PISC: Polymorphic Instruction Set Computers

Stamatis Vassiliadis, Georgi Kuzmanov, Stephan Wong, Elena Moscu-Panainte, Georgi Gaydadjiev, Koen Bertels, and Dmitry Cheresiz

Computer Engineering, EEMCS,
Delft University of Technology,
Mekelweg 4, 2628 CD Delft, The Netherlands
S.Vassiliadis@ewi.tudelft.nl,
WWW home page: http://ce.et.tudelft.nl/

Abstract. We introduce a new paradigm in the computer architecture referred to as Polymorphic Instruction Set Computers (PISC). This new paradigm, in difference to RISC/CISC, introduces hardware extended functionality on demand without the need of ISA extensions. We motivate the necessity of PISCs through an example, which arises several research problems unsolvable by traditional architectures and fixed hardware designs. More specifically, we address a new framework for tools, supporting reconfigurability; new architectural and microarchitectural concepts; new programming paradigm allowing hardware and software to coexist in a program; and new spacial compilation techniques. The paper illustrates the theoretical performance boundaries and efficiency of the proposed paradigm utilizing established evaluation metrics such as potential zero execution (PZE) and the Amdahl's law. Overall, the PISC paradigm allows designers to ride the Amdahl's curve easily by considering the specific features of the reconfigurable technology and the general purpose processors in the context of application specific execution scenarios.

1 Introduction

Overall performance measurements in terms of Millions Instructions Per Cycle (MIPS) or cycles per instruction (CPI) depend greatly on the CPU implementation. Potential performance improvements due to the parallel/concurrent execution of instructions, independent of technology or implementations, can be measured by the number of instructions which may be executed in zero time, denoted by PZE (potential zero-cycle execution) [1]. The rationale behind this measurement, as described in [1] for compound instruction sets is:

"If one instruction in a compound instruction pair executes in n cycles and the other instruction executes in $m \leq n$ cycles, the instruction taking m cycles to execute appears to execute in zero time. Because factors such as cache size and branch prediction accuracy vary from one implementation to the next, PZE measures the potential, not the actual, rate of zero-cycle execution. Additionally, note that zero-cycle instruction execution does not translate directly to cycles per instruction (CPI) because all instructions do not require the same number

of cycles for their execution. The PZE measure simply indicates the number of instructions that potentially have been removed from the instruction stream during the execution of a program."

Consequently, PZE is a measurement that indicates maximum speedup attainable when parallelism/concurrency mechanisms are applied. The main advantage of PZE is that given a base machine design the benefits of proposed mechanisms can be measured and compared. We can thus evaluate the efficiency of a real design expressed as a percentage of the potentially maximum attainable speedup indicated by PZE. An example is illustrated in Figure 1. Four instructions, executing in a pipelined machine are considered. The instruc-

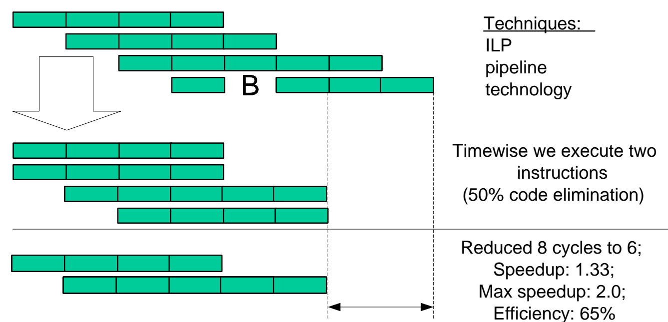

Fig. 1. PZE example.

tions from the example are parallelized applying different techniques, such as instruction level parallelism (ILP), pipelining, technological advances, etc., as depicted in Figure 1. Timewise, the result is that the execution of 4 instructions is equivalent to the execution of 2 instructions, which corresponds to a seeming code reduction by 50%, i.e., *2 out of 4 instructions potentially have been removed from the instruction stream during the program execution.* It means that the maximum theoretically attainable speedup (i.e., again potentially) in such a scenario is a factor of 2. In the particular example from Figure 1, the execution cycles count for 4 instructions is reduced from 8 to 6 cycles, allowing 1.33 times speedup, which compared to the maximum speedup of 2, suggests efficiency of 65%. The above example suggests that PZE allows to measure the efficiency of a real machine implementation by comparing to a theoretical base machine, i.e., PZE gives an indication of how close a practical implementation performs to the theoretically attainable best performance boundaries. These theoretical boundaries are described by Amdahl's law [2].

Amdahl's law and the new polymorphic paradigm. The maximum theoretically attainable (i.e., the potentially maximum) speedup, considered for the PZE, with respect to the parallelizable portion of the program code, is determined by Amdahl's law. Amdahl's curve, graphically illustrated in Figure 2, suggests that

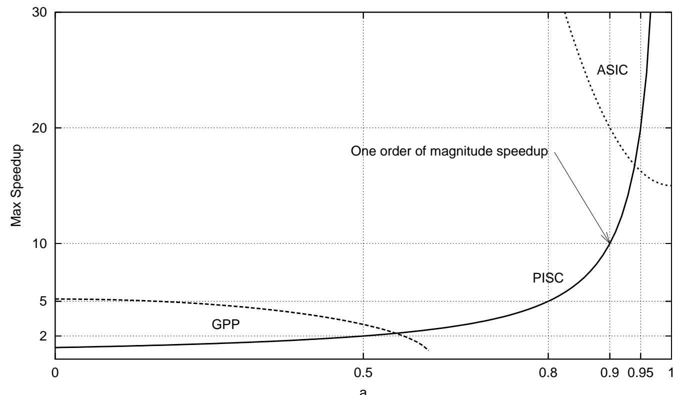

Fig. 2. The Amdahl's curve and PISC.

if, say half of an application program is parallelized and that its entire parallel fraction is assumingly executed in zero time, the speedup would potentially be 2. Moreover, the Amdahl's curve suggests that to achieve an order of magnitude speedup, a designer should parallelize over 90% of the application execution. In such cases, when over 90% of the application workload is considered for parallelization, it is practical to create an ASIC, rather than utilizing programmable GPP. The design cycle of an ASIC, however, is extremely inflexible and very expensive. Therefore, ASICs may not appear to be an efficient solution when we consider smaller portions (i.e., less than 90%) of an algorithm for hardware acceleration. Obviously, there exist potentials for new hardware proposals that perform better than GPPs and are more flexible alternative to design and operate than ASICs.

In this paper, we introduce a new architectural paradigm targeting the existing gap between GPPs and ASICs in terms of flexibility and performance. This new paradigm exploits specific features of the reconfigurable hardware technologies. In consistence with the classical RISC and CISC paradigms [3, 4], we refer to the new architectural paradigm as to a Polymorphic Instruction Set Computer (PISC). The practically significant scope of PISC covers between 50% and 90% application parallelization illustrated with the Amdahl's curve in Figure 2. This interval provides a designer with potentials to benefit from the best of two worlds, i.e., with a synergism between purely programmable solutions on GPPs and reconfigurable hardware. That is, the infinite flexibility of the programmable GPPs combined with reconfigurable accelerators results into a PISC - a programmable system that substantially outperforms GPP. Therefore, we believe that the gap between GPP and ASIC, illustrated in Figure 2, belongs to PISC. More specifically, we address the following research areas related to the PISC paradigm:

- New HW/SW co-design tools
- Processor architecture and microarchitecture

– Programming paradigm
– Compilation for the new programming paradigm

The remainder of this paper is organized as follows. In Section 2, we present a motivating example and derive key research questions. Section 3 describes the general approach to solve these research questions. The polymorphic architectural extension is presented in Section 4. Section 5 introduces some compiler considerations targeting the new architectural paradigm. Finally, the paper is concluded in Section 6.

2 Motivating example and research questions.

To illustrate the necessity of the PISC computing paradigm, we present a motivating example based on the Portable Network Graphics (PNG) standard [5]. PNG is a popular standard for image compression and decompression, it is a native standard for the graphics implemented in *Microsoft Office* as well as in a number of other applications. We consider the piece of C-code presented in Figure 3, which is extracted from an implementation of the PNG standard. This code

```
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++; } }
```

Fig. 3. The Paeth prediction routine according to PNG specification [5].

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, illustrated in Figure 4, selects from

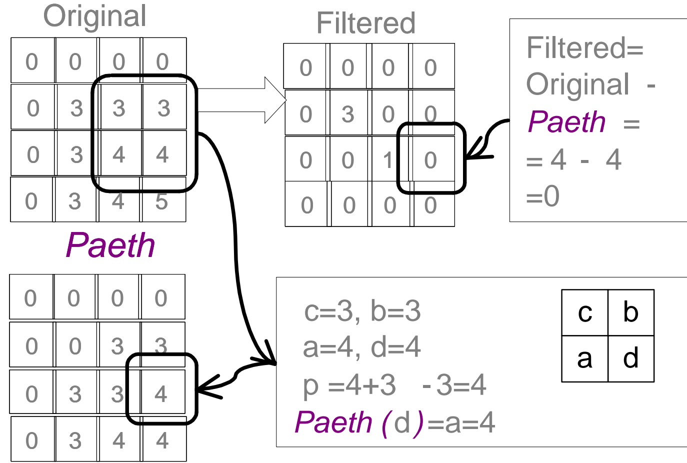

Fig. 4. The Paeth prediction scheme.

the 3 neighboring pixels a, b, and c, that surround d, 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. Figure 5 presents an implementation of the code fragment in pseudocode derived from the AltiVec assembly. In this figure, the general-purpose register GPRi of the underlying ISA is denoted by `ri`, `vri` denotes the i-th vector register of AltiVec.

Analysis of the motivating example presented above suggests the following. If the Paeth predictor must be computed for a row of 1024 pixels, the complete AltiVec code presented in Figure 5 will result in a dynamic instruction count of $8(prologue) + 64 \cdot [3(load) + 6(unpack) + 76(compute) + 1(pack) + 1(store) + 2(miscellaneous) + 3(pointerupdate) + 3(loopcontrol)] = 8 + 6464 \cdot 95 = 6088$ *instructions*. This high instruction count, which limits the performance, 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.

Considering Figure 5, we can substitute all loop iterations in the Paeth code with a single instruction and add only a few instructions to interface with the remainder of the program code. In such a case, we can expect considerable decrease of the instructions count and execution time improvements. The Paeth loop is transformed now in a single instruction [6] that takes 5 cycles to complete[1] and requires 20 setup instructions. The improvement attained is nearly two orders of magnitude reduction of the instructions count and two orders of

[1] One cycle is the duration of a single ALU operation

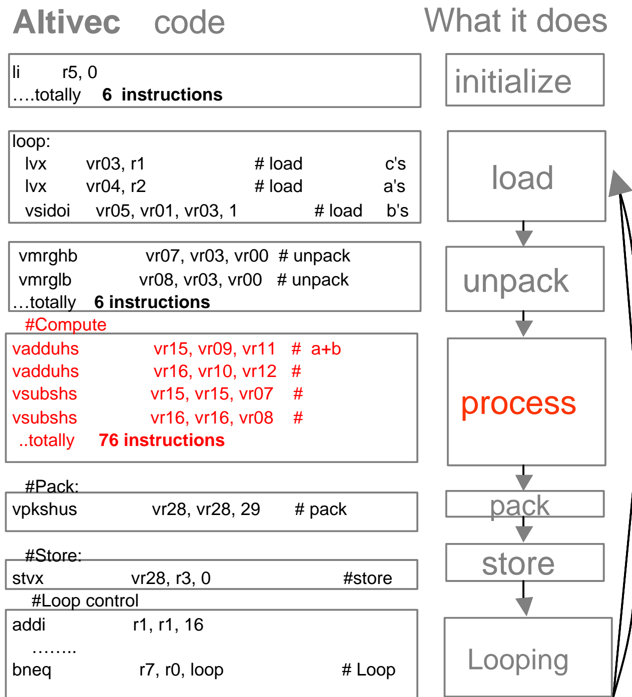

Fig. 5. AltiVec code for the *Paeth predict* kernel .

magnitude reduction of the execution time. Obviously, the scale of these improvements depends on the implementability of the Paeth coding into hardware as a single instruction. An efficient Paeth hardware implementation comprises 24 32-bit adders allowing a throughput of 16 pixels/cycle (i.e., 6 8-bit adders per pixel).

Research Questions. The Paeth encoding is just one computationally demanding kernel identified in a particular program. To implement an entire application efficiently, however, it is very likely that a number of such kernels should be identified within a single program execution context and each of them should be implemented in hardware. Therefore, traditional approaches, which introduce a new instruction for each portion of the application considered for hardware implementation, are restricted by the unused opcode space of the core processor architecture. Moreover, due to the large number of candidate kernels for hardware implementation, it may appear that their fixed hardware realization is impossible within limited silicon resources. The latter problem can be overcome, if the hardware can change its functionality at the designer's wish, i.e., using reconfigurable hardware. For many traditional reconfigurable approaches,

however, the above problems become even more dramatic if an arbitrary number of new operations should be considered for hardware implementation [7, 8]. In such scenarios,the traditional design methods can not be employed. The above observations arise the following research questions:

1. How to identify the code for hardware implementation?
2. How to implement "arbitrary" code?
3. How to avoid adding new instructions per kernel?
4. How to substitute the hardwired code with SW/HW descriptions say at source level?
5. How to generate the "transformed" program automatically?

With respect to the above questions, in this paper we address the following research topics:

1. New kind of tools.
2. Microarchitecture design.
3. Processor architecture (behavior and logical structure).
4. New programming paradigm allowing HW and SW to coexist in a program.
5. New compilation techniques.

3 General Approach

To solve the research questions stated in the previous section, we propose a synergism between a general-purpose processor (GPP) and a reconfigurable processor (RP) referred to as the Molen $\rho\mu$-coded processor. In the discussion to follow, we present the general concept of transforming an existing program to one that can be executed on the reconfigurable computing platform we propose and hints to the new mechanisms, intended to improve existing approaches.

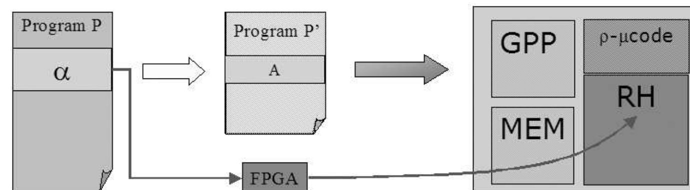

Fig. 6. Program transformation example.

The conceptual view of how program P (intended to execute only on the general-purpose processor (GPP) core) is transformed into program P' (executing on both the GPP core and the reconfigurable hardware) is depicted in Figure 6. The purpose is to obtain a functionally equivalent program P' from program P which (using specialized instructions) can initiate both the configuration and execution processes on the reconfigurable hardware. The steps involved in this transformation are the following:

1. identify code "α" in program P to be mapped in reconfigurable hardware.
2. show that "α" can be implemented in hardware in an existing technology, e.g., FPGA, and map "α" onto reconfigurable hardware (RH).
3. eliminate the identified code "α" and add "equivalent" code (A) assuming that A "calls" the hardware with functionality "α". The code A comprises the following:
   – Repair code inserted to communicate parameters and results to/from the reconfigurable hardware from/to the general-purpose processor core.
   – "HDL"-like hardware code and emulation code inserted to configure the reconfigurable hardware and to perform the functionality that is initialized by the "execute code".
4. compile and execute program P' with original code plus code having functionality A (equivalent to functionality "α") on the GPP/reconfigurable processor.

The mentioned steps illustrate the need for a new programming paradigm in which both software and hardware descriptions are present in the same program. It should also be noted that the only constraint on "α" is implementability, which possibly implies complex hardware. Consequently, the microarchitecture may have to support emulation [9] via microcode. We have termed this reconfigurable microcode ($\rho\mu$-code) as it is different from the traditional microcode. The difference is that such microcode does not execute on fixed hardware facilities. It operates on facilities that the $\rho\mu$-code itself "designs" to operate upon. We refer to such facilities as to configurable computing units (CCU). A processor supporting $\rho\mu$-code is referred to as a $\rho\mu$-coded processor and we also call it a Molen processor. More details on the Molen machine organization are presented in [11, 12].

The methodology of the transformation described previously for the reconfigurable computing platform is depicted in Figure 7. First, the code to be executed on the reconfigurable hardware must be determined. This is achieved by high-level to high-level instrumentation and benchmarking. This results in several candidate pieces of code. Second, we must determine which piece of code is suitable for implementation on the reconfigurable hardware. The suitability is solely determined by whether the piece of code is implementable (i.e., "fits in hardware"). Those parts can then be mapped into hardware via a hardware description language (HDL). In case the HDL corresponds to "critical" hardware in terms of, for instance, area, performance, memory and power consumption, the translation will be done manually. Otherwise, the translation can be done automatically or extracted from a library [13].

4 The Polymorphic ISA

In order to target the $\rho\mu$-code processor, we propose a sequential consistency programming paradigm [10]. The paradigm allows for parallel and concurrent hardware execution and requires only a one-time architectural extension of few

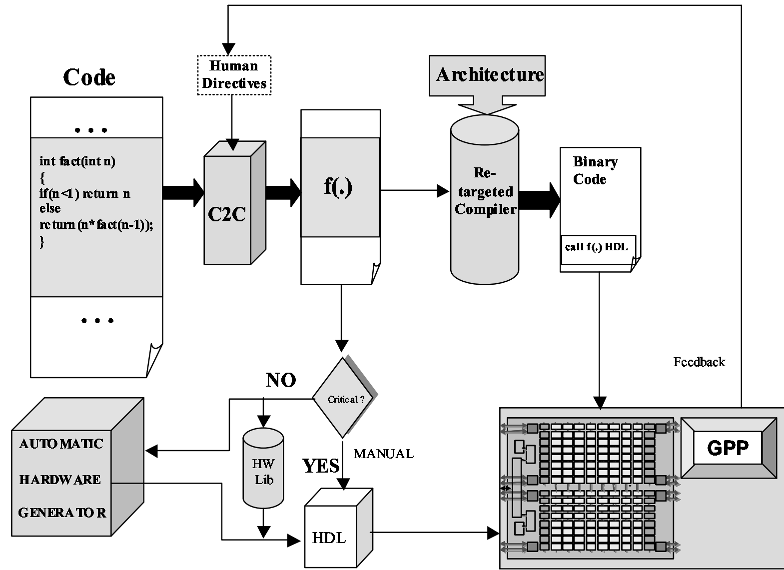

Fig. 7. Program transformation methodology for reconfigurable computing.

instructions to provide a large user reconfigurable operation space. The complete list of the eight required instructions, denoted as polymorphic ($\boldsymbol{\pi}o\lambda\nu\mu o\rho\phi\iota\kappa\acute{o}$) **I**nstruction **S**et **A**rchitecture (πISA), is as follows:

– Six instructions are required for controlling the reconfigurable hardware, namely:
  - Two **set** instructions: these instructions initiate the configurations of the CCU. Two instructions are added for partial reconfiguration:
    - the partial set (*p*-**set** <*address*>) instruction performs those configurations that cover common parts of multiple functions and/or frequently used functions.
    - the complete set (*c*-**set** <*address*>) instruction performs the configurations of the remaining blocks of the CCU (not covered by the *p*-**set**) to *complete* the CCU functionality.
  - **execute** <*address*>: controls the execution of the operations implemented on the CCU. These implementations are configured onto the CCU by the **set** instructions.
  - **set prefetch** <*address*>: prefetches the needed microcode responsible for CCU reconfigurations into a local on-chip storage facility (the $\rho\mu$-code unit) in order to possibly diminish microcode loading times.
  - **execute prefetch** <*address*>: the same reasoning as for the **set prefetch** instruction holds, but now relating to microcode responsible for CCU executions.

  - **break**: facilitates the parallel execution of both the reconfigurable processor and the core processor. It is utilized as a synchronization mechanism to complete the parallel execution.
- Two *move* instructions for passing values between the register file and exchange registers (XREGs):
  - **movtx** $XREG_a \leftarrow R_b$: (move to XREG) used to move the content of general-purpose register R_b to $XREG_a$.
  - **movfx** $R_a \leftarrow XREG_b$: (move from XREG) used to move the content of exchange register $XREG_b$ to general-purpose register R_a.

The <*address*> field in the instructions introduced above denotes the location of the reconfigurable microcode responsible for the configuration and execution processes. The parameters are passed via the exchange registers (XREGs). In order to maintain correct program semantics, the code is annotated. It is not imperative to include all instructions when implementing the Molen organization. The programmer/implementor can opt for different ISA extensions depending on the performance that needs to be achieved and the available technology. There are basically three distinctive πISA possibilities with respect to the Molen instructions introduced earlier - the *minimal*, the *preferred* and the *complete* πISA extension:

- **The minimal πISA:** This is essentially the smallest set of Molen instructions needed to provide a working scenario. The four basic instructions needed are **set** (more specifically: *c*-**set**), **execute**, **movtx** and **movfx**.
- **The preferred πISA:** In order to address reconfiguration latencies both *p*-**set** and *c*-**set** instructions are utilized. The two **prefetch** instructions (**set prefetch** and **execute prefetch**) provide a way to diminish the microcode loading times by scheduling them well ahead of the moment that the microcode is needed.
- **The complete πISA:** This scenario involves all πISA instructions including the **break** instruction. The **break** instruction provides a mechanism to synchronize the parallel execution of instructions by halting the execution of instructions following the **break** instruction.

Parallel execution. Parallel execution, for all πISA modifications is initiated by a **set/execute** instruction. For both minimal and preferred πISA, a parallel execution is ended by a general-purpose instruction as described in Figure 8(a). When a complete πISA is implemented and a sequence of instructions is performed in parallel, the end of the parallel execution is marked by the **break** instruction. It indicates where the parallel execution stops (see Figure 8 (b)).

Microarchitecture and its implementation. An example of a PISC is the Molen $\rho\mu$-coded processor introduced in [11]. More details on the Molen microarchitecture have been published in [12]. We have implemented a prototype design of a Molen processor using the Xilinx Virtex II Pro technology [14], which demonstrates many advantages of the PISCs and can be utilized for real-life application implementations. The core processor of the Molen prototype from [14] is the PowerPC hardcore embedded in the Xilinx virtex II Pro FPGAs.

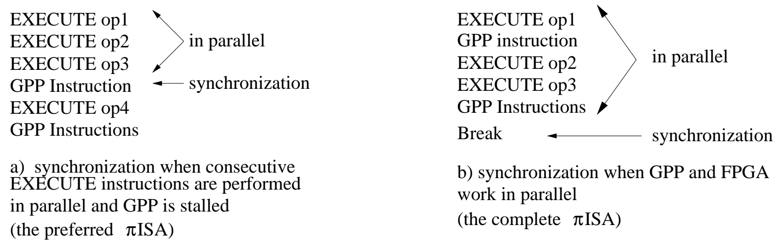

Fig. 8. Parallel execution and models of synchronization.

5 Compiler

The specific PISC compiling techniques will be illustrated with examples from the Molen compiler [15]. Currently, the Molen compiler relies on the Stanford SUIF2 [16] (Stanford University Intermediate Format) Compiler Infrastructure for the front-end and for the back-end on the Harvard Machine SUIF [17] framework. The following essential features for a compiler targeting custom computing machines (CCM) have currently been implemented:

- Code identification: for the identification of the code mapped on the reconfigurable hardware, we added a special pass in the SUIF front-end. This identification is based on code annotation with special pragma directives (similar to [18]). In this pass, all the calls of the recognized functions are marked for further modification.
- Instruction set extension: the compiler takes into account the instruction set extension and inserts the appropriate **set/ execute** instructions both at the medium intermediate representation level and at low intermediate representation (LIR) level.
- Register file extension: the register file set has been extended with the exchange registers. The register allocation algorithm allocates the XREGs in a distinct pass applied before the register allocation; it is introduced in Machine SUIF, at LIR level. The conventions introduced for the XREGs are implemented in this pass.
- Code generation: code generation for the reconfigurable hardware (as previously presented) is performed when translating SUIF to Machine SUIF intermediate representation, and affects the function calls marked in the front-end. The code generation schedules the **set** instructions to hide the reconfiguration latency and to guarantee that the functions can be mapped on the available area [19].

An example of the code generated by the extended compiler for the Molen programming paradigm is presented in Figure 9. On the left, the C code is depicted. The function implemented in reconfigurable hardware is annotated with a pragma directive named *call_fpga*. It has incorporated the operation name, *op1* as specified in the hardware description file. In the middle, the code generated

by the original compiler for the C code is depicted. The pragma annotation is ignored and a normal function call is included. On the right, the code generated by the compiler extended for the Molen programming paradigm is depicted; the function call is replaced with the appropriate instructions for sending parameters to the reconfigurable hardware in XREGs, hardware reconfiguration, preparing the fixed XREG for the microcode of the **execute** instruction, execution of the operation and the transfer of the result back to the general-purpose register file. The presented code is at medium intermediate representation level in which the register allocation pass has not been applied yet.

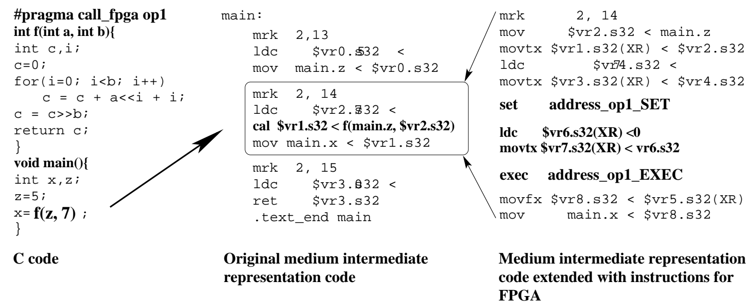

Fig. 9. Medium intermediate representation code.

The compiler extracts from a hardware description file the information about the target architecture such as the microcode address of the **set** and **execute** instructions for each operation implemented in the reconfigurable hardware, the number of XREGs, the fixed XREG associated with each operation, etc. The compiler may also decide not to use a reconfigurable hardware function and to include a pure software based execution.

6 Conclusions

We presented a new paradigm in computer architecture referred to as Polymorphic Instruction Set Computer (PISC). This new computing paradigm allows general purpose programming code and reconfigurable hardware descriptions to coexist within the same application program. We showed that a one-time instruction set extension of minimum 4 and maximum 8 polymorphic instructions is sufficient to implement an arbitrary number of application specific functionalities. Additional architectural features such as exchange registers and shared memory allow performance efficient communications, parameter and data exchange. We also presented the programming paradigm, supporting the polymorphic architectural extension and sketched some compiling considerations. Overall, we conclude that the PISC paradigm allows the designers to ride easily the Amdahl's curve towards the invention of more flexible and performance efficient computing machines.

References

1. S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, *SCISM: A scalable compound instruction set machine.* IBM J. Res. Develop. Vol. 38, No. 2, Jan 1994, pp. 59–78.
2. G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in *Proc. AFIPS 1967 Spring Joint Computer Conference*, 1967, pp. 483–485.
3. D. A. Patterson and D. R. Ditzel, *The case for the reduced instruction set computer*, SIGARCH Comput. Archit. News, Vol. 8, No. 6, Oct 1980, pp. 25–33.
4. D. Bhandarkar and D. W. Clark. *Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization.* Communications of the ACM, Sep 1991, pp. 310–319.
5. G. Roelofs. *PNG: The Definitive Guide.* O'Reilly and Associates, 1999.
6. E. A. Hakkennes and S. Vassiliadis, Hardwired Paeth codec for portable network graphics (PNG), in *Proc. Euromicro 99*, Sep 1999, pp. 318–325.
7. S. Hauck, T. Fry, M. Hosler, and J. Kao, The Chimaera Reconfigurable Functional Unit, in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 1997, pp. 87–96.
8. A. L. Rosa, L. Lavagno, and C. Passerone, Hardware/Software Design Space Exploration for a Reconfigurable Processor, in *Proc. Design, Automation and Test in Europe 2003 (DATE 2003)*, 2003, pp. 570–575.
9. S. Vassiliadis, S. Wong, and S. Cotofana, *Microcode Processing: Positioning and Directions*, IEEE Micro, vol. 23, no. 4, Jul 2003, pp. 21–30.
10. S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, The Molen Programming Paradigm, in *Proc. Third International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS'03)*, Jul 2003, pp. 1–7.
11. S. Vassiliadis, S. Wong, and S. Cotofana, The MOLEN $\rho\mu$-Coded Processor, in *Proc. 11th Int. Conf. on Field Programmable Logic and Applications (FPL 2001)*, Aug 2001, pp. 275–285.
12. S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, *The Molen Polymorphic Processor*, IEEE Transactions on Computers, vol. 53, Nov 2004, pp. 1363–1375.
13. J. M. P. Cardoso and H. C. Neto, *Compilation for FPGA-Based Reconfigurable Hardware*, IEEE Design & Test of Computers, vol. 20, no. 2, Apr 2003, pp. 65–75.
14. G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, The MOLEN Processor Prototype, in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, Apr 2004, pp. 296–299.
15. E. Moscu Panainte, K. Bertels, and S. Vassiliadis, Compiling for the Molen Programming Paradigm, in *Proc. 13th Int. Conf. on Field Programmable Logic and Applications (FPL)*, Sep 2003, pp. 900–910.
16. http://suif.stanford.edu/suif/suif2.
17. http://www.eecs.hardvard.edu/hube/research/machsuif.html.
18. M. Gokhale and J. Stone, Napa C: Compiling for a Hybrid RISC/FPGA Architecture, in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Apr 1998, pp. 126–135.
19. E. Moscu Panainte, K. Bertels, and S. Vassiliadis, Compiler-driven FPGA-area Allocation for Reconfigurable Computing, in *Proc. Design, Automation and Test in Europe 2006 (DATE 06)*, Mar 2006, pp. 369–374.