Project 4: Pipelined Control Unit

Anthony Gamerman & Zuguang Liu

EECE3026 |Professor Philip Wilsey

4/6/2021

Table of Contents

1 Assignment Summary

This section summarizes the assignment as a checklist for the final delivery of the project.

A pipelined processing unit shall be designed capable of executing a specified instruction set. The design should specify the data path, the control unit signaling, and the state machine, but the gate-level implementation is not needed.

1.1 Basic Characteristics

Basic characteristics of the machine is listed below.

- Word size of 16 bits
- Memory address and data bus size of 16 bits
- Byte addressable memory
- 64K byte main memory
- A 16-bit program status word (PSW) register illustrated in Figure 1-1, where `Z` is set when operation result is `0`, `N` is set when operation result is `0xFF`, `P` is set when machine is in privileged mode, and P is clear when machine is in user mode. Only the first 14 instructions can be executed in user mode while all 16 can be executed in privileged mode.

Figure 1-1 Program status word register spec.

- 8x16-bit general purpose register (GPR), where a constant `0` is stored at address 0, program counter (PC) is stored at address 7.
- 16-bit count-down timer
- 2's complement number representation

1.2 Instruction Specifications

There are 3 types of instruction formats (Figure 1-2) and 16 instructions (Figure 1-3) that follow them in the instruction set. `Z` and `N` bit in PSW are conditioned according to the operation result when `S` bit in instruction is set.

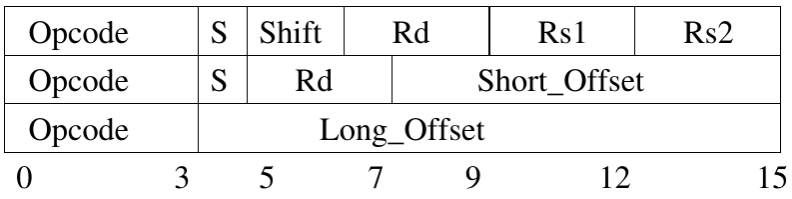

Figure 1-2 Instruction format.

Name	Opcode	Description
ADD	0	GPR[Rd] = GPR[Rs1] + left_shifted(GPR[Rs2],IR.Shift)
SUB	1	GPR[Rd] = GPR[Rs1] - left_shifted(GPR[Rs2],IR.Shift)
AND	2	GPR[Rd] = GPR[Rs1] **and** left_shifted(GPR[Rs2],IR.Shift)
SHL	3	GPR[Rd] = shift_left(GPR[Rs1]) by left_shifted(GPR[Rs2],IR.Shift)$_{3-0}$
SHRA	4	GPR[Rd] = shift_right(GPR[Rs1]) by left_shifted(GPR[Rs2],IR.Shift)$_{3-0}$
OR	5	GPR[Rd] = GPR[Rs1] **or** left_shifted(GPR[Rs2],IR.Shift)
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]

Figure 1-3 Instruction set.

1.3 Exception Specifications

There are two types of exceptions in this machine.

Program check violation exception is triggered when `CLK` or `LPSW` instruction is used in user mode. The behavior of the exception is shown in Figure 1-4 (A). This is a precise exception (completing all preceding instructions before exception happens).

Timeout exception is triggered at instruction boundary when countdown timer register is 0. Its behavior is shown in Figure 1-4 (B). This exception does not need to be precise.

(A)

1. MM[0] = PSW
2. MM[2] = PC
3. PSW = MM[4]
4. PC = MM[6]

(B)

1. MM[8] = PSW
2. MM[10] = PC
3. PSW = MM[12]
4. PC = MM[14]

Figure 1-4 Exception behavior: program check (A) and timeout (B).

1.4 Constraints

Some additional constraints of the assignment are listed below.

- No hard restriction on data path, but all paths should be used reasonably.
- Main memory (MM) has separate instruction and data ports that can be used simultaneously.
- Register file (GPR) has 3 read ports and 2 write ports and all can be accessed simultaneously.
- The pipeline should include at least three stages, corresponding to the fetch/decode/execute cycle.
- Read-only memory (ROM) may be used to contain up to 8 constants.

2 Design Overview and System Components

2.1 Pipeline Overview

The pipeline consists of four unique stages: F1, F2, D1 and E1, matching the fetch/decode/execute cycle instruction cycle. Although some states share hardware resources, each has its own portion of the data path and dedicated portion of the control unit. By this design, each state only uses 1 clock cycle to finish, resulting in an ideal instruction completion time of 4 clock cycles – with a possible 4 instructions in the pipeline. By comparison, the non-pipelined single bus design could execute 1 instruction in an average of approximately 6 clock cycles.

Unfortunately, due to hardware design constraints, certain hazards arise which can slow down the system. Although specific solutions to system hazards are discussed later, it should be noted that the handling of hazards never adds more than 2 clock cycles to an instruction completion time. For system exceptions, each trap sequence requires 6 clock cycles to carry out (using 2 “virtual” states). This means that the actual performance of the system, under typical use, completes instructions at an average between 4 and 6 clock cycles. Furthermore, most hazards, apart from control hazards, are solved while retaining all lagging pipeline instructions. This means that while processing time can increase, system efficiency is rarely sacrificed.

This 4-stage pipeline design was specifically chosen for its speed and efficiency benefits. Another design that was considered utilized 3 stages using 2 clock cycles each. This design would have simplified hardware by eliminating certain hazards but was bottlenecked to a processing time of 6 clock cycles with only 3 instructions in the pipeline at any given time. It was decided that reducing the processing bottleneck and increasing potential system efficiency was worth increased complexity and hardware requirements. The system presented here is optimized to its most reasonable full potential; While more improvements are possible, hardware additions begin to provide diminishing returns where added cost and complexity is not worth the processing benefits. Figure 2-1 below shows a sample of the stage timing from the startup of the machine as it goes through the 4 stages of the pipeline to execute instructions. This scenario is ideal, without encountering hazards or exceptions. Detailed sequence diagrams for these scenarios are presented in their respective sections.

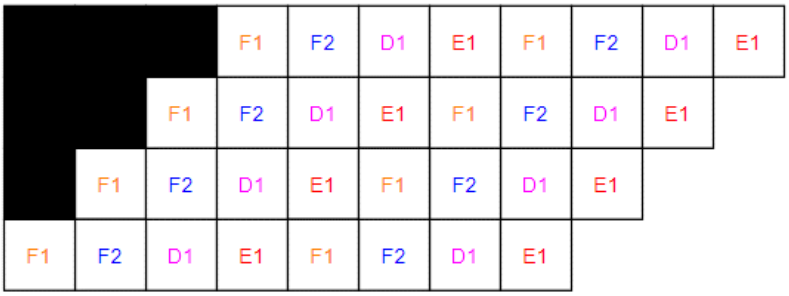

Figure 2- 1 Four-Stage, Four Instruction Pipeline Timing Example from Startup

2.2 Datapath Overview

A top-level architecture of the datapath is explained in this section. The design uses modular presentation. Each module is used extensively by the CPU to meet the design requirements and achieve optimization. The detail of each module is discussed in later sections.

All diagrams such as Figure 2-1 were made in Draw.io[1] A copy of the original project assignment can be found on Professor Wilsey's website[2]. Additionally, since the pipelined unit modifies the non-pipelined single bus solution from Project 3, many abstract circuit block functions and operations are defined in greater detail in Project 3 and not covered again in Project 4. Please reference Project 3[3] for further explanation of individual data blocks.

The CPU uses 4 busses and numerous point to point (P2P) connections to implement a 4 stage pipeline where each stage uses one clock cycle for completion. The busses are in black while the P2P connections are colored in blue. Additionally, tristate control buffers before circuit blocks are depicted with white arrows.

Abstract overviews of the 4 stages of the pipeline are shown below and a section of traps. These stages will be further discussed in their individual sections later in the report. Figure 2-2 below shows the first two stages, F1 and F2 which fetch the instruction to the IR from the I-MDR in F2, and put the PC from the GPR into the I-MAR and increment it in F1. Both stages have their own buses for transmitting the PC data during the fetch stages.

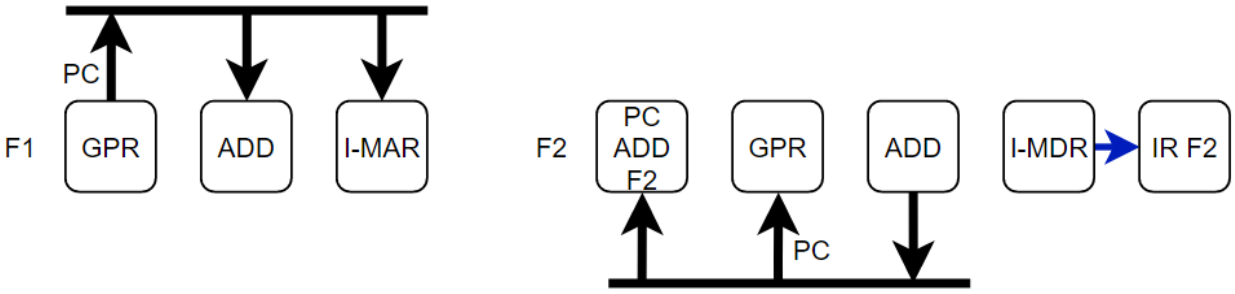

Figure 2-2 Abstract View of Stage 1 (F1) and Stage 2 (F2)

---

[1] Available at https://app.diagrams.net/.
[2] P. Wilsey, *Project 4*. 2021. Available at https://eecs.ceas.uc.edu/~wilseypa/classes/eecs3026/project/project4.pdf
[3] A. Gamerman and Z. Liu, *Project 3*. 2021.

Following the fetch, the decode state will use the data from the IR to perform the first part of the instructions by decoding them. This stage utilizes an IR for the decode state, with a built-in adder (discussed in a future section) with its own bus to write to the D-MAR, and the GPR. The GPR will also output to one of the inputs to the ALU and a temporary storage for the PC for the decode IR. This is further discussed in Section 5 which covers the decode and execute stages.

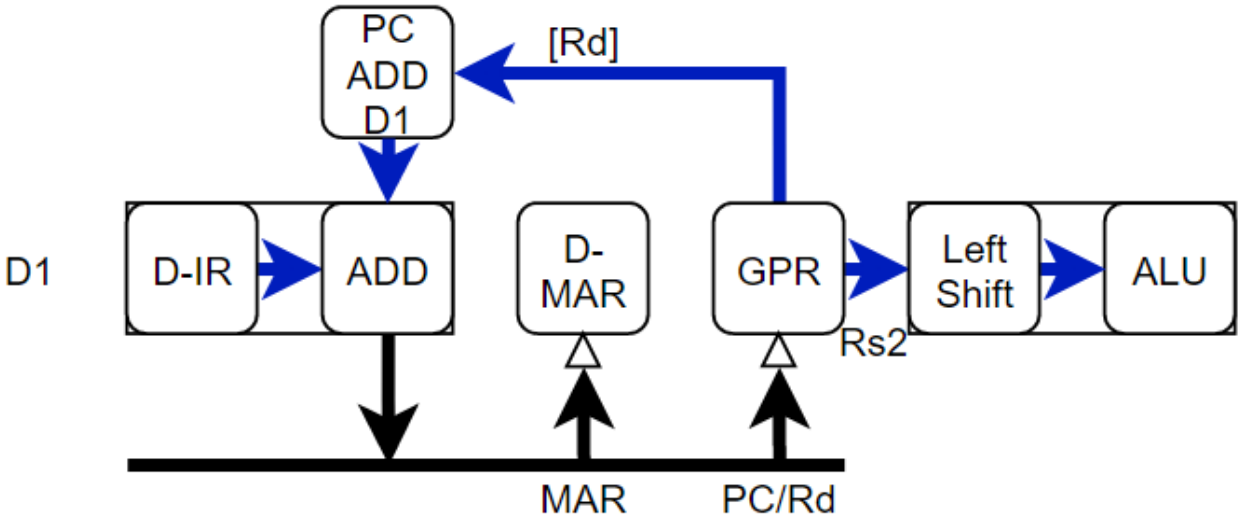

Figure 2- 3 Abstract View of Stage 3 (D1)

The fourth and last stage of the pipeline is the execute stage, E1 and is shown below in Figure 2-4. It utilizes a separate IR for the execute stage with a built-in adder with a bus to write to the GPR. The same bus can also be utilized for writing from the GPR to the ALU. Numerous point to point connections also allow for faster execution of instructions. The GPR can also directly write to the D-MDR and be written to from the ALU and the D-MDR. Additionally, the D-MDR can write to the PSW and Timer when necessary.

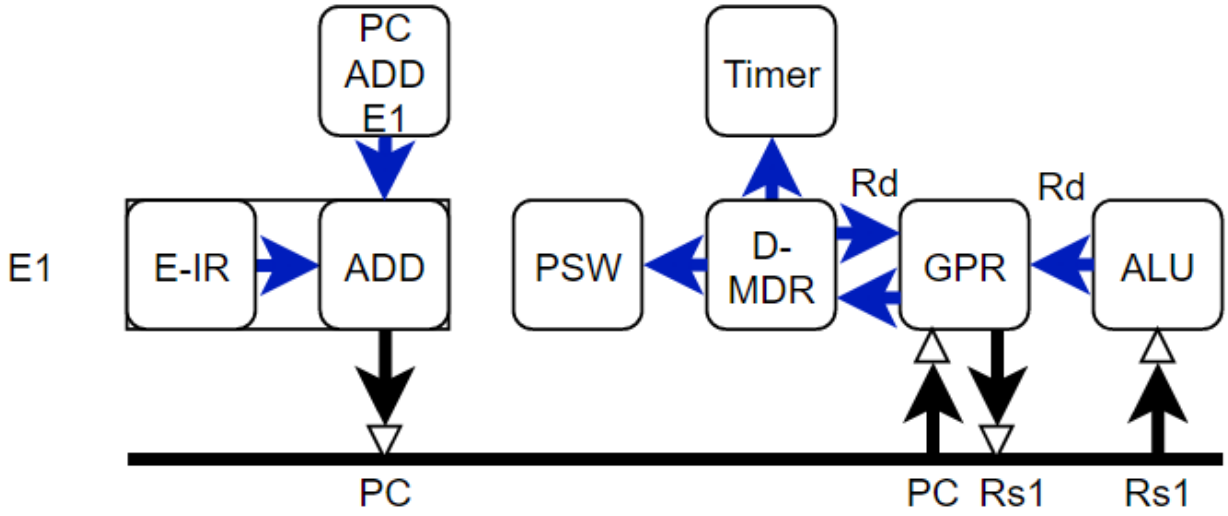

Figure 2- 4 Abstract View of Stage 4 (E1)

Finally, in the case of exceptions, the pipeline utilizes exclusively point to point connections to write to the D-MDR from the ROM, GPR and PSW based on the two types of exceptions, XP a Program Check Violation, and XT a Timeout. The D-MDR can also write to the GPR and PSW per the necessary instructions that are covered later in Section 6.

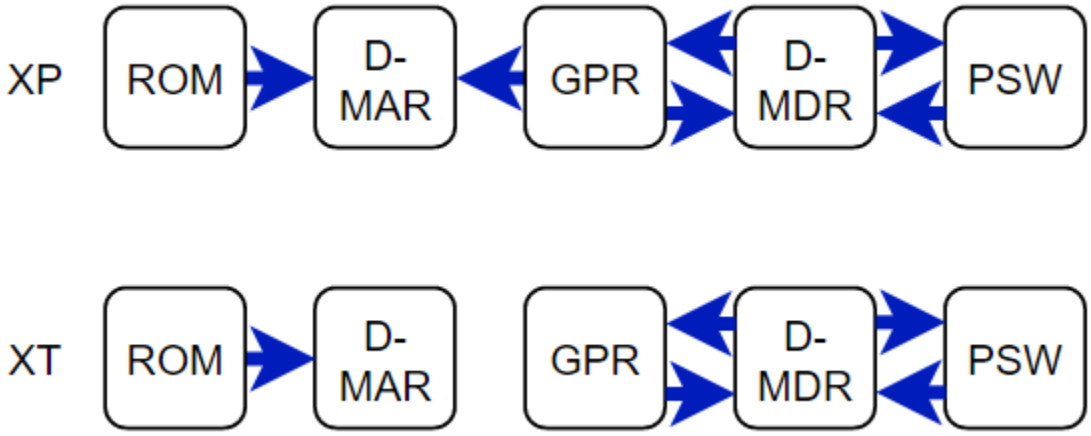

Figure 2- 5 Abstract View of Stage 5 (XP) and Stage 6 (XT)

There are several components that have modules within them, these are shown with the modules that are boxed together. These and the further connections between the abstract views will be further explained in their respective sections. The rest of Section 2 covers the individual modules shown in the abstract views.

2.3 IR, PC, and the Queue System

In order to ensure that instructions are executed in 4 clock cycles, the IR has been expanded from the previous Project. There are now 2 stages of the IR and a temporary register (F2-IR) and temporary PC registers associated with them.

The implementation of the queuing on IR and PC is 16 stacks of 4-bit and 3-bit shift registers, respectively. The memory organization of them can be seen in Figure 2-6, where each row is a shift register that can be controlled to shift, keep, or clear on each element, whereas a column represents a full IR/PC data. Inserting into the queue is done by writing the 16-bit wide data into the first element at every row, with the datapath and control signal discussed later about the first stage. The output of each flip-flop in one column effectively forms the read-out port for the IR and PC in the queue. Controlling the shifting behavior can be implemented by intercepting the clock on the flip-flops. Additionally, by having a digitally controlled buffer from the clock to the RESET of the flip-flops, the register can also be cleared synchronously with the rest of the machine. Figure 2-7 visualizes an example of this control scheme. Each clock cycle the data shifts from one IR and PC to the next. So, the data from the decode stage in D-IR and PC.ADD.D1 will be in E-IR and PC.ADD.E1, respectively in the following clock cycle for execution on the same information.

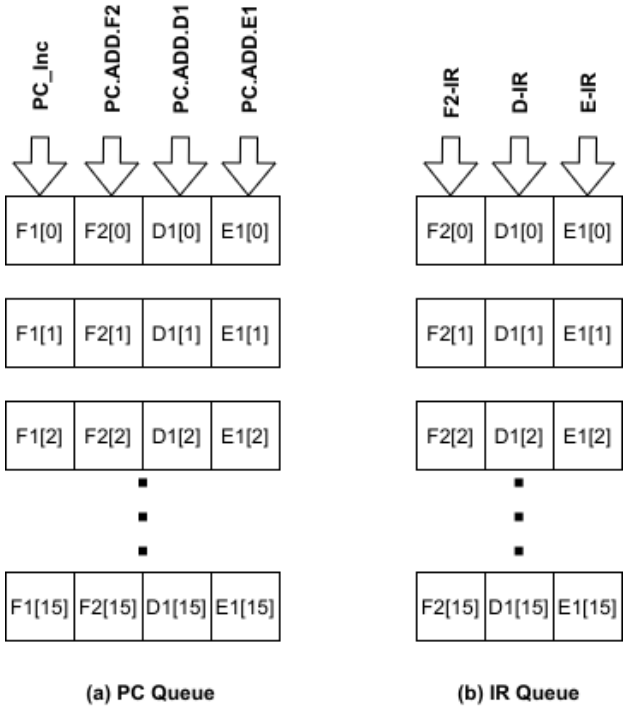

Figure 2- 6 Memory organization of the queues

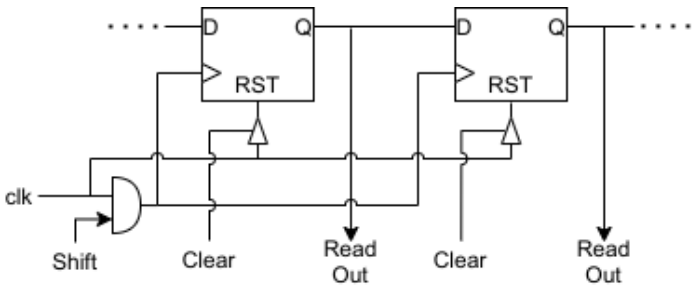

Figure 2- 7 Example implementation on the shift register control.

To ensure the correct information is worked on in each stage, the machine keeps multiple copies of the instruction and PC information as it moves forward in the pipeline. Each stage only has interface to its corresponding IR and PC data, such that they follow the expected program flow. After the instruction is fetched, it is inserted into the queue from start along with PC. Normally the queue moves forward each clock cycle. After the last job is executed, it gets removed from the queue. In case of hazards and exceptions, the Control Unit will decide about stalling (by keeping the original data) or flushing (by clearing the data) on different parts. An abstract view of the circuit blocks in the queue system used in Stages 3 and 4 is shown below in Figure 2-8 with the values between the PC.ADDs and IRs being controlled with the queue system shown in Figure 2-6 on the previous page.

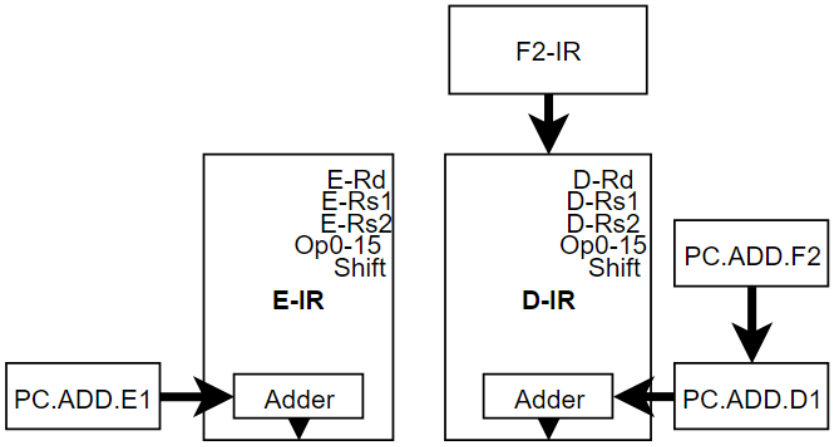

Figure 2- 8 Instruction Registers and Associated PC Temporary Storages

Additionally, to speed up operation, the two fully implemented IR's (D-IR and E-IR) have an adder built into them in order to be able to add the sign extended Short and Long offset to the PC directly inside the IR for instructions from OP codes 6 to 15. An example of this in the D-IR is shown below in Figure 2-9. This saves time and resources for the ALU since the offsets can be taken from the IR registers and added with the PC after processing in the same clock cycle, it allows for the ALU to be free for other OP code operations discussed later. The control signal for addition in the IR when held high will add the PC to the Long/short offset, and when low will simply pass the PC through the IR.

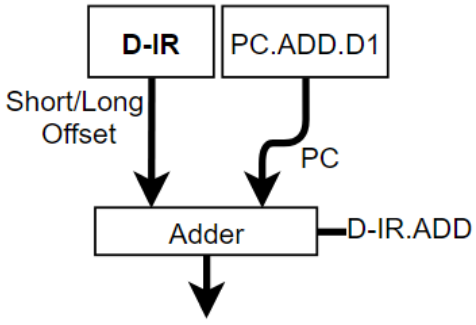

Figure 2- 9 Built in IR Adder

2.4 GPR

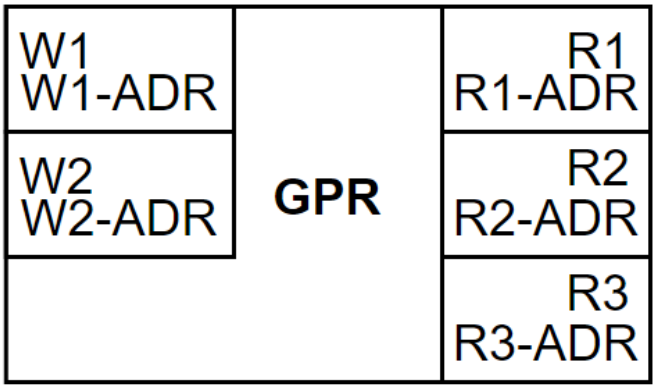

Figure 2- 10 GPR Abstract Circuit Block

Figure 2-10 above shows the GPR that is used in the pipelined CPU. The GPR consists of 2 write ports, their respective addresses and 3 read ports and their respective addresses. Since there are multiple read/write ports it was decided to allocate them to different functions of the pipeline. Write 1 and Read 1 (W1 and R1 respectively) were allocated for the fetch Stages 1 and 2. Write 2 is reserved for the decode and execute (Stages 3 and 4, respectively) while Read 2 is allocated to the decode and Read 3 is allocated to the execute.

The addresses are also controlled by their respective stages using multiplexers to select the outputs. Since W1 and R1 are only used for fetching the PC, their addresses, W1-ADR and R1-ADR are controlled with only the PC. However, the W2-ADR, R2-ADR and R3-ADR are controlled with multiplexers as shown below in Figure 2-11.

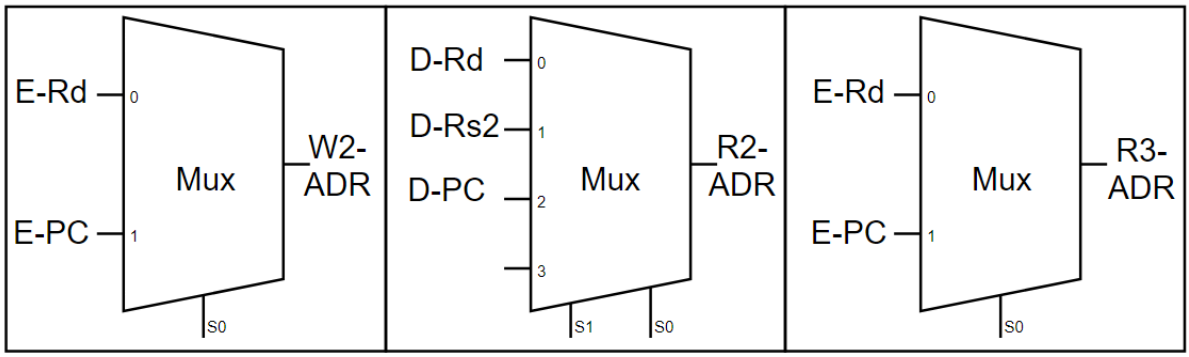

Figure 2- 11 GPR Address Selection MUXs

The addresses come from the decode and execution instruction registers, as stated by the D and E before the addresses. Additionally, the MUX selection bits are controlled by combinational logic from the OP codes that is shown in the Figure 2-12 below.

	E-Rd	E-PC	D-Rd	D-Rs2	D-PC
W2-ADR	OP0 – OP7	OP9 – OP13			
R2-ADR			OP13	OP0 – OP5	OP12
R3-ADR	OP0 – OP7	OP9 – OP13			

Figure 2- 12 Address MUX Selector Combination Logic OP Codes

2.5 Memory Interface

As specified in the project constraints, a memory model with a separate instruction and data ports is allowed to be used to issue an instruction read simultaneously data read/write. The two separate memory models for the instructions and data are shown below in Figure 2-13

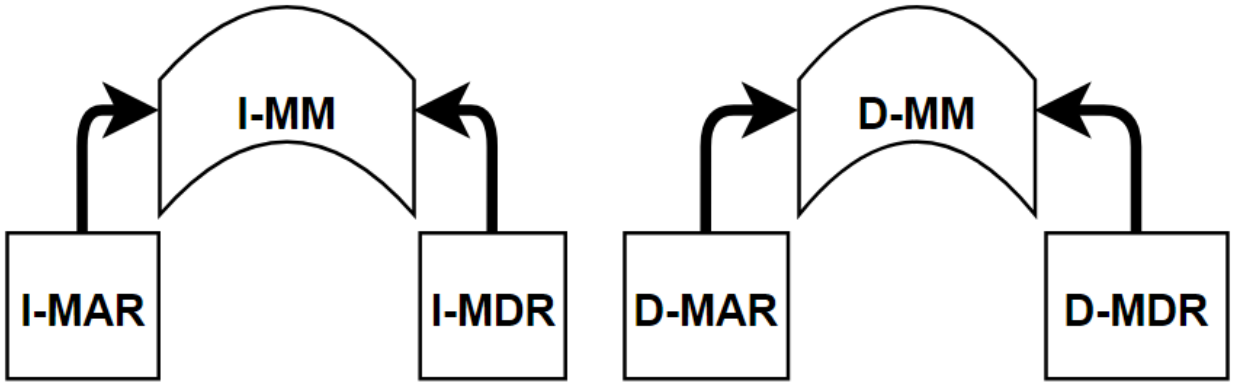

Figure 2- 13 Instruction and Data Memory Models

As also stated in the project constraints, the memory operation of WRITE_MM and READ_MM will complete in one clock cycle. Since the writing to and from is completed in one clock cycle, there are opportunities for hazards if the data in the MAR points to a different address than the correct data coming out of the MDR, an opportunity for data hazards arise. This is addressed in Section 7 for data hazards.

2.6 ROM, PSW, Timer

Below are the 3 additional circuit blocks that are featured in the CPU. The PSW and Timer operate identically to the single bus non-pipelined version from Project 3. However, there is one noticeable difference with the ROM. Whereas in Project 3, the ROM was comprised of a single register that held the value “8”, now the ROM is comprised of 7 registers, that hold the static values 2,4,6,8,10,12 and 14 for use within the trap conditions to speed up the operation instead of having to go through the ALU for incrementation. This is discussed further in Section 6 for exceptions/traps.

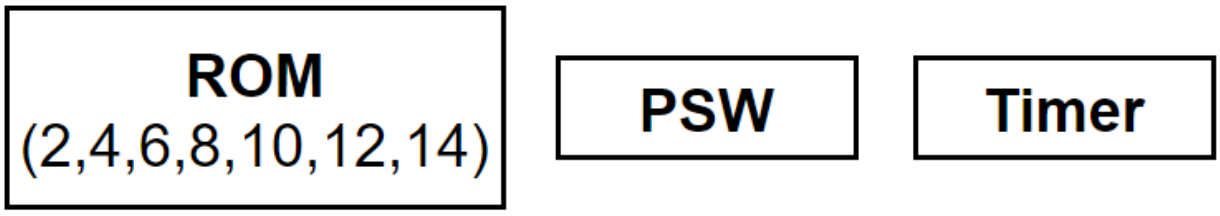

Figure 2- 14 ROM, PSW, and Timer Abstract Circuit Blocks

2.7 ALU

Due to various components having their own adders, the ALU has been simplified from the non-pipelined singular bus version. Figure 2-15. below shows the abstract circuit block of the ALU.

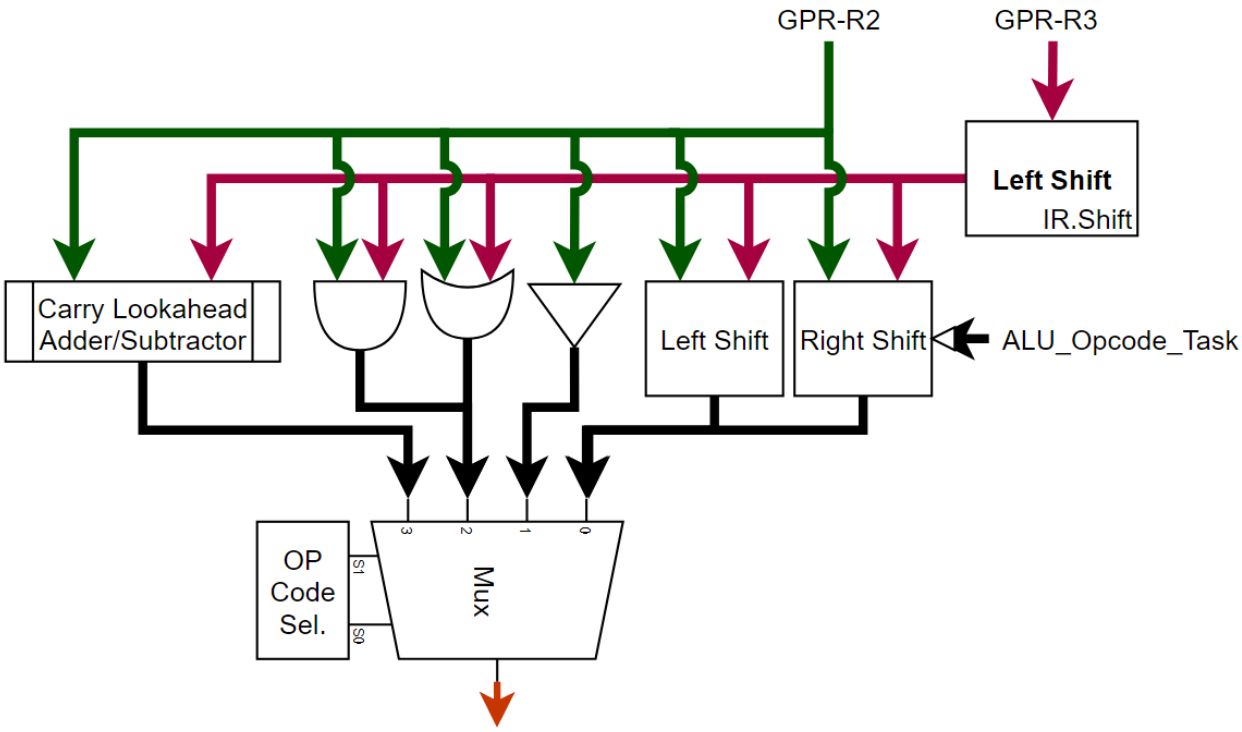

Figure 2- 15 ALU Abstract Circuit Block

The ALU now only takes in data from 2 buses, each that come from the GPR read ports into the GPR. The Left Shift block directly after GPR-R3 will shift the value GPR[Rs2] by IR.Shift based on the selection bits of the incoming data like in Project 3. However, the data can also not be shifted and fed straight through to the worked on by various operations inside the ALU with GPR[Rs1] coming from the other bus of data into the ALU. The task that the ALU performs and outputs via the multiplexer is selected using the ALU_Opcode_Task control signal identically to the previous Project 3 report.

3 Control Unit

The control unit supervises the operations of stages, then in each stage, it produces the control signals to finish the fetch/decode/execute cycle for each instruction. This section starts with discussing the supervision at the top level, then break down into separate stages on the control signals.

3.1 Top Level

The control unit on the top level is demonstrated in Figure 3-1. The Stage Controller enables and disables the stages of the pipeline. The Queue Controller keeps the queue in synchronization with the stage behavior by moving forward the shift registers of the IR and PC as discussed previously. At this level, the machine can be seen repeating on a normal state so long as there are no hazards or exceptions. In the event when they appear, the control unit enters states where it resolves the issue, then go back to looping state. These states are listed in Table 3-1, along with the GO state when the machine operates normally. The table also specifies whether to enable specific stages and the control over the queue, which will be justified in sections on hazards and exceptions. State transitions are demonstrated in Figure 3-2.

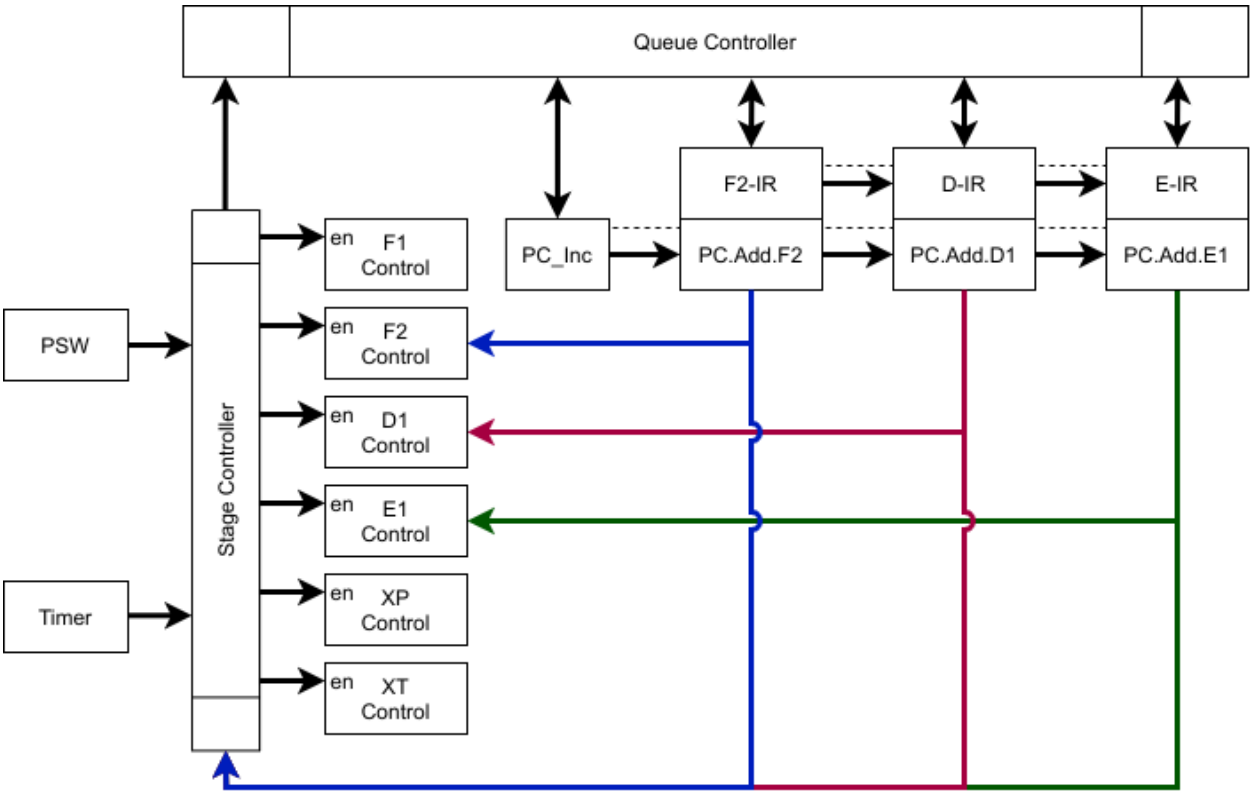

Figure 3-1 Control Unit Top Level Design

States	$F1_{en}$	$F2_{en}$	$D1_{en}$	$E1_{en}$	XP_{en}	XT_{en}	Move Queue	Additional Control
GO	1	1	1	1	0	0	1	-
XT1	0	0	1	1	0	0	1	-
XT2	0	0	0	1	0	0	1	-
XT3	0	0	0	0	0	1	0	-
XT4	0	0	0	0	0	1	0	-
XT5	0	0	0	0	0	1	0	-
XT6	0	0	0	0	0	1	0	-
XP1	0	0	0	0	1	0	0	-
XP2	0	0	0	0	1	0	0	-
XP3	0	0	0	0	1	0	0	-
XP4	0	0	0	0	1	0	0	-
S/DH	0	0	0	1	0	0	0	Clear PC.Add.E1, E-IR
CH	0	0	0	1	0	0	0	Clear all shift registers

Table 3-1 Control on stages and queue by states

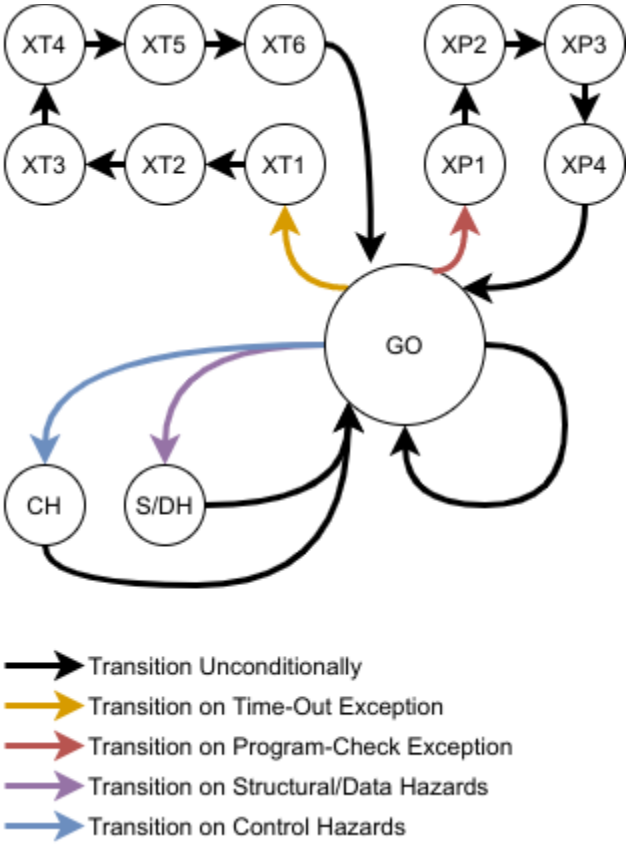

Figure 3-2 State transition diagram

3.2 Stage Controllers

When enabled, controllers in each stage dictates specific control signals on data path, memory unit and ALU. The first two stages (F1 and F2) fetch the instructions unconditionally, whereas the latter two stages (D1 and E1) determine the signal by combinational logic per OP-code. In this report, the implementation of the combinational logic at the gate level is not needed, so the conditions are descriptively listed in Table 3-2.

Stage	OP Code	Control Signals
F1	-	PC_Out, I-MAR_In, PC_Inc_In, READ_I-MM
F2	-	PC_Inc_Out, PC_In, PC.ADD.F2_In F2-IR_In IMDR_Out
D1	0,1,2,3,4,5	GPR[D-Rs2]_out, ALU_R2_In, ALU.Leftshift_IR.shift
	6,7	D-IR.Short_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD D-IR_ADD_OUT_3, D-MAR_In_3, READ_D-MM
	8	D-IR.Short_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD D-IR_ADD_Out_3, D-MAR_In_3
	9,10,11	- (NOP)
	12	GPR[D-Rd]_In_3, GPR[7]_Out_3
	13	GPR[D-Rd]_Out_3, PC.ADD.D1_In_3
	14,15	D-IR.Long_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD D-IR_ADD_Out_3, D-MAR_In_3, READ_D-MM
E1	0,1,2,3,4,5	GPR[E-Rs1]_out, ALU_R3_In, ALU_OP_Task, GPR[E-Rd]_In
	6	D-MDR_Out, Pre_SHU_NOT, GPR[E-Rd]_In
	7	D-MDR_Out, Pre_SHU_PASS, GPR[E-Rd]_In
	8	GPR[E-Rd]_Out, D-MDR_In, WRITE_D-MM
	9,10,11	E-IR.Long_Off_Out, E-IR_PC.ADD.E2_In, E-IR_ADD, E-IR_ADD_Out_3, GPR[7]_In_3
	12,13	E-IR.Short_Off_Out, E-IR_PC.ADD.E1_In, E-IR_ADD, E-IR_ADD_Out_4, GPR[7]_In_4
	14	D-MDR_Out_4, Timer_In_4
	15	D-MDR_Out_4, PSW_In_4

Table 3-2 Control signals on specific stages and OP-codes

3.3 Exception Handler

XP and XT are modules that specially handles the Program-Check and Timeout exceptions, when the pipeline is temporarily disabled and the machine behaves sequentially. Control signals at each sequential states are given in Table 3-3, which is justified in the Exception section.

Exception Handler	State	Control Signals	
XP	XP1	GPR[0]_Out_1, D-MAR_In_1	PSW_Out_2, D-MDR_In_2, WRITE_D-MM
	XP2	ROM2_Out_1, D-MAR_In_1	PC_Out_2, D-MDR_In_2, WRITE_D-MM
	XP3	ROM4_Out_1, D-MAR_In_1, READ_D-MM	D-MDR_Out_2, PSW_In_2
	XP4	ROM6_Out_1, D-MAR_In_1, READ_D-MM	D-MDR_Out_2, PC_In_2
XT	XT3	ROM8_Out, D-MAR_In_1	PSW_Out_2, D-MDR_In_2, WRITE_D-MM
	XT4	ROM10_Out_1, D-MAR_In_1	PC_Out_2, D-MDR_In_2, , WRITE_D-MM,
	XT5	ROM12_Out_1, D-MAR_In_1, READ_D-MM	D-MDR_Out_2, PSW_In_2,
	XT6	ROM14_Out_1, D-MAR_In_1, READ_D-MM	D-MDR_Out_2, PC_In_2

Table 3-3 Control signals on exceptions

4 Fetch Stage 1 and Fetch Stage 2

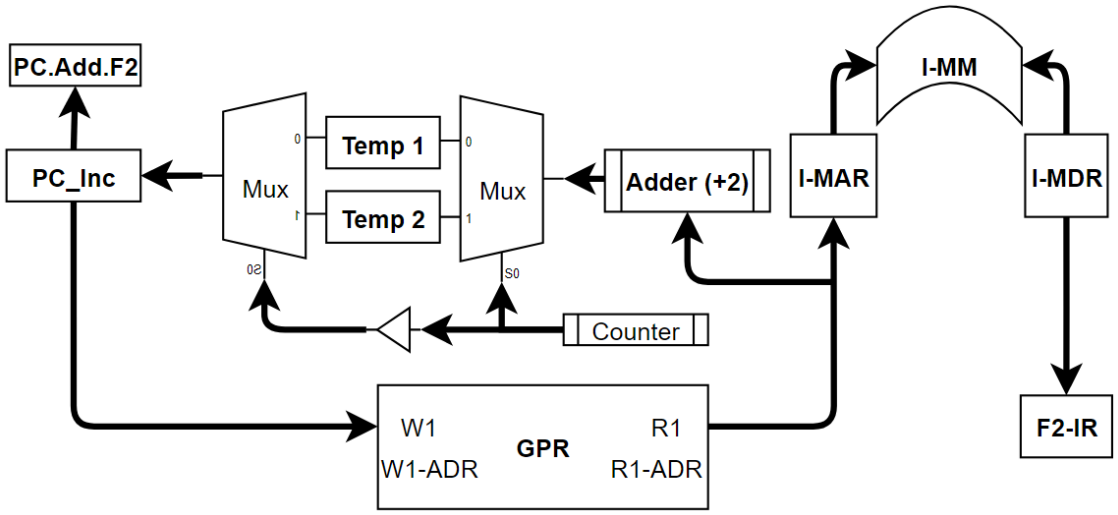

Figure 4-1 Stage 1 and 2 Hardware Implementation

Figure 4-1 above shows the combined hardware implementation for the two fetch stages. The hardware implementation of the stages uses the individual parts of the components listed in the previous sections and the two stages are broken up into their components and individually discussed below.

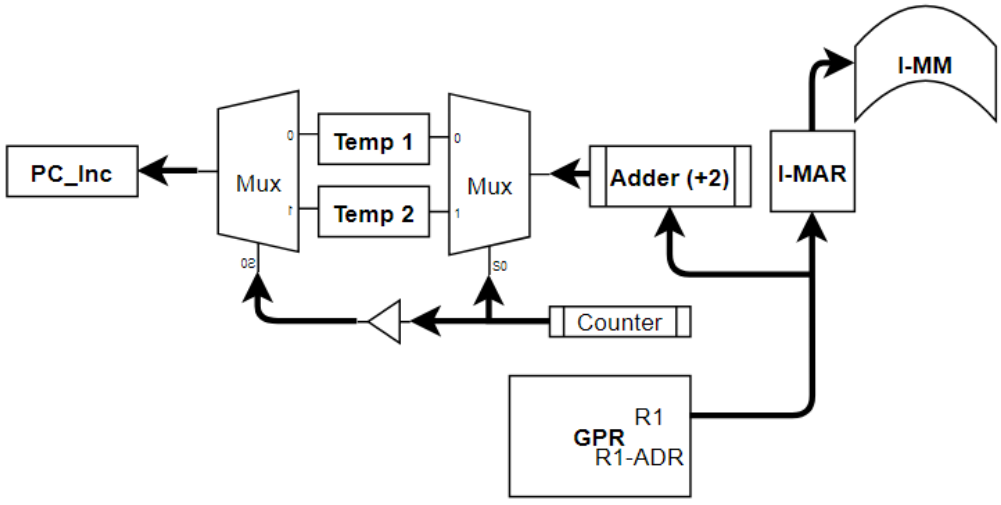

Figure 4- 2 Stage 1 Hardware Implementation

Stage 1 is the first clock cycle of fetch which takes the PC from the first read port of the GPR and feeds it into the Instruction MAR and reads the instruction main memory. At the same time, it also goes through an adder to increment the PC and stores it in the temp registers for insertion into PC_Inc. The adder was added in line with the PC as an optimization to save time for the PC to not have to go to the ALU. The two muxes and temp registers within them are needed for storing alternating incoming PC's. In one clock cycle the adder writes to Temp 1, in the following clock cycle it writes to Temp 2 and in the same cycle the PC in Temp 1 goes to PC_Inc and the operation keeps going. The counter input handles this as the selection bits to the Muxs. This temporary mux buffer prevents the PC_Inc from being overwritten by the following fetch before proper use.

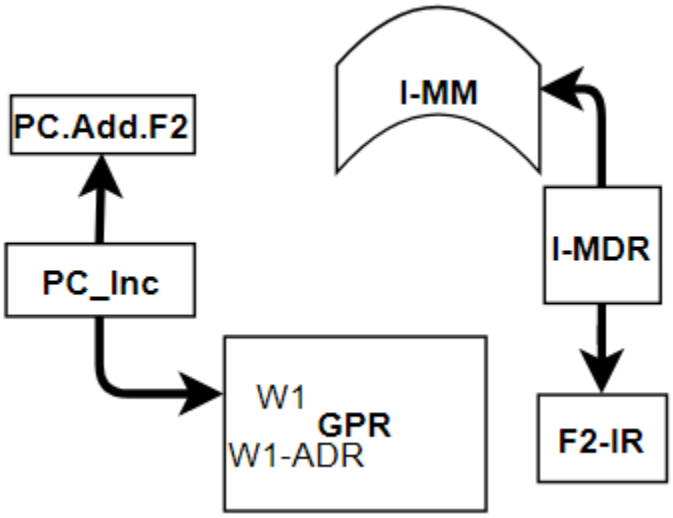

Figure 4- 3 Stage 2 Hardware Implementation

Stage 2 shown above in Figure 4-3 is the second clock cycle of fetch. It writes the incremented PC to the GPR and to a register labeled PC.Add.F2 which gets used in Stage 3 and will be covered in the next section. At the same time, the I-MDR will write to a temporary register for the IR named F2-IR which is used in Stage 3 and 4.

5 Decode Stage 3 and Execute Stage 4

Figure 5-1 below shows the full circuit hardware for the decode, Stage 3 and the execute, Stage 4. The stages are broken apart and further discussed in this section.

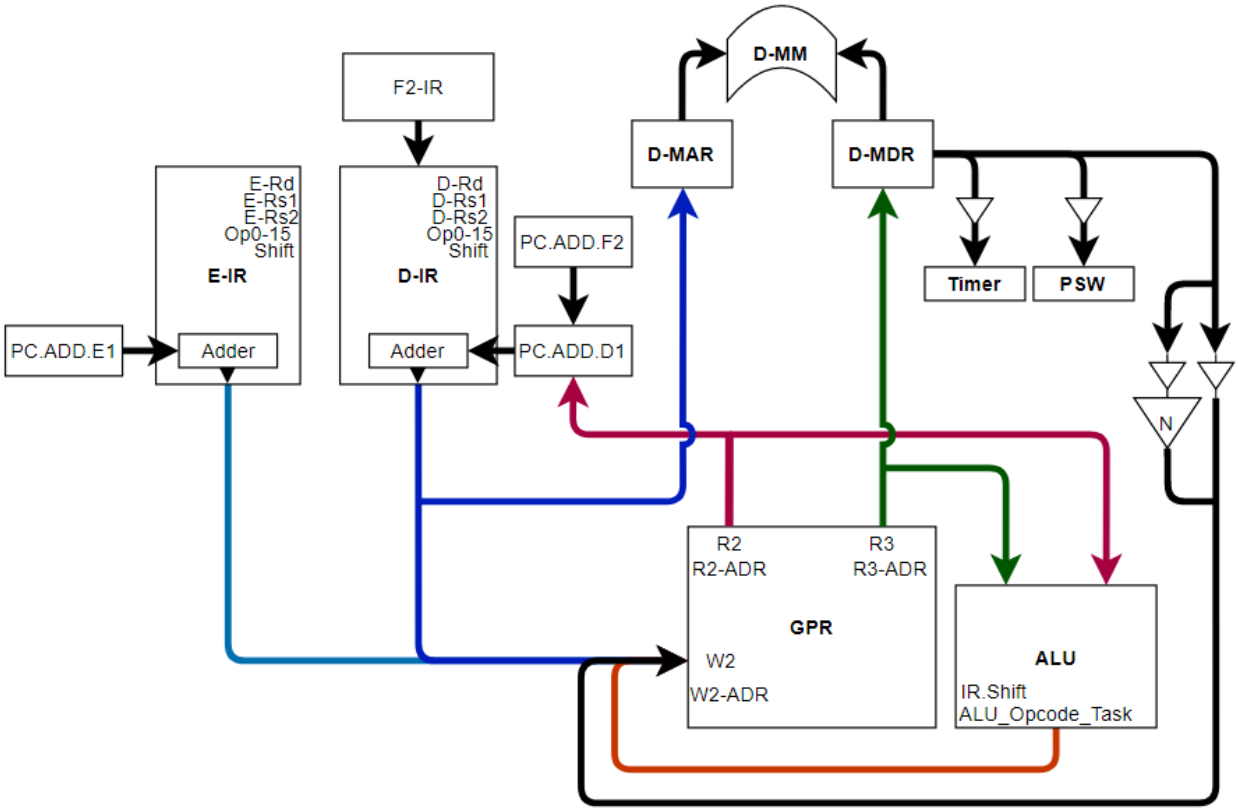

Figure 5- 1 Combined Stage 3 and 4 Hardware Implementation

In order to achieve the optimization of four clock cycles per instruction, the decode and execution hardware need to work together. Due to the simplification of the hardware, there is only one structural hazard that can arise which is when the decode IR attempts to write to W2 of the GPR at the same time as one of the same components from the execute stage. This is overcome with stalling the pipeline by one clock cycle to relieve the overlap of data writing. This is discussed further in the hazards section.

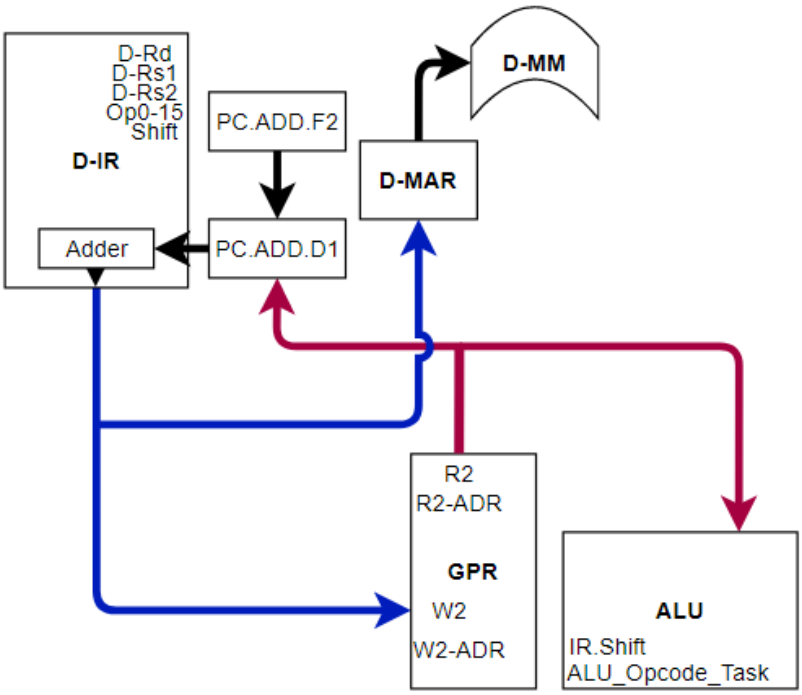

Figure 5- 2 Stage 3 Decode (D1) Hardware Implementation

The decode stage shown to the left in Figure 5-2. shows the decode IR. R2 of the GPR is reserved for read operations exclusively for the decode stage and due to ALU simplifications, GPR-R2 is one of the two only inputs in the ALU. This input goes into the left shifter of the ALU as described in the ALU subsection. In the decode stage, the D-IR is the only module that writes to the D-MAR and GPR per the control signals shown in Table 3-2

The execute stage shown to the right in Figure 5-3 shows the Execute IR. R3 of the GPR is reserved for read operations exclusively for the execute stage and is the only other input to the ALU. In this stage, in any one clock cycle only one of the lines will be high at the same time. This allows for multiple inputs to overlap into the W2 of the GPR. Additionally, on the output of the D-MDR, tristate buffers control input into the timer and PSW. Additionally, an external NOT gate, (labeled N) for the 16-bit data is placed on the output of the MDR and controlled with a tristate buffer before being written to the GPR. There is also a pass command that bypasses this and allows for direct writing from the MDR to the GPR. These are controlled with Pre-GPR-NOT and Pre-GPR-Pass, respectively.

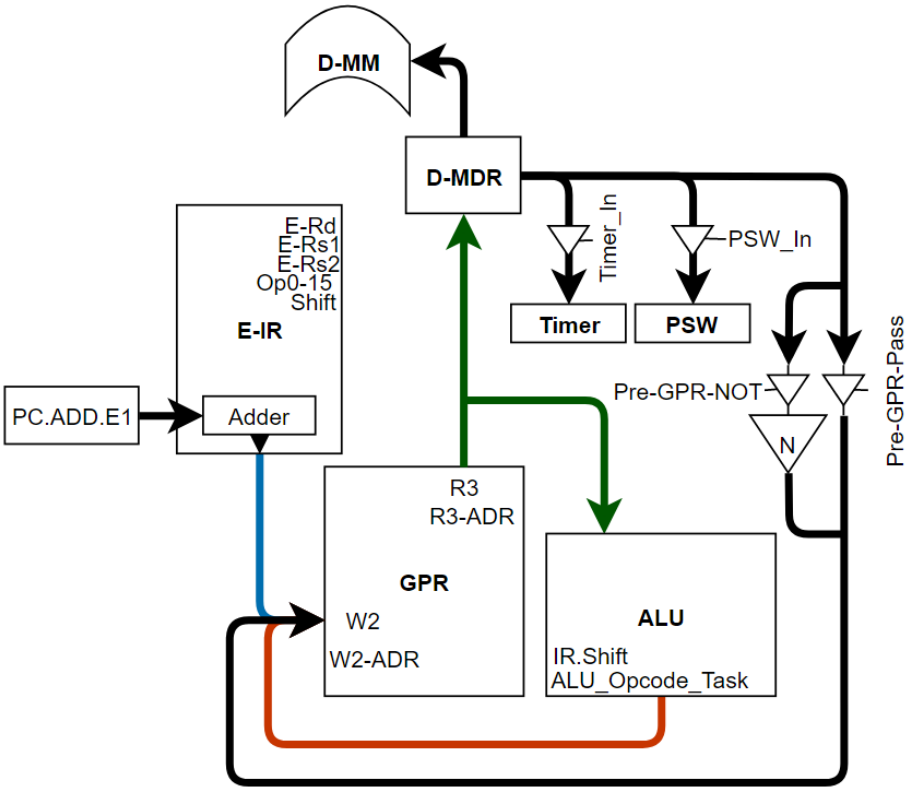

Figure 5- 3 Stage 4 Execute (E1) Hardware Implementation

6 Exceptions

Figure 6-1 below shows the hardware implementation of the two exceptions, Program check and Timeout. These use the same hardware as the decode and execute stages, however, exceptions do not run at the same time as the decode and execute.

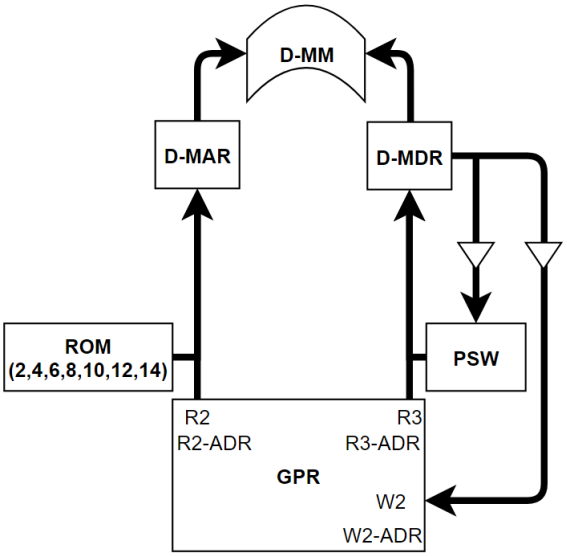

Figure 6- 1 Exceptions/Trap Execution Hardware Implementation

Exceptions are handled by temporarily disabling the pipeline and running sequentially, which use 4 clock cycles to complete for both Program-Check and Time-Out (Table 3-3). However, Time-Out exception is required to be precise, making the machine finish the instructions that has been fetched, then trigger the sequential flow. Thus, as soon as the Timer hits 0, the machine must use two more clock cycles to finish what is in D1 and E1 before commencing the exception sequence, resulting in states XT1-2 before XT3-6 (Table 3-1 and Figure 3-2). In contrast, Program-Check exception does not need to be precise, so no extra states are needed before the 4-step sequence.

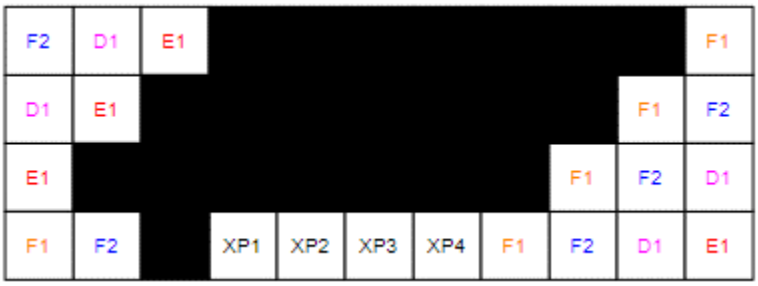

Figure 6- 2 Time-Out Exception Stage Timing Diagram Example

Figure 6-2 shows an example of how the machine would operate to execute a program check trap. It will wait for the currently pipelined instructions to finish executing, then execute the traps and restart normal operation of the machine. Admittedly, there are a lot of clock cycles lost to trap execution, however this allows for substantial optimization of the other stages and since traps are not expected to happen often, this is a worthwhile tradeoff for optimized decode and execute stage performance.

7 Hazards and Solutions

Three types of hazards, structural, data and control type, are analyzed and resolved separately in this section.

7.1 Structural Hazards

Structural hazards arise when the memory units cannot support enough I/O interface needed by the instructions. The main memory units used in this machine where structural hazards arise include the GPR and MM, so they are addressed separately below.

By the design requirements, GPR has 2 write ports and 3 read ports, and the memory can be updated synchronously. We manage the usage of these ports by the following policies:

> **SH#1**. Fetch (F1 and F2) are prioritized over decode and execute by reserving one read and one write port out of the available ports, illustrated by R1 and W1 in Figure 4-1.
>
> **SH#2**. Two read ports are left available for D1 and E1, so each can use one port without issue, illustrated by R2 and R3 in Figure 5-1.
>
> **SH#3**. The only one write port (W2 in Fig 5-2) must be shared by D1 and E1. When they simultaneously use the W2, machine prioritizes E1 over D1 while stalling the machine.

As for MM, there is only one interface via MDR and MAR, which is also updated asynchronously with a time offset of 1 clock cycle. Luckily, the requirement allows for the program and data parts of the memory to be accessed separately and parallelly, letting it avoid many potential structural hazards. Thus, the justification on the MM portion of the structural hazards follows:

> **SH#4**. F1 and F2 only access the program part of the MM (I-MM in Figure 4-1), whereas D1 and E1 only access the data part of the MM (D-MM in Figure 5-1). They do not interfere with each other.
>
> **SH#5**. Only one read command is used in F1 and F2, so there is no hazard in fetch stages.
>
> **SH#6**. Between D1 and E1, read and write must not happen on one clock cycle. When they do, machine prioritizes E1 over D1 while stalling the machine.

To summarize, only SH#3 and SH#6 need to be addressed specially. Furthermore, by comparing the control signals and looking up the corresponding OP-code from Table 3-2, we can effectively describe when exactly SH#3 and SH#6 occur by simply inspecting the OP-code part of the instruction, described in the following pseudo-code.

```
SH#3 = D1.OP == 12 && E1.OP == 9 to 13
SH#6 = ( D1.OP == 6 || D1.OP == 7 || D1.OP == 14 || D1.OP == 15 ) &&
        (E1.OP == 8)
```

This can be implemented by combinational logic on D-IR and E-IR, which are the two instruction blocks included in the queue system.

To prioritize E1 over D1, the machine completes the job on E1, then processes the job on D1 on the next clock, essentially stalling for one clock cycle. Implementation of this has been presented by S/DH state in Table 3-1 and Figure 3-2.

The pipeline was optimized to avoid as many structural hazards as possible, but two hazards remain possible. The Control Unit inspects OP-code part of D-IR and E-IR, and if the logic matches that of SH#3 or SH#6, it transitions the internal state to S/DH, where only E1 stage runs then the job is removed from the queue. This is the solution to structural hazards that effectively stalls for 1 clock cycle. Figure 7-1 shows a sample of this occurring where it is determined that a conflict will arise from D1 of the second instruction and the first instructions E1. To correct this, the machine stalls the D1 execution of the second clock cycle and executes E1 before continuing normal operation. This will in turn make the next few instructions execute in 5 clock cycles instead of the ideal 4 per instruction but it is still a strong optimization for avoiding the few structural hazards of the machine.

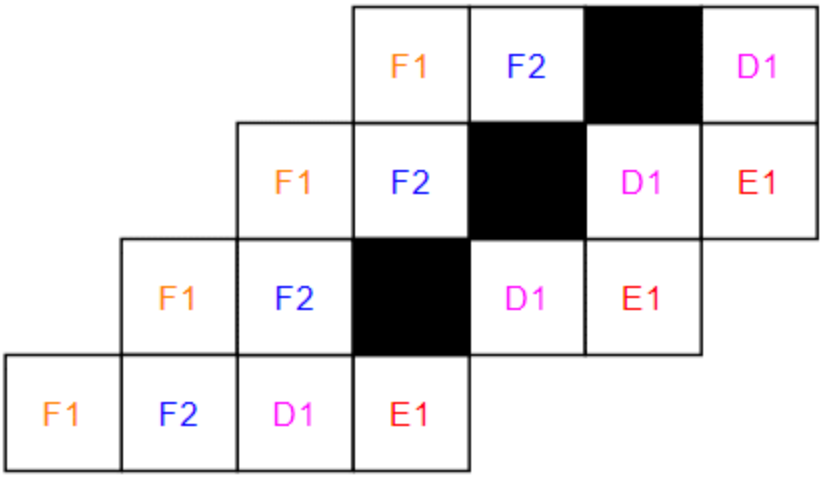

Figure 7- 1 Sample Structural Hazard/Data Hazard Clock Cycle Delay Diagram

7.2 Data Hazards

Data hazards occur when a memory address is involved in multiple instructions such that one is dependent upon the other (data dependencies). Similar to structural hazards, first potential causes of data hazards were analyzed, and a solution is proposed, which is then implemented descripively. The general policies for data hazards are:

> **DH#1**. A data hazard on the MM should also be a structural hazard. It can be addressed in the structural hazards.
>
> **DH#2**. Between D1 and E1, the same address on the GPR should not be read and written simultaneously. In the event it does, prioritize E1 over D1 while stalling the machine.

By inspecting the control signals in Table 3-2, DH#2 is further narrowed down to the following logical expression that can be easily implemented by combinational logic.

```
DH#2 = D1.Rs2 == E1.Rd || D1.Rd == E1.Rd || D1.Rd == E1.Rs1 ||
        ( D1.OP == 12 && E1.OP == 9 to 13 )
```

The solution to data hazards that arise is the same as structural hazard – stalling for one clock cycle. The Control Unit is designed such that within the same state, the overall behavior is the same (which is why the state is called S/DH, short for structural/data-hazards). The same Figure 7-1 above can be referenced for a sample timing diagram of a data hazard occurring and being resolved.

7.3 Control Hazards

Control hazards occur when a branch instruction interrupts the sequential program flow, leading to obsolete fetched and decoded instructions. The common solution to control hazards is a dynamic branch predictor, which predicts if the branch is taken, and alters the PC one stage ahead of the branch execution. However, since the data path design is determined before considering the control hazards, a limitation is placed to only update PC at execution (E1), making it ineffective even if the predictor predicts "taken". Thus, it was decided to use a branch predictor that statically predicts "not taken". Consequently, in the event of conditional branches (OP9 and OP10), PC would still update incrementally, until the branch instruction hits stage E1. If the branch does not happen (correct prediction), the machine moves streamlined without change. If the branch does happen (wrong prediction), the machine flushes the pipeline then starts anew from the branched PC.

To implement this solution, a flag was setup that notifies the Control Unit to transition to a special CH (short for control-hazard) state.

```
CH = E1.OP == 11 || ( E1.OP == 9 && PSW.N == 1) || (E1.OP == 10 && PSW.Z ==1)
```

When the machine enters CH state, following the commands in Table 3-1, it executes the branching, clears all data in the shift register, effectively resets the queue and flushes the pipeline. Figure 7-2 as a Gantt chart gives a visual illustration of this behavior, where (a) shows the case of correct prediction, and (b) shows the case of wrong prediction.

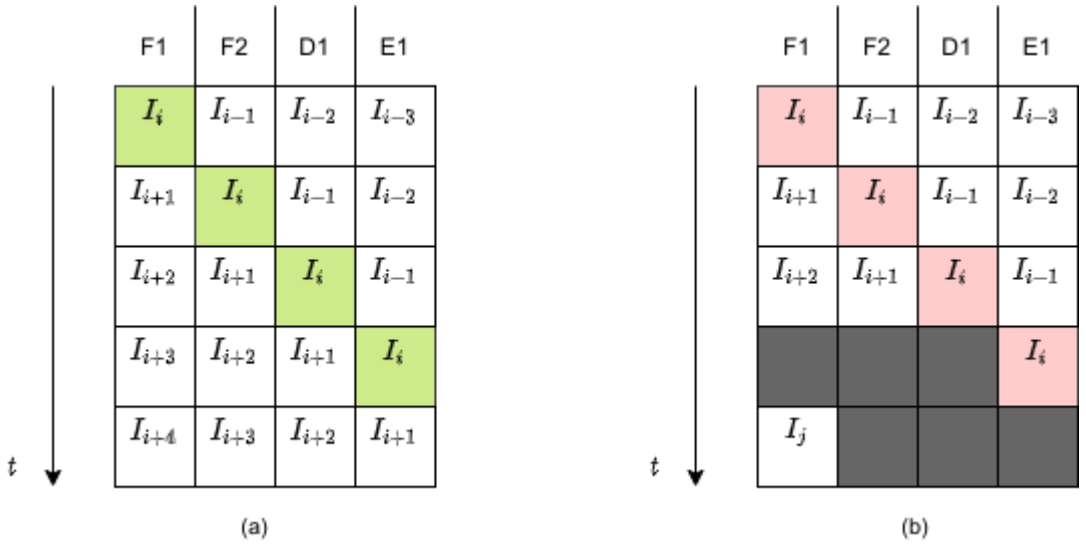

Figure 7-2 Control hazard clock cycle delay diagram. I_i is a branch instruction that is correctly predicted in (a), and incorrectly predicted in (b). The branch jumps to I_j.

8 Optimization Summarization and Appendix

This section is a re-cap of the various optimizations through this project and their locations in the report as well as an appendix showing the full control signals after optimization for each instruction.

Optimization 1, Two IR's and respective registers to hold PCs

In order to ensure that instructions are executed in 4 clock cycles, the IR has been expanded from the previous Project. There are now 2 fully function IRs and a temporary register (F2-IR) that stores the yet to be decoded instruction from the fetch state. (Page 8)

Optimization 2, Adder built into IRs to speed up IR operations

Additionally, to speed up operation, the two fully implemented IR's (D-IR and E-IR) have an adder built into them in order to be able to add the sign extended Short and Long offset to the PC directly inside the IR for instructions from OP codes 6 to 15. (Page 8)

Optimization 3, faster trap state execution due to more ROM values instead of going to the ALU

Whereas in Project 3, the ROM was comprised of a single register that held the value "8", now the ROM is comprised of 7 registers, that hold the static values 2,4,6,8,10,12 and 14 for use within the trap conditions to speed up the operation instead of having to go through the ALU for incrementation. (Page 11)

Optimization 4, Adder in line with the PC storage

At the same time, it also goes through an adder to increment the PC and stores it in the temp registers for insertion into PC_Inc. The adder was added in line with the PC as an optimization to save time for the PC to not have to go to the ALU. The two muxes and temp registers within them are needed for storing alternating incoming PC's. (Page 16)

Optimization 5, External NOT gate for Opcode 6

Additionally, an external NOT gate, (labeled N) for the 16-bit data is placed on the output of the MDR and controlled with a tristate buffer before being written to the GPR. (Page 19)

Optimization 6, Exceptions pause pipeline operation and execute which saves hardware

Exceptions are handled by temporarily disabling the pipeline and running sequentially, which use 4 clock cycles to complete for both Program-Check and Time-Out (Page 20)

ADD Op0 GPR[Rd] = GPR[Rs1] + left shifted(GPR[Rs2],IR.Shift)

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[Rs2]_Out, ALU_R2_In, ALU.Leftshift_IR.shift
4) GPR[Rs1]_Out, ALU_R3_In, ALU_OP_Task, GPR[Rd]_In

SUB Op1 GPR[Rd] = GPR[Rs1] - left shifted(GPR[Rs2],IR.Shift)

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[Rs2]_Out, ALU_R2_In, ALU.Leftshift_IR.shift
4) GPR[Rs1]_Out, ALU_R3_In, ALU_OP_Task, GPR[Rd]_In

AND Op2 GPR[Rd] = GPR[Rs1] and left shifted(GPR[Rs2],IR.Shift)

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In MDR_Out

Decode/Execute)

3) GPR[Rs2]_Out, ALU_R2_In, ALU.Leftshift_IR.shift
4) GPR[Rs1]_Out, ALU_R3_In, ALU_OP_Task, GPR[Rd]_In

SHL Op3 GPR[Rd] = shift left(GPR[Rs1]) by left shifted(GPR[Rs2],IR.Shift)3-0

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[Rs2]_Out, ALU_R2_In, ALU.Leftshift_IR.shift
4) GPR[Rs1]_Out, ALU_R3_In, ALU_OP_Task, GPR[Rd]_In

SHRA Op4 GPR[Rd] = shift right(GPR[Rs1]) by left shifted(GPR[Rs2],IR.Shift)3-0

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[Rs2]_Out, ALU_R2_In, ALU.Leftshift_IR.shift
4) GPR[Rs1]_Out, ALU_R3_In, ALU_OP_Task, GPR[Rd]_In

OR Op5 GPR[Rd] = GPR[Rs1] or left shifted(GPR[Rs2],IR.Shift)

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In MDR_Out

Decode/Execute)

3) GPR[Rs2]_Out, ALU_R2_In, ALU.Leftshift_IR.shift
4) GPR[Rs1]_Out, ALU_R3_In, ALU_OP_Task, GPR[Rd]_In

NOT Op6 GPR[Rd] = not MM[PC + Short Offset]

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) D-IR.Short_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD, | D-IR_ADD_Out_3, D-MAR_In_3, READ_MM
4) D-MDR_Out, Pre_GPR_NOT, GPR[Rd]_In

LD Op7 GPR[Rd] = MM[PC + Short Offset]

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) D-IR.Short_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD, | D-IR_ADD_Out_3, D-MAR_In_3, READ_MM
4) D-MDR_Out, Pre_GPR_PASS, GPR[Rd]_In

ST Op8 MM[PC + Short Offset] = GPR[Rd]

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) D-IR.Short_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD, | D-IR_ADD_Out_3, D-MAR_In_3, WRITE_MM
4) GPR[Rd]_Out, D-MDR_In

BRN Op9 if CC.N then PC = PC + Long Offset

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) NOP
4) E-IR.Long_Off_Out, E-IR_PC.ADD.E1_In, E-IR_ADD, E-IR_ADD_Out_3, GPR[7]_In_3

BRZ Op10 if CC.Z then PC = PC + Long Offset

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) NOP
4) E-IR.Long_Off_Out, E-IR_PC.ADD.E1_In, E-IR_ADD, E-IR_ADD_Out_3, GPR[7]_In_3

BR Op11 PC = PC + Long Offset

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) NOP
4) E-IR.Long_Off_Out, E-IR_PC.ADD.E1_In, E-IR_ADD, E-IR_ADD_Out_3, GPR[7]_In_3

JSR Op12 GPR[Rd] = PC; PC = PC + Short Offset

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[D-Rd]_In_3, GPR[7]_Out_3
4) E-IR.Short_Off_Out, E-IR_PC.ADD.E1_In, E-IR_ADD, E-IR_ADD_Out_4, GPR[7]_In_4

RTS Op13 PC = GPR[Rd] + Short Offset

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[D-Rd]_Out_3, PC.ADD.D1_In_3
4) E-IR.Short_Off_Out, E-IR_PC.ADD.E1_In, E-IR_ADD, E-IR_ADD_Out_4, GPR[7]_In_4

CLK Op14 Set timer to MM[PC + Long Offset]

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) D-IR.Long_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD | D-IR_ADD_Out_3, D-MAR_In_3, READ_MM
4) MDR_Out_4, Timer_In_4

LPSW Op15 PSW = MM[PC + Long Offset]

Fetch)

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) D-IR.Long_Off_Out, D-IR_PC.ADD.D1_In, D-IR_ADD | D-IR_ADD_Out_3, D-MAR_In_3, READ_MMM
4) MDR_Out_4, PSW_In_4

Program Check Violation

(P in PSW isnt set and someone tries to do OP14 or OP15)
MM[0] = PSW
MM[2] = PC
PSW = MM[4]
PC = MM[6]

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) GPR[0]_Out_1, MAR_In_1 | PSW_out_2, MDR_in_2, WRITE_MM,
4) ROM2_Out_1, MAR_In_1, | PC_out_2, MDR_in_2, WRITE_MM,
5) ROM4_Out_1, MAR_In_1, READ_MM | MDR_out_2, PSW_in_2,
6) ROM6_Out_1, MAR_In_1, READ_MM | MDR_out_2, PC_in_2

Timeout

(Timer = 0)
MM[8] = PSW
MM[10] = PC
PSW = MM[12]
PC = MM[14]

1) PC_Out, MAR_In, PC_Inc, READ_MM
2) PC_Inc_Out, PC_In, Y_In |IR_In, MDR_Out

Decode/Execute)

3) ROM8_Out_1, MAR_In_1, | PSW_out_2, MDR_in_2, WRITE_MM,
4) ROM10_Out_1, MAR_In_1, | PC_out_2, MDR_in_2, WRITE_MM,
5) ROM12_Out_1, MAR_In_1, READ_MM | MDR_out_2, PSW_in_2,
6) ROM14_Out_1, MAR_In_1, READ_MM | MDR_out_2, PC_in_2

Figure 8- 1 Optimized Control Signals for Each Instruction and Exception