Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter known, including display on the world wide web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Signature:

_____ 4/21/2010

Aaron L. Bush

Simulating a Pipelined CPU

by

Aaron L. Bush

Advisor Shun Yan Cheung

Department of Mathematics and Computer Science

Shun Yan Cheung
Advisor

Ken Mandelberg
Committee Member

Tracy Morkin
Committee Member

4/21/2010

Simulating a Pipelined CPU

by

Aaron L. Bush

Advisor Shun Yan Cheung

An abstract of
A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Sciences with Honors

Department of Mathematics and Computer Science

2010

Abstract

Simulating a Pipelined CPU

By Aaron L. Bush

The CS355 course in Computer Architecture teaches both entry-level and advanced pipelined CPU technology. For most of the computer components discussed in lectures, the course uses a circuit simulation program that allows students to gain an interactive experience with basic computer technology. However, the course has always lacked a simulation program for the pipelined CPU. Due to the complex nature of pipelining and the success with using the simulation software for other circuits, we have developed two versions of the pipelined CPU using the simulation program. By offering students a hands-on understanding of instruction pipelining, our simulated processors will greatly enhance the course.

Simulating a Pipelined CPU

by

Aaron L. Bush

Advisor Shun Yan Cheung

A thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Sciences with Honors

Department of Mathematics and Computer Science

2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preface

Computer technology is constantly being used to replace ordinary products that are used in everyday life. When thinking of computer technology, the first item that comes to mind is naturally the personal computer; however, the term computer *technology* actually encompasses a much broader collection of appliances. The chief component of this technology is the **central processing unit (CPU)**, more commonly known as a "processor" or a "core." In addition to its existence in the personal computer, CPU's are also used in a wide variety of other instances, ranging from common items like digital cameras and MP3 players to more complex applications like aeronautics and space travel. Nowadays, with the massive amounts of data that some of these appliances must handle and the necessity for lightning-fast process-

ing speeds, many of these devices are being manufactured with multi-core processors.

With the increasing presence of computer technology in everyday life, there is also a rising demand for individuals who are trained with a computer science background. The CPU is essential to most modern-day electronic devices. For this reason, it is imperative that university computer science departments are providing adequate instruction to their students on the theory and design of modern CPU's.

1.2 Background Information

At Emory University, the CS355 course in Computer Architecture provides students with the necessary knowledge in computer technology. This course highlights a wide range of digital components, beginning with very simple circuits, such as an addition circuit, and culminating with extremely complex systems like memory and processors. To assist both the professors and the students in lecturing on and learning the course material, the Department of Mathematics and Computer Science has installed a logic simulator named "Logic-Sim[2]" on all of its machines. For most of the topics discussed in this course, the computer science faculty has developed circuit simulation

programs so that the students can interact with the components and examine how they work. There is even a simulation program for a working processor. However, this program only simulates a basic, non-pipelined CPU. Modern processors use a technique known as instruction pipelining, which allows the CPU to simultaneously operate on several different instructions. Such a CPU can execute computer instructions at a faster rate.

While the non-pipelined CPU cannot adequately teach students how modern computers work, lectures on the subject matter still remain in the course syllabus, as they provide students with a good introduction and background knowledge for discussions about the pipelined CPU. Unfortunately, a Logic-Sim version of the pipelined CPU has not existed in the past. The CS355 course will be greatly improved if a pipelined CPU simulation is available, as students can get the necessary hands-on experience to thoroughly appreciate how these present-day processors function. For this reason, we have taken the time to develop and thoroughly test a working version of the pipelined CPU in Logic-Sim. Specific details about Logic-Sim, the design of the pipelined CPU, and our simulation results will be presented in this thesis.

Chapter 2

Related Work

2.1 Logic-Sim

The Logic-Sim utility introduced in the previous chapter is freeware that was developed by Professor Richard J. Reid of Michigan State University. In order to use the software, the user must write a descriptor file whose format is defined by the Logic-Sim documentation. The basic components of the simulator are the AND, OR, and NOT logic gates, as well as some other special-purpose circuits. Additionally, Logic-Sim allows the user to incorporate switches and probes (LED lights) into the circuit design for input and output, respectively. Each line of the descriptor file corresponds to one particular logic gate, switch, or probe, and it is encoded in such a manner that allows the outputs of some of the components to be wired directly to the inputs of others. The actual software then interprets the circuit descriptor

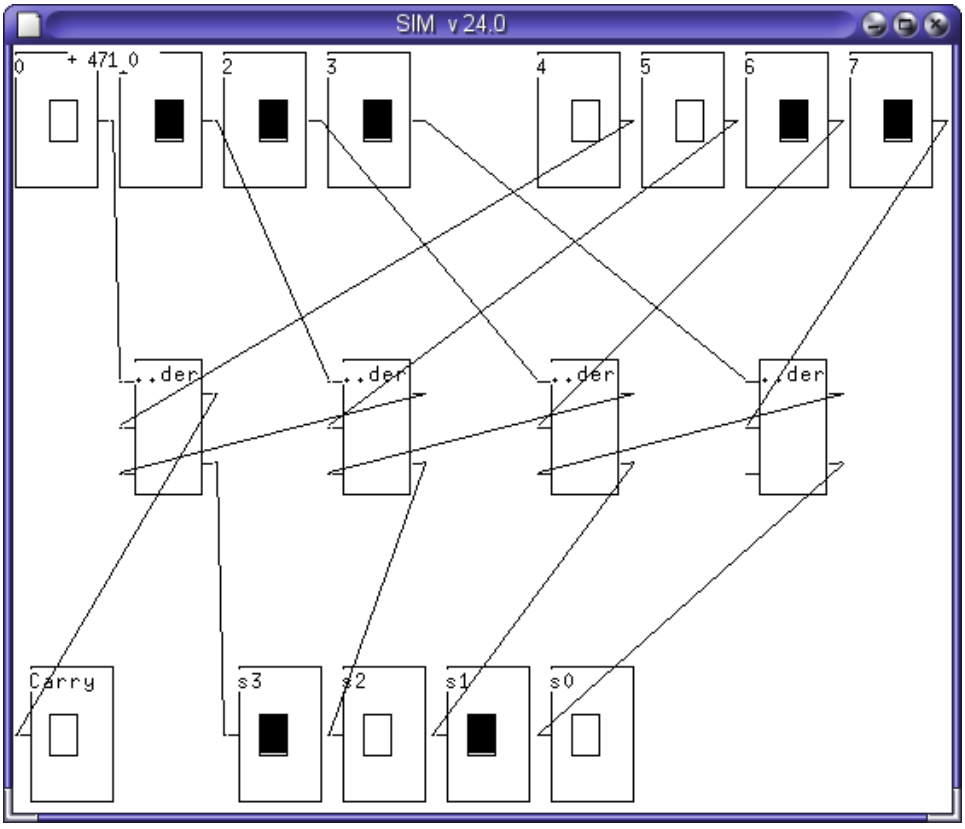

Figure 2.1: Logic-Sim Example: Addition circuit

file and generates a **graphical user interface (GUI)** that resembles the circuit design. The user can then interact with the GUI by pressing certain keys on the keyboard to toggle the input switches. As the switches are toggled, changes in the output probes can be monitored in order to observe the behavior of the given circuit[2].

Figure 2.1 is an example of a 4-bit addition circuit used in the CS355 course at Emory. Students can use the keyboard to toggle the input switches at the top-left and top-right corners of the screen. As shown, these switches are wired directly to the logic circuitry in the middle of the screen, and the output is connected to the probes at the bottom of the screen. From the current state of the circuit in this example, we can see that the user has inputted the values 7 and 3 (0111_2 and 0011_2, respectively) for the addends, and the addition circuit has correctly calculated the sum as 10 (1010_2).

2.2 Theory of Pipelining

When comparing computers today with those from several decades ago, one of the obvious differences is the improvement in the execution time of programs. Since a program is simply a set of machine instructions, computers have become faster because they have become more efficient at executing instructions. There are two main reasons why processors today are faster. One reason is the fact that the speed of the computer's clock has increased. For example, if the clock's period were reduced from 10 nanoseconds to one nanosecond, then the speed of the machine would increase tenfold. The processor speed's increase is due to better manufacturing technology that

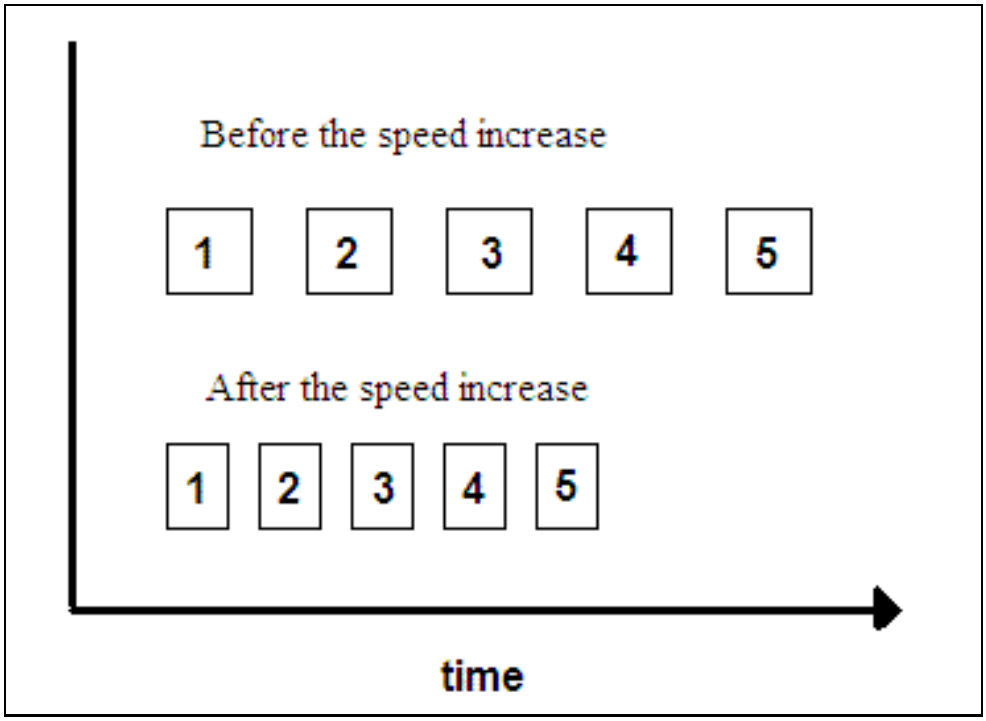

Figure 2.2: Performance increase due to faster clock

reduces the size of the circuit components. Figure 2.2 shows how the overall performance of the processor can be improved in this fashion. Notice that since the frequency of the clock has been changed, both the duration of each instruction and the gaps between each instruction are shortened proportionally. For various reasons, however, there is a physical restriction on the duration of a clock period[3].

The second reason is parallelism. In computer science, parallelism de-

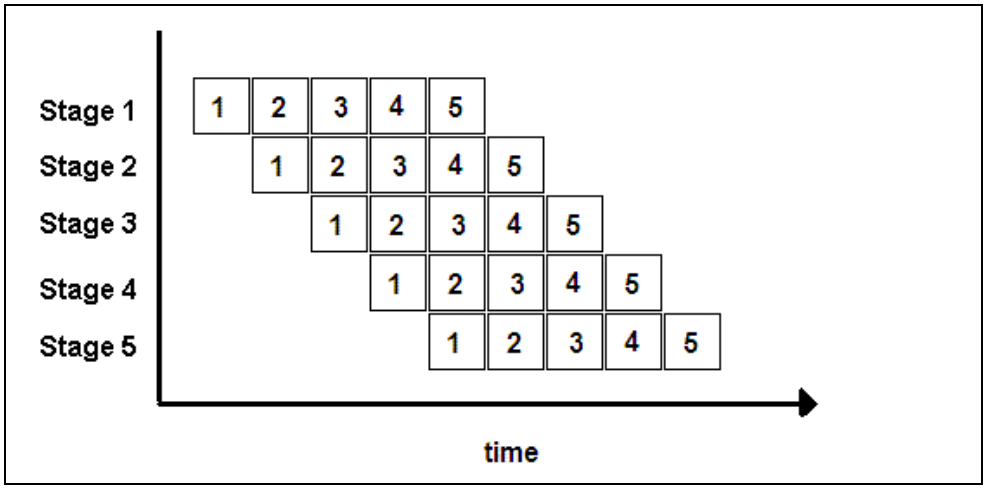

Figure 2.3: Performance increase due to instruction-level parallelism

scribes any method by which two or more jobs are worked on at the same time. Parallelism can be accomplished in two separate ways. Processor-level parallelism utilizes multiple cores, each of which is a complete CPU and is able to work on a separate task. Parallelism also exists in another form: pipelining. In pipelining, multiple instructions are simultaneously executed within one processor[3]. Using this strategy, the processor is engineered with multiple stages. Pipelining is similar to an assembly line in that new instructions enter the "assembly line" even while other instructions are still being worked on. Each stage in the pipeline has the ability to operate on its own instruction. An example of a processor with five stages is shown in Figure

2.3. At any given moment in time, up to five instructions can be active within the CPU. Pipelines can be combined with multi-core technology to increase the speed of processors even further.

Pipelining does not increase the speed at which individual instructions are completed. In fact, individual instructions could potentially have a longer execution time in the pipelined CPU than that same instruction in a non-pipelined version of the same CPU. Instead, the performance enhancement occurs because of the *overlapped* execution that occurs, which is apparent in Figure 2.3. So even though one instruction might require multiple clock cycles to make its way through the pipeline, in the long run, an average of one instruction will complete execution in each cycle. It is for this reason that pipelining allows the CPU to run faster. Under ideal conditions, this speedup technique increases a program's execution speed by a factor of the number of stages in the pipeline[1].

Chapter 3

The Basic Pipelined CPU

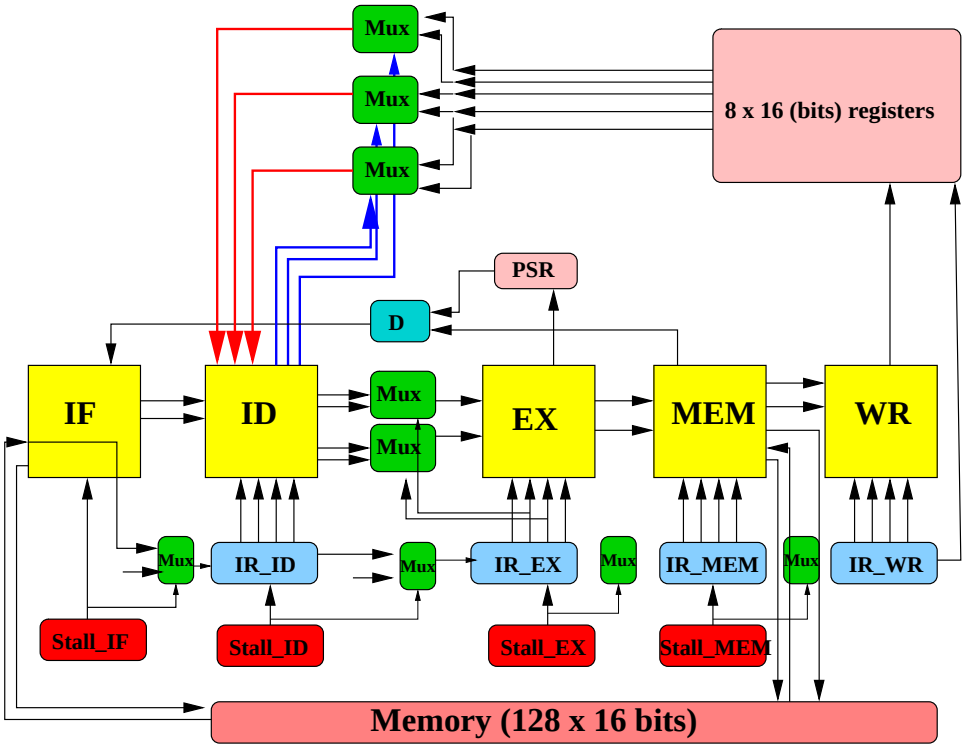

Figure 3.1: Design of the Basic Pipelined CPU

3.1 Overview

The Basic Pipeline discussed in the CS355 class has five stages, depicted by the boxes labeled IF, ID, EX, MEM and WR in Figure 3.1. Each stage, with the exception of the first one, has an **instruction register (IR)** associated with it. The IR stores the machine instruction so that each stage of the pipeline can know exactly what it needs to accomplish. The processor also has eight data registers which are used to store values locally within the CPU. The first register, `R0`, is read-only memory that always stores the value zero. The other registers, `R1` - `R7`, can store any 16-bit value. Also, the CPU is connected to the main memory that contains the program instructions. In addition to these components, there are numerous other circuits that are necessary to facilitate the execution of each instruction through the pipeline. The different circuits shown in the figure will be discussed in detail throughout this chapter.

At the first stage in the pipeline, the **Instruction Fetch (IF)** stage, an instruction is read from memory and then passed on to the **Instruction Decode (ID)** stage. There, the processor will fetch all values that could possibly be needed in order to execute the given instruction. The **Execution**

(EX) stage will examine the instruction to determine exactly which ones are used so that it can properly calculate the result. The last two stages of the pipelined CPU handle the updating of the result. The **Memory (MEM)** stage of the pipeline handles instructions that read from or write to memory. In the Basic Pipelined CPU, the MEM stage is also responsible for handling branch instructions, which will be explained in Section 3.2.1. The **Write Back (WR or WB)** stage of the pipeline handles instructions that write to a register in the CPU. The result of the executed instruction is stored in one of the eight registers so that it can be used as one of the operands in a later instruction. The rest of this chapter will explain exactly how the circuits work in unison to produce a properly working pipelined CPU.

3.2 Instruction Encoding

Every processor has a pre-defined set of instructions that it is capable of executing. Each instruction, along with its operands, is encoded in a machine language (binary code) that a processor understands. Different processors have different instruction encoding, which is why an assembly program written for an Intel processor cannot be run on a SPARC processor, and vice versa.

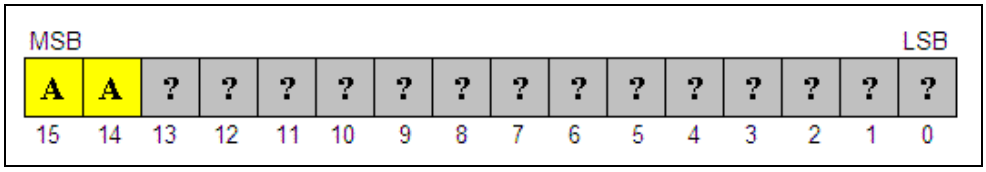

Figure 3.2: Instruction Encoding: Bits 15-14

A	A	Instruction Type
0	0	ALU (Logic)
0	1	Load (memory)
1	0	Store (memory)
1	1	Branch

Table 3.1: Instruction Encoding: Bits 15-14

For our simple pipelined processor (SPP), we have created our own simple machine language, which uses a fixed length of 16 bits to encode each instruction and its operands. The general format of the SPP instruction is given in Figure 3.2. Recall from Section 3.1 that the CPU used in CS355 has eight data registers. Within the instruction encoding, exactly three bits will be needed to reference a single one of these registers (since $\log_2 8 = 3$). Because some instructions have two registers as source operands and a third register for the destination of the output (e.g. `R4` = `R2` + `R7`), nine bits of the instruction are reserved for referencing registers. The remaining seven bits are used to encode the type of the instruction, as well some other flags that

tell the CPU exactly how to handle the given instruction. There are three types of instructions: ALU instructions, memory access instructions (LD and ST), and branch instructions. We have designed an instruction encoding for our CPU in such a way that the meaning of any given bit depends on the type of instruction. The meaning of bits 0 to 13 in the instruction depends on the values of bits 14 and 15. This strategy enables our machine language to encode more types of instructions than if every bit of the instruction had a fixed meaning. The first two bits, which are labeled `AA` as seen in Figure 3.2, are the only bits in the instruction that always have the same meaning. The four possible values and their definitions are shown in Table 3.1. The remaining bits have different meanings for different values of `AA`, which is why bits 0 through 13 are shown with question marks in the figure.

3.2.1 Branch Instructions

Generally speaking, instructions in a computer program are executed in consecutive order. However, most computer algorithms require that the processor execute some instructions out of order; typical examples of such program constructs are conditional statements and loops. In order to achieve this type of behavior, the CPU provides a special type of instruction, called

a **branch instruction**.

Branch instructions tell the CPU to retrieve the next instruction from a specific location in memory, rather than simply fetching the instruction from the next memory address. The specific location to which the CPU must jump can be encoded into the instruction in two different ways. In **direct branching**, the branch address is explicitly stored within the branch instruction. The number of bits needed to encode the destination address in direct branching depends on the size of the memory. If the CPU can address 65,536 (2^{16}) bytes of memory, the number of bits needed to encode the destination address in direct branching is 16. In our instruction encoding, there would be not be enough bits to encode the branch instruction in this manner.

The second kind of branching instruction is **relative branching**. The destination address is relative to the (current) branch instruction. Relative branching is the most often used type of branch instruction because it results directly from the translation of `if`-statements and `while`-statements. Direct branching is used to support subroutine calls, but since our CPU does not support subroutines, it does not support direct branching either. For didactical purposes, both types of branching are identical. Relative branching

is more practical though, because `if`-statements and `while`-statements are commonly used in programs.

As briefly mentioned, some branch instructions will only perform the branch if a certain condition is met. For example, the branch instruction resulting from an `if` statement should only be executed if the test from the conditional statement is passed. Within a program, conditional branches will almost always follow a compare instruction. **Compare instructions** subtract their operands and set the various flags according to the outcome. For example, if the result is negative, the N flag is set, and when the result is zero, the Z flag is set. The other flags are the overflow (V) flag and the carry (C) flag. From the setting of the N,Z,V and C flags, the logical relationship (equal to, less than, greater than, etc.) between the compare operands can be determined. Some condition bits are harder to compute than others (e.g. the overflow bit requires knowledge of the sign of the operands). Because our simulated CPU is for instructional purposes only, we have made some simplifications. Our CPU does not compute the V and C flags. As a result, some conditional branches are not supported. However, for the purpose of illustrating the pipeline concept, our design is adequate.

The branch condition is encoded by the bits labeled `BBB` in Figure 3.3.

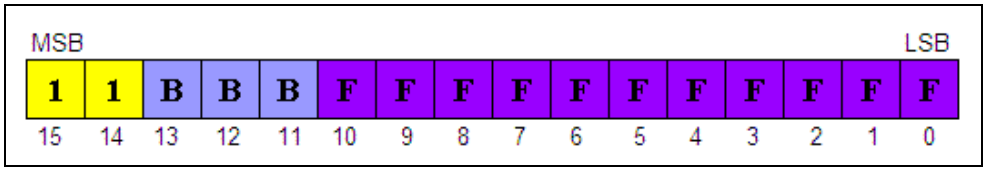

Figure 3.3: Instruction Encoding: Branch Instructions

B	B	B	**Branch Condition**
0	0	0	BRA (always)
0	0	1	BEQ (equal)
0	1	0	BNE (not equal)
0	1	1	BLT (less than)
1	0	0	BLE (less than or equal)
1	0	1	BGT (greater than)
1	1	0	BGE (greater than or equal)
1	1	1	*(not used)*

Table 3.2: Instruction Encoding: Branch Instructions

The different branch conditions are outlined in Table 3.2. When a conditional branch is being evaluated, the CPU uses these three bits along with the N and Z flags to determine whether or not the branch condition is met. If it does decide to branch, the destination address will be calculated by adding the memory address of the branch instruction to the signed offset value given in the bits `FFFFFFFFFFF` of Figure 3.3. If the branch is not taken, then the CPU will simply fetch the instruction from the next address in memory as it usually does for non-branching instructions.

3.2.2 Non-Branch Instructions

Apart from branching, there are three primary types of instructions.

1. Compute a value using the arithmetic logic unit (**ALU**)
2. Load (**LD**) a value from memory into one of the data registers
3. Store (**ST**) a value from one of the data registers to memory

All three of these types of instructions have a very similar instruction format, shown in Figure 3.4. For ALU and LD instructions, which are the only two types that will update one of the data registers, the destination register is encoded by `DDD`. In a ST instruction, however, `DDD` encodes the source register which contains the value to be written to memory. The `P` flag indicates whether or not the **program status register (PSR)** flags should be updated as a result of the ALU's computation. It is typically only used for ALU instructions, although it could theoretically be used for LD and ST instructions also. The PSR contains the N and Z flags that were discussed in

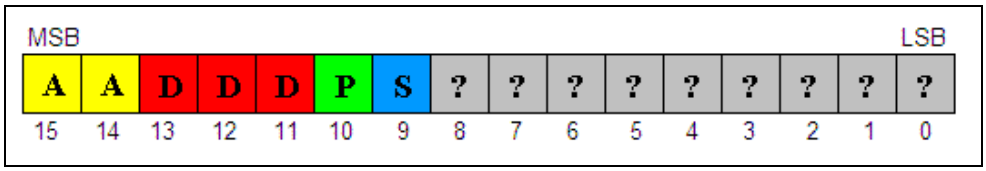

Figure 3.4: Instruction Encoding: Non-Branch Instructions

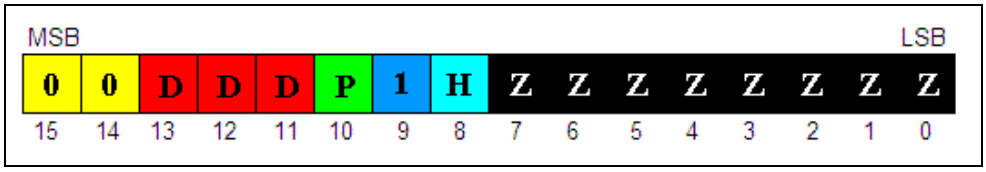

Figure 3.5: Instruction Encoding: SETHI and SETLO

Section 3.2.1. If the `P` bit is set and the result of the instruction is negative, then the EX stage of the pipeline will set the N flag. Likewise, if the `P` bit is set and the result is zero, then the Z flag will be set. If the instruction has the `P` bit cleared, then the PSR flags will remain unchanged.

The `S` bit denotes a sethi or setlo instruction. This flag is only used with ALU instructions. Figure 3.5 outlines the encoding format for these types of instructions. The `H` bit is set for a sethi instruction and cleared for a setlo instruction. The functionality of these instructions is very straightforward. For a **sethi instruction**, the upper (most significant) 8 bits of data register `DDD` is set to the value in the `ZZZZZZZZ` bits of the instruction, and the remaining bits are cleared. Similarly, a **setlo instruction** will store the value `ZZZZZZZZ` in the lower (least significant) 8 bits of the specified data register. However, in the setlo instruction, the other bits will remain unchanged. The sethi and setlo instructions are typically used in conjunction with each other

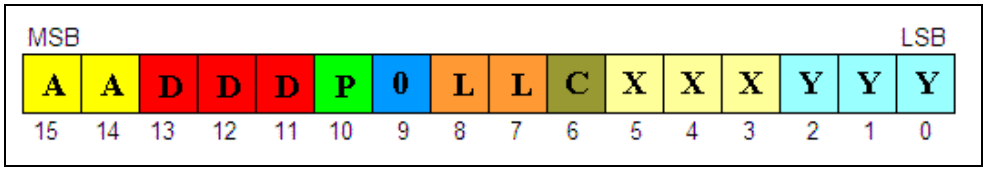

Figure 3.6: Instruction Encoding: ALU, LD, ST

(sethi followed by setlo) to store a 16-bit value into a data register.

When the `S` bit is cleared, the instruction is an ALU instruction that performs a computation on two source operands. The instruction has the format shown in Figure 3.6. The two `LL` bits specify one of the four ALU operations outlined in Table 3.3. The `C` bit indicates a constant operand. When the `C` flag is set, the bits `YYY` encode a constant in the range -4 to +3. Otherwise, the `YYY` bits represent a register number. The bits `XXX` always represent a register number.

L	L	ALU Operation
0	0	Addition
0	1	Subtraction
1	0	Binary AND
1	1	Binary OR

Table 3.3: Instruction Encoding: ALU, LD, ST

As we stated earlier, LD and ST instructions should always have the `S` bit cleared. Thus, the format of these instructions also follows Figure 3.6. For both of these memory-access instructions, the actual memory address will be determined by bits 0 through 8. When the instruction encodes the addition operation (`LL` equals `00`), as LD and ST instructions usually do, the memory address used by these instructions is the sum of register `XXX` and the constant `YYY` if the `C` bit is set. Otherwise, if the `C` bit is not set, the address is the sum of registers `XXX` and `YYY`. If the instruction encodes a different `LL` operation, then the memory address will be calculated accordingly.

Figures 3.7, 3.8, and 3.9 show sample instruction encodings for an ALU, LD, and ST operation.

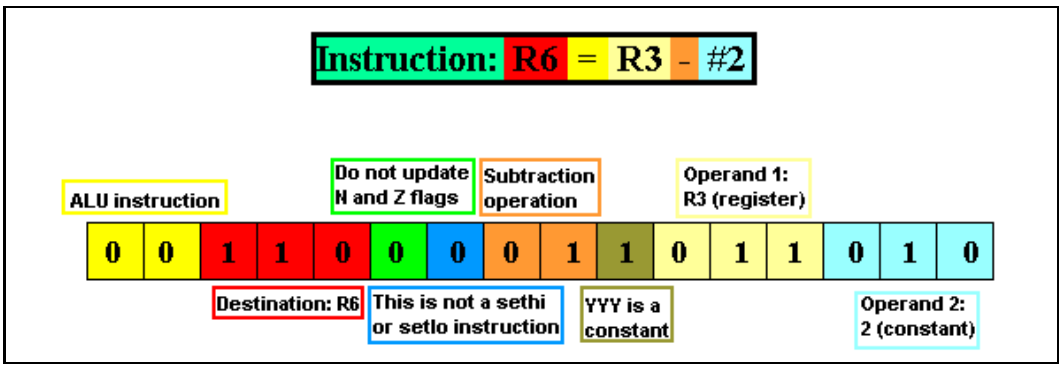

Figure 3.7: Example of ALU Instruction Encoding

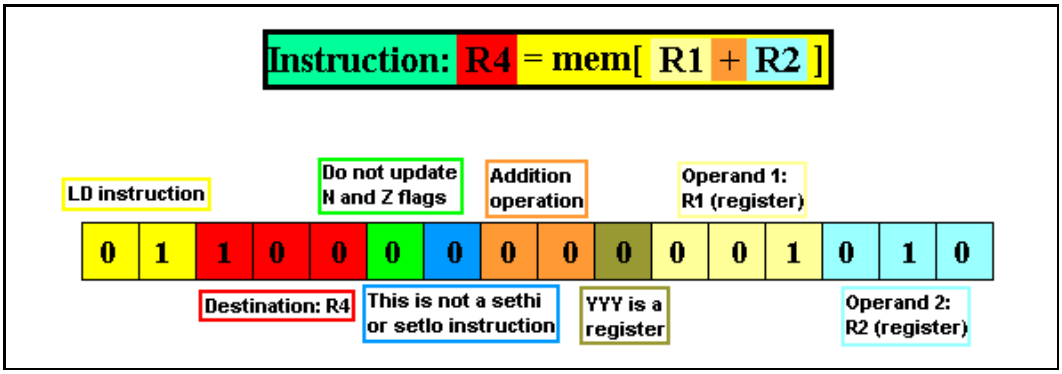

Figure 3.8: Example of LD Instruction Encoding

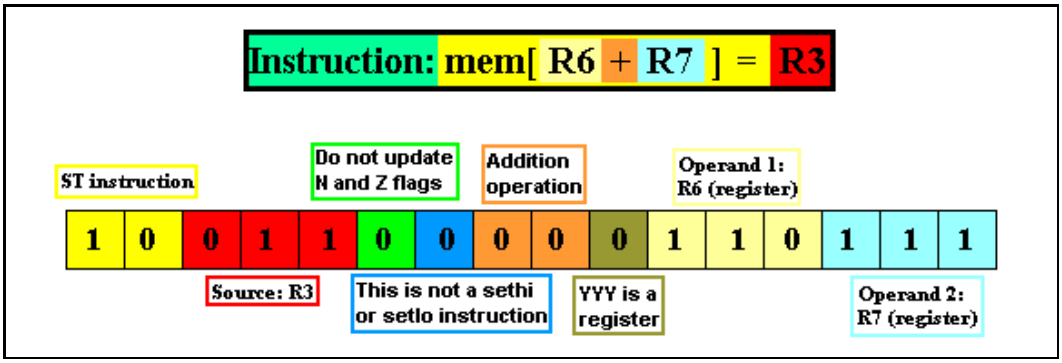

Figure 3.9: Example of ST Instruction Encoding

3.3 Instruction Fetch (IF) Stage

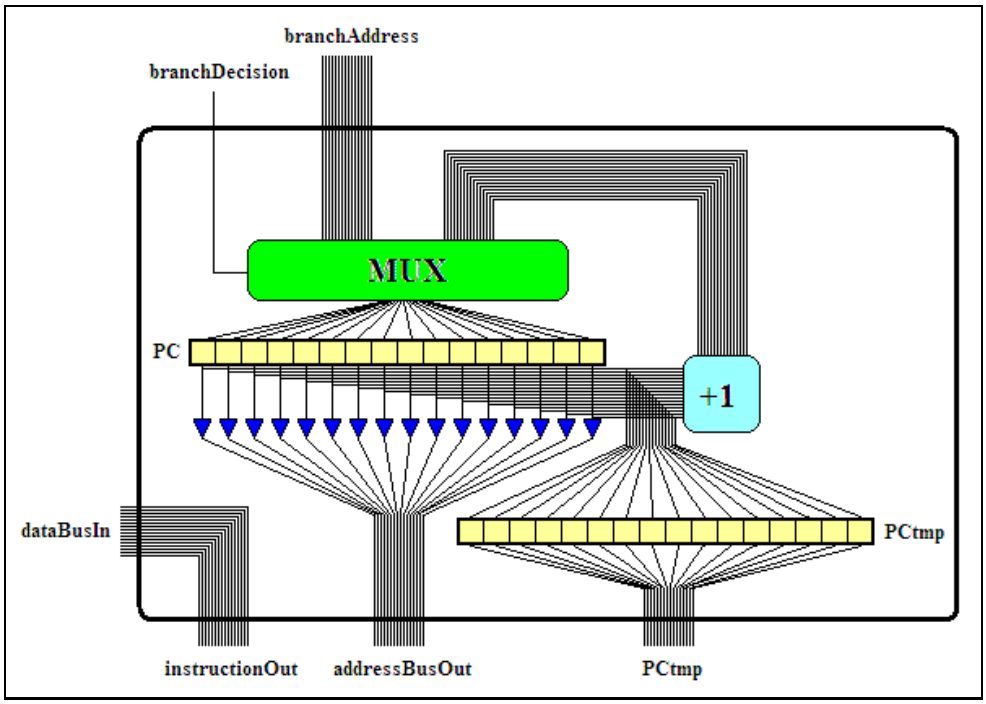

Figure 3.10: Circuit Diagram of the IF stage

The Instruction Fetch (IF) stage is the first stage of the pipelined CPU. The primary purpose of the IF stage (shown in Figure 3.10 with the inputs and outputs described by Table 3.4) of the pipeline is to read the next instruction from memory and store it in the instruction register (IR) of the next stage in the pipeline (the ID stage), so that it can be executed

Inputs	Outputs
reset*	memRequest*
clock*	memRead*
stall-IF*	addressBusOut
dataBusIn	instructionOut
branchDecision	PC-tmp
branchAddress	

** Not pictured in diagram*

Table 3.4: IF Stage Inputs and Outputs

by the CPU. In order to retrieve the instruction from memory, the CPU must know the memory address of the instruction. For this purpose, the IF stage maintains the address of the next instruction in a special purpose register called the **Program Counter (PC)**. After fetching an instruction, the IF stage computes the address of the next instruction by adding 1 to PC. However, in the case of branching, the branch address must be used as the location of the next instruction.

Within the IF stage, there is another special purpose register called **PC-tmp**. While the PC always keeps track of the address of the *next* instruction to be fetched, the PC-tmp keeps track of the address of the instruction that was *most recently* fetched into the IR of the ID stage. The PC is updated with the value from the multiplexor circuit (labeled "MUX") at each CPU

cycle. One of the inputs of the MUX is the value PC+1. The other input of MUX is the effective branch address of the instruction in the MEM stage. The multiplexor will either choose the output of the addition circuit (labeled "+1") or the branch address, depending on the signal outputted by the branch decision circuit (shown as D in Figure 3.1 and described in Section 3.6.1). The multiplexor thus selects the address of the next instruction to be stored in the PC.

Figure 3.10 shows that the output of the PC is also connected to a series of tri-state buffers (depicted as small triangles). **Tri-state buffers** are circuits that can be either enabled or disabled. When they are enabled, the input signal (either high or low) is passed directly through to the output. However, when they are disabled, the tri-state buffer will disconnect the input from the output. Tri-state buffers are used to prevent multiple outputs from being connected together, which can result in problems. If multiple outputs are connected to the address bus at the same time, then Logic-Sim will report errors. If this were to happen in a real computer, the hardware would be damaged. Therefore, all values written to the address bus (and the data bus for that matter) must pass through these tri-state buffers, and the timing of these devices must be carefully planned so that there is never more than one

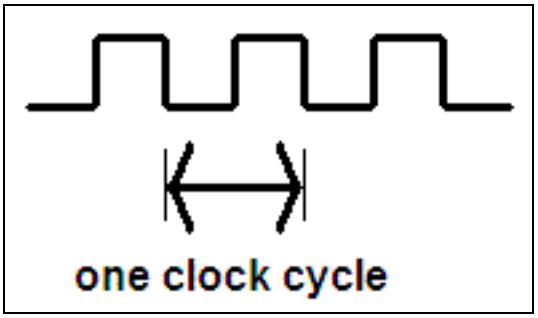

Figure 3.11: CPU Clock Cycle

value being written to the bus at any given time.

Clearly, timing within the CPU is critical. To maintain the integrity of the data, events must occur in a certain order. Each stage of the CPU completes its execution in one clock cycle. A clock cycle (Figure 3.11) consists of a **rising edge** (when the signal changes from low to high) and a **falling edge** (when the signal changes from high to low). During the rising edge of the clock, the IF stage sends out the `memRequest` signal (which informs memory to expect a request), and it also sends out the `memRead` signal (which tells memory to retrieve data instead of store data). Also during the rising edge, the tri-state buffers are activated so that the PC is sent on the address bus. At that point in time, the memory has all the necessary information to perform its task. In response to the request signals, the external memory

places the content of the given address on the data bus. As shown, this data simply passes through the IF stage to the `instructionOut` signal, which will be sent to the instruction register of the ID stage. During the falling edge of the clock, the PC-tmp register will be updated with the value from the PC register, and the PC register will be updated with the address of the next instruction (which was determined by the multiplexor). After the completion of a full cycle, one machine instruction has been fetched into the IR of the ID stage, and the CPU will be ready to fetch the next instruction.

There is also a stall input signal into the IF stage. The stall signal is part of a mechanism that will prevent the IF stage from accessing the memory during the execution of a LD or ST instruction. In effect, this signal masks the signal of the clock so that when the stall signal is on, nothing will be sent to the address bus during the rising edge of the clock. Also, PC and PC-tmp will not be updated. Details of the stalling mechanism will be given in Section 3.8.

3.4 Instruction Decode (ID) Stage

Every time a new instruction gets fetched, it is placed inside the instruction register (IR) of the Instruction Decode (ID) stage. The ID stage's

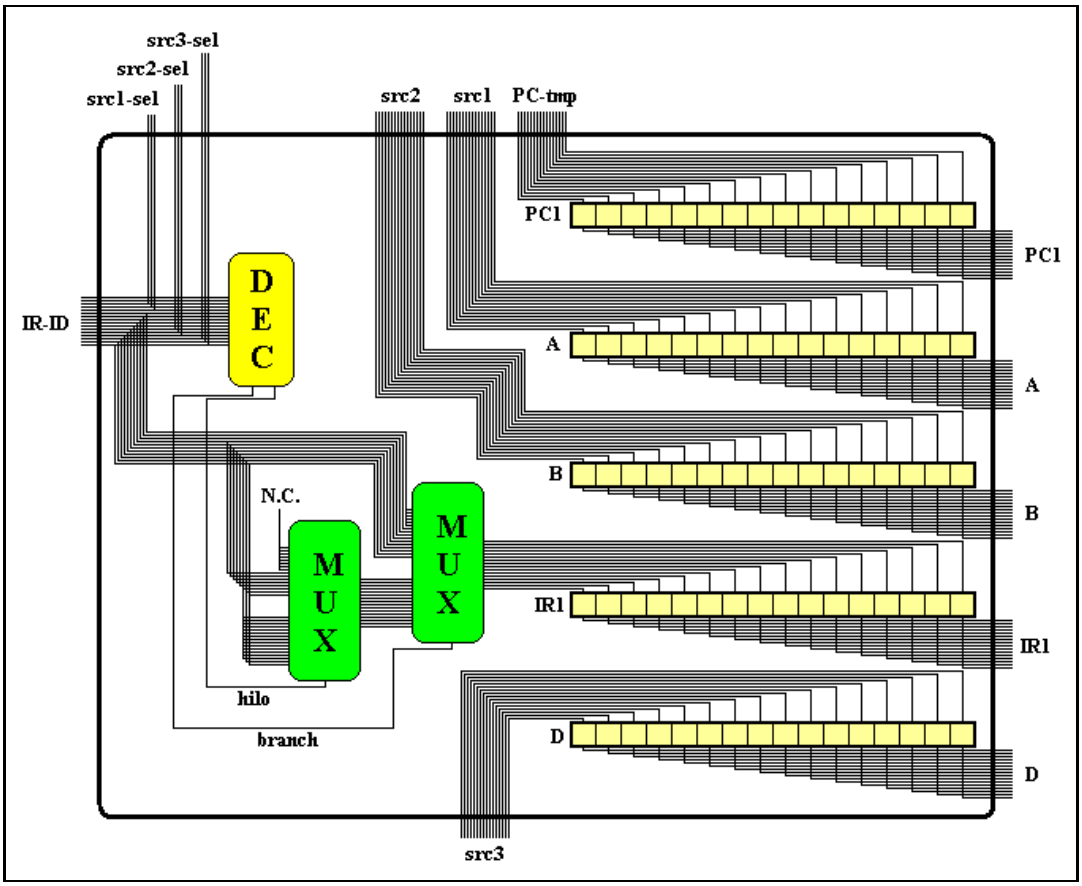

Figure 3.12: Circuit Diagram of the ID stage

Inputs	Outputs
reset*	src1-sel
clock*	src2-sel
stall-ID*	src3-sel
IR-ID	PC1
PC-tmp	A
src1	B
src2	IR1
src3	D

** Not pictured in diagram*

Table 3.5: ID Stage Inputs and Outputs

responsibility is to gather all possible operands that *might* be needed in order to execute an instruction. The circuit diagram is shown in Figure 3.12, and the inputs and outputs are listed in Table 3.5. These operands are placed in the five registers, labeled **PC1**, **A**, **B**, **IR1**, and **D**. The ID stage does not know the type of instruction being executed. As a result, some of the operands retrieved will not be used by the given instruction, but doing the extra work will not cause execution errors and makes the ID stage less complex to design.

During the first half of the clock cycle, the ID stage sends selection signals to the registers to obtain the desired values. The values of various operands will be ready prior to the end of the cycle. At the falling edge of the clock,

the five registers will be updated with their new values. The PC-tmp register in the IF stage will be written into the PC1 register in the ID stage.

Three of the five operands are encoded in the instruction. The first source operand is encoded by the `XXX` bits, and this register is fetched into the A register. The second source operand is `YYY`, and its value is stored in the B register. The `DDD` destination field encodes the source register for a ST instruction. This register is stored in the D register. The selection mechanism used to switch the values from the register bank to the respective registers is a multiplexor (shown at the top of Figure 3.1). Lastly, the IR1 register will contain the constant that could be encoded within the instruction itself.

In retrospect, the ID stage does have some knowledge of the instruction encoding. Since different types of instructions encode their operands differently, the instruction type must be known in order to use the correct bits in the IR. Specifically, the constant used in branch, sethi, and setlo instructions are encoded differently than all other instructions. Branch instructions encode a signed constant in bits 0 through 10. The sethi and setlo instructions encode an unsigned constant in bits 0 through 7. All other instructions can have a signed constant encoded in bits 0 through 2. Then based on the instruction type, the two multiplexors shown in the figure will extract the

proper constant. At the end of the clock cycle, the appropriate constant will be stored in the IR1 register.

Similar to the IF stage, the ID stage also has a stalling mechanism. When the stall signal is on, none of the registers will be altered. In the absence of stalling, all registers will be updated during the falling edge of the clock cycle, and the content of the IR in the ID stage will be passed on to the IR in the EX stage.

3.5 Execution (EX) Stage

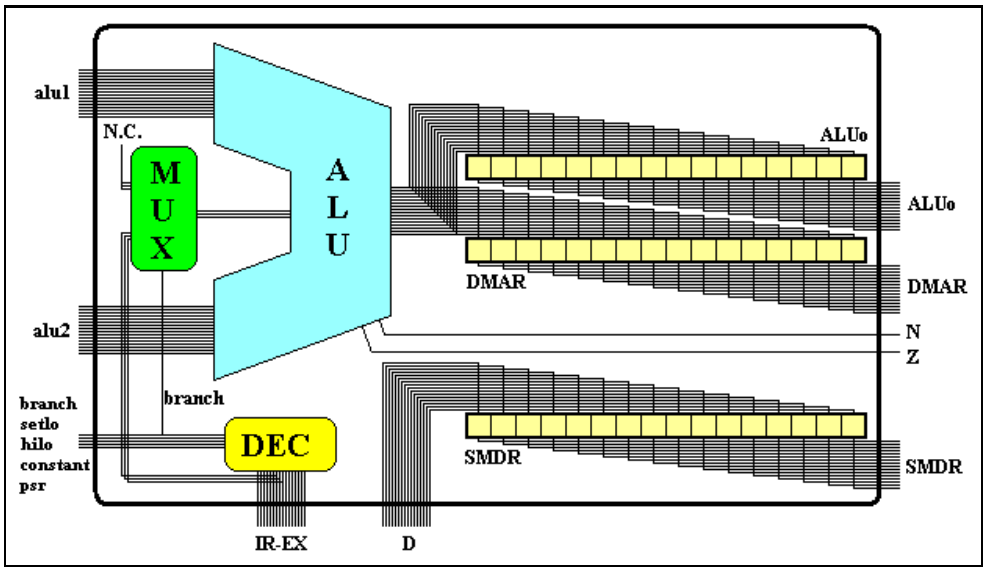

Figure 3.13: Circuit Diagram of the EX stage

After the ID stage has fetched *all* the values that the instruction *might* need, the Execution (EX) stage must then determine exactly *which* ones it *does* need so that it can perform the necessary calculations. In the first phase, we determine which of the five registers from the ID stage to use. This selection process is handled by several circuits that are external to the EX stage (depicted in Figure 3.1 as the multiplexors located in between the ID and the EX stages). The specifications of those multiplexors will be presented in Section 3.5.1, but for the discussion of the EX stage, it is only important that the necessary operands will be available through the `alu1` and `alu2` inputs. The circuit diagram and input/output mapping for the EX stage are shown in Figure 3.13 and Table 3.6 respectively.

The central component of the EX stage is the **arithmetic logic unit (ALU)**. The ALU that we have constructed takes `alu1` and `alu2` as its two, 16-bit inputs. It also uses a three-bit control signal to select the operation that the ALU should perform. Table 3.7 shows the six different ALU operations and their respective bit patterns. The third bit is not used in the sethi and setlo operations.

The multiplexor that is shown in Figure 3.13 is responsible for determining the correct, three-bit signal to send to the ALU. For a branch instruction,

Inputs	Outputs
reset*	ALUo
clock*	DMAR
stall-EX*	SMDR
IR-EX	N
alu1	Z
alu2	branch
D	setlo
	hilo
	constant
	psr

** Not pictured in diagram*

Table 3.6: EX Stage Inputs and Outputs

the two ALU operands will be PC1 (the address of the branch instruction) and IR1 (the offset value that was encoded into the instruction). Since all branch operations require the use of addition on these two operands, the instruction does not explicitly encode the ALU addition operation. For this reason, the multiplexor must be sure to output the value `000` to the ALU for all branch instructions. For any other type of instruction, the multiplexor can simply output bits 7 through 9 of the IR as the control signal for the ALU by taking advantage of the way the instruction is encoded.

The ALU's output is connected to two separate registers: the ALU output register (**ALUo**) and the **destination memory address register**

Bits			Operation
0	0	0	ADD
0	0	1	SUBTRACT
0	1	0	AND
0	1	1	OR
1	0	?	SETLO
1	1	?	SETHI

Table 3.7: Control Signal to the ALU

(DMAR). While both of these registers will contain the same value, the ALUo will be used by ALU operations, and the DMAR will be used as the address in LD and ST operations. As is the case with the ID stage, whenever the stall signal is not set, all of the registers in the EX stage will be written at the end of the clock cycle.

In addition to outputting the result of the computation to the ALUo and DMAR registers, the ALU also outputs the values of the N and Z flags. The **N flag** is set when the result is a negative number, which is the case when the most significant bit is set. The **Z flag** is set when the result is equal to zero. Even though the ALU outputs these flags for every single instruction, the EX stage does not contain any memory in which their values can be stored. Instead, the flags are stored in the program status register (PSR),

which is located outside of the EX stage. The PSR will be discussed further in Section 3.5.2.

The third register shown in Figure 3.13 is the **store memory data register (SMDR)**. This register will contain the value that gets written to memory in a ST operation. Since the ID stage already obtained this value in the D register, that value simply gets copied into the SMDR during the EX stage. Like the other registers, the SMDR will be updated at the end of the clock cycle, but only if the stall signal is not active.

The decoder used in the EX stage is very similar to the decoder in the ID stage. It reads the contents of the instruction register to determine the exact type of the instruction, including any of the instruction flags. In this case, the decoder has outputs that indicate whether the instruction is a branch instruction, a setlo instruction, either sethi or setlo, and whether the `C` flag is set. All four of these signals are used in the circuitry for selecting the correct ALU operands (Section 3.5.1). The branch signal is also used in the multiplexor that determines the three-bit control signal for the ALU. Additionally, the decoder outputs the value of the `P` flag. After combining this value with the clock and stall signals, the EX stage outputs a signal that tells the PSR (Section 3.5.2) when to store the N and Z flags.

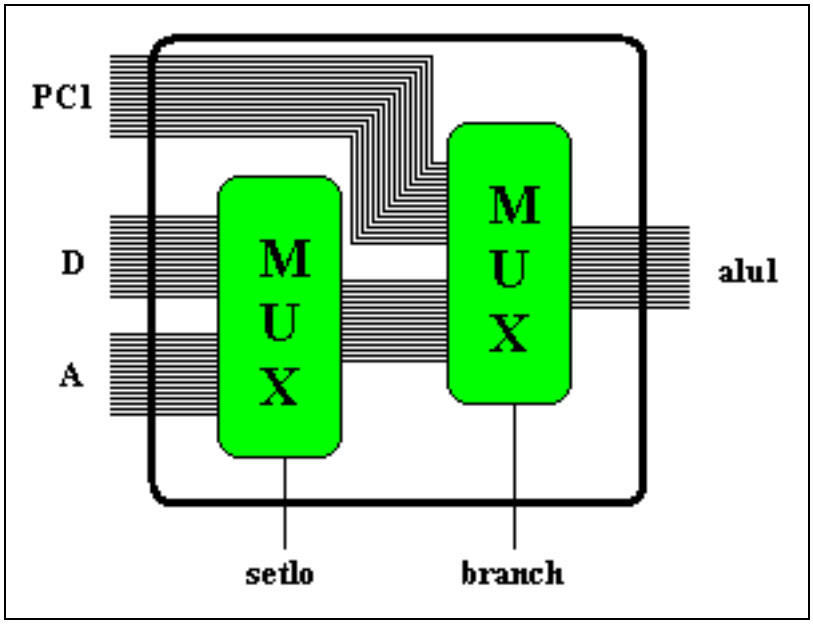

Figure 3.14: First Operand of the ALU

3.5.1 Selecting the Correct Operands for the ALU

The circuits to select the operands for the ALU implement a simple `if-else` logic using multiplexors. Figures 3.14 and 3.15 show the logic to select the first and second ALU operand, respectively. In the figures, the inputs that enter the circuit from the left side originate from the registers in the ID stage of the pipeline. Those that appear to enter from the bottom are selection signals from the EX stage. The wires that exit the right side of the circuits are the ALU operands, and these outputs are inputs of the EX stage.

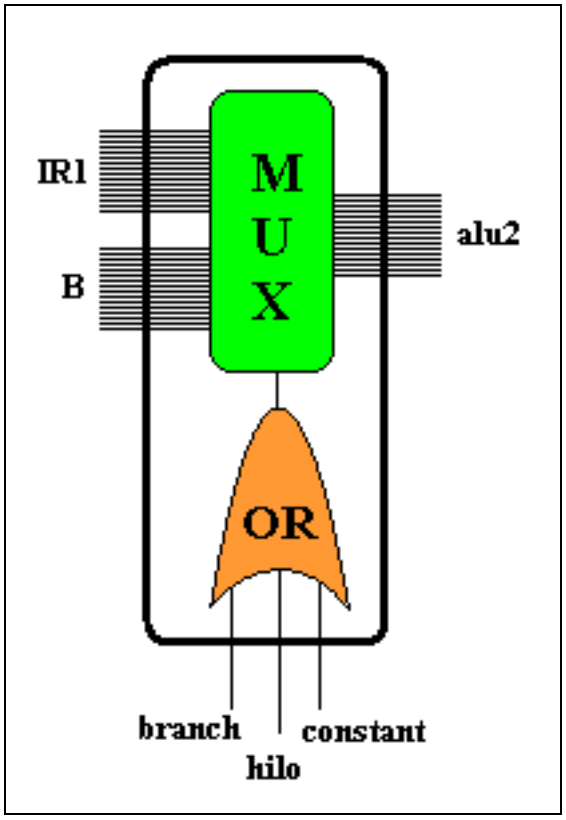

Figure 3.15: Second Operand of the ALU

The first operand, `alu1`, is determined as follows. If the instruction is a branch instruction, then `alu1` must select the PC1 register, which contains the address of the branch instruction. Otherwise, if the instruction is a setlo instruction, then the value of the D register must be used as the first operand. This is because the upper eight bits in the destination register are used in

the output of a setlo instruction. For the other types of instructions, the operand is a register value that is stored in the A register.

For the second source operand, the multiplexor must select between a register value in the B register and a constant value in the IR1 register. Branch instructions, sethi and setlo instructions, and instructions that set the `C` flag all require the use of a constant. All other instructions should select the B register.

3.5.2 The Program Status Register (PSR)

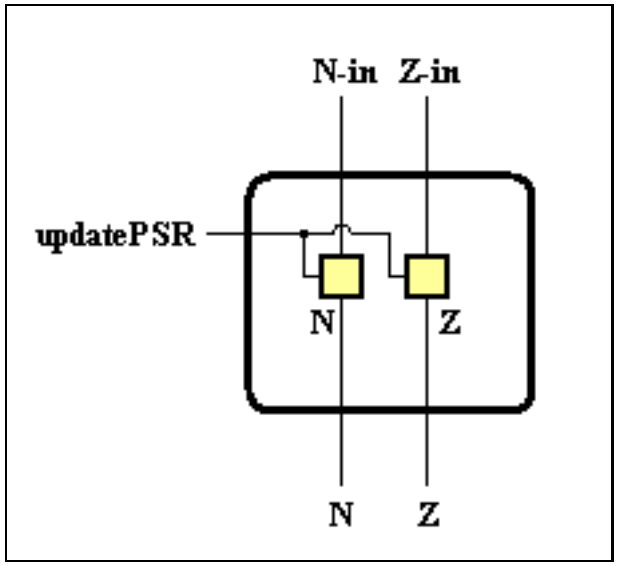

Figure 3.16: Program Status Register (PSR)

The wiring for the program status register (PSR), shown in Figure 3.16, is very straightforward. It receives three inputs from the EX circuit: `N-in`, `Z-in`, and `updatePSR`. The PSR circuit has two bits of memory inside of it–one bit for the N flag and one bit for the Z flag. Values of the `N-in` and `Z-in` inputs are only written to the PSR when the `updatePSR` signal changes from low to high (i.e. during the first half of the clock cycle). If the EX stage is stalled, however, the N and Z flags will not be updated. The two outputs of the PSR (the stored N and Z flags) will be used by the branch decision circuit (Section 3.6.1) to determine if a branch condition has been met.

3.6 Memory (MEM) Stage

The Memory (MEM) stage in the Basic Pipeline executes two types of instructions: memory-access (LD/ST) instructions and branch instructions. Figure 3.17 and Table 3.8 show the circuit diagram and the inputs and outputs of the MEM circuit, respectively. In the case of memory-access instructions, the EX stage has previously calculated the memory address and placed it in the DMAR. Additionally, the SMDR contains the source data needed for a ST instruction. With these values available, the MEM stage has all the information it needs to read from or write to memory. The decoder

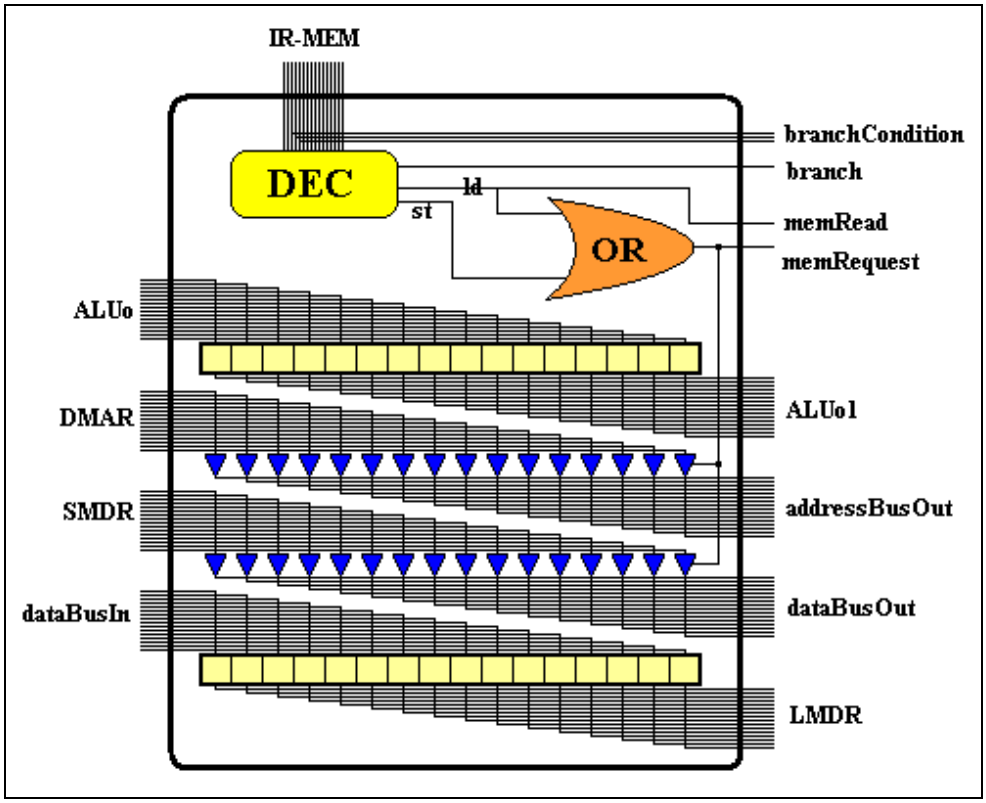

Figure 3.17: Circuit Diagram of the MEM stage

within the MEM circuit will output whether the instruction is a LD or ST instruction based on the instruction encoding. If it is either of the two, the `memRequest` signal is set during the rising edge of the clock cycle to indicate that a memory operation will be performed. The `memRead` signal is set for LD instructions and reset for ST instructions, so that the memory device

Inputs	Outputs
reset*	memRequest
clock*	memRead
stall-MEM*	addressBusOut
IR-MEM	dataBusOut
dataBusIn	LMDR
ALUo	ALUo1
DMAR	addressBusOut
SMDR	dataBusOut
	branch
	branchCondition

** Not pictured in diagram*

Table 3.8: MEM Stage Inputs and Outputs

can distinguish between retrieving data in the case of a LD instruction and storing data in the case of a ST instruction. The `memRequest` signal also serves as the control signal for the tri-state buffers that allow the DMAR to be written to the address bus. The tri-state buffers that allow the SMDR to be written to the data bus are only activated when both the `memRequest` signal is set and the `memRead` signal is cleared. Since the `memRequest` signal, which is activated during the rising edge of the clock, controls the output to both the address and data buses, memory reads and writes only occur during the clock's rising edge. In the case of the LD instruction, the value being read from memory is returned on the data bus and then stored in the **load**

memory data register (LMDR) during the clock's falling edge (at the end of the CPU cycle).

As branch instructions advance through the pipeline, the effective branch address is computed in the EX stage and stored in the ALUo register. Thus, the earliest stage in which a branch can be made is the MEM stage. While the connection is not shown in the figure, the ALUo signal is wired to the `branchAddress` input of the IF stage. Figure 3.17 does show that the MEM circuit outputs the three `branchCondition` (`BBB`) bits and another signal that indicates whether the instruction is a branch instruction. These two outputs are sent to the branch decision circuit described in Section 3.6.1.

3.6.1 Branch Decision Circuit

The branch decision circuit, shown in Figure 3.18, is responsible for determining if the branch conditions for a branch instruction have been met. The conditional branch instruction will use the values of the N and Z flags set by the previous instruction. From the current value of these two flags, the logical relationship between the two compare operands can be evaluated. For example, if the N flag is set, then the first compare operand is *less than* the second operand. If the Z flag is set, then the two compare operands

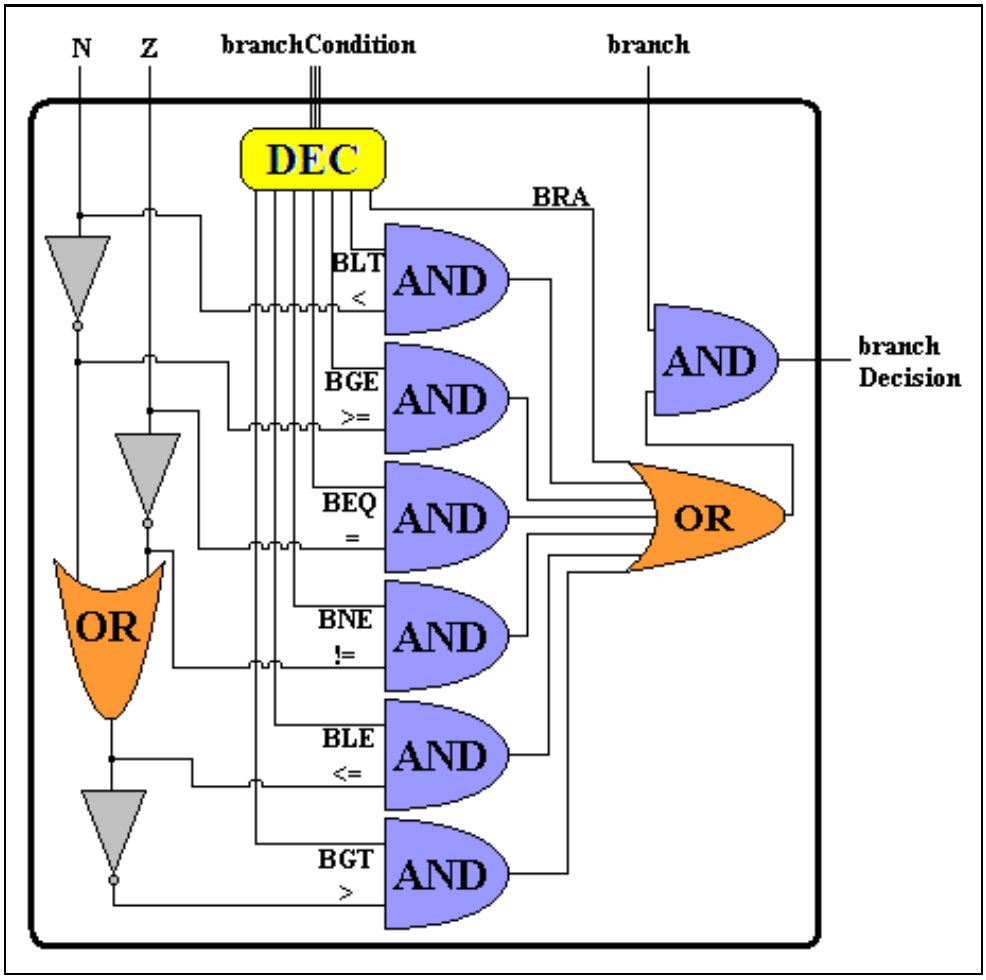

Figure 3.18: Branch Decision Circuit

are *equal to* each other. The other relationships can be computed using the circuitry on the left side of the figure.

The branch decision circuit also takes the three `branchCondition` bits of the branch instruction as an input. This signal is sent through a decoder to output the specific branch condition that was encoded into the instruction. For each possible branch condition, the circuit uses an AND gate to determine if that condition has been met. An OR gate collects all outputs to determine if any one of the branch conditions have been met. The branch always (BRA) instruction will always generate a positive output, while non-branching instructions always generate a negative result. This `branchDecision` output is sent to the IF stage of the pipeline so that the appropriate address for the next instruction can be selected.

3.7 Write Back (WR) Stage

The final stage of the pipeline is the Write Back (WR) stage, which updates the register bank for the LD and ALU instructions. Figure 3.19 shows the circuit diagram, and Table 3.9 lists the inputs and outputs of the WR stage. To detect the instruction type, the instruction register is sent through a decoder. If it is either a LD or ALU instruction, the WR

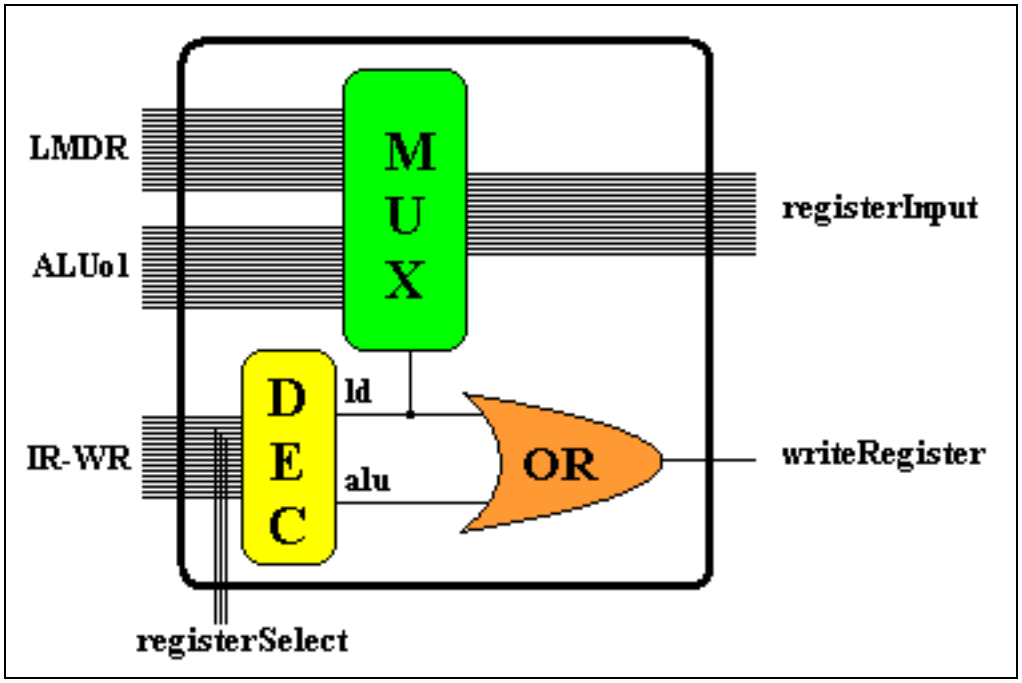

Figure 3.19: Circuit Diagram of the WR stage

circuit outputs the `writeRegister` signal. Bits 11 through 13, which encode the destination register, are used as output in the `registerSelect` signal. Lastly, the circuit must output the actual value to be written to the register. For a LD instruction, this value is found in the LMDR, and for an ALU instruction, the value is stored in the ALUo1 register. A multiplexor is used to select the correct input and output its value as `registerInput`. These three outputs are sent to the register bank, which is external to any of the

Inputs	Outputs
IR-WR	writeRegister
ALUo1	registerSelect
LMDR	registerInput

** Not pictured in diagram*

Table 3.9: WR Stage Inputs and Outputs

pipeline stages and not depicted in the figures. When the `writeRegister` signal is set, the register bank will update the selected register with the new input in the *middle* of the clock cycle. Performing the update in the middle of the clock cycle allows the instruction in the ID stage to obtain the updated register values at the end of the clock cycle.

3.8 Stalling

In the Basic Pipelined CPU, the stalling mechanism serves an important purpose: to prevent multiple CPU stages from using the address and data buses simultaneously. The two stages that can access memory are the IF stage and the MEM stage. The IF stage makes a memory request in every clock cycle, while the MEM stage only accesses memory during the execution of LD and ST instructions. When a memory-access instruction is in the MEM

stage, the conflict of deciding which of the two stages should access memory must be resolved, and the question of what to do with the other stage must be answered. Since the MEM stage is further along in the pipeline, we have to give the MEM stage priority at accessing the address and data buses. In order to prevent the IF stage from accessing memory in this scenario, a stall signal will be generated for the IF stage whenever the instruction register of the MEM stage contains a LD or ST instruction. Not only will the stall signal prevent the IF stage from accessing memory, but it will also stop the registers within the IF stage from being updated. In addition, a **NOP instruction**, which is just a "filler" instruction that has no effect on the CPU, is placed in the IR of the ID stage.

While the IF stage is the only stage that needs stalling in the Basic Pipelined CPU, in the Advanced Pipeline, which will be introduced in the next chapter, it is necessary to stall other stages in certain situations. The only stage to never require stalling is the WR stage, because all information and hardware required to execute an instruction are available without conflict. Notice that whenever a stage is stalled, all of the prior stages must also be stalled. Therefore, in addition to the IF stage stalling when the MEM stage contains a memory-access instruction, it will also receive the stall sig-

nal if the ID stage is stalled. Likewise, the ID stage will stall any time the EX stage is stalled, and the EX stage will stall whenever the MEM stage is stalled. In our version of the pipelined CPU, the MEM stage never receives the stall signal. However, we have still implemented the Stall_MEM circuit so that future extensions of the pipeline can easily modify this behavior.

3.9 The Basic Pipelined CPU in Logic-Sim

When designing a program in Logic-Sim, sets of logic gates can be grouped together in **macros**. For example, all of the logic gates associated with the IF stage of the pipeline could be defined together in a macro named "IF-Stage." Using macros has some distinct advantages. First of all, macros can be used multiple times throughout the program without having to redefine the elements of the macro each time. Secondly, the structure of a macro allows it to be easily tested and debugged. By dividing the entire CPU program into different macros, one for each pipeline stage plus a few others, we were able to test each component of the CPU individually to ensure that it worked. After combining all the macros to form the complete CPU, we tested the entire processor by running it with all the different types of instructions.

The graphical user interface (GUI) of the Basic Pipelined CPU is shown in

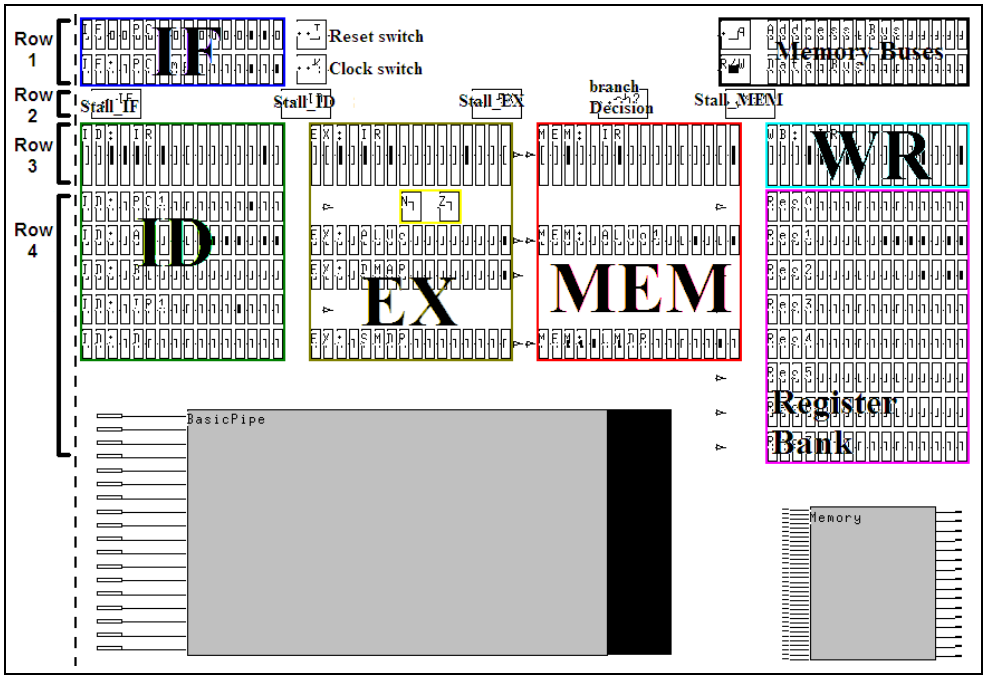

Figure 3.20: The Basic Pipelined CPU in Logic-Sim

Figure 3.20. The only two inputs for the entire program are the `reset` switch and the `clock` switch which are shown in row 1. The user can toggle these switches from the keyboard by pressing the `0` key and `1` key, respectively.

Everything else shown in the GUI is an output. In Figure 3.20, the first set of outputs shown in row 1 is the IF stage of the pipeline, which consists of the PC and PC-tmp registers. The last set of outputs in row 1 are the values on the system bus that connect the CPU and the main memory: the

`memRequest` and `memRead` signals, as well as the values on the address and data buses.

Row 2 contains five, single-bit outputs. Four of them are the stall signals for the IF, ID, EX, and MEM stages. We placed each of these probes between the instruction register of the stage it controls and the IR of the stage after it. We did this to make it clear that when the stall signal is set, the instruction in the first IR will not advance to the second one. The fifth output is the `branchDecision` signal, which we placed directly above the MEM stage because branches are executed during the MEM stage of the pipeline.

The outputs of the components in each stage are placed in rows 3 and 4 of the GUI. Row 3 contains the instruction register for each of the four pipeline stages that have an IR. We intentionally placed these registers side-by-side, so that the user can see the instructions progress through the pipeline, from the IR in one stage to the IR in the next. In row 4, directly underneath each instruction register, are the other registers that make up that given stage of the pipeline. The only exception is the WR stage since this stage does not contain any registers. We have placed the eight CPU registers beneath the IR of the WR stage. This choice seemed logical since these eight data registers are updated during the WR stage.

Chapter 4

The Advanced Pipelined CPU

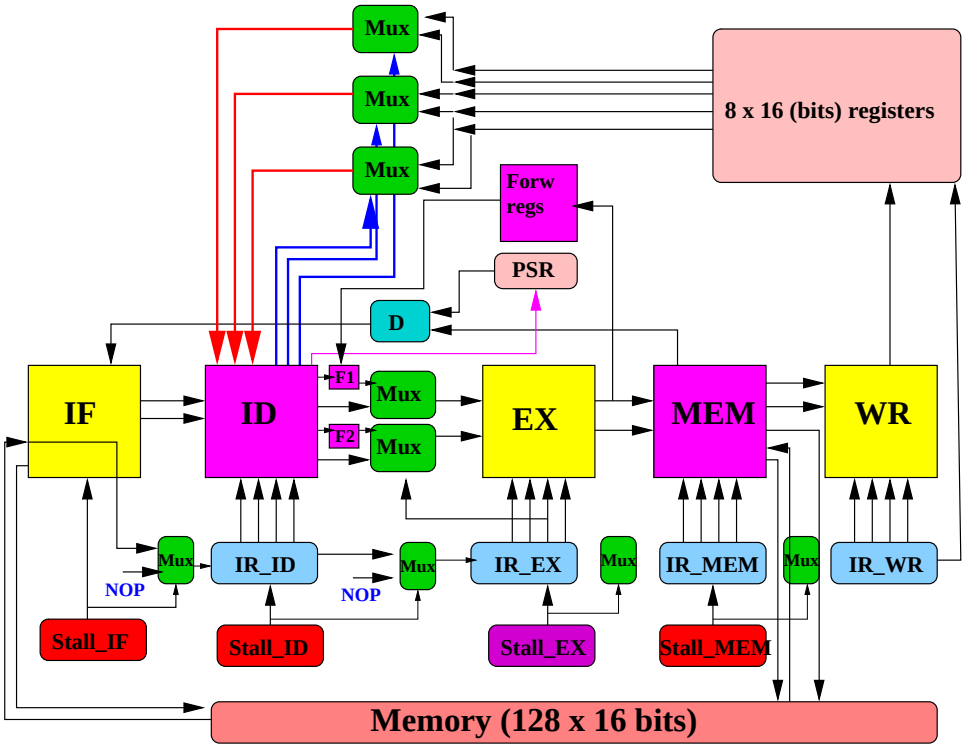

Figure 4.1: Design of the Advanced Pipelined CPU

4.1 Overview

In a non-pipelined CPU, each instruction is fully processed before the CPU fetches the next instruction. However, in a pipeline design, where multiple instructions can be executed simultaneously, the CPU will begin processing a new instruction before the ones before it have completed. This type of design can lead to various problems. For example, if one instruction writes a value to a register and the next instruction needs to use that value, then the second instruction will use the old value in the register before the first instruction can reach the WR stage and update the register. This specific example is a type of "read after write" data hazard. Other types of hazards also exist. Our Basic Pipelined CPU that we introduced in the previous chapter does not attempt to fix any of these problems. Thus, if a program written for the Basic Pipelined CPU does not account for these potential hazards, the CPU will produce incorrect results.

To address some of the drawbacks of the Basic Pipeline, the CS355 course also teaches the Advanced Pipelined CPU. We have also implemented this design in Logic-Sim. In this version, we address three different types of hazards that can occur within the Basic Pipeline. In order to correct these

issues, some of the circuits from the Basic Pipelined CPU need to be modified, and some new circuits must be added. Figure 4.1 shows the design of the Advanced Pipelined CPU. The ID, MEM, and Stall_EX circuits have been altered, and some forwarding registers and multiplexors have been added in the new design. This chapter will discuss the three different hazards, as well as the circuitry required to correct them.

4.2 Read after Write Data Hazard in ALU Instructions

As mentioned in Section 4.1, problems can arise in the Basic Pipelined CPU when the destination register of one instruction is used as the source register of an instruction following it. Problems such as this arise because instructions fetch their operands in the ID stage, but the new value is not yet available. This section deals with the case when an ALU instruction that updates a register is followed by one or more instructions that attempt to access that same register. As we will show, this type of data hazard can have an undesirable effect on only two instructions that come directly after the update instruction.

Consider the series of instructions below where the registers have the

initial values shown in Table 4.1. The first instruction will write the sum of `R2` and `R3`, which is equal to "20," to register `R1`. The next three instructions will all attempt to use this value as an operand. At the end of the first clock cycle, the first instruction will have been fetched into the instruction register in the ID stage. Table 4.2 shows the values of the pipeline registers after the first clock cycle.

- *(instr 1)* ADD R2, R3, R1
- *(instr 2)* ADD R4, R1, R4
- *(instr 3)* ADD R5, R1, R5
- *(instr 4)* ADD R6, R1, R6

During the next CPU cycle, the ID stage will fetch the operands for the first instruction. Since this is an ALU instruction that uses two register values as operands, only registers A and B in the ID stage will contain values that are relevant to the instruction. At the end of the cycle (Table 4.3),

Register Bank			
R0	0	R4	1
R1	123	R5	8
R2	11	R6	2
R3	9	R7	0

Table 4.1: Initial Register Values

ID		EX	MEM	WR	Regs	
IR	*(instr 1)*	IR	IR	IR	R0	0
PC1					R1	123
A		ALUo	ALUo1		R2	11
B		DMAR			R3	9
IR1					R4	1
D		SMDR	LMDR		R5	8
					R6	2
					R7	0

Table 4.2: Pipeline After 1 CPU Cycle

ID		EX		MEM	WR	Regs	
IR	*(instr 2)*	IR	*(instr 1)*	IR	IR	R0	0
PC1						R1	123
A	11	ALUo		ALUo1		R2	11
B	9	DMAR				R3	9
IR1						R4	1
D		SMDR		LMDR		R5	8
						R6	2
						R7	0

Table 4.3: Pipeline After 2 CPU Cycles

the first instruction will move into the IR in the EX stage, and the second instruction will have been fetched into the IR in the ID stage.

In the third clock cycle, the EX stage adds the values in registers A and B (11 + 9) and stores the result (20) in the ALUo register. This is the value that will be written to `R1` when the first instruction reaches the WR stage. However, since `R1` has not yet been updated with the value "20," the ID stage of the pipeline, which is working on the second instruction, will fetch

ID		EX		MEM		WR	Regs	
IR	*(instr 3)*	IR	*(instr 2)*	IR	*(instr 1)*	IR	R0	0
PC1							R1	123
A	1	ALUo	20	ALUo1			R2	11
B	**123**	DMAR	20				R3	9
IR1							R4	1
D		SMDR		LMDR			R5	8
							R6	2
							R7	0

Table 4.4: Pipeline After 3 CPU Cycles

the old value of `R1` (123) into register B. At the end of the cycle, instructions will advance to the next stage in the pipeline, and the registers will have the values in Table 4.4.

During the fourth CPU cycle, the MEM stage will simply copy the value from the ALUo register (20) into the ALUo1 register. Still, this value has not yet been written to `R1`. Thus, the ID stage will again fetch the old value "123" as an operand for the third instruction. The EX stage, which is performing the addition for the second instruction, will add the values from the A and B registers (1 + 123) and store the sum (124) in the ALUo register. Since 123 was not the intended operand for this instruction, an incorrect result has been calculated and will eventually be written in `R4`. Table 4.5 depicts the CPU at the end of the fourth cycle, after all instructions have advanced into

ID		EX		MEM		WR		Regs	
IR	*(instr 4)*	IR	*(instr 3)*	IR	*(instr 2)*	IR	*(instr 1)*	R0	0
PC1								R1	123
A	8	ALUo	**124**	ALUo1	20			R2	11
B	**123**	DMAR	124					R3	9
IR1								R4	1
D		SMDR		LMDR				R5	8
								R6	2
								R7	0

Table 4.5: Pipeline After 4 CPU Cycles

the instruction register of the next stage.

Recall from Section 3.7 that the WR stage updates the register bank in the *middle* of the cycle. As we will show here, this design will allow "instr 4," currently in the ID stage, to fetch the correct (updated) value of `R1` into the B register. Table 4.6 shows the pipeline during the middle of the fifth cycle, after `R1` has been updated with the value "20" from the ALUo1 register. Then, during the second half of the cycle, the ID stage can fetch the correct operands for "instr 4." However, when the EX stage calculates the result for "instr 3," it is using the incorrect operand, and thus, it will store the incorrect result (131) in the ALUo register. Table 4.7 shows the pipeline at the end of the fifth cycle.

At the end of the sixth cycle, the value "124" will be written to `R4`, and at

ID		EX		MEM		WR		Regs	
IR	*(instr 4)*	IR	*(instr 3)*	IR	*(instr 2)*	IR	*(instr 1)*	R0	0
PC1								R1	**20**
A	8	ALUo	124	ALUo1	20			R2	11
B	123	DMAR	124					R3	9
IR1								R4	1
D		SMDR		LMDR				R5	8
								R6	2
								R7	0

Table 4.6: Pipeline in the Middle of the 5th Cycle

ID		EX		MEM		WR		Regs	
IR		IR	*(instr 4)*	IR	*(instr 3)*	IR	*(instr 2)*	R0	0
PC1								R1	20
A	2	ALUo	**131**	ALUo1	124			R2	11
B	**20**	DMAR	131					R3	9
IR1								R4	1
D		SMDR		LMDR				R5	8
								R6	2
								R7	0

Table 4.7: Pipeline After 5 CPU Cycles

the end of the seventh cycle, `R5` will contain the value "131." Both of these values are wrong since they were calculated using the old value of `R1`, "123," instead of the correct value "20." When the eighth cycle finishes, `R6` will contain the value "22," which was the expected result. Thus, we can see that in the read after write data hazard for ALU instructions, two instructions following the update instruction will not be executed correctly. This fact will be important in the design of the solution, which we present in Section 4.2.1.

4.2.1 Solving the Data Hazard in ALU Instructions

Since the read after write data hazard for ALU instructions affects *two* instructions, we must add *two*, special-purpose **forwarding registers** to the pipeline. These forwarding registers, which we have named **FR1** and **FR2**, will store the two most recent outputs of the ALU in the EX stage. These registers will make their values available to other instructions during the gap in between the moment when the value is calculated and the moment when the value is stored to the destination register.

Associated with each forwarding register is a four-bit **tag register**. Three of the bits will be used to store the register number of the destination register that will be updated once the instruction that produced the result reaches the WR stage. The last bit indicates whether or not the tag is valid. Tag registers are only valid for ALU instructions that write to a register other than `R0` (since `R0` is read-only memory).

In order for the CPU to use the values from the forwarding registers, we modified the circuits that select the two operands for the ALU (described in Section 3.5.1). In the Advanced Pipeline, these multiplexors will first check tag registers **tag1** and **tag2** to see if they are valid and represent either of

the source registers for the instruction. If the tag register is valid and its register number matches one of the source register numbers, then the value from the associated forwarding register will be selected as an input for the ALU. Otherwise, one of the five registers in the ID stage will be selected as it did in the Basic Pipelined CPU.

4.3 Read after Write Data Hazard in LD Instructions

A similar data hazard occurs after a LD instruction. In the execution of a LD instruction, a value is first retrieved from memory and stored in the LMDR during the MEM stage of the pipeline. Meanwhile, the destination register of the LD instruction has not yet been updated. If either of the two instructions that follow the LD instruction attempt to use the value that was read from memory, the read after write data hazard will occur. Even though this data hazard, like the hazard after an ALU instruction, causes two instructions to execute incorrectly, we will show that the LD data hazard is more severe and cannot be entirely fixed with the forwarding register technique.

Consider the program below where the registers have the initial values

shown in Table 4.8 and the content of memory address 20 is the value "4000." The first instruction will load this value from memory into register `R1`, and then the next three instructions will all attempt to use this value as an operand. At the end of the first clock cycle, the first instruction will have been fetched into the instruction register in the ID stage, and the pipeline registers will have the values in Table 4.9.

- *(instr 1)* LD [R2 + R3], R1
- *(instr 2)* ADD R4, R1, R4
- *(instr 3)* ADD R5, R1, R5
- *(instr 4)* ADD R6, R1, R6

In the second CPU cycle, the values from registers `R2` and `R3` will be fetched into the A and B registers in the ID stage. At the end of the cycle, the first instruction will move to the EX stage and the second instruction will be fetched into the ID stage, as shown in Table 4.10.

Register Bank			
R0	0	R4	1
R1	123	R5	8
R2	11	R6	2
R3	9	R7	0

Table 4.8: Initial Register Values

ID		EX	MEM	WR	Regs	
IR	*(instr 1)*	IR	IR	IR	R0	0
PC1					R1	123
A		ALUo	ALUo1		R2	11
B		DMAR			R3	9
IR1					R4	1
D		SMDR	LMDR		R5	8
					R6	2
					R7	0

Table 4.9: Pipeline After 1 CPU Cycle

ID		EX		MEM	WR	Regs	
IR	*(instr 2)*	IR	*(instr 1)*	IR	IR	R0	0
PC1						R1	123
A	11	ALUo		ALUo1		R2	11
B	9	DMAR				R3	9
IR1						R4	1
D		SMDR		LMDR		R5	8
						R6	2
						R7	0

Table 4.10: Pipeline After 2 CPU Cycles

In the third cycle, the EX stage will add the values from the A and B registers (11 + 9) and store the result (20) in both the ALUo and DMAR registers. Additionally, the ID stage, which is working in the second instruction, will fetch the current values of `R4` and `R1` into the A and B registers, respectively. However, register `R1` has not yet been updated with the value "4000" from memory, and therefore, the incorrect operand will be fetched into the B register. When the cycle ends, the instructions will all advance to

ID		EX		MEM		WR	Regs	
IR	*(instr 3)*	IR	*(instr 2)*	IR	*(instr 1)*	IR	R0	0
PC1							R1	123
A	1	ALUo	20	ALUo1			R2	11
B	**123**	DMAR	20				R3	9
IR1							R4	1
D		SMDR		LMDR			R5	8
							R6	2
							R7	0

Table 4.11: Pipeline After 3 CPU Cycles

the next stage, and the third instruction will be fetched into the IR in the ID stage. The state of the CPU after the third cycle is shown in Table 4.11.

Recall that the stalling mechanism of the Basic Pipeline, described in Section 3.8, was designed to prevent multiple stages from accessing memory simultaneously. When the fourth CPU cycle finishes, since the MEM stage contains a LD instruction, the IF stage will be stalled, and a NOP instruction will be placed in the IR of the ID stage. During this cycle, the MEM stage will make the memory request and fetch the value from memory (4000) into the LMDR. The EX stage will add the values from the A and B register (1 + 123) and store the result in the ALUo register. This is an incorrect value because the program should use the value "4000" for the second operand and not "123." The ID stage will fetch the current values of `R5` and `R1` into the A

ID		EX		MEM		WR		Regs	
IR	NOP	IR	*(instr 3)*	IR	*(instr 2)*	IR	*(instr 1)*	R0	0
PC1								R1	123
A	8	ALUo	**124**	ALUo1	20			R2	11
B	**123**	DMAR	124					R3	9
IR1								R4	1
D		SMDR		LMDR	4000			R5	8
								R6	2
								R7	0

Table 4.12: Pipeline After 4 CPU Cycles

and B registers, respectively. Again, an incorrect value for `R1` will be fetched. Table 4.12 shows the pipeline registers after the fourth cycle has completed.

Now that the memory operation has completed, the new value for `R1` is available for the first time. As we have done in the ALU data hazard, we could implement a forwarding technique that would allow the value in the LMDR to be selected as one of the operands for the ALU. However, if you look at Table 4.12 carefully, you will see that the second instruction (instr 2) has already passed through the ALU in the EX stage and been placed in the MEM stage. In other words, it is too late to repeat this instruction. Thus, the read after write hazard is more severe for a LD instruction than it is for an ALU instruction. Forwarding would not be able to correct the outcome of "instr 2," and as such, this data hazard has a different solution, which we

will present in Section 4.3.1.

4.3.1 Solving the Data Hazard in LD Instructions

The main problem in a LD instruction is the fact that the data is fetched from memory. The operand is simply not available within the CPU, and instruction "instr 2" cannot proceed. For this reason, the logical solution to the problem would be to prevent "instr 2" from proceeding, which can be accomplished with stalling. We must stall the EX stage whenever the following are true:

- The MEM stage contains a LD instruction
- The EX stage contains an ALU, LD, or ST instruction
- One of the source registers of the instruction in the EX stage is the destination register of the LD instruction in the MEM stage

If we look at the example in the previous section and specifically at Table 4.11, which is a snapshot of the CPU after the third cycle, then we can see how stalling the EX stage will solve part of the data hazard problem. At that point in time, the MEM stage contains the LD instruction with `R1` as the destination register, and the EX stage contains an ALU instruction that uses `R1` as one of the source registers. All three conditions for stalling the EX

ID		EX		MEM		WR		Regs	
IR	*(instr 3)*	IR	*(instr 2)*	IR	NOP	IR	*(instr 1)*	R0	0
PC1								R1	123
A	1	ALUo	20	ALUo1				R2	11
B	**123**	DMAR	20					R3	9
IR1								R4	1
D		SMDR		LMDR	4000			R5	8
								R6	2
								R7	0

Table 4.13: Pipeline After 4 CPU Cycles (with stall)

stage are met. Therefore, with the new stalling mechanism in place, Table 4.13 shows the state of the CPU after the fourth CPU cycle.

Notice in Table 4.13 that the instruction "instr 2" is still in the EX stage. Therefore, this instruction can be executed with the correct value by forwarding the value of the LMDR to the input of the ALU. As with the solution for the ALU data hazard, we modify the circuits that select the operands for the ALU. Now, before checking the tag1 and tag2 registers, the multiplexors will first check if the instruction in the WR stage is a LD instruction with a destination register other than `R0`. If that is the case, and the destination register of the LD instruction is one of the source registers for the instruction in the EX stage, then the value in the LMDR will be selected as an input for the ALU. Otherwise, the circuits will function exactly as they

did before.

4.4 Control Hazard in Branch Instructions

Unlike a data hazard, a control hazard does not execute instructions using invalid data. Instead, a control hazard is an unusual behavior in the timing of the execution of a branch instruction. In the Basic Pipelined CPU, the branch decision is made when the branch instruction is in the MEM stage. As a result, the branch has a delay of three instructions. In other words, three instructions following the branch instruction in memory will enter the pipeline before the processor executes the branch. If the branch is taken, then logically those three instructions should not be executed, but there is no mechanism to remove instructions from the pipeline. In the Basic Pipelined CPU, the only safe way around this control hazard is to insert three NOP instructions after every branch instruction. However, these useless instructions reduce the efficiency of a pipelined CPU. A delay of three clock cycles is considered unacceptable, so additional hardware is included in the pipeline to reduce the branch delay to one clock cycle. This improvement is explained in Section 4.4.1.

4.4.1 Reducing the Branch Delay

In order to reduce the branch delay, we must first determine which operands are needed to execute a branch instruction (both conditional and unconditional). The following values are necessary:

- Address of the branch instruction (PC1)
- Value of the Offset (IR1)
- Branch condition type
- Value of the N and Z flags

Next, we must identify the earliest time that these values are available in the CPU. The address of the branch instruction is available immediately from the PC-tmp register in the IF stage. The value of the offset and the condition type are both encoded into the branch instruction, so these values are also available once the instruction is fetched into the instruction register in the ID stage. Also when the branch instruction is in the ID stage, an ALU (compare) instruction in the EX stage can update the N and Z flags in the first half of the CPU cycle, making these values available to the ID stage during the second half of the cycle. Therefore, the earliest stage in which a branch can be executed is the ID stage.

To improve the branch delay, we add a special-purpose addition circuit in the ID stage that sums the value of the PC-tmp register and relative address from the IR register in the ID stage. In the Advanced Pipelined CPU, this sum is forwarded to the `branchAddress` input of the IF stage. Additionally, the ID stage will output the `branchCondition` bits and the `branch` signal for the branch decision circuit that computes the final branch decision signal for the IF stage. The branch decision signal is set if the branch conditions have been met and reset otherwise. Now that the branch is being executed two stages earlier, in the ID stage instead of the MEM stage, the branch delay is reduced from three clock cycles to just one. As a result, only one NOP instruction will need to be placed after branch instructions.

4.5 The Advanced Pipelined CPU in LogicSim

Constructing the Advanced Pipeline is accomplished by expanding a small number of components in the Basic Pipeline and adding some forwarding registers. Most of the program modules remain the same. We only modify the macros that require design changes (ID, MEM, Stall_EX, ALU1 and ALU2) and create new macros for the forwarding and tag registers. Furthermore,

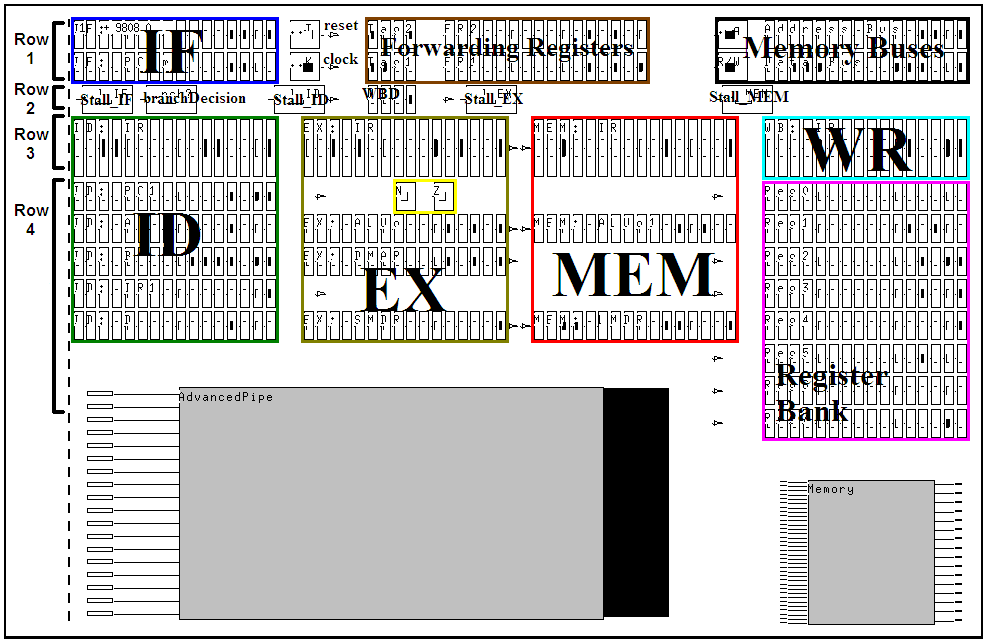

Figure 4.2: The Advanced Pipelined CPU in Logic-Sim

we only need to test the macros that were updated or added.

The GUI of the Advanced Pipelined CPU (Figure 4.2) is very similar to that of the Basic Pipeline. In row 1, we have added output probes for the two sets of forwarding and tag registers. Within the tag registers, the first bit indicates if the tag is valid, while the last three bits represent a register number. This represents the changes in the Advanced Pipeline that fix the read after write data hazard in ALU instructions.

In order to represent data forwarded from the LMDR to solve the read

after write data hazard in LD instructions, we have added a four-bit output named WBD (write back destination) in row 2. This output has the same form as a tag register, except that it uses the register number that is encoded as the destination register for the instruction in the WR stage. The WBD valid bit is only set when this instruction is a LD instruction that outputs to any register other than `R0`. Lastly, to show the reduction in the branch delay, the `branchDecision` output in row 2 has been moved above the ID stage (it was located above the MEM stage in the Logic-Sim program for the Basic Pipeline).

Chapter 5

Concluding Remarks

In this thesis, we have presented the design of two versions of a pipelined CPU. The Basic Pipeline is a simple, five-stage processor that correctly processes instructions, but it can produce undesirable results in certain situations. In the Advanced Pipeline, we identified a number of problem scenarios and implemented mechanisms to correct them. While we have fixed some of the data and control hazards, the Advanced Pipeline does not address every type of hazard that exists. A future version of the CPU could attempt to address other data hazards that exist due to the pipeline design.

As is the case in actual computer engineering, increases in speed and performance usually come at the cost of more complex circuits (and higher costs). Both of the processors that we have presented have five stages in the pipeline. Another of version of the pipelined CPU, with the same functionality as the ones in this thesis, could probably be constructed using only three

stages. However, the complexity of the additional hardware that would be needed to accomplish this is not worth making the change.

In evaluating this project, it is very important not to lose sight of the fact that the design of these two versions of the pipelined CPU are being used for instructional purposes. Moreover, the course in which this material is taught is an undergraduate course that is meant to give students only a basic understanding of computer architecture. The Basic Pipeline exists in the curriculum because it is straightforward and it demonstrates the data and control hazards. The CS355 course also introduces the Advanced Pipeline simply to demonstrate how some of these hazards can be fixed. It is by no means intended to be a perfect processor that has any commercial value. When teaching a subject, simpler is often times better. In this case, if there were fewer than five stages in the pipeline, too many things would be happening within any one stage, and it would be very difficult for students to follow.

Since Logic-Sim programs for the pipelined CPU have not existed in the past, professors had to lecture on the material using a series of figures to demonstrate how the instructions advanced through the different stages in the pipeline. A different set of figures would be needed to explain the func-

tionality of each type of instruction and to demonstrate the problems with each type of data and control hazard. With the simulated versions of the pipelined CPU, professors can easily explain all of this material in one, familiar location. By simply changing the instructions that are stored in memory, different programs can be run on the same simulator. Additionally, the Logic-Sim programs allow students to have a hands-on experience with the pipelined CPU. They can control the clock at their own pace and reset a program as many times as they would like until they have a clear understanding of how the processor works. Hopefully when our simulated versions of the Basic and Advanced Pipelines are introduced into the CS355 course, they will receive positive feedback from both professors and students, as well as constructive criticism to help us improve the design and layout of the GUI.

Bibliography

[1] Hennessy, John L. and David A. Patterson. *Computer Architecture A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.

[2] Reid, Richard J. "Interactive Digital Simulation." *SIGCSE Bulletin*, Vol. 18, No. 2 (June 1986): 58-62.

[3] Tanenbaum, Andrew S. *Structured Computer Organization*. 4th ed. Upper Saddle River, N.J.: Prentice Hall, 1999.