EC 413 COMPUTER ORGANIZATION
FINAL PROJECT REPORT
MONDAY, NOVEMBER 24, 2014
FANGLIANG YE
BU ID 64543053

---

1. Introduction

This is the final report for the EC413 course project, which combines all the results from milestone 1 overall design, milestone 2 CPU controller and milestone 3 working Datapath and finally create a fully functional multicycle CPU that implement a set of MIPS style instructions in Verilog environment as accomplishment of milestone 4. This last step includes some modifications to the original state machine of CPU controller together with some additional hardware to support extra instructions. All these changes and the complete design will be presented in detail in the following report.

2. Design

The Design will include two major categories: Control components and Datapath components. In order to decrease the complexity of the state machine for the controller, we implement main controller (as in milestone 2) together with several other controllers to facilitate the whole range of supported instructions. For the datapath, we will cover both combinational and sequential components include ALU, immediate extension logic, branch logic, MUXes, register file, all inter-state registers and memories separately for Instruction and Data.

2.1 Controller Design

2.1.1 Main Controller with Control Signal Mapping Unit

As usual, a complete instruction cycle will include all or part of the following subtasks: instruction fetch, instruction decode, execution, memory access, register write-back. In this project we implement 26 instruction with four different types (excluding the special instruction NOOP) as illustrated in the **Appendix I - Instruction List**. R-type instruction (01) takes 4 clock cycles excluding the subtask memory access; Jump and its variants (00) have different behaviors, all of them need three cycles to complete with careful configuration of control signals, but they still need different state; Branch instructions (10) can will need three cycles if additional branch test logic is provided; other I-type instruction(11) need to be further divided into three types: ones with memory access (SW/SWI and LW/LWI), ones without memory access (excluding LI and LUI) and the rest arithmetic/logic operation instructions.

We define the following states names with brief explanations:

State 1 - Instruction Fetch

fetch next instruction and write to the instruction register

State 2 - Instruction Decode
decode the instruction and calculate branch addr.
State 3 - Calculation with R2 and Immediate
(11) I-types, use Imm as second operand of ALU
State 4 - Memory Access Read
Read Data memory with calculated address from ALUout_reg
State 5 - Memory Read Complete
Write data from Data memory to the register file with specified write port
State 6 - Memory Access Write
Write data from second read output of register file to Data memory
State 7 - Execution with R2 and R3
ALU calculation with both operands from register file
State 8 - R/I-type Arithmetic/Logic operation Complete
Write back ALU result to the register file
State 9 - Branch Complete
Based on the result of ALU and branch logic to decide if write the PC register
State 10 - Jump Complete
Write to PC register with address calculated in State 2 (shared with branch)
State 11 - JAL Complete
In addition to Jump Complete, write register 31 with current PC + 4
State 12 - Calculation with R1 and Immediate
Use R1 as input select for read port 1 of register file and use that for ALU
State 13 - JR Complete
write address read from specified register to the PC register

In **Appendix II - State Machine**, one can find the state machine diagram. And in **Appendix IV - Control Signals List - Main Controller** part, one can find the corresponding control signal for each state.

In addition, we use one 14-bit control line as main controller output instead of separate individual signals and then we add a signal mapper mapping the control line to each control signals, thus reducing possible code complexity.

2.1.2 ALU Controller

The ALU controller is very similar to the one we used in the class. For the input, ALU controller itself takes control signal ALUOP from the main controller. For the output, basically, Branch types will always use SUB for their third cycle for comparison, other memory and PC addresses calculation will always need ADD, and for the rest arithmetic/Logic calculation, the control output will depend on the last four bits of the opcode itself.

ALUOP	‘b00	‘b01	‘b10
Control output	‘b0010 - ADD	‘b0011 - SUB	opcode[3:0]

2.1.3 PC Controller

PC register get written when PCWRITE signal asserted or both BRANCH and BRANCH_TAKEN signals asserted. PC Controller implement a simple logic to check whether one of above condition satisfied and output a PCupdate signal to enable write of PC register.

2.1.4 Read Port Controller for Register File

This controller is not existing in the original design of milestone 1. The purpose of read port controller is to deal with the cases where we need actually to use R1 as the read port select for both read port 1 and 2 of the register file. In particular, I-type Arithmetic/Logic instructions, such as ADDI, will need R1 to be read from the register file as the first operand of the ALU, as well as for LI/LUI and JR types while SW/SWI and Branch will need R1 to be read from the second read port and use the data from register for second operand of the ALU. This process is originally controlled by main controller, however, since these read port select need to be done in the State 2 - Instruction Decode, we separate this control signal from main controller to avoid additional states for Instruction Decode stage, thus every instruction can still share the same instruction decode state.

opcode[5:4]	‘b00		‘b01	‘b10	‘b11				
Opcode[3:0]	‘b0011	others	d.c.	d.c.	‘b1001	‘b1010	‘b1100	‘b1110	others
Control output*	‘b01	‘b00	‘b00	‘b10	‘b01	‘b01	‘b10	‘b10	‘b00

* lower bit for read port a select and high bit for read port 2 select

2.2 Datapath Design

Detailed design can be found in the **Appendix III - Datapath Design.** In the following part, we will explain how each combinational and sequential components of datapath works.

2.2.1 Arithmetic/Logic Unit (ALU)

In Arithmetic/Logic Unit, we implement functions, such as ADD, which can be identified by the last four bit of opcode. Additionally, we add special function in order to support additional instructions.

control	‘b0000	‘b0001	‘b0010	‘b0011	‘b0100	‘b0101	‘b0110
func_name	MOV	NOT	ADD	SUB	OR	AND	XOR
operation	R2	~R2	R2+R3	R2-R3	R2 | R3	R2 & R3	R2^R3
control	‘b0111	‘b1001	‘b1010	‘b1011	‘b1100		
func_name	SLT	LI	LUI	LWI	SWI		
operation	Set if R2< R3	{R2[31:16],R3[15:0]}	{R3[15:0], R2[15:0]}	R3	R3		

For LI/LUI, the lower/higher half word replace the R2’s corresponding half word. LWI/SWI only need the address from the zero extended immediate, then the ALU just need to return the second operand R3.

2.2.2 Immediate Extend Unit (ImmExt)

Immediate extension include signed and zero extension. This extension process is complicated by two factors. First the length of immediate is not fixed. Jump instruction typically has 26 bits immediate, while other I-type instructions have 16 bit immediate. Second, some of the I-type instructions require zero extension, while some need singed extension. Additional logic need to extend immediate depending on the opcode.

Opcode[5:4]	‘b00	‘b01	‘b10	‘b11				
Opcode[3:0]	d.c.	d.c.	d.c.	‘b0010	‘b0011	‘b0111	‘b1110	other
signed/zero ext.	long/signed	d.c.	signed	signed	signed	signed	signed	zero

2.2.3 Branch Logic Unit (Brancher)

Branch logic takes input from ALU (based on SUB function, compare the R2 and R1) as well as opcode to check the type of branch. Notice we read R1 as second operand, then all sign direction should be reversed. If the ALU result matches the branch type, Branch Logic output BRANCH_TAKEN signal to PC Controller.

2.2.4 MUXes

The design includes seven Muxes. Four muxes for register file input port: write port mux select R1 or $31 (JAL); read port 1 mux select R2 or R1; read port 2 mux select R3 or R1; write data mux selects output from ALUout, ALU_Mem, or PC value for JAL. Two muxes for ALU sources: ALU source A mux selects from current PC or data from register A (data from reading register file read output 1); ALU source B mux selects from register B (data from reading register file read output 2), 1 (for PC increment), extended immediate value (from ImmExt Unit), or 0 (for JR instruction). And at last PC source mux selects from nPC or calculated Jump/branch PC values.

2.2.5 Resisters and Memories

Register file for this project is provided as well as the instruction and data memories. In the datapath, we need register to store intermediate result between different cycles for the same instruction. Typically, we need PC register for PC value, instruction register to hold the instruction executing, ALU source registers A and B to store the read value from register file, and ALUout register to store the ALU result, data memory register to hold data read from the data memory and at last JAL register for holding the returning PC value (current PC + 4).

Combining all the components of controllers and datapath, we complete the design of a multicycle CPU. In the next section, we will discuss how we implement it in Verilog code and the simulation results.

3. Implementation and Simulation

3.1 Verilog Implementations

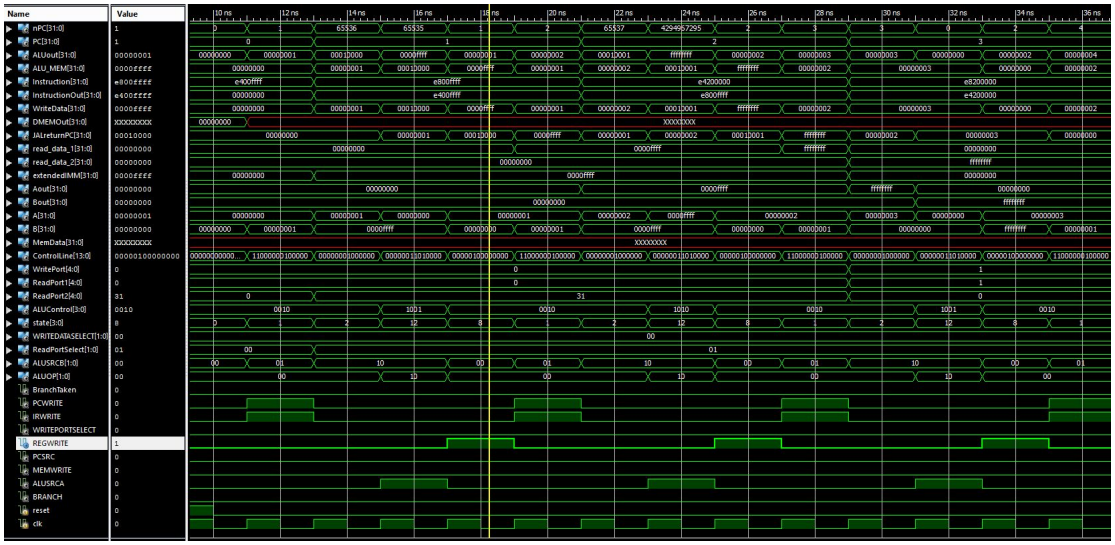
The main structure for the controller is basically the same with milestone 2 except for 3 additional states for supporting extra instructions. Detailed codes can be found in the attached code *Controller.v.* The rest of controllers and datapath components can also be found and they are basically following the design described in the above section.

3.2 Simulation Results and Discussion

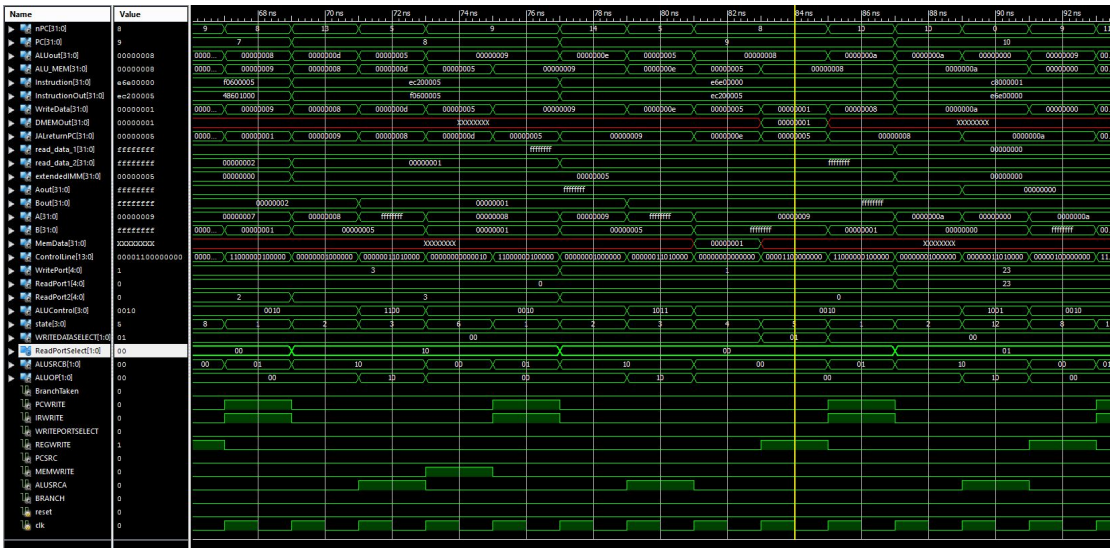
The test for the code is done by feed the instruction memory with different sequence of instructions for execution. There is three sets of different instruction sequences reload in the provided *IMem.v* file, *Program 1, 2 and 3*. We add a program 4 to test JAL and JR. The result is as follows.

Program 1

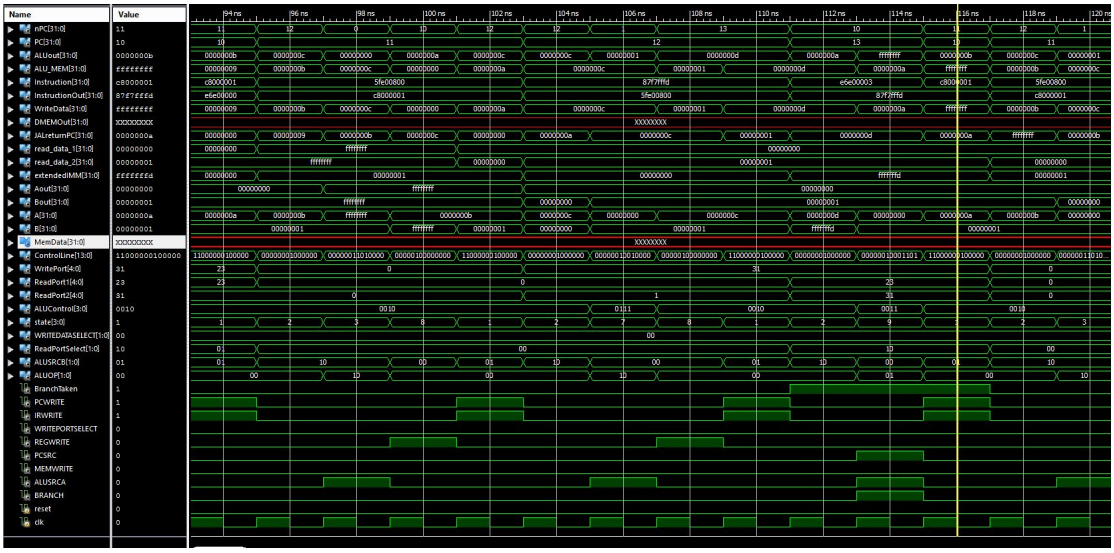
For first several instructions, the PC is incrementing correctly.



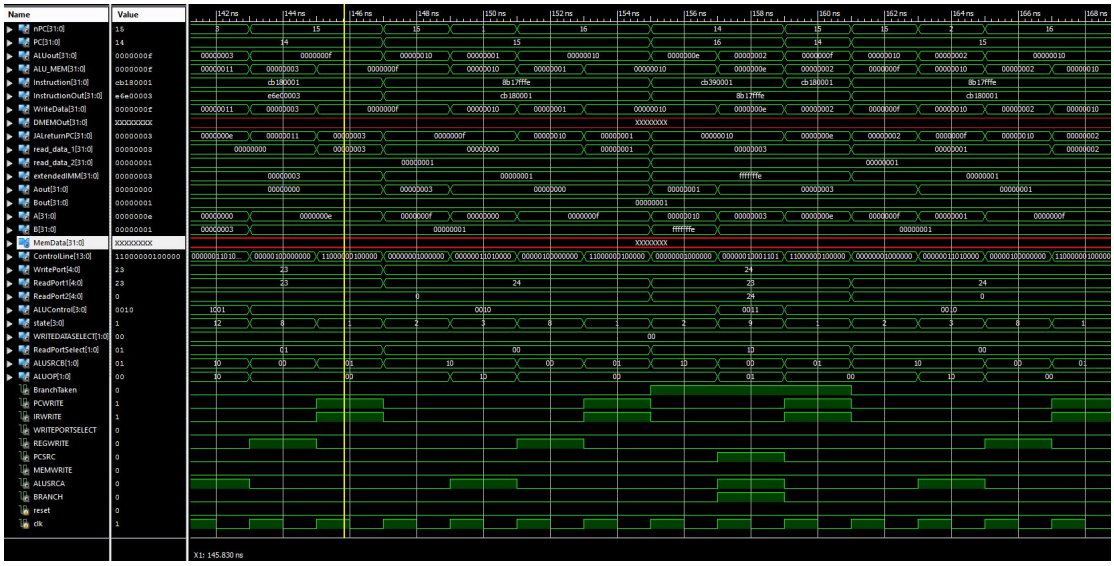
Instruction 7 and 8 is about SWI and LWI. We can see they are functioning properly.



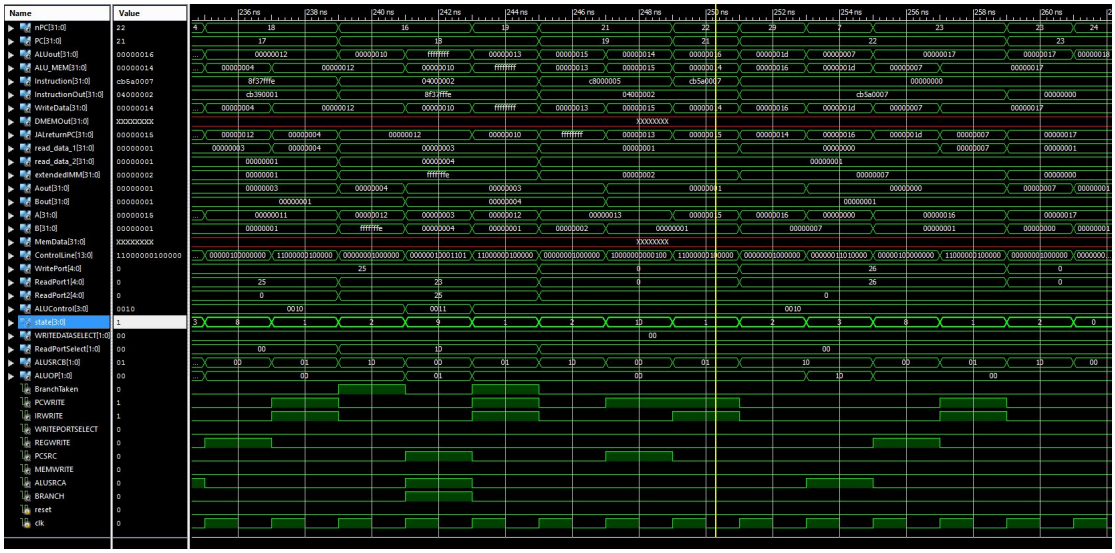
Instruction 10,11,12 is a simple loop and only execute twice.



Instruction 14,15 is another loop and this will execute 3 times.



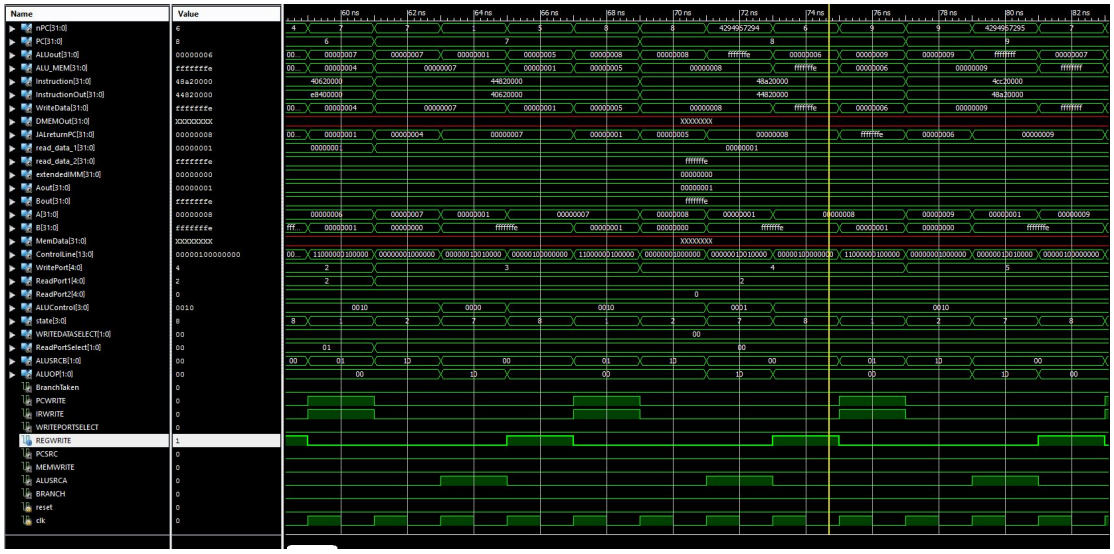
Instruction 16,17 are similar to 14 and 16. It is observed that all these branch instructions are working properly. Then we focus on instruction 18 which is a jump to instruction 21. It works well.

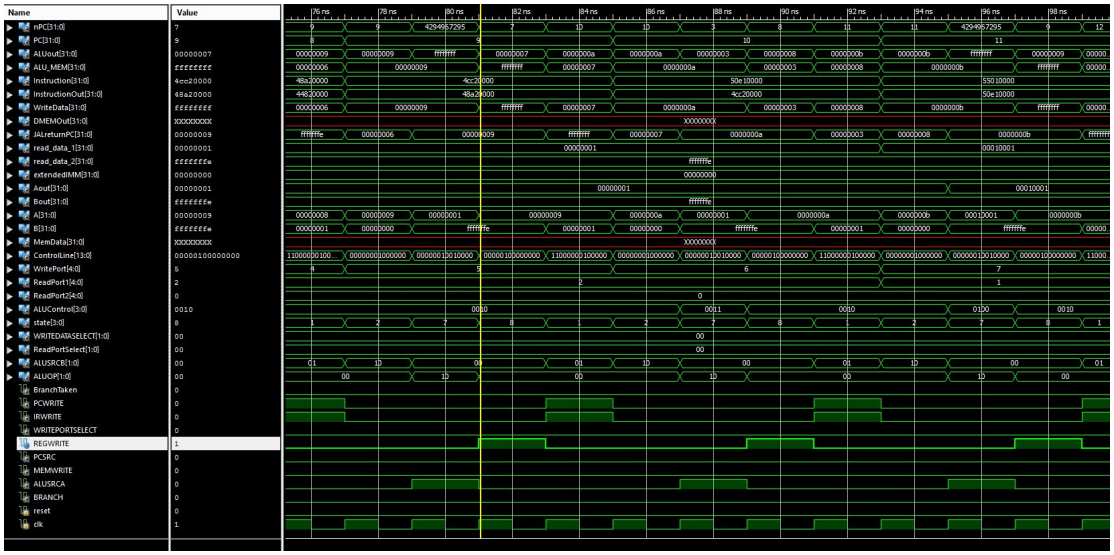


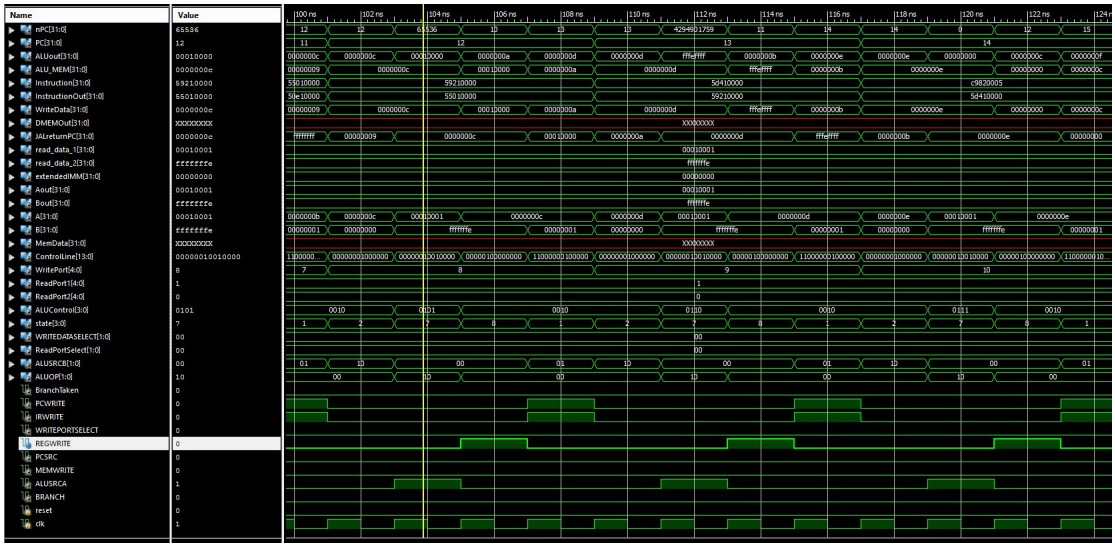
Program 2

This program is aimed to check all arithmetic and logic operation for both R-type and I-type.

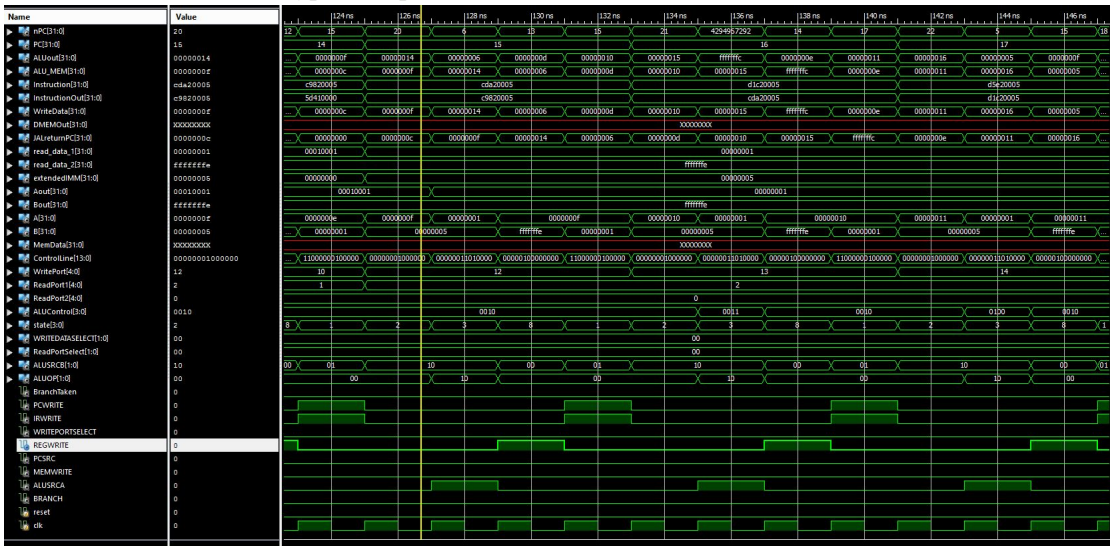
We should start with instruction 6. the results are correct based on the observation.

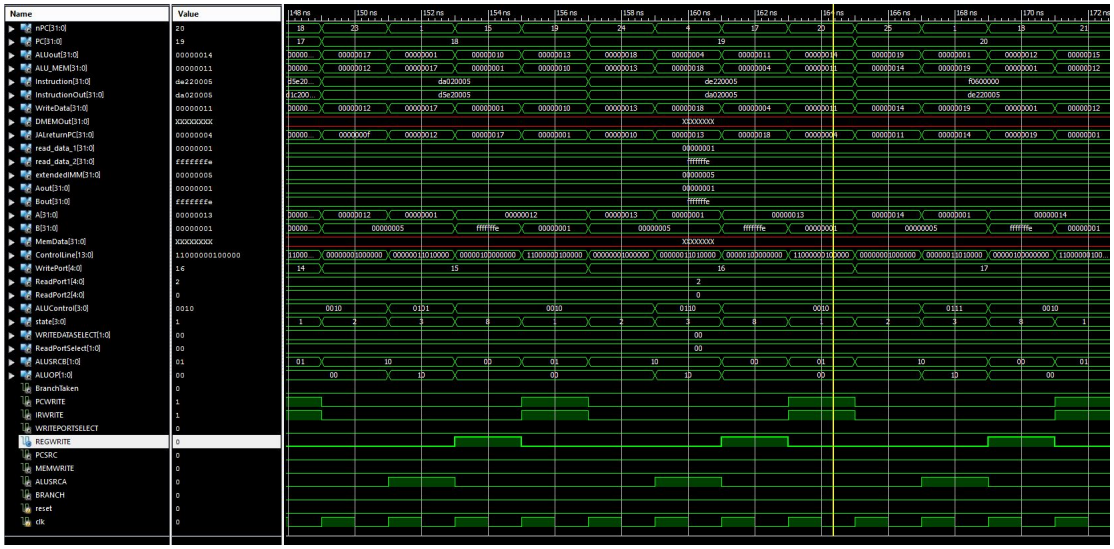




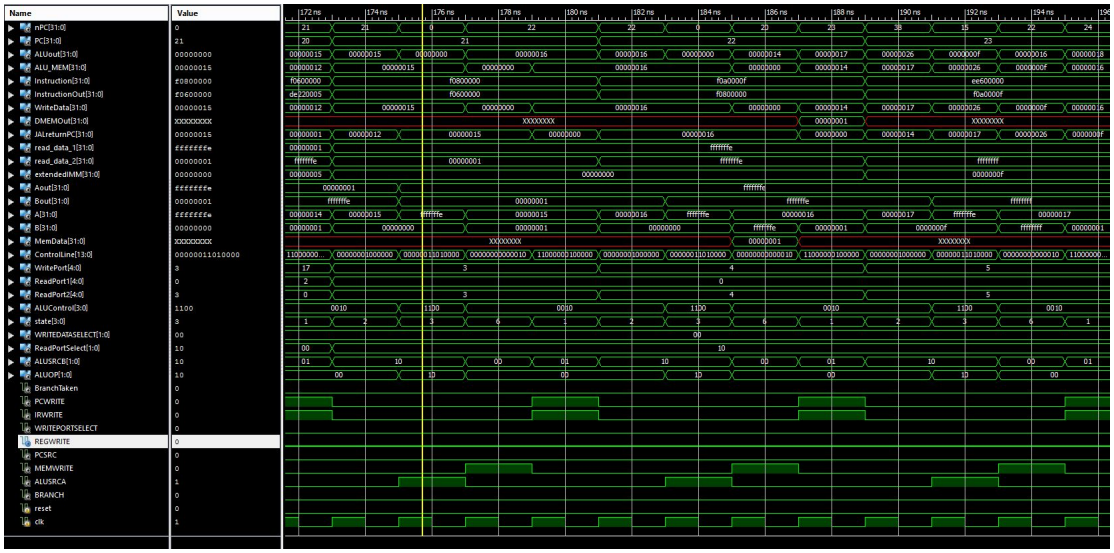


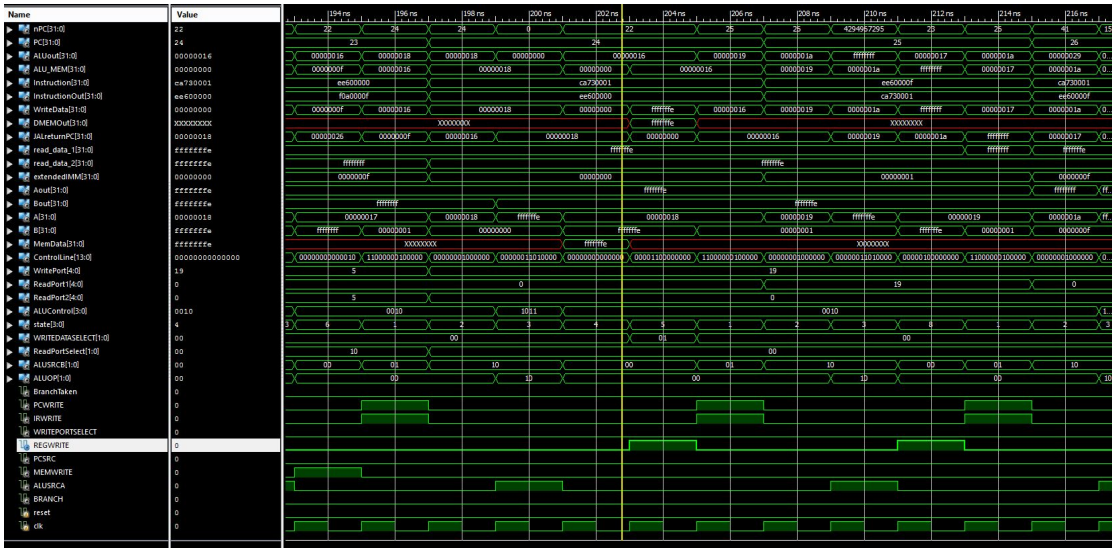
Instruction 14-19 is for I-type AL operation





Instruction 20 - 26 is for LWI and SWI testing.



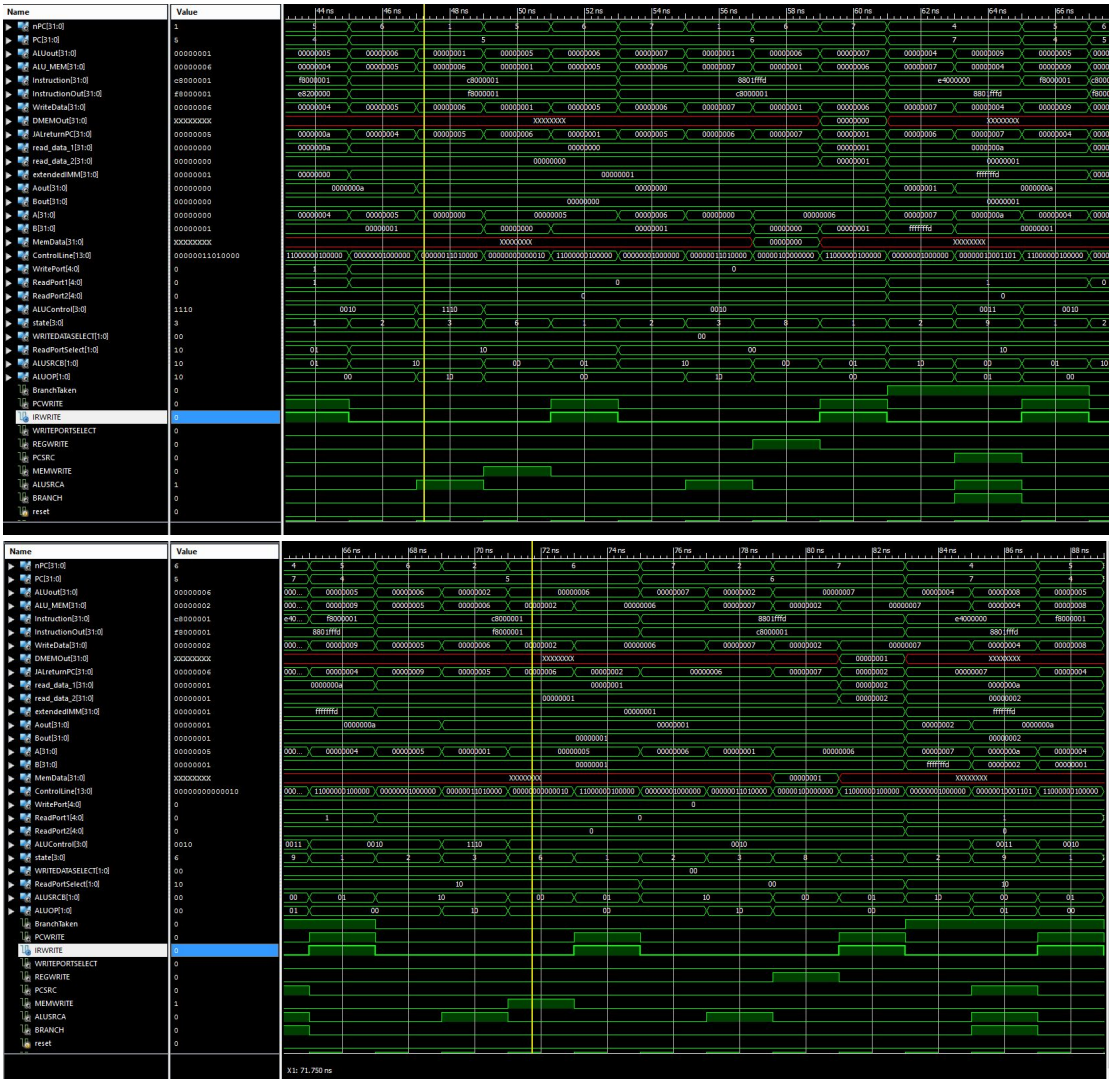


The result is quite satisfactory.

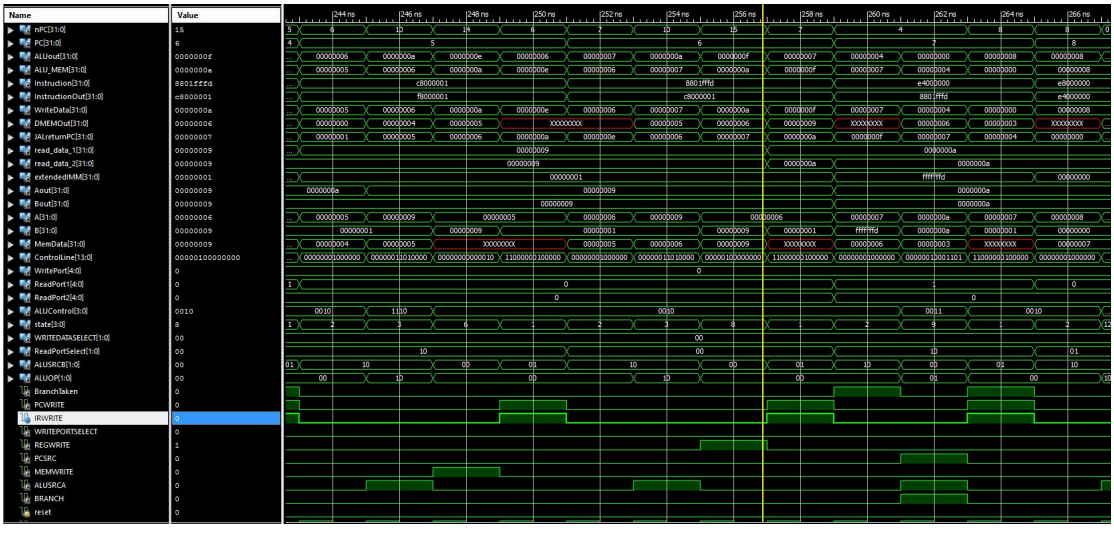
Program 3

Program 3 is for testing SW and LW

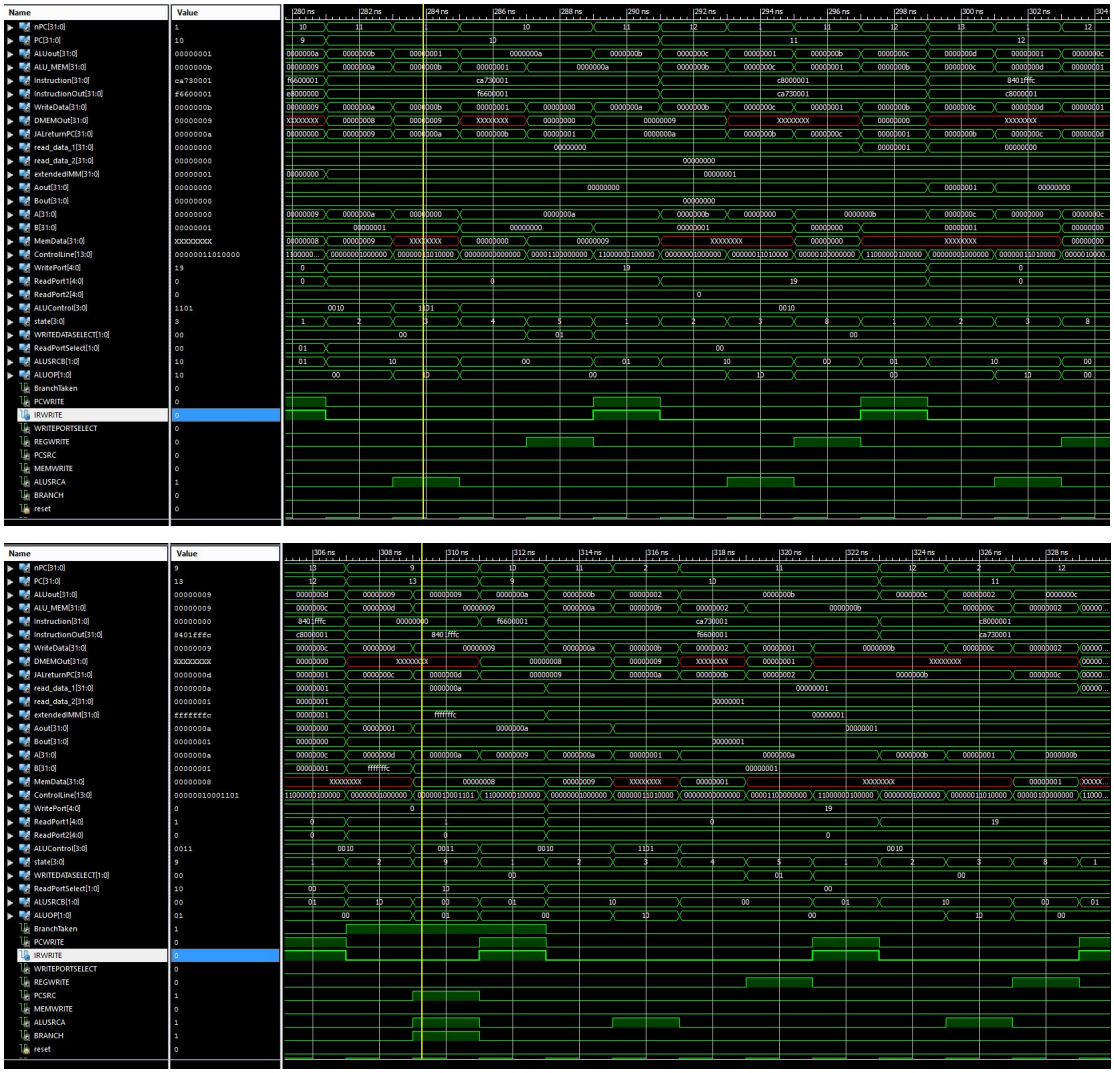
Instruction 4- 6 is a loop that store 0 - 9 to address 1 - 10.



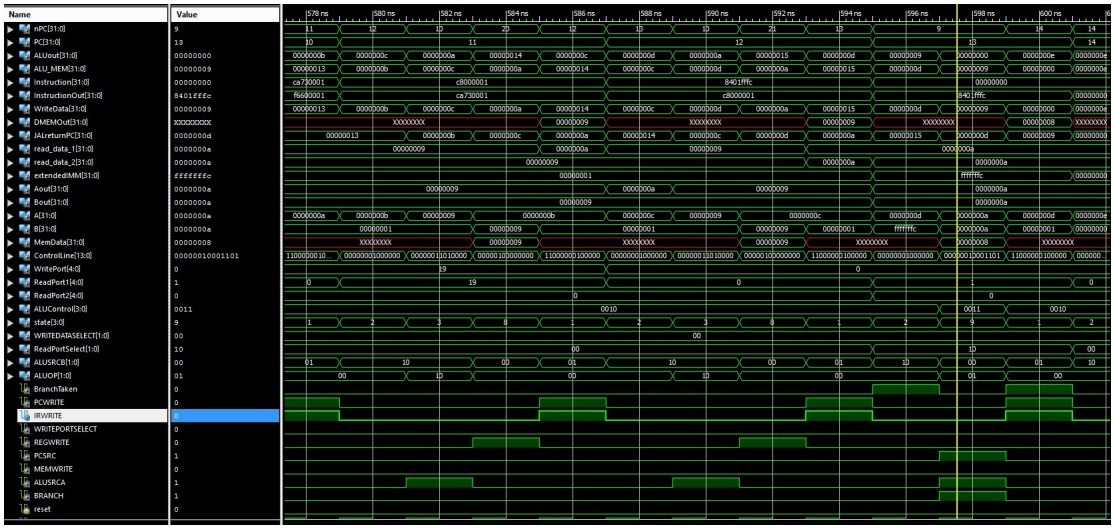
And last loop



Instruction 9-12 is another loop that load the memory.



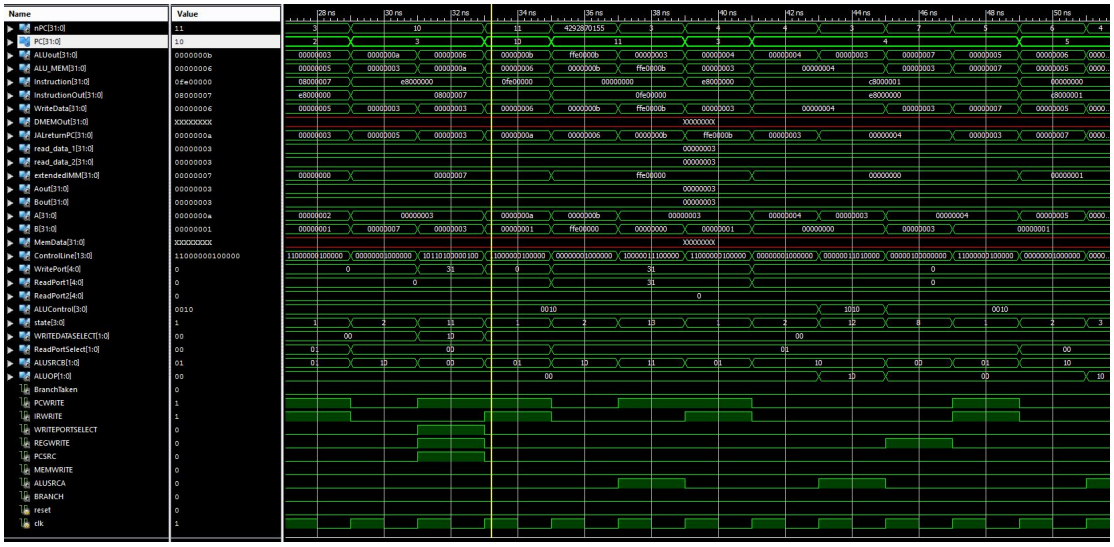
And last loop



Program 4

Program 4 is written in addition to test JAL and JR. We only need to focus on instruction 2,3,4,5,10.

First we will jump and link at instruction 2. Then jump register back to instruction 3.



Clearly, JAL store the current PC + 4 in the register 31 and then jump to instruction 10, and then instruction 10 which is a JR instruction , read register 31 and jump back to instruction 3.

4. Evaluation and Conclusion

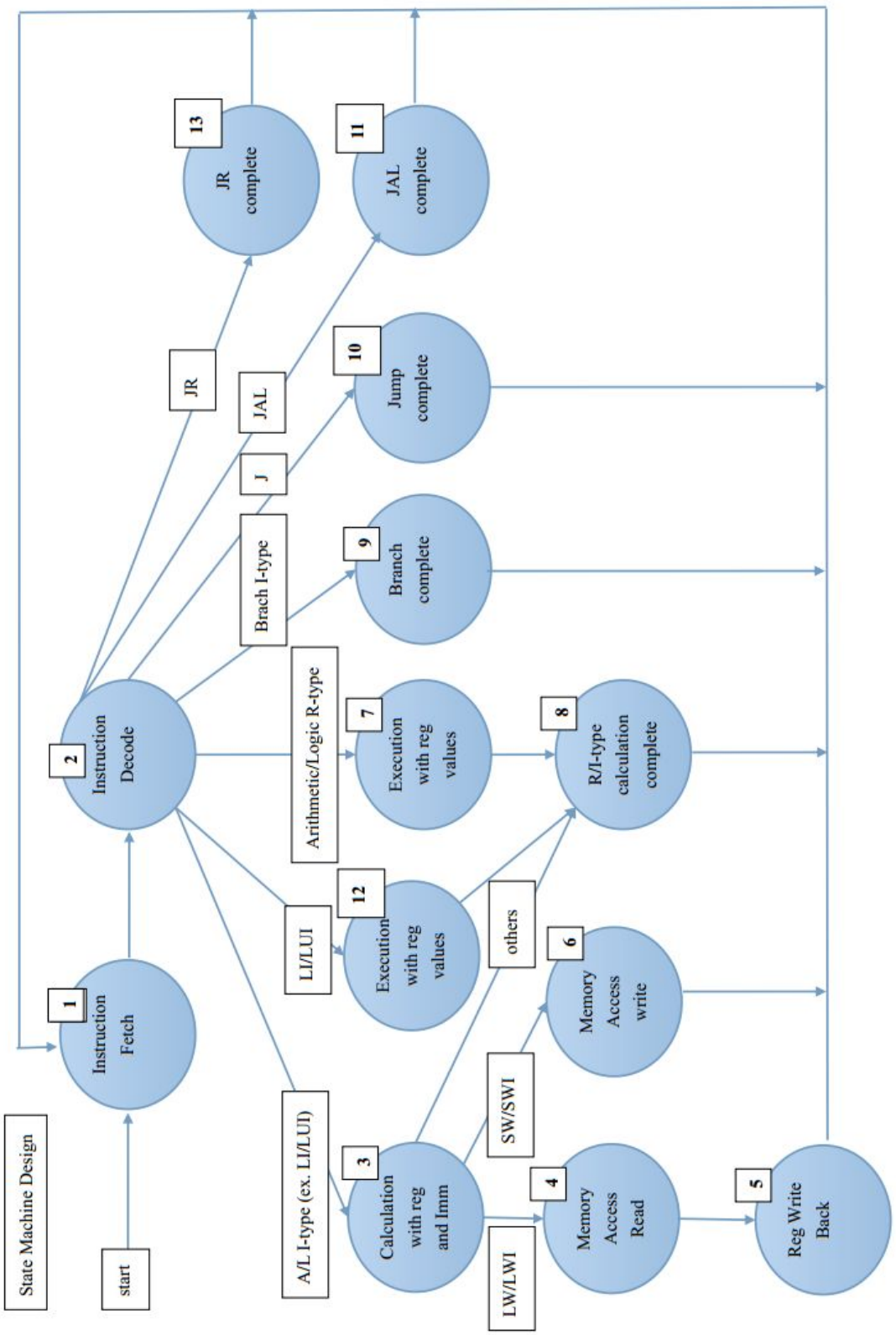
In this project, the major difficulties encountered when we try to extend the supported instructions. Though it looks very easy at first glance as just need to add one or more state in the state machine for the controller, in fact it turns out that we probably adding more control signals or additional hardware to make the newly added instruction work together with the existing ones. All verilog code for the datapath did not take long to write, while debugging with iSim takes me double the time of write up the codes themselves. Sometimes the mistake is trivial, such as using a wrong wire in connecting all components in the top module, it still hard to identify. One need to have a very clear picture of the whole system design and what happens in every cycle of each instruction in order to pinpoint where a mistake is made.

In conclusion, we implement a fairly complex multicycle CPU and the result we get from the iSim is convincing. We did not implement exception handling mechanism which further increase the complexity of the system. Through the design, implement and test process, we received some valuable practical experience in CPU design and understand multicycle CPU better.

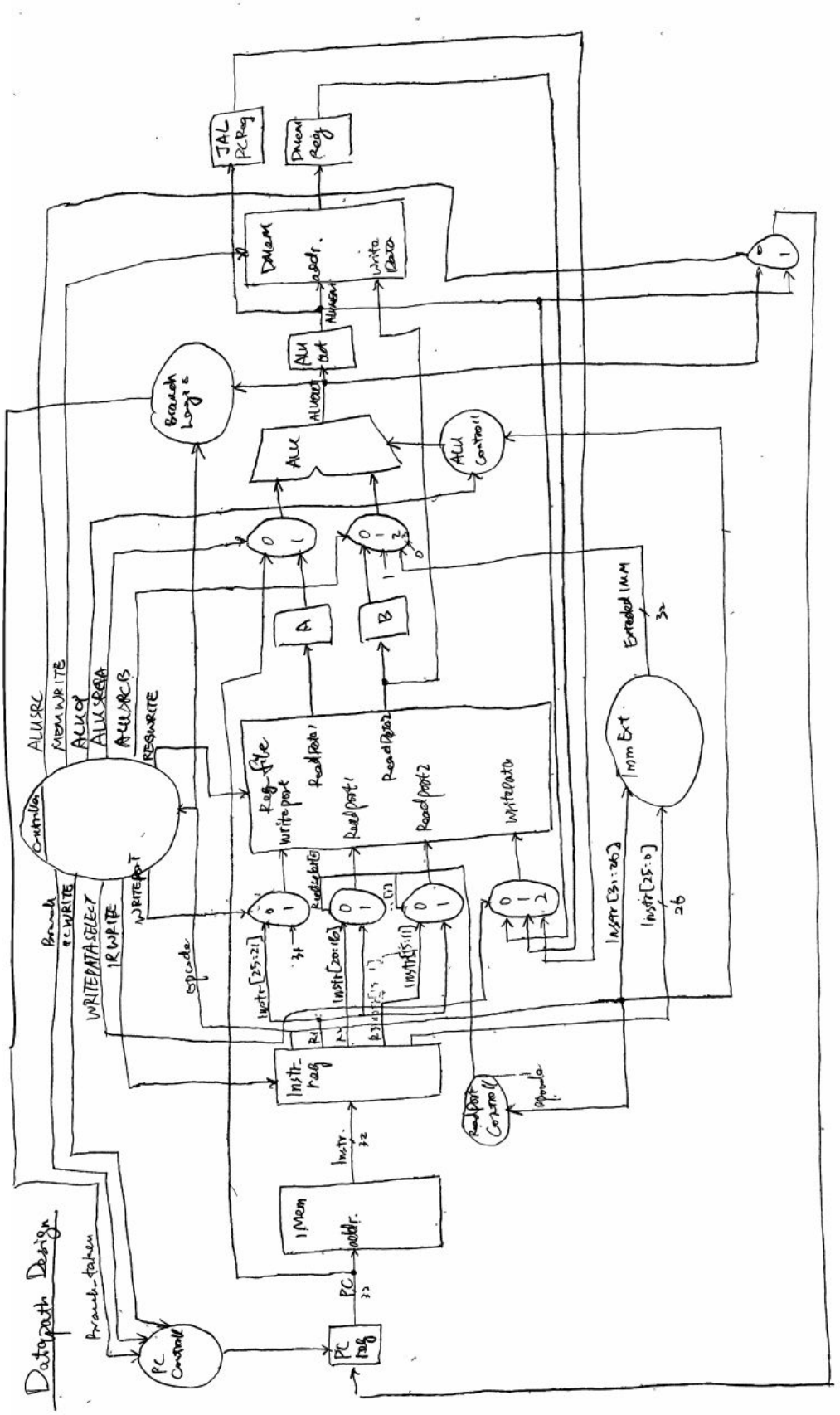
Appendix I - Instruction List

000000	NOOP	
010000	MOV	R1 = R2
010001	NOT	R1 = ~R2
010010	ADD	R1 = R2+R3
010011	SUB	R1 = R2-R3
010100	OR	R1 = R2 | R3
010101	AND	R1 = R2&R3
010110	XOR	R1 = R2^R3
010111	SLT	R1 = set if R2 < R3 else clear
000001	J	PC<-PC+SE(Limm)
000010	JAL	$31 = PC+4, PC<-PC+SE(Limm)
000011	JR	PC<-R1
100000	BEQ	PC<-PC+SE(IMM) if R1=R2
100001	BNE	PC<-PC+SE(IMM) if R1 != R2
100010	BLT	PC<-PC+SE(IMM) if R1 < R2
100011	BLE	PC<-PC+SE(IMM) if R1 <= R2
110010	ADDI	R1 = R2+SE(IMM)
110011	SUBI	R1 = R2-SE(IMM)
110100	ORI	R1 = R2|ZE(IMM)
110101	ANDI	R1 = R2&ZE(IMM)
110110	XORI	R1 = R2^ZE(IMM)
110111	SLTI	R1 = set if R2 < SE(IMM) else clear
111001	LI	R1[15:0]=ZE(IMM)
111010	LUI	R[31:16]=ZE(IMM)
111011	LWI	R1<-M[ZE(IMM)]
111100	SWI	M[ZE(IMM)]<-R1
111101	LW	R1 <- M[R2+SE(IMM)]
111110	SW	M[R2+SE(IMM)] <-R1

Appendix II - State Machine



Appendix III - Datapath Design



Appendix IV - Control Signals List

Main Controller

R-Type Control Signal List

Cycle no.	1	2	3	4	
State no.	1	2	7	8	
Control Signals:					
PCWRITE	1	0	0	0	
IRWRITE	1	0	0	0	
WRITEPORTSELECT	0	0	0	0	
WRITEDATASELECT	00	00	00	00	
REGWRITE	0	0	0	1	
ALUSRCA	0	0	1	0	
ALUSRCB	01	10	00	00	
ALUOP	00	00	10	00	
PCSRC	0	0	0	0	
MEMWRITE	0	0	0	0	
BRANCH	0	0	0	0	

Jump Control Signal List

Cycle no.	1	2	3		
State no.	1	2	10		
Control Signals:					
PCWRITE	1	0	1		
IRWRITE	1	0	0		
WRITEPORTSELECT	0	0	0		
WRITEDATASELECT	00	00	00		
REGWRITE	0	0	0		
ALUSRCA	0	0	0		
ALUSRCB	01	10	00		
ALUOP	00	00	00		
PCSRC	0	0	1		
MEMWRITE	0	0	0		
BRANCH	0	0	0		

JAL Control Signal List

Cycle no.	1	2	3		
State no.	1	2	11		
Control Signals:					
PCWRITE	1	0	1		
IRWRITE	1	0	0		
WRITEPORTSELECT	0	0	1		
WRITEDATASELECT	00	00	10		
REGWRITE	0	0	1		
ALUSRCA	0	0	0		
ALUSRCB	01	10	00		
ALUOP	00	00	00		
PCSRC	0	0	1		
MEMWRITE	0	0	0		
BRANCH	0	0	0		

JR Control Signal List

Cycle no.	1	2	3		
State no.	1	2	13		
Control Signals:					
PCWRITE	1	0	1		
IRWRITE	1	0	0		
WRITEPORTSELECT	0	0	0		
WRITEDATASELECT	00	00	00		
REGWRITE	0	0	0		
ALUSRCA	0	0	1		
ALUSRCB	01	10	11		
ALUOP	00	00	00		
PCSRC	0	0	0		
MEMWRITE	0	0	0		
BRANCH	0	0	0		

Branch Control Signal List

Cycle no.	1	2	3		
State no.	1	2	9		
Control Signals:					
PCWRITE	1	0	0		
IRWRITE	1	0	0		
WRITEPORTSELECT	0	0	0		
WRITEDATASELECT	00	00	00		
REGWRITE	0	0	0		
ALUSRCA	0	0	1		
ALUSRCB	01	10	00		
ALUOP	00	00	01		
PCSRC	0	0	1		
MEMWRITE	0	0	0		
BRANCH	0	0	1		

I-type without Mem access Control Signal List (exclude LI/LUI)

Cycle no.	1	2	3	4	
State no.	1	2	3	8	
Control Signals:					
PCWRITE	1	0	0	0	
IRWRITE	1	0	0	0	
WRITEPORTSELECT	0	0	0	0	
WRITEDATASELECT	00	00	00	00	
REGWRITE	0	0	0	1	
ALUSRCA	0	0	1	0	
ALUSRCB	01	10	10	00	
ALUOP	00	00	10	00	
PCSRC	0	0	0	0	
MEMWRITE	0	0	0	0	
BRANCH	0	0	0	0	

I-type without Mem access Control Signal List (LI and LUI)

Cycle no.	1	2	3	4	
State no.	1	2	12	8	
Control Signals:					
PCWRITE	1	0	0	0	
IRWRITE	1	0	0	0	
WRITEPORTSELECT	0	0	0	0	
WRITEDATASELECT	00	00	00	00	
REGWRITE	0	0	0	1	
ALUSRCA	0	0	1	0	
ALUSRCB	01	10	10	00	
ALUOP	00	00	10	00	
PCSRC	0	0	0	0	
MEMWRITE	0	0	0	0	
BRANCH	0	0	0	0	

I-type with Mem read access Control Signal List

Cycle no.	1	2	3	4	5
State no.	1	2	3	4	5
Control Signals:					
PCWRITE	1	0	0	0	0
IRWRITE	1	0	0	0	0
WRITEPORTSELECT	0	0	0	0	0
WRITEDATASELECT	00	00	00	00	01
REGWRITE	0	0	0	0	1
ALUSRCA	0	0	1	0	0
ALUSRCB	01	10	10	00	00
ALUOP	00	00	10	00	00
PCSRC	0	0	0	0	0
MEMWRITE	0	0	0	0	0
BRANCH	0	0	0	0	0

I-type with Mem write access Control Signal List

Cycle no.	1	2	3	4	
State no.	1	2	3	6	
Control Signals:					
PCWRITE	1	0	0	0	
IRWRITE	1	0	0	0	
WRITEPORTSELECT	0	0	0	0	
WRITEDATASELECT	00	00	00	00	
REGWRITE	0	0	0	0	
ALUSRCA	0	0	1	0	
ALUSRCB	01	10	10	00	
ALUOP	00	00	10	00	
PCSRC	0	0	0	0	
MEMWRITE	0	0	0	1	
BRANCH	0	0	0	0	