Assignment 2

Name: **Jaehak Kim (김재학)**

Student ID: **202220757**

Question:	1	2	3	4	5	Total
Points:	15	20	10	20	55	120
Score:						

1. Modify the single-cycle ARM processor to implement following instruction:

LSL with an immediate shift amount

Answer:

For LSL instruction, the shifter was connected right before SrcB. As shown in Figure 1, a Shift Block has been added to the circuit diagram provided in the problem. In the figure, the green lines represent the newly added Data path, and the orange lines represent the newly added Control signals. As a result of this improved circuit, not only the MOV operation but also Shift operations become possible.

A new control signal, referred to as ShOp, is extracted by decoding bits [6:5] of the instruction. This signal is introduced to support shift-related operations in the enhanced Data path. Actually, no complex decoding is required, as bits [6:5] are directly utilized as the ShOp signal. Additionally, bits [11:7] are forwarded to the Data path as shamt5, which specifies the shift amount. The shift operation determined by the ShOp signal is applied accordingly by shamt5.

The LSL (Logical Shift Left) operation is implemented by extending the MOV operation with register-based shifting. To support the MOV operation, it is also necessary to modify the ALUSrc control signal. In the machine code for MOV, the Rn field is encoded with the value 0. By performing an ALU operation between this value and Src2, a pass-through behavior can be achieved, effectively implementing the MOV instruction. Refer to Table 1 for the corresponding truth table of the control unit.

With this circuit configuration and the use of an ALU decoder, the Main Decoder can be utilized without any modification, maintaining compatibility with the original design.

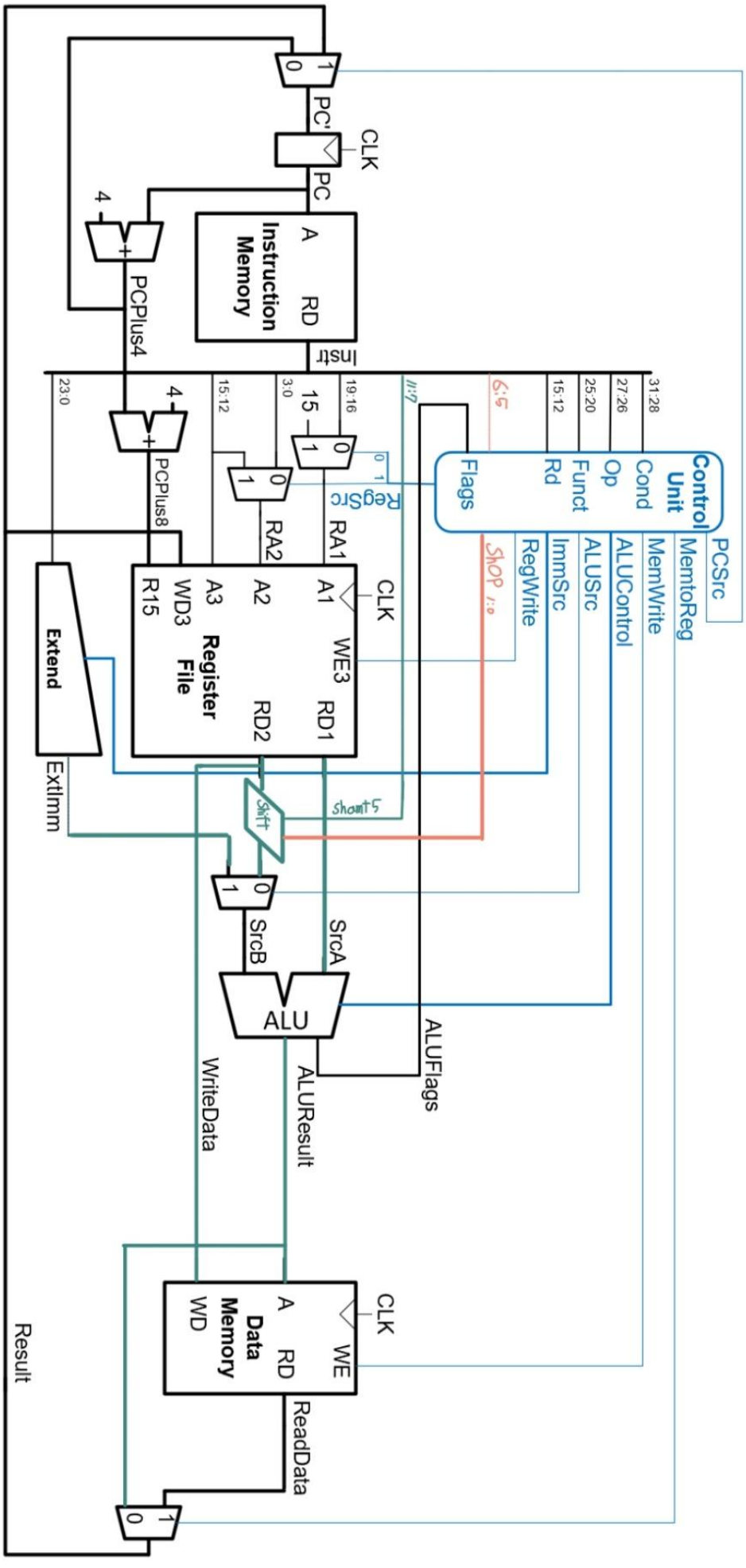
PC'
CLK
PC
4
A
RD
Instruction Memory
Instr
PCPlus4
PCPlus8
31:28
27:26
25:20
15:12
19:16
15:12
3:0
23:0
15
Control Unit
Cond
Op
Funct
Rd
Flags
PCSrc
MemtoReg
MemWrite
ALUControl
ALUSrc
ImmSrc
RegWrite
RegSrc
RA1
RA2
A1
A2
A3
WD3
R15
CLK
WE3
RD1
RD2
Register File
Extend
ExtImm
Shift
Shamt5
SrcA
SrcB
ALU
ALUFlags
ALUResult
WriteData
A
RD
WD
CLK
WE
Data Memory
ReadData
Result

Figure 1.

ALUOp	$\mathbf{Funct_{4:1}}$ *(cmd)*	$\mathbf{Funct_0}$ *(S)*	Type	$\boldsymbol{ALUControl_{1:0}}$	$\boldsymbol{FlagW_{1:0}}$
0	X	X	Not DP	00	00
1	0100	0	ADD	00	00
		1			11
	0010	0	SUB	01	00
		1			11
	0000	0	AND	10	00
		1			10
	1100	0	ORR	11	00
		1			10
	1101	0	MOV	00	00
		1			10

Table 1.

2. **Using a diagram similar to that in the page 109 of ca-w10 slides, show the forwarding and stalls needed to execute the following instructions on the pipelined ARM processor.**

   ADD R0, R4, R9
   SUB R0, R0, R2
   LDR R1, [R0, #60]
   AND R2, R1, R0

Answer: See Figure 2.

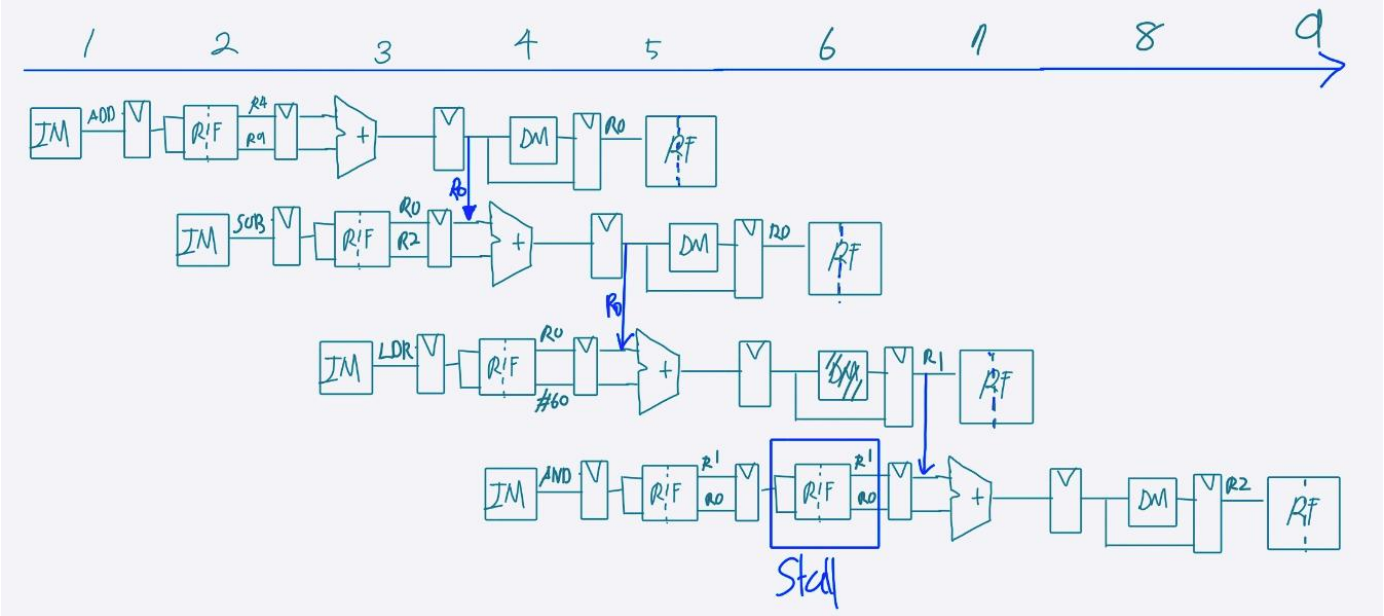

Figure 2.

3. **Suppose the ARM pipelined processor is divided into 10 stages of 400ps each, including sequencing overhead. Assume the instruction mix as:**

   - 25% loads;
   - 10% stores;
   - 13% branches;
   - And 52% data-processing instructions.

   Also assume that 50% of the loads are immediately followed by an instruction that uses the result, requiring six stalls, and that 30% of the branches are mispredicted. The target address of a branch instruction is not computed until the end of the second stage.

   (a) Calculate the average CPI. Please show a detailed calculation process including the CPIs of the loads and the branches instructions.

Answer:

$$Load\ CPI = 1(0.5) + 7(0.5) = 4$$

$$Branch\ CPI = 1(0.7) + 3(0.3) = 1.6$$

$$Average\ CPI = 0.25(4) + 0.1(1) + 0.13(1.6) + 0.52(1) = 1.828$$

   (b) Calculate execution time of computing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor.

Answer:

$$Excution\ time = (\#instruction) \times CPI\ \times\ T_C$$

$$= 100\ \times\ 10^9\ \times 1.828\ \times 400\ \times\ 10^{-12}\,\text{sec} = 73.12\ sec$$

4. **You are building a computer with a hierarchical memory system that consists of separate instruction and data caches followed by main memory. You are using the ARM multicycle processor running at 1 GHz.**

   (a) The instruction cache is perfect (i.e., always hits) but the data cache has a 15% miss rate. On a cache miss, the processor stalls for 200 ns to access main memory, then resumes normal operation. Taking cache misses into account, what is the average memory access time (AMAT)?

Answer:

$$AMAT = t_{cache} + MR_{cache}\ \times t_{MM}$$

$$= 1 + 0.15 * 200\ cycle = 31\ cycle$$

(b) How many clock cycles per instruction (CPI) on average are required for load and store word instructions considering the non-ideal memory system?

Answer:

$$CPI = \frac{AMAT}{Clock\ cycle\ time} = \frac{31}{1} = 31\ cycle$$

(c) Consider the benchmark application of Question 4 that has 25% loads, 10% stores, 13% branches, and 52% data-processing instructions. Taking the non-ideal memory system into account, what is the average CPI for this benchmark?

Answer:

$$Average\ CPI = (0.25 + 0.1) * 31 + (0.13 + 0.52) * 1 = 11.5$$

(d) Now suppose that the instruction cache is also non-ideal and has a 10% miss rate. What is the average CPI for the benchmark in part (c)? Take into account both instruction and data cache misses.

Answer:

$$Fetch\ AMAT = \ 1 + 0.1 * 200 = 21\ cycles$$

$$Average\ CPI + Fetch\ AMAT = \mathbf{32.5}$$

5. Single-cycle Arm

(a) Testing the single-cycle ARM processor:

Answer:

- See **Figure 3.**
- In the final STR instruction, It stores the value 0x00000007 at memory address 0x000000EC.
- See **Figure 4, 5, 6.**

Cycle	reset	PC	Instr	SrcA	SrcB	Branch	AluResult	Flags$_{3:0}$ [NZCV]	CondEx	WriteData	MemWrite	ReadData
1	1	00	SUB R0, R15, R15 E04F000F	8	8	0	0	?	1	8	0	x
2	0	04	ADD R2, R0, #5 E2802005	0	5	0	5	?	1	x	0	x
3	0	08	ADD R3, R0, #12 E280300C	0	C	0	C	?	1	x	0	x
4	0	0C	SUB R9, R3, #09 E2439009	C	9	0	3	?	1	X	0	X
5	0	10	ORR R4, R9, R2 E1894002	3	5	0	7	?	1	X	0	X
6	0	14	AND R5, R3, R4 E0035004	C	7	0	4	?	1	X	0	X
7	0	16	ADD R5, R5, R4 E0855004	4	7	0	B	?	1	X	0	X

Figure 3.

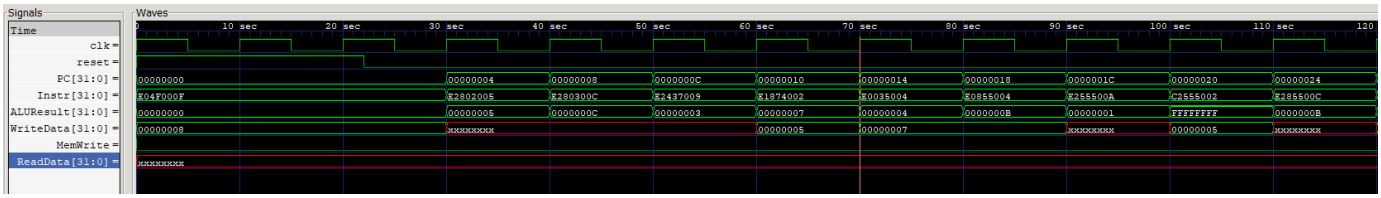

Figure 4.

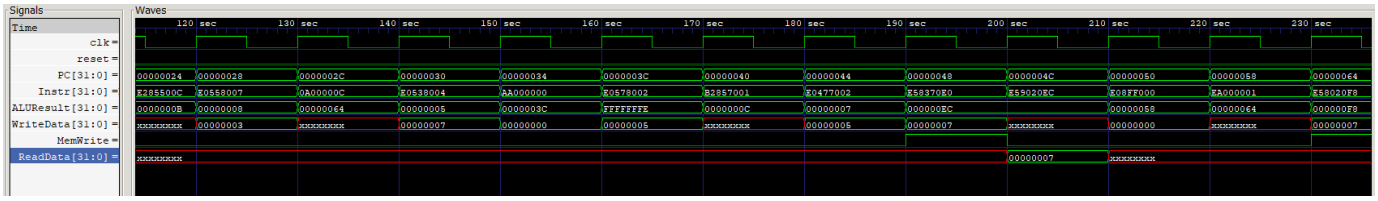

Figure 5.

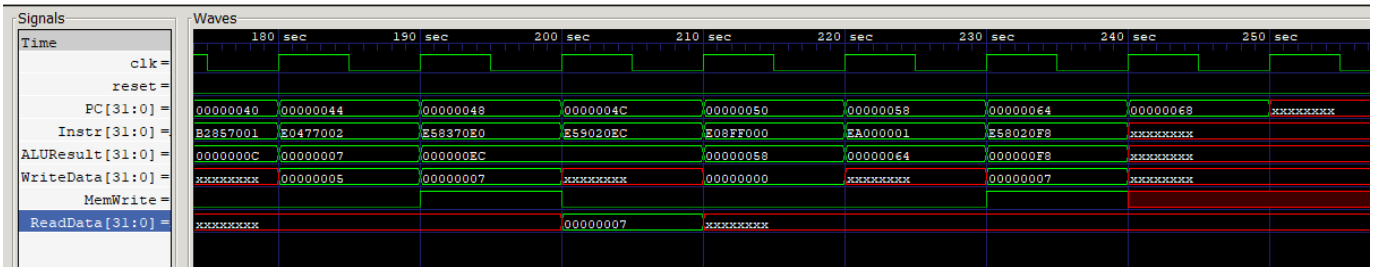

Figure 6.

Before	After

// check results
always @(negedge clk)
begin
if(MemWrite) begin
if(DataAdr === 100 & WriteData === 7) begin
$display("Simulation succeeded");
$stop;
end else if (DataAdr !== 96) begin
$display("Simulation failed");
$stop;
end
end
end
	// check results, modified for memfile
	always @(negedge clk)
	begin
	if(MemWrite) begin
	if(DataAdr === 248 & WriteData === 7) begin
	$display("Simulation succeeded");
	$stop;
	end else if (DataAdr !== 236) begin
	$display("Simulation failed");
	$stop;
	end
	end
	end

Table 2.

Additionally, the check result logic was modified to match the input memfile. As a result, when the code is executed, the message **"Simulation succeeded"** will be displayed. Please refer to **Table 2**.

(b) Modifying the single-cycle ARM processor by adding the LDRB instructions:

Answer:

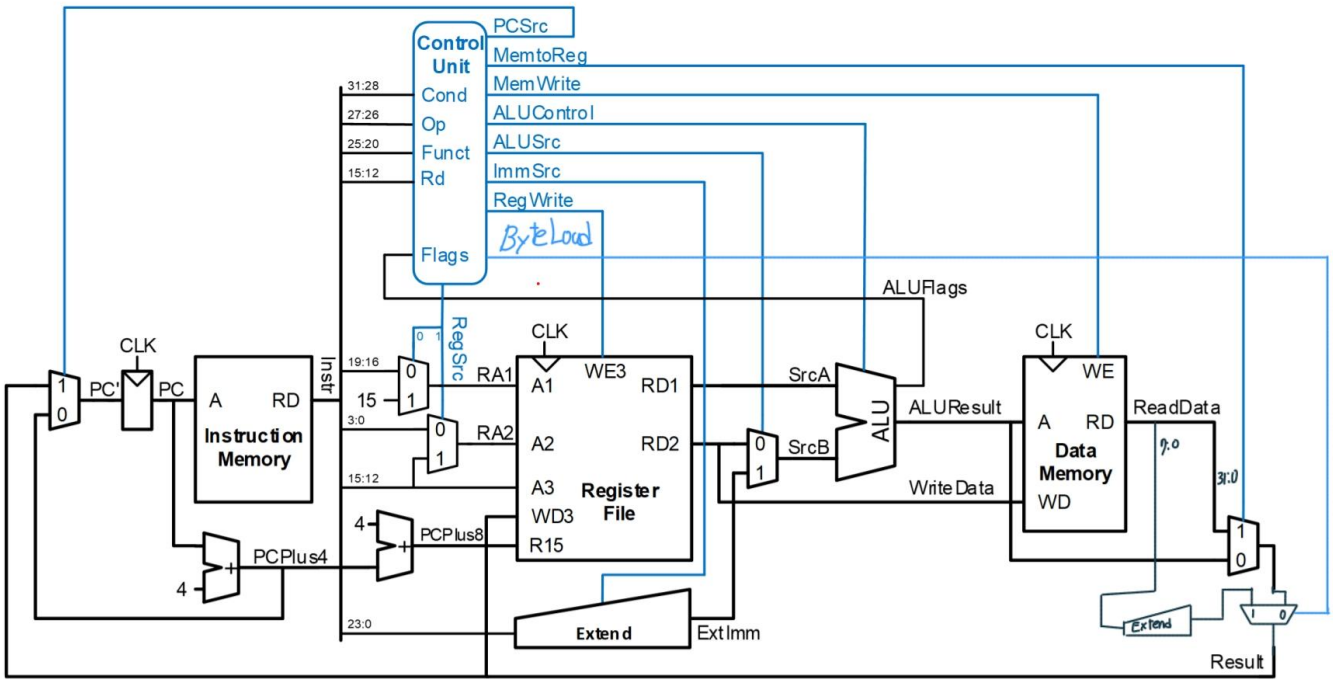

Figure 7.

As shown in Figure 7, 1 byte is extended from **ReadData**, and a new multiplexer is configured to allow selection of **Result** accordingly. In addition, the **Main Decoder** was modified as shown in **Table 3**, and the **ALU Decoder** was updated by replacing the **ORR** operation with the **EOR** operation to support the EOR instruction used in memfile2.

Op	Funct5	Funct0	Funct2	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp	ByteLoad
00	0	x	X	DPReg	0	0	0	0	Xx	1	00	1	0
00	1	X	X	DPImm	0	0	0	1	00	1	X0	1	0
01	X	0	0	STR	0	X	1	1	01	0	10	0	0
01	X	1	0	LDR	0	1	0	1	01	1	X0	0	0
01	X	1	1	LDRB	0	1	0	1	01	1	X0	0	1
10	x	X	x	B	1	0	0	1	10	0	X1	0	0

Table 3.

The code for this implementation was primarily modified within the **datapath** module, with corresponding changes clearly indicated through comments.

(c) Testing the single-cycle ARM processor:

Answer:

In conclusion, a successful result was not achieved due to an unresolved logical error that occurred during the process of loading data from memory. As shown in **Figure 8**, during consecutive **LDRB** operations, the same value is repeatedly loaded, resulting in an infinite loop. In the correct operation, the value **0xFE** should be stored at address **0x80**. The cause of this logical error appears to be that the **DataMemory** block processes data in **word-sized units**. To resolve this issue, it is necessary to implement support for byte-level data processing. It is unfortunate that the assignment concludes at this point, but I plan to fully implement the remaining features in the future.

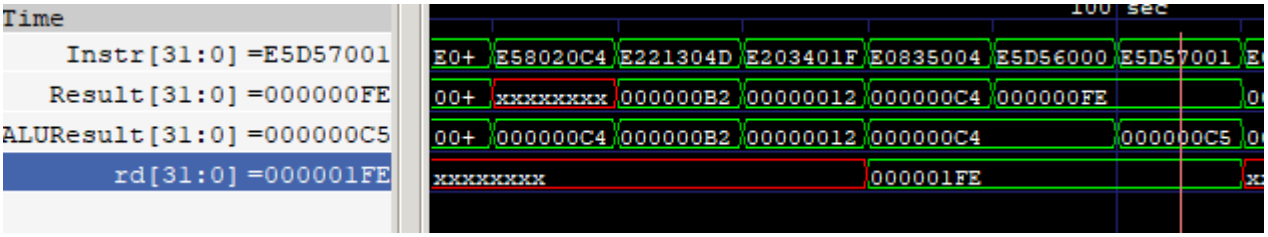

Figure 8.