EE312 Lab Report – Lab 3. Single Cycle CPU

20160030 고은석, 20160680 추헌호

1. Introduction

본 과제에서는 Verilog HDL을 통해 RV32I 기반의 Single Cycle CPU를 구현한다. Single Cycle CPU란 한 CLK 주기 안에서 instruction 하나의 처리를 완료하는 CPU로, 가장 기본적인 형태의 CPU라고 할 수 있다. 이 Single Cycle CPU의 원리를 기반으로 하여 Multi Cycle, Pipeline 등의 발전된 형태가 파생된다. 크게 다섯 가지 stage로 나누어져 있으며 (IF, RF, ALU, MEM, RF) 각 단계는 instruction의 fetch, register file로부터 데이터 읽기, 산술 및 논리 연산, 데이터 메모리에 읽고 쓰기, register file에 데이터 쓰기를 수행한다.

2. Design

Single Cycle CPU의 기본적인 구조는 Figure 1과 같다. 이는 A. Patterson, John L. Hennessy의 Computer Organization and Design - The Hardware Software Interface (RISC-V Edition)에서 제시된 Figure 4.17 (p.257)을 참고하여 설계했다. 상기한 교재에서 제시한 설계는 교육 목적의 가장 단순한 형태이기 때문에 본 과제를 위한 설계인 Figure 1에는 여러 소자가 추가적으로 배치되어 있으며, CTRL 모듈로부터 발생되는 신호의 종류에도 다소 차이가 있다. Instruction에 따른 신호 발생은 Table 1에 기술되어 있다.

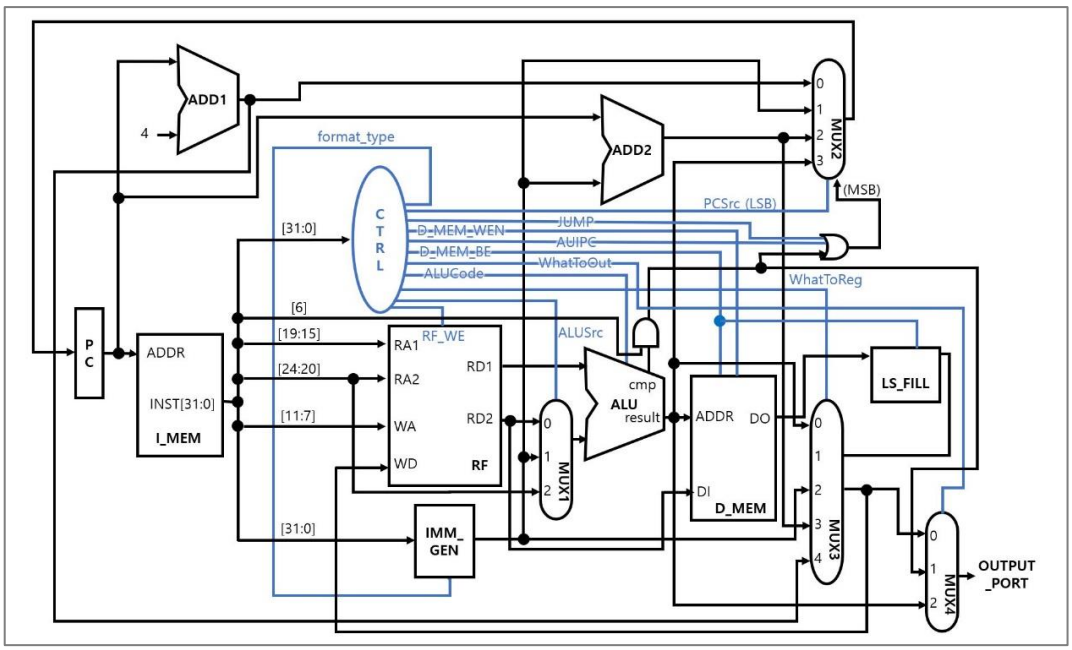

Figure 1. Datapath and Control Unit of Single Cycle CPU

Command	Format Type	RF_WE	PCSrc	WhatToReg	D_MEM_BE	D_MEM_WEN	ALUSrc	ALUCode	WhatTo Out	AUIPC	JUMP
LUI	U	1	0	2	x (set to 4'b1111)	0	x	x	0	0	0
AUIPC	U		x	3						1	
JAL	J		0	4 (PC+4)						0	1
JALR	I		1				1				
BEQ	B	0	0	x			0	4'b0100	1	0	0
BNE	B							4'b0101			
BLT	B							4'b0110			
BGE	B							4'b0111			
BLTU	B							4'b1000			
BGEU	B							4'b1001			
LB	I	1		1	4'b0001		1	4'b0000	0		
LH	I				4'b0011						
LW	I				4'b1111						
LBU	I				4'b1001						
LHU	I				4'b1011						
SB	S	0		x	4'b0001	1			2		
SH	S				4'b0011						
SW	S				4'b1111						
ADDI	I	1	0	0	x (set to 4'b1111)	0	2	4'b0110	0	0	
SLTI	I							4'b1000			
SLTIU	I							4'b1111			
XORI	I							4'b0011			
ORI	I							4'b0010			
ANDI	I							4'b1101			
SLLI	R							4'b1010			
SRLI	R							4'b1011			
SRAI	R							4'b0000			
ADD	R						0	4'b0001			
SUB	R							4'b1101			
SLL	R							4'b0110			
SLT	R							4'b1000			
SLTU	R							4'b1111			
XOR	R							4'b1010			
SRL	R							4'b1011			
SRA	R							4'b0011			
OR	R							4'b0010			
AND	R										

Table 1. Generation of signals at CTRL unit

3. Implementation

1) Template 구성

REG_FILE.v, Mem_Model.v, RISCV_CLKRST.v은 수정하지 않고 주어진 그대로 사용했다. 주어진 파일인 RISCV_TOP.v를 수정하였고, 추가로 ALU_RISCV.v와 CTRL.v를 추가했다. 추가된 파일 각각은 ALU와 CTRL unit 모듈을 담당한다. ALU_RISCV.v는 Lab. 1에서 구현했던 것을 수정하여 사용하였으며, Opcode에 따른 실행 내용은 Table 2에 나와 있다.

2) 모듈 설명

RISCV_TOP에는 Figure 1에서 나타낸 wire들이 모두 구현되어 있다. 그래서 CTRL 모듈, ALU 모듈이 전부 여기에서 사용된다. 추가로 RISCV_TOP.v에는 IMM_GEN과 LS_FILL이라는 간단한 모듈 두 개가 추가로 구현되어 있는데, 너무 간단한 모듈이라 따로 문서를 분리하지 않고

RISCV_TOP.v 안에 포함시켰다. IMM_GEN은 immediate generator라는 뜻으로 기본적인 sign extension을 수행하는 모듈이다. LS_FILL은 load 명령을 수행할 때 D_MEM_BE에 따라 D_MEM의 output에 sign/unsigned extension을 수행하는 모듈이다.

CTRL.v는 fetch된 instruction을 통해 Table 1에 나오는 11가지 신호를 발생시키는 모듈이다. 이 신호에 따라 CPU 내에서 어떤 루트를 따르게 될지 자동으로 결정된다.

ALU_RISCV.v는 ALU를 담당하는 모듈이며, 단 이 때 branch를 위해 compare 기능이 추가되어 있다. 교재의 경우 단순히 두 input의 subtraction을 통해 zero를 ALU의 output으로 두고 이를 branch에 활용하지만, 어차피 ALU 밖에서 구현되어야 할 로직이기 때문에 그냥 ALU 안에 포함시켰다. 그래서 zero를 사용하는 대신, opcode에 따라 comparison condition을 만족할 경우 cmp가 1이 되도록 구현되어 있다.

OP	operation	description
0000	A + B	32-bit addition
0001	A − B	32-bit subtraction
0010	A and B	32-bit and
0011	A or B	32-bit or
0100	A EQ B	Equal to?
0101	A NE B	Not equal to?
0110	A LT B	Lower than?
0111	A GE B	Greater than or equal to?
1000	A LTU B	Lower than? (Unsigned)
1001	A GEU B	Greater than or equal to?(Unsigned)
1010	A >> 1	Logical right shift
1011	A >>> 1	Arithmetic right shift
1100	A[0]A[15:1]	Rotate right
1101	A << 1	Logical left shift
1110	A <<< 1	Arithmetic left shift
1111	A xor B	32-bit xor

Table 2. Action of ALU depending on opcode

4. Evaluation

작성한 코드의 평가는 주어진 3개의 testbench 파일을 통해 하였으며, 각각의 경우에 포함된 모든 테스트를 통과하였다. 이는 Figure 2~4에 나타나 있다.

Figure 2. Result of simulation when 'TB_RISCV_inst.v' was used

(하단에 TV_RISCV_forloop.v라는 문구가 있지만 그건 Modelsim에서의 파일 이름이고, 실제 내용물은 inst입니다. 매번 이름을 바꾸기 번거로워 forloop라는 파일명을 쓰되 안의 코드만 inst, forlooop, sort를 바꿔가며 테스트했습니다.)

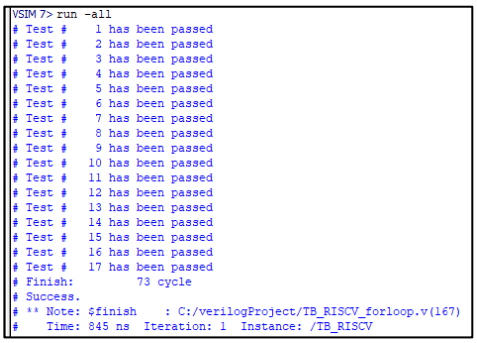

Figure 3. Result of simulation when 'TB_RISCV_forloop.v' was used

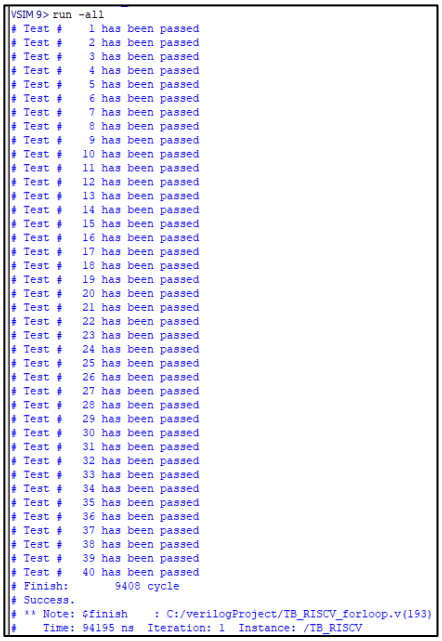

Figure 4. Result of simulation when 'TB_RISCV_sort.v' was used

5. Discussion

본 과제에서 어려웠던 점은 디버깅이었다. 테스트벤치는 테스트케이스의 Pass/Fail이 결정되는 지점에서 모든 요소를 다 판단하지 못하고, 단지 output port로 나오는 값만으로 성공 여부를 출력하기 때문에 우연히도 잘못된 지점부터 몇 개의 테스트는 성공으로 뜨는 경우가 있었다. 그래서 디버깅을 할 때 실패 지점만이 아니라 그 전 몇 개의 instruction도 같이 검토해야 했고, 그래서 더 복잡하게 느껴졌던 것 같다.

6. Conclusion

작성한 코드를 이용하여 확인해본 결과 실패한 testcase가 없었기에 Single Cycle CPU가 적절히 구현되었다고 판단했다. 본 과제를 통해 강의를 통해 배운 Single Cycle CPU의 구조에 대해 확실히 이해할 수 있게 되었고, 앞으로의 발전된 형태의 CPU 구현에도 큰 도움이 되리라 생각한다.