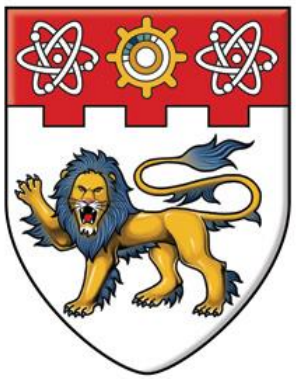


NANYANG TECHNOLOGICAL UNIVERSITY

CE3001: ADVANCED COMPUTER ARCHITECTURE

Final Report

TEAM: Dr.Teeth	DATE: [April 17, 2014]

Name	Matric No
HONG XIAOHUI	U1220523H
JIANG KEWEI	U1220848F
LONG HONGQUAN	U1220764A
TAN LI HAU	U1222847D

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

Contents

1. Brief Description

1.1. Introduction

This project aims to implement a 16-bit CPU.

1.1.1. Phase 1

We handled arithmetic instructions (ADD, SUB, AND, OR, SLL, SRL, SRA,and RL), memory instructions (LW and SW), load immediate instructions (LHB and LLB), control instructions (B, JAL, JR, and EXEC).In the first stage of the project, the processor accesses separate instruction and data memories. The instruction memory and data memory are both 16-bit .The access time for both memories is one cycle.

- ★ Our group have achieved the full specification for phase 1 .We implemented all the expected instructions successfully. All of them run successfully.
- ★ Two test cases including all instructions and data dependency were applied. Corner cases were considered in specific. Consequence was that our code run successfully under the test cases.
- ★ Our group added additional features which improve CPI and CPU performance, such as making decision for Branch instruction in ID stage and reducing CPI to 5 cycles and data forwarding to prevent stalls. They are illustrated in Design& Feature section.

1.1.2. Phase 2

In phase two, the difference from phase 1 is the instructions and data share one memory. This memory has a latency of 3 cycles for read and write. In accessing the memory, instruction fetch and data memory are not allowed to access it concurrently.

- ★ Our group have achieved the full specification for phase 2. All instructions i.e. arithmetic instructions (ADD, SUB, AND, OR, SLL, SRL, SRA,and RL), memory instructions (LW and SW), load immediate instructions (LHB and LLB), control instructions (B, JAL, JR, and EXEC) still works perfectly well.
- ★ Data memory is given priority to access memory when conflict of instruction fetch and data memory occurs. PC counter is stalled to wait for memory read before fetching for instruction.
- ★ Two test cases were applied to code. All instructions and data dependency are covered in these two test cases. Code run as expected under the test cases.
- ★ Additional features are fetching instruction in advance so than instruction will follow one another instead of every three cycle (IF needs three cycles). Details are illustrated in the Design and Feature of the Phase 2 shared memory design section.

1.2. Design & Feature

1.2.1. Pipeline

CPU was implemented based on the 5-stage pipeline design in Interim Report, as in figure 1. However, in order to control the cycles, we changed one single line for Memory.v file in the Interim Report. In the phase 2 design, we overcome this problem by reducing one unnecessary stage register and Memory.v is kept in original. Here only four stage register is used in which EXE and MEM stage are merged together. This is shown in Figure 2.Thus in this design, there are only 5 cycles in total in finishing IF, ID, EX, MEM, WB of one instruction (without structural hazard). This improves (reduces) CPI.

Due to this 4-stage design, we do not need one stall in solving data hazards of LW. Simplified solution is to forward the Data Memory output to ID/EXE pipeline and let the signal go to ALU input in the next cycle. (This will be further discussed in Data Hazard section.)

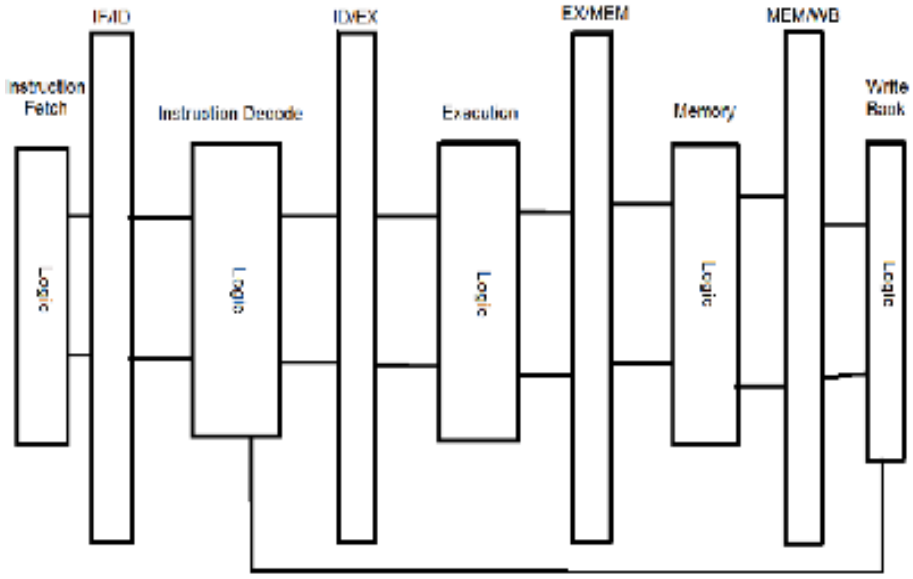


Figure 1

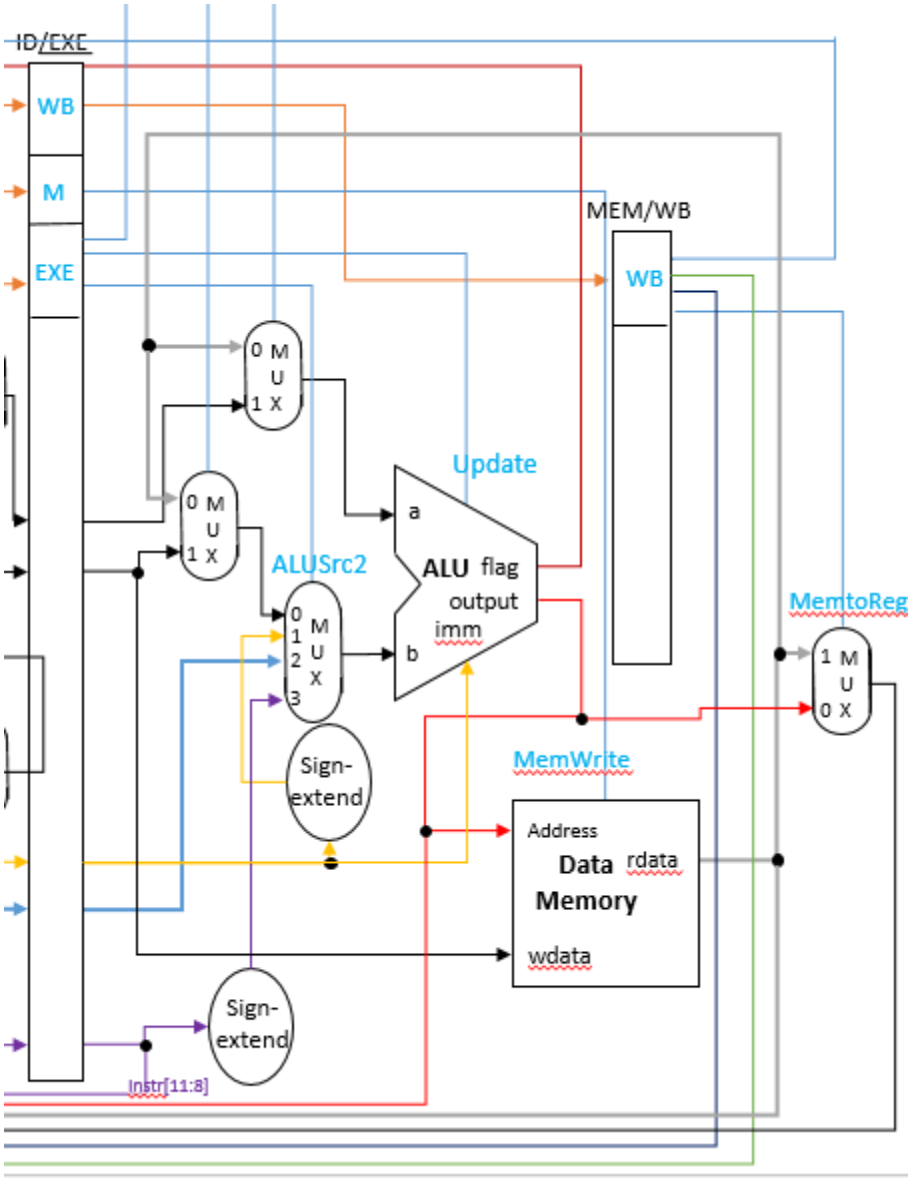


Figure 2

To solve the 1 cycle latency of Data Memory segment, the output of Data Memory, rdata, is directly written back to Register File without passing through a pipeline register. Simultaneously, to maintain the 1 cycle consistency, information other than the data read from Data Memory, traversed through EXE/WB pipeline. These information consisted of RegWrite, R15, and R15_enable. This is shown in Figure 2.

WB data to be written into Register File is only done in WB stage, instead of many WB data flowing from multiple pipeline directing to wdata in Register File. This avoids data dependency and conflicts in writing back to Register File.

1.2.2. Control Unit

Control Unit takes in current instruction and then controls signal input to each CPU component and multiplexers throughout the whole design which selects connection.
In order to match all the control signals with corresponding cycle and stage, we have divided the control signals into 4 different packages and pass them through the pipeline registers correspondingly.

- Decode stage control lines
- Execution/address calculation stage control lines
- Memory access stage control lines
- Write-back stage control lines

A detail list of ports except for clock and reset that design for control.v module is shown below.

Port Name	**Direction**	**Size**	**Description**
opcode	input	4-bit	instruction[15:11]
flag	input	3-bit	Flag information, 2-zero, 1-overflow, 0-negative
exec_in	input	1-bit	Indicates whether the previous instruction is EXEC or not
stall	input	1-bit	Stall signal used for EXEC instruction

Decode stage control lines

regdst1	output	1-bit	Select register address 1
regdst2	output	1-bit	Select register address 2
branch	output	1-bit	B-instruction, depending on the condition code and the flag
jal	output	1-bit	set 1 for JAL
jr	output	1-bit	set 1 for jr
r15enable	output	1-bit	enable R15 write
pc_load	output	1-bit	set 1 for successful control instruction, select PC input

Execution/address calculation stage control lines

aluop	output	1-bit	Operation code for ALU
flag_update	output	1-bit	enable flag update for R-type arithmetic instruction
ALU_sourse_2	output	2-bit	select ALU input source
exe	output	1-bit	used for EXEC instruction

Memory access stage control lines

memread	output	1-bit	Memory read enable
memwrite	output	1-bit	Memory write enable *Active low
memory	output	1-bit	used specifically for Phase 2, set when LW/SW for memory access

Write-back stage control lines

memtoreg	output	1-bit	Memory write to register
regwrite	output	1-bit	Register write enable

For R-type instruction, the CU mainly set Flag_Update, Register Write, ALU operation and corresponding ALU source signal.

For LW and SW instruction, the CU mainly set control signals such as memory read/write, register write, register, resource, memory to registers, memory access signal, as well as ALU operation signal.

For LLB and LHB, the CU mainly set control signals such as register write, register source, ALU source as well as ALU operation signal.

For Branch Instruction, CU decides whether to branch based on the flag signal the last instruction sent to it. ALU output determined flag. Flag signal is set by arithmetic instructions and is stored for instructions other than arithmetic instructions.
The algorithm used is as shown in following code, which is part of Control.v

```
            `BR:begin
//              wire n = flag[0];
//              wire o = flag[1];
//              wire z = flag[2];
                case (cond)
                `EQ : branch = (z)? 1 : 0;
                `NE : branch = (z)? 0 : 1;
                `GT : branch = ((z || n))? 0 : 1;
                `LT : branch = (n)? 1 : 0;
                `GEQ: branch = (z|| !(z || n))? 1 : 0;
                `LEQ: branch = (z || n)? 1 : 0;
                `O  : branch = (o)? 1 : 0;
                `T  : branch = 1;
                default: branch = 0;
            endcase
```

Figure 3

For other instructions such as JAL, JR, and EXEC, it usually set there corresponding signals as well as pc_load signal. For exam, JAL set jal signal and JR set jr signal.
As EXEC is a special case, we will discuss about it later.

1.2.3. Branch in decode stage

One of our most significant change of our design is that we moved all the control instructions performance into decode stage.

For instructions such as Branch and JAL, instead of using the ALU in EXE stage, we added a simple ADDER to calculate the desired location that might be loaded as PC input.

For EXEC and JR, as the desired location can be accessed in the same cycle by looking at register data out, it is even easier to perform jump in this stage instead of waiting for a cycle in EXE stage.

The benefits of this modification are significant.

- It reduced the cycles needed for completing a control instruction. Now we are able to perform branch in Decode Stage which only consumes 2 cycles.
- Simplified the implementation of flush significantly when encounter control hazard. This will be discussed in later part.
- Reduce the overhead work of ALU in Execute Stage, which may be able to focus specifically on other types of instructions. Also removed the complexity of ALU source mux.

1.2.4. ALU

For the instructions including R-type Instructions, LW, SW, LHB, LLB, the calculations are all done be one single ALU. No further adder is implemented here. The ALU is in the Ex stage. This reduces CPU components and time cost. Thus the design will improve CPU efficiency.
The ALU input is controlled by one 3-1 multiplexes. The implementation is shown as below.

OPCODE	ALUSrc1	ALUSrc2

R-TYPE	Register data out 1, Rs	0 - (Register data out 2)
LW	Register data out 1, Rs	1 - (Instruction[3:0], offset)
SW	Register data out 1, Rs	1 - (Instruction[3:0], offset)
LHB	Register data out 1, Rs	2 - (Instruction[7:0], imm)
LLB	Register data out 1, Rs	2 - (Instruction[7:0], imm)

Here should note that in our Interim Report, there are two mux which select both ALU source 1 and ALU source 2. This is due to the fact that we have moved the branch instructions all from EXEC stage into Decode stage.

1.2.5. FLAG Registers

In order to decide the conditional branch outcome, we implemented FLAG registers. There are three bits in the FLAG register: Zero (Z), Overflow (V), and Sign bit (N). Only ADD, SUB, AND, OR instructions can change the FLAG register.

The Z flag is set if and only if the output of the operation is zero.

The V flag is set by the ADD and SUB instructions if and only if the operation results in an overflow. Overflow must be set based on treating the arithmetic values as 16-bit signed integers. The AND and OR instructions always clear the V flag.

The N flag is set if and only if the result of the ADD and SUB instruction is negative. Note that N flag should be implemented as (sign-bit XOR overflow). The AND and OR instructions always clear the N flag.

Other Instructions, including shift/rotate instructions, load/store instructions and control instructions, do not change the contents of the FLAG register.

Store of Flag Register

In our phase 1 design, we directly connect ALU flag output with Control Unit. This is based on the assumption that the branch instruction is always after an R-type arithmetic instruction such as ADD, SUB, AND, OR. This may be true for theoretically when compiler strictly follows the rule.

However, this may not be true when come to the real situation in current trend of compilation. Due to Register Renaming or other possible instruction rescheduling which improves CPU performance, there might be the case that branch is following an instruction such as SLL or LHB etc. Thus, we changed our design which enable us to store the flag in a register which is accessed by Control Unit.

We created 3 registers to store the flag in the Control Unit, and it is updated only when the ALUOP is arithmetic instruction. Therefore, even though the ALU operation that can change the flag register is executed a few instruction before the Branch instruction, the zero, negative and overflow flag is stored in the Control Unit and is used to evaluate whether the branch to be taken or not.

1.2.6. EXEC instruction design

EXEC instruction is regarded as the most challenging one among all the instructions. In our interim report we failed to implement this instruction. Now we are able to perform EXEC instruction perfectly.

Basically, we view EXEC Rd instruction as two consecutive steps.

- First stage is similar to execute JR Rd, we load the jump address from Rd and perform jump in the similar way as JR. R15 is set to be PC+1 in the same stage.
- Then we jump back to original PC+1 by loading R15 into PC.

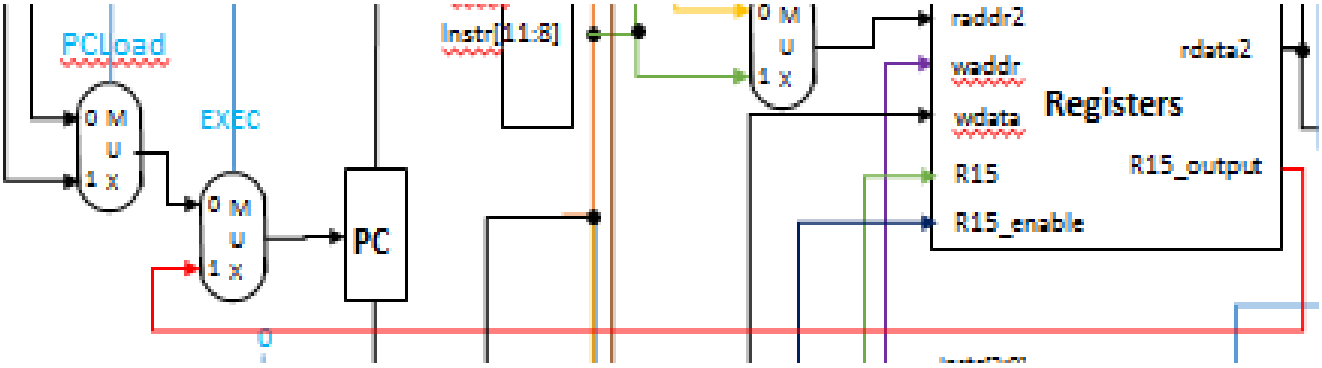


Figure 4

As shown in the figure, we just load R15 when EXEC signal is high. In order to make sure that the R15 is correctly loaded exactly one cycle after the executing of desire instruction, we delay the EXEC selection signal by one cycle by passing it through the ID/EXE register.

Special Corner Case

We also considered about the special situation that the EXEC targeted instruction itself cannot be a control instruction. In order to achieve this, we also pass the EXEC signal to Control Unit as a input from ID/EXE. Thus, for control instruction in decode stage, it will refer to the EXEC input signal to see whether the previous instruction is EXEC or not.

```
`JR:begin
    jr = 1;
    pc_load = !exec_in;
    regdst2 = 1;
end
```

Figure 5

This figure is one example code which shows how we enable and disable a pc load. When the exec_in is 1, the pc will not be changed even it is unconditional jump JR.

1.3. Hazard

Hazard consists of Data Hazard, Control Hazard and Structural Hazard. We considered Data Hazard and Control Hazard in phase 1 implementation, and Structural Hazard for phase 2 where shared main memory with 3 cycle delay is used. In our design, a hazard unit is used for detecting and handling all hazard.

1.3.1. Data Hazard

Data Hazard rises when the next instruction needs data information from the previous instruction.

Instruction in EXE	Instruction in ID Stage (Next Inst)	Condition where Data Hazard rises	Consider solution	Comment

ADD, SUB, AND, OR,SLL, SRL, SRA, RL,LHB, LLB	ADD, SUB, AND, OR, SW	ALU out==ID_Rs||ALU out==ID_Rt	Data forwarding	
ADD, SUB, AND, OR,SLL, SRL, SRA, RL,LHB, LLB	SLL, SRL, SRA, RL, LW, LHB, LLB	ALU out==ID_Rs	Data forwarding	
LW	ADD, SUB, AND, OR, SW	Mem out ==ID_Rs||Mem out ==ID_Rt	Data forwarding	However, in 5-stage pipeline, one more stall is needed. Thus using 4-stage pipeline saves one stall.
LW	SLL, SRL, SRA, RL, LW, LHB, LLB	Mem out ==ID_Rs	Data forwarding	However, in 5-stage pipeline, one more stall is needed. Thus using 4-stage pipeline saves one stall.

Instruction in MEM	Instruction in ID Stage (Second Next Inst)	Condition where Data Hazard rises	Consider solution	Comment
LW	ADD, SUB, AND, OR, SW	Mem out ==ID_Rs||Mem out ==ID_Rt	Data forwarding	
LW	SLL, SRL, SRA, RL, LW, LHB, LLB	Mem out ==ID_Rs	Data forwarding	

Solution - Data forwarding and stalling

Instruction in EXE	Instruction in ID Stage (Next Inst)	Forward Rd From	To

ADD, SUB, AND, OR,SLL, SRL, SRA, RL,LHB, LLB,	ADD, SUB, AND, OR,SLL, SRL, RA, RL,LHB, LLB SW,LW	ALU out	Register out
LW	ADD, SUB, AND, OR,SLL, SRL, RA, RL,LHB, LLB SW,LW	Mem out	ALU in

Instruction in MEM	Instruction in ID Stage (Second Next Inst)	Forward Rd From	To
LW	ADD, SUB, AND, OR,SLL, SRL, RA, RL,LHB, LLB SW,LW	Mem out	Register out

There are only three ways which covers all the above data forwarding situations.

- ALU out -> Register out
- Mem out -> Register out
- Mem out -> ALU in

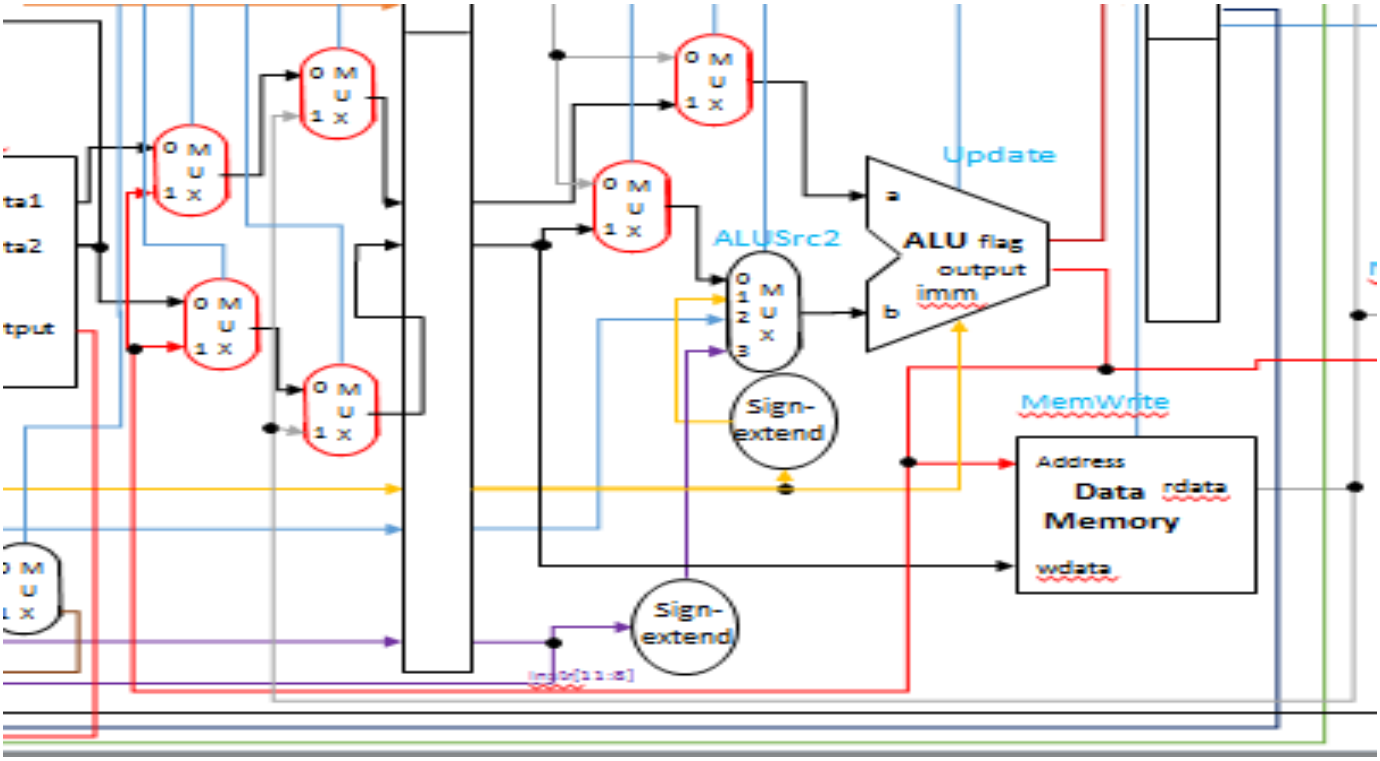


Figure 6

As shown in the graph, the 6 red mux are used to select the forwarding data, which are controlled by signals from Hazard Unit.
Among those, only one situation need stall, which is LW instruction followed by EXEC/JR instruction which consists of data dependency concern. Only in this case, the EXEC/JR instruction is stalled for one cycle waiting for the output of LW from memory out.
For the situations that LW followed by other instructions which need ALU, we directly forward data to ALU source as input instead of forwarding to decode stage. Thus, one cycle is saved by not stalling.

1.3.2 Control Hazard.

Control Hazard rises for conditional and unconditional jumps, subroutine calls and other program control instructions because of a delay in availability of an instruction. In this project, Control Hazard happens for Instruction B, JAL, JR, or EXEC instruction. If previous instruction is a branch and the branch is taken, current instruction should be stalled and should not be decoded again. If previous instruction is JAL, JR, or EXEC

Instruction in Decode Stage	Condition where Control Hazard rises	Consider solution
B	Branch taken	Instruction in IF/ID Stage is flushed.
JAL, JR, EXEC	Null	Instruction in IF/ID Stage is flushed.

Solution - Flush
Due to our special design, all the control instructions is taken in the decode stage, thus, we only need to flush one instruction which waste only one cycle in IF/ID register. Here we just simply use a flush mux where input 0 is original pc out and input 1 is a NOP operation. The flush signal is set to high if the previous instruction is a successful branch, JAL, JR or EXEC.

```
//flush instruction, from hazard unit or from rst
mul2_1_16bit instuction_flush(
        .in0(ifid_instruction_before_flush),
        .in1(stall_instruction),
        .select(flush_instruction || rst || stallreg),
        .out(ifid_instruction));
```

Figure 7

The reason why we are able to achieve flush by simply using a mux is only because just one instruction is flushed due to our design. Thus, no extra implementations such as TAGs are needed which increases the complexity of our design.

1.4. Phase 2 shared memory design

In phase 2, instruction memory and data memory is replaced by a single shared memory, which has a latency of 3 cycles for read and write. Because of having one shared memory, the instruction fetch pipeline stage and memory read/write pipeline stage cannot access the shared memory concurrently, therefore arise the structure hazard that need to be resolved. Therefore, a memory-top module is written to encapsulate the shared memory and the logic that controls the selection between instruction address and data memory address to be the address to access the memory.

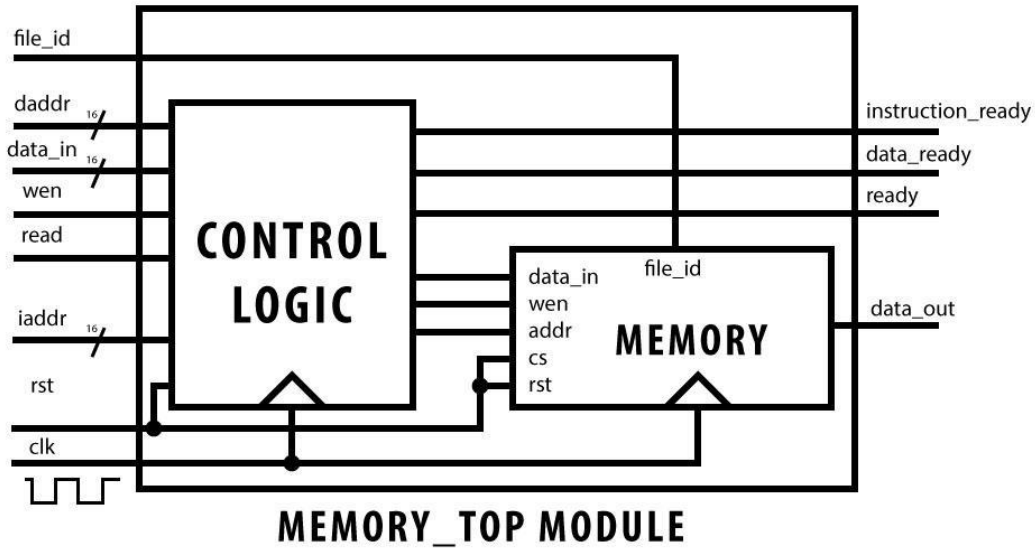


Figure 8

1.4.1. Design and Features

The following shows the port listing of the memory_top module:

```
module memory_top #(parameter ASIZE=16, DSIZE=16, LATENCY=3)
                   (input clk, rst,
                   input [ASIZE-1:0] iaddr, //instruction address
                   input [ASIZE-1:0] daddr, //data address
                   input read, //active high signal when LW
                   input wen,  //active low signal when SW
                   input [DSIZE-1:0] data_in,  //data input
                   input [3:0] file_id,
                   output [DSIZE-1:0] data_out,    //data output
                   output reg instruction_ready,   //active high when instruction is ready
                   output reg data_ready,  //active high when data is ready
                   output reg ready //active high when either data or instruction is ready
                   );
```

Figure 9

The output of the program counter is the instruction to be fetched and therefore is connected to the *iaddr*, and the output of the alu, *aluout* is connected to the *daddr*, as the address of the memory. Active low write enable signal (for instruction SW), active high read signal (for instruction LW), and data to be stored (for instruction SW), is connected to *wen*, *read*, and *data_in* respectively.

The *instruction_ready* flag is set to high when the data output at *data_out* is correspond to the instruction address at the input *iaddr*, similarly, the *data_ready* flag is set to high when the data output at *data_out* is

correspond to the data address at the input *daddr*. The *ready* flag is set high whenever the *data_ready* or *instruction_ready* is set to high.

Therefore, we used the *instruction_ready* to select the *data_out* to be used in the instruction fetch pipeline stage and *data_ready* to select the *data_out* for the memory pipeline stage. And, the *instruction_ready*, *data_ready*, and *ready* signal is connected to Hazard Unit to control the flow of the pipeline and stall the pipeline if needed.

The following figures summarize the connection of the inputs and outputs of the *memory_top* module.

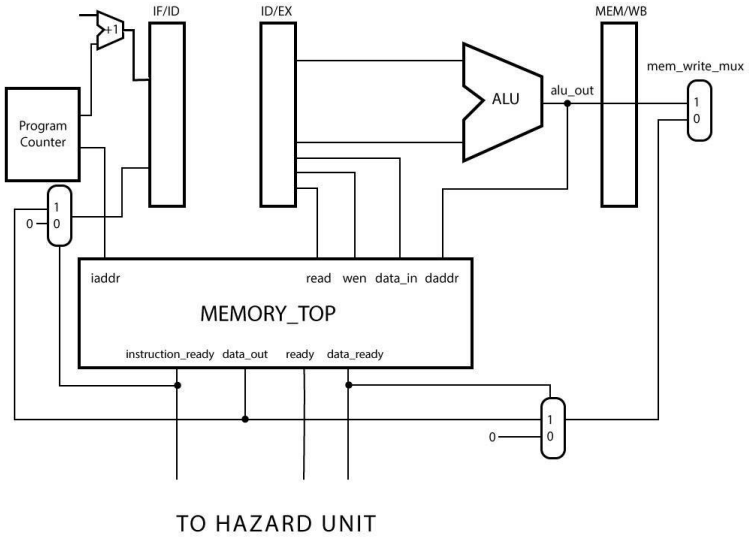


Figure 10

1.4.2 Structure Hazard

Structure hazard is one of the problem with the instruction pipeline of the processor. Structure hazard occur when two instructions need the same hardware resource at the same time.
In phase 2, the instruction memory and the data memory is combined into one unified shared memory and therefore it is possible that the memory pipeline stage and the instruction fetch pipeline stage access the memory concurrently. And thus arise the structure hazard, where a part of the processor's hardware (the shared memory), is needed by two or more instructions (Instruction Fetch and Store Word/Load Word), at the same time.

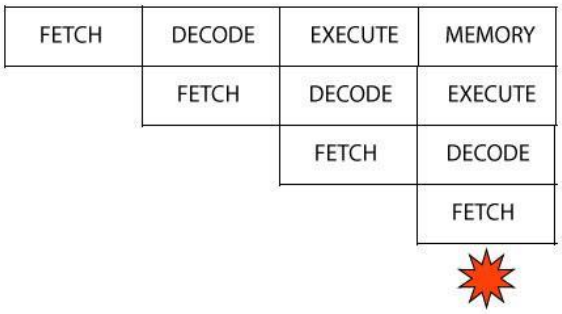


Figure 11

The diagram above shows that structure hazard arise in the pipeline when using a unified shared memory. Memory pipeline stage and instruction fetch stage occurs concurrently, however the memory can only be accessed by only one of the instruction, therefore, there is a need to resolve the structure hazard.

1.4.3 Solution

To avoid the structure hazard, we stall one of the instruction until the memory is available. In our design, we decided that data memory read, e.g. the instruction LW (Load Word), is prioritized. The reasoning behind is that if we are able to get the instruction from the memory, we still need to stall the pipeline to wait for the memory pipeline stage to access the memory. Therefore, if both data address and instruction address arrive at the same time, the data address is passed to *memory* first, and then the instruction address. And if the instruction has come out first from the memory data pipeline, we will ignore the instruction coming out from the memory *data_out* port (the *instruction_ready* flag and *data_ready* flag remains low).

This is done by having a register, *read_reg*, which will be set high when the memory starts to fetch data, and set low when the data comes out from the memory, as a flag to show that the memory is fetching data and *instruction_ready* flag will not be set high while the *read_reg* flag is high.

```
if(read) begin
    read_reg <= 1;
    ...
end

if(data_ready)  read_reg = 0;
```

Code snippet of the *read_reg* register
Figure 12

1.4.4 Performance

The new memory module has a latency of 3 cycles for read and write. Therefore, this significantly reduce the performance of our CPU by increasing the CPI (Cycles Per Instruction) by 2, i.e. 3 cycles for instruction fetch instead of 1 cycle, and if the instruction is a memory type of instruction, e.g. LW or SW, the CPI increase again by additional 2 cycle.

Therefore, besides having logic to select between *iaddr* and *daddr* in *memory_top*, we also write logic that tries to improve the performance of the memory module. We will discuss the methods taken in the following briefly and will show the results in the performance of the steps taken in the Test Case section later in the report.

1.4.4.1 Fetch instruction ahead

As it is observed by the locality principle, especially the spatial locality, and we observed that the instruction is fetched sequentially, we do not wait 3 cycles each for the instruction fetch. Instead, we try to fetch instructions stored next to the current instruction in the memory while waiting for the current instruction to be fetched. In the next cycle after fetching the current instruction, and no branch or jump is taken, the instruction address of the next instruction is set to the next memory address of the current instruction, and the instruction should be ready at *data_out*, and thus avoid of waiting for the latency of 3

cycles.

Therefore, if the consecutive instruction is arithmetic instruction or load immediate instruction, the performance of the CPU is improved by reducing the CPI as we have reduce the latency in memory read.

1.4.4.2 Data Memory Write Burst

Besides having instruction normally stored in consecutive memory address, we also observed that unless the next instruction is LW, the data stored into memory SW does not have any data dependencies to it's following instruction. Therefore, we do not need to stall or flush the memory pipe for SW immediately, SW is scheduled into the data pipe of the memory, and will be written into the memory after 3 cycles. And thus it will take only 1 cycle of latency for each SW instruction in the pipeline's perspective instead of 3. This can be clearly seen in the Phase 2 Test Case section.

2. Methodology

2.1. Design Methodology

2.1.1. Fully functional pipelined CPU without hazard

For phase 1, the design of the project was divided into 6 steps, and each step was completed before going on to the next step. Firstly, we designed a fully functional pipelined CPU without hazard detection. The implementing steps are conducted into the following methodology in the order:

1). Create modules that are the components of the pipelined CPU
2). Test individual modules for functionality
3). Piece all of the individual modules together
4). Test complete CPU with the instruction program written by ourselves
5). Add support and test for one instruction at a time
6). Create program to demonstrate functionality of CPU

Our pipeline implementation of the CPU was subdivided into modules which were created before being pieced together. Before this project, we had created many basic modules like ALU and Register File in the previous labs which were used in the single cycle CPU. Because of this reason, we decided to start from the basic single cycle design and then create the control unit for the datapath in the pipeline registers. Each instructions in these different pipeline stages. After that, test values were then put through the modules by specialized pipeline register had to be created and created the datapath in the control unit to deal with the memory to test their functionality.

After all of the modules were completed, we created the Phase1_top level for the CPU module which is used to join all of the individual modules together with logic part. This is the most difficult step because we need to add a large number of extra wires and registers in bits widths. The external clock which the test bench provided had to control these separate modules at the correct time correctly in the hardware and Verilog. In the CPU, all of pipeline registers were clocked so that they would write on the positive edge of the clock and the register file would write on the both positive and negative edge. The final datapath diagram is shown below.

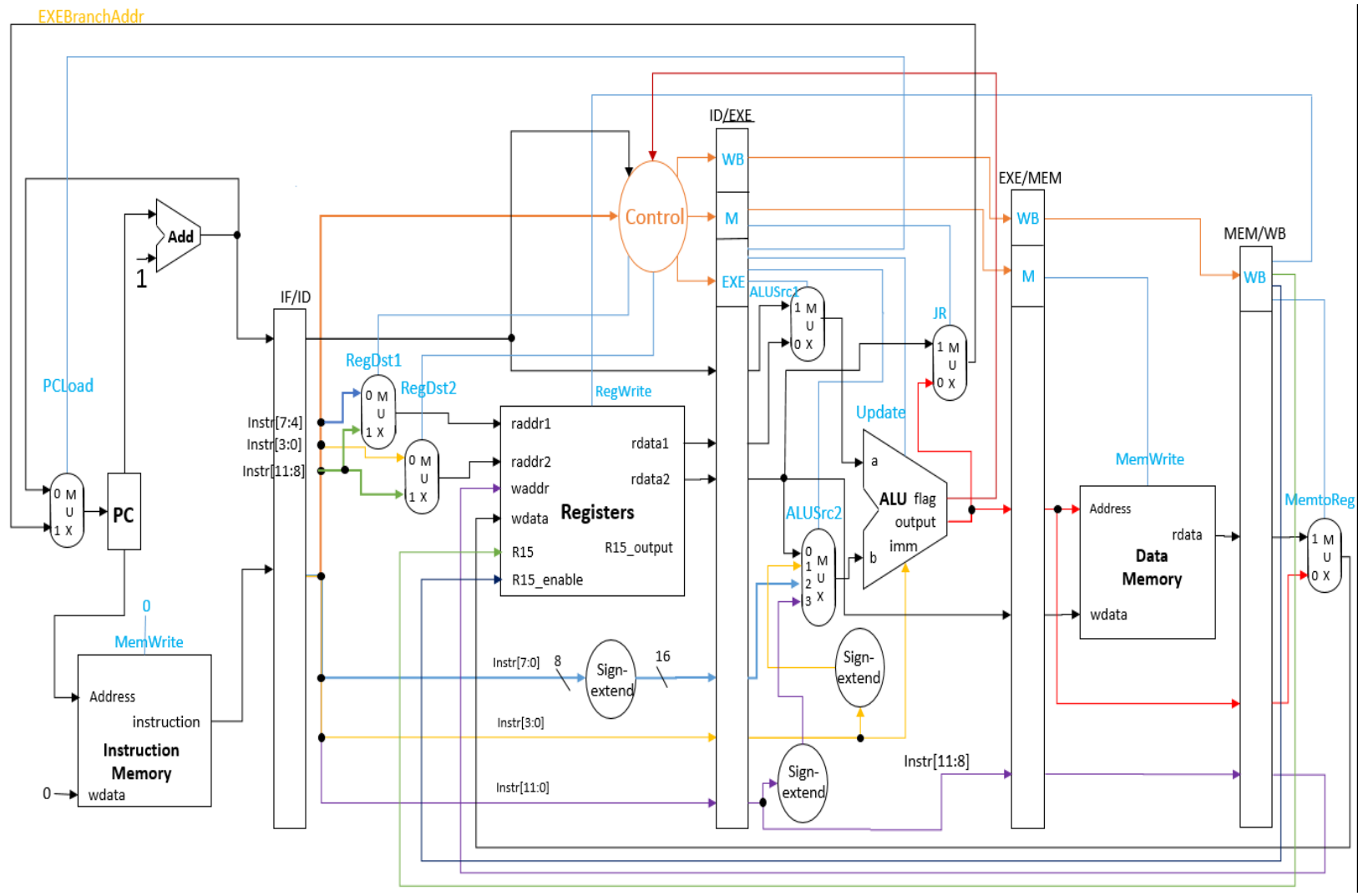


Figure 13

Once all the modules were packaged together and debugged already, we ran the instruction program which were written by ourselves as a test bench to debug the CPU when it is working.

2.1.2. Fix error and improve performance

After successful implementing of first step, we then look forward to detect errors (Hazard Detection), handle errors. That is, after we got a fully functional pipelining CPU, we start to design a Hazard Unit which detect and handle hazard.

We initially design the hazard types separately.

Firstly, we design the detection of Data hazard, and provide the solution by data forwarding and stall. Our initial design contains a Data Forwarding Unit which is specifically focusing on handling data forwarding. However, our final design combined Data Forwarding Unit into Hazard Unit, which reduced the complexity of the design.

Then we designed the detection of Control hazard, followed by providing the solution of stalling. **It is just**

at this step that we realized our initial design is inefficient. Here we redesigned our CPU by moving all the control instruction into decode stage.
In the same time, we also found and designed some ways to improve our CPU performance.

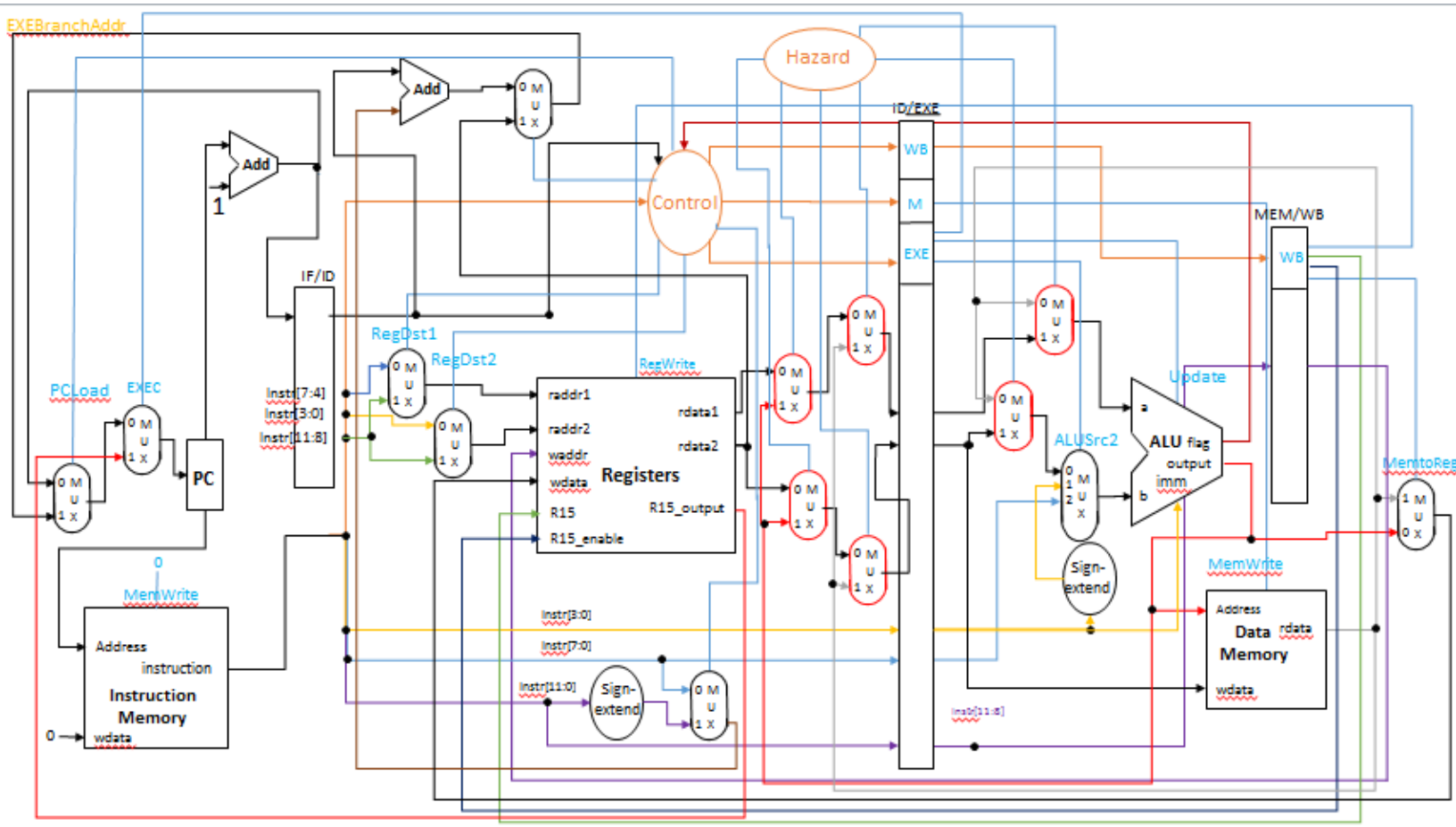


Figure 14
This is the new diagram which covers all the modifications for Phase 1

2.1.3. Proceed to phase 2

For design consideration for shared memory, based on the existing modules created in Phase 1 and the new memory module, we redesigned our CPU in as a whole and we divided them into steps for implementing.
We firstly consider to implement a CPU with shared memory which can freely work for all R-type instructions as well as the LLB and LHB.
Then we add the control instructions, which requires stall of PC and flush of instructions.
Finally we designed the approach of handling structural hazard.
In the same time, we also considered to implement a cache, and we made small progresses which will be shown later.

2.2. Testing Methodology

We employed an iterative and incremental testing methodology to fully test our program. The idea behind this methodology is that the system should be developed for repeated time (iterative) and in smaller portions at a time (incremental).We started with a simple implementation. More requirements were taken into considerations later, including pipeline, hazard detection and data forwarding, and extra features. We iteratively enhanced the evolving versions. At each iteration, design modifications are made, new functional capabilities are added, and test cases are created.

After we finish the implementation, we employed comprehensive test cases. In our consideration, in order to guarantee the CPU with fully functional operations, we decided to design 4 test cases to test our CPU to test all of the instructions and some special cases. In the 4 test cases, two of them were used to test phase 1 and the other two were implemented for phase 2.

2.2.1. Phase 1

2.2.1.1 Basic Integrated Test

In the first test case of phase 1, we used a test bench with integrated instructions that were written by ourselves to test our design which contains all of the instructions such as ADD, SUB, AND, OR, SLL, LW, SW, B and EXEC which are used to test our multi-cycle implementations for this project.
The test bench loaded some values from the memory and used them to test all of our instructions. Some of these instructions were dependent on previous instructions, but some weren't. For those instruction including data dependencies, the Hazard Detection Unit would detect the data hazard and execute the data forwarding to forward the data through the multiple cycles. For instance, SUB R5, R0, R1 and AND R6, R5, R3. In this example, R5 is used as destination register in the SUB instruction but also used as source register in the AND instruction that would cause the data dependency but the CPU would execute the data forwarding to handle this kind of problems. In our CPU, each instruction was stored into different registers so that we could just run the whole program and look at the end result to decide if we ended up with the correct values.

2.2.1.2 Advanced Fibonacci Number Test

For the second test case, we used the Fibonacci Number Calculation written by ourselves to test the functionalities of our CPU. This is a very complex test bench which contains **intensive** test of Branch, JAL, JR and EXEC. The reason of using the Fibonacci Number Calculation as the test case is that it is meaningful not just to integrate all the instructions in the one case to test our CPU. In the Fibonacci test case, we considered almost all of conditions which could happen in the operation of CPU including basic arithmetic instructions, LW, SW, LHB, LLB, B, JAL, JR and EXEC. In addition, the control hazard and data hazard were also in the consideration in the test case. In this test case, we also considered about some corner conditions, for example, after the EXEC instruction, the B instruction could not be implemented and the PC would be plus one to implement the next instruction.

2.2.2. Phase 2

2.2.2.1 Integrated Fibonacci Number Test

For the test 1 of the phase 2, we used the same Fibonacci Number Calculation test case as the phase 1 to test whether we can get the exact same results as the test case 2 of the phase 1. In the phase 2, we combined the instruction memory and data memory into one shared memory. The reason of using the same Fibonacci case is that it would be easy for us to figure out whether our CPU with the shared memory are able to handle all the instruction and operate the hazard detection successfully, especially the PC value would be stored after the B instruction executed.

2.2.2.2 Intensive Memory Load/Store Test

When we considered the test case 2 of the phase 2, we decided to write a test case to make an intensive memory test about LW and SW with the shared memory. In the test case, the structural hazard could be handled which was caused by the shared memory. In the addition, the data hazard of the LW and SW could be detected and the CPU could implement the data forwarding to handle it.

Based on the four test cases, we could implement all the functionalities into our CPU and handle the hazard detection and data forwarding with the separated memory and the shared memory. If we could get the results we wanted, we could conclude that our design is successful.

3. Current status and Progress of project

3.1. Achieved in Phase 1 (Interim Report)

- ✓ We are able to implement 5-stage pipeline and it runs almost perfectly under ideal pipeline.
- ✓ Instructions ADD, SUB, AND, OR, SLL, SRL, SRA, RL ,LW SW ,LHB LLB, B, JAL and JR can be compiled and run successfully.
- ✓ We can access and write to the memory, register, pc (for branch).
- ✓ The control unit is fully functional currently.
- ✓ Regardless of data dependency, we are free to write any test bench instructions by ourselves and it seems to work fine under ideal pipeline.

3.2. Achieved in Phase 2 (Final Report)

- ✓ Consolidated and upgraded all the functionalities we already got in our Interim Report. Our CPU can freely run very complicated test benches.
- ✓ Instruction EXEC runs perfectly, even can perfectly handle with corner case that the target instruction is control instruction.
- ✓ In our Interim report, Code in memory.v has been edited for one single line. Now we are able to use the original version which is with one cycle delay in nature, and the result matches perfectly. The maximum cycle keeps as 5 cycle including write back to register.
- ✓ Hazard Detection is perfectly implemented. They include data hazard, control hazards (phase 1) and structural hazard (phase 2). We are also able to handle all the hazard by data forwarding, stall, and instruction flush.
- ✓ We have improved the performance of our CPU by saving CPI, by having data forwarding instead of stalling the pipeline.
- ✓ For the shared memory in Phase 2, we also implement in the way such that we do not need to stall 3 cycles for every stage. We only need to stall in the case of LW/SW and control instruction.

3.3. Work Ahead (Future Upgrade)

We have designed a Cache Module which may significantly improve the performance. However, due to the time constrain we are not able to implement it out. We will discuss about our design in the discussion part.

4. Simulation and testing

4.1. Phase 1 Test case

Below is the test case we designed, such that there is no data dependency between each instruction, so that there will be no hazards, since our Hazard Detection module is yet to be implemented.

4.1.1. Basic Integrated Test Case

```
PCAddr MachineCode     Instruction
0000   8101    //       LW  R1, R0, 1 # R1 = 1
0001   8202    //       LW  R2, R0, 2 # R2 = 2
0002   8303    //       LW  R3, R0, 3 # R3 = 3
0003   8802    //       LW  R8, R0, 2 # R8 = 2. R8 is used as loop count
0004   0412    //       ADD R4, R1, R2 # R4 = 3
0005   1501    //       SUB     R5, R0, R1 # R5 = FFFF
0006   2653    //      *AND     R6, R5, R3  # R6 = 3
0007   4552    //       SLL     R5, R5, 2 # R5 = FFFC
0008   8A00    //L1:    LW  R10, R0, 0 # R10 = [0]
0009   0AA2    //      *ADD     R10, R10, R2 # R10 += 2;
000a   9A00    //      *SW  R10, R0, 0 #[0] += 2
000b   1881    //       SUB R8, R8, R1 # R8--
000c   C1FB    //       B   001, L1 #branch if not equal
000d   8A00    //       LW  R10, R0, 0 # R10 = 4
000e   0AA3    //      *ADD R10, R10, R3 # R10 = 7
000f   FA00    //       EXEC    R10 # R5 = FFF0
0010   9505    //      *SW  R5, R0, 5
```

Figure 15

Initial Memory

Address	Value (hex)
0000	0000
0001	0001
0002	0002
0003	0003
0004	0009
0005	0001

✓ Noted that in this test case, the * denotes that there is data dependency where the destination register of previous instruction is used as the resource register of the next instruction. In this test case, the following aspects can be tested.

- ✓ Correctness of all the instructions, R-type, LW, SW, Branch, EXEC (Other instructions such as LLB,LHB,JR and JAL is demonstrated later)
- ✓ Cycles taken for each instruction
- ✓ Data hazard detection and handling (Data Forwarding)
- ✓ Control hazard detection and handling (Instruction Flush)
- ✓ A corner case where Data dependency happens for EXEC.

Tracing the program in advance, the following result should be observed as our expectation.

```
R1 = 1
R2 = 2
R3 = 3
R4 = 3
R5 = FFFF -> FFFC -> FFF0
R6 = 3
R7 = 0
R8 = 2 -> 1 -> 0
R9 = 0
R10 = 0 -> 2 -> 2 -> 4 -> 4 -> 7

MEM[0] = 2 -> 4
MEM[5] = FFF0
```

Figure 16

Here is the simulation result picture taken from ModelSim.

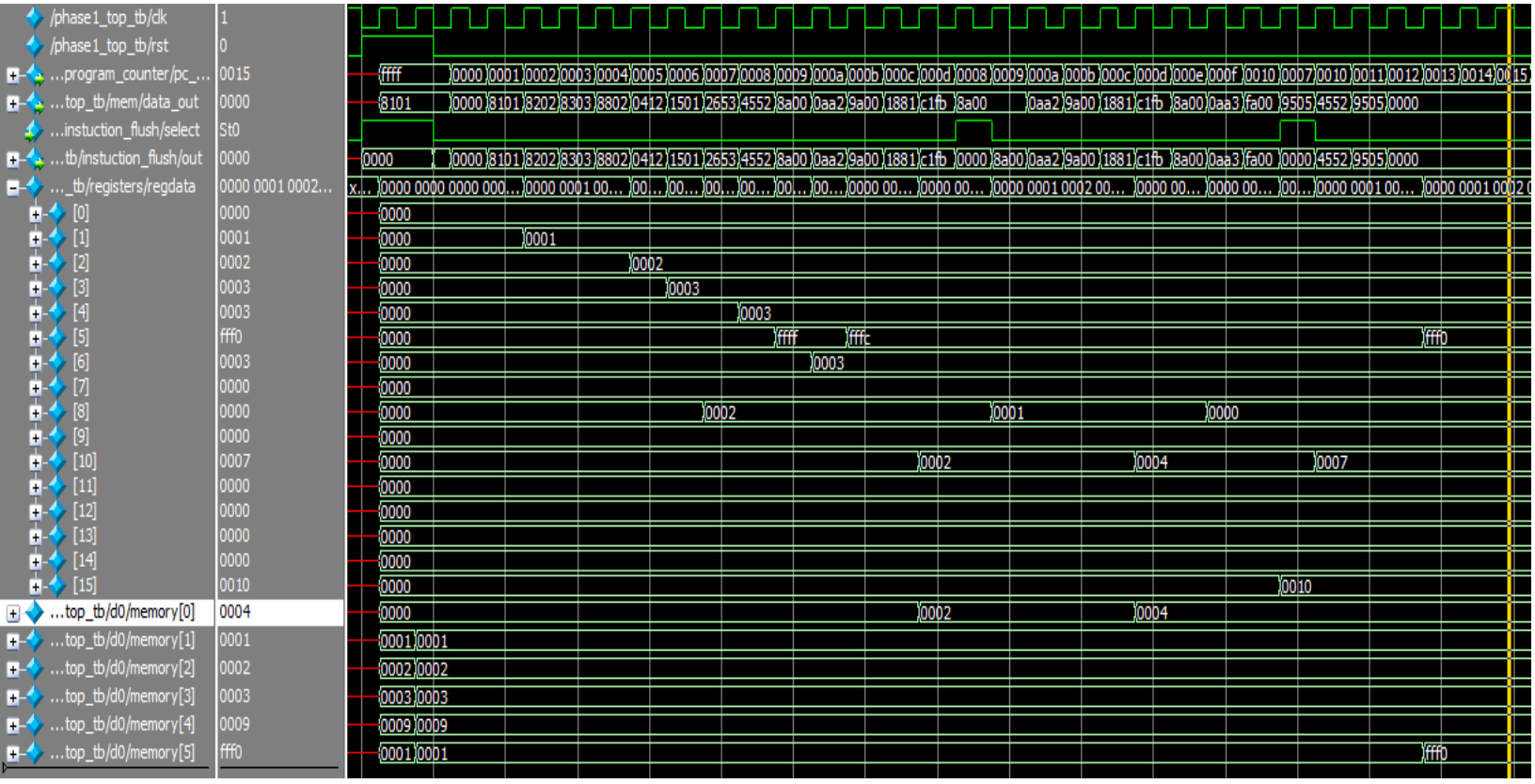


Figure 17

- In the figure above, we can see that ending value and change process of registers and memory is exactly the same as expected. For example, ending value of register[10] = 7, ending value of memory[0] = 4;
- By counting the cycles we can get control instruction

- The data dependency problem does is successfully handled, for exam, intruction 0x0006 AND R6, R5, R3 uses the destination register of the previous instruction as a source. If there is no data forwarding, the outcome would be 0x0000.
- When can also see from the figure when pc is 0x000c, it is a successful branch instruction where it jumps to 0x0008. We can get that only one instruction is fetched after the 0x000c, which is 0x000d. However, if we look down at the flush select and flush out, we will find that the flush signal is set at that cycle and the 0x000d instruction is flushed into NOP. Right after 0x000d, the desired pc is loaded which is 0x0008.
- When looking at the instruction 0x000f, which is an EXEC instruction with data hazard. Firstly we found that the data hazard is handled, moreover, the pc jumps to the desired target address - 0x0007, and gets back to executing 0x0010 which is original pc+1.

4.1.2. Advanced Fibonacci Number Test

```
//# test program for stage 2: Fibonacci Number Calculation
//# ----- INSTRUCTION 0-15: INITIALIZE ----- #
0000  B80B   //   LLB      R8, 11
0001  A800   //   LHB      R8, 0              # R8 = 11 is the input index
0002  B901   //   LLB      R9, 1              # R9 = 1
0003  BA10   //   LLB      R10, 0x10          # R10 = 0x10 is the memory location
0004  1880   //   SUB      R8, R8, R0         # CHECK IF R8 == 0
0005  C008   //   B        000, L0  # EARLY EXIT
0006  1189   //   SUB      R1, R8, R9         # CHECK IF R8 == 1
0007  C006   //   B        000, L0  # EARLY EXIT
0008  1889   //   SUB      R8, R8, R9         # R8 = R8 - 1
0009  B400   //   LLB      R4, 0              # R4 IS THE DESIRED FIBONACCI NUMBER
000a  B200   //   LLB      R2, 0              # R2 IS THE ZEROTH NUMBER
000b  B301   //   LLB      R3, 1              # R3 IS THE FIRST NUMBER
000c  94A0   //   SW       R4, R10, 0         # [0x10] = 0
000d  C702   //   B        111, L1
000e  98A0   //L0:         SW       R8, R10, 0         # [0x10] = R8
000f  C709   //   B        111, L3  # L3 IS FINISH
//# ----- INSTRUCTION 16-19: LOOPING COMPUTATION ------ #
0010  D00B   //L1:         JAL      L2                          # JUMP TO SUBROUTINE L2
0011  1889   //   SUB      R8, R8, R9         # R8=R8-1
0012  C1FD   //   B        001, L1            # IF R8!=0, CONTINUE
0013  7000   //   RL       R0, R0, 0          # NOP
//# ----- INSTRUCTION 20-27: FINISH ------ #
0014  8EA0   //   LW       R14, R10, 0        # should GIVE R14 = 89
0015  BC0D   //   LLB      R12, 0D
0016  BB1C   //   LLB      R11, 1C
0017  FB00   //   EXEC     R11                # should GIVE R4 = 144
0018  FC00   //   EXEC     R12                # should do nothing because 000d is a branch instruction
0019  C7FF   //L3:         B        111, L3  # jump to self: infinite loop
001a  7000   //   RL       R0, R0, 0          # NOP
001b  7000   //   RL       R0, R0, 0          # NOP
//# ----- INSTRUCTION 28-32 : SUBROUTINE ----- #
001c  0423   //L2:         ADD      R4, R2, R3         # R4 = R2+R3
001d  94A0   //   SW       R4, R10, 0
001e  0203   //   ADD      R2, R0, R3         # NEXT R2 = CURRENT R3
001f  83A0   //   LW       R3, R10, 0         # NEXT R3 = CURRENT R4
0020  EF00   //   JR       R15                # RETURN
```

Figure 18

- ✓ Here we use a relatively complex test bench, which is to calculate Fibonacci Sequence. In this test case, intensive control logic is implemented and tested, which proves that our design is able to handle more intensive situation.
- ✓ In addition to the first test case, instructions such as LLB, LHB, JR and JAL is tested and used frequently, especially in the L1 which contains a subroutine of L3 by making use of JAL and JR R15.
- ✓ A corner case where EXEC instruction leads to a target instruction which is unconditional jump.(0x0018)
- ✓ A corner case of correctly load and write to R15.

Tracing the program in advance, this Fibonacci Sequence will loop for 10 times and give a desired Fibonacci number of 144(0x0090) in register 4.
Here is the simulation result picture taken from ModelSim.

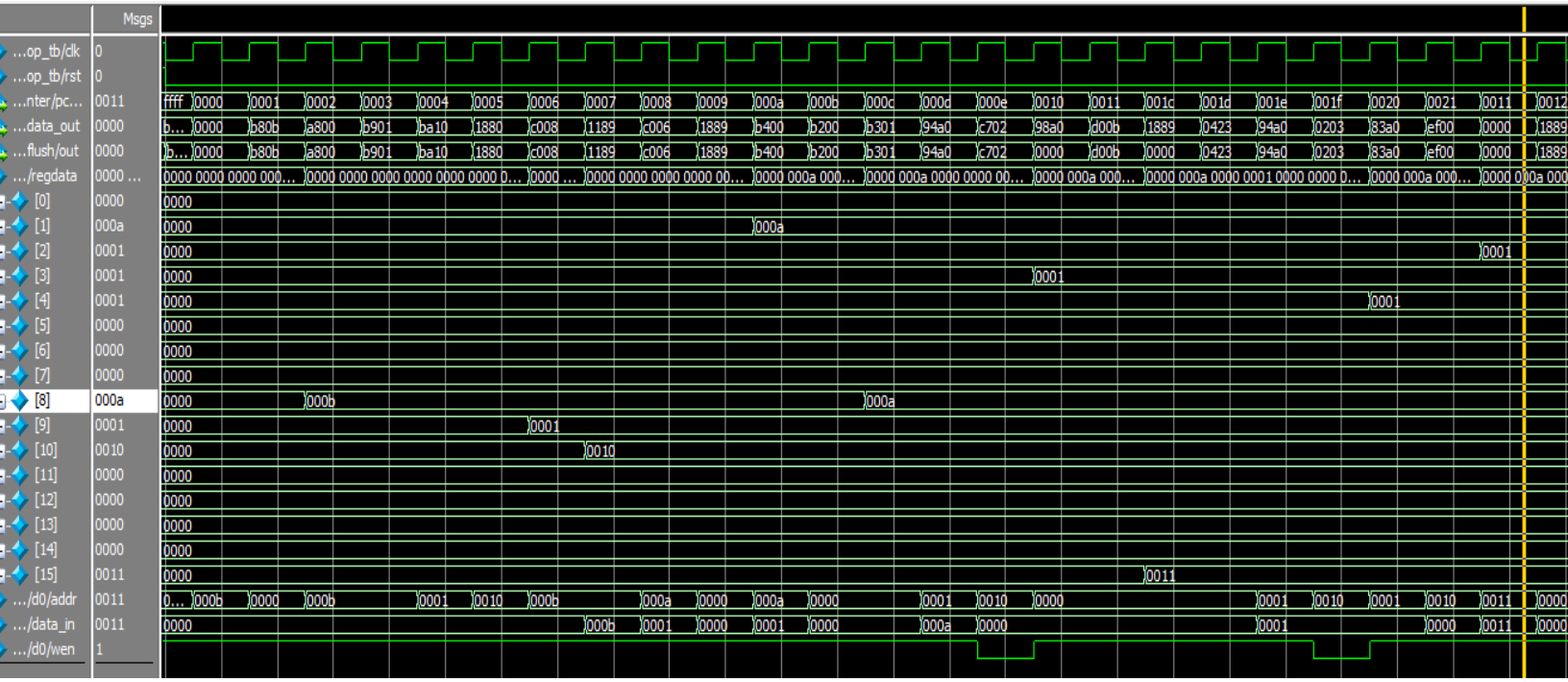


Figure 19

- The figure above shows the **initial part** of the test case. The instruction flow is from 0x0000->0x000d->unconditional branch to L2->0x0010->Jump And Link to subroutine L3->0x001c->0x0020->Using register R15 to jump back to 0x0011-> Keep looping…...
- By observing the program counter out, we found the control logic is perfectly correct. No matter for branch, JAL and JR.
- The initial data is correctly loaded into registers and memories. Where initial value for R2 is 1, initial value for R3 is 0, loop counter R8 is set to be 000a before looping.
- In the same cycle when pc out is 0020, we can see that the first desired result is correctly calculated for R4, where 0001 = 0001+0000.
- In the same cycle when pc out is 001c, we can see that R15 is loaded to be 0x0011. This is because of the instruction 0x0010 JAL. Thus we get that pc + 1 is correctly loaded into R15.
- It should also be noted that R15 is written only for JAL instruction here (also for EXEC but not shown here), for branch and JR, the R15 is not set.

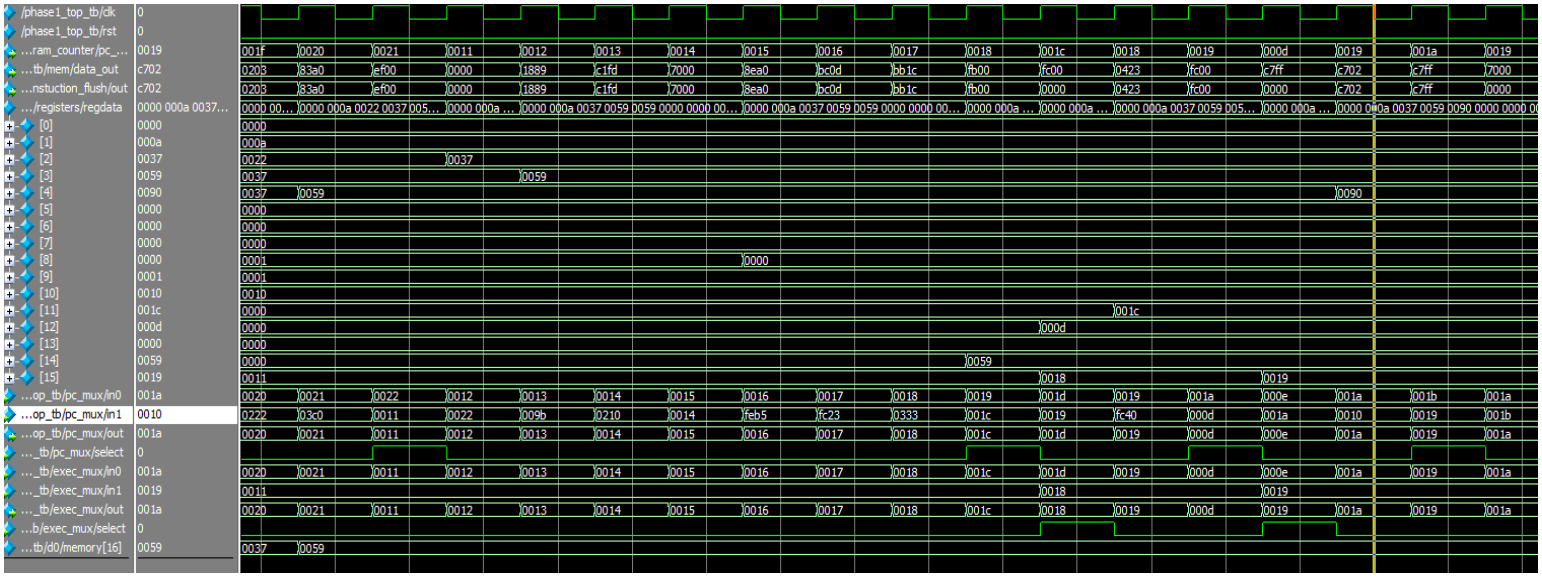


Figure 20

- The figure above shows the **end part** of the test case. Here as the register [8] = 0x0000, L2 will break and it will go to instruction 0x0014-0x0018 for FINISH. Then it will go to an infinite loop at instruction 0x0019 where is always jump to itself.
- Here the answer is correctly loaded into each register, as well as the memory [16] which ends with 0x0059.
- Here we focus on **one important corner case** when the target instruction derived from EXEC instruction is an unconditional branch instruction. We refer from the instruction

  0018 FC00 // EXEC R12

  As R12 stores 000d, it firstly jump to 0x000d

  000d C702 // B 111, L1

  As this is an unconditional jump, if we just look at this instruction, it will jump to L1 which is 0x0010. Here we look at the bottom of the graph (indicated by the yellow line), we can observe that even though the px_mux input 1 is 0x0010, the select, which is pc_load, is disabled. That is why the original pc+1(0x0019) is loaded into pc instead of 0x0010.

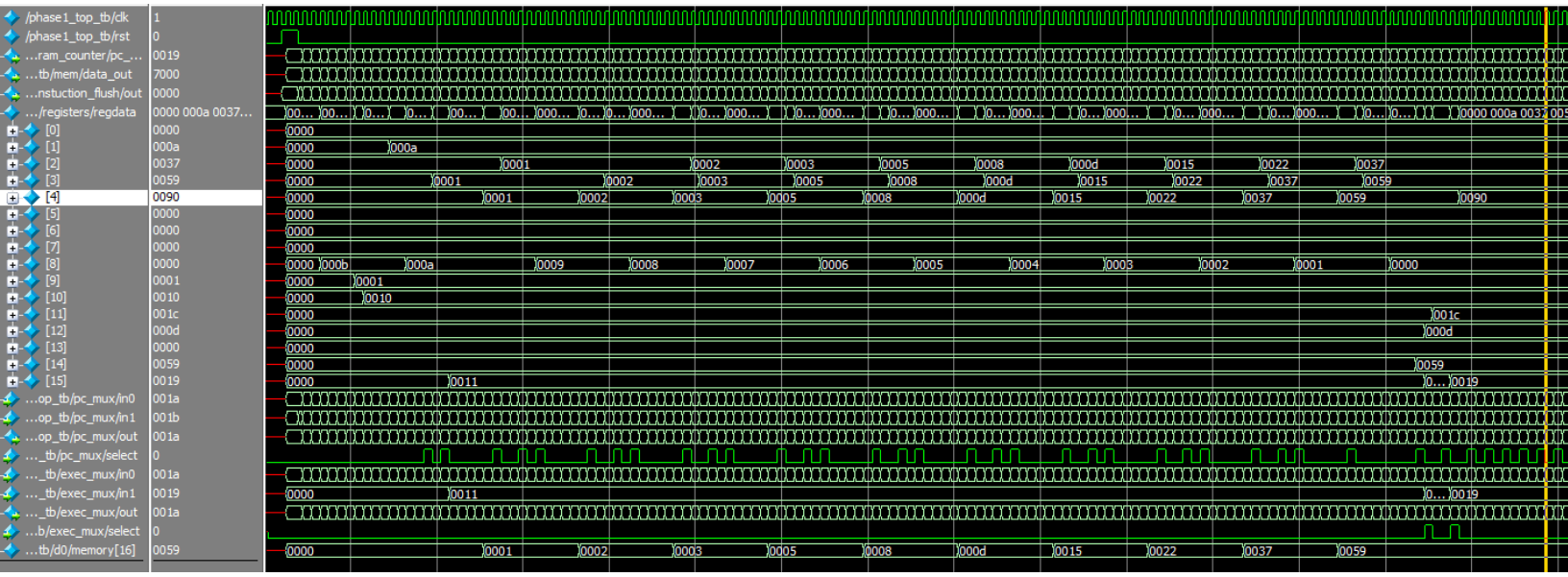


Figure 21

- This shows the overall result of the test program. We can see that in each loop, the result is correctly calculated, and the final result is 0x0090, which is 144 in decimal.
- This figure is later used to compare with the outcome of Phase 2 test.

4.2. Phase 2 Test case

4.2.1. Advanced Fibonacci Number Test

As the first test case for phase 2, we used the same test bench as the Fibonacci Number test of phase 1. It tests all the instructions other than LW and SW.

- By conducting this test, we mainly test on the correctness of our phase 2 design by comparing it with Phase 1 result.
- We will see how the program counter is stalled and how instructions are flushed when having a successful control instruction.

Here are the screenshots we took from ModelSim which probes our expectation

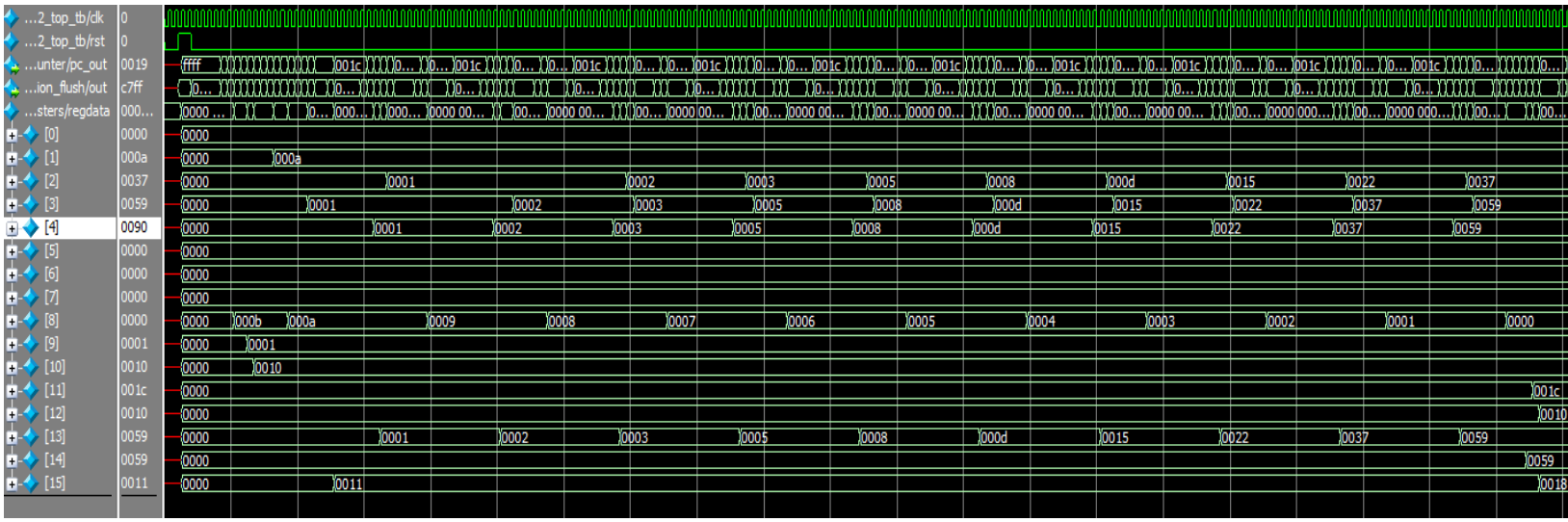


Figure 22

- From the figure above, by comparing it with the outcome we get from phase one test case two, we get exactly the same result. And excluding the number of cycles, we found that the process from start to end is also exactly the same.

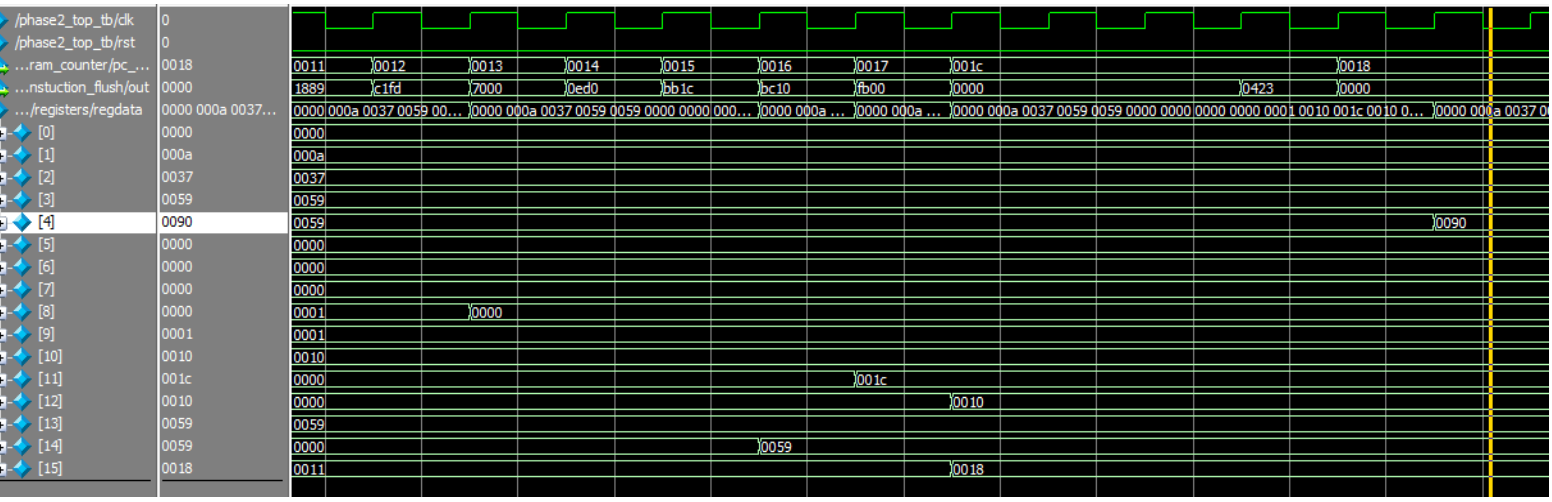


Figure 23

- When zooming into this above figure, we can found that when performing jumping, our program counter is stalled for two cycle and the first two instructions is flushed. The third cycle's instruction is successfully fetched.
- The detail demonstration of stall and flush in Phase 2 is explained in next test case.

4.2.2. Intensive Memory Load/Store Test

The instruction of this test case is as follows:

```
// a series of load word
0000  a911
0001  8494
0002  8595
0003  8190
0004  8291
// a series of arithmetic instruction
0005  0312
0006  0311
0007  0654
0008  0455
0009  0344
000a  0444
//a series of memory write
000b  9101
000c  9202
000d  9303
000e  9404
000f  9505
0010  9601
0011  9702
0012  9803
0013  9904
0014  9a05
//dummy instruction
0015  0344
0016  0455
0017  0322
0018  9105
//data memory
1100  0001
1101  0002
1102  0003
1103  0004
1104  0009
1105  0001
```

Figure 24

In this test case, we will be executing a consecutive of LW instructions, and then followed by a series of arithmetic instruction, followed by a consecutive of SW instruction. The result of this test case will be shown and the performance improvement made by our *memory_top* module to the memory module discussed in the following.

4.2.2.1 Consecutive Load Word (Instruction 0001 - 0004)

The following screenshot shows the wave form of the signal when executing consecutive LW instructions.

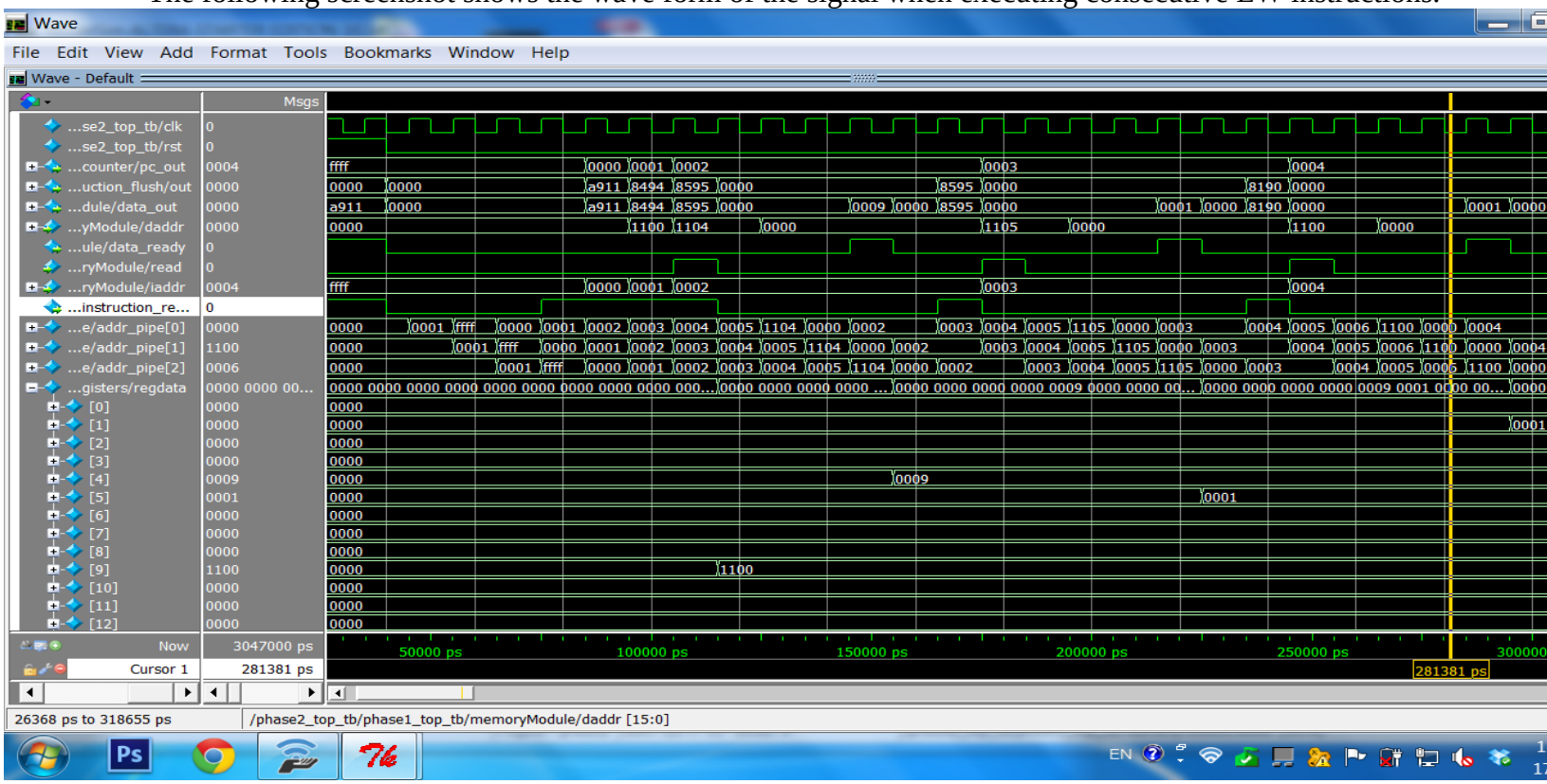


Figure 25

As seen in the screen shot of the wave line above, that the program counter is stalled while waiting for the data memory to be loaded from the memory. Take note that although the instruction is readily to be taken from the data_out of the memory, this is because *memory_top* has started fetching the instruction while the previous instruction is being fetched as discussed earlier. However the instruction is being flushed away or ignored as a memory data read is prioritized.

After getting the data, *data_ready* is set high, thus the memory module is able to fetch instruction and will set the *instruction_ready* flag to be high when the instruction is available. And the program counter exit from stall.

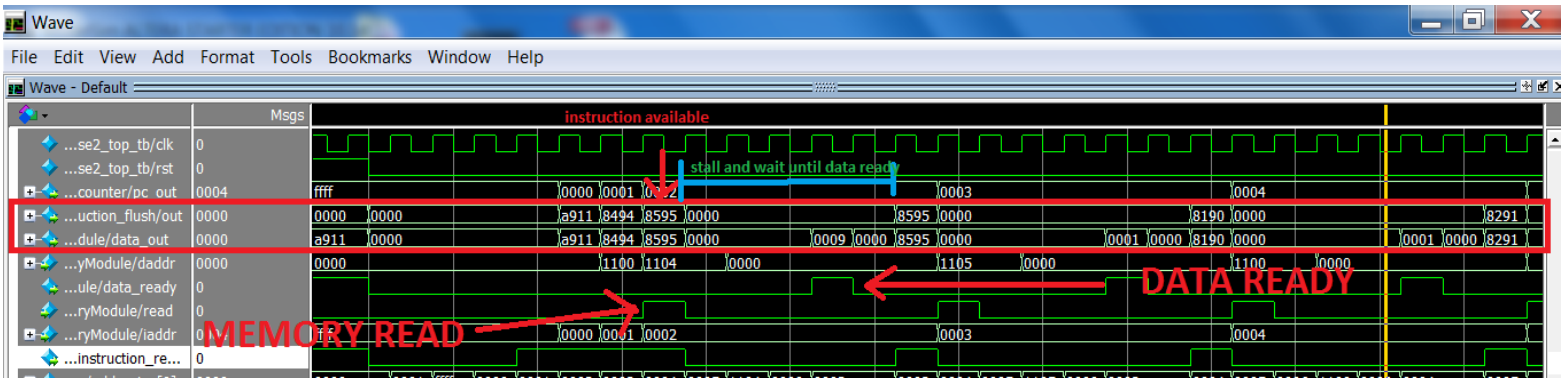


Figure 26

4.2.2.2 Consecutive Arithmetic Instruction (Instruction 0005 - 000a)

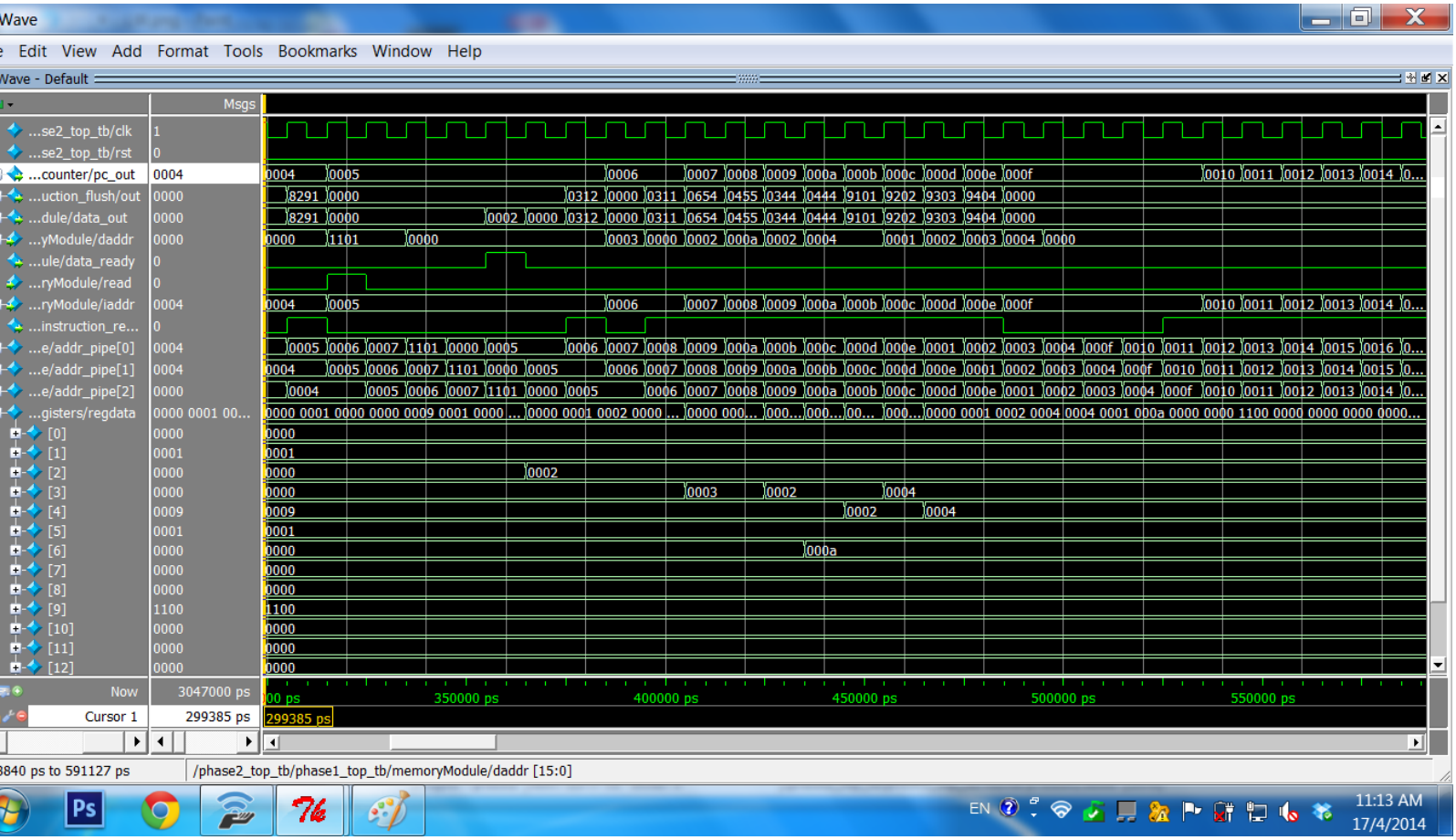


Figure 27

In this screenshot, we are going to look at how the performance of the memory improved by the *memory_top* module by pre-fetching the instruction. Having a consecutive instructions that without any branching or memory accessing, the latency of the instruction fetching can be reduced.

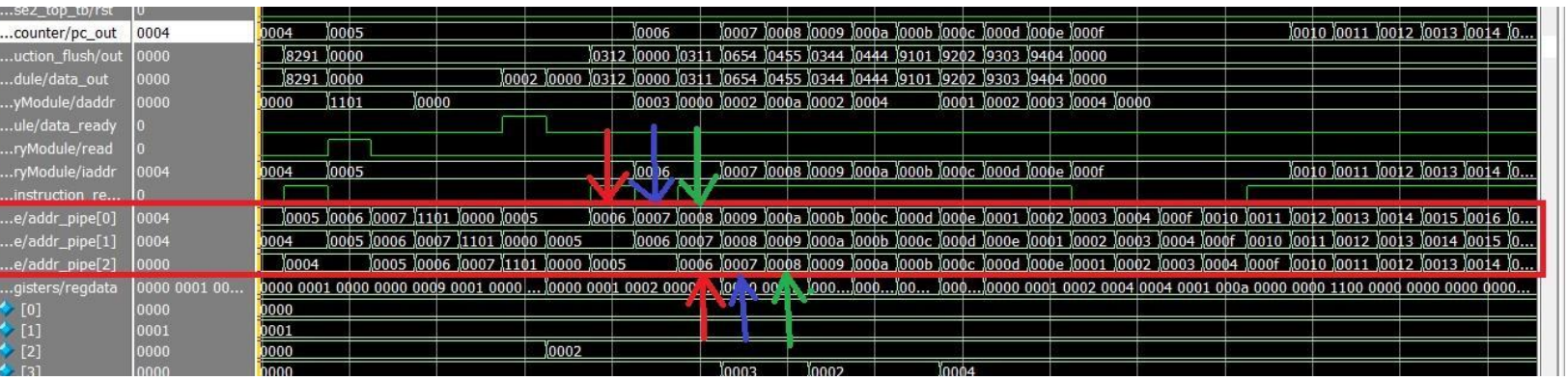


Figure 28

We can see that when waiting for the instruction at address 0005 to come out from the memory, address 0006 is being issued to the *addr* of the *memory* module (as shown as the red arrow). Therefore, fetching of instruction at address 0006 is reduced to 2 cycles and we can see that the program counter stalls only for 1 cycle.

And while waiting for the instruction at address 0006 to come, address 0007 is issued to *addr*. and in the next cycle, address 0008 is issued. In the next cycle, when program counter increment to 0007, the instruction 0007 is readily available, and so does the next cycle for 0008 and so forth.

In the long run without any branching or memory read/write, the CPI (Cycle Per Instruction) averages to nearly 1. Although, it is not practical to have a program without any branch or memory read/write, however, performance will greatly improve if the program have any consecutive arithmetic instruction or load immediate instruction that does not change the program counter nor read or write into memory.

4.2.2.3 Consecutive Store Word Instruction (Instruction 000b - 0014)

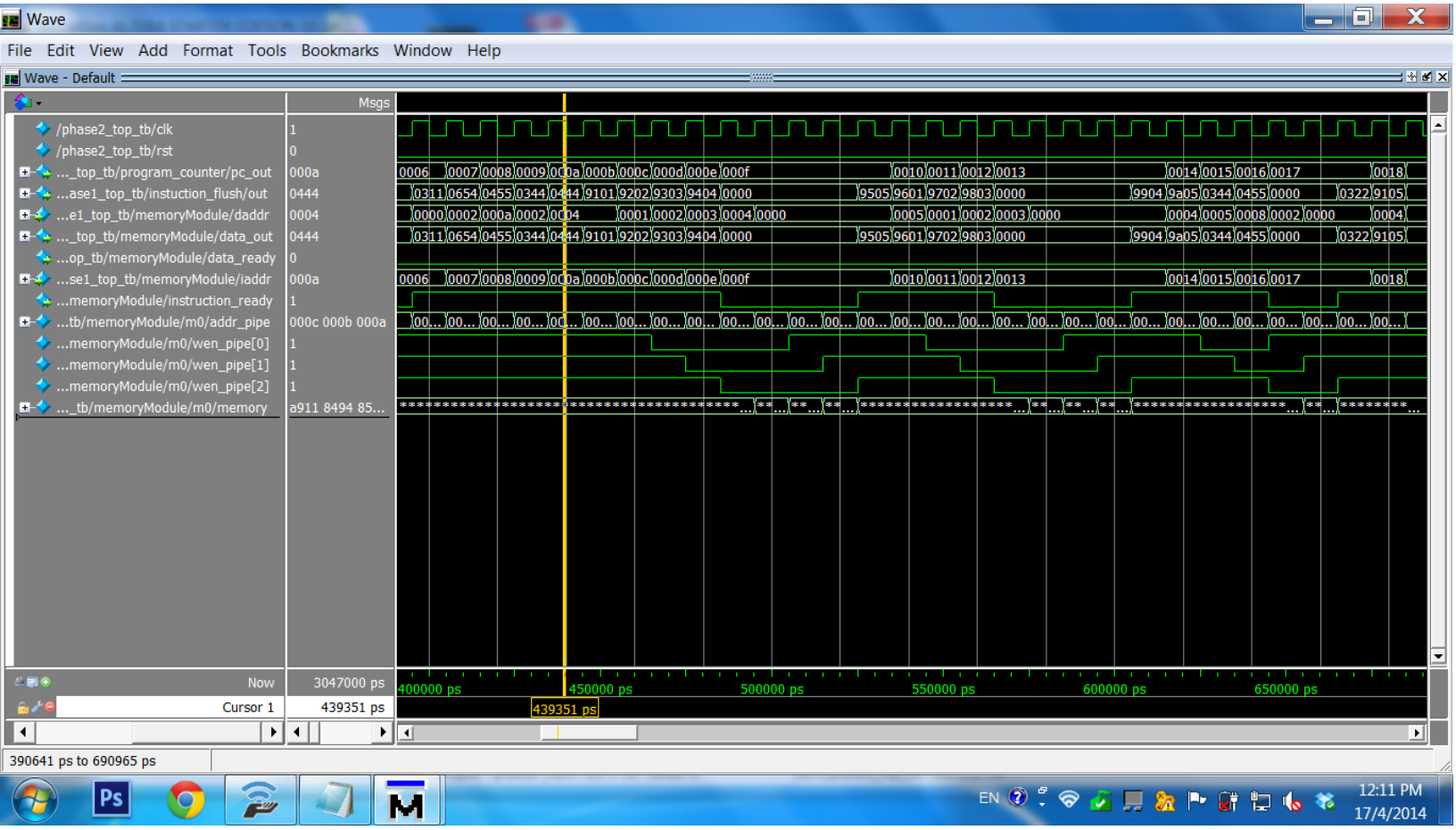


Figure 29

As mentioned in the earlier section, memory is written in burst mode if memory write is executed consecutively. This is because that we did not stall the pipeline for each Store Word instruction. Instead, we schedule the memory write that will be executed in 3 cycles later. If we execute Store Word consecutively, each store word is executed in 1 cycle and after 3 cycles, the instruction stall for 3 cycles for the memory write. Therefore, in average, the CPI (Cycles Per Instruction) is near to 2. If instead we flush the memory address pipeline for each Store Word instruction, we will be having 1+3 cycles for each Store Word instruction (1 for Instruction Fetch and 3 for Memory Write, as these 2 instruction cannot be executed concurrently)

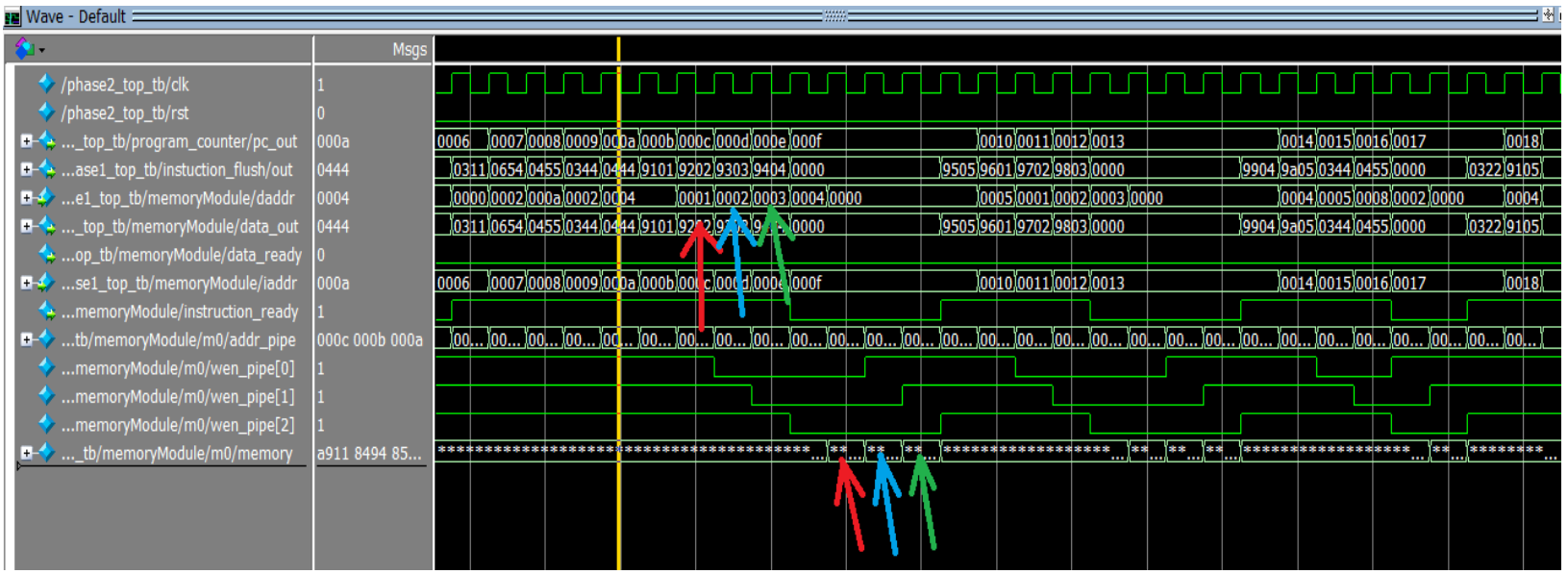


Figure 30

The top 3 arrows shows that the memory write that is being scheduled and the lower 3 arrows shows the memory being written respectively.

5. Discussion

$$\text{CPI} = \frac{Cycles}{Instructions}$$

For the test case 1 of phase 1, we got 29 cycles and 25 instructions. According to the formula above, we could calculate the $CPI_{p1\text{-}1} = 29/25 = 1.16$.

For the test case 2 of phase 1, we got 133 cycles and 96 instructions.
According to the formula above, we could calculate the $CPI_{p1\text{-}2} = 133/96 = 1.38$.

In this test case, the number of flush for the instructions is 32 because there are so many subroutines of branch needed to be flush in the Fibonacci Number Test.

For the test case 1 of phase 2, we got 315 cycles and 96 instructions.
According to the formula above, we could calculate the $CPI_{p2\text{-}1} = 315/96 = 3.28$.

In the comparison between test case 2 of phase 1 and test case 1 of phase 2, $CPI_{p2\text{-}1}$ is about 2 times than $CPI_{p1\text{-}2}$. The reason of this results is that there exist more data hazard and data forwarding because of the shared memory in the phase 2, and it costs lots of CPU resources and extends the cycles. If the test case of phase 2 only included arithmetic instructions like ADD, SUB, OR, ADD, etc., the CPI would be the same as the results that were obtained by the same case tested in phase 1.

6. Future Upgrade

In Phase 2, the instructions and data share a main memory. There is a latency of 3 cycles each time we access it. We recommend cache for future upgrade and we present our idea here.

In the future design, there should be an instruction cache and a data cache that cache the instruction and data, and therefore both cache can be accessed concurrently. We have considered of using direct-mapped cache for both caches, with block size of 4 words, which means that we will be storing 4 words in a block from the memory each time when there is a miss in accessing the cache.

The following diagram shows the rough sketch of the I-cache and D-cache that will be implemented in the future.

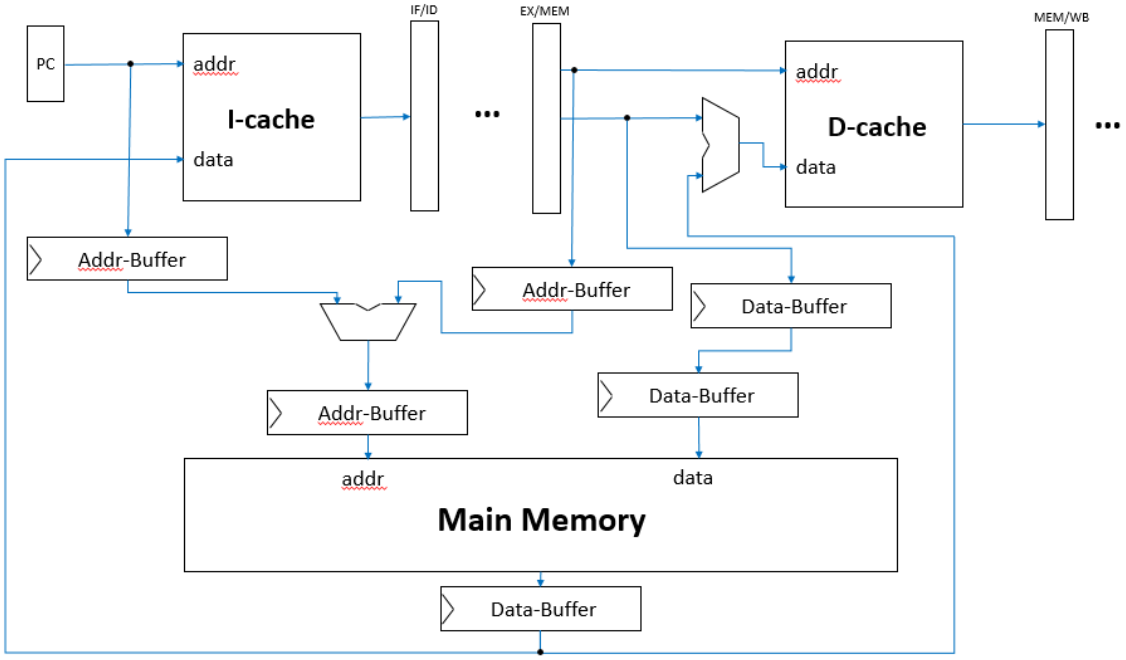


Figure 31

For the instruction cache, the pipeline will only be stalled when there is a fetch missed. The instruction will then be fetched from the memory and stored in to the I-cache and tagged appropriately.

For the data cache, the read operation is similar to the I-cache. However for writes, we would implement a write-back allocate policy. When there is a SW instruction in the pipeline, the cache would need to check if it hits in the cache. If it is a write hit, the data-cache would write the word to the cache and will be mark as “dirty”, and only write back to main memory when the block in cache is being replaced by new content. If it is a write miss, the processor would fetch the block from memory to cache and update the corresponding write data into the cache and mark it as “dirty”.

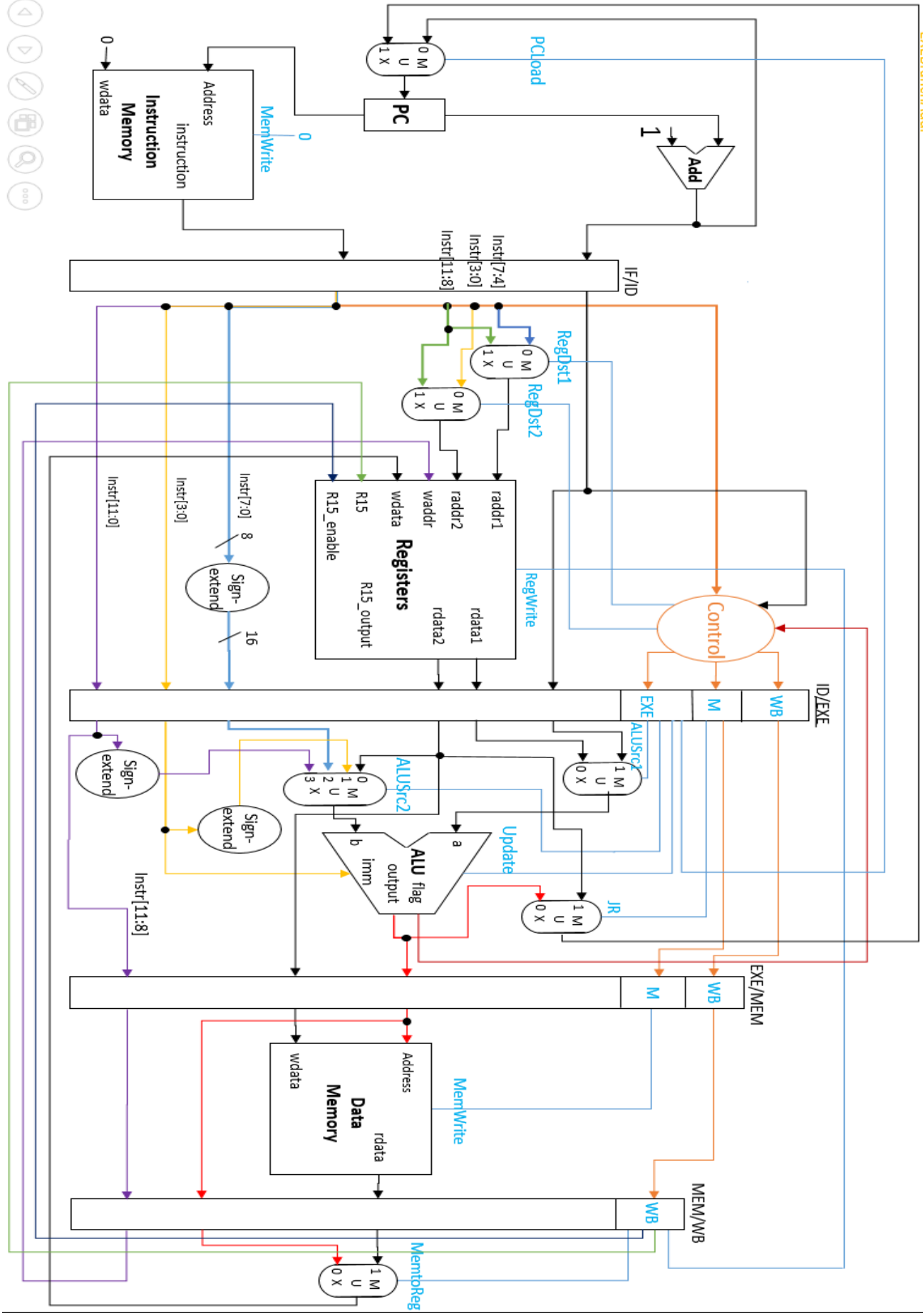
7. Conclusion

Overall, the implementation of a basic CPU with 16-bit MIPS-like ISA written by our team in Verilog for this project is successful in both phase 1 and 2 where we have considered the pipeline, hazard detection, data forwarding and shared memory.

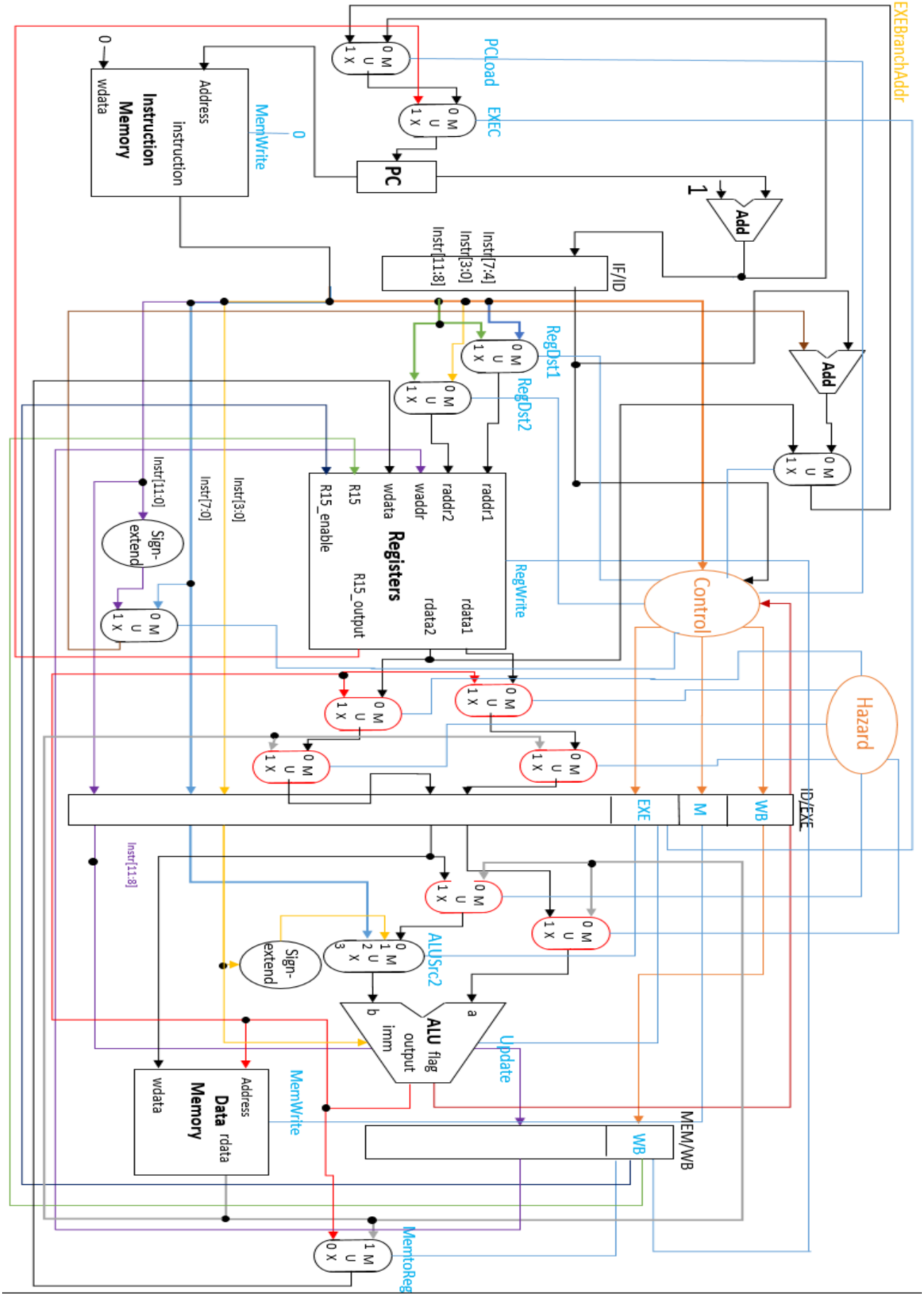
This project helped us understand how a pipelined CPU works, the importance of dividing the instructions into different register stages, how CPU time can be saved as opposed to a single-cycle or a multi-cycle implementation and the implementation of data hazard, data forwarding and shared memory combined from instruction and data memory.

It is a very complex project, mostly because of these different modules with a large number of registers and wires. We need to keep track of a lot of different wires and the clock issues that lots of components need to be clocked. Furthermore, the debugging takes us lots of time and we need to decide which of the components needed to be clocked and which ones didn't. Finally, in the future, we can implement the cache pattern as the Future Upgrade shown. It will improve our design and reduce more cycles and CPU resources.

8. Attachment 1 – Interim Report Design Diagram



9. Attachment 2 – Final Report Design Diagram for Phase 1



10. Attachment 3 – Final Report Design Diagram for Phase 2