2025 Spring

CSED311

컴퓨터구조

Lab 4-1 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

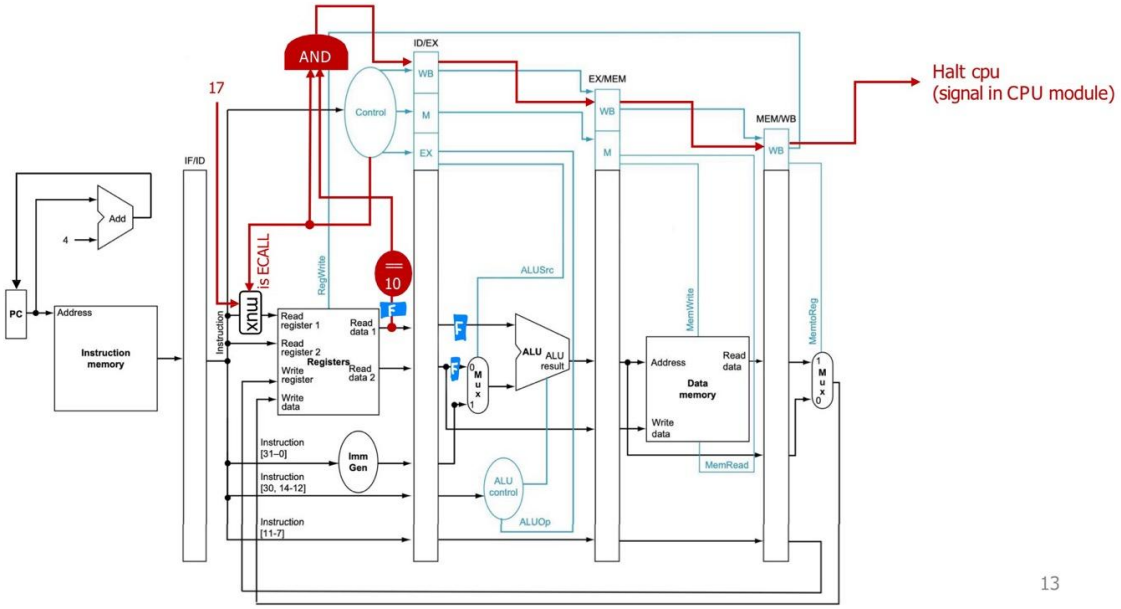
I. 서론

이번 Lab 에서는 pipelined CPU 를 구현하였다. Pipelined CPU 는 single-cycle CPU 를 몇 개의 단계로 쪼개어 (우리 디자인의 경우 5 단계), 한 cycle 에 여러 개의 instruction 을 동시에 처리할 수 있도록 업그레이드한 디자인이라고 볼 수 있다. Pipelined CPU 는 single-cycle CPU 보다 throughput 이 좋지만, 하나의 instruction 만을 집중했을 때 instruction 의 시작부터 끝까지 걸리는 시간(latency)도 같이 증가한다.

II. 디자인

A. 전체적인 Pipelined CPU의 구현

CPU 디자인은 Lab4-1 ppt의 13쪽에 있는 디자인을 참고했다. 밑의 그림에서 Stall과 data forwarding의 logic 정도를 추가했다. 밑에 보이는 세 개의 파란 "F" 블록들이 data forwarding이 일어난 부분이라고 보면 된다. (자세한 내용은 **II-B. Data Forwarding의 디자인**에서 다루겠다.)



- 서론에서 말한 바와 같이, pipelined CPU는 single-cycle CPU를 5단계(IF, ID, EX, MEM, WB)로 나누어, 각 단계에 다른 instruction을 처리할 수 있도록 설계되어 있다.
  - (stall을 고려하지 않는다면) Instruction은 positive clock edge을 만날 때마다 다음 단계로 넘어갈 수 있다.
  - 이 때 instruction의 처리에 필요한 data들은 각 단계 사이의 latch들을 이용해 넘겨줄 수 있다.
  - 이렇게 함으로써 한 cycle에 최대 5개의 instruction을 겹쳐 실행시킴으로써 throughput을 증가시킬 수 있다.

- 그러나, 동시에 실행되는 instruction들 간에 서로 dependency가 있는 경우 문제가 생길 수 있다. 가령, 현재 ID 단계에서 읽어야 하는 register의 정보가 아직 register file로 writeback되지 않았을 수 있다. (register을 update하는 instruction이 아직 EX나 MEM 단계에 머물러 있는 경우) 이를 RAW (Read After Write) dependency라고 부른다.
    - 이를 해결하기 위한 가장 기초적인 방법은 단순히 register file이 업데이트될 때까지 기다리는 것이다. 이를 Stalling이라고 부른다. (**II-C. Stall Detection의 디자인**)
    - 이에 더해 data forwarding의 방법을 이용해 stall cycle을 최소화할 수 있다.
- 우리의 pipeline CPU 디자인은 data forwarding과 stalling을 사용해 최대한 적은 수의 cycle로 돌아갈 수 있도록 구현하였다.

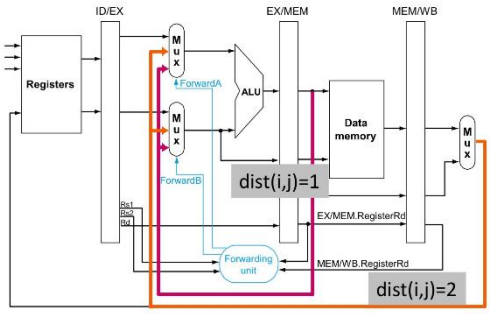
B. Data Forwarding의 디자인

현재의 문제 상황: ID단계에서 read하는 register이 제대로 업데이트되지 않은 경우, instruction 사이에 최대 2사이클을 기다려야 해야 할 수도 있다.

- writeback을 위한 data는 실제 writeback이 이루어지기 전에 생성됨을 알 수 있다.
    - ALU 결과의 경우 EX단계에서 생성된다. (EX/MEM latch에서 활용 가능)
    - load instruction의 경우 MEM단계에서 생성된다. (MEM/WB latch에서 활용 가능)
- Register file의 값을 활용해야 하는 경우, ECALL을 제외하면 EX 스테이지에서, ECALL의 경우 ID 스테이지에서 읽는 것이 가능하다.

위의 모든 내용을 정리하면 다음과 같다.

	R/I-Type	LW	SW	ECALL
IF				
ID				use
EX	use produce	use	use	
MEM		produce		
WB				



따라서 EX 스테이지의 ALU 앞의 두 군데, ECALL을 위한 halt 판별 회로도에서 한 군데 forwarding할 수 있으며, 각각 MEM, WB에서 당겨오거나 그냥 register file에서 읽어오는 것을 mux와 forwarding unit을 통해 결정할 수 있다. 옆의 그림은 ALU 앞의 forwarding logic에 대한 그림이다. (ECALL에 의한 것도 비슷한 논리로 만들 수 있다.)

구체적으로, 현재 read해야 하는 register이 뒷 스테이지가 write해야 하는 register과 일치할 때, register이 일치하는 instruction들 중 가장 늦는 instruction을 골라 그 stage에서의 data를 forwarding 한다. 이 때 "값이 아직 준비되지 않은 경우"는 없도록 stall을 적절히 해주어야 한다.

C. Stall Detection의 디자인

앞선 표에서 "일어나면 안 되는" 상황이 몇 개 있다.

	R/I-Type	LW	SW	ECALL
IF				
ID				use
EX	use produce	use	use	
MEM		produce		
WB				

- 한 사이클 내에서, EX stage 에서 register 값을 활용해야 하는데 MEM stage 가 load instruction 이어서 그 register 에 아직 load 되지 않은 경우
- 한 사이클 내에서, ID stage 에서 register 값을 활용해야 하는데 EX stage 에서 그 값을 아직 계산하지 못했거나, MEM stage 가 load instruction 이어서 그 register 에 아직 load 되지 않은 경우

이런 경우들에 대해서는 값이 제대로 준비되기 전에 읽기를 시도하므로 stall 이 필요하다. EX – MEM 이나 ID – EX 의 경우 1 회, ID – MEM 의 경우 2 회의 stall 을 하면 된다.

III. 구현

A. 전체적인 Pipelined CPU의 구현

- 이것의 경우 II-A에 있는 회로도를 거의 그대로 코드로 옮긴 것이라 크게 신경쓸 부분은 없다.

```
// Update EX/MEM pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    // Control values
    EX_MEM_mem_write <= 1'b0;
    EX_MEM_mem_read <= 1'b0;
    EX_MEM_mem_to_reg <= 1'b0;
    EX_MEM_reg_write <= 1'b0;
    EX_MEM_is_halted <= 1'b0;
    // Non-control values
    EX_MEM_alu_out <= 32'b0;
    EX_MEM_dmem_data <= 32'b0;
    EX_MEM_rd <= 5'b0;
  end
  else begin
    // Control values
    EX_MEM_mem_write <= ID_EX_mem_write;
    EX_MEM_mem_read <= ID_EX_mem_read;
    EX_MEM_mem_to_reg <= ID_EX_mem_to_reg;
    EX_MEM_reg_write <= ID_EX_reg_write;
    EX_MEM_is_halted <= ID_EX_is_halted;
    // Non-control values
    EX_MEM_alu_out <= alu_result;
    EX_MEM_dmem_data <= alu_forward_data_2;
    EX_MEM_rd <= ID_EX_rd;
  end
end
```

- 한가지 주목할 만한 부분은, single-cycle CPU와 비교해서 stage 사이를 구분짓는 latch들이 추가되었다는 것이다.
- 왼쪽은 이 latch의 예시로, EX와 MEM stage 사이의 latch들을 나타낸 코드이다.

B. Data Forwarding의 구현

Data forwarding과 관련한 모든 로직은 **`ForwardingUnit`** 안으로 modularization을 했다.

아래는 **`ForwardingUnit.v`** 내부에서 rs1의 forwarding 종류를 판별하는 식이다. rs2와 ecall의 신호도 밑과 거의 동일하게 짤 수 있다. MEM이 WB보다 먼저 검사된다는 사실은 "제일 늦는" instruction을 고르는 과정이므로 중요하다.

```
if (EX_rs1 != 5'b00000 && EX_rs1 == MEM_rd && MEM_reg_write) begin
    forward_rs1 = 2'b10;
end
else if (EX_rs1 != 5'b00000 && EX_rs1 == WB_rd && WB_reg_write) begin
    forward_rs1 = 2'b01;
end
else begin
    forward_rs1 = 2'b00;
end
```

rs1이 ex단계에서 쓰이는지의 여부를 확인하지 않아도 되는 이유는, 쓰이지 않는다면 어차피 mux로 골라지는 데이터는 그냥 무시되어 버리는 값이기 때문이다.

아래는 cpu.v 내부의 mux unit이다. rs2나 ecall의 forwarding도 아래와 거의 유사하게 선택된다.

```
assign alu_forward_data_1 = forward_rs1[1] ? EX_MEM_alu_out :
                            forward_rs1[0] ? writeback_data :
                                             ID_EX_rs1_data;
```

C. Stall Detection의 구현

우선 stall 을 해야 하는 경우 제어해야 할 것들에는 다음과 같다.

- PC & IR disable
- Stall 한 지점의 write control 0 으로 세팅하기

위의 두 로직은 다음 코드에서 나타난다.

```
always @(posedge clk) begin
    if (reset) begin
        current_pc <= 0;
    end
    else if (!is_stall) begin
        current_pc <= next_pc;
    end
end
```

> PC.v 에서. (PC update disable)

```
// Update IF/ID pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    IF_ID_inst <= 32'b0;
  end
  else if (!is_stall) begin
    IF_ID_inst <= instruction;
  end
end
```

>IF/ID latch update 에서. (IR latch disable)

```
if (is_stall) begin
  ID_EX_mem_write <= 1'b0;
  ID_EX_reg_write <= 1'b0;
end
else begin
  ID_EX_mem_write <= mem_write;
  ID_EX_reg_write <= reg_write;
end
```

> ID//EX latch update 에서. (write control 0)

이제 stall 을 언제 해야 하는지에 대해 정리하자. 이것에 대한 모든 logic 은 **`StallDetection.v`** 에 모아두었다.

register 의 data 를 read 할 때즈음에만 그 distance 가 벌어지면 되는 것이라, ID 스테이지에서 모든 stall 을 관할해도 상관없다. 이렇게 되면 EX – MEM 의 1 회 stall 은 ID – EX 1 회 stall 과 동일하게 취급할 수 있다.

- ECALL 이 아닌 경우 load instruction 에 대한 ID – EX 1 회 stall
- ECALL 인 경우 모든 instruction 에 대한 ID – EX 1 회 stall, load instruction 에 대한 ID – MEM 2 회 stall

을 코드로 담으면 다음과 같다. 이 때 "use_rsx"는 ID 스테이지에서 실제로 그 register 을 읽어야만 하는지를 검사하는 부분으로, 스크린샷에서는 그 로직을 미포함하였다.

```
// EX stage: stall if load
assign stall_by_load = ((ID_rs1 == EX_rd) && use_rs1 && EX_mem_read) ||
                       ((ID_rs2 == EX_rd) && use_rs2 && EX_mem_read);

// MEM stage: stall only if load; EX stage: stall for any write
assign stall_by_ecall = ((ID_opcode == `ECALL) && (EX_rd == 5'b10001) && (EX_reg_write)) ||
                        ((ID_opcode == `ECALL) && (MEM_rd == 5'b10001) && (MEM_mem_read));

always @(*) begin
    if (stall_by_load || stall_by_ecall) begin
        is_stall = 1'b1;
    end
    else begin
        is_stall = 1'b0;
    end
end
```

IV. 논의 사항

ECALL 을 조금 더 적은 사이클로 malfunction 없이 처리할 수 있었을까?

- is_ecall 컨트롤 시그널 자체를 EX 스테이지로 pipelining 을 시킨 후, halt 여부의 signal 을 EX 스테이지에서 만든다면 다른 register read instructions 과 마찬가지로 ECALL 의 "use"를 EX 스테이지로 옮길 수 있겠다는 생각이 든다. 이렇게 되면 평균적으로 하나의 ECALL 에 대해 하나의 stall 을 줄일 수 있을 것이라고 생각된다.

	R/I-Type	LW	SW	ECALL
IF				
ID				use
EX	use produce	use	use	
MEM		produce		
WB				

⇒

	R/I-Type	LW	SW	ECALL
IF				
ID				
EX	use produce	use	use	use
MEM		produce		
WB				

V. 결론

주어진 Non-control flow input 파일에 대해 testbench를 돌려 본 결과는 figure A와 같다. 사이클 수도 정답과 46으로 일치함을 확인할 수 있다.

같은 input 파일에 대해, 위의 결과와 single-cycle CPU에서의 결과를 비교해 보자. Figure B는 non-control flow input 파일을 Lab 2에서 구현한 single-cycle CPU의 testbench로 돌려 본 결과이다.

전체 cycle의 개수는 Pipelined CPU가 오히려 더 많음을 알 수 있다. 그럼에도 Pipelined CPU를 사용하는 이유는 한 cycle에 최대 5개의 instruction을 겹쳐 실행할 수 있어, throughput이 좋아지기 때문이다. cycle의 시간 자체도 더 짧다.

Pipelined CPU를 만드는 것이 생각보다 재미있었다. 제일 까다로웠던 부분은 ECALL을 처리하는 것이었는데, 다른 instruction들이 EX 단계에서 register의 정보를 필요로 하는 것과 다르게 ECALL은 ID 단계에서부터 필요해서, ECALL 용 stall logic을 추가로 생각했어야 했다는 점이 불편했다.

```
### SIMULATING ###
TEST END
SIM TIME : 94
TOTAL CYCLE : 46 (Answer : 46)
FINAL REGISTER OUTPUT
 0 00000000 (Answer : 00000000)
 1 00000000 (Answer : 00000000)
 2 00002ffc (Answer : 00002ffc)
 3 00000000 (Answer : 00000000)
 4 00000000 (Answer : 00000000)
 5 00000000 (Answer : 00000000)
 6 00000000 (Answer : 00000000)
 7 00000000 (Answer : 00000000)
 8 00000000 (Answer : 00000000)
 9 00000000 (Answer : 00000000)
10 0000000a (Answer : 0000000a)
11 0000003f (Answer : 0000003f)
12 fffffff1 (Answer : fffffff1)
13 0000002f (Answer : 0000002f)
14 0000000e (Answer : 0000000e)
15 00000021 (Answer : 00000021)
16 0000000a (Answer : 0000000a)
17 0000000a (Answer : 0000000a)
18 00000000 (Answer : 00000000)
19 00000000 (Answer : 00000000)
20 00000000 (Answer : 00000000)
21 00000000 (Answer : 00000000)
22 00000000 (Answer : 00000000)
23 00000000 (Answer : 00000000)
24 00000000 (Answer : 00000000)
25 00000000 (Answer : 00000000)
26 00000000 (Answer : 00000000)
27 00000000 (Answer : 00000000)
28 00000000 (Answer : 00000000)
29 00000000 (Answer : 00000000)
30 00000000 (Answer : 00000000)
31 00000000 (Answer : 00000000)
Correct output : 32/32
```

Figure A. Pipelined CPU의 testbench

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000a
11 0000003f
12 fffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

Figure B. Single-cycle CPU의 testbench