John Careaga
EC413

Final Project: Multi Cycle CPU

Finalized Design (MS1)

Overview

The Following 32-bit Multi-Cycle CPU is partitioned between a controller and a datapath that work together to execute instructions correctly and in order. The datapath carries out instructions over multiple cycles according to the controller's output signals. The overall design is shown in the diagram below:

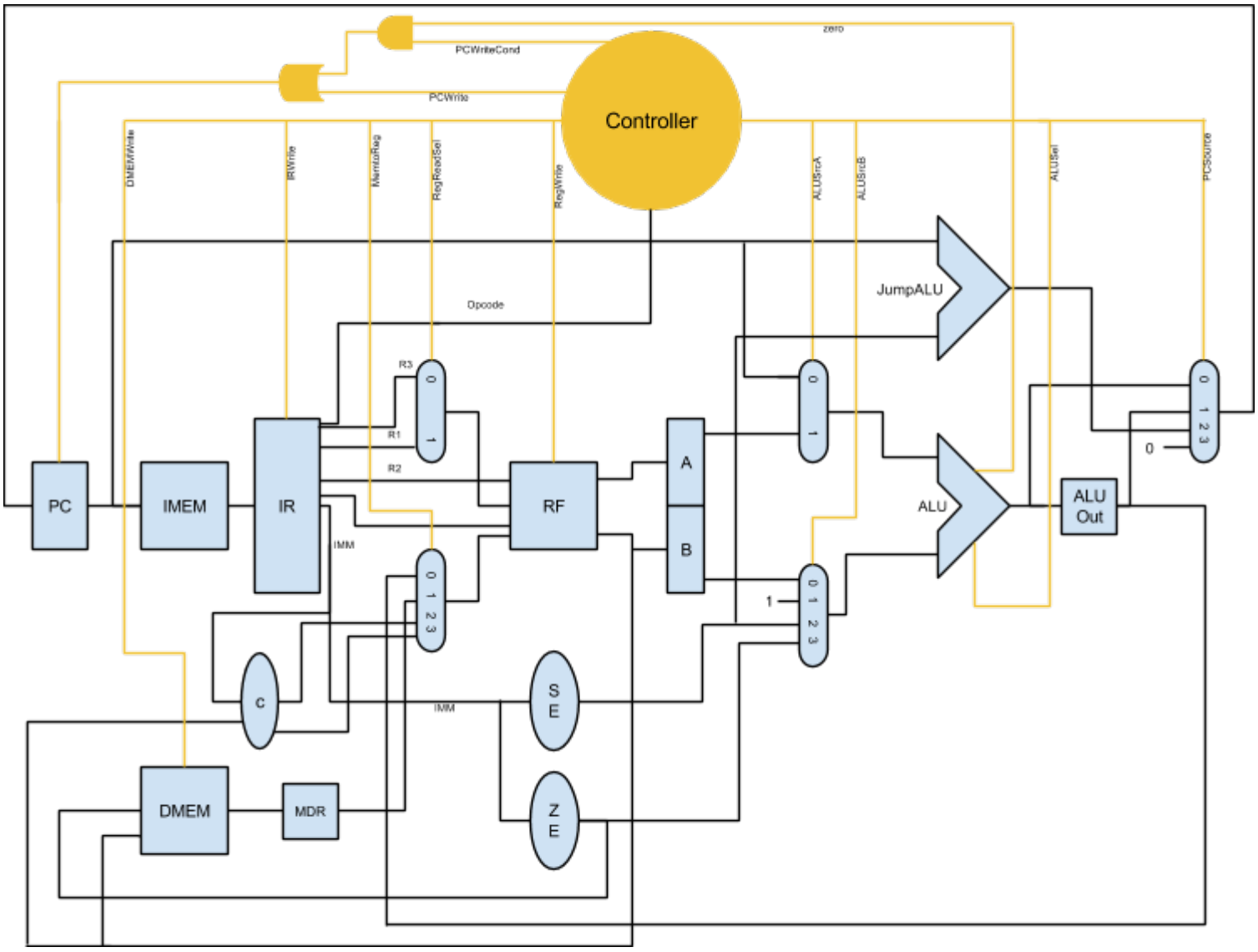


Fig. 1. Multi-Cylce CPU Design (I made this!)

The datapath components correspond to the blue objects and the black signal lines, while the control components corresponds to the orange objects and orange signal lines. These two main components are combined together to form the overall CPU.

The CPU contains the following:

- PC - Program Counter Register
- IMEM/DMEM - Instruction Memory / Data Memory Registers
- IR - Instruction Register
- MDR - Memory Data Register
- RF - Register File
- A, B - Register File Output Registers
- ALUOut - ALU output register
- ALU (main)
- JumpALU (for calculating Jump Target)
- SE - Immediate Sign Extender
- ZE - Immediate Zero Extender
- C - Outputs two concatenations: {ReadData2[31:16], imm} and {imm, ReadData2[15:0]}. This is not an actual module in the verilog datapath but the diagram illustrates what the verilog code achieves.
- Various Multiplexers to select from inputs
- Controller

Not shown is a Synchronous Reset input for the whole CPU. The reset clears the contents of the IR, MDR, A & B, and ALUOut registers. The IMEM/DMEM is not altered since it holds stored data. The reset sets the PC to the 0th instruction and sets the controller to the reset state.

Instruction Classification

Instructions are 32-bit words that contain encoded information to guide the controller's outputs and hence the datapath operations. The 2 most significant bits of the 6-bit OPcode determine the instruction type:

- 00 Jump
  - hardware: Muxes, JumpALU
- 01 Arithmetic/Logical R□type
  - hardware: RF, A, B, ALU, ALUOut, Muxes
- 10 Branch (I□type)
  - hardware: RF, A, B, ALU, ALUOut, Muxes
- 11 Arithmetic/Logical I□type
  - hardware: RF, A, ALU, ALUOut, Muxes, possibly DMEM & MDR,

All instructions require instruction fetches and instruction decodes. The hardware used in these steps include the PC, the IMEM, and the IR.

The Complete List of Instructions is shown below:

Complete List of Instructions

Op3	Op2	Op1	Op0	Op1	Op0	Output	Function Name
0	0	0	0	0	0		NOOP
0	1	0	0	0	0	R1 = R2	MOV
0	1	0	0	0	1	R1 = ~R2	NOT
0	1	0	0	1	0	R1 = R2 + R3	ADD
0	1	0	0	1	1	R1 = R2 - R3	SUB
0	1	0	1	0	0	R1 = R2 | R3	OR
0	1	0	1	0	1	R1 = R2 & R3	AND
0	1	0	1	1	0	R1 = R2 ^ R3	XOR
0	1	0	1	1	1	R1 = 1 if R2 < R3, else 0	SLT
0	0	0	0	0	1	PC ← PC[31:26] || Limm	J
1	0	0	0	0	0	IF (R1 == R2) THEN PC ← PC + SE(Imm) ELSE PC ← PC + 1	BEQ
1	1	0	0	1	0	R1 = R2 + SE(Imm)	ADDI
1	1	0	0	1	1	R1 = R2 – SE(Imm)	SUBI
1	1	0	1	0	0	R1 = R2 | ZE(Imm)	ORI
1	1	0	1	0	1	R1 = R2 & ZE(Imm)	ANDI
1	1	0	1	1	0	R1 = R2 ^ ZE(Imm)	XORI
1	1	0	1	1	1	R1 = 1 if R2 < SE(Imm), else 0	SLTI
1	1	1	0	0	1	R1[15:0] ← ZE(Imm)	LI
1	1	1	0	1	0	R1[31:16] ← ZE(Imm)	LUI
1	1	1	0	1	1	R1 ← M[ZE(Imm)]	LWI
1	1	1	1	0	0	M[ZE(Imm)] ← R1	SWI

Notes:

- Unless otherwise noted, all instructions advance the PC by 1.
- SE: Sign Extended.
- ZE: Zero Extended.

Figure 2. Instruction List

Finalized Controller (MS2)

Background

The controller is vital as it takes the role of the director: specifying which data to use, how to transform it, and what to do with it. For the Multi-Cycle CPU, each instruction has its own “path” of operation sequences. All instructions have an Instruction Fetch and an Instruction

decode cycle. From there, some instructions may then have an execution cycle, a memory access cycle, a write-back cycle, or a combination of these. The controller is thus a Moore Finite State Machine that inputs the instruction opcode from the Instruction Register and the current state, and outputs corresponding control signals.

State Diagram

Below is the controller state diagram. There are a total of 13 states shown below. Additionally, there is a Reset State sR that becomes the new state whenever the reset signal is high at a positive clock edge regardless of the current state. The Reset State is described in table 1. The Multi-Cycle CPU begins program execution at state zero - Instruction Fetch.

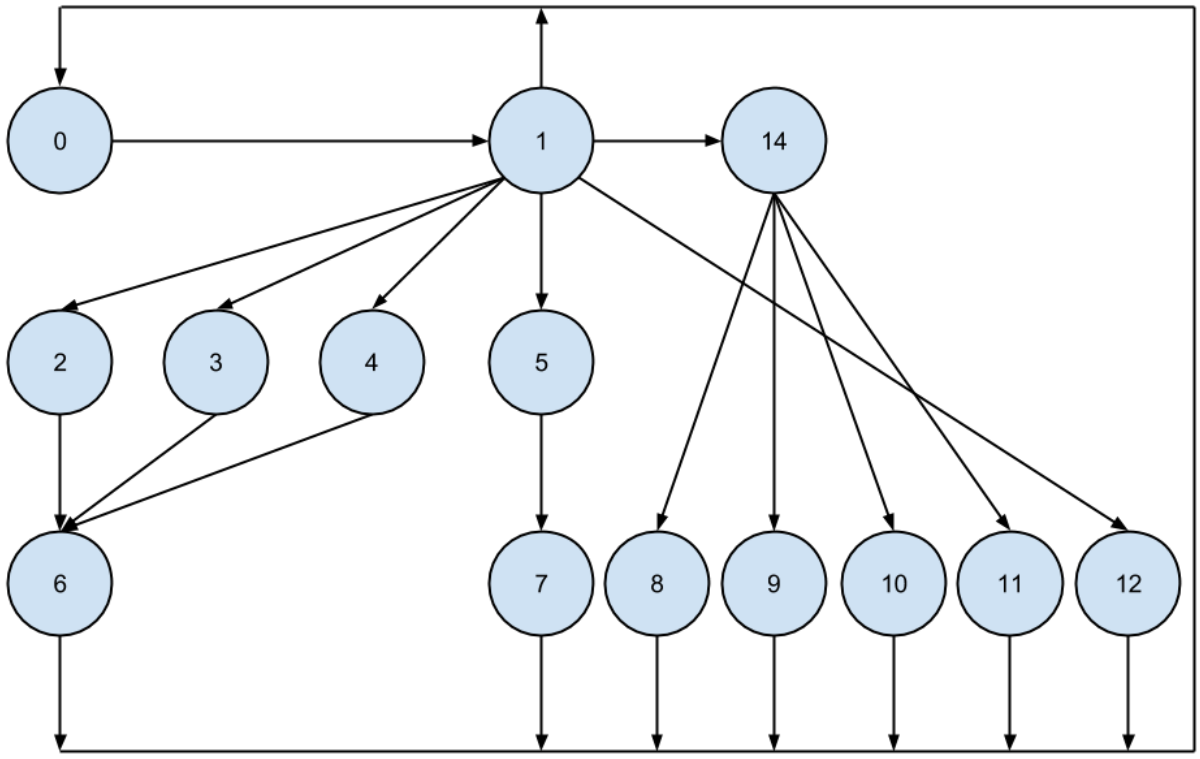


Fig. 3. State Diagram

State Description

Below is the description of each state from the Diagram above (and the sR state).

State	Description	Operation
0	IF	IR = IMEM[PC] PC = PC + 1
1	ID (and speculative Branch Target calculation)	A = $R2 = reg[IR[20-16]]

		B = $R3 = reg[IR[15-11]] ALUOut = PC + SE(IR[15-0]) NOP detection (returns to state zero)
2	R-type EX (ALU needed)	ALUOut = A op B
3	I-type sign-extended imm. EX (ALU needed)	ALUOut = A op SE(IR[15-0])
4	I-type zero-extended imm. EX (ALU needed)	ALUOut = A op ZE(IR[15-0])
5	LWI Mem Read	MDR = DMEM[ZE[IR[15-0]]]
6	Register Write-Back where ALUOut is used	R1 <- ALUOut
7	Register Write-Back where MDR is used	R1 <- MDR output
8	SWI Mem Write	DMEM[ZE[IR[15-0]]] <- $R1
9	Register Write-Back for LI	R1[15-0] = IR[15-0]
10	Register Write-Back for LUI	R1[31-16] = IR[15-0]
11	Branch Completion	if (A == B) then PC = ALUOut
12	Jump Completion	PC = PC[31-26] || IR[15-0]
sR (13)	Reset State	PC = 0 IR = 0 A = 0 B = 0 ALUOut = 0
14	$R1 necessary	Register File ReadData2 outputs $R1 instead of $R3

Table 1. State Description

Control Signal Description

PCWrite	**PC Writing Enabled/Disabled**
00	Writing Disabled
01	Writing Enabled

Table 2. PCWrite

PCWriteCond	Set if Branch instruction decoded
00	Disabled
01	Enabled

Table 3. PCWriteCond

DMEMWrite	DMEM Writing Enabled/Disabled
00	Writing Disabled
01	Writing Enabled

Table 4 DMEMWrite

IRWrite	IR Writing Enabled/Disabled
00	Writing Disabled
01	Writing Enabled

Table 5. IRWrite

MemtoReg	Register File Write Data input
00	ALUOut
01	MDR output data
10	write-data[Z immediate]
11	write-data[immediate Z]

Table 6. MemtoReg

PCSource	PC input
00	PC + 1
01	ALUOut (for branches)
10	Jump Target
11	hardwired 0 (for reset)

Table 7. PCSource

ALUSel	Main ALU operation select
0000	out = sourceA
0001	out = ~sourceA
0010	out = sourceA + sourceB
0011	out = sourceA - sourceB
0100	out = source A | sourceB
0101	out = sourceA & sourceB
0110	out = sourceA ^ sourceB
0111	out = 1 if sourceA < sourceB, else out = 0

Table 8. ALUSel

ALUSrcA	1st ALU input
00	PC
01	A

Table 9. ALUSrcA

ALUSrcB	2nd ALU input
00	B
01	hardwire 1 (for PC increment)
10	SE(immediate)
11	ZE(immediate)

Table 10. ALUSrcB

RegWrite	Register File Writing Enabled/Disabled
00	Writing Disabled
01	Writing Enabled

Table 11. RegWrite

RegReadSel	Select which register ReadData2 reads
00	R3
01	R1

Table 12. RegReadSel

Control Signal Specification

Below is the controller's output signals for each state. The control signal are chosen such that proper operation is achieved.

CTRL STATE

	0	1	2	3	4	5	6	7	8	9	10	11	12	sR	14
PCWrite	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0
PCWriteCond	0	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	1	dc	0	dc
DMEMWrite	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
MemtoReg	dc	dc	dc	dc	dc	dc	0	1	dc	2	3	dc	dc	dc	dc
PCSource	0	dc	dc	dc	dc	dc	dc	dc	dc	dc	dc	1	2	0	dc
ALUSel	2	2	op	op	op	dc	dc	dc	dc	dc	dc	3	dc	0	2
ALUSrcA	0	0	1	1	1	dc	dc	dc	dc	dc	dc	1	dc	0	0
ALUSrcB	1	2	0	2	3	dc	dc	dc	dc	dc	dc	0	dc	0	2
RegWrite	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0
RegReadSel	dc	0	dc	dc	dc	dc	dc	dc	dc	dc	dc	1	dc	dc	1

Table 13. Control signals for each state

Controller Verilog Implementation

The controller is a Moore Finite State Machine that inputs a clock signal, a reset signal, and a 6-bit opcode from the IR. The module outputs 11 control signals of varying width (see tables 2-12). The control signals are connected to the datapath control inputs. The 6-bit opcode input changes each time a new instruction is fetched and written to the IR. At every positive clock edge, the controller first checks to see if the reset signal is high. If it detects that it is, it unconditionally goes to the Reset State. In this state, the PC will be written to zero and the state will return to IF at the next clock cycle (provided reset signal returns to low).

Once the reset signal is low, the controller checks what the current state is. Based on this finding and the opcode input, the controller determines what the control signals need to be updated to and what state becomes the new current state. The behavior is coded to follow the state diagram in figure 3 and the state-dependent control signals specified in table 13.

The module code uses an always statement to update the controller every clock cycle. Within the always statement, An if-else statement is used to check for the reset signal. Inside the else block, is a case statement that looks for the current state. Once the current state is found, the state is updated according to the opcode input and the current state. The control signals for the new state are also driven.

The controller module uses many parameters to simplify the code and make it more readable. For example, a specific opcode input is labeled J to signify a jump instruction.

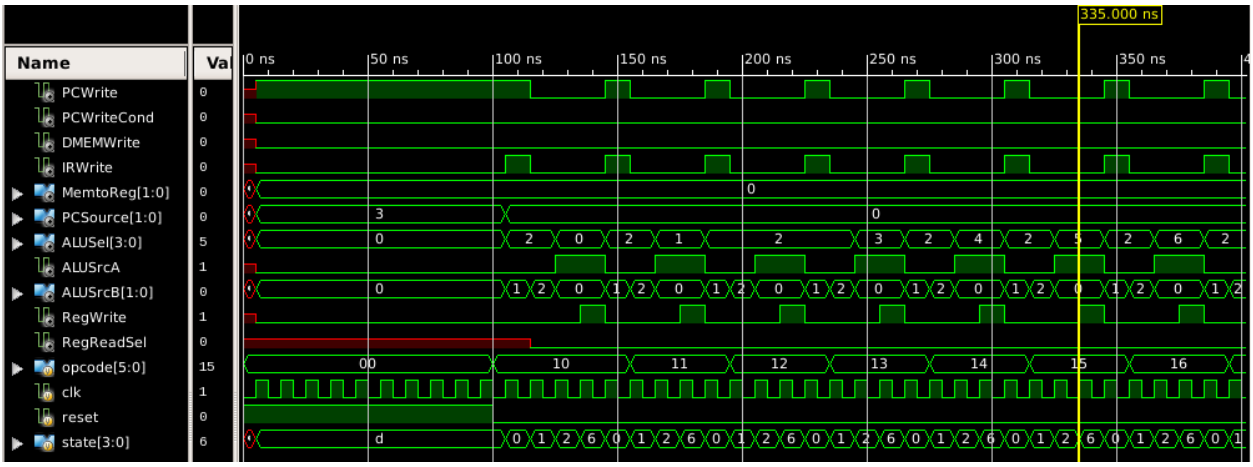
The Verilog Module with detailed comments is included in the Project Files.

Controller Testing - Verilog Test Fixture

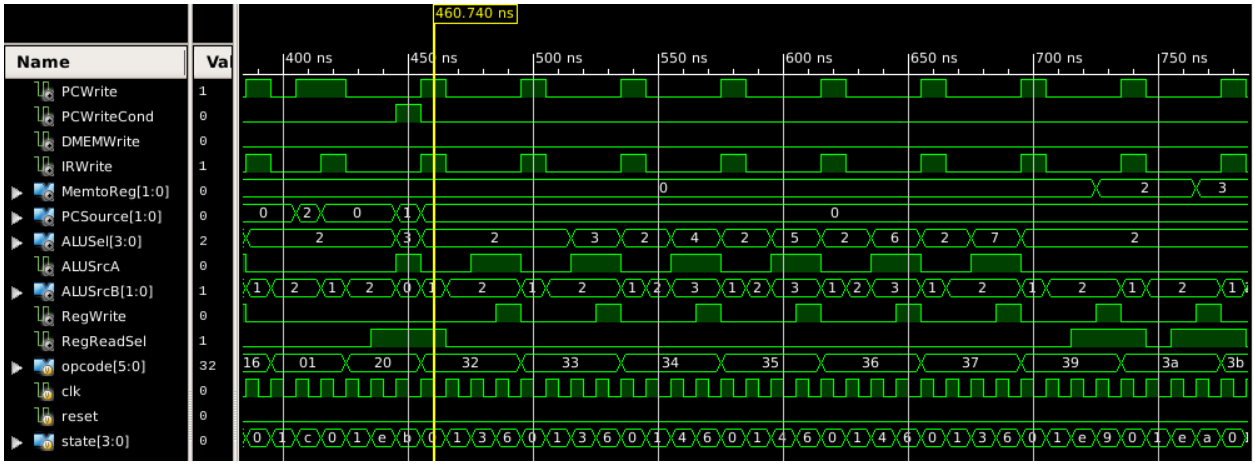
The test fixture begins with a reset and then begins with an instruction fetch with a specified opcode input. The opcode is updated to correspond to a new, different instruction after the current instruction has completed its state sequence. This simulates new IR values being written to at the end of each Instruction Fetch.

Below are the waveforms of the test results. We see that each instruction traversed its corresponding state sequence correctly. When there was a reset signal set, the controller went to the Reset State unconditionally, and then went back to state zero after the reset signal dropped. Additionally, note that the control signals are appropriately updated depending on the state of the controller. The controller uses the current state and the opcode to determine the next state, and uses the current state to determine the control signal outputs.

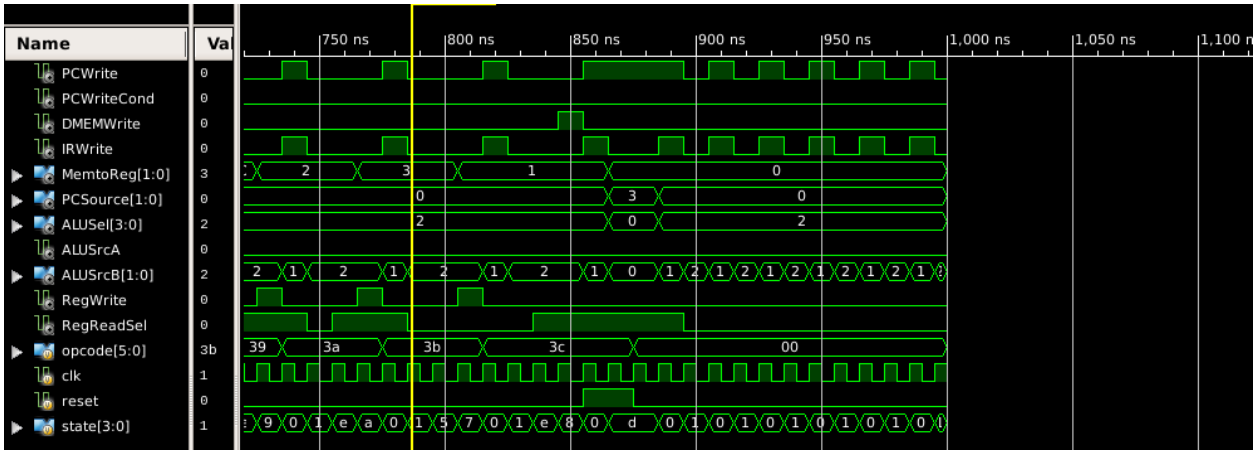
The Verilog Test Fixture is included in the Project Files with detailed comments.



R-type instructions



Jump instruction, BEQ instruction, I-type Arithmetic instructions, LI/LUI



LWI, SWI, reset, NOP

Finalized Datapath (MS3)

Datapath Implementation

The CPU's datapath follows exactly from the design diagram shown in figure 1. The verilog datapath module inputs the clock and reset signals, as well as the 11 control signals. The module outputs just the 6-bit opcode from the IR.

The datapath instantiates all the submodules shown in figure 1. These submodules are connected together in the datapath module using wires.

The submodules include:

- holding_reg
  - Registers for the PC, IR, A, B, and ALUOut.
  - Simple 32-bit register that outputs the contents combinationally and writes to the contents from an input sequentially. Includes write enable and reset inputs.
  - high reset clears the contents of the register.

    - Inputs: input_data, write, clk, reset
    - outputs: output_data
- zero_extend
    - Simple combinational module that inputs a 16-bit immediate and extends it to 32 bits using zeros.
    - inputs: input_data
    - outputs: output_data
- sign_extend
    - Simple combinational module that inputs a 16-bit immediate and extends it to 32 bits using the sign of the immediate.
    - inputs: input_data
    - outputs: output data
- read_mux
    - Combinational multiplexer specifically used to select the Register File's ReadSelect2 input.
    - Outputs one 5-bit word. The output is chosen from two 5-bit inputs (R1, R3).
    - inputs: input_data0, input_data1, select
    - outputs: output_data
- mux_2bit
    - Combinational multiplexer used when there are 4 inputs to choose from (ie, PCSource).
    - Outputs one 32-bit word. Output is chosen from four 32-bit inputs.
    - inputs: input_data0, input_data1, input_data2, input_data3, select
- mux_1bit
    - Same as mux_2bit, but there are only two data inputs to choose from instead of 4.
    - Used only for determining ALUsrcA
- jumpALU
    - Combinational ALU used for Jump instructions
    - Adds the input PC (PC+1) and the offset (the sign extended immediate from the IR) to get the jump target.
    - inputs: inputPC, offset
    - outputs: jump_target
- myALU
    - Main combinational ALU responsible for arithmetic and logical operations.
    - Control signal input ALUSel specifies desired operation
    - Includes a 1-bit "zero" output that is high when subtractions yield zero. This bit is used for BEQ instructions to indicate whether the contents of two registers are equal.
    - inputs: sourceA, sourceB, ALUSel
    - outputs: output_data, zero
- IMem
    - Memory used for holding programs. Inputs the PC and outputs a corresponding instruction.

    - inputs: PC
    - outputs: Instruction
- DMem
    - Memory used for holding data, as in SWI.
    - Combinational read, sequential write.
    - Write Enable control signal included
    - inputs: WriteData, Address, MemWrite, Clk
    - outputs: MemData
- nbit_register_file
    - Register File
    - 32 32-bit registers
    - Combinational read, sequential write.
    - inputs: read_sel_1, read_sel_2, write_address, write_data, RegWrite, clk
    - output: read_data_1, read_data_2

Assignment statements specify the fields from the IR that are routed to other places in the datapath. For example, bits 15-0 correspond to the immediate field of the instruction, and get routed to the sign and zero extenders, etc.

The PC's write enable input is derived from the OR and AND gates. This structural component is written in the datapath module, although it is shown as part of the control in figure 1.

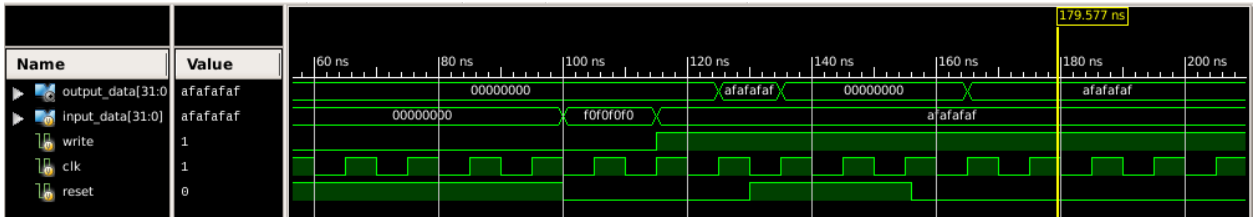
The Verilog Modules with detailed comments are included in the Project Files.

Datapath Component Testing - Verilog Test Fixture

Testing results for selected modules are shown here. Note that the Mux_2bit module follows the Mux_1bit, so testing both is redundant. Similarly, the zero extend logic is straightforward. The main ALU is similar to the ALU used in previous labs, which have already been tested, so additional testing is unneeded.

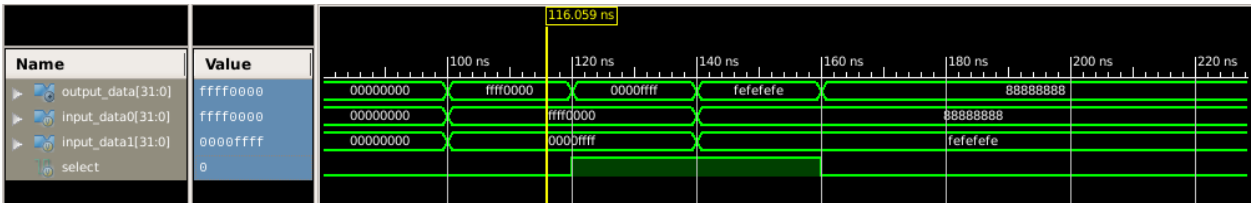
The test bench verilog files are included in Project Files.

Holding Register



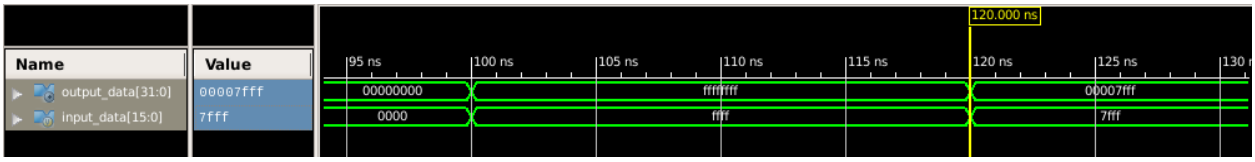
Sequential write only when write enable signal is high. Reset clears contents.

Mux_1bit



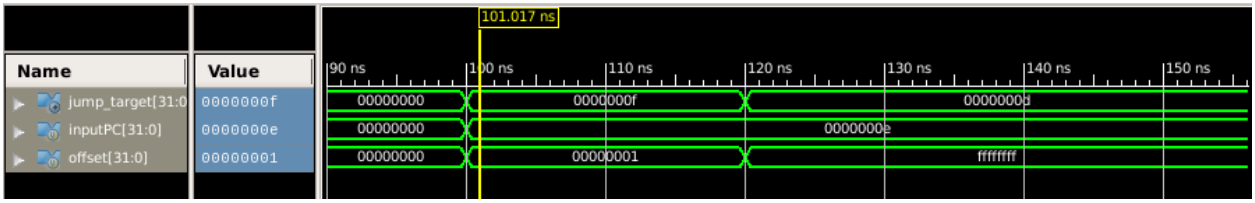
Mux output depends on the 1-bit select value

Sign Extend



The 16-bit input is sign extended for both negative and positive numbers

jumpALU



The jumpALU adds signed offsets to the PC (PC+1)

Finalized CPU (MS4)

Multi-Cycle CPU Implementation

The Multi-Cycle CPU implementation is very straightforward. The datapath and controller are simply connected together to form the overall CPU; all the hard work is done in the controller and datapath modules.

The verilog CPU module inputs only the clock and reset signals. There are no outputs. Wires represent the 11 control signals and the opcode.The datapath and controller modules are instantiated. The 11 control signals correspond to the outputs of the controller and to the inputs of the datapath. The opcode corresponds to the output of the datapath and to the input of the controller. Both modules input the clock and reset signals. The datapath and controller work seamlessly together to fetch and process instructions in memory. After the program is finished, default NOPs are continuously executed by the CPU. If a reset signal is set, the PC rolls back to zero and the program runs again.

The Verilog Module with detailed comments is included in the Project Files.

CPU Testing - Verilog Test Fixture

The test fixture begins with a high reset signal. After 100 ns, the reset signal changes to low and program execution begins as the state transitions from the Reset State to the IF State. No stimulus is needed while the CPU processes instructions stored in IMEM. After a selected

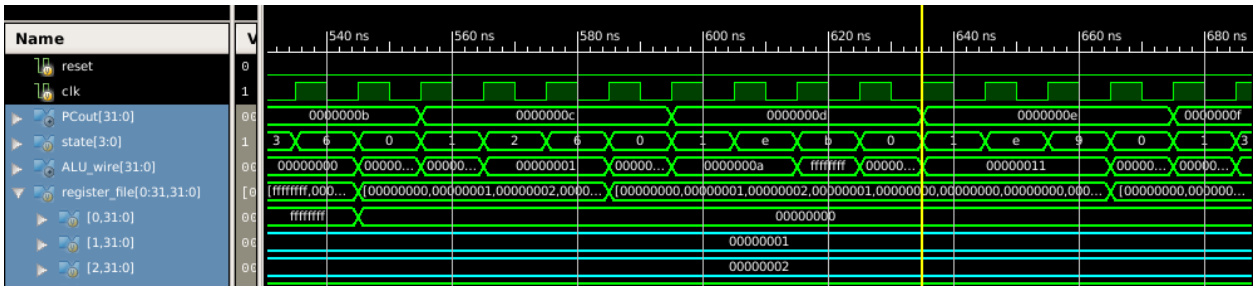
program from memory finishes execution, the reset signal is set again, and the program reruns. To see the programs stored in IMEM, open the Imem.v module. In said file, the expected results of the program can be compared to the results shown below.

The Verilog Test Fixture is included in the Project Files with detailed comments.

Below are the waveforms of the test results. For better views of the following pictures, open the screenshots provided in the Project Files.

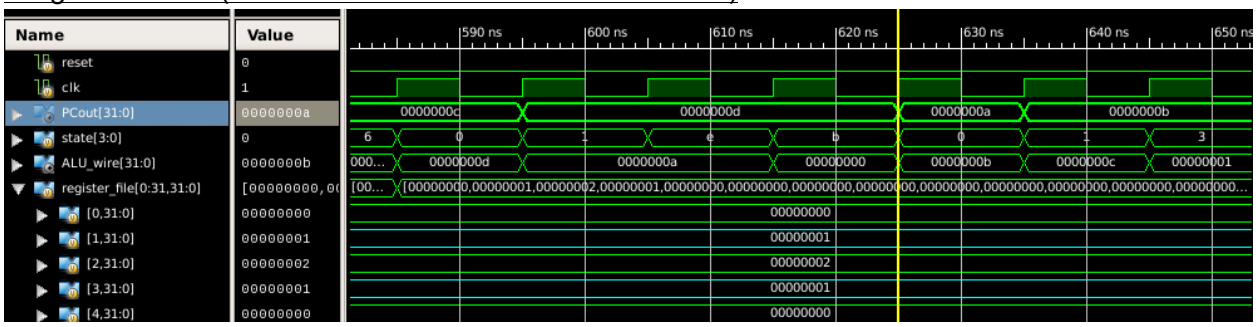
Program 1

Program 1 BEQ (no branch)



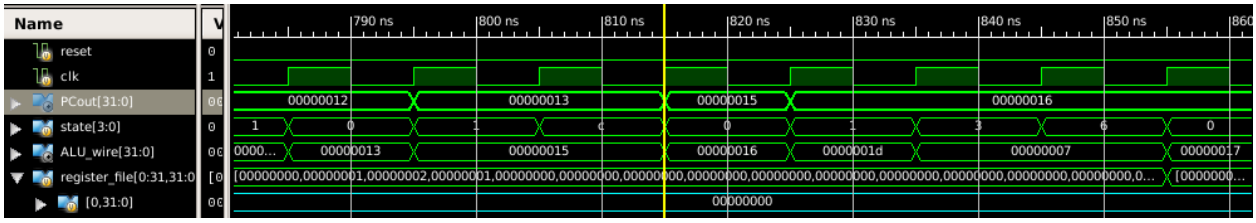
Instruction c (12) is a BEQ. The two blue signals are the register contents being compared. No branch is taken since the contents are not equal.

Program 1 BEQ (branch taken 3 instructions backwards)



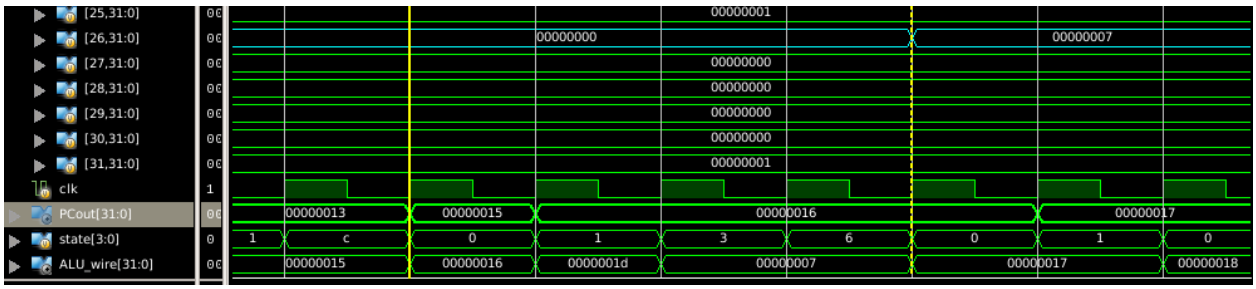
Same BEQ instruction (12) but edited so that the registers being compared contain the same value. Branch is taken. Branch target is: PC+1+(-3) = 12+1-3 = 10 = a.

Program 1 Jump taken, R0 unchanged



Unconditional jump taken. The jump instruction is 12 (18) and the offset is 2. Thus during IF, the new PC is 15 (21). The instructions that are skipped would have altered the contents of R0. We see here that they are not altered.

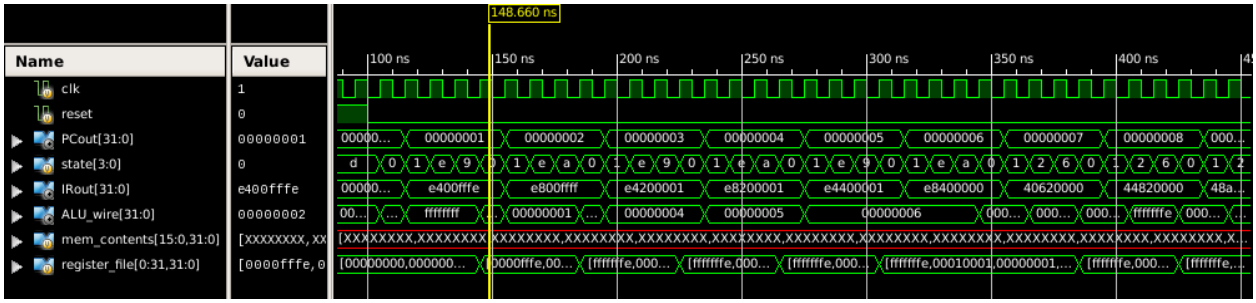
Program 1 After jump taken, R26 written to



After the jump is taken instruction 21 is processed, resulting in the update of R26 as expected.

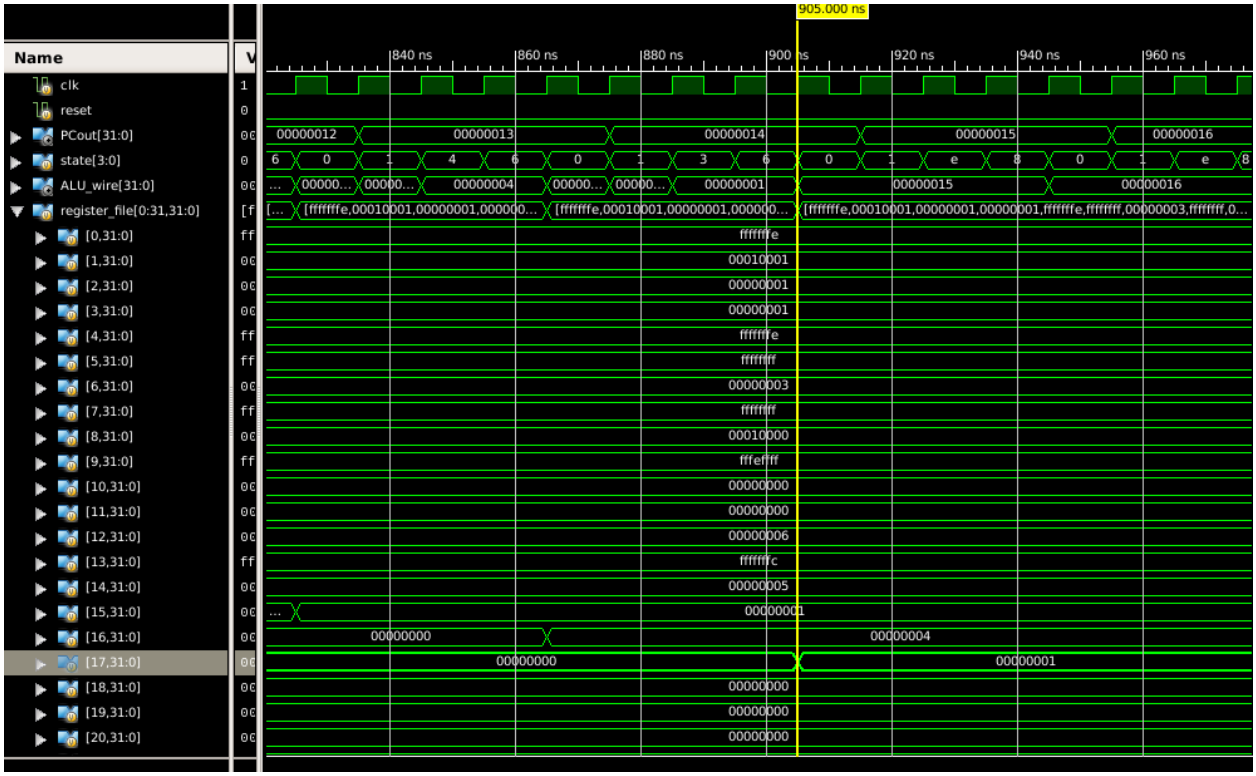
Program 2

Program 2 ALU instructions - LI, LUI, R-type, I-type Arithmetic/Logical



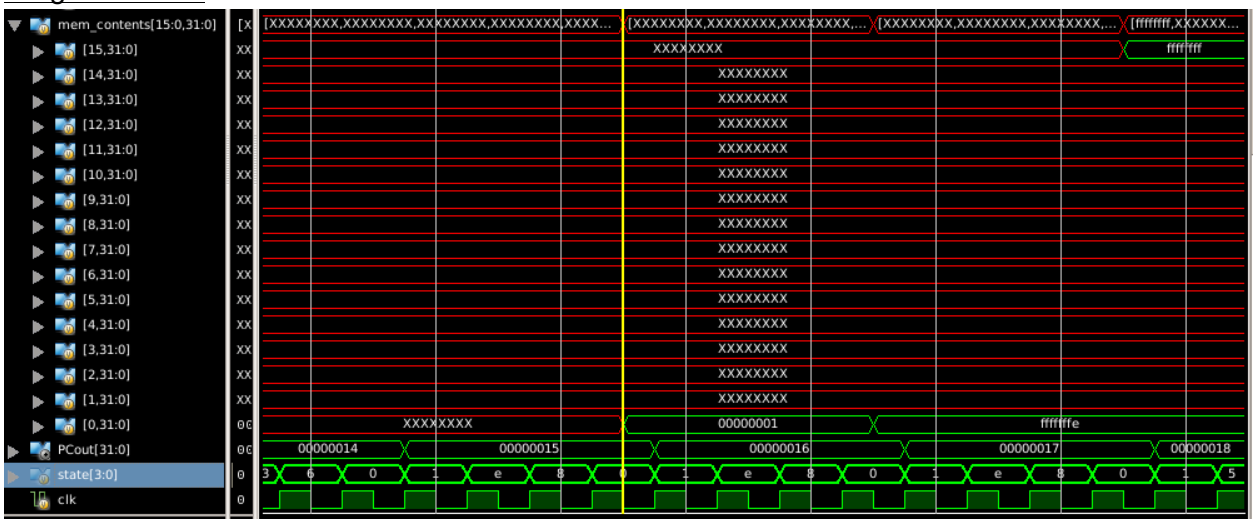
General view of PC incrementing, and the state value cycling through the state diagram depending on the instruction.

Program 2 Register File end contents - LI, LUI, R-type, I-type Arithmetic/Logical



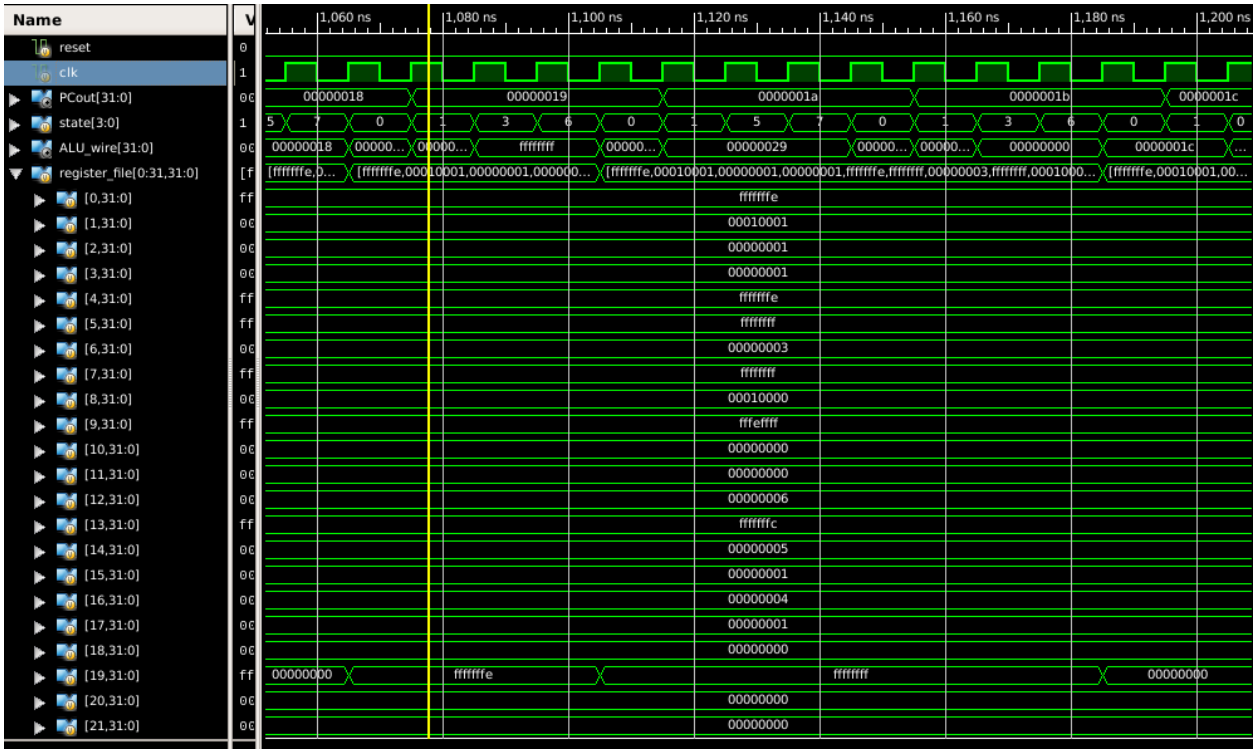
The contents of the updated register file. Comparing the values shown here with the expected results demonstrates correctness.

Program 2 SWI



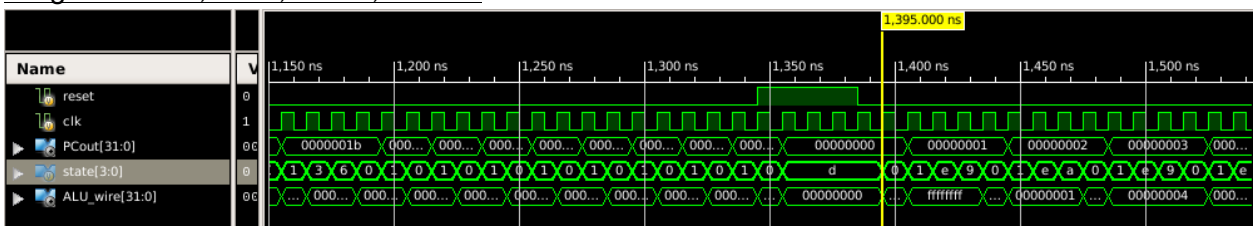
Contents from the Register File are stored into DMEM.

Program 2 LWI



LWI instructions that load contents stored in DMEM into R19.

Program 2 end, NOP, Reset, Restart



Program ends just before 1200 ns. From there no operations are executed. A reset signal send the state into the rS (d). The state remains sR until the reset signal drops. Then, the program reruns starting from PC = 0.