CSED311: Lab 4 (due Apr. 30)

김태연(20220140), 손량(20220323)

Last compiled on: Tuesday 30th April, 2024, 16:44

1 Introduction

RISC-V architecture를 바당으로 Control flow instructiond을 처리하지 않는 Pipelined CPU를 구현한다.

2 Design

이번에 구현한 RISC-V Multicycle CPU는 다음과 같은 주요 기능을 가진 Submodule로 나누었으며, 생성된 값들을 선택하기 위해 추가적으로 Multiplexer를 구현하여 사용했다. 기존 Single-cycle CPU에 추가적으로 Forwarding unit과 Forwarding path를 설계하여 Pipelining을 지원하였다.

2.1 Program Counter

2.2 Instruction Memory and Data Memory

2.3 Register File

2.4 ALU

기존의 Single cycle CPU와 동일한 모듈 배치를 이루고 있으며, 각 모듈의 Asynchronous 혹은 Synchronous한 Clock signal 또한 Single-cycle CPU와 동일한 구조를 가지고 있다. 다만, 현재 구현하고자 하는 CPU는 Control flow instruction이 없기 때문에 해당하는 Branch Instruction 등을 지원하지 않도록 설계되었다. `opcodes.v`에서 자세히 확인할 수 있다.

2.5 Inter-Stage Registers

Pipelined CPU가 일반적인 CPU와 가장 큰 다른 점은 CPU의 각 부분 별로 다른 Instruction을 동시에 처리할 수 있다는 점이다. 따라서, Instruction Fetch에서 Write Back에 이르는 구간 사이에 값을 전달하기 위해 Register를 두어야 한다. 편의상, 본 보고서에서는 이를 Interstage Register이라 부를 예정이며, 각 구간 별로 다음과 같은 Register가 존재한다.

2.5.1 IF–ID

Instruction Fetch –Instruction Decode 사이에서 저장되는 신호는 다음과 같다.

- `IF_ID_reg_write_enable` – 현재 CPU의 상황이 Hazard가 없을 경우에 해당 신호가 켜진다.
- `IF_ID_reg_inst_out` – Instruction Fetch Stage로부터 instruction을 전달받는다.

2.5.2 ID–EX

Instruction Decode – Execute 사이에서 저장되는 신호는 다음과 같다. Instruction Decode 단계에서 Control unit을 통해 다양한 Control signal이 생성되며, 사용될 Stage별로 pipeline의 단계를 거치며 각 stage에 도달할 때 까지 CPU clock에 synchronous하게 전달된다.

- `ID_EX_reg_wb_enable` – ID/EX Stage에 있는 Instruction이 추후 Write Back Stage에서 Register file으로 Write back이 이루어지는지를 나타낸다.
- `ID_EX_reg_mem_enable` – ID/EX Stage에 있는 Instruction이 추후 Memory Stage에서 Data Memory에 접근하는지 나타낸다.
- `ID_EX_reg_mem_write` – ID/EX Stage에 있는 Instruction이 추후 Memory Stage에서 Data Memory에 값을 작성하는지 나타낸다.
- `ID_EX_reg_op2_imm` – ID/EX Stage에 있는 Instruction이 현재 Execute Stage에서 ALU에 사용할 입력이 Register file의 값인지, Immediate generator의 값인지 나타낸다.
- `ID_EX_reg_is_halted` – ID/EX Stage에 있는 Instruction이 추후 halt를 호출하는지 나타낸다.
- `ID_EX_reg_ex_forwardable` – ID/EX Stage에 있는 Instruction이 Forward될 수 있는지를 전달한다. 이 값은 Control Unit과 Hazard Detection Unit으로부터 결정된다.
- `ID_EX_reg_rs1` – ID/EX Stage에 있는 Instruction에서 사용될 `rs1` register value이다.
- `ID_EX_reg_rs2` – ID/EX Stage에 있는 Instruction에서 사용될 `rs2` register value이다.
- `ID_EX_reg_rd_id` – ID/EX Stage에 있는 Instruction이 추후 Write Back Stage에서 작성할 register의 index이다.
- `ID_EX_reg_inst` – ID/EX Stage에 있는 Instruction으로부터 ALU Control Unit에 전달될 신호이다.
- `ID_EX_reg_imm` – ID/EX Stage에 있는 Instruction이 Immediate generator로부터 얻은 값을 전닳한다.

2.5.3 EX–MEM

Execute – Memory 사이에서 저장되는 신호는 다음과 같다.

- `EX_MEM_reg_wb_enable` – EX/MEM Stage에 있는 Instruction이 추후 Write Back Stage에서 Register file으로 Write back이 이루어지는지를 나타낸다.
- `EX_MEM_reg_mem_enable` – EX/MEM Stage에 있는 Instruction이 현재 Memory Stage에서 Data Memory에 접근하는지 나타낸다.
- `EX_MEM_reg_mem_write` – EX/MEM Stage에 있는 Instruction이 현재 Memory Stage에서 Data Memory에 값을 작성하는지 나타낸다.
- `EX_MEM_reg_is_halted` – EX/MEM Stage에 있는 Instruction이 추후 halt를 호출하는지 나타낸다.

- `EX_MEM_reg_ex_forwardable` – EX/MEM Stage에 있는 Instruction이 Forward될 수 있는지를 전달한다.
- `EX_MEM_reg_alu_output` – EX/MEM Stage에서 ALU의 출력을 전달하기 위한 Register이다.
- `EX_MEM_reg_rs2` – EX/MEM Stage에서 현재 Memory Stage에 값을 작성하기 위해 필요한 `rs2` 값의 출력을 전달하기 위한 Register이다.
- `EX_MEM_reg_rd_id` – EX/MEM Stage에 있는 Instruction이 추후 Write Back Stage에서 작성할 register의 index이다.

2.5.4 MEM–WB

Memory – Write Back 사이에서 저장되는 신호는 다음과 같다. Halt instruction 또한 다른 instruction처럼 CPU의 모든 pipeline을 통과한 이후 CPU가 종료될 수 있게 설계하였다. 따라서 halt instruction을 처리하기 위한 `is_halted` 신호가 현재 단계까지 전파된 것을 확인할 수 있다.

- `MEM_WB_reg_wb_enable` – MEM/WB Stage에 있는 Instruction이 현재 Write Back Stage에서 Register file으로 Write back이 이루어지는지를 나타낸다.
- `MEM_WB_reg_is_halted` – MEM/WB Stage에 있는 Instruction이 halt를 호출하는지 나타낸다.
- `MEM_WB_reg_rd_id` – MEM/WB Stage에 있는 Instruction이 현재 Write Back Stage에서 작성할 register의 index이다.
- `MEM_WB_reg_rd` – MEM/WB Stage에 있는 Instruction에서 register에 작성할 값을 전달받는다.

2.6 Control Unit

Instruction Fetch 과정을 거친 이후 Control Unit으로 `IF_ID_reg_inst_out` Interstage register의 값이 입력되며, 해당 instruction의 type으로부터 다음과 같은 종류의 Control signal을 만들어낼 수 있다. Signal이 생성되어 필요한 Pipelining stage까지 Interstage register를 통해 Clock Synchronous하게 전달된다.

- `opcode` – ALU로 전달될 Opcode이다.
- `wb_enable` – 해당 Instruction이 Write Back stage에서 Register file에 값을 쓰는 경우 활성화된다.
- `mem_enable` – 해당 Instruction이 Memory stage에서 Memory에 접근하는 (Read / Write) 경우 활성화된다.
- `mem_write` – 해당 Instruction이 Memory stage에서 Memory에 값을 쓰는 경우 활성화된다.
- `op2_imm` – ALU의 `in_2` 입력으로 Register file / Immediate generator의 출력 중 선택한다.
- `is_ecall` – 프로그램을 Halt하기 위한 `ecall`인 경우 활성화되며, 다른 Instruction과 마찬가지로 Pipeline을 통해 전달된다.

- `rs2_used` – 해당 Instruction에서 실제로 `rs2`가 두 번째 Register로써 사용되었는지 출력한다. `rs2`의 Forwarding을 위해 필요하게 되었다.
- `ex_forwardable` – 해당 Instruction에서 Execute stage의 ALU input으로 값을 전달할 수 있는지 확인한다. 해당 Signal은 ID/EX stage 및 EX/MEM stage에 전달되고, EX stage와 MEM stage의 마지막 단계에서 EX stage의 처음으로 값을 전달하여 ALU의 입력을 제공하고 있다.

2.7 Hazard Detection Unit

2.8 Forwarding Unit

본 Pipelined CPU의 구현에서는 Hazard를 detect하는 부분과 Data를 Forwarding하는 부분을 분리하여 구현하였다. Hazard Detection Unit에서는 Load instruction이 Execute Stage에 배치되면서 발생하는 Stall을 관리하고, Forwarding Unit에서는 필요한 경우 가능한 Forwarding 조합으로부터 ALU input multiplexer에 값을 전달한다. 자세한 내용은 3 **Hazard Detection and Data Forwarding** Section 에서 다루었다.

3 Hazard Detection and Data Forwarding

3.1 Hazard Detection

Hazard Detection Unit은 EX stage에 Load instruction이 배치됨으로써 발생하는 불가피한 "Stall" 을 처리하도록 설계하였다. Hazard Detection Unit의 출력인 `is_hazardous`는 각 `rs1`, `rs2` 별로 2개의 출력을 가진다. 이 중 하나만이라도 Hazardous하게 판단이 된 경우 `pc_write_enable` 및 `IF_ID_reg_write_enable` 신호를 Disable시켜 하나의 Clock cycle을 강제로 소비하여 Hazard를 해결한다 (Bubbling).

다음은 Hazard Detection Unit의 입/출력 신호와 그 출처이다.

- `enable` – 각각 `rs1` 및 `rs2`가 사용되는지에 따라 서로 다른 신호를 사용한다. `rs1`의 경우 1이고, `rs2`의 경우 해당 Field를 `rs2`로 사용하는 Instruction에 한해 활성화된다.
- `rs_id` – Instruction에서 사용되는 register id이다.
- `ex_mem_read` – Pipeline을 통해 전달된 `reg_mem_enable`과 `reg_mem_write` 신호로부터, Execute Stage의 Instruction이 Load인지 알아낸다.
- `ex_rd_id` – Pipeline을 통해 Write back 될 Load instruction의 register id이다.
- `is_hazardous` – 해당 Unit의 최종 출력으로, 해당 instruction이 Hazardous한지 나타낸다.

또한, Ecall 시 `x17` Register에 값이 쓰일 수 있기 때문에 이를 관리하기 위해 해당 `x17` register에 대해서 Hazard를 Observe하도록 `EcallHazardDetectionUnit` module을 제작하였다.

`HazardDetectionUnit.is_hazardous`로 출력된 값은 각각 `rs1`, `rs2`, `ecall` 에 대해 발생할 수 있으며, 셋 중 하나라도 활성화 된 경우 `cpu.v`의 `is_hazardous` wire에 신호를 주게 된다. 이에 따라 Program Counter를 update하는 signal인 `pc_write_enable`과 Instruction Fetch와 Instruction Decode 사이 Interstage register를 관리하는 signal인 `IF_ID_reg_write_enable`을 모두 비활성화하게 된다. 결과적으로 PC update와 Instruction Fetch를 진행하지 못하고, 현재 진행중이던 EX, MEM, WB만 Pipeline에서 Stage를 진행하며 Stall을 구현하였다.

3.2 Data Forwarding

Forwarding Unit은 각 단계 사이의 Inter-stage register로부터 ALU에 Forward된 값을 전달하며, Muliplexer로 선택하기 위해 Forwarded 여부를 같이 출력한다. Forwarding에는 다음과 같은 가능한 조합이 있으며, 두 조합 모두 만족하는 경우 Pipeline에서 가장 최근에 갱신되는 값인 EX/MEM stage의 값을 전달해야 한다.

- `EX/MEM` stage – `ex_forwardable` 신호가 Pipeline을 통해 전달되고, 현재 ALU의 `rs` 와 Forward destination register가 (0이 아닌 상태로) 같은 경우 ALU output을 Forwarding한다. 이 경우 Hazard distance = 1이다.
- `MEM/WB` stage – `mem_wb_enable` 신호가 Pipeline을 통해 전달되고, 현재 ALU의 `rs` 와 Forward destination register가 (0이 아닌 상태로) 같은 경우 ALU output을 Forwarding한다. 이 경우 Hazard distance = 2이다.

Hazard distance가 3보다 큰 경우에는 필요로 하는 Register value가 이미 Register File으로 Write back이 완료되었기 때문에 일반적인 흐름과 동일하게 실행할 수 있다. 이러한 Pipeline을 따라 전달되는 `forwardable` 신호는 Control Unit과 Hazard Unit에서 Hazardous하다고 판별된 Instruction에 활성화되며, `wb_enable` 신호는 Control Unit으로부터 생성된다.

다음은 Forwarding Unit의 입력 신호와 그 출처이다.

- `rs_id` – Forwarding 대상이 되는 Register ID이다.
- `ex_forwardable`, `ex_rd_id`, `ex_rd` – 각각 EX stage에서 Instruction이 Register file에 Write back을 시도하는지, 그 때 Destination Register ID, ALU result이다.
- `mem_wb_enable`, `mem_rd_id`, `mem_rd` – 각각 MEM stage에서 Instruction이 Register file에 Write back을 시도하는지, 그 때 Destination Register ID, Memory output이다.

앞서 설명한 Forwarding 가능한 조합이 존재하는 경우 Forwarding을 진행하여 모듈의 `is_forward = 1`, `forwarded_value = xxx_rd`로 설정한다. 각각 `rs1` 및 `rs2`의 Forwarding을 관리하기 위해 Multiplexer를 추가하였으며, 이 `sel`으로 `is_forward`, `mux_in_1`으로 `forwarded_value`가 전달되어 선택을 통해 필요한 값이 ALU에 입력되도록 구성했다.

4 Implementation

각 베릴로그 파일에 대한 세부 설명은 다음과 같다.

4.1 `top.v` – Top module

`cpu.v`에서 구현한 `cpu` 모듈을 사용하여 Pipelined CPU를 구현하였다.

4.2 `cpu.v` – 내부 Module을 연결하여 Pipelined CPU 구성

`cpu.v`는 다음과 같은 submodule을 연결하여 구성하였다.

- `PC pc` – Program Counter이다.
- `InstMemory imem` – Instruction Memory이다.
- `IFIDRegister if_id_reg` – Instruction Fetch stage에서 Instruction Decode stage로 넘어가는 Interstage register 전달 모듈이다.

- `ControlUnit ctrl_unit` – Control unit으로, ALU 및 각 Memory module, Hazard Detection, Forwarding 등에 사용되는 전반적인 Control signal을 instruction으로부터 생성한다.
- `MUX2X1 rs1_mux` – `ecall`의 경우 고정적으로 `x17`을 `rs1`으로 넘겨주기 위한 Muliplexer 이다.
- `RegisterFile reg_file` – Register file이다.
- `EcallUnit ecall_unit` – `ecall`을 처리하기 위한 Unit으로, Pipeline을 모두 수행한 이후에 해당 모듈에 Signal이 전달되도록 설계되었다.
- `ImmediateGenerator imm_gen` – Immediate generator이다.
- `HazardDetectionUnit rs1_hdu` – Execution stage에 `rs1`으로 인한 Hazard를 감지, Forwarding이 필요한 경우 Forwarding module인 `rs1_fwd` 에 신호를 제공한다.
- `HazardDetectionUnit rs2_hdu` – Execution stage에 `rs2`으로 인한 Hazard를 감지, Forwarding이 필요한 경우 Forwarding module인 `rs2_fwd` 에 신호를 제공한다.
- `EcallHazardDetectionUnit ecall_hdu` – Execution stage에 `x17`으로 인한 Hazard 가 발생할 경우, `ecall`의 동작을 보장하기 위한 전용 Hazard Detection Unit이다.
- `IDEXRegister id_ex_reg` – Instruction Decode stage에서 Execution stage로 넘어가는 Interstage register 전달 모듈이다.
- `ALUControlUnit alu_ctrl_unit` – ALU Control Unit으로, Single cycle CPU에서 Control flow instruction에 필요한 operation을 제외하고 구성되었다.
- `ForwardingUnit rs1_fwd` – EX/MEM stage와 MEM/WB stage의 `rs1` ID와 그 값, 현재 ALU에 들어간 `rs1` ID로부터 ALU에 Forwarding할 값과 Forward 여부를 결정한다.
- `MUX2X1 rs1_fwd_mux` – ALU의 `rs1` 입력으로 Register file의 값과 `rs1_fwd` Forwarding unit의 Forwarded value 중 필요한 값을 선택한다.
- `ForwardingUnit rs2_fwd` – EX/MEM stage와 MEM/WB stage의 `rs2` ID와 그 값, 현재 ALU에 들어간 `rs2` ID로부터 ALU에 Forwarding할 값과 Forward 여부를 결정한다.
- `MUX2X1 rs2_fwd_mux` – ALU의 `rs2` 입력으로 Register file의 값과 `rs2_fwd` Forwarding unit의 Forwarded value 중 필요한 값을 선택한다.
- `MUX2X1 mux_alu_in_2_select` – `rs2` 입력으로 `rs2_fwd_mux`를 통과한 최종적인 `rs2`의 값과 Immediate generator로부터 생성된 값 중 필요한 값을 선택한다.
- `ALU alu` – ALU를 구현하였다.
- `EXMEMRegister ex_mem_reg` – Execution stage에서 Memory stage로 넘어가는 Interstage register 전달 모듈이다.
- `DataMemory dmem` – Data memory이다.
- `MUX2X1 rd_mux` – Write back 될 value를 Register file의 출력 `rs2`와 Data Memory의 `dmem_dout`으로부터 선택한다.
- `MEMWBregister mem_wb_reg` – Memory stage에서 Write back stage로 넘어가는 Interstage register 전달 모듈이다.

4.3 `RegisterFile.v` – Register File

`RegisterFile` 모듈의 입력 신호 중 `rd`와 `rd_din`은 Write Back stage까지 Pipeline을 진행한 이후 입력되며, 이외의 입력은 Instruction Decode Stage에서 입력된다.

4.4 `InstMemory.v` – Instruction Memory

4.5 `DataMemory.v` – Data Memory

Pipelined CPU의 구현에서는 Instruction Memory와 Data Memory가 분리되어 있는 구조이다.

4.6 `ALU.v` – ALU

4.7 `ALUControlUnit.v` – ALU Control Unit

현재 구현의 정의에 따르면, ALU의 입력으로 Branch를 위한 Boolean operation이 주어지지 않았다. 따라서 Single Cycle CPU 구현에서 해당 부분을 제외하고 구성되었다.

4.8 `IFIDRegister.v` – Intsruction Fetch stage / Instruction Decode stage 간 Interstage Register 전달

4.9 `IDEXRegister.v` – Instruction Decode stage / Execution stage 간 Interstage Register 전달

4.10 `EXMEMRegister.v` – Execution stage / Memory stage 간 Interstage Register 전달

4.11 `MEMWBregister.v` – Memory Stage / Write back 간 Interstage Register 전달

각 Stage 간 전달되는 Interstage Register를 CPU clock에 Synchronous하게 다음 Stage로 전달해주는 Unit들이다.

4.12 `HazardDetectionUnit.v` – Hazard Detection Unit

Hazard Detection Unit을 구현했다.

4.13 `ForwardingUnit.v` – Forwarding Unit

EX, MEM stage의 register ID, register value로부터 Forwarding을 감지하고, Forwarding을 진행한다.

4.14 `ECallUnit.v` – `ecall` Unit

4.15 `EcallHazardDetectionUnit.v` – `ecall` Hazard Detection Unit

`ecall`을 구현하기 위한 모듈이며, 해당 모듈으로 들어오는 Signal은 모든 Stage를 지나서 전달된다. 이를 구현하기 위해 `cpu.v`의 Interstage register로 `is_halted`를 두어 구현했다. 또한, `halt`는 `x17` register에 값을 작성해서 이룰 수 있으며, 마찬가지로 Register의 Hazard가 발생할 수 있다. 특별히 Ecall만을 관리하기 위해 `EcallHazardDetectionUnit` 모듈을 만들어서 관리했다.

5 Discussion Conclusion

Single cycle CPU 구현에서 사용한 `non-controlflow_mem.txt`를 사용하여 Benchmark를 실행했으며, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다.

```
### SIMULATING ###
TEST END
SIM TIME : 98
TOTAL CYCLE : 48 (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
```

이와 **Lab 2**에서 구현한 Single-cycle CPU에서 같은 파일으로 수행한 Benchmark output을 비교하면 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39 (Answer : 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
Correct output : 32/32
```

Single-cycle CPU에서 얻은 Cycle은 39 Cycle으로, 해당 Single-cycle CPU의 Cycle은 Pipelined CPU의 모든 Stage를 (IF – ID – EX – MEM – WB) 포함하기 때문에 5 Microcycle이 필요하게 된다. 같은 Microcycle의 시간을 가진다고 가정했을 때, Single-cycle CPU는 39 $\times 5 = 195$ Mircocycle을 필요로 했던 반면, 이번에 구현한 Pipelined CPU는 결과와 같이 48 Microcycle의 시간이 걸렸다. 따라서 $195/48 \approx 4.06$ 배의 성능 향상을 이룰 수 있었다. 이론치인 5배의 성능 향상을 낼 수 없었는데, 이는 Load instruction에서 Stall이 발생하여 필연적인 Bubble을 형성하였기 때문이다.