A Flexible Simulator of Pipelined Processors

Ben Juurlink Koen Bertels Bei Li

Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands
Phone: +31 15 27 81572, fax: +31 15 27 84898
benj@ce.et.tudelft.nl

Abstract

A flexible, parameterizable simulator of pipelined processors is presented. The simulator allows to configure several (micro-)architectural features such as the pipeline depth, the stage in which branch execution occurs, whether or not register file forwarding is performed, and the number of branch delay slots. We use the simulator to perform experiments with three synthetic benchmarks: vector addition, vector summation, and sum of absolute differences. These kernels are representative for data parallel loops, reduction operations, and benchmarks containing many hard to predict branches, respectively.

Keywords: simulation, pipelined processors, microarchitecture, embedded processors, energy reduction.

1 Introduction

Since superscalar processors are very power hungry, the core of many embedded systems is an in-order issue, pipelined processor. This is exemplified by the ARM processors: the ARM7 implements a 3-stage pipeline, the ARM10 has 5 stages, and the ARM11 8 stages. The optimal number of pipeline stages usually depends on the application. If it performs many independent operations, a deep pipeline is preferable and no forwarding datapaths are needed. If operations are dependent, a shorter pipeline is preferable and forwarding may be required to avoid stalls. Furthermore, if energy consumption is a concern, a deep pipeline is favorable because deep pipelines often translate to lower supply voltages and, hence, reduced energy consumption.

In order to investigate these trade-offs, a simulator is needed that allows to configure the pipeline depth, the forwarding datapaths, etc. However, to our knowledge, there does not exist a simulator that allows to configure the microarchitecture. For example, in [7] the authors propose to pipeline cache accesses to reduce the cache supply voltage and, thereby, energy consumption, but to evaluate their proposal they had to modify the MARS simulator [2].

In this paper we present such a flexible, configurable simulator. As is typical for load/store RISC architectures, it is assumed that the execution of every instruction passes through the following steps: Instruction Fetch, Instruction Decode, Execute, Memory Access, Write-Back. Each of these steps (or super-stages) may be split up in an arbitrary number of sub-stages. For example, pipelined caches can be simulated by specifying that the Instruction Fetch and Memory Access super-stages consist of

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Figure 1: Simple RISC pipeline

two or more sub-stages. In addition, the simulator allows to specify several other microarchitectural features such as the sub-stage in which branch execution takes place, whether or not forwarding is performed, and in which sub-stages the registers are read or written. We use the simulator to determine the "optimal" pipeline for some important kernels.

This paper is organized as follows. Section 2 describes the conventional, 5-stage pipeline and discusses several micro-architectural trade-offs. Related work is described in Section 3. Section 4 presents our simulator called Sketch and discusses the processor parameters that can be varies. Section 5 present the results of some experiments using three synthetic benchmarks representing different type of applications: a data-parallel loop (vector addition), a reduction operation (vector accumulate), and a loop containing many branches (sum of absolute differences). Conclusions are drawn and directions for future work are given in Section 6.

2 Background

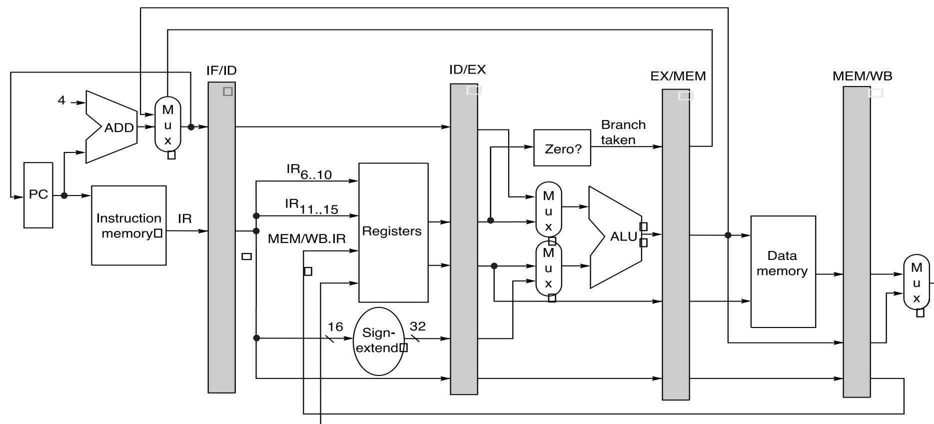
Figure 1 [5] depicts a conventional pipeline, as implemented for example by the MIPS R1000 processor.

It consists of five stages:

1. **Instruction Fetch (IF)**. In this stage an instruction is fetched from the instruction memory and the program counter is incremented.

2. **Instruction Decode/Operand Fetch (IF/OF)**. The instruction is decoded and its source operands are fetched from the register file. Note that the operands are fetched speculatively. This is possible because the register designators are always in the same location in the instruction format.

3. **Execute (EX)**. Depending on the the instruction, an ALU operation is performed, the effective address is calculated (for load/store instructions), or the branch condition is determined. The ALU computes the branch condition and an additional adder is used to compute the branch target address.



© 2003 Elsevier Science (USA). All rights reserved.

Figure 2: Pipelined datapath

4. **Memory access (MEM)**. In this stage load/store instructions access memory and branch instructions complete, meaning that if the branch is taken the PC is modified by the end of this stage.

5. **Write-back (WB)**. In this stage the result of the instruction is written back to the register file.

The pipelined datapath depicted in Figure 2 [5] suffers from two kinds of hazards: *data hazards* and *control hazards*. Data hazards occur because the result of an instruction is not written to the register file until the fifth stage (the write-back stage), while the register file is read during the second stage (the operand fetch stage). This implies that when an instruction I writes to a register r_d, the three instructions immediately following I in program execution order cannot use r_d as a source operand. A solution to this problem would be to stall a dependent instruction in the operand fetch stage until the instruction it depends on has written its result back to the register file, but in most cases this causes many stall cycles. In the R1000 processor data hazard stall cycles are avoided using two techniques. First, writing to the register file takes place during the first half of the processor cycle, while reading from the register file occurs during the second half of the cycle. This reduces the number of instructions that cannot use the destination register as a source operand to two. Second, a technique called *forwarding* or *bypassing* is applied. This is illustrated in Figure 3 [4]. Instead of waiting for a result to be written back to the register file, a Forwarding Unit is added that checks if the destination register of the instruction currently in the memory access stage or the detination register of the instruction in the write-back stage is equal to one of the source registers of the instruction currently in the execute stage. If this is the case, the multiplexors that determine the ALU inputs select the result of the instruction currently in the memory access or write-back stage rather than the values read from the register file in the previous cycle.

There is one situation, however, where the result of an instruction cannot be forwarded to the next instruction. This is when a load instruction is immediately followed by an instruction that uses the destination of the load as a source operand. This is because the result of a load is not available until the end of the fourth stage. The hazard detection unit depicted in Figure 3 [4] detects such cases and stalls the pipeline if necessary.

Note, however, that adding a forwarding unit increases the critical path of the execute stage.

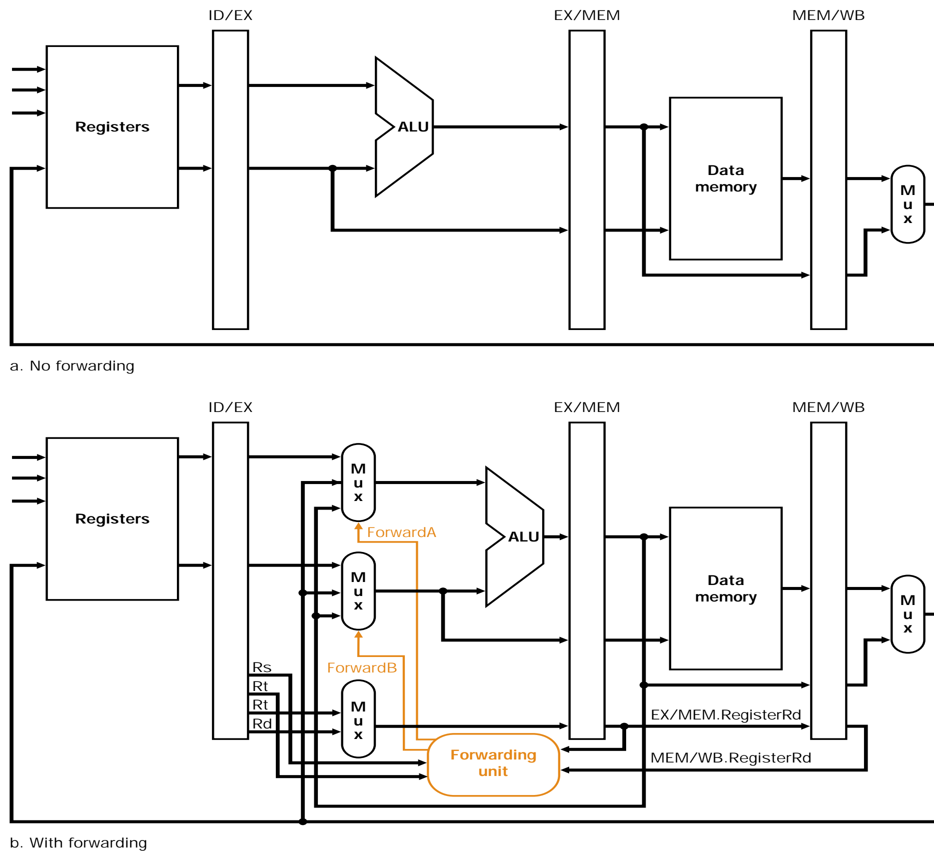


Figure 3: Datapath with forwarding

Depending on the critical path of the other stages, this may increase the cycle time. Therefore, in order to determine if forwarding is advantageous, micro-architectural simulations and cycle time estimates may be used.

The other kind of hazards the pipelined datapath depicted in Figure 2 [5] suffers from are control hazards. Because branch execution occurs in the fourth stage (the MEM stage), three instructions following the branch in program order have already been fetched into the pipeline. If the branch is taken, these instructions need to be flushed. In order to reduce the penalty of a taken branch, in the R1000 branch execution is moved to the OF stages, as illustrated in Figure 3 [4]. Immediately after the registers have been fetched from the register file, the branch condition is determined. In order to evaluate the branch condition in a very short time, the MIPS-I instruction set includes only a very limited set of easy to calculate branch instructions: branch if equal, branch if not equal, branch if less than zero, and branch if greater than zero. Branch if less than, for example, needs to be synthesized using the set if less than and branch if not equal instructions.

By moving branch execution to the OF stage, the branch dealy is reduced to one cycle. In the R1000 the branch penalty is further reduced by always executing the instruction immediately after the branch. It is up to the compiler to fill this slot with an useful instruction, or with a no-op if it cannot find a useful instruction. It is questionable if making such an implementation issue visible at the architectural level is good design. The superscalar R10000 processor does not need a branch delay slot, but it is kept for binary compatibility reasons. Furthermore, moving branch execution to the OF stage means that an ALU instruction followed by a branch that uses the result of the ALU instruction

incurs a stall of one cycle.

Moving branch execution to the OF stage can have a negative impact on the cycle time. Since the register file is read during the second half of the cycle, the time to read the register file plus the time needed to calculate the branch condition must me less than a half cycle. Maybe this did not affect the cycle time of the R1000, but it may affect the cycle time of other processors implemented in other technologies. So, the designer might ask: should I move branch execution to the OF stage or should I perform it in later stages, even if this means that the penalty of a taken branch is increased. This too can be determined using μ-architectural simulations and cycle time estimated, provided the simulator allows to specify when branch execution takes place.

The five-stage pipeline described above is a conventional pipeline. Contemporary processors use a deeper pipeline in order to achieve higher clock rates. This is also called *superpipelining*. The pipeline of the MIPS R4000 is a good example. [6].

Because cache accesses took considerably more time than either a register file access or an ALU operation, the designers of the R4000 decided to break up the IF and MEM stages into several stages. The IF stage is split into the following two stages:

IF First half of instruction fetch.

IS Second half of instruction fetch. Instruction cache access completes.

The MEM stage, on the other hand, is split into three stages:

DF First half of data cache access.

DS Second half of data fetch. Data cache access completes.

TC Tag check. Determine whether the data cache access yielded a hit.

Note that in order to be able to split cache accesses into several stages, the caches must allow pipelined cache accesses.

Deeper pipelines increase the load as well as the branch delays. The load delay of the R4000 is two cycles because the value loaded is available at the end of the DS stage. If the subsequent tag check indicates a miss, the pipeline is stalled until the data is available. The branch delay is three cycles, because branch condition evaluation is performed during the EX stage.

Summarizing, there are many μ-architectural design choices. However, we are not aware of a flexible, parameterizable simulation system that allows to evaluate these design choices. The simulator presented in the next section allows to configure μ-architectural features such as the number of pipeline stages, the stage in which branch execution occurs, whether forwarding is performed, etc.

3 Related Work

The SimpleScalar toolset [1] is probably the most often used simulation system for computer engineering studies. For example, many papers appearing at the International Symposium on Computer Architecture use SimpleScalar to verify the proposed techniques. SimpleScalar actually consists of several simulators of which `sim-outorder` is the most advanced. It allows to vary many processor and memory parameters such as the issue width (a single-issue, pipelined processor can be simulated by specifying an issue width of one), the window or Register Update Unit size, and the cache miss penalties. However, several micro-architecural features, in particular the pipeline organization, are

fixed and can only be changed by modifying the code substantially. For example, `sim-outorder` cannot be used to evaluate current deeply pipelined superscalar processors.

In [3], the authors present an analytical model based on which they want to determine the optimal pipeline depth for different sets of applications. They not only focus on the traditional SPEC benchmark but also include more modern applications such as web based and database applications. By They constructed a simulator based on the analytical model on which these workloads were then executed varying the pipeline depth. When not distinguishing between the two sets of applications, they found a Gaussian distribution of the pipeline lengths. When separating between the SPEC and 'modern' applications, it is found that the latter category requires a much deeper pipeline than the SPEC applications.

In [8], the relationship between performance and pipeline depth is explored. They show that the branch misprediction latency is the main cause of performance loss in deep pipelines. They also show that as the pipeline gets deeper, there is increasing pressure on the memory system, requiring larger on chip caches. When increasing the pipeline depth and cache size, a performance improvement of 35 to 90% can be achieved. However, increasing the pipeline depth implies the use of increasingly complex algorithms and more accurate timing tools. More accurate architectural simulators are required in order to validate the impact of such choices and fine tune the architecture.

4 The Simulator

This section describes our simulator Sketch.

The first and most important parameter that can be configured is the number of pipeline stages. Specifically, each stage in the conventional pipeline can be split into an arbitrary number of substages. To avoid confusion, the stages in the conventional pipeline will be referred to as *superstages*. The simulator takes the following command-line arguments which specify the number of substages in each superstage:

if Number of substages in the IF superstage.

of Number of substages in the OF superstage.

ex Number of substages in the EX superstage.

mem Number of substages in the MEM superstage.

wb Number of substages in the WB superstage.

By default, the number of substages in each superstage is one, corresponding to the conventional pipeline. In order to simulate the R4000 pipeline one has to specify

```
p3 -if=2 -mem=3 test
```

The current simulator cannot simulate pipelines that deviate significantly from the conventional pipeline. For example, it cannot simulate register-memory architectures such as the x86/IA-32 where one operand of an instruction can be a memory location. This is because for such architectures the EX and MEM superstages need to be replaced by an address stage that computes the effective address, a memory stages that loads the source operand from memory, and an execute stage. We remark, however, that the Pentium translates complex instructions to RISC-like μ-operations.

By default, branch execution takes place during the first substage of the MEM superstage.

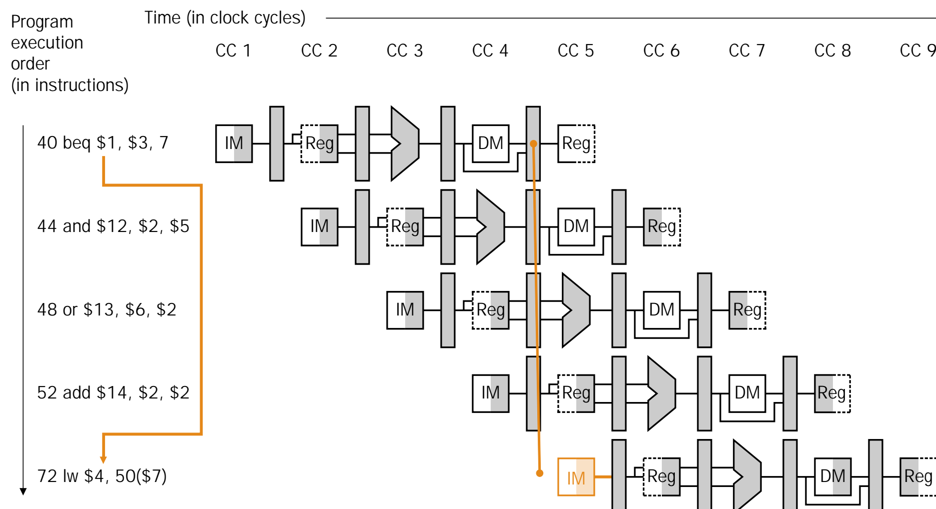


Figure 4: Branch delay

As illustrated in Figure 4 [4], this means that the branch delay is equal to the number of substages in the IF and OF superstages. One can specify that branch execution takes place during an earlier substage by specifying

```
p3 -branch=3.1 test
```

This specifies that branch execution occurs during the first substage of the third superstage (the EX) stage. The earliest substage branch execution can be performed is the last substage of the OF superstage.

After a branch the simulator continues fetching the next instructions in static program order until the branch is resolved. This is sometimes referred to as predict not taken. Not taken branches, therefore, do not occur a branch penalty while taken branches do. The current simulator does not implement delayed branches, in which the instruction(s) after the branch are always executed. If a branch instruction is taken, all instructions preceding it in the pipeline are flushed. To evaluate delayed branches, a parameterizable code reorganization tool would also be needed. Another extension we plan to incorporate in future versions of the simulator is branch prediction.

By default, the simulator does not perform forwarding. One can specify that forwarding should be performed and the number of data hazard stall cycles using the command-line argument

```
-dhs=<n>
```

where `<n>` is the number of cycles between the time a result is produced and the time it can be consumed. For most processors, `n=0`, meaning that the result of an arithmetic operation can be forwarded immediately after the last substage of the EX superstage and that the result of a load can be forwarded immediately after the last substage of the MEM superstage. However, as argued in Section 2, adding a forwarding unit may increase the cycle time, so it might be advantageous to allow one cycle for comparing if one of the source registers of the instruction currently in the first substage of EX is equal to the destination register of an instruction later in the pipeline. It is assumed that the result of a load can be forwarded from the last substage of the MEM superstage onwards. Forwarding from a earlier substage onwards and roll-back in case of a miss, as in the R4000, is not yet implemented.

```
Kernel 1: VADD
implemented as "v-add.s"
addi  $3, $0, 100
addi  $4, $0, 400
addi  $5, $0, 700
addi  $1, $0, 0
addi  $2, $0, 64
lw    $6, 0($3)
lw    $7, 0($4)
lw    $8, 1($3)
lw    $9, 1($4)
lw    $10, 2($3)
lw    $11, 2($4)
lw    $12, 3($3)
lw    $13, 3($4)
addi  $3, $3, 4
addi  $4, $4, 4
add   $6, $6, $7
add   $8, $8, $9
add   $10, $10, $11
add   $12, $12, $13
addi  $1, $1, 4
sw    $6, 0($5)
sw    $8, 1($5)
sw    $10, 2($5)
sw    $12, 3($5)
addi  $5, $5, 4
bne   $1, $2, -21
stop
```

```
Kernel 2: VSUM
implemented as "v-sum.s"
addi  $1, $0, 0
addi  $4, $0, 100
addi  $3, $0, 0
addi  $2, $0, 64
addi  $1, $1, 1
lw    $5, 0($4)
addi  $4, $4, 1
add   $3, $3, $5
bne   $1, $2, -5
stop
```

```
Kernel 3: SAD
implemented as "sad.s"
addi  $4, $0, 100
addi  $5, $0, 500
addi  $10, $0, 1
addi  $1, $0, 0
addi  $2, $0, 0
addi  $3, $0, 64
lw    $6, 0($4)
lw    $7, 0($5)
addi  $4, $4, 1
addi  $5, $5, 1
addi  $2, $2, 1
sub   $8, $6, $7
slt   $9, $8, $0
add   $1, $1, $8
bne   $9, $10, 2
sub   $11, $1, $8
sub   $1, $11, $8
bne   $2, $3, -12
stop
```

Figure 5: The kernels used for simulation

We remark that the current version of the simulator is a proof of concept. There are many features that have not yet been implemented. For example, the current simulator does not simulate a memory hierarchy: it is assumed that all memory accesses hit the cache. However, the cache hit ratio or the lack of a cache (since many embedded processors do not have a cache) infuences the optimal pipeline depth, since the memory latency may be hidden by splitting the IF and MEM superstages into many substages.

5 Usage

We use the simulator to perform experiments with three synthetic benchmarks: vector addition, vector summation, and sum of absolute differences. These kernels are representative for data parallel loops, reduction operations, and benchmarks containing many hard to predict branches, respectively. The assembly codes are given in Figure 5. Three experiments has been performed with the simulator, which are: varying the number of substages of Execution superstage, moving branch execution from MEM to EXE to ID/OF, and register file forwarding on/off. For each experiment, cycles of running the kernels are tested. What we do in each experiment and the corresponding results are given as follows.

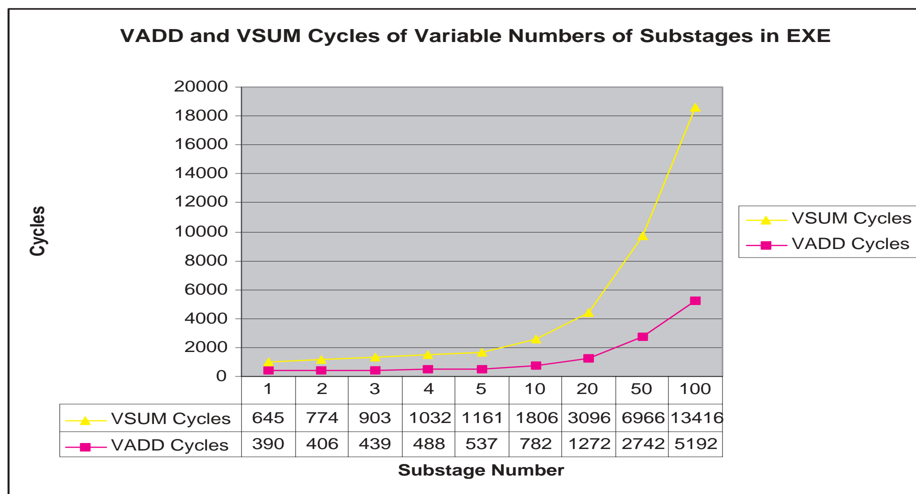


Figure 6: Experiment 1: Number of EXE substages VS Cycles

5.1 Experiment 1

In this experiment, we vary the number of substages of Execution(EXE) superstage. To change the number of substages of EXE superstage, the number can be assigned in command line, using ”-e=the number”.

For vector add(VADD) that contains few data dependencies (provided it is unrolled a couple of (four) times), the extra cycles will be few initially(first time: 16, second time: 33) and more later(after the number is bigger than 4, increase by 49 each time). For vector summation(VSUM) contains a true dependence (unless one performs ”scalar expension”) so its execution time will grow substantially. The test results and curves drawn based on results are depicted in Figure 6.

5.2 Experiment 2

In the second experiment, we move branch execution from MEM to EX to ID/OF. By default, branch is executed in MEM stage(in command: -b=4.1). If branch executes in EXE or ID/OF, we should change the value of ”-b” to ”3.1” or ”2.1” respectively.

For vector addition, the gain is small (contain only one branch per loop iteration and no data dependence.). For vector summation and SAD, the gain are large. It is likely that moving branch execution from MEM to EX will not affect the cycle time, so this will always improve performance. However, moving it to the ID/OF stage may lengthen the cycle time, because the the registers need to be fetched and the branch condition evaluated in the same cycle. As can be seen in Figure 7, moving branch execution to the OF stage reduces the number of cycles taken by the VSUM and SAD kernels substantially (19.5% and 18.8%), so execution time will be reduced provided the cycle time is not increased by 19.5% and 18.8%. However, the number of cycles taken by the VADD kernel is not reduced substantially, by 7.6%. For this kernels moving branch execution to the OF stage is advantageous only if a register file access takes less time than a machine cycle.

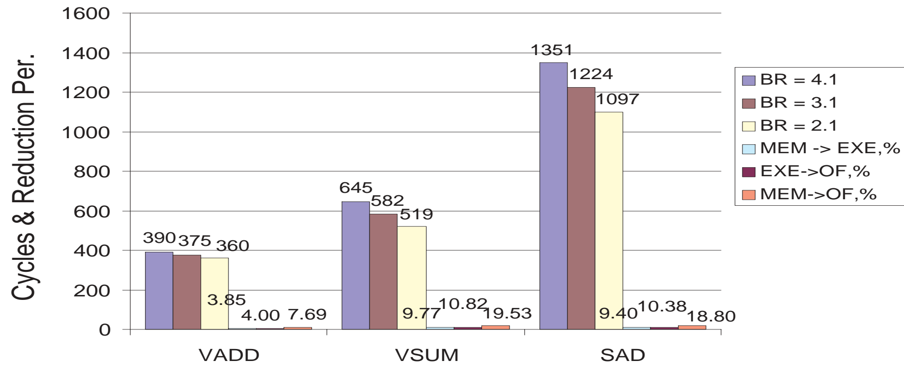


Figure 7: Experiment 2: Branch execution VS cycles

5.3 Experiment 3

In the third experiment, the test of register file forwarding on/off is performed. It is supposed to be worthwile if there are many dependencies over a short distance (in other words, if we cannot find three independent instructions between a definition and a use). For VSUM and SAD, there are more than one data dependencies. Turning forwarding on will reduce cycle times(9.92For VADD, there is no data dependence. So forwarding has nothing to do with this case. Test results are given in Table 1.

	ForwardingOFF Cycles	ForwardingON Cycles	Perc. of Cycle reduction
VADD	390	390	0
VSUM	645	581	9.92%
SAD	1351	1223	9.47%

Table 1: Experiment 3: Register file forwarding effect

However, there is an exception. Writing to the register file during the first half of a cycle and reading from the register file during the second half implies that a register file access can take at most half a machine cycle C. If a register file access takes more time, for example $0.6C$, then register file forwarding will decrease the performance of all kernels.

6 Conclusions and Future Work

We have presented a flexible, parameterizable simulator of pipelined processors. As for load/store RISC architectures, the simulator allows to configure several (micro-)architectural features such as the pipeline depth (meaning default five pipeline superstages can be divided into arbitrary substages), the stage in which branch execution occurs, whether or not register file forwarding is performed, and the number of branch delay slots. We used the simulator to perform experiments with three synthetic benchmarks: vector addition, vector summation, and sum of absolute differences. These kernels are representative for data parallel loops, reduction operations, and benchmarks containing many hard to

predict branches, respectively. With the help of our simulator, the cycle times of running each kernel are given. And the optimal pipeline configuration for each kernel can be decided.

We remark that the current version of the simulator is a proof of concept. There are many features that have not yet been implemented. For example, the current simulator does not simulate a memory hierarchy: it is assumed that all memory accesses hit the cache. Forwarding from an earlier substage onwards and roll-back in case of a miss, as in the MIPS R4000, is not yet implemented. These additional features may lead to new research topics and will improve our simulator in the future.

References

[1] Doug Burger et al. The SimpleScalar Architectuaral Research Toolset, Version 2.0. http://www.cs.wisc.edu/ mscalar/simplescalar.html, 2003.

[2] Jeff Ringenberg et al. The SimpleScalar-Arm Power Modeling Project. http://www/eecs.umich.edu/ jringenb/power, 2003.

[3] Puzak Hartstein. The Optimum Pipeline Depth for a Microprocessor. In *Proc. 29th Annual International Symposium on Computer Architetecture(ISCA02)*, 2002.

[4] John L.Hennessy and David A.Patterson. *Computers Organization & Design: the Hardware/Software Interface*. Morgan Kaufmann, 2ed edition edition, 1998.

[5] John L.Hennessy and David A.Patterson. *Computers Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition edition, 2003.

[6] S. Mirapuri, M. Woodacre, and N. Vasseghi. The Mips R4000 Processor. *IEEE Micro*, 12(2):10–22, 1992.

[7] J.C. Park, V. Mooney, K. Palem, and K.W. Choi. Energy Minimization of a Pipelined Processor Using a Low Voltage Pipelined Cache. In *Proc. 36th Annual Asilomar Conference on Signals, Systems and Computers*, 2002.

[8] Carmean Sprangle. Increasing Processor Performance by Implementing Deeper Pipelines. In *Proc. 29th Annual International Symposium on Computer Architetecture(ISCA02)*, 2002.