Report: Lab3

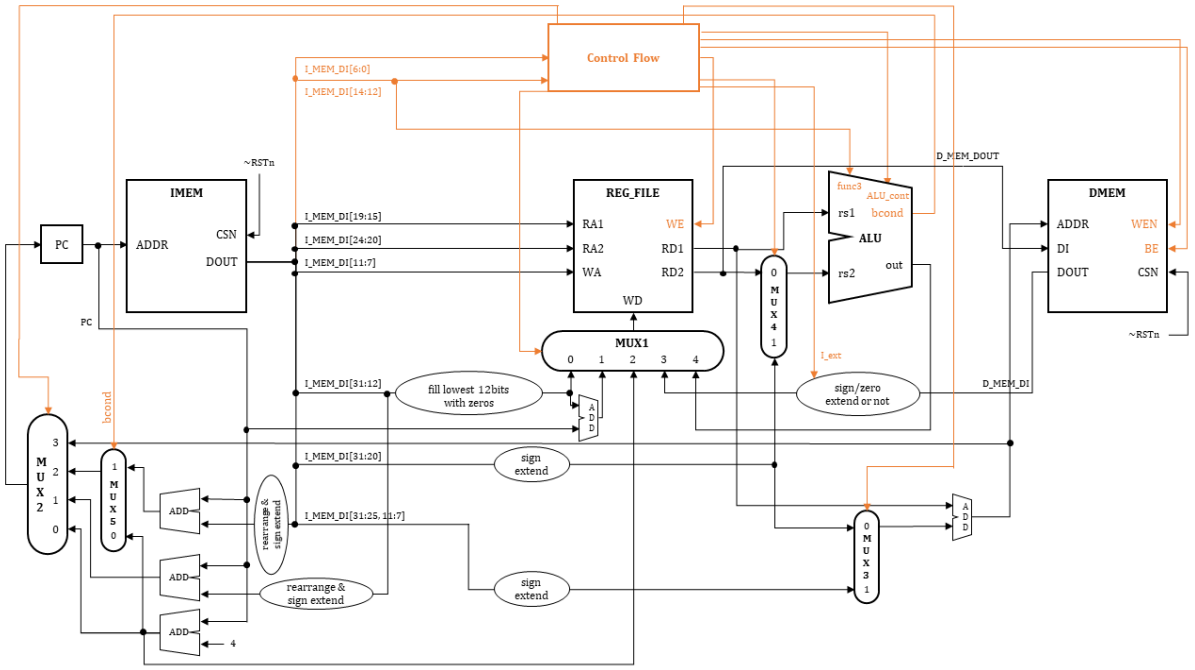
20190533 Hyemin Lee
20190235 Gangtae Park

1. Introduction

Lab3는 verilog를 이용해 RISC-V ISA를 기반으로 하는 Single-Cycle CPU를 제작하는 랩이다. Single-Cycle CPU를 구현하기 위해서는 Instruction Memory, Register File, Data Memory의 역할과 그들을 컨트롤하는 Control Flow와의 관계를 이해하는 것이 중요하다. 이들의 관계는 굉장히 복잡하고 고려해야할 사항들이 많기 때문에 Control & Data Path를 먼저 그리고 이를 기반으로 코드를 작성하였다.

2. Design

가장 먼저 PC가 Instruction Memory에서 instruction을 읽어오면서 instruction fetch(IF)를 실시한다. 읽어온 instruction은 I_MEM_DI에 저장되는데, instruction decode(ID)와 register operand fetch(RF)위해 I_MEM_DI를 여러 부분으로 나누어서 해석한다. Instruction의 type은 크게 6가지(R,I,S,U,B,J-type)로 나뉘는데, 각각의 type마다 data flow가 달라지므로 MUX를 이용해 원하는 역할을 수행하도록 하였다. MUX의 selector bit은 control flow module에서 계산되며, I_MEM_DI의 일부가 입력으로 들어간다. MUX 1은 RF_WD에 넣는 값을 결정하고, MUX 2는 PC값을 업데이트 할 때 이용되며 MUX 3은 D_MEM_ADDR에 들어가는 값을 결정한다. MUX 4는 ALU에 RF_RD2와 immediate 값 중 어느 것이 들어갈 지 결정하고, 마지막으로 MUX 5는 MUX 2와 함께 PC에 관여하는데, ALU에서 입력 받은 branch condition 일치 여부에 따른 다음 PC값을 넘겨준다.



3. Implementation

본 lab에서 구현한 CPU는 single-cycle cpu이기 때문에 한 clock 안에 instruction을 읽어오는 것부터 실행을 완료하고 다음 instruction의 주소값이 무엇인지 알아내는 과정까지 끝내야 한다. 우리는 CPU의 구현을 위해 data flow를 담당하는 RISCV_TOP.v, 'control_flow' module을 이용해서 control flow를 담당하는 control_flow.v, ALU의 작동을 담당하는 ALU.v까지 총 세 파일을 작성하였다.

RISCV_TOP module에는 위 그림에서 검정색으로 표시된 data flow에서 IMEM, REG_FILE, ALU, DMEM 모듈 자체의 구현을 제외한 모든 부분이 구현되어 있다. 즉, control flow에서 넘겨준 신호에 맞게 각 모듈들 사이의 상호작용과 데이터의 가공을 실시한다. RSTn이 1이면 모듈 상호작용에 관여하는 모든 변수를 초기화하고, clock이 negative edge가 될 때마다 PC(I_MEM_ADDR)를 다음 PC 값으로 업데이트 한다. Instruction이 변할 때마다 정해진 규칙에 맞게 IMEM, REG_FILE, ALU, DMEM에 입력할 값을 변경하고 각 모듈에서 나온 output값을 이용해서 instruction의 실행을 마무리하고 다음 PC를 계산한다.

Control flow module에서는 RISCV_TOP module에서 opcode(I_MEM_DI[6:0])와 func3(I_MEM_DI[14:12])을 입력 받아서 어떤 type의 instruction인지 판단하고, 이에 맞게 RF_WE, D_MEM_WEN, D_MEM_BE, I_ext, 그리고 각각의 MUX들의 selector값을 다시 output으로 내보낸다. 또한, ALU가 어떤 operation을 수행할지 결정해주는 ALU_cont를 ALU module로 전달한다. 아래 표는 각각의 opcode에 따라 control flow module에서 내보내는 output들을 정리한 것이다.

instruction	opcode	Semantics	RF_WE	D_MEM_WEN	D_MEM_BE	I_ext	MUX			
							1	2	3	4
LUI	0110111	GPR[rd]←[imm, zeros]	1	1	0	0	0	0	0	0
AUIPC	0010111	GPR[rd]←PC+[imm, zeros]	1	1	0	0	1	0	0	0
JAL	1101111	target←sign_extend(imm) GPR[rd]←PC+4 PC←PC+target	1	1	0	0	2	1	0	0
JALR	1100111	target←GPR[rs1]+sign_extend(imm) GPR[rd]←PC+4 PC←target	1	1	0	0	2	3	0	0
B-types	1100011	target←sign_extend(imm) if (bcond) PC←target	0	1	0	0	6	2	0	0

		else PC←PC+4								
I-type Loads	0000011	effective_address←sign_extend(imm)+GPR[rs1] GPR[rd]←MEM[translate(effective_address)] PC←PC+4	1	1	LB,LBU: 1 LH, LHU: 3 LW: 15	LW: 0 LB,LH: 1 LBU, LHU: 2	3	0	0	0
S-types	0100011	effective_address←sign_extend(imm)+GPR[rs1] MEM[translate(effective_address)]←GPR[rs2] PC←PC+4	0	0	SB: 1 SH: 3 SW: 15	0	5	0	1	1
I-types	0010011	GPR[rd]←GPR[rs1] OP sign_extend(imm) PC←PC+4	1	1	0	0	4	0	0	1
R-types	0110011	GPR[rd]←GPR[rs1] OP GPR[rs2] PC←PC+4	1	1	0	0	4	0	0	0
MULT, MODULO, IS_EVEN	0001011	GPR[rd]←GPR[rs1] OP GPR[rs2] PC←PC+4	1	1	0	0	4	0	0	0

ALU module은 lab1에서 구현했던 ALU에서 일부분만 변경하여 활용하였다. RISCV_TOP module에서 ALU operand인 rs1과 rs2를 입력받는다. Operator는 RISCV_TOP module에서 받은 func3(I_MEM_DI[14:12])와 control flow module에서 받은 ALU_cont를 조합해서 결정된다. 연산 결과인 ALU_out과 bcond(conditional jump condition 부합 여부)는 다시 RISVC_TOP module로 반환한다.

```
//MULT
ALU_out = rs1 * rs2
//MODULO
ALU_out = rs1 % rs2
//IS_EVEN
if (rs1[0] == 0) ALU_out = 1
else ALU_out = 0
//B-types
case (func3)
       func3: begin
              if (statement) bcond = 1
```

```
            else bcond = 0
end
//SLT, SLTU
if (rs1 < rs2) ALU_out = 1
else ALU_out = 0
```

4. Evaluation

Modelsim을 통해서 TB_RISCV_inst.v, TB_RISCV_forloop.v, TB_RISCV_sort.v의 testbench들을 시뮬레이션하였고 모든 testcase(각각 26개, 17개, 40개)를 모두 통과하였다.

5. Discussion

코딩하기에 앞서 CPU의 디자인을 구상하는 것부터 많은 노력이 필요했다. 수업시간에 배운 CPU 구조는 MIPS에 맞는 디자인이었기 때문에 RISC-V에 맞는 디자인으로 그림을 모두 다시 그려야 했다. 이를 위해 구현해야 하는 모든 instruction의 동작을 PPT 자료와 메뉴얼을 읽으면서 이해했고 이를 실제로 구현하기 위한 data flow와 control flow를 구상하기까지 약 4시간이 소요되었다.

CPU 디자인을 마쳤을 때, 이제 이 그림을 코드로 옮기기만 하면 되기 때문에 빨리 끝날 것이라 생각했다. 하지만 분명 디자인을 완료한 후였지만, 아무런 오류없이 모든 testbench를 통과하는 코드를 짜기까지 약 18시간이 소요되었다. 대부분의 시간이 디버깅을 하는 데에 사용되었는데, 여기에는 verilog 언어에 익숙하지 않은 탓에 생긴 문제도 있었고 메모리 구동 방식을 완벽히 이해하지 못해서 생긴 문제도 있었다. 그리고 분명 그림을 그대로 코드로 옮겼다고 생각했지만, 빠트린 부분들이 많았고 그 부분을 하나하나 찾아내는 데에 정말 많은 시간이 소요되었다.

많은 문제에 직면했지만, 한 가지 예시를 들자면 다음과 같다. RF_WD에 값을 선언하는 것에 어려움이 있었다. RF_WD가 wire로 지정되어 있었기 때문에 always 내에서 assign을 할 수가 없었고, 또한 always 밖에서도 case별로 나누어서 assign을 하기가 어려워서 새로운 함수로 지정해서 함수값을 바꿔주는 등의 방법을 시도했다. 최종적으로는 RF_WD_reg라는 register를 새로 만들어서 이 문제를 해결 할 수 있었다.

6. Conclusion

Lab3는 verilog를 이용해서 RISC-V 기반의 Single-Cycle CPU를 구현하는 랩이었다. 이번 Lab을 통해서 RISC-V에 어떤 instruction들이 있고, 어떻게 작동하는지 완벽히 이해할 수 있었다. Instruction Memory, Register File, 그리고 Data Memory 등의 module이 어떻게 작동하고 어떻게 상호작용하는지도 더욱 잘 이해할 수 있었고, 이 모든 data의 흐름을 결정하는 Control Flow의 구현 방법까지도 숙지할 수 있었다.