California State University, Fresno
Lyles College of Engineering
Department of Electrical and Computer Engineering

TECHNICAL REPORT

Project Title: Mips Processor
Course Title: ECE 174 Advanced Computer Engineering
Date Submitted: May 18, 2023
Course Instructor: Dr. El Razouk

Prepared By	Section Written
Omer Al Sumeri	Abstract, Conclusion
Aryan Singh	Literature Review, Procedure
Luigi Santiago-Villa	Theoretical Background, Results

INSTRUCTOR SECTION

Comments:

____
____
____
____

Final Grade: Omer Al Sumeri :
Aryan Singh :
Luigi Santiago-Villa:

TABLE OF CONTENTS

1. Abstract....................**4**
2. Literature Review.................... **4**
3. Theoretical Background.................... **4**
4.1 Instruction Stage Modules....................8
4.2 Control Module.................... 8
4.3 Register File.................... 9
4.4 Forwarding Unit....................10
4.5 Hazard Detection Unit....................11
4.6 ALU Module.................... 12
4.7 Data Memory.................... 12
4.8 Single Cycle Implementation....................13
4.9 Pipeline Registers....................14
4.10 Pipelined Implementation (Top Module).................... 15
4.11 Equipment Used....................15
5. Results.................... **16**
5.1 Load Word Instruction Simulation....................16
5.2 Store Word Simulation Results.................... 17
5.3 Add Simulation Results.................... 18
5.4 Add Immediate Simulation Results.................... 19
5.5 And Simulation Results.................... 19
5.6 Nor Simulation Results....................20
5.7 Or Simulation Results....................21
5.8 Set-Less-Then Simulation Results....................22
5.9 Subtraction Simulation Results....................23
5.10 Multiplication Simulation Results.................... 24
5.11 Division Simulation Results....................26
5.12 Forwarding Unit Simulation Results....................27
5.13 Hazard Unit Simulation Results.................... 28
5.14 Branch If Equal Simulation Results....................29
5.15 Jump Instruction Simulation Results....................29
5.16 Synthesization Results....................30
6. Conclusion....................**30**
7. References....................**31**
8. Appendices....................**32**
Appendix A.................... 32
Appendix B.................... 33
Appendix C.................... 34
Appendix D.................... 36
Appendix E....................37
Appendix F....................38
Appendix G.................... 39
Appendix H.................... 40

Appendix I..41
Appendix J..42
Appendix K..43
Appendix L..44
Appendix N..47
Appendix O..49
Appendix P..50
Appendix Q..51
Appendix R..52
Appendix S..53
Appendix T..54
Appendix U..55
Appendix V..56
Appendix W..57
Appendix X..58
Appendix Y..59
Appendix Z..60

1. Abstract

The objective of this project is to design and implement a processor on Modelsim that would run a program that is based off the Microprocessor with Interlocked Pipeline Stages (MIPS) instruction set. The processor will be 32 bits and will follow all of the properties of the original MIPS processor.

2. Literature Review

The main idea behind this project is to implement a basic MIPS processor ISA (Instruction Set Architecture) using Verilog. The literature associated and incorporated in this project is the five-stage pipelined data path which we learned in the Advanced Computer Architecture course. The organisational blocks that perform the major functions in this project are the Program Counter, Instruction Memory block, Register block, Arithmetic Logical Unit and the Data Memory block. Throughout this project we accommodate many more blocks like the Sign extend block, branching, Forwarding and Data Hazard blocks to prevent any Data and Control Hazards. A single cycle approach was taken initially to create a working MIPS implementation however, to reduce the implementation time and construct a more optimised architecture, the pipelines were introduced. So, through this project the pipelined implementation of the MIPS Instruction Set is designed and simulated.

3. Theoretical Background

With the pace and complexity of modern technology, it can become difficult and daunting to understand how a computer operates or comes to be. The first step in understanding and designing the modern computer is understanding its most critical component. The heart of a computer's operation is the processor. The computer processor is colloquially referred to as the brains of the computer. At its very simplest, all operations can be broken down from higher levels of abstraction to low machine level instructions. At its simplest, computers operate in binary but the language is difficult to understand. The first level of abstraction is found at the assembly level. The most commonly used assembly languages found in modern devices are ARM, x86, and MIPS, which is the focus of this lab.

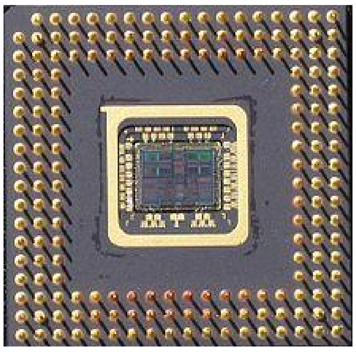


Figure 3.1: R4700 Processor

The MIPS processor is a Reduced Instruction Set Computer (RISC) processor. The name refers to a processor and instruction set architecture that is simpler and efficient. The simpler processor design provides the advantage of abstraction, and in doing so,it favours productivity. The MIPS instruction set can be broken down into three instruction formats. The two most common types R-type and I-type instructions form the foundation of most assembly languages. The R-type instruction consists of a label and 3 operands, as shown in figure 3.1. The 32 bit instruction address can further be broken down in the format shown in figure 3.2.

```
add  $a0,$s0,$zero
```

Figure 3.2: R-Type Instruction

The first 6 bits provide us with the opcode. The opcode defines the general operation to be performed on the contents of the registers. The next 15 bits define the three addresses used in the R-type instruction. The source register is listed first, followed by the target register, and a destination register. After the register addresses, we have five bits that hold the shift amount (shamt) and the remaining six bits define the specific function performed. The I-type instruction possesses the same first 16 bits as the R-type instruction but the remaining 16 bits are used to define the address to which the read or write result should be written to.

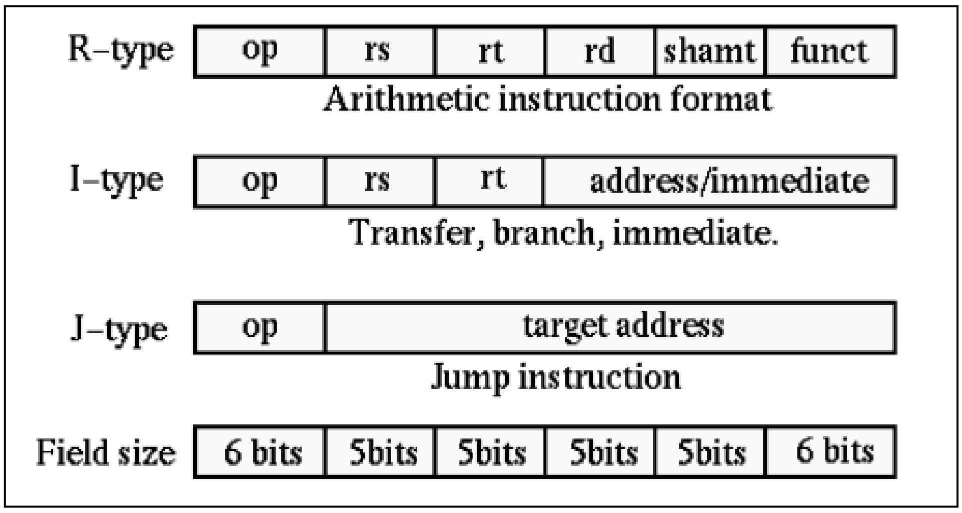


Figure 3.3: MIPS Instruction Format

The mips processor operates upon 32 registers where each register consists of four bytes. As shown in figure Table 3.1, the first instruction listed is the $zero register and aptly named, for it holds a constant value of zero. The $zero register is a prime example of the reserved registers that make up the 32 available registers. Among these reserved registers, we have the stack pointer $sp and the return address $ra, the latter of which is critical for recursive programming.

Table 3.1: MIPS Registers

Name	Register Number	Purpose
$zero	0	Constant 0
$at	1	Reserved for Compiler
$v0 - $v1	2 -3	Results
$a0 - $a3	4 - 7	Arguments
$t0 - $t7	8 - 15	Store temporary values
$s0 - $s7	16 - 23	Store saved values
$t8 - $t9	24 - 25	Store temporary values
$k0 - $k1	26 -27	Reserved for OS
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return Address

Table 3.2: MIPS Instructions and OP Codes

Operation	OP Code	Function Code
Add rd,rs,rt	000000	100000
AddI rt,rs,Immediate	001000	IMMEDIATE
Mult rs,rt	000000	011000
Div rs,rt	000000	011010
Nor rd,rs,rt	000000	100111
OR rd,rs,rt	000000	100101
AND rd,rs,rt	000000	100100
BEQ rs,rt,offset	000100	OFFSET
J target	000010	TARGET
SLT rd,rs,rt	000000	101010
MFHI	000000	010000
MFLO	000000	010010
Lw	100011	OFFSET
SW	101011	OFFSET

As previously stated, the computer “thinks” or operates in the machine language binary. The binary language is implemented by hardware as voltage signals through circuit elements. Each individual signal or *Bit* travels on a single wire throughout the circuit with multiple bits moving along buses. The modules created in this project, such as the ALU and memory, are created through the use of combinational and sequential elements. The output of combinational circuits is dependent on the input composition. Sequential modules are dependent on the specific combination of inputs and the previous state of the machine. The sequential moniker alluding to the use of a clock to synchronise its operations to. With this in mind, we now possess the basic knowledge and components to build a single simple processor.

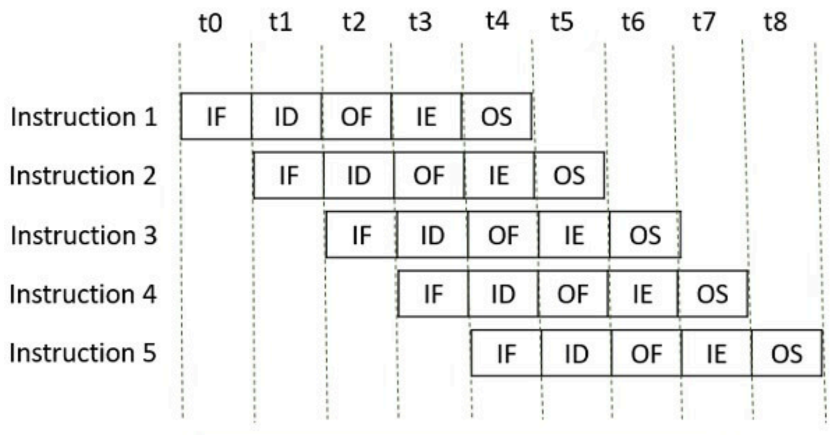


Figure 3.4: Pipelining Example

The Forwarding module is used in the Pipelined MIPS Datapath to reduce or even eliminate Data Hazards. In a Single Cycle MIPS datapath the value of an updated register is available only after the Memory Write back stage when the new value has been stored back into the register file. However, in pipelining, where multiple instructions are going through different stages of the pipeline simultaneously, then the updated value might be required before the Memory Writeback stage. In that case the Forwarding unit is used to ensure that the updated value of a register is passed onto the next instruction of the pipeline by forwarding the value straight after the execution stage and not waiting for the Memory Writeback to occur. This is how Forwarding is implemented.

4. Experimental Procedure

4.1 Instruction Stage Modules

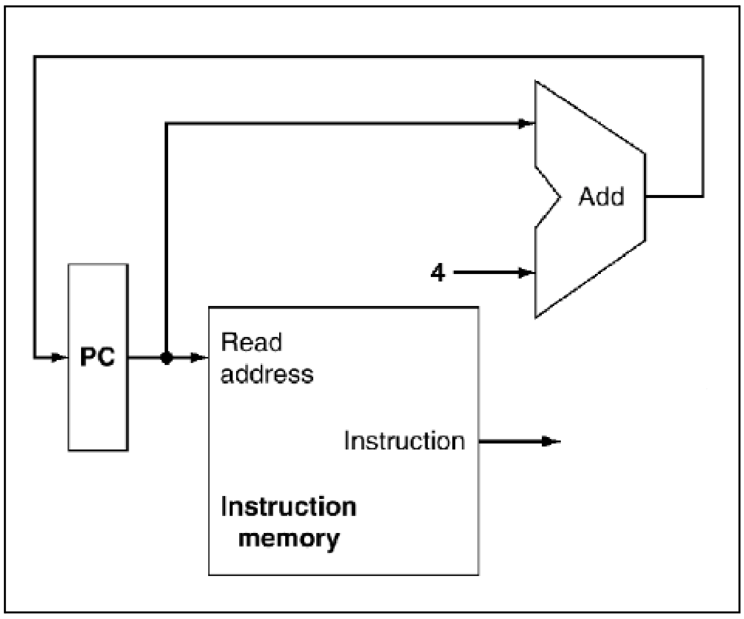


Figure 4.1: Instruction Stage Modules

MIPS instruction comprises various datapath elements connected together through wires and the first stage of the datapath includes the Instruction stage Modules, as can be seen in figure 4.1. This stage of the datapath includes the Program Counter, Instruction Memory and an Adder. The program counter is responsible for holding the address of the current instruction. In this stage the PC is also responsible for holding the address of the next instruction being input upon its incrementation from the Adder. The Program counter then forwards the instruction to the Instruction memory which, like the name suggests, stores the instruction in it for it to be decoded in the Instruction Fetch/Instruction Decode (IF/ID) pipeline register (Figure 4.6). So, overall this stage performs fetching and storing of the instruction.

4.2 Control Module

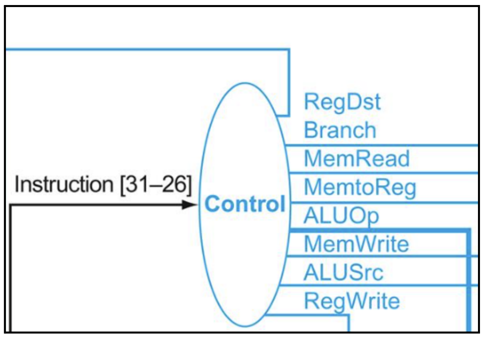


Figure 4.2: Control Module

The functionality of the pipelined datapath is determined by the Control Module (Figure 4.2). The Instruction OpCode is passed as an input to the control module and according to the OpCode the Control module decides which function will be carried out. The Control module has various signal outputs (Figure 4.5) which act as the signal inputs to all the datapath modules.

RegDst - The RegDst signal determines if the Destination register is specified as the Rd or the Rt register in the case of an R or an I type instruction.

Branch - The Branch signal goes into a mux to decide whether the instruction will branch off to another instruction or not. This functionality will be used in the case of beq (Branch on Equal) instruction.

MemRead - The MemRead signal enables memory to be read from the Data Memory in the case of Load instruction.

MemtoReg - The MemtoReg signal decides if the ALU Output or the Data Memory output value will be written back to the register file.

ALUOp - The ALUOp signal is a 2 bit signal that decides which operation will be executed in the ALU by passing 2 bits to the ALU Control module.

MemWrite - The MemWrite signal enables memory to be written to the Data memory in the case of Store instruction.

ALUSrc - The ALUSrc signal decides which operand from the Rt or the Rd will be passed to the ALU as the second source operand. This signal is only used in the Single Cycle Implementation and gets replaced by the Forwarding unit signals for the pipelined datapath.
RegWrite - The RegWrite signal enables the final output to be written to the Register File.
Every module receives its signal values which are generated by the Control module depending on the OpCode.

4.3 Register File

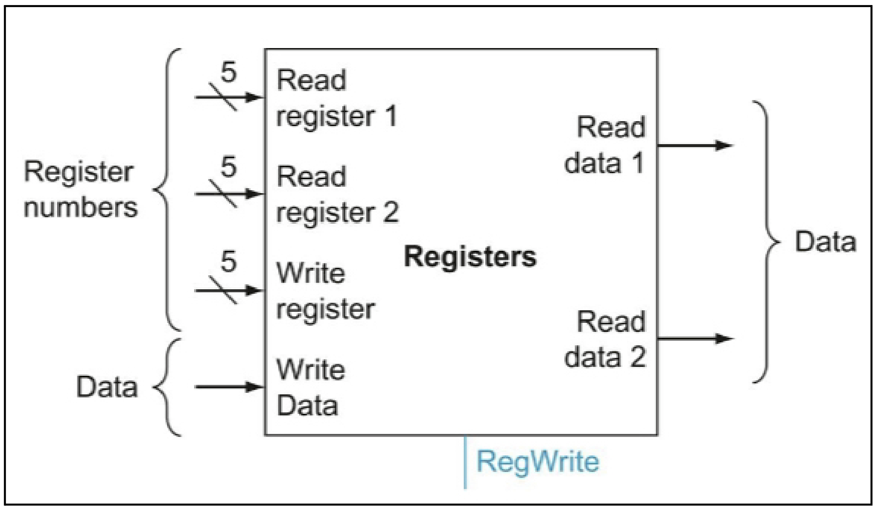


Figure 4.3: Register File

The output from the Instruction memory is decoded by using the first pipelining stage, Instruction Fetch and Instruction Decode (Figure 4.6). The IF/ID stage fetches the 32 bit instruction from the instruction memory and decodes it into different outputs based on the type of function and the Field size associated with each function, as seen in Figure 3.2. The decoded instruction is passed onto the register file which consists of various registers within it(Figure 4.3). The Read register 1 and Read register 2 are the two registers which read the 2 decoded data words in the case of an R or I type instruction. These 2 registers get 5 bit inputs each into them to specify one of the 32 registers (2^5=32) and the 2 output buses, namely Read data 1 and Read data 2, output the data in those registers to the following stages. The final output is also written back into the register file and for that purpose we have the Write register which stores the address for the register the data will be written back to and the Write Data has the actual output data in it that needs to be stored.

4.4 Forwarding Unit

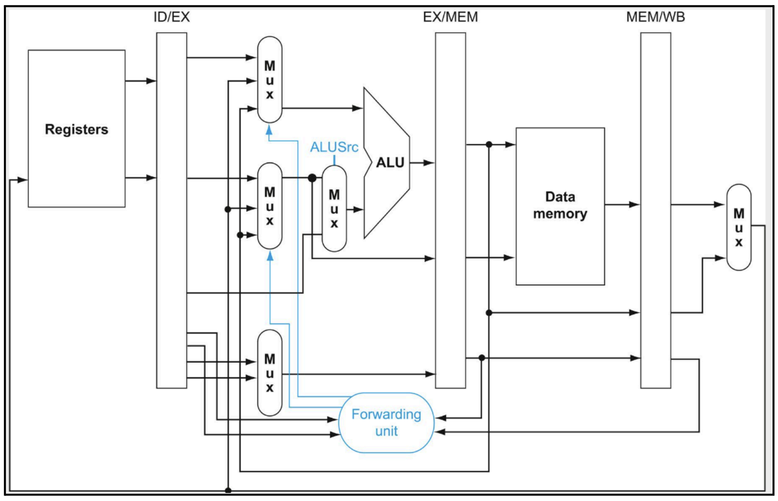


Figure 4.4: Forwarding Unit

The Forwarding unit is useful to avoid Data Hazards by forwarding the updated register values right after the Execution stage to the next instruction. It ensures that the Execution stage of the next instruction gets the updated values of the registers in it. The Forwarding module checks for an EX/MEM data hazard if the current instruction in its Execution stage has a previous instruction that will write to the register file and the destination of that instruction is one of the source registers that is in its Execution stage. To resolve it the forwarding unit sends the ForwardA signal to the Multiplexer which forwards the correct value from the EX/MEM register to the ALU's first input. Another hazard that the forwarding can resolve is the MEM/WB hazard. For this purpose the Forwarding module detects the same hazard but between the MEM/WB and the register value going into the Execution stage. To resolve this hazard the forwarding module sends the ForwardB signal to the Multiplexer which forwards the correct value from the MEM/WB register to the ALU's second input.

4.5 Hazard Detection Unit

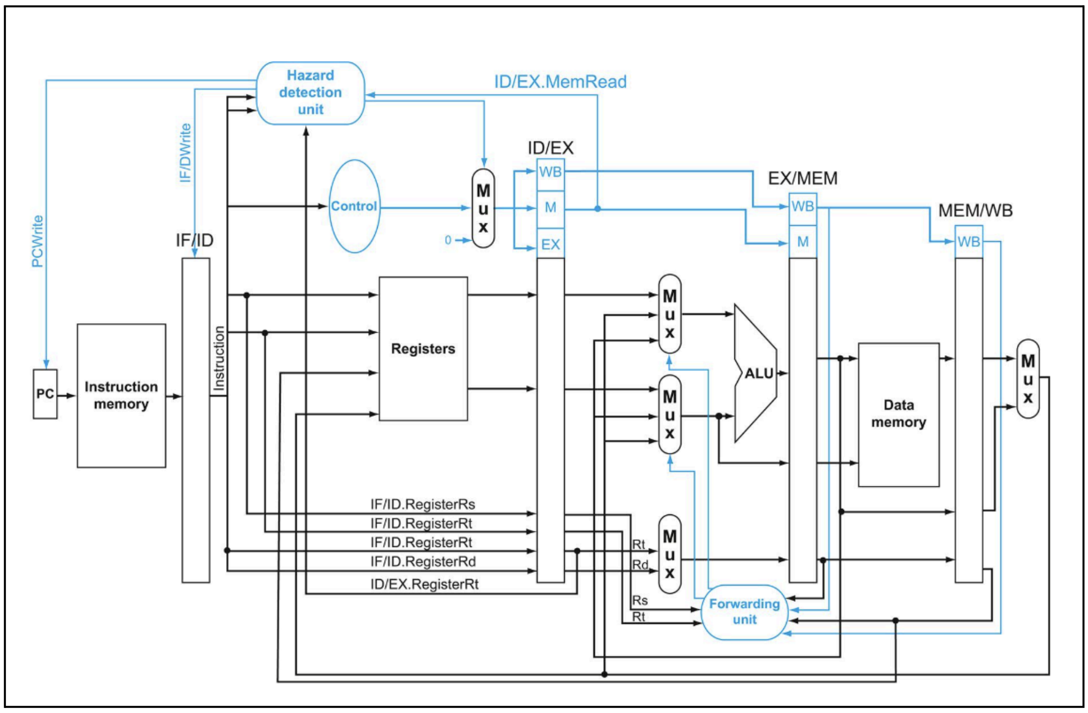


Figure 4.5: Hazard Detection Unit

For the case in which the forwarding unit cannot prevent the data hazard from occurring then in that case the hazard detection unit is used. This unit checks the case in which forwarding does not remove the data hazard like the case in which an instruction wants to read a register after a load instruction has written to the same register. In that case the data read from memory is done simultaneously while the next instruction is performing ALU operation. So to remove this hazard the data hazard unit stalls the entire pipeline by introducing a NOP (No Operation) instruction.The hazard detection unit also checks for any control hazards that occur during branching when the branching does not occur. To resolve this the unit sends out a NOP instruction which stalls the pipeline and flushes out the branch instruction that had already started executing.

4.6 ALU Module

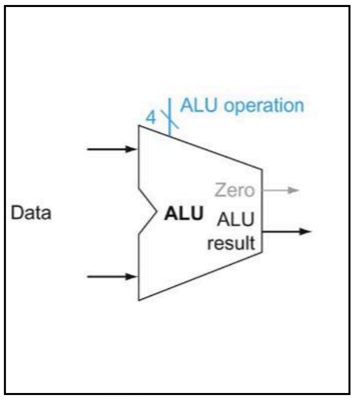


Figure 4.6: ALU

The outputs from the 2 data output buses are passed onto the ID/EX pipeline register which holds these 2 values as well as the target address in the case of a branch instruction. The target address is 16 bits and it passes through a sign extension unit to create a 32 bit address for the branching. So overall these 3 register values are passed onto the ID/EX register which holds onto them before passing them to the ALU.
The Arithmetic Logic Unit or ALU (Figure 4.5) is the next stage in the datapath where all the computations of all the instructions are executed. The ALU has the following instructions that are being implemented in it :

- Arithmetic/ logic operations : and, or, nor, add, sub, slt, addi, div, mult
- Data movement operations : lw, sw, mfhi, mflo
- Flow control operations : beq, j

All of these operations have their Operation Code and Function Code which is tabulated in Table 3.2. Out of all these operations the ALU chooses which operation to perform according to the ALU Operation Signal (Table 3.2) coming in from the Control Module. This signal decides which operation is to be executed on the register operands and the result of the operation is passed out onto the next stage of the pipeline.

4.7 Data Memory

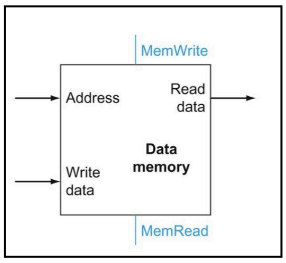


Figure 4.7: Data Memory

The ALU output is passed onto the EX/MEM pipeline register which again acts as a buffer similar to the other pipeline registers. This register holds the ALU output before passing it onto the Data memory module.
The Data Memory module is the other storage module which is used in the case of load word or store word instructions (Figure 4.6). The ALU output is stored in the Address input of the module since the load and store operations require an address to load from or store to. Using the MemWrite and MemRead control inputs, the Data Memory module decides whether to load a value from the memory using the Read Data output and send it back to the Register File for storing that value, or store data in memory using the Write Data input. In the case of an R type instruction there is no use of the Data Memory so this stage is skipped in that case as it can be seen in Figure 4.7.

The output from the Data Memory is passed into the MEM/WB pipeline register which acts as a storage register or a buffer in case of hazards. The MEM/WB register holds the output coming from the Data Memory in case of load instruction or the ALU output in the case of an R type instruction. The pipeline register then passes the stored values back to the Register file for storing.

4.8 Single Cycle Implementation

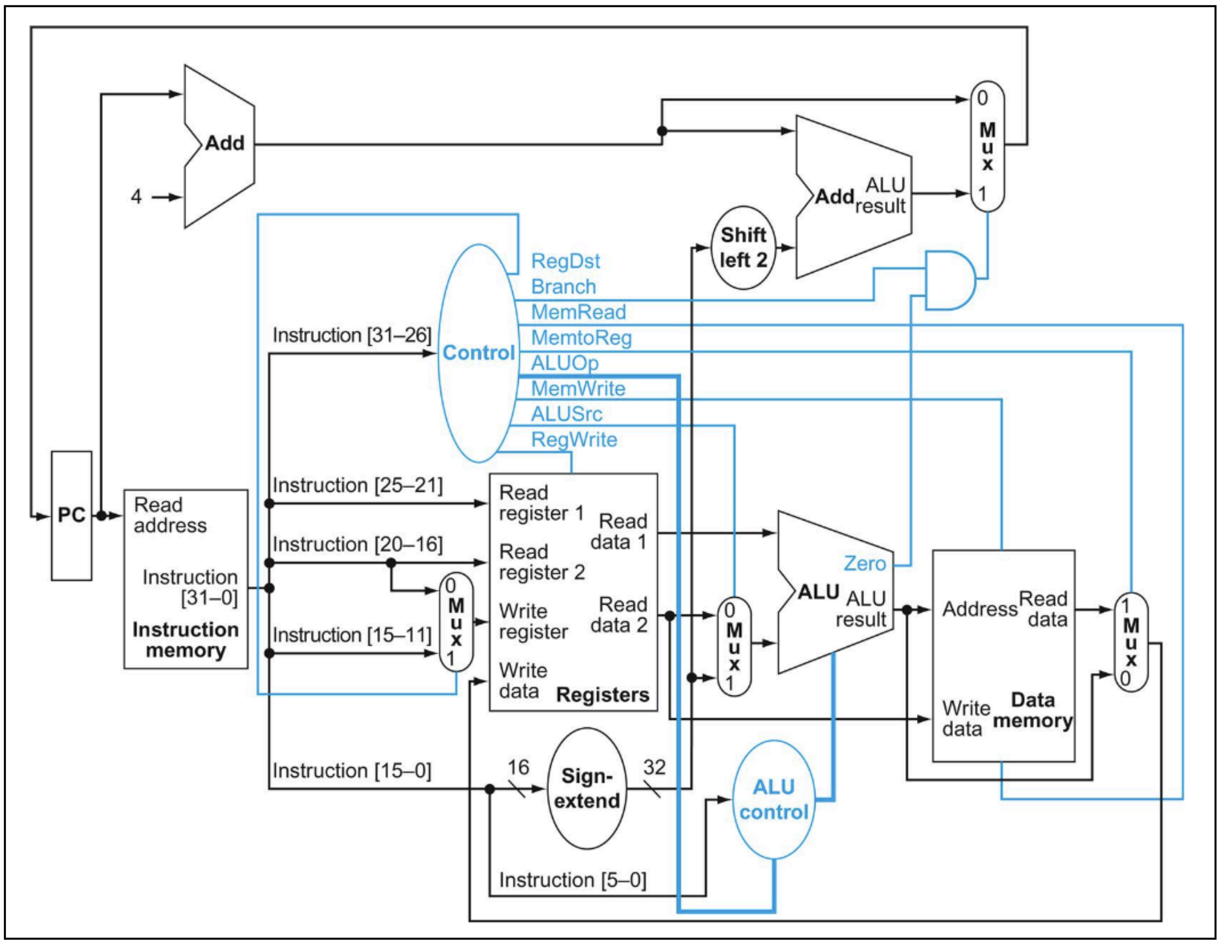


Figure 4.8: Single Cycle MIPS Datapath

Upon combining all of the above mentioned modules without the pipeline stages, we get the Single Cycle MIPS datapath (Figure 4.8). This datapath shows the complete flow of data from the program counter to the Final Data Memory with all the wires showing the port connections between each module. For this purpose a separate module was created called the Top Module (Appendix Z) which connected all of the modules shown in Figure 4.8 to make a single module.

4.9 Pipeline Registers

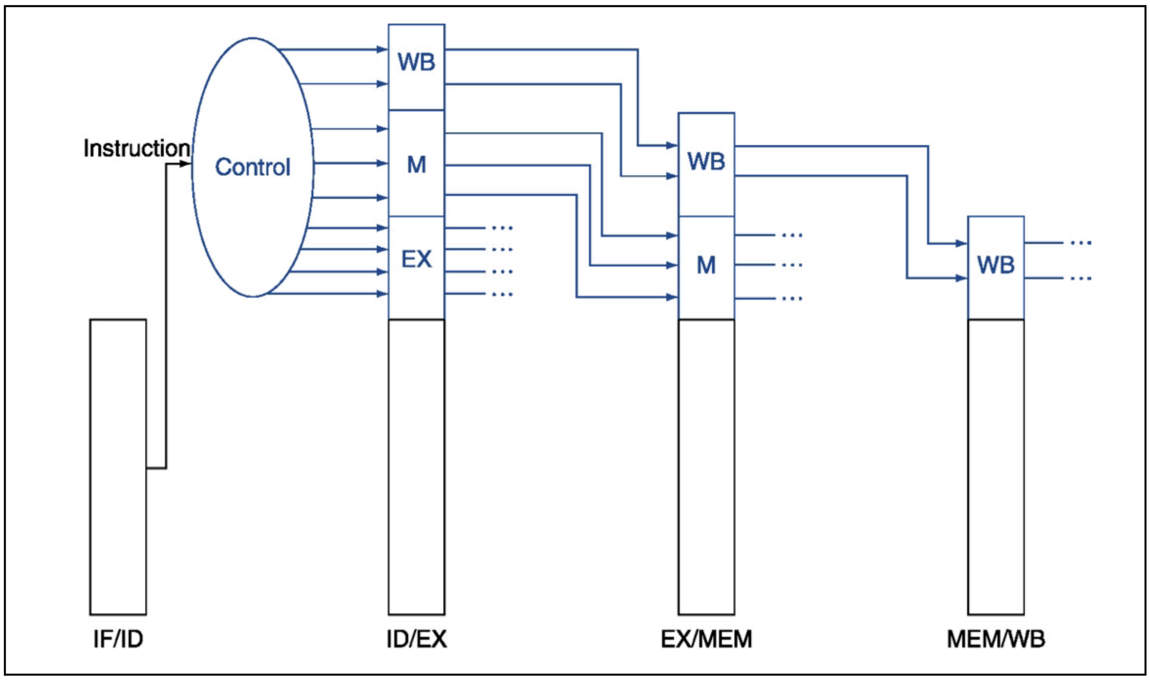


Figure 4.9: Pipeline Registers

All the pipeline stages that are used for the MIPS Implementation can be seen in Figure 4.9. The IF/ID pipeline is used to store the Instruction coming in from the Program Counter and then decode it according to the operation followed by the passing of the first 6 bits to the Control module and the rest to the register module depending on the Instruction to be executed.
The ID/EX pipeline module gets the outputs from the Register File along with the control module signals and holds those values before passing them onto the ALU module through multiplexers and to the Forwarding module for hazard resolution.
The EX/MEM pipeline module holds the outputs coming from the ALU and passes them onto the Data Memory module in the case of a load or store operation, or passes the ALU output to the MEM/WB pipeline register in the case of a R type instruction.
The MEM/WB pipeline module holds the output from the Data Memory in the case of a Load instruction or the ALU output in the case of a R type instruction, and passes the value onto the Register file for storing.

4.10 Pipelined Implementation (Top Module)

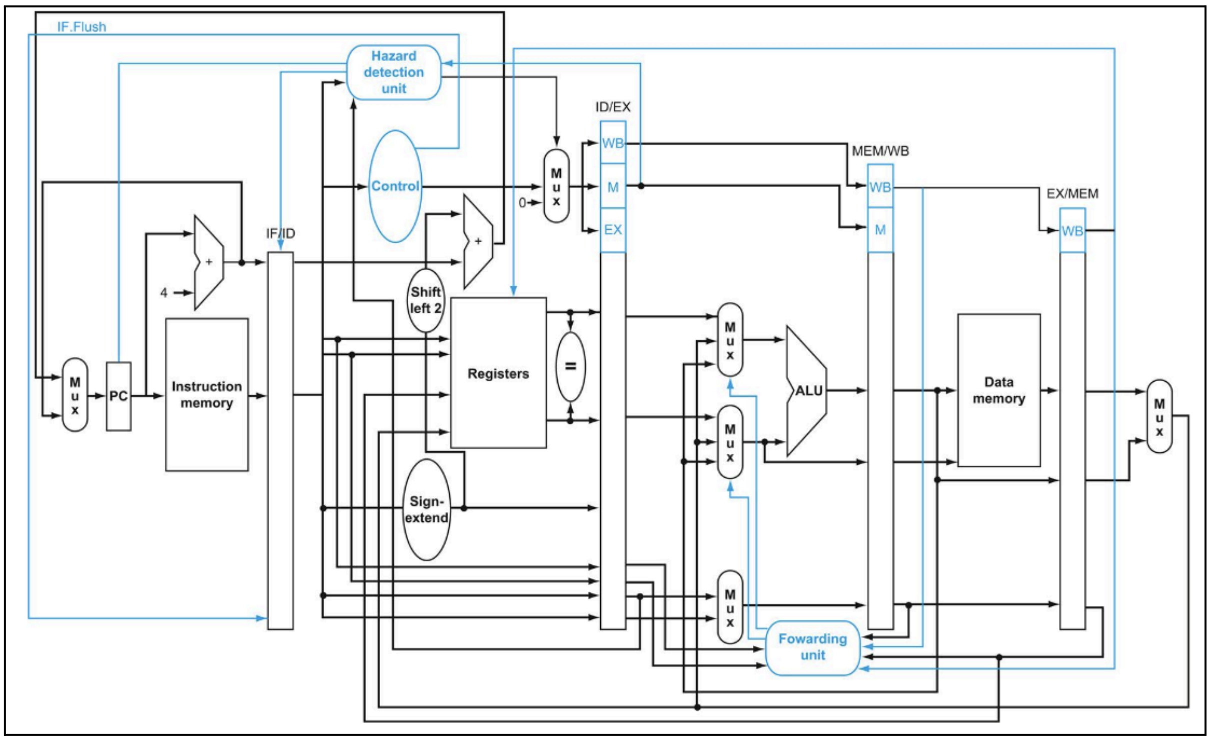


Figure 4.10: MIPS Pipelined Datapath

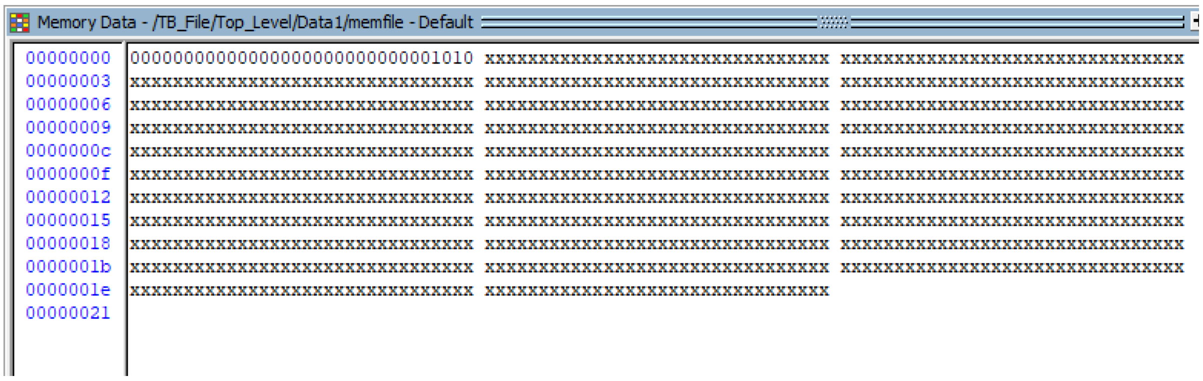
After the implementation of all the modules along with the pipelines, we get the final pipelined datapath as it can be seen in Figure 4.10. This final datapath has all the modules connected together using wires which were incorporated using a Top Level Module. All the modules are connected using ports and instantiations in the Top Level module (Appendix Z), and then finally, all the 3 types of instructions can be synthesised and simulated on this MIPS architecture design model.

4.11 Equipment Used

Equipment Used	Description
Quartus	High level hardware development and synthesis IDE
ModelSim	High level hardware development software
PC	Personal computer used for coding
DE2-115	FPGA development kit

5. Results

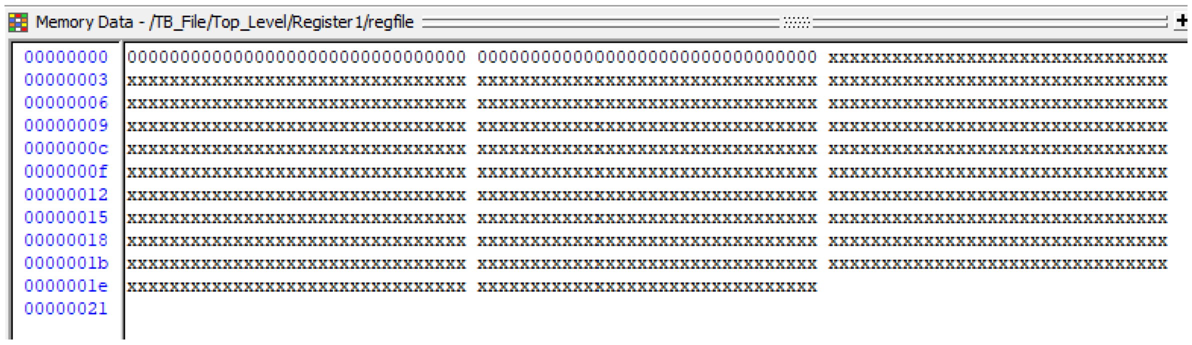
5.1 Load Word Instruction Simulation



```
Memory Data - /TB_File/Top_Level/Data1/memfile - Default
00000000 00000000000000000000000000001010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000021
```

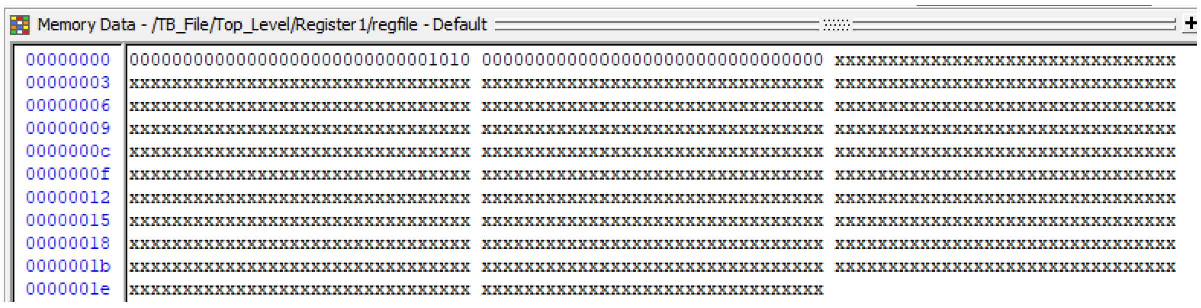
Figure 5.1 : Contents of Data Memory

The first operation tested is the Load word instruction. The instruction decoded during the instruction fetch is 0x8C200000 and breaks down to a destination register of $V0 and a source from the data memory located at address 0x0000. As shown in figure 5.1, the contents at address 0x0 in the data memory is d'10.



```
Memory Data - /TB_File/Top_Level/Register1/regfile
00000000 00000000000000000000000000000000 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000021
```

Figure 5.2 : Register Module Registers

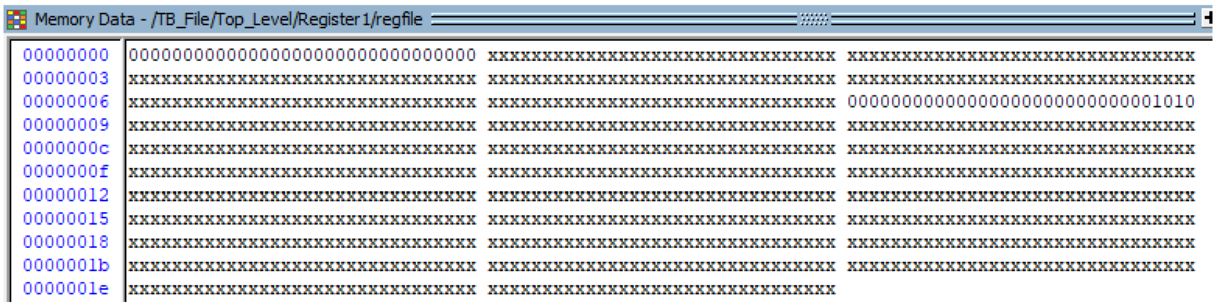


```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 00000000000000000000000000001010 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.3 : Registers After Load Instruction

Whilst the instruction is traversing through the IF/ID register, we can see that the contents of register $V0 and $V1 are first initialised to zero. After the word is brought out from memory and enters the write back stage, we see in figure 5.3 that the value was loaded to register $V0 located at address 0x00000000 in the register module. This gives us confirmation that the module is operating as expected.

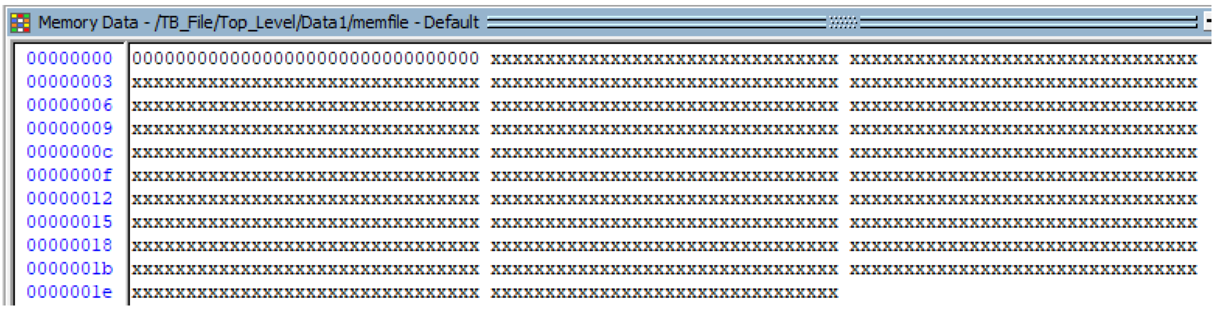
5.2 Store Word Simulation Results


Memory Data - /TB_File/Top_Level/Register1/regfile

```
00000000 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.4 : Register Module with Register $T8 Containing Value d'10

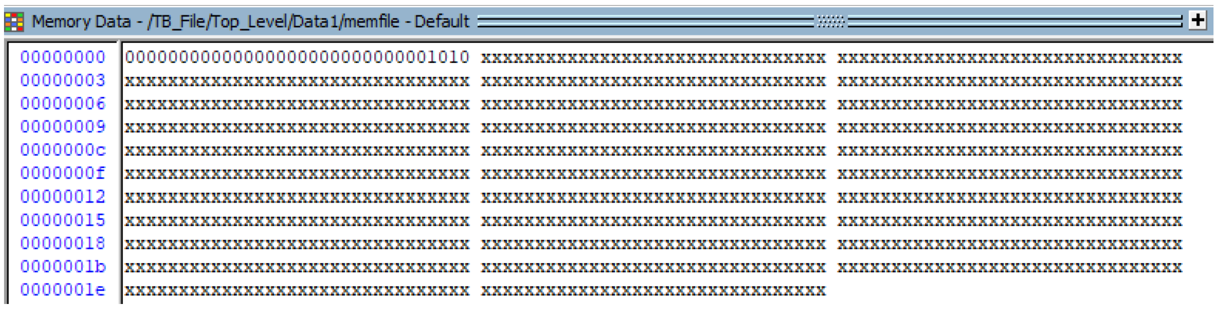
The second instruction tested in the pipelined mips processor is instruction h'AC080000. Once decoded in the IF/ID register pipeline, we see that this instruction code details a store word instruction. After the bits define the op code of 6 bits, the i type instruction then specifies the register that we will use to determine the memory address which consists of 5 bits. The following 5 bits will define the register containing the value we wish to store in memory.


Memory Data - /TB_File/Top_Level/Data1/memfile - Default

```
00000000 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.5 : Empty Data Memory

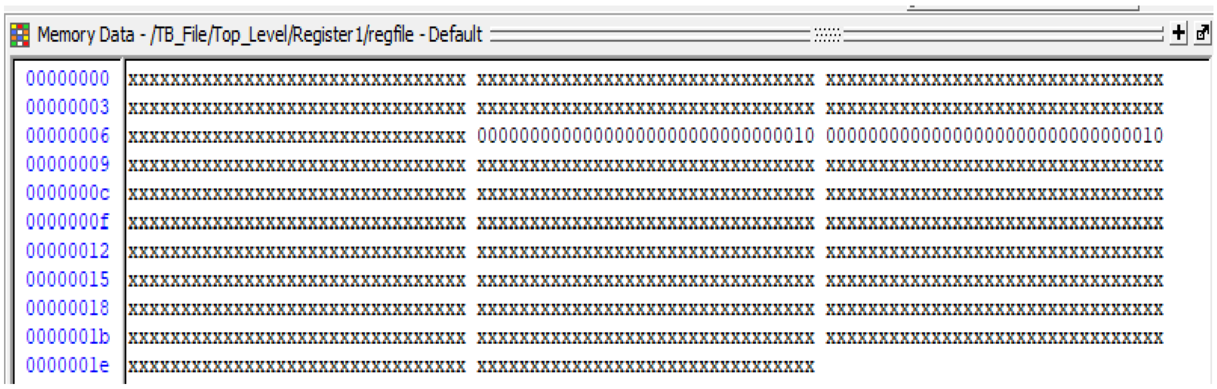
We therefore see that the instruction details that the data be written to address b'00000. First we can observe that the data located at b'00000 is initialised to zero, as shown above in figure 5.5. Furthermore, the instruction decodes a source register located in the register module at address b'01000. If we examine figure 5.4, it is shown that the value held at that address is indeed d'10. After 5 clock cycles, the signals will propagate fully through the processor and in the end, we will get the result shown in figure 5.6 where the value d'10 is written to address b'00000 in the data memory.



```
Memory Data - /TB_File/Top_Level/Data1/memfile - Default
00000000 00000000000000000000000000001010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.6 : Data Memory After Store Instruction

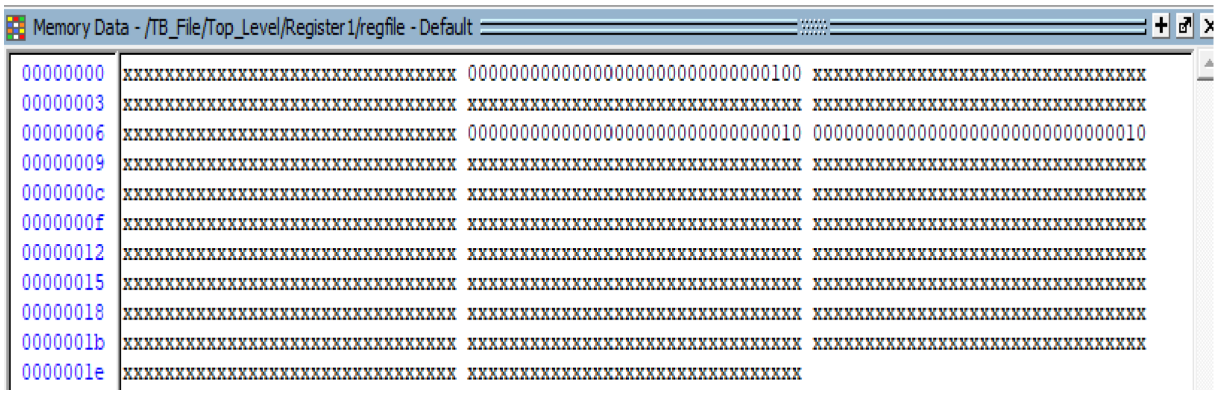
5.3 Add Simulation Results



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000010 00000000000000000000000000000010
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.7 : Register Contents Prior Addition

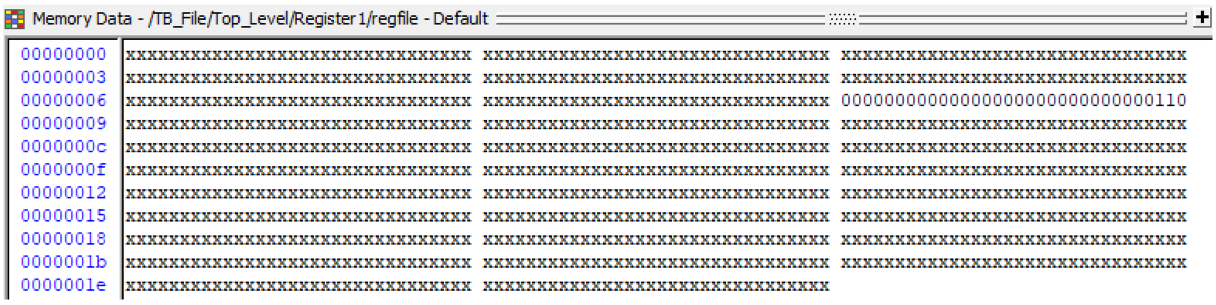
In figure 5.7, we see that register $T7 and $T8 are both initialised to d'4. The destination register $V1 located at address b'0001 is uninitialized and will remain so until the add operation is complete. After the result is computed by the ALU module and written through the EX/MEM and MEM/WB pipelines, it is written back to the destination register. Figure 5.8 shows that the value d'8 is indeed written to register $V1.



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000100 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000010 00000000000000000000000000000010
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.8 : Register Contents After Addition Operation

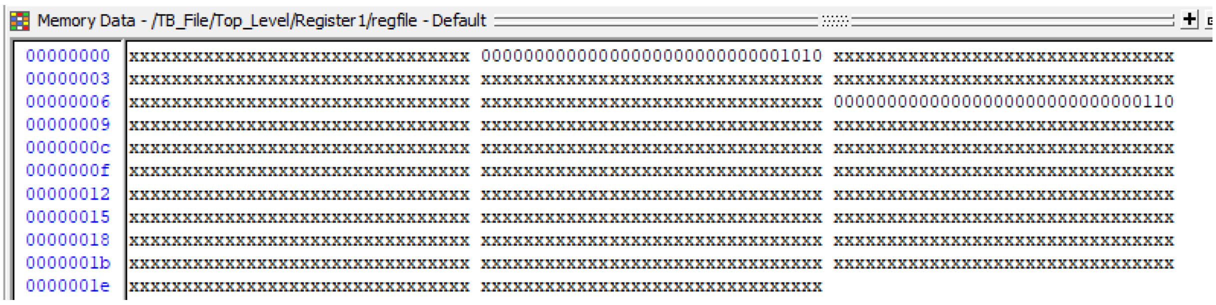
5.4 Add Immediate Simulation Results



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000110
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.9 : Register Contents After Initialization

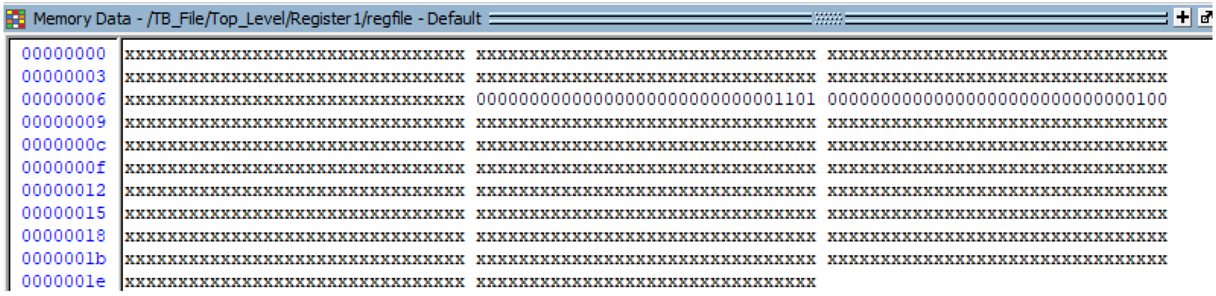
Similar to the operation performed in section 5.3, this operation will add the contents of the register in address b'1000 to the immediate value d'4. As specified in the r-type instruction h'21010004, the result of this add immediate operation is destined for register $V1 located at address h'0001. When we observe the contents of register $V1, we see that, after five clock cycles, the value d'10 is written to that address in the register module. This is because we are adding the d'6 located in $T8 and an immediate value of d'4. The immediate value is the final 16 bits in the instruction code and prior to addition, the value is sign extended to 32 bits and the mux located prior to the ALU is set to the results of the sign extension.



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000110
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.10 : Add Immediate Results

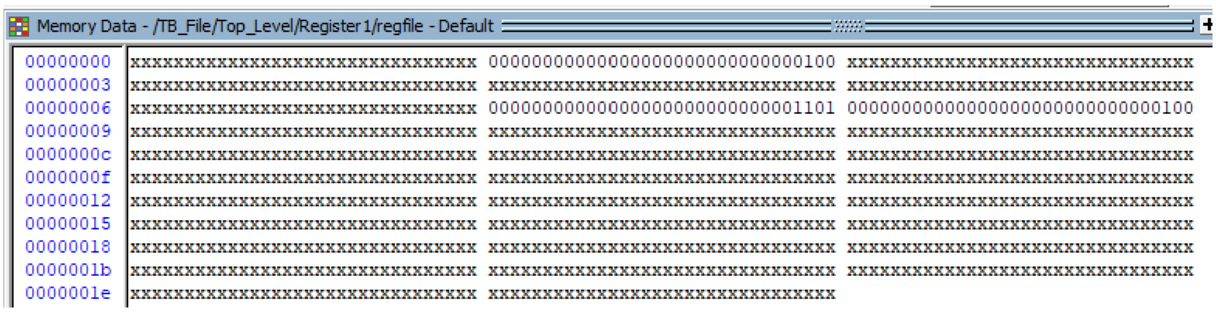
5.5 And Simulation Results



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001101 00000000000000000000000000000100
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.11: Registers $T7 and T8 With Initialized Values

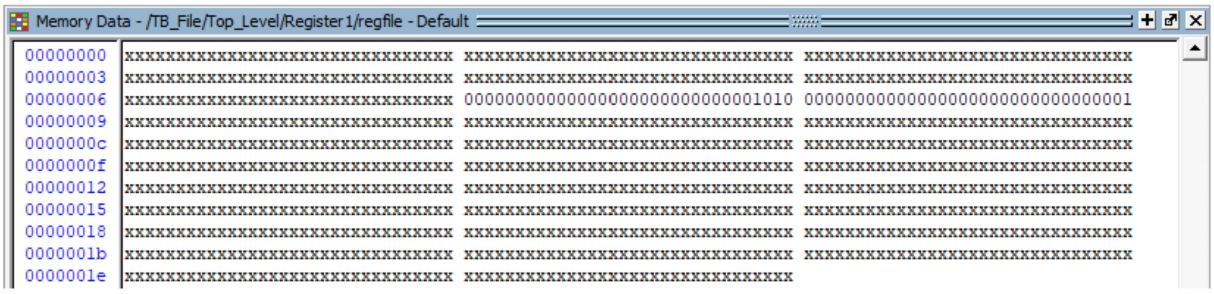
To test the operation of the ALU’s AND operation, the instruction h’00E80824 is passed from the instruction memory to the IF/ID stage pipeline. Here the instruction is decoded to an add instruction that will use values of $T7 and $T8 located at address h’0111 and h’1000 respectively. When the values of b’1101 and b’0100 are and together, we get a final 32 bit value of b'0100. Figure 5.12 shows that the result of the AND operation are indeed written to the register specified in the hex instruction which results in the value written to $V1 located at address h’0001.


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000100 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001101 00000000000000000000000000000100
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.12: Results of AND Operation Written To Destination Register

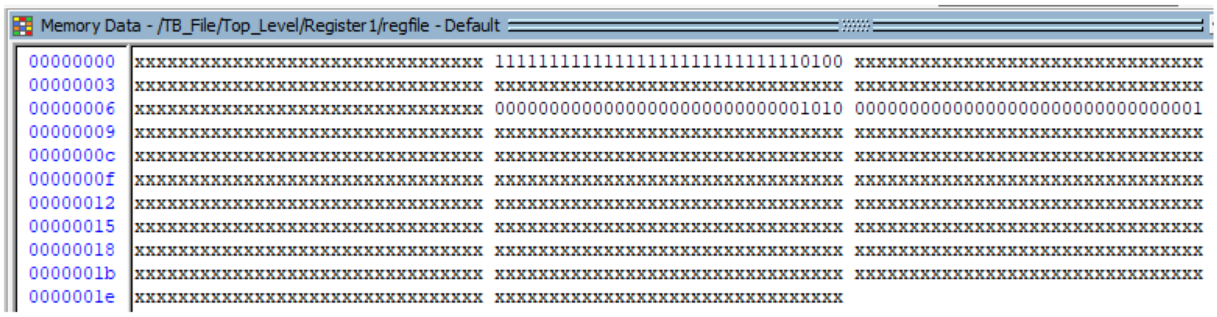
5.6 Nor Simulation Results


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 00000000000000000000000000000001
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.13: Register Contents Prior to Simulation of NOR Operation

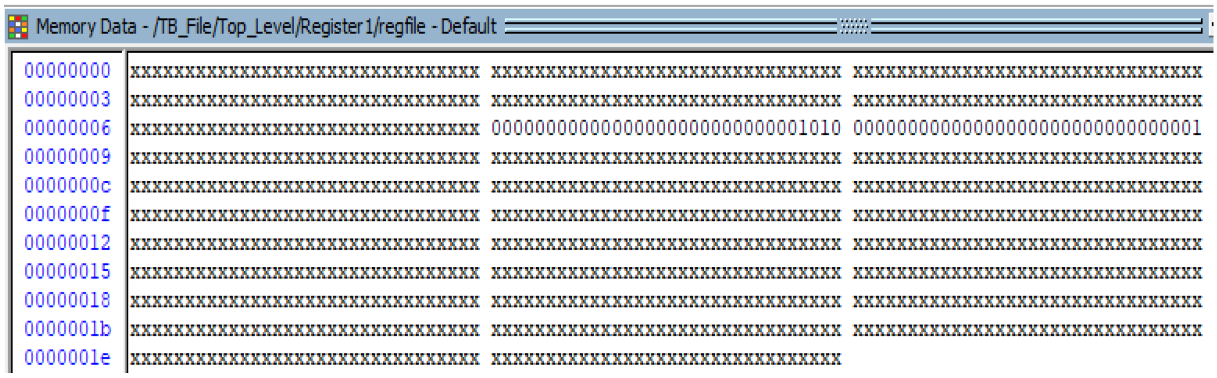
Similar to the previous operation, the instruction h’01070827 specifies a nor operation that will take the values located at addresses h’0110 and h’0111. The variable $T6 has a binary value of b’1010 and variable $T7 is initialised to the value b’0001, shown in figure 5.13. When we nor these values together, we expect to see the 32 bit value b’11111111111111111111111111110100 stored in the destination register located at address h’0001, which is labelled as $V1. After five Clock cycles, we can see that the expected value was written to the correct address confirming the operation of the ALU’s NOR calculation.



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 11111111111111111111111111110100 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 00000000000000000000000000000001
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.14: Nor Operation Results

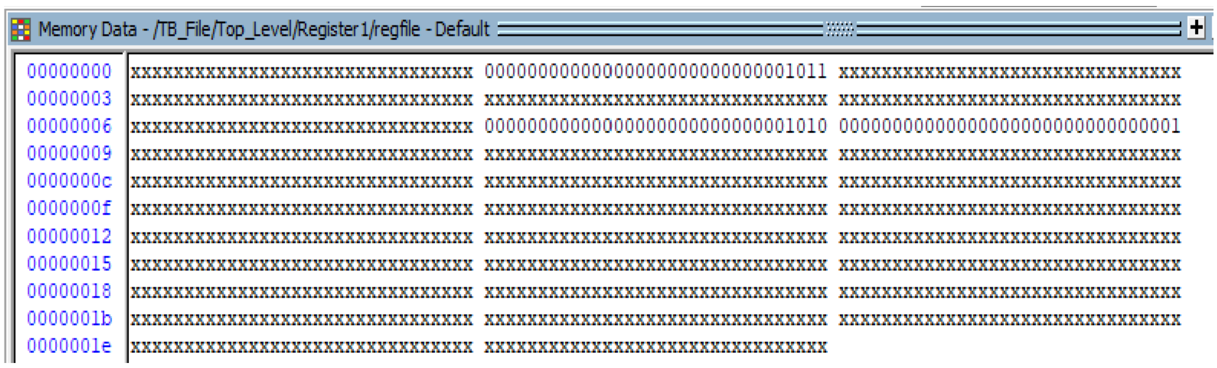
5.7 Or Simulation Results



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 00000000000000000000000000000001
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.15: Register Module Environment With Initialized Values

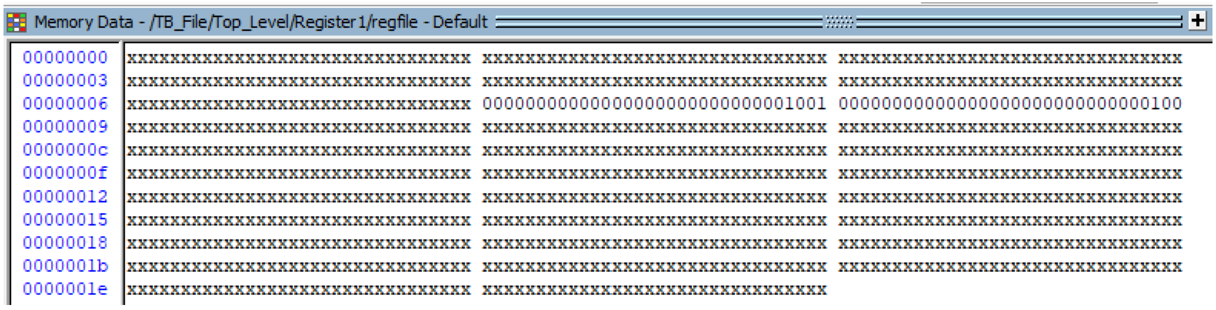
The next operation examined is the OR operation, which is specified with the instruction h'01070825. When decoded, we see that we are testing operation with the values b'1010, held in variable $T7, and $T8's value of b'0001. The contents of these variables is shown in figure 5.15. We expected a value of b'1011 to be stored in the variable $V1. After the values move through our pipelined processor, we see in figure 5.14 that the precalculated value is indeed stored in the correct variable address.



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001011 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 00000000000000000000000000000001
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.14: Register Module Updated with Or Simulation Results

5.8 Set-Less-Then Simulation Results

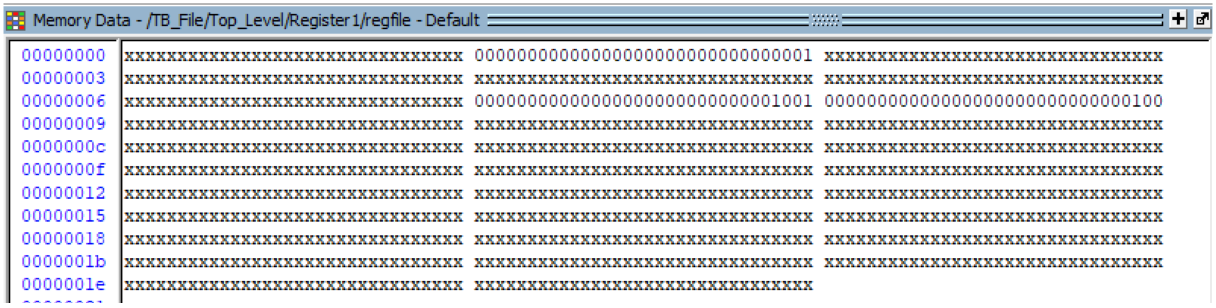


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001001 00000000000000000000000000000100
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.15: Contents of Register Module Prior to Set-Less-Then Operation

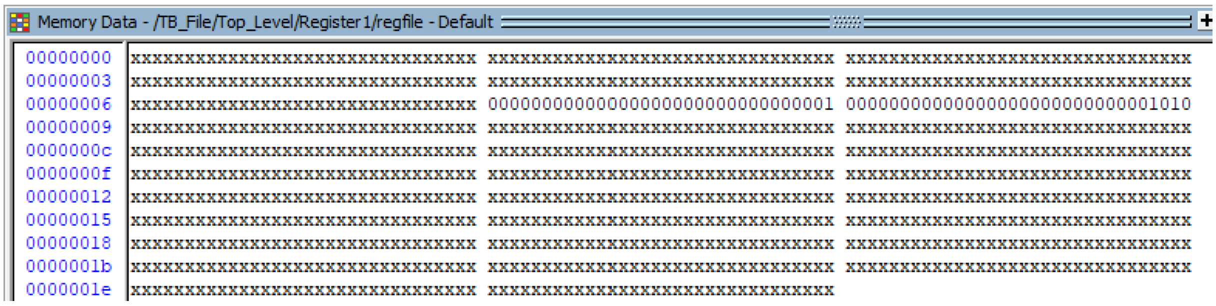
In order to fully test our processor's Set-Less-Than functionality, we ran the simulation for two separate scenarios from the instruction h'107082A. The first simulation will take two registers with variables $T8 and $T7. Aptly named for the register whose values we will use are specified in the instruction as the data located in register b'01000 and b'00111. In both cases we will be performing the comparison as $T8 < $T7 and storing the boolean value in register $V1 located at address b'0001 in the register module. First we initialise the value of $T8 and $T7 so that the comparison is true. In figure 5.15, we see that the variable $T8 is set to d'4 and variable $T7 is set to 9. Figure 5.16 confirms that we are indeed performing the Set-Less-Than operation, for the resulting value stored in register address b'00001 is the result of the comparison, and in this case, the result is a one or logical true.



Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000001 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001001 00000000000000000000000000000100
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.16 : Register Contents Post Set-Less-Then Operation

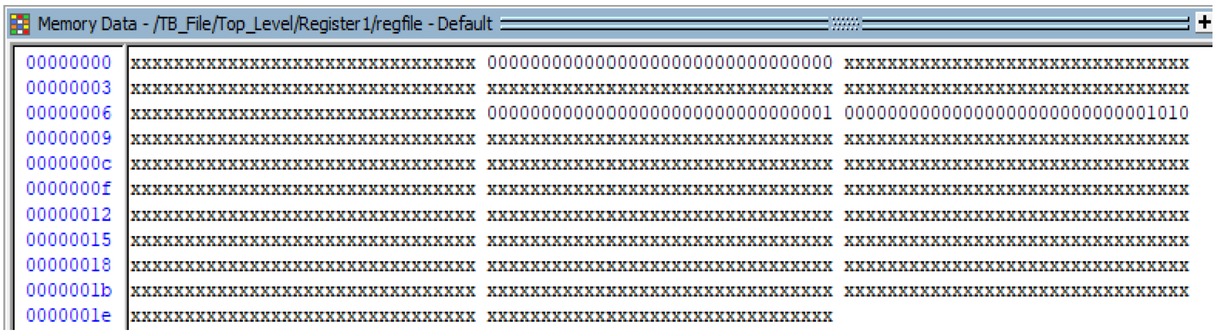


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000001 00000000000000000000000000001010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.17 : Registers $T7 and $T8 Initialized for Set-Less-Than Operation

The final step in testing the Set-Less-Than operation is to cause the comparison to return a false value or bit 0 result to the address in the same register as before b’00001. As in the previous test, the variables $T8 and $T7 hold the values upon which the comparison will be performed. In figure 5.18, the value at $T8’s address b’01000 is d’10. The value in $T7 is d’1 and we can see from figure 5.18 that this value is stored at address b’01000. The final result of zero was stored in variable $V1 found at address b’00001 in the register module.

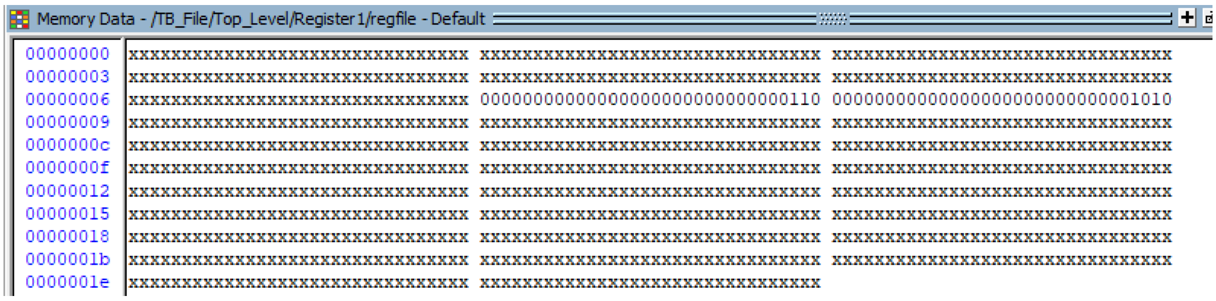


```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default

00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000001 00000000000000000000000000001010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.18 : Register $V1 With Results of Set-Less-Than Operation

5.9 Subtraction Simulation Results

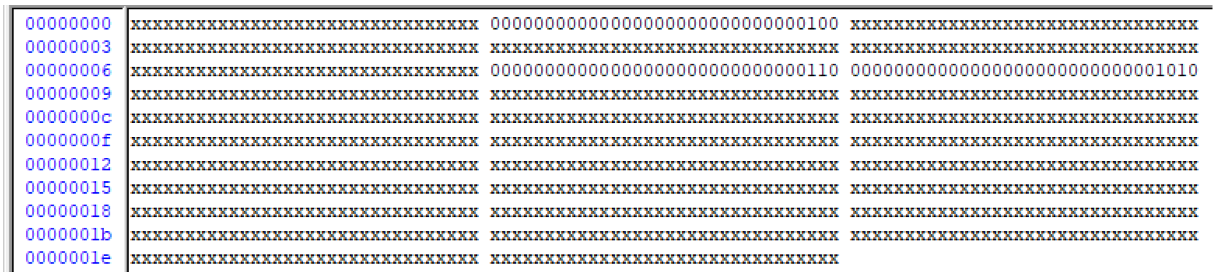


```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default

00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000110 00000000000000000000000000001010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.19: Registers $T7 and $T8 Initialized to 6 and 10

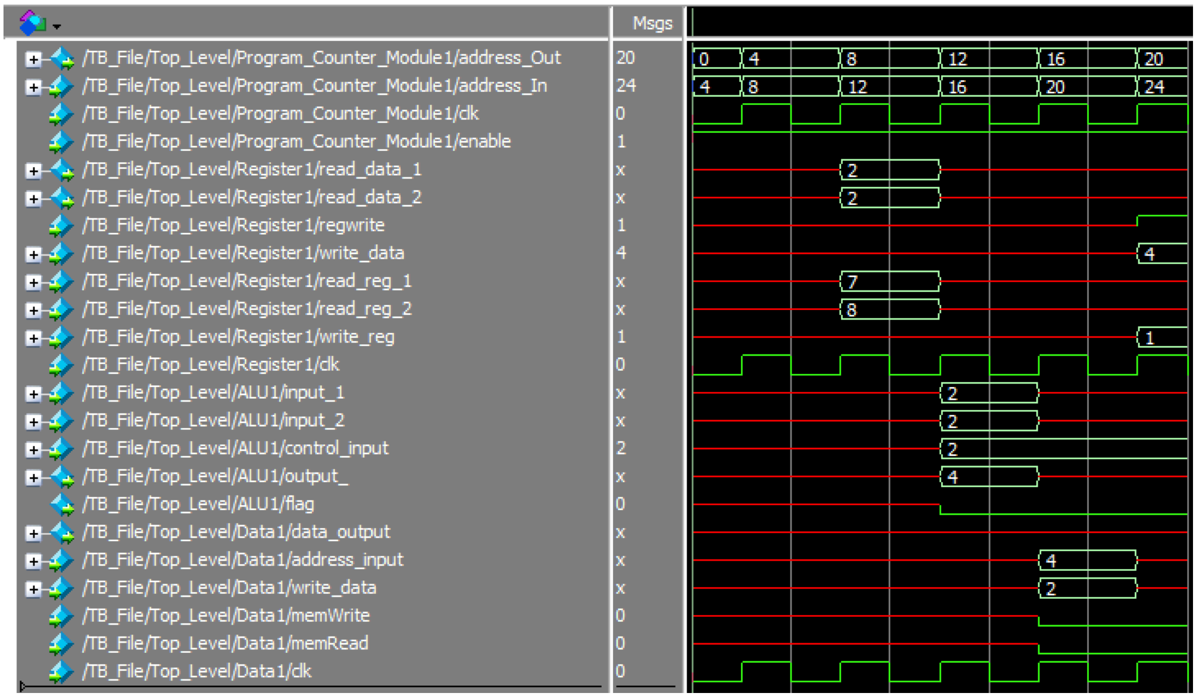
The Final function tested is the subtraction arithmetic operation. Testing the module with instruction h’01070822 we see in figure 5.19 that the registers $T7 and $T8 are set to values d’6 and d’10 respectively. After running our code for 5 clock cycles, we observed the result shown in figure 5.20 where the value d’6 is subtracted from d’10 and the value 4 was indeed stored in the instruction’s specified destination register located at address b’00001. This confirms that the simple subtraction is functioning as expected without error



```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000100 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000110 00000000000000000000000000001010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.20 : Results of Subtraction Written to Register $V1

The Final waveform for an Add operation was displayed as it can be seen in Figure 5.21. The Hex code 0x00E80820 is passed to the Instruction Memory and the complete implementation of each stage of the Top Module is allowed to execute. As a result, the complete waveform of a fully working Add operation using MIPS Architecture can be observed in Figure 5.21.

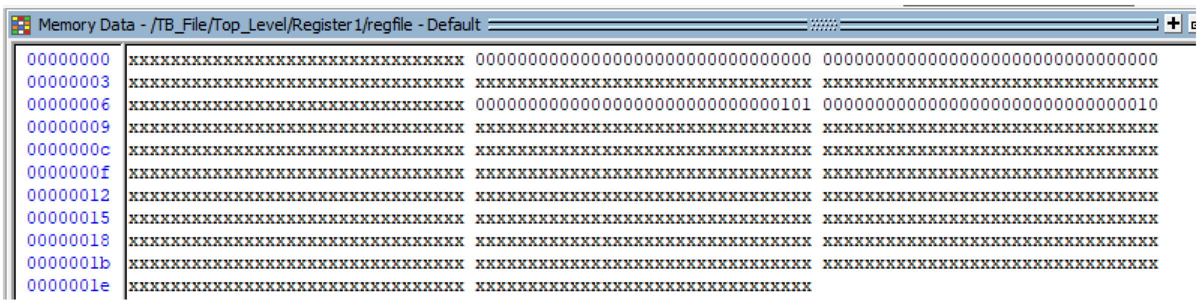


```
                                                        Msgs
/TB_File/Top_Level/Program_Counter_Module1/address_Out  20
/TB_File/Top_Level/Program_Counter_Module1/address_In   24
/TB_File/Top_Level/Program_Counter_Module1/clk          0
/TB_File/Top_Level/Program_Counter_Module1/enable       1
/TB_File/Top_Level/Register1/read_data_1                x
/TB_File/Top_Level/Register1/read_data_2                x
/TB_File/Top_Level/Register1/regwrite                   1
/TB_File/Top_Level/Register1/write_data                 4
/TB_File/Top_Level/Register1/read_reg_1                 x
/TB_File/Top_Level/Register1/read_reg_2                 x
/TB_File/Top_Level/Register1/write_reg                  1
/TB_File/Top_Level/Register1/clk                        0
/TB_File/Top_Level/ALU1/input_1                         x
/TB_File/Top_Level/ALU1/input_2                         x
/TB_File/Top_Level/ALU1/control_input                   2
/TB_File/Top_Level/ALU1/output_                         x
/TB_File/Top_Level/ALU1/flag                            0
/TB_File/Top_Level/Data1/data_output                    x
/TB_File/Top_Level/Data1/address_input                  x
/TB_File/Top_Level/Data1/write_data                     x
/TB_File/Top_Level/Data1/memWrite                       0
/TB_File/Top_Level/Data1/memRead                        0
/TB_File/Top_Level/Data1/clk                            0
```

Figure 5.21: Complete Waveform for Add Operation

Registers $7 and $8 were initialised with the value 2 as it can be seen in the second high clock signal and these 2 registers are the source register that are passed on as inputs to the ALU module. The ALU module performs Addition function on the values in these registers i.e. it adds 2 stored in $7 to the 2 stored in $8, to give the ALU output as 4. This output is then passed onto the register file where it gets stored back in Register $1.

5.10 Multiplication Simulation Results

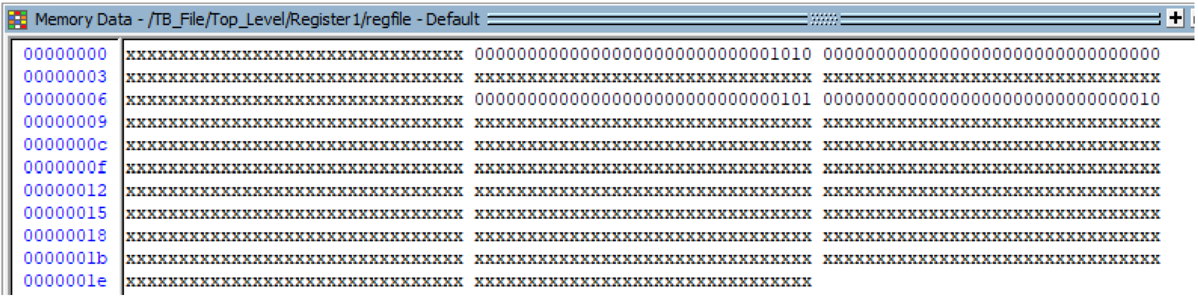


```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default

00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000000 00000000000000000000000000000000
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000101 00000000000000000000000000000010
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.22: Register File Prior to Multiplication

Figure 5.22 shows the contents of the register file initialised with the values d'5 held at address b'00111 and integer value d'2 stored at register address b'01000. In order to perform the multiplication instruction, the hex instruction code of h'01070018 was passed from instruction memory to the IF/ID pipeline register. Once decoded we see that we are multiplying the values held in $T7 and $T8. The result of this multiplication will not be stored to a destination register but stored with the Hi and Lo registers within the ALU module.


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 00000000000000000000000000000000
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000101 00000000000000000000000000000010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.23: Result of Multiplication Stored in Address b'00001

Since we are storing the values in the HI and LO registers, we require another instruction to move the values out of the HI/LO ALU registers. As shown in figure 5.24, after 3 clock cycles the value d'10 is stored in the LO register. Then when the PC counter increments to the next instruction, we get the instruction h'00000812 is passed into the pipeline. This instruction breaks down to a move from lo (MFLO) operation that will take the 32 bit value held in the ALU Lo register and store it in the address b'00001.

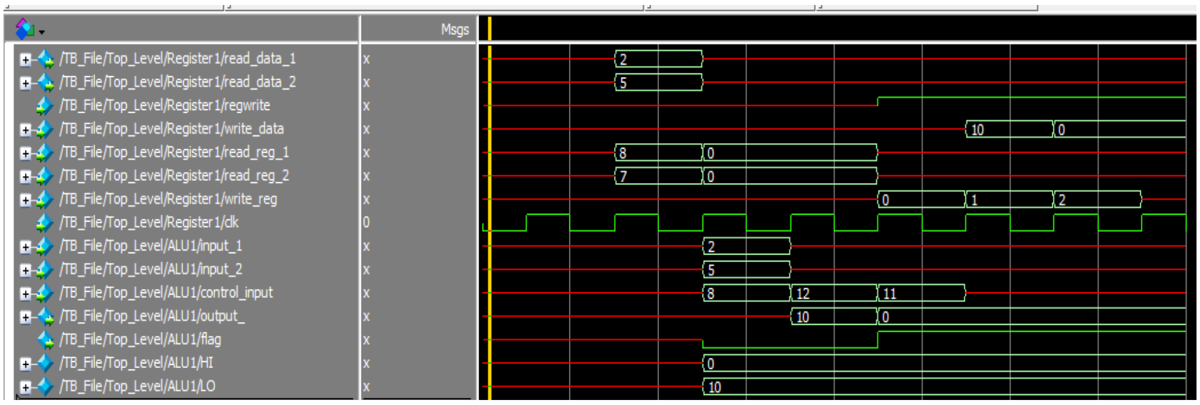
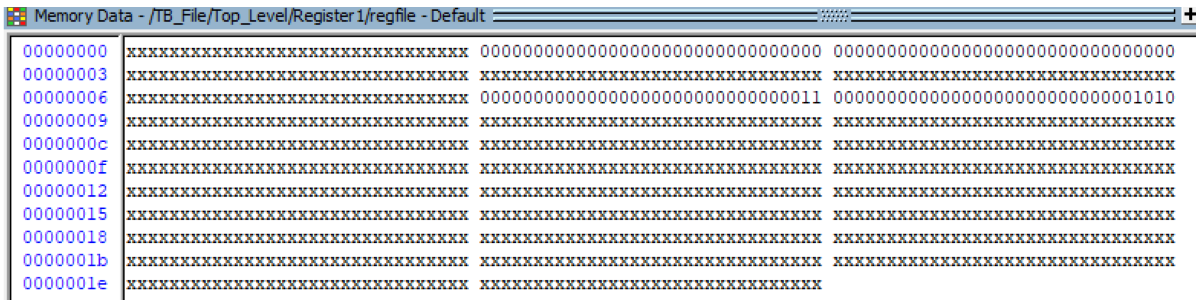

Msgs
/TB_File/Top_Level/Register1/read_data_1 x 2
/TB_File/Top_Level/Register1/read_data_2 x 5
/TB_File/Top_Level/Register1/regwrite x
/TB_File/Top_Level/Register1/write_data x 10 0
/TB_File/Top_Level/Register1/read_reg_1 x 8 0
/TB_File/Top_Level/Register1/read_reg_2 x 7 0
/TB_File/Top_Level/Register1/write_reg x 0 1 2
/TB_File/Top_Level/Register1/clk 0
/TB_File/Top_Level/ALU1/input_1 x 2
/TB_File/Top_Level/ALU1/input_2 x 5
/TB_File/Top_Level/ALU1/control_input x 8 12 11
/TB_File/Top_Level/ALU1/output_ x 10 0
/TB_File/Top_Level/ALU1/flag x
/TB_File/Top_Level/ALU1/HI x 0
/TB_File/Top_Level/ALU1/LO x 10

Figure 5.24: Multiplication Waveform

Furthermore, the next instruction to enter the pipeline is h'0000101 which translates to a move from high (MFHI) operation that will store the value stored in the high register and moves it to the destination register b'00010. Shown in figure 5.23, we see that the low value d'10 is indeed stored at address b'00001 the destination register of $V1 and the high value of d'0 is stored at address b'00010 which is the result register of $V2.

5.11 Division Simulation Results

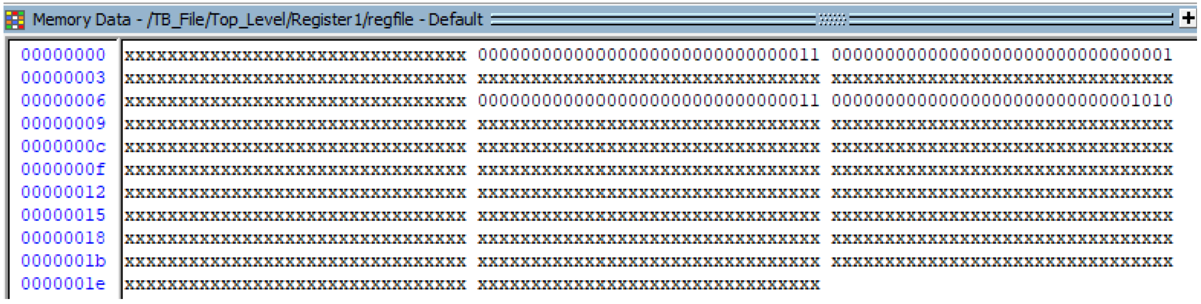


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000000 00000000000000000000000000000000
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000011 00000000000000000000000000001010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.25: Registers Initialised Prior to Division Operation

The division instruction comes into the pipeline with the value h'0107001A. This instruction will be broken down into a division operation that will divide the integer value d'3 stored at address location b'00111 and the value d'10 stored at register address b'01000



Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000011 00000000000000000000000000000001
00000003  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000011 00000000000000000000000000001010
00000009  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.26: Register Contents Post Division

As we can see in figure 5.26, as we can see the value stored in address b'00001 is d'3 which is the result we expect when dividing 10 by 3 and the remainder of d'1 is stored at register b'00010. The values stored in these registers come from the high and lo registers as shown in figure 5.27.

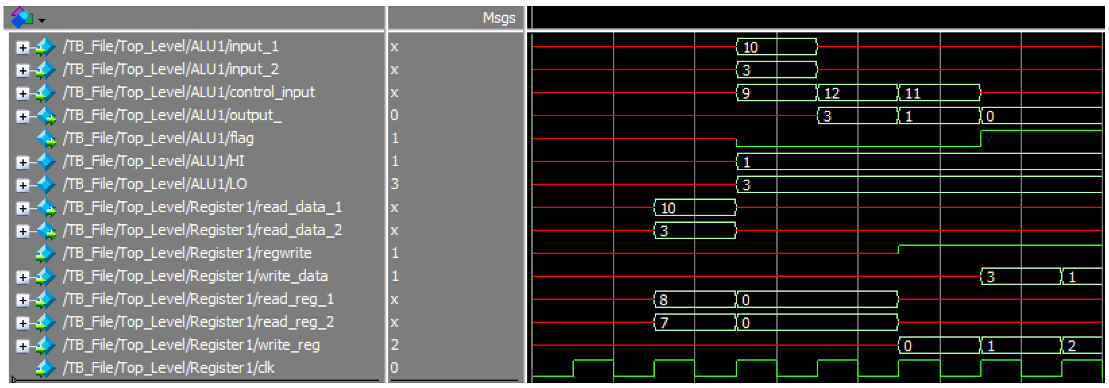
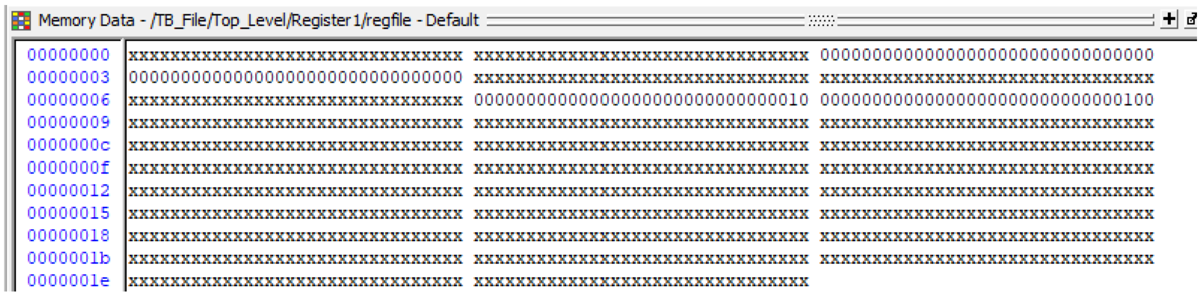


Figure 5.27: High and Low Register Waveforms

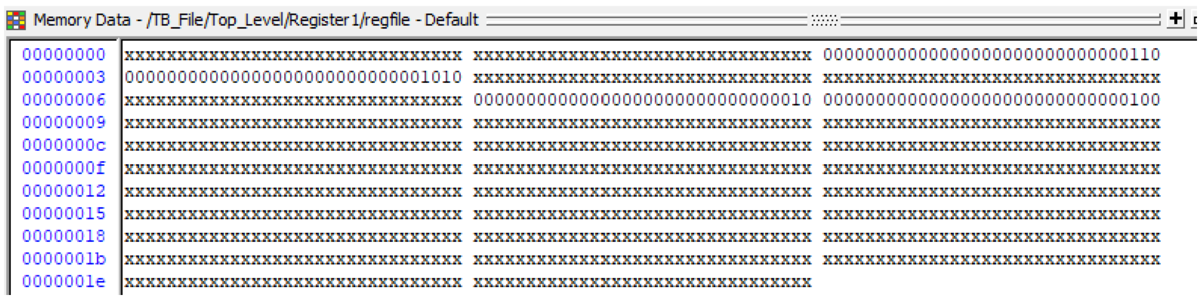
5.12 Forwarding Unit Simulation Results



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000000
00000003 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000010 00000000000000000000000000000100
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.28: Register Module Before Forwarding

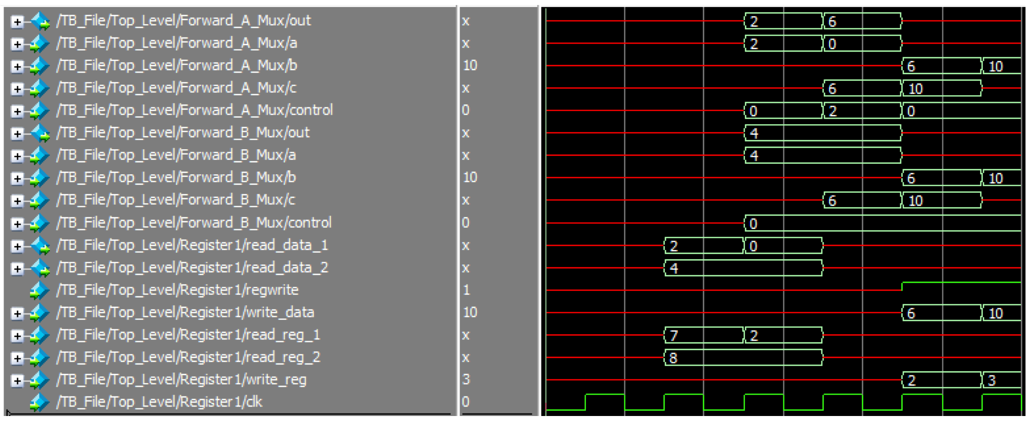
In order to test the forwarding, we placed instructions in the instruction memory. The first instruction will store the values held in registers $T7 and $T8 and store the value in register $T2. If the forwarding operated as expected we should see a value of d’6 written to register $T3



```
Memory Data - /TB_File/Top_Level/Register1/regfile - Default
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000110
00000003 00000000000000000000000000001010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000010 00000000000000000000000000000100
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.29: Register Contents After Forwarding

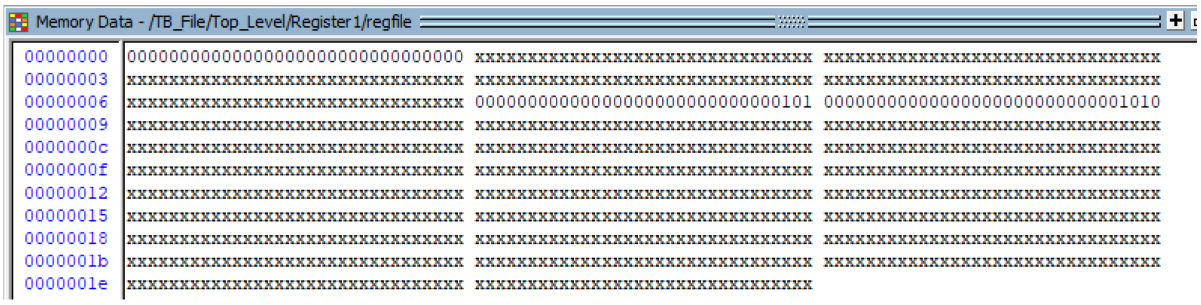
As shown in figure 5.29, we indeed stored the value d’6 into register $T3 located at address b’00110. In the figure below, we see the forward unit check whether the next instruction in the pipeline requires data that has not yet been returned. In figure 5.30, we see that the forwarding unit detects the need to forward and forwards it to the next instruction.



```
/TB_File/Top_Level/Forward_A_Mux/out          x
/TB_File/Top_Level/Forward_A_Mux/a            x
/TB_File/Top_Level/Forward_A_Mux/b            10
/TB_File/Top_Level/Forward_A_Mux/c            x
/TB_File/Top_Level/Forward_A_Mux/control      0
/TB_File/Top_Level/Forward_B_Mux/out          x
/TB_File/Top_Level/Forward_B_Mux/a            x
/TB_File/Top_Level/Forward_B_Mux/b            10
/TB_File/Top_Level/Forward_B_Mux/c            x
/TB_File/Top_Level/Forward_B_Mux/control      0
/TB_File/Top_Level/Register1/read_data_1      x
/TB_File/Top_Level/Register1/read_data_2      x
/TB_File/Top_Level/Register1/regwrite         1
/TB_File/Top_Level/Register1/write_data       10
/TB_File/Top_Level/Register1/read_reg_1       x
/TB_File/Top_Level/Register1/read_reg_2       x
/TB_File/Top_Level/Register1/write_reg        3
/TB_File/Top_Level/Register1/clk              0
```

Figure 5.30: Forwarding Unit Waveform

5.13 Hazard Unit Simulation Results

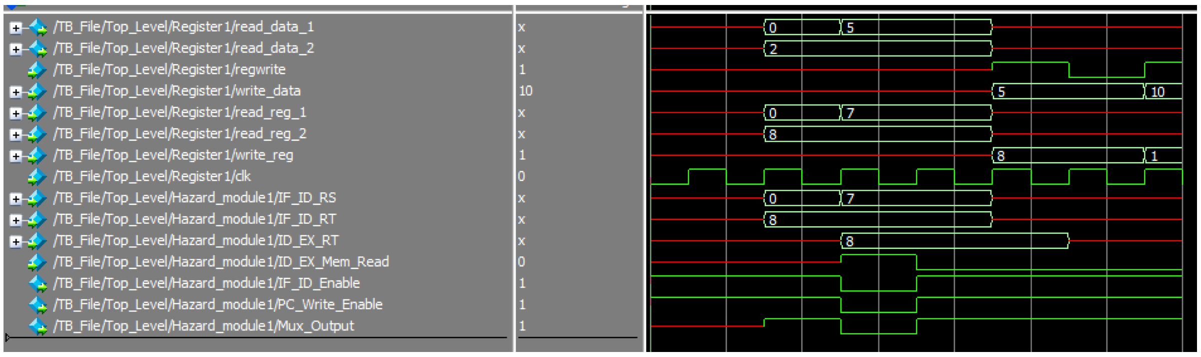


Memory Data - /TB_File/Top_Level/Register1/regfile

```
00000000 00000000000000000000000000000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000101 00000000000000000000000000001010
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.31: Register Module Before Hazarding

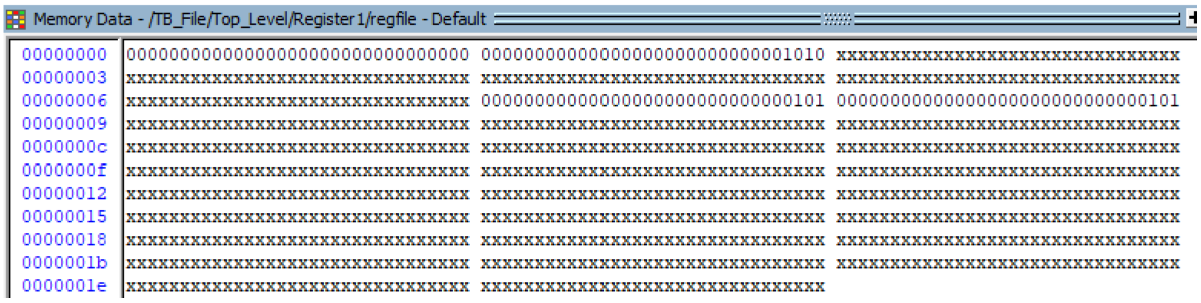
In order to test the hazard unit, we placed a load instruction into the pipeline of h'8C080000 and a following instruction of h'00E80820. This will create a load data hazard because the first instruction is a load instruction that is loading the value stored at address b'00000 to variable $T8 located at address b'01000. The instruction following the load is a simple add instruction.



```
/TB_File/Top_Level/Register1/read_data_1            x
/TB_File/Top_Level/Register1/read_data_2            x
/TB_File/Top_Level/Register1/regwrite               1
/TB_File/Top_Level/Register1/write_data             10
/TB_File/Top_Level/Register1/read_reg_1             x
/TB_File/Top_Level/Register1/read_reg_2             x
/TB_File/Top_Level/Register1/write_reg              1
/TB_File/Top_Level/Register1/clk                    0
/TB_File/Top_Level/Hazard_module1/IF_ID_RS          x
/TB_File/Top_Level/Hazard_module1/IF_ID_RT          x
/TB_File/Top_Level/Hazard_module1/ID_EX_RT          x
/TB_File/Top_Level/Hazard_module1/ID_EX_Mem_Read    0
/TB_File/Top_Level/Hazard_module1/IF_ID_Enable      1
/TB_File/Top_Level/Hazard_module1/PC_Write_Enable   1
/TB_File/Top_Level/Hazard_module1/Mux_Output        1
```

Figure 5.32: Hazard Unit Detection in Waveform

This instruction requires the use of the value held in variable $T8. As shown in the, in figure 5.32, we are detecting a need to handle a hazard and the pc write enable goes to zero to introduce a bubble into the pipeline. Figure 5.33 shows that the correct value was stored in register address b'01000 of d'5.

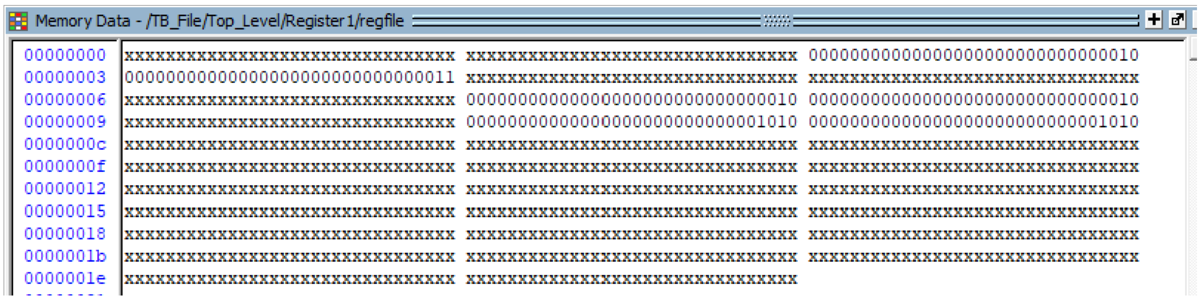


Memory Data - /TB_File/Top_Level/Register1/regfile - Default

```
00000000 00000000000000000000000000000000 00000000000000000000000000001010 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000003 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000101 00000000000000000000000000000101
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.33: Register Module After Hazard Detection

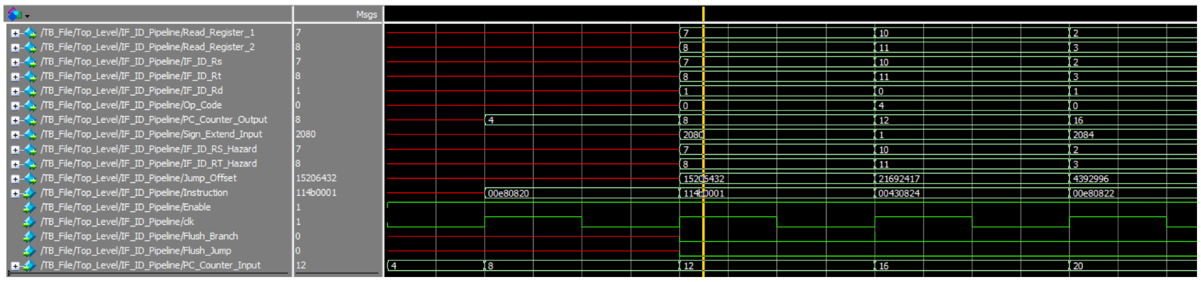
5.14 Branch If Equal Simulation Results



```
Memory Data - /TB_File/Top_Level/Register1/regfile
00000000 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000010
00000003 00000000000000000000000000000011 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000006 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000000010 00000000000000000000000000000010
00000009 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 00000000000000000000000000001010 00000000000000000000000000001010
0000000c xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000000f xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000012 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000015 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
00000018 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001b xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
0000001e xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Figure 5.35: Register Module Before Branch Instruction

In order to test if our branch if equal instruction is functioning correctly, we placed two instructions following the branch check. The contents of $t10 at address b'01010 and $t11 stored at b'1011. Since the values held there are correct, we will skip the and instruction in our pipeline and instead we will jump to instruction 12 which will perform a subtraction on the contents of address $T7 and $T8 and store it in to register $t3.

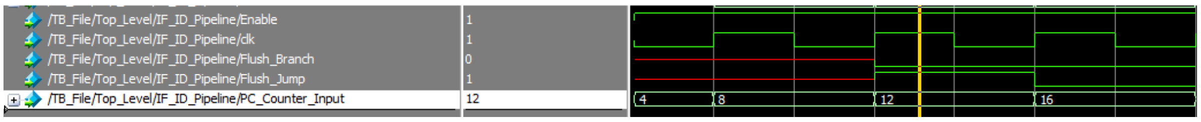


```
                                                   Msgs
/TB_File/Top_Level/IF_ID_Pipeline/Read_Register_1  7
/TB_File/Top_Level/IF_ID_Pipeline/Read_Register_2  8
/TB_File/Top_Level/IF_ID_Pipeline/IF_ID_Rs         7
/TB_File/Top_Level/IF_ID_Pipeline/IF_ID_Rt         8
/TB_File/Top_Level/IF_ID_Pipeline/IF_ID_Rd         1
/TB_File/Top_Level/IF_ID_Pipeline/Op_Code          0
/TB_File/Top_Level/IF_ID_Pipeline/PC_Counter_Output 8
/TB_File/Top_Level/IF_ID_Pipeline/Sign_Extend_Input 2080
/TB_File/Top_Level/IF_ID_Pipeline/IF_ID_RS_Hazard  7
/TB_File/Top_Level/IF_ID_Pipeline/IF_ID_RT_Hazard  8
/TB_File/Top_Level/IF_ID_Pipeline/Jump_Offset      15206432
/TB_File/Top_Level/IF_ID_Pipeline/Instruction      114b0001
/TB_File/Top_Level/IF_ID_Pipeline/Enable           1
/TB_File/Top_Level/IF_ID_Pipeline/clk              1
/TB_File/Top_Level/IF_ID_Pipeline/Flush_Branch     0
/TB_File/Top_Level/IF_ID_Pipeline/Flush_Jump       0
/TB_File/Top_Level/IF_ID_Pipeline/PC_Counter_Input 12
```

Figure 5.36: Branch If Equal Waveform

When we observe the waveform shown in figure 5.36, we see that the address is updated from the next instruction to the one after that one and the flush signal goes to zero indicating that a branch instruction did occur.

5.15 Jump Instruction Simulation Results

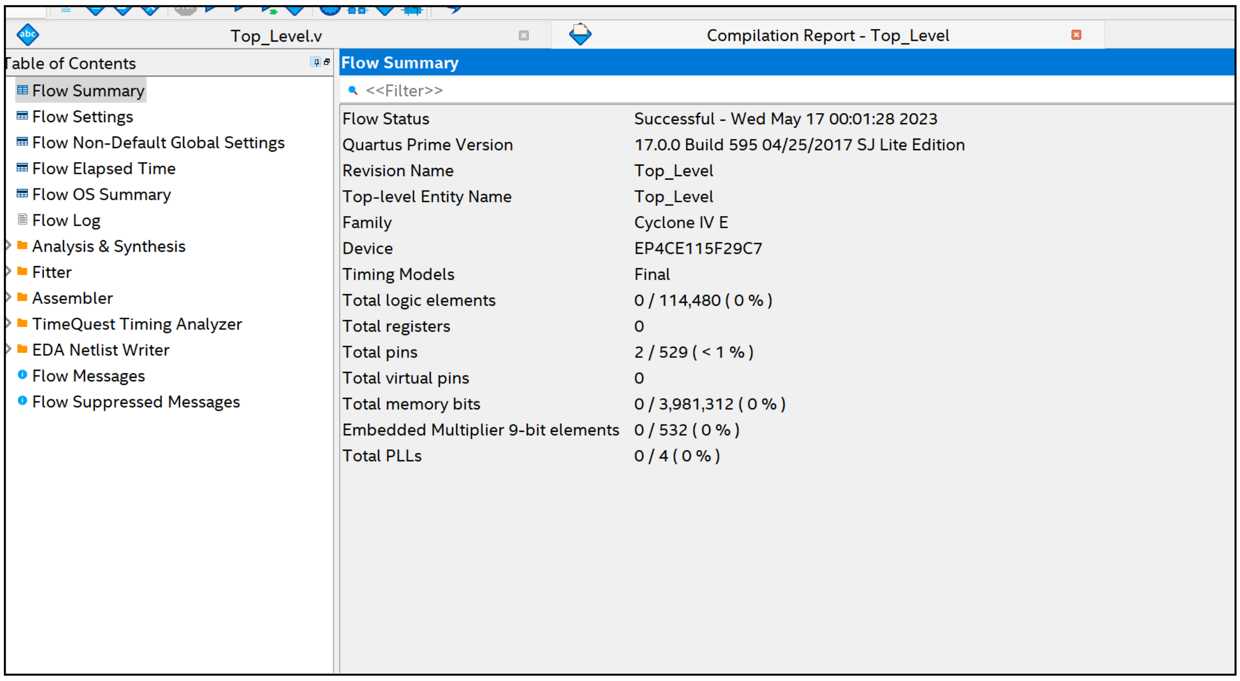


```
/TB_File/Top_Level/IF_ID_Pipeline/Enable           1
/TB_File/Top_Level/IF_ID_Pipeline/clk              1
/TB_File/Top_Level/IF_ID_Pipeline/Flush_Branch     0
/TB_File/Top_Level/IF_ID_Pipeline/Flush_Jump       1
/TB_File/Top_Level/IF_ID_Pipeline/PC_Counter_Input 12
```

Figure 5.38: Jump Instruction Waveform

The hex instruction detailing a jump is the hex value h'8000003. We can see from the waveform in figure 5.38, we see that flush jump has gone high and the flush branch goes low. This indicates that our jump is performing as expected.

5.16 Synthesization Results



Top_Level.v

Compilation Report - Top_Level

Table of Contents

- Flow Summary
- Flow Settings
- Flow Non-Default Global Settings
- Flow Elapsed Time
- Flow OS Summary
- Flow Log
- Analysis & Synthesis
- Fitter
- Assembler
- TimeQuest Timing Analyzer
- EDA Netlist Writer
- Flow Messages
- Flow Suppressed Messages

Flow Summary

<<Filter>>

Flow Status	Successful - Wed May 17 00:01:28 2023
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	Top_Level
Top-level Entity Name	Top_Level
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	0 / 114,480 (0 %)
Total registers	0
Total pins	2 / 529 (< 1 %)
Total virtual pins	0
Total memory bits	0 / 3,981,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 5.39: Synthesization Results

The synthesization results were unexpected. We had previously synthesised our project with block ram appearing in this report but as shown in figure 5.39, we must have produced an error that caused the block ram to stop synthesising.

6. Conclusion

Through this project we were able to design and implement a processor on Modelsim that would run a program that is based off the Microprocessor with Interlocked Pipeline Stages (MIPS) instruction to perform Arithmetic/Logic, Data Movement and Flow Control instructions depending on the instruction passed in by the user. For the design we implemented a 5 stage pipelined datapath to improve execution time and included hazard detection and Forwarding units to avoid Data and Control hazards. The complete pipelined cycle can be seen in Figure 4.10 and it was constructed through Verilog coded modules for each stage of the instruction and a Top Module connecting all the modules together for a complete cycle was also constructed. As a result the complete waveform (Figure 5.21) of a fully operational MIPS ISA was fabricated.

7. References

Patterson, David, A. and John L. Hennessy. Computer Organization and Design MIPS Edition. Available from: California State University - Fresno, (6th Edition). Elsevier S & T, 2020.

8. Appendices

Appendix A

```
module Adder_32_bit(input [31:0] a,b,output reg [31:0] out);
always@(a,b)
begin
        out = a + b;
end
Endmodule
```

Appendix B

```
module ALU_Module(input_1, input_2, output_, control_input,flag);
input [31:0] input_1, input_2;
input [3:0] control_input;
output reg [31:0] output_;
output reg flag;
reg [31:0] HI, LO;
always@( input_1, input_2,control_input)
begin
case(control_input)
0 :
   output_ = input_1 & input_2;
1:
output_ = input_1 | input_2;
2: output_ = input_1 + input_2;
6: output_ = input_1 - input_2;
7 : output_ = (input_1 < input_2 ? 1 : 0);
8:
   begin
   {HI, LO} = input_1 * input_2;
   end
9:
   begin
   if( ~(input_2 == 32'd0) )
   begin
   HI = input_1 % input_2;
   LO =  input_1 / input_2;
   end
   end
10 : output_ = ~(input_1 | input_2);
11: output_ = HI;
12: output_ = LO;
default : output_ = 0;

endcase

if (output_ == 0)
   begin
      flag = 1;
   end
else
   begin
   flag = 0;
   end

end
endmodule
```

Appendix C

```
module ALU_Module(input_1, input_2, output_, control_input,flag);

input [31:0] input_1, input_2;
input [3:0] control_input;

output reg [31:0] output_;
output reg flag;

reg [31:0] HI, LO;

always@( input_1, input_2,control_input)

begin

case(control_input)

0 :
   output_ = input_1 & input_2;
1:
output_ = input_1 | input_2;
2: output_ = input_1 + input_2;
6: output_ = input_1 - input_2;
7 : output_ = (input_1 < input_2 ? 1 : 0);
8:
   begin

   {HI, LO} = input_1 * input_2;

   end

9:
   begin
   if( ~(input_2 == 32'd0) )
   begin
   HI = input_1 % input_2;
   LO =  input_1 / input_2;
   end
   end
10 : output_ = ~(input_1 | input_2);

11: output_ = HI;

12: output_ = LO;
default : output_ = 0;
```

```
endcase

if (output_ == 0)
   begin
      flag = 1;
   end
else
   begin
   flag = 0;
   end

end

Endmodule
```

Appendix D

```
module AND_Gate(output out , input a ,b);
always@(*)begin
out = a &b;
end
endmodule
```

Appendix E

```
module Comparator( out , a , b);
output reg out;
input [31:0] a , b;
always@(*)begin
        if( a == b)
        begin
        out = 1;
        end
        else
        begin
        out = 0;
        end
end
endmodule
```

Appendix F

```
module Concatenation_28_32(out, a , b);
output reg [31:0] out;
input [27:0] a;
input [31:0] b;
always@(*)
begin
             out[31: 28] =  b [31: 28];
            out [27:0]   = a ;
end
endmodule
```

Appendix G

```
module Data_Memory_Module(
  output reg [31:0] data_output,
  input [31:0] address_input,
  input [31:0] write_data,
  input memWrite,
  input memRead,
  input clk
);

  reg [31:0] memfile[0:31];
initial
begin
memfile [32'd0] = 32'd10;
end
always@(negedge clk)
            begin
                        if (memWrite == 1)
                                    begin
                                                memfile[address_input] = write_data;
                        end
  end
always@(negedge clk)
  begin
                        if( memRead == 1)
                        begin
                                                data_output = memfile[address_input];
                        end
  end
endmodule
```

Appendix H

```
module EX_MEM_Pipeline_Module (
output reg RegWrite_Output, MemtoReg_Output, MemRead_Output, MemWrite_Output,
output reg [4:0] EX_MEM_RD_Fowarding, EX_MEM_RD_Next_Pipeline,
output reg [31:0] ALU_Result_Output,
output reg [31:0] RT_32_Bit_Output,
input RegWrite_Input, MemtoReg_Input, MemRead_Input, MemWrite_Input,
input [4:0] ID_EX_MUX,
input [31:0] ALU_Result_Input,
input [31:0] RT_32_Bit_Input,
input clk
 );
always@(posedge clk)begin
        RegWrite_Output = RegWrite_Input;
        MemtoReg_Output = MemtoReg_Input;
        MemRead_Output = MemRead_Input;
        MemWrite_Output = MemWrite_Input;
        EX_MEM_RD_Fowarding = ID_EX_MUX;
        EX_MEM_RD_Next_Pipeline = ID_EX_MUX;
        ALU_Result_Output = ALU_Result_Input;
        RT_32_Bit_Output = RT_32_Bit_Input;
end
endmodule
```

Appendix I

```
module Forward_Unit_Module(forward_A ,forward_B, ID_EX_RS, ID_EX_RT, EX_MEM_RD ,
MEM_WB_RD, EX_MEM_Reg_Write, MEM_WB_Reg_Write);
input [4:0] ID_EX_RS, ID_EX_RT , EX_MEM_RD, MEM_WB_RD;
input EX_MEM_Reg_Write, MEM_WB_Reg_Write;
output reg [1:0] forward_A, forward_B;
always@(ID_EX_RS, EX_MEM_RD, MEM_WB_RD)
begin
        if( ((ID_EX_RS == EX_MEM_RD) & (EX_MEM_Reg_Write == 1))| ((ID_EX_RS ==
EX_MEM_RD)& (ID_EX_RS == MEM_WB_RD) & (EX_MEM_Reg_Write == 1)) )
                begin
                        forward_A = 2'b10;
                end
        else if( (ID_EX_RS == MEM_WB_RD) & (MEM_WB_Reg_Write == 1) )
                begin
                        forward_A = 2'b01;
                end
        else
                        forward_A = 2'b00;
end
always@(ID_EX_RT, EX_MEM_RD , MEM_WB_RD)
begin
        if( ((ID_EX_RT == EX_MEM_RD) & (EX_MEM_Reg_Write == 1)) | (
(ID_EX_RT==EX_MEM_RD) & (ID_EX_RT == MEM_WB_RD) & (EX_MEM_Reg_Write == 1)
))
                begin
                        forward_B = 2'b10;
                end
        else if( (ID_EX_RT == MEM_WB_RD)  & (MEM_WB_Reg_Write == 1))
                begin
                        forward_B = 2'b01;
                end
        else
                        forward_B = 2'b00;
end
endmodule
```

Appendix J

```
module Hazard_Unit_Module (PC_Write_Enable,IF_ID_Enable, Mux_Output, ID_EX_Mem_Read ,
IF_ID_RS, IF_ID_RT, ID_EX_RT);
input [4:0] IF_ID_RS, IF_ID_RT, ID_EX_RT;
input ID_EX_Mem_Read;
output reg IF_ID_Enable;
output reg PC_Write_Enable;
output reg Mux_Output;

initial
begin
        IF_ID_Enable = 1;
        PC_Write_Enable =1;
end

always@(*)
begin
        if( ( (ID_EX_Mem_Read==1) & ((ID_EX_RT == IF_ID_RT) | (ID_EX_RT == IF_ID_RS) )
) )
                begin
                        Mux_Output = 0;
                        IF_ID_Enable= 0;
                        PC_Write_Enable = 0;
                end
        else
                begin
                        Mux_Output = 1;
                        IF_ID_Enable = 1;
                        PC_Write_Enable = 1;
                end
end
endmodule
```

Appendix K

```
module ID_EX_Pipeline_Module(ALU_Src_Out, Regwrite_Output, MemtoReg_Output,
MemRead_Output, MemWrite_Output, RegDst_Output, ALUOp_Output,Sign_Extend_Output,
ID_EX_Rs_Fowarding, ID_EX_Rt_Fowarding, ID_EX_Rt_MUX,ID_EX_Rd_MUX,
Read_Data_1_Output, Read_Data_2_Output, Regwrite_Input, MemtoReg_Input, MemRead_Input,
MemWrite_Input, RegDst_Input, ALUOp_Input, Sign_Extend_Input, IF_ID_Rs,
IF_ID_Rt,IF_ID_Rd, Read_Data_1_Input, Read_Data_2_Input,clk , ALU_Src_In);
output reg Regwrite_Output, MemtoReg_Output, MemRead_Output, MemWrite_Output,
RegDst_Output,ALU_Src_Out;
output reg [1:0] ALUOp_Output;
output reg [31:0] Sign_Extend_Output;
output reg [4:0] ID_EX_Rs_Fowarding, ID_EX_Rt_Fowarding,
ID_EX_Rt_MUX,ID_EX_Rd_MUX;
output reg [31:0] Read_Data_1_Output, Read_Data_2_Output;
input Regwrite_Input, MemtoReg_Input,  MemRead_Input, MemWrite_Input, RegDst_Input,
ALU_Src_In;
input [1:0] ALUOp_Input;
input [31:0] Sign_Extend_Input;
input [4:0] IF_ID_Rs, IF_ID_Rt,IF_ID_Rd;
input [31:0] Read_Data_1_Input, Read_Data_2_Input;
input clk;

always@(posedge clk)begin
        Regwrite_Output = Regwrite_Input;
        MemtoReg_Output = MemtoReg_Input;
        MemRead_Output = MemRead_Input;
        MemWrite_Output = MemWrite_Input;
        RegDst_Output = RegDst_Input;
        ALUOp_Output = ALUOp_Input;
        Sign_Extend_Output = Sign_Extend_Input;
        ID_EX_Rs_Fowarding = IF_ID_Rs;
        ID_EX_Rt_Fowarding = IF_ID_Rt;
        ID_EX_Rt_MUX = IF_ID_Rt;
        ID_EX_Rd_MUX = IF_ID_Rd;
        Read_Data_1_Output = Read_Data_1_Input;
        Read_Data_2_Output = Read_Data_2_Input;
        ALU_Src_Out = ALU_Src_In;
end
endmodule
```

Appendix L

```
module IF_ID_Pipeline_Module(Jump_Offset, IF_ID_RS_Hazard, IF_ID_RT_Hazard,
PC_Counter_Output, Op_Code, Read_Register_1, Read_Register_2, IF_ID_Rs, IF_ID_Rt,
IF_ID_Rd ,Sign_Extend_Input, Enable, Instruction, clk, PC_Counter_Input, Flush_Jump,
Flush_Branch);
output reg [4:0] Read_Register_1, Read_Register_2, IF_ID_Rs, IF_ID_Rt, IF_ID_Rd;
output reg [5:0] Op_Code;
output reg [31:0] PC_Counter_Output;
output reg [15:0] Sign_Extend_Input;
output reg [4:0] IF_ID_RS_Hazard, IF_ID_RT_Hazard;
output reg [25:0] Jump_Offset;
input [31:0] Instruction;
input Enable, clk , Flush_Branch, Flush_Jump;
input [31:0] PC_Counter_Input;
always@(posedge clk)begin
        if(Flush_Branch == 1 | Flush_Jump == 1)
        begin
        Op_Code = 0;
        Read_Register_1 =  0;
        Read_Register_2 =  0;
        IF_ID_RS_Hazard = 0;
        IF_ID_RT_Hazard = 0;
        IF_ID_Rs = 0;
        IF_ID_Rt = 0;// two go into the ID_EX Pipeline module
        IF_ID_Rd = 0;
        Sign_Extend_Input = 0;
        PC_Counter_Output = 0;
        Jump_Offset= 0;
        end

        else if( Enable == 0)
        begin
        Op_Code = Op_Code;
        Read_Register_1 =  Read_Register_1;
        Read_Register_2 =  Read_Register_2;
        IF_ID_RS_Hazard = IF_ID_RS_Hazard;
        IF_ID_RT_Hazard = IF_ID_RT_Hazard;
        IF_ID_Rs = IF_ID_Rs;
        IF_ID_Rt =IF_ID_Rt;// two go into the ID_EX Pipeline module
        IF_ID_Rd = IF_ID_Rd;
        Sign_Extend_Input =Sign_Extend_Input;
        PC_Counter_Output = PC_Counter_Output;
        Jump_Offset=Jump_Offset;
        end
        else
        begin
        Op_Code = Instruction[31:26];
```

```
            Read_Register_1 =  Instruction[25:21];
            Read_Register_2 =  Instruction[20:16];
            IF_ID_RS_Hazard = Instruction[25:21];
            IF_ID_RT_Hazard =Instruction[20:16];
            IF_ID_Rs = Instruction[25:21];
            IF_ID_Rt = Instruction[20:16]; // two go into the ID_EX Pipeline module
            IF_ID_Rd = Instruction[15:11];
            Sign_Extend_Input = Instruction[15:0];
            PC_Counter_Output = PC_Counter_Input;
            Jump_Offset = Instruction[25:0];
            end
end
endmodule
```

Appendix M

```
module Instruction_Mem_Module(address_In, Instruction,clk);
input [31:0] address_In;
input clk;
output reg [31:0] Instruction;
reg[7:0] memfile[0:255];
initial begin

 {memfile[3 ],memfile[2 ],memfile[1 ],memfile[0 ]}  =   32'hAC080000; // store word
 {memfile[7 ],memfile[6 ],memfile[5 ],memfile[4 ]}  =   32'h00430824; //AND
{memfile[3],memfile[2 ],memfile[1 ],memfile[0]}  =   32'h00E80820;
{memfile[7],memfile[6 ],memfile[5 ],memfile[4]}  =   32'h8000003;
{memfile[11],memfile[10 ],memfile[9 ],memfile[8]}  =   32'h00430824;
{memfile[15],memfile[14],memfile[13 ],memfile[12]}  =   32'h00E80822;
{memfile[19],memfile[18 ],memfile[17 ],memfile[16]}  =   32'h00430824;

{memfile[3],memfile[2 ],memfile[1 ],memfile[0]}  =   32'h00E80822;
end
always@(posedge clk)begin
        Instruction = {memfile[address_In+3], memfile[address_In+2], memfile[address_In+1] ,
memfile[address_In]};
end
endmodule
```

Appendix N

```
module Main_Control_Module(Jump_Signal, RegDst, Branch, MemRead, MemtoReg, ALUOp,
MemWrite, ALUSrc, RegWrite, Instruction_Op_Code);
output reg RegDst, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, Jump_Signal;
output reg [1:0] ALUOp; // keep a 6 bit input for all registers
input [5:0] Instruction_Op_Code;
always@( Instruction_Op_Code)
begin
        if(Instruction_Op_Code == 0)
                begin
                        RegDst = 1;
                        Branch = 0;
                        MemRead = 0;
                        MemtoReg = 0;
                        ALUOp = 2'b10;
                        MemWrite = 0;
                        ALUSrc = 0;
                        RegWrite = 1;
                        Jump_Signal=0;
                end

        else if( Instruction_Op_Code == 6'b10001)
                begin
                        RegDst = 0;
                        Branch = 0;
                        MemRead = 1;
                        MemtoReg = 1;
                        ALUOp = 2'b00;
                        MemWrite = 0;
                        ALUSrc = 1;
                        RegWrite = 1;
                        Jump_Signal=0;
                end

        else if ( Instruction_Op_Code == 6'b101011)
                begin
                        RegDst = 1'bx;
                        Branch = 0;
                        MemRead = 0;
                        MemtoReg = 1'bx;
                        ALUOp = 2'b00;
                        MemWrite =1;
                        ALUSrc = 1;
                        RegWrite = 0;
                        Jump_Signal=0;
                end
```

```
else if ( Instruction_Op_Code == 6'b000100)
                begin
                        RegDst = 1'bx;
                        Branch = 1;
                        MemRead = 0;
                        MemtoReg = 0;
                        ALUOp = 2'b01;  //from book page 579 Figure 4.18
                        MemWrite = 0;
                        ALUSrc = 0;
                        RegWrite = 0;
                        Jump_Signal=0;
                end
else if ( Instruction_Op_Code == 6'b00100)
                begin
                        RegDst = 1'bx;
                        Branch = 0;
                        MemRead = 1'bx;
                        MemtoReg = 0;
                        ALUOp = 2'b00; //from book page 579 Figure 4.18, copied from lw and sw
since they are using add
                        MemWrite = 0;
                        ALUSrc = 1;
                        RegWrite = 1;
                        Jump_Signal=0;
                end
else if ( Instruction_Op_Code == 6'b000010)
                begin
                        RegDst = 1'bx;
                        Branch = 0;
                        MemRead = 1'bx;
                        MemtoReg = 0;
                        ALUOp = 2'b00;
                        MemWrite = 0;
                        ALUSrc = 1;
                        RegWrite = 1;
                        Jump_Signal=1;
                end
end
endmodule
```

Appendix O

```
module MEM_WB_Pipeline_Module (
output reg [4:0] MEM_WB_Fowarding, MEM_WB_Rd_Register_Module,
output reg Regwrite_Output, MemtoReg_Output_Fowarding, MemtoReg_Output,
output reg [31:0] Data_Memory_Output_to_MUX,
output reg [31:0] ALU_Output_to_MUX,
input Regwrite_Input, MemtoReg_Input,
input [4:0] EX_MEM_RD,
input [31:0] Data_Memory_Output_IN,
input [31:0] ALU_Output_IN,
input clk,
);
always@(posedge clk)
begin
        Regwrite_Output = Regwrite_Input;
        MemtoReg_Output_Fowarding = MemtoReg_Input;
        MemtoReg_Output =MemtoReg_Input;
        MEM_WB_Fowarding = EX_MEM_RD;
        Data_Memory_Output_to_MUX = Data_Memory_Output_IN;
        ALU_Output_to_MUX = ALU_Output_IN;
        MEM_WB_Rd_Register_Module = EX_MEM_RD;
end
endmodule
```

Appendix P

```
module mux_1_module (out , a , b , c , control);
output reg [31:0] out;
input [31:0] a,b,c;
input [1:0] control;
always@(*)
begin
        if(control == 0)begin
                        out = a;
                end
        else if (control == 2'b01)begin
                out  = b;
                end
        else
                begin
                        out = c;
                end
end
Endmodule
```

Appendix Q

```
module Mux_2_to_1(out, a,b,control);
output reg [31:0]out;
input [31:0] a,b ;
input control;
always@(*)begin
if(control == 1)
begin
	out = b;
end
else
begin
out = a;
end
end
Endmodule
```

Appendix R

```
module Mux_2_to_1_5_bit(out , a , b , control);
output reg [4:0] out;
input [4:0] a, b;
input control;
always@(*)begin
        if(control == 1)
        begin
        out = b;
        end
        else
        begin
        out = a;
        end
end
endmodule
```

Appendix S

```
module Mux_Control_Module( ALU_Src_Out, ALUOP_Out, RegDst_Out, MemWrite_Out,
MemRead_Out, MemtoReg_Out, RegWrite_Out, ALUOP_In, RegDst_In, MemWrite_In,
MemRead_In, MemtoReg_In, RegWrite_In, Input_Hazard, ALU_Src_In);
output reg RegDst_Out, MemWrite_Out, MemRead_Out, MemtoReg_Out,
RegWrite_Out,ALU_Src_Out;
output reg [1:0] ALUOP_Out;
input RegDst_In, MemWrite_In, MemRead_In, MemtoReg_In, RegWrite_In, ALU_Src_In;
input [1:0] ALUOP_In;
input  Input_Hazard;



always@(*)begin
            if(Input_Hazard == 0)
            begin
            RegDst_Out = 0;
            MemWrite_Out =0;
            MemRead_Out = 0;
            MemtoReg_Out = 0;
            RegWrite_Out = 0;
            ALUOP_Out = 0;
            ALU_Src_Out = 0;
            end
            else
            begin
            RegDst_Out = RegDst_In;
            MemWrite_Out = MemWrite_In;
            MemRead_Out = MemRead_In;
            MemtoReg_Out =  MemtoReg_In;
            RegWrite_Out = RegWrite_In;
            ALUOP_Out = ALUOP_In;
            ALU_Src_Out = ALU_Src_In;
            end
            end
endmodule
```

Appendix T

```
module Mux_Last(out , a , b, control);
output reg [31:0] out;
input [31:0] a , b;
input control;
always@(*)
begin
        if(control == 1)
        begin
                out = a;
        end
        else
                begin
                        out = b;
                end
end
Endmodule
```

Appendix U

```
module Program_Counter_Module(address_Out, enable, address_In,clk);
output reg [31:0] address_Out;
input [31:0] address_In;
input clk;
input enable;
initial
begin
address_Out = 0;
end
always@(posedge clk)
        begin
                if(enable == 0)
                begin
                address_Out = 0;
                end

                else
                begin
                address_Out = address_In;
                end
        end
endmodule
```

Appendix V

```
module Register_Module(read_data_1, read_data_2, read_reg_1, read_reg_2, write_reg, write_data,
regwrite, clk);
output reg [31:0] read_data_1, read_data_2;
input regwrite;
input [31:0] write_data;
input[4:0] read_reg_1, read_reg_2, write_reg;
input clk;
reg [31:0] regfile[0:31];
initial
begin
        regfile[32'd7] = 32'd2;
        regfile[32'd8] = 32'd2;
        regfile[32'd2] = 32'd2;
        regfile[32'd3] = 32'd3;
        regfile[32'd0] = 32'd0;
end
always@(negedge clk)begin
if (regwrite == 1)
        begin
                regfile[write_reg] = write_data;
        end



end
always@(read_reg_1, read_reg_2)
begin
         read_data_1 = regfile[read_reg_1];
         read_data_2 = regfile[read_reg_2];
end
Endmodule
```

Appendix W

```
module Shift_Left_2(in , out);
output reg [31:0] out;
input [31:0] in;
always@(*)
begin
out = in << 2;
end
endmodule
```

Appendix X

```
module Sign_Extend_Module(Input_, Output_);
output reg [31:0] Output_;
input [15:0] Input_;
always@(*)
begin
                Output_[31:16] = 0;
                Output_[15:0] = Input_;
end
Endmodule
```

Appendix Y

```
module TB_File();
reg clk;
Top_Level_Module Top_Level(
 .clk(clk));
initial
begin
clk = 0;
end
always
#100 clk = ~clk;
endmodule
```

Appendix Z

```
module Top_Level_Module( input clk);
        wire [31:0] Adder_Input;
        assign Adder_Input = 32'd4;
        wire [31:0] w1,w2,w7,w11,w12,w39,w40,w41,w44,w47,w46,w102,w105;
        wire [5:0] w3;
        wire w4,w16, w17,w18,w19,w20,w21,w22,w23,w24,w25,w45,w104;
        wire w48,w49,w99;
        wire [25:0] w100;
        wire [27:0] w101;
        wire [1:0] w26,w34;
        wire [4:0] w5,w6,w8,w9,w98;
        wire [15:0] w10;
        wire [4:0] w13,w14,w15,w37;
        wire w27,w28,w29,w30, w31,w32,w38,w36,w42;
        wire A1,A2,A3,A4,A5,A20;
        wire [1:0] A6,A13,A14;
        wire [4:0]A7,A8,A18,A19,A21;
        wire [3:0] A9;
        wire [31:0] A10,A12,A15,A16,A17;
        wire [31:0] A11;
        wire l0,l1,l2,l5,l6,l7,l8,l11,l16;
        wire [4:0] l9,l15;
        wire [31:0] l3,l4,l10,l12,l13,l14,l17;
        Program_Counter_Module Program_Counter_Module1 // done
        (.address_Out(w1),
        .enable(w38),
        .address_In(w105), // needs to come from the output of the MUX;
        .clk(clk));
        Mux_2_to_1 MUX_For_PC_2nd_to_Last( // leftmost mux
        .out(w39),
        .a(w12),
        .b(w40),
        .control(l16) // Control Input for PC counter input
        );
         Instruction_Mem_Module InstructionMem // done
        (.address_In(w1),
         .Instruction(w2), // output
        .clk(clk)
        );
        IF_ID_Pipeline_Module IF_ID_Pipeline // done
        (// need to add teh 26 bits for Jump
        .IF_ID_RS_Hazard(w5),
        .IF_ID_RT_Hazard(w6),
        .PC_Counter_Output(w7),
        .Op_Code(w3),
.Read_Register_1(w8),
.Read_Register_2(w9),
.IF_ID_Rs(w13),
.IF_ID_Rt(w14),
.IF_ID_Rd(w15),
.Sign_Extend_Input(w10),
.Enable(w4),
.Instruction(w2),
.clk(clk),
.Flush_Branch(l16),
.Flush_Jump(w104),
.Jump_Offset(w100),
.PC_Counter_Input(w12) // 32 bit adder
);
Shift_Left_Input_26 Shift_Left_Jump_26(
.out(w101) ,
.in(w100));//
Concatenation_28_32 Concatenation_26_32_Jump(
.out(w102),
.a(w101) ,
.b(w7)// b should be w7);
Mux_Last MUX_For_PC_Last(
.out(w105) ,
.a(w102) ,
.b(w39),
.control(w104)
);

Sign_Extend_Module Sign_Extend (
        .Input_(w10),
        .Output_(w11) );

Shift_Left_2 Shift_Left_
(.in(w11),
.out(w41));

Adder_32_bit Adder_32_bit1
(.out(w40),
.a(w7),
.b(w41));
Adder_32_bit  Adder_Before_IF_ID
(.out(w12),
.a(w1),
.b(Adder_Input));

Main_Control_Module Main_ControlM1
(
        . Instruction_Op_Code(w3),
                    . RegDst(w17),
                    . Branch(w18),
                    . MemRead(w19),
                    . MemtoReg(w20),
                    . MemWrite(w21),
                    . RegWrite(w23),
                    . ALUOp(w26),
                    . ALUSrc(w45),
                    . Jump_Signal(w104)
                    );

Mux_Control_Module Mux_Control_Module1(
.ALUOP_Out(w34),
.RegDst_Out(w32),
.MemWrite_Out(w31),
.MemRead_Out(w30),
.MemtoReg_Out(w28),
.RegWrite_Out(w27),
.ALU_Src_Out(w48),
.ALUOP_In(w26),
.RegDst_In(w17),
.MemWrite_In(w21),
.MemRead_In(w19),
.MemtoReg_In(w20),
.RegWrite_In(w23),
.ALU_Src_In(w45),
.Input_Hazard(w16)
);

ID_EX_Pipeline_Module ID_EX_Module1 (
                    . Regwrite_Input(w27),
                    . MemtoReg_Input(w28),
                    . MemRead_Input(w30),
                    . MemWrite_Input(w31),
                    . RegDst_Input(w32),
                    . ALUOp_Input(w34),
                    . Sign_Extend_Input (w11) ,
                    . IF_ID_Rs (w13),
                    . IF_ID_Rt (w14),
                    . IF_ID_Rd (w15),
                    . Read_Data_1_Input (w46),
                    .Read_Data_2_Input (w47),
                    .clk(clk),
                    . ALU_Src_In(w48),
                    .Regwrite_Output (A1),
                    . MemtoReg_Output (A2),
                    . MemRead_Output (A4),
                    . MemWrite_Output (A5),
                    . RegDst_Output(A20),
                    . ALUOp_Output(A6),
                    . Sign_Extend_Output(A11),
                    . ID_EX_Rs_Fowarding (A7),
                    . ID_EX_Rt_Fowarding (A8),
                    . ID_EX_Rt_MUX(A18),
                    . ID_EX_Rd_MUX(A19),
                    . Read_Data_1_Output(A12),
                    . Read_Data_2_Output(A17),
                    . ALU_Src_Out(w49) );

Register_Module Register1 (
                    . read_reg_1 (w8) ,
                    .read_reg_2 (w9) ,
                    .write_reg (w37),
                    .write_data (l12),
                    .regwrite(w36),
                    .clk(clk),
                    .read_data_1 (w46),
                    .read_data_2 (w47));

Comparator Comparator_for_register_Module(
          .out (w99),
          . a(w46) ,
          .b(w47));

AND_Gate And_Gate_Branch(
.out(l16) ,
.a(w18) ,
.b(w99)
);

Hazard_Unit_Module Hazard_module1 (
                    . IF_ID_RS(w5),
                    . IF_ID_RT(w6),
                    . ID_EX_RT(A8),
                    . ID_EX_Mem_Read(A4),
                    . IF_ID_Enable(w4),
                    . PC_Write_Enable(w38),
                    . Mux_Output(w16));

mux_1_module Forward_A_Mux (
                    .a(A12),
                    . b(l12),
                    .c(l3),
                    . control(A13),
                    .out(A15));
```

```
mux_1_module Forward_B_Mux (
                    .a(A17),
                    . b(l12),
                    .c(l3),
                    . control(A14),
                    .out(A16));

Mux_2_to_1 MUX_ALU_Src(
.out(w44), // goes out to ALU
.a(A16), // fowarding unit B
.b(A11), // sign extend input
.control(w49) // ALU Src Signal);

mux_3_module Bottom_Mux_After_ID_EX (
                    . ID_EX_RT(A18),
                    . ID_EX_RD(A19),
                    .RegDst_Output(A20),
                    . Mux3_output(A21));

Mux_2_to_1_5_bit Bottom_Mux_After_ID_EX(
          .out(A21),
          .a (A18),
          .b (A19),
          .control(A20));

Forward_Unit_Module Forward_Module1 (
                    . ID_EX_RS (A7),
                    . ID_EX_RT (A8),
                    . EX_MEM_RD (l9), // Outputs from Luigi
                    . MEM_WB_RD (l15),
                    . EX_MEM_Reg_Write(l5),
                    . MEM_WB_Reg_Write(w36),
                    . forward_A(A13),
                    . forward_B(A14) );

ALU_Module ALU1 (
                    .input_1(A15),
                    .input_2(w44),
                    .control_input(A9),
                    .output_ (A10) );

ALU_Control_Module ALU_ControlM1 (
                    . Function_Code(A11),
                    . ALU_Op_Input(A6),
                    . ALU_Op_Output(A9) );
```

```
EX_MEM_Pipeline_Module EX_MEM_Module1 (
                        . RegWrite_Input (A1),
                        . MemtoReg_Input (A2),
                        . MemRead_Input (A4),
                        . MemWrite_Input (A5),
                        . ID_EX_MUX(A21),
                        . ALU_Result_Input (A10),
                        .RT_32_Bit_Input(A17),
                        .clk(clk),
                        .RT_32_Bit_Output(l17),
                        . RegWrite_Output (l5),
                        . MemtoReg_Output (l6),
                        . MemRead_Output(l1),
                        . MemWrite_Output(l2),
                        . EX_MEM_RD_Fowarding (l9),
                        . EX_MEM_RD_Next_Pipeline(w98),
                        . ALU_Result_Output(l3) );

Mux_Last Mux_Last1(
.out(l12),
.a(l13),
.b(l14),
.control(l11));

Data_Memory_Module Data1(
  .data_output(l10),
  .address_input(l3),
  .write_data(l17),
  .memWrite(l2),
  .memRead(l1),
  .clk(clk));

MEM_WB_Pipeline_Module MEM_WB_Module1 (
    .Regwrite_Input(l5),
    .MemtoReg_Input(l6),
    .EX_MEM_RD(w98),
    .Data_Memory_Output_IN(l10),
    .ALU_Output_IN(l3),
    .clk(clk),
    .MEM_WB_Fowarding(l15),
    .MEM_WB_Rd_Register_Module(w37),
    .Regwrite_Output(w36),
    .MemtoReg_Output(l11),
    .Data_Memory_Output_to_MUX(l13),
    .ALU_Output_to_MUX(l14));
endmodule
```