High-level Optimization of Pipeline Design

by

Jennifer P. L. Campbell

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, 2003

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Jennifer P. L. Campbell

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Jennifer P. L. Campbell

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Hardware designers are constantly working to improve circuit performance in an attempt to outdo their competitors and satisfy consumers. Throughput is the average number of instructions that can be processed by a microprocessor in a given time. Pipelining is an optimization technique used ubiquitously for improving circuit throughput.

We describe an automatic method for taking an unpipelined circuit, which has been decomposed into functional units, and pipelining it to optimize the throughput. Using a level of description, which we call the Data Dependency (DD) information, we describe the functional units in terms of the data they *need*, *compute* or *write* to the instruction set architecture (ISA) state. By working at a high level of abstraction, our technique can be used earlier in the design process than existing tools. We automatically add pipeline registers, and present rules based on the data dependencies between functional units that describe how to resolve control and data hazards by adding hazard detection and resolution circuitry. The problem of determining a correct pipeline configuration is modelled as a constraint satisfaction problem (CSP). Reinterpretation is used to perform the clock period calculation for a pipeline configuration. The throughput calculation must take into account delays in instruction processing caused by stall and kill hardware, and the calculation is automatically adjusted in our optimization method for each pipeline configuration. The DLX microprocessor is used as a running example to demonstrate our method. We also apply our optimization process to a simple multiply accumulator circuit (MAC).

Acknowledgements

I would like to thank Nancy Day, my supervisor, for her great ideas, patience, knowledge, and the remarkable amount of time she spent working with me. Nancy, your dedication to teaching is inspirational, and I am very fortunate and proud to be your first MMath student.

Thank you to Andrew Kennings and Peter van Beek, for taking the time to read both my proposal and my thesis. Funding for this work was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Government of Ontario and Nortel Networks through an Ontario Graduate Scholarship in Science and Technology, and the University of Waterloo. The members of the Watform lab helped make my time at UW both interesting and fun. A special thanks to Mark Aagaard for providing the electrical engineering perspective, and to Ann Meade and Jianwei Niu for their friendship.

As always, the support of my family was outstanding during the past two years. Thanks, Mum and Pam, for believing in me and encouraging me to pursue my dreams. I would like to thank Brent for the countless long distance phone calls and for helping me escape from my schoolwork. Thanks, Brent, for always knowing how to make me laugh. To Bull, Wayne, Josh, Mike, Pat, Woz, Rutherford, Skeoch, Morley, and Zero: thanks for providing a great distraction from my schoolwork and for including me as ”one of the boys”.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In 1965, Gordon Moore wrote an article, which began with the statement, “The future of integrated systems is the future of electronics itself” [21]. Thirty-eight years later, Moore’s prediction that integrated circuits would allow home computers to become a reality, has proven true. Circuits are now an integral part of our day-to-day lives and are used in everything from calculators and cell phones to cars. Hardware designers are constantly working to improve circuit performance in an attempt to outdo their competitors and satisfy consumers.

In this thesis, we present an automated method of optimizing the throughput of a circuit. In this chapter we introduce the background concepts, including pipelining circuits and pipeline hazards. Then we provide an overview of our method, list the contributions of this thesis, and discuss related work. We also introduce the DLX microprocessor, which we use as a running example. A summary of this work also appears in [4].

1.1 Pipelining Microprocessors

Throughput is the number of operations processed by a circuit in a given time (usually per clock period). The concept of throughput applies to any type of circuit and we use Hennessy and Patterson's academic DLX microprocessor [11] as an example throughout this thesis. For the throughput of microprocessors, the operations that we are concerned with are instructions. *Pipelining* [3] is commonly used to optimize throughput, by partitioning the function of the circuit into stages, so that multiple instructions can be processed concurrently.

The partitioning of instruction processing is done by introducing state-holding elements, called *pipeline registers* (*delays*), into the pipeline. We demonstrate this process using the DLX. The DLX is decomposed into 5 functional units, which appear in Table 1.1. A *functional unit* is a portion of the circuitry that performs a task, which contributes to the overall objective of processing instructions.

Acronym	Name
IF	Instruction Fetch
ID	Instruction Decode
EX	Instruction Execute
MEM	Memory Access
WB	Writeback

Table 1.1: DLX Functional Units

The *IF* unit fetches an instruction from the instruction memory at the current program counter (*pc*). The *ID* unit reads the values of the source operands from the register file. The third functional unit in the pipeline is *EX* and it performs arithmetic logic (ALU) operations, such as

add and multiply. In the *MEM* unit, memory is accessed (read from or written to). Finally, the *WB* unit writes the new value of the destination register to the register file.

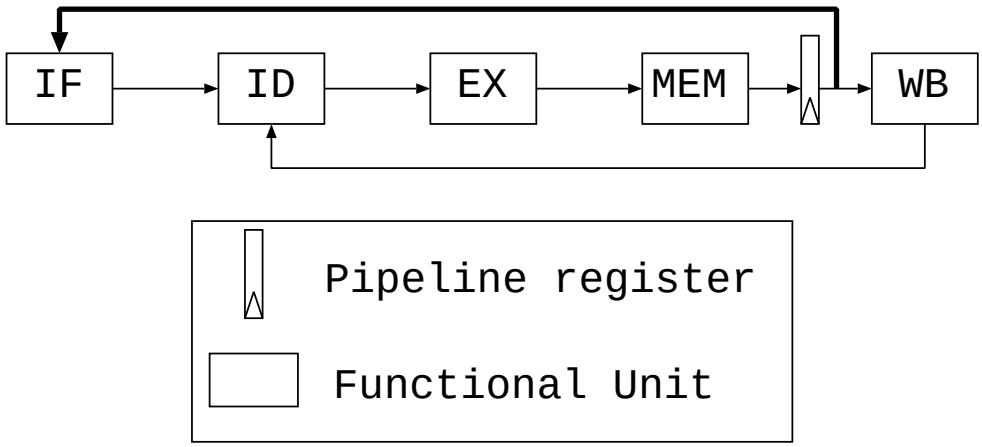


Figure 1.1: Unpipelined DLX

The DLX in Figure 1.1 is *unpipelined*, because there are no pipeline registers separating the functional units, except for the register between *MEM* and *WB*, which is necessary because we are using a write-before-read register file. The legend used in Figure 1.1 applies to the diagrams throughout this thesis. The functional units are represented by rectangular boxes and the wires between them are shown using arrows. The bold line represents the instruction path that carries a new *pc* back to the fetch unit, when a branch occurs.

We call the hardware between 2 pipeline registers a *segment*. The *clock period* (clock cycle) is measured in real time [1] and it is equal to the longest time taken by a segment to process an instruction. In Figure 1.2, the dashed lines show the 3 segments in the unpipelined DLX. There is an implicit pipeline register at the beginning of the pipeline, before the first functional unit.

[1] A common unit of time measurement is nanoseconds, but for generality, in this thesis we use “time units”.

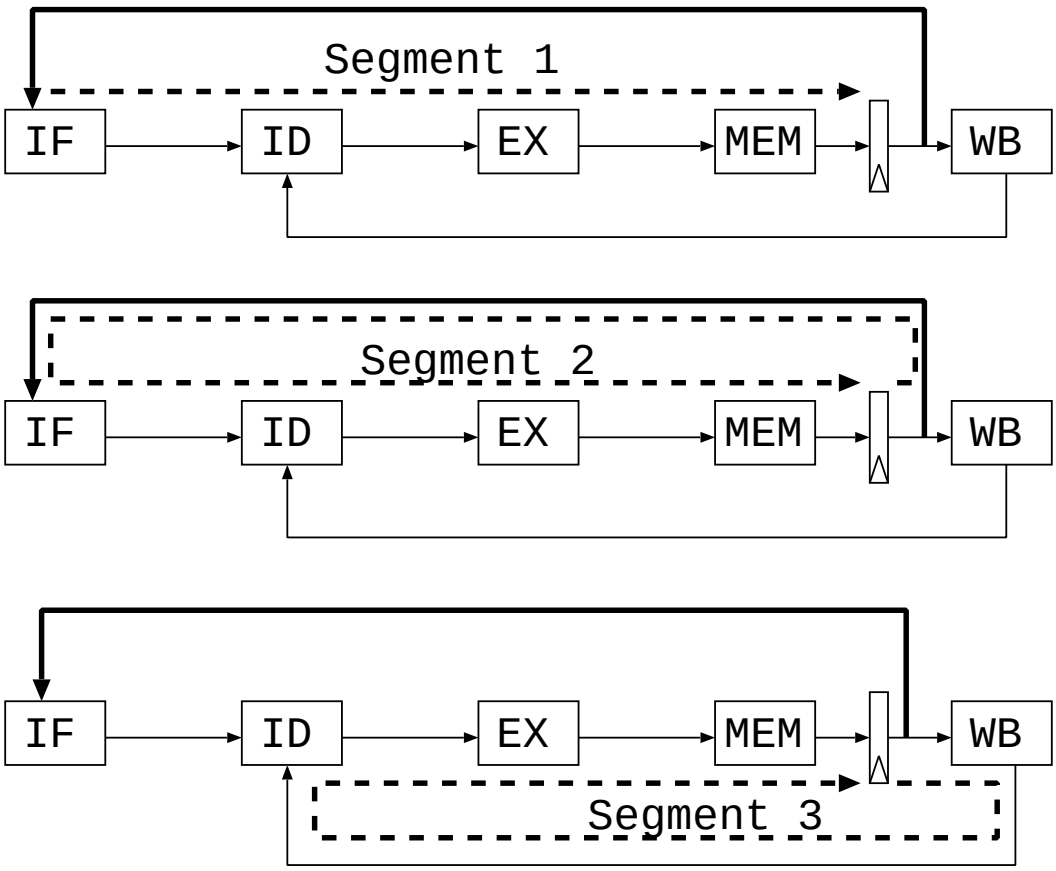
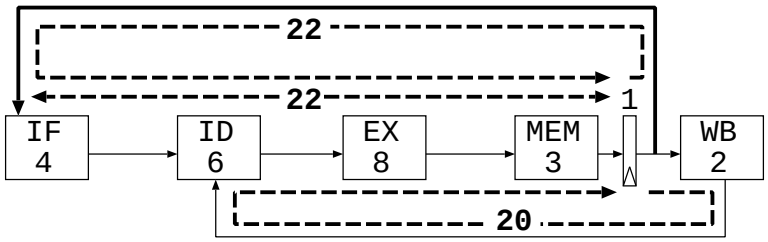


Figure 1.2: Stages in the Unpipelined DLX

Figure 1.3 shows the clock period of the unpipelined DLX using sample execution times (costs) for the functional units. The calculations for the cost of each segment are shown. The cost of Segment 1 and 2 is 22 time units, and cost of Segment 3 is 20 time units, so the clock period is 22 time units (equal to the segment with the highest cost).

In an unpipelined microprocessor, each instruction is entirely processed in a single clock cycle. Figure 1.4 shows how an instruction passes through every functional unit in one cycle. In an unpipelined microprocessor, the throughput is equal to the inverse of the clock period.




```
Segment 1 = IF + ID + EX + MEM + register
          = 4 + 6 + 8 + 3 + 1
          = 22

Segment 2 = IF + ID + EX + MEM  + register
          = 4 + 6 + 8 + 3 + 1
          = 22

Segment 3 = WB + ID + EX + MEM + register
          = 2 + 6 + 8 + 3 + 1
          = 20

Clock Period = Max(Segment 1, Segment 2, Segment 3)
             = 22
```

Figure 1.3: Unpipelined DLX with sample costs

Therefore, decreasing the clock period, increases the throughput. For example, in Figure 1.3, the clock period is 22 and the throughput is 1/22. In other words, 1 instruction is processed every 22 time units.

Pipelining can be used to reduce the clock period. Figure 1.5 contains a partially pipelined DLX and the clock period is reduced to 12, which improves the throughput to 1/12 or 1 instruction every 12 time units. A *fully pipelined* microprocessor has a pipeline register between each of the adjacent functional units. The fully pipelined DLX (Figure 1.6) is a 5-stage pipeline, because the 4 pipeline registers divide the linear datapath into five sections, which are called *stages*. The computation of an instruction is illustrated in Figure 1.7, which shows instructions travelling

Clock Cycle	Instruction 1	Instruction 2	Instruction 3
1	IF ID EX MEM WB		
2		IF ID EX MEM WB	
3			IF ID EX MEM WB

Figure 1.4: Instructions being processed in the unpipelined DLX

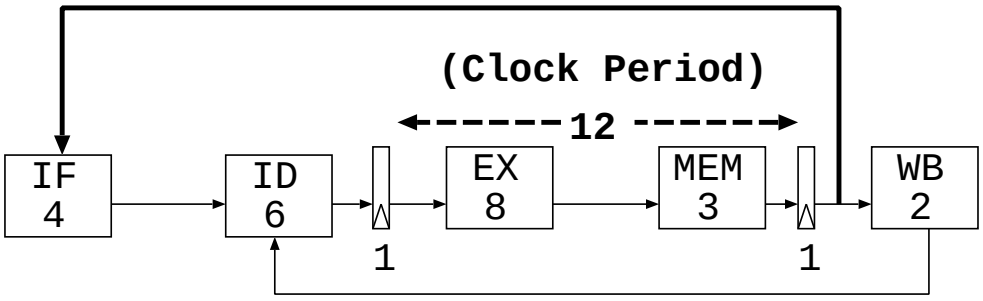


Figure 1.5: Sample circuit partially pipelined with cost

through the stages of the pipeline. "Instruction 1" enters the pipeline and the fetch task is completed by the *IF* unit. In the next clock cycle, the *ID* unit processes "Instruction 1", and the fetch unit processes "Instruction 2". At each clock cycle, a new instruction may enter the pipeline and each instruction already in the pipeline moves to its next stage. When an instruction moves out of the *WB* stage, it has been completely processed. The clock period must be long enough for each stage to complete its computation, which is why the clock period is equal to the segment with the highest cost. Less computation is done in a stage of the pipelined configuration (Figure 1.7) than the unpipelined configuration (Figure 1.4), meaning that the clock period is less.

Based on the examples above, it could be inferred that a microprocessor with pipeline reg-

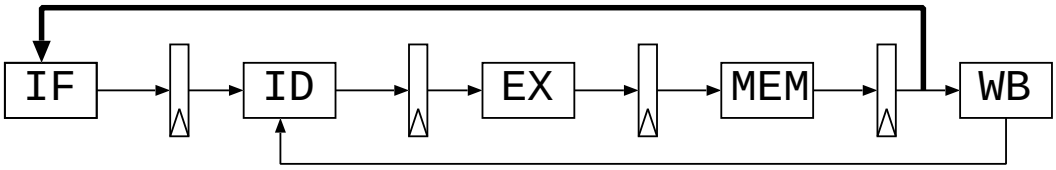


Figure 1.6: Fully-pipelined DLX

Clock Cycle	Instruction 1	Instruction 2	Instruction 3
1	IF		
2	ID	IF	
3	EX	ID	IF
4	MEM	EX	ID
5	WB	MEM	EX
6		WB	MEM
7			WB

Figure 1.7: Instructions being processed in the fully pipelined DLX

isters between every pair of adjacent functional units would have the highest throughput of any configuration. This conclusion is not always valid, because pipelining may introduce hazard situations.

1.1.1 Hazards

Pipelining may introduce *hazard* situations, which occur when the overlapping of execution stages of instructions causes incorrect output. These hazard situations must be detected and resolved, in order to ensure correct output. Hazards arise as a result of data dependencies, instructions that change the *pc*, and resource conflicts [12]. These 3 types of hazards are called data, control and structural, respectively.

For a pipeline to process instructions correctly, hazards must be resolved using control circuitry (bypass, stall or kill hardware). The addition of this control circuitry increases the cost of the stages to which it is added, which means that pipelining may actually decrease clock period or instruction throughput, rather than increase it.

Data Hazards

Data hazards occur when there are data dependencies between instructions. There are 3 types of data hazards: *read-after-write*, *write-after-read* and *write-after-write*. *Read-after-write* hazards occur when an instruction writes data to a register and a subsequent instruction reads an incorrect value for that register. In our work, we address only *read-after-write* hazards, because we assume in-order execution of instructions and we use a write-before-read register file.

Figure 1.8 illustrates a potential read-after-write hazard in the fully pipelined DLX. This fig-

ure shows the incorrect data values in bold. Figure 1.8 shows the situation where the instruction $R3 \leftarrow R1 + R2$ (the sum of *R1* and *R2* is stored in *R3*) is followed by $R4 \leftarrow R3 + R2$. The 2 instructions are shown as they travel through the pipeline with the data values of their source and destination registers in brackets. The register file is also shown with its values at the end of each clock cycle. In the *ID* stage, the instruction is decoded and the values of the source registers for that instruction are retrieved from the register file. The *EX* stage performs ALU operations. When the first instruction is in the *EX* stage, the value of $R3$ is calculated, but the register file will not be updated until the write back (*WB*) phase. This results in a data hazard, because the second instruction is in the *ID* stage and it needs the value of $R3$ (which should be 5), but when it gets the value from the register file, it gets the old value (which is 6). This data hazard causes the result of the second instruction to be incorrect (10 instead of 9), so $R4$ will be assigned an incorrect value and this value is written back to the register file.

This data hazard can be resolved by passing the value of $R3$, which is calculated in the *EX* stage, back to the *ID* stage, using bypass (forwarding) logic. A *bypass* circuit has 2 inputs: the instruction after it has been processed in the current stage and an instruction passed back from a subsequent stage. The bypass examines the instructions to determine whether there is a conflict (dependency). If there is a conflict, then the bypass uses the value of the source register that is being passed back from a subsequent instruction, otherwise it uses the current value (the value in that stage) of that source register.

The pipeline with the hazard circuitry added to fix this data hazard appears in Figure 1.9. This figure does not contain all of the necessary hardware for hazard prevention in the DLX, just the bypass that is used to prevent the data hazard discussed above. Figure 1.10 shows the correct execution for these 2 instructions using the bypass. The result for “Instruction 1” is calculated in the *EX* stage and is passed back to the *ID* stage. When the source registers for “Instruction 2”

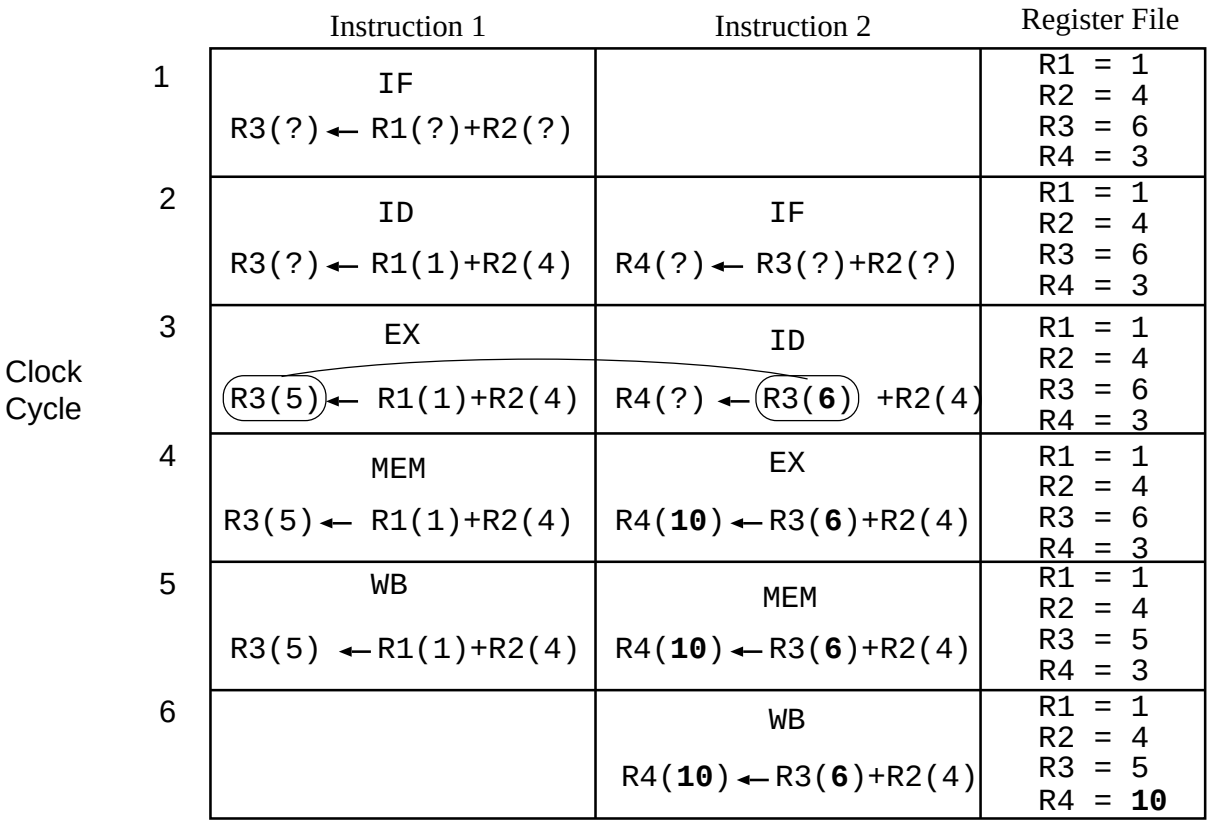


Figure 1.8: Fully pipelined DLX with a data hazard

are decoded, the correct value for *R3* is received by the bypass. Therefore, the correct value of *R4* is calculated by "Instruction 2".

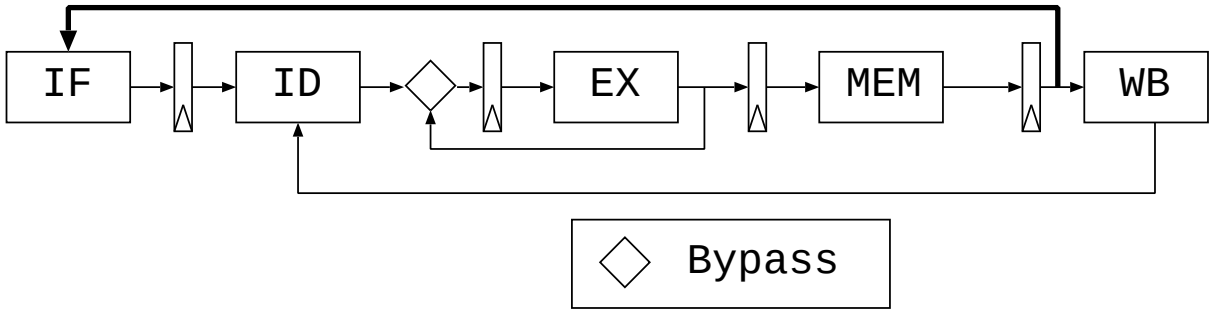


Figure 1.9: Fully pipelined DLX with a bypass

Not all data hazards can be resolved by forwarding; sometimes it is necessary to *stall* the pipeline. Figure 1.11 contains an example that must be resolved using both stall circuitry and a bypass. The instruction $R1 \leftarrow MEM[8 + R2]$ represents *loading* the value at the memory address "8 + val(*R2*)" into register *R1*. In Figure 1.11, the *load* instruction puts a value into *R1* in the *MEM* stage and that value is only written back to the register file in the *WB* stage. We need to stall the pipeline, because the value of *R1* is calculated by "Instruction 1" one stage after "Instruction 2" needs it. We stall the pipeline by inserting a *bubble* into the pipeline. A *bubble* is an instruction that does not perform any tasks, but simply "takes up space" [12] in the pipeline. We can then forward the value back from *MEM* to *ID*.

Figure 1.13 contains the same example with the hazard resolved. In addition to the bypass, 3 pieces of hardware were added to resolve the hazard: detect, stall and selector units. The *detect* unit checks for hazard situations by examining the current and past instructions to see if there is a data dependency (*load* instruction followed by an *ALU* instruction). If there is such a dependency, then the second instruction's status becomes a bubble.

Clock Cycle	Instruction 1	Instruction 2	Register File
1	IF R3(?) ← R1(?)+R2(?)		R1 = 1 R2 = 4 R3 = 6 R4 = 3
2	ID R3(?) ← R1(1)+R2(4)	IF R4(?) ← R3(?)+R2(?)	R1 = 1 R2 = 4 R3 = 6 R4 = 3
3	EX R3(5) ← R1(1)+R2(4)	ID R4(?) ← R3(5)+R2(4)	R1 = 1 R2 = 4 R3 = 6 R4 = 3
4	MEM R3(5) ← R1(1)+R2(4)	EX R4(9) ← R3(6)+R2(4)	R1 = 1 R2 = 4 R3 = 6 R4 = 3
5	WB R3(5) ← R1(1)+R2(4)	MEM R4(9) ← R3(6)+R2(4)	R1 = 1 R2 = 4 R3 = 5 R4 = 3
6		WB R4(9) ← R3(6)+R2(4)	R1 = 1 R2 = 4 R3 = 5 R4 = 9

Figure 1.10: Data hazard resolved by forwarding

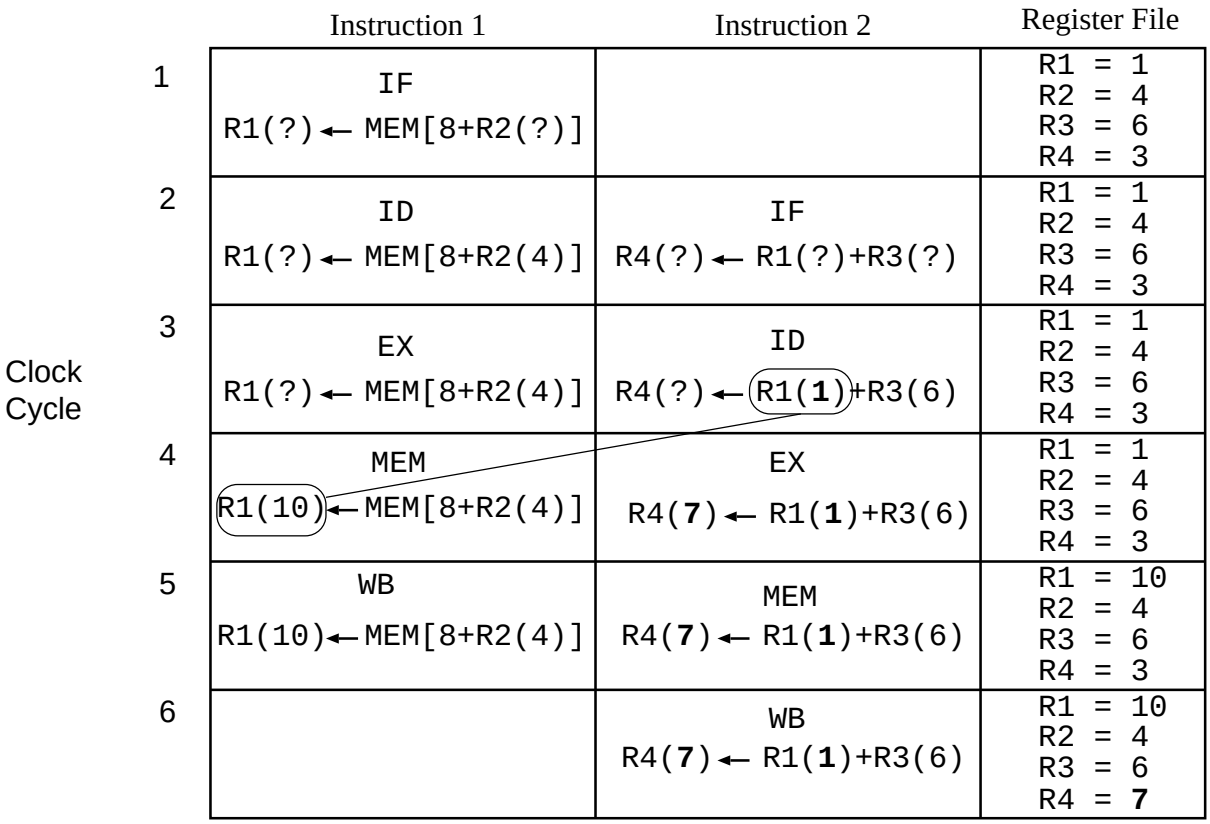


Figure 1.11: Fully pipelined DLX with a data hazard requiring a stall

The signal output from the detect unit is input to the *stall* unit. If this signal is not a bubble, then the stall unit does nothing. However, if this signal is a bubble, then the stall unit causes the instruction that was processed in the last clock cycle to be reprocessed, causing the pipeline to just “sit still”.

The *selector* unit also receives the signal from the detect unit, and it makes the *IF* unit fetch the previous instruction, if the pipeline is stalled. In order to stall the program counter, we must be able to distinguish which functional units fetch the instructions. The input to *IF* is a distinguished signal that is the feedback program counter based on the computation of an instruction [2]. We send our signal to stall the *pc* via this distinguished signal.

Figure 1.12 shows the pipeline with the hazard resolution circuitry and Figure 1.13 shows the corrected execution of the 2 instructions. It takes 7 clock cycles, rather than just 6, to process the 2 instructions. Stalls delay the processing of instructions. Therefore, pipelining may cause the need for hazard resolution hardware, which can actually decrease throughput, rather than increase it. In this case, the throughput calculation must be altered to account for the delay in processing.

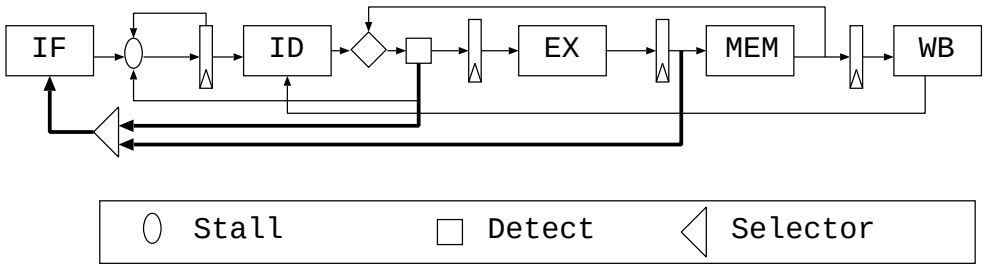


Figure 1.12: Data hazard resolved by forwarding and stalling

[2] In Chapter 2 (page 42), we explain how this signal is distinguished from the others.

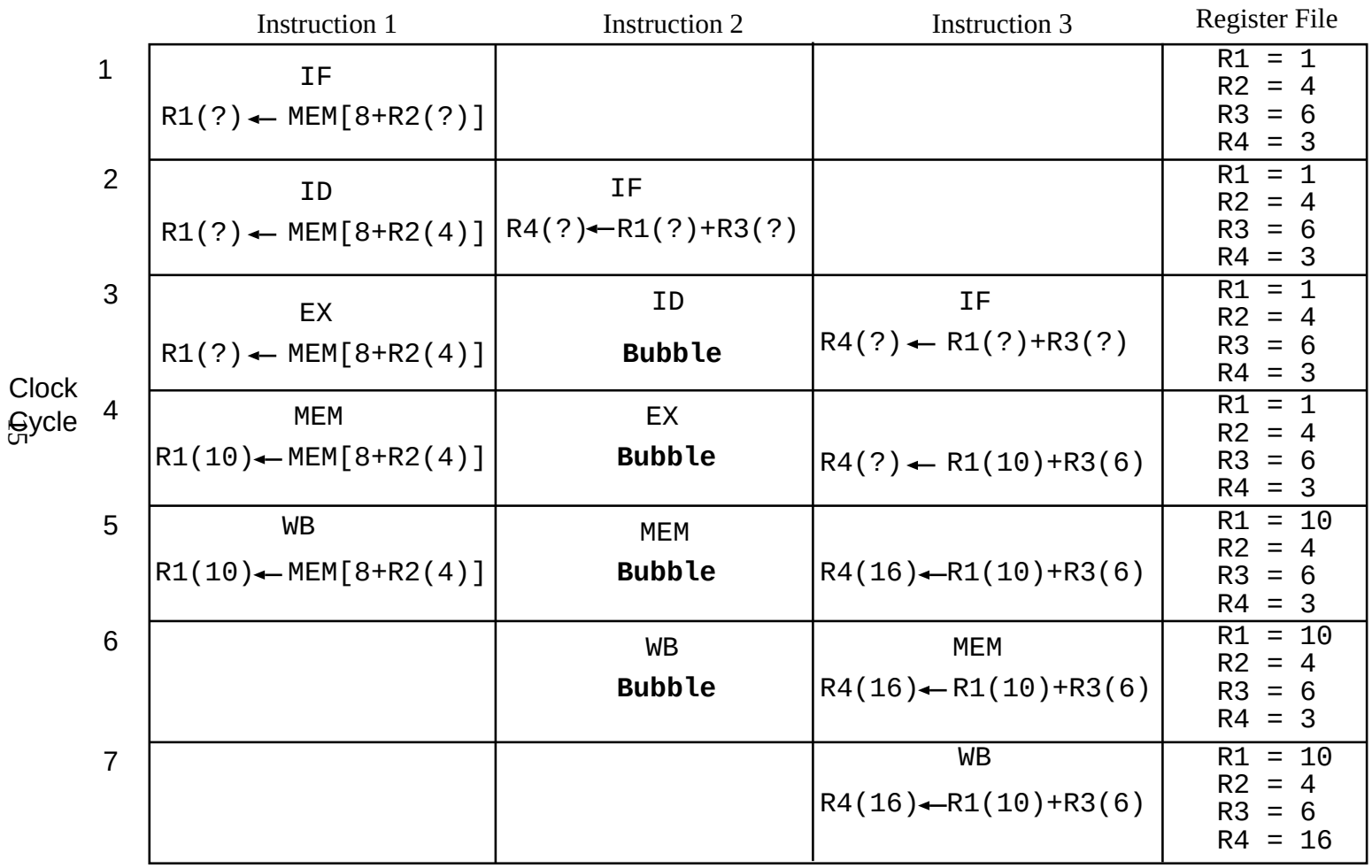


Figure 1.13: Fully pipelined DLX with a stall and bypass

Control Hazards

Control hazards arise when instructions change the *pc*. Unconditional and conditional branches both specify a new *pc*. Conditional branches must satisfy a condition (i.e., $R3 = 0$) in order for the *pc* to be changed, otherwise it is incremented as usual. An unconditional branch instruction specifies the address of the next instruction, so the *pc* must be changed, for the correct next instruction to be retrieved from memory. Addressing can be either *direct* or *relative*.

In order to process branches efficiently, branch prediction schemes are used. A *branch prediction scheme* makes a guess about whether the branch will be taken or not, and based on this guess, subsequent instructions are fetched until the branch condition is calculated. If the branch is incorrectly predicted, then subsequent instruction(s) are incorrectly fetched (using the wrong *pc*). We need to *kill* these instructions in the shadow of a mispredicted branch by making them bubbles, so that they do not have any visible effects.

Killing instructions reduces instruction throughput, so adding pipeline stages that cause control hazards, which are resolved using kills, may decrease throughput. Figure 1.14 contains the fully pipelined DLX with the kill circuitry added to resolve control hazards. The *EX* unit calculates the new *pc* for a branch instruction with relative addressing. The added hazard circuitry includes 2 kill units (to kill the instructions in the previous 2 stages), a delay, and a selector. A *kill* unit has 2 instructions as inputs. The first input is the instruction that is currently flowing through that stage and the second is an instruction that is passed back from a stage that calculated the new *pc*. The kill unit checks to see if the instruction being passed back is a mispredicted branch. If it is, then the other instruction becomes a bubble. The instruction that is passed back to the kill units is also input to the selector through a delay. The instruction passes through the delay, because it changes the *pc* in the next clock cycle. When we are stalling the pipeline and

the detect unit is input to the selector, it does not have to pass through a delay, because the *pc* is changed in the same clock cycle that the stall is detected.

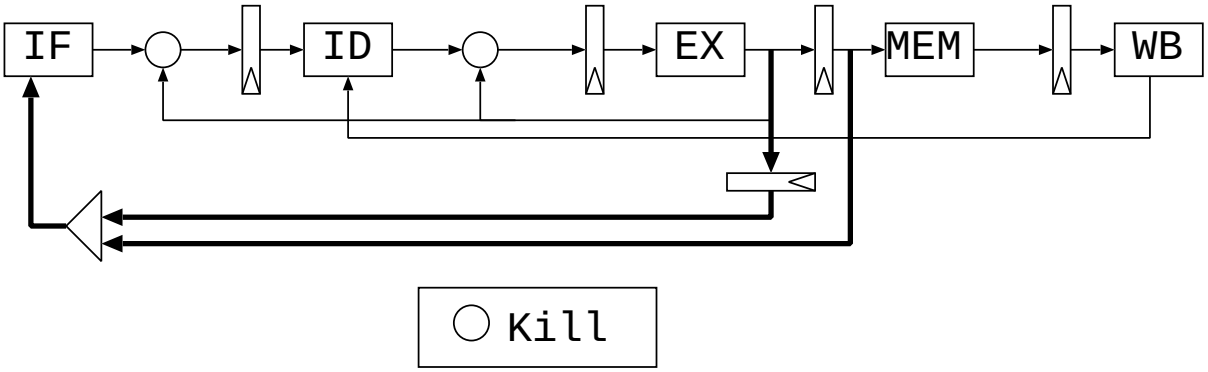


Figure 1.14: Fully pipelined DLX with kill circuitry

Structural Hazards

The third type of hazard is called *structural* and occurs because of hardware resource conflicts. An example of this type of hazard is multiple instructions requiring the use of the register-file write port at the same time [11]. *Structural* hazards are resolved by duplicating hardware or by have a compiler reorder instructions to avoid hazard situations. In our work, we assume structural hazards have already been resolved.

1.2 Thesis Description

Pipelining is a valuable technique that can be used to increase throughput, but determining the optimal pipeline configuration, taking into account hazard resolution circuitry for a given circuit, is a complex task. We use the term *pipeline configuration* to mean a pipeline layout with 0 to *n*

pipeline registers, where there are *n-1* pipeline stages, and the hazards have been resolved. For example, a circuit with 5 functional units in linear order has 16 possible pipeline configurations (including the unpipelined circuit), because there are 4 places where pipeline registers could be placed. To calculate the throughput for each of the 16 configurations, the appropriate hazard resolution circuitry must be added and taken into account in the clock period and throughput calculations.

> **In this thesis, we show that an unpipelined linear datapath, which has been decomposed into functional units, can be automatically pipelined optimally for throughput. If the introduction of pipeline registers causes hazards, then the appropriate hazard resolution hardware can be automatically added and the clock period and throughput calculations adjusted accordingly.**

Next, we provide a brief overview of our pipeline optimization process (Figure 1.15). To pipeline a datapath, we start with one that has been decomposed into functional units (the smallest possible pipeline stages), and pipeline the datapath to increase the throughput. We are not trying to synthesize the datapath itself. We assume that the datapath has already been decomposed into functional units. If adding pipeline registers introduces potential hazard situations, then hazard resolution hardware is added and that cost is taken into account in the throughput calculation.

Our process begins with a textual description of the dataflow of the circuit and timing information for each functional unit in the circuit. The user also provides an abstract view of the pipeline, which characterizes each functional unit in terms of its data dependency (DD) information. The 3 categories of data dependency that we consider are:

- computing data

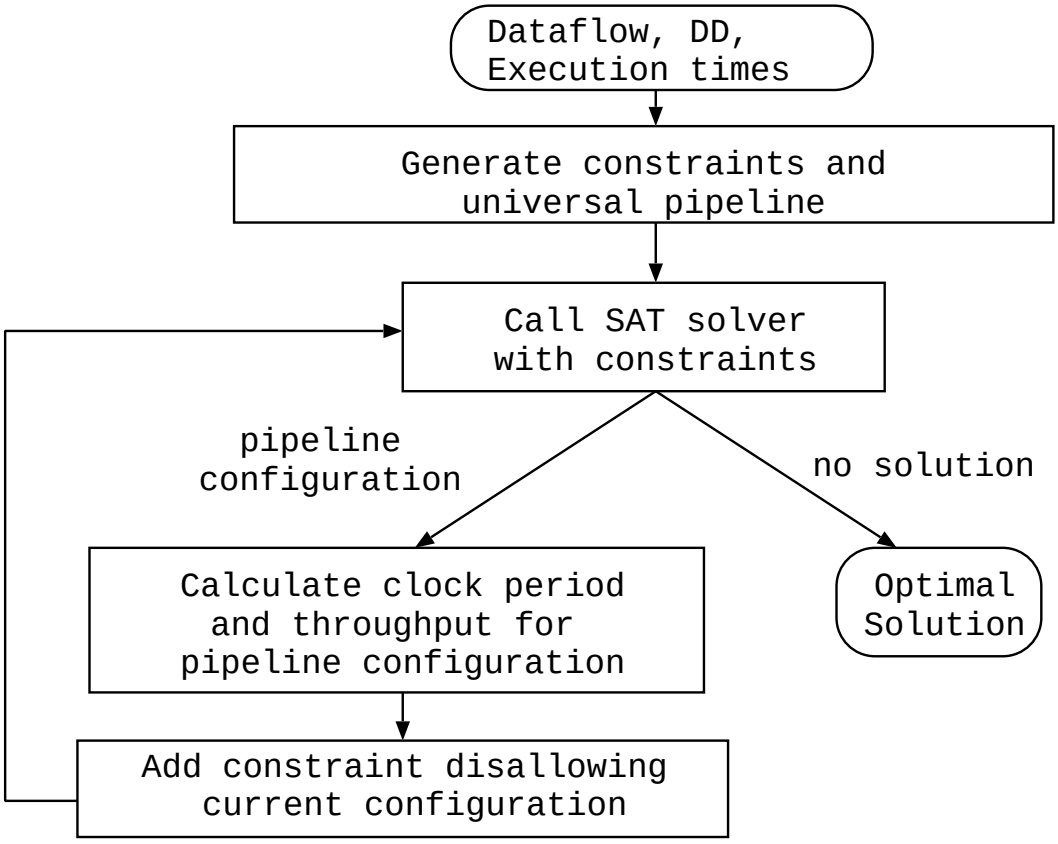


Figure 1.15: Pipeline Optimization Process

- needing data
- writing data

Examples of tasks at this level of abstraction are *computing* the program counter or *writing* data to the register file. This abstract view simplifies the description and analysis of hazards. Using this abstraction, we have written generic rules to detect and resolve hazards. The rules are based only on the location of the stages and the tasks that each stage performs (described using DD information). For a particular datapath, these rules can be instantiated to create a constraint satisfaction problem (CSP) [27, 26]. CSP involves finding solutions to problems consisting of

- a set of variables,
- a finite domain for each variable, and
- constraints on the values the variables can take.

A solution to a CSP problem is an assignment of domain values to the variables, such that no constraint is violated. Our variables are Booleans that represent the presence of absence of pipeline registers and hazard resolution circuitry. A CSP problem that involves only Boolean variables is called a Boolean satisfiability (SAT) problem. In our case, a solution from the SAT solver describes a pipeline configuration with all of its hazards resolved.

The rules are applied to the datapath to generate a *universal pipeline*, which is the union of all possible pipeline configurations. The universal pipeline contains all of the pipeline registers and hazard resolution hardware that could appear in any given pipeline configuration. The presence or absence of hardware is represented by Boolean variables, and a solution from the SAT solver is used to instantiate the universal pipeline to yield a pipeline configuration.

Reinterpretation [22] is the technique we use to calculate the clock period of a pipeline configuration. Reinterpretation determines the clock period using alternate definitions of the functions of the various pipeline components. Using reinterpretation, we can use a single circuit description for calculating the clock period, visualization, verification and simulation. The throughput is calculated using the clock period and information about the potential for stalls and kills in the pipeline configuration.

Our approach is iterative. Once the clock period and throughput have been calculated for a pipeline configuration, then a constraint stating that the solution should not be found again is added to the set of constraints, which are given to the SAT solver. The SAT solver returns another solution, and its clock period and throughput are calculated. This process continues until there are no solutions remaining, which means that all possible pipeline configurations have been considered and we have found the one with the optimal throughput.

The four main contributions of this thesis are:

- the creation of the Data Description abstraction for characterizing hazard situations,
- the development of a set of generic rules for hazard resolution,
- the instantiation of the rules to yield a SAT problem to which the solution is a pipeline configuration, and
- the use of reinterpretation to calculate the clock period and throughput for determining an optimal pipeline configuration.

1.3 Related Work

Although efforts have been made to automate optimization of pipeline layout, this work is often still done manually in industry. Several methods of automating microprocessor pipelining have been investigated, and each of these have varying degrees of usefulness to industry and some have significant weaknesses. Existing tools do not handle hazards and their resolution, which is critical in pipelined circuits such as microprocessors. Currently, pipeline layout is done using timing and area estimation tools to simulate the pipeline [5].

Get2Chip's Pipeline Master Synthesis, and Synopsys' Module and Behavioural Compilers, are both used to perform datapath synthesis [9]. Pipeline Master takes Verilog or Superlog as input and performs optimized pipeline datapath synthesis automatically. Pipeline Master may add hardware to the pipeline to improve performance. The output of the tool is register-transfer level (RTL) or gate-level code. Our tool does not do datapath synthesis and we do not further decompose the functional stages provided as input. We treat the functional units as black boxes, assuming those implementation details can be taken care of using an existing tool, and work to optimize at a higher level of abstraction. The Synopsys Behavioural Compiler is similar to Pipeline Master and it does pipeline rescheduling as well. This tool also applies at a lower level of abstraction than our method. By working at a higher level of abstraction, our technique can be used earlier in the design process than existing tools.

Kroening and Paul presented techniques for automating pipeline design and resolving hazards [14]. They begin with a *prepared sequential machine*, which is a pipeline that has been partitioned into stages and contains duplicate hardware to resolve structural hazards. They apply transformations to this pipeline, which add hardware components, and resolve data and control hazards. The added hardware may lead to a non-optimal implementation in terms of through-

put. This method requires some manual effort. A major difference between this work and our proposed work is that the pipeline we generate will be optimized for throughput.

Hassoun and Ebeling developed a technique called *architectural retiming*, which is used to improve circuit performance. They use *negative registers* to decrease clock period (which increases throughput) without increasing latency (the number of clock cycles it takes to complete the execution of a single instruction). A *negative register* differs from a regular register in that it performs a shift backward in time rather than forward, by using precomputation or prediction to calculate the negative register values. To do precomputation, duplicate hardware may be needed, so a trade-off between area and timing is made. Putting both negative and regular registers in the pipeline is equivalent to placing a wire in the path, because they do not affect latency. Architectural retiming is currently done manually. The negative/positive register pairs that are added do not introduce hazards, so no additional hazard resolution hardware is required, although some hazard circuitry may be present in the original, less efficient pipeline.

Chang and Hu [6] presented a specification style for microprocessors for the design of cycle-accurate simulators. They created an automated tool that can derive simulators from specifications. They assume that the microprocessor has already been pipelined and they are not trying to optimize the pipeline. The hazard resolution (bypass and stall) hardware is automatically derived from the specification. The need for hazard resolution hardware is determined using *context*, which is the state of the processor (register values, *pc* value, etc.) as seen by a particular instruction. When an instruction needs a value, it takes the most recently updated version of that value (using forwarding) and if the value is not available then the pipeline stalls. To simulate a processor, the functional units must be completely specified or, in other words, the tasks that each functional unit performs must be explicitly stated.

Marinescu and Rinard [16, 17] use a similar context-based approach to Chang and Hu. The

goal of their work is to take a set of user-specified modules and output an efficient synchronous (pipelined) circuit. Their pipelines are said to be “efficient”, but that does not necessarily mean optimal. By efficient, they mean more efficient than the original asynchronous unpipelined implementation and comparable to a hand-written Verilog model. Although the goal of their work is similar to ours, their method of achieving it is quite different. Unlike us, they further partition functional units in an effort to reduce clock period and increase throughput.

An alternative approach to using Boolean CSP, would be to use linear programming (LP), which is a technique that is often used in operations research to solve optimization problems [7]. Llewellyn describes linear programming problems as a set of variables with constraints on or between them, and an *objective* to attain [15]. This objective usually involves minimizing or maximizing a linear function. Our problem involves maximizing the instruction throughput, but LP cannot be used in our work, because the throughput calculation depends on the configuration of the pipeline. Hazard resolution hardware may delay the execution of instructions. Therefore, the throughput calculation varies with the different pipeline configurations.

Weinhardt states that *custom computing machines* combine software with hardware to make the program more efficient [28]. Weinhardt automates the generation of hardware accelerators from a sequential software program. The resulting hardware components can sometimes be pipelined to improve efficiency, and integer linear programming (ILP) is used to automate this task. ILP is a form of linear programming, although only integer variables can be used in ILP. ILP involves instantiating a set of constrained integer variables in order to minimize a cost function. The goal of Weinhardt’s ILP problem is maximizing pipeline throughput, while minimizing the number of field-programmable gate array (FPGA) flip-flops. The program determines the quantity and location of flip-flops. This problem is different from ours because hazards are not resolved using stalls. If stalls are not used for hazard resolution, then the throghput calculation

is static.

Although we are tackling a different problem, our method works at the same level of abstraction as the microprocessor descriptions of the Hawk hardware description language [20]. This abstraction level views the pipeline as having instructions, the register file or memory flowing along the wires between functional units. Matthews and Launchbury present an elementary microarchitecture algebra, which can be used to unpipeline a given pipelined datapath [19]. The problem that they address is the opposite of what we are trying to do, and the goal of their work is verification. Initially, the pipeline contains hazard resolution hardware including bypass logic. The algebraic rules remove the pipe stages and the hazard resolution hardware. The rules are local and there is not a systematic method of applying the rules in order to unpipeline efficiently the datapath. In fact, the rules could be applied infinitely.

1.4 Thesis Outline

Chapter 2 describes our rules for the inclusion of hazard resolution circuitry based on the DD information about the circuit and the grouping of the functional units into stages. It contains definitions and examples of our 5 placement rules for hazard resolution circuitry. We describe how the rules can be used to generate a set of Boolean constraints as a problem for a SAT solver that will result in a possible pipeline configuration.

Chapter 3 contains a description of our throughput optimization. We describe our use of reinterpretation to calculate the clock period. We also show how hazard resolution hardware that delays the processing of an instruction alters the throughput calculation. Chapter 3 also includes implementation details of our throughput optimization method.

Chapter 4 contains the results of the 2 case studies that we performed on the DLX and a multiply accumulator (MAC). In Chapter 5, we conclude and discuss the limitations of our work and our ideas for future work.

Chapter 2

Hazard Resolution Rules

In this chapter, we describe our rules to resolve read-after-write data hazards and control hazards. Our rules depend on information about the data dependencies among the functional units of the circuit. We introduce the data dependency (DD) information used to capture this abstraction. We use Hennessy and Patterson's DLX microprocessor [11] as a running example and show all possible pipeline configurations of the DLX with the required hazard detection and resolution circuitry.

The hazard resolution rules are a key part of our optimization process. We use our rules to generate constraints and create a Boolean satisfiability problem for determining possible pipeline configurations, including hazard circuitry.

2.1 DLX: ISA State

Figure 2.1 contains the DLX microprocessor that we introduced in Chapter 1. In Figure 2.2, we partition the original DLX functional units to expose all state-holding elements, which in

this case are just the elements of the instruction set architecture (ISA) state, which includes the program counter (*pc*), the register file (*rf*) and memory (*mem*). From now on, all references to the DLX will be to the ISA state representation. Our method of calculating the clock period considers the amount of computation between all registers (not just pipeline registers), so it is dependent on ISA state. The dashed lines in Figure 2.2 show the partitioning of the ISA state diagram into the corresponding functional units in Figure 2.1.

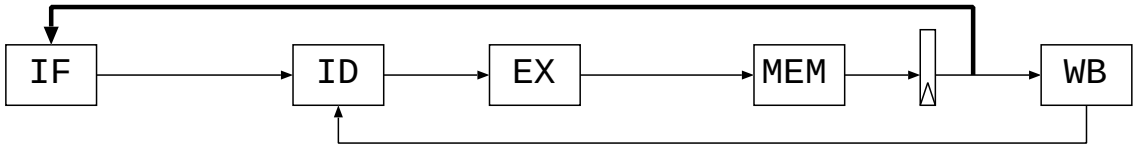


Figure 2.1: Unpipelined DLX Microprocessor

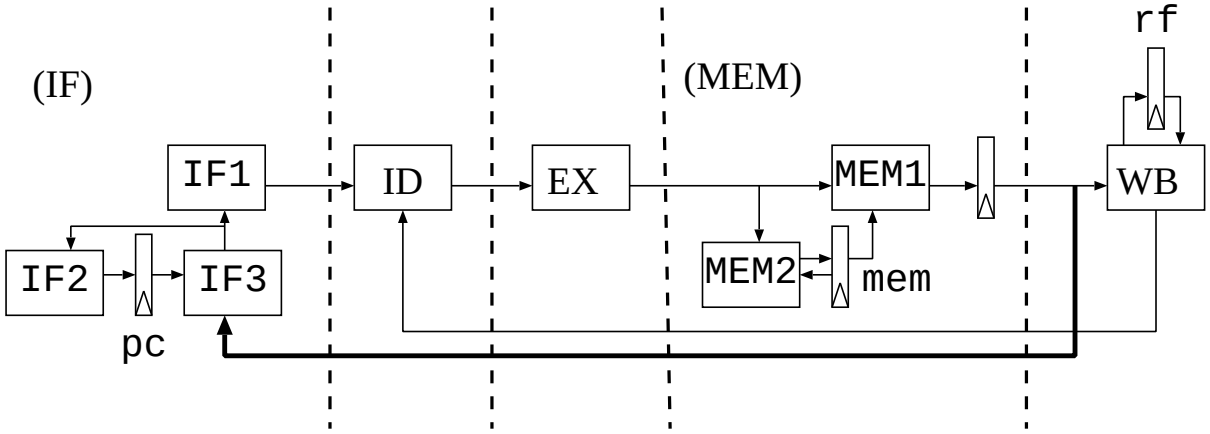


Figure 2.2: Unpipelined DLX Microprocessor (ISA state)

2.2 Data Dependencies (DD)

One contribution of this thesis is finding an appropriate level of abstraction to describe the characteristics of a pipeline that cause hazards. We call this description level the data dependency (DD) information. DD is similar to a def-use relation among the functional units. It is used to describe the data that is *computed*, *needed*, or *written* by each functional unit. The DD information includes only data that is affected by multiple functional units. When more than one functional unit manipulates data there may be a dependency that could result in incorrect output. By data, we mean the ISA state (program counter, register values or memory).

Using this information, we have created generic rules that indicate where hazard detection and resolution hardware should be placed, depending on the pipelining of the datapath. These rules are described in Section 2.5 and they are used to generate constraints that describe the situations in which hazard resolution circuitry is needed.

The DD representation of the DLX appears in Figure 2.3. The data called *regVal* represents a register value other than the *pc*. The DD information for the DLX was determined by considering the tasks that each functional unit performs. For example, the *ID* stage reads the value of the source registers from the register file. Therefore, it *needs* data *regVal*. The *ID computes* the *pc*, because this unit is where the *pc* of an unconditional branch with direct addressing is calculated. The *EX* stage also *computes* the *pc*, because the destination *pc* of branches using relative addressing is calculated here.

There can be slight variations in the functionality of the DLX that result in different DD information. If the branch address calculation is moved to the *ID*, then the *EX* would no longer *compute* the *pc*, resulting in a different set of DD information for the DLX. We use the DD information in Figure 2.3 for our DLX example.

```
Needs(IF3, pc)
Needs(ID, regVal)
Computes (ID, pc)
Computes(EX, regVal)
Computes (EX, pc)
Computes(MEM1, regVal)
Writes(WB, regVal)
```

Figure 2.3: DLX DD Information

2.3 SAT Constraints

Our optimization process uses constraint satisfaction programming (CSP) to perform the search for a pipeline configuration. We use our rules to generate constraints for a Boolean satisfiability (SAT) solver. Given a set of Boolean variables and constraints on those variables, the SAT solver finds a satisfying solution.

In our problem, we use Boolean variables to represent the presence or absence of the components of a pipeline. Components include pipeline registers and hazard resolution circuitry. When we apply rules to a pipeline, we use the DD information to generate constraints on the components involved. In the following sections, we describe each hazard resolution rule. We also illustrate the Boolean constraints generated from the application of each rule.

Throughout the remainder of this chapter, a set of symbols will be used to represent the Boolean operators. These symbols appear in Table 2.1 in the order of precedence used in expressions (e.g., $\neg$ has the highest precedence).

Boolean Operator	Symbol
FOR ALL	$\forall$
NOT	$\neg$
AND	$\wedge$
OR	$\vee$
IMPLIES	$\rightarrow$
IFF	$\leftrightarrow$

Table 2.1: Boolean Operators

2.4 Naming Conventions

In this chapter, we discuss the addition of hazard resolution circuitry into pipelines. In some pipelines there may be multiple bypass, stall, kill or selector units. We use the naming conventions in Table 2.2 to name the added control hardware. The names are dependent on the location of the hardware in relation to the functional units. When we use the term *between X and Y* in the table, we mean that X comes before Y on the datapath. When we say *X back to Y*, we mean that X comes after Y on the datapath. The names are dependent not only on location, but also on the direction of the data flow. These names are used as the names of the Boolean variables representing the presence or absence of a hardware component.

2.5 Hazard Resolution Rules

This section contains the definition of our 5 hazard resolution rules. Each rule is explained using an example occurrence in the DLX pipeline. The rules appear in Figure 2.4. All of the hazard

Situation	Name
Pipeline register between functional units X and Y	XY
Bypass from X back to Y	XBypY
Stall between X and Y	XStY
Detect from X back to Y	XDetY
Kill from X back to Y	XKillY
Select between X and Y	XSelY

Table 2.2: Hardware Naming Conventions

resolution rules are written in terms of the DD information about functional units, the stages that the functional units are in, and the distance between stages. Our rules are expressed using characteristic functions. If the statement "Stage(*A*, *i*)" is true, then functional unit *A* is *in* stage *i*. The predicates Needs, Computes and Writes refer to DD information. The expression "Earlier(*A*, *B*)" means that *A* is *earlier* in the dataflow than *B*. The "Earlier" relation is a partial order. The variables representing functional units, stages and data are all universally quantified.

If the conditions for the rules are met, then hazard resolution circuitry is needed. For Rules 1, 2, and 3, the hazard resolution circuitry is expressed in terms of the functional units involved. For Rules 4 and 5, the hazard resolution circuitry required is described in terms of stages, because stalling and killing are associated with stages rather than functional units.

2.5.1 Bypass Rules: Rules 1, 2 and 3

As we saw in the introductory chapter, bypasses are used to resolve some hazards caused by read-after-write data dependencies. Bypasses forward data from a functional unit to a preceding

- **Rule 1 (byp)**:

  $\text{Stage}(A, i) \wedge \text{Stage}(B, i+1) \wedge \text{Needs}(A, x) \wedge \text{Computes}(B, x)$

  $\wedge\ (\forall C.\ \text{Stage}(C, i+1) \wedge \text{Earlier}(B, C) \rightarrow \neg\text{Writes}(C, x))$

  $\wedge\ (x \neq \textit{pc})$

  $\leftrightarrow\ \text{forward}(B, A, x)$

- **Rule 2 (byp)**:

  $\text{Stage}(A, i) \wedge \text{Stage}(B, i+y) \wedge \text{Needs}(A, x) \wedge \text{Computes}(B, x)$

  $\wedge\ (\forall C.\text{Stage}(C, i+y) \wedge \text{Earlier}(B, C) \rightarrow \neg\text{Writes}(C, x))$

  $\wedge\ x \neq \textit{pc} \wedge y \geq 2$

  $\leftrightarrow\ \text{forward}(B, A, x)$

- **Rule 3 (byp)**:

  $\text{Stage}(A, i) \wedge \text{Stage}(B, i) \wedge \text{Earlier}(A, B) \wedge \text{Needs}(A, x) \wedge \text{Computes}(B, x)$

  $\wedge\ (\forall C.\ \text{Stage}(C, i) \wedge \text{Earlier}(B, C) \vee \text{Stage}(C, i+1) \rightarrow \neg\text{Writes}(C, x))$

  $\wedge\ x \neq \textit{pc}$

  $\leftrightarrow\ \text{delayedForward}(B, A, x)$

- **Rule 4 (stall)**:

  $\text{Stage}(A, i) \wedge \text{Stage}(B, j) \wedge \text{Stage}(C, j+y)$

  $\wedge\ \text{Needs}(A, pc) \wedge \text{Needs}(B, x) \wedge \text{Computes}(C, x)$

  $\wedge\ i \leq j \wedge y \geq 2$

  $\leftrightarrow\ \text{stall}(i) \wedge \text{stallpc}$

- **Rule 5 (kill):**

  $\text{Stage}(A, i) \wedge \text{Stage}(B, i+x) \wedge \text{Needs}(A, pc) \wedge \text{Computes}(B, pc) \wedge x > 0$

  $\leftrightarrow\ \text{kill}(i+x) \wedge \text{kill}(i+x-1) \wedge ... \wedge \text{kill}(i) \wedge \text{fixpc}$

Figure 2.4: Hazard Resolution Rules

functional unit, which *needs* the data before it has been *written* to the ISA state.

Rule 1

Rule 1 states that a bypass is required when functional unit *A* in stage *i needs* data that functional unit *B* in stage $i+1$ *computes*, but does not *write* back to ISA state. A bypass is needed in the DLX when *ID* and *EX* are in different stages, and *EX* is not in the same stage as *WB*. The stage with *ID needs* a value that is *computed* in *EX*, but not *written* to ISA state until the stage with *WB*. A bypass is used to pass back the data from the stage with *EX* to the stage with *ID*, which needs it in order to ensure that the proper value of the source register is present at the end of the stage with *ID*. If the bypass is not present, then the incorrect value of the source register will be retrieved from the register file by *ID* and it will be passed on to the subsequent stage. An example of this type of hazard is in Chapter 1 (page 10). This hazard is caused by a data dependency between *ID*, which is in stage *1*, and *EX* in stage *2*. An instantiation of Rule 1 for the DLX DD information is:

$$
\begin{aligned}
&\textbf{Stage}(ID, 1) \wedge \textbf{Stage}(EX, 2) \\
&\wedge \textbf{Needs}(ID, regVal) \wedge \textbf{Computes}(EX, regVal) \\
&\wedge (\textbf{Stage}(EX, 2) \wedge \textbf{Earlier}(EX, EX) \rightarrow \neg \textbf{Writes}(EX, regVal)) \\
&\wedge (regVal \neq pc) \\
&\leftrightarrow \textbf{forward}(EX, ID, regVal)
\end{aligned}
$$

This instance of Rule 1 introduces a bypass, when there are 2 pipeline registers present: *IDEX* and *EXMEM1*. The presence of the *IDEX* register means that *EX* is in the stage immediately following the stage with *ID*, satisfying Stage*(ID, 1)* and Stage*(EX, 2)*. The DLX DD information

states that the *needs-computes-writes* dependency component of the rule is satisfied. The presence of *EXMEM1* ensures that *regVal* is not *written* by *EX* or functional units later than *EX* in stage 2, satisfying the universally quantified portion of the rule. There are no other functional units in stage 2. This instance of Rule 1 appears in Figure 2.5, with the optional components appearing with dashed lines. These dashed-line components are either all present or all absent, meaning that *EXBypID* is present if and only if *IDEX* and *EXMEM1* are present.

The Boolean constraint that is generated for this example instance of Rule 1 is:

IDEX $\wedge$ **EXMEM1** $\leftrightarrow$ **EXBypID**

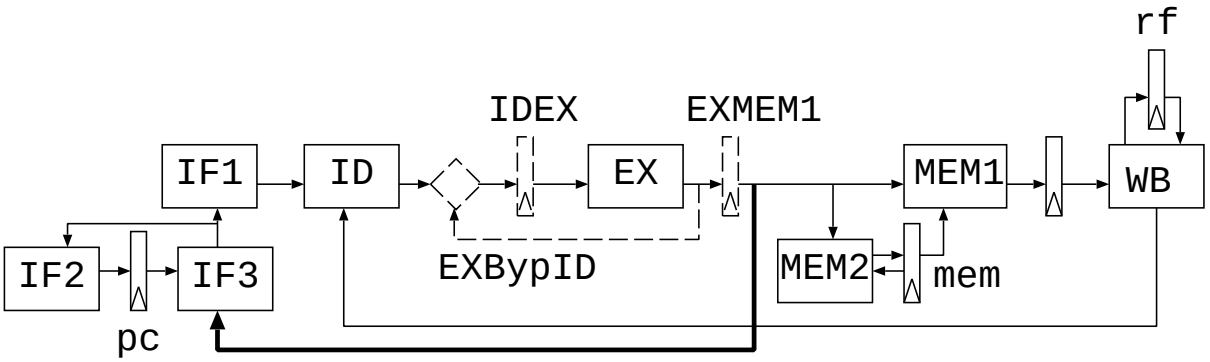


Figure 2.5: Example Instance of Rule 1

The presence of *EXMEM1* is not always necessary to ensure that *WB* is not in the same stage as *EX*. An alternative instantiation of this rule requires that *EXMEM1* is absent, and *MEM1WB* is present, and the data would then be forwarded from *MEM1* to *ID*.

Rule 2

The second bypass rule, applies in situations when the *need* for a value and its *computation* (without a *write* to ISA state) are separated by 2 or more stages. In the DLX, Rule 2 applies to *ID* and *MEM1* when there are pipeline registers between *ID* and *EX* (*IDEX*), *EX* and *MEM1* (*EXMEM1*), and *MEM1* and *WB* (*MEM1WB*). An example instance of Rule 2 is the following:

$$\textbf{Stage}(ID, 1) \wedge \textbf{Stage}(MEM1, 3)$$
$$\wedge \textbf{Needs}(ID, regVal) \wedge \textbf{Computes}(MEM1, regVal)$$
$$\wedge (\textbf{Stage}(MEM1, 3) \wedge \textbf{Earlier}(MEM1, MEM1) \rightarrow \neg\textbf{Writes}(MEM1, regVal))$$
$$\wedge regVal \neq pc \wedge 2 \geq 2$$
$$\leftrightarrow \textbf{forward}(MEM1, ID, regVal)$$

The Boolean constraint that is generated for the application of Rule 2 on this example instance involves 3 pipeline registers:

IDEX $\wedge$ **EXMEM1** $\wedge$ **MEM1WB** $\leftrightarrow$ **EXBypID**

An illustration of this instance of the rule appears in Figure 2.6. The presence of the first 2 registers ensures that *ID*, which *needs* data, and *MEM1*, which *computes* data, are 2 stages apart (as the rule requires). The last register, *MEM1WB*, is needed to guarantee that *MEM1* and *WB* are in different stages, so that the universally quantified condition that data is not *written* by the functional unit *MEM1* or any functional units later than *MEM1* in stage *3* is met. There are no other functional units in stage 3. *MEM1WB* is always present because we are using a write-before-read register file, so a constraint stating that *MEM1WB* is true is added to the set of constraints.

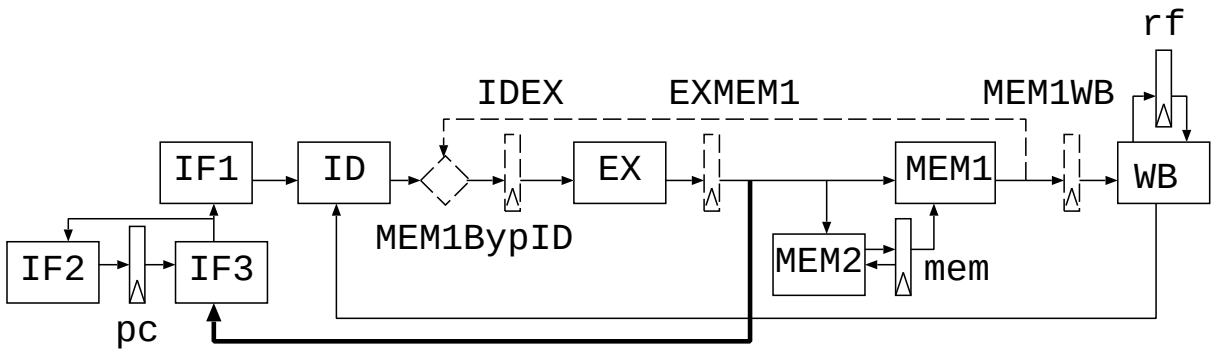


Figure 2.6: Example Instance of Rule 2

This rule does not completely resolve the hazard. In Chapter 1 (page 14), we saw an example of a data hazard that requires both a bypass and stall to resolve it. The example above is the same situation, so the bypass alone will not fix the data hazard. Later in this chapter, Rule 4 is introduced and it will provide the remaining hardware that is needed to resolve this hazard.

Rule 3

The last bypass rule involves a *needs-computes* dependency within a single pipeline stage that consists of multiple functional units, but the data value is not *written* until a later stage. For example, if the *ID* and *EX* units are in the same pipeline stage, meaning that there is no pipeline register, *IDEX*, between them, then a dependency exists because the *ID* unit *needs* data and the *EX* unit *computes* data after the instruction has passed through the *ID* stage. The universally quantified portion of the rule states that *EX*, functional units later than *EX* in stage *1*, and functional units in stage *2* cannot *write* data. Therefore, to satisfy the rule neither *EX* (which is in stage *1*) nor *MEM1* (which is in stage *2*) can *write* data. The data that *EX* computes must be passed back to the *ID* unit to resolve the hazard. An example occurrence of Rule 3 is:

$$\textbf{Stage}(ID, 1) \wedge \textbf{Stage}(EX, 1) \wedge \textbf{Earlier}(ID, EX)$$
$$\wedge \textbf{Needs}(ID, regVal) \wedge \textbf{Computes}(EX, regVal)$$
$$\wedge (\textbf{Stage}(EX, 1) \wedge \textbf{Earlier}(EX, EX) \rightarrow \neg\textbf{Writes}(EX, regVal))$$
$$\wedge (\textbf{Stage}(MEM1, 2) \rightarrow \neg\textbf{Writes}(MEM1, regVal))$$
$$\wedge regVal \neq pc$$
$$\leftrightarrow \textbf{delayedForward}(EX, ID, regVal)$$

In this situation, we cannot simply place a bypass from *EX* to *ID* directly, because this would introduce a combinational loop (Figure 2.7). We need to ensure that this bypass is *delayed* by a pipeline register, so we bypass from the subsequent stage to *ID* (Figure 2.8), which means passing the value that is output by the *EXMEM1* pipeline register back to *ID*.

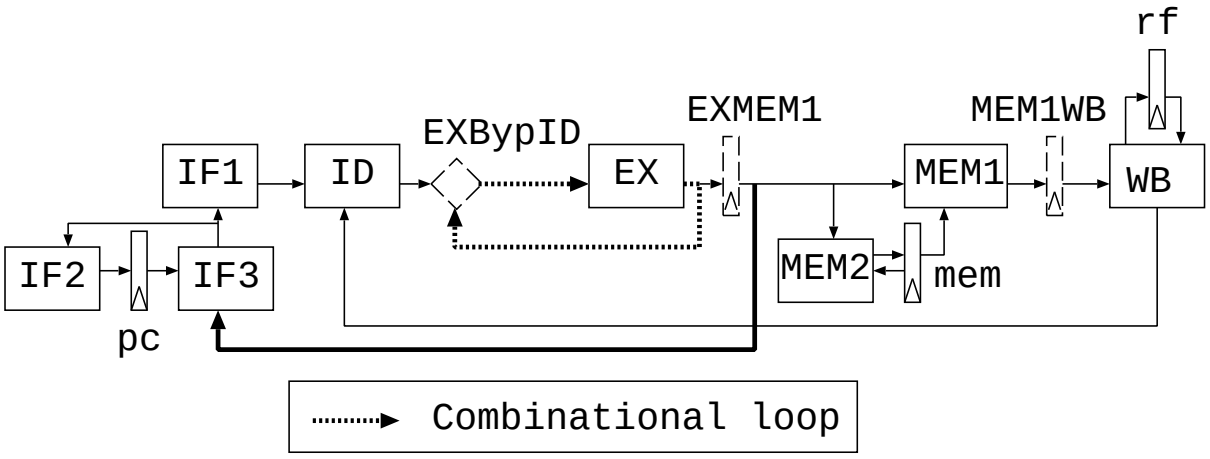


Figure 2.7: DLX With Bypass Resulting in a Combinational Loop

The Boolean constraint for this instance of Rule 3 depends both on the presence and absence

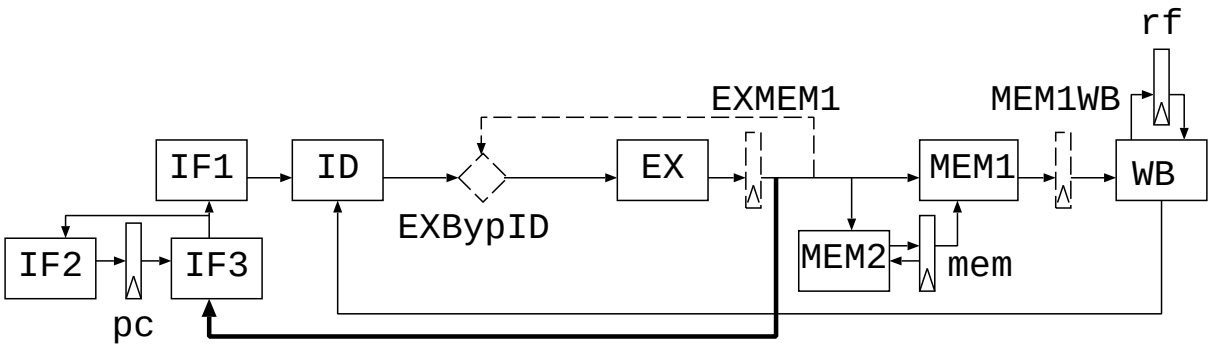


Figure 2.8: Example Instance of Rule 3

of pipeline registers:

$\neg$ **IDEX** $\wedge$ **EXMEM1** $\wedge$ **MEM1WB** $\leftrightarrow$ **EXBypID**

If *MEM1* also *computes regVal*, then we would need a bypass from *MEM1* to *ID* by Rule 1. The duplicate hardware is visible in Figure 2.9. In this situation, we amend Rule 3 and forward the value at the end of stage *2* (in this case *MEM1*) so we can resolve the hazards for Rule 1 and Rule 3 using a single bypass. If we do not need Rule 1, then we apply only Rule 3 and forward from before *MEM1*, which decreases the clock period.

2.5.2 Stalling Rule: Rule 4

A pipeline stall is required when the *need* for a value and its *computation* are separated by 2 or more stages. Resolution for this hazard involves introducing bubbles into the pipeline, and sending a signal to stall the program counter. The following is an example instance of this rule

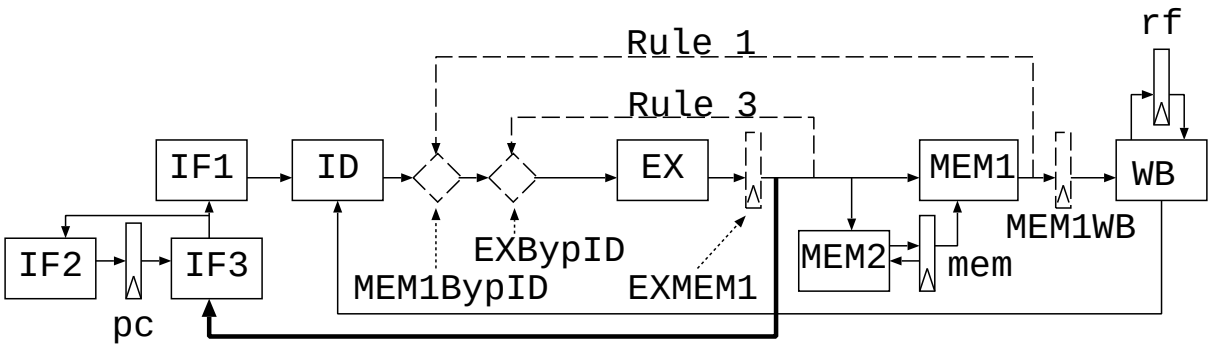


Figure 2.9: Rules 1 and 3 applied to DLX: Duplicate hardware

in the DLX:

Stage$(IF, 1) \wedge$ **Stage**$(ID, 2) \wedge$ **Stage**$(MEM1, 4)$

$\wedge$ **Needs**$(IF, pc) \wedge$ **Needs**$(ID, regVal) \wedge$ **Computes**$(MEM1, regVal)$

$\wedge\ 1 \leq 2 \wedge 2 \geq 2$

$\leftrightarrow$ **stall**$(1) \wedge$ **stallpc**

Figure 2.10 contains the pipeline corresponding to the instance of Rule 4 described above. Three pieces of hazard resolution circuitry were added: a detect unit, a stall unit and a selector. Although it is not revealed in Figure 2.10, the detect unit has an internal register, which we call the *history register*. This is shown in Figure 2.11. The detect unit uses the history register to determine whether or not a stall situation has occurred. An alternative to using a history register is passing the data from the next pipeline register on the datapath back to the detect unit. A *load* operation followed by an *ALU* operation causes the need for a stall. If a stall situation occurs, then the detect unit makes the instruction a bubble and passes a signal to the stall unit. The stall

unit causes instructions to “sit still” in the pipeline.

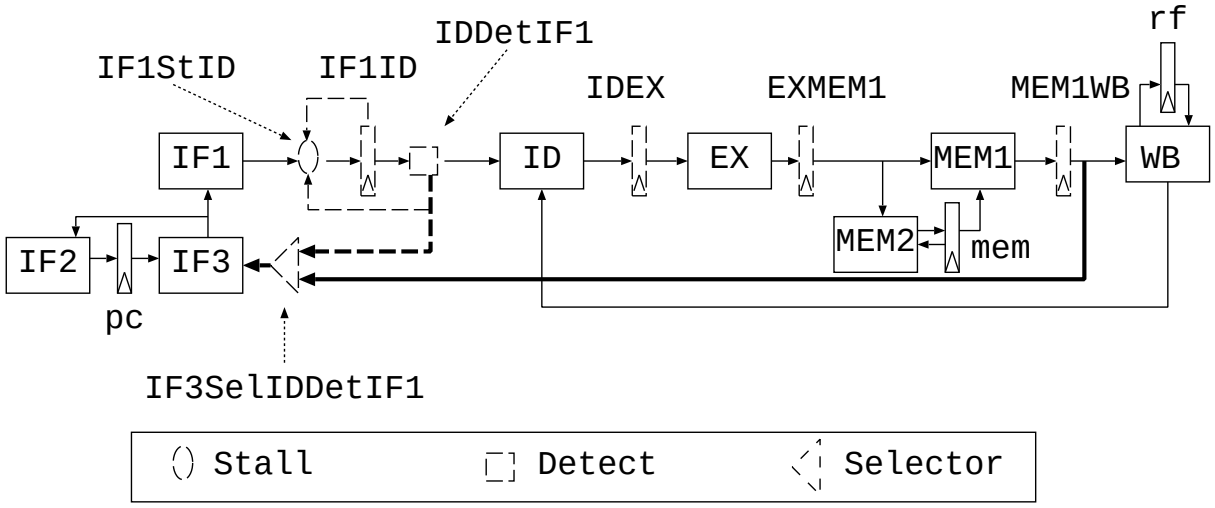


Figure 2.10: Example Instance of Rule 4

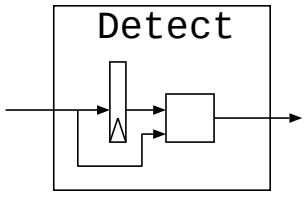


Figure 2.11: Detect Unit with State Revealed

The signals to revise the *pc* are coordinated, so that kill units and instructions later in the pipeline have precedence. The signals sent back to the *IF3* unit are combined in a “*pc* selector” unit, illustrated in Figure 2.10. The arrows with dotted lines are used to match labels to some of the hardware. The bold lines represent instructions that may change the *pc* travelling back to the fetch unit through a selector. In a microprocessor without stall or kill circuitry, the distinguished

pc signal comes from a delay following the last functional unit on the datapath that *computes* the *pc*.

A stall will delay the processing of an instruction. We call a stall that causes a delay of 1 cycle a *1-cycle stall*. When the *need* for a value and its *computation* are *n* stages apart, they cause a *(n-1)-cycle stall*.

The Boolean constraint for this instance of Rule 4 is a conjunction of 3 expressions. The first expression is based on the application of the rule. In order for this rule to apply, 4 pipeline registers must be present:

IF1ID $\wedge$ **IDEX** $\wedge$ **EXMEM1** $\wedge$ **MEM1WB** $\leftrightarrow$ **IF1StID**

Rule 4 does not explicitly state that the *IF* and *ID* must be in different stages. Therefore, the *IFID* register is not always necessary, but we included it in this instance of the rule. The second part of the constraint states that the detect unit should be present if and only if the stall unit is present:

IF1StID $\leftrightarrow$ **IDDetIF1**

The final part of the constraint involves the selector unit. There could be multiple selector units in the pipeline, but each is associated with a particular piece of stall or kill hardware. The selection unit (IF3SelIDDetIF1) is present if and only if the stall unit (IF1StID) is:

IF1StID $\leftrightarrow$ **IF3SelIDDetIF1**

If the need for a value and its computation are separated by more than 2 stages, then the

detect unit must stall the pipeline more than once. Our current implementation only handles 1-cycle stalls, but by inserting different detect units we could accomodate multiple cycle stalls.

We assume that stall situations can be detected after the fetch unit. If stall situations can only be detected after the instructions have been decoded, then we would need to alter our rule or add another rule to ensure that the detect unit is placed in the correct location.

2.5.3 Branching Rule: Rule 5

Instructions in the shadow of a mispredicted branch must be killed in the pipeline. Any branch prediction scheme could be used with our method. In simulation, we have chosen to use the *predict branch not taken* scheme. When a conditional branch enters the pipeline, the fetch unit will continue to fetch subsequent instructions as if the branch is not taken, until the branch condition is checked and the branch address is calculated in the *EX* stage. If the branch is taken, then the 2 instructions that are currently in the *IF* and *ID* stages are killed. Rule 5 states that it must fix the *pc* ("fixpc"). This means that if the branch is taken, then the instruction containing the branch target address (the new *pc*) must be sent to the fetch unit. This instruction passes through a delay and a selector unit, before reaching the *IF3* unit. The selector acts as a switch and is needed to choose the appropriate *pc* to use when there are multiple sources of the next *pc* value. An example instance of a 2-cycle kill (a kill that affects instructions in 2 previous stages) in the DLX, appears in Figure 2.12 and the instantiation of the rule is below:

$$\textbf{Stage}(IF, 1) \wedge \textbf{Stage}(EX, 3)$$
$$\wedge\ \textbf{Needs}(IF, pc) \wedge \textbf{Computes}(EX, pc) \wedge 2 > 0$$
$$\leftrightarrow\ \textbf{kill}(2) \wedge \textbf{kill}(1) \wedge \textbf{fixpc}$$

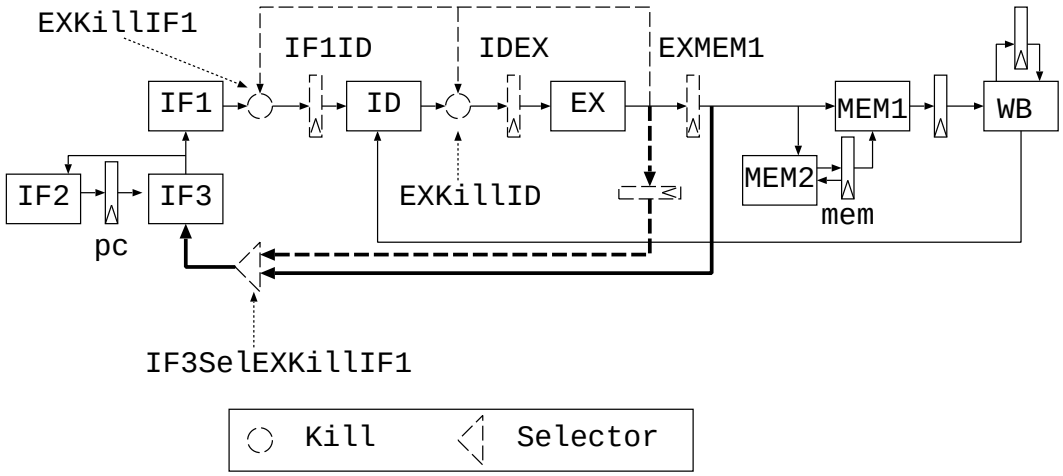


Figure 2.12: Example Instance of Rule 5

In Figure 2.12, the delay *EXMEM1* is necessary for this instance of the rule to occur. If *EXMEM1* is not present, but *IF1ID* and *IDEX* are, then the input to the kill units would come from after *MEM1* rather than *EX*.

Figure 2.12 describes our current implementation where the output of *EX* passes through registers on 2 different wires. An alternative is to not introduce the extra delay on the line to the selector unit and to have the input to the selector come from *EXMEM1* instead. Then we could remove the selector since both its inputs are the same, and have the wire go directly from *EXMEM1* to *IF3*.

The Boolean constraints for this instance of Rule 5 are:

IF1ID $\wedge$ **IDEX** $\wedge$ **EXMEM1** $\leftrightarrow$ **EXKillIF1**

EXKillIF1 $\leftrightarrow$ **EXKillID**

EXKillIF1 $\leftrightarrow$ **IF3SelEXKillIF1**

In this case, the 2 inputs to the selector are the same signal. In the future, we plan to eliminate the selector and the second wire (from *EXMEM* into the selector) in situations where both inputs to a selector are the same.

2.6 Combining Selector Units

The stall and kill rules both require the *pc* to be altered. Figure 2.13 shows how selector units can be combined to determine which new *pc* should be used. In Figure 2.13 there are 2 selector units. They work in combination to determine whether the next *pc* should be:

- the current *pc* resulting from a stall,
- the *pc* resulting from a branch, or
- the incremented *pc*.

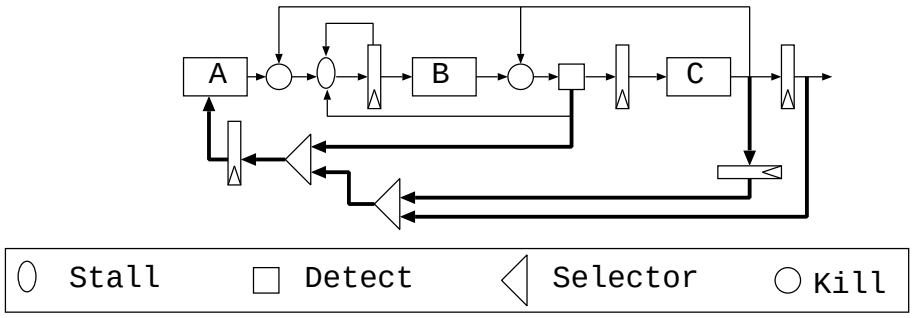


Figure 2.13: Example of Multiple Selector Units

In the selector, kill signals take precedence over stall signals. The next level of precedence ensures that signals coming from later in the pipeline take priority over signals earlier in the pipeline.

2.7 Application of Rules

Our hazard resolution rules are applied to an unpipelined circuit to determine the required places to put hazard hardware. The rules are applied in a particular order, so that the added circuitry is placed in the pipeline in the correct order of precedence. First, we generate the pipeline registers. Then the rules are applied in the following order: Rule 4, Rule 5, and Rules 1-3. When hazard resolution circuitry is placed in the pipeline between X and Y, it is placed immediately after X. For example, when there is both a detect unit and a bypass in the same stage, we want the bypass to come before the detect unit. This order is important, because we want the forwarded value from the bypass to be received before the detect unit makes the decision to stall or not.

To apply a rule, we traverse the datapath and use a functional unit as an "anchor". Each functional unit takes a turn being the anchor, as we work from earlier to later along the datapath.

We illustrate how Rule 1 is applied, using *ID* as an anchor. The functional unit *ID* satisfies the condition *needs* data *regVal*. Therefore, we traverse the remainder of the datapath (from *EX* to *WB*) to determine whether or not the rest of the rule can be satisfied using the DD information. The next unit that is reached is *EX*. The unit *EX computes* data, but does not *write* data. The presence of *IDEX* ensures that *ID* and *EX* are 1 stage apart. The presence of *EXMEM1* satisfies the condition that no data is *written* by functional units later in the stage with *EX*, because when *EXMEM1* is present there are no functional units after *EX* in that stage. This pipeline configura-

tion satisfying this instance of Rule 1 appears in Figure 2.14 and is expressed as follows:

Instance 1

Stage$(ID, 1) \wedge$ **Stage**$(EX, 2)$

$\wedge$ **Needs**$(ID, regVal) \wedge$ **Computes**$(EX, regVal)$

$\wedge$ (**Stage**$(EX, 2) \wedge$ **Earlier**$(EX, EX) \rightarrow \neg$**Writes**$(EX, regVal))$

$\wedge\, regVal \neq pc$

$\leftrightarrow$ **forward**$(EX, ID, regVal)$

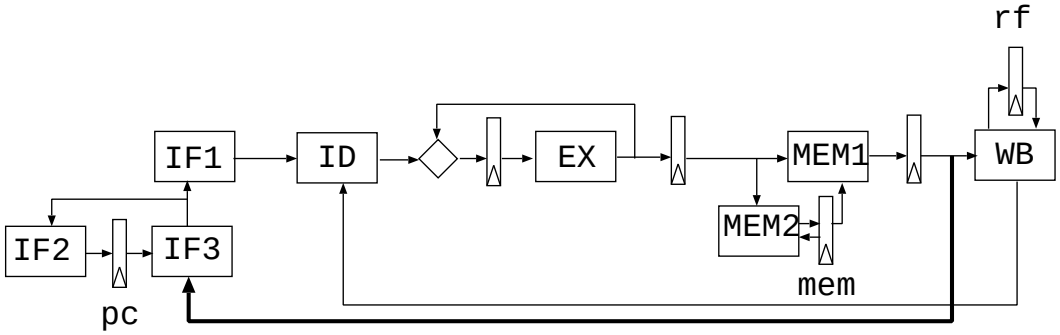


Figure 2.14: DLX Configurations Satisfying Instance 1 of Rule 1

Continuing along the datapath, *MEM1* is the next functional unit reached. The *MEM1* unit also *computes* data. There are 2 pipeline registers between *ID* and *MEM1*. Rule 1 states that the functional units involved in the *needs-computes* dependency (in this case *ID* and *MEM1*) must be 1 stage apart. Therefore, exactly 1 of the 2 registers between *ID* and *MEM1* (*IDEX* and *EXMEM1*) must be present. The rule also states that data cannot be *written* by functional units later in the stage with *MEM1*, so the *MEM1WB* register must be present to ensure that *WB* is in the stage after *MEM1*. The pipeline configurations satisfying this instance of Rule 1 appear in

Figure 2.15. This instance of the rule is expressed as follows:

Instance 2

Stage$(ID, 1) \wedge$ **Stage**$(MEM1, 2)$

$\wedge$ **Needs**$(ID, regVal) \wedge$ **Computes**$(MEM1, regVal)$

$\wedge$ (**Stage**$(MEM1, 2) \wedge$ **Earlier**$(MEM1, MEM1) \rightarrow \neg$**Writes**$(MEM1, regVal))$

$\wedge\, regVal \neq pc$

$\leftrightarrow$ **forward**$(MEM1, ID, regVal)$

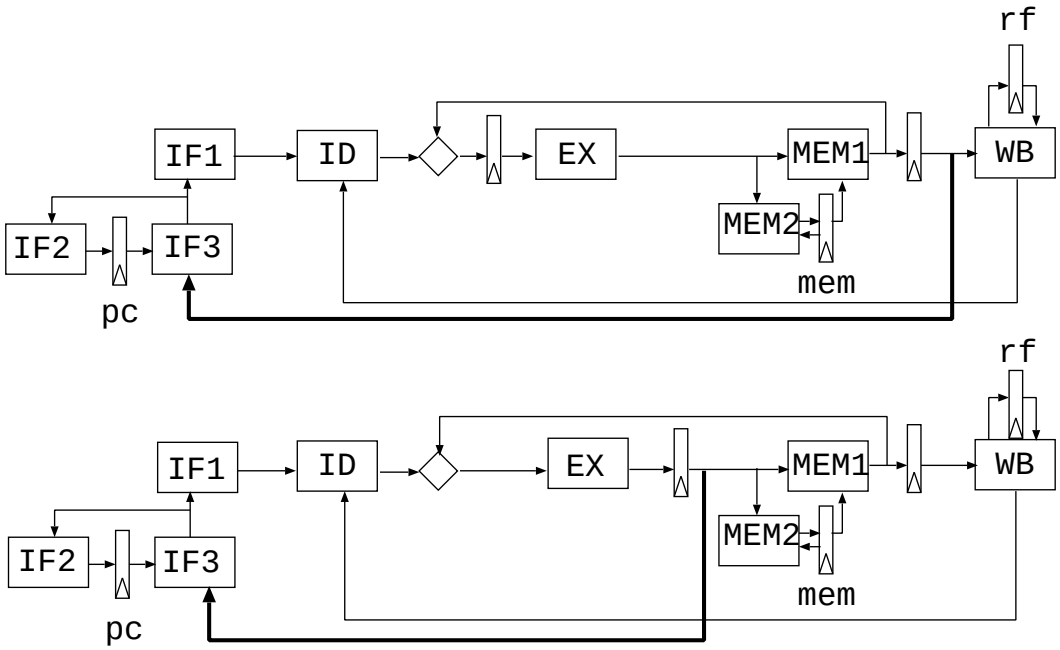


Figure 2.15: DLX Configurations Satisfying Instance 2 of Rule 1

We continue traversing the datapath and reach the last functional unit, *WB*, which does not *compute* data, so our application of Rule 1 using *ID* as an anchor is complete. Applying Rule 1 using the anchor *ID* is equivalent to applying Rule 1 to the entire pipeline, because *ID* is the only

functional unit that *needs* data.

The constraints that are generated by applying Rule 1 to the DLX are:

- **Instance 1**

  IDEX $\wedge$ EXMEM1 $\leftrightarrow$ EXBypID

- **Instance 2**

  (IDEX $\wedge\neg$ EXMEM1 $\vee$ $\neg$ IDEX $\wedge$ EXMEM1) $\wedge$ MEM1WB $\leftrightarrow$ MEM1BypID

There are 3 more pipeline configurations of the DLX that satisfy Rule 1 when the pipeline register *IF1ID* is present. These are similar to the three presented, except the stage numbers are incremented by one.

2.8 Universal Pipeline

As we apply our rules to the pipeline, we generate a collection of hazard resolution circuitry that may be present in the pipeline and a set of constraints describing the situations where that circuitry is needed. Rather than creating multiple pipeline configurations when applying our rules, we create one circuit called the *universal pipeline*, which contains the hazard resolution circuitry in every possible place that it could appear. The universal pipeline is the union of all possible pipeline configurations. Figure 2.16 is the universal pipeline for the DLX. It has potentially extraneous hazard resolution circuitry (depending on the configuration). Values for the Boolean variables indicating the presence or absence of circuitry allow us to create a particular pipeline configuration from the universal pipeline.

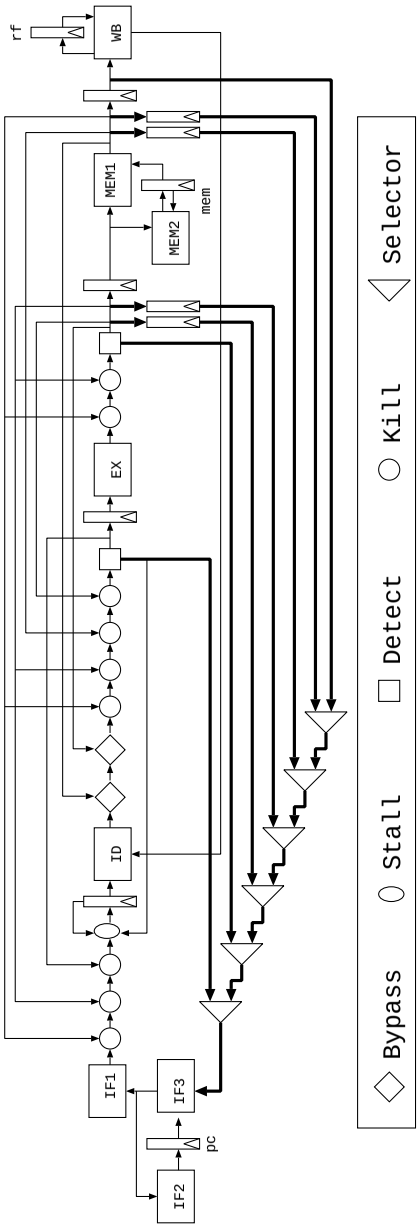


Figure 2.16: Universal Pipeline

2.9 DLX Pipeline Configurations

There are 4 pipeline registers that may be added to the DLX and one of these (*MEM1WB*) is mandatory, because we are using a write-before-read register file. Therefore, there are 2^3 or 8 possible pipeline configurations, including the unpipelined configuration. Figures 2.17 to 2.23 contain all of these configurations (except the unpipelined DLX, which appears in Figure 2.2) with all necessary hazard resolution circuitry based on our rules. The caption of each figure contains a key that indicates which pipeline configuration is illustrated. For example, in Figure 2.17, there is a pipeline register between *IF* and *ID*, so the pipeline key is: *IF* | *ID EX MEM*. These are all correct pipeline configurations, in that they have all hazards resolved to produce correct output. All of these pipeline configurations are solutions to the CSP problem consisting of the constraints generated by the application of our rules to the DLX. In the next chapter, we discuss out method for determining which configuration is optimal for throughput given timing information.

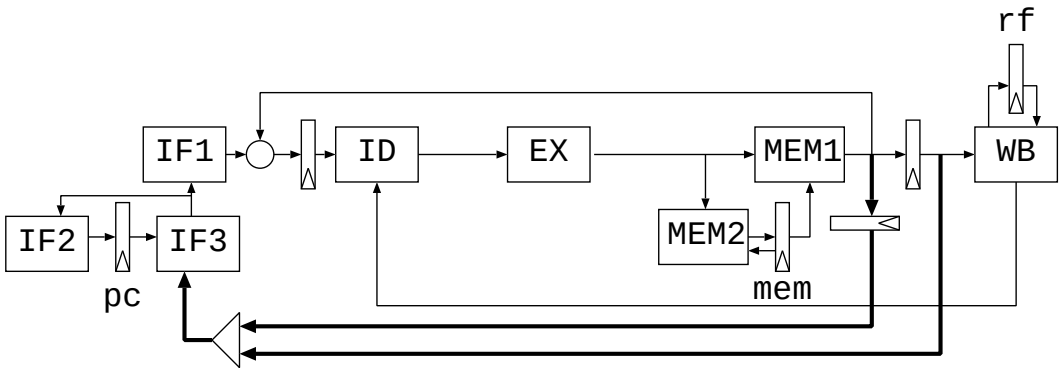


Figure 2.17: DLX Configuration 1 (IF | ID EX MEM)

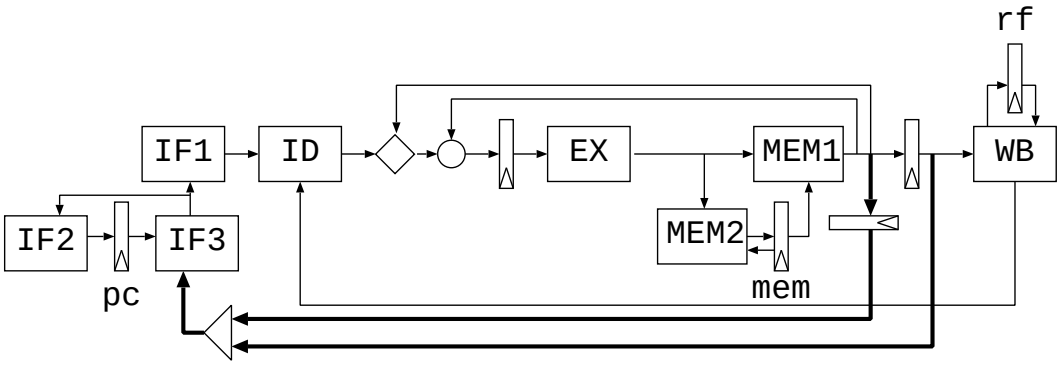


Figure 2.18: DLX Configuration 2 (IF ID | EX MEM)

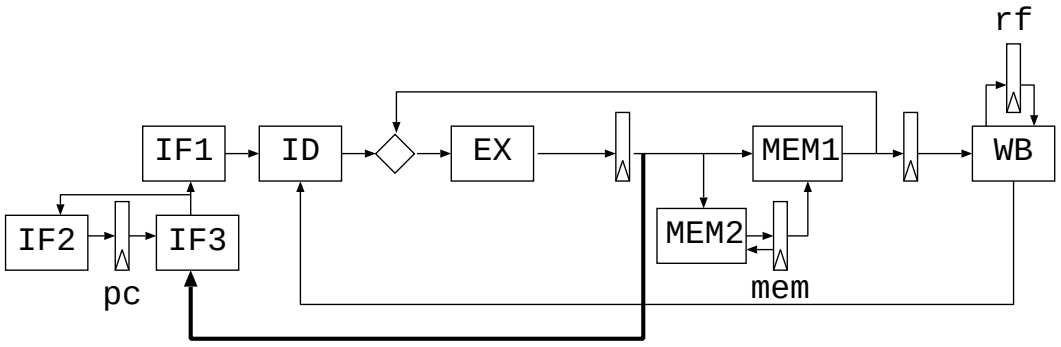


Figure 2.19: DLX Configuration 3 (IF ID EX | MEM)

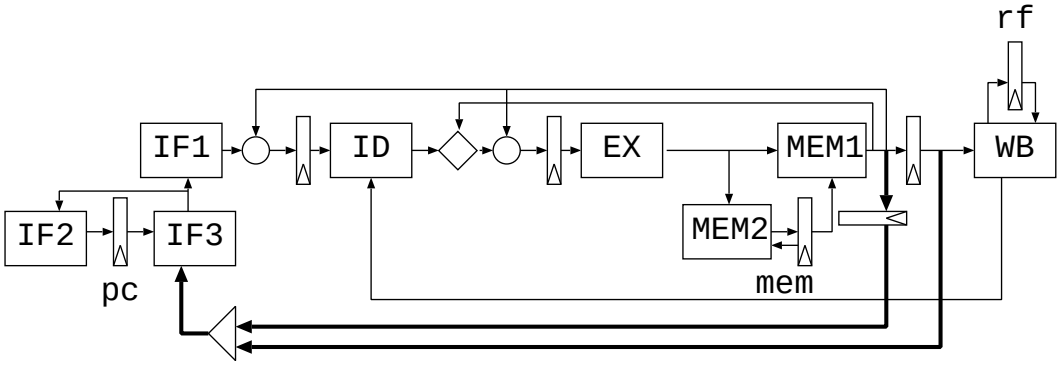


Figure 2.20: DLX Configuration 4 (IF | ID | EX MEM)

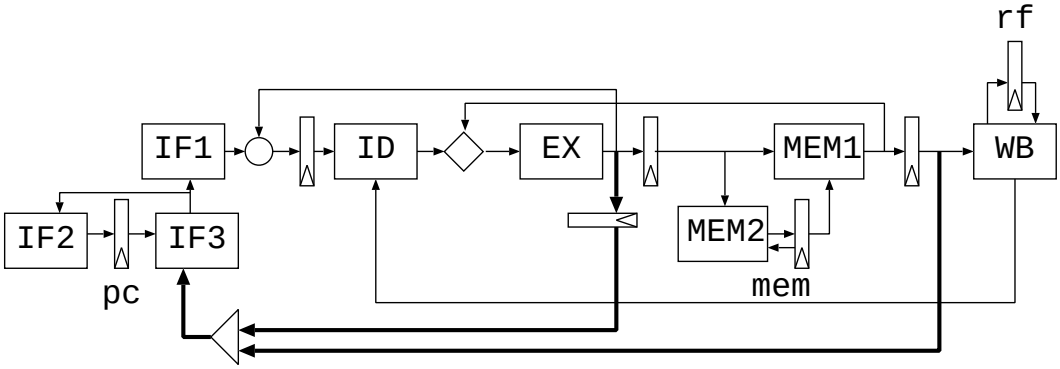


Figure 2.21: DLX Configuration 5 (IF | ID EX | MEM)

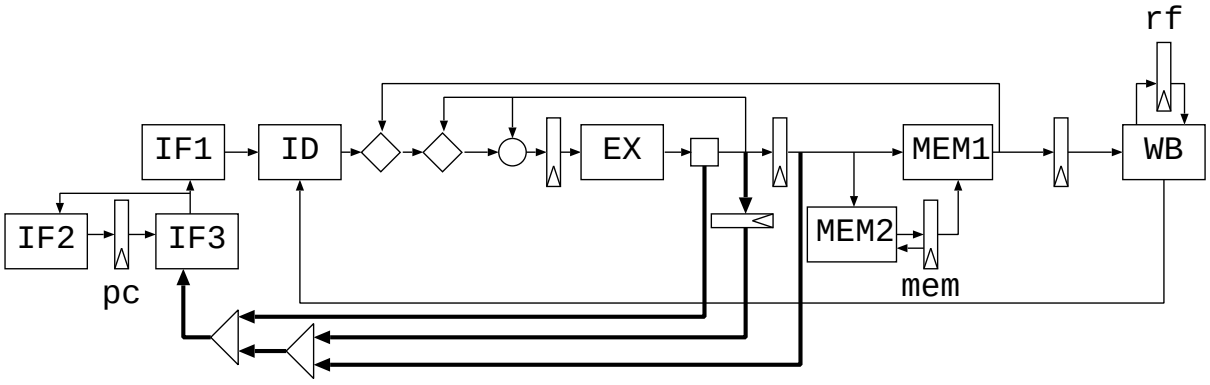


Figure 2.22: DLX Configuration 6 (IF ID | EX | MEM)

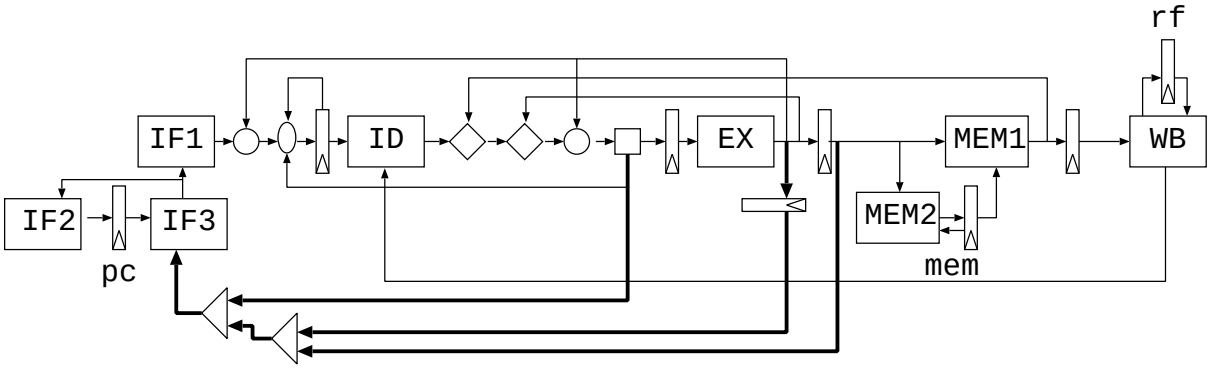


Figure 2.23: DLX Configuration 7 (IF | ID | EX | MEM)

2.10 Validation

In order to verify that our rules are correct and complete, we tested the DLX pipeline configurations using simulation. In simulation, the input to the pipeline configuration is a program, which is a list of instructions. Simulation allows us to "execute" our instructions and examine the results to ensure that they are as expected. The goal is to match the output of our pipelined configurations with that of the unpipelined processor.

Simulation was used to debug our rules to ensure the necessary hardware was placed in the correct location for all pipeline configurations. We took a systematic approach and tested an instance of each of the known hazard situations on all of the pipeline configurations generated by our method. In Chapter 5, we discuss future work of formally verifying the rules.

2.11 Summary

We have presented 5 rules for hazard resolution. These rules are based on the Data Dependency (DD) information, which is a description of the data that a functional unit *computes*, *writes* or *needs*. The rules are also dependent on the grouping of the functional units into stages and the distance between pipeline stages.

The pipeline rules are applied to yield a set of constraints, which describe all of the situations in which hazard hardware is required. The constraints can then be passed to a SAT solver and the resulting solution is an instantiation of the variables, which describes the presence or absence of hazard resolution circuitry. A solution is one possible configuration of the pipeline.

Chapter 3

Throughput Optimization

In this thesis, we provide an automated method of finding the optimal pipeline configuration (in terms of throughput) by examining every configuration. The process starts with an unpipelined datapath and a description of the functional units, and produces the optimal pipeline. We add hazard detection and resolution hardware as needed, and take the additional hardware into account in our *clock period* and *throughput* calculations. This chapter describes our approach to throughput optimization.

3.1 User Input

The layout of the circuit is provided by the user as a textual description of a dataflow diagram. A unique name for each functional unit is needed, along with its previous (parent) and successor (child) nodes. The user must also provide any non-pipeline registers, such as the register representing the register file in the ISA state. The user may also specify that a particular pipeline register need always be present.

There are 2 types of paths in the dataflow diagram: instruction and non-instruction. An *instruction path* is 1 or more wires along which instructions travel. An example of a *non-instruction path* is a wire carrying the register file. It is necessary to distinguish between the 2 types of paths, because non-instruction paths should not be pipelined. If 2 units are connected by a non-instruction path, then they are considered to be a single functional unit that should not be further decomposed.

Figure 3.1 is the user input for the DLX. Each line represents a different component (functional unit or register). The first word on a line is the name of the component and the second word is its type. "*PREV:*" and "*SUCC:*" are keywords (all words followed by a colon are keywords) and they are followed by the names of the child and parent nodes of the component, respectively. An "(I)" is used to indicate that the path carries an instruction. For example, line 2 of Figure 3.1 states that the *ID* unit is a successor of the *IF1* unit. The "(I)" after *ID* means that the wire between *IF1* and *ID* is an instruction path. The input and output of the circuit are also specified by the user.

To optimize the pipeline, we need the timing cost of each functional unit. Timing cost information is also needed for pipeline registers and for hazard resolution circuitry. Sample timing information for the DLX appears in Figure 3.2. The number of inputs (arguments) to each functional unit is also included. This information is used for generating the data structure and functional descriptions of the units (discussed later in this section). The "HAZTIMING" section contains the cost of each type of hazard resolution circuitry.

To resolve hazards we need the Data Dependency information for the functional units, which was described in Chapter 2. Figure 3.3 contains the DD information for the DLX (previously seen in Figure 2.3 on page 29).

The descriptions of the circuit layout, the timing information and the DD information are all

```
CIRCUIT:
IF1 FUNC PREV: IF3 SUCC: ID (I)
PC REG PREV: IF2 SUCC: IF3
IF2 FUNC PREV: IF3 SUCC: PC
IF3 FUNC PREV: PC, MEMWB SUCC: IF1, IF2
ID FUNC PREV: IF1 (I), WB SUCC: EX (I)
WB FUNC PREV: MEMWB (I), RFREG SUCC: RFREG, ID
RFREG REG PREV: WB SUCC: WB
EX FUNC PREV: ID (I) SUCC: MEM1 (I), MEM2 (I)
MEM1 FUNC PREV: EX (I), MEMREG SUCC: MEMWB (I)
MEM2 FUNC PREV: EX (I), MEMREG SUCC: MEMREG
MEMREG REG PREV: MEM2 SUCC: MEM1, MEM2
MEMWB REG PREV: MEM1 (I) SUCC: IF3, WB (I), OUT (I)

INPUT:
IF1

OUTPUT:
MEM1WB
```

Figure 3.1: DLX Circuit Description

```
TIMING:
IF1 ARGS: 2 TIME: 25
IF2 ARGS: 1 TIME: 20
IF3 ARGS: 2 TIME: 25
ID ARGS: 2 TIME: 30
EX ARGS: 1 TIME: 35
MEM1 ARGS: 2 TIME: 40
MEM2 ARGS: 2 TIME: 35
WB ARGS: 2 TIME: 19

HAZTIMING:
DELAY: 3
BYP: 3
STALL: 2
DETECT: 2
KILL: 2
```

Figure 3.2: DLX Timing Description

```
DD:
Needs(IF3, pc)
Needs(ID, regVal)
Computes (ID, pc)
Computes(EX, regVal)
Computes (EX, pc)
Computes(MEM1, regVal)
Writes(WB, regVal)
```

Figure 3.3: DLX DD Information

input to our tool through a text file. We chose this input format for ease of implementation. In the future, a format more compatible with existing tools could be used.

3.2 Data Structures

We used the functional programming language ML [25] to implement our optimization process. The input is parsed using *mosmllex* and *mosmlyacc* [25, 18], and a data structure representing the circuit and a functional interpretation of the circuit (described in the next section) is created. We have designed a graph data type consisting of nodes that store information including the name of a component, a list of previous nodes, a list of successor nodes and a "visited" flag (used for traversing the data structure). A node is also categorized as either a functional unit, register, bypass, kill, stall, detect or selector. Knowing the type of node is necessary for manipulating the circuit and adding hardware (pipeline registers and hazard resolution circuitry). Before creating our own data type, we investigated existing types to determine if there was a suitable match. In our data structure, each node in the graph (circuit) can have multiple children and parents, and we felt that using a data type that was designed specifically for our problem would be the best approach.

Once the input is parsed and the data structure is created, we traverse the data structure and apply the rules to all possible configurations. The output is:

- the universal pipeline, and
- a set of constraints describing situations that required hazard circuitry.

The constraints are then converted to *conjunctive normal form* (CNF), which is a Boolean expression that contains only the logical operators *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$). The CNF expres-

sion is a conjunction of disjunctions of literals. Once the constraints are in CNF, they are input to the SAT solver *SATO* [29]. SATO returns an instantiation of the variables that satisfies all of the constraints (if a solution exists). The connection from *ML* to *SATO* was made possible using modified portions of Gordon's HolSatLib [10] and the DIMACS graph format [1].

3.3 Universal Pipeline Code

Before introducing the method that we used to calculate the *clock period*, we must first discuss the *universal pipeline code*. Besides the internal data structure, the textual input is also used to create an *ML* function representing the universal pipeline, which is the union of all possible pipeline configurations. A portion of universal pipeline code for the DLX is in Figure 3.4.

We represent a circuit using mutually-recursive functions. Within the let statement, each line starting with "fun" or "and" describes a signal, which is the output of a functional unit, pipeline register or hazard resolution hardware. The input to the pipeline, *INP*, is a program that consists of 1 or more instructions.

In simulation, the data manipulated by the circuit is a stream (list) of values. The *lift* functions apply combinational functions to the element of a stream. The input to a pipeline register (delayPar) is a stream and the output is the input stream shifted by 1 to represent the stream in the next clock cycle. All signals are functions of units so that we can delay the computation of infinite lists in a functional language with eager evaluation [24].

The Boolean variables used in the constraints that represent the presence or absence of components, are used as the conditions of *if* statements in the universal pipeline code. For example, if the Boolean variable *!IFIDbv* is set to *true* (in a given SAT solution), then there will be a pipeline register (delay) between *IF* and *ID*. If it is set to *false*, then the output of that function is

one of the inputs, making it equivalent to a wire with no cost. Therefore, given the values for the Boolean variables, we can create a particular pipeline configuration for a circuit.

In order to test our method using simulation, the universal pipeline code is designed using an existing library of functions that simulate the tasks that components perform. For example, library components include an ALU, a pipeline register, and a multiplexer (MUX). Our library also includes the functional units of the DLX.

3.4 Reinterpretation

Reinterpretation [22] is used to calculate the *clock period*. Reinterpretation determines the *clock period* of the circuit using alternate definitions of the functions of the various pipeline components. This allows us to use the same description of the circuit (the universal pipeline code) for simulation, visualization, verification and timing analysis.

We now explain the concept of reinterpretation through an example using the DLX. The simulation interpretation defines the components in terms of the tasks that they perform. For example, the simulation interpretation of the ID unit, as well as the other definitions it relies on, is in Figure 3.5. Instructions flowing through the circuit are represented using a parcel data type as a record-like *ML* structure. The register file is a list of integers. There are two inputs to the *ID* function: the instruction (*INP*) and the register file (*rf*). The *ID* function reads the values of the two source registers from the register file and outputs the instruction containing these source values.

To determine the *clock period* we can use an alternative interpretation. Instead of considering what the functional units do, the units are defined in terms of how long it takes to perform the tasks. In this case, timing costs runs along the wires, not instructions. Each function is equal

```
fun pipe INP =
   let
      fun IF1 () = lift2 IF1FUNC IMEM IF3  ()
      and IMEM () = delayPar "IMEM" initPar IMEM ()
      and PC () = delayPar "PC" initPar IF2
      and IF2 () = lift1 IF2FUNC IF3  ()
      and IF3 () = lift2 IF3FUNC PC SelDEL  ()
      and ID () = lift2 IDFUNC IF1ID RFREG  ()
      and WB () = lift2 WBFUNC MEMWB RFREG  ()
      and IF1ID () =
            if !IF1IDbv
              then (delayPar "IF1ID" initPar IF1 ())
            else (IF1 ())
      and IF1StID () =
            if !IF1StIDbv
              then (lift3 Stall IF1KillID IF1ID IDDetIF1 ())
            else (IF1KillID ())
      and IDDetIF1 () =
            if !IDDetIF1bv
              then (detect IF1ID ())
            else (IDSTEX  ())
      and IF1KillID () =
            if !IF1KillIDbv
              then (lift2 kill IF1KillEX IDDetIF1 ())
            else (EXKillIF1 ())

      ...

      and EXBYPID () =
            if !EXBYPIDbv
              then (lift2 bypassPar MEM1BypID EX  ())
            else (MEM1BypID ())
   in
      MEM1WB
   end;
```

Figure 3.4: DLX Universal Pipeline Code

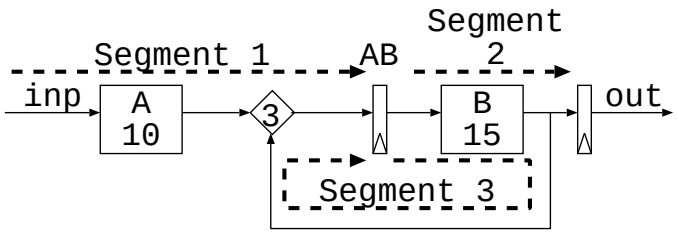
```
fun ssrc1 (Parcel (_,_,_,(src1,_),_,_)) = src1;

fun ssrc2 (Parcel (_,_,_,(src2,_),_,_)) = src2;

fun sdata1 (Parcel (_,_,_,(_,data1),_,_)) = data1;

fun read rf r = List.nth(rf, r);

fun readRf rf r = read rf (pos r);

fun mux a b c = if a then b else c;

fun replaceData1 (Parcel (v,pc,opc,(src1,_),x1,x2)) data1 =
  Parcel (v,pc,opc,(src1,data1),x1,x2);

fun replaceData2 (Parcel (v,pc,opc,x1,(src2,_),x2)) data2 =
  Parcel (v,pc,opc,x1,(src2,data2),x2);

fun ID inp rf =
  let
    fun readRegs i rf =
      let
        val v1 = readRf rf (ssrc1 i)
        and v2 = mux (ssrc2 i = RImm)
                     (sdata2 i)
                     (readRf rf (src2 i))
      in
        replaceData2 (replaceData1 i v1) v2
      end
    fun out () = lift2 readRegs inp rf ()
  in
    out
  end;
```

Figure 3.5: Functional Interpretation of ID

to the maximum value of its inputs plus the time it takes to perform that function (a constant provided by the user). The inputs to the functional unit are the maximum times that it has taken the inputs to reach this point in the circuit, since the last register. Figure 3.6 contains the timing interpretation for the *ID* unit, where *TIME_ID* is a constant representing the time cost of the *ID* unit.

```
fun ID inp newrf = Int.max(inp, newrf) + TIME_ID;
```

Figure 3.6: Timing Interpretation of ID




```
fun pipe inp =
  let
    fun A () = AFUNC inp ()
    and byp () = bypass A B ()
    and AB () = delayPar "AB" initPar byp ()
    and B () = BFUNC AB ()
    and out ()= delayPar "out" initPar B ()
  in
    out
  end;
```

Figure 3.7: Example of Reinterpretation

Figure 3.7 contains a small example of using reinterpretation to calculate clock period. The input to the circuit is 0 time units. Reinterpretation uses the definitions of the functional units to

calculate the costs of the stages. For example, in determining the cost of “Segment 1”, the costs of *A* and a bypass are summed. The key to the process is the delay units. The delay unit signifies the end of the segment and it compares the cost of the segment with the current maximum clock period. The delay then starts calculating the cost of the next segment at 0 time units. Another role of the delay is to break loops in the pipeline. Each delay is assigned a unique name, and once a delay is traversed that name is stored, so that it will not be visited again. For example, “Segment 2” begins at delay *AB* and follows a loop back to *AB*, at which point it ends. The process will not start searching a new path from delay *AB*, because that delay has already been visited.

An alternative method for calculating the clock period is to traverse our *ML* data structure and calculate the costs of each stage. Our data structure is quite complex, and the output from a node might be input to as many as 6 other nodes, so reinterpretation is a much simpler option.

3.5 Optimization

Our goal is to optimize the throughput, which is dependent on the clock period. In a pipeline without hazards that cause a delay in processing, the throughput calculation is the inverse of the clock period. Throughput has to take into account the presence of stalls and kills, because they will delay the processing of instructions. Hazards are categorized based on the number of cycles that they delay (stall or kill) the pipeline. The user must provide the information about the frequency of instructions. For example, a load instruction immediately followed by a dependent ALU instruction will result in a pipeline stall, so the user must provide information about how often this instruction pair occurs. The throughput will always be less than or equal to 1, because only 1 instruction is fetched each clock cycle. The formula for *throughput* is:

$$\frac{1}{(\%\textit{no-delay})*cp \; + \; 2*\%(\textit{1-cycle-delay})*cp \; +\ldots+ \; (n+1)*\%(\textit{n-cycle-delays})*cp}$$

cp	clock period
%*no-delay*	percentage without delays
%*1-cycle-delay*	percentage that result in a one cycle delay
%*n-cycle-delays*	percentage that result in an n cycle delay

In a pipeline that has stall hardware that causes a 1-cycle delay in processing, the clock period calculation must take that into account. For example, if *load* instructions are following by dependent *ALU* instructions 7% of the time, then a 1-cycle stall occurs 7% of the time, so the throughput calculation becomes:

$$= \frac{\frac{1}{(\%\textit{no-delay})*cp \; + \; 2*\%(\textit{1-cycle-delay})*cp}}{\frac{1}{(0.93)*cp \; + \; 2*(0.07)*cp}}$$

Once the throughput has been calculated for a given pipeline configuration, a constraint stating that the current solution is not allowed is added to the current set of constraints. We iterate through the process of finding solutions and calculating throughput until there is no solution to the SAT problem (i.e., we have examined every possible configuration). The result of the process is an instantiation of the universal pipeline, which is optimal for throughput.

3.6 Summary

The optimization process starts with user input, which is provided in the form of a text file. The user provides a description of the dataflow of the circuit, timing information for functional

units and hazard hardware, and DD information for the circuit. The text file is parsed and an *ML* internal data structure representing the circuit is created. The data structure is traversed and the five hazard hardware placement rules are applied to generate a set of constraints, and the universal pipeline code.

Those constraints are passed to a SAT solver, which returns solutions that are pipeline configurations. A solution from the SAT solver provides values for the Boolean variables, which are applied to the universal pipeline to generate a particular pipeline configuration. The clock period is calculated using reinterpretation and the throughput is found for each configuration. The output of the process is the pipeline configuration that is optimal for throughput.

Chapter 4

Case Studies

To demonstrate our optimization method, we have performed 2 case studies. The first case study involved applying our method to Hennessy and Patterson's DLX pipeline. The DLX was the test pipeline that we used to develop our optimization method. The second case study was performed on a *multiply accumulator* (MAC), which is unrelated to the DLX and was not considered when developing our technique. The timing costs for hazard resolution circuitry appear in Table 4.1. These costs are used in both case studies.

4.1 DLX

The DLX is used by Hennessy and Patterson to illustrate pipelining and pipeline hazards [11]. This academic pipeline is linear and composed of only 5 stages, so at first glance it seems that pipelining it would be a trivial task. We use our optimization method to show that full pipelining is not always optimal and making relatively minor adjustments to the functional unit costs can change which pipeline configuration is optimal.

Hardware	Cost
Pipeline Register	3
Bypass	2
Stall	2
Detect	2
Kill	2
Selector	3

Table 4.1: Hazard Resolution Circuitry Timing Cost

There are 8 possible pipeline configurations for the DLX. We experimented with different costs to see which pipeline configuration was optimal. Running our implementation to generate an optimal pipeline configuration takes less than 15 seconds on a Pentium III Xeon server. Figures 4.1 to 4.3 contain 3 optimal pipeline configurations based on 3 different sets of functional unit costs. From these results, we see that changing the cost of a single functional unit, *EX*, by as little as 1 time unit can result in a different optimal pipeline configuration. Figure 4.4 contains the *ML* function solution that corresponds with the pipeline configuration diagram in Figure 4.1.

Figures 4.5 to 4.8 contain 4 more pipeline configurations with their timing cost, clock period and throughput. By comparing the pipeline configurations in Figures 4.7 and 4.8, we see that a smaller clock period does not necessary correlate to higher throughput. The pipeline in Figure 4.7 has a clock period of 118 and its throughput (0.00722) is higher than the throughput of the pipeline in Figure 4.8 (0.00707), which has a clock period of 108.

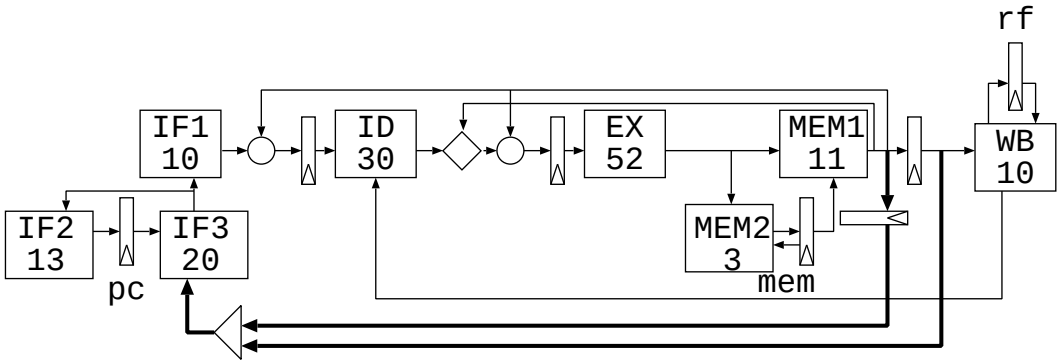


Figure 4.1: DLX Results: Example 1 (IF | ID | EX MEM)

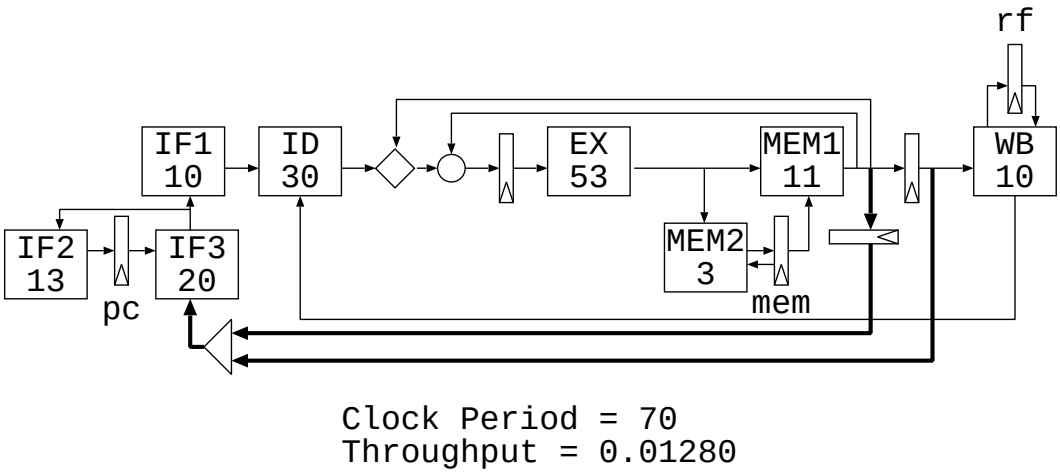


Figure 4.2: DLX Results: Example 2 (IF ID | EX MEM)

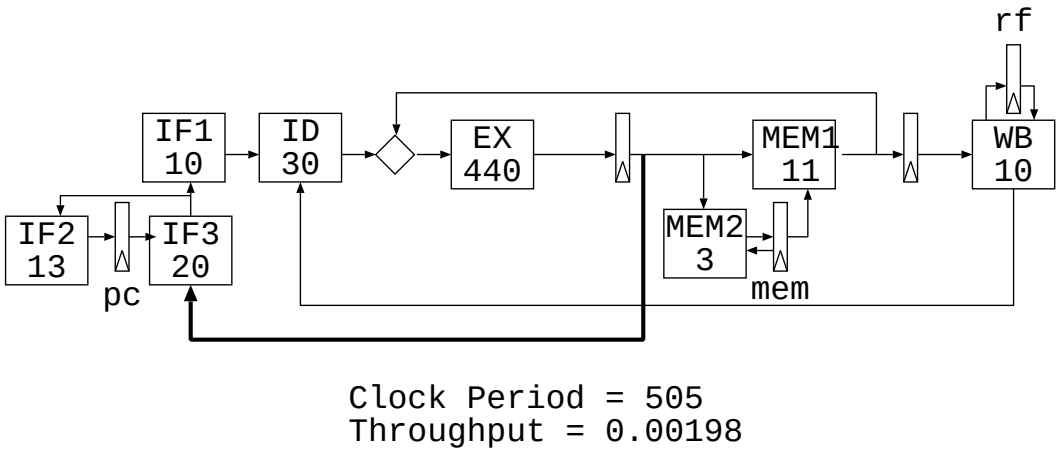


Figure 4.3: DLX Result: Example 3 (IF ID EX | MEM)

4.2 MAC: Multiply Accumulator

For our second case study, we pipeline a simple *multiply accumulator* (MAC) [23]. A MAC is used to perform 3 operations:

- A = B + C
- A = B $\times$ C
- A = B + (C $\times$ D)

The unpipelined MAC appears in Figure 4.9. Wires representing the input of the program are included in the figure for clarification. The MAC is composed of 4 functional units, but it could be further decomposed. The *ADD* unit reads the value of 2 registers from the register file and adds them. The *MUL* unit reads 2 register values from the register file and multiplies them. The *RND* unit performs normalization, rounding, and renormalization for floating point numbers. Finally, the *WB* unit writes the result to the floating-point register file.

```
fun pipe INP =
   let
      fun IF1 () = lift1 IF1FUNC IF3  ()
      and PC () = delayPar "PC" initPar IF2 ()
      and IF2 () = lift1 IF2FUNC IF3  ()
      and IF3 () = lift2 IF3FUNC PC IF1KillIDSelIF1  ()
      and ID () = lift2 IDFUNC IF1DELID WB  ()
      and WB () = lift2 WBFUNC MEM1WB RFREG  ()
      and RFREG () = delayPar "RFREG" initPar WB ()
      and EX () = lift1 EXFUNC IDEX  ()
      and MEM1 () = lift2 MEM1FUNC EXMEM1 MEMREG  ()
      and MEM2 () = lift2 MEM2FUNC EXMEM1 MEMREG  ()
      and MEMREG () = delayPar "MEMREG" initPar MEM2 ()
      and MEM1WB () = delayPar "MEM1WB" initPar MEM1 ()
      and IF1DELID () = delayPar "IF1ID" initPar MEM1KillIF1 ()
      and IDEX () = delayPar "IDEX" initPar MEM1KillID ()
      and MEM1KillIF1 () = lift2 kill IF1 MEM1 ()
      and MEM1KillID () = lift2 kill MEM1BypID MEM1 ()
      and MEM1BypID () = lift2 bypassPar ID MEM1  ()
      and IF1KillIDSelIF1 () = lift2 selector MEM1KillIF1DEL MEMDELWBOUT ()
      and MEM1KillIF1DEL () = delayPar "MEM1KillIF1DEL" initPar MEM1 ()
   in
      MEM1WB
   end;
```

Figure 4.4: Example Solution

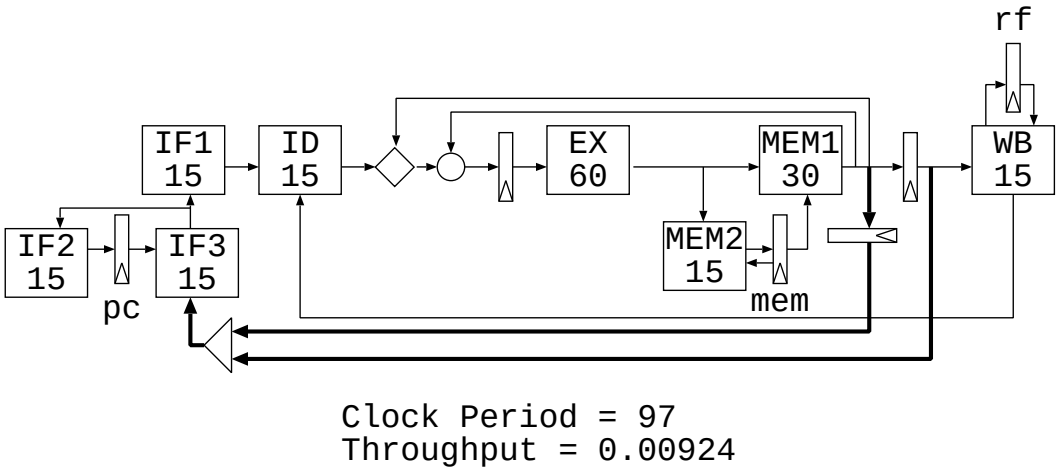


Figure 4.5: DLX Result: Example 4 (IF ID | EX MEM)

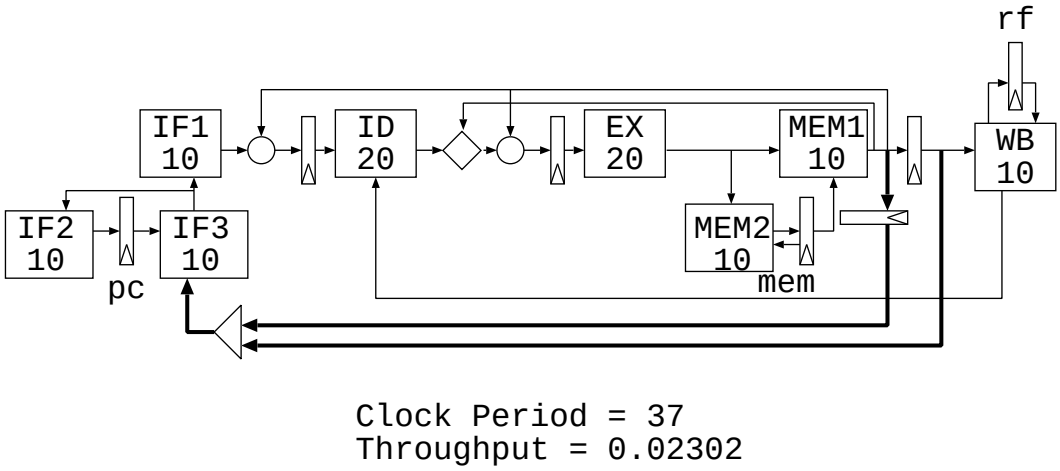


Figure 4.6: DLX Result: Example 5 (IF | ID | EX MEM)

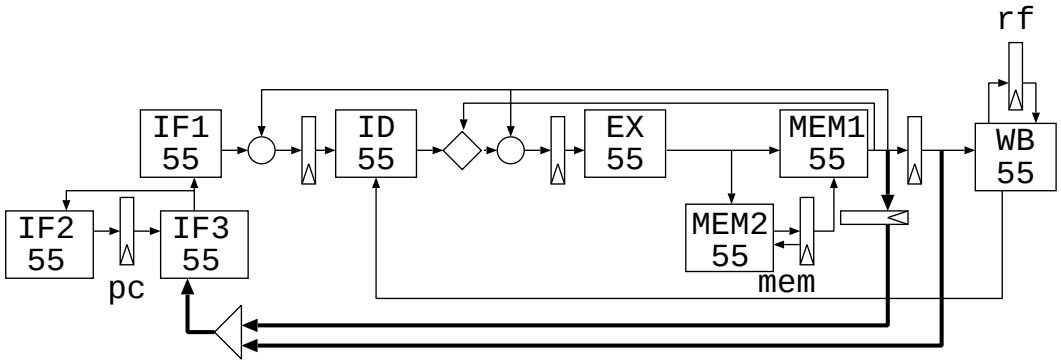


Figure 4.7: DLX Result: Example 6 (IF | ID | EX MEM)

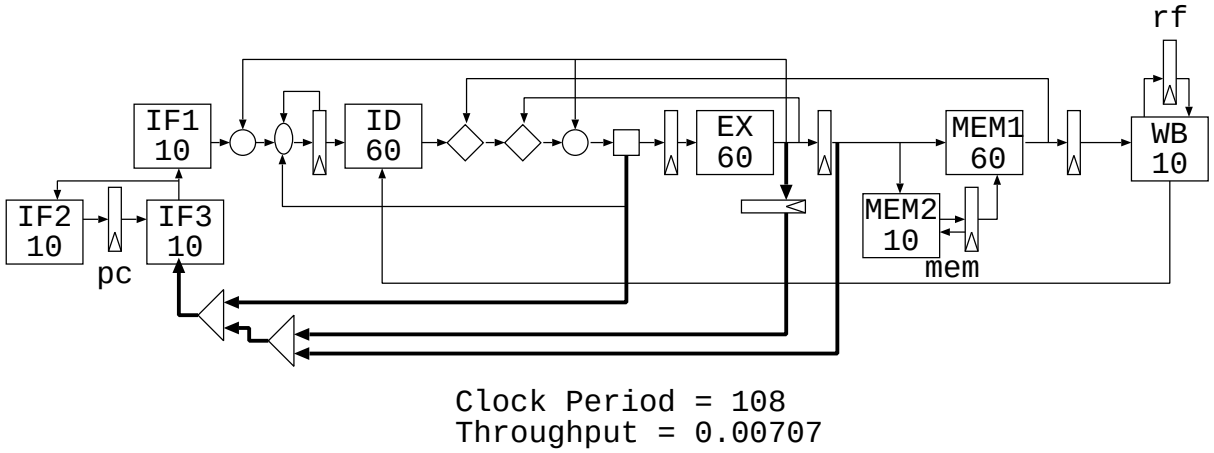


Figure 4.8: DLX Result: Example 7 (IF | ID | EX | MEM)

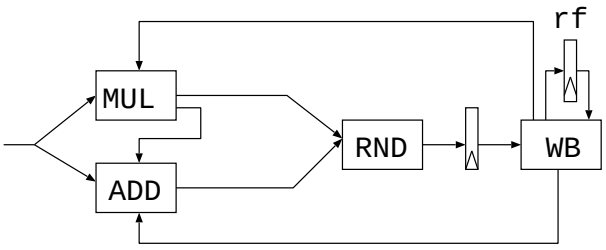


Figure 4.9: Unpipelined MAC

This pipeline has 2 structural hazard situations. There may be contention for the *ADD* unit from the inputs to the unit. The other structural hazard occurs when both *ADD* and *MUL* pass data to the *RND* unit. Our rules do not resolve structural hazards. For the purpose of this case study, we assume that a compiler orders instructions so that no structural hazards occur.

Figure 4.10 contains the textual circuit description and the DD information for the MAC. This pipeline only has *Needs-Computes* dependencies for *regVal*. The only hazard resolution hardware needed is *bypasses*.

There are 3 possible locations for pipeline registers in the MAC. The *RNDWB* register is always present, because we are using a write-before-read register file. There are 4 pipeline configurations of the MAC, including the unpipelined circuit. An interesting feature of this problem, is that the universal pipeline is the same as the fully pipelined MAC. The MAC configurations appear in Figures 4.11 to 4.13.

We experimented with timing costs for the functional units to determine which pipeline configurations would be optimal. The resulting pipelines are in Figures 4.14 to 4.16. There are only 2 pieces of hazard resolution hardware that may be required, so the fully pipelined configuration is often the best. In Figure 4.14, the cost of the functional units is only 2, and the bypass hardware costs 3 (Table 4.1), but the optimal configuration is still fully pipelined. When the cost of the

```
CIRCUIT:
ADD FUNC PREV: WB, INP (I), MUL (I) SUCC: RND (I)
MUL FUNC PREV: WB, INP (I) SUCC: ADD (I), RND (I)
RND FUNC PREV: ADD (I), MUL (I) SUCC: RNDDELWB (I)
RNDDELWB REG PREV: RND (I) SUCC: WB (I), OUT (I)
WB FUNC PREV: RNDDELWB (I) SUCC: MUL, ADD
RF REG PREV: WB SUCC: WB

DD:
ADD Needs regVal
MUL Needs regVal
RND Computes regVal
WB WritesBack regVal

INPUT:
ADD, MUL

OUTPUT:
WB
```

Figure 4.10: MAC Input (without timing information)

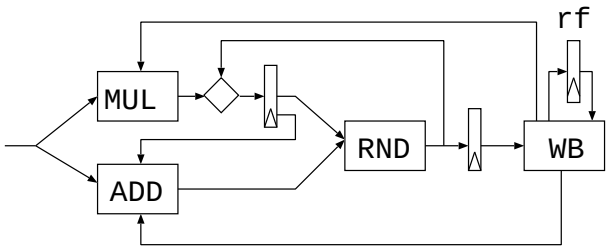


Figure 4.11: MAC Configuration 1

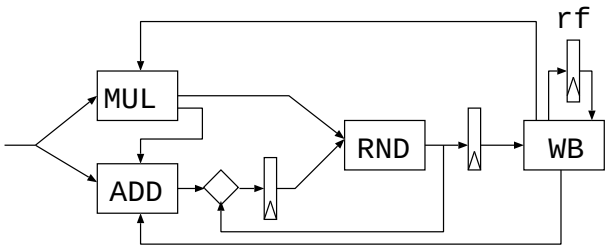


Figure 4.12: MAC Configuration 2

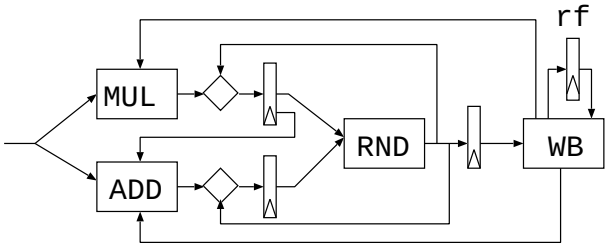
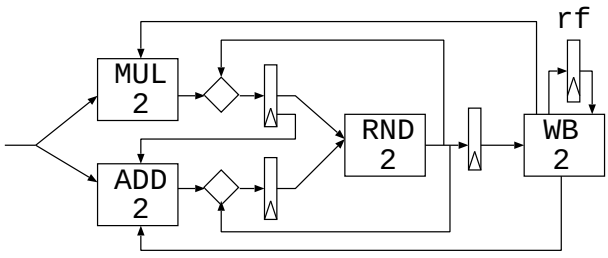


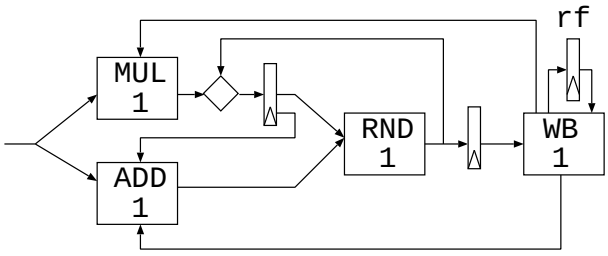
Figure 4.13: MAC Configuration 3

functional units is reduced to 1 (Figure 4.15), then a non-fully pipelined configuration becomes optimal. In Figure 4.16, the *MUL* unit is much more expensive than the other functional units, and a pipeline configuration with only 2 pipeline registers is optimal.



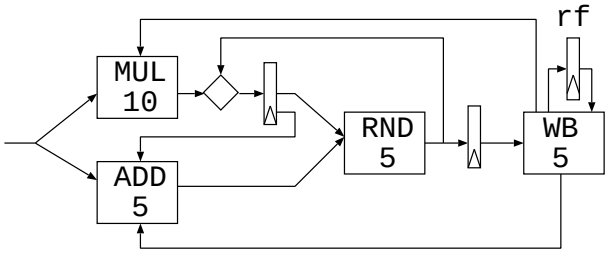

```
Clock Period = 15
Throughput = 0.06667
```

Figure 4.14: MAC Results: Example 1



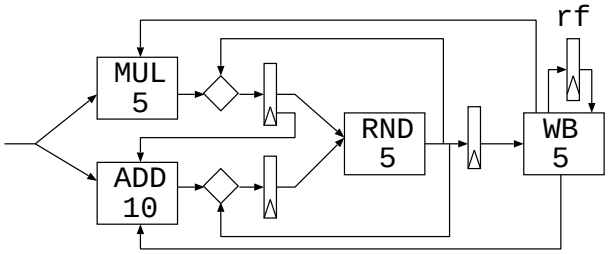

```
Clock Period = 7
Throughput = 0.14286
```

Figure 4.15: MAC Results: Example 2




```
Clock Period = 20
Throughput = 0.05
```

Figure 4.16: MAC Results: Example 3




```
Clock Period = 20
Throughput = 0.05
```

Figure 4.17: MAC Results: Example 4

4.3 Summary

The DLX and MAC were used to demonstrate the pipeline optimization process, and to show the variety of pipeline configurations that can result. The resulting pipeline configurations for the DLX are quite varied. We showed that changing the cost of a single functional unit by a relatively small amount could affect which pipeline configuration is optimal. On the other hand, the MAC optimal pipelines are much more consistent. In the MAC, there is very little hazard hardware required, so the fully pipelined configuration often had optimal throughput.

Chapter 5

Conclusion and Future Work

The main goal of our work is to pipeline a microprocessor optimally for throughput given a dataflow decomposed into functional units, timing costs for each pipeline component, and data dependency (DD) information. The partitioning of a pipeline into stages may cause hazard situations. The presence or absence of hazards changes the throughput calculation, making pipelining a non-trivial task. This thesis presents a method for optimally pipelining a circuit for throughput by adding the appropriate hazard circuitry to the pipeline automatically and adjusting the throughput calculation accordingly.

5.1 Contributions

The four main contributions of this thesis are:

- the creation of the Data Description abstraction for characterizing hazard situations,
- the development of a set of generic rules for hazard resolution,

- the instantiation of the rules to yield a SAT problem to which the solution is a pipeline configuration, and
- the use of reinterpretation to calculate the clock period and throughput for determining an optimal pipeline configuration.

The *DD* description level considers 3 types of data manipulation, namely *needs*, *writes* and *computes*. Data dependencies occur when functional units, in the same or different stages, manipulate the same piece of data. Using this DD information, we have developed a set of generic rules for the placement of hazard resolution circuitry. By working at a high level of description, our method can be applied early in the design process.

Our five hazard resolution rules use the DD information to outline the circumstances where there is a need for hazard resolution circuitry. Three rules handle the need for a bypass (forwarding logic). The fourth rule describes stall situations, which occur when the need for data and its computation are 2 or more stages apart. The last rule introduces kill circuitry, which is needed to resolve hazards caused by instructions that change the *pc*.

The rules are considered for all possible pipeline configurations and a set of Boolean constraints is generated. A universal pipeline is created by taking the union of all possible pipeline configurations. A SAT solver is used to find a satisfying assignment of the variables representing the presence or absence of pipeline registers and hazard circuitry. The resulting solution is a set of variable assignments, which represents a correct pipeline configuration. The universal pipeline can then be instantiated with the variable values to yield that pipeline configuration.

Reinterpretation is used to calculate the clock period. This technique allows us to use the same functional description of the circuit for simulation, visualization, verification and timing analysis. The throughput calculation is automatically adjusted to account for hazard circuitry that

delays the processing of instructions (stalls and kills). The throughput is calculated and compared with the current maximum. A constraint stating that the current solution is not allowed is then added to the current set of constraints and the SAT solver finds another solution. This process continues until all solutions have been found by the SAT solver. The result is that the maximum throughput has been calculated and the optimal pipeline configuration code is generated.

5.2 Case Studies

We performed case studies on the DLX and on a MAC. Optimal pipeline configurations were found for these 2 circuits using particular timing information. We showed that a relatively small change in the cost of a single functional unit could result in different optimal pipeline configurations. We also showed an example of a pipeline with a lower clock period, which produced a higher throughput than a second pipeline that had a higher clock period. These results were due to the hazard resolution circuitry present. This demonstrates the complexity of the problem and shows how something as seemingly simple as the DLX is still difficult to pipeline.

5.3 Limitations

There are several restrictions on the types of circuits that we can optimize. We do not handle out-of-order execution of instructions. Out-of-order execution may introduce write-after-read or write-after-write data hazards into the pipeline and we do not have rules to resolve these types of data hazards.

We cannot apply Rules 4 and 5 to dataflows containing forks or joins because we do not have rules that take into account instructions changing the *pc* and executing in parallel. The MAC

case study is an example of the bypass rules being applied to a pipeline containing a fork and join. Another limitation is that although our method handles data and control hazards, it does not handle structural hazards.

We assume that writebacks occur at the beginning of the computation of the next step (i.e., we have a write-before-read register file). Using a write-before-read register file is only a limitation of our current implementation and it would be possible to implement rules for a read-after-write register file.

5.4 Future Work

The next phase of our work is performing formal verification. We have used simulation to test our rules, but we have not formally verified them. An advantage of using reinterpretation is that we can also provide an interpretation suitable for verification. In the future, we plan to show that our pipelines are correct using the Burch-Dill commuting diagram approach [13]. We can use commuting diagrams to create a statement of correctness, which shows that a pipelined circuit has the same behaviour as an unpipelined circuit. We can then use the Stanford Validity Checker (SVC) to verify the statements of correctness [2]. Day et al [8] have already taken the approach of using commuting diagrams with SVC to verify circuits written in the specification style we use.

We would also like to verify the rules themselves. To do this, we would need a generic model of how pipelines behave. We would then verify that any pipeline that satisfies our rules is correct.

Another idea for future work is developing hazard resolution rules for structural hazards. Once the situations that cause hazards have been characterized, then duplicate hardware could be placed in the pipeline to resolve them.

Finally, we would like to investigate how our approach scales as we consider larger circuits with many more functional units.

Bibliography

[1] Satisfiability suggested format. *DIMACS*, May 1993.

[2] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science (LNCS)*, pages 187–201, 1996.

[3] Werner Buchholz, editor. *Planning a Computer System: Project Stretch*. McGraw-Hill, 1962.

[4] Jennifer P. L. Campbell and Nancy A. Day. High-level optimization of pipeline design. To appear in IEEE High Level Validation and Test Workshop (HLDVT), San Francisco, California, November 2003.

[5] Tim Chan, Amit Chowdhary, Bharat Krishna, Artour Levin, Gary Meeker, and Naresh Sehgal. Challenges of CAD development for datapath design. Technical report, Intel Technical Journal, 1st Quarter, 1999.

[6] Felix Sheng-Ho Chang and Alan Hu. Fast specification of cycle-accurate processor models. In *IEEE International Conference on Computer Design*, pages 488–492. IEEE Computer Society Press, 2001.

[7] George B. Dantzig. *Linear Programming and Extensions*. RAND Corporation, 1963.

[8] Nancy A. Day, Mark D. Aagaard, and Byron Cook. Combining stream-based and state-based verification techniques for microarchitectures. In *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 126–142. Springer-Verlag, November 2000.

[9] R. Goering. Get2chip describes pipeline synthesis advances. *EE Times*, December 2001.

[10] Mike Gordon. HolSatLib documentation, October 2001.

[11] John L. Hennessy and David A. Patterson. *Computer Organization and Design (2nd ed.): The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 1997.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture (3rd ed.): A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2003.

[13] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In David L. Dill, editor, *Proceedings of the Sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 68–80, Standford, California, USA, 1994. Springer-Verlag.

[14] Daniel Kroening and Wolfgang J. Paul. Automated pipeline design. In *Design Automation Conference*, pages 810–815. ACM Press, 2001.

[15] Robert W. Llewellyn. *Linear Programming*. Holt, Rinehart and Winston, Inc., 1964.

[16] Maria-Cristina Marinescu and Martin Rinard. High-level specifciation and efficient implementation of pipelined circuits. In *ASP-DAC Asia and South Pacific Design Automation Conference*, pages 655–661, 2001.

[17] Maria-Cristina V. Marinescu and Martin Rinard. High-level automatic pipelining for sequential circuits. In *14th International Symposium on System Synthesis (ISSS)*, pages 215–220, 2001.

[18] Tony Mason and Doug Brown. *lex & yacc*. O'Reilly and Associates, 1990.

[19] John Matthews and John Launchbury. Elementary microarchitecture algebra. In *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 288–300. Springer-Verlag, 1999.

[20] John Matthews, John Launchbury, and Byron Cook. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101, 1998.

[21] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[22] J. O'Donnell. Generating netlists from executable functional circuit specifications in a pure functional languge. In *Functional Programming Glasgow*, Workshops in Computing, pages 178–194. Springer, 1992.

[23] Brett Olsson, Robert Montoye, Peter Markstein, and MyHong NguyenPhu. RISC system/6000 floating-point unit. Technical Report SA23-2619, International Business Machines Corporation, 1990.

[24] L.C. Paulson. *ML For the Working Programmer*. Cambridge University Press, 1996.

[25] Sergei Romanenko, Claudio Russo, and Peter Sestoft. Moscow ML owner's manual, June 2000.

[26] Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[27] Barbara M. Smith. A Tutorial on Constraint Programming. Technical Report 95.14, School of Computer Studies, University of Leeds, 1995.

[28] Markus Weinhardt. Pipeline synthesis and optimization of reconfigurable custom computing machines. Technical Report iratr-1997-1, Universitat Karlsruhe, 1997.

[29] H. Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275, 1997.