Eindhoven University of Technology

MASTER

The design of a simulator generator for parallel architectures

Waucomont, M.J.P.H.

Award date:
1993

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

ED 409
7065

The Design of A Simulator Generator for Parallel Architectures

M.J.P.H. Waucomont

Department of Electrical Engineering
Digital Information Systems Section
Eindhoven University of Technology

October 25, 1993

Coach: ir. W.J. Withagen
Supervisor: prof. ir. M.P.J. Stevens

The department of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of student projects and graduation reports.

Abstract

This paper reports the research into enhancing the existing sequential simulator generator SIMGEN (and the accompanying sequential processor description language SIMDES) to accomodate the simulation of parallel processors. In the context of this report, 'parallel' refers to instruction level parallelism within a single processor. The research deals with finding a new model and the impact of this model on the description language.

Adapting the simulator generator requires finding a new simulation model. In order to be able to simulate architectures with instruction level parallelism, the model must incorporate implementation details such as pipeline registers and control. Control consists of a control unit which resembles the control unit of a sequential implementation plus additional units like a hazard detection unit, and a forwarding unit. It also contains provisions to cope with control hazards as well as interrupts.

Research on the description language has shown that the modification has a severe impact on the nature of the description language. For sequential simulation the description of the behavior of the instructions plays the key role. Description of the behavior of the instructions suffices to describe the behavior of a sequential processor. For parallel processors the behavioral description of the instructions must be replaced by a description of the behavior of the implementation. This requires descriptive constructs to describe the behavior of the pipe stages and the control units.

This report describes the complications involved in describing and simulating high performance pipelines. First the theory of pipelining is presented. Then, the hardware required is shown at an appropriate level of abstraction. Finally, the problem space is narrowed down by choosing a simplified example in order to be able to present a definition of the description language.

Contents

Chapter 1

Introduction

"Simulare certe est hominis"
(To simulate is certainly a human trait)
Terentius, Adelphi 734

1.1 Microprocessor Simulation

Over the past years simulation of microprocessor architectures has become increasingly important. Designers of microprocessors want to be able to evaluate the performance of an architecture prior to its implementation. This evaluation can be done at different levels of abstraction.

1.2 SIMPROACH

One of the current research projects at the Digital Information Systems Section of the department of Electrical Engineering at the Eindhoven University of Technology is the SIMulator design system for PROcessor ArCHitectures (SIMPROACH). SIMPROACH will become a toolkit that can be used to obtain quantitative information about the performance of different microprocessor architectures. In SIMPROACH three levels of abstraction are distinguished. Increasing level numbers indicate an increasing level of detail and a decreasing level of abstraction.

Level 0 is represented by the *Architecture Workbench* [BCFZ89, Mul90], a processor performance analysis system developed at the Stanford University Computer Science Laboratory. At this level only the elementary processor characteristics are described. Slight alterations of these characteristics immediately translate into changes of the simulation results. This tool is used to get a first impression of the performance of an architecture.

Level 1 is used to model the Instruction Set Architecture (ISA). At this level there is minimal information regarding the implementation of the processor. Its behavior is specified by its instructions and the resources on which it operates, like registers and memory. The result of the research done so far is a sequential instruction set simulator generator [Tak92]. Takken's work uses a description of the architecture to produce a simulator for it.

Level 2 will be used to model the processor implementation. At this level either a more detailed behavioral or even a structural simulation using a hardware description language like VHDL or IDaSS will be provided.

Currently, little research regarding Level 2 has been done. The research described in this report focusses on the possibilities of simulating processors with instruction level parallelism at Level 1 by adding a little bit more information about the implementation to the sequential description.

1.3 Level 1

Distinguishing different levels of abstraction is *the* way to cope with the complexity of the design of microprocessors. The Instruction Set Architecture or ISA is a very important interface between the levels of abstraction. It is the interface between the hardware and the low-level software. Often the ISA is defined as the abstraction or definition of the system as seen by a machine language programmer or compiler writer. It is considered to be the definition of the conceptual structure and the functional behavior of a processor as opposed to factors such as the processor's logic design and circuit technology.

The approach most often found in literature to gather information at this level of abstraction is to describe the architecture in a description language [MJ76, Cor81, KT87, Tak92]. Either the description is interpreted or a simulator for the architecture is generated. In the case where a simulator is generated, it bears

a model of the processor in it that is chosen by the designer of the simulator generator, and not by the designer of the architecture. This type of simulator thus has a fixed number of execution phases, for example Instruction Fetch (IF), DEcode (DE), and EXecute (EX). This division into phases limits the designer's freedom in taking decisions.

Sequential simulators use this three stage cycle: IF, DE and EX. The phases are executed sequentially and repeatedly, until either the end of the code to be simulated is reached or program execution is interrupted to monitor the simulation progress. Executing IF, DE and EX simulates the behavior of exactly one instruction.

In the instruction fetch phase the instruction is fetched from the simulator's memory. In the decode phase the instruction type is determined and the appropriate syllables are extracted. There are many ways to recognize the different instruction types. Takken discusses different instruction decoding strategies in his Master's thesis [Tak92]. In the execute phase the behavior of the instruction is simulated by calling the appropriate function.

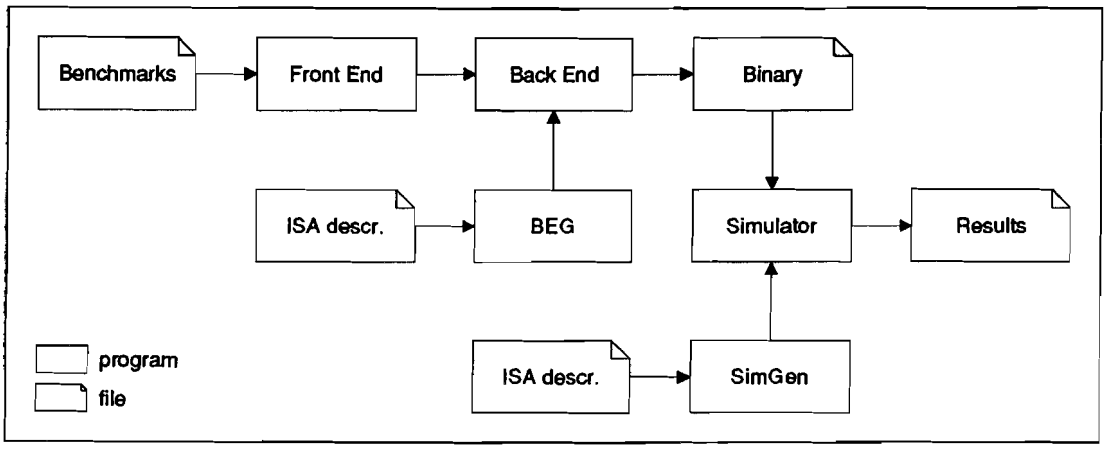

Figure 1.1: Overview of the components of Level 1.

Takken recently finished a simulator generator. It generates simulators that work according to the principles just described. The simulator generator is called SIMulator GENerator (SIMGEN). It uses an input language called processor SIMulation DEScription language (SIMDES). The output is C code that implements the architecture as well as a mechanism to gather statistics.

On Level 1 of SIMPROACH the information regarding the performance of an

architecture is gathered by simulating instructions on such a generated simulator. Benchmarks are compiled to obtain executables consisting of a sequence of instructions to be simulated. The term 'benchmarks' should be taken in the broadest sense of the word. A benchmark is a set of programs designed to get an overall impression of the performance of a processor. The compiler back end is generated by a so called Back End Generator (BEG) which, just like the simulator generator, has a description of the architecture as its input. Right now these two descriptions are different, but for future versions a generic architecture description for the simulator generator as well as the back end generator will be designed.

At the time of this writing, an ANSI C front end for the compiler is finished. SIMGEN is being tested, with SIMDES getting a lot of attention. Several processors are being modelled in this language to determine the usability of the language rather than to test the suitability of SIMGEN.

1.4 A Parallel Simulator Generator at Level 1

The rest of this report describes the research done to investigate the possibilities to develop a new simulator generator for SIMPROACH, to be called Parallel Simulator Generator (PSIMGEN), and a new description language, Parallel Simulation Description Language (PSIMDES). The simulators generated by PSIMGEN will be used to gather information about the parallel execution of instructions, such as Clocks Per Instruction (CPI), information about pipeline stalls, etc. In the context of SIMPROACH this is the logical next step after the simulators generated by SIMGEN, which only perform a purely functional simulation of instructions and provide information like instruction count, register usage, etc.

In order to be able to simulate parallel architectures an appropriate model of a processor has to be developed. In this report, first an indication of the level of abstraction is given. Much attention is paid to the theory of pipelining. Then, an indication is given of the amount of implementation detail required to correctly model pipelined processors.

This information is used to complete a detailed design of a model and a description language for a simplified example. This can be a guideline for a possible implementation, but it also indicates how rapidly the complexity of the model and the language increase when more detail is added.

Chapter 2

Levels of Abstraction

"People here don't know the art of leveling"
Bruce Watson, Stratums Eind, 1992

An important complication in defining a suitable model for the simulation of a processor with instruction level parallelism, is to reach the right level of abstraction within the limits set by Level 1. This chapter shows two different abstraction levels at which a processor can be viewed. Both are extremes. For ease of explanation a sequential implementation is used, but the line of reasoning can be extended to the more complicated case of an implementation with instruction level parallelism.

2.1 An Extremely Abstract Model

The von Neumann architecture (Figure 2.1) is the predominant model of computer architectures. There is one read-write memory which contains both data and instructions. This memory is addressable by location. The Central Processing Unit (CPU) or processor executes instructions from consecutive locations unless an instruction modifies the control flow explicitly.

Von Neumann's model therefore is strictly sequential. A simulator based on this model fetches an instruction, decodes it, and finally applies the appropriate operations on the resources—registers, memory, and I/O ports. All instructions

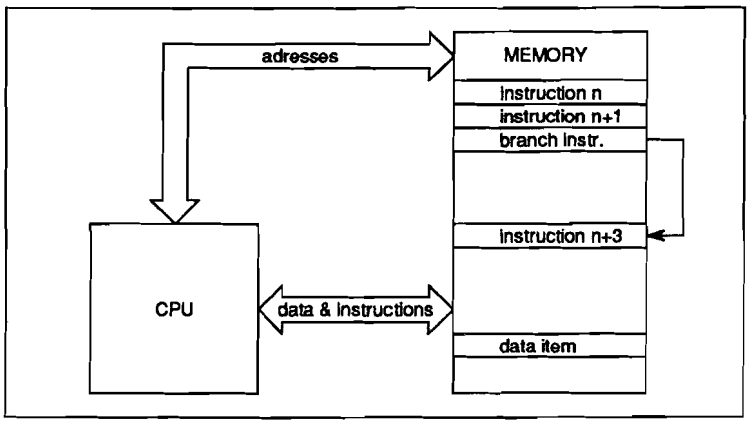

Figure 2.1: The von Neumann architecture.

are executed sequentially in the same manner. This describes the behavior of the processor. Basically, this is the model as used by Takken [Tak92].

The information about the architecture that is required to generate a simulator of this type can be split into three categories. The first category describes the resources. The second category describes the layout of the instruction types, so they can be recognized by the decode portion of the simulator. The last category provides a functional description of the behavior of each instruction type. This description is used in the execute section to simulate the instruction's behavior.

The von Neumann model results in simulators that stress the importance of the instruction set. More exactly, the importance of the functional specification of the effect of the instructions. The model itself is implicitly present in the simulator. The input of the simulator generator consists mainly of the semantics of the instructionset.

2.2 An Implementation Oriented Model

The processor is the part that executes the instructions. It consists of a control unit and a datapath. The datapath is the part of the processor through which data flows. It includes functional units such as an Arithmetic and Logic Unit (ALU) as well as registers that may contain data, all organized around buses. A bus is a shared path between registers and/or functional units.

The control unit controls the actions of the datapath during every clock cycle of the execution of any instruction. The behavior of the control unit can be described by a finite state machine (with output—as in a Mealy or a Moore machine), which changes state each clock cycle. Operations to be performed are associated with the states. Each instruction takes several clock cycles to complete. If we consider the processor to be a finite state machine, the contents of the datapath determines the state the processor is in.

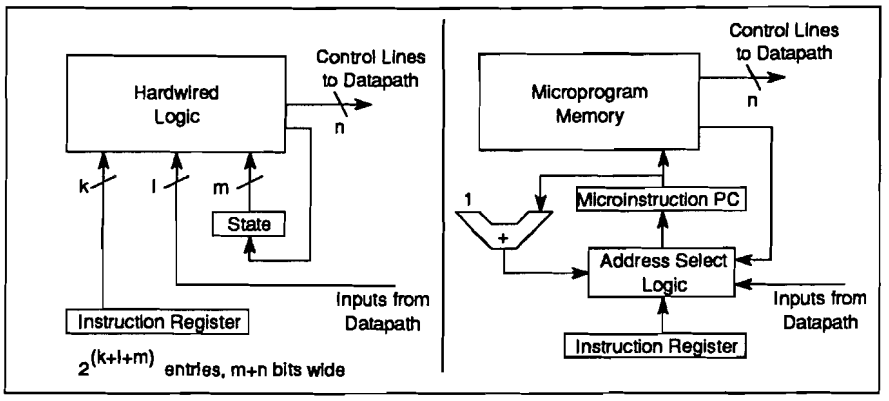

Figure 2.2: Hardwired versus Microprogrammed Control.

There are two major techniques for implementing the control unit. The first is called hardwired control. In this technique the opcode field of the current instruction is fed into the hardwired control, together with the previous state and some control inputs from the datapath. This results in a new state and in control signals that tell the datapath how to manipulate the data. Hardwired controls tend to be large. Straightforward implementation would require a table of several megabytes of Read Only Memory (ROM). Fortunately this is in general a sparse table, so its size can be reduced by keeping only the rows with unique information. This approach increases the complexity of the address decoding logic. The implementation is known as a Programmed Logic Array (PLA).

Further reduction of hardware requirements can be achieved by using computer-aided design programs to minimize the number of minterms. Yet another thing that has to be taken in account here is that the size of the PLA also depends on the assignment of ordinal numbers to the states. There are computer-aided design programs that help to assign similar state numbers to states that perform similar operations. This reduces the size of the PLA considerably. Lastly, the instruction bits are also inputs to the control PLA. Just like the numbering of states, the selection of appropriate opcodes affects the cost of control.

The second technique is called microprogrammed control. Essentially the control unit is implemented as a miniature computer with its own instruction set. It consists of a table that specifies the control of the datapath and a second table that determines the control flow at the micro level. Thus, the microinstructions—called this way by their inventor M. Wilkes in 1949 [Wil69]—specify all the control signals for the datapath, plus the ability to decide which microinstruction should be executed next.

In this case the control unit is an interpreter for the instruction set, written for the microarchitecture. The structure of the resulting microprogram resembles the state diagram that describes the processor very closely. For each state in the diagram there is one microinstruction.

The brief descriptions of these two ways for implementing control indicate that choosing a model that is very close to the hardware results in simulators that are both extremely laborious to describe as well as very difficult to simulate. This is not very useful in the context of simulation at SIMPROACH Level 1. To obtain a more useful model of the hardware a higher level of abstraction is required. This is not only true for the sequential architecture described in this section, but also for architectures with instruction level parallelism.

Chapter 3

The Theory of Pipelining

"Fallacy: pipelining is easy."
Hennessy & Patterson [HP93]

It is clear that the von Neumann model presented in the previous chapter is not an appropriate model to describe architectures with instruction level parallelism. To be able to describe such architectures information about the parallel implementation has to be added. But too much detail has to be avoided since it leads to overwhelming descriptions as well as extremely complex simulators.

Before trying to determine the amount of implementation detail that needs to be incorporated in the model, the technical evolution that has led to processors with instruction level parallelism deserves a closer look. This will give a better understanding of the complications involved in designing a parallel implementation. At the end of each section that covers one of these complications, its impact on the model is treated briefly.

3.1 Pipelining

The performance of a system based on the von Neumann model can be increased by speeding up the clock. This can be done up to the point where one of the components of the system fails. This component determines the maximum performance of such a system.

The performance can also be increased by partitioning the instruction execution in a number of phases. Overlapping these phases for successive instructions ensures that the limiting component is always busy. This leads to a dramatic improvement of the performance without the risks that come with only speeding up the clock. Partitioning the instruction execution reduces the complexity of the tasks to be accomplished in one clock cycle. A nice side-effect is that the clock-speed easily *can* be increased. This leads to an even bigger performance increase. The technique of overlapping multiple instructions in execution is called **pipelining**.

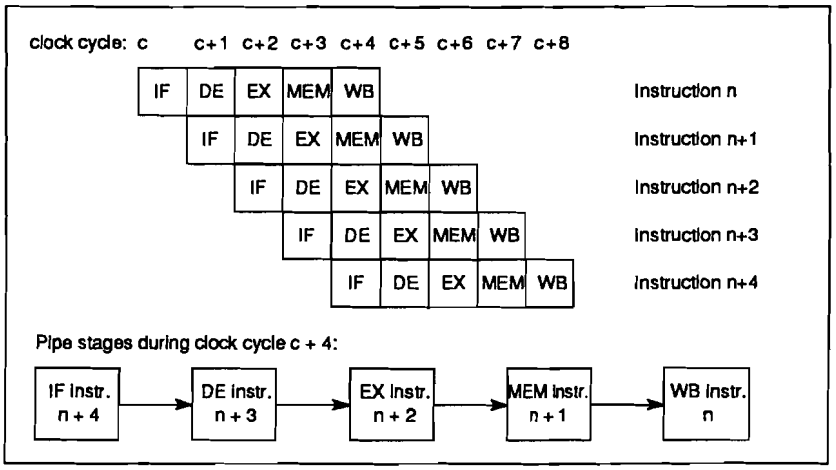

Figure 3.1: Pipelining: overlap in time keeps all stages busy.

A pipeline can be described as a collection of processing stages through which the information flows. Each stage performs part of the processing as dictated by the way the task is partitioned. The result of the computation of each stage is transfered to the next stage in the pipeline. The final result is obtained after the data has passed through all stages. The resulting simplified model of a pipelined processor can be seen in the bottom half of figure 3.1.

Pipelined processors try to start a new instruction every machine cycle. A phase in the execution, depicted by a processing stage in figure 3.1, is called a **pipe stage** or a pipe segment. An instruction stays in a stage for the duration of one machine cycle before being moved to the next stage. On most pipelined processors a machine cycle is equal to one clock cycle, but a multiphased clock can be used.

3.2 Performance of Pipelined Processors

SIMPROACH is intended to gather information about the performance of pipelined processors. Three characteristics are used when comparing CPU performance:

- Instruction count.
- Duration of the clock cycle.
- Average number of clock cycles per instruction (CPI).

The instruction count depends on the instruction set architecture and the compiler technology. The latter is an often overlooked aspect which might even deserve its own place in the enumeration of characteristics. It could be a weak point of SIMPROACH. The quality of the compiler influences the simulation results. The back end generator plays an important role here, because the quality of the generated executables depends heavily on the quality of the back end. The quality of the back end depends on how well the back end generator is tuned for the architecture—normally, back end generators make assumptions about the kind of architecture they are going to be used for. For now, it is assumed that this is not a major objection.

The duration of the clock cycle depends on hardware technology and organization. There is not much to be said about this in the context of SIMPROACH other than that there exists a clock—or a two phase clock if necessary—which is used to clock the system. The number of elapsed clock cycles can be used to compute the CPI.

CPI is defined as the number of clock cycles required to complete a program, divided by the instruction count for that program. The CPI for the ideal pipeline from figure 3.1, CPI_i, can be expressed in the CPI for the non-pipelined version CPI_n and the Pipeline depth PD.

$$CPI_i = \frac{CPI_n}{PD}$$

By pipelining a processor a large **speedup** can be achieved. Speedup is defined as the ratio of the average instruction time AIT_n on a non-pipelined processor over the average instruction time AIT_p on the pipelined version of the processor.

$$\text{Speedup} = \frac{\text{AIT}_n}{\text{AIT}_p}$$

Using

$$\begin{cases} \text{AIT}_n = \text{CPI}_n \times \text{CT}_n \\ \text{AIT}_p = \text{CPI}_p \times \text{CT}_p \end{cases}$$

where CT_n and CT_p are the clock cycle time for the non-pipelined processor and the pipelined processor respectively, yields

$$\text{Speedup} = \frac{\text{CPI}_n}{\text{CPI}_p} \times \frac{\text{CT}_n}{\text{CT}_p}$$

Substituting the formula for the ideal CPI in order to remove CPI_n from the equation yields

$$\text{Speedup} = \frac{\text{CPI}_i \times \text{PD}}{\text{CPI}_p} \times \frac{\text{CT}_n}{\text{CT}_p}$$

Finally, CPI_p can be removed by substituting

$$\text{CPI}_p = \text{CPI}_i + \text{PSC}$$

where PSC is the average number of pipeline stall cycles. (Stalls will be explained in the next sextion.) This yields

$$\text{Speedup} = \frac{\text{CPI}_i \times \text{PD}}{\text{CPI}_i + \text{PSC}} \times \frac{\text{CT}_n}{\text{CT}_p}$$

The factor CT_n/CT_p expresses the potential increase in clock rate due to pipeline overhead. In practice, this factor is approximately 0.9 [Sme91]. The factor $(\text{CPI}_i \times \text{PD})/ (\text{CPI}_i + \text{PSC})$ is equal to $\text{CPI}_n/\text{CPI}_p$. In the ideal case this factor is equal to the number of pipe stages PD. In practice, the maximum speedup which can be achieved is within thirty percent of PD.

The fact that the theoretical maximum for speedup cannot be reached is due to pipeline overhead as well as to situations that lead to pipeline stalls—that is, situations in which the pipeline is not completely filled. Since SIMPROACH is intended to get an impression of the performance, care has to be taken to describe and simulate those situations.

3.3 Pipeline Hazards

Figure 3.1 gives an *impression* of what is going on in a pipelined processor. But it is an extremely simplified model. The most eminent simplification is that it assumes that the pipeline is always filled. In reality this is not true. By overlapping the execution of instructions their relative timing has changed. This leads to dependencies. These dependencies can lead to situations in which the next instruction in the instruction stream cannot be executed. These situations are called pipeline **hazards**. Hazards are the reason why high performance pipelines are hard to design. There are three classes of hazards.

1. *Structural hazards* occur when there are not enough resources available to allow simultaneous overlapped execution of all possible combinations of instructions.
2. *Data hazards* occur when an instruction needs the results of a previous instruction and these results have not yet been written back.
3. *Control hazards* occur when pipelining instructions that change the Program Counter.

A possible solution for hazards is to delay the instructions following the instruction that causes the hazard. This is known as **stalling** the pipeline. As can be seen in the formula for speedup, stalls result in lower performance. The more detailed description of hazards that follows shows solutions that can reduce the number of stalls and thus increase performance.

3.3.1 Structural Hazards

A structural hazard can be caused by a single instruction or a combination of instructions that demands more resources than those that are available.

A single instruction can cause a structural hazard if it differs extremely from the instructions for which the pipeline was designed. If for example all instructions have two source operands and one destination operand and there would be one instruction which needs three source operands, this would lead to a structural hazard. A possible solution, at the cost of extra hardware, would be to design the processor to be able to read three operands in one cycle instead of two.

Most of the time structural hazards are caused by a combination of instructions which require the same functional unit. This is often caused by a functional unit which is not replicated often enough or is not fully pipelined. A good example for such a unit is the floating point divide unit. Pipelining such a unit would be very expensive in terms of hardware. On the other hand, considered the relatively low frequency of floating point divisions, replicating this unit a number of times would also be too expensive.

Accesses to memory are also a candidate for structural hazards. If for example instruction n in figure 3.1 does a memory access in clock cycle $c+3$, the instruction fetch of instruction $n+3$ will have to be delayed to clock cycle $c+4$. Using dual ported memory would eliminate this problem, but it is expensive.

A last example of a situation where a structural hazard occurs is an architecture with a register file with only one write port. Under some circumstances, the pipeline might want to perform two writes in one clock cycle. The cheap solution is to stall one of the instructions if this situation occurs. But this stall could be avoided altogether if an extra write port would be added to the register file.

The Model

In the ideal situation—ideal in terms of performance—functional units are fully pipelined, duplicated as often as necessary, and register files and memory have as many ports as necessary.

Assuming ideal conditions is not realistic. In practical designs the number of pipeline stalls due to structural hazards is not zero, thus eliminating the effect of structural hazards on the performance figures.

3.3.2 Data Hazards

Pipelining changes the relative timing of the instructions, which leads to a different access pattern to the operands than one would expect from examining the program code. This results in data hazards. There are three types of data hazards: Read After Write (RAW), Write After Read (WAR) and Write After Write (WAW). Consider two instructions i and j, with i occurring before j. The most common type of hazard is the RAW hazard. It occurs when j tries to read a source before i writes it. In this case, j incorrectly reads the old value.

WAR hazards occur when j tries to write a destination before it is read by i. In this case i incorrectly gets the new value.

A WAW hazard occurs when j tries to write an operand before it is written by i. This way the writes are performed in the wrong order, incorrectly leaving the value written by i in the destination. This type of hazard can only happen in pipelines that write in more than one pipe stage.

One way to deal with data hazards is to just accept the fact that they exist and forbid the compiler to generate code sequences with dependencies. This means that the compiler has to insert independent instructions between instructions which are dependent and would thus lead to conflicts. If no such instructions are present in the instruction sequence to be scheduled, `nop` instructions have to be inserted. `Nop` instructions do not do anything but consume a clockcycle—hence the name, 'no operation'—and therefore are always independent. This approach is called static scheduling. The drawback of static scheduling is that the `nop` instructions occupy clock cycles, but do not do anything useful.

Another way is to stall the instructions in the pipeline until the hazard is resolved. This strategy requires additional hardware to detect the hazard and to stall the pipeline. This piece of hardware is called a **pipeline interlock**.

Using a pipeline interlock to detect hazards and stall the pipeline guarantees correct execution of dependent instructions without forcing the compiler to resolve the dependencies. But with static scheduling as well as pipeline interlocks the cost of correctness is lower performance.

Some stalls resulting from data hazards can be handled by **forwarding** the result from the output of one unit to the input of the unit that needs the result. Forwarding requires extra hardware. Figure 3.2 shows how forwarding can be used

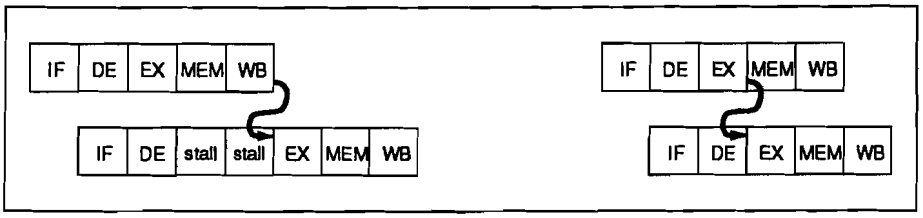

Figure 3.2: Data hazards resolved by stall versus forwarding

to eliminate a stall. The information needed to execute the second instruction is available after the execute part of the first instruction has been completed, but will not be written back until the write back stage. Instead of waiting two cycles, the information can be forwarded to the execute stage of the second instruction.

An advanced technique using special hardware to handle data hazards is dynamic scheduling. The special hardware rearranges the instruction execution to reduce stalls. Dynamic scheduling reduces compiler complexity at the cost of a significant increase of hardware complexity. The latter observation once again stresses the tight entanglement of the hardware and the compiler technology.

Applying dynamic scheduling results in *out-of-order execution* and thus in *out-of-order completion* of instructions. Over the years various ways to cope with out of order execution have been invented, for example scoreboarding or Tomasulo's algorithm. Basically they allow instructions to execute out of order when there are sufficient resources available and when there are no data dependencies. They are used in early, heavily pipelined machines. More recently there is a tendency towards having the compiler resolve all data hazards by rescheduling dependent instructions and adding `nop` instructions if necessary. This technique is called static scheduling.

The Model

Excluding hardware solutions for data hazards completely would force a designer to use static scheduling. This would obfuscate the performance figures obtained from simulation. The figures would reflect the performance of the combination of instruction set design and compiler rather than the performance of the instruction set design alone. Therefore, there at least have to be provisions to model hazard detection and forwarding.

3.3.3 Control Hazards

Control hazards lead to having to discard part of the contents of the pipeline and thus to a situation in which the pipeline is not completely full. Consider a branch instruction. If the branch is taken, instruction execution will have to restart at the newly computed destination address, which is stored in the Program Counter. There are several ways to take branches.

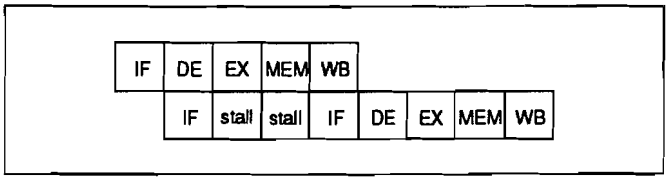

Figure 3.3: Primitive way to take branches

The most primitive way is to stall the pipeline as soon as the decode stage discovers that an instruction is a conditional branch. The pipeline is then stalled until the destination address is known. Since the destination is computed in the execute stage, the Program Counter normally is not changed until after the MEM stage. The IF stage is restarted as soon as the target is known. It is clear that this is the most ineffective way. No matter whether a branch is taken or not, the IF stage is delayed for three consecutive cycles.

An improvement can be made by assuming that the branch will not be taken. In this case, the branch is treated as a normal instruction and the instructions after the branch are allowed to enter the pipeline. This is called the *predict-not-taken* strategy.

If the branch is not taken, the next three instructions have flowed into the pipeline and execution can continue without losing cycles. Only if the branch is taken, the information of the three instructions that have flowed into the pipeline in the mean time is invalid and therefore has to be discarded. This is called **flushing** the pipeline.

The observation that a number of instructions have flowed into the pipeline already, has lead to yet another solution where taking the branch is delayed until the intructions that have flowed into the pipeline have been completed. The drawback of this method is that compilers have to take this into account when scheduling instructions.

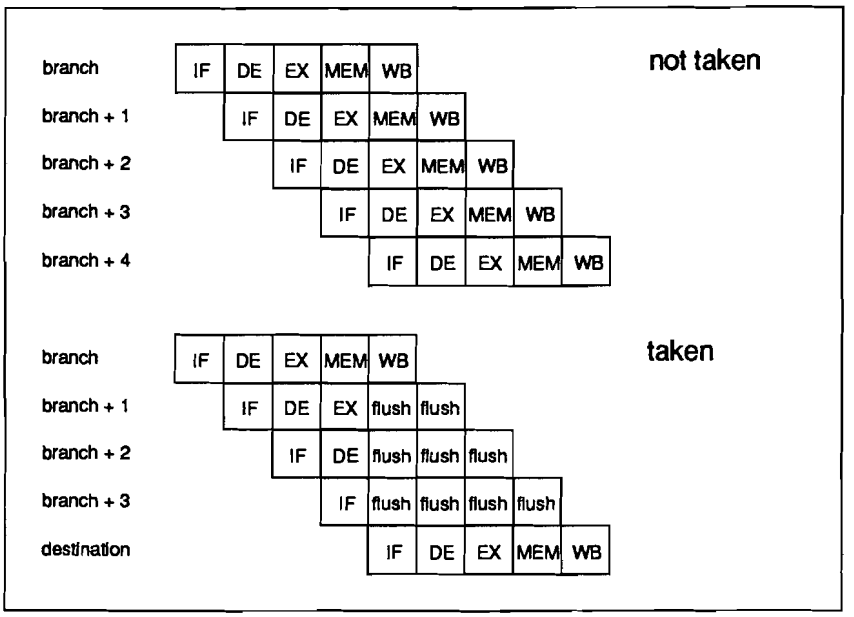

Figure 3.4: Predict-not-taken scheme, taken and not taken

In the case of branches that are taken, both methods result in a situation in which not all pipe stages are busy. This is inevitable. But the number of stalls due to branches can be reduced in a number of ways. The most well-known option is using branch prediction. This can either be done by the compiler—by adding hints to the branches—or in hardware.

The Model

Considered the relatively high frequency of occurence of instructions that change the Program Counter, it is clear that the way an architecture copes with these instructions has a large influence on its performance. This stresses the need for at least rudimentary provisions in the model to cope with control hazards.

3.4 Interrupts

Interrupts were invented to signal real-time events like I/O requests, pagefaults, etc. Later, they were also used to detect arithmetic errors. When an interrupt

occurs, the state of the machine has to be saved, together with an indication of the cause. Then, control must be transfered to a routine that handles the interrupt. After the interrupt is handled, program execution must restart at the intruction that caused the interrupt.

Interrupt handling in a pipelined machine is difficult, because instructions in the pipeline change the state of the processor on every clock cycle. It is hard to tell whether an instruction can safely change the state of the processor.

An interrupt can occur in the middle of an instruction. This means that the processor state of before the instruction execution has to be reconstructed. A possible solution to safely shut down the pipeline and save the state when an interrupt occurs, is to follow the next steps:

1. Force a trap instruction into the pipeline on the next instruction fetch.
2. Prevent state changes for instructions that will not be completed before the interrupt is handled by turning off all the writes for the faulting instruction and the instructions that follow it in the pipeline.
3. The first thing the interrupt handling routine does is save the PC of the faulting instruction, so it can be used to return from the interrupt. In case of delayed branches a number of PCs that is more than the length of the branch delay will have to be saved and restored.

A pipeline is said to have **precise interrupts** if the pipeline can be stopped so that the instructions before the faulting one are completed and those after it can be restarted from scratch.

The Model

Leaving interrupt out of the sequential simulator was a justifiable omission. In a sequential machine interrupt handling takes place after completion of an instruction. Therefore they are not relevant for the semantics of the instructions.

In addition to the IF, DE and EX stage, Takken's simulator had a TRAP stage. In this stage a trap handler could be modeled. This has never been done in practice [Hon93], but that does not matter, since this does not affect the simulation results seriously.

In case of a pipelined processor, proper attention has to be paid to interrupts *during the design.* Adding interrupts to an implementation later will lead to a complicated implementation at best. It is even more likely that adding interrupts later will make it impossible to realize the design at all. Thus interrupts cannot be left out of the design at Level 1 for pipelined processors.

3.5 Multicycle Operations

Some instructions—like floating point instructions—can only be completed in one clock cycle if either the clock is slowed down tremendously or if enormous amounts of logic are used in the floating point units, or both. It is more effective to allow for a longer latency for these operations.

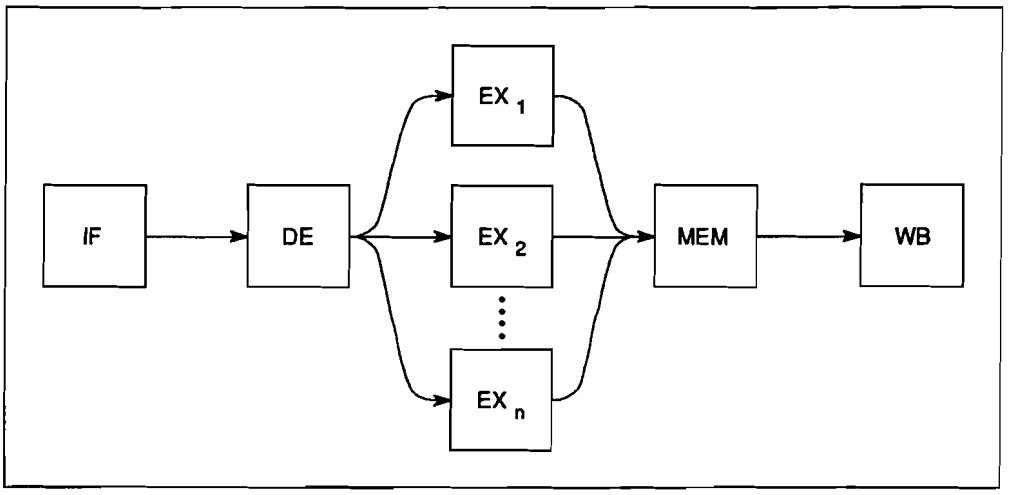

Figure 3.5: Pipeline with multiple execute alternatives

If it is assumed that the floating point operations have the same pipeline as the integer instructions, the situation can be depicted as in Figure 3.5. It shows a pipeline with multiple execute alternatives. Each alternative may take as many cycles to complete as needed. Thus it is allowed to overlap instructions whose running times differ. This results in out-of-order execution.

It introduces the possibility of WAW and WAR hazards, contention for register access at the end of the pipeline and it greatly complicates the implementation of precise interrupts.

The Model

Finding a way to cope with these difficulties at Level 1 turned out not to be possible. The primitive model presented in Chapter 5 offers a way to approximate the behavior of a pipeline for instructions which need more cycles in the execute stage, but for reasons to be discussed in that chapter, that model had to be rejected.

Reference

Pipelining is very complex. In this chapter a simplified description has been given which suffices for the next chapter. The interested reader is referred to [HP90].

Chapter 4

The Hardware

"Science is organized common sense
where many a beautiful theory
was killed by an ugly fact."
Thomas H. Huxley

4.1 Adding Implementation Details

In the simplified model of Figure 3.1 a processor is a sequence of pipe stages through which the information flows. The rest of Chapter 3 outlines the complications of high-performance pipeline design. SIMPROACH is intended to gather information that gives a fair indication of the performance, so a designer needs to have the means available to describe his solutions for the problems. To accomplish this, it is necessary to incorporate information about the implementation. This information is needed to be able to model the *communication* between the pipe stages.

In this context, communication denotes two things. First, in a pipeline information is passed on from one stage to the next stage—or better: stages. The model needs facilities to describe this communication. Secondly, there must be a way to describe the control. The control determines what each of the pipestages has to do during every cycle of the simulation.

A designer has to choose a good partitioning of the instruction execution. One of the goals of SIMPROACH is to allow the designer to experiment with this partitioning in order to find an optimal one. Therefore he must be able to describe the pipestages and how they communicate. To arrive at a level of implementation detail which allows for convenient description as well as simulation, this chapter shows how a sequential implementation evolves into a pipelined implementation.

4.1.1 Sequential Implementation

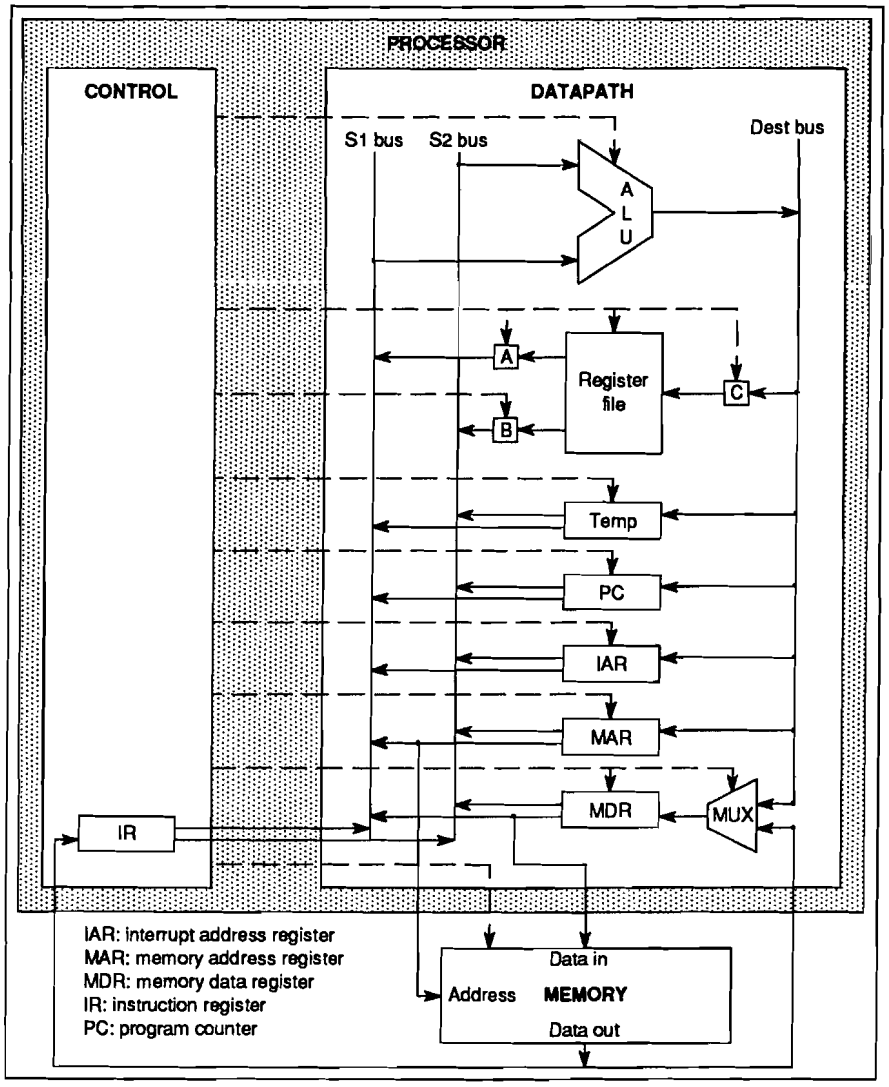

Figure 4.1: A possible sequential implementation of the MIPS R2000.

Figure 4.1 is used by Hennessy and Patterson [HP90] in their chapter on basic processor implementation techniques. It shows a possible sequential implementation of their example architecture DLX. The DLX architecture resembles the MIPS R2000 architecture very much. The fact that they use the MIPS R2000 architecture in their second book [HP93] stresses this fact. Because of its simplicity, it is an ideal model for study.

The figure shows a processor consisting of a control unit and a datapath. The datapath contains the function units and the registers. These are all connected to buses. What happens in the datapath is determined by the control unit, based on the current instruction. The control lines are shown as dashed lines.

Figure 4.1 is rather deceptive. The control looks easier than the datapath. But as indicated in Chapter 2 it is the most complicated part of the processor. A SIMDES description would not mention control at all. Simulation of a sequential machine is not a big problem because only the assumption that it works according to the von Neumann model has to be made.

A SIMDES description of this architecture would only contain the information that is required to simulate the behavior of the instructions. This includes the number of registers, the width of the registers, the Program Counter, the size and width of the memory, and of course a description of the behavior for each instruction. It would not mention the Memory Address Register, the Memory Data Register, or the Temporary Register because these are not essential for the behavior of the sequential processor. Neither is the control.

4.1.2 Unfolding the Datapath

A sequential implementation of the datapath for a subset of the MIPS R2000 instruction set is shown in Figure 4.2. The subset used consists of the memory reference instructions `lw` (load word) and `sw` (store word), the arithmetic-logical instructions `add`, `sub`, `and`, `or`, and `slt` (set on less than) and finally the branch equal instruction `beq` and the jump instruction `j`. Even though this is a very limited subset the design of the control will turn out to be rather complicated.

Breaking the instruction into steps corresponding to the functional unit operations that are needed, makes the division into functional units visible, as it is required for the parallel implementation. The control for a sequential implementation can be implemented as a finite state machine. The state it is in, depends

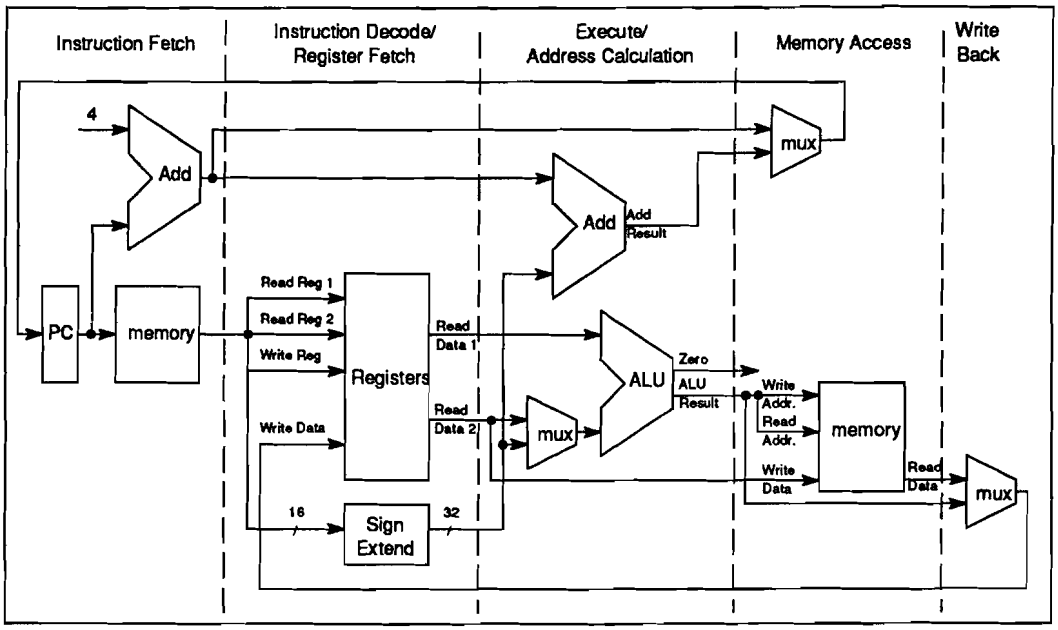

Figure 4.2: Unfolded datapath.

on the type of instruction the processor is executing.

For the simple subset of the MIPS R2000 instruction set, the complete control for the sequential implementation, including the interrupt-handling extension for arithmetic overflow and illegal instructions, can be realized with a finite state machine with thirteen states. For an instruction set with more instructions of widely varying types, the control unit could easily require thousands of states with hundreds of sequences [HP93]. Description of the control by means of a finite state machine will then become very cumbersome.

Another option is to implement the control as a program which implements the machine instructions in terms of microinstructions. The underlying idea is to represent the the control lines symbolically, so that the microprogram is a representation of the microinstructions. This requires devising a syntax for the microinstruction assembly language.

The microinstructions are syntactically represented as a sequence of fields whose functions are related. Some of the fields of the microinstruction determine the value of the control lines for the data path. The remaining fields specifies how to select the next microinstruction.

No matter whether the choice is made in favor of microcoded control or in favor of the finite state machine approach, the fact remains that designing the control is difficult. To avoid the problems associated with the design of a microinstruction language or a finite state machine, the designer will be allowed to express the control algorithmically in the description language.

4.1.3 Parallel Implementation

For parallel execution and thus also for simulation of parallel execution, the execution of an instruction has to be partitioned into a number of stages. The MIPS R2000 pipeline as described in [HP93] consists of the following five execution stages.

1. IF – instruction fetch
2. ID – instruction decode and register fetch
3. EX – execution and effective address calculation
4. MEM – memory access
5. WB – write back

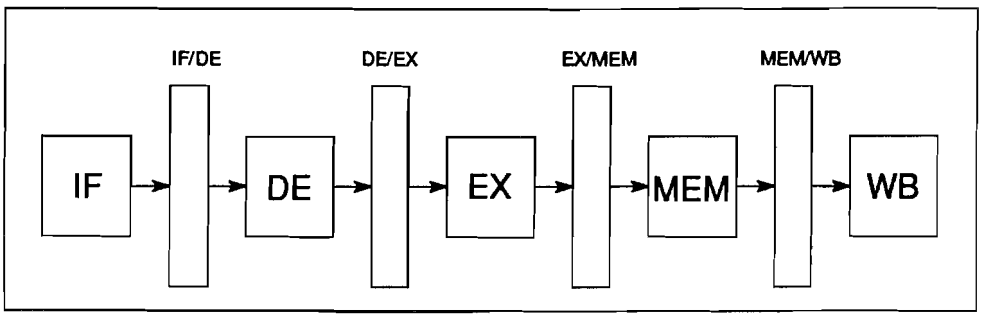

Figure 4.3: Stylized unfolded datapath with pipeline registers added.

Unfolding the datapath as illustrated in Figure 4.2 [HP93] yields a view that resembles the simplified model of a pipeline presented in Figure 3.1. But it is still the datapath of the sequential implementation. To turn it into a real pipeline, the functional units have to be separated to correspond to the pipe stages.

The functional units that can be distinguished in the figure resemble the execution stages. By adding registers between these functional units the datapath is turned into a datapath for pipelined execution. Figure 4.3 shows a stylized version of Figure 4.2 with the pipeline registers added.

Comparing Figure 4.4 with Figure 4.2 shows that adding the pipeline registers is not the same as simply replacing the dotted lines with registers. The *Write register* number in the sequential implementation is supplied by the instruction that is currently executing. Replacing the dotted line by pipeline registers would erroneously cause the *Write register* number of instruction i, which needs this number to write back the result in clock cycle c, to be replaced by the *Write register* number of instruction $i - 3$.

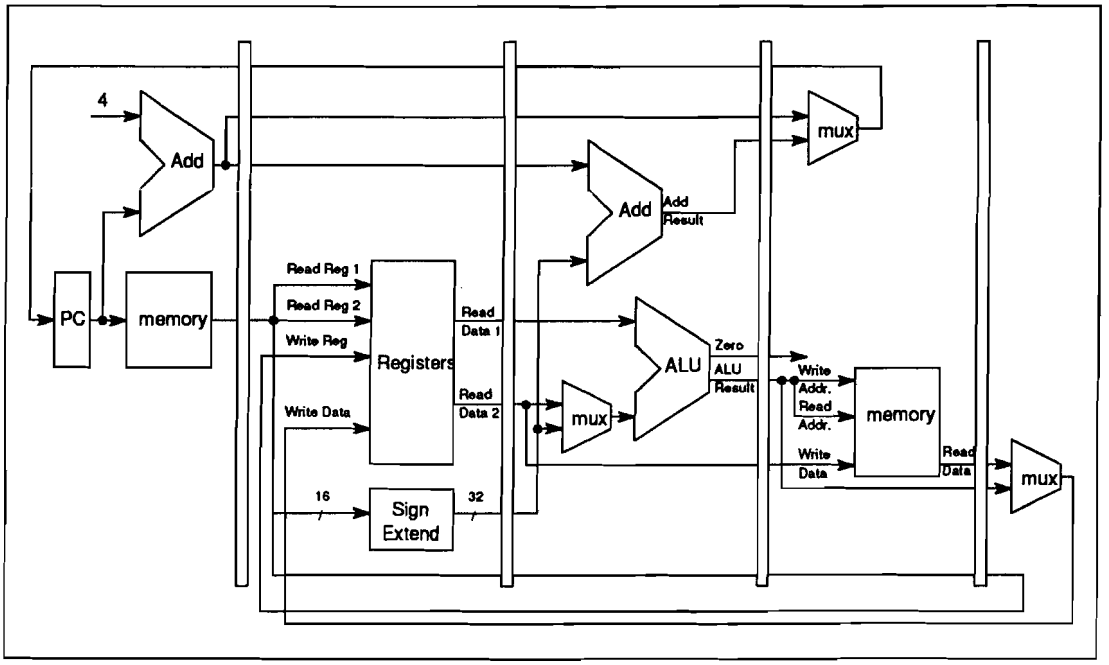

Figure 4.4: Datapath with pipeline registers.

This brings us to the the following important point. Not only information needed in the next pipe stage needs to be passed on to that stage through the pipeline register. Also information needed in later stages must be passed on, otherwise the information is lost at the very moment on which a next instruction enters that pipeline stage. Care has to be taken that each pipeline stage contains the portion of the instruction needed for that stage and all later stages. This is called *preserving* the instruction.

4.1.4 The Main Control of a Pipelined Datapath

The previous section suggests that in order to make a pipelined implementation it suffices to just take the control of the sequential implementation and plug it into a parallel implementation. Besides the normal control information as found in the sequential implementation extra control units have to be added to deal with the extra information typical to the pipelined implementation. To be able to distinguish the normal control unit from the additional control units, it is called the *main control unit* in the rest of this text.

In practice the main control unit is the easiest part of the control section of a pipelined implementation. It can be seen as an extra block of logic in the decode stage that determines the state of the control signals of the instruction currently in the decode stage. These control signals are then passed on to the next stages through the pipeline registers.

Theoretically, pipelining does not change the meaning of the control signals. In this naive point of view implementing control in the pipelined model means setting the control lines to the appropriate values in each stage and for each instruction. The easiest way to accomplish this is to pass the control information on in the pipeline registers. Each pipe stage uses the information that is associated with the instruction it is operating on. The control information intended for the next stages, is passed on through the pipeline registers. This is depicted in Figure 4.5.

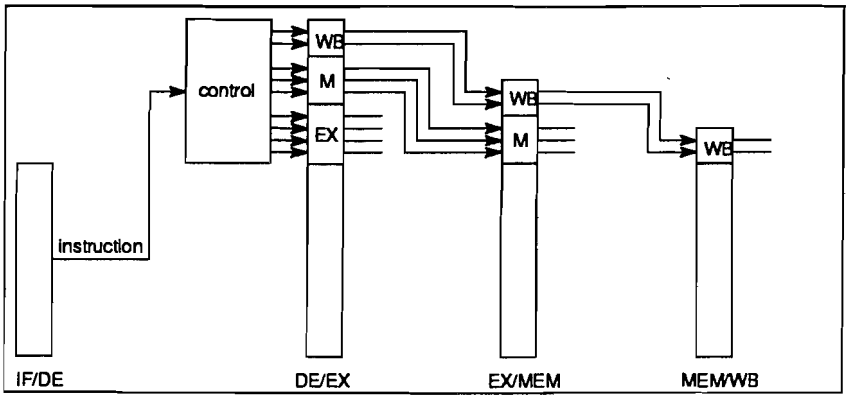

Figure 4.5: How the control information is passed on.

In contrast to the sequential implementation of the control, special hardware is no longer required to sequence the control. Sequencing the control is solved

by the pipeline structure itself. All instructions take the same number of clock cycles, and all control information is computed during instruction decode and then passed along by the pipeline registers, so no next state hardware is needed anymore.

The descriptions of the nine control signals of the R2000 are listed below because the information is needed in the rest of this report.

The following four (`aluOp` is a two bit signal) control signals are used in the execute stage.

`regDst` When active, the number of the destination register comes from the `rd` field of the instruction. When not active, it comes from the `rt` field of the instruction.

`aluOp` Is a two bit control signal which controls the ALU operation.

`aluSrc` When active, the second ALU operand is the sign extended lower 16 bits of the instruction. When not active, the second ALU operand comes from the second register file output.

The following three control signals are used in the memory stage. Note that in the memory stage the `zero` output from the ALU is anded with the `branch` control signal to yield the `pcSrc` signal. The `zero` output is not originating from the main control unit, yet it has to be passed on in the pipeline register.

`memRead` When active, the data memory contents at the address given by *Read Address* are put on the *Read Data* output. No effect when not active.

`memWrite` When active, the data memory contents at the address given by *Write Address* are replaced by the value on the *Write Data* input. No effect when not active.

`branch` Is ANDed with the *Zero* output of the ALU to yield the `pcSrc` signal.

`pcSrc` When active, the PC is replaced by the output of the adder that computes the branch target. When not active, the PC is replaced by the output of the adder that computes the value of PC + 4.

Finally, in the write back stage there are two more control signals.

memtoReg When active, the value fed to the *Write Data* input of the register file comes from the data memory. When not active, it comes from the ALU.

regWrite When active, the register given by the *Write Register* number input is written into with the value on the *Write Data* output. No effect when not active.

4.2 The Other Control Units

Like in the previous section, also of the other control units only a brief description will be given. The following sections merely introduce the hardware required to solve certain problems, along with the signals specific for the R2000. The operation of the units is described superficially. In the next chapter this information will be used in order to come to a model that can be simulated, and a description language.

4.2.1 Data Hazards

The hazard detection unit is used to detect possible hazards. The hazard detection unit is a piece of logic with as its inputs the instruction currently in the decode stage, and the `rt` and `rd` fields of the instruction in the DE/EX register, and the `writeRegister` field of the EX/MEM register and the MEM/WB register, and the `regWrite` bits from the DE/EX- the EX/MEM- and the MEM/WB pipeline registers.

From this information the unit determines whether there is a hazard condition. If a hazard is detected, part of the pipeline has to be stalled. This is achieved by allowing the Program Counter as well as the IF/DE pipeline register to be written and inserting zeroes into the control fields of the DE/EX pipeline register. The latter is done by using the output of the hazard detection unit as the control line of a multiplexor which either selects the output of the main control unit or zeroes.

In many cases hazards can be solved by forwarding an intermediate result to the ALU. In order to do so, multiplexors have to be connected to the ALU inputs. Based on the control lines of the multiplexors, they either select the ALU operand

either from the normal register or from the prior ALU result in the appropriate pipeline register.

If `regWrite` is active in either the EX/MEM pipeline register or the MEM/WB pipeline register, the `readRegister1` and `readRegister2` number of the instruction in the DE/EX pipeline register are compared with the `writeRegister` number of the EX/MEM or the MEM/WB pipeline register, respectively, to see if there is a RAW. These comparisons are used to determine the multiplexor control lines `aluSelA` and `aluSelB`. The `aluSrc` control line is used as control line for a third multiplexor which determines whether the B input of the ALU is taken from the multiplexor controlled by `aluSelB` or the sign extended immediate.

4.2.2 Control Hazards

If a branch is taken, the `pcSrc` signal is active. This signal can be used to implement a predict-not-taken scheme by using it as a control line to flush the fetch, decode and execute stage. The decode and execute stages are flushed by using a multiplexor to insert zeroes into the control bits of the pipeline registers of these stages. In the fetch stage it is directly fed into the pipeline register. If that bit is set, the main control unit will make all control signals 'not active' during that instruction's execution.

4.2.3 Interrupts

Just like with the branch instruction in the predict-not-taken scheme, instructions following the one that caused the interrupt will have to be flushed. The address of the instruction that caused the interrupt will have to be saved, so that program execution can restart as soon as the interrupt routine is finished. To start fetching instructions from the address of the interrupt routine, an input on the PC multiplexor that sends this fixed addess to the Program Counter has to be added.

4.2.4 Parallel Function Units

In the case of parallel function units, extensive interlocks are required to take care of situations where the result of one operation is the operand of a second

operation. The interlocks are implemented in a generalized queue and reservation scheme called the scoreboard, where a record is maintained for the usage state of each register, functional unit and interconnecting bus. Each new instruction causes an entry to be added to the scoreboard. This delays the instruction issue if necessary, but does not delay the issue of subsequent instructions. The details of this are beyond the scope of this report.

Chapter 5

Describing and Simulating the Hardware

'Ceci n'est pas une pipe'
Painting by Marguerite

The model presented in the previous chapter essentially says that a pipeline can be viewed as a collection of stages that pass on information to each other. In reality, all stages are active at the same time. The key problem when implementing the simulator is that this parallelism has to be described in a sequential language. This means that a way has to be found to sequentially execute the behavioral descriptions of the separate pipestages in such a fashion that the effect is the same as when they were executed at the same time. The user of SIMPROACH has to be safguarded for these complications.

The design of the description language as well as the design of the simulator go hand in hand and therefore they are presented together in one chapter. The reader will have to take the operational approach of this chapter for granted, since this is the only acceptable way to find a solution.

Figure 4.3 shows the basic model. It does not describe how the pipe stages communicate with each other. This chapter presents two ways to cope with the communication between the pipe stages. First, observing the behavior of the entire system results in the use of handshake wires between the stages. As will be shown, this method has its drawbacks.

In the second alternative, the implementation (which is shown in Chapter 4) is described in more detail. This leads to a model which is more realistic and also easier to describe and to simulate.

5.1 Terminology and Conventions

In general, a processor has $N(N > 0)$ pipe stages. For ease of explanation it is assumed that the first is a fetch stage, the second is a decode stage and that the third is an execute stage. This can be assumed without a loss of generality. Furthermore, it is assumed that for each pipe stage there is a function that mimics its behavior and that this function bears the same name as the corresponding pipe stage. A stage can be referred to by its name or by its number.

The pipeline registers introduced in the previous chapter are modeled by inserting a buffer between the pipe stages. A buffer is associated with a pipe stage. Buffer n (where $1 \leq n < N$) contains the information produced by stage n. These buffers resemble the latches often found between pipe stages in actual implementations.

Simulation of one clockcycle is defined as calling the functions N, $N-1 \ldots 1$. Even though in the simulation these functions are called sequentially, the events are said to have taken place in one clock cycle. This definition introduces the important notion of time into the model. The evaluation order of the pipe stages is reversed to assure that the data produced by stage n in clock cycle c are based on the data produced by stage $n-1$ in clock cycle $c-1$.

Throughout this chapter the programming language C [KR78] will be used to describe code fragments. Where the actions are irrelevant for the explanation or become too elaborate to write out completely, plain English in single quotes is used.

5.2 A Primitive Model

The definition of a clock cycle as given in the previous section introduces a problem. How does a pipe stage 'know' it has to do something? For example, in the first cycle after simulation is started or after the pipeline is flushed completely, only the fetch stage is able to do its job. On the second clock cycle, there is in-

formation available for the decode stage, so both the decode stage as well as the fetch stage are able to do their job, but the others are not able to do anything.

For that purpose the buffers contain, besides the data, also execution control information. A stage is allowed to execute only when the previous stage has data available and the next stage is ready to receive data. The execution control information in buffer $n-1$ and buffer n determines if stage n is allowed to execute. For normal operation a stage only needs to modify the execution control information in the buffers that are connected to it, but for exceptions (for example branches) they can modify the execution control information of the entire pipeline.

If stage n is allowed to execute, it uses the value stored in buffer $n-1$ to compute the value for buffer n. In the execution control field of buffer $n-1$ it marks that the value which stage $n-1$ has put in there during the previous clock cycle has been used. When stage $n-1$ is called, it will interpret this as '*the next stage is ready to receive*'. The result of the operation of stage n in the current cycle will be stored in buffer n. Stage n will mark the execution control field to let stage $n+1$ know there is new data available. In the next clock cycle stage $n+1$ will interpret this as '*the previous stage has data available*'.

To be able to model multicycle operations, a 'busy' field is added to the execution control field. Now, stage n is allowed to execute if stage $n-1$ has produced data AND if stage $n+1$ has consumed the data that n has produced earlier AND if stage $n+1$ is not busy. Figure 5.1 shows a pipeline with the two handshake lines as just described.

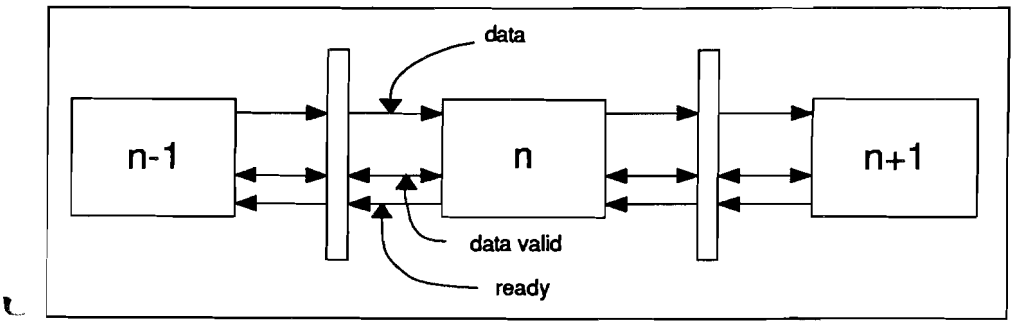

Figure 5.1: Model with handshake lines

This gives enough information about the model to construct a very primitive simulator. To be able to describe it conveniently, a few primitives need to be introduced.

- `PRODUCE (data, n)` puts the data produced by stage n in buffer n. It also sets a flag to denote that the data now in buffer n has been produced and thus is ready to be consumed.
- `CONSUME (n)` reads the information produced by stage $n-1$ and clears the flag, denoting that the data have been consumed.
- `SETBUSY (n)` sets the busy flag for stage n.
- `SETREADY (n)` clears the busy flag for stage n.
- `ISBUSY (n)` is `TRUE` if the busy flag of stage n is set.
- `ISREADY (n)` is `TRUE` if the busy flag of stage n is clear AND the value produced by $n-1$ is valid AND the value produced by n in the previous cycle has been consumed by $n+1$ in this cycle.

The control can now be described with the following piece of pseudo C code:

```
void
control()
{
  flushed = FALSE;
  n = N -1;
  while ((NOT (flushed)) AND (n >= 0)) {
    if (ISBUSY(n) | ISREADY (n))
      'call stage n';
    n = n - 1;
  }
}
```

The control 'knows' the number of the stage it is calling. This number is used internally by the simulator, but the designer need not be aware of it. It should be hidden in the final description language. But in the C fragment representing the framework for the execute stage as shown below, the stage number is visible.

```
void
execute (int n)
{
  if (cycles_left == 0) {
    CONSUME(n);
    'do whatever needs to be done in this stage for this instruction'
    'determine cycles_left for this instruction'
  }
```

```
  else
    cycles_left = cycles_left -1;
  if (cycles_left == 0) {
    PRODUCE (data, n);
    SETREADY (n);
  }
  else
    SETBUSY (n);
}
```

Here, it is assumed that the execute stage is the only stage which might need more cycles to complete an operation. The framework for the other stages is similar to the one just presented, but it lacks the check for the number of cycles left.

5.2.1 What Is Wrong with the Primitive Model

In this model, instead of trying to create a one-to-one correspondence with the physical implementation of a processor, an abstraction of how the pipe stages communicate is used. This approach results from the general misconception that a model in which there is no need to describe the implementation must be easier than a model which requires more information about the implementation. But the attempt to keep the model simple results in very complicated descriptions and simulators.

The first confusing aspect is that it is not clear what information has to be passed on to the next stages. If a designer wants to, he can see the partitioning of what happens in which stage seperately from the partitioning of the execution of the instructions. This can lead to simulators that do not reflect the true nature of the architecture.

The second, but most confusing aspect, is that this model forces the designer to attempt to describe the datapath *and* the control simultaneously. Actually, the control is partially hidden in the `control` function and partially taken care of by the stages.

For example, if a branch is taken, the execute stage will have to modify the execution control information for the entire pipeline in such a fashion that on the next clock cycle only the fetch stage will execute. This can be done in the description of the execute stage. A designer might want to model this by defining

a function to accomplish this. This function can then be said to be part of the 'control', which would agree with the designers' point of view of a processor.

After modifying the execution control information of all stages, this function will have to set the variable `flushed` to true, which forces `control` to abort the execution of the current cycle and thus causes the information in the stages before it, which has become invalid, to be discarded. In the next clock cycle, execution will resume at the new Program Counter.

The previous example indicates how control hazards can be modeled. It is also possible to model stalls as a solution to structural hazards. To establish this, all resources have to be furnished with a usage counter and a maximum for the usage. Take for example single-ported memory. The maximum for its usage is one. At the beginning of a clock cycle, the usage counter of this memory is cleared to zero. If the memory stage writes information to memory, it increases the usage counter. The instruction fetch stage checks the usage counter and sees that the maximum has been reached. Therefore it can not complete its operation in this cycle.

Also data hazards can be modeled. If it becomes clear in the decode stage that an instruction is going to write to a certain register, this register can be tagged. If an instruction following this instruction needs the value of that particular register it will have to wait until the register is written and the tag is cleared.

It might be possible to describe and simulate many of the features of a high performance pipeline, but the fact that the control is partially described in the pipeline stages results in extremely complicated and messy descriptions and in non-standard simulators. This means that the designer virtually has to provide the entire simulator framework and thus might even be better off by hand coding the complete simulator.

5.3 The Improved Model

As a starting point for the improved model, the same basic model as in the previous section is taken. But this time, a description is chosen that is closer to the implementation, which means that control is taken care of by separate units as seen in Chapter 4. This results in descriptions that are easy to write and understand, and also in a uniform simulator framework.

In the ideal case the information travels 'from the left to the right' (from the first stage to the last) and thus the behavior of the entire pipeline can be simulated by executing the stages 'from the right to the left' (from the last stage to the first). In cycle c stage n uses the information produced by stage $n-1$ in cycle $c-1$. So by evaluating the pipeline from the right to the left, information is used before it is overwritten. The information produced by a stage is stored in a buffer, representing a pipeline register.

Contrary to the primitive model, where the designer could pass on anything he wanted in the pipeline registers, the choice of what is passed on is now determined by the implementation. In reality pipeline register n contains the information that has to be passed on to stages $n+1$ and further. The designer has to read the fields required from the specification of the implementation.

To keep the simulator simple, the contents of the pipeline registers is made identical for all stages. This means that in some stages fields have a meaning, while in others they do not mean anything. It is up to the designer to use the correct fields in each pipe stage.

The meaning of a field in for example pipeline register DE/EX depends on what the decode stage produces and the execute stage consumes. As a working example, the rest of this section presents a way to describe the information required to simulate the MIPS R2000 pipeline as described in [HP93].

The following type definitions are used:

```
typedef int CtrlLine;          /* A control line.  */
typedef unsigned int Aword;    /* A word in the architecture, 32 bits.  */
typedef Aword RegField;
```

For the MIPS R2000, the pipeline register contains the following information:

```
struct pipeReg {
  RegField programCounter;
  RegField instruction;
  RegField readData1;
  RegField readData2;
  RegField signExtended;
  RegField writeRegisterRt;
  RegField writeRegisterRd;
  RegField writeRegister;
  RegField addResult.
```

```
  RegField zero;
  RegField aluResult;
  RegField readData;
  RegField writeData;
}
```

Figure 5.2 shows in which pipeline register which fields have a meaning. A dot in the table means that the value of the field is required in the next stage or one of the stages after that and thus has to be set by the stage associated with the pipeline register. Since the `zero` output of the ALU has to be passed on from the execute stage to the memory stage, it is also mentioned in this overview. The `pcSrc` signal is generated by anding the `branch` and `zero` signals. The fact that both `pcSrc` and `zero` do not directly come from the control can easily be seen in the table.

	DE/EX	EX/MEM	MEM/WB	WB
`regDst`	•			
`aluOp`	•			
`aluSrc`	•			
`memRead`	•	•		
`memWrite`	•	•		
`branch`	•	•		
`zero`		•		
`pcSrc`			•	
`memtoReg`	•	•	•	
`regWrite`	•	•	•	•

Figure 5.2: Where the particular fields have a meaning

5.3.1 Control

The control as present in a sequential implementation of this processor has nine control lines. Their meaning is explained in Chapter 4. In the parallel implementations their values are determined in the decode stage, based on the instruction currently being decoded, and passed on to the next stages through the pipeline registers. This can be seen as passing them on along with the instruction to which they belong. The control lines have to be added to `pipeReg`. Since the control lines are exclusively used in one pipe stage, they are listed per stage.

```
/* Control line used in DE.  */
CtrlLine regWrite;
/* Control lines used in EX.  */
CtrlLine aluSrc;
CtrlLine aluOp;
CtrlLine regDst;
/* Control lines used in MEM.  */
CtrlLine memRead;
CtrlLine memWrite;
CtrlLine branch;
CtrlLine pcSrc;
/* Control lines used in WB.  */
control memtoReg;
```

Finally, the definition of the resources can be very simple:

```
Aword Mem[10000];
Aword Reg[32];
```

Now the resources and fields of the pipeline registers have been defined, the behavior of the stages can be expressed as operations on the incoming values and on the resources, depending on the incoming control lines, yielding outgoing values which can be used as the inputs for the next stages. In the simulator framework, pipeline register n is associated with stage n.

A stage only needs to 'know' what to do but it does not need to 'know' its number. The simulator framework uses these numbers to determine the order in which the stages are called. For the description of the pipestages the numbering can be hidden for the designer.

In the descriptions of the pipestages that follow below, an incoming value is denoted by `in.itsName`. This is a shorthand notation for `stages[n].pipeReg.itsName`. If an outgoing value is written, it is denoted in a similar way by `out.itsName`. For control lines, this indication is not necessary in the final description language, since they are always incoming.

The model is changed so that the writeback stage has a pipeline register associated with it as well. This is due to an exception which has caused great difficulty during this research. The essence of the problem is that information is sent back into the pipeline, *against the flow of information* mentioned before. Both `writeRegister` as well as `writeData` come from the writeback stage...

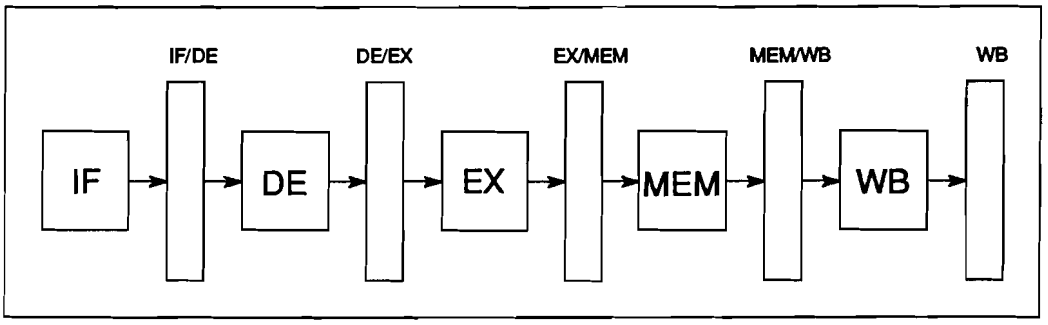

Figure 5.3: The extended model with a register added after the last stage

If the write back stage would *not* have a pipeline register associated with it and the stages would be evaluated from right to left, the information needed to correctly complete the operation of the decode stage would no longer be available.

This problem is solved by adding a register after the write back stage—as can be seen in Figure 5.3—and by indirectly allowing the decode stage to violate the rule that in general a stage is only allowed to use information from a previous cycle and a previous stage to fulfill its task in the current cycle.

The pipestages are now described in a C-like language.

```
void
writeback (int n)
{
  if (in.memtoReg == 1)
    out.writeData = in.readData;
  else
    out.writeData = in.aluResult;
  out.writeRegister = in.writeRegister;
  out.programCounter = in.programCounter;
}


void
memory (int n)
{
  if (in.branch == 1 & in.zero == 1)
    PC = in.addResult;
  else
    PC = PC + 4;
```

```
  if (in.memRead == 1)
    out.readData =  Memory[in.readAddress];
  if (in.memWrite == 1)
    Memory[in.writeAddress] = in.writeData;
}


void
execute (int n)
{
  out.addResult = in.programCounter + in.signExtended;
  if (in.regDst == 1)
    out.writeRegister = in.writeRegisterRt;
  else
    out.writeRegister = in.writeRegisterRt;
  /* Need temporary value for alu input.  */
  AWord aluIn;
  if (in.aluSrc == 1)
    aluIn = in.signExtended;
  else
    aluIn = in.readData2;
  switch aluOp {
    1: out.aluResult = in.readData1 + aluIn;
       break;
    2: out.aluResult = in.readData1 - aluIn;
       break;
    3: /* use function code */
       break;
  }
}


void
decode (int n)
{
  'Determine instruction type'
  'Forward all required stuff to next stage'
  'Determine control lines'
  'Forward eight control lines to the next stages'
  if (regWrite == 1)
    Reg[WBout.writeRegister] == WBout.writeData;
}
```

In the decode stage the instruction type can be determined using the strategy developed by Takken [Tak92]. Determining the value of the control lines can be done by allowing the designer to algorithmically specify the behavior of the control in a function and having the simulator call that function in the decode stage. Finally the fetch stage can be described by the following piece of code:

```
void
fetch (int n)
{
  out.instruction = Mem[PC];
  out.programCounter = PC + 4;
  PC = WBout.programCounter;
}
```

5.3.2 Data Hazards

Up to now an ideal pipeline has been described in a fashion which allows for simulation. Control has been added and the hurdle of sending information upstream has been taken by adding an extra register after the write back stage. It is not usable yet to perform real simulations because it does not cover hazards nor interrupts.

Now it is time to tackle the problem of data hazards. First the problems arising from dependencies will be solved by stalling the pipeline when a data hazard occurs. Data hazards occur if in a clock cycle the instruction in the decode stage needs to read a register that will be written by an instruction in either the execute stage, the memory stage or the write back stage. Whether or not a hazard condition occurs can be determined by examining the control information in the pipeline registers. This control information tells what is going to happen in the next clock cycle.

Denote the pipeline registers between two stages with the names of its adjacent stages separated by an underscore. For example, the register between the decode stage and the execute stage is denoted with `ID_EX`. A hazard, caused by the instruction in the execute stage wanting to write to a register from which the instruction in the decode stage needs to read, is called an EX hazard.

Using the nomenclature just introduced, an EX hazard can be detected by evaluating the next expression:

```
DE_EX.regWrite & (
((DE_EX.regDst == 0) & (DE_EX.writeRegisterRt == IF_DE.readRegister1)) |
((DE_EX.regDst == 1) & (DE_EX.writeRegisterRd == IF_DE.readRegister1)) |
((DE_EX.regDst == 0) & (DE_EX.writeRegisterRt == IF_DE.readRegister2)) |
((DE_EX.regDst == 1) & (DE_EX.writeRegisterRd == IF_DE.readRegister2))
)
```

A MEM hazard can be detected by evaluating:

```
EX_MEM.regWrite & (
(EX_MEM.writeRegister == IF_DE.readRegister1) |
(EX_MEM.writeRegister == IF_DE.readRegister2)
)
```

And finally, a WB hazard can be detected by evaluating:

```
MEM_WB.regWrite & (
(MEM_WB.writeRegister == IF_DE.readRegister1) |
(MEM_WB.writeRegister == IF_DE.readRegister2)
)
```

The WB hazard does not exist if the read delivers what is written, as is the case for many implementations of register files and also for the MIPS R2000.

If a hazard is detected, the pipeline has to be stalled. That means that in the next clock cycle, the PC and the `IF_DE` pipeline register are not allowed to be updated. The situation in the fetch stage has to be preserved until the hazard has gone. That also implies that the decode stage is not allowed to decode the instruction. Instead, zeroes are substituted for the control information that goes into the `DE_EX` pipeline register. Normally, this information would be extracted from the instruction in the `IF_DE` pipeline register. The instructions in the execute, memory and write back stage are allowed to proceed.

In the simulator framework the Hazard Detection Unit is implemented by adding a function which is evaluated after all stages have been evaluated. This way the control lines indicating that a hazard is about to occur, can be used in the next clock cycle to take the measures as just described. The function for the hazard detection unit looks is as follows:

```
void
hazardDetectionUnit ()
{
  int exHazard, memHazard;
  exHazard = DE_EX.regWrite & (
  ((DE_EX.regDst == 0) & (DE_EX.writeRegisterRt == IF_DE.readRegister1)) |
  ((DE_EX.regDst == 1) & (DE_EX.writeRegisterRd == IF_DE.readRegister1)) |
  ((DE_EX.regDst == 0) & (DE_EX.writeRegisterRt == IF_DE.readRegister1)) |
  ((DE_EX.regDst == 1) & (DE_EX.writeRegisterRd == IF_DE.readRegister1))
  );
  memHazard = EX_MEM.regWrite & (
  (EX_MEM.writeRegister == IF_DE.readRegister1) |
  (EX_MEM.writeRegister == IF_DE.readRegister2)
  );
  hazardDetected = memHazard | exHazard;
}
```

Now, the `fetch()` and `decode()` function have to be modified to take the appropriate actions in case a hazard occurs. The other functions remain unchanged.

```
void
fetch (int n)
{
  if (!hazardDetected) {
    out.instruction = Mem[PC];
    out.programCounter = PC + 4;
    PC = WBout.programCounter;
  }
}
```

```
void
decode (int n)
{
  if (hazardDetected) {
    'Set all control lines to zero'
    'Forward the control lines to the next stage'
  }
  else {
    'Determine instruction type'
    'Forward all required stuff to next stage'
    'Determine control lines'
    'Forward eight control lines to the next stages'
    if (regWrite == 1) {
      Reg[WBout.writeRegister] == WBout.writeData;
  }
}
```

5.3.3 Forwarding

Instead of stalling the pipeline when a hazard is detected, the intermediate results that are in the pipeline registers already can be forwarded to the inputs of the ALU. The hazard conditions are tested by the following pieces of code. The tests yield the values for the select lines for the multiplexers at the inputs of the ALU. The first set of expressions test for an EX hazard, the second set of expressions tests for a MEM hazard.

Hazards can occur in both the execute and memory stages at the same time. In this case, priority goes to the EX hazard because it is found in the instruction nearest the instruction in the in the decode stage in the program execution order. To prevent the conditions for the MEM hazard from setting the `aluSelA` and `aluSelB` in this case, test conditions have to be added to compare the `EX_MEM.writeRegisterRt` with `DE_EX.readRegister1` and `DE_EX.readRegister2` respectively.

```
void
forwardingUnit()
{
  if (EX_MEM.regWrite & (EX_MEM.writeRegisterRt == DE_EX.readRegister1)) {
    aluSelA=1;
  }
  if (EX_MEM.regWrite & (EX_MEM.writeRegisterRt == DE_EX.readRegister2)) {
    aluSelB=1;
  }
  if (MEM_WB.regWrite & (MEM_WB.writeRegisterRt == DE_EX.readRegister1)
                      & (EX_MEM.writeRegisterRt != DE_EX.readRegister1)) {
    aluSelA=2;
  }
  if (EX_MEM.regWrite & (MEM_WB.writeRegisterRt == DE_EX.readRegister2)
                      & (EX_MEM.writeRegisterRt != DE_EX.readRegister2)) {
    aluSelB=2;
  }
}
```

The description of the execute stage has to be adapted to select the correct ALU input according to the ALU select lines determined this way.

When a register use immediately follows its load, forwarding does not work and thus a Hazard Detection Unit to stall the pipeline is required. The new Hazard

Detection Unit works in the presence of the Forwarding Unit. Its control can be simplified to:

```
void
hazardDetectionUnit ()
{
  hazardDetected = (DE_EX.regWrite &
                    (((DE_EX.regDst == 0) &
                      (DE_EX.writeRegisterRd == IF_DE.readRegister1)) |
                     ((DE_EX.regDst == 0) &
                      (DE_EX.writeRegisterRd == IF_DE.readRegister1))
                    )
                   );
}
```

5.3.4 Branch Hazards

If a predict-not-taken scheme is to be implemented, the pipeline has to be flushed when a branch is taken. If a branch is taken, can be told from the st pcSrc signal. The execute stage and the decode stage can be rewritten to take this into account by deasserting the control lines. The fetch stage now has to select the destination address rather than the Program Counter + 4.

```
void
branchTakenDetection ()
{
  branchDetected = pcSrc;
}
```

5.3.5 Interrupts

The same mechanism as used for mispredicted branches can be used, but this time the deasserting of the control lines is caused by the interrupt. The execute stage has to be rewritten to retain a copy of the address of the instruction that caused the interrupt, so program execution can be restarted after the interrupt is serviced. The fetch stage has to be rewritten to allow selection of the interrupt service routine address if an interrupt has occured.

```
void
interruptOccurredDetection ()
{
  interruptOccurred = 'set if an interrupt occurred';
}
```

5.3.6 Putting it All Together

The simulator framework now consists of a loop to call the pipe stages in reverse order, followed by a call to the control units to determine the state of the control lines. The control lines are variables local to `oneCycle`. They are determined at the end of cycle n and used by the stages during cycle $n + 1$.

```
void
oneCycle()
{
  n = N - 1;
  while (n >= 0) {
    'call stage n';
    n = n - 1;
  }
  hazardDetectionUnit();
  forwardingUnit();
  brancTakenDetection();
  interruptOccuredDetection();
}
```

Chapter 6

The Language

"Language is only the instrument of science,
and words are but the signs of ideas."
Samuel Johnson.

In the previous chapter it is shown that it is possible to describe the behavior of a pipelined implementation by describing the behavior of the pipe stages and various control units. The description language used was a C-like language. SIMDES bears a strong resemblance in functionality to C. This chapter presents the language constructs which have to be added to SIMDES so that the behavior of a parallel implementation can be described.

The complete SIMDES grammar and production rules can be found in Appendix A. In SIMDES the behavior of the instructions is described in order to be able to simulate the behavior of the processor. In PSIMDES this is completely replaced by the behavioral description of the implementation. This means that there are elements in SIMDES that do not have to be included in PSIMDES. A prime example is the `instruction_block`, since in PSIMDES the behavior of the instructions is indirectly specified by describing the behavior of the hardware.

When designing SIMDES a modular approach was chosen. In SIMDES a description consists of modules or blocks. This facilitates expansion of the language with new constructs. Blocks not needed by one particular tool can be discarded by its parser. This chapter only deals with what has to be added to the language, not with what can be omitted. The extensions to SIMDES are given in the same style

and notation as the SIMDES grammar and production rules in the appendix.

6.1 Pipeline Registers

As shown in Chapter 5 the pipeline registers are kept identical for all stages in order to facilitate simulation.

```
pipe_register           : |[ 'piperegister' : registers
                                              controllines ]|
controllines            : |[ 'controllines' : controlline_declarations ]|
controlline_declarations : controlline_declaration
controlline_declarations : controlline_declaration ; controlline_declarations
controlline_declaration  :
controlline_declaration  : identifiers designator1 { Int }
```

The production rules for `registers` remain unchanged. The above syntax declares a pipeline register containing registers and control lines for the main control unit. These registers and control lines are a union (in the mathematical sense of the word, not a C union) of all information that could possibly be needed in the communication between any two pipe stages, as explained in Chapter 5.

For the example architecture of the previous chapter the declaration of the pipeline register could look like this:

```
|[ piperegister :
    |[ registers : programCounter{32};
                   instruction{32};
                   readData1{5};          # read register number
                   readData2{5};
                   signExtended{32};
                   writeRegisterRt{5};
                   writeRegisterRd{5};
                   writeRegister{5};
                   addResult{32};
                   zero{1};
                   aluResult{32};
                   readData{32};
                   writeData{32}; ]|
    |[ controllines : regWrite{1};
                      aluSrc{1};
                      aluOp{2};
                      regDst{1};
```

```
                    memRead{1};
                    memWrite{1};
                    branch{1};
                    pcSrc{1};
                    memtoReg{1}; ]|
]|
```

6.2 Pipe Stages

The block `pipestages` contains the behavioral descriptions of the pipestages. The stages are described by means of SIMDES functions without any parameters and with return type `void`. The number of stages is arbitrary. The order in which the stages are defined is the order in which the stages are called. If there are N stages, the first function describes stage N, the second describes stage $N-1$, etc. Finally, the last function describes stage 1.

```
pipe_stages          : |[ 'pipestages' : stages ]|
stages               : stage stages
stages               :
stage                : function
```

Each stage has a pipeline register associated with it. Stage n can read a field from the pipeline register of the previous pipeline stage $(n-1)$ by reading the value of `in.fieldName`. It can write to a field of pipeline register n by assigning a value to `out.fieldName`.

6.2.1 Main Control

The main control unit is always in the decode stage. It generates the control lines for the pipeline register of the decode stage, according to the type of instruction being processed. Its behavior can be expressed algorithmically. The algorithm specified in the block `main_control` modifies variables that represent the control lines. These can be local to the decode stage. After the main control is processed, the code of the decode stage explicitly writes the control lines into the pipeline register.

```
main_control         : |[ 'maincontrol' : statement_list ]|
```

6.3 Control Units

The control units that deal with the problems specific to the pipelined implementation are defined in the block `controlunits`. Just like the pipestages, they are functions without parameters and with return type `void`.

```
control_units          : |[ 'controlunits' : units ]|
units                  : unit units
units                  :
unit                   : function
```

6.4 One Cycle

The block `onecycle` forms the simulator framework. Together with the block that defines the pipeline registers, it replaces the description of the behavior of the instructions for the sequential implementation.

```
one_cycle              : |[ 'onecycle' : pipe_stages control_units ]|
```

Communication between the pipe stages takes place through the pipeline registers. As described earlier, a stage can only read from the pipeline register of its predecessor and write in its own pipeline register. Only the control units have access to all pipeline registers.

Chapter 7

Conclusions

"It is difficult to say what is impossible,
for the dream of yesterday is the hope of today
and the reality of tomorrow."
Robert H. Goddard

7.1 Effect on the Model

The assumption that pipelined architectures can be described and simulated at virtually the same level of abstraction as sequential architectures by '*adding a little bit of information about the implementation*' is misguided. Description of a parallel architecture's behavior requires an approach that is completely different from the one used for sequential architectures. A description of the behavior of the implementation rather than a description of the behavior of the instructions. Since the design of pipelined architectures is very complicated, it is very hard to devise a generic model at Level 1 which covers all the possible strategies to cope with all the complications involved.

The first section of Chapter 3 might give the reader the impression that pipelined architectures are just as easy to model at Level 1 of SIMPROACH as sequential architectures. But the other sections reveal the complications that are involved in pipeline design. Chapter 4 offers a description of the hardware required to solve the most trivial problems in pipeline design for a very limited subset of

an instruction set. In Chapter 5 these solutions are put together in a model. Chapter 6 gives a language for describing an implementation in this model.

A possible way to cope with the complications could be to ignore the implementation and use a simplified model, but as shown in chapter 3, omitting certain implementation details can lead to a severe degradation of the quality of the simulation results. Simulations are intended to gather performance figures. By depriving a designer of the means to implement a high performance pipeline in the simulation environment, the results are handicapped from the start. Depending on the degree of simplification, the performance figures can even be computed without simulation at all.

7.2 Effect on the Language

The only solution that is *guaranteed* to work is to allow for as much implementation detail as necessary to implement all the known strategies that are used to create high performance pipelined architectures. This can only be done well at Level 2 of SIMPROACH. Covering all the complications involved in pipeline design requires such an amount of detail, that the only way to do it right seems to be to use an existing (hardware description) language and spend research time on *how to model pipelines in an existing language* rather than spend it on *designing a new language to describe hardware.*

Arguments in favor of this statement can be found in the literature. Furber [Fur89] reports that Modula was used to simulate the ARM completely before it was implemented. References go back as early as Blaauw [Bla76], who states that "APL is particularly suited [...] since it allows expression at the high architectural level, at the lowest implementation level, and at all levels between." Saying this, he stresses the importance of letting the designer determine the model and the level of abstraction himself.

One of the original design goals was to get as much information as possible about the functional behavior of the architecture out of the existing SIMDES description. The thought behind it was that once a designer takes the decision—based on the results of a functional simulation by a simulator generated by SIMGEN—to continue with a design, he could add information for parallel simulation to the description. Next, simulation on a simulator generated by PSIMGEN would yield information about the parallel execution of instructions.

Unfortunately, this requirement cannot be met. Only the description of the instruction types and their syllables can be reused. The description of the behavior of the instructions—accounting for over 95% of the information in a SIMDES file—must replaced.

A description of the pipestages and the control takes its place. In the suggested description language for pipelined architectures, the description of the pipestages and the control implicitly describes the behavior of the instructions.

The effect of extending the Level 1 simulator generator for pipelined architectures on the description language SIMDES is dramatic. The language turns into a hardware description language. As shown in Chapter 6, adding constructs to describe the pipe stages and the control to the description language makes the language more complicated. Add to this the fact that one year after the completion of SIMGEN no one has yet succeeded in completing the description of a present-day processor in SIMDES and it becomes clear that trying to model a pipelined processor at Level 1 might not be such a good idea.

7.3 Recommendations

The basic idea behind SIMPROACH is good. It covers the transformation of an idea for an architecture into a logical circuit diagram by building a sequence of successively more detailed models.

To be able to model pipelined architectures, the required amount of information about the implementation is so large, that it becomes attractive to describe the architecture in a hardware description language like VHDL. By taking advantage of VHDL's behavioral code, it is possible to simulate at the register transfer level[Jen91].

One caveat here is that VHDL is suitable for event-driven simulation. A pipelined architecture is basically a synchronous system. The dynamic scheduling of the model components which VHDL performs during simulation is not necessary. The overhead for managing the event queue and dispatching the models leads to time consuming compilation simulation runs.

The fact that there are no general systems available for the simulation of pipelined architectures might be explained by the fact that in order to fully describe the ar-

chitecture and to get a realistic impression of the performance of the architecture it is necessary to be able to describe the pipeline strategy in full detail. Usually this is done in a general purpose language, for example C. The cycle level MIPS R2000 simulator 'cycle level SPIM', developed at the University of Wisconsin, is a good example[Lar93, Rog93].

Cycle level SPIM uses many of the ideas found in this report, for example the reverse order evaluation of the pipe stages. The author had the advantage of having the full power of C available to express the solutions to the complications of implementing the MIPS R2000 pipeline. It seems to be a good idea to explore the possibilities of decribing pipelined architectures in a general purpose language and try to find guidelines for such simulators. The resulting type of simulator will have a significant advantage over simulation using VHDL in both compilation time as well as execution time.

Concentration should be shifted to the study of techniques and libraries for simulating, using a language such as C or C++. A C++ class library could be the ideal method for implementing the flexibility, efficiency, and reusability that is needed for SIMPROACH.

Acknowledgements

I wrote this report without any help from anyone, and if you believe that, my friends and I have the ultimate architecture simulation tools we'd like to sell to you.

Carel Braam, Luc Cluitmans, Pieter Schoenmakers, Bruce Watson, Willem Jan Withagen.

The report is dedicated to my grandfather, Johan Waucomont († 16 Oct 1983), who taught me to never leave a job unfinished.

Appendix A

SimDes Grammar and Production rules

The SIMDES grammar can be characterized using the following items:

1. Start symbol
2. Non-terminals
3. Terminals
4. Production rules

1. **Start symbol**

   Start symbol : `description`

2. **Non-terminals**

   Non-terminals in order of occurrence:

   `description`
   `resources`
   `memory`
   `pc`
   `registers`
   `register_declarations`
   `register_declaration`
   `ports`

port_declarations
port_declaration
aliases
alias_declarations
alias_declaration
statistics
statistic_declarations
statistic_declaration
identifiers
rest_identifiers
designator1
initial_values
integer_list
rest_integer_list
implementations
rest_implementations
local_resources
fetchbuffer
function_block
functions
function
parameters
rest_parameters
parameter
return_type
fetch_block
instruction_block
instructions
instruction
opcode
rest_opcode
operand
rest_operand
traphandler_block
locals
local
statement_list
block
statement
alternatives
function_arguments
rest_arguments
designator2

3. Terminals

Terminal symbols which occur in the production rules are handed to the parser by the scanner. These symbols are placed within quotes, except for the Ident and Int symbols.

Terminals:

```
description
resources
memory
PC
ports
registers
aliases
implementation
functions
instructions
traphandler
statistics
fetch
fetchbuffer

if
then
else
for
to
downto
do
while
repeat
until
case
default
return
void

Ident
Int
```

4. Production rules

The SIMDES grammar is a context-free grammar. A non-terminal is replaced by (n)one or more non-terminals or terminals. All rules consist of a non-terminal on the left-hand side of the colon and an optional non-terminal/terminal mix on the right-hand side.

```
non-terminal          : (non-terminal | terminal)*
```

Production rules:

```
/* global description production rules */


description           :
description           : |[ 'description' : Ident      resources
                                           statistics implementations]|


/* production rules resources */


resources             : |[ 'resources' : memory     pc
                                         registers  ports
                                         aliases           ]|

memory                : |[ 'memory' : Ident [ Int ] { Int } ]|

pc                    : |[ 'PC' : Ident { Int } = { Int } ]|

registers             :
registers             : |[ 'registers' : register_declarations ]|
register_declarations : register_declaration
register_declarations : register_declaration ; register_declarations
register_declaration  :
register_declaration  : identifiers designator1 { Int } initial_values

ports                 :
ports                 : |[ 'ports' : port_declarations ]|
port_declarations     : port_declaration
port_declarations     : port_declaration ; port_declarations
port_declaration      :
port_declaration      : identifiers designator1 { Int }

aliases               :
aliases               : |[ 'aliases' : alias_declarations ]|
alias_declarations    : alias_declaration
```

```
alias_declarations     : alias_declaration ; alias_declarations
alias_declaration      :
alias_declaration      : Ident = Ident designator1 { Int }
alias_declaration      : Ident = Ident designator1 { Int .. Int }

identifiers            : Ident rest_identifiers
rest_identifiers       :
rest_identifiers       : , Ident rest_identifiers

designator1            : [ Int ] designator1
designator1            :

initial_values         : = { integer_list }
initial_values         :

integer_list           : Int rest_integer_list
rest_integer_list      :
rest_integer_list      : , Int rest_integer_list


/* statistics global to description */


statistics             :
statistics             : |[ 'statistics' : statistic_declarations ]|
statistic_declarations : statistic_declaration
statistic_declarations : statistic_declaration ; statistic_declarations
statistic_declaration  :
statistic_declaration  : identifiers designator1 { Int }


/* implementation block production rules */


implementations        : implementation rest_implementations
rest_implementations   :
rest_implementations   : implementation rest_implementations

implementation         : |[ 'implementation' : Ident
                                               local_resources
                                               statistics
                                               function_block
                                               fetch_block
                                               instruction_block
                                               traphandler_block ]|
```

```
/* resources local to implementation */


local_resources        : |[ 'resources' : fetchbuffer registers
                                          aliases                ]|

fetchbuffer            : |[ 'fetchbuffer' : Ident [ Int ] { Int } ]|


/* functions local to implementation */


function_block         :
function_block         : |[ 'functions' : functions ]|

functions              :
functions              : function functions
function               : Ident ( parameters ) : return_type
                             { locals | statement_list }

parameters             :
parameters             : parameter rest_parameters
rest_parameters        :
rest_parameters        : , parameter rest_parameters
parameter              : identifiers { Int }

return_type            : { Int }
return_type            : void


/* fetch_block local to implementation */


fetch_block            : |[ 'fetch' : { locals | statement_list } ]|


/* instructions local to implementation */


instruction_block      : |[ 'instructions' : instructions ]|
instructions           :
instructions           : instruction instructions
instruction            : Ident ( opcode |  operand )
                             { locals | statement_list }
instruction            : Ident ( opcode )
                             { locals | statement_list }
```

```
opcode                  : Int rest_opcode
rest_opcode             :
rest_opcode             : , Int rest_opcode

operand                 :
operand                 : Ident { Int } rest_operand
operand                 : Ident { Int .. Int } rest_operand
rest_operand            :
rest_operand            : , Ident { Int } rest_operand
rest_operand            : , Ident { Int .. Int } rest_operand


/* traphandler local to implementation */


traphandler_block       :
traphandler_block       : |[ 'traphandler' : ]|
traphandler_block       : |[ 'traphandler' :
                                { locals | statement_list } ]|


/* locals used to declare locals preceding statement blocks */


locals                  : local ; locals
locals                  : local
local                   :
local                   : identifiers designator1 { Int }


/* From statement_list to statement */


statement_list          : block ; statement_list
statement_list          : block

block                   : { statement_list }
block                   : statement

statement               :
statement               : 'if' expression 'then' block
statement               : 'if' expression 'then' block 'else' block
statement               : 'for' Ident designator2 = expression
                                        'to' expression 'do' block
statement               : 'for' Ident designator2 = expression
                                  'downto' expression 'do' block
```

```
statement              : 'while' expression 'do' block
statement              : 'repeat block 'until' expression
statement              : 'case' expression
                                { alternatives  'default' : block }
statement              : 'return'
statement              : 'return' ( expression )
statement              : Ident ( function_arguments )
statement              : var   = expression
statement              : var  *= expression
statement              : var  /= expression
statement              : var  %= expression
statement              : var  += expression
statement              : var  -= expression
statement              : var  &= expression
statement              : var  ^= expression
statement              : var  |= expression
statement              : var <<= expression
statement              : var >>= expression
statement              : var  ++
statement              : var  --

alternatives           :
alternatives           : Int : block alternatives

function_arguments     :
function_arguments     : expression rest_arguments
rest_arguments         :
rest_arguments         : , expression rest_arguments

designator2            :
designator2            : [ expression ] designator2

var                    : Ident designator2
var                    : Ident designator2 { expression }
var                    : Ident designator2 { expression .. expression }


/* expressions used in statements */


expression             : ( expression )
expression             : Int
expression             : Ident designator2
expression             : Ident ( function_arguments )
```

```
/* ternary operator { slice } */

expression              : expression { expression .. expression }

/* binary operators */

expression              : expression *  expression
expression              : expression /  expression
expression              : expression %  expression
expression              : expression +  expression
expression              : expression -  expression
expression              : expression &  expression
expression              : expression ^  expression
expression              : expression |  expression
expression              : expression == expression
expression              : expression != expression
expression              : expression <  expression
expression              : expression >  expression
expression              : expression <= expression
expression              : expression >= expression
expression              : expression << expression
expression              : expression >> expression

/* unary operator */

expression              : ~expression
```

Bibliography

[BCFZ89] B. Bray, K. Cuderman, M. Flynn, and A. Zimmerman. The computer architect's workbench. In G. X. Ritter, editor, *Proceedings of the IFIP 11th World Computer Congress (San Fransisco)*, pages 509–514. IFIP, Elsevier Science Publishers B.V., 1989.

[Bla76] Gerrit A. Blaauw. *Digital System Implementation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Cor81] P. Corcoran. Simulator generator system. *Proceedings of the IEEE*, 128(2):61–63, March 1981.

[Fur89] Stephen B. Furber. *VLSI RISC Architecture and Organization.* Marcel Dekker, Inc., New York, 1989.

[Hon93] Gaston F. J. A. Honings. *Describing an 80386 using PASS.* Report EB436, Eindhoven University of Technology, 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffmann Publishers, Inc., San Mateo, California, 1990.

[HP93] John L. Hennessy and David A. Patterson. *Computer Organization & Design: The Hardware/Software Interface.* Morgan Kauffmann Publishers, Inc., San Mateo, California, 1993.

[Jen91] Glen Jennings. *Approaches to Register Transfer Modeling of VLSI Systems.* PhD thesis, Lund University, Sweden, February 1991.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1978.

[KT87] Kovvali S. Kumar and James H. Tracey. Modeling and description of processor-based systems with DTMSII. *IEEE Transactions on Computer Aided Design*, CAD-6(1):116–127, January 1987.

[Lar93] James R. Larus. *SPIM S20: A MIPS R2000 Simulator.* anonymous ftp: cs.wisc.edu, University of Wisconsin-Madison, 1993.

[MJ76] Robert A. Mueller and Gearold R. Johnson. A generator for microprocessor assemblers and simulators. *Proceedings of the IEEE*, 64(6):921–931, June 1976.

[Mul90] J. M. Mulder. The computer architect's workbench. In *COMPEURO '90, Proceedings of the 1990 IEEE International Conference on Computer Systems & Software Engineering (Tel-Aviv, Israel, 8-10 May 1990)*, pages 524–525, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Rog93] Anne Rogers. *Cycle Level SPIM.* anonymous ftp: cs.wisc.edu, Princeton University, 1993.

[Sme91] Jean-Paul C. F. H. Smeets. *High Performance Implementation of the C-processor.* Report AIO-EB013, Eindhoven University of Technology, 1991.

[Tak92] Rob Takken. Sequential instruction set generator. Master's thesis, Eindhoven University of Technology, October 1992.

[Wil69] M. V. Wilkes. The growth of interest in microprogramming: A literature survey. *Computing Surveys*, 1(3):139–145, September 1969.