Complex Streamed Media

Processor Architecture

Complex Streamed Media Processor Architecture

PROEFSCHRIFT

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van de Rector Magnificus Dr. D.D. Breimer,
hoogleraar in de faculteit der Wiskunde en
Natuurwetenschappen en die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op donderdag 13 maart 2003
te klokke 16:15 uur

door

Dmitry Cheresiz
geboren te Novosibirsk, Rusland
in 1974

samenstelling van de promotiecommisie

promotors:	Prof. dr. H.A.G. Wijshoff	
	Prof. dr. S. Vassiliadis	Delft University of Technology
co-promotor:	Dr. B.H.H. Juurlink	Delft University of Technology
referent:	Prof. dr. M. Valero	Universitat Politècnica de Catalunya, Barcelona, Spain

overige leden:	Prof. dr. G. van Dijk	
	Prof. dr. D. de Groot	
	Prof. dr. F. Peters	
	Dr. P.M.W. Knijnenburg	

Complex Streamed Media Processor Architecture
Dmitry Cheresiz. - [S .l. : s.n.].-Ill.
Thesis Universiteit Leiden. - With ref.
ISBN: 90-6734-195-9

Preface

This dissertation presents the results of the research which I carried out in the years 1997-2002, during which I worked as a research assistant at the Department of Computers Science of Leiden University and collaborated with members of the Computer Engineering Laboratory of Delft University of Technology. In this dissertation a novel processor architecture targeted at multimedia applications is described and evaluated. This architecture, called the *Complex Streamed Instruction Set (CSI)*, includes a set of complex instructions which operate on arbitrary-length data streams. Most of the results reported in this dissertation have been presented at international conferences. A description of the CSI architecture and evaluation of its performance on the image/video coders/decoders was presented at the *International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*. The results of the performance evaluation of CSI on image-processing kernels were presented at the *European Conference on Parallel Computing (Euro-Par'2001)*. A study of the performance improvements provided by CSI for the 3-dimensional graphics processing was presented on the *14-th IASTED International Conference on Parallel and Distributed Computing (PDCS-2002)*, while the study of performance scalability of CSI-enhanced processors was presented at the *Euro-Par'2002 Conference*. Finally, the example implementation of the CSI execution unit was presented at the *Euromicro Symposium on Digital System Design (2002)*.

I am very grateful to many people who helped and encouraged me during the last years. I enjoyed the fruitful collaboration and deep discussions with my colleagues at Leiden University and Delft University of Technology. Many friends in Holland, Russia, and other countries made life more interesting and joyful. Finally, I want to deeply thank my parents and brothers for their constant support and encouragement. This dissertation is dedicated to them.

Contents

Chapter 1

Introduction

Multimedia applications, such as audio/video compression and decompression, or two- and three-dimensional graphics processing, provide new highly valuable and appealing services to the consumer and form, consequently, a new important workload for computing systems. Large amounts of data which are required for representing multimedia information have to be processed by the applications. Often, the processing has to be done in real time, imposing even higher requirements on system performance. Progress in processor technology and particular features of multimedia applications enable consumer desktop systems to meet these requirements. In this thesis we propose a new architectural and design paradigm that will allow a general-purpose desktop system to provide the required performance for a wide range of multimedia applications, originating from application domains such as voice, image, and video coding and decoding, two-dimensional image-processing, and three-dimensional graphics. In this chapter, we will briefly discuss existing approaches to the design of computing systems targeted at multimedia processing and state a number of open questions, which will be answered in this dissertation. This chapter has the following structure. In Section 1.1 we give a short overview of multimedia-oriented computing systems and discuss one important class of them, the systems based on general-purpose processors enhanced with multimedia Instruction Set Architecture (ISA) extensions, in more detail. In Section 1.2 we present an example how a typical media kernel can be implemented using one of the existing media ISA extensions and state a number of questions that arise from the analysis of this example. In Section 1.3 we briefly sketch the structure of the dissertation.

1.1 Background and Related Work

Existing design approaches of systems capable of efficient execution of multimedia applications can be roughly divided in two main classes: *non-programmable* and *programmable*. When a non-programmable design approach is taken, application-specific integrated circuits (ASICs) are employed. The system may consist of just a single ASIC which executes a particular application, or it may consist of a simple core processor and a number of ASICs that are used to perform the most time-consuming parts of an application. The main feature of

such systems is that the ASICs are not programmable and can execute only one particular application (or a part of it). Therefore, such an approach is usually employed in embedded systems that have to perform a limited set of predefined tasks, such as a GSM phone which has to perform voice coding and decoding according to GSM standard, or a digital photo camera which needs to perform, for example, JPEG image coding/decoding. The ASIC-based approach, however, is not suitable for general-purpose desktop systems, which is the main scope of the present dissertation, for the following reasons. First, such a system has to execute a wide range of already existing media applications. Therefore, it would need a large number of ASICs, which are typically implemented as separate chips, that allow the execution of all these applications. This approach increases the cost of such a system unacceptably. Second, and more importantly, a general-purpose desktop system should be flexible enough to execute not only a fixed set of existing applications but also newly evolving media applications. This is not possible for an ASIC-based system. Because we are basically interested in general-purpose desktop computing systems, the ASIC-based design approach will be no longer considered in this dissertation.

Programmable system designs can be roughly divided in two main categories: systems equipped with a special-purpose programmable media (co)processor which is used to perform media applications or their most important parts, and systems based on a general-purpose processor provided with a media ISA extension. Examples of commercial programmable media processors are IBM's *MFAST* [48], Philips Trimedia *TM-1000* [60], and Texas Instruments *TMS320c80* [32]. There exist also some research prototypes of programmable special-purpose media coprocessors, for example, the *Imagine* stream processor developed at Stanford University [35, 46, 53, 54]. The main drawback of systems that employ programmable media processors is that an application programmer should be aware of the presence of such a coprocessor in a system in order to utilize it. This requires developing customized software. In order to make such development economically appealing for a software developer, a large number of systems with a particular media coprocessor should be present on the market, which is currently not the case for the class of consumer general-purpose desktop systems consisting of PC-based systems. Therefore, in order to provide sufficient media-processing capabilities, such systems were usually developed using the second programmable design approach mentioned above. The systems are based on a general-purpose processor enhanced with specialized multimedia architecture extensions. Below we give a short overview of such extensions.

Short-Vector Media ISA Extensions. Most of the vendors of processors currently employed in consumer and workstation class desktop systems have enhanced their instruction set architectures (ISAs) with media extensions. For example, Intel has extended their $x86$ architecture with the *Multimedia Extension* (MMX) [3, 49, 50] which accelerates integer media applications, such as speech, image, and video coders/decoders (which will be collectively referred to as *codecs*) and 2-D image processing. Later Intel provided the *Internet Streaming SIMD Extension* (SSE) [34, 64] which includes support for floating-point intensive media applications, such as 3-D graphics. AMD, another main producer of $x86$-compatible CPUs, has supported the MMX extension in their processors, as well as provided the $x86$ architecture with its proprietary $3DNow!$ extension [12], which is primarily intended to accelerate 3-D graphics workloads. Sun Microsystems has enhanced its *UltraSPARC* ISA with the *Visual In-*

Extension	Vendor	Implementation	Year
3DNow!	AMD	AMD Athlon	1999
MVI	DEC	Alpha 21264	1998
MMX/SSE	Intel	Pentium III	1999
AltiVec	Motorola	Motorola 7400 (G4)	1999
VIS	Sun	UltraSPARC III	1998

Table 1.1: Commercial media ISA extensions and their implementations.

struction Set (VIS) extension [31, 65] targeted at integer media benchmarks. DEC (currently Compaq) have provided the *Motion Video Instructions* extension (MVI) [27] for the *Alpha* architecture, while Motorola has enhanced the *PowerPC* architecture with the *AltiVec* extension [24, 44] which provides support for both integer and floating-point media applications. All of these extensions were actually implemented (see Table 1.1). The main design decisions used in extensions listed above are rather similar and are motivated by the following typical features of the multimedia applications. First, media applications spend large fractions of their execution time in loops which perform the same computation on elements of long data streams. This makes such applications well suited for efficient processing by architectures based on the *Simple Instruction Multiple Data* (SIMD/Vector-Processing) approach. Second, while traditional integer applications, such as those included in the *CINT2000* package of the SPEC's *SPEC CPU2000* suite [19] operate on 32-bit integer data types, the data types that are most frequently used in integer media applications are 8- or 16-bit integer values. These values mostly represent pixels of an image or audio samples. In order to exploit the parallelism inherent in media applications using the SIMD approach, each of the listed extensions provides a number of 64-bit (or, sometimes, 128-bit) *multimedia registers* that can contain a short vector consisting of several 8-bit, 16-bit, and, for some of the extensions, 32-bit integer values. For example, the VIS extension defines a set of 16 64-bit multimedia registers, each of which may contain a vector consisting of eight 8-bit, four 16-bit, or two 32-bit integer values. Each of the extensions also provides a set of instructions that perform the operations on all the elements of a short vectors contained in the media registers in parallel. In this way, parallelism is exploited according to the SIMD/Vector-Processing paradigm. Therefore, such extensions are referred to as *short vector SIMD extensions*. The older media extensions (VIS, MMX, MVI, and MDMX) use pairs of 32-bit floating point registers already present in the corresponding ISAs as 64-bit multimedia registers. The more recent media extensions such as AltiVec and SSE do not use FP registers but provide a separate set of 128-bit media registers. The 128-bit media register of AltiVec can contain, for example, a vector of 16 8-bit elements, thus increasing the amount of parallelism that can be exploited by a single instructions. Some of the extensions, namely, 3DNow!, SSE, and AltiVec, provide support for SIMD parallel computations on 32-bit single-precision floating-point numbers. The 3DNow! extension packs two such numbers in a single 64-bit media register and contains so-called *paired-single* instructions that operate on these numbers in parallel. The AltiVec and SSE extensions pack four 32-bit floating point numbers in their media registers and provide instructions that can operate on all four values in parallel. We remark here that the Intel SSE extension uses its registers solely for packing 32-bit FP numbers, not for packing integer values.

In Table 1.2 we summarize, for all of the mentioned media extensions, the sizes and

Extension	Register Size	Number of Registers
3DNow!	64 bits	8
MVI	64 bits	32
MMX	64 bits	8
SSE	128 bits	8
AltiVec	128 bits	32
VIS	64 bits	16

Table 1.2: Structures of the media register files of short-vector media ISA extensions.

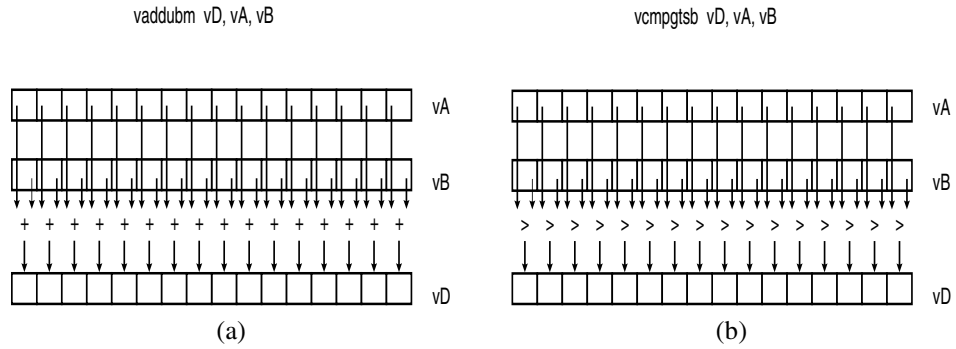


Figure 1.1: AltiVec's instructions *vaddubm* and *vcmpgtsb*.

the numbers of the available media registers. In order to provide a systematic way to view the architectures, we describe the typical structure of the instruction set provided by a short-vector extension. The instructions can be roughly split in the following categories.

1. **Data Transfer Instruction.** These instructions are used to transfer data to or from media registers. For example, such instructions include the loads and stores of the media registers from (or to) the memory.

2. **Arithmetic and Logical Instructions.** These instructions perform arithmetic or bit-wise logical operations, such as addition, multiplication, OR, etc., operating in parallel on all elements of the short vectors contained in the media registers. For example, the instruction `vaddubm vD, vA, vB` (Vector Add Unsigned Bytes Modulo), presented graphically in Figure 1.1(a), adds in parallel sixteen 8-bit unsigned integer values contained in the AltiVec short vector register `vA` to the corresponding 8-bit elements contained in the `vB` register and places the obtained sums in the corresponding locations of the destination register `vD`. We note here that if the result of an addition of two elements results in an overflow, the instruction specifies that modulo arithmetic should be used, i.e., that the overflow bit (9-th bit) should be discarded. This results in a wrap-around effect. Such effects, however, are not always desirable. It may, for example, create artifacts when used in image-processing. Suppose that two gray-scale images have to be added, and that the luminance values of the pixels are represented by 8-bit unsigned values (0 represents a black pixel and 255 represents a white pixel).

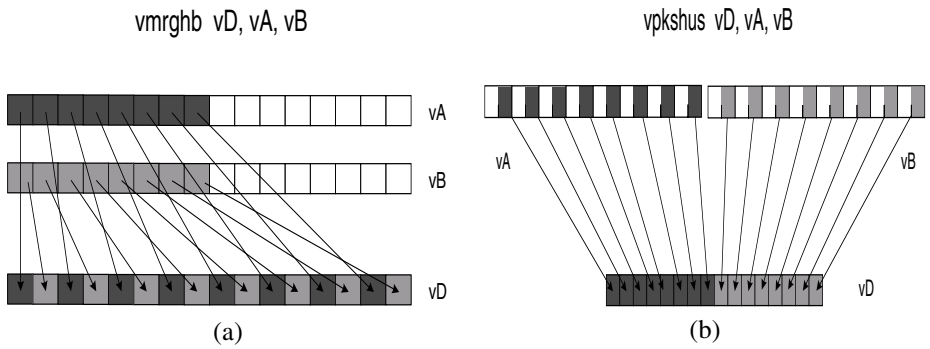


Figure 1.2: AltiVec's instructions *vmrghb* and *vpkshus*.

Then, if the luminance values of the two corresponding pixels are 1 and 255, respectively, that is one pixel is almost black and another is white, the sum of two pixels will be 0 (black) due to the wrap-around effect. To avoid such undesirable situations some of the extensions provide instructions that perform addition/subtraction operations using saturation arithmetic, i.e., if the result of an operation overflows (underflows), it is clipped (or *saturated*) to the maximum (minimum) value that can be represented by the element's data type. For example, AltiVec provides the `vaddubs` instruction (Vector Add Unsigned Byte Saturate) that performs addition of 16 pairs of corresponding 8-bit unsigned elements contained in 2 AltiVec registers and, contrary to `vaddubm`, does not discard the most-significant bit but saturates the 9-bit value to the 0..255 range. If values 1 and 255 are added using this instruction, the result will be 255.

3. **Comparison Instructions.** These instructions perform parallel comparisons of the corresponding short vector elements. For example, the AltiVec instruction `vcmpgtsb vD, vA, vB` (Vector Compare Greater Signed Byte), presented graphically in Figure 1.1(b), compares each of the 16 signed (2's complement) 8-bit elements of `vA` with the corresponding element of `vB`. If `vA`'s element is greater than the corresponding `vB`'s element, the corresponding element of the destination register `vD` is set to 11111111 and otherwise it is set to 00000000. The results of such parallel comparisons can be used, for example, as a mask for bitwise logical operations in order to provide conditional selection of elements of a short vector contained in a media register, as shown in [50]. An example illustrating this will be presented later in this chapter.

4. **Packing and Unpacking.** While multimedia data is stored in memory in a certain format (referred to as the *storage format*), it has often to be processed in a different, usually wider format, which is called the *computation format*. For example, if a stream of 8-bit unsigned values has to be added to a stream of 16-bit signed values, the elements of the first stream have to be converted (or *unpacked*) from their 8-bit storage format to the signed 16-bit computation format by zero extension from 8 to 16 bits. Such unpack operations are usually performed by interleaving the bytes of the first stream with zero bytes. For example, the AltiVec instruction `vmrghb vD, vA, vB`

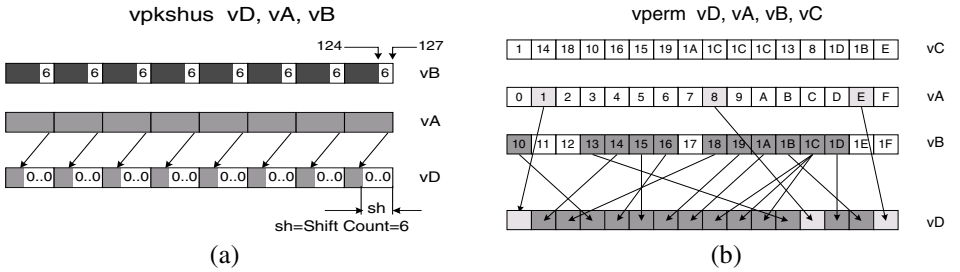


Figure 1.3: AltiVec's instructions *vslh* and *vperm*.

(Vector Merge High Bytes), depicted graphically in Figure 1.2(a), interleaves the 8 high-order bytes of `vA` with 8 high-order bytes of `vB`. If `vA` contains all zeroes and `vB` contains elements of the first stream, the instruction will perform the required unpacking. Short-vector ISA extensions usually contain a large number of unpack instructions that perform unpacking between most of the allowed storage and computation formats. The reverse operation to unpacking, i.e. conversion from a wider computation to a narrower storage format, is called $packing$. For example, the AltiVec instruction `vpkshus vD, vA, vB`, which is depicted graphically in Figure 1.2(b), packs 16 signed 16-bit elements contained in registers `vA` and `vB` to 16 unsigned 8-bit elements by saturating each of the 16-bit values to the range 0..255. Media ISA extensions usually contain large number of pack instructions that perform packing from most of the allowed computation to allowed storage formats.

5. **Parallel Element Shifts.** Short-vector media extensions usually provide instructions that perform parallel multiplication or division of elements contained in a media register by numbers that are powers of two. Such instructions shift in parallel each of the elements of a media register. For example, the AltiVec instruction `vslh vD, vA, vB` (Vector Shift Left Integer Halfword), which is depicted graphically in Figure 1.3(a), performs the following operation: each 16-bit element in `vA` is shifted left by the number of bits specified in the low-order four bits of the corresponding element in `vB`. Bits shifted out of bit 0 of the element are lost. Zeros are supplied to the vacated bits on the right. The result is placed into the corresponding element of `vD`. The parallel shift instructions are crucial for supporting fixed point integer arithmetic which is often employed in media applications. A common operation is the multiplication of an n-bit integer by an m-bit fixed point fractional constant (with the binary point left of its most significant bit), producing an $(n + m)$-bit result with the binary point located left of the m-th least significant bit. At the end of computation, the final result is rounded by adding the fixed point fraction that represents 0.5, and then it is shifted right by m bits to eliminate the fractional bits.

6. **Data Shuffles.** These instructions allow the elements contained in media registers to be reordered. For example, the AltiVec instruction `vperm vD, vA, vB, vC`, which is depicted graphically in Figure 1.3(b), fills the destination register with the 1-byte elements of source registers `vA` and `vB` under the control of `vC` according to the following

procedure. A 32-byte source vector is formed by concatenating `vA` and `vB`. For each integer i in the range $0..15$, the contents of the byte element in the source vector specified in bits $3..7$ of byte element i in `vC` are placed into byte element i of `vD`. Such reordering operations are often needed when the data stream that has to be processed has non-unit stride, because in order to process the stream elements using short-vector SIMD instructions, they have to be stored consecutively in a media register.

In the section to follow we consider the challenges that arise from our analysis of the short-vector ISA extensions and state the questions which have to be answered in order to resolve these challenges.

1.2 Challenges

Short-vector SIMD extensions have proven to provide significant performance benefits for multimedia applications executed on general-purpose CPUs equipped with these extensions, as was shown for example in [52, 69]. For an extensive comparative study of the performance improvements provided by the extensions listed earlier in this chapter, the interested reader is referred to [61]. While significant performance improvements can be achieved by employing short-vector SIMD extensions, these extensions have a number of architectural features that may limit further performance improvements. For example, Ranghanathan et. al. [52] observe that for MPEG/JPEG codecs, on average, 41% of the executed VIS instructions are instructions associated with overhead operations, such as packing/unpacking and data re-shuffling. This large number of overhead instructions increases the instruction traffic and, hence, the pressure on the decode/issue logic of a superscalar processor and can make it, potentially, a performance bottleneck. In the remaining part of this section we will present an example that will illustrate the features of short-vector extensions that may potentially limit further performance improvement. Using this example, we state a number of questions we attempt to answer in this thesis in order to provide an architecture will allow us to overcome these limitations. We remark that the quantitative performance evaluation will show that such an architecture, which resolves the questions mentioned above, provides substantial performance gains for media applications, when implemented using an appropriate mechanism.

We consider the piece of C-code presented in Figure 1.4, which is extracted from an implementation of the *Portable Network Graphics standard (PNG)*. This code fragment implements an important stage of the PNG coding process. It computes the *Paeth prediction* for each pixel d of the current row, starting from the second pixel. The Paeth prediction scheme selects from the 3 neighboring pixels a, b, and c, that surround d as depicted in Figure 1.5, the pixel that differs the least from the value $p = a + b - c$ (which is called the initial prediction). The selected pixel is called the *Paeth prediction* for d. If the pixel rows contained *length* $+ 1$ elements, *length* prediction values are produced. This prediction scheme is used during the image filtering stage of the image coding and decoding performed according to the PNG Standard. PNG is a popular standard for image compression and decompression, it is a native standard for the graphics implemented in *Microsoft Office* [17] and in a number of other applications. Figure 1.6 presents an implementation of the code fragment in pseudo-code derived from the AltiVec assembly. In this figure, by `ri` we denote the general-purpose register `GPRi` of the underlying ISA, by `vri` we denote the i-th vector register of AltiVec.

```
void Paeth_predict_row(char *prev_row, char *curr_row, char *predict_row, int length)
{ char *bptr, *dptr, *predptr;
char a, b, c, d;
short p, pa, pb, pc;

bptr = prev_row+1;
dptr = curr_row+1;
predptr= predict_row+1;

for(i=1; i < length; i++)
{ c = *(bptr-1); b = *bptr;
a = *(dptr-1); d = *dptr;
p = a + b - c;                                   /* this is the initial prediction */
pa = abs( p - a );                               /* distance of each member */
pb = abs( p - b );                                                  /* to the */
pc = abs( p - c );                                                  /* initial estimate */
if ((pa ≤ pb)&&(pa≤pc) *predptr = a;
else if (pb≤pc) *predptr = b;
else *predptr = c;

bptr++; dptr++; predptr++;
}
}
```

Figure 1.4: The Paeth prediction routine according to PNG specification [55].

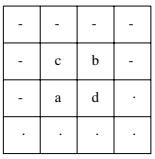


Figure 1.5: Definition of *a, b, c,* and *d* according to PNG specification

The # symbol is used to denote the comments. It is assumed that the function parameters *prev_row, curr_row, predict_row,* and *length* are contained in the registers `r1, r2, r3`, and `r4`, respectively. It is also assumed that the length of the input pixel streams is larger than the length of the output stream, which is equal to *length*, by one. The registers `r0` and `vr0` are assumed to contain 32 and 128 zeroes, respectively. Because of the fixed size of the AltiVec media registers, the data streams to be processed are divided into 16-byte sections that fit into the media registers. Each iteration loads sections of the source streams, each of which contains 16 8-bit pixels, unpacks the pixel data to the computation (16-bit) format, performs the main computation calculating the Paeth predictor value for 16 pixels, and stores the resulting

```
prologue:
       li      r5, 0            # initilize loop counter
       addi    r6, r4, -16      # initilize termination counter
       cmp.leq r7, r5, r6       # check if we have at least 17 pixels
       beq     r7, r0, epilogue # if not--jump to epilogue, no AltiVec computation
       lvx     vr01, r1, 0
       lvx     vr02, r2, 0
       addi    r1, r1, 16
       addi    r2, r2, 16
loop:
   #Load Data
       lvx     vr03, r1            #vr3 contains next section  c's
       lvx     vr04, r2            #vr4 contains next section  a's
       vsidoi  vr05, vr01, vr03, 1  #vr5 contains current section of b's
   #Unpack
       vmrghb          vr07, vr03, vr00  # unpack 8 c's
       vmrglb          vr08, vr03, vr00  # unpack 8 other c's
       vmrghb          vr09, vr04, vr00  # unpack 8 a's
       vmrglb          vr10, vr04, vr00  # unpack 8 other a's
       vmrghb          vr11, vr05, vr00  # unpack 8 b's
       vmrglb          vr12, vr05, vr00  # unpack 8 other b's
   #Compute
       vadduhs         vr15, vr09, vr11  # perform 8 a+b
       vadduhs         vr16, vr10, vr12  # perform 8 a+b
       vsubshs         vr15, vr15, vr07  # subtract 8 c's
       vsubshs         vr16, vr16, vr08  # subtract 8 c's
           # vr15, vr16 contain 16 p's
       vsubshs         vr17, vr15, vr10  # get 8 (p-a)-s
       vsubshs         vr18, vr00, vr17  # get 8 (a-p)-s
       vcmpgtsh        vr19, vr00, vr17  # mask fro (p-a)<0
       vnor            vr20, vr17, vr00  # mask fro (p-a)>=0
       vand            vr21, vr18, vr19  # fill (a-p)-s
       vand            vr22, vr17, vr20  # fill (p-a)-s
       vor             vr23, vr21, vr22  # vr23 contains 8 abs(p-a)

           # with  7 more instructions  we obtain rest 8 abs(p-a)'s in vr 24
           # thus 16 pa's = abs(p-a)'s in vr23, vr24
           # similarly, with 14 more instructions  we obtain
           #      16 pb's = abs(p-b)'s in vr25, vr26
           # similarly, with 14 more instructions  we obtain
           #      16 pc's = abs(p-c)'s in vr27, vr28
       vcmpgtsh        vr17, vr23, vr25  # pa > pb mask for 8 els
       vnor            vr18, vr17, vr00  # pa <= pb mask
       vcmpgtsh        vr19, vr23, vr27  # pa > pc mask for 8 els
       vnor            vr20, vr19, vr00  # pa <= pc mask
       vand            vr21, vr18, vr20  # (pa<=pb)&&(pa<=pc) mask
       vand            vr28, vr09, vr21  # select a's
       vcmpgtsh        vr23, vr25, vr27  # pb > pc mask
       vnor            vr23, vr23, vr00  # pb <=pc mask
       vnor            vr25, vr21, vr00  # !(pa<=pb)&&(pa<=pc) mask
       vand            vr25, vr25, vr23  # (!(pa<=pb)&&(pa<=pc))&&(pb<=pc) mask
       vand            vr27, vr11, vr25  # select b's
       vor             vr28, vr28, vr27  # merge a's and b's
       vnor            vr27, vr25, vr00  # mask for c's
       vand            vr27, vr07, vr27  # select c's
       vor             vr28, vr28, vr27  # merge c's with a's and b's
           # Now vr22 contains 8 result values in 16-bit format
           # Similarly, with 15 more instructions we obtain 8 other result
           # values and place them in vr29
   #Pack:
       vpkshus         vr28, vr28, 29    # pack to 8-bits, vr28 contains 16 results
   #Store:
       stvx            vr28, r3, 0       # store the result
   #Miscellaneous
       vor             vr01, vr03, vr00  # move vr03 to vr01 for next iteration
       vor             vr02, vr04, vr00  # move vr04 to vr02 for next iteration

   #Update pointers
       addi            r1, r1, 16
       addi            r2, r2, 16
       addi            r3, r3, 16
   #Loop control
       addi            r4, r4, 16
       cmp.leq         r7, r4, r6       # termination condition
                                        # at least 17 pixels should be left
       bneq            r7, r0, loop     # go to next iteration
Epilogue:
       # if there are pixels left, process them in scalar mode
```

Figure 1.6: AltiVec code for *Paeth_predict_row*.

16-byte section of the destination stream back to memory.

In the following, we analyze the presented code example. According to the tasks they perform, the instructions can be organized in several groups, as described below.

- **Main computation.** All the instructions which are located between the `Compute:` and `Pack:` labels realize this task. This part of the code contains 76 instructions, some of which are omitted from the picture for brevity. Out of these 76 instructions, 4 instructions are used to compute p, and $14 \cdot 3 = 42$ instructions are used to compute the absolute differences pa, pb, and pc. Thirty instructions are used to perform comparisons and conditionally select the values of a, b, and c in order to form the Paeth prediction for each pixel of the currently processed section. It can be easily observed that the main computation of the kernel requires multiple instructions for each stream section. We remark that an interesting scenario could occur if the multiple operations required for the main computation could be grouped and compounded into a single complex operation which doesn't require more cycles than the simple ones. In such case, a new instruction that specifies this complex operation can be introduced. This single complex instruction can then be used to substitute the multiple simple instructions used to implement the main operation. It has been shown that the main computations which are performed by a number of important kernels, such as the *Sum of Absolute Differences (SAD)*, the *Paeth Prediction*, and others, lend themselves to such a technique [25, 66].

- **Sectioning.** This task performs all the computations associated with processing the streams in sections. It consists of the following sub-tasks.

  - Loop setup. This task consists of initializing the loop counter and checking if the loop has to be executed at all. If yes, some instructions necessary for initializing the first iteration are executed. This task is performed by the instructions located between the `prologue:` and `loop:` labels.
  - Pointer updates. This task includes advancing the data stream pointers at the end of each iteration, so that at the following iteration they point to the next sections of the streams. It is performed by the instructions located between the labels `Update pointers` and `Loop control`.
  - Loop control. This task consists of incrementing the loop counter, computing the loop termination condition, and branching. It is performed by the instructions located between the labels `Loop control` and `Epilogue`.

- **Data Transfers.** This task consists of loading the sections of the source data streams into media registers performed by three instructions located at the `Load Data` label, and storing the computed results back to the destination stream performed by the `stvx` instruction located at the `Store:` label.

- **Unpacking.** This task is responsible for the conversion of the source stream data from the storage (8-bit unsigned integer) to the computational (16-bit signed integer) format. The instructions located between the labels `Unpack` and `Compute` perform this task. Since the 8-bit values are unsigned and, therefore, have to be zero-extended to 16 bits, the loaded data bytes are merged with zero bytes from the `vr0` register.

- **Packing.** This task is responsible for the conversion of the results from the computation to the storage format and is performed by the instruction `vpkshus` located at the `Store` label.

Except for the instructions employed for the main computations, all other instructions are used for tasks which constitute the overhead. We note here that other media kernels, when implemented using a short-vector SIMD extension like AltiVec, may need to perform additional overhead tasks not present in the code example we considered. One of these tasks is described below:

- **Data Re-shuffling.** This task usually arises when the data streams have non-unit strides. Load instructions of short-vector SIMD extensions, however, can transfer to the media registers only bytes located consecutively in memory. In order to be processed, stream elements occupying non-consecutive locations in the loaded sections, have to be extracted and stored consecutively in a media register. This task was not necessary in the code fragment we consider because the stream data was stored with unit stride.

Summarizing the observations presented above, we see the following. Implementations of media codes using the short-vector ISA extensions require execution of large amounts of instructions. For example, if the Paeth predictor must be computed for a row of 1024 pixels, the AltiVec code presented in Figure 1.6 will result in a dynamic instruction count of 8 *(prologue)* $+ 64 \cdot [3$ *(load)* $+ 6$ *(unpack)* $+ 76$ *(compute)* $+ 1$ *(pack)* $+ 1$ *(store)* $+ 2$ *(miscellaneous)* $+ 3$ *(pointer update)* $+ 3$ *(loop control)*$] = 8 + 64 \cdot 95 = 6088$ instructions. This high instruction count is caused by the following features of the short-vector media extensions. First, if the main operation to be performed is relatively complex, it requires multiple instructions. Second, the overhead tasks associated with stream sectioning, loading, storing, packing, unpacking, and data rearrangement require separate instructions. The large number of instructions to be executed is likely to severely limit the performance that can be achieved by using short-vector media extensions. Since the amount of parallelism that can be exploited by a single instruction of such extensions is limited by the size of the media register, more instructions have to be executed in parallel in order to exploit higher levels of data-level parallelism and to increase performance. This, however, increases the pressure on the decode-issue logic of a superscalar processor. In order to sustain this pressure, the issue width of the processor has to be increased because, otherwise, the decode-issue logic is likely to become a bottleneck. It is generally accepted that such increase incurs huge hardware costs and, furthermore, can increase cycle time. One way to attack this problem is to increase the size of the media registers in order to increase the amount of parallelism that can be exploited by a single instruction. This approach, however, results in the change of the architecture of a short-vector media extension and, hence, binary incompatibility. Old codes will have to be recompiled, or even rewritten, which is undesirable.

The previous discussion which illustrated the shortcomings of short-vector register architectures (and, in fact, of vector register architectures in general) suggests a number of questions which have to be answered in order to provide an efficient architecture for media applications.

- Can the number of instructions needed to specify the main useful computation be reduced with the introduction of complex instructions that are general enough to be used in multiple applications?
- Can the instructions associated with the sectioning (or, at least, their execution overhead) be eliminated? Or, more specifically;
  - Can loop setup instructions be eliminated?
  - Can pointer update instructions be disposed of?
  - Can loop control instructions be avoided?
- Can load and store instructions be eliminated?
- Can instructions associated with conversion between storage and computational formats (packing and unpacking) be disposed of?
- Can the instructions associated with the reorganization of data in consecutive sections be avoided?

The previous observation (that an increase in the amount of data-level parallelism that can be achieved by an increase of the size of multimedia registers implies software incompatibility) rises another question:

- Can multimedia ISA extensions be designed in such a way that increasing amounts of parallelism can be exploitable by a single instruction, while software compatibility will be retained?

Some of the existing media ISA extensions attempt to answer a limited number of the questions listed above. For example, several extensions, such as VIS and MMX, try to reduce the number of instructions needed to perform certain complex media computations. VIS and MMX include the special-purpose `pdist` and `psadbw` instructions, respectively, which collapse relatively complex computations required to compute the sum of absolute differences of 8-bit values contained in a media register, in a single instruction. The *Matrix-Oriented Multimedia (MOM)* media extension proposed by Corbal et. al. [10, 11] attempts to solve the question of allowing a single instruction to increase the amount of parallelism as well as to allow different numbers of data items to be processed in parallel without losing software compatibility. The MOM ISA extension is a vector-like architecture. It provides a number of 16-element vector registers, where each element is a 64-bit entity, which can be seen, essentially, as a media register similar to those employed in short-vector media extensions like MMX. An element of a MOM register can contain, for example, eight 8-bit values packed in 64 bits. A MOM register, therefore, can contain up to 128 8-bit elements. MOM instructions operate on data contained in MOM register. A single instruction can exploit the parallelism to a degree of up to 128 8-bit data items processed in parallel. Different implementations of MOM can, therefore, perform a varying number of parallel operations in a single instruction, depending on the amount of parallel processing hardware present in a particular implementation. If no more than 128 8-bit elements have to be processed in parallel, the size of the MOM register does not need to be increased, and, hence, the software will be compatible.

Another proposal which employs vector processing for multimedia applications is the *Vector-IRAM (V-IRAM)* [36, 41] architecture developed within the *Berkeley Intelligent RAM (IRAM)* project [38, 47]. The main idea of this project is integrating in a single chip main memory and the CPU enhanced with a vector facility. This approach allows to achieve higher bandwidth and lower memory latency and has shown its high potential [37, 45]. The V-IRAM architecture is a register-to-register vector architecture enhanced with subword processing capabilities. Contrary to short-vector ISA extensions, V-IRAM employes long vectors. Furthermore, it includes strided and indexed vector load/store instructions. Additionally, load instructions can promote the loaded values to wider computation format. These techniques allow V-IRAM to reduce the number of overhead instructions needed for address generation, loop control, packing and unpacking.

In this thesis we will propose a novel media ISA extension, called the *Complex Streamed Instruction Set (CSI)*. The main features of the proposed architecture give the following answers to the questions raised earlier in this section.

- The number of instructions needed to specify the main useful computation can be reduced for many kernels. This is achieved by including instructions that can perform complex arithmetic operations, such as the *SAD*, or the *Paeth prediction*.
- The sectioning overhead instructions associated with pointer updates and loop control can be eliminated. Explicit sectioning in CSI is avoided by employing memory-to-memory instructions that perform the associated tasks implicitly in hardware.
- Load and store instructions can be avoided.
- Overhead instructions associated with packing and unpacking can be disposed of.
- Instructions associated with arranging the data consecutively can be eliminated. The elimination of these instructions, as well as those used for loads, stores, packing, and unpacking, is achieved by allowing a CSI instruction to specify these tasks and perform them implicitly in hardware simultaneously with the main computation.
- CSI codes don't require recompilation or rewriting to benefit from bigger amounts of parallel hardware available in aggressive CSI implementations because the amount of data which is processed in parallel is not a part of architecture and, hence, doesn't appear in the codes.

We will present a design of a hardware unit capable of executing CSI instructions. We validate the proposed ISA extension by studying the performance of a wide range of multimedia benchmarks originating from image/video coding and decoding, two-dimensional image processing, and three-dimensional graphics application domains. The performance exhibited by superscalar processors enhanced with the CSI execution unit are compared with performance exhibited by the same processors enhanced with hardware capable of executing instructions from several of the existing media ISA extensions. Since the performance of such extensions can greatly depend on the quality of the codes, the main criterion for selecting particular ISA extensions for comparison with CSI was availability of vendor codes for a sufficient number of media applications. According to this criterion, as the reference

media extension oriented toward integer media applications, we have selected the Sun's Visual Instruction Set (VIS), since an implementation of a large number of important media kernels is provided in the VIS Developer's Kit (VSDK) [30]. As a reference extension oriented toward floating-point media applications, we have selected Intel's Internet Streaming SIMD Extension (SSE), for which implementations of the most important kernels employed in floating-point intensive three-dimensional graphics applications are provided by the vendor in [14, 15]. The results of these studies allow us to answer all of the open questions listed earlier in this chapter.

1.3 Overall Structure of the Thesis

In Chapter 2 we introduce the *Complex Streamed Instruction Set* architecture, or *CSI*. We describe the architectural state of CSI, which consists of the CSI memory state and the CSI register state. Then we describe the operands of CSI instructions, concentrating on the description of the operand types that are particular for CSI, such as the CSI arithmetic streams and the CSI bit streams. These are, respectively, the arbitrary length sequences of arithmetic or logical (single bit) elements located in storage according to certain regular access patterns. We show how such operands are specified by sets of control registers, which are a part of the CSI register state. For example, we show how a CSI arithmetic stream can be specified by a *stream control register set (SCR-set)*. Next we present the conditional execution support, for which CSI bit streams and the stream-mask mode of operation are used. After this, we give a detailed description of the interruption handling mechanism employed in CSI. This mechanism is based on representing the execution of a CSI instruction as a sequence of units of operation (UOPs) and on the automatic update of the control registers that represent the number of UOPs already processed, i.e., the extent to which an instruction has been executed at the point of interrupt. We show that this mechanism allows resumption of the execution of an interrupted instruction from the point of interruption. This is an important condition for achieving high performance with CSI, since the CSI instructions operate on arbitrary-length data streams and, hence, can be rather long-running.

The set of CSI instructions is being described in Chapter 3. First, the instruction formats are given. Then we describe the CSI instruction classes in which instructions are organized depending on the number of elements they process, interruptibility, and dependence on the stream-mask mode. Thereupon, the complete set of CSI instructions is presented. CSI instructions are organized in the following groups depending on the type of operations they perform: *Simple Arithmetic and Logical, Comparison-Related, Accumulation-Related, Stream Reorganization, Bitstream-Operating, Special-Purpose*, and *Auxiliary*. For each of these groups, we list the instructions belonging to it and we describe how they are encoded and executed. After this, we describe the auxiliary operations of converting stream data between storage and computation formats (packing and unpacking), which can be performed by a CSI instruction next to the execution of its main operation.

An example implementation of a hardware unit capable of executing CSI instructions is described in Chapter 4. We first describe a general design of such a unit. Since the operations performed by typical CSI instructions, such as loading source stream elements, unpacking, performing the main operation, packing, and storing the results, are usually independent of

the same operations performed on other stream elements, they can be overlapped. Therefore, the proposed design has a pipelined structure. The design of a unit depends on the levels of memory hierarchy to which it is interfaced. In this chapter we provide a detailed description of a unit that interfaces to the first-level data cache. In the presented implementation, a whole cache block is transferred to or from the unit in a single cache access. We analyze the computations needed for generating address information. Although these computations turn out to be complex, we show that they can be performed in a pipelined fashion and present a design of an address generation unit, organized as a three-stage pipeline. This unit can compute address information for a new cache block every cycle.

In Chapter 5 we present the results of extensive experimental validation of the proposed CSI ISA extension. The proposed CSI extension is validated by comparing the performance of superscalar processors enhanced with the CSI execution unit to the performance of these processors enhanced with hardware units capable of executing instructions of several existing media ISA extensions. We compare the performance of the CSI-enhanced superscalar processors with that of VIS- and SSE-enhanced processors using a large number of benchmarks originating from the following multimedia application domains: image and video coding/decoding, two-dimensional image processing, and three-dimensional graphics processing. We also study how the performance of these machines scales if the amount of parallel SIMD processing hardware is increased, and identify the bottlenecks of the VIS- and SSE-enhanced processors.

Finally, in the last chapter, conclusions are drawn, in which we summarize our findings, present our main contributions, and propose directions for future research.

Chapter 2

CSI Architecture: General Description

In this chapter we describe a novel multimedia-oriented ISA extension, called the *Complex Streamed Instruction* Set architecture, or *CSI*. In the remainder of this chapter, we assume that CSI is an extension to a general-purpose 32-bit RISC-like instruction set architecture which has 32 32-bit general-purpose registers, 32 32-bit floating-point registers, and an address space of 2^{32} bytes. The byte-ordering convention is assumed to be *big-endian*. Bits, bytes, halfwords, etc. are numbered from left to right with a count starting at zero. As usual, the most significant bit of a byte (halfword/word) is the leftmost one, i.e., the 0th bit according to our numbering convention. The definitions of the CSI architecture presented in this chapter can be easily modified if the underlying general-purpose ISA has a different number of registers, or register size, or the byte-ordering convention of the memory space as the one assumed above.

As was shown in the introduction, existing short-vector multimedia ISA extensions such as VIS, MMX, and others, have a number of architectural features which limit the performance. The CSI is a memory-to-memory vector-like architecture which allows us to avoid limitations of short-vector media ISA extensions. Before we start the description, we briefly list these limitations and discuss how they are resolved in CSI.

- **Limited amount of parallelism.** The number of elements on which a single instruction can operate in parallel is not limited in CSI because instructions operate on arbitrary length data streams stored in memory. Furthermore, CSI can address two-dimensional strided data streams allowing to expose more parallelism than is possible with the standard one-dimensional strided vector addressing.

- **Large number of overhead instructions.** In CSI, overhead instructions needed for sectioning, data alignment, rearranging, and conversion are avoided. The associated tasks are performed implicitly in hardware by a single CSI instruction simultaneously with its main computation.

- **Inefficient implementation of complex operations.** Standard short-vector media

extensions implement a complex operation by a sequence of simple instructions. However, many complex operations common in multimedia applications, such as inverse discrete cosine transformation (IDCT), variable-length coding, Paeth coding/prediction, macroblock padding and others, can be implemented directly in hardware at acceptable cost. Such complex operations are performed by a single CSI instruction.

This chapter is organized as follows. In Section 2.1, the state of the CSI architecture is described. Section 2.2 presents the possible operands of CSI instructions, concentrating on the description of the stream operand types, which are particular to CSI. Section 2.3 describes how interrupts are handled in the CSI architecture, presenting the mechanism that allows an interrupted CSI instruction to be restarted from the point of interruption. Section 2.4 presents some conclusions.

2.1 CSI State

The programmer-visible state of CSI consists of the *CSI memory space* and the *CSI register space*. The CSI memory space, also referred to as the *storage space*, coincides with the memory (storage) space of the general-purpose ISA which the CSI extends. It is, therefore, a byte-addressable linear space of 2^{32} bytes, that uses the big-endian ordering convention (see the assumptions in the introductory part of this chapter). The CSI register space is depicted in Figure 2.1 and consists of the *stream control registers*, the *mask stream control registers*, the *stream status register*, the *integer packed accumulator register (pacc_int)*, and the *floating-point packed accumulator register (pacc_fp)*.

Stream Control Registers. There are 96 32-bit *stream control registers (SCRs)* organized in 16 sets with 8 registers per set. Each set of the stream control registers, or *SCR-set*, contains the parameters that completely specify a *CSI arithmetic stream*, as well as some parameters that specify the position of the current element when the stream is processed by a CSI instruction. A CSI arithmetic stream contains elements that represent arithmetic data, i.e., binary integer or floating-point numbers, and is formed by groups consisting of a certain fixed number of consecutive bytes and following a two-dimensional strided access pattern. For a detailed description of the structure of a CSI arithmetic stream and the meaning of its parameters, such as *base, horizontal stride, vertical stride* and others, the reader is referred to Section 2.2.

An SCR-set consists of the following eight 32-bit registers, numbered from 0 to 7.

0. **Base.** This register initially contains the base address of the stream. During the execution of an interruptible CSI instruction, it is implicitly modified and contains the address of the stream element to be processed next by the instruction. This information is used to resume execution after such an instruction has been interrupted. Detailed descriptions of how the register is updated and how it is used to resume execution after an interruption are presented in Section 2.3.

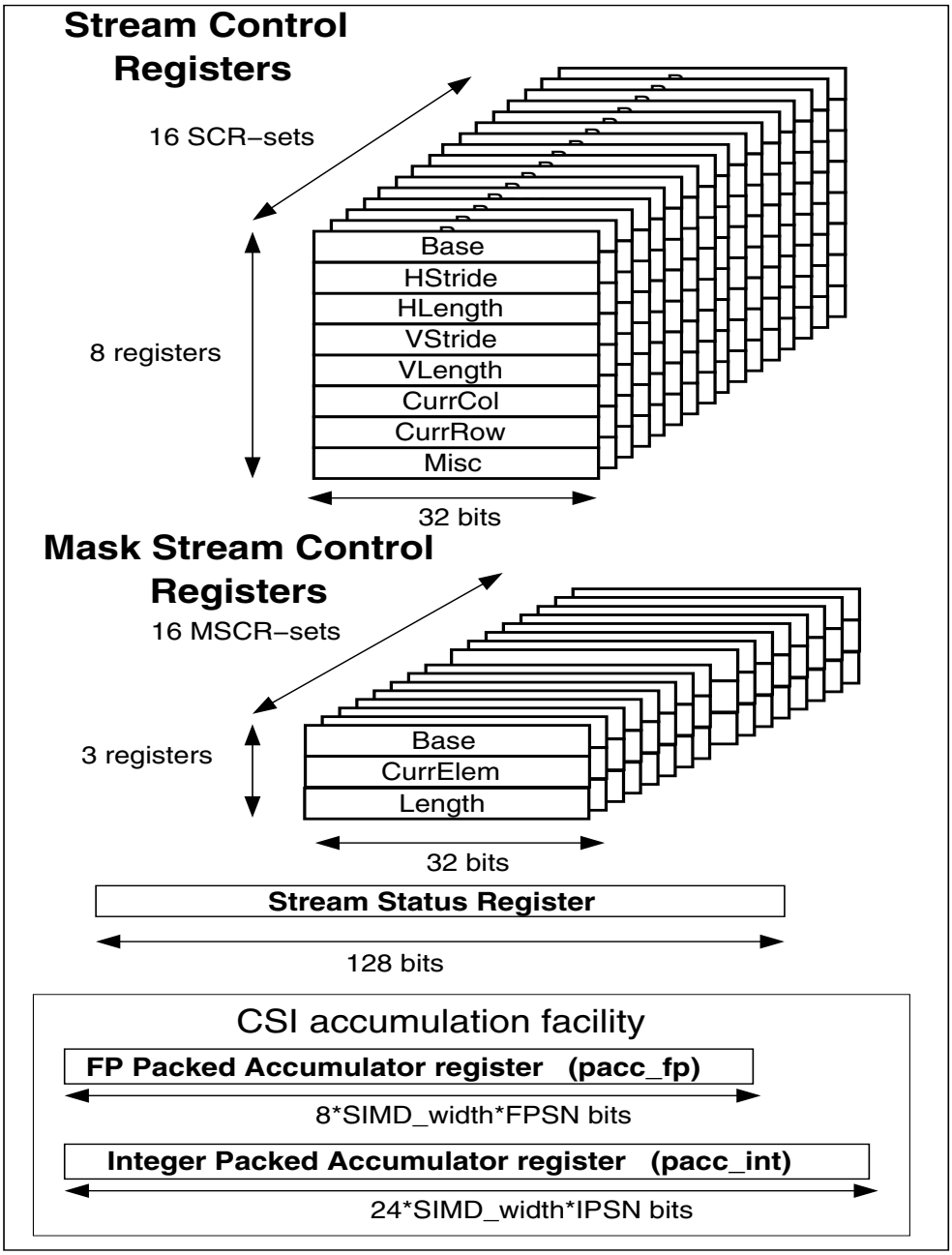


Figure 2.1: CSI register space.

1. **HStride.** This register contains the horizontal stride of the stream, i.e. the difference between the addresses of the first bytes of the two consecutive groups belonging to the same row.

2. **HLength.** This register contains the horizontal length of the stream, i.e., the number of elements in a row.

3. **VStride.** This register contains the vertical stride of the stream, i.e., the difference between the addresses of the first bytes of the first groups belonging to two consecutive rows.

4. **VLength.** This register contains the vertical length of the stream, i.e., the number of rows in it.

5. **CurrCol.** This register contains the position of the element to be processed by an interruptible CSI instruction within its row. This position, as well as the register, are termed as the *Current Column*. It is used to resume execution after such an instruction has been interrupted. The register is normally zero at the start of execution, and it is set to zero at completion. Details concerning the operation of the register and how it is used to resume instruction execution after an interruption are presented in Section 2.3.

6. **CurrRow.** This register contains the number of the row to which the element to be processed by an interruptible CSI instruction belongs. This row number, as well as the register, are termed as the *Current Row*. The register is used to control resumption of execution after such an instruction has been interrupted. It is normally zero at the start of execution, and it is set to zero at completion. Details concerning the operation of the register and how it is used to resume instruction execution after an interruption are presented in Section 2.3.

7. **Misc.** This register (called *Miscellaneous*) contains a number of miscellaneous stream parameters. It contains the following fields (see Figure 2.2).

   - *GrSize.* This 2-bit field specifies the size of a group in bytes. The possible group sizes are 1, 2, 3, and 4 bytes.
   - *ElSize.* This field specifies the size in bytes of a stream element in its storage format. The allowed values are 1,2, and 4, specified as the 2-bit numbers 00, 01, and 11, respectively.
   - *FP.* When set, this 1-bit flag specifies that the stream elements are floating-point numbers. It is used only if the element size is 4 bytes, because for other cases the size immediately implies that elements are integer.
   - *Sign.* This 1-bit flag specifies if the elements should be interpreted as signed or as unsigned integers.
   - *ProcessSize.* This 2-bit field specifies how many bytes are required to represent a stream element in its computation format. Possible values are 1, 2, and 4, encoded in the field as 2-bit numbers 00, 01, and 11.

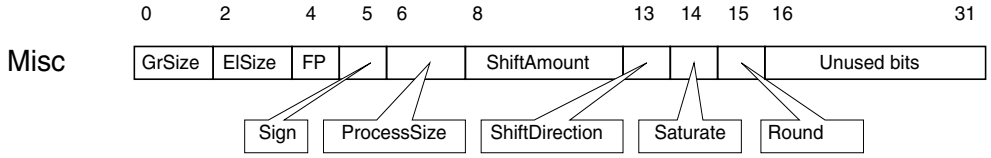


Figure 2.2: Format of **Misc** stream control register.

- *ShiftAmount.* This field specifies the number of bits by which an integer stream element will be shifted during conversion between the storage and computation formats. Details about the usage of this field, as well as the usage of three other fields described below, are presented in Section 3.4.
- *ShiftDirection.* This 1-bit field specifies the direction of shift operation which is performed on an element during conversion between the storage and computation formats. If the field is zero the direction is 'left', otherwise the direction is 'right'.
- *Saturate.* This 1-bit flag is used differently for source and for destination streams. For the source streams, it specifies whether the saturation arithmetic should be used during the computation. For a destination stream, it specifies whether saturation should be performed during the conversion of an element from computation to storage format.
- *Round.* This 1-bit flag is used only if the stream is a destination stream. It specifies whether rounding should be performed during the conversion of an element from computation to storage format.

We remark here that the *ShiftAmount, ShiftDirection, Saturate*, and *Round* fields are used to control the process of conversion of stream elements between the computation and the storage formats. Such conversion can occur only when the formats differ and, therefore, only when stream elements are integer. Floating-point numbers used in CSI have a unique format (IEEE 754 32-bit) and, hence, do not require conversion. The stream control registers (SCRs) can be accessed or modified by moving their contents to or from the general-purpose registers of the processor using the CSI instructions `csi_mf_scr` (*move from an SCR*) and `csi_mt_scr` (*move to an SCR*), respectively. For details, see Section 3.3.7.

Mask Stream Control Registers. There are 48 32-bit *mask stream control registers (MSCRs)* organized in 16 sets with 3 registers per set. Each set of the mask stream control registers, or an *MSCRS-set*, contains the parameters that fully specify a *CSI mask stream* (also called a *CSI bit stream*), as well as the position of the current element when the stream is processed by a CSI instruction. A CSI mask stream is a sequence of individual bits stored consecutively in memory. For a detailed description of the structure of a CSI mask stream, the reader is referred to Section 2.2.2. An MSCR-set consists of the following three 32-bit registers, numbered from 0 to 2.

0. **Base.** This register contains the address of the byte that contains the first bit of the mask stream. During the execution of a CSI instruction, the register is implicitly modified

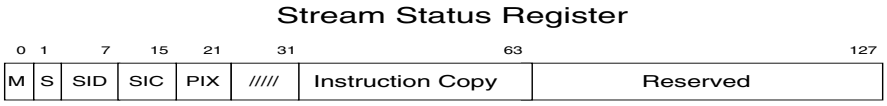


Figure 2.3: Format of the Stream Status Register (SSR).

and contains the address of the byte that contains the mask bit to be processed by an interruptible CSI instruction. The register is used to control resumption of the operation after such an instruction has been interrupted. Details concerning the operation of the register and the effect of an interruption are presented in Section 3.4.

1. **CurrElem.** This register contains the number of the mask stream element to be processed by a CSI instruction. This register is implicitly modified during the execution of an interruptible CSI instruction and facilitates resumption of execution if the instruction was interrupted. The register name stands for the *'Current Element'*.

2. **Length.** This register contains the number of bits in a CSI mask stream.

The mask stream control registers (MSCRs) can be accessed or modified by moving their contents to or from the general-purpose registers of the processor using the CSI instructions `csi_mf_scr` and `csi_mt_scr`, respectively.

The Stream Status Register. This is a 128-bit register containing five fields describing the current status of the instruction that is executed and the current mode of operation. The fields are arranged as in Figure 2.3 and are described as follows.

- **Stream Mask Mode Bit (M).** When this bit is 1, the *stream mask mode* is enabled and the CSI instructions are executed under the control of the elements of the *mask stream* specified by each instruction. When the bit is zero, the mode is off (disabled). For details, see Section 2.2.3.

- **Stream Sequential Mode Bit (S).** When this bit is 1, the *stream sequential mode* is enabled, the CSI instructions operate on their elements sequentially, element by element. When this bit is zero, the *stream SIMD mode* is enabled and CSI instructions operate on groups of consecutive elements in parallel SIMD fashion. For details, see Section 2.3.3.

- **Stream Identifier (SID).** This 5-bit field facilitates the handling of exceptions caused by executing CSI instructions. If an *access exception* was recognized, i.e., if an exception was recognized during access to a stream element, the SID fields designates the SCR-set or the MSCR-set that describes the corresponding stream. If the most-significant bit of the field is zero, the 4 other bits designate the number of an SCR-set, otherwise they designate an MSCR-set number.

- **Stream Interruption Code (SIC).** This field facilitates the handling of exceptions caused by executing CSI instructions. If an exception was recognized for a CSI instruction and has caused the program interruption, the SIC field designates the type of

the exception. The exception types which can cause CSI instruction interruptions are presented in Section 2.3.3.

- **Instruction Copy (IC).** This field simply contains a copy of the current CSI instruction. It is also used to facilitate exception handling. For example, an exception-handling routine may access the information contained in this field to perform the result fix-up for arithmetical exceptions.

The 128-bit stream status register is split into four 32-bit parts numbered from 0 to 3. The part consisting of the bits 0..31 (most significant bits) has number 0, the part consisting of the bits 32..63 has number 1, and so forth. The parts can be accessed or modified by moving them to or from the general-purpose registers of the CPU using the instructions `csi_mf_ssr_pacc` (*move from SSR or from an accumulator register*) and `csi_mt_ssr_pacc` (*move to SSR or to an accumulator register*), respectively. Bits 22..31 and 64..127 are reserved for possible future use and are stored as zeroes. If `csi_mt_ssr_pacc` specifies a value other than zero for these bit positions, a specification exception is recognized.

Accumulation Registers. Two accumulation registers **pacc_int** and **pacc_fp** are used by accumulation-related CSI instructions to perform parallel accumulations of integer and floating-point numbers, respectively. The sizes of these registers are implementation-dependent and determined by the parameter *SIMD_width*. This parameter designates the width of the datapath of the CSI execution unit in bytes, i.e., it designates how many bytes can be processed in parallel on a given CSI execution unit. The size of **pacc_int** depends also on the *integer partial sum number (IPSN)*, which is defined as the latency (in cycles) of the integer addition. Similarly, the size of **pacc_fp** depends on the *floating-point partial sum number (FPSN)*, the latency of the 32-bit floating-point addition. The sizes of **pacc_int** and **pacc_fp** are equal to $24 \cdot SIMD\_width \cdot IPSN$ bits and $8 \cdot SIMD\_width \cdot FPSN$ bits, respectively. The accumulation registers are organized as vector registers. For a detailed description of their structure and usage, the reader is referred to Section 3.3.3.

2.2 CSI-Instruction Operands and Results

An operand of a CSI instruction can be contained either in the CSI register space (a *register operand*) or in the CSI memory space (a *storage operand*). A register operand, therefore, can be one of the following registers: a scalar (integer or floating-point) register of the underlying ISA, a control registers belonging to a SCR/MSCR-set, the stream status register, the accumulator register *pacc_int*, or the accumulator register *pacc_fp*. A CSI storage operand (also called a *memory* operand) can be either a *CSI arithmetic stream* or a *CSI bit stream*. The CSI arithmetic streams are the streams of 8-, 16-, or 32-bit elements which represent arithmetic data. They are contained in storage at a certain regular sequence of addresses. The format of arithmetic streams is described in detail in Section 2.2.1. The CSI bit streams, which are also referred to as *CSI mask streams*, are streams of individual bits stored consecutively in memory. The format of the CSI bit streams is described in Section 2.2.2. The addresses of the elements of a CSI storage operand must be aligned on certain integral boundaries which

Data type	Mnemonic	Size in bits	Alignment
Bit	*b1*	1	1-byte
Binary integer			
- 8-bit signed	*s8*	8	1-byte
- 8-bit unsigned	*u8*	8	1-byte
- 16-bit signed	*s16*	16	2-byte
- 16-bit unsigned	*u16*	16	2-byte
- 32-bit signed	*s32*	32	4-byte
- 32-bit unsigned	*u32*	32	4-byte
Floating-point			
- 32-bit IEEE 754	*f32*	32	4-byte

Table 2.1: Data types of CSI stream elements.

depend on the element type. The allowed element data types and the corresponding alignment requirements are summarized in Table 2.1. If the appropriate alignment requirements for a storage operand of an instruction are not satisfied, a specification exception is recognized when the instruction is executed. An instruction specifies a storage operand (i.e., an arithmetic or bit stream) by designating the SCR- or MSCR-set that describes this operand. An instruction must designate a valid register or valid SCR-set (MSCR-set) for each operand. Otherwise, a specification exception is recognized. Finally, we remark that if the elements of an arithmetic stream represent signed integer numbers, the two's complement convention is used.

2.2.1 CSI Arithmetic Streams

A CSI arithmetic stream consists of 8-bit, 16-bit, or 32-bit elements which are contained in storage and represent arithmetical data. The elements are organized in groups that consist of up to 4 consecutively stored bytes. The stream is formed by groups that have the same size and follow a two-dimensional strided access pattern. The stream data which follows such an access pattern forms a sub-matrix of a larger matrix. Such streams commonly arise in multimedia applications when, for example, an operation has to be performed on a sub-image of a larger image. Stream elements are accessed in a certain regular access pattern. In this section we describe the format of an arithmetic stream, i.e., the type of the access pattern which the elements may follow. The sequence is specified by the parameters *element size, group size, base address, horizontal stride, horizontal length, vertical stride*, and *vertical length*. In order to assist the reader in understanding the general format of a CSI arithmetic stream and the meaning of its parameters, we present a several examples prior to the exact definitions.

Examples. Figure 2.4 (a) presents an example CSI arithmetic stream, which has element size 2 (bytes), group size 2 (bytes), base address 18, horizontal stride 4 (bytes), horizontal length 4 (elements), vertical stride 32 (bytes), and vertical length 4 (rows). The stream thus contains 12 16-bit elements. The address sequence at which the elements are accessed is $18, 22, 26, 30, 50, 54$, etc. Figure 2.4 (b) presents a different stream which has the same parameters except of the element size, which is 1 byte, and the horizontal length, which

is 8 elements. This stream, therefore, has 16-bit groups, each of which consists of two 8-bit elements. The total number of groups is the same as for the first stream: 12, but the total number of elements is equal to 24. The address sequence accessing the elements is $18, 19, 22, 23, 26, 27, 30, 31, 50, 51$, etc.

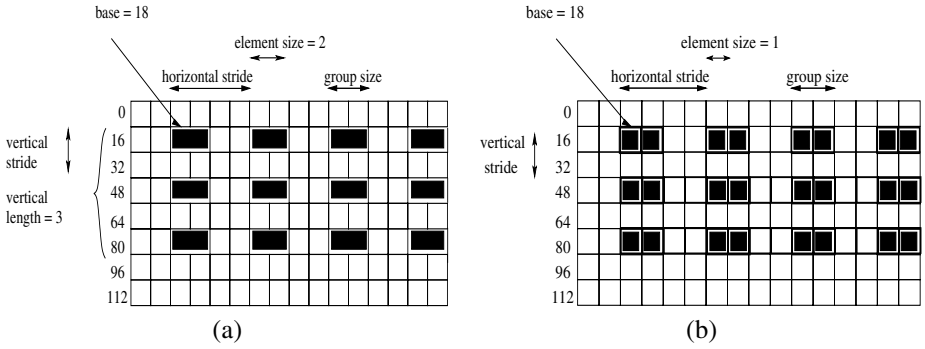


Figure 2.4: Two arithmetic streams with all parameters being the same, except for *element size* and *horizontal length*.

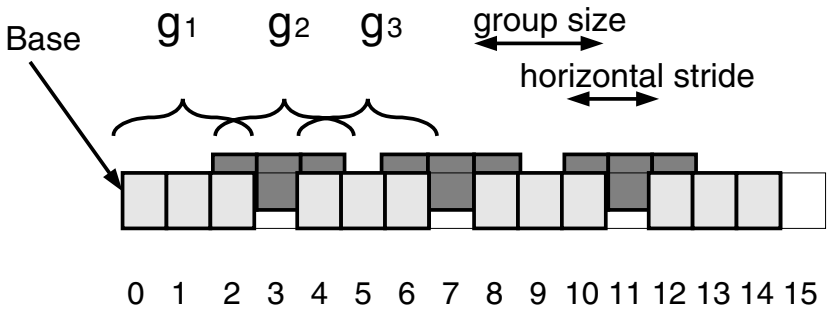


Figure 2.5: Examples of a CSI streams with overlapping groups.

We note that the horizontal stride (in bytes) can be less then the number of bytes in a group. In such case, the groups will overlap and a byte of data can belong to several groups. Consider, for example, the stream depicted in Figure 2.5. The stream has a group size of 3 bytes, base address zero, horizontal length 7, horizontal stride 2, vertical length 1, and vertical stride zero, i.e., the stream is formed by a single row consisting of 7 3-byte groups. The first group, g_1, consists of bytes $0, 1$, and 2, the second group, g_2, consists of the bytes $2, 3$, and 4, the third group, g_3, of bytes $4, 5$, and 6, and so forth. Therefore, bytes $2, 4$, etc. belong to two different groups. The stream elements are the 1-byte values which are accessed according to the address sequence $0, 1, 2, 2, 3, 4, 4, 5$, etc. Note that they are not located at constant stride and if the horizontal stride would have been 1, they wouldn't even be consecutive. Such situations, when the operation should be performed on overlapping groups of elements, are common in multimedia. Examples of such arithmetic streams are the ones arising in the

convolution of an image with an $n \times n$ matrix, which is frequently used in image-processing, in the 2:2 upsampling/downsampling routines employed in MPEG-2 codecs, and in many other kernels. In fact, the notion of group is introduced to allow instructions to access such streams. We now present the definitions of a CSI arithmetic stream and its associated stream parameters. The arithmetic streams will be defined in such a way that both types of example streams presented above, i.e., streams with 2-dimensional strided access pattern and streams with overlapping groups of consecutive bytes, will fit in the definition.

Definitions. A *CSI arithmetic stream* is a sequence S of data items $s_{i,j,k}$ which have the same arithmetic data type $t \in \{u8, s8, u16, s16, u32, s32, f32\}$ and are fetched from a lexicographically ordered finite sequence of natural numbers (addresses) $A = (a_{i,j,k})$. This address sequence is parametrized by natural numbers *Base, VStride, HStride, VLength, HLength, GrSize,* and *ElSize* according to the following formulas:

$$a_{i,j,k} = \mathit{Base} + i \cdot \mathit{VStride} + j \cdot \mathit{HStride} + k, \text{ where}$$

$$0 \le i < \mathit{VLength},\ 0 \le j < \mathit{HLength}\frac{\mathit{ElSize}}{\mathit{GrSize}},\ 0 \le k < \frac{\mathit{GrSize}}{\mathit{ElSize}}.$$

The address of the first element, $a_{0,0,0}$ is referred to as the stream *Base*. The sequence of elements $\{s_{i,j,k} |\ 0 \le k < \frac{\mathit{GrSize}}{\mathit{ElSize}}\}$ is called the (i,j)-th *group* of the stream S. The parameter *ElSize* (*element size*) describes the size of a stream element in bytes and, therefore, is required to be equal to 1, 2, or 4. If *ElSize* is equal to 2 or 4, it is required that $\mathit{GrSize} = \mathit{ElSize}$. If *ElSize* is equal to 1, the *GrSize* is required to be 1, 2, 3, or 4. Therefore, a group consists of *GrSize* consecutive bytes, $1 \le \mathit{GrSize} \le 4$, and represents either a single *GrSize*-byte element or *GrSize* consecutive 1-byte stream elements (except for the case $\mathit{GrSize} = 3$, which always corresponds to a group of three 1-byte elements). The sequence of elements $\{s_{i,j,k} |\ 0 \le j < \mathit{HLength}\frac{\mathit{ElSize}}{\mathit{GrSize}},\ 0 \le k < \frac{\mathit{GrSize}}{\mathit{ElSize}}\}$ is called the i-th *row* of the stream S. It consists of all the (i,j)-th groups, the total number of which is denoted as *HGroupLength* and called the *horizontal group length* of the stream. Obviously, $\mathit{HGroupLength} = \mathit{HLength}\frac{\mathit{ElSize}}{\mathit{GrSize}}$. The number of elements in a row, denoted as *HLength*, is referred to as the *horizontal length* of a stream. It is required to be such that *HGroupLength* is guaranteed to be integer. The addresses of the first elements of two consecutive groups, $a_{i,j,0}$ and $a_{i,j+1,0}$, differ by the constant *HStride*, called the *horizontal stride*. The addresses of the first elements of the first groups of the two consecutive rows, $a_{i,0,0}$ and $a_{i+1,0,0}$, differ by the constant *VStride*, which is called the *vertical stride* of the stream. The number of rows in the stream, *VLength*, is referred to as the *vertical length*. We remark once again that the elements of a CSI arithmetic stream are lexicographically ordered. A CSI instruction which operates on an arithmetic stream will process the elements according to this order. Therefore, the element $a = s_{i,j,k}$ will be processed earlier than the element $b = s_{x,y,z}$ if $i < x$, or if $i = x$ and $j < y$, or if $i = x, j = y,$ and $k < z$. In other words, a will be processed earlier than b either if a lies in the row that precedes that of b, or if a and b lie in the same row and the group of a precedes that of b, or if a and b lie in the same group and a precedes b in this group.

To specify an arithmetic stream, its parameters are stored in the stream control registers that belong to the same SCR-set. An arithmetic stream is determined by the data type of

its elements and the address sequence. The data type is specified by setting appropriately the *ElSize, FP,* and *Sign* fields of the *Misc* control register of the SCR-set. The address sequence is determined by the parameters *GrSize, ElSize, Base, HStride, HLength, VStride,* and *VLength*. It is specified by setting appropriately the corresponding control registers of the set (for *GrSize* and *ElSize* – by setting the corresponding fields of the **Misc** register). An instruction addresses an arithmetical stream operand by specifying the SCR-set that describes it. During the execution of a CSI instruction which accesses an arithmetical stream, the base address contained in the **Base** register of the SCR-set, as well as the values contained in the **CurrCol** and the **CurrRow** registers, are modified as successive stream elements are accessed. Suppose that at current point of the instruction execution, the next stream element to be processed is $s_{i,j,k}$. Then the current value of **Base** will be set to $a_{i,j,k}$, that of **CurrCol** – to $\frac{GrSize}{ElSize} \cdot j + k$, and that of **CurrRow** – to i. Such automatic updating of the address of a stream element to be processed, and of its position within the stream, facilitates the restart of a CSI instruction after an interrupt, from the point at which the interrupt has occurred. For more details concerning the interrupt-handling mechanisms in CSI, the reader is referred to Section 2.3.3.

2.2.2 CSI Bit Streams

A CSI bit stream, also referred to as the CSI mask stream, consists of individual bits in storage (i.e., in memory) that form a continuous sequence. A CSI bit stream is characterized by two parameters: the *base address* and the *length*. The base address is the address of the byte in storage that contains the first element of the stream. The first element of the stream is required to be the 0'th bit, i.e., the most significant bit of the byte located at the base address. The number of elements in the stream is called the length of the stream. Note, that the length of a bit stream is not required to be a multiple of 8 and, therefore, several bits on the right in the last byte of the stream may not belong to the stream. CSI bit streams are usually used as source mask operands in the *masked* CSI arithmetic instructions, which are executed when the *mask mode* is enabled. Employing such instructions allows CSI to implement the loops with data-dependent control. Bit streams are commonly produced using comparison instructions.

An instruction accesses a bit stream by specifying an MSCR-set. When a CSI instructions processes a bit stream, the starting address in the **Base** register of the MSCR-set is implicitly increased by the number of bytes accessed during execution, and the **CurrElem** is incremented by the number of elements processed. At any point of execution, the **Base** register contains the address of the byte containing the bit that has to be processed, and the **CurrElem** register contains the number of this bit. The automatic update of the registers facilitates the restart of an interrupted CSI instruction from the point of interruption. Details concerning the interruption-handling mechanisms in CSI can be found in Section 2.3.3.

2.2.3 Conditional Execution

Data-parallel applications, such as multimedia applications, operate on long streams of data and usually perform the same operation on each stream element. There are situations, however, in which the operation to be performed on each element is dependent on the element

itself, as the example presented in Figure 2.6 shows. The necessity to be able to handle such

```
for(i=0; i<N; i++){
   if(A[i] > 0.0)
      A[i] = A[i]+B[i];
}
```

Figure 2.6: Simple loop with data-dependent control.

cases has been recognized in the vector processing domain. A solution commonly used there is *masking*. In this section we present the concept of masking and its implementation in CSI as well as a different mechanism, based on stream reorganization operations, such as conditional splitting and merging of CSI streams. This will allow us to implement more complex conditional loop constructs efficiently. This technique is also useful for simple loops similar to the one presented above in case when the number of elements on which the computation has to be performed is relatively small compared to the total number of elements.

Masking

We now briefly explain the concept of masking using the code presented above as an example. In a vector architecture, such as the *IBM System/370 Vector architecture* [4], groups of consecutive elements of the arrays `A` and `B` will be represented as *arithmetic vectors*. For brevity, we will refer to these vectors simply as `A` and `B`. The code fragment can then be implemented as follows. First, a *mask vector* is produced: the i-th element of this vector is 1, if the corresponding condition is true (i.e., if `A[i] > 0.0`), and 0 otherwise. This operation can be implemented using the *vector compare instruction*. Thereafter, a *masked* arithmetic vector instruction is executed which uses vectors `A`, `B`, and the mask vector as the source operands and vector `A` as the destination operand. In present case, the *masked vector add* instruction will be executed. It operates as follows: if the i-th element of the mask vector is 1, then the i-th elements of vectors `A` and `B` are added and the result is written to the i-th element of vector `A`. Otherwise, a *no-op* is executed and the i-th element of vector `A` is left unchanged.

The concept of masked execution is adopted in CSI with minor modifications. CSI arithmetic streams correspond to arithmetic vectors and CSI mask streams (bit streams) correspond to mask vectors. Most of the CSI arithmetic instructions use the format that explicitly designates four operands, namely, two *source data operands*, one *source mask operand*, and one *destination data operand*. Each of the source and destination data operands can be a CSI arithmetic stream or a scalar register (general-purpose or floating-point) and are designated using the corresponding SCR-set or register identifiers. The mask stream operand is a CSI bit stream designated via the corresponding MSCR-set. The *CSI execution unit*, i.e., the hardware unit which implements CSI instructions, can operate in two modes: the *masked stream mode* and the *normal (unmasked) stream mode*. The mode of operation is determined by the stream mask-mode bit of the stream status register (SSR). This bit can be set to 0 or to 1 using the `csi_mt_ssr` instruction. When the stream mask-mode bit of the SSR is set to zero, the source mask operand is ignored and the instructions are *unmasked*. Unmasked instructions operate normally, performing the designated operation on all stream elements. If

the stream mask-mode bit of the SSR is set to 1, the CSI instructions that are subject to the masked stream mode are *masked*. Masked arithmetical CSI instructions process operand elements only if they are in positions which correspond to mask bits that are ones. In positions which correspond to zero-valued mask bits no arithmetic or operand-access exceptions are recognized and the corresponding target locations (the storage locations for CSI arithmetical stream operands or the scalar registers for register operands) remain unchanged.

The instructions that are subject to the stream mask-mode are those belonging to the *IM* instruction class (see Section 3.2). For example, all the simple arithmetic and logical instructions described in Section 3.3.1, such as `csi_add`, `csi_mul`, etc., belong to the IM instruction class and, therefore, are subject to the mask mode. We note here that there are instructions that use the CSI bit streams as their source or destination operands but are not subject to the stream mask-mode. The execution of such instructions is independent of the value of the stream mask-mode bit of the SSR. Examples are the stream compare instruction `csi_cmp`, logical instructions performed on the CSI bit streams, such as `csi_and_bitstr`, and instructions for splitting and merging of arithmetical streams, such as `csi_split` and `csi_merge` presented later in this section. These instructions are declared to be independent of the mask-mode status because they have to process CSI bit streams, but the operations they have to perform do not fit into the description of the processing in mask-mode presented above. Consider, for example, the `csi_cmp` instruction. This instruction compares corresponding elements of two arithmetic streams and, depending on the value of the *modifier* that specifies the type of comparison and the outcome of the comparison, sets the corresponding element of the CSI bit stream to 1 or to 0 (see Section 3.3.2). The instruction uses all four operand fields: two field to designate source arithmetic streams, one field to encode the modifier, and one field to designate the destination bit stream. If the instruction would be dependent on the stream mask-mode, the following would happen. When the mask-mode would be disabled, the instruction would ignore the mask operand. When the mode would be enabled, it would be able to access the mask operand, i.e., the destination bit stream. However, this wouldn't make sense since, according to the description of operating under the stream mask-mode, the instruction would have to use the designated bit stream as a source while, in fact, this bit stream has to be used as the destination.

Stream Reorganization Techniques

In this section we briefly describe several operations which provide a more efficient implementation of certain conditional loops. The operations presented in this section perform conditional extraction/insertion of data from one arithmetic stream into another, conditional splitting of an arithmetical stream into two, and conditional merging two arithmetic streams into one.

Stream Extraction and Insertion. Consider once again the code fragment presented in Figure 2.6. Masked execution enables the implementation of that loop using just two CSI instructions: one `csi_cmp` and one `csi_add`, which is executed in stream-mask mode. Recall that during the execution of masked `csi_add` a no-op is executed for the positions that correspond to the zero-valued mask stream elements. If the number of such positions is

relatively large, the execution time of the masked instruction is dominated by no-ops, which do not perform any useful computations.

CSI provides a mechanism that avoids executing no-ops in the implementation of loops which contain conditional `if`-constructs similar to those presented above. Those elements for which the `if` condition is true can be extracted from an arithmetic stream. Thereupon, these elements are stored in a shorter temporary stream which can be processed without any no-ops. The computed values are inserted back into the original stream at the required positions. In order to implement such mechanism, CSI provides two instructions: `csi_extract` and `csi_insert`. Below we give a brief description of these instructions. For more details one the reader is referred to Section 3.3.4.

- The instruction `csi_extract` has the format
  `csi_extract SCRSk, SCRSi, MSCRSj.`
  The operands `SCRSi` and `MSCRSj` designate the SCR-set that describes the input arithmetic stream and the MSRS-set that specifies the input mask stream, respectively. The length of the input mask stream and that of the input arithmetic stream must be equal. The instruction performs the following operation. Each element of the mask stream is checked. If the element is zero, the corresponding element of the arithmetic stream is ignored as well as any access exceptions recognized during access. If the mask element is 1, the element of the input arithmetic is *extracted*, and stored in the output arithmetic stream described by the SCR-set `SCRSk`. This instruction should only be executed when the stream mask-mode is enabled.

- The instruction `csi_insert` has the format
  `csi_insert  SCRSk, SCRSi, MSCRSj.`
  Operands `SCRSi` and `MSCRSj` designate the SCR-set that describes the input arithmetic stream and the MSRS-set that specifies the input mask stream, respectively. The SCR-set `SCRSk` describes the output arithmetic stream. The length of the mask stream must be the same as the length of the output stream. The instruction is executed as follows. It checks elements of mask stream one after another. If the mask element is zero, the corresponding element of the output arithmetic stream, designated by SCR-set `SCRSk`, is skippedand the current element of the input stream remains the same. If the mask element is 1, then the current element of the input stream, e_i, is *inserted* into the output stream at the position which corresponds to the mask element. Then e_i is discarded and current element of the input stream is set to e_{i+1}, i.e., e_{i+1} will be the next element of the input stream considered for insertion. The instruction should be executed only when the stream mask-mode is enabled.

Stream Splitting and Merging. CSI instructions that perform conditional stream extraction and insertion are intended to be used for implementing loops with `if`-constructs. However, when a loop which contains an `if-else`-construct has to be implemented, employing such instructions has certain drawbacks, as the following example demonstrates. Consider the loop presented in Figure 2.7. This loop can be implemented in CSI using conditional extraction and insertion instructions as described below. First, the mask stream for the condition `A[i] > 0.0` is generated. Then, the elements of the arrays `A` and `B` that correspond to the ones in the mask stream, i.e., the elements for which the condition is satisfied, are extracted

```
for(i=0; i<N; i++){
   if(A[i] > 0.0)
      A[i] = A[i]+B[i];
   else
      A[i] = A[i]-B[i];
}
```

Figure 2.7: A loop with *if-else*.

and stored into two temporary streams. These temporary streams are then added in unmasked mode and the results are inserted back into array `A` at the appropriate positions. After this, another mask stream is created by negating each element of the mask stream obtained earlier. This second mask stream corresponds to the `else`-part of the `if-else` statement. The elements of `A` and `B` that should be processed in the `else`-part are then extracted and stored in two temporary streams. This is done using two `csi_extract` instructions. Then the temporary streams are subtracted element-wise and the results are inserted back into array `A` under control of the second mask stream.

We observe that each of the arrays `A` and `B` (or, more precisely, each of the arithmetic streams that represent these arrays) is 'scanned' twice. During the first 'scan', the arrays elements that should be processed in the `if`-part of the conditional part are extracted, and during the second one the elements that correspond to the `else`-part are extracted. CSI provides a `csi_split` instruction which is intended to perform such extraction more efficiently, just in a single 'scan'. Furthermore, we observe that, in the extract-insert implementation of the loop described above, the destination stream representing array `A` is 'scanned' twice also during the insertion of the results. CSI provides the `csi_merge` instruction that allows to insert the results in a single 'scan'. Below, we give a brief description of the stream splitting and merging instructions.

- `csi_split` instruction has the format:
  `csi_split SCRSk, SCRSl, SCRSi, MSCRSj.`
  SCR-sets `SCRSi`, `SCRSk`, and `SCRSl` designate one input and two output arithmetic streams, respectively. `MSCRSj` designates the input mask stream. The instruction operates as follows. Each element of the input arithmetic stream is read. If the corresponding entry of the mask stream is 1, the element is written to the output arithmetic stream described by `SCRSk`. If the corresponding mask streams entry is 0, the element is written to the second output stream, which is described by the SCR-set `SCRSl`. The instruction is **not** subject to the stream-mask mode. For correct operation, the length of the input arithmetical stream, the length of the mask stream, and the sum of the lengths of the output streams should be equal.

- `csi_merge` instruction has the format:
  `csi_merge SCRSl, SCRSi, SCRSj, MSCRSk.`
  SCR-sets `SCRSi`, `SCRSj`, and `SCRSl` designate two input and one output arithmetic streams, respectively. `MSCRSk` designates the input mask stream. The instruction operates as follows. An element of the mask stream is read. If it is 1, then the first element of the first input stream (which is described by `SCRSi`) is removed from this

stream. Otherwise, the first element of the second input stream (described by `SCRSj`) is removed. The removed data element is then written to the destination stream. The instruction is **not** subject to the stream-mask mode. For correct operation, the length of the mask stream, the sum of the lengths of the input arithmetic streams, and the length of the input arithmetical stream should be equal.

2.3 Interruption Handling in CSI

In this section we present the mechanisms that allow most CSI instructions to be interrupted by the supervisor program (the operating system) during their execution and then to be restarted from the point of interruption, after the exceptional condition that caused the interruption is handled by the operating system and control is returned to the user program. Providing a mechanism for resumption of execution from the point of interruption is important, because a single CSI instruction can process a very large number of elements and, therefore, can be rather long running. If after an interrupt the instruction would have to be executed from the very beginning, performance would degrade severely. The CSI architecture is somewhat similar to vector architectures, where a single instruction performs computations on a long sequence of elements. Therefore, vector instructions can be rather long-running as well and, hence, require a restart mechanism in order to avoid severe performance degradation in the presence of interrupts. Such a mechanism has been employed, for example, in the *IBM System/370 Vector Architecture* [4] and in its newer version, the *IBM System/390 Vector Architecture*[13]. In these architectures, execution of a vector instruction is represented as execution of a sequence of units of operation (UOPs), where a UOP is defined to perform the specified operation on the operand elements located at corresponding positions of the operand vectors. Interrupts are allowed to occur between UOPs. To allow instructions to be restarted from the point of interrupt, the system keeps track of which UOP is currently being performed. The current UOP is identified by a certain register which is automatically updated during the instruction execution. The instruction restart mechanism in the CSI architecture is based on the same ideas and allows CSI instructions to be interrupted and restarted.

2.3.1 Supervision

A general-purpose uniprocessor system usually has to be able to operate in multiprogrammed mode. This means that several user programs can be concurrently active and have to share system resources such as processor time, memory space, and peripheral devices. Therefore, such a system contains a *supervisor program* (also called the *operating system*, or *OS*) that administers resource sharing. The supervisor may need to interrupt a user program in certain cases, for example, when system resources should be dedicated to another user program or when an external event requires supervisor intervention, or when the currently executed user program's instruction causes some exceptional situation, such as division by zero. Supervisor programs handle such situations by seizing control from the user program and performing the necessary actions, such as activating another user program, handling external events or exceptional situations caused by the interrupted program. After this, the OS may return control to the interrupted program. The transfer of control from a user program to the supervisor is

referred to as an *interruption* or, simply, an *interrupt*. The code executed by the supervisor when handling the interruption is called the *interruption handling routine*, the *trap-handling routine*, or simply the *interruption (trap) handler*.

Since CSI is an extension to a general-purpose ISA, CSI assumes presence of a supervisor program (OS). It is also assumed that the interruption mechanism of the OS is implemented as follows. The underlying general-purpose ISA contains the *Process Status Word* register, or *PSW*. PSW includes the address (also called the *program counter (PC)*) of the current instruction, the *Process Identifier (PID)*, which is a unique number assigned to the user program by the OS and controlling, for example, the memory access permission of the program, and other information used to control instruction sequencing and to determine the state of the CPU. PSW governs the program currently being executed and is called the *current PSW*. The CPU has an interruption capability, which permits the CPU to switch rapidly to another program (usually, the supervisor program) in response to exceptional conditions and external events. When an interruption occurs, the CPU places the current PSW in an assigned storage location, called the old-PSW location, for a particular class of interruption. The CPU fetches a new PSW from a second assigned storage location. This new PSW determines the next program to be executed. When it has finished processing the interruption, the interrupting program may reload the old PSW, making it again the current PSW, so that the interrupted program can continue.

We note here that when a program is interrupted and then restarted, its state (also referred to as the *program's context*) has to be restored to be the same as at the moment of interruption in order to ensure correct execution. The program's state consists of the memory state and the register state. In a virtual memory environment, the memory state of a program is determined by the PID field of the PSW. Therefore, to restore the program's context, it is sufficient to restore its register state, which consists of all the architecture registers, such as PSW, scalar registers and the CSI registers (SCRs, MSCRs and the SSR). This is usually done by saving the registers at the point of interruption at a certain predefined memory location called the *save area*, and reloading them when control is returned to the interrupted program. The CSI architecture contains large number of registers: 96 SCRs, 48 MSCRs, two accumulator registers, and the 128-bit SSR. CSI provides a mechanism which permits reducing the overhead associated with saving and restoring the SCRs and the MSCRs. As it was presented in Section 2.1, the stream status register (SSR) contains the *in-use* bit for each SCR-set and each MSCR-set. Only the SCR(MSCR)-set for which the in-use bit is set to 1 has to be saved at the point of interruption and restored when control is returned to the interrupted program.

We assume that the OS recognizes several classes of interruptions, such as *I/O*, *supervisor call*, *program*, and others. Different classes of interruptions correspond to different types of situations which require the OS intervention. For example, the *input/output (I/O) interruption* provides means by which the CPU responds to conditions originating in I/O devices; the supervisor call interruption is often performed when the supervisor decides to interrupt the current user program in order to activate another user program. Each class of interruptions has a distinct pair of old-PSW and new-PSW locations permanently assigned in physical memory. In the remainder of this chapter we discuss the *program interruptions* caused by CSI instructions, i.e., the interruptions caused by the exceptional conditions that arise during execution of a user program's CSI instructions, such as floating-point exponent overflow. We present the mechanism that allows us to interrupt the CSI instructions during their execution

and to restart them from the point of interruption after the interrupt is handled. Although the mechanism will be described in the context of program interrupts, it can be used also for other types of interrupts, such as an I/O interrupt or a supervisor call.

2.3.2 Effect of Program Interruptions During Execution

We now briefly describe the actions taken during a program interrupt. The general description of interruption handling presented above, in case of a program interruption, looks as follows.

1. *Jump to the interrupt-handler.*
   First, the current user program's PSW is stored at a certain predefined old-PSW location. Then the CPU fetches new PSW (i.e., the supervisor's PSW) from another predefined storage location.

2. *Store the Exception Code*
   As defined above, program interrupts are caused by exceptional situations (also called *exceptions*) which arise during the execution of a user program instruction. It is assumed that the general-purpose architecture which underlies CSI defines a unique identifier called the *exception code* for each type of exception. It is also assumed that at the moment when an exception is recognized and the program interruption is started, the exception code is stored in a predefined location of the PSW, called the *PSW exception-code field*, or *EXCode* field. We note that execution of a CSI instruction may cause exceptions which are not defined in the underlying ISA. It is assumed that each of these exceptions is assigned a unique exception code (distinct from the exceptions codes defined in the underlying ISA, and from each other) and this code is stored in the EXCode field of PSW as well. It is also assumed that a certain part of the PSW, called *exception mask field*, is used to enable and disable certain kinds of exceptions. Each of such exception types is assigned a separate bit of the field. User programs may disable a certain exception type by setting the corresponding bit to zero. If the bit is set to zero, then the corresponding type of exception is ignored and does not cause program interruption. Furthermore, this mechanism is useful for the system programmer who implements the interruption-handling routines: subsequent interruptions can be disallowed by the new PSW introduced by an interruption.

3. Store the Stream Identifier (optional).
   If the exception concerns the operand access exception, the identifier (i.e., the number of the corresponding SCR/MSCR-set) of the operand stream, which contains the interruption-causing element, is stored in the SID field of the stream status register. The information stored in SID, together with the EXCode field of PSW, allows the interruption routine to handle the exception. This information can also be used for debugging purposes, especially in the *CSI sequential* mode of operation, which is described later in this section.

4. *Execute the interruption-handling routine.*
   Before handling the exception, the interruption routine may store some parts of the CSI register state and parts of the register state of the underlying ISA. The stored parts

of the state should be sufficient to restore the state of the interrupted program (the so-called *program context* at the point of interruption. Then, the routine handles the exception and determines how the further execution of the interrupted program should proceed. If the user program was interrupted during the execution of a CSI instruction then, depending on the type of exception, the execution of the program can be continued by processing the stream elements that were being processed at the point of interruption (operations performed on these elements are referred to as the *current unit of operation*; a detailed definition of a unit of operation is presented later in this section). The execution may be also resumed by processing the elements that follow the ones processed in the current unit of operation. For certain kinds of exceptions, the interrupt-handler may also decide to cancel further execution of the interrupted instruction and to resume the interrupted program by executing the next instruction. In the worst case, the interrupted program may be terminated. The routine determines the way the interrupted program should be resumed by setting the program counter in the old PSW and by adjusting the stream operand parameters that determine the unit of operation that will be performed by the interrupted instruction when it is re-executed. The detailed description of these actions is presented later in this section Finally, the routine reloads the register state that was saved at the beginning of the interruption thus restoring the context of the interrupted program.

5. *Return to the Interrupted Program.*
   After the trap-handling routine is executed, the OS returns control to the interrupted program by loading the PSW from the old-PSW location.

2.3.3 Interruptible CSI Instructions

Most CSI instructions that operate on CSI arithmetic or bit streams are interruptible and consist of multiple UOPs. After the last stream element has been processed without a program interruption, the instruction is completed in the *final unit of operation*. This final UOP consists of advancing the program counter (located in the corresponding field of PSW) to the next instruction. Performing the final UOP cannot create any program-interruption. If a program interruption occurs while processing the last element of a stream, the instruction remains partially completed, because the final unit of operation has not yet been carried out. Therefore, all elements of a stream, including the last one, are processed in the same way. Only the final unit of operation of advancing the program counter is performed if the operand streams are specified in such a way that no stream elements have to be processed. A non-final unit of operation for a given instruction is determined by the instruction itself and by the current mode of operation, which can be set to either *CSI parallel mode* or to *CSI sequential mode*. For example, when the CSI sequential mode is enabled, a unit of operation for most of the instructions consists of processing a single stream element. More details concerning the definitions of a UOP for different modes are presented later in this section.

CSI instructions that operate on (arithmetic or mask) stream operands are always executed sequentially and the UOPs are processed in their natural order determined by the stream addressing. For an arithmetic stream processed in sequential mode, this means, for example, that the UOP that processes a certain element is performed before all other units

of operation, which process the elements from the previous rows and the preceding elements from the same row, are done. We remark that preceding elements from the same row may have a larger address in case of overlapping groups of elements for arithmetic streams. Any exceptions resulting from the execution of a stream instruction are recognized sequentially as well. A program interruption always occurs due to the first exception which is recognized and for which interruption is allowed (i.e, the corresponding exception-mask bit in PSW is set to 1).

In order to guarantee the correct execution of a program in the presence of interrupts and to allow an interrupted instruction to be continued from the point of interruption, the following rules are imposed by CSI.

1. *Changes to register contents which have to be made by the interrupted CSI instruction beyond the point of interruption, are not yet performed at the time of an interruption.* The registers mentioned in this rule are the registers that are modified implicitly during the execution, such as **Base, CurrCol**, and **CurrCol** SCRs of the SCR-set describing an operand stream of the instruction, or a scalar register designated as the destination operand.

2. *Changes to memory locations, which have to be made by the interrupted instruction beyond the point of interruption, are allowed to occur for memory locations beyond the point of interruption, i.e., for locations that correspond to stream elements that follow the element being processed in the current unit of operation.* For example, such changes may happen in the situations similar to the following one. Assume that CSI is implemented in a system with virtual memory and the CSI execution unit is interfaced to a virtually-addressed cache. Assume that the `csi_add` instruction is executed in sequential mode and that the current unit of operation processes the ith elements of the operand streams. The outcome of the unit of operation will be known after the outcome of the store access to the ith element of the destination stream is determined. Suppose that this store access misses in the L1 cache and, furthermore, causes an access exception. If storing the $(i+1)$-th element of the destination stream results in a cache hit, it might already have been performed by the time the data for the ith element is retrieved from lower levels of the memory hierarchy and the exception is recognized. Note, that in such case the unit of operation that corresponds to the $(i+1)$-th element is considered as not yet performed.

3. *Changes to memory locations beyond the last element specified by the instruction and to locations for which access exceptions exist are not yet performed at the point of interruption.*

4. *Changes to memory locations or register contents which are due to be made by instructions following the interrupted instruction have not yet been made at the time of interruption*.

Units of Operation in the Sequential and the Parallel Processing Modes

As was mentioned above, the definition of the unit of operation for a given CSI instruction depends on instruction itself, as well as on the current mode of operation of the CSI execution

unit. CSI execution has two mutually exclusive modes of operation: the *CSI sequential mode*, which is also called the *single-element mode*, and the *CSI non-sequential mode*, which is also called the *parallel mode*. Which of these two is the current one is determined by the *CSI sequential stream mode bit* of the stream status register. If the bit is set to 1 then the execution unit operates in the CSI sequential stream mode, otherwise, it operates in the CSI parallel mode. Note that these two modes are orthogonal to the stream mask/non-mask mode. The sequential mode allows us to identify the cause of an interruption precisely and is provided primarily for debugging purposes. The parallel mode is provided for high performance and does not allow us to identify the cause of an interruption exactly.

Definitions of a unit of operation for less common types of instructions are presented in their individual descriptions (see Chapter 3). Now we describe the operations that are performed in a single unit of operation for the most common type of CSI instructions, namely, the CSI arithmetic instructions that perform simple arithmetic operations on all pairs of corresponding elements of two input arithmetic streams and stores the result into the corresponding location of the destination arithmetic stream. We use `csi_add` as an example of such a common instruction and assume that two input arithmetic streams, the (optional) input mask stream, and the output arithmetic stream are designated by the SCR/MSCR-sets `SCRSi`, `SCRSj`, `MSCRSk`, and `SCRSl`, respectively.

- If the sequential mode of operation is enabled, a unit of operation consists of processing a single stream element. More precisely the following operations are performed in the ith unit of operation, depending on the mask-mode status. If the mask mode is disabled, the i-th elements of the input streams described by `SCRSi` and `SCRSj` are loaded (mask stream is ignored), added, and then stored as the i-th element of the destination stream. If the mask mode is enabled, the i-th elements of the input streams `SCRSi`, `SCRSj`, and `MSCRSk` are loaded. If the loaded mask stream element is 1, then the i-th elements of `SCRSi` and `SCRSj` are added and stored at the appropriate position in the destination stream. Otherwise, the store is not performed, and the i-th element of `SCRSl` remains unchanged. Furthermore, any operand access exceptions or arithmetic exceptions are not recognized and do not cause program interruption.

- In CSI parallel mode, a unit of operation usually consists of processing several consecutive stream elements in parallel. The number of elements processed in parallel is determined by the *processing width* of the CSI unit and by the *processing size* of the elements. The processing width is equal to the width of the datapath of the functional subunits (adders, multipliers, etc) of the CSI execution unit. It is expressed in bytes and is implementation dependent. The processing size of the elements is the number of bytes that are required to represent a stream element in its processing format. This parameter is determined by the value of the *ProcessSize* field of the **Misc** control register of the SCR-set that describes the stream.
  The number of elements processed in parallel in a unit or operation is defined as the quotient of the processing width and the processing size.
  Assume, for example, that for the `csi_add` instruction, the arithmetical stream elements require 16 bits (2 bytes) for their processing representation and that the processing width of the unit is 16 bytes. Then the number of elements processed in a single unit of operation is equal to $16/2 = 8$ elements, except for the unit that processes the

last few stream elements and that may require less operations. If stream mask mode is disabled, the ith unit of operation for the instruction consists, in general, of loading 8 consecutive elements starting from the $(8 \cdot i)$-th element of both arithmetic operands and then adding them pairwise and in parallel. The resulting 8 values are stored at the corresponding 8 consecutive locations of the destination stream. If stream mask mode is enabled, then the unit of operation consists of loading 8 consecutive elements of the input arithmetic streams and of the mask stream. For the positions that correspond to nonzero mask values, the input arithmetic elements are added and the result is stored in the destination stream. For the positions that correspond to zero mask values, the store is not performed and any arithmetic or operand access exceptions are not recognized.

Any exceptions recognized for the unit of operation are associated with the whole unit of operation, not with the one particular element that causes the exception. This means, for example, that if an exception occurs for an UOP, the particular element that caused this exception cannot be identified. This potentially causes the results to be imprecise, i.e., the results may differ from those obtained when the same instruction is processed in sequential mode. For example, an exception caused by a particular element may cause the corresponding unit of operation to be *inhibited*, in which case the destination stream locations remain unchanged for all the positions that belong to this unit of operation. In the sequential mode of operation, only the unit of operation of the exception-causing element will be inhibited and the corresponding destination location will remain unchanged, while the other locations will be modified.

The execution of instructions in sequential mode is likely to provide lower performance than execution in the parallel mode. It allows, however, to identify exactly at which position an exception and subsequent interruption have occurred and what the cause was. This is achieved by the following mechanism. At the point of interruption, the code of exception that caused this interruption is stored in the *EXCode* field of the PSW. Furthermore, if the exception is an operand access exception, the identifier (i.e., the number of the corresponding SCR/MSCR-set) of the operand stream, which contains the interruption-causing element, is stored in the SID field of the stream status register. Since the **Base, CurrCol** and **CurrRow** (**CurrElem**) registers of this SCR-set (MSCR-set) are not yet modified at the point of interruption, their contents identify the position of the element that causes the interruption. If the interrupt is caused by an arithmetic exception, only the exception code is stored at the EXCode field of PSW. This information, together with the copy of the currently executed instruction stored in the *Instruction Copy* field of the SSR, allows the interruption handler routine to identify the cause of exception and the corresponding destination location. Note that if the destination operand is a stream operand, the location of the particular element that corresponds to the current position can be determined by the values of the control registers. Using this information, the interrupt-handler may perform any desired result fix-up. The sequential mode of operation thus provides sufficient information for debugging. Is supposed to be used primarily for this purpose.

The parallel execution mode is provided for high-performance. This mode does not provide means to identify exactly the element that caused interruption. However, the information retained in this mode is sufficient to resume execution of an interrupted instruction from the point of interruption. Details are presented later in this section.

Instruction Restart Mechanism

Now we present in more detail the mechanism that allows an interrupted CSI instruction to be restarted from the point of interruption. This mechanism is based on the automatic update of stream operand parameters that determine the current unit of operation. The parameters of a stream operand are described by the contents of the stream control registers from the SCR(MSCR)-set that designates this stream operand. For an arithmetic operand stream, these parameters are the address of the first element that has to be processed in the current UOP, the number of the row to which this element belongs, and the position of the element within the row. These values are contained, respectively, in the **Base**, **CurrRow**, and **CurrCol** control registers of the SCR-set that describes the stream. In a similar way, for an operand bitstream, the parameters that determine the current UOP are the address of the byte that contains the first stream element (i.e, a single bit) which is processed in the current UOP, and the position of this bit within the bitstream (and, hence, within the byte). These values are contained in the **Base** and the **CurrElem** control registers of the corresponding MSCR-set.

The mechanism that allows a CSI instruction to be restarted from the point of interruption is based on the following rule:
For the operand streams, the operand parameters that determine the positions of the elements being processed in the current unit of operation, are not updated until the interrupt handler determines the outcome of this unit of operation. When the outcome is determined, the handler adjusts the parameters.
The parameters are adjusted depending on the outcome, which, in turn, is determined by the type of the interrupt-causing exception. In most cases, the operand parameters are adjusted, so that the same UOP or the next UOP of the same instruction will be performed upon return from the interrupt. For some types of instructions and exceptions, the resumption of the current instruction does not make sense. In such cases, the interrupt handler cancels the instruction by setting the program counter field of the old PSW to point to the next instruction of the interrupted program.

In general, a unit of operation of a CSI instruction can have one of the five following outcomes: *completion, inhibition, nullification, suppression*, or *termination*. Termination of an UOP causes termination of the instruction. If instruction is terminated, the contents of the storage locations and of the registers that might have been changed by the instruction in the current UOP are unpredictable. The program counter in the old PSW is set to the address of the next instruction, so that the execution of the interrupted program will resume with that instruction. Termination can occur only as the result of an hardware malfunction and will not be discussed further. In the following we describe how execution is resumed for the four other UOP outcomes by presenting the effect of the interruption on the instruction address in the old PSW stored during the interruption, on the (implicitly updated) operand parameters, such as **Base, CurrCol**, and **CurrRow** control registers, and on the result location for the current unit of operation.

1. *Completion.*
   The instruction address in the old PSW contains the address of the interrupted instruction. The result location for the current UOP contains the new result, which is defined depending on the type of the interrupt-causing exception. For example, the floating-point exponent overflow exception is defined to place the overflowed number in the

UOP Outcome	Program Counter at	Stream Parameters at	Result Location
Completed	Current instruction	Next UOP	Changed
Inhibited	Current instruction	Next UOP	Unchanged
Nullified	Current instruction	Current UOP	Unchanged
Suppressed	Next instruction	Current UOP	Unchanged

Table 2.2: Effect of different outcomes of a unit of operation.

result location. The parameters of operand streams are adjusted such that, if the interrupted instruction is re-executed, its execution is resumed with the next UOP.

2. *Inhibition.*
   The effect is the same as of completion, except that the result location for the current UOP remains unchanged. In other words, inhibition of an UOP means that this UOP performs a *no-op*.

3. *Nullification.*
   The instruction address in the old PSW designates the interrupted instruction. The result location for the current UOP remains unchanged. The operand parameters are adjusted such that, if the instruction is re-executed, execution is resumed with the current UOP. For example, if the CSI architecture is implemented in a processor with virtual memory and with a TLB (translation lookaside buffer) [26], the *TLB miss* access exception will cause nullification. Interruption occurs before performing any arithmetic operation on the stream elements that have to be processed in the current unit of operation. Since memory accesses and execution can be decoupled in a particular implementation of the CSI execution unit, access exceptions that nullify the current unit of operation may be recognized for stream elements beyond those to be processed in this unit. Thus, the exception is not necessarily caused by the access to the elements that have to be processed in the current UOP.

4. *Suppression.*
   This type of outcome has the same effect as nullification, except that the instruction address in the old PSW designates the next sequential instruction. For example, if CSI is implemented in a virtual-memory environment, suppression can be specified as the outcome of an UOP that raises a *protection violation* access exception. Similar to nullification, suppression of the current unit of operation may be caused by elements located beyond those processed in this unit.

Table 2.2 summarizes the differences between the effects of non-terminating types of outcome of a unit of operation (UOP) on the program counter of the interrupted program, on the (implicitly updated) stream control registers which determine the next elements to be processed, and on the contents of the result location. We recall here that apart from the update of the program counter, control registers, and the result location, the following values are updated upon interruption: the *StreamID (SID)* field of the SSR, and the *ExCode* field of the interrupted program's PSW. These changes are performed irrespectively of the outcome of the interruption-causing UOP. Finally, we remark that when an interruption occurs after com-

Exception Type	UOP Outcome
Access Exceptions	
TLB miss	Nullified
Address	Suppressed
Protection	Suppressed
Arithmetic Exceptions	
FP exponent underflow	Completed
FP exponent overflow	Completed
FP divide	Inhibited
FP square root	Inhibited
Specification exception	suppressed

Table 2.3: Types of exceptions and associated UOP outcomes.

pletion, inhibition, nullification, or suppression of an UOP, all prior units of operation have been completed.

Exception Types

The exceptions that cause (non-terminating) non-program interrupts (such as the *I/O* or the *supervisor call* interrupts) are defined to have no effect on the interrupted instruction. Therefore, all these kinds of exceptions will cause the current UOP to be completed. On the contrary, if an exception caused a *program interrupt*, the outcome of the current UOP can be different from completion and is determined by the exception type. All of the exceptions that may cause a program interrupt, i.e., the exceptions which may arise during execution of a CSI instruction, may also arise during execution of a scalar instruction of the general-purpose ISA that underlies CSI. Therefore, all these exceptions are defined in the underlying ISA. Since the underlying ISA is not exactly defined, the names of exception types presented below and conditions that cause them may have slightly different definitions for different underlying ISAs. It is, however, likely that any such ISA will define exceptions that are similar to those we describe. In the following, we present such general types of exceptions, assuming that the underlying ISA has virtual memory, and that the processor contains the TLB, which caches the results of the several last virtual-to-physical address translations. Table 2.3 summarizes the effect of the exceptions on the current unit of operation. The exceptions are structured in three groups: the *operand access* exceptions, the *arithmetical* exceptions, and the *specification* exceptions.

The first group consists of exceptions that may arise due to access to a stream element in storage. It includes the following exceptions: *TLB miss*, *address*, and *protection*. The *TLB miss* exception arises when the virtual address of the element cannot be translated using the TLB lookup. When this exception is recognized, the OS takes control by initiating the interruption routine that performs translation. The obtained physical address together with the virtual address that was translated are placed in the TLB. The effect of TLB miss on the current UOP is nullification, i.e. the interrupted CSI instruction will resume execution with the current UOP. When the physical address of the element is known, it is checked for validity. To be more precise, it is checked whether this address is actually present in the physical memory

and whether the interrupted program has rights to access the address. If the address is not present in physical storage, the *address* exception is recognized. If the interrupted program did not have rights to access the address, the *protection* exception is recognized. Any of these exceptions cause the suppressing of the current UOP, i.e., the cancellation of the interrupted instruction. This is logical, since these exceptions mean that interrupted instruction cannot access a stream element, and therefore, cannot perform the specified operation. Interrupted program resumes execution from the next following instruction.

The second group consists of the exceptions that are caused by performing arithmetic operations. The group includes the following exceptions: the floating-point (FP) *exponent underflow*, the *FP exponent overflow*, the *FP divide*, and the *square root*.

- When the floating-point exponent underflow exception is recognized, the unit of operation is **completed** by placing the true zero in the result location.
  For the simple arithmetic instructions such as `csi_add` and `csi_mul`, the result location of an UOP is either a single storage location in the destination stream (if the CSI sequential mode is enabled), or several consecutive such locations (if the CSI parallel mode is enabled). In the latter case, the true zero is placed in all the destination locations that correspond to the UOP for which the exception was recognized. For the accumulation-related instructions such as `csi_acc` and `csi_mul_acc`, the destination location is the *floating-point packed accumulator register (pacc_fp)*, which is a vector register that can hold several 32-bit FP numbers. The true zero is placed not only into the vector element that corresponds to the particular operation that caused the underflow, but in all the elements of *pacc_fp*. The process of FP accumulation and the usage of the FP accumulator register is presented in more detail in the description of `csi_acc` instruction, Section 3.3.3. Furthermore, we remark that if the underflow occurs during multiplication operation of the `csi_mul_acc`, `csi_mul_add`, or `csi_mul_sub` instruction, the true zero is placed in the result location without performing the addition or subtraction, although these operations are specified by the instruction.

- When the floating-point exponent overflow exception is recognized, the unit of operation is **completed** by placing the overflowed number in the destination location. The destination locations that are updated are defined in the same way as for the underflow exception. Similarly to the underflow handling, if the overflow occurred when performing multiplication during the execution of the `csi_mul_acc`, `csi_mul_add`, or `csi_mul_sub` instruction, the addition or subtraction are not performed on the overflowing element.

- The floating-point-divide exception is recognized during execution of the CSI divide instruction `csi_div`, which operated on streams of floating point numbers, for the unit of operation that contain at least one position in which the divisor element is zero. The unit of operation is **inhibited**.

- The FP square root exception is recognized during execution of the CSI divide instruction `csi_div`, which operates on streams of floating point numbers, for the unit of

operation that contain at least one position in which the divisor element is zero. The unit of operation is **inhibited**.

Note, that in the parallel mode, the whole unit of operation, which consists of processing multiple elements, has the same outcome. For example, if an UOP is inhibited due to the square root exception that resulted from operation on a certain element being processed in the unit, all the destination locations that correspond to this UOP will remain unchanged, although for some of them the operation could have been performed. This means that the semantics of the instruction (i.e., the way it changes the program state), when executed in parallel mode will differ from the semantics of the same instruction executed in the sequential mode. In some cases, however, programmer can avoid such differences in semantic by masking out the exceptions. For example, before performing the `csi_sqrt` instruction, programmer may insert the `csi_cmp` instruction which will generate mask stream that has zeroes at the positions that correspond to the negative elements of the source arithmetic stream. After this, the `csi_sqrt` may be executed in the masked mode and will cause no exceptions (and, therefore, will have the same outcome as the sequential version or the scalar code). Furthermore, masking out the exceptions to prevent interruptions is a good programming practice to provide high performance, because interruptions take a lot of time. The FP divide exceptions may be ignored in the similar way as the square root ones. For the overflow/underflow exceptions such mechanisms are not provided We remark, however, that the stream data that cause such exceptions is likely to represent erroneous input.Note that the semantics of a CSI instruction in the sequential mode in most of the cases (except of accumulation-related instructions) will coincide with the semantics of the scalar code of the underlying ISA that implements the same calculations. Deviation from the sequential semantic is the price we pay for the high performance of the parallel mode.

Finally, the third group consist of a single element, the *CSI specification exception*. This exception is recognized during the decoding of a CSI instruction in the cases when invalid operands are designated. Specification exception is associated with the first UOP and causes the first UOP (and, hence, the whole instruction) to be suppressed. Execution of the interrupted program resumes with the next following instruction. Specification exception is recognized in the following most common cases:

- When the `csi_mt_ssr_pacc` attempts to load values in the stream status register that are other than all zeros in reserved bit position.
- When the SCR-set that describes the operand stream of an instruction contains parameters that will result in unaligned accesses. More precisely, such a situation arises if the **Base** register contains address that is not aligned according to the element size, or if the **HStride** or **VStride** registers specify strides (in bytes) that are not multiple of the stream element size.

For particular CSI instructions, specification exception may be recognized in some additional cases. Details are presented in individual instruction descriptions in Section 3.3.

2.4 Conclusions

In this chapter we have described the CSI architecture in general. We presented the state of the CSI architecture, which consists of the CSI memory space and CSI register space. The CSI memory space is assumed to coincide with the memory space of the general-purpose ISA to which CSI serves as an extension. We presented a detailed description of the CSI register space, that consists of stream control register sets (SCR-sets), mask stream control register sets (MSCR-sets), the stream status register, and two accumulator registers. The structure of the sets, the format of individual registers and the meaning of their contents were explained. Then we described the operands of CSI instructions, concentrating on the description of the operand types that are particular for CSI, namely, the CSI arithmetic streams, the CSI bit streams, and the accumulator registers. The conditional execution support, for which the bit streams have been introduced and used in CSI, was explained. After this we gave a detailed description of the interruption handling mechanism employed in CSI. This mechanism is based on representing the execution of a CSI instruction as a sequence of units of operation (UOPs) and on the automatic update of the control registers that represent the extent to which an instruction has been executed at the point of interrupt. We showed that this mechanism allows resumption of the execution of an interrupted instruction from the point of interruption, which is an important condition for achieving high performance with CSI.

In the following chapter we will present the CSI instruction set, giving the descriptions of each CSI instruction, its format, and the operation it performs. We will also describe how CSI instructions are executed and will give some examples of coding multimedia kernels with CSI.

Chapter 3

CSI Architecture: ISA and Execution

In the previous chapter we presented a general description of the CSI architecture, concentrating on describing the architectural state of CSI, the operand types specific to CSI, such as CSI arithmetic streams, and the interruption-handling mechanism. For illustration purposes, we gave definitions of some CSI instructions, such as `csi_add`. In this chapter the complete set of CSI instructions, i.e., the CSI Instruction Set Architecture, or CSI ISA, is described. We present possible instruction formats, types of operands used in each format, main operations specified by the instructions, definitions of units of operation, and other relevant information. As the majority of the CSI instructions are similar to vector instructions, the general organization of the CSI ISA is based on that of the *IBM System 370/390 Vector Architecture* [4, 13].

Section 3.1 presents the instruction formats. Many CSI instructions have common characteristics, such as how they access operand data or react on exceptions. Instructions are grouped in several classes with respect to these features. These instruction classes are described in Section 3.2. Section 3.3 presents the complete set of CSI instructions, grouped according to the type of operations they perform. In Section 3.4 of this chapter we describe the instruction execution. Section 3.5 presents some conclusions.

3.1 Instruction Formats

CSI is assumed to be an extension to a general-purpose RISC-like instruction set architecture (ISA). Throughout this chapter we assume that this ISA has 32-bit instructions, and that the 6 most-significant (leftmost) bits of an instruction specify its opcode. It is assumed that CSI instructions are encoded as coprocessor instructions. This means that some particular value of the opcode, called the *CSI coprocessor opcode*, is assigned to all CSI instructions. When the host processor decodes an instruction with the opcode equal to the CSI coprocessor opcode, the instruction is recognized as a CSI instruction and is sent to the CSI execution unit which performs further decoding and execution. If the general-purpose ISA that underlies CSI is not

Format	Types and Numbers of Operands			
Mnemonic	Arith. streams	Bit streams	Scalar Registers	other
a3b1	3	1		
a2b1r	2	1	1/none	none/pacc
a2b1	2	1		
abr	1	1	1	
b3		3		
br		1	1	
pr			1	pacc
r_scr			1	SCR/MSCR
r_ssr_pacc			1	SSR/pacc

Table 3.1: Instruction formats: operand types and quantities.

32-bit, or has different encoding than described before, the definitions given in this chapter can be modified easily.

As stated before, the 6 most-significant bits (MSB) of all CSI instruction have the same value, namely the CSI coprocessor opcode. The remaining 26 bits are used to encode a CSI instruction specifying the operation and the operands. Bits 25..31 of a CSI instruction are called collectively the *CSI opcode field* and specify the operation which should be performed. This field of a CSI instruction will be further referred to simply as the *CSI opcode* or just *opcode*, except of where this may be confusing and the CSI opcode can be confused with the (main) opcode, i.e., with the field that consists of the 6 most significant bits. Bits 6..24 are used to encode the operands of an instruction and they are collectively referred to as the *operand bits*. CSI instructions are organized in several groups according to the instruction format they use.

A CSI instruction can have one of the following formats: *a3b1, a2b1r, a2b1, abr, br, r_scr, r_ssr_pacc*. The formats describe the way how the operand bits are organized in a number of fields. Different instructions that have the same format may, however, interpret these fields differently. The interpretation of the fields is determined by the CSI opcode of the instruction. Instructions, which have the same format, usually have the same number and the same types of operands. Table 3.1 summarizes all possible CSI instruction formats and presents the types of operands available for each format. The mnemonics of the format names have the following meaning: 'a' stands for arithmetic streams, 'b' — for bit stream, 'r' — for scalar register, 'p' — for packed accumulator register, 'cr' — for control register, and 'sr' — for status register. For example, all instructions that have the *a3b1* format designate four operands located in storage, three are CSI arithmetic streams and one is the CSI bitstream. The symbol *pacc*, which appears in the "other" column, represents any of the two packed accumulator registers *pacc_int* and *pacc_fp*. The expression "X/Y", which appears, for certain formats, in the "Scalar Registers" and "other" columns mean that both "X" and "Y" can be designated as operands for the corresponding format. For example, all the instructions that have the *a2b1r* format designate two arithmetic streams, and one bitstream operand. Furthermore, an instruction in this format may designate as a fourth operand a scalar register or one of the accumulator registers. Figure 3.1 depicts the most common instruction formats graphically. Below we describe in detail these common formats, while the less common ones

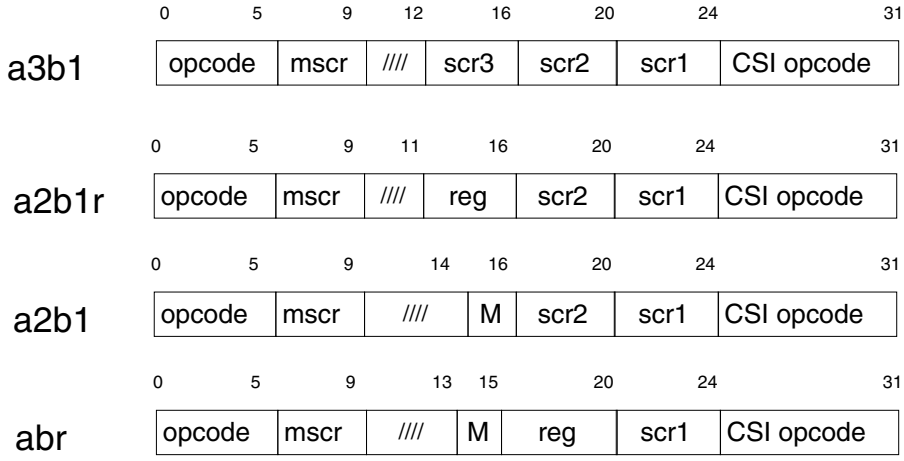


Figure 3.1: Most common CSI instruction formats.

will be presented together with the individual instruction descriptions later in this chapter.

1. The *a3b1* format specifies 4 operands. Following instructions have the *a3b1* format:
   `csi_add`, `csi_sub`, `csi_mul`, `csi_div`,
   `csi_and`, `csi_or`, `csi_xor`,
   `csi_mul_add`, `csi_split`, and `csi_merge`.
   For all the instructions except for the last three, *scr1* designates the SCR-set of the destination arithmetic stream, *scr2* and *scr3* designate the SCR-sets of the first and the second source arithmetic streams, and *mscr* designates the MSCR-set of the source mask operand stream. The mask operand is ignored if the instruction depends on the CSI mask mode and the mask mode is switched off. For the last three instructions the interpretation of the operand fields is slightly different and will be presented in the individual descriptions of these instructions.

2. The *a2b1r* format is another common format. The following instructions have this format:
   `csi_add_reg`, `csi_sub_reg`, `csi_mul_reg`, `csi_div_reg`,
   `csi_and_reg`, `csi_or_reg`, `csi_xor_reg`,
   `csi_acc_section`,`csi_mul_acc`, and `csi_mul_add_reg`.
   For all the instructions except for the last three, the operand fields specify the operands as follows. Fields *scr1, scr2* and *mscr* have the same meaning as in the *a3b1* format. The *reg* field specifies the second source operand, which is a constant contained in a scalar register. The field may designate either a general-purpose or a floating-point register. The type of the register is determined by the element type of the operand arithmetic streams, i.e., if the *FP* flag of the **Misc** register of the corresponding SCR-sets is set then the *reg* field designates a floating-point register, and otherwise it designates a general-purpose register. The meaning of the operand fields for the last three instructions will be presented in their individual descriptions.

3. The *a2b1* format is used by the `csi_sqrt`, `csi_cvt_fp_w`, `csi_cvt_w_fp`, and `csi_cmp` instructions. For the first three of them, the *scr1, scr2* and *mscr* fields are interpreted in the same way as for the formats presented above, and the *modifier* field *(M)* is not used. For `csi_cmp`, the *scr1* and *scr2* fields designate SCR-sets of the two source arithmetic streams, *mscr* designates the MSCR-set of the destination bit stream. In the modifier field (M) the type of comparison is encoded.

4. The *abr* format is used by the `csi_cmp_reg`, `csi_max`, `csi_min`, and `csi_acc` instructions. For the `csi_cmp_reg`, the *scr1*, *mscr*, and *M* fields have the same meaning as for the `csi_cmp`. The *reg* designates the scalar register that contains the second source operand, which is a constant. For the `csi_max` and `csi_min` instructions, *scr1* and *mscr* designate the source arithmetic and mask streams, respectively. The *reg* operand field designates the destination scalar register. For all these three instructions, whether a GP or an FP register is designated by the *reg* field is determined by the element type of the source operand stream. The operand field interpretation used by the `csi_acc` instruction is different and will be presented in the description of this instruction in Section 3.3.3.

3.2 CSI Instruction Classes

CSI instructions are grouped in several classes depending on their interruptibility, dependence on the mask-mode, and the amount of data they process. Table 3.2 summarizes the features of these instruction classes.

Class	Number of Data Items Processed	Interruptible?	Mask-mode ?
IM	Determined by SCR/MSCR-sets	Yes	Yes
IC	Determined by SCR/MSCR-sets	Yes	No
IP	Determined by SIMD-width and PSN	Yes	No
N	None	No	No

Table 3.2: CSI instruction classes.

The instruction classes distinguish:

- Whether the instruction is interruptible (IM, IC, IP) or not interruptible (N-class).
- Whether selection of stream elements to be processed depends on the setting of the stream mask mode (IM-class).
- Whether the number of the elements processed is variable and controlled by the corresponding (mask) stream control registers (IM, IC), or it is fixed and is determined by the SIMD-width and the partial-sum number (IP-class).

IM- and IC-Class Instructions. Most CSI arithmetic instructions, such as `csi_add`, `csi_and`, or `csi_and_bitstr` are either in class IM or IC. Instructions in both classes are interruptible. Class-IM instructions are also under the control of the stream-mask

mode; class-IC instructions are independent of the stream-mask mode. For an arithmetic stream, the elements are processed in sequence (described in Section 2.2.1) from the $((\textit{CurrRow} - 1) \cdot \textit{HLength} + \textit{CurrCol})$-th element to the $(\textit{VLength} \cdot \textit{HLength})$-th element, where *HLength, VLength, CurrRow* and *CurrCol* are the values of the corresponding stream control register from the SCR-set that describes the stream. For a CSI bitstream, the elements, each of which is a single bit, are processed in sequence from the *CurrElem*-th to the *Length*-th element, where *CurrElem* and *Length* are the values of the corresponding mask control registers from the MSCR-set that describes the stream. Normally, the **CurrRow, CurrCol** and **CurrElem** registers of the operand streams SCR/MSCR-sets contain zeroes at the beginning of the execution.

The number of elements that are processed for each operand is called the operand *net count*. For a CSI arithmetic stream, the net count is equal to

$$\textit{net count} = max((\textit{VLength} - \textit{CurrRow}) \cdot \textit{HLength} + (\textit{HLength} - \textit{CurrCol}), 0).$$

This means that if *CurrRow* $>$ *VLength*, or if *CurrRow* $=$ *VLength* and *CurrCol* $>$ *HLength*, the net count is zero. For CSI bitstreams, the net count is equal to $max(\textit{Length} - \textit{CurrElem}, 0)$. For CSI instructions which combine arithmetic stream source operand with a scalar register source operand, the scalar operand is considered to be replicated as many times as indicated by the net count of the corresponding arithmetic stream operand. If a CSI instruction is interrupted during execution, $X - Y$ elements of an operand stream have been processed, where X and Y are the values of the operand's net count at the beginning of execution and at the time of interruption, respectively. Y is then equal to the number of the next element to be processed. If the net count for at least one stream operand is zero at the start of instruction execution, the **CurRow, CurrCol** or **CurrElem** registers from the SCR/MSCR-sets that describe the stream operands of the instruction are set to zero, and execution is completed immediately. No elements are processed, and no operand-access exceptions occur. Operands in storage, floating-point, and general-purpose registers that are due to be modified, and the SCR/MSCR-set in-use bits remain unchanged.

IP-Class Instructions. The IP instruction class consists of instructions that operate on the packed accumulator registers *pacc_int* and *pacc_fp*. The number of arithmetic values they process is determined by the size of the registers and, for *pacc_int*, also by the way the register is formatted. The *pacc_fp* register is $8 \cdot \textit{SIMD\_width} \cdot \textit{FPSN}$ bits wide, where *SIMD_width* is the width in bytes of the CSI execution unit datapath, and *Floating-Point Partial-Sum Number (FPSN)* is the latency of the floating-point addition (in cycles). Both these numbers are fixed for a given implementation of the CSI execution unit. The floating-point packed accumulator register contains $\textit{pacc\_fp\_num} = \frac{1}{4}\textit{SIMD\_width} \cdot \textit{FPSN}$ single-precision FP numbers. The integer packed accumulator register *pacc_int* is $24 \cdot \textit{SIMD\_width} \cdot \textit{IPSN}$ bits wide and may contain $\textit{SIMD_width} \cdot \textit{IPSN}$24-bit binary integers, or $\frac{1}{2}\textit{SIMD\_width} \cdot \textit{IPSN}$ 48-bit binary integers. The interpretation of its contents as 24-bit or as 48-bit numbers is determined by the instructions that use the accumulator. The *Integer Partial Sum Number (IPSN)* is the latency of the integer addition. For simplicity, in the remainder of this dissertation, we assume that $\textit{IPSN} = 1$, and will omit it from the formulas.

The elements of a packed accumulator register used by an IP-class instruction are processed in sequence from *PIX*-th one to the last one, where *PIX* is equal to the contents of the

Packed Accumulator Interruption Index (PIX) field of the stream status register (SSR). For the instructions that operate on *pacc_fp*, the last element is the $\frac{1}{4}$*SIMD_width* $\cdot$ *PSN*-th one. For the instructions that operate on *pacc_int* the last element is either the *SIMD_width*-th 24-bit element or the $\frac{1}{2}$*SIMD_width*-th 48-bit element, depending on the register format used by the instruction. The total number of elements in the accumulator is denoted as *pacc_int_num*. The total number of the elements processed by an IP-class instruction is called the net count. For the instructions that use *pacc_fp*, the net count is equal to $max($*pacc_fp_num* $-$ *PIX*$, 0)$, and for the instructions operating on *pacc_int* is equal to $max($*pacc_fp_num* $-$ *PIX*$, 0)$. Normally, the accumulator interruption index PIX is zero at the beginning of the execution. If an IP-class CSI instruction is interrupted during execution, $X - Y$ elements of an operand stream have been processed, where X and Y are the values of the operand's net count at the beginning of execution and at the time of interruption, respectively. Y is then the number of the next operand element to be processed. If the net count is zero at the start of instruction execution, no accumulator register elements are processed, no scalar registers that are due to be modified are changed, the PIX field of the SSR is set to zero, and the instruction is completed immediately.

N-Class Instructions. The N-class consists of the CSI instructions that do not process any data, but just move data between the scalar and the CSI registers. For example, the `csi_mf_scr` and `csi_mt_scr` instructions that move data between the stream control registers of the SCR/MSCR-sets and the general-purpose registers belong to this class. Instructions belonging to this class consist of a single unit of operation that performs the specified action. These instructions are not interruptible.

3.3 CSI Instruction Set

In this section we give an overview of the CSI instruction set. The instructions are grouped according to the type of operation they perform. For each of the presented instructions, we describe the main operation it performs, give the definition of the unit of operation, and describe the types of the exceptions that may occur during the execution.

3.3.1 Simple Arithmetic and Logical Instructions

This group is formed by the most common instructions, namely, the instructions that perform simple unary or binary arithmetic operation on the elements of one or two source arithmetic streams (optionally, under control of the source mask stream) and store the results in the destination arithmetic stream. The instructions that belong to this group, and their main features, are summarized in Table 3.3. The *Element Types* column of this table presents the stream element data types in their processing format that are allowed for each of the listed instructions. The abbreviation *"all arith."* used in the 5-th column stands for *"all arithmetic types"*, i.e. the types *u8, s8, u16, s16, u32, s32*, and *f32*.

All instructions presented in Table 3.3 are rather similar. The first source operand is always a CSI arithmetic stream. The second source operand (if present) is a CSI arithmetic stream or a scalar register. Since all of the instructions are from the IM class, if the

Instruction	Operation	Format	Class	Element types
Arithmetic				
`csi_add`	add	*a3b1*	*IM*	*all arith.*
`csi_add_reg`	add	*a2b1r*	*IM*	*all arith.*
`csi_sub`	subtract	*a3b1*	*IM*	*all arith.*
`csi_sub_reg`	subtract	*a2b1r*	*IM*	*all arith.*
`csi_mul`	multiply	*a3b1*	*IM*	*all arith.*
`csi_mul_reg`	multiply	*a2b1r*	*IM*	*all arith.*
`csi_mul_add`	multiply and add	*a3b1*	*IM*	*all arith.*
`csi_mul_add_reg`	multiply and add	*a2b1r*	*IM*	*all arith.*
`csi_div`	divide	*a3b1*	*IM*	*f32*
`csi_div_reg`	divide	*a2b1r*	*IM*	*f32*
`csi_sqrt`	square root	*a2b1*	*IM*	*f32*
`csi_cvt_w_fp`	convert integer to FP	*a2b1*	*IM*	*f32, u32*
`csi_cvt_fp_w`	convert FP to integer	*a2b1*	*IM*	*f32, u32*
Logical				
`csi_and`	bitwise AND	*a3b1*	*IM*	*u8, u16, u32*
`csi_and_reg`	bitwise AND	*a2b1r*	*IM*	*u8, u16, u32*
`csi_or`	bitwise OR	*a3b1*	*IM*	*u8, u16, u32*
`csi_or_reg`	bitwise OR	*a2b1r*	*IM*	*u8, u16, u32*
`csi_xor`	bitwise XOR	*a3b1*	*IM*	*u8, u16, u32*
`csi_xor_reg`	bitwise XOR	*a2b1r*	*IM*	*u8, u16, u32*

Table 3.3: Simple arithmetic and logical instructions.

stream-mask mode is enabled, they also may use the source mask stream operand. Instructions perform an arithmetic or logic operation on the corresponding elements of the source operand streams and write the results to the corresponding location of the destination arithmetic stream.

The only exceptions to the rules which describe the way the instruction operands are interpreted and used, are formed by the `csi_mul_add` and `csi_mul_add_reg` instructions. Consider the `csi_mul_add SCRSi, SCRSj, SCRSk, MSCRS` instruction, where `SCRSi, SCRSj` and `SCRSk` designate the SCR-sets of the operand arithmetic streams *opnd1, opnd2*, and *opnd3*, respectively, and `MSCR` designates the operand mask stream *mask1*. The instruction uses the operand streams slightly differently from the other instructions from Table 3.3, which have the same *a3b1* format. The difference is that the *opnd1* stream is used both as the source and the destination stream. The operation performed by the instruction is specified is follows (assuming that the stream mask mode is enabled):

$$opnd1[i] = \begin{cases} opnd1[i] + opnd2[i] \cdot opnd3[i], & \text{if } mask1[\text{i}] = 1 \\ no\text{-}op & \text{otherwise} \end{cases}$$

If the stream-mask mode is disabled, the instruction performs the multiply-add operation on all the triples of the corresponding stream elements. The `csi_mul_add_reg` differs in a similar way from all the other instructions which are presented in Table 3.3 and have the *a2b1r* format.

Since the operations performed by the instructions presented in Table 3.3 are very similar, we describe these operations and give the definitions of the unit of operation (UOP) just for some of them, namely, `csi_add`, `csi_add_reg`, and `csi_sqrt`. These are typical binary arithmetic, binary arithmetic with register operand, and unary arithmetic instructions.

Units of Operation

We first describe the actions performed during the execution and give the unit of operation (UOP) definitions for the `csi_add` instruction. Consider the `csi_add SCRSi, SCRSj, SCRSk, MSCRSl` instruction. The SCR-sets `SCRSi`, `SRCSj`, and `SCRSk` describe the destination, the first source arithmetic, and the second source arithmetic streams, respectively. If the stream-mask mode is disabled, the mask stream operand described by the MSCR-set `MSCRl` is ignored. The following operations are then performed during the execution. Elements of the source arithmetic streams are loaded from memory. If the storage and processing formats of the source stream elements are different, the loaded elements are converted to the processing format, or *unpacked* (for a detailed description of unpacking procedure, see Section 3.4). Corresponding elements of the source arithmetic streams are then added. When required, the sum is converted from the processing format of the destination stream to its storage format, or *packed* (see Section 3.4). After this, the obtained value is stored at the corresponding location of the destination stream. If the stream-mask mode is enabled then mask stream elements are loaded first and the operations described above are performed only for the arithmetic stream elements for which the corresponding mask element is 1. When the mask element is zero, no arithmetic data is loaded, addition is not performed, and the destination stream is not modified. We now define the unit of operation for the instruction.

Case 1: Stream-Mask Mode Disabled. If the stream-sequential mode is enabled, a unit of operation (UOP) consists of loading a pair of corresponding source stream elements, unpacking them (optional), performing the addition, packing the sum (optional), and storing the resulting element in the destination stream. Suppose now that the stream-parallel mode is enabled. As was described in Section 2.3.3, for a given implementation of the CSI execution unit, the *SIMD_width* is defined as the width of the unit's datapath in bytes. Consider the first source operand stream *src_opnd1*, which is described by the SCR-set `SCRSj`. We define the following value:

$$src\_opnd1\_SIMD\_els = \frac{SIMD\_width}{SCRSj.Misc.ProcessSize}$$

This value is equal to the number of the *src_opnd1*'s elements that are needed to fully utilize the datapath of the execution unit, i.e. it is equal to the number of elements of the first source stream that require together exactly *SIMD_width* bytes for their processing format representation. Similarly, the values *src_opnd2_SIMD_els* and *dest_opnd_SIMD_els* are defined. It is required that the streams are specified in such a way that these values are equal to *src_opnd1_SIMD_els*, otherwise, the specification exception is recognized. A unit of operation consists of the following operations: loading *src_opnd1_SIMD_els* elements of each of the source streams, optional unpacking of the loaded elements to the computation format, performing in parallel *src_opnd1_SIMD_els* additions of the corresponding elements, optional

packing of the results, and storing the obtained *src_opnd1_SIMD_els* sums to the destination stream.

Case 2: Stream Mask Mode is Enabled. We now assume that the stream-mask mode is enabled. Suppose that the stream-sequential mode is enabled. We describe the operations that are performed in the i-th unit of operation. If the byte that contains the i-th bit of the mask stream is not yet loaded, then this byte is loaded. The i-th mask stream bit is checked. If it is zero, no other operations are performed in the i-th UOP. If it is one, the UOP consists of the same sequence of operations as for the disabled mask mode. Suppose that the stream-parallel mode is enabled. The value *src_opnd1_SIMD_els* is defined in the same way as described above. The value *mask_opnd_SIMD_bytes* is equal to the number of mask stream bytes that contain mask bits that correspond to *src_opnd1_SIMD_els* arithmetic stream elements. It is defined according to the formula:

$$mask\_opnd\_SIMD\_bytes = \left[\frac{SIMD\_width}{8}\right] + 1,$$

where $[.]$ denotes the integer part of a number. An UOP consists of the following operations. First, *mask_opnd_SIMD_bytes* bytes of the mask stream are loaded. The elements of the source streams that correspond to the ones in the mask stream are loaded, added in parallel and stored in the destination stream. The elements that correspond to zeroes in the mask stream, are not accessed, and the corresponding locations of the destination stream remain unchanged.

For the `csi_add_reg` instruction the definitions of an UOP are quite similar to the definitions we presented above. The only difference is that instead of loading elements from the second source stream, and adding them to the elements of the first source stream, the elements of the first stream are added with the value contained in the scalar register designated by the *reg* field of the instruction (see the definition of the *a2b1r* instruction format). If the source stream contains floating-point numbers, the *reg* field designates an FP register, and the value it contains is used as it is. If the source stream contains integer numbers (i.e., the *u8, s8, u16, s16, u32, s32* data types), the *reg* field designates a 32-bit GP register. For the *u32* and *s32* data types, the value it contains is also used as it is. When the stream elements in their processing format are of the *u8* and *s8* data types, the 8 least significant bits of the GP register form the number which is added to the stream elements. When the stream elements are of the *u8* and *s8* data types, the number which has to be added to them is formed by the 16 least-significant bits of the GP register. For the `csi_sqrt` instruction, the UOP definitions are also similar to those of `csi_add` instruction. The only differences are the absence of the accesses to the second operand stream and the different main operation performed on the source stream elements.

Exceptions

Exceptional conditions that can occur during the execution of the instructions presented in Table 3.3 are grouped in three classes: specification exceptions, access exceptions, and floating-point exceptions.

Specification Exceptions. This class of exception consists of a single type of exception, called the *specification exception*, which is recognized when the operand stream parameters are specified illegally for a given instruction. The following types of illegal specifications are common for all the instructions presented in Table 3.3:

- The elements of operand streams are specified to have different processing formats. For example, for the `csi_add_reg` instruction, the elements of the source arithmetic stream are specified to be processed as *u8* values, and the elements of the destination stream are specified to have the *s8* processing format.
- The storage and the processing formats of a source arithmetic stream make unpacking impossible. This situation arises if the number of bytes needed to represent an element in its storage format is bigger than the number of bytes required for its processing format representation.
- The storage and the processing formats of the destination arithmetic stream make packing impossible. This situation arises if the number of bytes needed to represent an element in its storage format is smaller than the number of bytes required for its processing format representation.
- The **Base, HStride** or **VStride** registers of the SCR-set that describe an operand arithmetic stream are specified erroneously with respect to the alignment requirements of the storage format of stream elements. The alignment requirements are presented in Table 2.1. For example, if the stream elements are specified to be stored in *u16* format, which requires the 2-byte alignment, the values contained in the registers mentioned above should be multiples of two.
- The storage or the processing formats of an operand stream elements are not the ones that are allowed for a given instruction. The formats that are allowed are presented in the *Element Type* column of Table 3.3.
- The *Sign* (or *Saturate*) bits of the **Misc** control registers of the SCR-sets that describe the operand arithmetic streams are not the same.
- The $GrSize$, $ElSize$, or $ProcessSize$ fields of the $Misc$ register of the SCR-set that describes the operand stream are specified illegally. The legal values for these fields are presented in Section 2.1.

For some instructions there are additional conditions that cause the specification exception. These conditions are summarized below.

- For the `csi_cvt_w_fp` instruction: the format of the source stream elements is not *u32*, or the format of the destination stream elements is not *f32*.
- For the `csi_cvt_fp_w` instruction: the format of the source stream elements is not *f32*, or the format of the destination stream elements is not *u32*.
- For the `csi_mul` and `csi_mul_reg` instructions: the element format of at least on of the source streams is specified to be *u32* or *s32*.

- For the `csi_mul_add` and `csi_mul_add_reg` instructions: the element format of at least one of the source streams which has to be multiplied is specified to be *u32* or *f32*.

Operand Access Exceptions. These exceptions arise when the operand elements located in memory are accessed. If the instruction depends on the stream-mask mode and the mode is enabled then the elements of the arithmetic stream operands that correspond to the zero mask bits are not accessed and do not cause access exceptions. The exceptions and their effect on a unit of operation were presented in Table 2.3. When the stream-parallel mode is enabled, several consecutive elements of an operand stream are accessed in a single UOP. Recall from Section 2.3.3 that exceptions are attributed to the whole UOP. Therefore, for example, if an access to a single element in a UOP causes a *TLB_miss* exception, the whole UOP is nullified and will have to be re-executed after the exception is handled.

Floating-point Exceptions. These exceptions can arise during the execution of the arithmetic instructions that operate on floating-point data. The situations that can cause floating-point exceptions are summarized below.

- For the `csi_add`, `csi_add_reg`, `csi_sub`, `csi_sub_reg`, `csi_mul`, and `csi_mul_reg` instructions, the *FP exponent overflow/underflow* exceptions arise if the instructions operate on floating-point elements and the operation performed on a pair of elements results in floating-point exponent overflow/underflow.
- For the `csi_div` and the `csi_div_reg` instructions, the *FP divide* exception arises when the division has to be performed on a couple of elements and the divisor is zero.
- For the `csi_sqrt` instruction, the *FP square root* exception is recognized when the source stream element is negative.

3.3.2 Comparison-Related Instructions

This group is formed by the instructions that perform compare operation during their execution. Table 3.4 summarizes the instructions that belong to the group and their main features.

Instruction	Operation	Format	Class	Element types
`csi_cmp`	compare	*a2b1*	*IC*	*all arithmetic*
`csi_cmp_reg`	compare	*abr*	*IC*	*all arithmetic*
`csi_max`	maximum	*abr*	*IM*	*all arithmetic*
`csi_min`	minimum	*abr*	*IM*	*all arithmetic*

Table 3.4: Comparison-related instructions.

Main Operation and Operand Interpretation

The `csi_cmp` instruction has the *a2b1* format (see Section 3.1). The *scr1* and *scr2* operand fields designate the SCR-sets of two source arithmetic streams and the *mscr* designates the MSCR-set of the destination mask stream. The instruction compares an element of the first operand stream with the corresponding element of the second operand stream. If the result of comparison is true, the corresponding bit of the destination mask stream is set to 1, otherwise it is set to zero. The type of the comparison that has to be performed is encoded in the 2-bit modifier field, as it is shown in Table 3.5.

Modifier field	Comparison type
00	less
01	less or equal
10	equal
11	not equal

Table 3.5: Comparison type specification.

The `csi_cmp_reg` instruction has the *abr* format. The *scr1, mscrs* and *M* fields have the same meaning as for `csi_cmp`. The *reg* field designates a scalar register that contains the second source operand. Whether a general-purpose or a floating-point register is designated is determined by the data type of the source stream elements. For the *u8, s8, u16* and *s16* data types, the value contained in the register is used according to the same rules as for the simple arithmetic instructions with a register operand (see Section 3.3.1). The instruction compares the elements of the first operand stream with the value contained in the scalar register and, depending on the comparison outcome, sets the corresponding bits of the destination bit stream.

The `csi_max` instruction has the *abr* format as well. The *scr1* and *mscr* fields designate the source arithmetic and mask streams, respectively. The *reg* field designates the destination scalar register. Whether a general-purpose or a floating-point register is designated is determined by the data type of the source stream elements. The modifier field is always set to designate the ”less or equal” comparison type. The instruction finds the maximal element in the source stream (ignoring, if executed under mask-stream mode, the elements that correspond to zeroes in the mask stream) and places the element in the destination scalar register. The `csi_min` is rather similar to `csi_max`. The only difference is that the modifier is set to specify the 'less' comparison and the instruction finds the minimal element in the source stream.

Units of Operation

The activation of the stream-mask mode for comparison-related instructions of the IM class have the same effect as for the simple arithmetic and logical instructions, and no effect on the instructions of the IC class. Activation of the stream-parallel mode for all instructions listed in Table 3.4 have the same effect as for simple arithmetic and logical instructions. Therefore, in the following we present the unit of operation definitions under the assumption that the stream-mask and stream-parallel modes are disabled.

Data Type	Initialization Value	
	for `csi_max`	for `csi_min`
u8	0	$2^8 - 1$
s8	-2^7	$2^7 - 1$
u16	0	$2^{16} - 1$
s16	-2^{15}	$2^{15} - 1$
u32	0	$2^{32} - 1$
s32	-2^{31}	$2^{31} - 1$
f32	-infinity	+infinity

Table 3.6: Register initialization for minimum and maximum instructions.

For the `csi_cmp` instruction, a unit of operation consists of the following actions: loading a couple of corresponding elements from the source arithmetic streams, unpacking (if necessary), comparing them, and setting the corresponding bit of the destination bit stream to 1 or to 0, depending on the comparison's outcome. Unpacking may be needed, for example, if the source arithmetic streams that have to be compared have different formats, such as *u8* and *u16*, respectively. For the `csi_cmp_reg` an UOP is defined in a similar way. The only difference is that no elements of the second operand stream are fetched, and the elements of the first operand stream are compared with the constant value contained in the scalar register and that unpacking is never needed.

For the `csi_max` instruction, a unit of operation consists of the following actions: loading an element of a source arithmetic stream, comparing it with the value contained in the destination register (as was specified above, the 'less' comparison is used), and, if the outcome of the comparison is false, i.e., if the loaded element is bigger than the value contained in the scalar register, replacing the contents of the register with the value of the loaded element. As we specified above, if the source arithmetic stream consists of the elements that have a subword data type (i.e., on of the *u8,s8, u16,* or *s16* types), the value contained in the destination register is interpreted and has to be modified correspondingly. For example, if the source stream elements are of the *s8* data type, the 8 least significant bits are used for comparison, and if the contents of the destination GP have to be replaced, the replacing stream element is sign-extended from 8 bits to 32 bits and then is put into the register.

For the `csi_min` instruction, an UOP is defined similarly to that of `csi_max`. The only difference is that replacement occurs when the result of comparison, which is specified as 'less', is true, i.e., the replacement occurs if the compared stream element is less than the value contained in the scalar register.

Exceptions

For all of the instructions listed in Table 3.4, an operand access or a specification exception occurs under the same conditions as the ones described in Section 3.3.1.

Programming Notes

Because for the `csi_max` and `csi_min` instructions the scalar register operand is compared with every element of the source arithmetic stream, before starting the execution of these instructions, the value of the register should be initialized, respectively, to the minimal or the maximal values that can be represented by the data type of the source stream elements (in the processing format). Table 3.6 presents the values to which the scalar operand should be initialized for all possible data types.

3.3.3 Accumulation-Related Instructions

This group is formed by the instructions that perform accumulations during their execution. Table 3.7 summarizes the instructions that belong to the group and their main features.

Instruction	Operation	Format	Class	Elements
`csi_acc`	accumulate	*abr*	*IM*	*all arith.*
`csi_mul_acc`	multiply-accumulate	*a2b1r*	*IM*	*all arith.*
`csi_acc_psum`	accumulate partial sums	*pr*	*IP*	
`csi_zero_pacc`	zero accumulator	*pr*	*IP*	
`csi_acc_section`	accumulate section	*abr*	*IM*	*all arith.*

Table 3.7: Accumulation-related instructions.

General Structure of the CSI Accumulation Facility

Most of the accumulation-related CSI instructions are using the *CSI accumulation facility*, which is a part of the CSI register state consisting of two registers, the *floating-point packed accumulator register pacc_fp*, and the *integer packed accumulator register pacc_int*. Each of these registers is, essentially, a vector register in which several elements are packed. A detailed description of the format of these registers is presented later in this section. The reasons why the CSI accumulation facility is introduced is described below.

Although accumulation of a stream of numbers requires just a sequence of add operations, the definition of instructions that can perform accumulations efficiently requires some engineering effort for the following reason. When simple arithmetic and logic or compare instructions are executed, they produce a data stream the contents of which are not affected by the order in which the individual elements are produced. This is because operations on individual elements are independent from each other. Execution of operations on individual elements can, therefore, be easily overlapped as in a pipelined arithmetic unit. Furthermore, several operations on individual elements can be executed in parallel if multiple arithmetic units are available. The same observations hold for minimum and maximum instructions, the result of which is independent of the order in which elements are compared. The accumulation operation, however, is different. The order in which additions are performed may affect, for example, whether an overflow or underflow takes place. If a CSI accumulation instruction would have had to follow the scalar program semantics (as it is done for simple arithmetic and logic instructions), the stream elements would have had to be added in their original order.

This, however, does not allow us to use the arithmetic pipeline at top speed, if the latency of addition is more than one cycle. Suppose, for example, that a stream of floating-point numbers has to be accumulated and that the CSI execution unit has a single 32-bit wide arithmetic pipeline which requires two cycles to add two FP numbers. Then, if the stream elements have to be accumulated in their original order, the result of each addition will become available and the following addition will be performed after two cycles. Thus, accumulation will be performed at the speed of one addition every two cycles, while the arithmetic pipeline can perform additions at the top speed of one addition every cycle. Our solution to this problem is based on ideas employed in the *IBM System/370 Vector Architecture* [4].

Floating-Point Accumulation. In the situation we consider (single FP adder with latency of two cycles), the CSI architecture contains the *floating-point packed accumulator register (pacc_fp)*, which is a 64-bit register holding two 32-bit FP numbers. In order to perform additions at the top speed of the FP adder pipeline, i.e., at the speed of one addition per cycle, elements are added in an order that differs from the original sequential one. The accumulation operation is performed in two steps: first, the stream elements are reduced to two partial sums using the arithmetic pipeline at full speed and the sums are placed in the *pacc_fp* register. Second, these partial sums are added sequentially at the speed of one addition per two cycles to form the final result.

We now describe the first step in more detail. Let two floating-point values contained in the *pacc_fp* register to be denoted as $pacc\_fp_i$, for $i = 0, 1$. The *pacc_fp* is also called the *FP partial sum vector*. The number of elements in this vector (two in our case) is equal to the pipeline depth of the floating-point adder or, equivalently, to the latency of the 32-bit floating-point addition. This vector length is called the *floating-point partial sum number (FPSN)*. Accumulation of the source operand stream B to produce the partial sum vector $(pacc\_fp_0, pacc\_fp_1)$ is performed in the following order: the element B_0 is added to $pacc\_fp_0$, the element B_1 is added to $pacc\_fp_1$, the B_2 is added to $pacc\_fp_0$, B_3 is added to $pacc\_fp_1$, and so forth. Hence, the operand element B_i is added to partial sum vector element $pacc\_fp_{(i \bmod 2)}$.

In general, if the CSI execution unit has a single (32-bit) floating-point pipeline, i.e. if *SIMD_width* $= 4$, and the floating-point PSN is equal to p, then during the first stage of accumulation, B_0 is added to $pacc\_fp_0$,..., B_{p-1} is added to $pacc\_fp_{p-1}$, B_p is added again to $pacc\_fp_0$, B_{p+1} – to $pacc\_fp_1$, etc., so that an operand element B_i is added to the partial sum element $pacc\_fp_{(i \bmod p)}$. A CSI execution unit may have several 32-bit FP arithmetic pipelines. The number of pipelines is equal to $\frac{1}{4}$*SIMD_width*, where *SIMD_width* is the total width of the datapath of the CSI execution unit in bytes. In the most general case, when the CSI execution unit has m arithmetic pipes, each of which can perform addition in p cycles, the CSI accumulation facility will contain a $(32 \cdot m \cdot p)$-bit FP packed accumulator register *pacc_fp* that contains $(m \cdot p)$ 32-bit FP elements: $pacc\_fp = (pacc\_fp_0, \ldots, pacc\_fp_{mp-1})$. During the first stage of accumulation, operand stream elements are accumulated to produce the partial sum vector in such a way that an operand element B_i is added to the partial sum vector element $pacc\_fp_{(i \bmod mp)}$. After the first stage is finished, the partial sum vector element $pacc\_fp_j$, where $0 \le j < mp$, will contain the partial sum $\sum_{k=0}^{f(j,l,mp)} B_{j+k \cdot mp}$, where l is the number of elements in the stream B, and the function f is defined according as

follows:

$$f(j,l,x) = \begin{cases} \lceil \frac{l}{x} \rceil, & \text{if } j + x\lfloor \frac{l}{x} \rfloor < l \\ \lfloor \frac{l}{x} \rfloor, & \text{otherwise} \end{cases}$$

Integer Accumulation. Accumulation of integer elements is performed in a way similar to accumulation of floating-point elements. First, the elements are added in an order different from the original one to form several partial sums, which are placed in the *integer packed accumulator register (pacc_int)*, which is also referred to as the *integer partial sum vector*. After this, the partial sums are added to each other to form the final result.

Consider an implementation of the CSI execution unit for which *SIMD_width* $= n$, which means that the datapath of the execution unit is n bytes wide, and, hence, n add operation on 8-bit values can be performed in parallel. Then, the size of the integer accumulator register is defined to be $24 \cdot n$ bits ($3 \cdot n$ bytes). The register can contain n 24-bit elements $pacc\_int24_0, \ldots, pacc\_int24_{n-1}$. Recall that we assumed that the *integer partial sum number* is 1, i.e., that the latency of the integer add operation is 1 cycle. Therefore, the partial sum vector that contains n elements is sufficient to perform each cycle n additions in parallel. While for the floating-point accumulator register the size of its elements is equal to the size of the source operand elements (both the register and the operand elements are 32-bits), for the integer accumulator register, the size of its elements (24-bit) is larger than the size of the elements that have to be added (8-bit). The reason is that when the 32-bit floating-point numbers are added, the sum is still represented in 32-bits without loss of the precision, while when m 8-bit binary integer numbers are added, $8 + log_2 m$ bits are needed to represent the sum without loss of precision. Furthermore, allocating 24 bits for each registers element allows to perform the multiply-accumulate operation on two streams of 8-bit numbers fully utilizing the whole width of the multipliers datapath: n multiplications of 8-bit numbers can be performed in parallel (since *SIMD_width* $= n$), resulting in n 16-bit products, which are then accumulated in parallel in n partial sums contained in *pacc_int* register. Since each partial sum element is 24 bits wide, CSI can perform multiply-accumulate operation without loss of precision on data streams that contain up to $2^{log_2(24-16)} = 256$ 8-bit elements.
We note that the partial sum vector elements are 24 bits and, therefore, an 8- or 16-bit element that has to be added to a certain partial sum is (sign- or zero-) extended to 24 bits. Furthermore, the n-element accumulator register allows $n =$ *SIMD_width* additions to the partial sums to be performed in parallel, which requires n 24-bit integer addition pipes, while the SIMD execution unit is assumed to be able to operate on just n 8-bit numbers in parallel. Therefore, it is implied that an implementation of CSI that is capable of utilizing the full performance potential provided by a n-element ($24n$-bit) partial sum vector will contain a number of integer addition pipelines which are capable of performing n 24-bits additions in parallel and that are used exclusively to add values to the elements of the partial sum vector *pacc_int*. The $(24 \cdot n)$-bit CSI integer packed accumulator register can also contain $\frac{n}{2}$ 48-bit elements $pacc\_int48_0, \ldots, pacc\_int48_{\frac{n}{2}-1}$, which are used to hold partial sums when 24-bit or 32-bit values have to be accumulated. Accumulation of 24-bit values is required when executing multiply-accumulate operation on two arithmetic stream, one of which contains 16-bit numbers and the another one 8-bit numbers, since multiplication of such elements results in 24-bit products. Accumulation of 32-bit values is performed when the multiply-accumulate operation is executed on two streams of 16-bit values (implying 32-bit products), or when the

accumulation operation is executed on stream of 32-bit elements. Allocating of 48 bits for partial sums allows CSI to execute without loss of precision the multiply-accumulate operation on two streams that contain up to 2^{16} 16-bit values each.

Main Operations and Operand Interpretation

We now present the instruction formats used by the accumulation-related CSI instructions and the main operation these instructions perform.

Accumulation Instruction. The CSI accumulation instruction `csi_acc` uses the *abr* format discussed in Section 3.1. The instruction has three operands: the source arithmetic stream, the source mask stream, which are designated explicitly, and the destination operand, which is an implicitly designated accumulator register. The operand field *scr1* designates the SCR-set of the source arithmetic stream, the elements of which have to be accumulated. The *mscr* field designates the MSCR-set of the source mask stream, which is used if the stream-mask mode is enabled. The 2-bit modifier field (M) is used to designate the destination accumulator register. If $M = 0$ the destination register is $pacc\_fp$; if $M = 1$, the destination is $pacc\_int$ formatted to contain 24-bit partial sums, and if $M = 2$, the destination is $pacc\_int$ formatted to contain 48-bit partial sums.

The instruction reduces the elements of the source arithmetic stream according to the rules presented in Section 3.3.3 and places the partial sums into either integer of floating-point accumulator register. If the stream elements are floating-point numbers, the partial sums should be placed into the *pacc_fp* register. If the elements are integer numbers, the partial sums should be placed in the *pacc_int* register. Whether the *SIMD_width* 24-bit or $\frac{1}{2}$*SIMD_width* 48-bit partial sums are produced, is determined by the data type of the stream elements. Accumulation of 8-bit or 16-bit elements should result in *SIMD_width* partial sums (M should be specified equal to 1), and accumulation of 32-bit elements should result in $\frac{1}{2}$*SIMD_width* partial sums (M should be specified equal to 2). If a programmer specifies the destination accumulator not according to the rules we just described, a specification exception is recognized.

Multiply-Accumulate Instruction. The CSI multiply-accumulation instruction `csi_mul_acc` uses the *a2b1r* format, which is presented graphically in Figure 3.1. The instruction has four operands: two source arithmetic streams, the source mask stream, and the destination accumulator register. The operand fields *scr1* and *scr2* designate the SCR-sets of two arithmetic streams, and the *mscr* field designates the MSCR-set of the mask stream, which is used if the stream-mask mode is enabled. The destination operand is designated by the modifier field M in the same way as it is done for the `csi_acc` instruction.

Let A and B be the source arithmetic streams. The `csi_mul_acc` instruction multiplies the corresponding elements of the stream and reduces this sequence of products to several partial sums which are placed into one of the accumulator registers. The reduction of the sequence of pairwise products $A_i \cdot B_i$ to partial sums is performed in the same way as it is done by the `csi_acc` instruction. Which of the two accumulator registers should be used as the destination of the multiply-accumulate instruction is determined by the data types of

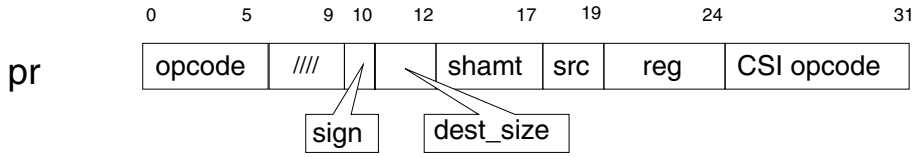


Figure 3.2: The *pr* instruction format.

the elements of the source streams. If the elements are floating-point numbers, the *pacc_fp* register is used ($M = 0$). If the data types of the elements of two streams in their processing format are 8-bit and 8-bit then the destination is the *pacc_int* register containing 24-bit partial sums ($M = 1$). If the elements of both streams are 16-bit, then the destination is the *pacc_int* register formatted to contain 48-bit partial sums ($M = 2$). If a programmer specifies the destination accumulator not according to the rules we just described, a specification exception is recognized. Combinations of stream data types different than (8-bit, 8-bit) or (16-bit, 16-bit), are not allowed and cause specification exception as well.

Partial Sum Accumulation Instruction. The second part of accumulation process is performed by the CSI instruction `csi_acc_psum` (accumulate partial sums in an accumulator register). This instruction adds all the elements of a partial sum vector to produce the final result, which is placed in a scalar register. The `csi_acc_psum` instruction has the *pr* format which is presented graphically in Figure 3.2. The 2-bit *src* field designates which of two accumulator registers should be used as a source operand, and, for *pacc_int*, the format which has to be used. The *dest* field designates the destination register. Whether a GP or an FP scalar register serves as the destination operand is determined by the type of the accumulator: if the source operand is *pacc_fp* then *dest* designates an FP register, otherwise, if the source operand is *pacc_int* then *dest* designates a GP register. Three other fields, *shamt*, *dest_size* and $sign$, contain integer numbers which are used to control shifting and truncation of the final result, needed to be performed when the destination is a GP register. These operations are described below.

If the value encoded in the *src* field is zero, `csi_acc_psum` adds the 32-bit floating point partial sums contained in $pacc\_fp$ sequentially and places the result in the floating-point register designated by the *dest* field.

If the value of the *src* field is 1, `csi_acc_psum` adds 24-bit binary integer partial sums contained in $pacc\_int$ sequentially. The obtained 24-bit number is then rounded (by adding the 24-bit binary integer number $2^{shamt-1}$, which represents the value 0.5) and truncated by shifting (arithmetically if $sign = 1$, or logically, if $sign = 1$) to the right by the number of bits specified in the *shamt* field. After roudning and truncating, the result is sign-extended (if $sign = 1$) or zero-extended (if $sign = 0$) to 32 bits. The *dest_size* field controls the conversion of the obtained number x to the appropriate data type. The field specifies the size of the data type of the final result in bytes. If this size is 1 byte, the 32-bit value x is truncated to its 8 least significant bits (LSB) and the obtained 8-bit value is then sign-extended to 32-bits (if the value of the *sign* field is 1) or zero-extended to 32 bits (if the value of the *sign* field is 0). If the size is specified to be 2 bytes, the 16 LSB of x are taken and sign- or zero-extended

to 32 bits. If the size is specified to be 4 bytes, the value of x is taken as it is. The 32-bit value obtained after the conversion operation we just described is placed in the GP register designated by the *dest* field.
If the value of the *src* field is 2, `csi_acc_psum` adds 48-bit binary integer partial sums contained in $pacc\_int$ sequentially. The obtained 48-bit number is then rounded and truncated by shifting to the right by the number of bits specified in the *shamt* field. Next, the least significant 32 bits of this value are taken. Finally, the obtained value is converted to the data type of the final result in the same way as is it is done for 24-bit partial sums. The reason why the rounding and shifting operations are performed when accumulating integer partial sums is explained in Section 3.3.3.

Zero Accumulator Register Instruction. The CSI instruction `csi_zero_pacc` uses the *pr* format. The 2-bit *src* field designates the destination accumulator register in the same way as `csi_acc_psum`. All other fields are not used. The instruction sets all the elements of the destination accumulator register to zeroes.

Accumulate Section Instruction. The CSI instruction `csi_acc_section` instruction uses the *a2b1r* format. The instruction has four operands: the source arithmetic streams, the source mask stream, the source register operand, and the destination arithmetic stream. The *scr1* and *scr2* fields designate the SCR-sets of the destination and the source arithmetic streams, respectively. The *mscr* designates the MSCR-set of the source mask stream, and *reg* designates the source general-purpose register, which contains the *section_size* parameter.

The instruction accumulates every *section_size* consecutive elements of the source arithmetic stream and writes the produced values to the destination stream. The *section_size* parameter can be any number from 1 to 16 and is determined by the 4 least significant bits of the GP register GP[*reg*], which is designated by the *reg* field, according to formula: $section\_size = 4\text{LSB}(\text{GP}[reg]) + 1$. The instruction is introduced in order to perform accumulations of short sequences of stream elements, because such operation cannot be performed efficiently using the `csi_acc` instruction. The need to perform such operation often arises in multimedia applications when a sequence of (short) n-element vectors has to be multiplied with an $n \times n$ matrix, or when matrix-matrix multiplications of $n \times n$ matrices have to be performed. The typical values of n used in multimedia codes are 3, 4, or 8 (i.e., less than 16), which explains the limitation posed by CSI on the *section_size* parameter. For accumulations of longer sequences of i consecutive stream elements, the conventional accumulation instruction `csi_acc` can be employed and is likely to be efficient. We also note here that `csi_acc_section` does **not** use the accumulator registers.

Units of Operation.

The activation of the stream-mask and the stream-parallel modes have the usual effect on the `csi_acc`, `csi_mul_acc`, and the `csi_acc_section` instructions. The `csi_acc_psum` is independent of both of these modes. Therefore, in the following we give the definition of an UOP under the assumption that these two modes are disabled.

For the `csi_acc` instruction, a unit of operation consists of the following actions: loading a source stream element, unpacking it to the processing format (optional), and adding

the element in its processing format to the appropriate element of an accumulator register, according to the rules described in Section 3.3.3.

For the `csi_mul_acc` instruction, an UOP consists of loading two corresponding elements of the source arithmetic streams, unpacking them (optional), multiplying the values in their processing format with each other and adding the obtained product to the appropriate element of an accumulator register, according to the rules described in Section 3.3.3. If the multiply-accumulate instruction is performed on FP numbers, and multiplication has caused an FP overflow or underflow exception, the addition is not performed.

For `csi_acc_psum` instruction, an UOP consists of the following three operations. First, the current element of the partial sum vector (i.e. the current element of either *pacc_fp* or *pacc_int*), which is pointed to by the *partial sum interruption index (PIX)* field of the stream status register, is added to the first element of the partial sum vector (i.e., $pacc\_fp_0$ or $pacc\_int_0$). Second, the value of PIX is incremented by 1 to point to the following partial sum. The additions of partial sums are finished when all the elements of the partial-sum vector are processed, i.e., when PIX has a value greater than the number of elements in the partial-sum vector. Third, the *pre-final accumulation* UOP is executed. For floating-point accumulation, this UOP consists of placing the value of the $pacc\_fp_0$, which contains the sum of all partial sums into the destination register. For integer accumulation, the UOP consists of shifting and truncating the 24-bit or 48-bit value contained in $pacc\_int_0$ according to the rules described later in this section (see the **Programming Notes** part) and placing the resulting 32-bit value in the destination GP register. Maintaining the number of the current partial-sum element using the PIX allows the `csi_acc_psum` instruction to continue execution from the point of interruption, if an interruption has occurred. We note that PIX is always initialized to 1 at the beginning of execution. If the partial-sum vector consists of just a single element, no additions have to be performed and the pre-final UOP is executed immediately.

For the `csi_zero_pacc` instruction, an UOP consists of setting the current element of the partial sum vector (i.e., the current element of either *pacc_fp* or *pacc_int*), which is pointed to by the PIX field of the stream status register, to zero and of incrementing the PIX.

For the `csi_acc_section` instruction a unit of operation consists of loading *section_size* consecutive elements of the source arithmetic stream, unpacking them (if necessary) to the processing format, accumulating the obtained values to a single number, packing (if required) the obtained value to the storage format of the destination stream, and storing the resulting number at the appropriate location of the destination stream. If the source stream elements $src_{i+1}, \ldots, src_{i+s}$ are accumulated in the current UOP, the next UOP will accumulate values $src_{i+s+1}, \ldots, src_{i+2s}$, where $s = section\_size$. The last UOP that processes stream elements may accumulate less than *section_size* elements, if the source stream length is not a multiple of *section_size*.

Exceptions

For all the instructions listed in Table 3.7, except for `csi_acc_psum`, a specification exceptions occurs in the same situations as for simple arithmetic or logic instructions given in Section 3.3.1. The conditions that cause a specification exception and that are particular to certain accumulation-related instructions, are the following ones:

- For the `csi_acc_psum` instruction: if the *dest_size* or the *src* fields are specified

illegally, i.e., if *dest_size* is specified to be 2 (implying that the accumulation result is a 24-bit value), or if *src* $= 3$ (recall that *src* values 0, 1, and 2 are sufficient to specify all possible source operands for the instruction).

- For the `csi_zero_pacc`, if *src* $= 3$.
- For the `csi_mul_acc` operating on integer data, the data type of the source streams elements in their processing format is specified to be 32-bit (*u32, s32*) for at least one of the operand streams.
- For the `csi_acc` and `csi_mul_acc` instructions: if the destination accumulator register, or its format, does not correspond to the data type of the source streams, i.e., if the rules presented in the descriptions of these instructions in Section 3.3.3 are not satisfied.

For all the instructions listed in Table 3.7, except for `csi_acc_psum` and `csi_zero_pacc`, an operand access exceptions may occur under the same conditions, and will have the same effect as for simple arithmetic logic instructions (see Section 3.3.1). The `csi_acc_psum` and `csi_zero_pacc` instructions do not access data in the storage and, therefore, cannot cause an access exception. For all the instructions listed in Table 3.7, the floating-point exponent overflow/underflow can occur under the same conditions, and will have the same effect as for simple arithmetic logic instructions (see Section 3.3.1).

Programming Notes

Rounding of the Final Result for Integer Accumulation. As described above, when the `csi_acc_psum` instruction accumulates integer values, the result of the accumulation can be rounded off and then truncated by shifting to the right by a certain number of bits before being placed in the destination register. The reason why these operations may be needed is that binary integer data is often used to represent fractions in fixed-point format. We illustrate such situations by the following example. Suppose that a stream of 8-bit unsigned values representing integer numbers in the range 0..255 has to be multiplied with a stream of fractional numbers and then accumulated and rounded off to produce an integer result. Suppose further that the fractions in the second stream are represented in the fixed point format with two fractional bits, i.e., suppose that the decimal point is placed after the 2 least significant bits. For example the value 0.75 in such format will be represented as: $0.75 = 1 \cdot 2^{-2} + 1 \cdot 2^{-1} = 000000.11$. To perform the required multiplication and accumulation, the elements of the stream are multiplied with each other using the `csi_mul_acc`, which produces a sequence of 16-bit values. These values thereupon are accumulated to 24-bit partial sums. The partial sums have two fractional bits as well. The `csi_acc_psum` instruction first adds the partial sums to each other, producing a 24-bit result. This number will also have two fractional bits. We recall, however, that an integer result is needed. Therefore, the instruction will be specified to round off and truncate the two least significant bits of the 24-bit sum before converting it to the required integer format and placing in the destination GP register. Rounding off is performed by adding the value which represents 0.5 in the 24-bit binary fixed-point format. In our example the binary point is positioned to the left

of the 2nd least significant bit and 0.5 is therefore is represented by the 24-bit binary integer $2^{-1} = 0\ldots0.10$.

Initializing Accumulator Registers. In order to obtain the correct results when performing accumulate or multiply-accumulate operations, the programmer needs to initialize the partial sums contained in an accumulator register using the `zero_pacc` instruction.

Accumulator Registers and Interruptions. All the instructions listed in Table 3.7, except for `csi_acc_section`, use the accumulator registers. The accumulator registers contain intermediate results of the instructions. In order to guarantee the correctness of the final result in the presence of an interruption, the interrupt-handling routine of the supervisor has to save the accumulator register contents at the point of interruption and to restore them when control is returned to the user program. This can be done by using the `csi_mf_pacc` and `csi_mt_acc` instructions, which move the data between the accumulator and the scalar registers. These instructions are described in Section 3.3.7.

3.3.4 Stream Reorganization Instructions

The instructions belonging to this class, and their main features, are summarized in Table 3.8. These instructions perform conditional extraction/insertion of elements from one arithmetic stream into another and conditional splitting/merging two arithmetic streams. All instructions have a mask stream operand, which provides the condition values. Instructions, however, are of the *IC* class and, therefore, are independent from the stream-mask mode.

Instruction	Operation	Format	Class	Element types
`csi_extract`	extract stream	*a2b1*	*IC*	*all arithmetic*
`csi_insert`	insert stream	*a2b1*	*IC*	*all arithmetic*
`csi_split`	split stream in two	*a3b1*	*IC*	*all arithmetic*
`csi_merge`	merge two streams	*a3b1*	*IC*	*all arithmetic*

Table 3.8: Stream reorganization instructions.

Main Operation and Operand Interpretation

The `csi_extract` instruction has the *a2b1* format. The *scr1* and *scr2* operand fields designate the SCR-sets of the destination and the source arithmetic streams, respectively, and the *mscr* designates the MSCR-set of the source mask stream. The instruction reads the source arithmetic stream elements that correspond to ones in the mask stream and writes them into the destination stream.

The `csi_insert` instruction has the *a2b1* format as well. The operand fields are interpreted in the same way as for `csi_extract`. The instruction reads the elements of the source stream and inserts them into the destination stream positions that correspond to ones in the mask stream.

The `csi_split` instruction has the *a3b1* format. The *scr1, scr2*, and *scr3* operand fields designate the SCR-sets of the the first destination, the second destination, and the source streams, respectively. The *mscr* field designates the MSCR-set of the mask stream. The instruction reads the elements of the source stream. The elements that correspond to zeroes in the mask stream are written to the first destination stream, and those corresponding to ones are written to the second destination stream.

The `csi_merge` instruction has the *a3b1* format. The *scr1, scr2*, and *scr3* operand fields designate the SCR-sets of the the destination, the first source, and the second source streams, respectively. The *mscr* field designates the MSCR-set of the mask stream. For each element of the mask stream, the instruction performs the following operation: if the mask element is zero, the current element from the first source stream is read, otherwise the element from second source stream is read. The read element is written to the destination stream.

Units of Operation

Although the stream reorganization instructions use CSI mask streams, they are independent of the stream-mask mode. The activation of the stream-parallel mode has the usual effect on the execution of the instructions. Therefore, in this section we present the definition of a unit of operation (UOP) under the assumption that the stream-parallel mode is disabled.

For `csi_extract`, an UOP consists of the following actions. The current element of the mask stream is checked (this may require a load of the byte that contains this element, if the byte was not loaded before). If the element is 1, the current element of the source stream is read, otherwise, no read is performed. In any case, the **Base**, **CurrCol** and **CurrRow** control registers of the corresponding SCR-set are updated so that they point to the next element. If the mask bit was 1 and, hence, the source stream element was read, this element is written to the current position in the destination stream and control registers of the corresponding SCR-set are updated, so that they point to the next element of the destination stream. If the mask bit was 0, the write and the control register update for the destination stream are not performed.

For `csi_insert`, an UOP consists of the following actions. The current element of the mask stream is checked. If the element is 1, the current element of the source stream is read and the **Base**, **CurrCol** and **CurrRow** control registers of the corresponding SCR-set are updated, so that they point to the next element. If the mask stream element is zero, no read and control register update for the source stream are performed. Then, if the element was read, it is written to the current position in the destination stream, otherwise no write to the destination stream is performed. In any of these two cases, the control registers of the destination stream are updated so that they point to the next element in the destination stream.

For `csi_split`, an UOP consists of the following actions. The current element of the source stream is read and the control registers of the corresponding SCR-set are updated so that they point to the next element. Then, the current mask bit is checked. If it is zero, the source stream element, which was read, is written to the first destination stream and the control registers of this stream are updated so that they point to its next element. If the mask bit is one, the read element is written to the second destination stream and the control register of this stream are updated.

For all the stream reorganization instructions just described, checking the mask stream element, which is performed in a unit of operation, implies that the **CurrElem** register of the MSCR-set, which describes the mask stream, is incremented by one.

Exceptions

For all the instructions listed in Table 3.8, a specification or an operand access exception may occur in the same situations as for the simple arithmetic or logical instructions. These situations are described in Section 3.3.1.

Furthermore, the parameters of the arithmetic streams should be specified in such a way that no packing or unpacking will be performed, i.e., for each operand stream the *Misc* register of the corresponding SCR-set should specify its *ElSize* and *ProcessSize* fields in such a way that *ElSize* = *ProcessSize*.

Programming Notes

Recall that the number of elements in a CSI arithmetic or CSI masked stream is also called the *stream length*. For the correct execution of the stream reorganization instructions, a programmer needs to initialize the operand stream parameters that control the stream length correctly. That is, the following conditions should be satisfied.

- For the `csi_extract` instruction:
  - The lengths of the source arithmetic and the mask streams should be equal.
  - The length of the destination stream should be equal to the number of ones in the mask stream.
- For the `csi_insert` instruction:
  - The lengths of the destination arithmetic and the (source) mask streams should be equal.
  - The length of the source arithmetic stream should be equal to the number of ones in the mask stream.
- For the `csi_split` instruction:
  - The lengths of the source arithmetic and the mask streams should be equal.
  - The length of the first destination stream should be equal to the number of zeroes in the mask stream.
  - The length of the second destination stream should be equal to the number of ones in the mask stream.
- For the `csi_merge` instruction:
  - The lengths of the destination arithmetic and the (source) mask streams should be equal.
  - The length of the first source stream should be equal to the number of zeroes in the mask stream.
  - The length of the second source stream should be equal to the number of ones in the mask stream.

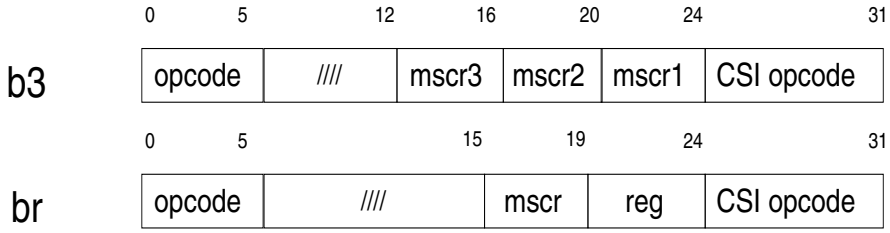


Figure 3.3: Formats of bitstream-operating instructions.

For the calculation of the number of 1's in a mask stream the programmer is advised to employ the `csi_cnt_ones` instruction described in Section 3.3.5

3.3.5 Bitstream-Operating Instructions

The instructions that belong to this group, and their main features, are summarized in Table 3.9. The three logical instructions perform standard logical operations on the corresponding elements (bits) of two bitstreams, and the `csi_cnt_ones` instruction counts the number of ones in a bitstream. All the instructions are of the IC-class, i.e., they are interruptible and are independent of the stream-mask mode.

Instruction	Operation	Format	Class	Element type
`csi_and_bitstr`	AND bitstreams	*b3*	*IC*	*b1*
`csi_or_bitstr`	OR bitstreams	*b3*	*IC*	*b1*
`csi_xor_bitstr`	XOR bitstreams	*b3*	*IC*	*b1*
`csi_cnt_ones`	count ones in a bitstream	*br*	*IC*	*b1*

Table 3.9: Bitstream-operating instructions.

Main Operation and Operand Interpretation

The three logical instructions (`csi_and_bitstr`, `csi_or_bitstr`, and `csi_xor_bitstr`) use the same instruction format and interpret the operand fields in the same way. The instructions use the *b3* instruction format which is presented graphically in Figure 3.3. The *mscrs1, mscrs2* and *mscrs3* operand fields designate the MSCR-sets of the destination, the first and the second source bit streams, respectively. A bitstream logical instruction performs the specified operation on the corresponding elements (i.e., bits) of the source bitstreams and writes the result to the corresponding position of the destination bitstream. The `csi_cnt_ones` instruction uses the *br* format, which is depicted in Figure 3.3. The *mscr* field designates the MSCR-set of the source bitstream and *dest* designates the destination GP register. The instruction counts the number of ones in the source bitstream and places this number in the destination GP register.

Units of Operation.

The activation of the stream-parallel mode has the usual effect on the execution of the instructions listed in Table 3.9. Therefore in this section we present the definitions of a unit of operation under the assumption that the stream-parallel mode is disabled which means that instructions are executed in the stream-sequential mode. The three instructions that perform binary logical operations are very similar to each other and differ only in the main operation they perform. Therefore, we present the way the operand fields and UOP definition only for `csi_and_bitstr`. For this instruction, an UOP consists of loading two bytes that contain the corresponding elements of the source bitstreams, performing the bitwise AND operation on these two bytes and writing the results into the destination bitstream. The **Base** and **CurrElem** of the MSCR-sets that describe the streams are updated in the same UOP, so that they point to the bitstream elements that have to be processed in the next UOP. More precisely, the **Base** register of each set is incremented by 1, and the **CurrElem** register of each set is incremented by $min\{8, \textit{Length} - \textit{CurrElem}\}$, where *Length* denotes the length of the bitstream described by the MSCR-set. For the `csi_cnt_ones`, the following operations are performed in a unit of operation. The source bitstream byte that contains the current bitstream element (i.e., the byte at the address contained in the **Base** register of the corresponding MSCR-set) is read. The number of ones in this byte is computed. This value is added to the contents of the destination GP register and the sum is placed again in the destination GP register. In the same UOP, the control registers of the source streams MSCR-set are updated to point to the following byte, as we have decribed above.

Exceptions and Programming Notes

The only exceptions that may be recognized for the instructions listed in Table 3.9 are the operand access exceptions that can arise when the operand data in storage is accessed. These exceptions have the usual effect on the current UOP (see Table 2.3).

For the `csi_cnt_ones` instruction, the number of ones contained in the byte which is processed in a current UOP is added to the contents of the destination GP. If an interruption has occurred, the contents of this register at the point when control is returned to the interrupted instruction should be the same as its contents at the point of interruption, in order for the result of the instruction to be correct. The supervisor's interrupt-handling routine is responsible for this. It should save the value of the register at the point of interruption and restore it upon the exit from the interruption. Since, for the `csi_cnt_ones` instruction, the number of ones in a byte of the source stream is added to the contents of the destination GP, the application programmer should initialize the destination GP to zero just before `csi_cnt_ones` in order to obtain correct results. The `csi_cnt_ones` instruction is likely to be used together with the stream reorganization instructions, in order to calculate the lengths of conditionally extracted/inserted or splinted/merged streams.

3.3.6 Special-Purpose Instructions

This is an important group of instructions, most of which are particular for CSI. These instructions are included in order to accelerate a number of the most time-consuming kernels for several important multimedia applications, such as MPEG/MPEG-2/MPEG-4 codecs. The instructions in this group perform relatively complex computations on the input data streams and are also referred to as *complex CSI instructions*. Actually, the inclusion of these non-standard complex stream-operating instructions motivated the name of the whole CSI (Complex Streamed Instruction) set. The special-purpose instructions have been selected for inclusion in CSI according to the following criteria:

- The selected instruction should allow an efficient implementation of a code segment that represents a significant fraction of execution time for an important multimedia application.
- The instruction should allow a direct hardware implementation with non-prohibitive hardware requirements in terms of area and delay.

For example, the `csi_sad` instruction is selected because it computes the sum of absolute differences of two data streams which is, when performed on two 16×16 pixel blocks, the main operation of the most time-consuming function of the MPEG-2 encoder, the *motion compensation*. The instruction allows a hardware implementation with acceptable hardware requirements, as was shown in [25]. Currently, we provide complex instructions targeted at acceleration of certain code fragments of MPEG, PNG, MPEG-2 and MPEG-4 codecs, which are typical representatives of image and video coding/decoding application domains. All of the complex instructions currently included in the CSI instruction set, and their main features, are listed in Table 3.10. Below, we give a brief descriptions of these instructions.

Instruction	Operation	Format	Class	Element types
`csi_sad`	SAD of 2 streams	*a2b1r*	*IC*	*u8, s8, u16, s16*
`csi_huff`	Huffman encoding	*a2b1r*	*IC*	*u16*
`csi_vld`	VLD decode	*a2b1r*	*IC*	
`csi_idct`	IDCT	*a2b1*	*IC*	$s16$
`csi_paeth`	Paeth predict/code/decode	*a3b1*	*IC*	$s8, u8$
`csi_acq`	ACQ function of MPEG-4	*a2b1r*	*IC*	$b1$

Table 3.10: Special-purpose instructions.

Main Operations and Operands

The `csi_sad` instruction uses the *a2b1r* format. The *scr1* and *scr2* fields designate the SCR-sets of the source streams, and *reg* designates the destination GP register. Let $(A_i)_{i=0,\ldots,n}$ and $(B_i)_{i=0,\ldots,n}$ be two source streams, A and B. The instruction computes the *sum of absolute differences (SAD)* of the corresponding stream elements, i.e, the value $SAD(A,B) = \sum_{i=0}^{n} |A_i - B_i|$. The result is placed in the destination GP register. An implementation of an arithmetic unit which can perform the SAD operation efficiently is presented in [25].

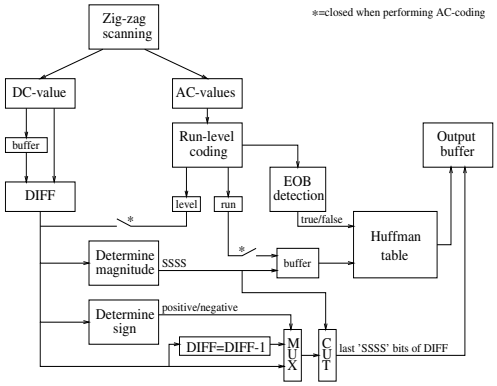


Figure 3.4: Huffman coding.

The `csi_huff` instruction uses the *a2b1r* format. The instruction performs the *Huffman coding* of the arithmetic stream described by SCR-set designated by *scr2* and writes the obtained data to the bitstream specified by the MSCR-set designated by *scr1*. The first parameter of the coding process, the so-called *old DC value*, is contained in the GP register designated by *reg*. The second parameter, the *Huffman table* that has to be used, is designated by the *mscr* field. Below we describe the coding process in more detail. Huffman coding is performed on a stream of 64 16-bit signed integer numbers. The first stream element is called the *DC* coefficient, and all others are called the *AC* coefficients. The process of Huffman coding is parametrized by two values: a scalar, which represents the DC coefficient of the previously encoded block, and a second value that is the number of *Huffman table* to be used. The process of huffman coding is presented graphically in Figure 3.4 and is performed as follows. For DC coding, the DC of the previously encoded block, which is contained in the source GP register, is subtracted from the current DC (i.e., from the first stream element): $DIFF = DC - old\_DC$. If *DIFF* (represented in 2's complement) is negative, 1 is subtracted from it. Then, the magnitude *SSSS* of *DIFF* is calculated, i.e., *SSSS* is the number of bits needed to represent *DIFF*. Finally, *SSSS* is Huffman encoded using the Huffman table designated by *mscr* field and the lower *SSSS* bits are appended to it. The result is written to the destination stream.

The encoding of all other stream elements (i.e, AC coefficients) is performed by the following steps. First, *run-level coding* is performed, producing a sequence of the *run-level pairs*, where *run* represents a number of zeroes until a non-zero AC coefficient (*level*) is encountered. If the sequence of AC values ends with a run of zeroes, a special number, called *End of block (EOB) symbol* is produced. Each run-level pair is then processed as follows. If $level < 0$, then $level = level - 1$. Next, the magnitude *SSSS* of the *level* is determined, and *run* together with *SSSS* are Huffman encoded. When an EOB symbol is detected, it is encoded as well. The result of Huffman encoding is then written to the destination bitstream. For the internal organization of an execution unit that can perform Huffman coding the interested reader is referred to [68].

Finally, we observe that the computations performed by the `csi_huff` instruction are

inherently sequential, The instruction does not perform the same computations on a stream of independent data elements, as most of simple CSI instructions do. Therefore, the instruction execution cannot be organized in a number of uniform operations. The execution consists of a single UOP and, hence, is not interruptible. We remark that the impossibility of interruptions is not likely to cause performance degradation for the instruction because the data stream length is limited to 64.

The `csi_vld` instruction uses the *a2b1r* format. The instruction performs operation which is, essentially, the reverse to that of `csi_huff`. It reads the source bitstream, which represents coded data (so-called (*run, level*) pairs) and is described by the MSCR-set designated by *mscr*, decodes it, writing the decoded *run* and *level* values to the destination streams specified by the SCR-sets designated by *scr1* and *scr2*, respectively. The type of decoding (luminance/chrominance block, VL decoding table, and others) are contained in the register pair (2 consecutive 32-bit registers) designated by *reg*. A description of the hardware that can perform Variable-length decoding (VLD) can be found in [57].

The `csi_idct` instruction uses the *a2b1* format, and interprets the operand fields as follows. The *scr1* and *scr2* fields designate the SCR-sets of the destination and the source arithmetic streams, respectively. The input stream is read in groups of 8 consecutive elements. Next, the *one-dimensional inverse discrete cosine transformation (1-D IDCT)* is performed on each group, producing 8-element groups of transformed values, and the obtained results are written to the destination stream. In general, the result of the 1-D IDCT performed on an 8-element sequence $(f_j)_{0\leq j\leq 7}$ is the sequence of 8 elements $(F_i), 0 \leq i \leq 7$, defined as follows:

$$F_i = \sum_{0\leq j\leq 7} \Lambda_j \cdot \cos(\frac{\pi i}{16}(2j+1)) \cdot f_j,$$

where $\Lambda_j = \frac{1}{\sqrt{2}}$, for $j = 0$ and $\Lambda_j = 1$ for all other cases. It can be observed that the 8-element group of transformed values can be seen as the (column) vector $F = (F_0, \ldots, F_7)^t$ obtained from the vector $f = (f_0, \ldots, f_7)^t$ by means of multiplication with the constant 8×8 matrix $A = (a_{ij})$: $F = A \cdot f$, where $a_{ij} = \Lambda_j \cdot \cos(\frac{\pi i}{16}(2j+1))$. We remark than the input data of IDCT, as well as the matrix coefficients are represented in 16-bit fixed point format. Furthermore, it was also observed that due to the special structure of the matrix A, the number of multiplications and additions needed for 1-D IDCT can be significantly reduced. A number of such efficient IDCT computation methods has been proposed in the DSP domain [7]. A 1-D IDCT thus requires a small limited number of 16-bit arithmetic operations, making it possible to develop efficient hardware implementations of the algorithm at acceptable cost. For an example of such an implementation, the interested reader is referred to [56].

The `csi_paeth` instruction uses the $a3b1$ format and interprets the operand field as follows. The $scr2$, and $scr3$ fields designate the SCR-sets of the destination and of the two source arithmetic streams, which represent two consecutive source image scanlines. The $scr1$ field designates the SCR-set of the destination stream, which represents the scanline of the destination image. The value x of the 2 leftmost bits of the $mscr$ fields are used to encode the type of operation performed by the instruction. The instruction can perform three different operations: the *Paeth prediction* ($x = 0$), the *Paeth encoding* ($x = 1$), and the *Paeth decoding*($x = 2$). Let a, b, c, and d be 4 corresponding pixels of the two source streams, positioned as shown in Figure 1.5, and b and d be the current pixels of these streams. Then

the result of the Paeth prediction operation is simply the value of the predictor *pred* for the pixel d (computed according to the algorithm presented in Figure 1.4). The results of the Paeth encoding and decoding for the current pixel are defined as $d - pred$ and $d + pred$, respectively. It was shown by other researchers that these operations allow a direct hardware implementation with acceptable area and delay requirements [25].

The `csi_acq` instruction has the $a2b1r$ format. It computes the *Accepted Quality Function* of the two 16×16 binary alpha blocks (BABs) according to the MPEG-4 standard. The BABs are specified by the SCR-sets designated by $scr1$ and $scr2$. The computed ACQ value is stored in GP register reg. The $mscr$ field designates the GP register which contains the *alpha threshold* parameter of the ACQ function. The computation of the ACQ function for two 16×16 BABs is based on computation of the SAD function on their 4×4 sub-blocks. For a detailed description of the operations involved in computation of the ACQ function, as well as for an example hardware implementation of it, the interested reader is referred to [39].

We state here once again that the proposed set of complex should be considered as a prototype, and by no means as being complete. New complex instructions are likely to be included after thorough analysis of the considered image/video codecs, as well as after analysis of the applications originating from other multimedia domains, such as audio or image processing.

Units of Operation

For compactness of the presentation we will describe the units of operation just for the `csi_sad` instruction. The `csi_sad` instruction is of IC-class and thus independent of the stream-mask mode. The definition of unit of operation depends, however, on whether the stream-parallel mode is enabled or not. Suppose first that the mode is disabled (i.e., the stream-sequential mode is enabled). An UOP consists of performing the following operations. The corresponding elements A_i and B_i of two source streams are loaded, their absolute difference $|A_i - B_i|$ is computed and is added to the contents of the first 24-bit element $pacc\_int24_0$ of the *pacc_int* register. Let m denote the *SIMD_width* for a given CSI implementation, and n denote the size in bytes of the stream elements in their processing format (i.e., $n = 1$ or $n = 2$, see Table 3.10). If the stream-parallel mode is enabled, a UOP consists of loading $\frac{m}{n}$ corresponding elements of the source streams, computing the SAD for these two sequences of numbers, adding the obtained value to the contents of $pacc\_int24_0$, and placing the sum back to $pacc\_int24_0$. After all stream elements are processed, in the *pre-final* UOP the contents of $pacc\_int24_0$ are zero-extended to 32-bits and placed in the destination GP register. Then, the instruction is completed in the *final* UOP which performs the same operations as for any other instruction: advancing the program counter and resetting the *CurrRow* and *CurrCol* registers of the appropriate SCR-sets to zero.

Exceptions and Programming Notes

For compactness of the presentation we will describe the exceptional conditions only for the `csi_sad` instruction. The operand access or specification exceptions may occur for the instruction in the same situations as for the simple arithmetic and logical instructions (see Section 3.3.1). Note that the instruction uses the 0'th element $pacc\_int24_0$ of the packed ac-

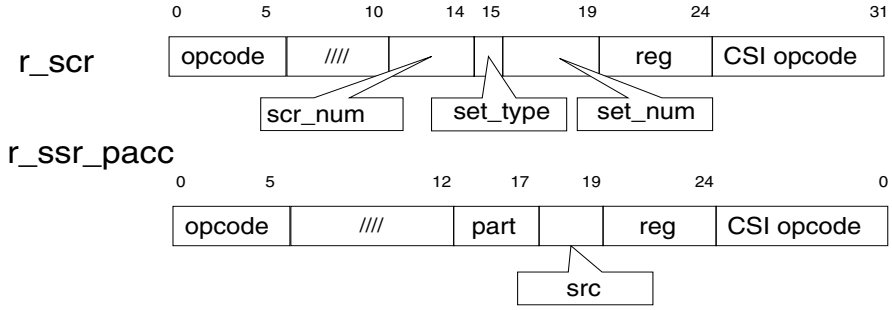


Figure 3.5: Formats of auxiliary instructions.

cumulator register *pacc_int*. In order to obtain correct results, the programmer should ensure that the value of the element is set to zero prior to execution of `csi_sad`. This can be done by using either the `csi_pacc_zero` or the `csi_mt_ssr_pacc` instruction, which are described in Section 3.3.3 and Section 3.3.7, respectively.

3.3.7 Auxiliary Instructions

The instructions belonging to this group, and their main features, are summarized in Table 3.11. The instructions are used to move data between the scalar registers of the underlying ISA and the CSI registers. The instructions are of the NC-class and thus are not interruptible.

Instruction	Operation	Format	Class	Data type
`csi_mt_scr`	move to SCR	*r_scr*	*NC*	*u32*
`csi_mf_scr`	move from SCR	*r_scr*	*NC*	*u32*
`csi_mt_ssr_pacc`	move to SSR/pacc	*r_ssr_pacc*	*NC*	*u32*
`csi_mf_ssr_pacc`	move from SSR/pacc	*r_ssr_pacc*	*NC*	*u32*

Table 3.11: Auxiliary instructions.

Main Operation and Operands

The `csi_mt_scr` instruction uses the *r_scr* format which is shown graphically in Figure 3.5. The *reg* field designates the source GP register, *set_num* designates the number of the destination SCR/MSCR-set, *scr_num* designates the number of a destination control register from the set, and the *set_type* field specifies whether the destination is SCR set (*set_type* $= 0$) or a MSCR-set (*set_type* $= 1$). The instruction moves the 32-bit value contained in the source GP register (treated as a bit pattern, i.e., as a value of *u32* type) to the *scr_num*-th control register of the SCR/MSCR-set designated by the *set_num* and *type* fields. The control registers within the SCR/MSCR-sets are numbered according to the conventions presented in Section 2.1. The `csi_mf_scr` instruction performs the reverse operation to that of `csi_mt_scr`, moving the contents of a control register to a GP register. It uses the *r_scr*

format, and the operand fields are interpreted as follows. The *reg* field designates the destination GP register. The *set_num, set_type* and *scr_num* fields designate the source control register in the same way as it is done for the `csi_mt_scr` instruction.

The `csi_mt_ssr_pacc` instruction moves the contents of a 32-bit scalar (general-purpose or floating-point) register to a certain 32-bit part of the stream status register (SSR), the integer packed accumulator register (*pacc_int*), or the floating-point accumulator register *pacc_fp*. The value is moved as a bit-pattern, without any type conversion, i.e., the operation can be seen as a move of a value that has *u32* data type. Note that all of these three registers have a size in bits that is a multiple of 32. Indeed, the number of bits in the SSR is 128 and thus a multiple of 32. The sizes of *pacc_fp* and *pacc_int* are $8 \cdot$ *SIMD_width* and $24 \cdot$ *SIMD_width* bits, respectively. Since *SIMD_width* is always a multiple of 4, both of these values are multiples of 32. Each of the three registers can be seen as a number of consecutive 32-bit parts. The parts are numbered from 0. Thus, the SSR consists of 4 32-bit parts $SSR = (SSR\_32_0, SSR\_32_1, SSR\_32_2, SSR\_32_3)$. The *pacc_fp* register consists of $\frac{1}{4}$*SIMD_width* parts:

$$pacc\_fp = (pacc\_fp\_32_0, \ldots, pacc\_fp\_32_{\frac{1}{4}SIMD\_width-1}).$$

The *pacc_int* register consists of $\frac{3}{4}$*SIMD_width* parts:

$$pacc\_int = (pacc\_int\_32_0, \ldots, pacc\_int\_32_{\frac{3}{4}SIMD\_width-1}).$$

The `csi_mt_ssr_pacc` instruction has the *r_ssr_pacc* format and interprets the operand fields as follows: the 5-bit *reg* field designates the destination GP register, and the 2-bit *long_reg* field designates the source register, which is either SSR ($long\_reg = 0$), or *pacc_fp* ($long\_reg = 1$), or *pacc_int* ($long\_reg = 2$). The *part* field designates the number of the 32-bit part of the source register.

The `csi_mf_ssr_pacc` instruction performs the operation opposite to that of `csi_mt_ssr_pacc` and uses the *r_ssr_pacc* format as well. The *reg* operand field designates the source operand GP register, and the *long_reg* and *part* fields designate the destination 32-bit part in the same way as it is done for `csi_mt_ssr_pacc`.

Exceptions

All the auxiliary instructions are of the NC class. They consist of a single UOP and are not interruptible. Execution of an auxiliary instruction can, however, result in a specification exception, in which case the instruction is suppressed. A specification exception occurs under the following conditions:

- For the `csi_mt_scr` and `csi_mf_scr` instructions: If the designated control register does not exist. Namely, if *scr_num* designates a number bigger than 7 for an SCR-set, or bigger than 2 for an MSCR-set.
- For the `csi_mt_ssr_pacc` and `csi_mf_ssr_pacc` instructions:
  - If the designated part of the designated *long register* (i.e., the part of *SSR*, *pacc_int* or *pacc_fp*) does not exist. For example, if *long_reg* designates SSR and $part > 3$.
  - If *long_reg* designates a non-existing long register, i.e., if $long\_reg = 3$.

Programming Notes

The instruction that moves data to control registers (`csi_mt_ssr_pacc`) is usually employed to set up the stream parameters. Since an SCR-set contains 8 registers, 8 instructions are needed to initialize the stream parameters for a single arithmetic stream. This overhead, however, is not likely to decrease the performance substantially, because the instructions will be executed by the fast general-purpose host processor. Furthermore, it is common in multimedia applications that the same operation has to be performed on a number of streams that have the same parameters, except for the base address. For example, in the *motion estimation* routine of the MPEG-2 encoder, the *sum of absolute differences (SAD)* operation has to be performed on a large number of 16×16 pixel blocks. In such a case, only the *Base* register of the corresponding SCR-set has to be initialized for each computation except for those of the first block (*CurrRow* and *CurrCol* are initialized automatically to zeroes, and all other control registers do not have to be changed). This allows the amortization of the overhead associated with the setup of stream parameters over a large number of CSI computational instructions. The instruction that can move data to the SSR (`csi_mt_ssr_pacc` can be used to activate/deactivate the stream-mask mode or the stream-sequential/stream-parallel mode by setting up the corresponding bits of the SSR. All of the auxiliary instructions are likely to be used by the supervisor program to save/restore the CSI register state upon entering/exiting interruption routines.

3.4 Program Execution

In this section we first describe how the execution of a program that contains CSI instructions is performed in general. Then we present more details about the actions which are performed during execution of CSI instructions that operate on CSI (arithmetic and bit) streams, concentrating in particular on the description of the packing/unpacking transformations.

We first recall the assumptions which are made earlier in Section 2.1 and Section 2.3.3. It is assumed that CSI serves as an extension of a general-purpose 32-bit RISC-like ISA, which is also referred to as the *underlying ISA*, or the *baseline ISA*. The underlying ISA contains the *Process Status Word (PSW)* register, the *Process Identifier (PID)* field of which is used to identify the currently executed program (also referred to as the current *process*). The instruction that has to be executed at any given moment of the execution is located in the storage at the address contained in the *Program Counter (PC)* field of the PSW. The execution of the current instruction consists of performing the main operation it specifies and updating the PC to point to the next instruction to be executed. For all the instructions, except for branch and jump instructions, the PC is incremented by 4 to point to the next instruction in the storage (recall that all instructions are 32-bits, i.e. 4 bytes, and the storage is byte addressable). The instructions of the underlying ISA consist of a single unit of operations, in which the specified main operation and the PC update occur. If such an instruction causes an exception, the UOP may be completed, nullified, suppressed, or terminated, according to the UOP outcome specified by the baseline ISA for a given exception type. Termination results in the termination of the executed program. The effects of the other three outcomes are summarized in Table 3.12. If an instruction has not caused any exceptions, it is completed.

UOP Outcome	Program Counter at	Result Location
Completed	Next instruction	Changed
Nullified	Current instruction	Unchanged
Suppressed	Next instruction	Unchanged

Table 3.12: Effect of different outcomes of baseline ISA's instructions.

The NC-class CSI instructions consist of a single UOP and are executed in the same way as the baseline ones. The CSI instructions of all other classes (IM, IC and IP) consist of multiple UOPs and are seen by the programmer as being performed sequentially, one UOP after another, in the order described below. For IM- and IC-class instructions, the UOPs consist of processing the stream data and are performed in the natural order, which is determined by the order at which the stream elements are accessed according to the definition of the operand streams (see Section 2.2.1 and Section 2.2.2). For IP-class instructions, the UOPs consist of processing of the elements of the accumulator registers *pacc_int* or *pacc_fp*. The UOPs are performed in the order induced by the numbering of the elements. The IM-, IC-, and IP-class instructions can be interrupted during the execution with interruptions being allowed between UOPs. After an interruption is handled, the instructions may be resumed from the UOP at which the interruption occurred, or from the next following UOP, according to the outcome of the current UOP (i.e., the last UOP not completed at the point of interruption). The possible UOP outcomes for interruptible instructions have been presented earlier and their effects on the CSI state are summarized in Table 2.2.

As stated before, the semantics of the executed program implies that instructions are executed in order, i.e., sequentially, one after another. The host processor that fetches and decodes (sometimes just partially) all the instructions and performs execution of the baseline ones, is likely to be implemented as a superscalar engine, which may execute instructions out of their program order. The NC-class and IP-class CSI instructions operate only on the CSI register state and, therefore, can be easily implemented in the host superscalar processor in such a way that their out-of-order execution does not result in a change of the program semantics. The IM- and IC-class CSI instructions, however, operate on the CSI streams located in storage and after being identified by the host processor are further decoded and executed by the *CSI execution unit* (also referred to as the *CSI coprocessor*). Therefore, if such CSI instructions are executed out-of-order with the scalar load/store instructions of the baseline ISA, which are executed by the host processor, the original program semantics can change. The designer of the host processor should take precautions so that such situations do not occur. For a superscalar host processor this can be achieved, for example, by employing the following mechanism. As soon as a CSI IM- or IC-class instruction is detected by the host processor, the processor stops fetching and decoding any further instructions. The CSI execution unit waits till all the preceding scalar memory instructions have been executed. Then it decodes the CSI instruction and executes is. The host processor waits until all the storage updates specified by the CSI instructions are performed and then resumes fetching and decoding.

Packing and Unpacking. In the individual descriptions of the CSI instructions, it was mentioned that for instructions operating on arithmetic streams, an UOP may perform packing or

unpacking of the stream elements. These operations may be performed when an arithmetic stream contain binary integer elements. Below, we describe these operations in detail. Consider an arithmetic stream str. Two formats, the *storage format* and the *processing format*, are specified for stream elements by the *Misc* register of the corresponding SCR-set. The storage format is characterized by the number of bytes a stream element occupies in storage. It is specified by the *ElSize* field of the *Misc* register. The processing format is characterized by the number of bytes an element requires when being processed and by the interpretation of the element as a signed or an unsigned number. The size of an element in bytes in its processing format is specified by the *ProcessSize* field of the *Misc* register. If the *Sign* bit is set the values are treated as signed, otherwise as unsigned.

The unpacking (i.e., conversion from storage to computation format) is performed for the elements of the arithmetic streams that are source operands of an instruction. We recall that *ElSize* holds the number which is one less than the number of bytes required for an element in storage. *ProcessSize* holds the number which is one less than the number of bytes required to represent an element in its processing format. Unpacking consists of the following operations:

- If *Sign* $= 0$, the $8 \cdot (ElSize + 1)$ bits, which represent a stream element in its storage format, are zero-extended to $8 \cdot (ProcessSize + 1)$ bits (of course, the bits are not changed if *ElSize* $=$ *ProcessSize*). If *Sign* $= 1$, the $8 \cdot (ElSize + 1)$ bits representing an element are sign-extended to $8 \cdot (ProcessSize + 1)$ bits. Of course, if *ElSize* $=$ *ProcessSize*, these operations are not performed.

- The result of the operations we just described, is then shifted, as an $8 \cdot (ProcessSize + 1)$ bit value, by the number of bits specified in the *ShiftAmount* field of the *Misc* register. If *ShiftDirection* = *1* the shift direction is right and the shift is arithmetic if *Sign* $= 1$, or logical if *Sign* $= 0$. If *ShiftDirection* $= 0$, the direction is left and shifting is performed logically, irrespectively of the value of the *Sign* field.

The left direction is more commonly used during unpacking and represents the multiplication of stream elements by a power of 2. For example, suppose that a stream *str1* of unsigned 8-bit integer values has to be added to a stream *str2* of signed 16-bit values that represent fixed-point binary integers with 4 fractional bits. Then, the elements of the first stream will be converted to the format of the second one by zero-extending them to 16 bits and shifting left (logically) by 4 bits. Figure 3.6 presents graphically how this operation will be performed, for example, for the number $129 = 10000001$. We remark that if *ElSize* $>$ *ProcessSize* for a source arithmetic stream of an instruction, the specification exception is recognized, since unpacking is impossible.

The packing (i.e., conversion from the computation to the storage format) is performed for the elements of the arithmetic stream that is the destination operand of an instruction. Suppose the number x is a destination stream element in its computation format, i.e., it is a result of the main operation performed by a CSI instruction. Let *Round, ProcessSize, ElSize, ShiftDirection*, and *ShiftAmount* represent the values of the corresponding fields of the *Misc* register of the SCR-set that describes the destination stream. We recall here that according to the definitions given in Section 2.2.1, the sizes of the elements in storage or processing

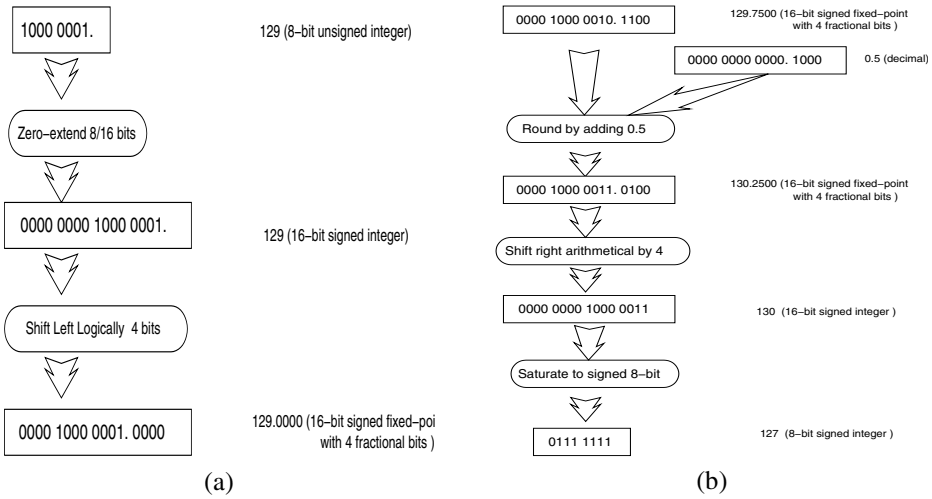


Figure 3.6: Unpacking and packing in CSI.

format, which are specified by the *ElSize* and the *ProcessSize* fields, are larger by 1 than the actual values of these fields. For example, if *ElSize* $= 0$, the size of a stream element in the storage format is equal to *ElSize* $+ 1 = 1$ byte. Packing of x to its storage format consists of the following operations:

- If the *Round* bit is set to 1, the $8 \cdot$ *(ProcessSize* $+ 1)$-bit binary integer value $2^{ShiftAmount-1}$ is added to x. This value represents the number 0.5 in the $8 \cdot$ *(ProcessSize* $+ 1)$-bit fixed-point format with *ShiftAmount* fractional bits.

- If *ShiftDirection* $= 1$, the obtained value is shifted right by the *ShiftAmount* bits. In such a case, the shift is logical if *Sign* $= 0$, or arithmetical if *Sign* $= 1$. If *ShiftDirection* $= 0$, the obtained value is shifted left (logically) by the *ShiftAmount* bits.

- If *Saturate* $= 1$, the obtained value is then saturated to the number in the range which can be represented in the storage format. If *Sign* $= 0$, the representable range is $[0, 2^{8\cdot(ElSize+1)} - 1)]$, and if *Sign* $= 1$, the representable range is $[-2^{8\cdot(ElSize+1)-1}, 2^{8\cdot(ElSize+1)-1} - 1]$. If *Saturate* $= 0$, nothing is done to the result of shift.

- The $8 \cdot$ (*ElSize* $+ 1$) rightmost bits of the obtained number represent the stream element in the storage format and are extracted to be stored in the destination stream.

Note. Programmers should not specify *Round* $= 1$ if the shift direction for the destination stream is left. However, the right direction is usually used for packing. It represents division of the destination stream element by a a power of 2 and is used when the element in the processing format contains fractional bits and the element in the storage format should be an integer without fractional bits. Shifting to the right discards the fractional

bits. Figure 3.6(b) presents, for example, how a signed 16-bit number with 4 fractional bits $129.75 = 0001000000001.1100$ is packed to the 8-bit storage format to produce the (signed) number $127 = 01000000$.

Saturated Addition and Subtraction. Saturation may be performed not only during packing but, for the CSI addition and subtraction instructions, directly during the execution of the main operation. Addition and subtraction performed with saturation arithmetic are very important since they allow these instructions to produce meaningful results using shorter processing formats. Suppose, for example, that two streams of 8-bit unsigned numbers have to be added. Addition of two such numbers may result in an unsigned 9-bit result. If the processing format was specified to be 8-bit, the result of addition may overflow and the most significant 9th bit will be discarded. Such behavior results in the wrap-around effect, i.e., the result of addition is the sum modulo 8 (for example, $255 + 1 = 256 \bmod 8 = 0$). This effect, however, is undesirable. It may, for example, create artifacts when used in image-processing, as it was shown in Section 1.1. One way to avoid a wrap-around effect is to specify larger processing format, for example 16-bit, and to specify the destination format to be 8-bit. Then, the value 256 will be saturated during the packing stage producing the desirable result 255. However, specifying the larger processing format (this process is usually called *promotion*) results in reducing the number of stream elements that can be processed in parallel. Suppose, for example, that *SIMD_width* of the CSI execution unit is 8. If the elements are promoted to 16-bit format, only 4 of them can be processed in parallel.

Within CSI this inefficiency can be avoided by using the same technique as employed by some other multimedia ISA extensions such as MMX. Namely, CSI allows addition and subtraction instructions to be performed directly with saturation arithmetic. Two stream elements that have y-bit storage format can be added or subtracted in the same y-bit processing format resulting in a $(y + 1)$-bit sum. This sum is then immediately saturated to the specified range ($[0, 2^y - 1]$ if values are unsigned, or $[-2^{y-1}, 2^{y-1} - 1]$ if values are signed) to produce an y-bit result, which can then be stored at the destination stream without packing. This capability allows the CSI to utilize the functional units more efficiently. For example, if *SIMD_width* $= 8$, and two streams of 8-bit values have to be added, employing of saturation arithmetic allows 8 additions to be performed in parallel, while if standard arithmetic was used and the values were promoted to 16-bit, only 4 additions could have been performed. The type of arithmetic used by the CSI addition/subtraction instructions is specified via the *Saturate* bit of the *Misc* control register that corresponds to the first source arithmetic stream. If the bits is set to 1, the instruction uses the saturation arithmetic, otherwise it uses the standard one. All other CSI instructions are independent of the value of the *Saturate* bit of their operand streams.

3.5 Conclusions

In this chapter we presented the CSI instruction set in detail and described how CSI instructions are executed.

First, the CSI instruction formats were listed, and the most common of them were described in detail. For each of them we showed how an instruction that has a certain format is

divided in a number of fields and how the fields are interpreted. Next we described the CSI instruction classes, into which instructions are divided with respect to the number of elements they process, interruptibility, and dependency on the stream-mask mode. For instructions that belong to the same class, the number of elements they process is determined in the same way. Such instructions have the same behavior in the presence of interrupts and react in the same way on enabling/disabling of the stream-mask mode. After this, the complete CSI instruction set was presented. CSI instructions were organized in the following groups with respect to the type of the operations they perform: *Simple Arithmetic and Logical, Comparison-Related, Accumulation-Related, Stream Reorganization, Bitstream-Operating, Special-Purpose*, and *Auxiliary*. For each group, we described the instructions that belong to it and the way they are encoded and executed. Thereupon we have presented the auxiliary operations of converting stream data between storage and computation formats (packing and unpacking), which can be performed by a CSI instruction next to the execution of its main operation. In this chapter we also provided a detailed description of the CSI accumulation facility and how it is used. We would like to note that the CSI as presented in this dissertation is a prototype architecture that is intended to prove a concept rather than an architecture proposed to be readily developed into a product. It is likely with an additional investigation that more instructions, particularly those performing complex operations, can be added to complete the instruction set. We encourage such study as a future research topic.

In the following chapters we will present the design of a hardware unit which can execute the CSI instructions and evaluate the performance benefits of superscalar general-purpose processors enhanced with such a unit.

Chapter 4

An Example Implementation

In the previous chapters we have presented the CSI architecture, describing the CSI state, the set of CSI instructions, the operations they perform and how they are executed. In order to prove feasibility of a CSI implementation, in this chapter we present the design of a unit that can execute CSI instructions. This unit is further referred to as the *CSI unit* or the *stream unit*. Since CSI instructions are memory-to-memory, the organization of a particular CSI unit depends on the decision to which level in the memory hierarchy it interfaces.

The chapter is organized as follows. Section 4.1 describes a general design of the stream unit, without specifying the level of memory hierarchy it interfaces to. Section 4.2 presents a detailed organization of the CSI unit under the assumption that the unit is interfaced with the L1 data cache. The discussion in this section is focused on the control organization of such a unit. Section 4.3 presents a detailed description of the address-generation hardware for a L1-interfaced CSI unit. We show that the complex, two-dimensional, address-generation calculations needed to address CSI arithmetic streams can be performed in a pipelined fashion and implemented using a three-stage pipeline with acceptable delay. In this section we present a simplified version of the address-generation hardware that is capable of addressing the two-dimensional arithmetic streams for which two consecutive groups do not overlap, i.e., streams for which each group contains just a single stream element (see Section 2.2.1). In Section 4.4 we present some concluding remarks.

4.1 General Organization of the CSI Unit

We recall that a typical CSI instruction, such as `csi_add`, has two input stream and one output stream. The instruction loads the source streams from memory, unpacks (if necessary) the stream elements from storage to computational format, performs a certain operation on corresponding elements, packs (again if necessary) the results, and stores the resulting output stream back to memory. Since these operations are independent, they can be pipelined. The CSI execution unit is, therefore, organized as a pipeline in which stream data flows through a sequence of stages that perform these operations. The datapath of this unit is depicted in Figure 4.1. For clarity, some parts (for example, floating-point hardware) have been omitted.

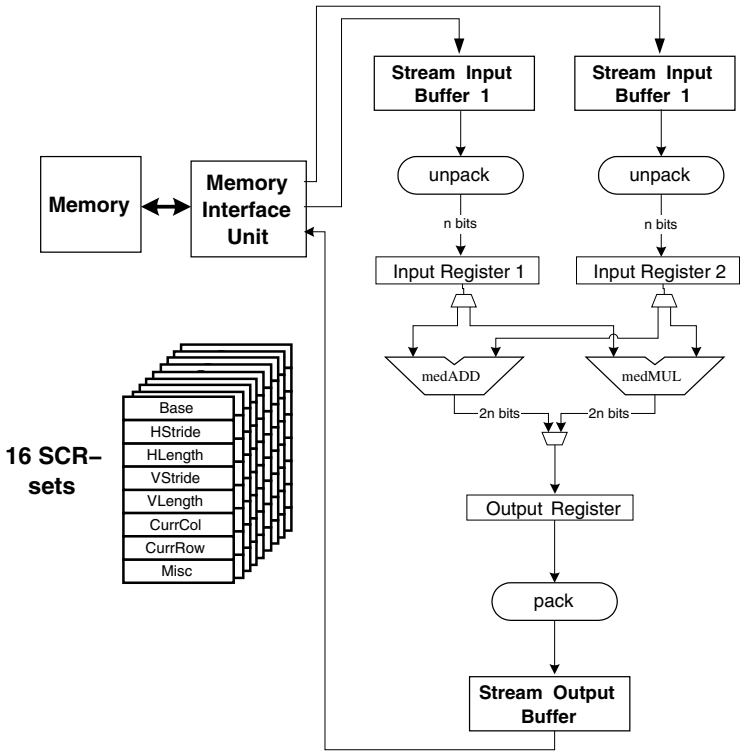


Figure 4.1: Datapath of the CSI execution unit

Section 4.2 describes the control logic.

The main hardware entities of the streaming execution unit are the *stream control register sets* (SCR-sets), *memory-interface unit* (MIU), the *stream input* and *stream output* buffers, the *pack* and *unpack* units, and one or more SIMD-like functional units. In Figure 4.1, two SIMD functional units, medADD and medMUL, are shown that perform addition-related and multiply-related operations, respectively. Detailed descriptions of the parts of the CSI execution unit will be presented later in this chapter.

The memory interface unit is responsible for transferring data between the memory hierarchy and the stream input buffers. In addition, if the source stream elements are not stored consecutively, it must also extract and store them consecutively in the stream buffers. If the destination stream elements are not stored consecutively, the unit must perform the reverse operation, scattering data into appropriate memory locations.

Each unpack unit converts stream data from storage format to computational format (if required). Unsigned fixed-point numbers are zero-extended (additional bits are filled with zeroes) while signed numbers are sign-extended. Additionally, the values can be scaled by means of shifting in order to position the binary point. These operations are controlled by the fields of the Misc register of the corresponding SCR-set.

The functional units medADD and medMUL perform SIMD parallel operations on the

data contained in the input latches. If these input latches are n bits wide, these units process either $n/8$ bytes in parallel, $n/16$ halfwords, or $n/32$ words. The value of n is implementation dependent. It can be 64, 128, or even larger. As mentioned before, this feature allows CSI codes to take advantage of a wider datapath without recompilation. The output register is $2n$ wide so that no overflow occurs during computation. The medADD unit performs the usual addition, subtraction, and bitwise logical operations, as well as addition-related operations such as the *Paeth* operation [25]. It also expands the output to $2n$ bits by padding it with zeroes in order to produce the same number of bits as the *medMUL* unit. The medMUL unit performs the packed multiply operation and could also incorporate more complex media operations, hardwired or firmwired. For example, the unit can be enabled to perform the Sum of Absolute Difference (SAD) and the Paeth prediction operations [25], and the operations described in [58, 59].

From the output register, data flows to the stream output buffer via the pack unit. The pack unit converts, if necessary, the data from computational format to storage format under control of the *Misc* register of the destination stream. When no conversion is needed, data is passed through the unit without being changed.

The design of the CSI execution unit is strongly influenced by the memory hierarchy level it is connected to. It can be connected to the first-level (L1) cache, or it can bypass the L1 cache and go directly to the L2 cache or even main memory. We decided to interface it to a 2-ported L1 cache. The motivation for this design decision is as follows. First, Ranganathan et al. [52] observed that with realistic L1 cache sizes, most multimedia applications achieve high hit rates. Our experiments ([8, 9, 33]) support this observation: with a 32KB direct-mapped L1 data cache, JPEG and MPEG-2 coders/decoders as well as the 3D graphics benchmark *viewperf* exhibited hit rates of over 99%. Second, Burger et al. [6] has predicted that the pin bandwidth will be a critical consideration for future microprocessors, because the number of available off-chip connections (pins) cannot be easily increased. However, since the L1 cache is on-chip, widening the path between the cache and the CSI execution unit, so that an entire cache line can be brought in in a single access, does not require extra off-chip connections. Furthermore, such a design provides high data bandwidth without increasing the number of cache ports, which is undesirable since multi-ported caches are expensive and, furthermore, may increase the cache hit time. Therefore, in the remainder of this chapter we assume that the CSI execution unit is interfaced with the L1 cache.

4.2 Organization of the CSI Unit Interfaced to L1 Cache

This section presents a detailed description of the CSI exection unit, concentrating on the design of the parts which are particular to its L1 cache interface and on the organization of the control. The design of the most complex part of the unit, the address-generators, is presented in Section 4.3.

Figure 4.2 depicts the CSI datapath and the control lines. Thick lines represent paths through which data and addresses move and thin lines are control lines. Rounded rectangular and trapezium shapes are used for computational units and boxes for storage between pipeline stages. In this figure, the stream address-generators (boxes labelled AG), the load and store queues (LQ and SQ), and the extract and insert units represent collectively the implementa-

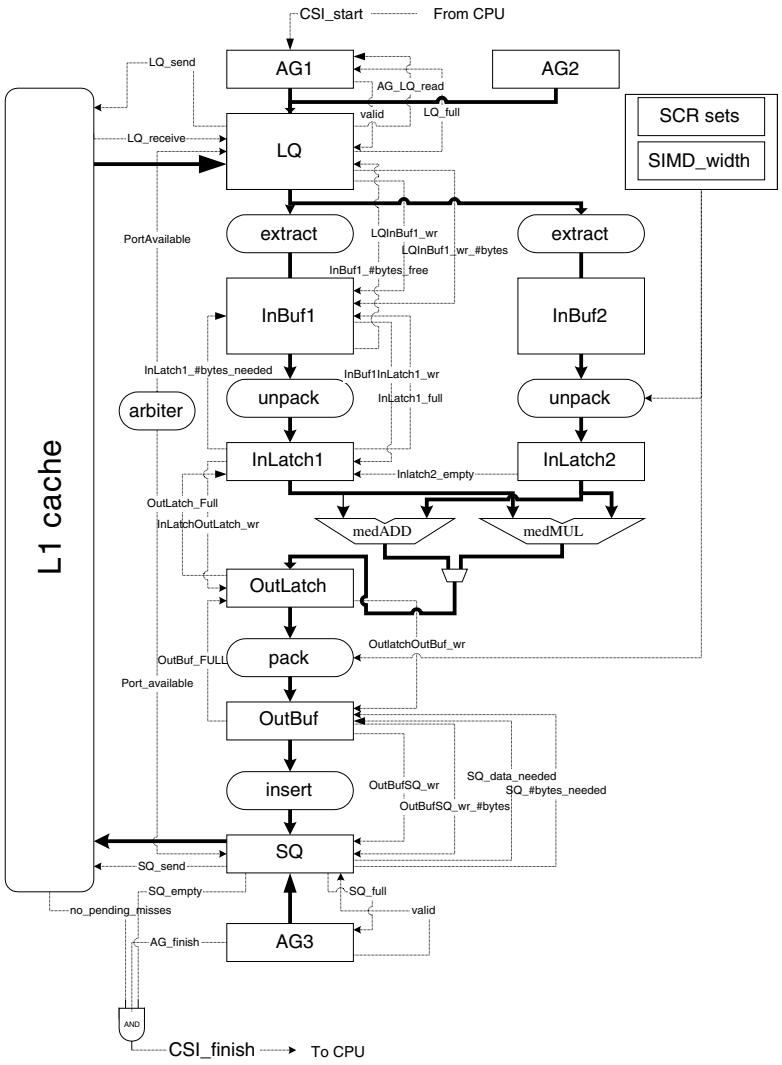


Figure 4.2: Datapath of L1-interfaced CSI unit with the control lines added.

tion of the memory-interface unit (the box labelled MIU in Figure 4.1). The latencies of the pipeline stages are estimated in terms of machine cycles, where one cycle is assumed to be approximately twice as long as the delay of a 32-bit adder.

General Control Organization

The control logic is distributed across the pipeline stages. Each control signal is associated with a storage entity located between two consecutive pipeline stages. The control organiza-

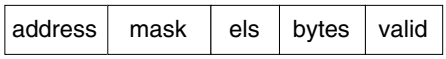


Figure 4.3: Format of an AG record.

tion guarantees that if storage entity S_i is full, no data generated by the preceding pipeline stage will be written to S_i and destroy valid data in it. Therefore, one signal sent by the control logic of S_i to the control logic of the previous storage entity S_{i-1} is, usually, the signal S_i*_full*. According to this control organization, the CSI pipeline operates in general as follows: at the beginning of each cycle the control logic associated with S_i receives the control signals from the control associated with the next storage element S_{i+1}. Based on these signals and its current state, the control logic associated with S_i updates its state and decides whether the data can be transferred to the next pipeline stage. If so, the transfer is performed and the control logic associated with S_{i+1} is notified. After that, it checks the notification control signal generated by the control logic associated with the previous storage entity S_{i-1} if data is transferred from S_{i-1} to S_i. Based on this signal, the control logic associated with S_i again updates its state.

Stream Address Generators

Each address generator associated with an input stream (AG1 and AG2) produces a sequence of records consisting of addresses and some bookkeeping information needed to extract stream elements from a cache block. As illustrated in Figure 4.3, each AG record consists of the following fields.

- The *addr* field contains the address (aligned at a cache block boundary) of the cache block from which stream data should be extracted.
- Let *bsize* denote the size of an L1 data cache block in bytes. The *mask* field is a (*bsize* $\cdot \log_2$(*bsize*))-bit *position mask*. It indicates which bytes in the cache block contain stream data. If a byte in the block belongs to the stream, the corresponding $\log_2$(*bsize*)-bit value is equal to the order of this byte among all stream bytes in the block. For example, for the first byte in the block that belongs to the stream, the corresponding position mask value is 1. For bytes that do not belong to the stream, the value is equal to zero.
- The *els* field contains the number of stream elements contained in the cache block.
- The *bytes* field contains the number of bytes belonging to the stream and contained in the cache block. It is equal to the value of the *els* field multiplied with the value of the *ElSize* field of **Misc** control register from the corresponding SCR-set.
- Finally, the *valid* field is a 1-bit flag which signifies if the record is valid. Invalid records may be generated by an AG when the end of a row of a 2-dimensional stream has been reached.

In Section 4.3 we describe how these fields are computed. Here we describe the control signals received and generated by each AG and how they influence their operation.

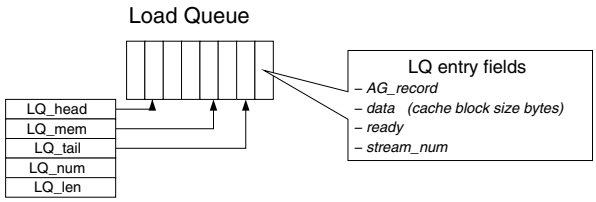


Figure 4.4: Organization of the Load Queue.

- The *CSI_start* signal is generated by the host CPU. If this signal is set, the fields of the first AG record for the cache block corresponding to the base address of the stream are calculated and the AG starts generating the records.
- The *LQ_full* control signal is generated by the load queue (LQ). When this signal is set, the AG pipeline is stalled. We remark that since each AG is organized as a 3-stage pipeline, up to three valid AG records can be in-flight when the signal is received. To guarantee that none of them is lost, each AG contains a FIFO buffer with 3 entries.
- The AG_i*_LQ_wr* signal is generated by the control of AG_i ($i = 1, 2$) to notify the LQ that an AG record is transferred. The signal is asserted if *LQ_full* is deasserted and the first record in the FIFO of AG_i is valid. In this case, the record is dequeued from the FIFO and sent to the LQ.

The Load Queue

The Load Queue (LQ) fetches data from the L1 data cache and passes it further through the pipeline. Figure 4.4 depicts its organization. It is organized as a circular queue where each entry consists of an AG record and three extra fields, *data*, *ready*, and *stream_num*. The *data* field is *bsize* bytes large and contains the cache block fetched from the L1 cache. The *ready* field is a 1-bit flag indicating that the data has arrived. The *stream_num* field indicates to which input stream (1 or 2) the entry belongs. In the experiments reported in [33, 67], an 8-entry LQ was used.

The internal state of the LQ consists of registers that hold information needed to implement a circular queue (LQ_head, LQ_tail, LQ_length, and LQ_size), and of the LQ_mem register which points to the first entry that has not yet been sent to the L1 cache. We now describe the signals received by the control associated with the LQ from other parts of the CSI pipeline, the actions they initiate, and the signals generated by the LQ control and sent to other parts of the pipeline. The four sequences of actions described below are performed in parallel during each cycle.

1. The control associated with the LQ checks the number of the input stream associated with the first LQ entry. Suppose it is the first stream. Then the LQ control receives the *InBuf1_#bytes_free* signal from the control of InBuf1 and generates the *LQ_InBuf1_wr* signal. This signal is asserted if the cache block for the first LQ entry has arrived and InBuf1 has enough free space to accommodate all bytes in the block that belong to the stream. In this case, the *data* and *mask* fields of the first LQ entry are sent to the extract

unit which extracts the stream data from the block according to the mask and places it into the input buffer InBuf1. The $\log_2(bsize)$-wide control signal *LQInBuf1_wr_#bytes* notifies the control of InBuf1 of the number of stream bytes contained in the block. After this, the first LQ entry is discarded.
Since the calculations for generating the *LQInBuf1_wr* signal are simple, they will only take a fraction of a cycle. We therefore deduce that the LQ control is able to perform the same calculations for the second LQ entry in the same cycle and, therefore, pass two cache blocks to the extract stage during a single cycle. This makes the design balanced because, for a two-ported L1 cache, two cache blocks can be delivered to the LQ in one cycle.

2. If there were less than two free entries in the load queue at the end of the previous cycle, then the *LQ_full* signal is asserted to stall the AGs. Two free entries are needed because two cache blocks can be delivered in a cycle. Otherwise, two entries are appended to the LQ, the *LQ_AG_read* signals are sent to AG1 and AG2, and the first valid AG record of each AG are transferred to the *AG_record* fields of the next two LQ entries.
3. The *ports_available* signal is received from the cache port arbiter responsible for sharing the cache ports between the load and store queues. If the signal is set and there are two LQ entries which have not yet accessed the L1 cache, the LQ control sends their *address* fields to the cache ports and increments the LQ_mem register.
4. The *data_available* signal is received from the L1 cache controller. If it is asserted, cache blocks are received from the ports and written to the *data* field of the corresponding LQ entries and the ready flags of these entries are set. Since we assume a 2-ported L1 cache, two cache blocks can be received in a single cycle.

Extract Units and Stream Input Buffers

Each extract unit receives data of a loaded cache block from an LQ entry, extracts the bytes belonging to the stream, and stores them consecutively in one of the input buffers (see Figure 4.5). An extract unit is implemented as a $bsize \times bsize$ switch controlled by the *mask* field of the LQ entry. It is assumed that the delay of an extract unit is less than one cycle. Each $\log_2(bsize)$-bit entry of the *mask* field controls the position to which the corresponding byte is routed. If the entry is zero, the corresponding byte is discarded.

Each stream input buffer is a shift buffer that stores stream elements consecutively. ¿From there the data is passed to the unpack units. Each input buffer has two control registers. The buf_data register points to the position of the first byte (least recently received from the extract unit) which is not yet passed to the *Unpack* stage. In the buf_size register the size of the buffer (in bytes) is hardwired. We now describe the control organization of the first input buffer and its interface with other parts of the CSI pipeline. The second buffer operates identically. In each cycle the following sequence of actions is performed.

1. The *InLatch1_full* signal is received from the next storage entity in the CSI pipeline, InLatch1. This signal is asserted if InLatch1 is filled with stream data that has not yet been consumed by the SIMD execution units. If the signal is deasserted, the control logic calculates how many bytes of stream data are needed to fill InLatch1. If InBuf1 contains the required number of bytes, they are sent to the *Unpack* stage and removed from the

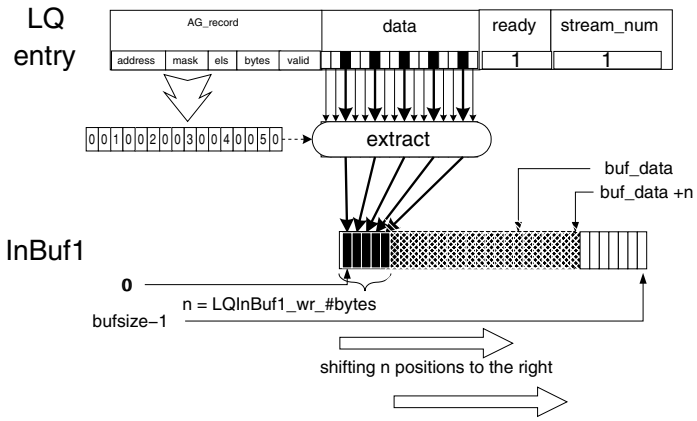


Figure 4.5: Extracting the bytes that belong to the stream.

buffer by adjusting the value contained in the buf_data register. The *InBufInLatch1_wr* signal is then asserted to notify InLatch1.

2. The *LQInBuf1_wr* signal is received from the LQ. If it is asserted, the contents of InBuf1 are shifted n positions to the right and the n leftmost bytes are transferred from the extract unit to the n leftmost positions in InBuf1, where n is the value of the *LQInBuf1_wr_#bytes* control signal generated by the control of the LQ (cf. Figure 4.5). After this, the buf_data register is incremented by n and the value of *buf_size-buf_data* is sent to the control of the LQ via the *InBuf1_bytes_free* signal.

Unpack Units

Each unpack unit receives stream data from its corresponding input buffer and converts (unpacks) it, if needed, from storage to computational format. Unpacking consists of converting an m-bit value to a larger width by sign-extending (for signed fixed-point numbers) or zero-extending (for unsigned numbers) it. Additionally, the promoted value can be shifted, which is performed to reposition the fractional point. The converted data is stored in the input latches located in front of the SIMD functional units.

SIMD Functional Units

After the input data has arrived at the input latches in the computational format, the arithmetic or logical operation specified by the CSI instruction is performed by the SIMD functional units. Two input latches, InLatch1 and InLatch2, are located in front of the SIMD execution units, each of which is n bits wide, where $n = 8 \cdot$ *SIMD_width* and where *SIMD_width* is the number of bytes processed in parallel by the SIMD functional units. The following sequence of actions is performed for InLatch1 in each cycle. The same actions are performed for InLatch2.

1. The *OutLatch_full* signal is received from the output latch of the SIMD units, *Out-*

Latch. If *InLatch1_full* and *InLatch21_full* indicate that both input latches are full and *OutLatch_full* is deasserted, the SIMD computation on the data contained in the input latches is triggered and the *InLatchOutLatch_wr* is asserted to signal to the OutLatch that a transfer has taken place. After that, the *InLatch1_full* and *InLatch21_full* are reset.

2. The *InBuf1InLatch1_wr* signal is received from the control of InBuf1. If it is asserted, the unpack unit writes data to InLatch1 and the *InLatch1_full* flag is set.

The results of the SIMD execution units flow to OutLatch, the storage entity located after the SIMD functional units. The control associated with OutLatch generates a 1-bit flag *OutLatch_full*. In each cycle the following actions are performed. The *OutBuf_full* signal is received from the next storage entity in the CSI pipeline, the stream output buffer Outbuf. If this signal is deasserted and the Outlatch contains a valid result, then the result is sent to the *Pack* stage and the notification signal *OutLatchOutBuf_wr* is asserted. Simultaneously, the *OutLatch_full* flag is set to the value of the *OutBuf_full* signal and passed to InLatch1. If the flag was set, the input latch will stop triggering new SIMD operations in the next cycle.

We remark that the latency of the medMUL unit is assumed to be at least two cycles and that the unit is fully pipelined. Because of this, there can be two or more valid results in-flight at the moment the *OutLatch_full* signal is asserted. To guarantee that this data is not lost, the output latch is not organized as a single register but as a FIFO buffer with s entries, where s is the number of pipeline stages needed to implement the medMUL unit so that it has a throughput of one result per cycle.

Pack Unit

The pack unit converts (*packs*), if required, the output stream data from computational to storage format. Packing is the reverse operation of unpacking: the elements are shifted (usually, to the right) and then truncated. Additionally, elements can be rounded prior to shifting and saturated when being truncated.

Stream Output Buffer and Insert Unit

The stream output buffer OutBuf operates similarly to the the stream input buffers. It receives the *SQ_data_needed* and *SQ_bytes_needed* signals from the store queue (SQ) in order to determine whether data should be transferred from OutBuf to the SQ through the insert hardware unit. If the transfer should occur, the control of the output buffer asserts the *OutBufSQ_wr* and *OutBufSQ_wr_#bytes* signals and sends them to the SQ. After this, the control logic associated with OutBuf generates the *OutBuf_full* signal and sends it to OutLatch. The signal is asserted if the number of free bytes in OutBuf is less than the number of bytes in the output latch. The insert unit performs the reverse operation of the extract units. It takes from the output buffer the stream bytes that belong to the same cache block and inserts them in the appropriate positions of the *data* field of the SQ entry which requested the insert operation. The positions are determined by the *mask* field of the entry. We note that only the elements which correspond to non-zero mask positions are written.

The Store Queue

The store queue (SQ) is organized and operates similarly to the load queue. It is responsible for storing the output stream data in memory. During each cycle the following three sequences of actions are performed in parallel.

- The SQ control receives the *Port_available* signal from the cache arbiter. If it is active and there are entries in the queue which have already received data from the output buffer but is not yet submitted to the L1 cache, they are sent to the cache and the *SQ_send* signals are asserted. It is assumed that two entries can be transferred to L1 during one cycle.
- If there is a free entry in the SQ and a valid record in the FIFO of the output stream AG AG3, the *SQ_AG_read* is asserted and the first valid record from the FIFO is transferred to the *AG_record* field of the SQ entry.
- The *OutBuf_wr* and *OutBuf_wr_#bytes* signals are received from the output buffer. If they are set, *OutBuf_wr_#bytes* are transferred from the buffer to the *data* field of the appropriate SQ entry. After this, the *SQ_data_needed* and *SQ_#_bytes_needed* signals are generated. The first signal is asserted if there are SQ entries for which addresses are generated but data has not yet been received. The second one is set equal to the number of bytes needed by the oldest entry.

4.3 CSI Address Generators

The task of the CSI address generators is to produce the addresses of cache blocks containing stream elements and information needed to extract data from a cache block. In this section we describe how these AGs can be implemented. Each AG should produce the following information:

- The addresses $B_{1,1}, \ldots, B_{1,k_1}$, $B_{2,1}, \ldots, B_{2,k_2}$, ..., $B_{m,1}, \ldots, B_{m,k_m}$ of all cache blocks that contain stream elements. Each address $B_{i,j}$ is a multiple of the L1 cache block size *bsize*. k_i is the number of cache blocks containing elements of row i of the 2-dimensional stream.
- The sequence $mask_{1,1}, \ldots, mask_{m,k_m}$ of position masks. Each mask $mask_{i,j}$ consists of *bsize* $\log_2(bsize)$-bit values. The k-th value is equal to zero if the cache block does not belong to the stream. Otherwise, it indicates the order of the k-th byte among all bytes that belong to the stream.

Consider, for example, the 2-dimensional stream illustrated in Figure 4.6 and assume that the block size is 8 bytes. The base address of this stream is 804, the horizontal stride is 3, the row length is 4, the element size is 1, and the number of rows is 2. For this stream the AG should generate successively the addresses 800, 808, 904, 912, and the mask vectors $(0,0,0,0,1,0,0,2)$, $(0,0,1,0,0,2,0,0)$, $(1,0,0,2,0,0,3,0)$, and $(0,1,0,0,0,0,0,0)$.

This section is structured as follows. Section 4.3.1 presents formulas for calculating the addresses and the mask vectors. Since thesse calculations are rather complex and do not appear to fit in one machine cycle, a pipelined implementation capable of generating one AG record each cycle is developed and presented in Section 4.3.2.

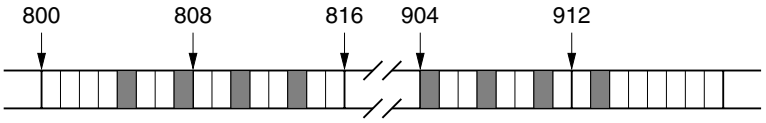


Figure 4.6: A stream stored in memory.

4.3.1 Formulas for Calculating Addresses and Mask Vectors

Let *hstr* denote the horizontal stride of a stream. Throughout this section we assume that *hstr* $\leq$ *bsize*. If this is not the case each cache block contains only one stream element and, therefore, CSI instructions will not achieve a higher throughput than scalar instructions. We also assume that *hstr* > 0.

We introduce the following notations and definitions.

1. $A_{i,j}$ denotes the address of the first stream element in the cache block starting at address $B_{i,j}$.
2. $ofs_{i,j}$ is the offset of the first stream element to the block boundary: $ofs_{i,j} = A_{i,j} - B_{i,j}$
3. $els_{i,j}$ is the number of stream elements contained in the block starting at $B_{i,j}$.
4. By $rel_{i,j}$ (*Row Elements Left*) we denote the number of stream elements contained in the blocks starting at $B_{i,j}, \ldots, B_{i,k_i}$: $rel_{i,j} = \sum_{l=j}^{k_i} els_{i,l}$
5. Let x and y be integers. We use the symbol $\%$ for the modulo operation ($x\%y = x \bmod y$) and the symbols $\sim$ and $\&$ for the bitwise NOT and AND operations.

The calculations performed by each address generator are based on the following formulas.

Proposition 1 *Let SCRS be the stream control register set for which an address generator generates addresses. Then the following equations hold:*

$$A_{1,1} = \textit{SCRS.Base},\ A_{i+1,1} = A_{i,1} + \textit{SCRS.Vstride} \tag{4.1}$$

$$B_{i,1} = A_{i,1}\&\sim(\textit{bsize} - 1),\ B_{i,j+1} = B_{i,j} + \textit{bsize} \tag{4.2}$$

$$ofs_{i,1} = B_{i,1}\&(\textit{bsize} - 1) \tag{4.3}$$

$$els_{i,j} = \lceil(\textit{bsize} - ofs_{i,j})/\textit{hstr}\rceil \tag{4.4}$$

$$rel_{i,1} = \textit{SCRS.HLength},\ rel_{i,j+1} = rel_{i,j} - els_{i,j} \tag{4.5}$$

Given the offset $ofs_{i,j}$ of the first stream element in a block to the block boundary, the offset $ofs_{i,j+1}$ in the next block can be calculated using the following theorem.

Theorem 4.1 *If $ofs_{i,j}$ is the offset of the first stream element in a block to the block boundary, then*

$$ofs_{i,j+1} = (ofs_{i,j} + hstr - bsize \% hstr) \% hstr \tag{4.6}$$

PROOF: Let k be the smallest integer such that

$$ofs_{i,j} + k \times hstr \geq bsize.$$

Then

$$\begin{aligned} ofs_{i,j+1} &= (ofs_{i,j} + k \times hstr) \% bsize \\ &= ofs_{i,j} + k \times hstr - bsize. \end{aligned}$$

Because $ofs_{i,j+1}$ must be smaller than *hstr*, we can take the modulo *hstr* on both sides and obtain

$$\begin{aligned} ofs_{i,j+1} &= ofs_{i,j+1} \% hstr \\ &= (ofs_{i,j} + k \times hstr - bsize) \% hstr \\ &= (ofs_{i,j} - bsize) \% hstr. \end{aligned}$$

In order to implement it in hardware, we want to add a positive term to $ofs_{i,j}$. We, therefore, exploit the fact that $(a + b)\%n = (a\%n + b\%n)\%n$ and add the term *hstr* to the inner expression to obtain

$$ofs_{i,j+1} = (ofs_{i,j} + hstr - bsize \% hstr) \% hstr.$$

□

Generation of Block Addresses. Equations (4.1), (4.2), and (4.5) are sufficient to construct the sequence of the block addresses $B_{i,j}$. Suppose that $B_{i,j}$ is calculated. If the next block containing stream elements belongs to the same row, then its address is obtained using the second part of (4.2). Otherwise, it is the first block of the following row. In this case, first, the address $A_{i+1,1}$ is obtained using the second part of (4.1). The required address $B_{i+1,1}$ is then given by the first equation of (4.2). The row termination decision and which calculations should be performed can be based on (4.5). The row termination condition is simply $rel_{i,j+1} \leq 0$.

Generation of Mask Vectors. The mask $mask_{i,j}$ can be derived from the base mask *bmask* which is determined by the stream horizontal stride and the element size and is constant for a given stream. The calculations are based on (4.6), (4.4), (4.5), and the following definitions.

Definition 4.1 *msb_mask(bsize,x) denotes the $bsize \cdot \log_2(bsize)$-bit binary number for which the x most significant bits (i.e., leftmost bits) are set to 1 and all other bits to 0.*

Definition 4.2 *Given a stream with base address 0, horizontal stride hstr, row length HLength, and element size elsize. Let HLength $\geq$ bsize. By bmask(bsize, hstr, elsize) we denote the mask for the cache block starting at address 0.*

Consider, for example, the base mask *bmask(8, 3, 1)*. This mask is a 24-bit vector consisting of eight 3-bit numbers. Elements 0, 3, and $6 = 2 \cdot 3$ are equal to 1, 2, and 3, respectively. All other elements are equal to 0 because the corresponding bytes do not belong to the stream. So

$$\textit{bmask(8, 3, 1)} = 001\ 000\ 000\ 010\ 000\ 000\ 011\ 000.$$

Because the element size *elsize* is always a power of two, it is easy to show that if the mask for a stream with a given stride and element size of one (byte) *bmask(bsize, hstr, 1)* is known, the mask for a stream with an element size *elsize*) of 2 or 4 bytes, *bmask(bsize, hstr, elsize)*, can be obtained using a short sequence of binary shift and AND operations.

The following proposition presents formulas for calculating $mask_{i,j}$.

Proposition 2 *Let $w = \log_2(bsize)$. $mask_{i,j}$ can be calculated using the following equations:*
If the cache block is not the last block in a row, then

$$mask_{i,j} = bmask(bsize,hstr,elsize) \gg (ofs_{i,j} \cdot w), \tag{4.7}$$

where $\gg$ stands for the shift right logical operation.
If the cache block is the last block in a row, then

$$\begin{aligned} mask_{i,j} \quad = \quad & (bmask(bsize,hstr,elsize) \gg (ofs_{i,j} \cdot w)) \ \ \& \\ & msb\_mask(bsize,(y+ofs_{i,j}) \cdot w) \end{aligned} \tag{4.8}$$

where $y = rel_{i,j} \cdot hstr + elsize$.

PROOF: Equation (4.7) is obvious. Let $C_{i,j}$ be the last block in the i'th row, i.e., let $j = k_i$. Then the number $rel_{i,j}$ gives the number of stream elements in $C_{i,j}$. Equation (4.8) is based on the observation that $rel_{i,j}$ can be less than $els_{i,j}$. We notice that for the mask $mask'_{i,j}$ obtained using only the first part of (4.8), it might be necessary to zero the last several nonzero mask values, because they correspond to bytes which lie after the row end. This is exactly what is done by the bitwise AND operation on the right-hand side of (4.8). Indeed, the number of the leftmost bytes in the block which might belong to the stream is equal to $z = rel_{i,j} \cdot hstr+elsize+ofs_{i,j}$. The remaining *bsize-z* bytes definitely do not belong to the stream. The mask $msb\_mask(bsize, (y+ofs_{i,j}) \cdot w)$ used in (4.8) corresponds to this situation, having each of the z leftmost w-bit values set to $11\ldots1$ and each of the remaining *(bsize-z)* values set to $00\ldots0$. This concludes the proof. □

4.3.2 A Pipelined Implementation

The formulas presented in Proposition 1, Theorem 4.1, and Proposition 2 are sufficient to generate an AG record for each cache block that contains stream elements. It is important

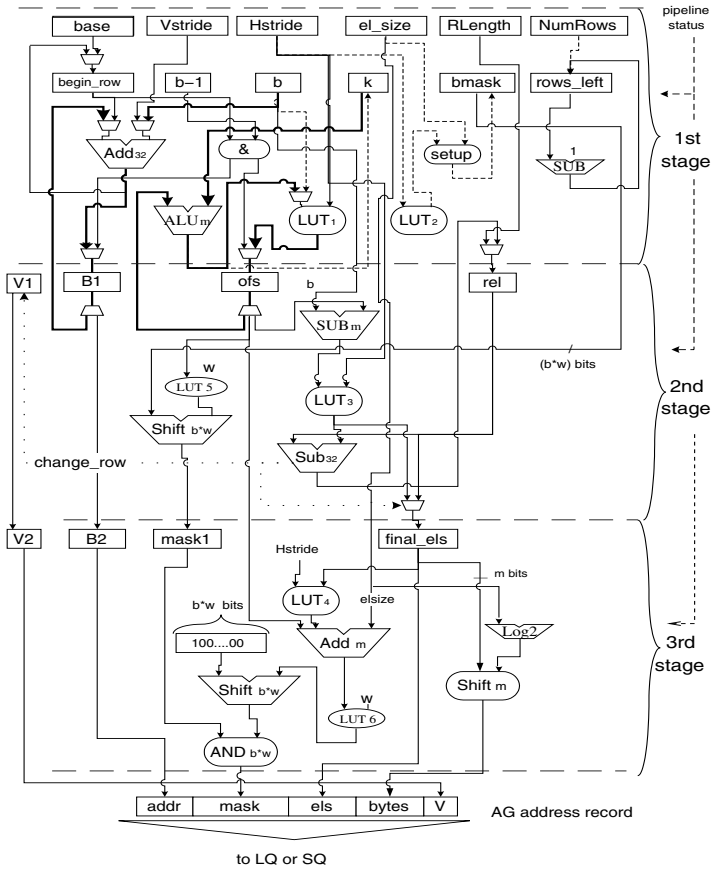


Figure 4.7: Organization of a CSI address generator.

to determine whether these calculations can be implemented in hardware at acceptable cost, what their latency are, and whether they can be pipelined. The possibility of pipelining is critical (if the latency is more than one cycle), because otherwise the rate at which AG records are produced will be less than the rate at which they are consumed by the L1 cache ports, and the performance of the CSI unit might become limited by the address generators. In this section we present a pipelined implementation of the CSI AGs with a latency of three machine cycles, where the machine cycle time is assumed to be approximately twice as long as the delay of a 32-bit adder.

An address generator is organized as a 3-stage pipeline as depicted in Figure 4.7. In this picture, rectangles denote the internal registers and rounded and trapezium shapes are used for processing hardware. Below we describe the pipeline stages and the operations they perform in detail. To make the figure clearer, the number b is sometimes used instead of *bsize*. The symbol w denotes $\log_2(bsize)$, and m denotes $w + 2$.

First Stage. Let *str* be the stream assigned to the AG. The stream is described by a certain SCR-set S. During the first stage the block addresses $B_{i,j}$, the offset of the first stream element in the block $ofs_{i,j}$, and the number of the elements remaining in the same row $rel_{i,j}$ are computed for the current cache block $C_{i,j}$. These values are stored in the pipeline latches B1, ofs, and rel, respectively.

The registers in the top row hold copies of the corresponding stream control registers from the SCR-set S. The registers in the second row contain the following values: begin_row holds the address $A_{i,1}$ of the first stream element in the current row $A_{i,1}$, b-1 holds the value of *bsize-1*, and b holds the value of *bsize*. The register k contains the number $hstr - bsize\%hstr$. The bmask register contains the base mask *bmask(bsize, hstr, elsize)*. Finally, rows_left is a counter which contains the number of stream rows (excluding the current one) for which address information has yet to be generated.

Depending on whether $C_{i,j}$ is the first block of the first row, the first block of some other row or not the first block of any row, data can flow along different paths through the first stage. There are three possible situations.

- $C_{i,j}$ is the first block in a row that is not the first one, i.e., $i > 1$ and $j = 1$. The paths used in this case are shown as thin lines in Figure 4.7. The following actions take place simultaneously:
  - The values of $A_{i,1}$ contained in the begin_row register and of $(bsize\text{-}1)$ contained in the b-1 register are fed into the functional unit labeled "&". This unit simultaneously computes the values of $A_{i,1}\&(bsize\text{-}1)$ and of $A_{i,1}\&^{\sim}(bsize\text{-}1)$. These values are written into the B1 and ofs registers, respectively.
  - begin_row is incremented by the contents of VStride using the 32-bit adder Add_{32}.
  - The value contained in HLength is transferred to the rel latch.
  - The rows_left counter is decremented by one.

  Each of these actions can be done in one cycle.
- $C_{i,j}$ is not the first block of the ith row. The paths used in this case are shown as thick lines in Figure 4.7. The following actions take place in parallel:
  - Register B1 is incremented by *bsize* using the 32-bit adder Add_{32}. After that, B1 contains the *bsize*-aligned address of $C_{i,j}$.
  - The contents of the ofs latch (which is equal to $ofs_{i,j-1}$ at the beginning of the cycle) is incremented by the value contained in the register k using the m-bit arithmetic unit ALU_m. The result is routed to the LUT_1 unit together with the contents of HStride. This unit computes $a \bmod b$, where a is its first input and b its second. The result is written to ofs. These two operations implement Equation (4.3). Thus, on the edge of the cycle *ofs* will contain the value of $ofs_{i,j}$.

  Incrementing B1 fits in one cycle. The m-bit addition using the ALU_m unit will take a fraction of a cycle. The subsequent m-bit modulo division will most likely not fit into one cycle if LUT_1 is implemented as an m-bit divider. However, we observe that the width of the first input of LUT_1 is limited by $2 \cdot bsize$:

$$ofs_{i,j-1} + hstr - bsize\%hstr \leq bsize + hstr \leq 2 \cdot bsize.$$

Furthermore, the second input is constant for a given stream. Therefore, if the remainders (x mod *hstr*) are precalculated for all x ($0 \leq x \leq 2 \cdot$ *bsize*), the LUT_1 unit can be implemented as a look-up table with $2 \cdot$ *bsize* entries. For a typical block size of *bsize* $= 32$, it will have 64 entries. The critical path in this case goes through ALU_m and LUT_1. It is assumed that the critical path fits in one cycle.
According to our experience, most multimedia streams have horizontal strides of 1, 2, 3, or 4. Therefore, if the hardware budget allows, the look-up tables for these strides can be hardwired in the LUT_1 unit. One more table is needed for the case that the stride is different. In this case the table should be loaded with the precalculated remainders before the AG can start generating addresses. This setup operation might take a few cycles. The parts of the AG datapath used for this setup operation are not shown in Figure 4.7. Note that the total number of different tables that should be precalculated is limited by *bsize*, because *hstr* $\leq$ *bsize*. A similar approach can be applied to the LUT_2 look-up table described below.

- Finally, if $C_{i,j}$ is the first block of the first row (i.e., $i = j = 1$), the same actions will take place as for any other first block in a row but, prior to this, some setup actions are performed. The parts of the datapath of the first stage which are utilized during these actions are shown as dashed lines in Figure 4.7. These setup operation may require several cycles.
  - If the stride is non-standard, the look-up tables LUT_1 and LUT_2 are loaded.
  - The content of the Base register is transferred to begin_row and the content of the NumRows register to rows_left.
  - The contents of the b and HStride registers are routed to LUT_1, and the resulting value (*bsize*$\%$*hstr*) is written to *ofs*. From there it is passed to the ALU_m unit which computes *hstr* $-$ *bsize*$\%$*hstr*. The result is written to the register k.
  - Simultaneously, the number *hstr* contained in HStride is sent to LUT_2 which returns the value of *bmask(bsize, hstr, 1)*. This value is sent to the setup unit which generates the mask *bmask(bsize, hstr, elsize)* by means of shift and bitwise AND operations and writes it to the bmask latch.

Which parts of the datapath of the first stage are used and which of the three operation sequences described above are performed is determined by the *pipeline_status* control signal generated by the pipeline stage controller. This signal depends on the *CSI_start* signal provided by the host CPU, the *change_row* signal generated by the second stage of the pipeline, and the stream horizontal stride *hstr*.

Second Stage. During this stage the total number of stream elements in the block is computed and some initial mask calculations are performed. This stage has two possible modes of operation: *setup* and *process*, which are activated by the *pipeline_status* signal. When the stage is operating in the *setup* mode, the look-up table LUT_3 is loaded (if the horizontal stride is non-standard). The corresponding parts of the datapath are not shown in Figure 4.7.

When the stage operates in the *process* mode, the following actions are performed in parallel.

- The V1 flag which indicates if the record generated for $C_{i,j}$ is valid (i.e., whether $j \leq k_i$) is copied to V2.
- The block address is copied from B1 to B2.
- The mask stored in bmask is shifted to the right by $ofs_{i,j} \cdot w$ bits and written to mask1. This action carries out the first computation on the right-hand side of Equation (4.8). The multiplication of $ofs_{i,j}$ with w is implemented by the look-up table LUT_5.
- The SUB_m unit calculates *bsize-ofs*$_{i,j}$. The result is passed to the look-up table LUT_3, which implements integer division by *hstr*. The output of LUT_3 is equal to $els_{i,j}$ calculated according to Equation (4.4). It is then subtracted from $rel_{i,j}$ stored in rel. If the result is negative then $C_{i,j}$ is the last block in the row and the *change_row* signal is generated. If the signal is asserted, the V1 flag is set to zero, invalidating the record for $C_{i,j+1}$ for which the calculations are started by the first pipeline stage during the current cycle. The signal also controls the multiplexer in front of the final_els latch and is communicated to the pipeline stage controller.

The critical path of this stage goes through SUB_m, LUT_3, and SUB_{32}. It is assumed that the critical path fits in one machine cycle.

Third stage. During this stage the mask computations are completed. This stage also has two possible modes of operation: *setup* and *process*, which are activated by the *pipeline_status* signal. When the stage operates in the *setup* mode and the stream has a non-standard stride, the look-up table LUT_4 is loaded. When the stage operates in the *process* mode, the following actions are performed in parallel.

- The registers V2 and B2 are copied to the corresponding fields of the appropriate entry of the AG FIFO buffer.
- The look-up table LUT_4 computes the product of the m-bit number contained in the final_els latch and the horizontal stride. The product is routed to the 3-input m-bit adder Add_m which calculates $z = y + ofs_{i,j}$, where y is computed according to the formula presented in Proposition 2. The hardwired $(b \cdot w)$-bit constant $10\ldots0$ is shifted arithmetically to the right by $z \cdot w$ bits, resulting in the mask *msb_mask(bisze,* $z \cdot w$) (see the last term of Equation (4.8)). By bitwise ANDing this mask with the mask contained in mask1 the final mask is obtained which is written to the corresponding field of the appropriate entry of the AG's FIFO buffer. The multiplication of z by w is implemented by LUT_6. We note that the $(b \cdot w)$-bit shifter Shift_{b*w} is simpler than a common $(b \cdot w)$-bit shifter because shifting can be done only by amounts that are a multiple of the constant w.
- The number contained in the final_els register is shifted to the right by $\log_2(elsize)$ bits, producing the number *final_els* $\cdot$ *elsize*, which is written to the *bytes* field of the FIFO entry.

The critical path of this stage goes through the LUT_4, Add_m, LUT_6, Shift_{b*w}, and AND_{b*w} units. It is assumed that the critical path fits in one machine cycle.

This concludes the description of the AG. The presented pipelined implementation is able to produce a new AG record every cycle. We observe that since the *change_row* signal is

generated during the second stage, a one-cycle bubble will appear in the pipeline when a new row is started. However, the *change_row* signal is, in fact, generated by the carry bit produced by the Sub_{32} unit of the second stage. The carry can be produced before the subtraction itself has finished and, therefore, the signal can become available before the end of the cycle. Since the unit labeled "&" that calculates the block address of the first block in the next row $B_{i+1,1}$ consists of just one level of logic gates, the bubble can be removed if an extra 32-bit adder which calculates duting every cycle $A_{i+1,1} = A_{i,1} + \mathit{VStride}$ is added to the first AG pipeline stage and connected to the unit labeled *"&"*.

4.4 Conclusions

In this chapter we presented how a CSI execution unit can be implemented. First, a general design of such a unit was given. A CSI typical instruction loads the stream data, unpacks it if required, performs the main operation, packs the results and stores them. Since these operations on different stream elements are, usually, independent, they can be pipelined. Therefore, the proposed general organization of such a unit has a pipeline structure. The design of this unit depends on the level of memory hierarchy to which it is interfaced. In this chapter we provided a detailed description of a unit that interfaces to the first-level data cache. This design decision was motivated by the particular characteristics of common multimedia applications, such as the MPEG-2/JPEG codecs. These applications exhibit high cache hit rates and, furthermore, require high data throughput. In the presented implementation, the whole cache block is transferred to or from the unit in a single cache access. When the hit rates are high, such a design can provide a high data throughput without increasing the number of cache ports, thus satisfying the needs of the streaming multimedia applications, which require high data bandwidth. Then we analysed the computations needed for the generation of address information. Although these computations turn out to be complex, we have shown that they can be performed in a pipelined fashion. We described a detailed design of a pipelined address generation unit that can compute the adress information for a new cache block every machine cycle. The proposed implementation is organized as a three-stage pipeline. We also identified the critical paths of the proposed stream address-generator.

In the following chapter we will present some experimental results for the proposed implementation. We will study the performance of the CSI-enhanced superscalar processors on the wide range of multimedia benchmarks, such as MPEG-2/JPEG codecs, image-processing and 3D graphics benchmarks, and compare it with the performance of the superscalar processors enhanced with traditional short-vector multimedia ISA extensions, such as Sun's *VIS* and Intel's *SSE*.

Chapter 5

Experimental Validation

In Chapters 2 and 3 we have presented the Complex Streamed Instruction Set Architecture (CSI), a multimedia-oriented ISA extension intended to be used on general-purpose superscalar processors. CSI has been designed so that a single CSI instruction can process arbitrary-length data streams located in memory, loading the source stream data, unpacking it from storage to computation format, performing the main operation, packing the results from computation to the storage format, and storing these values to the destination stream. This approach, as well as the extension of the instruction set with the special-purpose arithmetic instructions, has been taken in order to avoid the shortcomings of the existing short-vector multimedia ISA extensions, such as Sun's *VIS*, Intel's *MMX* and *SSE*, Motorola's *AltiVec*, and others. These shortcomings have been listed in Chapter 1. In Chapter 4 a design of a hardware unit that can execute CSI instruction has been presented.

In this chapter we validate the proposed architecture by studying the performance benefits it provides for superscalar processors enhanced with it on a number of important benchmarks representative of the particular application subdomains, such as image and video coding and decoding, 2-D image processing, and 3-D graphics. We study the following media benchmarks: JPEG and MPEG-2 encoders/decoders (referred to as *codecs*), which belong to the image and video coding/decoding application subdomain, *add8, blend8, scale8*, and some other image-processing kernels from Sun's VIS developer Kit (2-D image processing subdomain), and SPEC's *viewperf* (3-D graphics subdomain). We compare the performance of CSI-enhanced superscalar processors to that of processors extended with Sun's VIS or Intel's SSE media extensions. This chapter is structured as follows. In Section 5.1 we present our experimental methodology and tools. Section 5.2 provides the initial validation of the CSI set, studying the performance of CSI-enhanced and VIS-enhanced superscalar processors on MPEG-2 and JPEG codecs, as well as on a number of the most time-consuming kernels extracted from these applications. Section 5.3 presents a performance study of CSI- and VIS-enhanced processors on a number of image-processing kernels and evaluates the performance behavior with respect to the main memory bandwidth. The goal of the study is to investigate how well CSI- and VIS-enhanced processors perform when the cache hit rate is relatively low and the influence of main memory becomes significant. In Section 5.4 an industry-standard 3D performance evaluation benchmark, SPEC's *viewperf*, is studied. This

is the only floating-point intensive benchmark we consider. In this study, the performance of CSI-enhanced CPUs is compared with that of the CPUs capable of executing instructions from the Intel's *SSE* ISA extension proposed specifically to improve floating-point performance of 3D graphics applications. In Section 5.5, as well as in Section 5.4, we investigate how the performance of CSI-, VIS-, and SSE-enhanced CPUs scales when the number of parallel processing units is increased. These studies are motivated by the current trends in microprocessor technology, which allow a designer to fit dozens of functional units on the same chip. The challenge is to supply these numerous units with instructions and data to utilize them efficiently. Finally, in Section 5.6, some conclusions are presented.

5.1 Experimental Methodology and Tools

In order to evaluate the performance of the proposed ISA, we simulated superscalar processors without a multimedia ISA extension, processors with the VIS or SSE extension, and processors extended with CSI instructions. We used the `sim-outorder` simulator of the *SimpleScalar* toolset (release 3.0) [2, 5] to simulate a superscalar processor without any media extensions. This is a cycle-accurate execution-driven simulator of an out-of-order superscalar processor with a 5-stage pipeline based on the *Register Update Unit (RUU)* [62]. In order to evaluate the performance of the VIS/SSE-enhanced and CSI-enhanced processors, we modified the `sim-outorder` simulator so that it can simulate the VIS, SSE and CSI instructions. A corrected version of the SimpleScalar's memory model based on the SDRAM specifications [23] was used in all the studies, except for the ones reported in Section 5.3, where the standard SimpleScalar's memory model based on page-mode DRAM was employed. This was done because we needed to simulate varying bandwidth of the main memory, while the SDRAM specification fixes the width of the memory bus and the bus frequency, thus fixing the bandwidth. The simulator simulates programs compiled to the Portable ISA (PISA) architecture, which is derived from the MIPS-IV ISA [51]. Each PISA instruction has a 16-bit *annotate* field that can be modified post-compile with annotations to instructions in the assembly files. This interface can be used to synthesize new instructions without having to change the assembler. We used this mechanism to synthesize CSI and VIS and SSE instructions.

In general, the experiments were performed as follows. First, we profiled the benchmarks using the `sim-profile` tool provided in the *SimpleScalar* toolset (release 2.0) [5] and identified the most compute-intensive kernels. Consequently, the functions that contained a substantial amount of data-level parallelism and whose key computation could be replaced by VIS/SSE and CSI instructions were rewritten manually. We had to rewrite them by hand, because, to our knowledge, there is no publicly-available compiler that generates VIS and SSE code. Loops were unrolled so that the loop bodies could be replaced by a set of equivalent VIS/SSE instructions. The vendor codes, such as the ones provided in [14, 15, 30] were used when available. Finally, using the `sim-outorder` simulator, we simulated the execution of the modified benchmarks, some of the kernels of which were rewritten using CSI/VIS/SSE instructions, on the superscalar processors enhanced with the corresponding (CSI, VIS or SSE) execution units. The `sim-outorder` simulator provides rather detailed machine descriptions and allows the specification of a wide range of machine parameters,

such as the issue width, the size of the instruction window (RUU), the number of functional units and their latency/recovery, and the parameters of the cache/memory configuration, such as the types and the sizes of caches, the presence of TLB, and many others. In different experiments we studied different aspects of the performance and explored a wide range of processors, varying their key parameters. These parameters will be presented individually in each of the four experimental studies we performed.

5.2 Performance on MPEG-2/JPEG codecs

In this section we perform an initial experimental validation of the CSI architecture, studying the performance benefits it provides for the MPEG-2/JPEG codecs, and comparing it with the enhancements provided by the Sun's VIS extension. We studied four benchmarks from the MediaBench [40] test suite: `mpeg2enc` (MPEG-2 encoder), `mpeg2dec` (MPEG-2 decoder), `cjpeg` (JPEG encoder), and `djpeg` (JPEG decoder). These programs are representative of video and image processing applications. For the MPEG benchmarks, we used the *test* bitstream, which consists of three 128×128 frames. For the JPEG benchmarks, the *rose* input was used, which is a 227×149 pixel image. We profiled the benchmarks using the `sim-profile` tool provided in the SimpleScalar toolset and selected the most compute-intensive kernels: `Add_Block` (MPEG2 frame reconstruction), `Saturate` (saturation of 16-bit elements to 12-bit range in MPEG decoder), `dist1` (sum of absolute differences for motion estimation), `ycc_rgb_convert` and `rgb_ycc_convert` (color conversion between YCC and RGB color spaces in JPEG), and `h2v2_downsample` (2:1 horizontal and vertical downsampling of a color component in JPEG), and `idct` (inverse discrete cosine transform). We studied the kernel-level as well as application-level performance of 4-way and 2-way issue superscalar processors augmented with CSI or with VIS execution units. In the section to follow we describe the parameters of the simulated machines.

Modeled Processors

The base system is a 4-way superscalar processor with out-of-order issue and execution. The main processor parameters are listed in Table 5.1. VIS instructions operate on the floating-point register file. All VIS instructions have a latency of 1 cycle except the `pdist` (which computes the SAD) and the packed multiply instructions, both of which have a latency of 3 cycles. There are two VIS adders that perform partitioned add and subtract, merge, expand and logical operations, and two VIS multipliers that perform the partitioned multiplication, compare, pack and pixel distance operations. This is modeled after the UltraSPARC [65] with the following exceptions. In the UltraSPARC, the `alignaddr` instruction cannot be executed in parallel with other instructions [31] but this limitation is not present in the architecture we modeled. Furthermore, the UltraSPARC has only one 64-bit VIS multiplier. We assumed two because the width of the datapath of the stream unit is assumed to be 128 bit (i.e., the unit can process 16 bytes, 8 halfwords, or 4 words in parallel). The degrees of parallelism of the VIS-enhanced and the CSI-enhanced architectures are, therefore, comparable. The CSI execution unit is modeled after the L1-interfaced implementation that was described in Chapter 4, which means that it includes two extract units, two pack subunits, two main

Issue width	4-way
Reorder buffer size	16
Load-store queue size	8
Branch Prediction	
Bimodal predictor size	2K
Branch target buffer size	2K
Return-address stack size	8
Functional unit type and number	
Integer ALU	4
Integer MULT	1
Cache ports	2
Floating-point ALU	4
Floating-point MULT	1
VIS adder	2
VIS multiplier	2
CSI execution unit	1
datapath width	128 bits

FU latency/recovery (cycles)	
Integer ALU	1/1
Integer MUL	
multiply	3/1
divide	20/19
Cache port	1/1
FP ALU	2/2
FP MUL	
FP multiply	4/1
FP divide	12/12
sqrt	24/24
VIS adder	1/1
VIS multiplier	
multiply and pdist	3/1
other	1/1
CSI extract/insert	1/1
CSI pack/unpack	1/1
CSI medADD	1/1
CSI medMUL	3/1

Table 5.1: Processor configuration.

SIMD processing units *medADD* (used for addition-related operations) and *medMUL* (used for multiplication-related operations), one pack unit, and one insert unit. The latencies of the subunits of the CSI execution unit are taken to be equal to the latencies of the VIS units that perform the corresponding operations.

The parameters of the memory subsystem are listed in Table 5.2. Because the benchmarks used in this study have small instruction working sets and exhibit very high instruction cache hit rates. Therefore, and in order to reduce simulation time, a perfect instruction cache is assumed. All processors studied are configured with two L1 data cache ports. As described in Chapter 4, for the CSI execution unit, a whole cache line is fetched or stored in a single cache access. Furthermore, we note here that the CSI execution unit does not require extra ports to cache, the two available ports are shared between the load and store accesses generated by CSI instructions, and are also used for memory accesses performed by scalar load/store instructions. The ratio of CPU to memory clock frequency has been set to 4.0. Hence, the clock frequency of simulated CPU is $4.0 \cdot 100 = 400$ MHz.

Because one CSI instruction can replace two embedded loops, the requirements for the machine's fetch, decode and issue bandwidth will be greatly reduced. In order to evaluate this effect, we also simulated a 2-way superscalar processor in addition to a 4-way system. Finally, we remark that since VIS instructions are register-to-register and operate on the floating-point register file, they do not interfere with the existing processor pipeline. CSI instructions, however, are memory-to-memory and, therefore, require extra care: one must ensure that the execution of a CSI instruction does not overlap with scalar memory instructions. We, therefore, took the following conservative approach: when a CSI instruction is detected in the instruction stream, the decode pipeline is stalled. The processor waits until all

Instruction cache	ideal
Data caches	
L1 line size	32 bytes
L1 associativity	direct-mapped
L1 size	32 KB
L1 hit time	1 cycle
L2 line size	128 bytes
L2 associativity	2-way
L2 size	1 MB
L2 hit time	6 cycles
L2 replacement	LRU

type	SDRAM
row access time	20 ns
row activate time	20 ns
precharge time	20 ns
bus frequency	100 MHz
bus width	64 bits

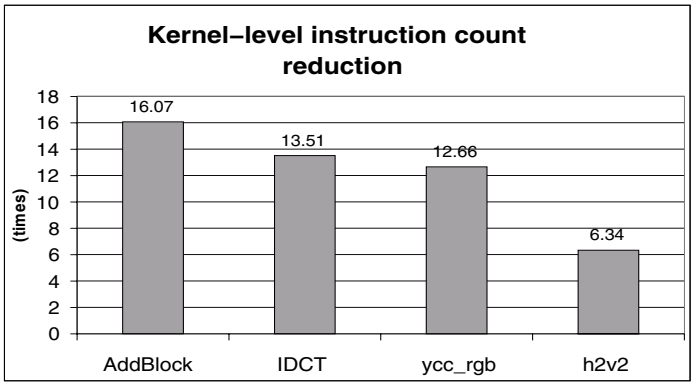
Table 5.2: Memory configuration.

active memory instructions commit, after which it issues the CSI instruction. Fetching and decoding resumes after the CSI instruction has finished.

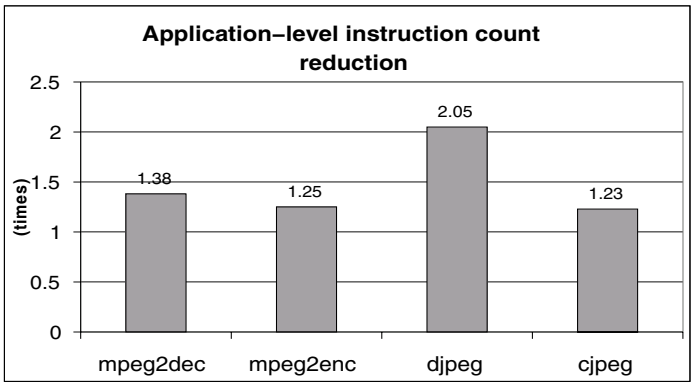
Experimental Results

In this section we study the performance behavior of the MPEG-2/JPEG coders and decoders on superscalar processors enhanced with CSI and on the same processors enhanced with VIS. Prior to presenting the performance results, we recall the main shortcoming of existing commercial media ISA extensions, which was stated in the introduction and motivated the development of CSI. It was observed there that the short-vector media ISA extensions, such as VIS, require a high number of instructions to be executed, thus making the decode/issue logic of a superscalar CPU a potential bottleneck. CSI attempts to avoid this problem by employing different hardware mechanisms in order to reduce the number of instructions which have to be executed. Figure 5.1(a) depicts, for a number of kernels which are extracted from the MPEG-2/JPEG codecs, the ratio of the dynamic instruction count exhibited by the 4-way issue superscalar VIS-enhanced CPU to the instruction count exhibited by the same processor enhanced with the CSI execution hardware. It can be observed from this figure that CSI, as expected, provides significant reductions in the instruction count, which range from factor 16.07 to 6.34, with the average reduction of a factor of 12.14. Figure 5.1(b) depicts the instruction count reductions attained by the 4-way issue superscalar CSI-enhanced CPU with respect to the same CPU enhanced with VIS on the level of complete applications. This figure shows that instruction count reductions achieved due to CSI on the kernel level translate in the significant reductions on the application level.

We now present the speedups attained by the CPUs enhanced with VIS or with CSI execution hardware. Speedups will be given with respect to the 2-way base system. We first present results for several kernels from our benchmarks. Next, we analyze how kernel-level speedup translates to application speedup. Figure 5.2 depicts the speedups attained for the seven kernels selected from the benchmarks. When the issue width is 2, the VIS-enhanced architecture achieves a speedup of 1.4 to 5.9 with an average of 3.1, whereas the CSI-enhanced architecture attains speedups ranging from 5.2 to 42.3 (21.2 on average). When the issue width is 4, the average speedup (w.r.t. to the 2-way system) of the VIS-enhanced



(a)



(b)

Figure 5.1: Instruction count reductions attained by CSI w.r.t. VIS.

architecture is 4.2 (1.9 to 7.2) and the average speedup of the CSI-enhanced architecture is 22.5 (5.6 to 42.2) CSI clearly outperforms VIS. Especially on the `Saturate` kernel the CSI-enhanced architecture performs much better than the architecture extended with VIS instructions. Whereas the VIS-enhanced architecture attains speedups of 1.43 (2-way issue) and 2.03 (4-way issue), the CSI-enhanced architecture attains speedups of 32.1 and 33.2, respectively. The reason is that in this kernel the 16-bit elements of a stream have to be clipped to a 12-bit range ($[-2048, 2047]$) and, simultaneously, the clipped values have to be accumulated. We observe that clipping (or, equivalently, saturating) of a signed 16-bit value to this range can be obtained by applying the following sequence of operations. First, the value is sign-extended to 32 bits and then is shifted left logically by 4 bits. Then, the shifted value is saturated to the range representable by signed a 16-bit value (i.e. $[-2^{15}, 2^{15} - 1]$) and, thereupon, the obtained 16-bit value is shifted arithmetically to the right by 4 bits. Such sequence of operations can be performed on all the elements of the stream by a single CSI instruction `csi_add_reg dest, src, reg`. To achieve this, the register operand `reg` should contain zero, and the parameters of the source and the destination streams, such as *ElSize, ProcessSize, Sign, Saturate, ShiftAmount,* and *ShiftDirection* should be set in a certain way so that the required sign-extension, shifting and saturation will be performed

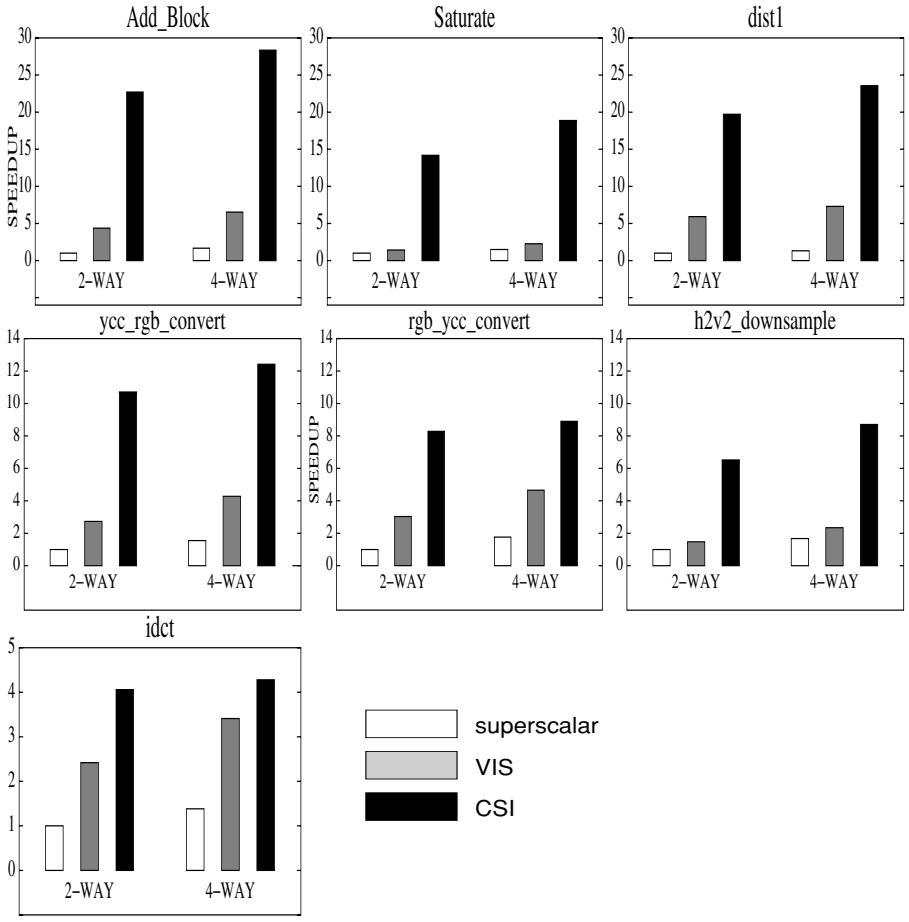


Figure 5.2: Speedups on kernel level.

during processing. After the stream of clipped values is produced, it can be accumulated by a single CSI instruction `csi_acc`. Hence, the whole `Saturate` kernel requires just two CSI instructions. On the other hand, the VIS implementation of this kernel requires the stream to be sectioned into chunks that fit into 64-bit VIS registers. Therefore, the kernel is implemented as a loop which processes a single stream section at each iteration. Furthermore, each iteration requires more than 20 VIS instructions such as loads, comparisons, conditional stores, and additions, in order to implement saturation and accumulation. Therefore, the VIS implementation exhibits high instruction counts. The large number of instructions that have to be executed limits the performance of VIS.

It can be observed that the smallest performance improvement of the CSI-enhanced architecture over the VIS-enhanced architecture occurs for the `idct` kernel, which performs the two-dimensional *Inverse Discrete Cosine Transformation (IDCT)*. The reason is that the VIS version of this kernel is based on a highly optimized DSP algorithm proposed in [7]. This

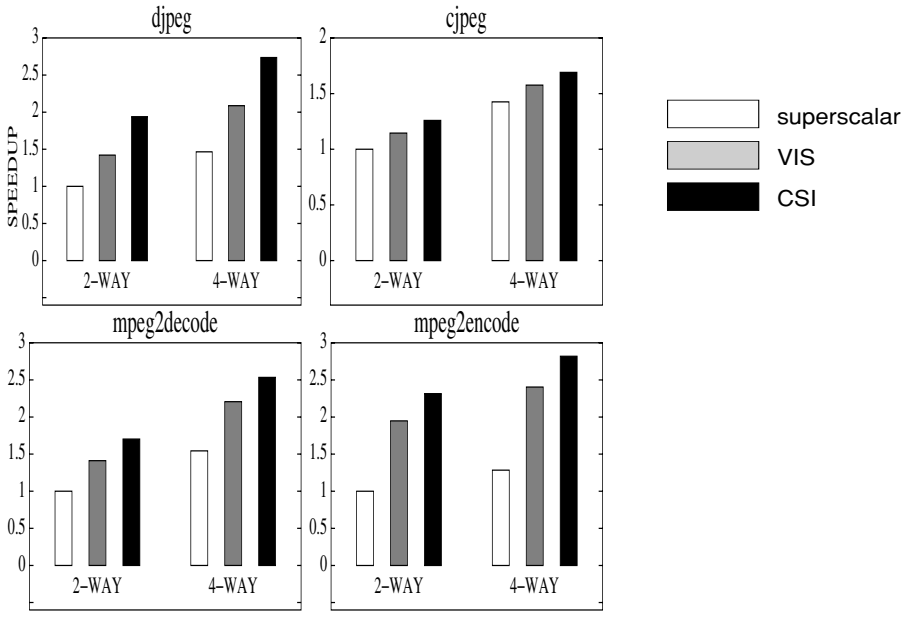


Figure 5.3: Speedups on application level.

algorithm can be utilized by the special-purpose CSI instruction `csi_idct`, but in such a case the CSI execution unit would require specialized hardware to perform these computations. Since the VIS-enhanced processors we simulate do not have any specialized hardware, equipping the CSI execution unit with such hardware will make the comparisons between the CSI- and the VIS-enhanced processors unfair. Because of this, we assume that the CSI execution unit has no special-purpose hardware and, hence, cannot execute the `csi_idct` directly. This instruction has to be emulated by means of simple arithmetic CSI instructions. The DSP algorithm, however, does not operate on long vectors and, hence, cannot be efficiently implemented using simple arithmetic CSI instructions. Therefore, the CSI version of the `idct` kernel is based on the standard definition of the IDCT as two matrix multiplications. Because of this, the CSI version of `idct` executes many more operations than the VIS version, but nevertheless a speedup is obtained.

The results for complete applications are depicted in Figure 5.3. For a 2-way issue machine, the VIS-enhanced architecture achieves speedups of 1.42 (on the `djpeg` benchmark), 1.17 (`cjpeg`), 1.40 (`mpeg2dec`) and 1.93 (`mpeg2enc`), whereas the CSI-enhanced architecture attains speedups of 1.94, 1.28, 1.70 and 2.28, respectively. For a 4-way issue machine, the respective speedups are 2.08, 1.59, 2.13 and 2.37 for the VIS-enhanced architecture, and 2.75, 1.74, 2.48 and 2.77 for the CSI-enhanced architecture. Of course, due to Amdahl's Law, the speedups for complete programs are less impressive than those for kernels. Nevertheless, when the issue width is two, the CSI-enhanced architecture yields a performance gain over VIS of 20% on average (range of 8% to 36%), and when the issue width is four, the average speedup of CSI over VIS is 18% (range of 8% to 32%). We remark that when the issue rate

is 2, the CSI-enhanced architecture attains higher speedups with respect to the VIS-enhanced architecture than when the base system is a 4-way processor. This means that the performance of the stream unit is rather insensitive to the processor issue width. This makes the CSI architecture highly suitable for embedded systems, where high issue rates and out-of-order issue and execution are too expensive. The same observation has been made in [11] for the MOM ISA extension. Finally, we make a remark concerning the memory-to-memory organiation of the CSI architecture. Early vector computers such as *TI ASC*, *Star-100*, and *Cyber-205* [28] were also memory-to-memory and could process vectors of arbitrary length. However, these machines suffered from long startup times, which was mainly due to overhead instructions needed for setting up the vector parameters and due to long memory latency. The implementation of a CSI-enhanced processor we study does not suffer from these problems for the following reasons. Since CSI is implemented next to a superscalar core, the overhead needed for setting the parameters (i.e., initializing the registers in SCR-sets) is small. Second, because the L1 cache hit rates are high and the CSI execution unit is interfaced to this cache, the memory latency experienced by this unit is short (mostly, it is equal to the L1 cache access latency, which is one cycle).

Conclusions

We have evaluated the performance provided by the proposed Complex Streamed Instruction (CSI) set on the MPEG-2/JPEG codecs, which are typical representatives of the image/video coding/decoding application domain. On a number of important kernels, we observed speedups ranging from 2.1 to 22.4 relative to an architecture extended with VIS instructions. These local improvements resulted in application speedups of up to 36%.

5.3 Performance on Image-Processing Kernels

In the previous section we studied the relative performance of the CSI and VIS media extensions on the image and video codecs, `cjpeg` (JPEG encoder), `djpeg` (JPEG decoder), `mpeg2encode` (MPEG-2 encoder), and `mpeg2decode` (MPEG-2 encoder). These applications exhibited very high hit rates for a processor equipped with a medium-sized 32 Kbyte L1 data cache. In this section we evaluate the performance on another multimedia application domain, 2-D image processing. Contrary to image and video codecs, image-processing kernels are characterized by little data reuse. Since we study the implementation of the CSI execution unit which is interfaced to the L1 cache, it is important to evaluate if such an implementation can provide significant speedups when the cache performance is relatively poor. Since the cache performance is relatively poor, and the kernels have really streaming nature, often accessing data just once, the memory system becomes an important performance factor. In particular, the memory bandwidth is likely to become a decisive factor for the overall system performance. Therefore, we evaluate the performance of the VIS- and CSI-enhanced processors with respect to the main memory bandwidth. Such a study is also motivated by the current trends in the design of the memory DRAM chips. While many contemporary PCs have a memory bandwidth of 0.8 GB/s using the *Synchronous DRAM (SDRAM)* devices with the clock rate of 100 Mhz and a 64-bit clocked at the same frequency, *Double Data*

Rate-Synchronous DRAM (DDR SDRAM) [20] and *Direct RAMBUS* [29] provide 1.6 GB/s of bandwidth and are likely to provide even higher bandwidths in the future.

To evaluate the performance of CSI on image processing kernels, we simulated their execution on three different processors: a 4-way superscalar processor without media ISA extensions, the same processor extended with VIS, and the same processor augmented with CSI. The following benchmarks taken from the VIS Software Development Kit (VSDK) were selected: `add8` (adding two images using mean of corresponding pixels), `blend8` (alpha-blending of two images), `scale8` (linear scaling) and `convolve3x3` (convolution with a 3x3 kernel). These kernels usually serve as building blocks for more complex image transformations such as image compositing, blurring, sharpening, etc. As input, we used 332x345 images in Sun rasterfile format with 3 color components. Three different executables of each kernel were created: a baseline PISA version, a VIS version, and a CSI version. The baseline PISA versions were obtained by compiling the C code taken from the VSDK using the `gcc` compiler with option `-O4`. For VIS and CSI we manually rewrote the assembly files, using for VIS a 1-1 translation of the VIS codes provided in the VSDK.

Modeled Processors

The base system is a 4-way superscalar processor with out-of-order issue and execution. Its main parameters are the same as of the baseline processor studied in Section 5.2 and are listed in Table 5.5. The ratio of CPU clock frequency to the bus/memory clock frequency was set to 5, which results in the CPU clock frequency of 500 Mhz, assuming that the bus and the memory are clocked, according to the PC SDRAM specification, at 100 Mhz. The VIS-enhanced processor is modeled after the UltraSPARC [65], except that we assumed two 64-bit VIS multipliers whereas the UltraSPARC has only one. We chose this configuration because CSI instructions are assumed to process two 128-bit packed data types in parallel. Thus, both CSI-enhanced and VIS-enhanced processors have the SIMD execution hardware that can operate on 16 bytes of data in parallel. In fact, the hardware of the VIS-enhanced processor is somewhat more aggressive, because it is equipped with 2 VIS adders and 2 VIS multipliers. Since the adders and the multipliers can be used independently of each other, then if at certain cycle there are 4 instructions, which are ready to be executed, and 2 of them require a VIS adder for execution, while 2 other require a VIS multiplier, all four of them can be issued at that cycle and thus 32 bytes of data will be processed in parallel. Such configuration of the VIS execution units was chosen so that VIS-enhanced processor can match the throughput of the CSI unit independently of the instruction mix. Configuring the VIS-enhanced processor with only 1 VIS adder and 1 VIS multiplier would allow such processor to match the CSI-enhanced CPU only for the instruction mixes for which the ratio of addition-related to multiplication-related VIS instructions is 1:1.

The cache and memory parameters of the modeled processor are summarized in Table 5.3. The memory latencies are expressed in CPU clock cycles. Converted to absolute time, they correspond to access latencies of 20-60ns, which is close to those of contemporary DRAM chips. The front-side memory bus (between the L2 cache and the memory controller) is clocked at 100 MHz as in current PCs [16]. In order to study the effect of memory bandwidth on the performance of the modeled processors, we vary the bus width from 8 bytes (current PC standard) to 16 and 32 bytes, which corresponds to bandwidths ranging from 0.8

Instruction cache	ideal
Data caches	
L1 line size	32 bytes
L1 associativity	direct-mapped
L1 size	32 KB
L1 hit time	1 cycle
L2 line size	128 bytes
L2 associativity	2-way
L2 size	128 KB
L2 replacement	LRU
L2 hit time	6 cycles

Main memory	
type	page-mode
page size	4 KB
first page access	30 cycles
next page access	10 cycle
bus clock frequency	100 MHz
bus width	8/16/32 bytes

Table 5.3: Memory system parameters.

to 3.2 GB/sec. Contemporary PCs have a memory bandwidth of 0.8 GB/s using a 64-bit wide bus and *DDR SDRAM* [20] and *Direct RAMBUS* [29] already provide 1.6 GB/s of bandwidth.

Experimental Results

We now present and discuss the performance improvements attained by the CSI- and VIS-enhanced CPUs. We also analyze the different components of the execution time in order to identify the bottlenecks.

Figure 5.4 depicts the execution times of the baseline, VIS-enhanced and CSI-enhanced processor on each benchmark. The bars labeled '8b', '16b', '32b' depict the execution time when the bus width is 8 bytes, 16 bytes, or 32 bytes, respectively. The execution time is normalized with respect to the time taken by the baseline superscalar system with an 8-byte wide bus. Figure 5.4 shows that the baseline system is compute-bound and hardly benefits from higher memory bandwidths. This is evidenced by the fact that the `add8` kernel (which requires a small amount of computation per pixel) scales better than `convolve3x3` (which is computationally more expensive). The VIS ISA extension significantly improves the performance across all benchmarks. Nevertheless, the processor core becomes the bottleneck when the bandwidth is increased from 1.6 GB/s to 3.2 GB/s, since the performance improvements are smaller than when going from 0.8 GB/s to 1.6 GB/s. The CSI-enhanced processor provides additional performance gains ranging from a factor of 1.2 to 3.9. This can be seen more clearly from Figure 5.5, which depicts the speedups of CSI over VIS. It should be observed that the speedup increases with the bus width.

In order to further identify the performance bottlenecks, we analyze the different components of the execution time. Every cycle is classified either as *non-stall*, *cpu stall* or *memory stall*. A cycle is classified as non-stall if the number of instructions retired that cycle is equal to the issue rate (4). If not, we consider the first instruction that could not retire. If it is a memory access instruction, it is classified as memory stall. Otherwise, it is classified as cpu stall. Since instructions retire in order, the first instruction that cannot retire prevents retirement of all the instructions that follow it. We used the described classification instead of the *busy time* metric used in [52] because it clarifies why less than the maximum number of instructions are retired. For the CSI architecture, a cycle is classified as non-stall if a SIMD

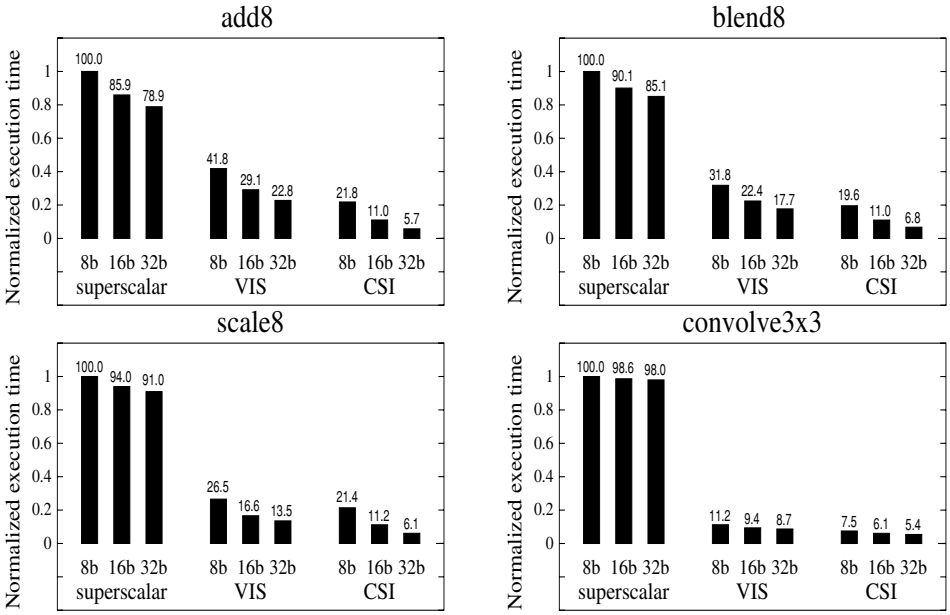


Figure 5.4: Normalized execution time of the image processing kernels.

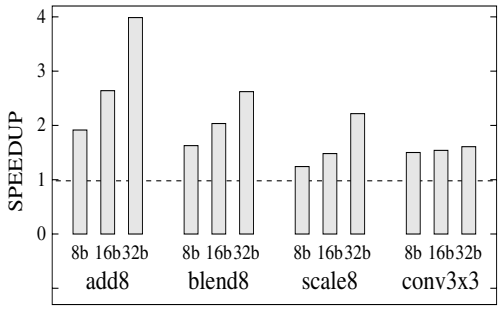


Figure 5.5: Speedup of CSI over VIS.

operation was initiated at that cycle. Otherwise, it is classified as memory stall, meaning that data has not yet been delivered by the memory. The CSI-enhanced processor does not experience cpu stall cycles because there are no dependencies between stream elements and there are enough resources to initiate a SIMD operation every cycle. Figure 5.6 breaks down execution times in non-stall, cpu-stall and memory stall cycles. It shows that neither the superscalar nor the VIS-enhanced processor can fully utilize the increased memory bandwidth. The superscalar system gets already saturated when the bus width is 8 bytes, since the CPU component of the execution time (non-stall plus cpu-stall) increases only marginally. The VIS-enhanced system utilizes the available bandwidth much better, but the growth of its CPU component is mostly due to the cpu-stall component. Increasing the memory bandwidth of

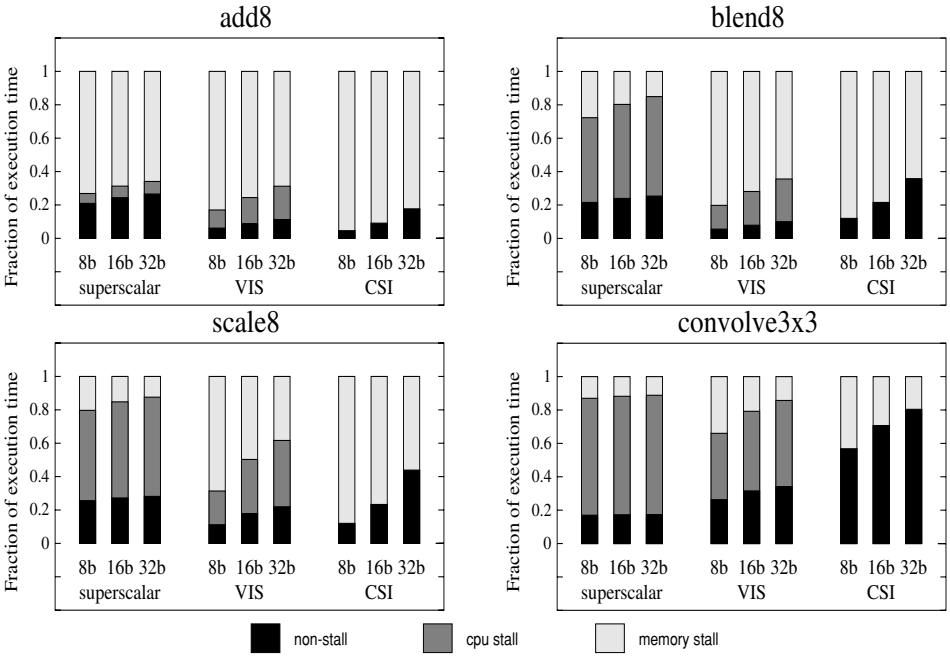


Figure 5.6: Execution time partitioned in non-stall, cpu stall and memory stall cycles.

the CSI-enhanced system, on the other hand, leads to a significant reduction in memory stall cycles and a corresponding increase in non-stall cycles.

We remark that some execution may occur during cpu stall cycles. The fact that a given cycle is cpu stall means that the oldest instruction in the instruction window (i.e., in the RUU) cannot retire because its result is not ready yet. There can be multiple reasons why an instruction has not finish, the main being the following. Either the instruction was issued and started execution too late because no functional unit was available, or the instruction was decoded and placed in the RUU too late because there were no free entries. These observations suggest that the performance of conventional and VIS-enhanced superscalar processors may be improved by increasing the number of functional units and/or by increasing the size of the RUU. We conducted experiments using these techniques. Adding two VIS adders and two VIS multipliers produced no effect. Doubling the sizes of the RUU and the load/store queue indeed improved performance for the VIS-enhanced system by a factor of up to 1.20. However, this improvement is still significantly less than the one achieved by CSI (cf. Figure 5.5). This shows that the performance of conventional and VIS-enhanced superscalar processors is limited by the execution engine and cannot be improved much resource duplication techniques. Hence, it seems that in order to improve performance of the CPU enhanced with a short-vector media extension like VIS, one has to pack more parallel operations in a single instruction. This, however, requires a change of the architecture of the media extension (for example, the size of the multimedia registers should be increased) and, therefore, will result in binary incompatibility.

Conclusions

We have extended the work presented in Section 5.2 by studying a new application domain with different memory system behavior and by evaluating the effect of memory bandwidth. We have evaluated the performance of the CSI multimedia ISA extension on a set of image processing kernels characterized by little data reuse. We have also studied the scalability of the proposed architecture with respect to the memory bandwidth. The results show the following:

- On the high-bandwidth system (3.2 GB/s), the CSI extension improves performance by a factor of 1.6 to 3.9 (2.6 on average) compared to the VIS-enhanced processor.
- On the low-bandwidth system (0.8 GB/s), the speedups range from 1.2 to 1.9 with an average of 1.56.
- Neither the superscalar nor the VIS-enhanced processors are able to fully utilize the high memory bandwidth provided by current high-performance and future generations DRAM architectures. We identified the CPU core as the bottleneck and observed that increasing the size of the reorder buffer and the number of functional units does not alleviate the problem.

5.4 Performance on 3D graphics

In this section we study the performance benefits provided by the CSI extension for an application that belongs to another important domain of multimedia applications: three-dimensional (3-D) graphics processing. This domain includes applications such as CAD tools, 3D games, virtual reality, etc. The high computational demands and real-time constraints of these applications has long kept them in the realm of high-end workstations, where specialized hardware was employed to provide the required performance. Fast progress in processor technology and a great demand for such applications are now bringing them to consumer desktop systems. In order to reduce cost, much processing is shifted from specialized hardware to general-purpose CPUs. While the image/video coding/decoding and 2D image processing applications, which we have studied earlier in this chapter, are usually operating on 8- and 16-bit integer values, the 3-D graphics applications are floating-point intensive, operating usually on 32-bit single-precision floating-point (FP) numbers. To achieve higher performance, several processor vendors have provided short-vector SIMD extensions that contain instructions capable of performing several 32-bit floating point operations in parallel. AMD's *3DNow!* [12] extension operates on the usual 64-bit double precision floating-point registers, treating their contents as two independent 32-bit FP numbers and includes so-called *paired-single* instructions that can operate on two 32-bit FP numbers in parallel. Paired-single SIMD instructions have been quantitatively evaluated by Yang et al. [69] and positive results have been presented. However, the general deficiencies of short-vector SIMD extensions listed in Chapter 1 apply to these extensions as well, with the exception of the overhead associated with packing/unpacking since these operations are not needed for floating-point numbers. The work presented here is to a large extend motivated by this observation. We perform experiments that study the performance provided by CSI and compare it with that

provided by a short-vector floating-point ISA extension. As our reference extension, we do not employ the AMD's *3DNow!*, which has paired-single FP instructions. Instead, we us more aggressive Intel's *Internet Streaming SIMD Extension (SSE)* which increases the parallelism that can be exploited by a single instruction by extending the register state with a set of 128-bit register that can contain 4 32-bit FP numbers. It includes instructions that can perform 4 32-bit floating-point operations in parallel. We evaluate the 3D graphics performance of superscalar CPUs extended with CSI by comparing them with the performance of CPUs enhanced with the SSE extension. For our study we use the industry-standard 3-D performance evaluation benchmark, SPEC's *viewperf* [63].

Three-dimensional graphics processing. Before presenting the results of experiments, we provide the reader with a short description of the computations performed during 3-D graphics processing. A 3D application, such as a CAD tool, 3D game, or virtual reality application, creates a description of a 3D scene and stores it in a database. This information is then passed to the graphics processing system. In general, 3D graphics processing is a 3-stage pipeline consisting of the following steps [21]: (1) database traversal, (2) geometry calculation, and (3) rasterization. The first stage is responsible for extracting the information necessary for displaying a 3D scene created by an application. 3D objects are usually described by a set of polygons which model the object's surface. The information needed for display includes the coordinates of the polygon vertices, the type of primitive which has to be constructed from these vertices (e.g., line, polygon, triangle), material of the object, lighting model, camera position, and many others. This information is passed to the second stage, geometry computation, which calculates the color and coordinates of each vertex in the *screen* coordinate system. This stage is the most floating-point intensive one and is the primary target for acceleration using SSE and CSI instructions. The final stage, rasterization, takes the color and screen coordinates of each vertex as input, calculates the pixel values for each point on the display, and stores them in the frame buffer from where they are taken and displayed on the monitor screen.

In contemporary desktop systems, the CPU performs the database traversal and the geometry processing stages, and special-purpose hardware (the so-called graphics card) is employed for rasterization. The performance of graphics cards has increased significantly in the last few years due to advances in technology as well as by employing a number of architectural techniques, such as caching, memory tiling and interleaving, and others. Detailed descriptions of some of them can be found in [42]. According to some reports [18], the CPU has become the bottleneck for 3D graphics, because CPUs "only" double in speed every 18 months, while the performance of graphics processors increases by a factor of eight in the same period of time. Thererefore, a significant performance improvement in the geometry processing task is required. Although some contemporary graphics cards can perform parts of geometry computations, such cards are rather expensive. In the study presented later in this section we assume that these computations are performed by CPU, since such an organization will be a cheaper and more flexible solution, provided it can achieve the desired performance. In our experiments we will attempt to accelerate geometry computations by employing SSE and CSI instructions. To understand the ways this problem can be attacked, the geometry stage is described in more detail.

The computations carried out in the geometry stage are also organized in a pipeline. The

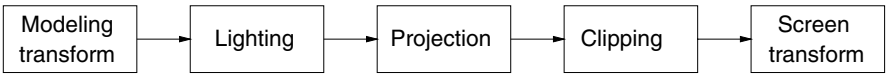


Figure 5.7: 3D geometry pipeline.

data representing a 3D scene flows through a number of kernels, as depicted in Figure 5.7. The 3D geometry pipeline uses several coordinate systems to describe the objects, namely *modeling coordinates*, *world coordinates*, *view coordinates* and *screen coordinates*. The modeling transform stage uses 3- and 4-dimensional matrix-vector multiplications to transform modeling coordinates to world coordinates. The lighting stage computes the color of each vertex using the properties of the object's material and parameters describing the lights present in the scene. The projection stage transforms vertex coordinates from the world to the view coordinate system using 4D matrix-vector multiplication. During the clipping stage the objects are clipped to the viewable domain to avoid unnecessary rendering of polygons positioned outside the area. The screen transformation stage converts 4D view coordinates (x, y, z, w) to 3D screen coordinates. It consists of two consecutive transformations. The first one divides the x, y, and z component by the w component. The second one multiplies the obtained vector with a 3×3 matrix and adds a displacement vector. All stages except lighting are mandatory. For example, applications using only wireframe objects will bypass the lighting stage. However, the applications primarily targeted by floating-point SIMD extensions as well as by CSI, such as 3D games and virtual reality, use lighting extensively.

Experimentation setup. First of all, we had to decide to which level of memory hierarchy the CSI execution unit should be interfaced. For this purpose, we evaluated the behavior of the 3D geometry pipeline on the SPEC *viewperf* benchmark and observed that very high data cache hit rates are exhibited. Our results show that, for a 4-way superscalar processor with a 32 Kb 4-way set-associative L1 data cache with a line size of 64 bytes, the geometry pipeline exhibits hit rates of 98.5% on average. We remark here, that such high hit rates can be explained by the fact that, according to [1], about 90 floating-point operations are performed by the geometry pipeline in order to process a single vertex. This results in high temporal locality. The benchmark also exhibits spatial locality, since vertex data is stored consecutively. Because of this good cache behavior, we chose to interface the MIU to the L1 data cache, as in the studies reported earlier in this chapter.

In order to evaluate the performance of the proposed ISA extension, we simulated a superscalar processor without a multimedia ISA extension, a processor extended with SSE instructions and a processor extended with CSI instructions. We studied the SPEC's *viewperf* benchmark using the *DX-06* dataset. This benchmark is distributed with five datasets with widely varying characteristics, most importantly of which is the number of vertices per primitive, because it determines the length of the streams. It varies from 3.4 in *Awadvs* to around 400 in *CDRS*. The *DX-06* dataset lies between these extreme cases and has an average of 95 vertices per primitive, as well as acceptable simulation times. As usual, we used the `sim-outorder` simulator, modified to simulate the execution of CSI and SSE instructions. A corrected version of the SimpleScalar memory model based on SDRAM specifications given in [23] has been used. The geometry computations in *viewperf* are performed via API calls to the *Mesa* library, which is the public-domain implementation of the *OpenGL* graphics library [22]. To simulate the benchmarks on a standard superscalar

processor and on superscalar processors with SSE and CSI extensions, we created three different versions of *Mesa*: *libMesaGL_scalar.a*, *libMesaGL_SSE.a* and *libMesaGL_CSI.a*. We obtained three versions of the *viewperf* executable by linking it with these libraries. The scalar library has been obtained by compiling the *Mesa* sources using *gcc* with the `-O2 -funroll-loops` optimization flags. SSE and CSI versions of the library have been created as follows. First, the source files containing the kernels `xform_points_4fv` and `gl_color_shade_vertices_fast` have compiled to assembly using the `-O2` optimization flag. After that, these kernels have been manually rewritten using SSE or CSI instructions and the object files have been created. Finally, the remaining *Mesa* source files have been compiled to objects and the required libraries have been created. We decided to modify these kernels because, according to our experiments, on a 4-way superscalar machine, they account for 42% and 16% of the total geometry execution time, respectively. In all the libraries the call to the `render_vb` function was removed because it implements the rasterization stage of the graphics pipeline and this task is usually handled by the graphics card, not by the CPU.

Coding the Geometry Kernels Using SSE and CSI. The `xform_points_4fv` function implements 4x4 floating-point matrix-vector multiplication of a sequence of 4D vectors with a fixed 4x4 matrix. For brevity, this function will be referred to as the `xform` kernel. This kernel is a part of the *modeling* and *projection* stages of the pipeline. The parameters to this kernel are the number n of vectors to be transformed, pointers M to the transformation matrix, P to the input array of 4D vectors which have to be transformed and Q to the output array of transformed vectors. For the *DX-06* dataset, arrays P and Q contain $n = 96$ vectors on average. CSI implements this kernel by means of a loop with 4 iterations. Each iteration i calculates $Q[j][i]$, for $1 \leq j \leq n$, the stream formed by ith coordinates of transformed vectors, using just two CSI instructions, `csi_mul` and `csi_acc_section`. The whole kernel is, therefore, implemented with 8 CSI instructions. For the SSE implementation, the vendor code provided in [15] was used.

The `gl_color_shade_vertices_fast` kernel carries out the lighting stage of the pipeline. For brevity, this kernel will be referred to as the **light** kernel. It is a complex kernel consisting of about 80 lines of C code. A simplified pseudo code description is given in Figure 5.8(a). Before implementing this kernel with CSI instructions, some transformations were performed on the source code level, as depicted in Figure 5.8(b). The outermost loop was split into three loops. The first loop initializes the RGB color components of each vertex to the ambient light constants. The second loop calculates the contributions of the light sources to the vertex colors. Loop interchange was performed on this loop, because the number of lights is likely to be small (typically 1 or 2) which results in short streams and poor CSI performance. The calculations of the light source contribution contain several *if-then-else* statements and have been implemented using conditional CSI instructions. The third loop converts the RGBA color components of each vertex from floating-point to integer format. It has been implemented using the `csi_fp_cvt.w` instruction. For SSE we did not perform similar source code transformations because it would require spilling intermediate results from SSE registers to memory, which would most likely decrease performance. Instead, we based the SSE implementation of this kernel on [14] and used corresponding code samples provided by Intel.

```
GLfloat R,G,B,A;
/* Alpha is constant for all vertices */
A= ..;
for (j=0; j<n; j++){ /* each vertex*/
   /*set color to ambient light */
   R = ctx->Light.BaseColor[0];
   G= ..; B=..;
   for (each light source){
      calculate light's R,G,B contribution;
      R+=contribution_of_this_light_R;
      G+=..;B+=..;
   }
   /*convert FP to integer for R,G,B,A*/
   frontcolor[0][j]=(GLfixed)(R);
   frontcolor[1][j]=(GLfixed)(G);
   frontcolor[2][j]=(GLfixed)(B);
   frontcolor[3][j]=(GLfixed)(A);
}
```

(a) Original code.

```
GLfloat tmp_R[],tmp_G[],tmp_B[],A;
A= ..;
for(j=0; j<n; j++){ /* each vertex*/
   /*set to ambient light constants*/
   tmp_R[j] = ctx->Light.BaseColor[0];
   tmp_G[j]= ..; tmp_B[j]=..;
}
for(each light source){
   for(j=0; j<n; j++){ /* each vertex*/
      calculate R,G,B contribution of light;
       tmp_R[j]+=contribution_of_this_light_R;
       tmp_G[j]+=..;tmp_B[j}+=..;
   }
}
/* convert FP to integer for R,G,B,A */
for(j=0; j<n; j++){ /* each vertex*/
   frontcolor[0][j]=(GLfixed)(tmp_R[j]);
   frontcolor[1][j]=(GLfixed)(tmp_G[j]);
   frontcolor[2][j]=(GLfixed)(tmp_B[j]);
   frontcolor[3][j]=(GLfixed)(A);
}
```

(b) Restructured code.

Figure 5.8: Original and restructured code of the lighting kernel.

Table 5.4: Processor configuration.

Clock rate	666 MHz	
Issue width	4	
Register update unit size	32	
Load-store queue size	16	
Branch Prediction		
Bimodal predictor size	2K	
Branch target buffer size	2K	
Return-address stack size	8	
Functional unit types	*number*	*latency/recovery (cycles)*
Integer ALU	4	1/1
Integer MULT	4	
multiply		3/1
divide		20/19
Cache ports	2	1/1
Floating-point ALU	4	2/1
Floating-point MULT	4	
FP multiply		4/1
FP divide		12/12
sqrt		24/24

Modeled Processors

The base system is a conventional 4-way superscalar processor with out-of-order issue and execution based on the Register Update Unit (RUU). The processor parameters are listed in Table 5.4. The *viewperf* benchmark exhibits a very high instruction cache hit rates. Therefore, and in order to reduce simulation time, a perfect instruction cache is assumed. The processor is configured with the 32 KB 4-way set-associative L1 data cache with a cache line size of 64 bytes and with a 1 MB 2-way set-associative L2 data cache having 128 byte lines. The access times of the caches are 1 and 6 cycles, respectively. The main memory is a standard 166 MHz SDRAM memory with row access, row activate, and precharge times of 2

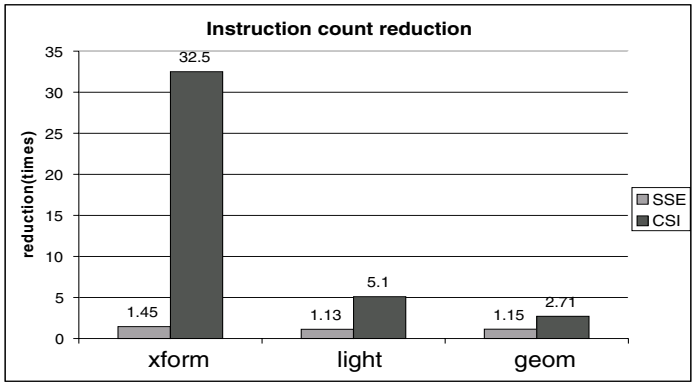


Figure 5.9: Reduction in dynamic instruction counts provided by CSI and SSE.

(memory) cycles. The memory bus is 64 bytes wide and is clocked at 166 MHz as well. The ratio of CPU to memory clock frequencies was set to four, resulting in a CPU clock rate of 666 MHz. The conventional 4-way superscalar processor was configured with 4 single-precision FP adders and 4 single-precision FP multipliers. Both the SSE-enhanced and CSI-enhanced processors have one SIMD FP execution unit capable of performing 4 FP numbers in parallel. The latencies of SSE instructions are equal to the latencies of the corresponding scalar instructions. For CSI instructions, latencies of the main SIMD computations they perform (such as add, multiply, etc.) are taken to be the same as the latencies of the corresponding SSE/scalar instructions. The latency of instruction itself is usually much longer, and is not fixed, since the data streams it processes can be of arbitrary length.

We also remark that CSI instructions are executed in-order. Furthermore, CSI instructions are only issued when all preceding loads and stores have completed. This is necessary to ensure semantic correctness of the executed program. It is important to realize that the CSI execution unit performs parallel operations on several floating-point numbers, but does not execute several CSI instructions in parallel.

Experimental Results

In this section we present the speedups attained by the SSE and CSI multimedia ISA extensions over the conventional superscalar processor, identify the bottlenecks of the SSE-enhanced processor, and study several other important parameters.

Instruction Traffic Reduction. We remark that in previous research [9] the necessity to execute a large number of instructions was identified as a factor that limited the performance of integer SIMD extensions. Similar observations can be applied to floating-point SIMD extensions, such as SSE. Prior to presenting the relative performance of the CSI- and SSE-enhanced processors, we study the dynamic instruction counts exhibited by such machines. Figure 5.9 depicts the dynamic instruction counts of the SSE and CSI versions of the kernels with respect to the number of instructions exhibited by the scalar versions of the kernels. It

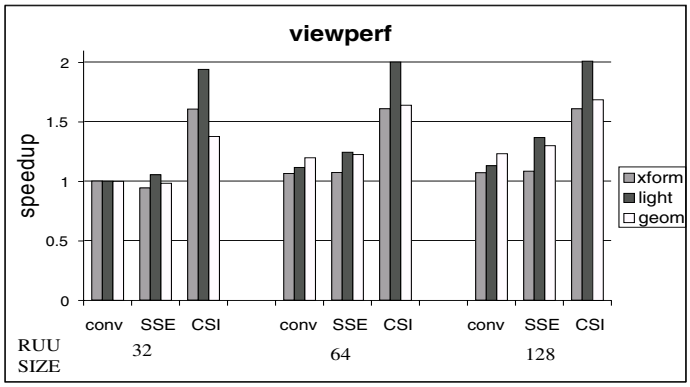


Figure 5.10: Speedups w.r.t. the baseline processor for varying RUU sizes.

can be seen that on the **xform** kernel (which can be fully streamed) CSI reduces the dynamic instruction count by a factor of 30, while SSE decreases the instruction traffic only by factor of 1.45. Furthermore, CSI reduces the dynamic instruction count of the **light** kernel by a factor of 5.1, while SSE reduces it by a factor of 1.13. The relatively smaller reductions exhibited by this kernel are due to the fact that some parts of the kernel use pointer structures and indirect addressing and, therefore, cannot be implemented using CSI and SSE instructions. The reductions on the kernel-level translate to a reduction of 2.71 for CSI and 1.15 for SSE for the whole geometry pipeline.

Relative Performance of the Conventional, SSE- and CSI-enhanced processors. We now study the relative performance of all three processor designs and investigate the influence of the instruction window size on their performance by varying the size of the RUU and the LSQ (load-store queue). Recall that all machines considered are able to execute 4 FP operations in parallel. Figure 5.10 depicts the speedups of the 4-way conventional (referred as 'conv' in the picture), SSE- and CSI-enhanced processors for varying RUU sizes with respect to the baseline system: the 4-way superscalar processor equipped with an RUU with 32 entries. CSI clearly outperforms SSE and conventional processors, especially for smaller RUU sizes. The reason for this is the following. The conventional superscalar processor exploits the parallelism present in the 3D geometry pipeline in the form of instruction-level parallelism (ILP). The SSE-enhanced processor exploits ILP as well as data-level parallelism (DLP). When the size of the RUU decreases, the ability of these processors to utilize the ILP also decreases, and so does the performance. For the CSI architecture, the parallelism present in computations is exploited in a single instruction and, therefore, insensitive to the RUU size. The ability of CSI to provide high performance with the smaller instruction window is rather beneficial, since this is an expensive resource as its cost grows quadratically with respect to its size.

Figure 5.10 also shows that the performance of the SSE-enhanced processor is close to the performance of the conventional processor. We remark, however, that for a conventional

4-way issue CPU the number of floating-point operations, which it can execute in parallel, cannot be more than four, while for the SSE-enhanced processor with equal issue width this number can be increased up to 16. Since we are interested in the scalability of the performance with respect to number of the floating-point execution units, in the remainder of the paper we no longer include the results for the conventional superscalar processor.

Performance Scalability and Bottlenecks. Next we study the performance scalability of the SSE- and CSI-enhanced machines with respect to the number of floating-point units, the issue width, and the number of cache ports in order to identify the bottlenecks. The following machines are considered. As the baseline processor we take the 4-way issue SSE-enhanced CPU with a 128-entry RUU. The other parameters as in Table 5.5. We also consider an 8-way issue processor extended with SSE, for which we double the RUU size and the number of functional units. The processors are configured with 1, 2, or 4 SSE execution units. The same processors are enhanced with a CSI execution unit having a 128-, 256-, or 512-bit wide datapath. So, these CPUs are also able to perform 4, 8, or 16 single-precision FP operations in parallel. Each of the processors is then configured with 2 or 4 cache ports.

Figure 5.11 depicts the speedups of the SSE- and CSI-enhanced processors over the baseline 4-way SSE-enhanced processor with two cache ports that is capable of performing four FP operations is parallel. The results for CPUs equipped with a two-ported cache are denoted with 'mem2', and those for CPUs with a four-ported cache with 'mem4'. The bars denoted with '4fp', '8fp', and '16fp' show the speedups attained by the SSE- or CSI-enhanced CPUs capable of performing 4, 8, or 16 FP operations in parallel, respectively. The presented results lead to the following observations. First, the 4-way SSE-enhanced CPUs cannot efficiently exploit more than one SSE unit (4 parallel FP operations). Since increasing the number of cache ports does not improve the performance significantly, we conclude that the issue width of such CPUs constitutes the bottleneck. This conclusion is supported by the observation that increasing the issue width of an SSE-enhanced CPU from 4 to 8 speeds up the geometry pipeline approximately by the factor of 1.8, if sufficient cache ports are provided. The performance of the 8-way SSE-extended CPU strongly depends on the number of cache ports, leading to the conclusion that the cache ports constitute the bottleneck for this machine. We also observe that even provided with 4 ports, the 8-way SSE-enhanced CPUs cannot utilize more then 2 SSE units (8 parallel FP operations).

The results also show that, contrary to the SSE-enhanced processors, the CSI-enhanced CPUs can efficiently exploit the hardware that allows to perform up to 16 FP operations in parallel and that they can provide high performance without increasing the issue width. This is expected, since the parallelism is exploited by CSI in a single instruction and not in the form of ILP. This is particularly evident for the **xform** kernel which is fully streamed. A larger issue width does improve, however, the performance of the geometry pipeline as a whole because it improves the performance of the scalar (non-streamed) code sections.

In addition, we observe that performance of a CSI-enhanced CPU is rather insensitive to the number of cache ports. This is also expected, because the L1-interfaced implementation of CSI we study delivers a whole cache line in a single access and, therefore, performs fewer accesses. This is illustrated in Figure 5.12, which depicts the ratio of the number of L1 cache accesses performed by the SSE- and CSI-enhanced processors to the number of the accesses performed by the conventional superscalar processor. The ability of CSI to exploit a

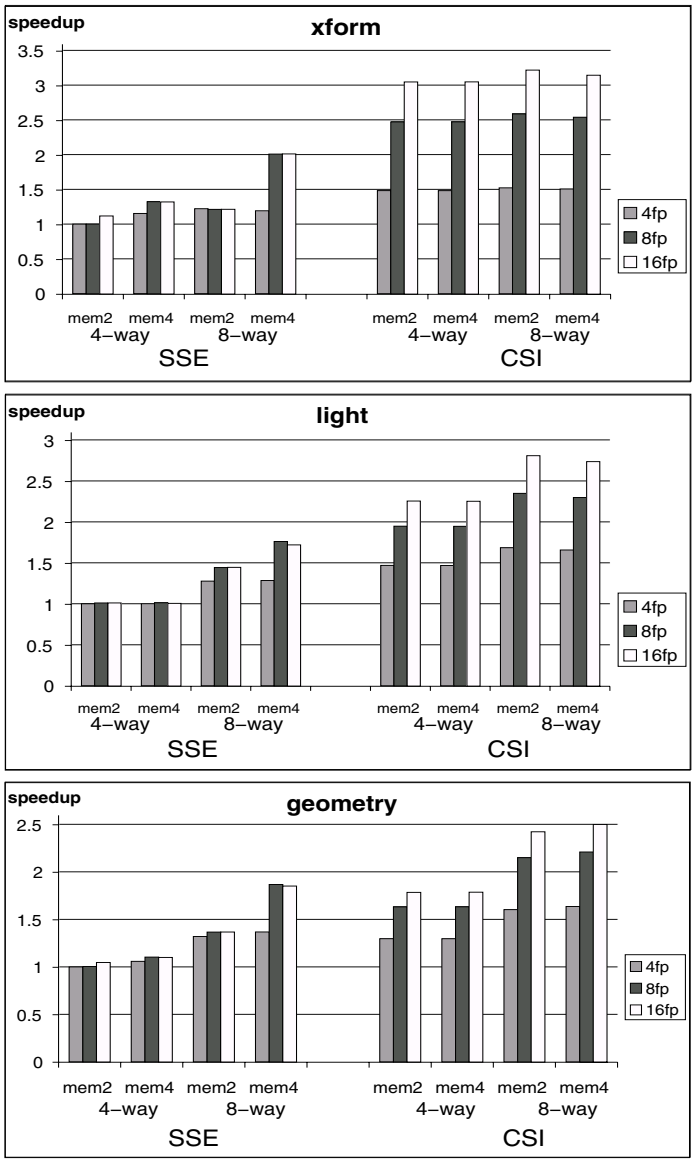


Figure 5.11: Speedups of CSI and SSE CPUs over the baseline 4-way SSE CPU.

large number of parallel FP processing hardware without having to increase the issue width or the number of cache ports is very important, because increasing any of these parameters bears huge hardware costs and can negatively affect the cycle time. On the other hand, the increasing scale of integration and transistor budgets allows to put dozens of functional units of a single chip and the main challenge is how to utilize them efficiently. CSI provides a

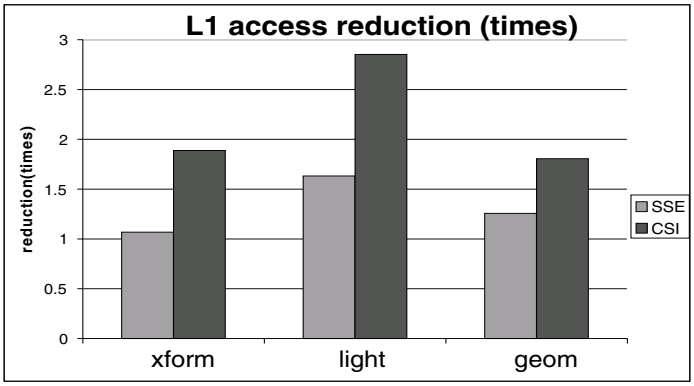


Figure 5.12: Reduction of L1 data cache accesses.

solution, scaling very well when the amount of parallel execution hardware is increased. We observe that a 4-way CSI-enhanced CPU equipped with a two-ported cache and capable of performing 16 parallel FP operations outperforms the very aggressive 8-way SSE-enhanced CPU with a four-ported cache on the **xform** and **light** kernels, and performs just 10% slower on the whole geometry pipeline.

L1 Cache Performance. Concerning the memory-to-memory nature of the CSI architecture, the following issue has to be addressed. When a complex sequence of computations is implemented in CSI, intermediate results have to be written to temporary streams in memory. For the chosen CSI implementation with the stream unit connected to the L1 data cache, this may result in cache pollution and thrashing, when every cache miss causes a cache block writeback. To evaluate whether this problem really occurs, we study the replacement traffic between the L1 and L2 cache and the number of L1 misses. Thrashing can be characterized using the *L1 thrash rate* which is defined as the average number of L1 writebacks per L1 cache miss. Figure 5.4 shows that for both **xform** and **light** kernels, as well as for the entire geometry computation, the CSI-enhanced processor exhibits thrash rates that are within 10% of those of the standard processor. This fact can be explained by the following observation. In general, thrashing is likely to be severe for CSI if the number of the distinct memory locations which has to be accessed is close to the cache size, so that the accesses to these locations are likely displace the useful data from the cache. We remark, however, that the memory locations (and, consecutively, the cache locations), which are occupied by the elements of a temporary stream and hold intermediate results, can be reused by another temporary stream as soon as these intermediate results become *dead*, i.e. used for the last time. Using this technique, we reduced the number of cache locations required for the temporary streams. From the two considered kernels, **light** involved more complex computations. In our implementation it required 7 temporary streams of length n, and one stream of length $3n$, where n is the number of vertices in a primitive. We recall, that, for the DX-06 dataset, $n = 95$ on average. The elements of temporary streams were always 32-bit (integer or floating-point)

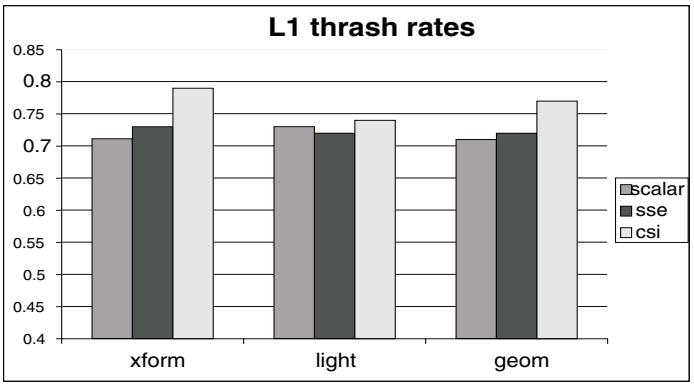


Figure 5.13: L1 cache thrash rates.

values. Hence, the average number of bytes occupied by temporary streams was equal to $4 \cdot (7 \cdot 95 + 3 \cdot 95) = 3800$ bytes. Since the L1 cache size was set to 32 Kbytes, temporary streams did not cause significant cache pollution and thrashing.

Conclusions

We have studied the applicability and the performance benefits provided by the CSI extension on the 3-D geometry computations. It was shown that the most important parts of the 3D geometry pipeline can be implemented using the proposed extension. The performance benefits provided by CSI were compared with those achieved by CPUs equipped with functional units capable of executing Intel's *SSE* extension. They were evaluated by implementing several routines of the *Mesa* 3D library using CSI and SSE instructions and by evaluating the library performance using the industry-standard 3D performance evaluation benchmark *viewperf*.

For the 4-way issue machine with an RUU of 32 entries and floating-point resources capable of performing 4 single-precision floating-point operations is parallel, the speedups of CSI over SSE on the kernels `xform` and `light` were 1.70 and 1.84, respectively. These speedups translated to an application speedup of 1.40. The performance scalability of SSE and CSI with respect to the number of floating-point computing resources was also studied, showing that CSI can exploit more parallelism. For the processors capable of executing 8 single-precision FP operations in parallel and having an 128-entry instruction window, the speedups of CSI over SSE were equal to 2.46, 1.92, and 1.62 for the `xform` and `light` kernels, and the whole geometry stage, respectively. The performance bottlenecks of the SSE-enhanced superscalar CPUs on the 3D geometry computation are identified. Results show that the performance of a 4-way machine is limited by the issue width and that of a 8-way machine is limited by the number of the cache ports. Furthermore, studied a number of miscellaneous performance-related parameters. We observed that CSI reduces the dynamic instruction counts of the `xform` and `light` kernels by the factors of 30 and 5.1 and that of the whole geometry pipeline by the factor of 2.71. SSE provides the reduction only by the factors of 1.45, 1.13, and 1.15, respectively. We also observed that a processor equipped with

the proposed L1-interfaced implementation of the CSI execution unit performs significantly less L1 cache accesses than a processor equipped with the SSE execution hardware (1.44 time less accesses on the geometry stage). Finally, we observe that the memory-to-memory nature of the CSI extension, which requires storing the results of the intermediate computations to the temporary streams, does not result in a significant increase of thrashing in the L1 cache.

5.5 Performance Scalability and Bottlenecks

Among other issues studied in Section 5.4, we investigated how the performance of the SSE- and CSI-enhanced processors on the floating-point intensive benchmark *viewperf* scaled when the amount of SIMD processing hardware is increased. We recall that this is an important issue because of the current trends in microprocessor technology, which allow a designer to fit dozens of functional units on a single chip. This poses a challenge to design an implementation that can exploit these units efficiently, providing them with sufficient amounts of instructions and data. We observed that CSI allows to exploit additional SIMD processing hardware efficiently, while SSE doesn't do it so well, and identified the bottlenecks of the SSE-enhanced processors. In this section, we perform a similar study for integer multimedia benchmarks, namely JPEG/MPEG-2 codecs and image-processing kernels. The main goal of the experiments is to evaluate how the performance of a superscalar processor extended with VIS or CSI scales with the amount of SIMD execution hardware. We also explore the design space of such processors by varying key parameters such as the instruction window size and the issue width in order to identify the performance bottlenecks. The benchmark set consists of MPEG-2 and JPEG codecs (`mpeg2encode`, `mpeg2decode`, `cjpeg`, `djpeg`) from the *Mediabench* suite [40] and of several image processing kernels (`add8`, `scale8`, `blend8`, `convolve3x3`) taken from the VIS Software Development Kit (VSDK) [30]. For this study, we use the same executables of the MPEG-2/JPEG codecs as those that were developed and studied in Section 5.2. For the VSDK kernels, we employed the executables that were used for the study presented in Section 5.3. The input datasets for the codecs and VSDK kernels were the same as the ones used in experiments reported in Section 5.2 and Section 5.3, respectively.

Modeled Processors

A wide range of superscalar processors was simulated by varying the issue width from 4 to 16 instructions per cycle and the instruction window size (i.e, the number of entries in the RUU unit) from 32 to 512. Table 5.5 summarizes the basic parameters of the processors and lists the functional unit latencies. Processors were made capable of processing VIS or CSI instructions by adding the VIS functional units or the CSI unit. The CSI unit is interfaced to L1 cache, as it was done in the previous studies. Each time the issue width is doubled, the number of integer and floating-point units is scaled accordingly. The number of cache ports is fixed at 2, however, because multi-ported caches are expensive and hard to design (for example, currently there are no processors with a 4-ported cache). Note, that the CSI-enhanced processors do not require more ports to cache then the VIS-enhanced ones. The CSI-enhanced processors use two cache ports, sharing loads and stores generated by the CSI

Clock frequency	666 MHz
Issue width	4/8/16
Instruction window size	16-512
Load-store queue size	8-128
Branch Prediction	
Bimodal predictor size	2K
Branch target buffer size	2K
Return-address stack size	8
FU type and number	
Integer ALU	4/8/16
Integer MULT	1/2/4
Cache ports	2
Floating-point ALU	4/8/16
Floating-point MULT	1/2/4
VIS adder	2/4/8
VIS multiplier	2/4/8

FU latency/recovery (in cycles)	
Integer ALU	1/1
Integer MUL	
multiply	3/1
divide	20/19
Cache port	1/1
FP ALU	2/2
FP MUL	
FP multiply	4/1
FP divide	12/12
sqrt	24/24
VIS adder	1/1
VIS multiplier	
multiply and pdist	3/1
other	1/1

Table 5.5: Processor parameters.

Instruction cache	ideal
Data caches	
L1 line size	32 bytes
L1 associativity	direct-mapped
L1 size	32 KB
L1 hit time	1 cycle
L2 line size	128 bytes
L2 associativity	2-way
L2 size	128 KB
L2 hit time	6 cycles
L2 replacement	LRU

Main memory	
type	SDRAM
row access time	2 bus cycles
row activate time	2 bus cycles
precharge time	2 bus cycles
bus frequency	166 MHz
bus width	64 bits

Table 5.6: Memory parameters.

unit between them. The VIS- and the CSI-enhanced CPUs were simulated with different amounts of SIMD-processing hardware by varying the number of the VIS functional units and the width of the CSI unit's datapath, respectively. The parameters of the cache and memory subsystems are summarized in Table 5.6. They are, essentially, the same as the ones presented in Section 5.2, except that the memory and bus clock frequencies are set to 166 MHz instead of 100 MHz. The ratio of CPU clock frequency to memory clock frequency was set to 4, corresponding to a CPU clock rate of 666 MHz.

Experimental Results

In this section we study the impact of the RUU size on the performance of the CSI- and VIS-enhanced processors. Then we analyze the performance behavior of these processors with respect to the number of SIMD functional units. After this, the performance bottleneck of the VIS-enhanced processors is identified. Finally, we present the speedups attained by CSI relative to VIS.

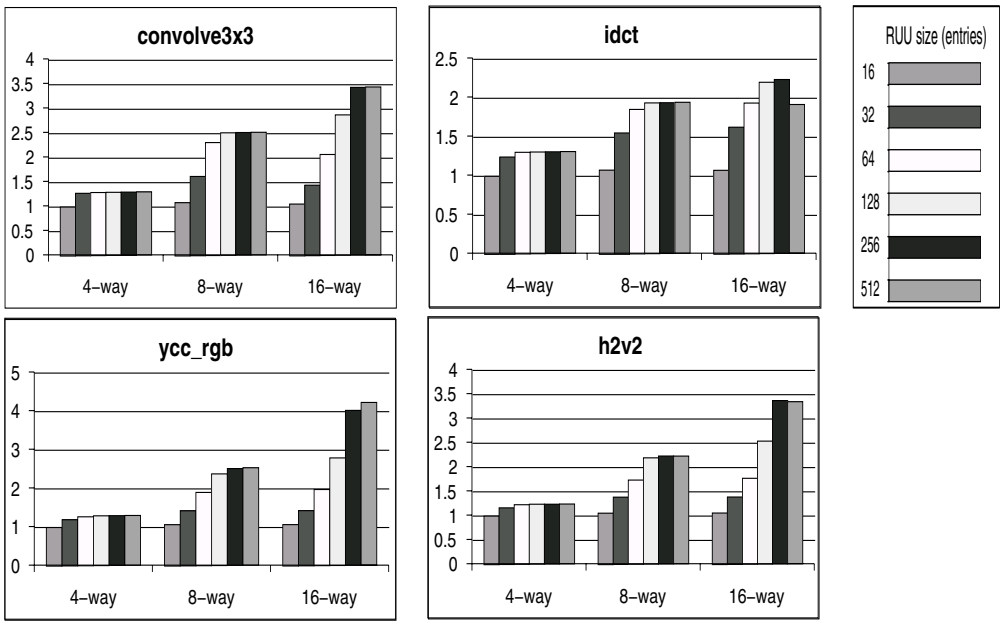
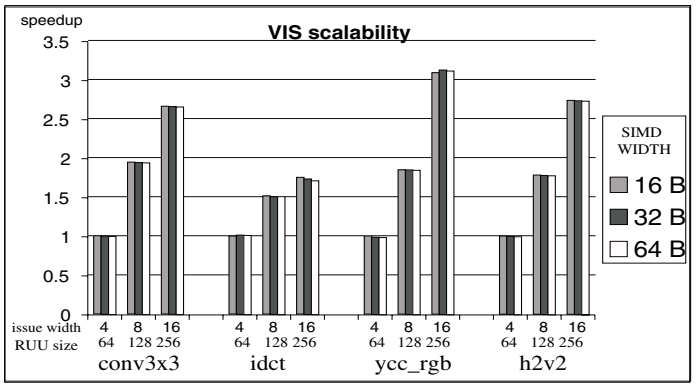


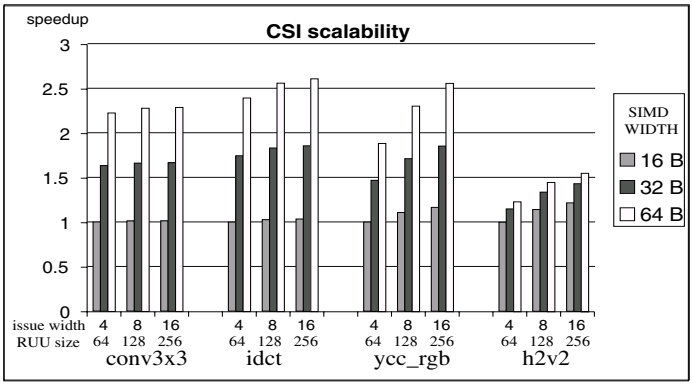
Figure 5.14: Kernel-level speedups for different issue widths and RUU sizes

Increasing the Window Size. First, we study the influence of the instruction window (RUU) size on the performance. Because the RUU is a rather costly resource, it is important to minimize its size. Figure 5.14 depicts the kernel-speedups attained by 4-way, 8-way, and 16-way VIS-enhanced processors with RUU sizes varying from 16 to 512 entries. The speedups are relative to the 4-way VIS-enhanced processor with a 16-entry RUU. These results show that for the studied kernels increasing the window size has a positive effect on the performance of a VIS-enhanced processor. A similar study performed on 4-, 8- and 16-way CSI-enhanced processors has shown that the window size has negligible influence on performance of these processors. This is not surprising for the following reason. The kernels usually operate on long data streams, performing the same operation independently on all stream elements. In VIS, long data streams are split into sections which fit into VIS registers and a separate instruction is generated for each section. The instructions are independent which means that the VIS ISA translates the parallelism present in the kernels into instruction-level parallelism (ILP). Larger instruction windows allow a VIS-enhanced superscalar CPU to expose and utilize larger amounts of ILP. On the other hand, in CSI the parallelism is exposed by a single CSI instruction which processes the whole stream. Therefore, a CSI-enhanced CPU does not need large instruction windows in order to exploit it. These results mean that the CSI-enhanced CPUs can be implemented with smaller (and cheaper) instruction windows then the VIS-enhanced ones.

The simulation results also show that, for each issue width, increasing the RUU size beyond a certain limit yields diminishing returns. The speedup of the 4-way VIS-enhanced processor saturates when the RUU consists of 64 entries, the 8-way machine when the RUU size is 128, and the 16-way processor reaches its peak performance when the RUU size is



(a)



(b)

Figure 5.15: Scalability of the CSI and VIS performance w.r.t. available SIMD hardware.

256. Therefore, we select these instruction window sizes both for VIS- and CSI-enhanced processors for all the experiments presented in the rest of the paper.

Increasing the Amount of Parallel Execution Hardware. The next question we consider is: how does the performance of the superscalar processors equipped with VIS or CSI scale with the amount of parallel execution hardware? This issue is important because of current trends in microprocessor technology. Larger scales of integration and, hence, growing transistor budgets allow to put dozens of functional units on a chip. The challenge for the designer is to keep these units utilized. The amount of SIMD processing hardware is characterized by the number of bytes which can be processed in parallel. For VIS this is determined by the number of VIS adders and multipliers. For example, to operate on 16 bytes in parallel, the machines are configured with 2 VIS adders and 2 VIS multipliers. For CSI, the amount of parallelism is determined by the width of the datapath of the CSI execution unit. In order to process 32 or 64 bytes in parallel the number of VIS units of the VIS-enhanced processor is doubled or quadrupled, whereas for CSI, the width of its datapath is increased appropriately.

As the baseline processors, we consider 4-, 8-, and 16-way superscalar CPUs with instruction windows of 64, 128, and 256 entries, respectively. Each baseline processor has been equipped with sufficient SIMD hardware so that it can process 16, 32, or 64 bytes in

parallel. Let a VIS-extended processor with an issue width of m, a window size of n, and a SIMD processing width of k bytes be denoted by *VIS*(m, n, k). A similar notation will be used for CSI-enhanced CPUs. Figure 5.15(a) depicts the speedups attained by all simulated VIS-enhanced processors relative to *VIS*$(4, 64, 16)$. The speedups achieved by all simulated CSI-enhanced CPUs relative to *CSI*$(4, 64, 16)$ are plotted in Figure 5.15(b). All VIS and CSI kernels exhibited similar behavior and, therefore, we only present the performance figures of four selected kernels. Figure 5.15(a) shows that, for a fixed issue width, increasing the number of the VIS functional units does not yield any benefit. Contrary to VIS, CSI is able to utilize additional SIMD execution hardware (see Figure 5.15(b)), and its performance scales almost linearly with the amount of SIMD resources. It can also be observed that the performance of CSI-extended processors is less sensitive to the issue width than that of VIS-enhanced processors. This is expected since CSI does not exploit instruction-level parallelism and, therefore, does not need to issue a lot of instructions in parallel.

Figure 5.16 presents the speedups attained by CSI-enhanced processors with respect to VIS-extended processors equipped with the same amount of SIMD execution hardware, i.e., the figure shows the speedups of *CSI*(m, n, k) over *VIS*(m, n, k) for various values of m, n, and k. It shows that CSI clearly outperforms VIS, especially for the 4-way and 8-way issue configurations, provided sufficient amount of parallel processing hardware is available. We remark that 4- and 8-way issue processors are less costly and can probably achieve a higher clock frequency than a 16-way one. We also notice that in some cases *VIS*(m, n, k) processor outperforms corresponding CSI-enhanced processor, *CSI*(m, n, k). For example, on the `convolve3x3` kernel, *VIS*$(8, 128, 16)$ outperforms *CSI*$(8, 128, 16)$. This can be explained by the following observation. *VIS*$(8, 128, 16)$ machine is equipped with two VIS adders and two VIS multipliers so that it can always issue two VIS instructions in a single cycle (i.e., process $8 \cdot 4 = 16$ bytes in parallel), provided the instructions are independent. In some cases, however, this machine can issue four VIS instruction in a single cycle and, hence, process $8 \cdot 4 = 32$ bytes in parallel. Such a situation can occur, for example, if at a certain cycle 4 VIS instructions, of which two require a VIS adder and two other require a VIS multiplier, are ready to be issued. On the other hand, the *CSI*$(8, 128, 16)$ processor, which is equipped with a CSI unit having 16-byte wide datapath, can process not more than 16 bytes in parallel. This observation means that, in fact, a VIS-enhanced processor may exploit higher levels of parallelism than the corresponding CSI-enhanced processor and, hence, be faster. A kernel is likely to exhibit such behavior if in its VIS implementation the ratio of the number of instructions that need a VIS adder to the number of instructions that need a VIS multiplier is close to $1 : 1$. The `conv3x3` kernel is likely to exhibit such a ratio, since it performs 9 multiplications and 8 additions per each output pixel.

Identifying the Bottleneck. It is important to identify where the performance bottleneck of the VIS-enhanced machines lies. Figure 5.15(a) shows that the VIS-enhanced machines perform better when the issue width is increased. This suggests that the issue width is the limiting factor for VIS. To investigate this, we study the IPCs attained by the VIS-enhanced CPUs. For each issue width, Figure 5.17 depicts the attained IPCs normalized to the corresponding ideal IPCs (e.g., for a 4-way machine, the ideal IPC is 4). It shows that, for most kernels, the performance achieved by the 4-way and 8-way superscalar VIS-enhanced processors is close to ideal. These CPUs achieve IPCs which are within 78% to 90% of the

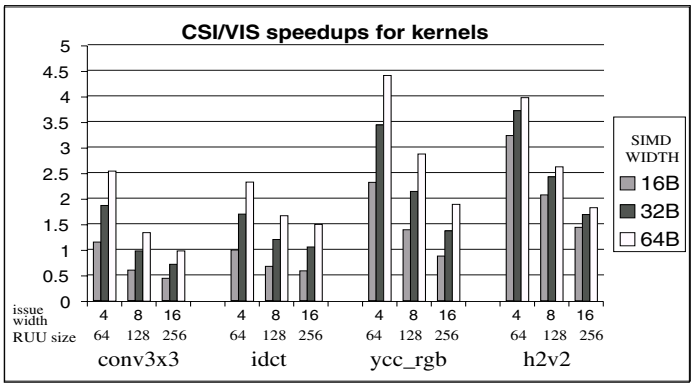


Figure 5.16: Speedup of CSI over VIS for several kernels.

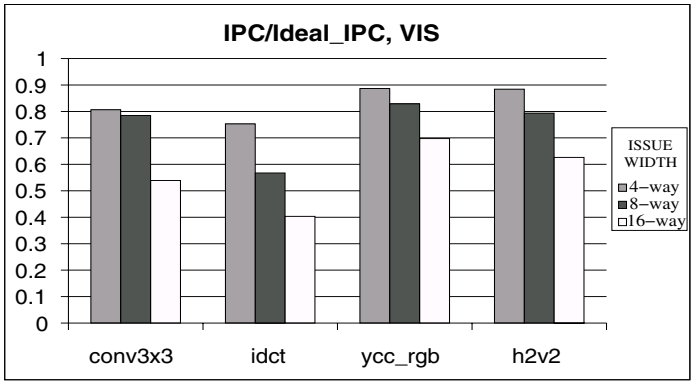
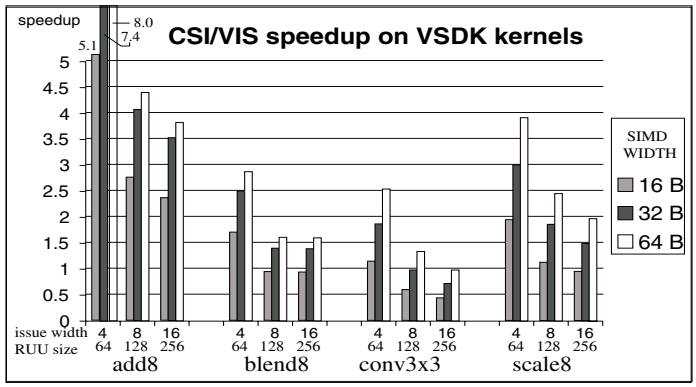
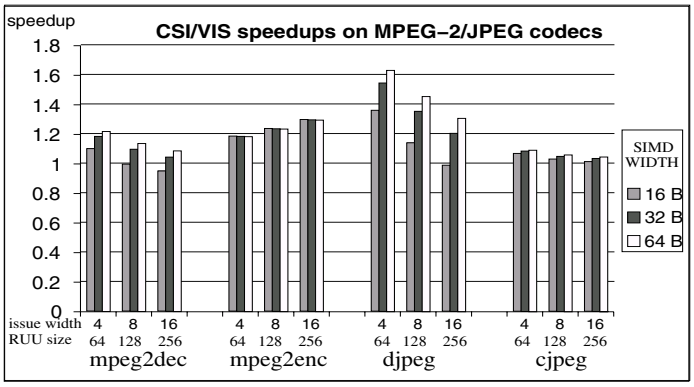


Figure 5.17: Ratio of achieved IPC to ideal IPC for several VIS kernels and issue widths.

ideal IPC for all kernels except `idct`. This means that they cannot be accelerated more than by factors ranging from 1.11 ($= 1/0.9$) to 1.28 ($= 1/0.78$). Corresponding CSI-enhanced processors outperform these processors by much larger factors (see Figure 5.16). Therefore, even if they achieve their ideal IPCs, performance of these VIS-enhanced CPUs will not approach that of corresponding CSI-enhanced CPUs, This observation shows that the issue width indeed limits the performance of the VIS-enhanced CPUs. The IPCs attained by the 16-way machine are far from the ideal and are within 55% to 70% of the ideal IPC, for all kernels except `idct`, meaning that the accelerations of 42-81% are possible. However, such wide-issue machines are not likely to be built in the near future due to their complexity and cost constraints. In conclusion, Figure 5.18 depicts the VSDK kernel and application speedups achieved by CSI over VIS. Of course, the speedups achieved by CSI on MPEG-2/JPEG codecs are smaller than those achieved on VSDK kernels due to Amdahl's law. Still rather impressive application-level speedups are achieved by CSI, especially for smaller (and more realistic) issue widths of 4 and 8. For example, when the issue width is 4 and 32 bytes can be processed in parallel, CSI achieves speedups ranging from to 1.08 (`cjpeg`) to 1.54



(a)



(b)

Figure 5.18: Speedup of CSI over VIS on VSDK kernels and MPEG/JPEG codecs.

(`djpeg`) with an average of 1.24.

Conclusions

We have evaluated how the performance of the CSI and VIS multimedia ISA extensions scales with the amount of SIMD parallel execution hardware. We have also performed experiments to identify the bottlenecks of the VIS-enhanced superscalar CPUs. Our results can be summarized as follows:

- The kernel-level performance of CSI increases almost linearly with the width of the CSI datapath. It improves by a factor of 1.56 when the width of the CSI datapath is doubled and by an additional factor of 1.27 when the width is quadrupled. Furthermore, the performance of CSI-enhanced processors is less sensitive to the issue width than that of VIS-enhanced ones.

- The VIS-enhanced CPUs do not perform substantially better when the number of VIS functional units is increased. The issue width is the limiting factor for the 4-way and 8-way issue processors and the decode/issue logic, therefore, constitutes a bottleneck.

Since the issue width already constitutes a bottleneck for realistic (4-way and 8-way) VIS-enhanced processors, we did not perform a study that investigates the influence of the number of cache ports on the performance of these processors, as it was done in Section 5.4. The reason why issue width is a limiting performance factor for integer media benchmarks executed on 4-way or 8-way VIS-enhanced CPUs, while it is not for the floating-point intensive `viewperf` application executed on SSE-enhanced CPUs having the same issue widths, can be explained as follows. Integer media benchmarks require much higher numbers of overhead instructions than the 3-D graphics application `viewperf`. This happens due to the following facts. First, processing floating-point data does not require packing and unpacking, while processing of 8-bit and 16-bit integer data, which is typical for integer media applications, does. Second, the data that has to be processed in 3-D geometry computations performed by `viewperf` is usually stored consecutively, while the data streams processed by integer benchmarks often exhibits non-unit strides (for example, the color conversion routines of JPEG codecs need to access data with stride 3, and the upsampling/downsampling routines of MPEG codecs access data with stride 2). Non-unit strides result in additional overhead instructions which are required to arrange data consecutively in multimedia registers (VIS registers in our case). On a very aggressive 16-way processor the performance impact of the extra instruction traffic generated by overhead instructions will be less, and the number of cache ports may become the limiting performance factor for integer media benchmarks, as it was for the floating-point ones on the 8-way SSE-enhanced CPUs. We did not study this in more detail because such an aggressive processor is not likely to be implemented in the near future. Furthermore, implementation of the L1 caches with a number of ports larger than two is also not likely to become common, since such caches are more expensive and have longer access times. In fact, the current trend in the CPU design, namely, increasing the processor's clock frequency by means of superpipelining, requires simple caches in order to maintain an access time (hit latency) of 1 cycle.

The results of this experimental study have the following implications for multimedia ISA and processor design. To increase the performance of a CPU extended with a short-vector media ISA on integer media benchmarks there are two main options. The first one is to reduce the pressure on the decode-issue logic by increasing the size of the multimedia registers. The second one is to increase the issue width of the CPU. Both of them bear huge costs: software incompatibility for the first and expensive hardware for the second. CSI offers a solution which provides scalable performance without having to increase the issue width, as well as software compatibility.

5.6 Conclusions

In this chapter we presented the results of an extensive experimental validation of the proposed CSI ISA extension. We studied the performance benefits that CSI provides for the superscalar processors enhanced with the CSI execution hardware on a large number of benchmarks originating from the following multimedia application domains: image and video coding/decoding, two-dimensional image processing, and three-dimensional (3-D) graphics processing. We compared the performance improvements which are provided by CSI with the improvements provided by the contemporary commercial short-vector media ISA extensions,

namely, the Sun's VIS extension and the Intel's SSE extension. We have also studied how the performance of the CSI-, VIS-, and SSE-enhanced CPU depends on the key parameters of the CPU, such as the issue width, the size of the instruction window, the main memory bandwidth, and the amount of parallel SIMD processing hardware.

The results of these studies showed the following: for all the studied applications, which are typical representatives of the image/video coding/decoding, two-dimensional image processing, and three-dimensional graphics application domains, the CSI-enhanced superscalar processors clearly outperform the processors enhanced with the contemporary commercial short-vector media extensions, namely Sun's VIS and Intel's SSE. The performance of the CSI-enhanced CPUs scales much better than that of the VIS- ans SSE-enhanced processors when the amount of the SIMD processing hardware or the memory bandwidth are increased. Such increases of the SIMD processing resources and memory bandwidth available to a processor are expected due to the current trends in the microprocessor and memory technology. Furthermore, the performance of the processors enhanced with the proposed CSI execution hardware interfaced to L1 data cache does not require an increase of the number of cache ports beyond two in order to utilize the increased amount of the SIMD processing hardware.

Chapter 6

Conclusions

In this dissertation we have presented a new media processor architecture. The discussion has been organized in 4 chapters, excluding the introduction. A brief outline of the achievements is as follows.

In Chapter 2 we have described the CSI architecture in general. First, we presented the architecture state of CSI, which consists of the CSI memory space and the CSI register space. The CSI memory space was assumed to coincide with the memory space of the general-purpose ISA to which CSI serves as an extension and a a detailed a detailed description of the CSI register space was presented. Then we described the operands of CSI instructions, concentrating on the description of the operand types that are particular for CSI, such as the CSI arithmetic streams and the CSI bit streams. The conditional execution support, for which the bit streams are introduced and used in CSI, was explained. After this we gave a detailed description of the interruption handling mechanism employed in CSI. This mechanism is based on representing the execution of a CSI instruction as a sequence of units of operation (UOPs) and on automatic update of the control registers that represent the extent to which the instruction was executed at the point of interruption. We showed that this mechanism allows resumption of the execution of an interrupted instruction from the point of interruption, which is an important condition for achieving high performance with CSI.

In Chapter 3 we presented a detailed description of the complete CSI instruction set and described how CSI instructions are executed. First, the CSI instruction format were listed, and the most common of them described in detail, showing, for each of them, how an instruction that has a certain format is divided in a number of fields and how these fields are interpreted. Then we described the CSI instruction classes, in which instructions are divided with respect to the number of elements they process, interruptibility, and dependency on the stream-mask mode. For instructions that belong to the same class, the number of elements they process is determined in the same way, such instructions have the same behavior in the presence of interrupts, and react in the same way on enabling/disabling of the stream-mask mode. Thereupon, the complete CSI instruction set was presented. CSI instructions were organized in the following groups with respect to the type of the operations they perform:

Simple Arithmetic and Logical, Comparison-Related, Accumulation-Related, Stream Reorganization, Bitstream-Operating, Special-Purpose, and *Auxiliary*. For each such group, we listed the instructions that belong to it, and described the way they are encoded and executed. We have also presented the auxiliary operations of converting stream data between storage and computation formats (packing and unpacking), which can be performed by a CSI instruction next to execution of its main operation. Finally, we described the CSI accumulation facility and how it is used.

In Chapter 4 we presented how a CSI execution unit can be implemented. First, a general design of such a unit was described. A typical CSI instruction loads the stream elements, unpacks them, if required, performs the main operation, packs and then stores the results. Since these operations, when performed on different stream elements, are usually independent, their execution can be overlapped. Therefore, the proposed general organization of such a unit has a pipeline structure. The design of such a unit depends on the level of memory hierarchy to which it is interfaced. In this chapter we provided a detailed description of a unit that interfaces to the first-level data cache. This design decision was motivated by the particular characteristics of the common multimedia applications, such as the MPEG-2/JPEG codecs. These applications exhibit high cache hit rates and, furthermore, require high data throughput. In the presented implementation, the entire cache block is transferred to or from the unit in a single cache access. When the hit rates are high, such a design provides high data throughput without increasing the number of cache ports, thus satisfying the needs of the streaming multimedia applications, which require high data bandwidth. We analyzed the computations needed for generation of address information for such a design. Although these computations turned out to be complex, we have shown that they can be performed in a pipelined fashion. A detailed design of a pipelined address generation unit that can compute address information for a new cache block every machine cycle was presented. The proposed unit is organized as a three-stage pipeline and the critical paths of this pipeline were identified.

In Chapter 5 we presented the results of extensive experimental validation of the proposed CSI ISA extension. We studied the performance exhibited by the CSI-enhanced superscalar processors on a large number of benchmarks originating from the following multimedia application domains: image and video coding/decoding, two-dimensional image processing, and three-dimensional (3-D) graphics processing. For superscalar processors, we compared the performance improvements provided by CSI with the improvements provided by contemporary commercial short-vector media ISA extensions, namely, the Sun's VIS extension and Intel's SSE extension. We have also studied how the performance of the CSI-, VIS-, and SSE-enhanced CPU depends on the key processor parameters, such as the issue width, the size of the instruction window, the main memory bandwidth, and the amount of parallel SIMD processing hardware. The results of these experimental studies showed the following. For all the studied applications, which are typical representatives of the image/video coding/decoding, two-dimensional image processing, and three-dimensional graphics application domains, the CSI-enhanced superscalar processors clearly outperform the processors enhanced with the contemporary commercial short-vector media extensions, namely Sun's VIS and Intel's SSE. The performance of the CSI-enhanced CPUs scales much better than that of the VIS- ans

SSE-enhanced processors when the amount of the SIMD processing hardware or the memory bandwidth are increased. Such increases of the SIMD processing resources and memory bandwidth available to a processor are expected due to the current trends in the microprocessor and memory technology. Furthermore, the performance of the processors enhanced with the proposed CSI execution hardware interfaced to L1 data cache does not require an increase of the number of cache ports beyond two in order to utilize the increased amount of the SIMD processing hardware.

The CSI architectural concepts have been presented in a number of conferences. The original concept was presented at the 26-th *Euromicro Conference (2000)*. The performance evaluation of the CSI on the image/video codecs and on the image-processing kernels were presented at the *International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, and at the *European Conference on Parallel Computing (Euro-Par'2001)*, respectively. The results of the study comparing the performance of the CSI- and SSE-enhanced superscalar processors on the 3D graphics were presented on the 14-th *IASTED International Conference on Parallel and Distributed Computing (PDCS-2002)*, while the study of the performance scalability of the CSI- and the VIS-enhanced processors was presented at the *Euro-Par'2002 Conference*. The example implementation of the CSI execution unit was presented at the *Euromicro Symposium on Digital System Design (2002)*. Several additional materials concerning the CSI architecture are currently under preparation.

6.1 Major Contributions

In the introduction we presented the reader with a number of questions, which had to be answered in order overcome the limitations inherent in the contemporary commercial short-vector media ISA extensions. These questions were illustrated by an example implementation of a typical multimedia kernel with an existing commercial short-vector SIMD multimedia extension, Motorola's *AltiVec*. For convenience, we pose these questions again here. One of the main questions that needed to be answered was: **Can the number of instructions needed to implement a kernel be reduced?** This general question translates in the following more specific questions.

- Can the number of instructions needed to specify the main computation be reduced ?
- Can the instructions associated with the sectioning be eliminated? Or, more specifically;
  - Can loop setup instructions be eliminated?
  - Can pointer update instructions be eliminated?
  - Can loop control instructions be avoided ?
- Can load and store instructions be eliminated?
- Can the instructions associated with conversion between storage and computational formats (packing and unpacking) be eliminated?

```
#Initialization
   sub r4, r4, 1 # compute (length-1), the length of all streams
   li r5, 1 # put 1 in r5
   csi_mt_scr r1, SCR1, 0  # set Base SCR for the 1st source stream
   csi_mt_scr r5, SCR1, 1 # set HStride to 1
   csi_mt_scr r4, SCR1, 2 # set HLength to length
   csi_mt_scr r0, SCR1, 3 # set VStride to 0, since streams are 1-dimensional
   csi_mt_scr r1, SCR1, 4 # set VLength to 1 (single row)
   csi_mt_scr r0, SCR1, 5 # set CurrCol to zero
   csi_mt_scr r0, SCR1, 6 # set CurrRow to zero
       #load the constant to the 16 MSB (leftmost) bits of a register
       # constant will set the fields of the Misc SCR to specify
       # GrSize = 2 (bytes), ElSize =1 (byte), FP=0, Sign =0, ProcessSize=2 (bytes),
       # ShiftAmount=0, ShiftDirection =0, Saturate = 0, Round =0
   lui r6, 0x41
   csi_mt_scr r6, SCR1, 7 # set Misc

       # similarly, 9 more instructions will initialize the SCR-set SCR2
       # to the parameters of the second source stream
       # similarly, 9 more instructions will initialize the SCR-set SCR3
       # to the parameters of the destination stream
#Main operation
   csi_paeth SCR3, SCR1, SCR2, 0 # compute paeth prediction for the whole row
       # operand '0' is used to specify that
       # csi_paeth has to perform prediction, not coding/decoding
```

Figure 6.1: CSI code for *paeth_predict_row*.

- Can the instructions associated with the reorganization of data so that it is located consecutively be eliminated?

In order to illustrate how the studies of the CSI architecture reported in this dissertation allow to answer these questions, we present the reader with the CSI implementation of the kernel *paeth_predict_row* in Figure 6.1. An *AltiVec* implementation of this routine was presented in the introduction. It is assumed that the parameters of the implemented procedure *prev_row*, *curr_row*, *predict_row,* and *length* are contained in the general purpose registers $r1, r2, r3$, and $r4$, respectively. It is assumed that the $r0$ register contains zero. For brevity, some of the instructions have been omitted in the presented CSI code. We now present the answers to the questions listed above, in the same order as they were stated.

- With respect to the reduction of the number of instructions needed to specify the main operation we conclude that the number of these instructions can be reduced to a single one. This is achieved by incorporating in CSI the special-purpose media instructions, such as `csi_paeth`. We note here that it was shown by other researchers that for such instructions hardware implementations with non-prohibitive requirements (in terms of area and delay) can be developed. For example, for the computations performed by `csi_paeth`, such implementation was presented in [25].

- The questions related to the reduction of instructions associated with sectioning overhead can be answered as follows.

  – Instructions associated with loop set up cannot be eliminated. In the presented CSI code these instructions are substituted with the CSI instructions needed to initialize stream parameters that are contained in the SCR-sets. These are the $2 + 9 \cdot 3 = 29$ instructions located between the labels `Initialization` and `Main operation`. We remark here, that although 29 initialization instructions are required, they will be executed just once and by the main processor, which is likely to be a fast multiple-issue superscalar CPU. It can be shown that on a 4-way

superscalar machine these instructions can be scheduled in such a way that every cycle 4 of them are issued (except of the last cycle, at which only the remaining instruction will be issued). Therefore, the initialization code will be executed on such processor in just $28/4 + 1 = 8$ cycles. When streams that have to be processed are sufficiently long, which is likely to be the case since they represent image scanlines, the initialization overhead will be negligible. Furthermore, we remark here that if more rows of an image will have to be processed, only the **Base**, **CurrCol**, and **CurrRow** registers of each SCR-set have to be reinitialized, which reduces the overhead even more.

- The overhead instructions associated with the update of the pointers to the data streams are completely eliminated in CSI. This is achieved by the automatic updates of the **Base** control registers which are performed implicitly in hardware by the main computation instruction `csi_paeth`.
- The loop overhead instructions are completely eliminated. This is achieved by automatically updating the **CurrCol** control registers and checking the termination condition performed by the `csi_paeth` instruction.

- With respect to the load and store instructions we observe that these are completely eliminated. This was achieved, in general, due to the memory-to-memory nature of the CSI architecture. In the presented example, the associated operations are performed implicitly by `csi_paeth`. We remark here, that because of the pipelined organization of the CSI execution unit, these operations will be overlapped with the main computations.

- The instructions associated with the conversions between storage and computation formats are completely eliminated. The conversion operations are performed implicitly by the `csi_paeth` instruction and, for the implementation of the CSI execution unit presented in this dissertation, will be overlapped with the main computation.

- The instructions associated with the reorganization of data in consecutive order are completely eliminated. Although such instructions were not needed for the AltiVec implementation of the presented kernel, they are often required in other media kernels. CSI allows to dispose of these instructions. The associated task is performed implicitly by the main computation instruction. In the presented implementation of the CSI execution unit, the memory-interface subunit will perform this task.

Another open question stated in the introduction is the following one.

- Can a multimedia ISA extension be designed in such a way that increasing amounts of parallelism will be exploitable by a single instruction, while the software compatibility will be retained ?

We give a positive answer to this question. CSI is an example of such an architecture. While the data streams that have to be processed are completely specified by the SCR-sets, the number of elements that can be processed in parallel is not fixed in CSI architecture. It is determined by the *SIMD_width* parameter and is implementation dependent. A more aggressive

implementation of CSI with an increased amount of parallel SIMD processing hardware will be able to execute old CSI codes while fully utilizing the available parallel SIMD hardware.

Generally speaking, the main contributions of our proposal include the following.

- We eliminate the need for sectioning and the associated overhead.
- We avoid the overhead instructions associated with conversion of the stream data between computation and storage formats and with data rearrangement.
- We avoid explicit load and store instructions and reduce the associated overhead by performing memory accesses in parallel with the main computation of a CSI instruction.
- The codes written using the proposed extension can utilize the increased amount of the parallel execution hardware without being recompiled.

We should note the following: the processor architecture presented in this dissertation is a vector architecture. Consequently, the concepts presented here are applicable to all vector architectures with no (or, at most, trivial) changes. This means that the CSI proposal can be part of any vector architecture extension, including the general-purpose vector facilities.

6.2 Proposed Research Directions

While a comprehensive description and experimental validation of the CSI architecture has been provided in this dissertation, there exist a number of interesting issues which can be addressed in the future. Below, we present possible directions for future research.

- Compare the CSI with a wider range of the media extensions, industrial as well as those proposed in academic research. In particular, a comparison of the performance improvements provided by CSI with those that can be achieved by employing the *Matrix-Oriented Multimedia (MOM)* ISA extension will be interesting, since this proposal addresses some of the issues addressed by CSI. Such a study was not performed in this dissertation for the following reasons. In order to make a fair comparison of the CSI and MOM architectures, high-quality MOM codes are required, which are not publicly available at the current moment.
- Since the CSI architecture, as currently proposed, is a prototype architecture, the applications studied in this dissertation, as well as other media applications, can be investigated in order to find additional instructions (in particular, those performing complex operations) that can included in order to complete the CSI instruction set.
- In the general design of the CSI execution unit presented in Chapter 4, a designer was given a freedom in the implementation of the memory-interface subunit (MIU). This subunit can be interfaced to the level-one (L1) data cache (as was proposed in that chapter), as well as to the L2 cache, or even to the main memory. Since the CSI instructions operate on long data streams, they are likely to be latency tolerant, which can provide sufficient performance for CSI implementations with L2- or memory-interfaced

MIUs. The L2- or memory-interfaced CSI implementations can be viable solutions for the embedded systems domain, since large and sophisticated L1 caches are usually not employed in such systems due to cost constraints. Since CSI instructions have access to information about the complete data streams they access, it is particularly interesting to study an implementation of a CSI execution unit which is interfaced to the main memory via a smart controller capable of reordering the requests in order to exploit the multibanked internal structure of contemporary DRAM chips, such as the controllers proposed by McKee et al. [43, 70].

- CSI provides support for conditional execution in two different ways: first, by providing masked stream instructions and, second, by means of instructions that allow to conditionally extract, insert, split and merge data streams. A comparative study of these two techniques can be made so that an application programmer will be able to decide which of two methods is better for a particular code fragment that has to be streamed.

- While CSI codes are currently produced by hand, the issue of compilation for CSI architecture has to be studied. We expect that, at least, codes for kernels that perform simple arithmetic and logical operations in a stream fashion can be generated by a compiler, since automatic code generation for such type of kernel has been proven to be possible by vectorizing compilers in the scientific computing application domain.

Bibliography

[1] K. Akeley and T.Jermoluk. High-Performance Polygon Rendering. *Computer Graphics*, 22:239–249, 1998.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, 2002.

[3] R. Bhargava, L.K. John, B.L. Evans, and R. Radhakrishnan. Evaluating MMX Technology Using DSP and Multimedia Applications. In *Proc. Int. Symp. on Microarchitecture (MICRO 31)*, pages 37–46, 1998.

[4] W. Buchholz. The IBM System/370 Vector Architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[5] D. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report 1342, Univ. of Wisconsin-Madison, Comp. Sci. Dept., 1997.

[6] D. Burger, J.R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 78–89, 1996.

[7] W.C. Chen, C.H. Smith, and S.C. Fralick. A Fast Computational Algorithm for the Discrete Cosine Transformation. *IEEE Trans. on Communication*, 25:1004–1009, 1977.

[8] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff. Architectural Support for 3D Graphics in the Complex Streamed Instruction Set. In *Proc. Int. Conf. on Parallel and Distributed Computing and Systems (PDCS 14)*, 2002.

[9] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H. Wijshoff. Performance Scalability of the Multimedia Instruction Set Extensions. In *Proc. Euro-Par'02*, 2002.

[10] J. Corbal, R. Espaca, and M. Valero. On the Efficiency of Reductions in micro-SIMD media extensions . In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[11] J. Corbal, R. Espasa, and M. Valero. Exploiting a new level of DLP in multimedia applications. In *Proc. Int. Symp. on Microarchitecture (MICRO)*, 1999.

[12] AMD Corp. AMD 3DNow! Technology. Document available via http://www.amd.com/K6/k6docs/pdf/21928.pdf.

[13] IBM Corp. *Enterprise Systems Architecture/390 Vector Operations. Document Number SA22-7207-00*. IBM Corp., 1st edition, 1991.

[14] Intel Corp. Diffuse-Directional Lighting, Application Note AP-596. Document available via http://www.intel.com/software/products/college/ia32/strsimd/appnotes/ap596.

[15] Intel Corp. Streaming SIMD Extensions - Matrix Multiplication, Application Note AP-930. Document available via http://developer.intel.com/design/pentiumiii/sml/245045.htm.

[16] Intel Corp. PC SDRAM Specification, Revision 1.7. Document available via http://www.intel.com/technology/memory/pc133sdram/spec/sdram133.htm, 1999.

[17] Microsoft Corp. Microsoft Office Resource Kit, Chapter 21. Document available via http://www.microsoft.com/office/ork/021/021.htm.

[18] NVIDIA Corp. Transform and Lighting, Technical Brief . Document available via http://www.nvidia.com/docs/IO/1345/ATT/ TransformAndLighting.pdf.

[19] SPEC CPU2000 V1.2 Documentation. Document available via http://www.spec.org/cpu2000/docs/.

[20] Samsung Electronics. 256Mbit K4H560438B DDR SDRAM part specification. http://www.samsungelectronics.com/semiconductors/DRAM/DDR_SDRAM/256M_bit/K4H560838B/.

[21] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics - Principles and Practice*. Addison-Wesley, 1996.

[22] C. Frazier and R. Kempf, editors. *OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.1*. Addison-Wesley Pub Co., 1997.

[23] M. Gries. The Impact of Recent DRAM Architectures on Embedded Systems Performance. In *Proc. EUROMICRO 26*, 2000.

[24] L. Gwennap. AltiVec Vectorizes PowerPC. *Microprocessor Report*, 12(6), 1998.

[25] E. Hakkennes and S. Vassiliadis. Multimedia Execution Hardware Accelerator. *Journal of VLSI Signal Processing*, 28(3):221–234, 2001.

[26] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[27] David P. Hunter and Eric B. Betts. Measured effects of adding byte and word instructions to the Alpha architecture. *Digital Technical Journal of Digital Equipment Corporation*, 8(4), 1996.

[28] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 2nd edition, 1984.

[29] Rambus Inc. Direct RDRAM 64/72-Mbit data sheet. Document available via http://www.rambus.com/docs/64dDDS.pdf, 1998.

[30] Sun Microsystems Inc. VIS Software Development Kit. Available via http://www.sun.com/processors/vis/vsdk.html.

[31] Sun Microsystems Inc. VIS Instruction Set User's Manual. Document available via http://www.sun.com/microelectronics/vis/, March 2000.

[32] Texas Instruments Inc. TMS320c80 Digital Signal Processor Data Sheet. Document available via http://www-s.ti.com/sc/ds/tms320c80.pdf, 1997.

[33] B. Juurlink, D. Tcheressiz, S. Vassiliadis, and H. Wijshoff. Implementation and Evaluation of the Complex Streamed Instruction Set. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[34] J. Keshava and V. Pentkovski. Pentium III Processor Implementation Tradeoffs. *Intel Technology Journal*, May 1999.

[35] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing With Streams. *IEEE Micro*, 21(2):35–47, 2001.

[36] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *Proc. 12th Hot Chips Conference*, 2000.

[37] C. Kozyrakis and D. Patterson. Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *Proc. Int. Symp. on Microarchitecture (MICRO-35)*, 2002.

[38] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9):75–78, 1997.

[39] G. Kuzmanov, S. Vassiliadis, and J. van Eindhoven. MPEG-4 Addressing and ACQ Function. In *Proc. 12th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC)*, 2001.

[40] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems. In *Proc. Int. Symp. on Microarchitecture (MICRO 30)*, 1997.

[41] D. Martin. Vector Extensions to the MIPS-IV Instruction Set Architecture (The V-IRAM Architecture Manual) Revision 3.7.5. Document available via http://http://iram.cs.berkeley.edu/, 2000.

[42] J. McCormack, R. McNamara, C. Gianos, N.P. Jouppi, T.Dutton, J. Zurawski, and L. Seiler amd K. Correll. Implementing Neon: A 256-Bit Graphics Accelerator. *IEEE Micro*, 19(2):58–69, 1999.

[43] S.A. McKee, W.A. Wulf, J.H. Aylor, R.H. Klenke, M.H. Salinas, S.I. Hong, and D.A.B. Weikle. Dynamic Access Ordering for Streamed Computations. *IEEE Micro*, 49(11):1255–1271, 2000.

[44] H. Nguyen and L.K. John. Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology. In *Proc. Int. Conf. on Supercomupting (ICS)*, 1999.

[45] T. Nguyen, A. Zakhor, and K. Yelick. Performance Analysis of an H.263 Video Encoder on VIRAM. In *Proc. Int. Conf. on Image Processing (ICIP)*, 2000.

[46] J.D. Owens, W.J. Dally, U.J. Kapasi, S. Rixner, P. Mattson, and B. Mowery. Polygon rendering on a stream architecture. *Proc. 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 2000.

[47] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, /1997.

[48] G.G. Pechanek, C.W. Kurak, C. J. Glossner, C.H.L. Moller, and S. J. Wals. M.F.A.S.T.: A Highly Parallel Single Chip DSP with a 2D IDCT Example. In *Proc. Int. Conf. on Signal Processing Applications and Technology*, pages 69–72, 1995.

[49] Alex Peleg and Uri Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[50] Alex Peleg, Sam Wilkie, and Uri Weiser. Intel MMX for Multimedia PCs. *Communications of the ACM*, 40(1):24–38, 1997.

[51] C. Price. *MIPS IV Instruction Set, revision 3.1*. MIPS Technologies, Inc., 1995.

[52] P. Ranganathan, S. Adve, and N.P. Jouppi. Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions. In *Proc. Int. Symp. on Computer Architecture (ISCA 26)*, 1999.

[53] S. Rixner, W.J. Dally, U.J. Kapasi, P.R. Mattson, and J.D. Owens. Memory access scheduling. In *Proc. Int. Symp. on Computer Architecture (ISCA 27)*, pages 128–138, 2000.

[54] S. Rixner, W.J. Dally, B. Khailany, P.R. Mattson, U.J. Kapasi, and J.D. Owens. Register organization for media processing. In *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 375–386, 2000.

[55] G. Roelofs. *PNG: The Definitive Guide*. O'Reilly and Associates, 1999.

[56] M. Sima, S. Cotofana, J. van Eindhoven, and S. Vassiliadis. An 8-point IDCT Computing Resource Implemented on a Trimedia/CPU64 Reconfigurable Unit. In *Proc. 12th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC)*, 2001.

[57] M. Sima, S. Cotofana, J. van Eindhoven, and S. Vassiliadis. Variable-Length Decoder Implemented on a TriMedia/CPU64 Reconfigurable Functional Unit. In *12th Annual Workshop on Circuits, Systems, and Signal Processing (ProRISC)*, 2001.

[58] M. Sima, S. Cotofana, J. van Eindhoven, S. Vassiliadis, and K. Vissers. 8x8 IDCT Implementation on an FPGA-augmented TriMedia. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 2001.

[59] M. Sima, S. Cotofana, J. van Eindhoven, S. Vassiliadis, and K. Vissers. MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented Trimedia Processor. In *Proc. Int. Conf. on Computer Design (ICCD)*, 2001.

[60] G. Slavenburg, S. Rathnam, and H. Dijkstra. The Trimedia TM–1 PCI VLIW Media Processor. In *Proc. 8-th HOT Chips Symposium*, pages 171–177, 1995.

[61] N.T. Slingerland and A.J. Smith. Measuring the performance of multimedia instruction sets. *IEEE Trans. on Computers*, 51(11):1317–1332, 2002.

[62] G. Sohi and S. Vajapayem. Instruction Issue Logic for High-Performance Interruptible Pipelined Processors . In *Proc. Int. Symp. on Computer Architecture (ISCA)*, 1987.

[63] Standard Performance Evaluation Corporation (SPEC). SPECviewperf 6.1.2. Document available via http://www.specbench.org/gpc/opc.static/opcview.htm.

[64] S. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, May 1999.

[65] M. Tremblay, J.M. O'Conner, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, 1996.

[66] S. Vassiliadis, B. Blaner, and R. Eickemeyer. SCISM: A Scalable Compound Instruction Set Machine Architecture. *IBM Journal of Research and Development*, 38(1), January 1994.

[67] S. Vassiliadis, B. Juurlink, and E. Hakkenes. Complex Streamed Instructions: Introduction and Initial Evaluation. In *Proc. EUROMICRO 26*, 2000.

[68] S. Wong, S. Cotofana, and S. Vassiliadis. General-Purpose Processor Huffman Encoding Extension. In *Int. Conf. on Information Technology: Coding and Computing*, 2000.

[69] C. Yang, B. Sano, and A .Lebeck. Exploiting Parallelism in Geometry Processing with General Purpose Processors and Floating-Point SIMD Instructions . *IEEE Trans. on Computers*, 49(9):934–946, 2000.

[70] L. Zhang, J.B. Carter, W.C. Hsieh, and S.A. McKee. Memory System Support for Image Processing. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 1999.

Samenvatting

Computers verwerken data door middel van het uitvoeren van machine instructies. Een machine instructie specificeert de data die verwerkt moet worden (als collectief operands genoemd) en de bewerking die op hen uitgevoerd moet worden. De programmeur zorgt voor de instructies. De data kunnen zich bevinden in verschillende typen van opslag in de computer. Alle verschillende typen opslag in de computer die zichtbaar zijn voor de programmeur heten de architectonische staat. De architectonische staat tezamen met alle instructies welke hierop opereren heten tezamen de Instructie Set Architectuur (ISA) of, in het kort, de architectuur. Een individuele processor die een bepaalde architectuur heeft wordt ook wel een implementatie van deze architectuur genoemd. Hierbij plaatsen wij als kanttekening dat een architectuur verschillende implementaties kan hebben. Een voorbeeld hiervan zijn de x386, x486, Pentium en Pentium-2 processoren van Intel, die allemaal implementaties zijn van dezelfde x86 architectuur.

De applicaties die op een computer uitgevoerd worden hebben zeer verschillende karakteristieken, en het is moeilijk om een architectuur te ontwikkelen waarvan een implementatie gemaakt kan worden welke alle mogelijke applicaties kan uitvoeren. Om deze reden introduceren en ontwikkelen computer wetenschappers nieuwe architecturen die gericht zijn op een specifiek applicatie domein. Populaire 'desktop' computers gebruiken over het algemeen generieke (general-purpose) processoren als Central Processing Unit (CPU). Dit type processor moet in staat zijn om elke mogelijke applicatie die potentieel interessant is voor de gebruiker uit te voeren, en als gevolg hiervan wordt hun architectuur gekenmerkt door flexibiliteit en algemeenheid. Een voorbeeld hiervan is de Intel Pentium processor familie, of de Sun UltraSparc familie. De architecturen van deze CPU's, de Intel x86 en de Sun UltraSparc, werden ongeveer 20 jaar geleden ontwikkeld en zijn gericht op beslissingsintensieve generieke applicaties met weinig parallellisme in de berekeningen. Een instructie bestaat in dit geval meestal uit een enkele operatie die uitgevoerd wordt op data die zich in registers of geheugen bevinden,

Het laatste decennium zijn multimedia applicaties, zoals audio/video compressie en decompressie, twee en drie dimensionale beeldverwerking, steeds belangrijker geworden omdat zij nieuwe zeer kostbare en aantrekkelijke diensten bieden aan de consument. Deze applicaties verschillen sterk van de eerder genoemde generieke applicaties. Een van de voornaamste verschillen is dat zij veel grotere hoeveelheden data verwerken. Verder dient deze verwerking meestal 'real-time' te gebeuren, wat nog zwaardere eisen oplegt aan de verwerkingscapaciteit van het systeem. Indien dergelijke applicaties worden uitgevoerd door een generieke processor dan is er voor elk individueel stukje data een aparte instructie nodig.

Als gevolg hiervan moet er binnen een beperkte tijd een groot aantal instructies uitgevoerd worden. De generieke processoren bleken niet in staat om de benodigde instructie verwerkingssnelheid te behalen die noodzakelijk is voor deze multimedia applicaties. Een van de beperkende factoren is de volgende. De meeste populaire processoren zijn van het 'superscalar' type. Gedurende elke klok cyclus kan een dergelijke processor tegelijk aan verschillende nieuwe instructies beginnen. Maar door de complexiteit van de instructie decodeer logica, die bepaalt aan welke instructies begonnen kan worden, is dit aantal beperkt. De bovengrens, ook wel 'issue-width' genoemd, is de maximale snelheid waarmee instructie verwerkt kunnen worden, en dit bleek onvoldoende te zijn voor multimedia applicaties.

In tegenstelling tot general-purpose toepassingen, bevatten multimediale toepassingen een groot deel van parallelisme op data niveau: dezelfde berekening wordt vaak uitgevoerd, zelfstandig en op veel gegevens. Daarom is het mogelijk om een general-purpose ISA uit te breiden met *multimediale registers* en instructies die op deze opereren. Een multimediale register kan een korte vector bevatten (meestal 8 of 4 elementen). Eén media instructie verwerkt parallel alle elementen in een vector. In het ideale geval kan het aantal uitgevoerde instructies gereduceerd worden met een factor van 4 of 8 afhankelijk van het aantal elementen per korte vector. Voorbeelden van dergelijke ISA uitbreidingen zijn *Multimedia Extension (MMX)* en *Streaming SIMD Extensions (SSE)* van Intel en *Visual Instruction set (VIS)* van Sun. Korte vector ISA uitbreidingen staan een reductie toe in het aantal uit te voeren instructies and verbeteren de prestaties. Desalniettemin zijn de geboden voordelen niet genoeg om aan de eisen van toekomstige media toepassing te voldoen. In Hoofdstuk 1 geven we de redenen daarvoor. Als eerste worden de ideale reducties van het aantal uitgevoerde instructies met factoren van 4 en 8 bijna nooit behaald. Dit wordt veroorzaakt door de extra instructies die moeten worden uitgevoerd om de lange vectoren te verdelen in stukken die in de media registers passen, het converteren van data tussen opslag en verwerk formaten, en andere taken. Verder het is nodig om meer instructies uit te voeren om zoedoende service kwaliteit van deze toepassingen te verbeteren, (bijvoorbeeld, het verhogen van de resolutie of het aantal beelden per seconde in digitale video) Omdat het aantal bewerkingen dat door een korte-vector instructie wordt uitgevoerd gelimiteerd is zal het aantal uitgevoerde instructies ook toenemen. Hierdoor zullen superscalar processoren die uitgebreid zijn met dergelijke extensies dezelfde beperkingen ondervinden als conventionele processoren en zullen zij niet in staat zijn instructies uit te voeren met het vereiste tempo.

In dit proefschrift presenteren we en evalueren we een nieuw media ISA extensie zonder de beperkingen van de korte-vectoren media extensies. Dit stelt general-purpose processoren in staat om de benodigde prestaties te leveren voor het ondersteunen van een wijde scala aan multimediale toepassingen zoals stem-, beeld- en video coderen en decoderen, twee dimensionale beeldverwerking en drie dimensionale graphics. Hoofdstukken 2 en 3 beschrijven de voorgestelde *Complex Streamed Instruction Set (CSI)* architectuur. Het basis idee achter deze architectuur is het reduceren van het aantal uitgevoerde instructies. Dit wordt gerealiseerd door gebruik te maken van een aantal technieken. Allereerst, kunnen de CSI instructies uitgevoerd worden op data stromen van willekeurige lengte en ongelimiteerd aantal elementen. Ten tweede, voert een CSI instructie niet alleen de basis operatie uit op de elementen van de data stroom, maar ook andere taken die nodig zijn voor het uitvoeren van datastroom operaties zoals de toegang tot de datastromen in het geheugen, het splitsen daarvan in secties die parallel bewerkt kunnen worden en het converteren van de elementen tussen hun opslag

en bewerkingsformaten (*packing* en *unpacking*). Als laatste bevat CSI een aantal specifieke (special purpose) instructies. Een dergelijke instructie specificeert een complexe aritmetische operatie en voegt de corresponderende sequentie van simpele operaties toe in een enkele instructie.

CSI instructies zijn complex en daarom is het van belang de implementeerbaarheid te onderzoeken. Dit is het onderwerp van Hoofdstuk 4 waarin een voorbeeld implementatie van een CSI executie eenheid gepresenteerd wordt. Eerst wordt een algemeen ontwerp van een dergelijke eenheid beschreven. Een typische CSI instructie laadt de data elementen, converteert ze indien nodig, voert de kern operatie uit, converteert de resultaten en slaat ze op. Omdat deze operaties meestal onafhankelijk van elkaar zijn kan hun executie overlappen. Om deze reden is er gekozen voor een zogenaamde pipeline structuur voor het algemeen ontwerp van de CSI eenheid. Het ontwerp van deze eenheid is afhankelijk van het niveau van geheugen hiërarchie aan welke deze is verbonden. In dit hoofdstuk geven we een uitgebreide beschrijving van een eenheid die voor de verbinding zorgt met de L1 data cache. In de gepresenteerde implementatie wordt een hele cache lijn getransporteerd van of naar de CSI eenheid per cache toegang. Wanneer de ”hit” ratio hoog is, kan een dergelijk ontwerp een hoge data doorvoer snelheid bereiken zonder het aantal ”cache” poorten te verhogen en zodanig aan de vereiste van multimedia toepassingen voldoen.

Hoofdstuk 5 presenteert de resultaten van een uitgebreide experimentele validatie van de voorgestelde CSI ISA uitbreiding. Wij hebben de prestaties van een superscalar processor uitgebreid met CSI bestudeerd op een groot aantal benchmarks uit verschillende media domeinen zoals video codering/decodering, twee dimensionale beeldverwerking en drie dimentionale (3D) graphics. Voor superscalar processoren hebben wij de prestatieverbeteringen van CSI vergeleken met de prestatieverbeteringen van huidige commerciële processoren die gebruik maken van korte-vector ISA uitbreidingen, nl. de VIS uitbreiding van Sun en de SSE uitbreiding van Intel. Deze experimentele studies hebben de volgende resultaten opgeleverd. Het gebruik van de CSI uitbreiding biedt de mogelijkheid tot het dramatisch reduceren van het aantal uitgevoerde instructies (tot 16.07 keer op ”kernel” niveau en tot 2.05 keer op toepassingsniveau). Deze verminderingen resulteren in een aanzienlijke verkorting van de executie tijd. Voor alle bestudeerde toepassingen presteert de CSI-uitgebreide superscalar processor duidelijk beter dan processoren uitgebreid met de huidige commerciële korte-vector media uitbreidingen (VIS en SSE). Verder schaalt de prestatie van de CSI-uitgebreide processor beter dan die van VIS- en SSE-uitgebreide processoren als de SIMD hardware middelen of geheugen bandbreedte wordt verhoogd. Dergelijke verhogingen van SIMD verwerkingsmiddelen en geheugen bandbreedte kunnen worden verwacht gezien de huidige trend in microprocessor en geheugen technologie. Deze resultaten tonen aan dat CSI een realiseerbare oplossing biedt voor het probleem van efficiënte executie van media toepassingen op general-purpose processoren.

Curriculum Vitae

Dmitry Cheresiz was born on September 18, 1974 in Novosibirsk, Russia. After graduating from Novosibirsk high school #130 in 1991 (cum laude), he started his studies at the Faculty of Mathematics and Mechanics of Novosibirsk State University. In 1995 he successfully finished his studies (cum laude). In 1997 Dmitry Cheresiz joined as a Ph.D. student the High Performance Computing group of the Computer Science Department of Leiden University headed by Prof. Dr. H.A.G. Wijshoff. The Ph.D. research of Dmitry Cheresiz was part of a joint project between this group and the Computer Engineering Laboratory of Delft University of Technology headed by Prof. Dr. S. Vassiliadis.