ABSTRACT

The objective of the fourth and fifth project is to bring the hardware modules that have designed and tested in the previous projects together with some additional capabilities into a simple single-cycle harvard architecture CPU. Based on that, the single cycle datapath will be converted and extended into a pipeline datapath and a modified control unit and control scheme to manage data flow will be implemented. In support of the pipelined data path, the CPU is designed to manage data and control/instruction hazards, and addition functions like LW, SW, J ,JR, BGE will be added into. In addition, the corresponding machine code for each instruction and the necessary signals to control the datapath elements will be developed. Besides, a C program will be implemented, it will be hand compiled, loaded and executed in the designed CPU. This project will strengthen and reinforce the knowledge of the Verilog HDL and the C. The design is implemented on the Cyclone V FPGA and verified each design using a test suite and the Signaltap logic analyzer. After building the applications, the functionalities of the programs are tested. In order to debug and test programs, specific test plans have been designed for each part of the project. "Quartus II", "SignalTap", are utilized to test the overall project.

INTRODUCTION

This project intends to convert and extend the single cycle datapath into a pipelined datapath and add modified control unit and control scheme to manage data flow through the system. The execution of the program will follow the fetch, decode, execute, write back, next instruction cycle in a pipelined context. The project is designed for practicing understanding single cycle CPU, understanding basic operation of a pipelined datapath, understanding C code fragment and assembly level program, etc. Through the project, tools like "Quartus II", "SignalTap" are used. In the following sections, the lab procedure is introduced first. The whole design project process is then discussed in detail including design objective, design procedure, system description, software implementation and hardware implementation. After that, the data analysis procedure is presented by specifying test plan, test specification and test cases. Then the data analysis in details will be discussed in the presentation, discussion, and analysis of the results section. Finally, a summary and a conclusion can be found the summary and conclusion sections.

DISCUSSION OF THE PROJECT

1.1 Designing and Building Verilog HDL Applications

1.1.1 Design Specification

1.1.1.1 Design Description

In this design, a harvard architecture central processing unit (cpu) is designed. The cpu implements a pipelining design for the whole computation process in order to upgrade the efficiency of the cpu. In this design, the cpu is able to execute MIPS instructions which are shown below:

- Add Immediate (addi)
- Store Word (sw)
- Load Word (lw)
- Branch Greater Than (BGT)
- Jump (J)
- Addition (add)
- Subtraction (sub)
- AND (and)
- OR (or)
- XOR (xor)
- Set Less Than (slt)
- Shift Left Logical (sll)
- Jump Register (jr)
- No Operation (nop)

The design contains 5 main modules that are instruction fetch module, instruction decode module, execution module, memory access module, and write back module. In order to pass the data within these 5 modules, 4 transit data register modules are designed between each two modules. And also, the control signal register modules are implemented for synchronizing control signals with the data flow. Furthermore, 2 control modules are implemented to control the data paths of the MIPS instructions. One of the control modules is to control the overall data path in the whole cpu, and the other one is aiming for control the operation of the ALU module in the cpu. Finally, a forwarding unit is designed to dealing with the data hazard which may happen in the MIPS instruction, as well as a hazard detection unit is also designed to dealing with the control hazard in the MIPS instruction.

1.1.2 Design Procedure

In this design, the system is able to handle MIPS instruction set to calculate the data in the system and executing the program that is specified by the MIPS instruction. As discussed before, the design includes the following modules:

- Instruction Fetch Module (IF)
- Instruction Decode Module (ID)
- Execution Module (EX)
- Memory Access Module (MEM)
- Write Back (WB)
- Instruction Fetch to Instruction Decode Transit Register Module (IF_ID)
- Instruction Decode to Execution Transit Register Module (ID_EX)
- Instruction Decode to Execution Control Signal Transit Register Module (ID_EX_CtPath)
- Execution to Memory Access Transit Register Module (EX_MEM)
- Execution to Memory Access Control Signal Transit Register Module (EX_MEM_CtPath)
- Memory Access to Write Back Transit Register Module (MEM_WB)
- Memory Access to Write Back Control Signal Transit Register Module (MEM_WB_CtPath)
- Control Module (control)
- ALU Control Module (ALU_Control)
- Forwarding Unit (ForwardingUnit)
- Hazard Detection Unit (Hazard_Detection_Unit)

In this section, the report will discuss each module in details separately.

1.1.2.1: Instruction Fetch Module (IF)

The Instruction Fetch Module (IF) consists of two major sub-modules: program counter and instruction memory. The program counter (pc) is the module that actually drive the whole system functioning. The program counter's design simply takes the calculated instruction address into the module and output it at the next clock cycle. The other sub-module is the instruction memory module (instruction_mem). The instruction memory module is a memory module with a size of 128 and is able to store word with a width of 32 bits because the standard MIPS instructions have width of 32 bits. The instruction memory module actually store the user's program in binary in sequence. It takes the output of the program counter as input which specifies the location of the instruction that will be executed in the following procedures, and then outputs the instruction to the Instruction Fetch to Instruction Decode Transit Register Module. Furthermore, the IF module also has three 32 bits 2-to-1 multiplexers to select the correct input of the program counter. These three multiplexers are branch multiplexer, jump multiplexer, and jump register multiplexer which are controlled by branch signal, jump signal, and jump register signal generated by the control module.

1.1.2.2: Instruction Decode Module (ID)

The Instruction Decode Module (ID) decodes the instructions taken from the IF_ID module. ID mainly contains two major sub-modules that are register file module and sign extended module. The register file is a memory module with a size of 32 and a word width of 32 bits which maintains all of the data that needs to be calculated in the program. Moreover, another sub-module is inside the ID module as well which is the sign extended module. Due to the fact that the word width is 32 bits but the immediate instruction's immediate field has a width of 16 bits, the immediate field should be extended to 32 bits in order to match all the other data for the further operations. Other than these two modules, a comparator is also in the ID module in order to detect the branch beforehand, so that the flush operation will be kept simple.

1.1.2.3: Execution Module (EX)

The execution module is the one that does the major calculation operations. There is only one main sub-module in the EX module which is ALU module. The ALU module takes two inputs from previous stage and does calculations based on the command output by the ALU control module. However, ALU module's inputs have three different sources that are the outputs from ID module that are the data stored in the register file, the calculated data from memory access module, and the output data from write back module. The forwarding unit will handle the selection of the source of the inputs which will be discussed in the following sections.

1.1.2.4: Memory Access Module (MEM)

The Memory Access Module (MEM) is the main memory in the cpu module. Practically, the memory access module consists of a SRAM module with a size of 2K and a word width of 16 bits. The data will be stored into the main memory if it is required by instruction. Otherwise, the data will go around this stage and be forwarded to the write back module.

1.1.2.5: Write Back Module (WB)

The Write Back Module (WB) is the last stage of executing an instruction. It consists of a single multiplexer that decides if the data that would be written back the the register file is from either from main memory or from the output of the ALU module.

1.1.2.6: Control Module (control)

The Control Module (control) generates the control signals to determine the control path of the data. Different types of instructions have different combinations of control signals. In this design, there are three major types of instructions that are R-type, I-type, and Jump.

R-type instructions usually involve ALU operations. The source register fields are rs and rt, the destination register field is rd. Furthermore, an R-type instruction writes a register (RegWrite = 1) , but neither reads nor writes to data memory. The control module will output a control signals to ALU control module in order to determine the actual behavior of the ALU module. In addition, the R-type instructions include addition (add), subtraction (sub), and (AND), or (OR), xor (XOR), set less than (SLT), shift left logical (SLL), jump register (JR) and none operation (NOP). In the MIPS standard, the R-type instructions share the same Op-code which is 000000, and the operation is determined by each instruction’s function field (i.e., the last 6 bits of the instruction).

Other instructions such as add immediate (addi), load word (lw), and store word (sw) are I-type. An Addi instruction which write the chosen value (RegWrite = 1). A lw-type instruction, the ALUSrc fields is set to perform the address calculation. The memory read set to perform the memory access. The MemtoReg decides between sending the AU result or the memory value to the register file. A sw-type instruction, the ALUSrc fields is set to perform the address calculation. The memory write set to perform the memory access. The branch great than instruction (BGT) is used for comparing situations. In C code, it corresponds to the if statement. If the statement evaluates to true, the program counter will be moved to the position where the if statement instructions located. The jump instruction is a R-type instruction, which will move the program counter to the new location where the instruction indicates.

Table 1.1.2.6 shows the detail control signals for each instruction.

Table 1.1.2.6-control signals for each type of instruction

Input Or Output	Signal Name	R_form at	Addi	lw	sw	bgt	Jump	default
Inputs								
Outputs	ALUOp	10	00	00	00	01	11	11
	ALUSrc	0	1	1	1	0	0	0
	Branch	0	1	0	0	1	0	0
	MemRead	0	0	1	0	0	0	0
	MemtoReg	0	0	1	x	x	0	0
	MemWrite	0	0	0	1	1	0	0
	RegDst	1	x	0	x	x	0	0
	RegWrite	1	1	1	0	0	0	0
	Jump	0	0	0	0	0	1	0

1.1.2.7: ALU Control Module (ALU_control)

ALU control module takes the input of the control module (ALUOp) to determine the operation of the ALU module. Table 1.1.2.7 shows the details of the binary instructions of the ALU control module. Notice that I-type instructions have the same ALU control input as addition since in this design, the address of the data memory is added with $zero which maintains 0. The ALU module will also decode the function field of the instruction which is the lowest 6 bits if the Op-code is 000000 which specifies the instruction is a R-type instruction.

Table 1.1.2.7-ALU control module operation instruction look-up table

Instruction Opcode	ALUOp	Instruction Operation	Function Field Code	ALU Control Input
I-Type	00	Load word	XXXXXX	001
I-Type	00	Store word	XXXXXX	001
BGT	01	Branch greater than	XXXXXX	010
R-Type	10	add	100000	001
R-Type	10	subtract	100010	010

R-Type	10	AND	100100	011
R-Type	10	OR	100101	100
R-Type	10	XOR	101000	101
R-Type	10	SLT	101010	110
R-Type	10	SLL	000000	111
R-Type	10	NO Operation	001000	000

1.1.2.8: Forwarding Unit Module (ForwardingUnit)

The forwarding unit is will handle the data hazard to forward the output of the first two instructions to the third instruction's input if needed. The general case of the forwarding is to compare the $s and $t in the execution stage to $d in the memory access stage and that in the write back stage. If either $s or $t matches $d in either memory access stage or write back stage, the forwarding unit will forward the result calculate in corresponding stage to the operand of the ALU which requires a forwarded input.

1.1.2.9: Hazard Detection Unit Module (HazardDetectionUnit)

The Hazard Detection Unit (HazardDetectionUnit) is to handle control hazard or stall. As the hazard detection unit detects that the branch is taken in the instruction decode stage, the hazard detection unit will enable the flush signal to flush the following three instructions. Also, if the hazard detection unit detects load word instruction followed by R-type instructions, it will stall the next instruction by keep the next instruction in its stage for one clock cycle and clear all of the control signals for this clock cycle.

1.1.3 System Description

1.1.3.1 Instruction Fetch Module (IF)

- Inputs:
    - 32 bits branch true case increment counter
    - 32 bits jump register true case increment counter
    - 32 bits jump true case increment counter
    - Branch control signal
    - Jump control signal

    - Jump register control signal
    - PCWrite control signal
    - 32 bits program counter
- Outputs:
    - 32 bits next instruction
- Side effects:
    - If the program executes in the normal order, the program counter increments in this module instead of forwarding to the next stage.

1.1.3.2 Instruction Decode Module (ID)

- Inputs:
    - 32 bits data that will be written to register file
    - 32 bits instruction from IF module
    - 32 bits ALU output from memory access module
    - 32 bits ALU output from execution module
    - 1 bit control signal that shows the result of comparing two data
- Outputs:
    - 32 bits operand 1
    - 32 bits operand 2
    - 32 bits extended data
    - All of the control signals
- Side effects:
    - The ID module requires to forward the data in the EX and MEM to this module.

1.1.3.3 Execution Module (EX)

- Inputs:
    - 32 bits branch true case increment counter
    - 32 bits jump register true case increment counter
    - 32 bits jump true case increment counter
    - Branch control signal
    - Jump control signal
    - Jump register control signal
    - PCWrite control signal
    - 32 bits program counter
    - 32 bits operand 1
    - 32 bits operand 2

- Outputs:
    - 32 bits ALU output
- Side effects:
    - Data hazard

1.1.3.4 Memory Access Module (MEM)

- Inputs:
    - 32 bits ALU output from EX module
    - 32 bits operand 2 from EX module
    - MemWrite, MemRead control signals
- Outputs:
    - 32 bits data read from data memory
    - 32 bits ALU output from EX module
- Side effects:
    - $d needs to be forwarded to next stage

1.1.3.5 Write Back Module (WB)

- Inputs:
    - 32 bits data read from data memory
    - 32 bits ALU output from EX module
    - MemtoReg control signal
- Outputs:
    - 32 bits data which will be written back to register file
- Side effects:
    - None

1.1.3.6 Control Module (control)

- Inputs:
    - 6-bits Operation Code Id, Function field
- Outputs:
    - 1-bit Branch, Memory Read, Memory Write, Memory to Register
    - 1-bit Destination Register, Write Register, ALUSrc
    - 1-bit Jump Instruction
    - 2-bits ALU Operation code

- Side effects:
    - The Jump Register is a R-type instruction instead of Jump type

1.1.3.7 ALU_Control Module (ALU_control)

- Inputs:
    - 2 bits ALU operation code
    - 6 bits Function field code
- Outputs:
    - 3 bits ALU control scheme
- Side effects:
    - None

1.1.4 Software Implementation

The overall harvard architecture central processing unit (cpu) will be tested and implemented by using a simple C code program. The way to test the harvard architecture central processing unit is to rewrite the C code to MIPS assembly language, then encode MIPS assembly language into binary instruction and hard-coded it into memory instruction. The overall MIPS is shown in appendix. Each instruction in the C code is rewritten to MIPS assembly code using corresponding R or I type instructions, together with registers v0 - v1, t0 - t7, s0 - s7, t8 -t9.

Commands	MIPS Assembly Code	Encoded Binary Instruction

int A = 7	$s0 (10000), $zero, 7
	addi $t0 (1000), $zero, 7
	sw $t0 (1000), 0($s0 (10000))
	sw $t0 (1000), 0($s0 (10000))
		0010 0000 0001 0000 0000 0000 0000 0111
		0010 0000 0000 1000 0000 0000 0000 0111
		1010 1110 0000 1000 0000 0000 0000 0000
		1000 1110 0000 1000 0000 0000 0000 0000

int B = 5	addi $s1 (10001), $zero, 5
	addi $t1 (1001), $zero, 5
	sw $t1 (1001), 0($s1 (10001))
	lw $t1 (1001), 0($s1 (10001))
		0010 0000 0001 0001 0000 0000 0000 0101
		0010 0000 0000 1001 0000 0000 0000 0101
		1010 1110 0010 1001 0000 0000 0000 0000
		1000 1110 0010 1001 0000 0000 0000 0000

int C = 3	addi $s2 (10010), $zero, 3
	addi $t2 (1010), $zero, 2
	sw $t2 (1010), 0($s2 (10010))
	lw $t2 (1010), 0($s2 (10010))
		0010 0000 0001 0010 0000 0000 0000 0011
		0010 0000 0000 1010 0000 0000 0000 0010
		1010 1110 0100 1010 0000 0000 0000 0000
		1000 1110 0100 1010 0000 0000 0000 0000

Int D = 5	addi $s3 (10011), $zero, 5;
	addi $t3 (1011), $zero, 4;
	sw $t3 (1011), 0($s3 (10011))
		0010 0000 0001 0011 0000 0000 0000 0101
		0010 0000 0000 1011 0000 0000 0000 0100
		1010 1110 0110 1011 0000 0000 0000 0000

	lw $t3 (1011), 0($s3 (10011))	1000 1110 0110 1011 0000 0000 0000 0000
int* dPtr = &D	add $t8 (11000), $s3 (10011), $zero;	0000 0010 0110 0000 1100 0000 0010 0000

unsigned int E = 0x5A5A	addu $s4 (10100), $zero, 0x5A5A
	addu $t4 (1100), $zero, 0x5A5A
	sw $t4 (1100), 0($s4 (10100))
	lw $t4 (1100), 0($s4 (10100))
		0000 0000 0001 0100 0101 1010 0101 1010
		0000 0000 0000 1100 0101 1010 0101 1010
		1010 1110 1000 1100 0000 0000 0000 0000
		1000 1110 1000 1100 0000 0000 0000 0000

unsigned int F = 0x6767	addu $s5 (10101), $zero, c
	addu $t5 (1101), $zero, 0x5A5A
	sw $t5 (1101), 0($s5 (10101))
	lw $t4 (1101), 0($s4 (10101))
		0000 0000 0001 0101 0110 0111 0110 0111
		0000 0000 0000 1101 0110 0111 0110 0111
		1010 1110 1010 1101 0000 0000 0000 0000
		1000 1110 1010 1101 0000 0000 0000 0000

unsigned int G = 0x3c	addu $s6 (10110), $zero, 0x3c
	addu $t6 (1110), $zero, 0x3c
	sw $t6 (1110), 0($s6 (10110))
	lw $t6 (1110), 0($s6 (10110))
		0010 0100 0001 0110 0000 0000 0011 1100
		0010 0000 0000 1110 0000 0000 0011 1100
		1010 1110 1100 1110 0000 0000 0011 1100
		1000 1110 1100 1110 0000 0000 0011 1100

unsigned int H = 0xff	addu $s7 (10111), $zero, 0xff
	addu $t7 (1111), $zero, 0xff
	sw $t7 (1111), 0($s3 (10111))
	lw $t7 (1111), 0($s3 (10111))
		0010 0000 0001 0111 0000 0000 1111 1111
		0010 0000 0000 1011 0000 0000 0000 0100
		1010 1110 1110 1111 0000 0000 0000 0000
		1000 1110 1110 1111 0000 0000 0000 0000

int temp0 = A - B	sub $v0 (0010), $t0 (1000), $t1 (1001) addi $t9 (11001), $zero, 3	0000 0001 0000 1001 0001 0000 0010 0010 0010 0000 0001 1001 0000 0000 0000 0011
(A - B) ? 3	bgt $t9 (11001), $v0 (0010), L1:line 40 (00101000)	0001 1100 0101 1001 0000 0000 0000 0100
C = C << 3	sll $t2 (1010), $t2 (1010), $t9 (11001)	0000 0001 0101 1001 0101 0000 0000 0000
*dPtr = 7	addi $t3 (1011), $zero, 7	0010 0000 0000 1011 0000 0000 0000 0111
G = E & F	and $t6 (1110), $t4 (1100), $t5 (1101)	0000 0001 1000 1101 0111 0000 0010 0100
Jump to L2	L2: line 43 (00101011)	0000 1000 0000 0000 0000 0000 0010 1011
L1: C = C + 4	addi $t2 (1010), $t2 (1010), 4	0010 0001 0100 1010 0000 0000 0000 0100
D = C – 3	sub $t3 (1011), $t2 (1010), $t9 (11001)	0000 0001 0101 1001 0101 1000 0010 0010
G = E | F	or $t6 (1110), $t4 (1100), $t5	0000 0001 1000 1101 0111 0000 0010 0101

	(1101)	
L2: A = A + B	add $t0 (1000), $t0 (1000), $t1 (1001)	0000 0001 0000 1001 0100 0000 0010 0000
temp1 = E ^ F	xor $v1 (0011), $t4 (1100), $t5 (1101)	0000 0001 1000 1101 0001 1000 0010 0110
G = (E ^ F) & H	and $t6 (1110), $v1 (0011), $t7 (1111)	0000 0000 0110 1111 0111 0000 0010 0100

1.1.5 Hardware Implementation

Figure 1.1.5.1 shows the high level hardware schematic. The IF module lays at the left most of the schematic, the ID follows the IF module, and then are followed by EX, MEM, and WB in sequence. Between each two modules, the transit registers are shown in the schematic as well. All of the control modules and the hazard handling units are labeled in blue in the figure.

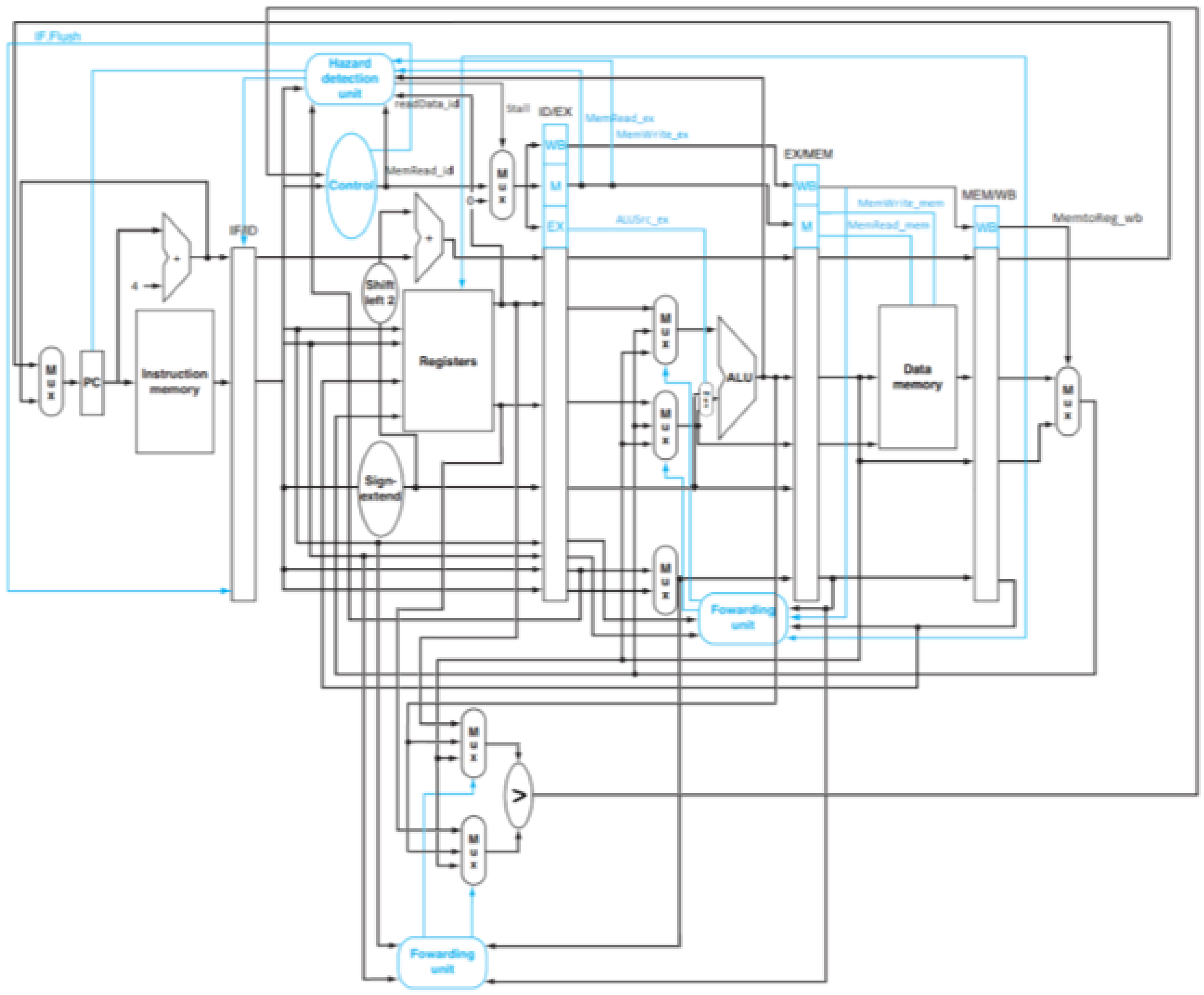

Figure 1.1.5.1-Schematic of the CPU module

TEST PLAN

In order to make sure that the entire system works as expected and neither hazards nor errors happens during the executions of code, the system will be tested in different ways. First of all, the system will be tested separately. The execution of each of the thirteen instructions will be tested by itself. The instructions are NOP, ADD, SUB, AND, OR, XOR, SLT, SLL, LW, SW, J, JR, BGT. After that, the combination of instructions will be tested. This will make sure that the system will be able to handle hazards in the right way. In the end, the performance of the pipelining will also be tested and the input and output of the system will be examined thoroughly to prove its correct performance.

TEST SPECIFICATION

The performance of the system will be tested by following C code.

```
int A = 7;
 int B = 5;
int C = 3;
int D = 5;
int* dPtr = &D;
unsigned int E = 0x5A5A;
unsigned int F = 0x6767;
unsigned int G = 0x3C;
unsigned int H = 0xFF;

if ((A – B) > 3 {
        C = C + 4;
        D = C – 3;
        G = E | F;
} else {
        C = C << 3;
        *dPtr = 7;
        G = E & F;
}
A = A + B;
G = (E ^ F) & H;
```

Figure 2.1 Test C Code

The C code shown above will be inputted into the system and the result of the system should be exactly the same as the results calculated by the C code. The performance of each line in the C code and its corresponding position in the register file are shown below in the table. The calculated values are all shown in hex.

Table 2.1 Instruction Execution Results

Variable and constants	Register name	Register address	Initial values	Executions happened during if statement	Executions happened during else statement	After the if else statement

A	t0	1000	7			A = A+ B A = 7 + 5 A = 12
B	t1	1001	5			
C	t2	1010	3	C = C + 4 C = 7	C = C << 3 C = 18	
D	t3	1011	5	D = C - 3 D = 7 - 3 D = 4	*dPtr = 7 D = 7	
E	t4	1100	5A5A			
F	t5	1101	6767			
G	t6	1110	3C	G = E | F G = 5A5A | 6767 G = 7F7F	G = E & F G = 5A5A & 6767 G = 4242	G = (E ^ F) & H G = 3D3D & FF G = 3D
H	t7	1111	FF			
dPtr	t8	11000	Address of D (5)			
3	t9	11001	3			
A - B	v0	0010	2			
E ^ F	v1	0011				E ^ F = 3D3D

In this case, the "if" statement will not be executed since A = 7, B = 5, and A - B = 2 which will always smaller than 3. Therefore, the output of the system should be the same as the table shown below. The values are also shown in hex.

Table 2.2 Instruction Performance Table

Variable name	Initial value	Intermediate value	Final value
A	7	7	12
B	5	5	5
C	3	18	18
D	5	7	7

E	5A5A	5A5A	5A5A
F	6767	6767	6767
G	3C	4242	3D
H	FF	FF	FF

The limitation for this test is that it will not be able to test the branch statement, since the "if" statement will not be reached under the current situation. To resolve this problem, the value of A or B could be changed to produce a result which will be higher than 3 after calculation.

TEST CASES

In order to performance the test, the C code shown in the above section is first converted to MIPS instructions. The converted instructions are then change to binary code, which, were used as inputs to the system. Both the MIPS version and the binary version of the instructions are shown in the Appendix section in Figure t.3. Those instructions will be loaded into the instruction memory at the beginning of execution and then when the program counter starts counting up, it will go through the instructions one by one and execute them in order. When branch or jump instructions occur, the program will be changed to point to specific places in the instruction memory and start executing from there. The output of the system will be stored into the register files and the values in there will be compared against the results that are shown in the above section to prove the correctness of the system.

PRESENTATION, DISCUSSION, AND ANALYSIS OF THE RESULTS

The pipelined system is tested by implementing "Signal Tap". In this section of the report, the outcome of the "Signal Tap" is shown in Appendix Figure t.4.a - t.4.l which represent the waveform of writing operation for address bus and data bus.

ANALYSIS OF ANY ERRORS

The results of the test match the expected outcome. The output of the written data to the register files under different instructions matches the value shown in the test specification section above.

ANALYSIS OF WHY THE PROJECT MAY NOT OF WORKED AND WHAT EFFORTS WERE MADE TO IDENTIFY THE ROOT CAUSE OF ANY PROBLEMS

The design may not of worked if the C program is not converted to standard MIPS instructions properly. The MIPS instructions were checked bit by bit during the design process. Also, the forwarding unit that is mentioned in the textbook only works for the specific case. In general, either the $s or the $t matches $d in either MEM module or WB module, the data should be forwarded.

SUMMARY

In this project, the performance of the single cycle design was improved by incorporating an instruction cycle pipeline. Machine code was developed to support control of the pipeline and flow through the datapath to ensure proper execution of each instruction and to manage the data and control hazards. After that, a complete simple C code fragment was implemented into an assembly level program comprising instructions from the supported instruction set. The machine code was loaded into the instruction memory for the design on the DE1-SoC. Finally, the operation of the pipeline and complete instruction cycle for each instruction comprising the code fragment was demonstrated and tested using SignalTap.

CONCLUSION

This project gives a very good introduction in designing and building robust, well-functioning, and user-friendly applications. During this project, various tools such as “SignalTap” have also been introduced. Those tools are very useful in testing and making sure the design is working properly and they will also be very useful for testing future projects. Ultimately the test results produced by the designs for this project are consistent with both expected results as well as with each other. Through simulation programs as well as failure mode analysis, no errors were found in all the Verilog HDL applications. Through the tests of the C code, no mistakes were found even with boundary cases. All the confusions have been successfully resolved and all the problems were resolved. Thus, in conclusion, the designs and tests for this project were successful.

APPENDICES

```
***************** int A = 7; *************************
******************************************************
** #0 addi $t, $s, imm
** #0 addi $s0 (10000), $zero, 7;
** opcode $s    $t    imm
** 001000 00000 10000 0000000000000111
** 2    0    1    0    0    0    0    7       HEX
** 0010 0000 0001 0000 0000 0000 0000 0111    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*******************************************************
** #1 addi $t, $s, imm
** #1 addi $t0 (1000), $zero, 7;
** opcode $s    $t    imm
** 001000 00000 01000 0000000000000111
** 2    0    0    8    0    0    0    7       HEX
** 0010 0000 0000 1000 0000 0000 0000 0111    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*******************************************************
*******************************************************
** #2 sw $t, offset($s)
** #2 sw $t0 (1000), 0($s0 (10000));
** opcode $s    $t    offset
** 101011 10000 01000 0000000000000000
** a    e    0    8    0    0    0    0       HEX
** 1010 1110 0000 1000 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*******************************************************
*******************************************************
** #3 lw $t, offset($s)
** #3 lw $t0 (1000), 0($s0 (10000));
** opcode $s    $t    offset
** 100011 10000 01000 0000000000000000
** 8    e    0    8    0    0    0    0       HEX
** 1000 1110 0000 1000 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*******************************************************

***************** int B = 5; **************************
*******************************************************
** #4 addi $t, $s, imm
** #4 addi $s1 (10001), $zero, 5;
** opcode $s    $t    imm
** 001000 00000 10001 0000000000000101
** 2    0    1    1    0    0    0    5       HEX
** 0010 0000 0001 0001 0000 0000 0000 0101    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*******************************************************
** #5 addi $t, $s, imm
** #5 addi $t1 (1001), $zero, 5;
** opcode $s    $t    imm
** 001000 00000 01001 0000000000000101
** 2    0    0    9    0    0    0    5       HEX
** 0010 0000 0000 1001 0000 0000 0000 0101    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #6 sw $t, offset($s)
** #6 sw $t1 (1001), 0($s1 (10001));
** opcode $s    $t    offset
** 101011 10001 01001 0000000000000000
** a    e    2    9    0    0    0    0       HEX
** 1010 1110 0010 1001 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #7 lw $t, offset($t)
** #7 lw $t1 (1001), 0($s1 (10001));
** opcode $s    $t    offset
** 100011 10001 01001 0000000000000000
** 8    e    2    9    0    0    0    0       HEX
** 1000 1110 0010 1001 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************

****************** int C = 3; ***************************
*********************************************************
** #8 addi $t, $s, imm
** #8 addi $s2 (10010), $zero, 3;
** opcode $s    $t    imm
** 001000 00000 10010 0000000000000011
** 2    0    1    2    0    0    0    2       HEX
** 0010 0000 0001 0010 0000 0000 0000 0011    BINARY
** 010 00ss ssst tttt iiii iiii iiii iiii     ENCODING
*********************************************************
** #9 addi $t, $s, imm
** #9 addi $t2 (1010), $zero, 2;
** opcode $s    $t    imm
** 001000 00000 01010 0000000000000010
** 2    0    0    a    0    0    0    2       HEX
** 0010 0000 0000 1010 0000 0000 0000 0010    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #10 sw $t, offset($s)
** #10 sw $t2 (1010), 0($s2 (10010));
** opcode $s    $t    offset
** 101011 10010 01010 0000000000000000
** a    e    4    a    0    0    0    0       HEX
** 1010 1110 0100 1010 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #11 lw $t, offset($s)
** #11 lw $t2 (1010), 0($s2 (10010));
** opcode $s    $t    offset
** 100011 10010 01010 0000000000000000
** 8    e    4    a    0    0    0    0       HEX
** 1000 1110 0100 1010 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************

***************** int D = 5; *************************
*********************************************************
** #12 addi $t, $s, imm
** #12 addi $s3 (10011), $zero, 5;
** opcode $s    $t    imm
** 001000 00000 10011 0000000000000101
** 2    0    1    3    0    0    0    4       HEX
** 0010 0000 0001 0011 0000 0000 0000 0101    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #13 addi $t, $s, imm
** #13 addi $t3 (1011), $zero, 4;
** opcode $s    $t    imm
** 001000 00000 01011 0000000000000100
** 2    0    0    b    0    0    0    4       HEX
** 0010 0000 0000 1011 0000 0000 0000 0100    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #14 sw $t, offset($s)
** #14 sw $t3 (1011), 0($s3 (10011));
** opcode $s    $t    offset
** 101011 10011 01011 0000000000000000
** a    e    6    b    0    0    0    0       HEX
** 1010 1110 0110 1011 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #15 lw $t, offset($s)
** #15 lw $t3 (1011), 0($s3 (10011));
** opcode $s    $t    offset
** 100011 10011 01011 0000000000000000
** 8    e    6    b    0    0    0    0       HEX
** 1000 1110 0110 1011 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************

***************** int* dPtr = &D; *********************
*********************************************************
** #16 add $d, $s, $t
** #16 add $t8 (11000), $s3 (10011), $zero;
** opcode $d    $s    $t          function field
** 000000 10011 11000 00000 00000 100000
** 0    0    1    3    5    0    2    0       HEX
** 0000 0010 0110 0000 1100 0000 0010 0000    BINARY
** 0000 00ss ssst tttt dddd d000 0010 0000    ENCODING
*********************************************************

**************unsigned int E = 0x5A5A; ****************
********************************************************
```

```
** #17 addu $t, $s, imm
** #17 addu $s4 (10100), $zero, 0x5A5A;
** opcode $s    $t    imm
** 001000 00000 10100 0101101001011010
** 0    0    1    4    5    A    5    A       HEX
** 0000 0000 0001 0100 0101 1010 0101 1010    BINARY
** 0000 00ss ssst tttt iiii iiii iiii iiii    ENCODING
**********************************************************
** #18 addu $t, $s, imm
** #18 addu $t4 (1100), $zero, 0x5A5A;
** opcode $s    $t    imm
** 001000 00000 01100 0101101001011010
** 0    0    0    C    5    A    5    A       HEX
** 0000 0000 0000 1100 0101 1010 0101 1010    BINARY
** 0000 00ss ssst tttt iiii iiii iiii iiii    ENCODING
**********************************************************
**********************************************************
** #19 sw $t, offset($s)
** #19 sw $t4 (1100), 0($s4 (10100));
** opcode $s    $t    offset
** 101011 10100 01100 0000000000000000
** a    e    8    C    0    0    0    0       HEX
** 1010 1110 1000 1100 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
**********************************************************
**********************************************************
** #20 lw $t, offset($s)
** #20 lw $t4 (1100), 0($s4 (10100));
** opcode $s    $t    offset
** 100011 10100 01100 0000000000000000
** 8    e    8    C    0    0    0    0       HEX
** 1000 1110 1000 1100 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
**********************************************************

**************unsigned int F = 0x6767; ****************
*********************************************************
** #21 addu $t, $s, imm
** #21 addu $s5 (10101), $zero, c;
** opcode $s    $t    imm
** 001000 00000 10101 0110011101100111
** 0    0    1    4    5    A    5    A       HEX
** 0000 0000 0001 0101 0110 0111 0110 0111    BINARY
** 0000 00ss ssst tttt iiii iiii iiii iiii    ENCODING
**********************************************************
** #22 addu $t, $s, imm
** #22 addu $t5 (1101), $zero, 0x5A5A;
** opcode $s    $t    imm
** 001000 00000 01101 0110011101100111
** 0    0    0    C    5    A    5    A       HEX
** 0000 0000 0000 1101 0110 0111 0110 0111    BINARY
** 0000 00ss ssst tttt iiii iiii iiii iiii    ENCODING
**********************************************************
```

```
*********************************************************
** #23 sw $t, offset($s)
** #23 sw $t5 (1101), 0($s5 (10101));
** opcode $s    $t    offset
** 101011 10101 01101 0000000000000000
** a    e    8    C    0    0    0    0       HEX
** 1010 1110 1010 1101 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
*********************************************************
** #24 lw $t, offset($s)
** #24 lw $t4 (1101), 0($s4 (10101));
** opcode $s    $t    offset
** 100011 10101 01101 0000000000000000
** 8    e    8    C    0    0    0    0       HEX
** 1000 1110 1010 1101 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************


************unsigned int G = 0x3c; **********************
*********************************************************
** #25 addu $t, $s, imm
** #25 addu $s6 (10110), $zero, 0x3c;
** opcode $s    $t    imm
** 001000 00000 10110 0000000000111100
** 2    0    1    2    0    0    0    2       HEX
** 0010 0100 0001 0110 0000 0000 0011 1100    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #26 addu $t, $s, imm
** #26 addu $t6 (1110), $zero, 0x3c;
** opcode $s    $t    imm
** 001000 00000 01110 0000000000111100
** 2    0    0    a    0    0    0    2       HEX
** 0010 0000 0000 1110 0000 0000 0011 1100    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #27 sw $t, offset($s)
** #27 sw $t6 (1110), 0($s6 (10110));
** opcode $s    $t    offset
** 101011 10110 01110 0000000000000000
** a    e    4    a    0    0    0    0       HEX
** 1010 1110 1100 1110 0000 0000 0011 1100    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #28 lw $t, offset($s)
** #28 lw $t6 (1110), 0($s6 (10110));
** opcode $s    $t    offset
** 100011 10110 01110 0000000000000000
** 8    e    4    a    0    0    0    0       HEX
** 1000 1110 1100 1110 0000 0000 0011 1100    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
```

```
*********************************************************

***********unsigned int H = 0xff; *********************
*********************************************************
** #29 addu $t, $s, imm
** #29 addu $s7 (10111), $zero, 0xff;
** opcode $s    $t    imm
** 001000 00000 10111 0000000011111111
** 2    0    1    3    0    0    0    4       HEX
** 0010 0000 0001 0111 0000 0000 1111 1111    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #30 addu $t, $s, imm
** #30 addu $t7 (1111), $zero, 0xff;
** opcode $s    $t    imm
** 001000 00000 01111 0000000011111111
** 2    0    0    b    0    0    0    4       HEX
** 0010 0000 0000 1011 0000 0000 0000 0100    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #31 sw $t, offset($s)
** #31 sw $t7 (1111), 0($s3 (10111));
** opcode $s    $t    offset
** 101011 10111 01111 0000000000000000
** a    e    6    b    0    0    0    0       HEX
** 1010 1110 1110 1111 0000 0000 0000 0000    BINARY
** 1010 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
** #32 lw $t, offset($s)
** #32 lw $t7 (1111), 0($s3 (10111));
** opcode $s    $t    offset
** 100011 10111 01111 0000000000000000
** 8    e    6    b    0    0    0    0       HEX
** 1000 1110 1110 1111 0000 0000 0000 0000    BINARY
** 1000 11ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************

***************** if (A - B) > 3; *********************
*************** int temp0 = A - B; ********************
*********************************************************
*** #33 sub $d, $s, $t
*** #33 sub $v0 (0010), $t0 (1000), $t1 (1001);
*** opcode $s    $t    $d          function field
*** 000000 01000 01001 00010 00000 100010
*** 0    1    0    9    6    8    2    2      HEX
*** 0000 0001 0000 1001 0001 0000 0010 0010   BINARY
*** 0000 00ss ssst tttt dddd d000 0010 0010   ENCODING
*********************************************************
*** #34 addi $t, $s, imm
*** #34 addi $t9 (11001), $zero, 3;
*** opcode $s    $t    imm
*** 001000 00000 11001 0000000000000011
*** 2    0    0    f    0    0    0    3      HEX
*** 0010 0000 0001 1001 0000 0000 0000 0011   BINARY
*** 0010 00ss ssst tttt iiii iiii iiii iiii   ENCODING
*********************************************************
******************** (A - B) ? 3 ************************
*********************************************************
*** #35 bgt $s, $t, offset
*** #35 bgt $t9 (11001), $v0 (0010),  L1:line 40 (00101000);
*** opcode $s    $t    offset
*** 000111 11001 00010 0000000000000100
*** 1    d    d    0    0    0    1    a      HEX
*** 0001 1100 0101 1001 0000 0000 0000 0100   BINARY
*** 0001 11ss ssst tttt iiii iiii iiii iiii   ENCODING
*********************************************************
*********************** else ****************************
******************** C = C << 3 *************************
*********************************************************
*** #36 sll $d, $s, $t
*** #36 sll $t2 (1010), $t2 (1010), $t9 (11001);
*** opcode $d    $s    $t          function field
*** 000000 01010 01010 11001 00000 000000
*** 1    d    d    0    0    0    1    a      HEX
*** 0000 0001 0101 1001 0101 0000 0000 0000   BINARY
*** 0000 00ss ssst tttt dddd d000 0000 0000   ENCODING
*********************************************************
********************* *dPtr = 7 *************************
*********************************************************
** #37 addi $t, $s, imm
** #37 addi $t3 (1011), $zero, 7;
** opcode $s    $t    imm
** 001000 00000 01011 0000000000000111
** 2    0    1    3    0    0    0    4       HEX
** 0010 0000 0000 1011 0000 0000 0000 0111    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
********************* G = E & F *************************
*********************************************************
*** #38 and $d, $s, $t
*** #38 and $t6 (1110), $t4 (1100), $t5 (1101)
*** opcode $d    $s    $t          function field
*** 000000 01110 01100 01101 00000 100100
*** 1    d    d    0    0    0    1    a      HEX
*** 0000 0001 1000 1101 0111 0000 0010 0100   BINARY
*** 0000 00ss ssst tttt dddd d000 0010 0100   ENCODING
*********************************************************
********************* Jump to L2 ************************
*********************************************************
*** #39 j target
*** #39 j L2
*** opcode L2: line 43 (00101011)
*** 000010 00000 00000 0000000000101011
*** 1    d    d    0    0    0    1    a      HEX
*** 0000 1000 0000 0000 0000 0000 0010 1011   BINARY
*** 0000 10ii iiii iiii iiii iiii iiii iiii   ENCODING
```

```
*********************************************************
***************** if (A - B) > 3 **********************
***************** L1: C = C + 4 ***********************
*********************************************************
** #40 addi $t, $s, imm
** #40 addi $t2 (1010), $t2 (1010), 4;
** opcode $s    $t    imm
** 001000 01010 01010 0000000000000100
** 2    0    1    3    0    0    0    4       HEX
** 0010 0001 0100 1010 0000 0000 0000 0100    BINARY
** 0010 00ss ssst tttt iiii iiii iiii iiii    ENCODING
*********************************************************
******************** D = C - 3 ************************
*********************************************************
*** #41 sub $d, $s, $t
*** #41 sub $t3 (1011), $t2 (1010), $t9 (11001);
*** opcode $s    $t    $d          function field
*** 000000 01010 11001 01011 00000 100010
*** 0    1    0    9    6    8    2    2      HEX
*** 0000 0001 0101 1001 0101 1000 0010 0010   BINARY
*** 0000 00ss ssst tttt dddd d000 0010 0010   ENCODING
*********************************************************
******************** G = E | F ************************
*********************************************************
*** #42 or $d, $s, $t
*** #42 or $t6 (1110), $t4 (1100), $t5 (1101)
*** opcode $s    $t    $d          function field
*** 000000 01100 01101 01110 00000 10 0101
*** 1    d    d    0    0    0    1    a      HEX
*** 0000 0001 1000 1101 0111 0000 0010 0101   BINARY
*** 0000 00ss ssst tttt dddd d000 0010 0101   ENCODING
*********************************************************
***************** L2: A = A + B **********************
*********************************************************
** #43 add $d, $s, $t
** #43 add $t0 (1000), $t0 (1000), $t1 (1001);
** opcode $d    $s    $t          function field
** 000000 01000 01000 01001 00000 100000
** 0    0    1    3    5    0    2    0       HEX
** 0000 0001 0000 1001 0100 0000 0010 0000    BINARY
** 0000 00ss ssst tttt dddd d000 0010 0000    ENCODING
*********************************************************
***************** temp1 = E ^ F ********************
*********************************************************
*** #44 xor $d, $s, $t
*** #44 xor $v1 (0011), $t4 (1100), $t5 (1101)
*** opcode $s    $t    $d          function field
*** 000000 01100 01101 00011 00000 100110
*** 1    d    d    0    0    0    1    a      HEX
*** 0000 0001 1000 1101 0001 1000 0010 0110   BINARY
*** 0000 00ss ssst tttt dddd d--- --10 0110   ENCODING
*********************************************************
***************** G = (E ^ F) & H ********************
```

```
*********************************************************
*** #45 and $d, $s, $t
*** #45 and $t6 (1110), $v1 (0011), $t7 (1111)
*** opcode $d    $s    $t          function field
*** 000000 01110 00011 01111 00000 100100
*** 1    d    d    0    0    0    1    a      HEX
*** 0000 0000 0110 1111 0111 0000 0010 0100   BINARY
*** 0000 00ss ssst tttt dddd d000 0010 0100   ENCODING
*********************************************************
```

Figure 2.3 MIPS Version and Binary Version Instructions

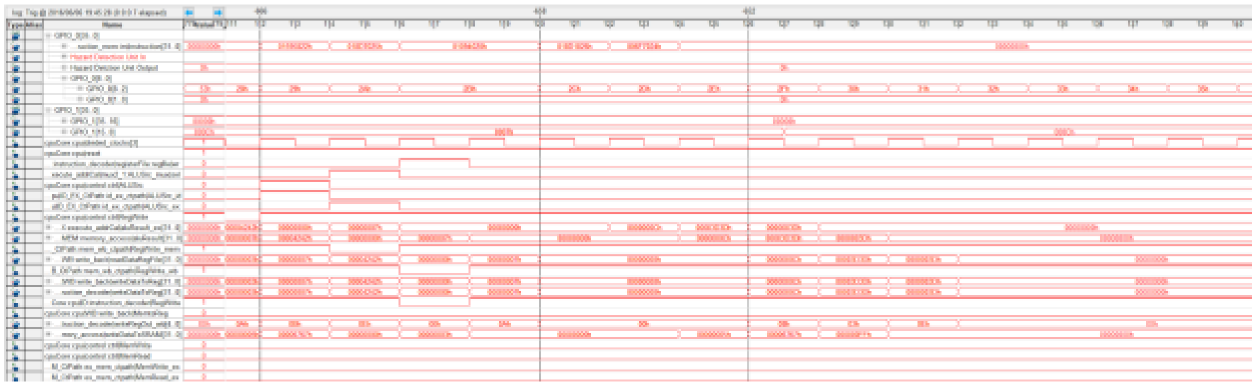

Figure 2.4.a SignalTap Result for Variable A (Register t0)

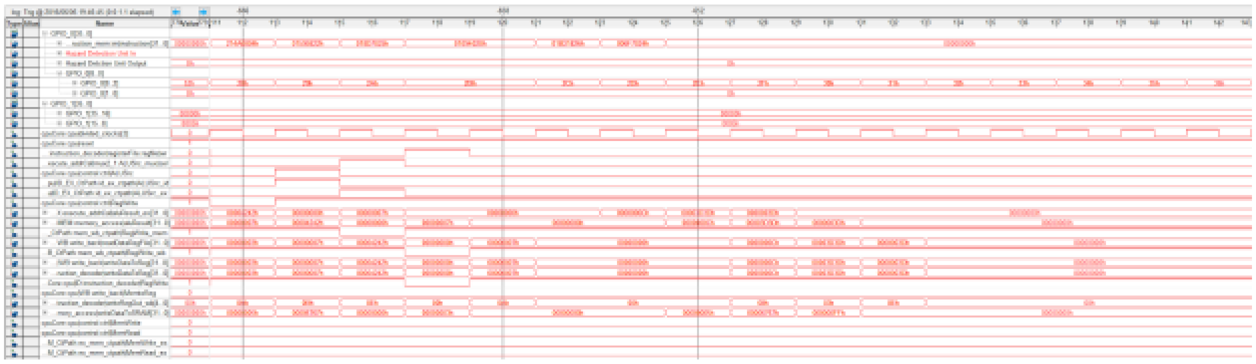

Figure 2.4.b SignalTap Result for Variable B (Register t1)

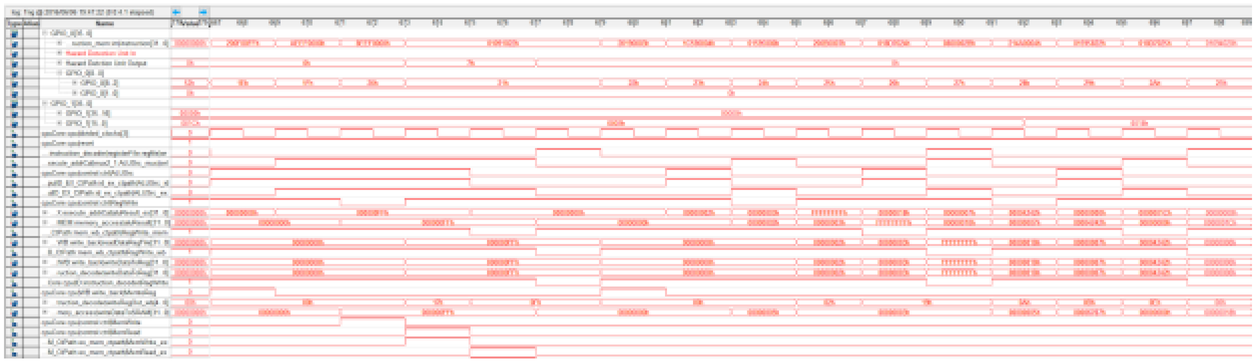

Figure 2.4.c SignalTap Result for Variable C (Register t2)

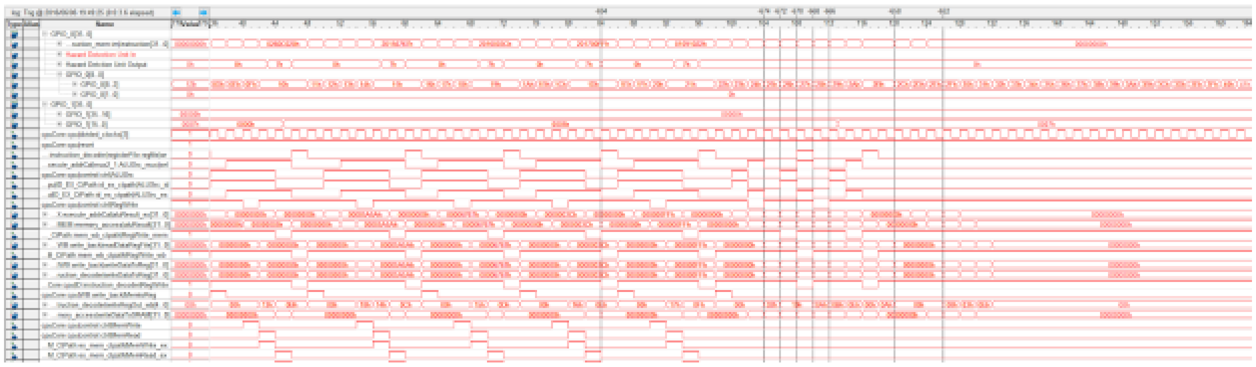
Figure 2.4.d SignalTap Result for Variable D (Register t3)

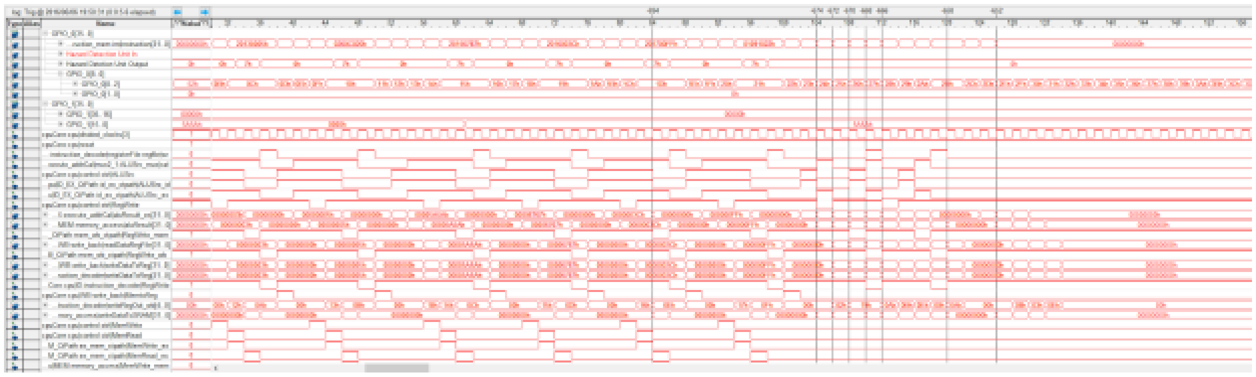
Figure 2.4.e SignalTap Result for Variable E (Register t4)

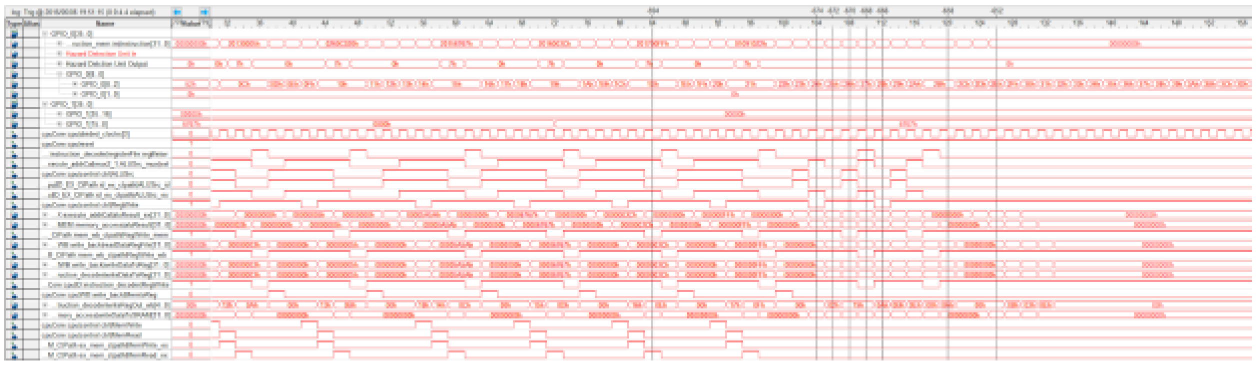
Figure 2.4.f SignalTap Result for Variable F (Register t5)

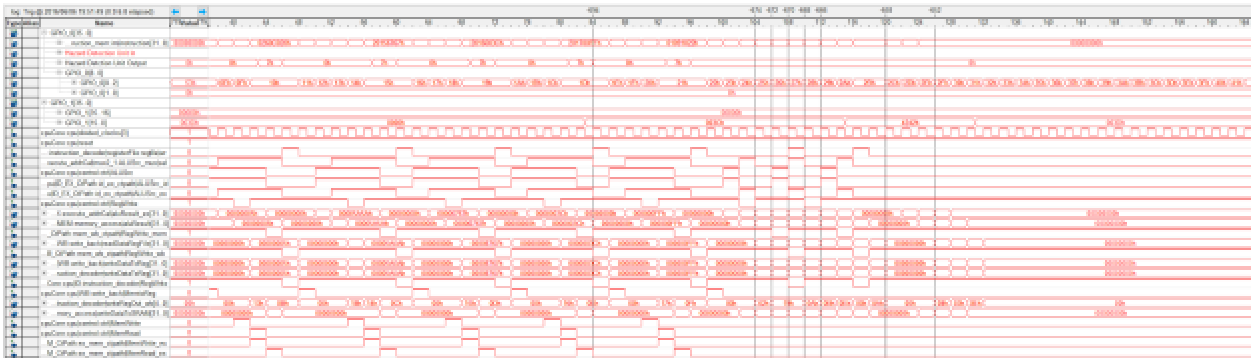
Figure 2.4.g SignalTap Result for Variable G (Register t6)

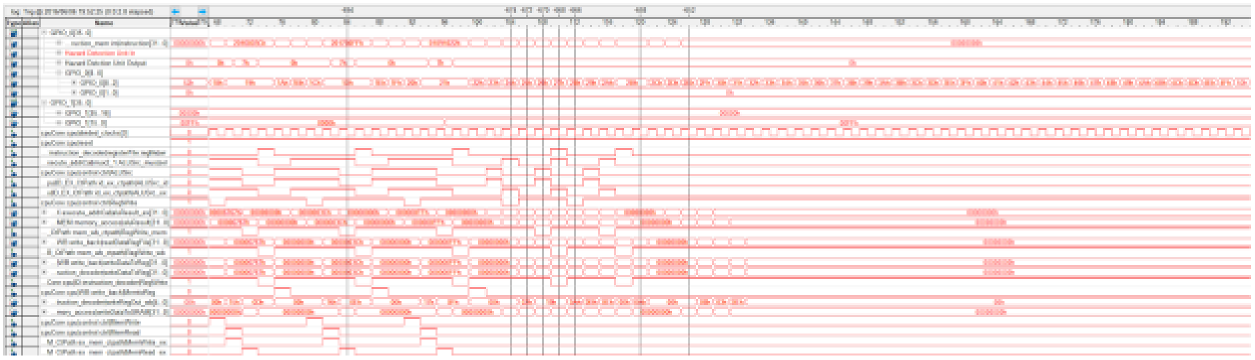
Figure 2.4.h SignalTap Result for Variable H (Register t7)

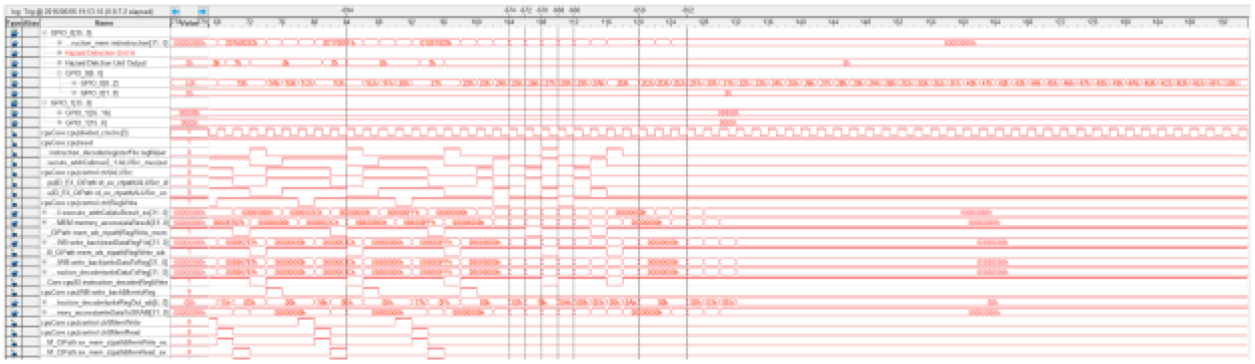
Figure 2.4.i SignalTap Result for Variable dPtr (Register t8)

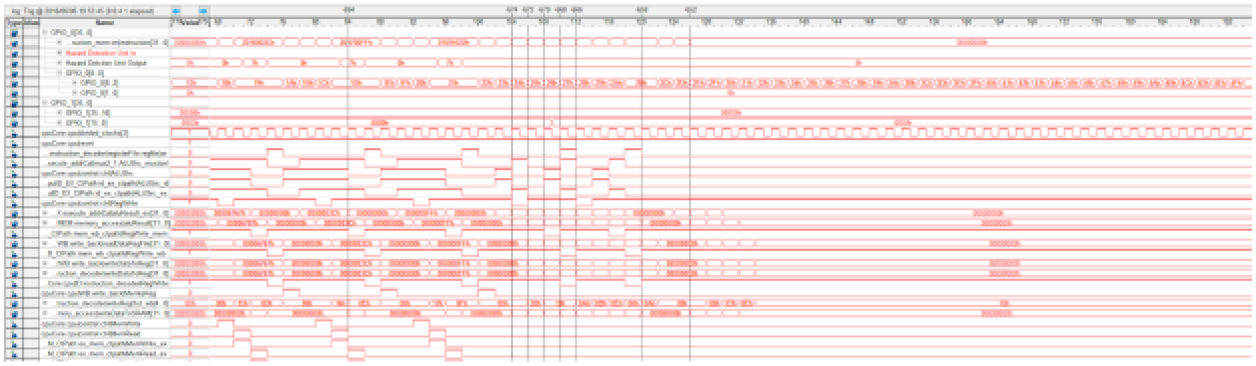
Figure 2.4.j SignalTap Result for Storing Constant Three (Register t9)

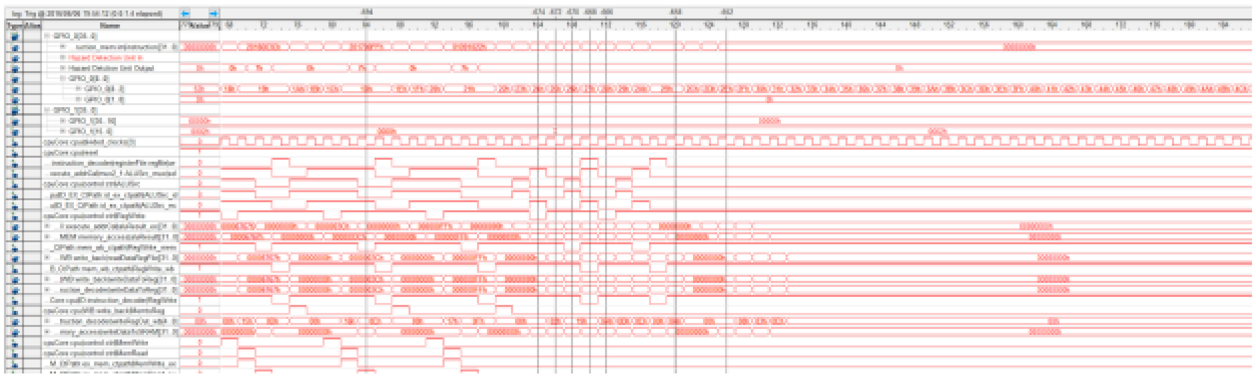

Figure 2.4.k SignalTap Result for Storing A - B (Register v0)

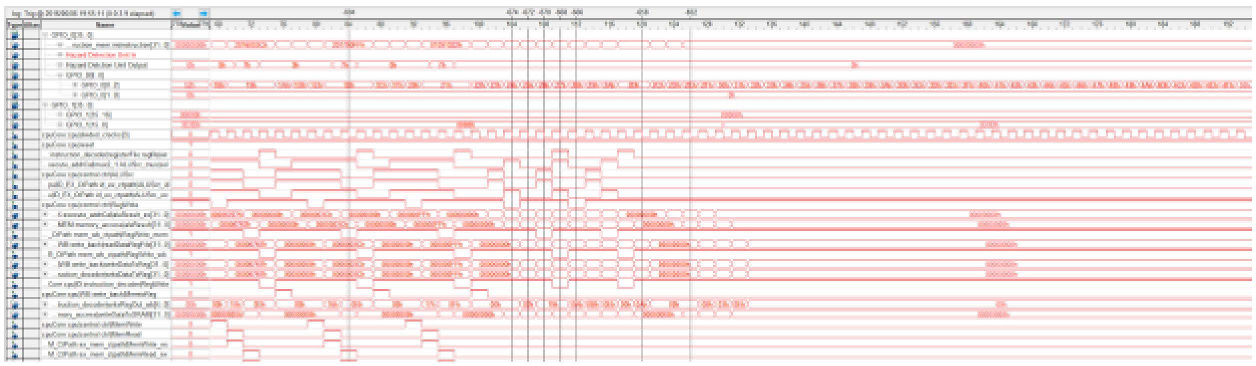

Figure 2.4.l SignalTap Result for Storing E ^ F (Register v1)

MEMBER CONTRIBUTION

Member	Contribution
X.O.	Design and testing and debug the program, related parts in report
L.H.	Design and testing and debug the program, related parts in report
Y.W.	Design and testing and debug the program, related parts in report