CSED311: Lab 4 (due May. 14)

김태연(20220140), 손량(20220323)

Last compiled on: Tuesday 14th May, 2024, 13:17

1 Introduction

RISC-V architecture를 바당으로 Control flow instruction을 처리하지 않는 Pipelined CPU의 구현에 Branch prediction을 덧붙여 Control flow instruction을 처리할 수 있도록 설계한다.

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 지원하여 Data hazard를 피했으며, GShare branch predictor를 디자인하여 Control hazard를 피했다.

2.1 Program Counter

Program counter는 CPU clock에 synchronous하게 현재 program counter를 출력하고, 들어오는 32-bit 입력으로 현재 program counter를 갱신한다. 해당 모듈 내부에는 branch prediction 기능을 가지고 있지 않다.

2.2 Instruction Memory and Data Memory

구현한 CPU는 instruction memory 영역과 data memory 영역을 분리하여 가지고 있으며, 각 memory의 특징은 다음과 같다.

- Instruction Memory
  - Size – 1024 Words = 4KiB를 가지고 있다.
  - Propagation – `imem_dout`을 통해 출력된 instruction은 IF/ID interstage register으로 입력되어 CPU clock에 synchronous하게 전달된다.
- Data Memory
  - Size – 16384 Words = 64KiB를 가지고 있다.
  - Input – Forwarding이 완료된 EX/MEM interstage의 `rs2`를 입력으로 받는다.
  - Output – `dmem_dout`와 ALU의 출력 중 multiplexer를 통해 선택되어 MEM/WB interstage register으로 전달된다.

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. `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. 이 중 Control flow instruction을 처리하면서 새롭게 도입하거나 수정된 Interstate register들은 **굵은** 글씨와 + 기호를 사용해 명시하였다. Control Hazard는 EX stage에서 모두 Resolve되기 때문에, control hazard와 관련하여 추가로 설치한 interstage register은 모두 IF/ID 혹은 ID/EX stage에 새로 추가된 것을 확인할 수 있다.

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_valid +`** – 현재 Instruction의 validness를 전달하기 위해 사용되는 값이다. IF/ID stage에서는 항상 1(Valid)를 전달한다.
- **`IF_ID_reg_bubble +`** – IF/ID stage에서 instruction fetch를 중단해야 하는 경우 활성화되는 신호이다. 해당 값은 `ctrl_hdu_is_hazardous`로부터 전달되어, control hazard가 발생하는 경우 신호가 활성화된다.
- `IF_ID_reg_inst_out` – Instruction Fetch Stage로부터 instruction을 전달받는다.
- **`IF_ID_reg_pc +`** – IF/ID stage를 지나고 있는 instruction의 program counter 값을 전달한다.

ID (Instruction Decode) stage에서 `IF_ID_reg_bubble` 신호가 활성화되어 전달된 경우, ID/EX interstage register의 입력으로 전달되는 memory 접근(read/write) 및 instruction validness에 대한 모든 control input을 비활성화한다. (`wb_enable`, `mem_write_in`, `is_halted_in`, `valid_in`) 이로써 더 이상 해당 invalid instruction이 pipeline 내에서 전파되는 것을 방지할 수 있다.

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_op1_pc`** + – ID/EX Stage에서 EX stage의 ALU의 입력으로 Regsiter file의 값을 사용할 지, Program counter의 값을 사용할 지 선택하는 신호이다.
- **`ID_EX_reg_op2_sel`** + – ID/EX Stage에서 EX stage의 ALU의 입력으로 Register file의 값, Immediate generator, +4 중 어떤 값을 사용할 지 선택하는 신호이다.
- `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_valid`** + – 현재 ID/EX Stage를 지나고 있는 Instruction이 valid한지 나타내기 위한 입력이다. 이전 Stage로부터 전달된 Validness와 Bubble의 여부, Data 및 Control Hazard 여부를 모두 고려했을 때 문제가 없는 경우 해당 신호를 활성화한다.
- **`ID_EX_reg_is_branch`** + – Instruction decode 결과 현재 Instruction이 Branch type인지 다음 Stage로 전달한다.
- **`ID_EX_reg_is_jalr`** + – Instruction decode 결과 현재 Instruction이 `JALR`인지 다음 Stage로 전달한다.
- `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.
- **`ID_EX_reg_pc`** + – 현재 ID/EX Stage를 지나는 Instruction이 놓이던 Instruction address를 저장하고 있다.

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. 이 중 Control flow instruction을 처리하면서 새롭게 도입하거나 수정된 Control unit Signal들은 **굵은** 글씨와 + 기호를 사용해 명시하였다.

- `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.
- **`op1_pc +`** – ALU의 `in_1` 입력으로 현재 Pipeline 단계를 진행 중인 program counter의 값 / Register file의 값 중 선택한다.

- `op2_sel` + – ALU의 `in_2` 입력으로 Register file / Immediate generator의 출력 / +4 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.
- `is_branch` + – 해당 Instruction이 Branch type인지 여부를 전달한다.
- `is_jalr` + – 해당 Instruction이 `JALR` type인지 여부를 전달한다.

2.7 Hazard Detection Unit

2.8 Hazard Detection

Hazard Detection을 구현하기 위해 `HazardDetectionUnit` 모듈을 구현하였다. 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.

2.9 Data Forwarding

Data Forwarding을 구현하기 위해 `ForwardingUnit` 모듈을 두었다. Forwarding Unit은 각 단계 사이의 Inter-stage register로부터 ALU에 Forward된 값을 전달하며, multiplexer로 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.

3 Branch Prediction

해당 CPU의 Branch predictor으로 GShare branch predictor를 구현하였으며, 다음과 같은 구성 요소를 가지고 있다.

- BTB – 32 (2^5)개의 32-bit Branch target buffer entry가 존재한다.
- Tag table – 32개의 27 (32 - 5) bit entry가 존재하여 instruction address의 LSB 27bit를 저장한다. 각 entry는 BTB에 대응될 때 index에 따라 tag가 동일한지 확인한다.
- Global BHSR – 해당 BHSR은 LSB 5bit와 XOR하여 BTB index를 결정하게 된다. 따라서 5bit 크기의 Global branch history shift register을 가지게 된다.
- PHT – 32 (2^5)개의 2-bit saturation counter를 가지는 pattern history table이며, 각 entry는 BTB에 대응되어 counter의 상태에 따라 branch 여부를 결정하게 된다. 각 0(`STRONG_NOT_TAKEN`) 부터 3(`STRONG_TAKEN`)으로 갈 수록 branch prediction에 대한 확신이 증가하게 된다.

Branch predictor은 다음 Program counter를 예측하는 부분과, 실제 EX stage의 결과로부터 Branch predictor를 갱신하는 부분으로 나뉘어져 있다. 다음 Program counter의 예측은 Combinational하게 생성되며, Branch predictor 내부 상태의 갱신은 Clock에 Synchronous하게 동작한다. 각각을 구현하기 위한 입/출력은 다음곽 같다.

- Branch Prediction
  - `current_pc` – 현재 Branch predictor를 거치는 instruction의 program counter가 입력된다.
  - `predicted_pc` – 현재 Branch predictor의 상태와 program counter로부터 예측한 다음 instruction의 address를 출력한다.
- Branch Predictor Update
  - `update_pred` – Branch predictor 내부 상태를 갱신하기 위해 필요한 활성화 Signal이다. `cpu.v`의 구현 상 `ID_EX_reg_is_branch`가 입력되어 Branch instruction일 때만 GShare branch predictor의 상태를 갱신할 수 있다.
  - `branch_inst_address` – Branch predictor 갱신의 대상이 되는 branch instruction의 주소를 입력한다.
  - `resolved_next_pc` – Adder를 거치고, EX stage를 통과하여 실제로 계산되고 선택된 다음 Instruction의 주소를 반환한다.
  - `predictor_wrong` – 해당 Branch instruction이 Hazard를 띌 수 있는지 여부가 주어지게 된다.

`branch_inst_address`에 4를 더한 값이 `resolved_next_pc`와 같은지 비교를 통해 Branch taken / NOT taken 여부를 알아낼 수 있고. `predictor_wrong` (예측 결과의 일치 여부) 입력으로부터 실제로 Branch predictor가 어떤 예측을 하였는지 알 수 있다. 총 4가지의 경우((Prediction taken / NOT taken) ×(Actual taken / NOT taken))의 조합이 가능하다.
실제 Branch taken 여부에 따라 PHT가 갱신되고 BHSR이 업데이트된다.

- Actual Taken – PHT의 2-bit Counter를 증가시킬 수 있는 경우 증가시키고, BHSR shift 이후 1으로 LSB를 masking한다.
- Actual NOT Taken – PHT의 2-bit Counter를 감소시킬 수 있는 경우 감소시키고, BHSR shift 이후 0으로 LSB를 masking한다.

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. 이 중 Control flow instruction을 처리하면서 새롭게 도입되거나 수정된 Module은 **굵은** 글씨와 + 기호를 사용해 명시하였다.

- `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 multiplexer이다.
- `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 Hazard가 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.
- **`MUX2X1 op1_mux`** + – ALU의 `in1` 입력으로 현재 Program counter 및 Register file의 값 중 선택하는 multiplexer이다.
- `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`** + – ALU의 `in2` 입력으로 `rs2_fwd_mux`를 통과한 최종적인 `rs2` 의 값과 Immediate generator에서 생성된 값, +4 중 필요한 값을 선택한다.
- `ALU alu` – ALU를 구현하였다.
- **`PCGenerator pc_gen`** + – Execution stage를 거친 실제 연산 결과로부터 다음 instruction을 fetch해야하는 Program counter를 계산한다.

- `ControlHazardDetectionUnit ctrl_hdu` + – 해당 Instruction이 Control Hazard를 가지고 있는지 판단한다.
- `MUX2X1 mux_next_pc` + – 다음 Program counter를 Branch predictor으로부터 갱신할 것인지, `pc_gen`을 통해 계산된 Program counter으로 갱신할 것인지 결정하는 multiplexer이다.
- `BranchPredictor branch_predictor` + – GShare Branch Predictor를 통해 다음 Instruction fetch가 일어날 Instruction address를 예측한다.
- `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 `PC.v` – Program Counter

Clock에 Synchronous하게 현재 PC를 출력하고, 들어오는 32-bit 입력으로 현재 PC를 갱신하는 모듈이다.

4.4 `PCGenerator.v` – 다음 PC 계산

Unconditional Jump (`JAL`, `JALR`)의 경우 해당 Offset을 더하여 다음 PC를 출력한다. Branch의 경우 입력된 `imm`만큼을 더해 다음 PC를 출력하게 된다. 그 이외의 경우는 다음 Instruction (PC + 4)를 출력한다.

4.5 `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.

4.6 `ControlUnit.v` – Control Unit

Single Cycle CPU의 기능에 더해, 다양한 Data hazard / Control Hazard를 Resolve하기 위한 Control signal output이 존재한다.

4.7 `InstMemory.v` – Instruction Memory

4.8 `DataMemory.v` – Data Memory

Pipelined CPU의 구현에서는 Instruction Memory와 Data Memory가 분리되어 있는 구조이다.

4.9 `ALU.v` – ALU

4.10 `ALUControlUnit.v` – ALU Control Unit

4.11 `ImmediateGenerator.v` – Immediate Generator

Single Cycle CPU의 구성과 동일한 ALU 설계와 ALU Opcode, Immediate generator의 형태를 가지고 있다.

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

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

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

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

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

4.16 `BranchPredictor.v` – GShare Branch Predictor

GShare branch predictor를 구현하였으며, 해당 모듈 내에서 Prediction 성공 여부로부터 Branch predictor 내부 구성 요소(BHSR, PHT, BTB 등)의 갱신이 Clock synchronous하게 이루어지며, 다음 Program counter의 예측은 해당 모듈의 Combinational block에 구현하였다.

4.17 `HazardDetectionUnit.v` – Data Hazard Detection Unit

Data Hazard Detection Unit을 구현했다.

4.18 `ControlHazardDetectionUnit.v` – Control Hazard Detection Unit

Control Hazard Detection Unit을 구현했다.

4.19 `ForwardingUnit.v` – Forwarding Unit

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

4.20 `ECallUnit.v` – `ecall` Unit

4.21 `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.

4.22 `Adder.v` – Adder

4.23 `MUX2X1.v` – 2-to-1 Multiplexer

4.24 `MUX4X1.v` – 4-to-1 Multiplexer

Pipelined CPU의 구현에 필요한 Multiplexer 및 Adder를 개별의 모듈로 분리하였다.

5 Discussion

5.1 Perfomance of GShare Branch Predictor

GShare branch predictor의 경우, BHSR을 통해 이전까지의 Branch 기록과 XOR을 통해 같은 Address에서 Branch를 수행하더라도, 결국 BTB Table의 index가 달라지게 된다. 기존의 Branch history에 따라 Branch 여부를 좀 더 자세히 택할 수 있다는 점에서 일반적인 Global counter를 사용하는 Branch prediction보다 좋은 Branch prediction rate를 보이게 된다.

5.2 Comparing Policy

또한, 현재 Branch predictor의 사양을 다르게 해 보며 각각의 Testbench를 가동했을 때 최종 CPU Cycle은 다음과 같았다. 다음과 같이 3개의 Branch predictor를 사용해 비교하였다.

- `PC = 0` – 다음 Program counter를 항항 0으로 설정하여 항상 Bubble을 일으키도록 설정하였다. (대조군)
- Always NOT taken – 항상 PC+4를 다음 PC로 prediction하도록 설정하였다.
- GShare – 이번 구현에서 우리가 사용한 Branch predictor이다.

	`PC = 0`	Always NOT taken	GShare
`basic`	86	36	36
`non-controlflow`	119	48	48
`ifelse`	104	44	44
`loop`	668	323	301
`recursive`	2690	1209	1109

Always NOT Taken의 경우와 GShare branch predictor를 비교해 보았을 때 복잡한 Testbench에서 다음과 같은 성능 향상을 확인할 수 있었다.

- `loop` – 22 Cycle 감소 (107.3% Faster)
- `recursive` – 100 Cycle 감소 (109.0% Faster)

5.3 Comparing Cache Size

현재 Branch Predictor의 구현은 32 (2^5)개의 Entry를 가진 PHT와 BTB를 통해 구현되었다. 해당 Entry의 수를 바꿔 보면서 얼마나 효율적인지 실험해 보고자 했다. `BHSR_WIDTH`를 달리하면서 Testbench `recursive`를 수행하였을 때 실험 결과는 다음과 같다.

BHSR_WIDTH	recursive
2	1195
3	1156
4	1140
5	1109
6	1061
7	1063
8	1065
9	1070
10	1067
20	1078

최적의 BTB 및 PHT 크기는 `BHSR_WIDTH`가 6일 때, 해당 Cache들의 크기가 $2^6 = 64$가 되어 이때 최적의 크기를 가지게 된다. 이보다 더 작은 경우, Branch prediction에 유효하게 쓰일 정보가 BTB에 들어가 있음에도 이를 invalidate하고 새로운 정보로 덮을 수 있으며, BHSR의 크기가 작기 때문에 충분히 다양한 index의 BTB entry로 분산시키지 못하였을 것으로 추측할 수 있다. 반면 이보다 더 큰 경우, BHSR에 의해 과도하게 PHT 상에 Branching 정보가 분포되어 실제로 같은 Address와 같은 Branch 여부를 지녔음에도 BHSR으로 인해 다른 BTB entry에 대응될 수 있을 것이다.

6 Conclusion

`basic_mem.txt`를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다.

```
### SIMULATING ###
TEST END
SIM TIME : 74
TOTAL CYCLE : 36 (Answer : 36)
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 00000013 (Answer : 00000013)
11 00000003 (Answer : 00000003)
12 fffffff7 (Answer : fffffff7)
13 00000037 (Answer : 00000037)
14 00000013 (Answer : 00000013)
15 00000026 (Answer : 00000026)
16 0000001e (Answer : 0000001e)
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
```

non-controlflow_mem.txt를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다.

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

ifelse_mem.txt를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다.

```
### SIMULATING ###
TEST END
SIM TIME : 90
TOTAL CYCLE : 44 (Answer : 44)
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 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 0000000a (Answer : 0000000a)
15 00000028 (Answer : 00000028)
16 00000000 (Answer : 00000000)
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
```

loop_mem.txt를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다.

```
### SIMULATING ###
TEST END
SIM TIME : 604
TOTAL CYCLE : 301 (Answer : 323)
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 00000000 (Answer : 00000000)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 0000000a (Answer : 0000000a)
15 00000009 (Answer : 00000009)
16 0000005a (Answer : 0000005a)
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
```

`recursive_mem.txt`를 사용하여 Benchmark를 실행했을 때, 다음과 같은 결과를 얻었으며 모든 Testcase 및 Register가 맞는 것을 확인했다.

```
### SIMULATING ###
TEST END
SIM TIME : 2220
TOTAL CYCLE : 1109 (Answer : 1188)
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 0000000d (Answer : 0000000d)
11 00000000 (Answer : 00000000)
12 00000000 (Answer : 00000000)
13 00000000 (Answer : 00000000)
14 00000001 (Answer : 00000001)
15 0000000d (Answer : 0000000d)
16 00000015 (Answer : 00000015)
17 0000000a (Answer : 0000000a)
18 00000000 (Answer : 00000000)
19 00000000 (Answer : 00000000)
20 00000000 (Answer : 00000000)
21 00000022 (Answer : 00000022)
22 00000000 (Answer : 00000000)
23 00000037 (Answer : 00000037)
24 00000059 (Answer : 00000059)
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
```