UNIVERSITY OF Cincinnati

Project 4: Pipelined Control Unit

Presented by:
Devin Burnside
Aaron Sens

EECE3026
Due Date: 4/4/2017

Table of Contents

I. Specifications

The basic characteristics of this machine are: a word size of 16 bits, a data bus size of 16 bits, byte addressable memory, a 16 bit program status word (PSW), 16 instructions (14 user and 2 privileged) 8 general purpose registers, 16 bit program counter, 16 bit countdown timer, and 2's complement number representation. The register file containing the GPR's has 3 read ports and 2 write ports that can all be accessed simultaneously.

The main memory unit in this machine is split up into two segments: one for instructions and one for data. Instruction reads can occur simultaneously with instruction writes. Memory operations only take one clock cycle to complete. There is also a read only memory unit (ROM) containing the constants 0,2,4,6,8,10,12, and 14 which are used to resolve exceptions.

Opcode	S	I1	I2	Rd	Rs1	Rs2
Opcode	S	Rd		Short_Offset		
Opcode	Long_Offset					

0 3 5 7 9 12 15

Table 1: Instruction Format

I1/I2	Operand
I1 = 0	OP1 = Reg[Rs1];
I1 = 1	OP1 = MM[Reg[Rs1]];
I2 = 0	OP2 = Reg[Rs2];
I2 = 1	OP2 = MM[Reg[Rs2]];

Table 2: Operand Location Determination Bits

Name	Opcode	Description
ADD	0	GPR[Rd] = OP1 + OP2
SUB	1	GPR[Rd] = OP1 - OP2
AND	2	GPR[Rd] = OP1 **and** OP2
SHL	3	GPR[Rd] = shift_left(OP1) by $OP2_{3-0}$
SHRA	4	GPR[Rd] = shift_right(OP1) by $OP2_{3-0}$
OR	5	GPR[Rd] = OP1 **or** OP2
NOT	6	GPR[Rd] = **not** MM[PC + Short_Offset]
LD	7	GPR[Rd] = MM[PC + Short_Offset]
ST	8	MM[PC + Short_Offset] = GPR[Rd]
BRN	9	if CC.N then PC = PC + Long_Offset
BRZ	10	if CC.Z then PC = PC + Long_Offset
BR	11	PC = PC + Long_Offset
JSR	12	GPR[Rd] = PC; PC = PC + Short_Offset
RTS	13	PC = GPR[Rd] + Short_Offset
CLK	14	Set timer to MM[PC + Long_Offset]
LPSW	15	PSW = MM[PC + Long_Offset]

Table 3: Instructions and their Semantics

II. High Level Design

The objective of this project is to design a pipelined control unit to implement the instruction set based on the specification from Project 3.

A high level block diagram of the machine is shown in **Figure 1.**

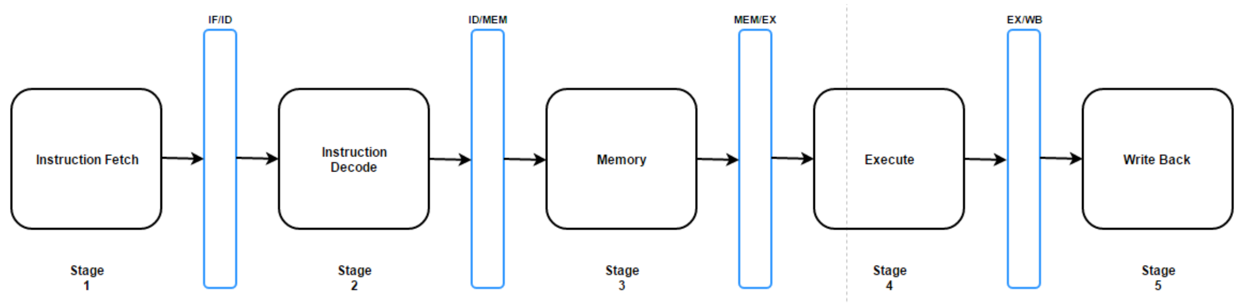


Figure 1: High Level Block Diagram of Machine

There are 5 pipeline stages in this machine: instruction fetch, instruction decode, memory, execute, and write back. Each of these stages is separated by a pipeline register. These registers store the data from the previous stage so that it can move sequentially through the pipeline, or forward the data to a later stage to prevent data hazards. The five pipeline registers are: IF/ID, EX/MEM, MEM/EX, and EX/WB.

III. Data Path Stages and Components

A complete image of the pipelined control unit is shown in the **Appendix**. The operations that occur in each stage of the pipeline are shown in this section via a stage by stage explanation. These operations are carried out by the control unit, which will be explained further in section **IV**.

Note: Control signals are indicated by dotted green circles, values from pipeline registers are indicated by blue circles, values sent/received by components within a stage are represented by black circles.

A. Instruction Fetch Stage

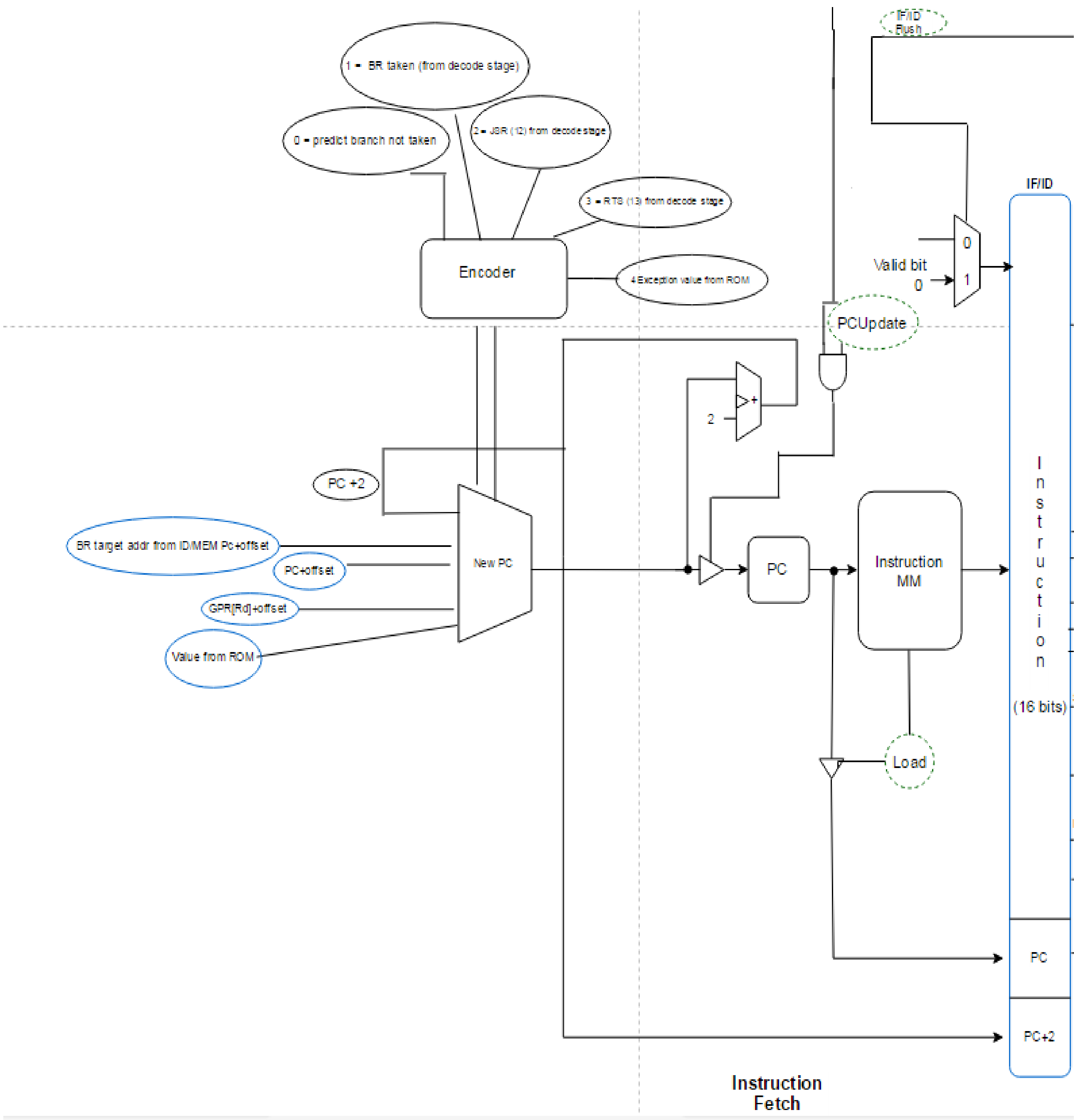

IF/ID Flush
1 = BR taken (from decode stage)
0 = predict branch not taken
2 = JSR (12) from decode stage
3 = RTS (13) from decode stage
Encoder
4 Exception value from ROM
IF/ID
Valid bit
0
0
1
PCUpdate
2
+
PC +2
BR target addr from ID/MEM Pc+offset
PC+offset
GPR[Rd]+offset
Value from ROM
New PC
PC
Instruction MM
I n s t r u c t i o n
(16 bits)
Load
PC
PC+2
Instruction Fetch

Figure 2: Instruction Fetch Stage

The fetch stage fetches a new instruction from the instruction memory based on the current PC value. This value is stored in the IF/ID pipeline register. The value of the current PC and PC+2 are also stored in the IF/ID pipeline register. The value for the next PC is decided from PC+2, a branch target address, jump subroutine address (JSR), return to subroutine (RTS) address, or ROM value based on combinational logic and the current instruction. It defaults to PC+2.

B. Instruction Decode Stage

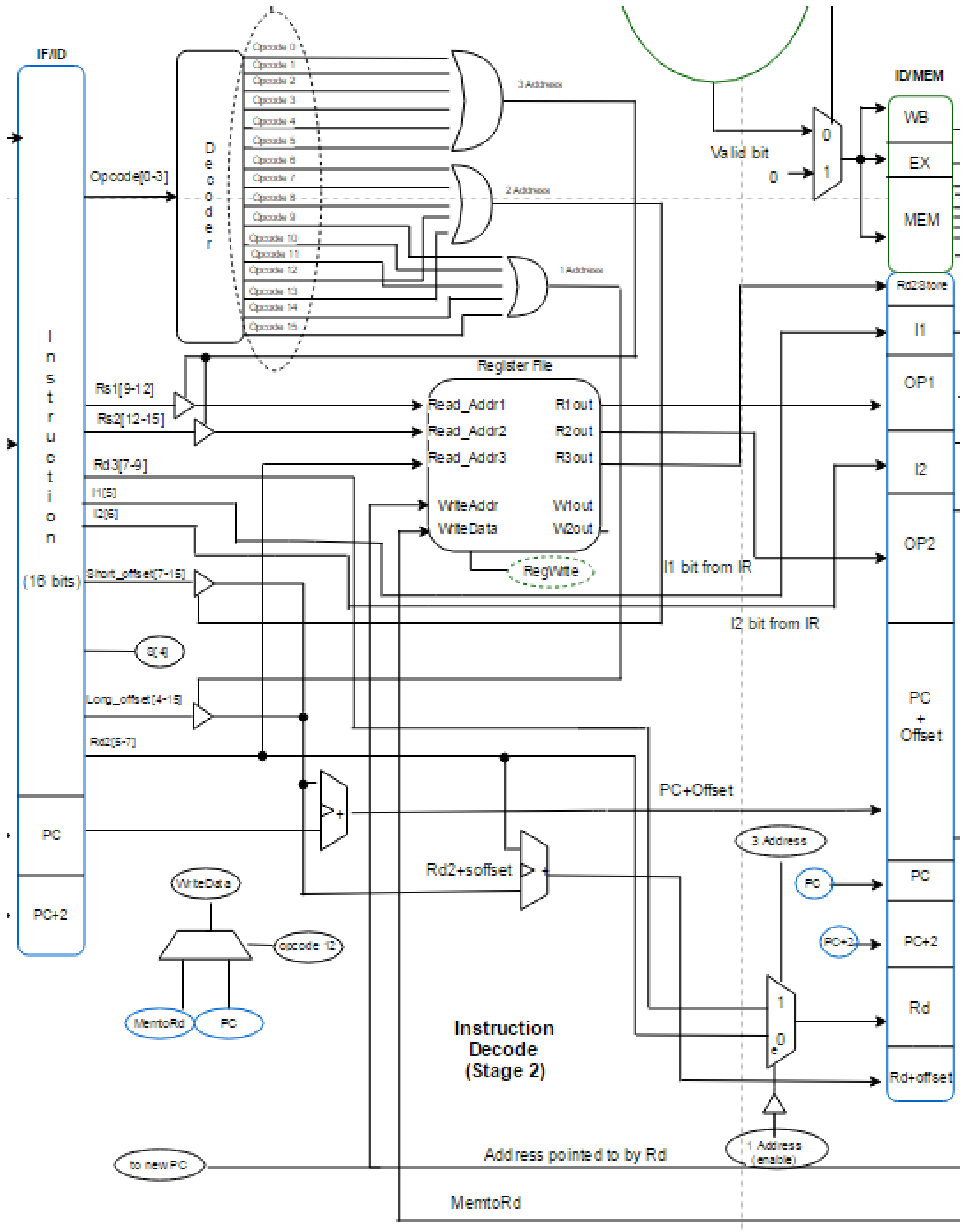


Figure 3: Instruction Decode Stage

The instruction decode stage reads the instruction bits in the IF/ID pipeline register from the fetch stage. It then decodes the instruction and sends it to the control unit to generate control signals. The instruction decode stage also calculates any PC+offset or Rd+offset address using combinational logic depending on the opcode being read, and stores it in the ID/MEM pipeline register. Addresses for operands OP1 and OP2 are also generated based on the instruction bits that are propagated to the register file. Additionally, data can be written into the register file can be written in from stages further down on the pipeline when the machine operation dictates it.

C. Memory Stage

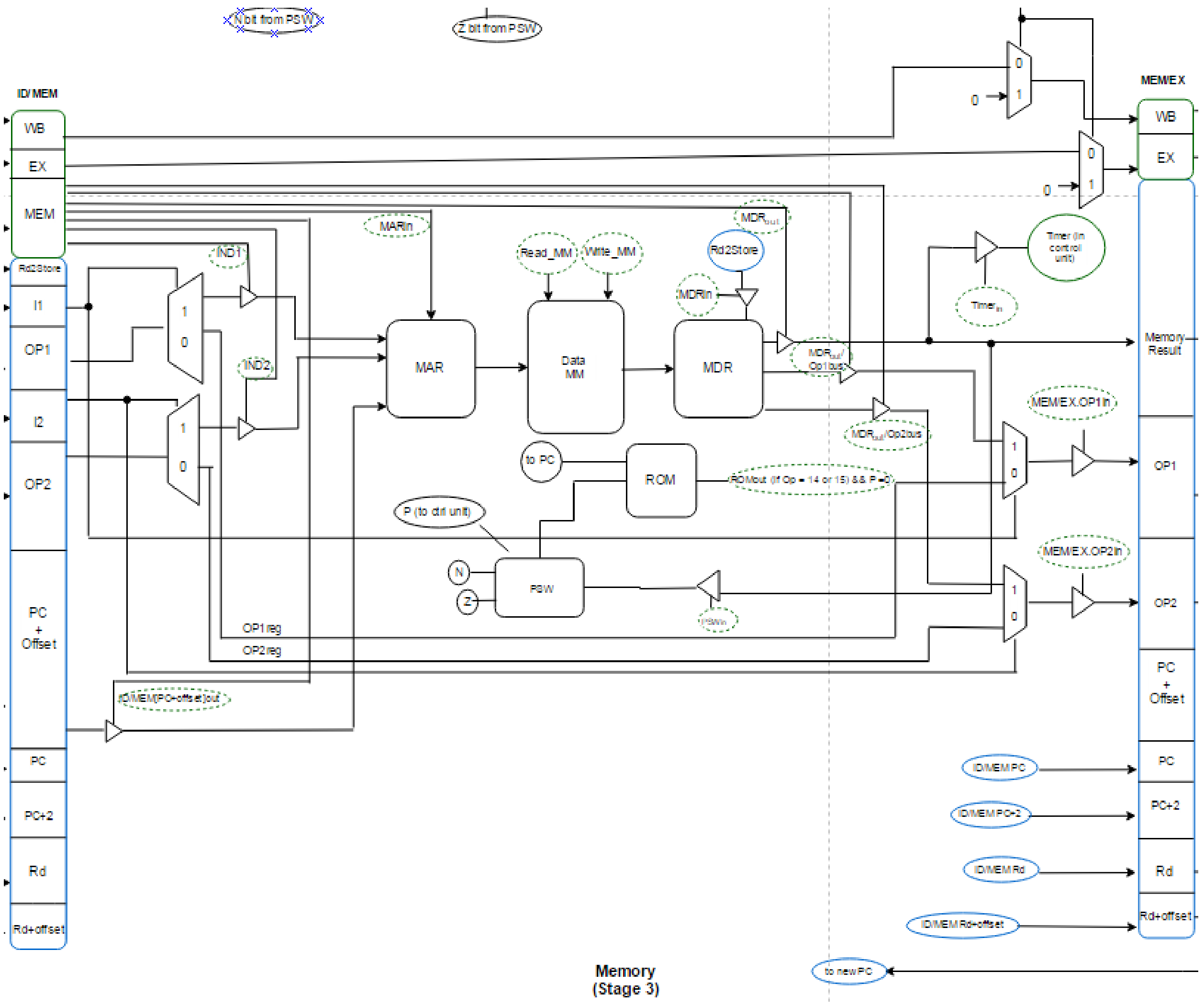


Figure 4: Memory Stage (3)

The memory stage reads or writes addresses in data main memory based on information in the ID/MEM pipeline register and then stores it in the MEM/EX pipeline register. Data main memory may not be accessed during this stage depending on the instruction. In this case the operands OP1 and OP2 are immediately stored in the MEM/EX pipeline register to be used in the execute stage. Program check violation and timeout exception operations are performed in this stage.

All of the PC, PC+offset, and Rd values stored in the ID/MEM pipeline register are automatically stored in the MEM/EX register as soon as the stage moves for data forwarding purposes.

D. Execute Stage

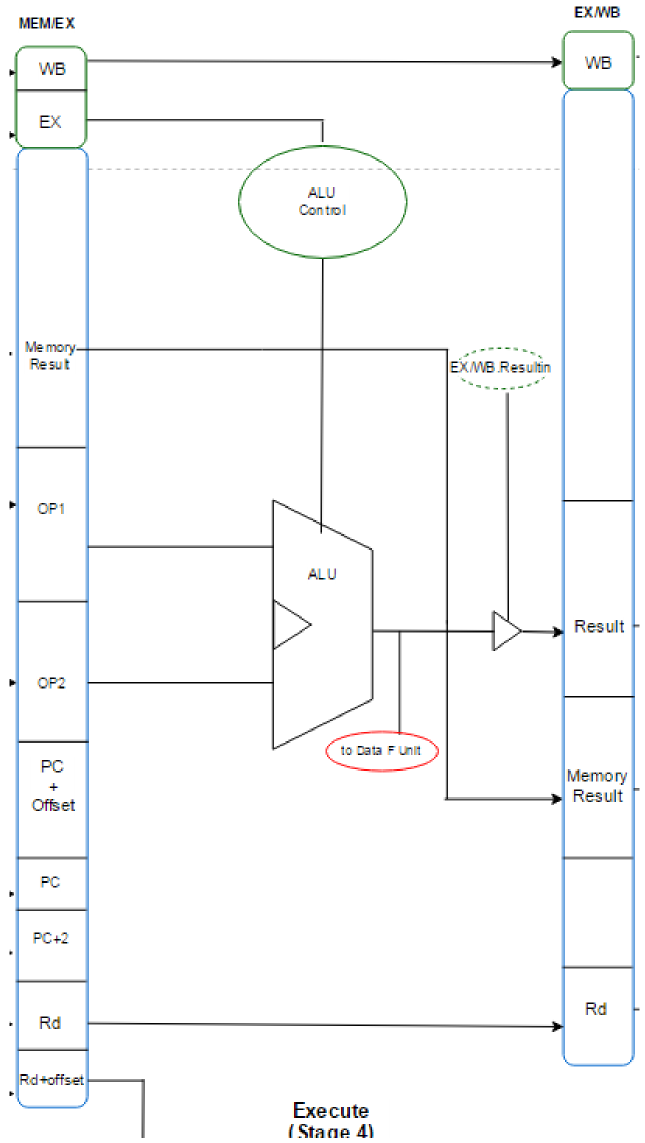


Figure 5: Execute stage (4)

The execute stage performs ALU operations on operands stored in the MEM/EX pipeline register which were generated during the memory stage. It then stores the values of these results in the EX/WB pipeline register. If the current machine operation does not require an ALU operation, the result from data main memory stored in the MEM/EX pipeline register is instead stored in the EX/WB pipeline register so it can be written back.The register destination value Rd is propagated to the EX/WB pipeline register. This allows it to be written back to the register file in the write back stage.

- ALU & ALU Control Unit

The arithmetic logic unit (ALU) located in the execute stage can perform several different operations on operands stored in the MEM/EX pipeline register. The ALU control unit decides which operation to perform based on instructions from the main control unit.

Selector			Operation
0	0	0	Do Nothing
0	0	1	NOT
0	1	0	AND
0	1	1	OR
1	0	0	Add
1	0	1	Subtract
1	1	0	Logical Shift Left
1	1	1	Arithmetic Shift Right

Table 4: Operations and Selector Bits of ALU (set by ALU control unit)

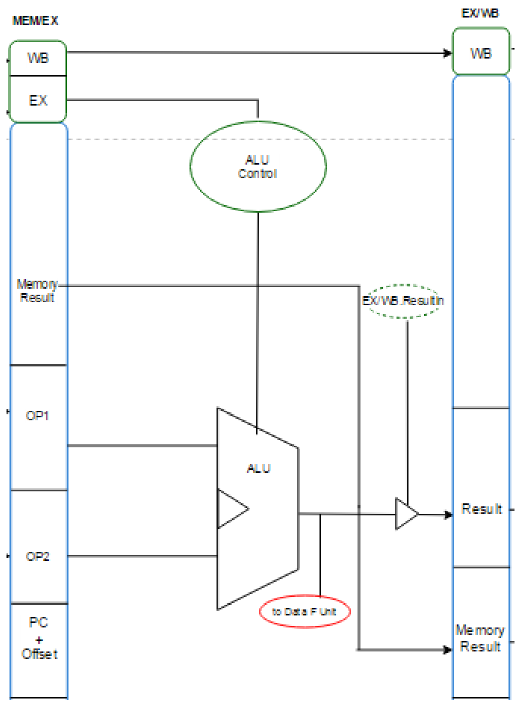


Figure 6: ALU and ALU Control Unit

E. Write Back Stage

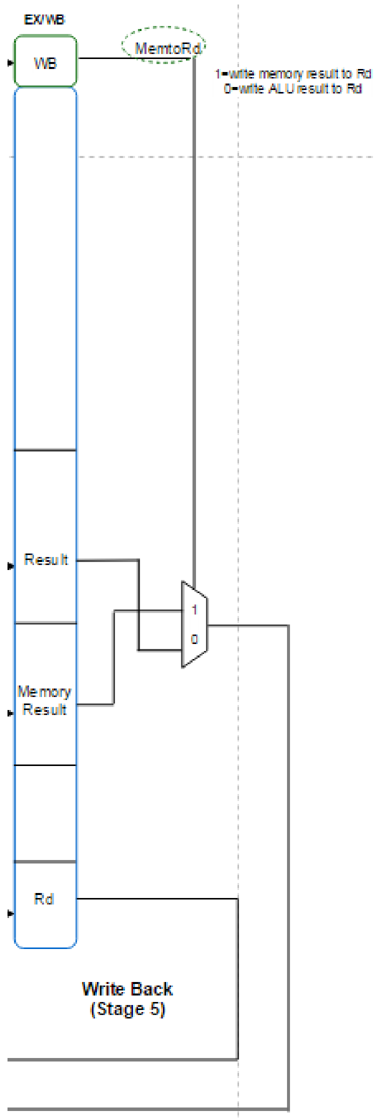


Figure 7: Write Back Stage (5)

The write back stage takes either the memory result or the ALU result stored in the EX/WB pipeline register and writes it back to the address pointed to by the Rd (in the EX/WB pipeline register) into the register file. The current instruction the control unit is executing will dictate what is written back (if any).

F. Control Hazard Detection Unit

If a branch operation is performed by the machine the control hazard detection unit stalls the pipeline until the hazard clears. It does this using combinational logic to determine if the currently executed opcode involves a branch (opcodes 9,10, or 11) and stalls the pipeline for one clock cycle until the branch is taken and a branch instruction is no longer being executed.

The control hazard detection unit also stalls the pipeline if the machine enters a trap state (exception). It will stall the machine until the exception has been resolved in the memory stage. Stalls are further explained in section **III.H**
Control hazards and exceptions are further explained in section **V.B.**

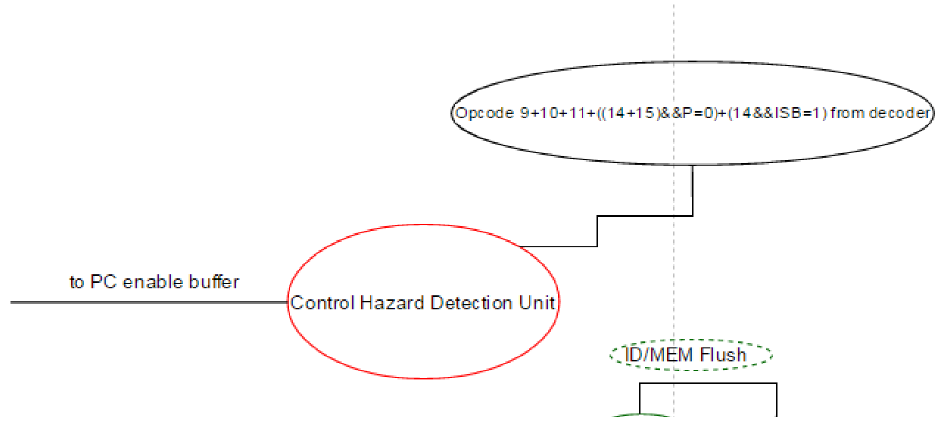

Opcode 9+10+11+((14+15)&&P=0)+(14&&ISB=1) from decoder
to PC enable buffer
Control Hazard Detection Unit
ID/MEM Flush

Figure 8: Control Hazard Detection Unit (Located in Instruction Decode stage)

G. Data Forwarding Unit

Occasionally during the operation of the machine a value will be needed by a later instruction even though it has not been stored in a pipeline register yet. The data forwarding unit uses combinational logic based on the instructions being carried out and the registers they are using and forwards the necessary Rd or ALU result values to the appropriate pipeline register. Data hazards and data forwarding are further explained in section **V.A.**

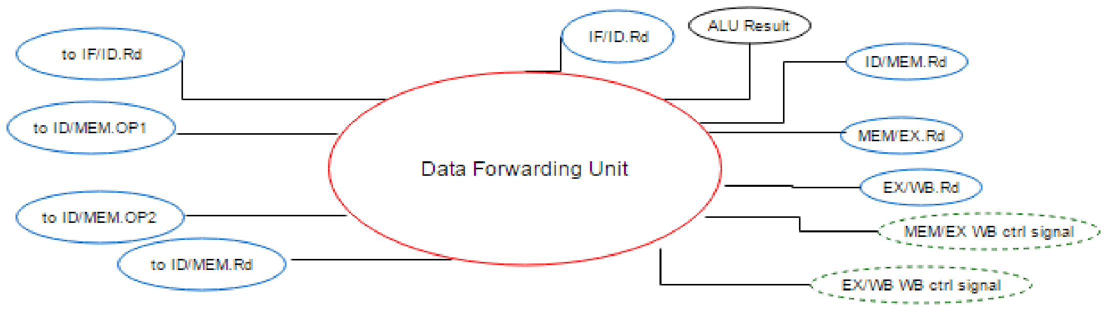


Figure 9: Data Forwarding Unit

H. Pipeline Stage Progression

The machine fetches a new instruction and inserts it at the beginning of the pipeline when all stages finish executing their current instruction. This is accomplished by the control unit (refer to **IV.E)**. There are two situations where the normal progression of the pipeline is altered: stalls and flushes. Both stalls and flushes are accomplished by using nops, which sends a 0 to every control bit. In other words, a nop is an instruction to wait and do nothing for that stage.

- Stalls

The machine stalls when the control hazard detection unit (refer to **III.F)** detects a control hazard or exception. Stalling prevents the PC from updating (and therefore fetching a new instruction) and causes nops to be issued to each of the pipeline registers until the hazard clears. For branches, the hazard clears when the branch target address is set and sent to the new PC. For exceptions, the hazard clears after memory operations are completed in the memory stage. The control hazard detection unit implements stalls as shown in **Figure 10** below.

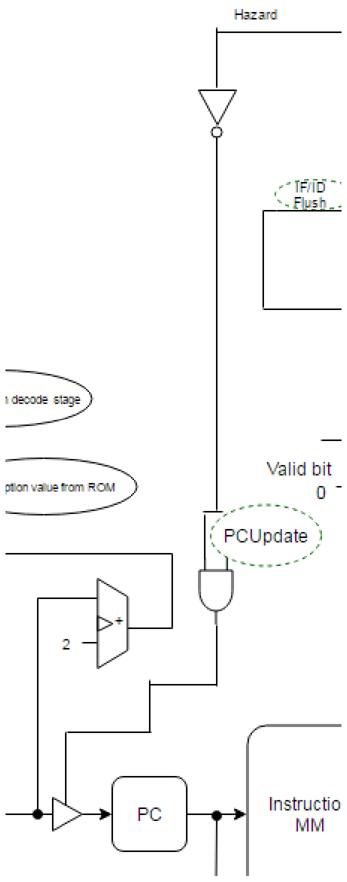


Figure 10: How the Control Hazard Detection Unit Stalls the Pipeline

If the control hazard detection unit detects a control hazard or exception, it prevents the PC from being updated as shown in **Figure 10.** In order for the PC to be updated with a new value, there must not be a hazard, and the PCUpdate control signal must be enabled. (refer to **V.B** for control signal explanations**)**. Taken branches stall the pipeline for 1 clock cycle while the PC is updated with the new address from the branch instruction. Exceptions stall the PC for 3 clock cycles until they perform their memory operations.

- **Flushes**

In addition to stalling the pipeline, the machine must also flush instructions for taken branches. This is because any instruction fetched after the branch is invalid until the branch target address is loaded into the PC. The control unit does this by setting out a flush instruction to the corresponding pipeline register, which sets the valid bit for that register to zero (refer to **IV.B** and **IV.C** for control signal explanations**).** When the valid bit is set to 0, the current instruction in the pipeline register becomes invalid, and exits the pipeline without doing anything. The pipeline register still contains the instruction information in the register, but it *acts* as a nop.

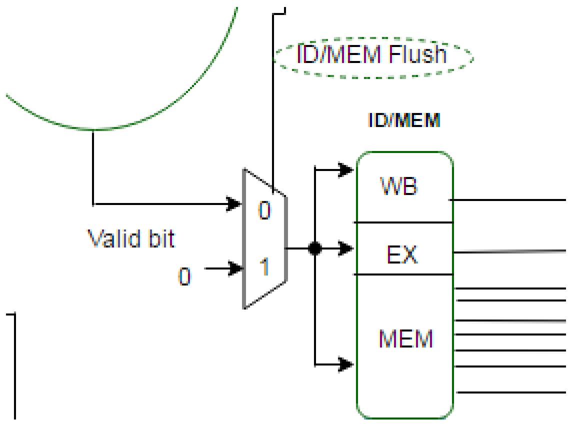


Figure 11: Valid Bit for Pipeline Flushes

V. <u>Control Unit</u>

A. Control Signal Explanation

There are a total of 23 different control signals generated by the control unit for this machine. These 23 bits create a control string which is output by the control unit. These control strings either enable or disable the ALU, pipeline registers, and other memory elements depending on their value. (0 = disable, 1 = enable). For ROM$_{out}$ functions, the address is specified by combinational logic in the control unit.

B. Control Signal List

The complete list of control signals and their functionalities are shown below. Bits 24-32 are denoted as don't cares (x) because no control signal is assigned to them.

Control Signal Name	Control Signal Bit Number (0-31) (least sig bit- most sig bit)	Control Signal Functionality
Nop	10000000000000000000000xxxxxxxxx	No operation. Signals the stage to do nothing until each of the other stages have completed. (sets all control signals to 0 for 1 clock cycle)
Load	01000000000000000000000xxxxxxxx	Loads the PC into Instruction MM (provided the hazard detection unit is not outputting a stall) while also enabling a buffer to send the PC to the IF/ID register.
PC$_{update}$	00100000000000000000000xxxxxxxxx	Loads the new PC address into the PC register.
IND1	00010000000000000000000xxxxxxxxx	Enables a buffer to send the contents of ID/MEM.OP1 into the MAR to be read into Data MM. Used for indirect addressing.
IND2	00001000000000000000000xxxxxxxxx	Enables a buffer to send the contents of ID/MEM.OP2 into the MAR to be read into Data MM. Used for indirect addressing.
ID/MEM.[PC+offset]$_{out}$	00000100000000000000000xxxxxxxxx	Enables a buffer to send the contents of ID/MEM.[PC+Offset] into the MAR to be read or written into Data MM.
MAR$_{in}$	00000010000000000000000xxxxxxxxx	Enables the MAR.
Read_MM	00000001000000000000000xxxxxxxxx	Reads the data stored in Data MM at the address pointed to by the MAR and sends it to the MDR.

Write_MM	00000000100000000000000xxxxxxxxx	Writes the data in the MDR in Data MM at the address pointed to by the MAR.
MDR_{in}	00000000010000000000000xxxxxxxxx	Enables a buffer that sends Rd2 into the MDR to be written into Data MM.
MDR_{out}	00000000001000000000000xxxxxxxxx	Enables a buffer that sends the contents in the MDR to MEM/EX.MemoryResult
MDR_{out}/OP1bus	00000000000100000000000xxxxxxxxx	Enables a buffer that sends the contents in the MDR to a multiplexer that determines whether to send it to MEM/EX.OP1 or not based on the current instruction.
MDR_{out}/OP2bus	00000000000010000000000xxxxxxxxx	Enables a buffer that sends the contents in the MDR to a multiplexer that determines whether to send it to MEM/EX.OP1 or not based on the current instruction.
IF/ID.Flush	00000000000001000000000xxxxxxxxx	Sets the valid bit to zero in the IF/ID register. This invalidates that instruction in the pipeline register until a new one is loaded in.
ID/MEM.Flush	00000000000000100000000xxxxxxxxx	Sets the valid bit to zero in the ID/MEM register. This invalidates that instruction in the pipeline register until a new one is loaded in.
MEM/EX.Flush	00000000000000010000000xxxxxxxxx	Sets the valid bit to zero in the MEM/EX register. This invalidates that instruction in the pipeline register until a new one is loaded in.
PSW_{in}	00000000000000001000000xxxxxxxxx	Enables buffer to send the contents of the MDR from the Data MM to the PSW.
ROMout	00000000000000000100000xxxxxxxxx	Outputs the constant value stored in the ROM specified by the current control string and combinational logic
ALUop (operation)	00000000000000000010000xxxxxxxxx	Enables the ALU control unit- ALU controls specified in the ALU control unit section of the report (**III.D)**
RegWrite	00000000000000000001000xxxxxxxxx	When asserted, the register address in the reg file pointed to by Rd in the EX/WB register is written with the data value indicated by the MemtoRd signal. When deasserted, nothing is written back to the register file.
MemtoRd	00000000000000000000100xxxxxxxxx	When asserted, signals to write the result from a memory operation to Rd.

		When deasserted, signals to write the result from an ALU operation to Rd. **Note:** This just selects the value to be written back. The data is not written back unless RegWrite is asserted.
IF/ID.PC$_{out}$	00000000000000000000010xxxxxxxxx	Sends the PC value stored in the IF/ID register to the MDR to be written in to data main memory to resolve exceptions.
PSWout	0000000000000000000000001xxxxxxxx	Outputs the data in the PSW to be written into the MDR when an exception occurs.

Table 5: Control Signal List

C. Machine Operations with Control Signals (separated by pipeline stage in which they occur)

Fetch	Decode	Memory	Execute	Write Back
1.Load	1.nop	**if I1 &I2 =0**	1.ALUop/(operation)	1.RegWrite
2.PC$_{Update}$		1.nop		
		if I1 = 0 I2 = 1		
		1.IND2, MARin, Read_MM		
		2.MDRout/OP2bus		
		if I1 = 1 and I2 = 0		
		1. IND1, MARin, Read_MM		
		2.MDRout/OP1bus		
		else (I1 = 1 I2 =1)		
		1.IND1, MARin, Read_MM		
		2.MDRout/OP1bus		
		3.IND2, MARin, Read_MM,		
		4.MDRout/OP2bus		

Table 6: Control Signals for Opcodes 0-5

The control signals for opcodes 0-5 are the same except for ALUop/(operation) in the Execute stage. The (operation) is determined by the ALU control unit and the signals are as follows:

Opcode 0 = ADD=ALUop/(100)
Opcode 1 = SUB=ALUop/(101)
Opcode 2 = AND=ALUop/(010)
Opcode 3 = SHL=ALUop/(110)
Opcode 4 = SHRA=ALUop/(111)
Opcode 5 = OR=ALUop/(011)

Fetch	**Decode**	**Memory**	**Execute**	**Write Back**
1.Load	1.nop	1.ID/MEM.[PC+offset]$_{out}$, MAR$_{in}$, Read_MM	1.ALUop/(NOT)	1.RegWrite
2.PC$_{Update}$		2.MDR$_{out}$/OP1bus		

Table 7: Control Signals for Opcode 6 (NOT)

Fetch	**Decode**	**Memory**	**Execute**	**Write Back**
1.Load	1.nop	1.ID/MEM.[PC+offset]$_{out}$,MAR$_{in}$,Read_MM	1.nop	1.MemtoRd, RegWrite
2.PC$_{Update}$		2.MDR$_{out}$		

Table 8: Control Signals for Opcode 7 (LD)

Fetch	**Decode**	**Memory**	**Execute**	**Write Back**
1.Load	1.nop	1.MAR$_{in}$,MDR$_{in}$,Write_MM	1.nop	1.nop
2.PC$_{Update}$				

Table 9: Control Signals for Opcode 8 (ST)

Fetch	**Decode**	**Memory**	**Execute**	**Write Back**
1.Load	1.nop	**if S = 1 and CC.N = 1**	1.nop	1.nop
2.PC$_{Update}$		1.IF/ID.Flush, ID/MEM.Flush		
		2..PC$_{Update}$		
		else S=0 or CC.N=0		
		1.MEM/EX.Flush		

Table 10: Control Signals for Opcode 9 (BRN)

Fetch	Decode	Memory	Execute	Write Back
1.Load	1.nop	**if S = 1 and CC.Z = 1**	1.nop	1.nop
2.PC_{Update}		1..IF/ID.Flush, ID/MEM.Flush		
		2..PC_{Update}		
		else S=0 or CC.Z=0		
		1.MEM/EX.Flush		

Table 11: Control Signals for Opcode 10 (BRZ)

Fetch	Decode	Memory	Execute	Write Back
1.Load	1.nop	1..IF/ID.Flush, ID/MEM.Flush	1.nop	1.nop
2.PC_{Update}		2..PC_{Update}		

Table 12: Control Signals for Opcode 11 (BR)

Fetch	Decode	Memory	Execute	Write Back
1.Load	1.RegWrite	1. .PC_{Update}	1.nop	1.nop
2.PC_{Update}				

Table 13: Control Signals for Opcode 12 (JSR)

Fetch	Decode	Memory	Execute	Write Back
1.Load	1.nop	1.PC_{Update}	1.nop	1.nop
2.PC_{Update}				

Table 14: Control Signals for Opcode 13 (RTS)

Fetch	Decode	Memory	Execute	Write Back
1.Load	if PSW.P=1	1.ID/MEM.[PC+offset]$_{out}$	1.nop	1.nop
2.PC$_{Update}$	1. nop	2.MARin, Read_MM		
	if PSW.P =0, ISB = 1	3.MDRout, Timer$_{In}$		
	Timeout exception/ trap			

Table 15: Control Signals for Opcode 14 when P=1 (CLK)

Fetch	Decode	Memory	Execute	Write Back
1.Load	**if PSW.P=1**	1.ID/MEM.[PC+offset]$_{out}$, MARi$_{n}$, Read_MM	1.nop	1.nop
2.PC$_{Update}$	1. nop	2. MDR$_{out,}$ PSW$_{in}$		
	if PSW.P =0			
	pgm check violation/exception			

Table 16: Control Signals for Opcode 15 when P=1 (LPSW)

Fetch	Decode	Memory	Execute	Write Back
1.Load	**PSW.P=0**	1. PSWout, ROMout, MARin, Write_MM	1.nop	1.nop
2.PC$_{Update}$		2. IF/ID.PCout, ROMout, MARin, Write_MM		
		3. ROMout, MARin, Write_MM, PSWin		
		4. ROMout, MARin, Read_MM, PCUpdate		

Table 17: Control Signals for Opcode 15 when P=0
(LPSW program check violation/exception)

Fetch	Decode	Memory	Execute	Write Back
1.Load	PSW.P = 0, ISB=1	1. PSWout, ROMout, MARin, Write_MM	1.nop	1.nop
2.PC_{Update}		2. IF/ID.PCout, ROMout, MARin, Write_MM		
		3.ROMout, MARin, Read_MM, PSWin		
		4. ROMout, MARin, Read_MM, PCUpdate		

Table 18: Control Signals for Timeout trap/exception

D. Control Strings

Below is a control string example of the control signals from opcodes 0-5. Each of the control strings are listed from the least significant bit to the most significant bit (0-31) in which correspond to the **Control Signal List** in section **IV.B**. The control strings are listed in the order they execute on each clock cycle.

Fetch
01000000000000000000000xxxxxxxxx
00100000000000000000000xxxxxxxxx

Decode
10000000000000000000000xxxxxxxxx

Memory
if I1 &I2 =0
10000000000000000000000xxxxxxxxx
if I1 = 0 I2 = 1
00001011000000000000000xxxxxxxxx

00000000000010000000000xxxxxxxxx
if I1 = 1 and I2 = 0
00010011000000000000000xxxxxxxxx
0000000000010000000000xxxxxxxxxx
else (I1 = 1 I2 =1)
00010011000000000000000xxxxxxxxx
00000000000100000000000xxxxxxxxx
00001011000000000000000xxxxxxxxx
00000000000010000000000xxxxxxxxx

Execute
00000000000000000010000xxxxxxxxx

Write Back
00000000000000000001000xxxxxxxxx

Table 19: Control String List for Opcodes 0-5

E. State Machine/Control Unit

The control strings for this machine are generated by the machine's control unit which is represented in this design by **Figure 12** shown below.

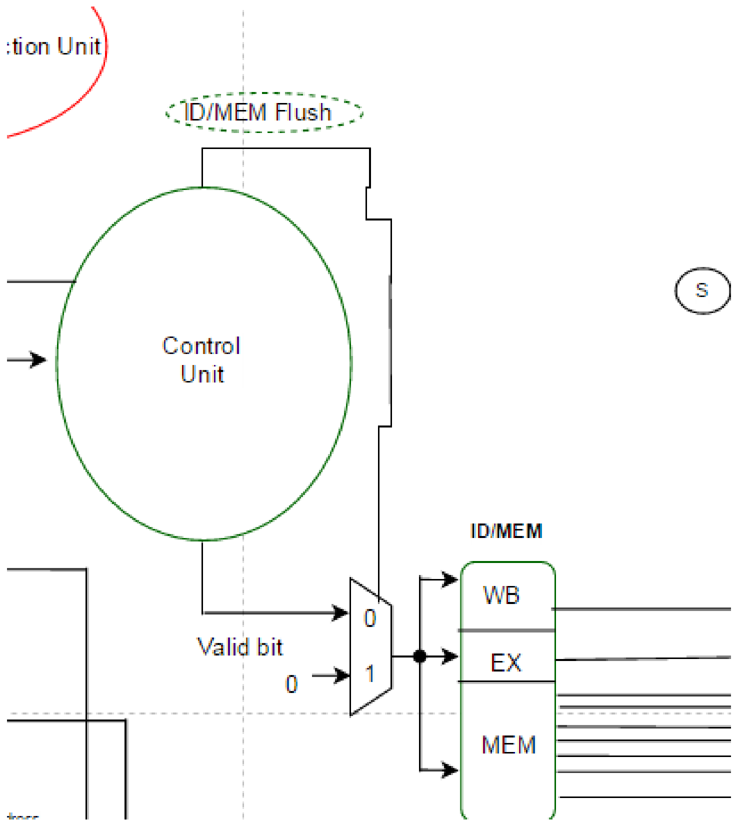


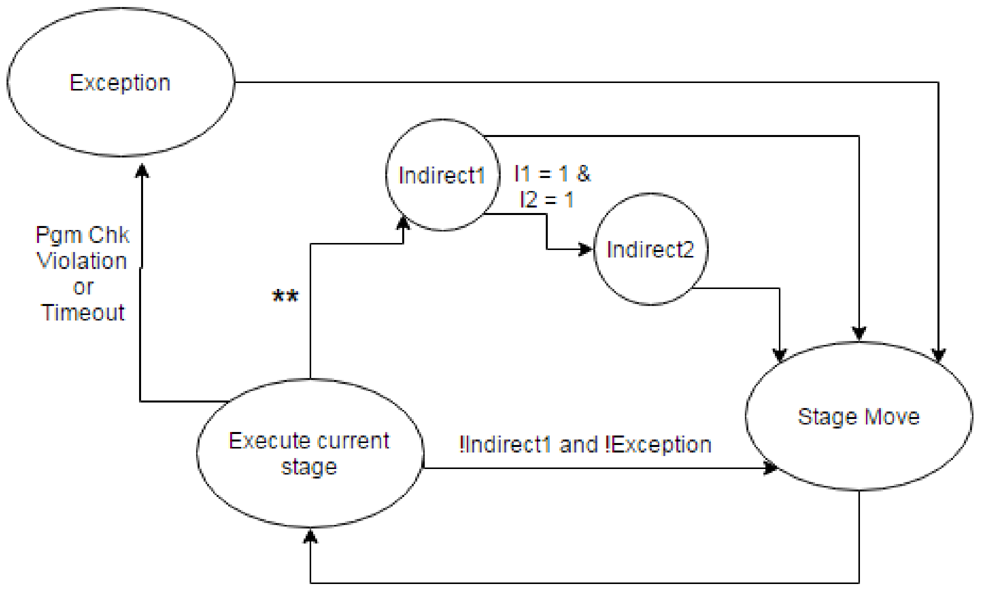
Figure 12: High level Control Unit (Located in Instruction Decode stage)

The control unit generates control strings based on the opcode from the current instruction loaded into the IF/ID pipeline register which was decoded in the instruction decode stage. When these control strings are generated, they are propagated to each pipeline register downstream from the control unit: ID/MEM, MEM/EX, EX/WB, respectively. One example of this is the WB/EX/MEM block shown in **Figure 12** above. The reason for propagating controls to each of the pipeline registers is that it allows the control strings to execute in later stages in the pipeline while other instructions are being executed in earlier stages. This is because control data for the current instruction is not propagated to the next pipeline stage until the current stage is done executing. In other words, this is how multiple instructions are executed at the same time. A brief explanation of the control signals propagated in each pipeline register is shown in **Table 20** below.

Pipeline Register	Control Propagation Functionality
IF/ID	Passes current instruction to the control unit from fetch stage. Thus, no signal propagation from the control unit is necessary. It is still controlled by the control unit whenever the PC changes via the PCUpdate signal.
ID/MEM	Buffers controls for the WB, EX, and MEM stages, while executing controls in the MEM stage.
MEM/EX	Buffers controls for the MEM and EX stages, while executing the MEM stage.
EX/WB	Buffers and executes controls for the WB stage.

Table 20: Functionality of each Pipeline Register

The state machine which generates the control strings and propagates them to the pipeline registers is shown in **Figure 13** below.



**I1=1 OR I2=1 OR Opcode(6,7,11) OR (Opcode 9, S=1, N=1) OR (Opcode 10 S=1, Z =1) OR (Opcode 15 & P=1)

Figure 13: State Machine for Pipelined Control Unit

A table of the present state and requirements to move to the next state is shown in **Table 21.** It is also shown in the state diagram in **Figure 13.**

Present State	**Conditions**	**Next state**
Execute current stage	!Indirect1&!Timer&!Exception	Stage Move
Execute current stage	I1=1 OR I2=1 OR Opcode(6,7,11) OR (Opcode 9, S=1, N=1) OR (Opcode 10 S=1, Z =1) OR (Opcode 15 & P=1)	Indirect 1
Execute current stage	If Opcode =14 & P=1	Timer
Execute current stage	If there is a program check violation or timeout	Exception
Indirect 1	(I2=0&I1=0) OR Opcode(6,7,11,9,10,15) substates complete	Stage Move
Indirect 1	(I1=1&I2)=1	Indirect 2
Indirect 2	Substates complete	Stage Move
Timer	Substates complete	Stage Move
Exception	Substates complete	Stage Move
Stage move	No stall detected	Execute Current Stage

Table 21: Next State Table

This machine is implemented using both sequential and combinational logic.
The execute current stage state will iterate through the substate(s) of the current pipeline state, and upon completion will move to the stage move state. The stage move state generates the PC_{Update} control signal. This allows another instruction to be fetched while the previous instructions are being carried out later in the pipeline by the control information stored in the respective pipeline registers. **Table 22** shows each state with its corresponding number of substates that must execute before a stage move can occur. Substates are implemented in the control unit using combinational logic based on the current instruction and PSW in the machine.

State	**Number of Substates (equal to number of clock cycles)**
Execute current stage (Fetch)	**2**
Execute current stage (Decode, Memory, Execute, Write back)	**1**
Indirect 1	**2**
Indirect 2	**2**
Timer	**3**
Exception	**4 (stalls for 3 cycles instead of bubbles)**

Table 22: Substate Table

If the current state being executed has more than 1 substate, the control unit will insert a bubble into the pipeline for each subsequent substate after 1. For example, fetch inserts 1 bubble into the pipeline and timer inserts 2. Bubbles are inserted by the control signal nop, which sends a control string of all 0’s to the requisite pipeline register. This indicates for that stage to do nothing until the current stage is completed.

V. Hazards

A. Data Hazards

A data hazard is when a instruction cannot execute during the proper clock cycle because data that is needed to execute the instruction is not yet available. There are two ways to solve these hazards which include stalling the pipeline and data forwarding. Stalling wastes a lot of clock cycles, so we implemented data forwarding in our machine.

Data forwarding is done by adding hardware to the pipeline which forwards or transmits the desired value to the pipeline segment that needs that value. In this machine this added hardware is our Data Forwarding Unit in section **III.G**. Stalling is discussed in section **III.H.**

From the instructions given in the specifications of the project, there are several potential cases in which can cause a data hazard. These cases are denoted by x's in the table below.

Opcode	Rd is Used	Rd is Written to
0-5	x	x
6		x
7		x
8	x	
9		
10		
11		
12		x
13	x	
14		
15		

Table 23: Potential Data Hazards

When an instruction uses Rd, that instruction could have a read after write (RAW) data hazard because the instruction may refer to a result that has not yet been made available. An example of a RAW data hazard with opcode 0 may be as follows:

Instr.1 R2 = R1+**R3**
Instr.2 **R3** = R1 + R4

When an instruction writes to Rd, that instruction can cause a write after read (WAR) data hazard if there is concurrent execution. An example of a WAR data hazard with opcode 2 is:

Instr.1 R2 = R5 AND **R4**
Instr.2 **R4** = R5 AND R1

Where in a situation Instr.2 may complete before Instr.1 (concurrent execution).

In this machine there are three main potential data hazards.

Problem 1:
For Opcodes 0-5, Rs1 or Rs2 in the current instruction may utilize Rd of the previous instruction.

Solution:
Anytime the ALU performs an operation, the result is sent to the data forwarding unit where it is compared with the Rd any of the following instructions currently in the pipeline. Therefore, the ALUresult is compared with the ID/MEM.Rd, MEM/EX.Rd, and EX/WB.Rd values in the corresponding pipeline registers. If the combinational logic within the forwarding unit determines that any of the Rd values match, further logic in the forwarding unit determines to output the ALU result and overwrites the value in either ID/MEM.OP1 or ID/MEM.OP2.

Problem 2: Opcode 13 (RTS) needs the Rd result from the previous instruction before being able to add it to the short offset.

Solution: IF/ID.Rd is sent to the forwarding unit to be compared with any Rd values of previous instructions further down the pipeline. These are stored in the ID/MEM.Rd, MEM/EX.Rd, and EX/WB.Rd pipeline registers. If the Rd value in the IF/ID register matches with any of these Rd values down the pipeline, IF/ID.Rd is overwritten to that value.

Problem 3: Opcode 8 (ST). If the Rd that is supposed to be stored in memory is from a previous instruction.

Solution: To prevent the hazard, ID/MEM.Rd is sent to the data forwarding unit to be compared with the Rd of the following instructions which are MEM/EX.Rd and EX/WB.Rd. If the values are equal, the value is overwritten into ID/MEM.Rd.

B. Control Hazards

A control hazard occurs when an instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed. This hazard is caused by branch instructions in this machine (refer to **Table 3** opcode 9,10, or 11).

Branch instructions cause the PC to be updated with the value PC+Long_offset in the next instruction.

To avoid these control hazards the machine predicts that branches are not taken for every instruction that is fetched. This is a static prediction scheme because the prediction does not change. This is implemented with a multiplexer and encoder as shown in **Figure 3**. The zero line to the encoder propagates the value PC+2 to the PC for the next instruction. The zero line is enabled by default for every instruction unless the instruction dictates the PC be set to a different value. If a branch is taken it enables the 1 line on the encoder which loads the branch target address into the PC.

***Clarification:** Although the PC can be set to several different values with the multiplexer, it only has to stall when a branch target address is selected with combinational logic, as the previous instructions have to be flushed. The other possible PC values are loaded as soon as their respective instructions are completed.*

Static branch prediction makes it so there is only a stall for one clock cycle if the branch is taken. If there was no branch prediction scheme the machine would have to finish the execution of every other instruction in the pipeline registers before taking the target address of the branch instruction. This would take 4 stalls as opposed to 1 when branch prediction is implemented. (refer to **III.H** for how stalls are implemented).

C. Structural Hazards

A structural hazard is when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

In this design a potential structural hazard was avoided by placing the memory stage in front of the execute stage to support indirect addressing. In order to compute the effective address the operand must first be read into memory in before executing the instruction because of a potentially indirect address. Trying to execute before reading into memory would result in a structural hazard.

VI. Appendix

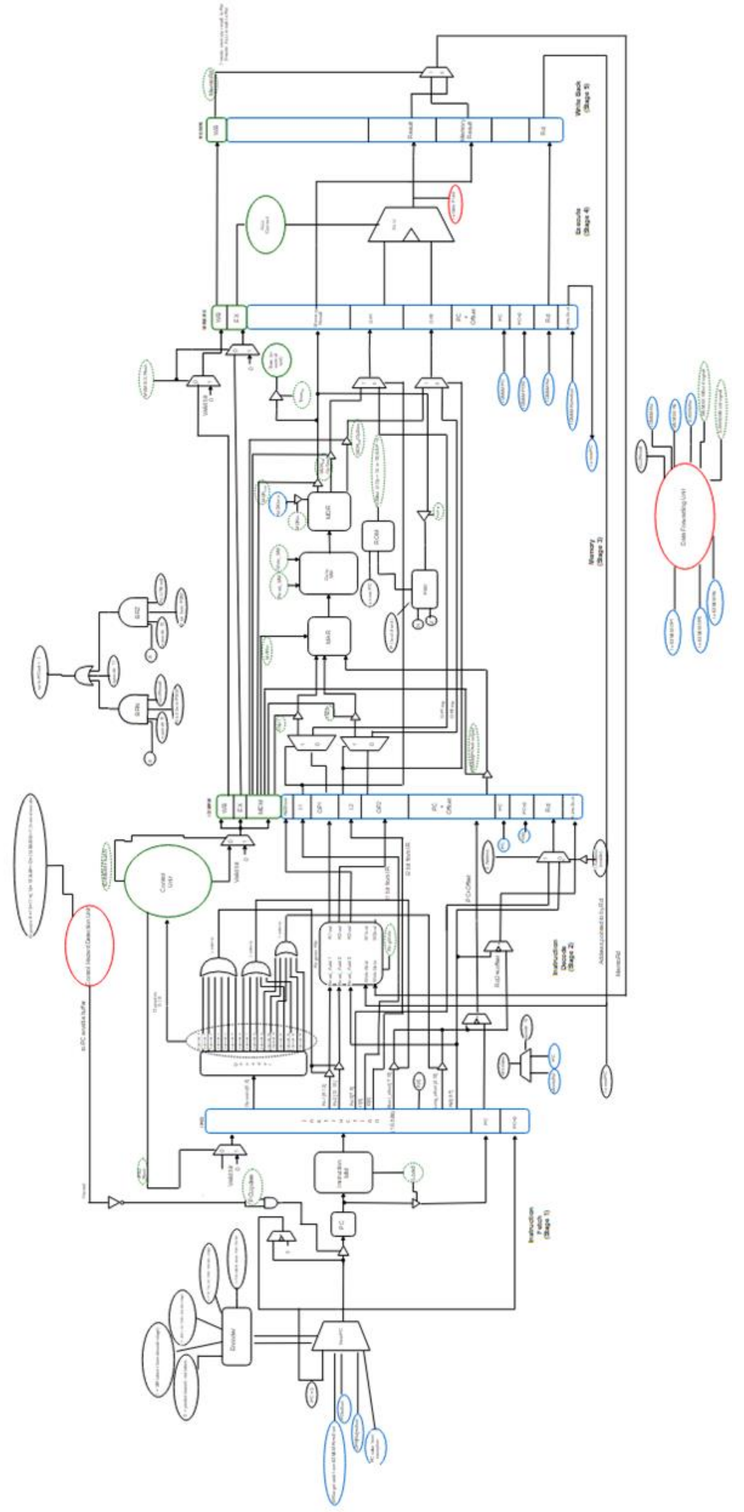


Figure 14: Complete Pipelined Control Unit Design

VII. <u>References</u>

Dr. Philip Wilsey's website
http://eecs.ceas.uc.edu/~paw/

Design of Pipelined MIPS Processor - Carnegie Mellon University
https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/info/pipeline-slide.pdf

MIPS Pipeline - Cornell University
http://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/09-pipelined-cpu-i-g.pdf

Organization of Computer Systems: § 5: Pipelining - University of Florida
https://www.cise.ufl.edu/~mssz/CompOrg/CDA-pipe.html

Patterson, David A., and John L. Hennessy. *Computer Organization and Design the Hardware/software Interface*. Amsterdam: Morgan Kaufmann, 2016. Print.