컴퓨터구조 (CSED311)

Lab 4-1 Report

20200220 오상윤

(1) Introduction

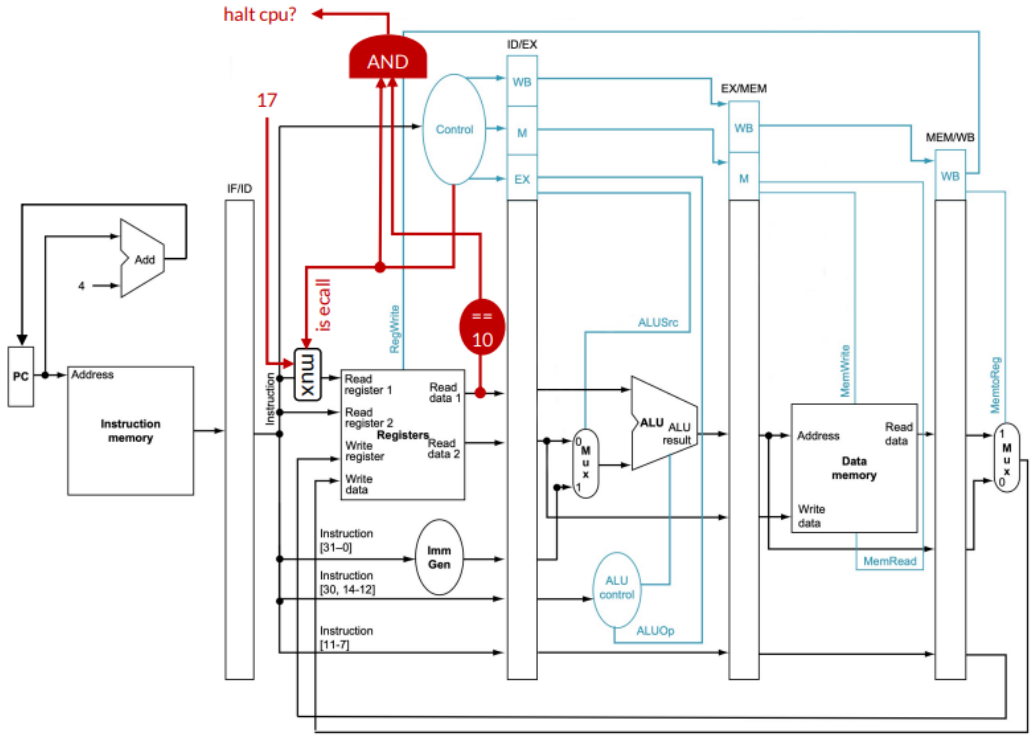
베릴로그를 이용하여 5 stage pipelined cpu를 구현하는 lab이었다. pipelined cpu는 cpu에서 한 번에 한 instrunction을 처리했던 single cycle cpu, multi cycle cpu와 다르게 처리 단계를 5 stage로 나누어 여러 개의 instruction을 stage 별로 parallel 하게 처리하여 resource를 보다 효율적으로 사용함과 동시에 throughput을 크게 증가시킬 수 있다.

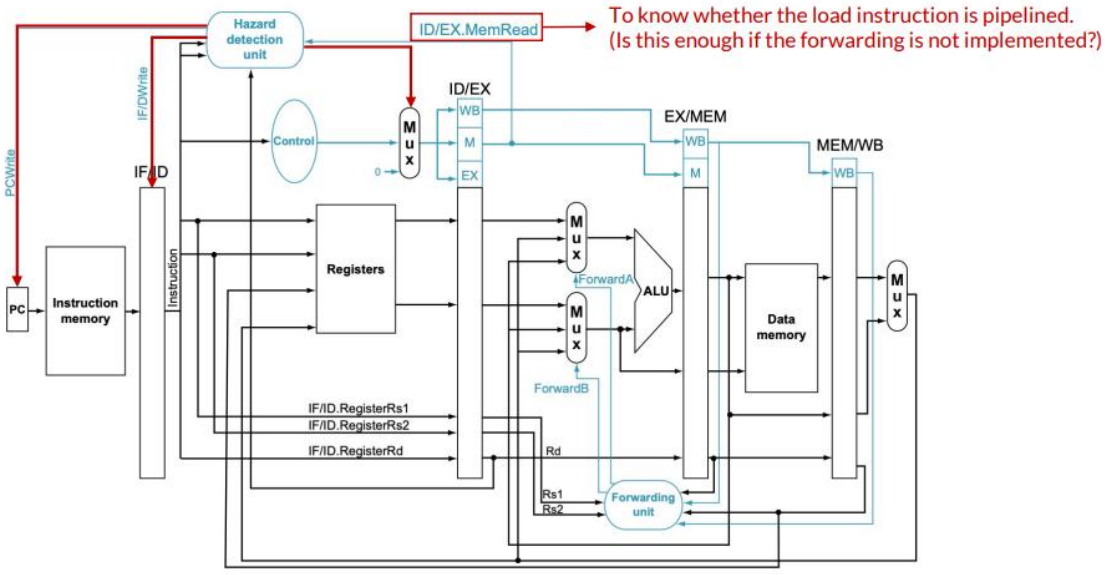
주어진 cpu module은 지난 lab의 single cycle cpu, multi cycle cpu와 마찬가지로 input으로 reset과 clk를, output으로 machine의 중지를 나타내는 is_halted를 갖는다. clk의 positive edge가 한 cycle을 나타내며, cpu module은 ECALL instruction을 만났을 때 x17 레지스터의 값이 10인 경우 is_halted의 값을 1로 출력하여 machine의 중지를 알린다.

이번 Lab 4-1에서는 non-control flow instruction 만을 다루었으며 따라서 처리 가능한 instruction은 R-type, I-type(JALR 제외), S-type의 instruction이며, LOAD instruction은 LW, STORE instruction은 SW만을 처리 가능하다. 파일 구성은 cpu module을 담고 있는 cpu.v, PC module, Adder module을 담고 있는 pc.v, ControlUnit module을 담고 있는 ControlUnit.v, ImmediateGenerator module을 담고 있는 ImmediateGenerator.v, ALUControlUnit module, ALU module을 담고 있는 ALU.v, Mux module, MuxForIsEcall module, MuxForForward module을 담고 있는 mux.v, HazardDetectionUnit module을 담고 있는 HazardDetectionUnit.v, ForwardingUnit module, ForwardingMuxControlUnit module을 담고 있는 ForwardingUnit.v, constant들의 define을 담고 있는 opcodes.v로 구성되어 있다.

(2) Design

Lab4-1 pdf 에 나와 있는 Datapath Design을 토대로 하여 전체적인 module을 Design 하였다.





우선 cpu는 각 stage에서 다른 instruction을 처리하기 위해 ID stage에서 사용하기 위한 값들을 담아두기 위한 IF/ID registers, EX stage에서 사용하기 위한 값들을 담아두기 위한 ID/EX registers, MEM stage에서 사용하기 위한 값들을 담아두기 위한 EX/MEM registers, WB stage에서 사용하기 위한 값들을 담아두기 위한 MEM/WB registers를 갖는다. 각 register들은 clock cycle 마다 update 되며, Control Unit으로부터 만들어진 signal과 rs1, rs2, rd 등 이후 stage들에서 필요한 값들이 이 register들을 통해 전달되고 저장된다.

cpu 내부의 각 module에 대한 설명은 아래와 같다.

1. PC module

   매 cycle마다 pc 값을 변경하는 module이다. 초기의 default pc값은 0x0으로 설정하며, 이후 PCwrite signal을 받을 때마다 logic을 통해 새롭게 계산된 새로운 pc 값 next_pc 를 current_pc 값으로 넣어준다. cpu는 current_pc를 받아 InstMemory module에 input으로 전달하여 해당 line의 instruction을 얻어낸다.

2. Adder module

   next_pc 값을 계산하기 위해 사용되는 module이다. non-control flow의 구현이므로 입력값으로 항상 current_pc, 4를 받아 더해서 output으로 next_pc 값을 내보낸다.

3. ControlUnit module

   opcode를 input으로 받은 다음 해당 opcode에 따라 control signal을 적절하게 설정한다. opcode가 LOAD인 경우 mem_read와 mem_to_reg를, STORE인 경우 mem_write를, I-type또는 S-type인 경우 alu_src를, S-type도 아니고 B-type도 아닌 경우 reg_write을, ECALL인 경우 is_ecall을 1로 설정하는 combinational logic이다. cpu는 각각의 signal들을 다른 module들과 ID/EX registers에 적절하게 연결하여 signal에 따라 올바르게 동작하도록 한다.

4. ImmediateGenerator module

   instruction에서 opcode를 뽑아내 각 opcode에 따라 적절하게 immediate value를 생성하는 module이다. I-type, S-type, B-type, J-type에 따라 immediate value의 생성 방법이 다르며, 구체적인 생성은 아래 implemetation에서 설명할 것이다. cpu는 생성된 immediate value를 이용해 ID/EX registers 값을 update하고 이후 alu의 소스로 적절히 이용한다.

5. ALUControlUnit module

   ALUOp를 input으로 받고, 현재 instruction에서 opcode, func3, func7을 뽑아내 ALU가 수행할 operator를 적절하게 지정한다. ALUOp가 00인 경우는 ADD를, 01인 경우는 SUB를 지정하고, 이외의 경우(10인 경우)에는 func3, func7의 값을 통해 operator을 지정한다. ALUControlUnit module이 alu_op를 통해 operator를 지정하면, ALU module이 이를 input으로 받아 해당하는 연산을 수행한다.

6. ALU module

   ALUControlUnit으로부터 alu_op를 받아 input들에 대해 적절한 연산을 수행한 다음, 연산 결과를 alu_result에 담아 output으로 내보낸다.

7. Mux module, MuxForIsEcall module

   1-bit signal에 따라 두 개의 input 값 중 하나의 값을 output으로 전달한다.

8. MuxForForward module

   2-bit signal에 따라 3개의 input 값 중 하나의 값을 output으로 전달한다.

9. HazardDetectionUnit module

   각 stage의 register 번호를 적절하게 비교하여 필요한 상황에서 hazard를 발생시키는 module이다. 이번 lab에서 data forwarding까지 구현하였기 때문에 load instruction의 바로 다음 instruction이 load instruction의 destination register의 값을 참조하는 경우에서 1 stall 이 걸리도록 설계하였으며, 또한 ECALL instruction의 경우 바로 이전 instruction destination register가 x17이거나 이전의 이전 instruction이 load instruction이면서 destination register가 x17인 경우 hazard signal을 발생시키도록 하였다. HazardDetectionUnit module이 hazard signal을 발생시키면 cpu는 PC module과 IF/ID registers, 그리고 ControlUnit module에 적절한 signal을 전달하여 stall이 이루어지도록 한다.

10. ForwardingUnit module, ForwardingMuxControlUnit module

    각 stage의 register 번호를 적절하게 비교하여 필요한 상황에서 Data를 forwarding 할 수 있도록 신호를 발생시키는 module들이다. dist=1,2에 해당하는 instruction 간의 data forwarding(특정 instruction이 바로 이전 또는 이전의 이전 instruction의 destination register의 값을 참조하는 경우)은 ForwardingUnit module이 처리하며, dist=3에 해당하는 instruction 간의 data forwarding(특정 instruction이 이전 3번째 instruction의 destination register의 값을 참조하는 경우)은 ForwardingMuxControlUnit module이 처리하도록 하였다. 또한 추가적으로 ECALL instruction의 바로 이전 instruction이 x17에 data를 write하는 경우 ECALL instruction이 ID stage일 때 EX/MEM stage로부터 data를 forwarding 해와야하므로 이 역시 ForwardingMuxControlUnit module이 처리하도록 하였다. ForwardingUnit module과 ForwardingMuxControlUnit module이 data forwarding이 필요할 때 적절하게 신호를 발생시키면 cpu에서 해당 신호에 따라 mux를 통해 signal에 해당하는 data가 forwarding 되도록 한다.

(3) Implementation

1. PC module

input으로 reset, clk, next_pc, pc_write 를 받으며, output으로 current_pc를 갖는다. clock synchronous 하게 동작하며 clk의 positive edge마다 pc_write 값이 1인 경우 current_pc의 값을 next_pc로 바꾸어 준다. default PC의 값이 0x0이므로 reset이 1인 경우, pc를 32'b0으로 초기화하도록 하였다.

2. Adder module

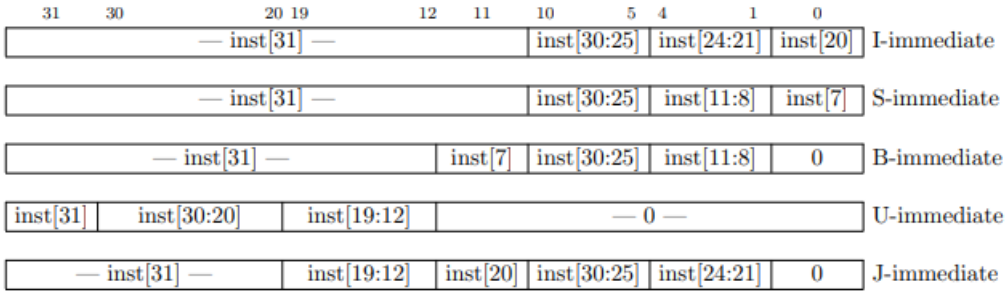
input으로 input1, input2를, output으로 output_adder를 갖는다. input1과 input2의 값을 더하여 output_adder에 asynchrounous하게 넣어주는 combinational logic이다.

3. ControlUnit module

input으로 part_of_inst을 받으며, output으로 mem_read, mem_to_reg, mem_write, alu_src, reg_write, alu_op, is_ecall 을 갖는다. cpu로부터 받는 part_of_inst는 instruction의 하위 비트 7개로 opcode를 나타내며, opcode에 따라 combinational logic을 통해 asynchronous하게 control signal들을 적절하게 바꿔 output으로 내보낸다.

4. ImmediateGenerator module

input으로 part_of_inst, output으로 imm_gen_out을 갖는다. cpu로부터 받는 part_of_inst는 32 bit의 instruction이며 하위 비트 7개로 opcode를 뽑아내 opcode에 따라 asynchronous하게 instruction으로부터 immediate value를 적절하게 생성하여 imm_gen_out으로 내보낸다. opcode를 통해 type을 얻어낸 다음 각 type에 따라 immediate value의 생성방법은 아래의 riscv-spec 자료의 그림을 따른다.



5. ALUControlUnit module

input으로 part_of_inst, ALUOp, output으로 alu_op를 갖는다. cpu로부터 받는 part_of_inst는 32 bit의 instruction이며, instruction에서 opcode, func3, func7을 뽑아내 ALUOp값과 func7, func3를 고려하여 asynchronous하게 alu에게 전달할 operator인 alu_op를 적절하게 지정하여 내보낸다.

6. ALU module

input으로 alu_op, alu_in_1, alu_in_2를 받으며, output으로 alu_result 를 갖는다.

asynchronous하게 동작하는 combinational logic이며, alu_op에 따라 operator에 맞게 alu_in_1과 alu_in_2간의 연산을 수행하여 alu_result로 내보낸다.

7. Mux module, MuxForIsEcall module

   input으로 input0, input1, signal을 받으며, output으로 output_mux를 갖는다. asynchronous하게 동작하는 combinational logic이며, signal이 0인 경우 input0을, 1인 경우 input1을 output_mux의 값으로 지정한다. 두 module은 input0, input1, output_mux의 bit 수만 차이가 있으며 구조는 동일하다.

8. MuxForForward module

   input으로 input00, input01, input10, signal을 받으며, output으로 output_mux를 갖는다. asynchronous하게 동작하는 combinational logic이며, signal이 00인 경우 input00을, 01인 경우 input01을, 10인 경우 input 10을 output_mux의 값으로 지정한다.

9. HazardDetectionUnit module

   input으로 rs1, rs2, id_ex_rd, id_ex_mem_read, id_ex_opcode, ex_mem_mem_read, ex_mem_rd, is_ecall을 받으며 output으로 is_hazard를 갖는다. 바로 MEM stage의 instruction이 load instruction이면서 해당 instruction의 rd register를 EX stage의 instruction이 참조하는 경우 hazard를 발생시키기 위해, rs1과 id_ex_rd가 같거나 rs2와 id_ex_rd가 같으면서 id_ex_mem_read가 1일 때 is_hazard를 asynchronous하게 1로 설정한다. 또한 ECALL instruction을 위한 hazard를 발생시키기 위해 is_ecall이 1일 때, 즉 ID stage에 ECALL instruction이 위치할 때 id_ex_rd가 17이고 id_ex_opcode가 register에 write back 하는 type instruction의 opcode 인 경우 is_hazard를 asynchronous하게 1로 설정하며, ex_mem_mem_read가 1이고 ex_mem_rd가 17일 때 역시 is_hazard를 asynchronous하게 1로 설정한다.

10. ForwardingUnit module

    input으로 id_ex_rs1, id_ex_rs2, ex_mem_rd, ex_mem_reg_write, mem_wb_rd, mem_wb_reg_write를 받으며, output으로 forward_A, forward_B를 갖는다. id_ex_rs1, ex_mem_rd, mem_wb_rd, ex_mem_reg_write, mem_wb_reg_write의 값을 적절하게 비교하여 rs1 register에 대해 data forwarding이 필요한지 판단하며, id_ex_rs2, ex_mem_rd, mem_wb_rd, ex_mem_reg_write, mem_wb_reg_write의 값을 적절하게 비교하여 rs2 register에 대해 data forwarding이 필요한지 판단하여 각 rs1과 rs2에 대해 EX/MEM register로부터 forwarding이 필요한 경우 forward_A 또는 forward_B의 값을 01로 asynchronous하게 바꿔주고, MEM/WB register로부터 forwarding이 필요한 경우 forward_A 또는 forward_B의 값을 10으로 asynchronous하게 바꿔준다.

11. ForwardingMuxControlUnit module

input으로 rs1, rs2, rd, ex_mem_rd, is_ecall을 받으며 output으로 mux_rs1_dout, mux_rs2_dout을 갖는다. rs1과 rd가 같거나 rs2와 rd가 같은 경우 각각 mux_rs1_dout과 mux_rs2_dout의 값을 asynchronous하게 0으로 바꿔주고 아닌 경우 1로 바꿔준다. 또한 추가적으로 ecall instruction 바로 이전 instruction에서 x17의 값을 write하는 경우를 찾아내기 위해 ex_mem_rd가 17이고 is_ecall 이 1일 때 mux_rs1_dout을 10으로 바꿔준다.

(4) Discussion

Data Forwarding 을 구현하는 과정에서, 처음 구현을 시도하였을 때는 rs1 과 rs2 에 대해 동시에 forwarding 이 일어나는 경우를 생각하지 못하였다. 때문에 주어진 testbench file 을 돌렸을 때 동시에 forwarding 이 필요한 상황에서 rs1 에 대해서만 forwarding 이 일어나 원하는 결과를 얻지 못하였다. rs1, rs2 에 대해 각각 forwarding 이 필요한지를 확인하도록 바꿔주어 문제를 해결할 수 있었다. 또한 is_hazard signal 이 1 이 되어 stall 이 일어났을 때, ID/EX register 들의 값을 모두 0 으로 설정하여 아무 instruction 이 들어가지 않은 것처럼 동작하도록 구현하였는데, 이 때문에 모두 0 으로 설정한 값들이 EX/MEM register 와 MEM/WB register 에 도착하였을 때 EX_MEM_rd 와 MEM_WB_rd 값 또한 각각 0 으로 설정되어 있어, 0 번 register 를 참조하는 instruction 에 대해 data forwarding 이 일어나면서 0 번 register 의 값이 들어가야 할 자리에 0 이 아닌 값이 forwarding 되는 문제가 있었다. 이는 EX_MEM_rd 와 MEM_WB_rd 값이 0 이 아닌 경우에만 forwarding 이 일어나도록 수정하여 문제를 해결할 수 있었다.

주어진 Non-control flow input file 을 pipelined cpu 와 single-cycle cpu 에서 각각 돌려보았다. pipelined cpu 는 59 cycles, single-cycle cpu 에서는 51 cycles 를 확인할 수 있었다. single-cycle cpu 의 1 cycle 은 한 instruction, 즉 5 개의 stage 를 모두 돌기 위한 1 cycle 이지만, pipelined cpu 의 1 cycle 은 1 개의 stage 를 돌기 위한 1 cycle 이므로, 주어진 Non-control flow input file 을 기준으로 pipelined cpu 가 single-cycle cpu 보다 약 5 배의 성능 향상을 이루었다고 볼 수 있다.

(5) Conclusion

이번 Lab 을 통하여 주어진 skeleton code 에서 적절하게 code 를 디자인하여 pipelined CPU 를 구현해보면서 pipelined cpu 의 작동 방식으로 익히고 내부에서 data 가 어떻게 흐르는지를 익힐 수 있었다. 또한 직접 주어진 testbench file 를 통해 single-cycle cpu 와 cycle 수를 비교해봄으로써 성능의 향상 또한 확인해볼 수 있었다.