Table of Contents:

1. Abstract

The purpose of this lab was to fully integrate all of the past modules, namely the SRAM, the ALU, and the register memory into a single cycle MIPS architecture machine. This meant the creation of a program counter, instruction memory, and system control unit. This also included the creation of commands, namely the ALU operations, branch greater than, store word, load word, jump, and jump register. After this basic framework, the next step was pipeline all commands, which included adding in several interim registers and rewiring the design appropriately. This also created the need for hazard control and forwarding, as data could be calculated or retrieved incorrectly due to timing issues. This report begins with the design specifications of the system, then moves onto the procedures of how they were designed and implemented. Next, the test cases and analyses of the corresponding tests are discussed, and end with the summary.

2. Introduction

Adding on to the last lab, several modules are changed, and others are newly constructed and integrated. The memory and ALU are changed to be single cycle, and the framework for running through a program is implemented. The program counter is a single register that controls where the program is, and the instruction memory takes in the program counter value to find the instruction at that address. This also necessitates the inclusion of several checks to ensure the program counter can shift when encountering a branch, jump register, or jump command. With the addition of pipelining, the modules were further divided into several stages, and new controls involving forwarding and hazard control were added to avoid data and branch hazards. These modules were designed in a mix of behavioural and RTL level Verilog, and were tested through the use of iVerilog and several small test cases, in addition to large scale testing with a provided lab specification using SignalTap to view outputs.

3. Discussion Of the lab

3.1 Design Specification

3.1.1 Designing and Building the Program Counter and Instruction Memory

The first part of this project is to design and construct a control system to be able to both store the commands and access them. This leads to the creation of the program counter, a register that simply saved its input every positive edge of the clock, and instruction memory, which read an instruction out every positive edge of a clock cycle based on its input program counter address. The output of the program counter was directly connected to the input of the instruction memory, and the input to the program counter was the output of a variety of muxes, which were controlled by various signals which will be discussed more in the control section (3.1.2).

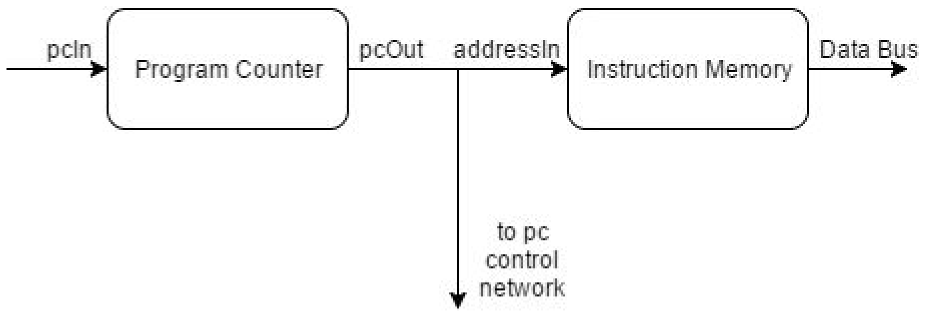


Figure 1. Program Counter and Instruction Memory

It is possible for the program counter to increment by four every clock cycle, to represent the size of a word. However, it was decided for simplicity, clarity, and ease of calculation that the current system would increment by one every clock cycle. The program counter would increment by one every clock cycle barring any address operations. These values would then go into instruction memory and return out the requisite instruction out to the data bus. This was mainly tested in iVerilog and eventually in the full integration with iVerilog and SignalTap.

3.1.2 Designing and Building Control

The next step of this design involves the creation of a control unit. Since the whole system is being combined together, a control unit is needed to change the state of the i/o bits for the memory modules and to manage all the data paths. Another control is also needed to decode or interpret the bits into what operation is being performed. This involves reading in the data bus and using combinational logic to change the control bits. The bits in particular are sramCS, sramWE, regWE, branch, jr, jump, aluSrc, regDst, memToReg, and aluOp, as shown below.

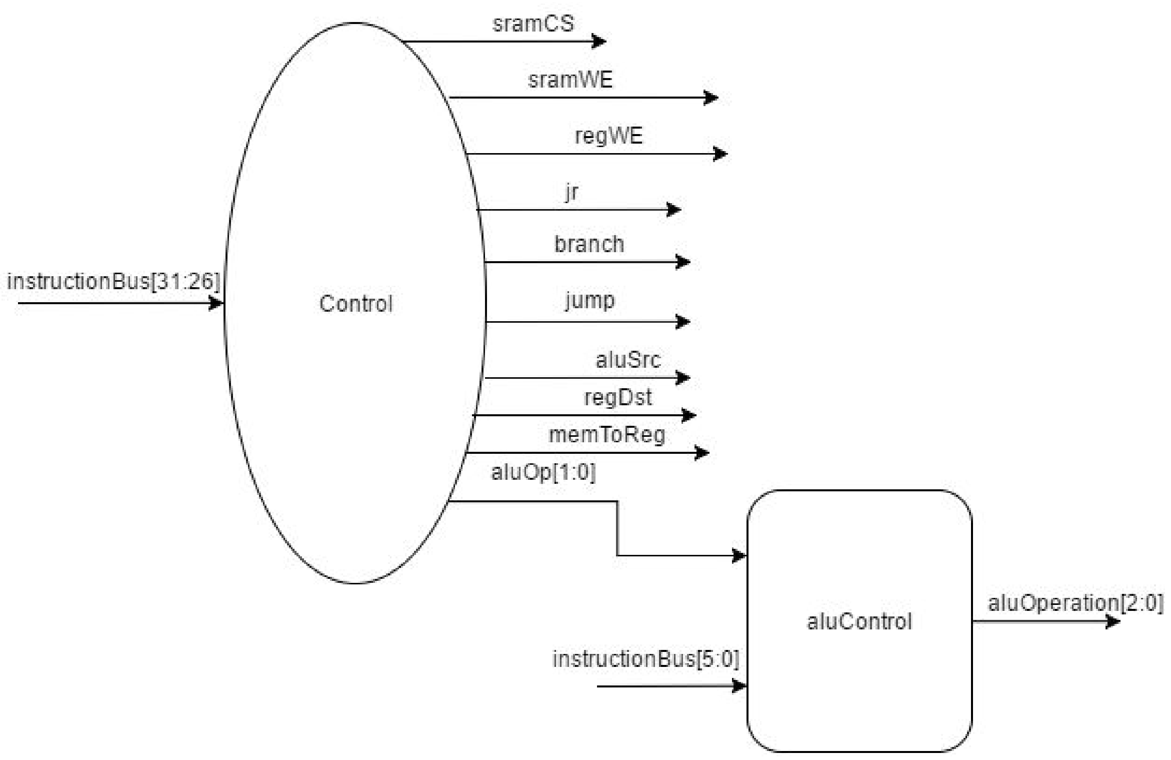


Figure 2. Control Bits

The bits sramCS and sramWE control the input and output out of the SRAM module. Both are low true, meaning no operation if sramCS is a 1, reading if sramWE is a 1, and writing if sramCS is a 0. To reach the single cycle operation, the register always reads out values, but does not always save them. The equivalent no operation, then, is regWE being a 1, or not writing anything back in (low true). The bit regDst controls which set address serves as the write address for the SRAM. A 1 denotes 15 to 11 in the data bus, whereas a 0 denotes 20 to 16. Similarly, memToReg controls where the output data back into the register comes from. A 1 is from the SRAM, and a 0 is from the ALU. The next 3 bits: jr, jump, and branch, all indicate a change or potential change in the program counter. Thus, they control a network of muxes that control the address going into the program counter, as seen below.

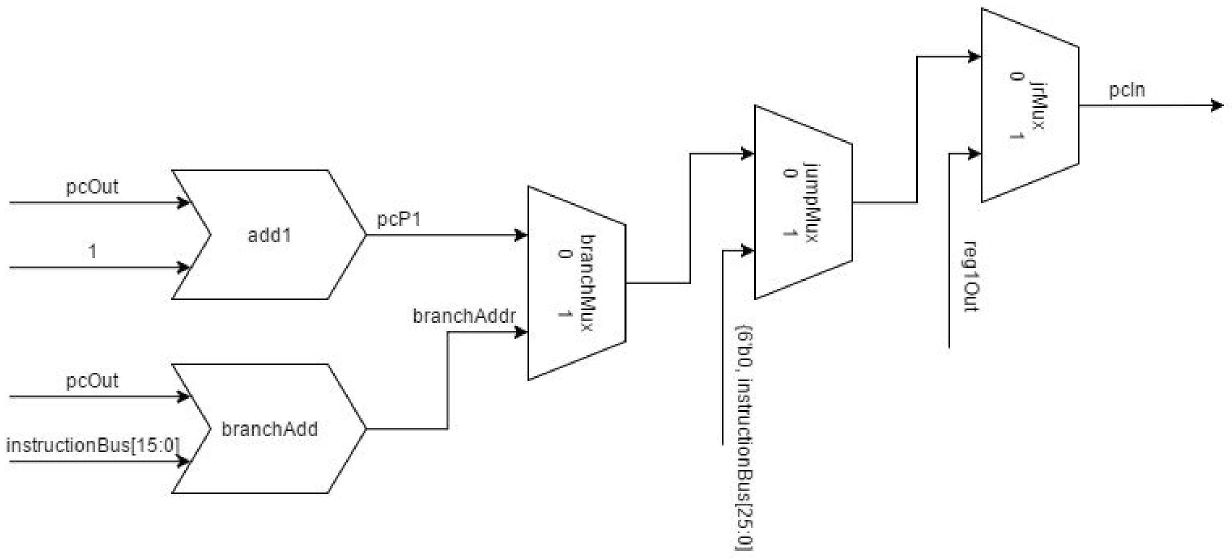
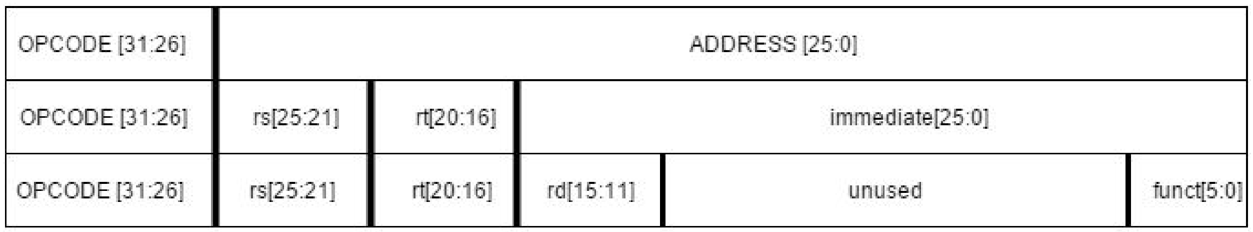


Figure 3. Program Counter Mux Control

For the ALU, aluSrc controls the input into the second ALU input. Specifically, it chooses between a second register output or an immediate 32-bit value. The ALU operation is determined by the aluOp output from control unit and the bottom 6 bits of the data bus known as the function field. There are several types of operations, namely R-type, store word/load word, jump, and branch greater than. R-type are ALU operations or jump register, jump is the opcode and an immediate 26 bit address, store word/load word has two 5-bit register addresses and a bottom 16-bit SRAM address, and branch greater than an immediate 16-bit offset value should the branch be taken, which can be seen below.



OPCODE [31:26]	ADDRESS [25:0]				
OPCODE [31:26]	rs[25:21]	rt[20:16]	immediate[25:0]		
OPCODE [31:26]	rs[25:21]	rt[20:16]	rd[15:11]	unused	funct[5:0]

Figure 4. Operation Type Bit Distributions

3.1.3 Designing and Building Pipeline Functionality

In order to add pipeline functionality, the machine must be subdivided into each step, so all modules can be kept running simultaneously. Between each of these steps is a register, to hold the values from the step before and after the current step finishes. The current design has the steps split into fetch to instruction decode, instruction decode to execute, execute to memory, and memory to write back. Many of these registers contained forwarded values, because in order to maintain correct timing, they must be in the same step as their data, and so the circuit is reconnected accordingly. An overview of design can be seen in Figure 5.

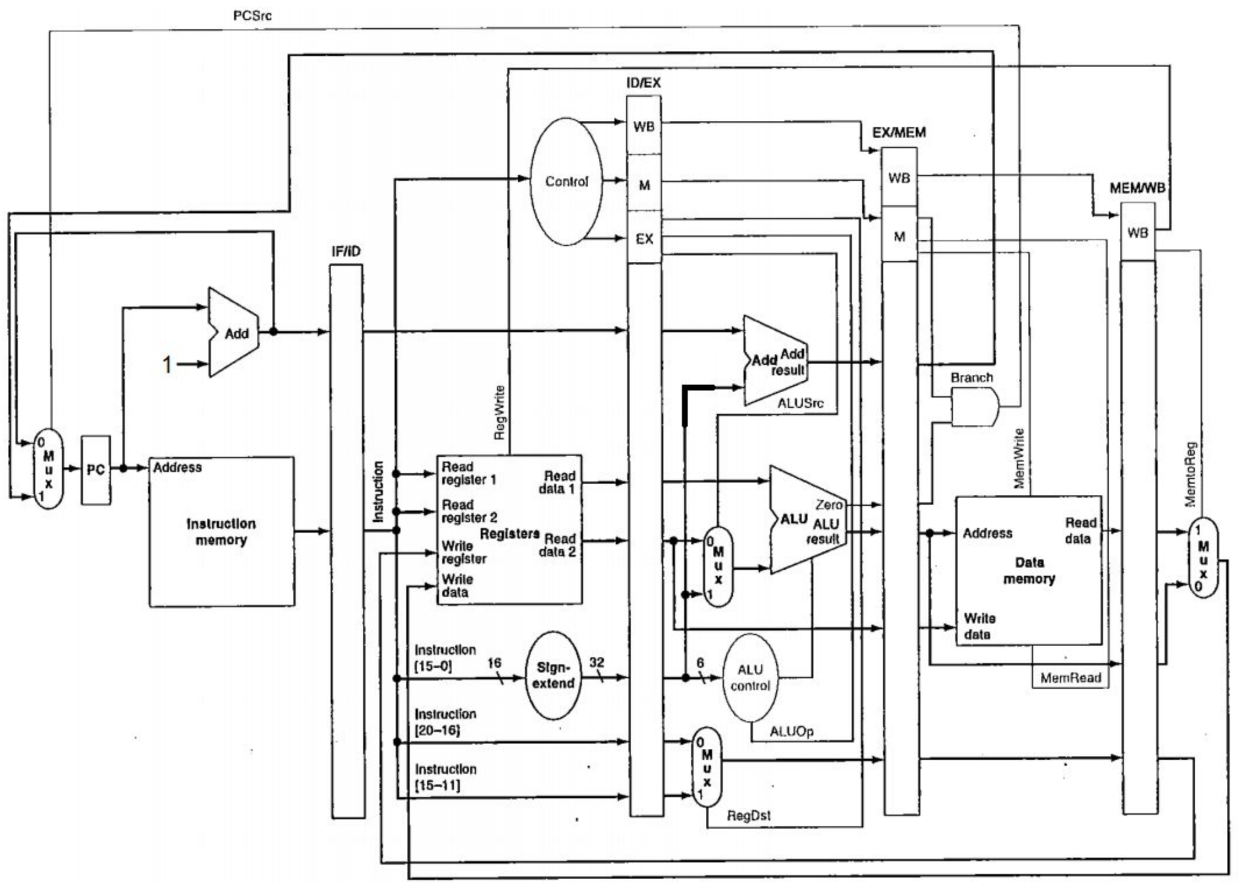


Figure 5. Circuit with Pipeline Registers (source: Patterson and Hennessy Chap. 4)

The write back bits are regWE and memToReg, the memory bits are sramWE, branch, jump, and jr, and the execution bits are regDst, aluSrc, and aluOp. However, when switching over from a single cycle to a pipelined design as above, timing issues arise. For example, when accessing the same register directly after an operation is done on it to change its value will return a wrong answer. This is because the result of the first operation has not actually been saved back into the register yet, and thus the wrong value is being input into the ALU. This necessitates the inclusion of a forwarding unit. Essentially, whenever an operation is performed, the address of where it will write to is saved. Then, if a future command attempts to use the written address as one of its inputs, the forwarding unit, as seen in Figure 6, will change which values are input into the ALU, or in other words, forwarding the result to the ALU so it can be used before it is saved. Another case is when loading a word, there must be a stall of 1 cycle so the instruction can actually load the data out of data memory before it can be forwarded. Essentially, the stalled instruction becomes a no operation, and the previous instruction is made into the stalled instruction.

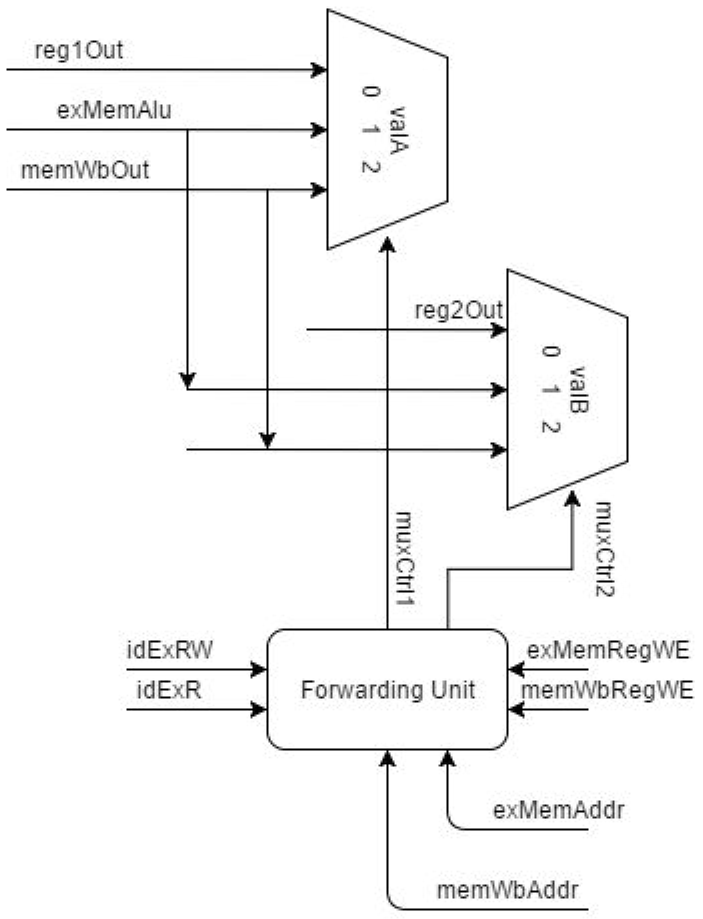


Figure 6. Forwarding Unit

There are a few more hazards which are particularly dangerous. These have to do with jumping, jump register, and branch. Since jump and jump register are evaluated in the fetch-decode stage, there is a delay of 1 clock cycle. Thus, the instruction stored directly before the jump is invalid. This is solved with the inclusion of flush, which flushes the values out of the fetch-decode register and makes it a no operation command. There is a second issue, namely branch. Branch is calculated in the decode-execution stage. This is partially solved by adding another flush to the decode-execution register. However, it is inefficient to stall for two cycles without doing any operation. Thus, dynamic branch prediction is used instead. The machine begins by predicting that it will take the branch. If it does, there is no problem. If it does not, it reverts to the branch instruction's address plus one, and flushes the registers. When predicting the not to take the branch, and branching, it goes to the calculated address, and the registers are flushed. Figure 7 is a diagram of the design used as reference.

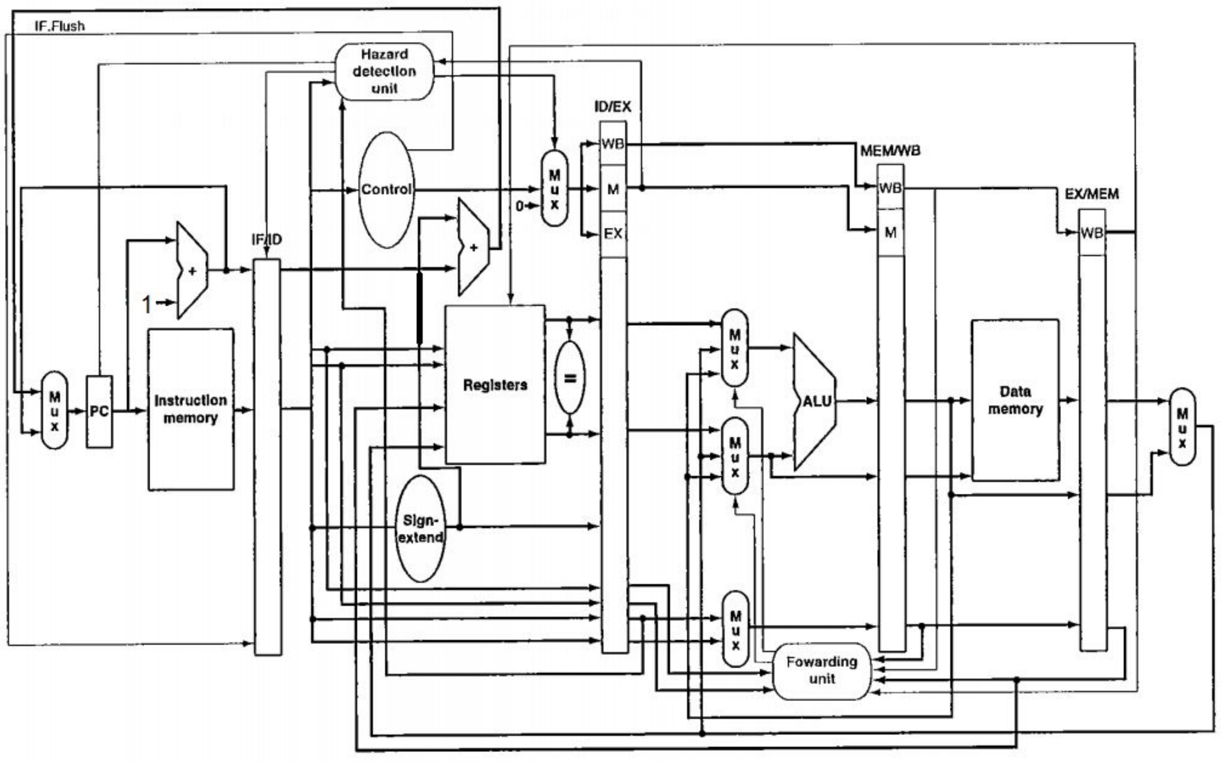


Figure 7. Reference Design (source: P & H text Chap. 4)

3.2 Design Procedure

3.2.1 Program Registers and System Control

Behavioral Level Model:

The first step was to simply implement an iterator and its associated memory to be able to step through the program. This involved the creation of an adder that incremented the program counter by 1 every clock cycle. The next step was to be able to calculate the addresses necessary for jumping, jump register, and branching. Jump register was simply the address stored within the referenced register. Jump was the immediate 26 bit output given in the command, and thus could be wired directly, and branch was an offset.

This led to the creation of a carry-look ahead adder to be able to calculate the needed address within one clock cycle, accepting the program counter value and the immediate 16-bit offset. These values were then input into a chain of muxes controlled by their respective signals. If it was going to branch, it would select the branch address rather than the old address plus one. This output would be compared to the jump address, and later the jump register address before being placed back into the program counter.

The actual control unit, and ALU control, were implemented through the use of a behavioural state machine, which changed states based on input. Control was dependent on the top 6 bits of the data bus, or the opcode. ALU control was based on the bottom 6 bits of the data bus, or the function field, as well as the aluOp output from control, determining what kind of format it was using. The cases can be seen below.

Opcode	sramWE	regDst	branch	regWE	aluOp	aluSrc	memToReg	jump	jumpReg
default	1	1	0	1	00	1	1	0	0
000000 (jr)	1	x	x	1	xx	x	x	0	1
000000 (alu)	1	1	0	0	10	0	0	0	0
100011 (lw)	1	0	0	0	00	1	1	0	0
101011 (sw)	0	1	0	1	00	1	x	0	0
000100 (bgt)	1	x	1	1	01	x	x	0	0
000010 (j)	1	x	0	1	00	x	x	1	0
001000 (addi)	1	0	0	0	11	1	0	0	0

Table 1. Control Bit Cases

aluOp	funct	output
00	N/A	001 (add)
01	N/A	000 (NOP)
10	100000	001 (add)
10	100010	010 (sub)
10	100100	011 (and)
10	100101	100 (or)
10	100110	101 (xor)
10	101010	110 (slt)
10	000000	111 (sll)
11	N/A	001 (addi)

Table 2. ALU Operation State Machine

3.2.2 Pipeline Registers

After constructing the single cycle computer, the next step was to divide the circuit into its pipeline stages. This led to the creation of ifId, idEx, exMem, and memWb, all intermediary registers. All clocked in new values at the positive clock edge. The registers ifId and idEx additionally had the flush inputs, which would flush all internal values to zero. Implementations of flush are in the next section. The purpose of ifId was to simply hold the output of the instruction memory as well as a saved value of the program counter, needed for branch hazards later. After ifId, the data was used to set control flags and get data out of the register. In idEx, the necessary stored elements were the outputs of the register, the read address (25:21), the read/write address (20:16), and the write address (15:11), needed later for the forwarding unit. It also stored the sign-extended immediate bits and the control bits, to be forwarded with each step so each step would be synchronized with the correct set of bits. Specifically, idEx contained execution bits, memory bits, and write-back bits, exMem contained memory and write-back bits, and memWb contained only write-back bits. Next, exMem contained the output of the ALU, the B input into the ALU, the write-back address to the register, and a flag register. The flag register was to check for branch hazards, discussed in the next section, and the B input was to control the data input into the SRAM. Finally, in memWb, the register contained the output of the ALU, the output from SRAM, the write-back bits, and the write-back address.

3.2.3 Hazards - Data and Branching

With the design now hooked up in a pipeline architecture, it became necessary to deal with data and branch hazards. The forwarding unit was implemented using behavioural level Verilog, for convenience and clarity, as it is essentially combinational logic. It takes in the two addresses from idEx, and checks if either of them are equal to the write-back addresses stored in memWb or exMem, where exMem is checked before (the logic is after) memWb, to ensure the most recent value is used. Each comparison controls one of the two input muxes. If the read address operand matches, it affects the first mux, and the same extends to the second mux when dealing with the second operand address. A 2-to-1 mux is implemented after the 3-to-1 mux for input B to maintain the ability to input immediate values. The forwarding unit also takes in the write-back control bits from exMem and memWb. This is to ensure accidental data is not forwarded back, if it is not being saved and there is an edge case with overlapping addresses.

The hazard unit was implemented using behavioural level Verilog and contains a register for a stored address along with a state machine for dynamic branch prediction. Whenever the machine predicts correctly, it moves towards the respective state of strongly predicts or strongly does not predict, if in the normal predict stage. If it predicts wrong, it moves from the strong state to the weak state, or from the weak state to the weak state of the other polarity. The states can be seen in the table below.

State	Meaning	If Taken	If Not
00	Strong Y	00	01
01	Weak Y	00	10
10	Weak N	01	11
11	Strong N	10	11

Table 3. Dynamic Branch Prediction

The hazard unit takes in the read/write register address, sramWE from idEx, the ifId data bus, whether or not the branch was taken, the current and last ALU instructions, a jump check, which includes both jump and jump register, an input address, and the clock. The input address is used for stalling, the current ALU instruction for dynamic prediction and stalling, the last instruction for stalling, the data bus for stalling, sramWE for stalling, and the read/write register for stalling. Due to time constraints and the lack of requirement in the specification, the actual stall mechanism has been left unimplemented, though the framework is still present.

Flushing happens in two particular cases: after a bad branch and after a jump or jump register. When jumping or jump registering, one instruction, the one stored in ifId is no longer valid and thus must be flushed to a no operation. When dealing with a bad branch, the values in ifId and idEx must be set to no operations, as the calculation for actually taking the branch occurs in the idEx stage, thus resulting in a delay of 2 cycles.

3.1 System Description

3.3.1 Program Counter, Control, ALU Control

- Summary:
  Six modules: pc (program counter), instructionMem (instruction memory), control (system bit control), aluControl (ALU control), add1 (program counter iterator), pcAdder (branch offset calculator)
- Algorithm Description
  - 32-bit program counter
  - 128x32-bit instruction memory
  - Combinational logic control unit
  - Combinational logic ALU control unit
  - Add 1 adder
  - Carry-look ahead adder of program counter and immediate value
- Inputs:
  - pc: pcIn, clk

    - instructionMem: pcOut, clk
    - control: dataBus[31:0] (eventually changed to ifIdBus)
    - aluControl: aluOp[1:0], dataBus[5:0] (becomes ifIdBus)
    - add1: pcOut, 1
    - pcAdder: pcOut, dataBus[15:0] (changes to ifIdPcP1 and ifIdBus)
    - pcOut is determined by mux network, which is controlled by control bits
- Output:
    - pc: pcOut
    - instructionMem: iMemOut
    - control: regDst, branch, sramWE, memToReg, jump, jumpReg, aluOp, aluSrc, regWE
    - aluControl: aluOpOut[2:0]
    - add1: pcP1
    - pcAdder: pcBranch
- Timing Constraints:
    - Register modules are clocked by the positive edge of the clock and run in 1 cycle
    - Other modules are combinational and run with only gate delay
- Error Handling:
    - Signal Tap logic analyzer
    - iVerilog
    - GTKWave

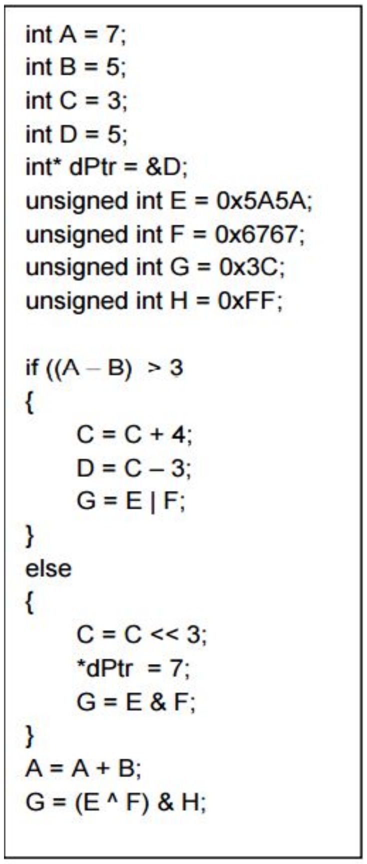
3.3.2 Pipeline, Hazard Unit, Forwarding Unit

- Summary:
    - Six modules: ifId (fetch-decode register), idEx (decode-execute register), exMem (execute-memory register), memWb (memory-write back register), fwdUnit (forwarding unit), hazardUnit (hazard unit)
- Algorithm Description
    - ifId 64-bit register, storing program counter value and instruction memory output
    - idEx 153-bit register, storing register outputs, extended immediate bits, program counter value, branch predict, register addresses, write-back control bits, memory control bit, and execution control bits
    - exMem 58-bit register, storing ALU output, write address, flags, ALU B input, write-back bits, and memory bits
    - memWb 71-bit register, storing ALU output, SRAM output, write address, and write-back bits
    - fwdUnit is combinational logic such that when writing back in a later step, it compares that address to the input operand addresses and adjusts the ALU inputs accordingly.
    - hazardUnit checks dynamic state and predicts accordingly. It adjusts the state once the branch is calculated, and sets the flag if the program counter needs to reset.
    - Flushing controlled by hazard unit output and jumping control bits.
- Inputs:

    - ifId: iMemOut, pcP1, flushIf, clk
    - idEx: reg1Out, reg2Out, immediate, pcP1IfId, ifIdBus[25:11], wbCtrl, mCtrl, exCtrl, flushId, clk
    - exMem: aluOut, regWAddress, wbIdEx, mIdEx, z, v, c, n, aluB[10:0], clk
    - memWb: sramOut, exMemAlu, exMemW, wbExMem, clk
    - hazardUnit: regWAddr, mIdEx[3] (sramWE), ifIdBus, bSel, exBits[1:0] (aluOp), aluOp (current aluOp), jOut (jump and jump register), idExPc, clk
    - Register is between ifId and idEx, ALU is between idEx and exMem, and SRAM is between exMem and memWb.
- Output:
    - ifId: ifIdBus, pcP1IfId
    - idEx: idExR1, idExR2, idExImm, idExPc, idExRegRW, idExRegR, idExRegW, wbIdEx, mIdEx, exBits, bpOut (branch predict)
    - exMem: exMemAlu, exMemW, wbExMem, mExMem, flags, exMemB
    - memWb: memWbSram, memWbAlu, memWbW, wbMemWb
    - fwdUnit: aluMuxCtrl1, aluMuxCtrl2
    - hazardUnit: bPredict, stall, stallAddr
- Timing Constraints:
    - Register modules are clocked by the positive edge of the clock and run in 1 cycle
    - Other modules are combinational and run with only gate delay
    - Bad branches lose 2 cycles, jumps lose 1
- Error Handling:
    - Signal Tap logic analyzer
    - iVerilog
    - GTKWave

3.4 Software Implementation

In terms of software, we were provided with a C code fragment which we hand-compiled into MIPS assembly language, which was then converted into corresponding machine code for our CPU.



```
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 8. C Code Required for Instruction Set

Each line of the code fragment broken up into individual instructions for execution. The *If, Else* statement is implemented with dynamic branch prediction to determine whether the program counter should continue as normal or jump to a different instruction.

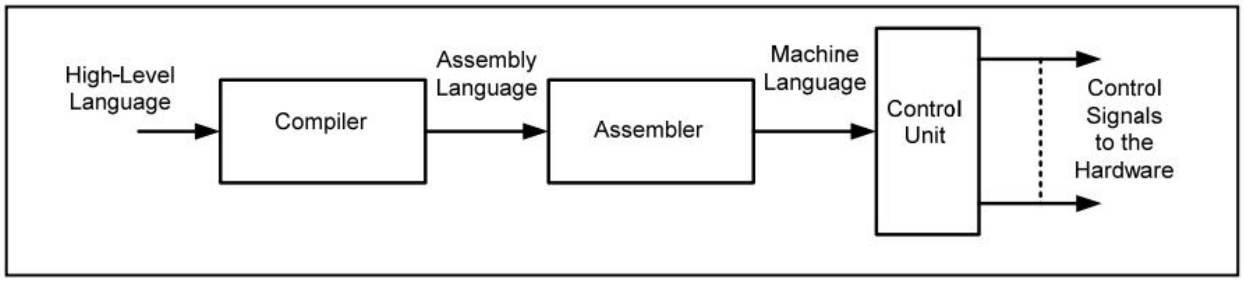


Figure 9. C Code Compiler Modules

3.5 Hardware Implementation

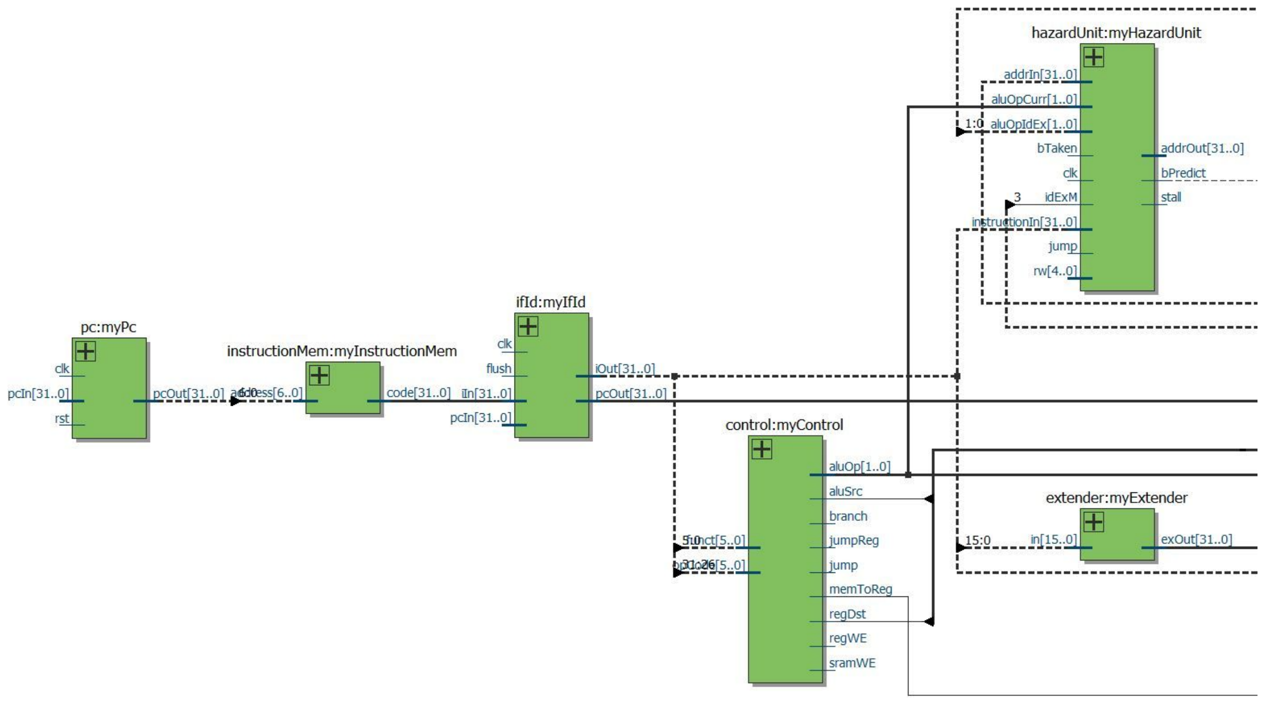
3.5.1 System Top-Level Block Diagram

A top-level diagram of the entire system is shown in Figure 9. The pipeline process can be broken down into five steps:

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

From left to right:

The program counter, pc, tells which next instruction is to be executed. This corresponding instruction is then fetched from instructionMem. This instruction is saved at this stage with the ifid register. This instruction is sent to the hazardUnit, in case of a failed prediction such that the system can easily revert back to the last correct instruction. The instruction is then sent to control to be decoded and the output is then saved in the idEx register. From here, the desired ALU operation and function code from the instruction is sent to aluControl which controls the ALU. The output calculated within the ALU is saved into the exMem register which is connected to SRAM for possible storage, or into memWb which determines if the result should be written back into the register files regMem. Along the pipeline, the forwarding unit,, fwdUnit, works to forward information quickly back to different stages as necessary to streamline the process.



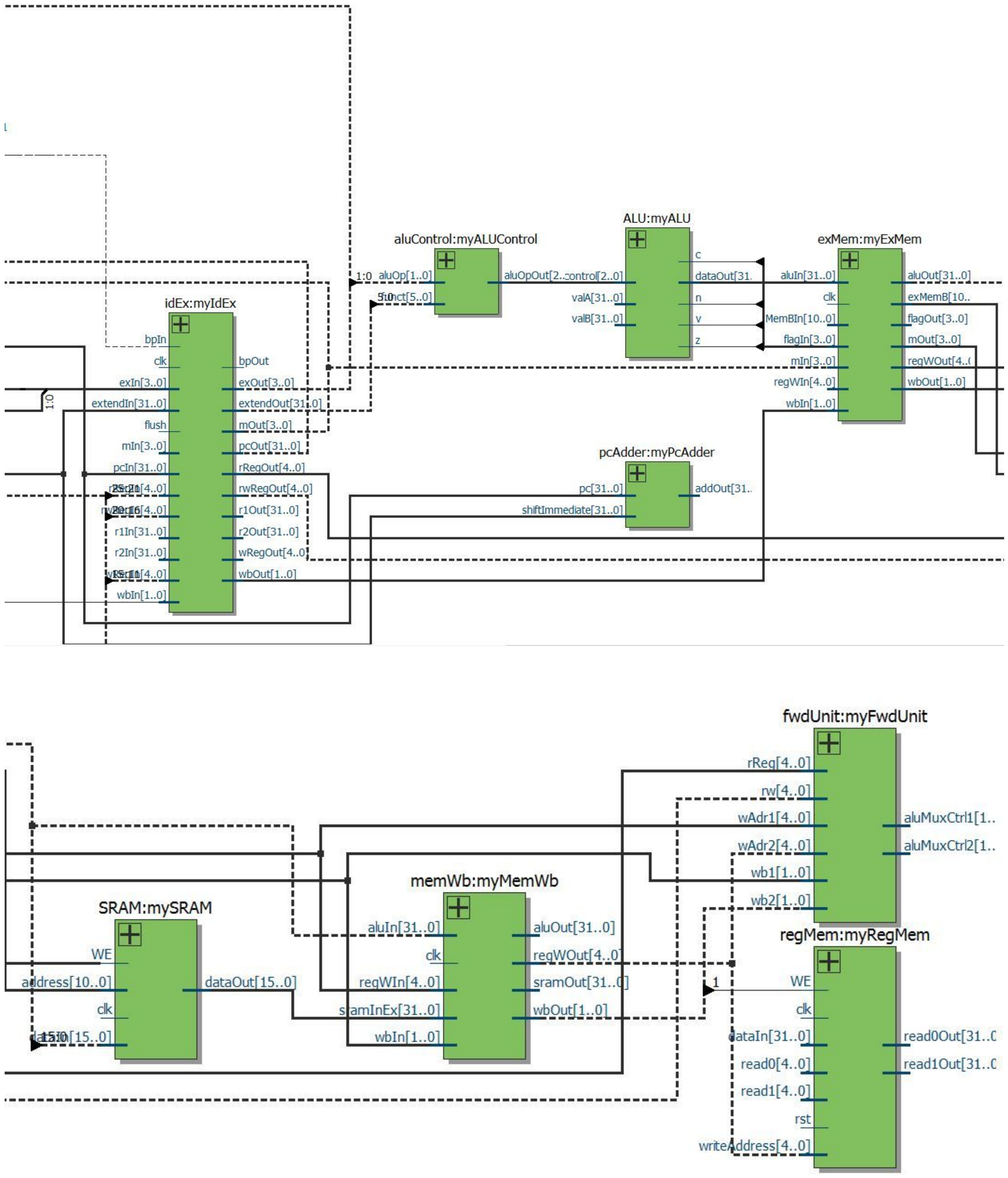


Figure 10. RTL Viewer of System in Quartus

4 Test Procedure

4.1 Test Plan

4.1.1 Program Counter, Control, ALU Control

- **GTKWave**: We created individual testbenches for each of the six modules for this section to be tested within iVerilog and GTKWave. In these cases, they were not rigorously tested - we forwent complicated cases in favor of testing the basic functionality, as we decided more thorough testing would be easier within the integrated system. For the Program Counter, we confirmed it was simply able to take in the given instruction, which would be selected through outside modules/muxes. Later on in integration, we verified muxes that provided input to the Program Counter were controlled as intended. For the Control, we confirmed the state machine worked as intended to decode the given instruction, breaking it down to the 9 main control signals, and passing along information to the ALU Control which assisted in the decoding. We ran these tests against the clock to confirm operation within one clock cycle.

4.1.2 Pipeline, Hazard Unit, Forwarding Unit

- **GTKWave**: Just as above, we created a simple individual testbench for the Hazard Unit. For the Pipeline, complete with the Forwarding Unit, we again integrated the system as a whole. For each possible ALU operation, we created individual text files that would run through and test the entire pipeline. After each operation on its own was properly executed, we tested various combinations of multiple operations before finally testing the provided C code fragments. These tests were also ran against the clock to confirm operation within one clock cycle.
- **DE1-SoC and Signal Tap**: Using the instruction files from the provided C code fragments, we verified the core functionality of our design using the Signal Tap Logic Analyzer. The signals corresponding to the current Program Counter instruction and ALU Outputs were tracked in real time to further confirm the system as working.

4.2 Test Specification

4.2.1 Program Counter, Control, ALU Control

Because the Program Counter simply acts as a register, we simply tested to confirm it would hold whatever input was provided until the next clock edge.

For the Control, we confirmed our state machine functionality by testing each valid possible opCode and function code:

- Inputs: opCode, funct;
- Outputs: regDst, branch, sramWE, regWE, aluOp, aluSrc, memToReg, jump, jumpReg;

We also verified these signals properly controlled each mux that controlled the program counter input, to insure that the program counter received the proper input when intended.

For the ALU Control, we again confirmed state machine functionality by observing the output aluOpout given and aluOp code and function code.

- Inputs: aluOp, funct (corresponding to NOP, ADD, SUB, AND, OR, XOR, SLT, SLL)
- Outputs: aluOpOut (code corresponding to ALU operations)

4.2.2 Pipeline, Hazard Unit, Forwarding Unit

For the Hazard Unit, we only tested part of its functionality - specifically the dynamic branch prediction state machine. Other functions would be tested with the pipeline as a whole.

- Input: bTaken (if branch was taken)
- Output:branchP (next branch prediction)

Because we confirmed functionality of the individual pieces, part, we tested the Pipeline and Forwarding unit once the system was integrated.

- Inputs: instruction text file
- Outputs: aluOut, pcOut

This was a matter of creating various text files testing the operation of different instructions. The output from the ALU and program counter was observed to confirm the right numbers were being used and received.

4.3 Test Cases

4.3.1 Program Counter, Control, ALU Control

The Program Counter itself was easily confirmed to hold its provided input each clock cycle. Because it was designed as a register, it correctly held any input given until the next clock edge. We input various 32 bit numbers, for example 32’b0 and 32’b1, to confirm this. The muxes controlling program counter input were each tested as well. For example, if we jump or jump register, the new instruction should be passed through to the program counter, rather than the next sequential instruction. The Control and ALU Control were simply tested with one of each type of operation: Jump/Branch, Load/Store Word, and Arithmetic type. For the arithmetic cases, we used values 1 and 2 in the testing of each operation. These test cases changed somewhat during the pipeline process.

4.3.2 Pipeline, Hazard Unit, Forwarding Unit

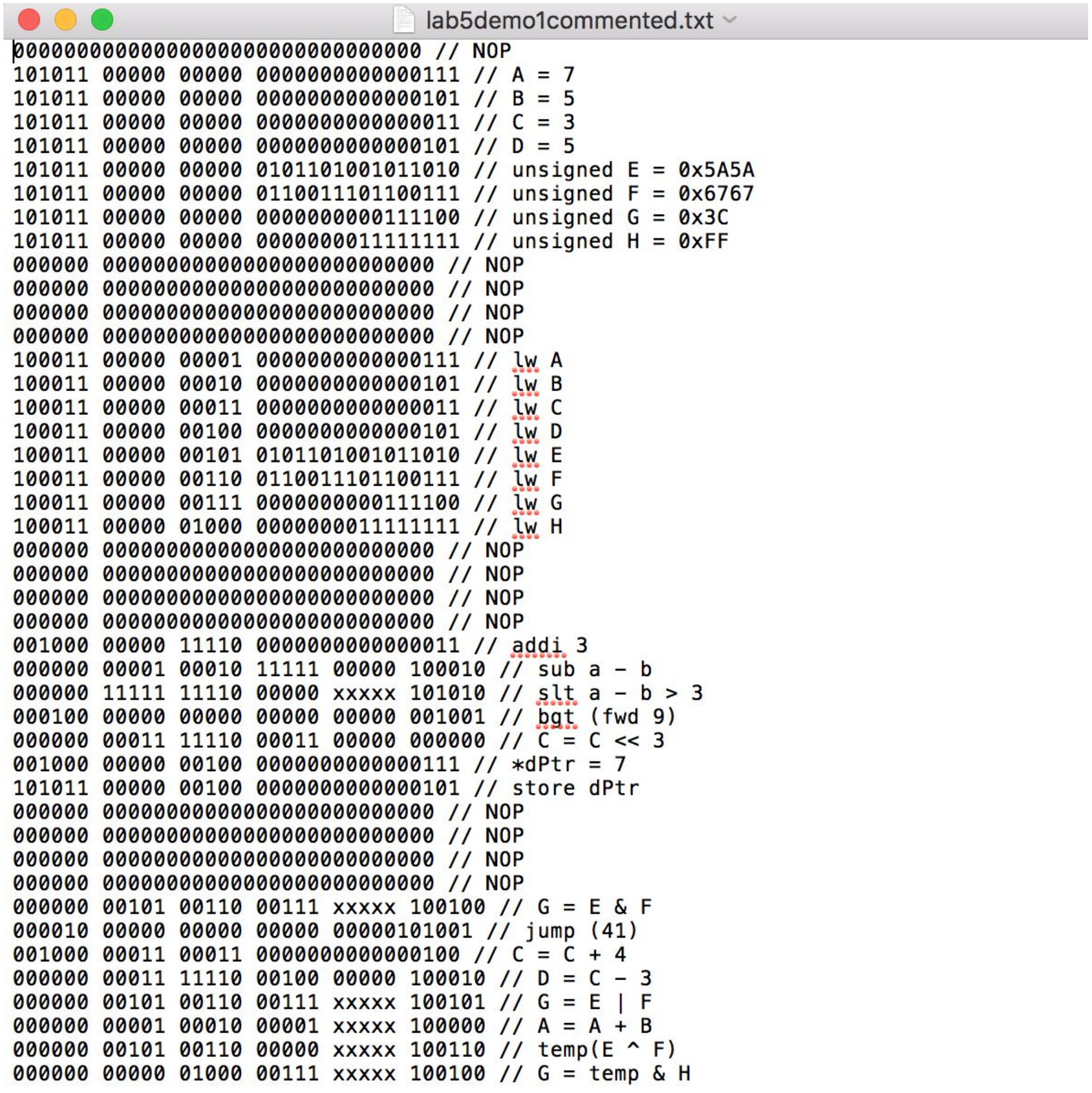
During the pipeline process the previous sections were now tested in the following way:

- **Program Counter** : The test cases are MIPS instruction set files we made based on the C program in Figure 8. Each type of instruction set is decoded by control units and the output from control unit is ALUop which determines branch, store/load words, and arithmetic operations. The instruction set file also contains the case of jump/ jump register. When branch prediction is wrong, a mux controlling program counter input should cause the program counter to reset before moving on to correct next instruction. We tested two different cases for the control to this: both incorrect and correct branch prediction. If the result from ALU for "A-B" is not greater than 3, the program counter jumps to else case. As seen Figure 11, after branch greater than operation done, program counter fetches the instruction forwarded by 9 because the condition is not satisfied. Our test case is to see the pattern of program counter addressing a correct instruction in the case of branch and jump.

- **Control** : To test the control unit, we should check how program counters behave when control outputs jump and jump registers as well as ALU output. Since the control unit manages all the data path in CPU, the behavior depends on the control state. In our test case, control have to enable register memory and data memory when we load/ store the data and add immediate values.

- **ALU Control** : ALU is controlled by 2bits ALUop outputs from control unit; 2b'00 is the load/store word, 2'b10 is arithmetic operation , and 2'b01 is branch greater than operation. In our test case, ALU control output 3'b000 which is no operation in ALU control bits in the case of branch.

To test the pipeline modules, there were several smaller test cases, and the two required test cases, seen in figure 8. Specifically, each operation was isolated and tested within the bounds of the pipeline machine. This meant jump, jump register, branch greater than, load word, store word, add immediate, and all of the ALU operations (add, subtract, AND, OR, XOR, set less than, and shift left logical). Each test case utilized dummy variables. In the jump and jump register cases, a value of 13 was stored in register 2, and then jumped to with the jump register command. The thirteenth command was a jump command that jumped to the end of the program. To test store word and load word, a value of 1 was stored into the SRAM address of 1, then loaded out to both registers 1 and 2. To test if the values were loaded out properly, the add ALU operation was then performed on the two registers. For the other ALU operations, the values of 9 and 12 were immediately added into the registers and then compared. They were ANDed, ORed, XORed, subtracted, and shifted. Then, both cases of branch greater than were tested. 2 and 1 were immediately added into registers, and one case involved comparing 2 to 1, and the other 1 to 2. This created a case where the branch would fail and one where the branch would succeed. It would be expected in both cases to take the branch, and in the case where the branch failed, for the program counter to return to it's previous value after 1 cycle. Finally, after all of the small scale testing, the large C program in Figure 8 was implemented. This combined all of the previous small scale testing together and allowed for testing of various hazards as well as how it would perform with more complex strings of instructions.



lab5demo1commented.txt

```
00000000000000000000000000000000 // NOP
101011 00000 00000 0000000000000111 // A = 7
101011 00000 00000 0000000000000101 // B = 5
101011 00000 00000 0000000000000011 // C = 3
101011 00000 00000 0000000000000101 // D = 5
101011 00000 00000 0101101001011010 // unsigned E = 0x5A5A
101011 00000 00000 0110011101100111 // unsigned F = 0x6767
101011 00000 00000 0000000000111100 // unsigned G = 0x3C
101011 00000 00000 0000000011111111 // unsigned H = 0xFF
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
100011 00000 00001 0000000000000111 // lw A
100011 00000 00010 0000000000000101 // lw B
100011 00000 00011 0000000000000011 // lw C
100011 00000 00100 0000000000000101 // lw D
100011 00000 00101 0101101001011010 // lw E
100011 00000 00110 0110011101100111 // lw F
100011 00000 00111 0000000000111100 // lw G
100011 00000 01000 0000000011111111 // lw H
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
001000 00000 11110 0000000000000011 // addi 3
000000 00001 00010 11111 00000 100010 // sub a - b
000000 11111 11110 00000 xxxxx 101010 // slt a - b > 3
000100 00000 00000 00000 00000 001001 // bgt (fwd 9)
000000 00011 11110 00011 00000 000000 // C = C << 3
001000 00000 00100 0000000000000111 // *dPtr = 7
101011 00000 00100 0000000000000101 // store dPtr
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00000000000000000000000000 // NOP
000000 00101 00110 00111 xxxxx 100100 // G = E & F
000010 00000 00000 00000 0000101001 // jump (41)
001000 00011 00011 0000000000000100 // C = C + 4
000000 00011 11110 00100 00000 100010 // D = C - 3
000000 00101 00110 00111 xxxxx 100101 // G = E | F
000000 00001 00010 00001 xxxxx 100000 // A = A + B
000000 00101 00110 00000 xxxxx 100110 // temp(E ^ F)
000000 00000 01000 00111 xxxxx 100100 // G = temp & H
```

Figure 11. Instruction set text file

5. PRESENTATION/DISCUSSION/ANALYSIS OF THE RESULTS

5.1 Program Counter, Control, ALU Control

Because testing for the Program Counter, Control, and ALU Control involved only simple cases, it was easy to verify the basic core functionality of the modules. Through iVerilog and GTKWave, we confirmed the Program Counter was able to take in the selected instruction each clock cycle. The state machine for the Control and ALU control was also able to decode instructions covering Branch/Jump, Immediate, Arithmetic Type cases, turning on the 9 control signals as appropriate. Though these modules were functioning, it was only during integration that we discovered small mistakes in our control, leading to signals inadvertently turned on/off when unintended. These typos in control were appropriately debugged in the end, and eventually functioned completely as intended.

5.2 Pipeline, Hazard Unit, Forwarding Unit

To test the pipeline architecture, there were several test cases simply testing one aspect or another of the design. This included one for the ALU operations, where 32'b1001 and 32'b1101 were added, subtracted, ANDed, ORed, XORed, set less that, and shift left logical. The output of these tests matched expected output. More tests were run for jump and jump register, and those outputs were also correct, implying that the jump muxes and other pieces of architecture were synthesized correctly. Store word and load word also matched expected output. Next, tests were done on the branch greater than cases, both taking and failing the branch. This isolated case displayed correct output. When actually synthesizing on the board, however, the branch case demonstrated an error where it would jump to the required offset, but then encounter no operation encounters to the end of the program, rather than jumping back or continuing on with the rest of the stored instructions.

6. ANALYSIS

6.1 Analysis of any Errors

In designing a single cycle cpu, we encountered that register memory is not able to read and store data because of timing issue. Since single cycle cpu has only one register memory reading and writing at the same time, that is, reading out addressed data and writing back the result from ALU in one clock cycle. Our previous design couldn't handle that case since we had two enables, OE(Output Enable) and WE(Write Enable) that are activated in positive edge of the clock input. However, we figured out that register can read data anytime not affected by clock edge or enable since our control unit only controls writing. After we fixed the register memory by removing OE for reading case, we could see outputs in Gtkwave. For pipelining cpu, wire connections made errors several times since the cpu design got more complicated than single cycle one. After checking all the data path from mux controls for branch prediction, forward units, and hazard unit, we figured out that 3:1 mux design was different from expected

The main issue when synthesizing the pipeline MIPS architecture onto the board was that somehow, after branching to the required offset, instead of executing the instruction there or jumping back to the address directly after the branch, there was simply just no operations until the program counter reset and began again from 0. This implies one of three things. Firstly, that the values into instruction memory were not properly saved. This would mean that at the branch value and after, some error caused the instructions to m=not be saved in properly. The second case is an error with flush, where somehow flush would be set to 1 after the branch and thus just flush all output until the end. The third case is an error somewhere else in integration, which is unlikely due to every other command working. Thus to identify and fix the problem, it would be ideal to SignalTap the output again, this time including the flush bits, and the output of the ifId and instruction memory registers.

6.2 Analysis of Why the Project may not have Worked and What Efforts Were Made to Identify the Root Cause of any Problems

The design worked perfectly in iVerilog and GTKWave, as well as in synthesis on the day before. However, the failure of the branch operation leading to no operations until reset implies one of three things. Metastability, instructions not being properly saved in instruction memory, or an actual integration/flush error in the code. If given enough time, to check for these issues, the first step would be to SignalTap and look at the values coming out of instruction memory and the ifId register. This would allow easy analysis of each step to isolate where the error is occurring, catching if the instructions are not saving and if the flush command is faultily connected. Given some time, this one error should be quickly identified and fixed.

7. Summary and Conclusion

7.1 Summary

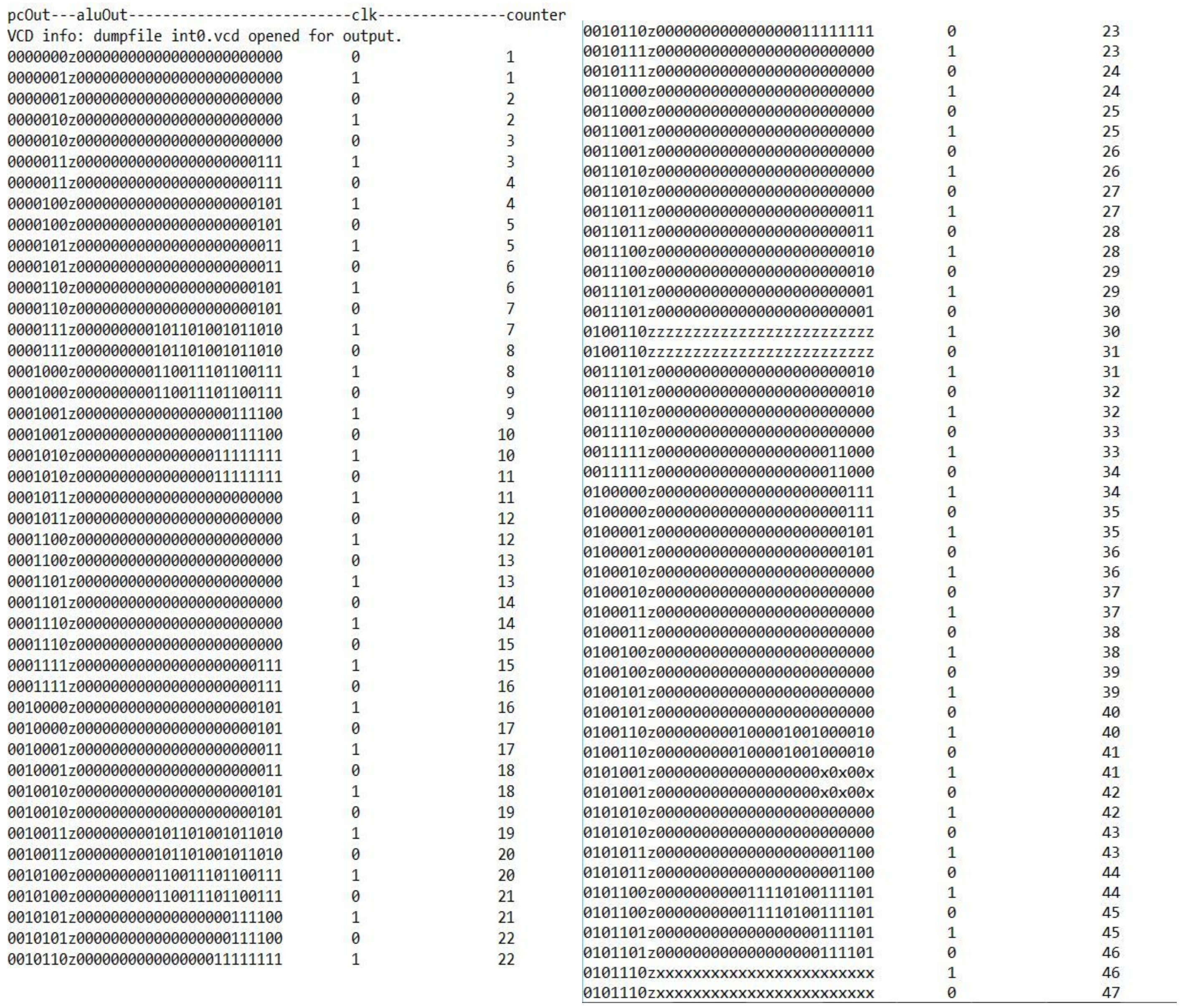
This lab focused on the construction and integration of a control system and pipeline characteristics. The control components were a program counter, instruction memory, system control, and ALU control, which all had to run in a single cycle and be able to execute various commands. It had to support the ALU operations as well as loading and storing words, branching, jumping and jumping to addresses stored in registers. This necessitated the ability for control and ALU control to be able to manipulate the other bits running the system. After this design was finished, the next step was to implement a pipeline architecture. This meant the construction of several pipeline registers along the data path, specifically four; one between fetching and decoding, one between decoding and execution, one between execution and memory, and one between memory and write-back. However, with the implementation of this design, there arose timing issues, where branches could run wrong commands due to being calculated on a later step, and data being accessed in succession would be incorrect, as it would not be able to use the updated value due to it not yet being saved into memory. This led to the creation of a forwarding unit, allowing for results in the later stages to be forwarded for use in the earlier steps, removing the need to stall, and a hazard unit, to add dynamic branch prediction and to appropriately flush and reset the program counter if the branch prediction was incorrect. In actual testing, the programmed design compiled and worked perfectly in iVerilog and GTKWave. However, when synthesized onto the board, there was one error involving the branch command. Specifically, it would predict the branch, but after it jumped, instead of continuing on with the program or resetting, back, the system seemed to encounter a no operation instead of the commands saved at that register in instruction memory, thus not doing anything else until the program counter reset and began again from the beginning. This implies there was some potential error when connecting modules when transferring from iVerilog to system Verilog, or that there was some error with values not properly being saved into instruction memory. Regardless, all of the other commands worked perfectly, and could be seen through the SignalTap screenshots, and given enough time, the small error could be identified and fixed.

7.2 Conclusion

Although there was a slight error in the actual synthesis onto the board, the code worked perfectly in iVerilog and GTKWave. However, even with this pipeline architecture, there is still much room for improvement. Although a smart compiler can avoid these problems, there are problems when loading a word and then using it in an operation the step after, as it must propagate an additional cycle before the data is actually fetched by SRAM. This would lead to the need for a stall mechanism, where the program counter would stall and set the stored ifId register to a no operation. Though not completed in the current code, the framework support is there for such a mechanism. This design not only covered the basics of how each module worked, but how they performed together in tandem, with the need to accept inputs and output to different stages to deal with timing issues. This project also left much room for improvement, seen in the implementation of our four-state dynamic branch prediction. Thus, this project covered much of the MIPS pipeline architecture, while leaving much room for later improvement, development, and thought.

8. APPENDIX:

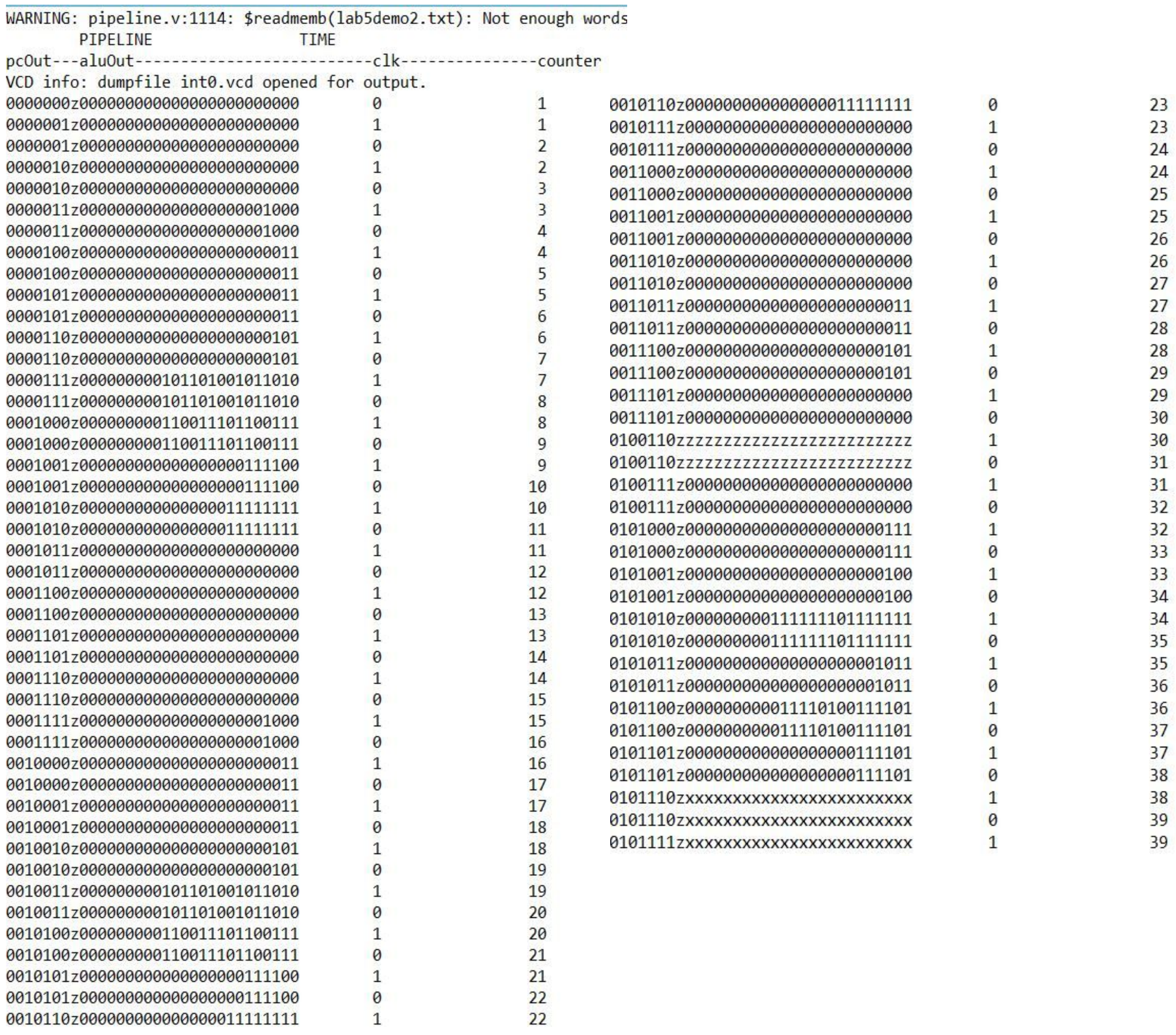
Command Outputs:



```
pcOut---aluOut--------------------------clk---------------counter
VCD info: dumpfile int0.vcd opened for output.
0000000z000000000000000000000000        0               1
0000001z000000000000000000000000        1               1
0000001z000000000000000000000000        0               2
0000010z000000000000000000000000        1               2
0000010z000000000000000000000000        0               3
0000011z000000000000000000000111        1               3
0000011z000000000000000000000111        0               4
0000100z000000000000000000000101        1               4
0000100z000000000000000000000101        0               5
0000101z000000000000000000000011        1               5
0000101z000000000000000000000011        0               6
0000110z000000000000000000000101        1               6
0000110z000000000000000000000101        0               7
0000111z000000000101101001011010        1               7
0000111z000000000101101001011010        0               8
0001000z000000000110011101100111        1               8
0001000z000000000110011101100111        0               9
0001001z000000000000000000111100        1               9
0001001z000000000000000000111100        0              10
0001010z000000000000000011111111        1              10
0001010z000000000000000011111111        0              11
0001011z000000000000000000000000        1              11
0001011z000000000000000000000000        0              12
0001100z000000000000000000000000        1              12
0001100z000000000000000000000000        0              13
0001101z000000000000000000000000        1              13
0001101z000000000000000000000000        0              14
0001110z000000000000000000000000        1              14
0001110z000000000000000000000000        0              15
0001111z000000000000000000000111        1              15
0001111z000000000000000000000111        0              16
0010000z000000000000000000000101        1              16
0010000z000000000000000000000101        0              17
0010001z000000000000000000000011        1              17
0010001z000000000000000000000011        0              18
0010010z000000000000000000000101        1              18
0010010z000000000000000000000101        0              19
0010011z000000000101101001011010        1              19
0010011z000000000101101001011010        0              20
0010100z000000000110011101100111        1              20
0010100z000000000110011101100111        0              21
0010101z000000000000000000111100        1              21
0010101z000000000000000000111100        0              22
0010110z000000000000000011111111        1              22
```

```
0010110z000000000000000011111111        0               23
0010111z000000000000000000000000        1               23
0010111z000000000000000000000000        0               24
0011000z000000000000000000000000        1               24
0011000z000000000000000000000000        0               25
0011001z000000000000000000000000        1               25
0011001z000000000000000000000000        0               26
0011010z000000000000000000000000        1               26
0011010z000000000000000000000000        0               27
0011011z000000000000000000000011        1               27
0011011z000000000000000000000011        0               28
0011100z000000000000000000000010        1               28
0011100z000000000000000000000010        0               29
0011101z000000000000000000000001        1               29
0011101z000000000000000000000001        0               30
0100110zzzzzzzzzzzzzzzzzzzzzzzzz        1               30
0100110zzzzzzzzzzzzzzzzzzzzzzzzz        0               31
0011101z000000000000000000000010        1               31
0011101z000000000000000000000010        0               32
0011110z000000000000000000000000        1               32
0011110z000000000000000000000000        0               33
0011111z000000000000000000011000        1               33
0011111z000000000000000000011000        0               34
0100000z000000000000000000000111        1               34
0100000z000000000000000000000111        0               35
0100001z000000000000000000000101        1               35
0100001z000000000000000000000101        0               36
0100010z000000000000000000000000        1               36
0100010z000000000000000000000000        0               37
0100011z000000000000000000000000        1               37
0100011z000000000000000000000000        0               38
0100100z000000000000000000000000        1               38
0100100z000000000000000000000000        0               39
0100101z000000000000000000000000        1               39
0100101z000000000000000000000000        0               40
0100110z000000000100001001000010        1               40
0100110z000000000100001001000010        0               41
0101001z000000000000000000x0x00x        1               41
0101001z000000000000000000x0x00x        0               42
0101010z000000000000000000000000        1               42
0101010z000000000000000000000000        0               43
0101011z000000000000000000001100        1               43
0101011z000000000000000000001100        0               44
0101100z000000000011110100111101        1               44
0101100z000000000011110100111101        0               45
0101101z000000000000000000111101        1               45
0101101z000000000000000000111101        0               46
0101110zxxxxxxxxxxxxxxxxxxxxxxxx        1               46
0101110zxxxxxxxxxxxxxxxxxxxxxxxx        0               47
```

Figure 12. Output of First Executed C Program



```
WARNING: pipeline.v:1114: $readmemb(lab5demo2.txt): Not enough words
       PIPELINE                TIME
pcOut---aluOut--------------------------clk---------------counter
VCD info: dumpfile int0.vcd opened for output.
0000000z000000000000000000000000       0                1
0000001z000000000000000000000000       1                1
0000001z000000000000000000000000       0                2
0000010z000000000000000000000000       1                2
0000010z000000000000000000000000       0                3
0000011z000000000000000000001000       1                3
0000011z000000000000000000001000       0                4
0000100z000000000000000000000011       1                4
0000100z000000000000000000000011       0                5
0000101z000000000000000000000011       1                5
0000101z000000000000000000000011       0                6
0000110z000000000000000000000101       1                6
0000110z000000000000000000000101       0                7
0000111z000000000101101001011010       1                7
0000111z000000000101101001011010       0                8
0001000z000000000110011101100111       1                8
0001000z000000000110011101100111       0                9
0001001z000000000000000000111100       1                9
0001001z000000000000000000111100       0               10
0001010z000000000000000011111111       1               10
0001010z000000000000000011111111       0               11
0001011z000000000000000000000000       1               11
0001011z000000000000000000000000       0               12
0001100z000000000000000000000000       1               12
0001100z000000000000000000000000       0               13
0001101z000000000000000000000000       1               13
0001101z000000000000000000000000       0               14
0001110z000000000000000000000000       1               14
0001110z000000000000000000000000       0               15
0001111z000000000000000000001000       1               15
0001111z000000000000000000001000       0               16
0010000z000000000000000000000011       1               16
0010000z000000000000000000000011       0               17
0010001z000000000000000000000011       1               17
0010001z000000000000000000000011       0               18
0010010z000000000000000000000101       1               18
0010010z000000000000000000000101       0               19
0010011z000000000101101001011010       1               19
0010011z000000000101101001011010       0               20
0010100z000000000110011101100111       1               20
0010100z000000000110011101100111       0               21
0010101z000000000000000000111100       1               21
0010101z000000000000000000111100       0               22
0010110z000000000000000011111111       1               22
0010110z000000000000000011111111       0               23
0010111z000000000000000000000000       1               23
0010111z000000000000000000000000       0               24
0011000z000000000000000000000000       1               24
0011000z000000000000000000000000       0               25
0011001z000000000000000000000000       1               25
0011001z000000000000000000000000       0               26
0011010z000000000000000000000000       1               26
0011010z000000000000000000000000       0               27
0011011z000000000000000000000011       1               27
0011011z000000000000000000000011       0               28
0011100z000000000000000000000101       1               28
0011100z000000000000000000000101       0               29
0011101z000000000000000000000000       1               29
0011101z000000000000000000000000       0               30
0100110zzzzzzzzzzzzzzzzzzzzzzzzz       1               30
0100110zzzzzzzzzzzzzzzzzzzzzzzzz       0               31
0100111z000000000000000000000000       1               31
0100111z000000000000000000000000       0               32
0101000z000000000000000000000111       1               32
0101000z000000000000000000000111       0               33
0101001z000000000000000000000100       1               33
0101001z000000000000000000000100       0               34
0101010z000000000111111101111111       1               34
0101010z000000000111111101111111       0               35
0101011z000000000000000000001011       1               35
0101011z000000000000000000001011       0               36
0101100z000000000011110100111101       1               36
0101100z000000000011110100111101       0               37
0101101z000000000000000000111101       1               37
0101101z000000000000000000111101       0               38
0101110zxxxxxxxxxxxxxxxxxxxxxxxx       1               38
0101110zxxxxxxxxxxxxxxxxxxxxxxxx       0               39
0101111zxxxxxxxxxxxxxxxxxxxxxxxx       1               39
```

Figure 13. Output of Second Executed C Program

GTKWave Outputs:

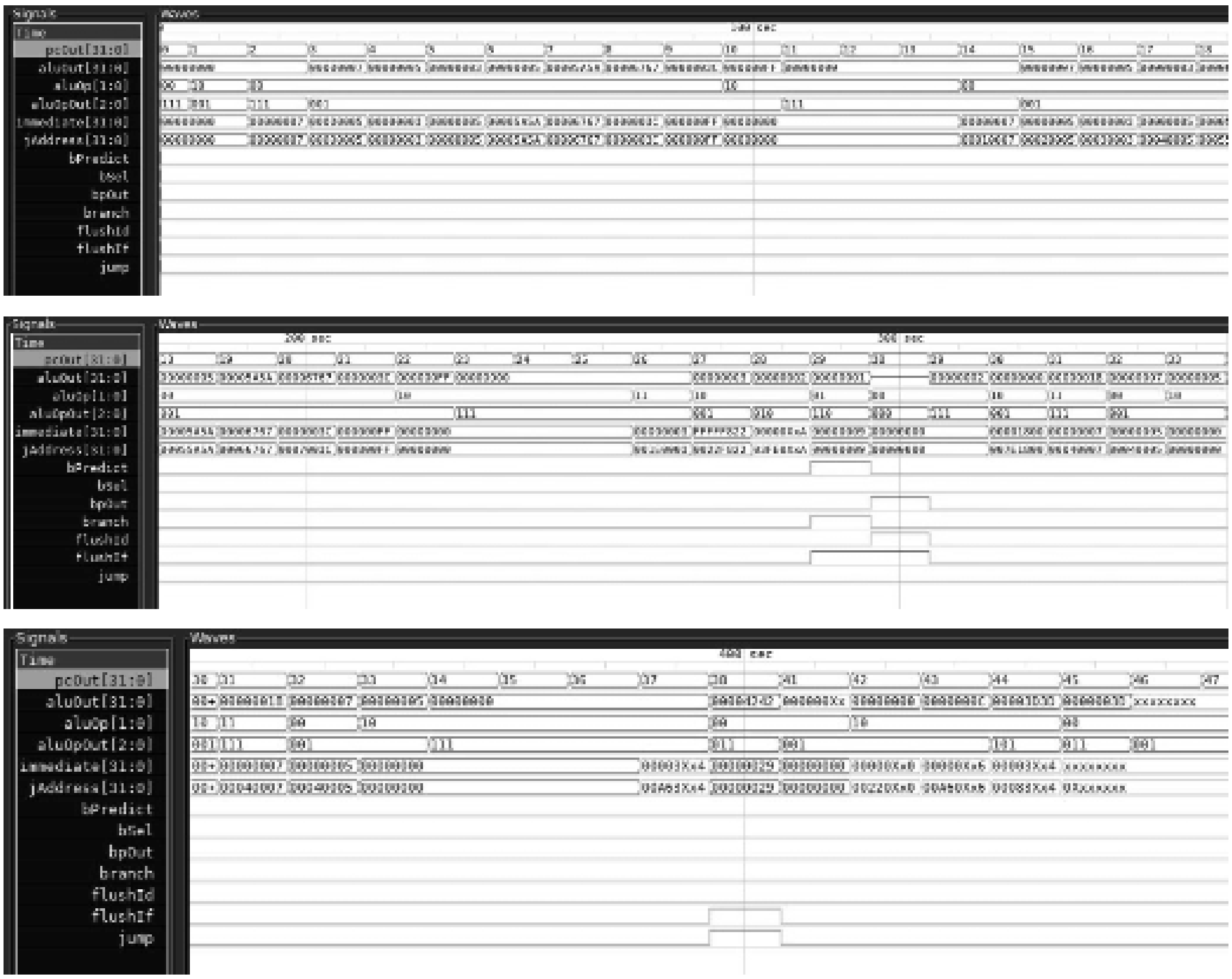


Figure 14. GTKWave of First Executed C Program

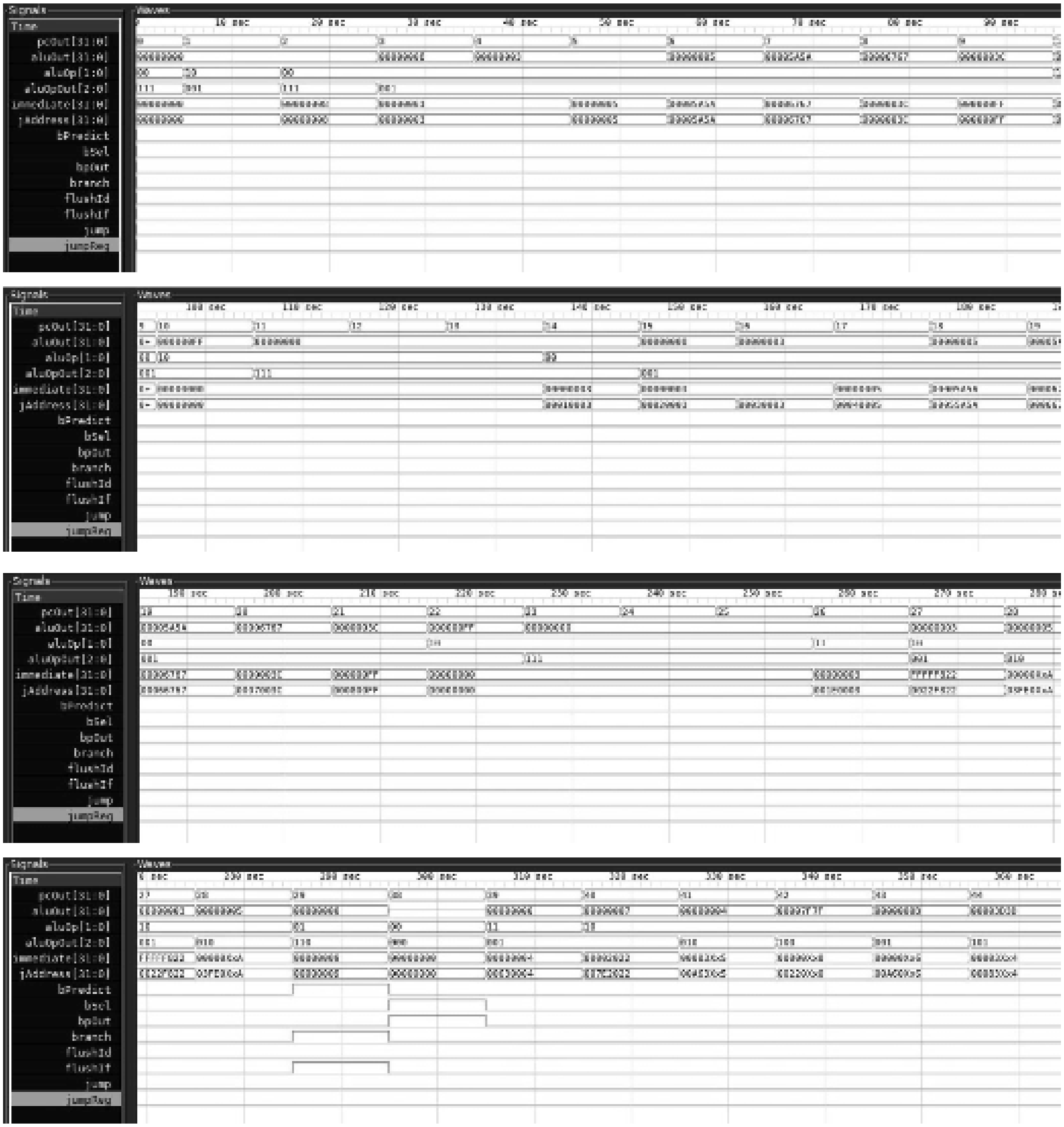


Figure 15. GTKWave of Second Executed C Program

Signal Tap Outputs:

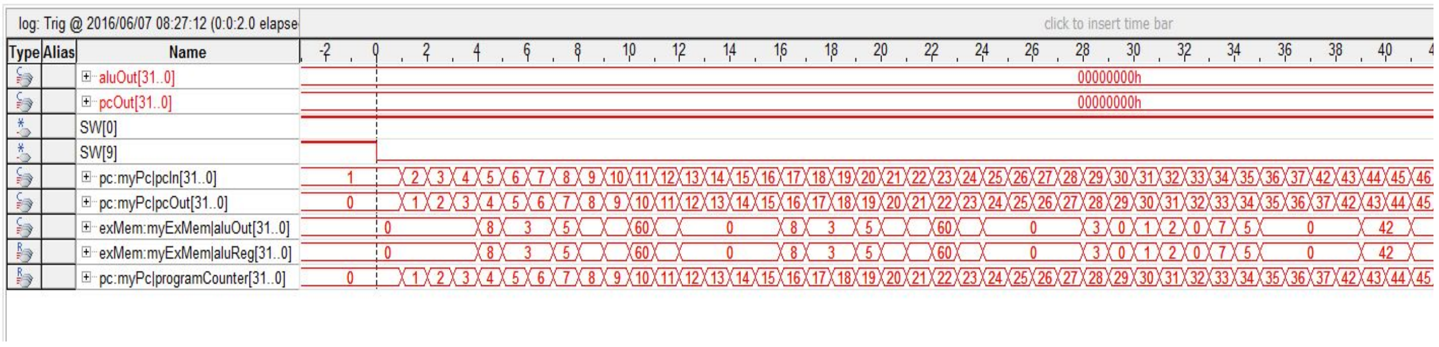

log: Trig @ 2016/06/07 08:27:12 (0:0:2.0 elapse
click to insert time bar
Type
Alias
Name
aluOut[31..0]
pcOut[31..0]
SW[0]
SW[9]
pc:myPc|pcIn[31..0]
pc:myPc|pcOut[31..0]
exMem:myExMem|aluOut[31..0]
exMem:myExMem|aluReg[31..0]
pc:myPc|programCounter[31..0]
00000000h
00000000h

Figure 16. Signal Tap of First Executed C Program

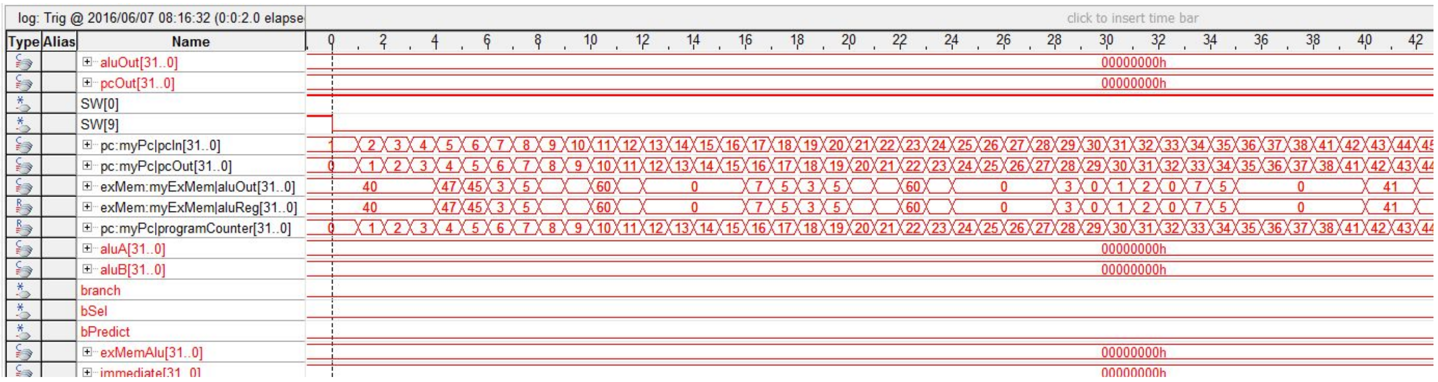


Figure 17. Signal Tap of Second Executed C Program