ECE 212 – Digital Circuits II

Lab9 – Digital Clock

Otieno Maurice(Scribe)

Alex Villalba

https://github.com/otienomaurice1/ece212_alex_maurice.git

Introduction:

In this lab we designed the controller for the multicycle processor. The controller consists of the main decoder and the Alu decoder. The main decoder produces several controls and select signals that control the registers and multiplexers of the data path. The controller takes in the OPCODE and the FUNCT fields of the instruction and uses them to determine this control and select signals. The main decoder is essentially a finite state machine in which each state corresponds to a particular stage of the instruction that is currently being computed. All instructions will be fetched and decoded using the same states. Inside the main decoder we determine the type of instruction to execute by looking at the opcode. An R-TYPE instruction would have two more states for executing and writing the result to register file. A LW instruction would have three more for computing memory address, accessing memory and writing back the result to the register file. A store word would have two more states for computing memory address and writing the data to memory. Both Jump and BRANCH would have one more state each for executing the jump and branch respectively. Each of the states would specify the output control and select signals for the data path for the stage of that instruction. One of the outputs of the main decoder is the Alu op which is a signal to determine instruction will be executed by the Alu. The Alu decoder takes in the funct field that corresponds to a given R-TYPE instruction and determines what the Alu will compute.

We described the behavior of the controller in system Verilog and wrote testbenches for each instruction.

The expected outputs for various control and data signals are shown on the table 1 below.

State	pcwrite	memwrite	irwrite	regwrite	alusrca	branch	iord	memtoreg	regdesg	alusrcb[1:0]	pcsrc[1:0]	aluop[1:0]
FETCH (0)	1	0	1	0	0	0	0	0	0	01	00	00
DECODE (1)	0	0	0	0	0	0	0	0	0	11	00	00
MEMADR (2)	0	0	0	0	1	0	0	0	0	10	00	00
MEMRD (3)	0	0	0	0	0	0	1	0	0	01	00	00
MEMWB (4)	0	0	0	1	0	0	0	1	0	01	00	00
MEMWR (5)	0	1	0	0	0	0	1	0	0	01	00	00
RTYPEEX (6)	0	0	0	0	1	0	0	0	0	00	00	10
RTYOEWB (7)	0	0	0	1	0	0	0	0	1	01	00	00
BEQEX (8)	0	0	0	0	1	1	0	0	0	00	01	01
ADDIEX (9)	0	0	0	0	1	0	0	0	0	10	00	00
ADDIWB (10)	0	0	0	1	0	0	0	0	0	01	00	00
JEX (11)	1	0	0	0	0	0	0	0	0	0	10	00

Table 1 – Main Decoder Output Table

Results:

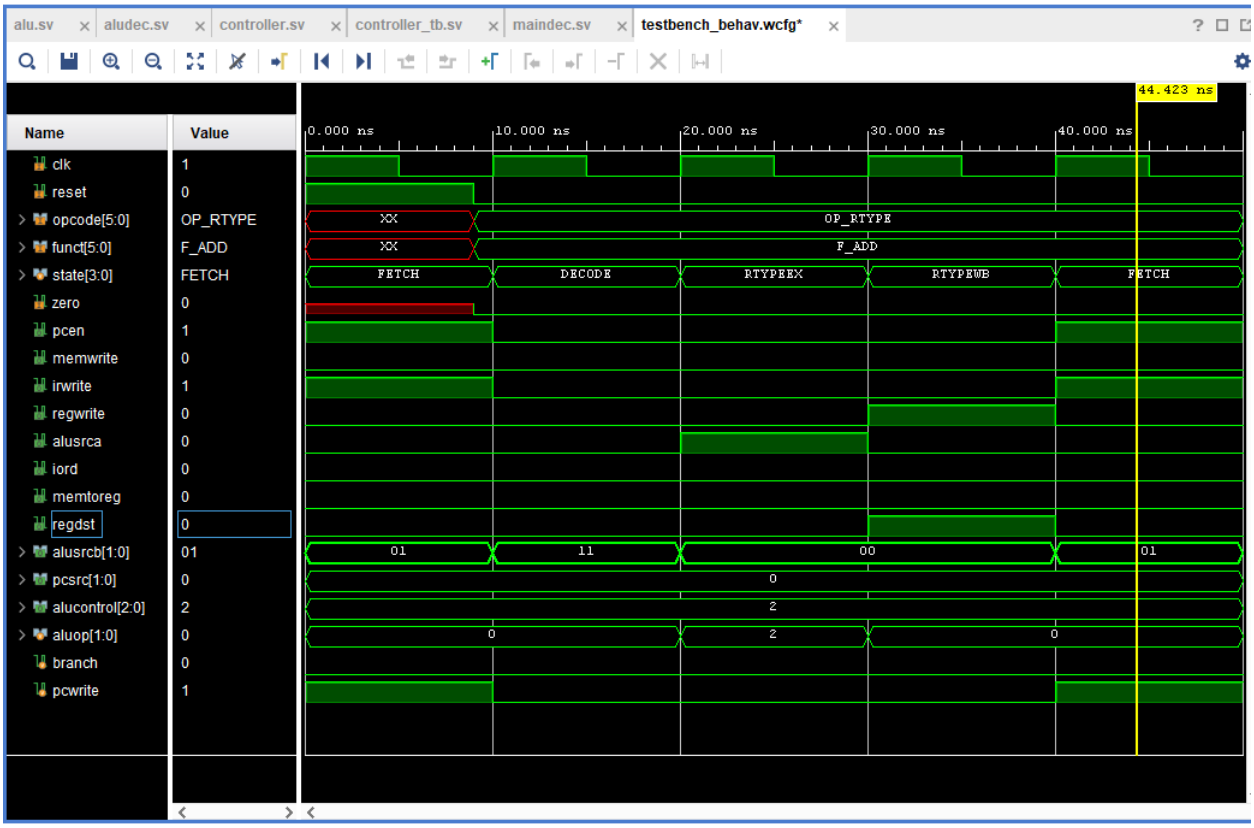

Figure 1 simulation scope for add instruction in mips_multicycle processor

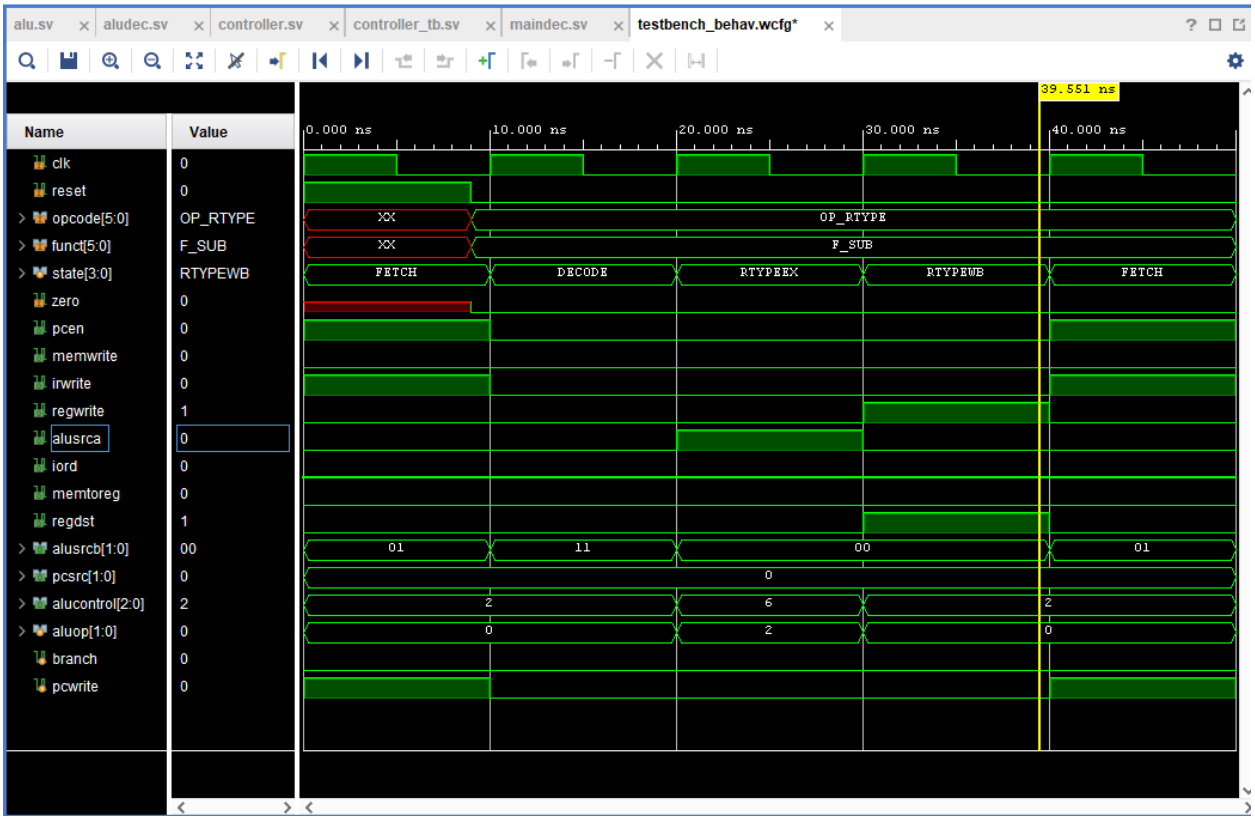

Figure 2simulation scope for SUB instruction in mips_multicycle processor

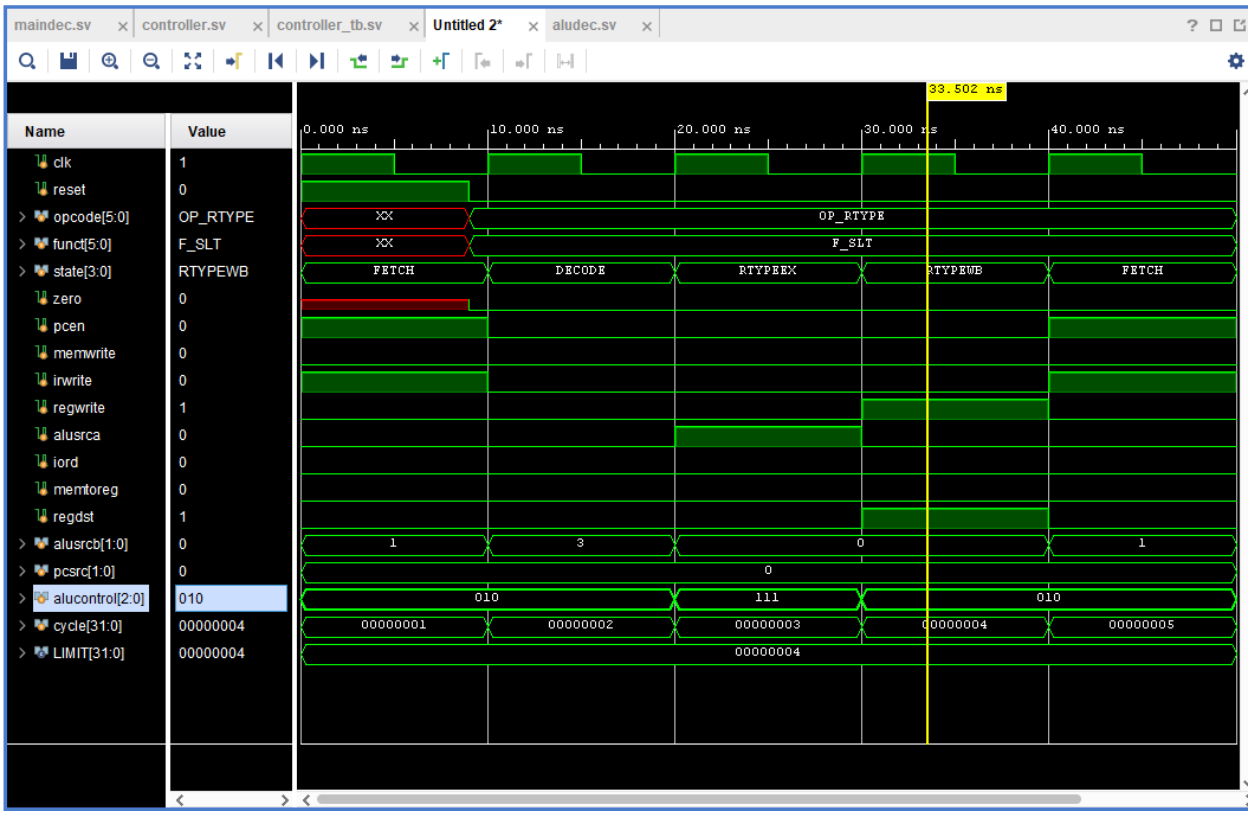

Figure 5 simulation scope for SLT instruction in mips_multicycle processor

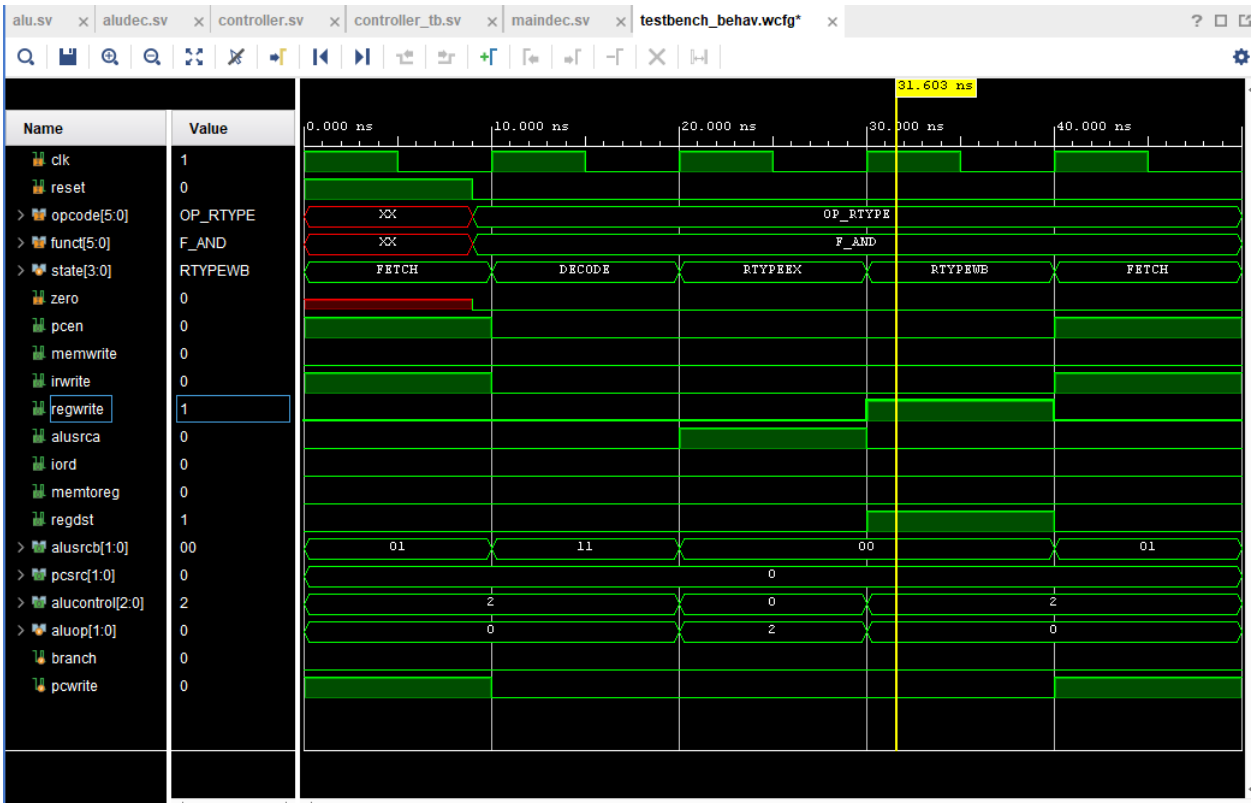

Figure 3simulation scope for AND instruction in mips_multicycle processor

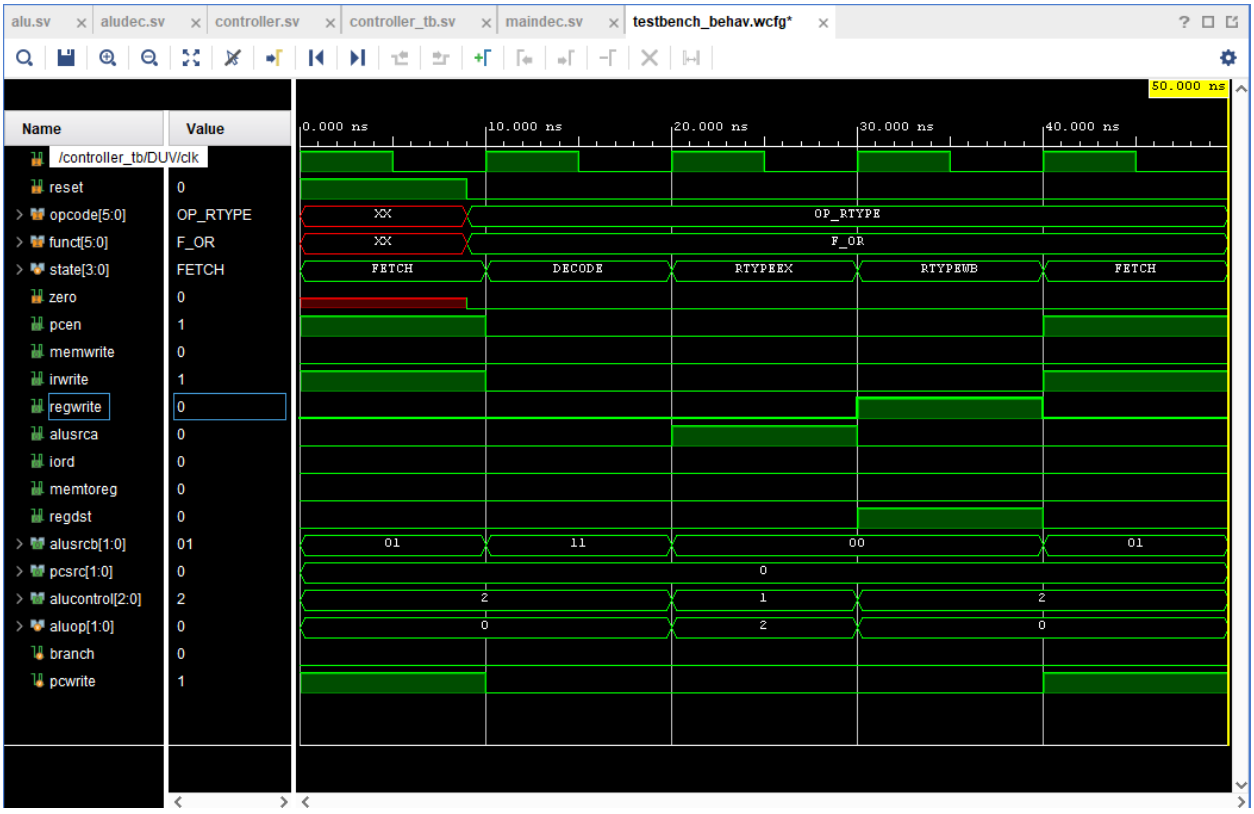

Figure 4 simulation scope for OR instruction in mips_multicycle processor

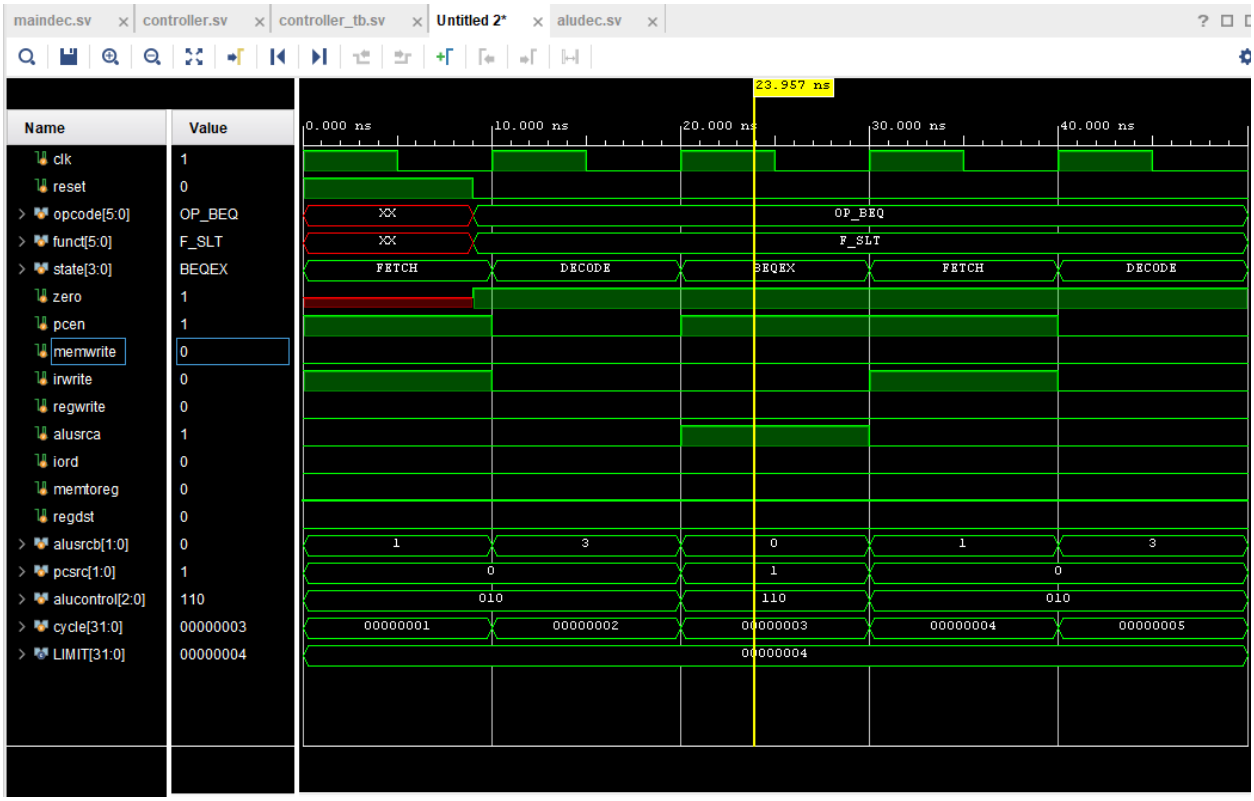

Figure 5simulation scope for BEQ instruction when branch is taken in mips_multicycle processor

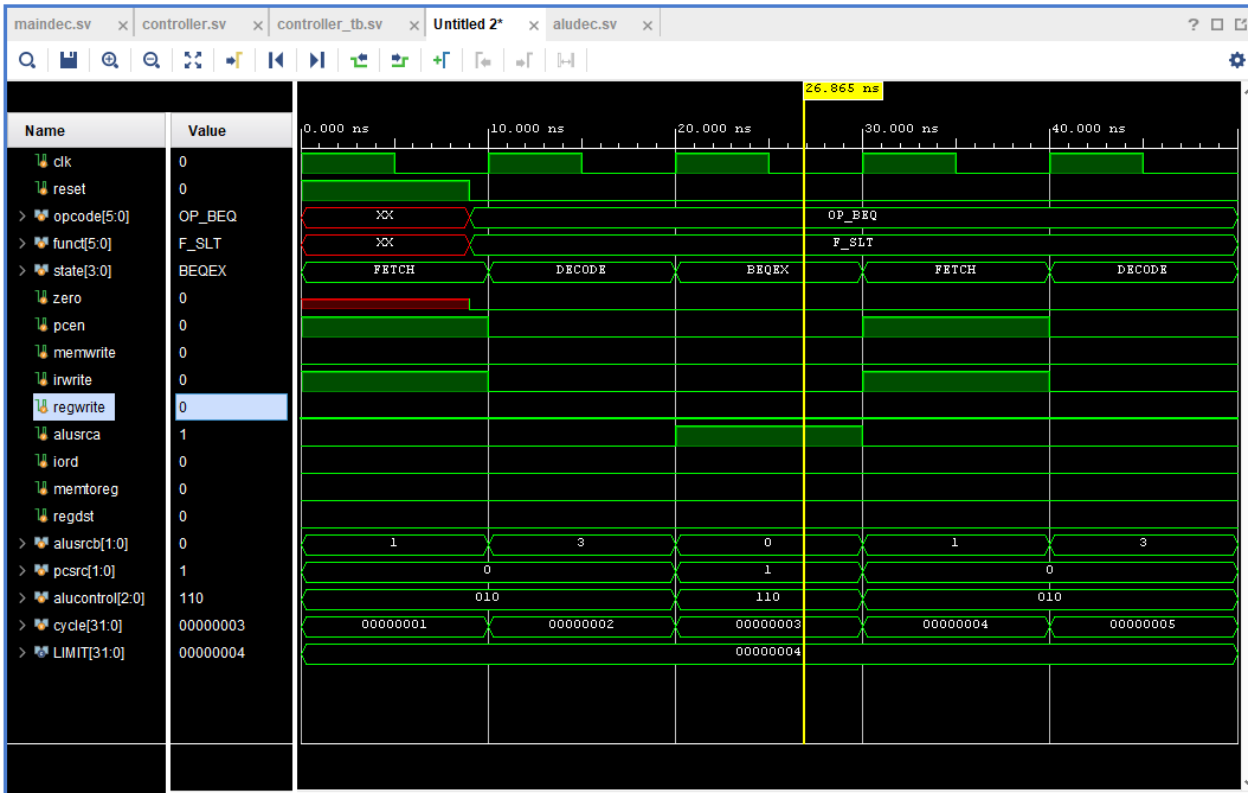

Figure 6 simulation scope for BEQ instruction when branch is not taken in mips_multicycle processor

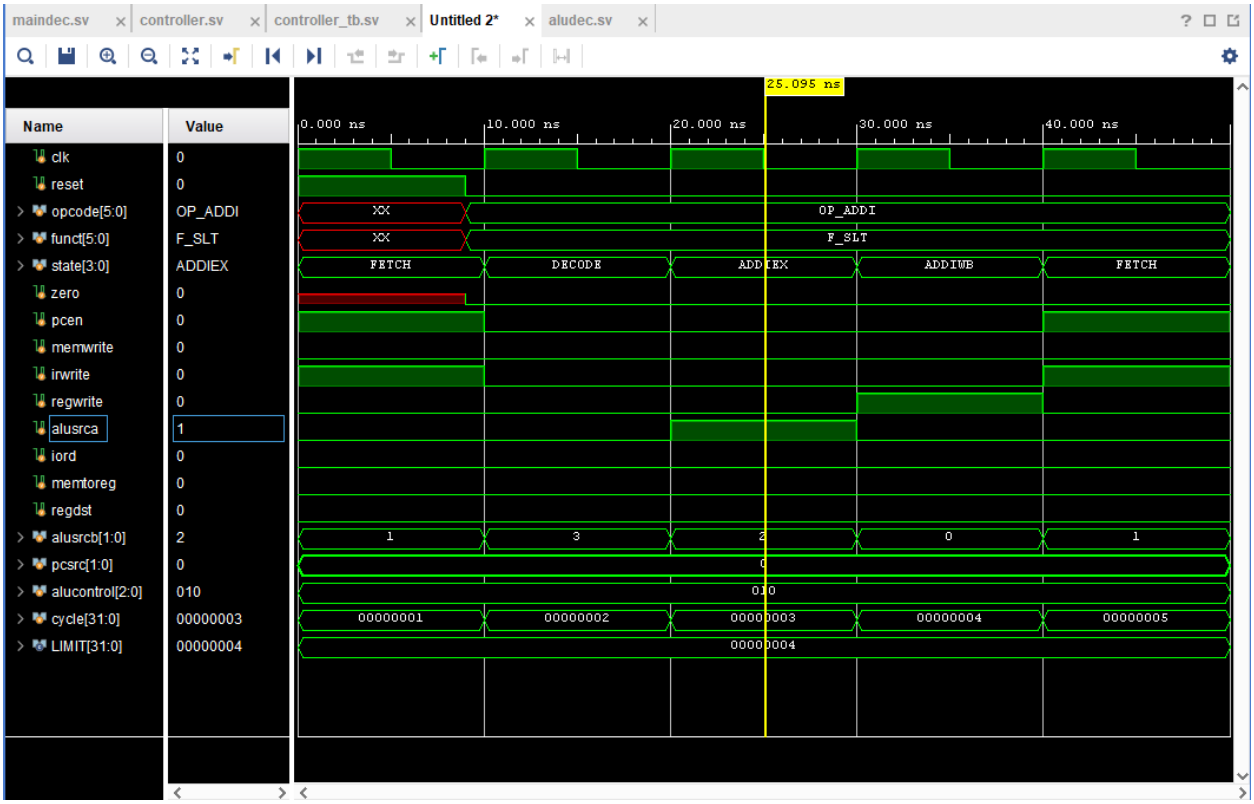

Figure 7simulation scope for the ADDI mips_multicycle processor

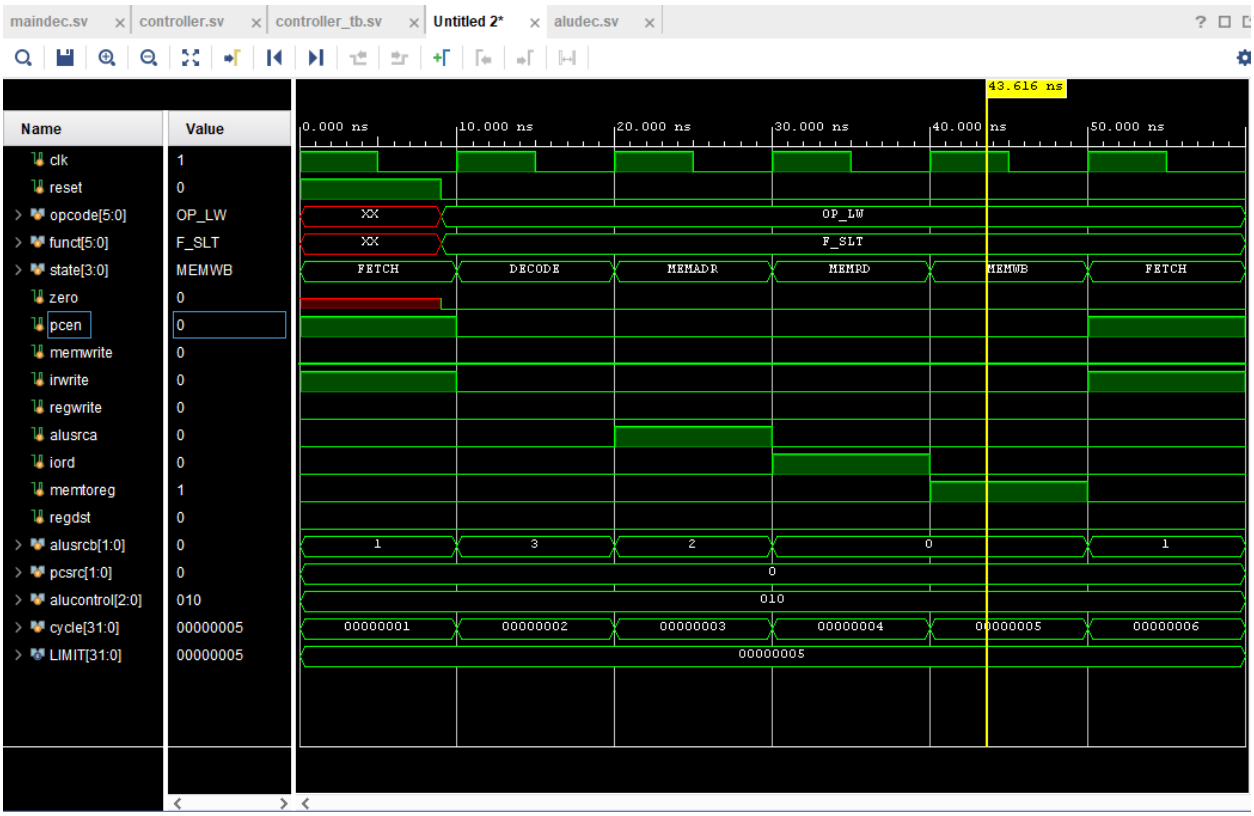

Figure 8 simulation scope for the LW instruction in mips_multicycle processor

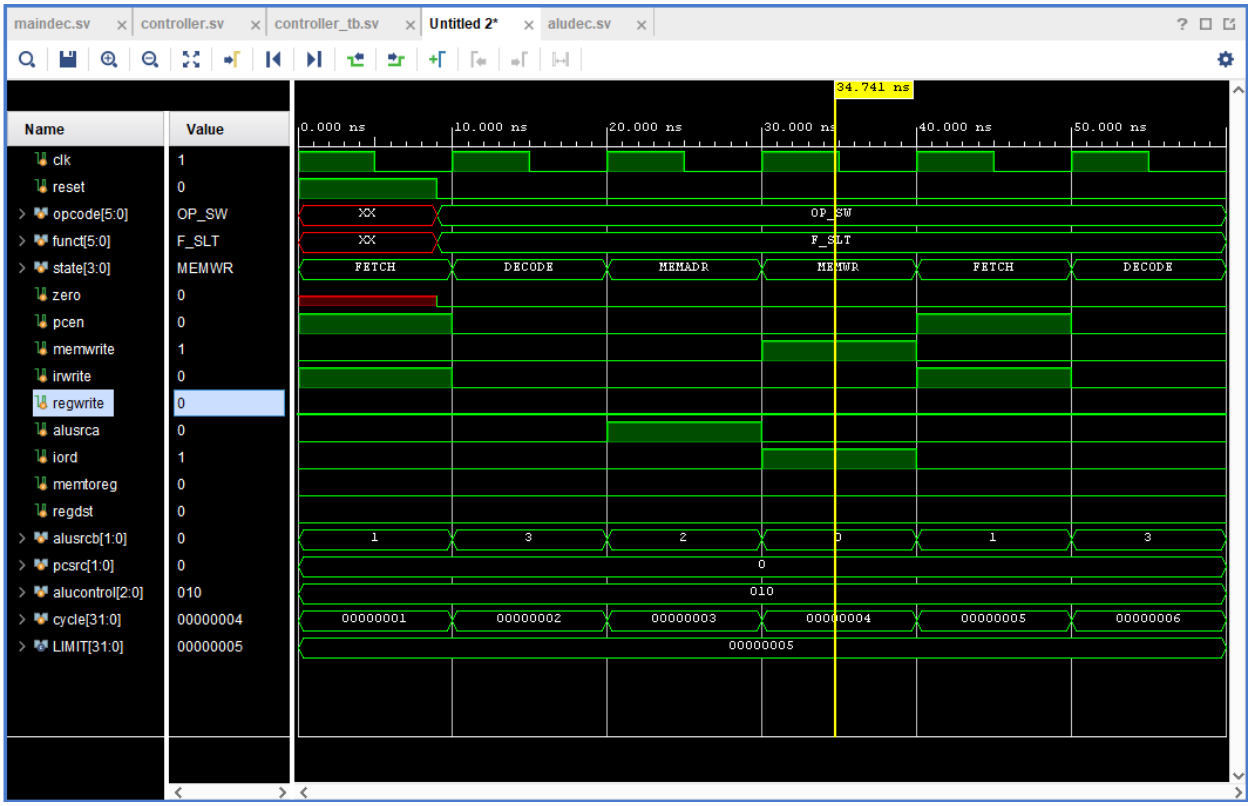

Figure 9simulation scope for the SW instruction in mips_multicycle processor

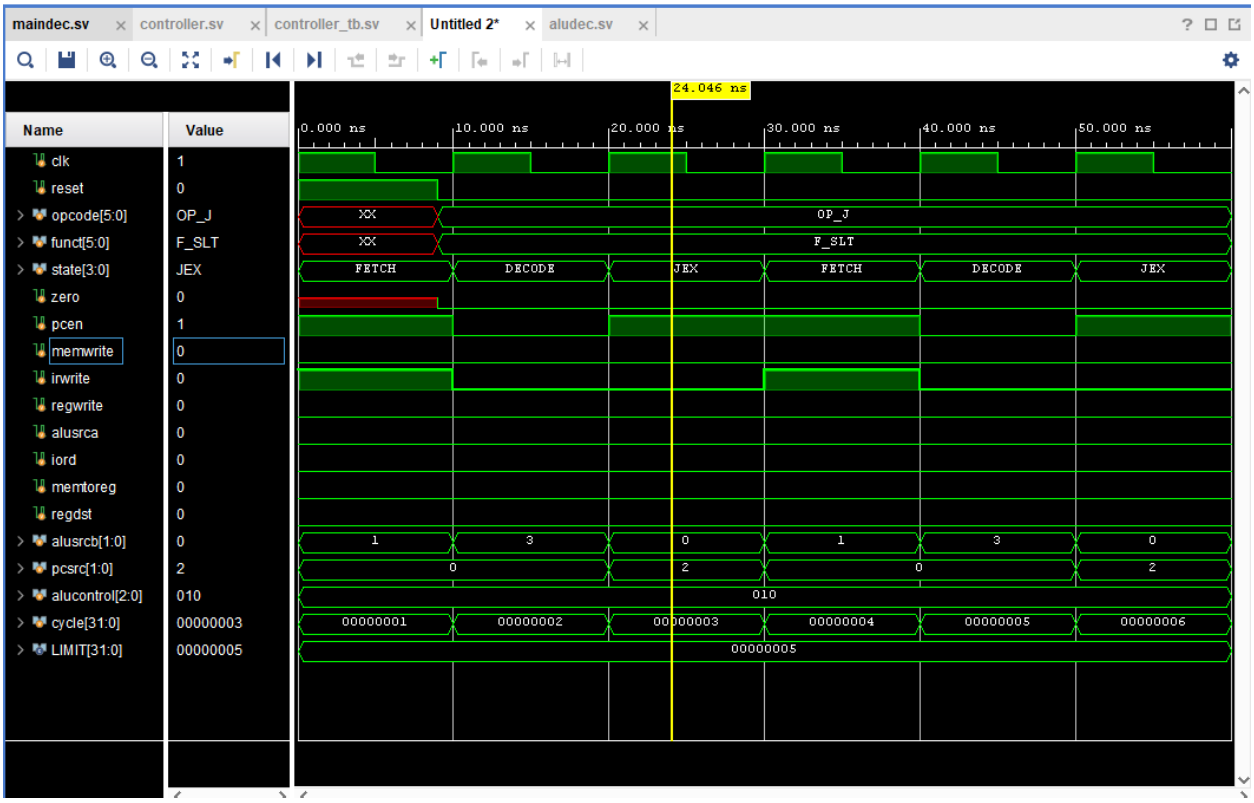

Figure 10simulation scope for the J instruction in mips_multicycle processor

Conclusion

This lab was relatively easier to write, simulate and debug due to its straight forward nature. The only error we ran into involved assigning non-enum type values to enum type variables opcode and funct. We quickly corrected the mistake by importing the enum variables in the testbench file rather than explicitly assigning values to them.

Time spent on the lab

3 hrs