

컴퓨터 구조 Lab5 Cache

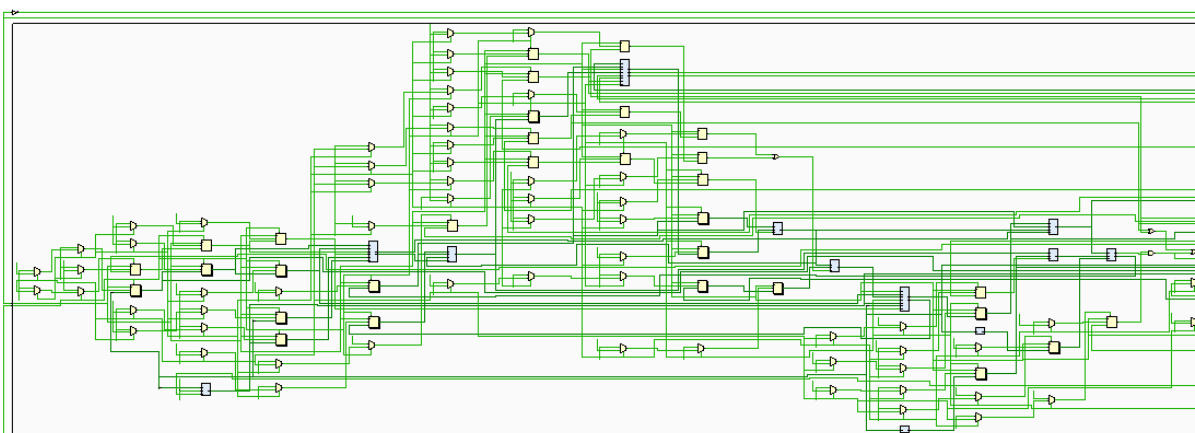
20210207 이지현

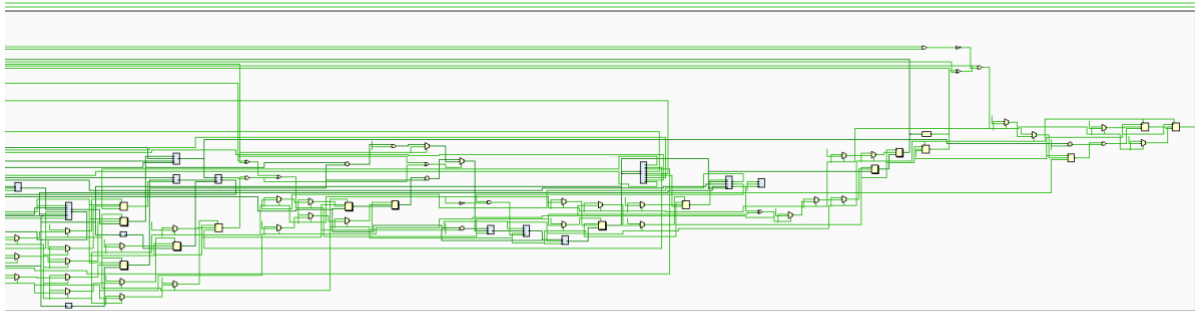
20210794 정유진

1. Introduction

- 1.1. Lab 5는 Vivado를 이용하여 Direct-mapped Cache가 존재하는 5-stage pipelined RISC-V CPU를 구현하는 것을 목표로 한다.
- 1.2. Cache state는 CACHE_STOP 3'b000, CACHE_END 3'b100, TAG_COMPARE 3'b001, READ_MISS 3'b101, WRITE_HIT_CACHE_CONFLICT 3'b011, WRITE_MISS_ALLOCATE 3'b010, WRITE_MISS_ALLOCATE_WRITE 3'b110로 총 7 state가 존재하고, 1과 4는 필수 state로 그 안의 hit/miss/conflict + dirty bit을 이용하여 state를 이동하며 캐시 처리를 해준다.
- 1.3. 주어진 스켈레톤 코드는 Cache.v, InstMemory.v, DataMemory.v, CLOG2.v가 있고 추가로 만든 V 파일은 ALU.v, ALUControlUnit.v, ImmediateGenerator.v, ControlUnit.v, PC.v, mux5bit.v, mux.v, mux4.v, HazardDetection, forwardingUnit, top.v, cpu.v가 있다.
- 1.4. 테스트 코드는 naïve_matmul, optimal_matmul가 있으며, Ripes result txt 레지스터 값과 베릴로그 상에서 얻은 레지스터 값과 비교하며 정상적인 동작을 확인할 수 있다.
- 1.5. Pipelined CPU의 design과 implementation을 설명할 때 각각의 모듈이 synchronous인지 asynchronous인지 확인하며 각각의 스테이지를 설명하고자 한다.

2. Design



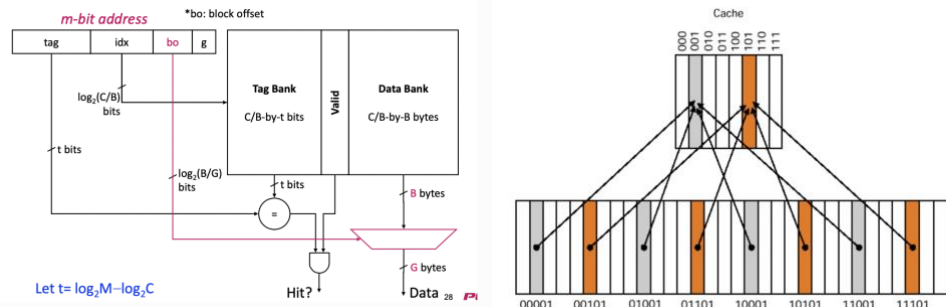


2.1. Pipelined CPU 동작이 이루어지는 과정에 대한 순차적인 설명은 다음과 같다.

- Instruction processing은 보통 5가지 절차로 이루어진다.
- 파이프라인 CPU는 HW 활용률(utilization) 향상을 통한 throughput 향상을 위해 한 stage씩 instruction이 쪼개서 작업이 진행되고, 여기서는 5개의 stage의 파이프라인을 가진다. => 5 stage pipelined cpu
 - IF : instruction fetch로, 인스트럭션 실행을 위해선 메모리로부터 읽어와야한다.
 - ◆ Instruction memory, PC 가 주로 수행한다.
 - ID : instruction decode로, 레지스터 파일에서 레지스터 값을 읽어오며 (operand fetch) instruction에 따른 control signal, immediate value를 발생시킨다.
 - ◆ ControlUnit, immediateGenerator, RegisterFile 가 주로 수행한다.
 - EX : execution으로, ALU 등을 통한 연산이 이루어진다.
 - ◆ ALU, ALUControlUnit 가 주로 수행한다.
 - MEM : Cache memory access를 하고, miss나 conflict 시 data memory access를 통해 접근하고자 하는 메모리를 읽고 쓰거나 dirty bit를 설정한다.
 - ◆ Cache, Data memory 가 주로 수행한다.
 - WB : write back으로, 레지스터 파일을 업데이트 하고 메모리에서 읽은 값을 rd에 저장하는 등의 과정이 이루어진다.
 - ◆ RegisterFile 가 주로 수행한다.
- 단, 이 과정에서 발생하는 data hazard는 stall과 data forwarding을 통해서 해결할 수 있는데, 이를 위해 hazard detection unit과 forwarding unit을 추가해서 조건에 따라 stall되거나 앞서 생성된 와이어값을 레지스터에 적용되지 않은 상태에서도 forwarding해서 가져와 쓸 수 있도록 하였다.
- 위와 같은 메커니즘으로 파이프라인 CPU는 작동된다.

2.2. The design of the cache (Directed mapped cache)

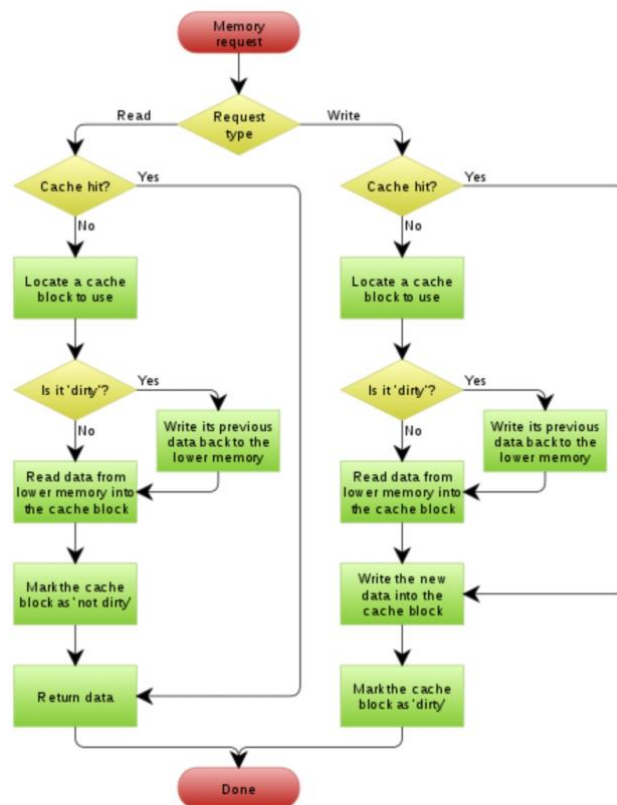
- Directed mapped cache는 특정 memory 주소는 특정 cache의 block에만 들어갈 수 있는 방식으로, 아래와 같은 구조를 가진다.



- **각 memory block은 하나의 single cache block에 mapping된다.**
- Tag bit를 통해 어떤 Index의 block data가 찾고 있는 data인지를 식별할 수 있다.
- 즉, tag bit와 Tag bank 값이 동일하면 hit가 되고, valid bit도 함께 확인해야 한다.
- mapping되는 location을 결정할 때, idx를 이용하는데, idx 사이즈는 $\log_2(C/G)$ bits (C =capacity, G =granularity)가 된다.
- 이 때, 데이터에 접근할 때 byte단위의 접근이 필요하고, 이를 위해 block offset이 필요하다.
- **Write back / Write through policy**를 이용했으므로, write Hit 발생시 cache만 계속해서 참조하면 되고, data memory와 달라도 dirty bit만 1로 설정해놓고 업데이트 하지 않는다. 또한, write miss가 발생하면 data memory를 업데이트하고 current cache에도 업데이트를 해 놓는다. 즉, miss된 data를 위한 공간을 캐시에 할당해 놓게 된다.

2.3. Explain replacement policy

- Directed mapped cache 방식을 이용했으므로 replacement policy가 따로 필요하지 않다. 각 memory block은 하나의 cache block에 매핑되므로, 해당하는 block의 값이 바뀌게 될 때 dirty bit의 세팅 여부에 따라 cache/dmem data를 교체해주어야 한다.
- 정확한 알고리즘은 다음과 같다.

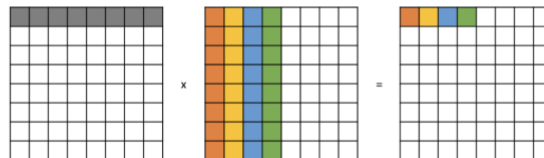


2.4. naïve_matmul vs opt_matmul (이론적 + 실험적 hit ratio)

```

// matrix op
for (int m = 0; m < M; ++m) {
  for (int n = 0; n < N; ++n) {
    for (int k = 0; k < K; ++k) {
      c[m][n] += a[m][k] + b[k][n];
    }
  }
}

```



- naïve 코드는 일반적인 행렬 곱셈 알고리즘으로, 3중 반복문을 사용하여 행렬 연산을 수행한다. 이 경우, 행렬의 element를 순차적으로 접근하지 않고, 행렬의 행과 열을 동시에 반복하면서 접근하기 때문에 cache locality가 좋지 않은 코드이며, 이는 곧 cache miss를 유발해 cache-friendly한 코드가 아니다.

```



// matrix op
for (int tile_m = 0; tile_m < M; tile_m += TILE) {
  for (int tile_n = 0; tile_n < N; tile_n += TILE) {
    for (int tile_k = 0; tile_k < K; tile_k += TILE) {
      for (int m = tile_m; m < tile_m + TILE; ++m) {
        for (int n = tile_n; n < tile_n + TILE; ++n) {
          for (int k = tile_k; k < tile_k + TILE; ++k) {
            c[m][n] += a[m][k] + b[k][n];
          }
        }
      }
    }
  }
}

```



- opt_matmul (tiled implementation) 코드는 행렬의 블록 단위로 접근하여 데이터의 locality를 활용할 수 있다. 행렬의 크기에 맞추어 block 크기를 선택하여 행렬의 블록 단위로 곱셈을 수행하고, 이미 업로드된 캐시의 값을 재사용할 수 있다. 블록 단위로 나누어 곱셈을 하는 방법은 matrix의 temporal locality를 활용하여 miss rate를 줄이는 대표적인 방법이다. 이는 곧 hit를 증가시켜 캐시 미스를 줄여 cache-friendly한 코드라고 할 수 있다.

- 우리가 작성한 코드를 통해 total access와 memory access를 출력해보았을 때 naïve에서와 opt에서의 값은 다음과 같다.

>  total_access[31:0]	6278	>  total_access[31:0]	6278
>  memory_access[31:0]	119174	>  memory_access[31:0]	128854

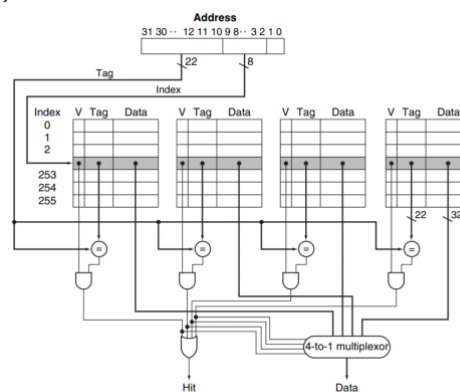
- **memory access**는 저장소 접근했을 때의 시간을 출력하고, 이는 곧 **write-back/write-through**의 기준으로 naïve에서의 hit rate가 더 크다는 결론을 얻을 수 있다.

2.5. Why is the cache hit ratio different between two matmul algorithms?

- 위에서 이론적으로는 언급한 것처럼 블록 단위로 접근하면 temporal locality를 가지고 있는 matrix를 최대한 활용하여 cache에 업로드된 값을 재사용할 수 있게 된다. 이는 곧 miss가 덜 발생한다는 것을 의미하고, hit ratio가 일반 행렬 곱보다 더 커지는 이유이다.
- 하지만, 위에서 작성한 코드로 직접 실행을 하였을 때 이론적으로 예측한 결과 **optimal**의 hit ratio가 더 많을 것이라고 추측했지만 naïve가 더 많음을 확인할 수 있다.
- **opt**의 더 hit ratio가 크기 위해서는 행렬의 크기가 캐시의 크기보다 훨씬 크다고 가정해야 타일을 나누는 것이 효과적이다. 하지만 이번 테스트 벤치의 **opt** 예시는 캐시와 행렬의 크기가 거의 동일했기 때문에 **cache-friendly**한 효과를 얻지 못했다고 말할 수 있다.

2.6. what happens to the cache hit ratio if you change the # of sets and # of ways?

- 캐시 라인을 페이지 단위로 묶은 것을 Way라고 하고, 각 Way 내에 있는 같은 인덱스를 묶은 것을 Set이라고 한다.
- way수를 증가시키면 더 많은 tag-data bank를 가져 데이터가 매핑될 수 있는 방법의 수가 늘어난다. 특정 set에 속하는 data bank 및 매핑 방법의 수가 증가하는 것이다. 즉, 같은 곳에 매핑될 가능성이 줄어들어 **conflict**의 가능성이 줄어들고, **hit rate가 증가한다**. (hit이 일어날 수 있는 방법의 가지수도 많아 이를 위한 overhead gate도 존재하게 된다)



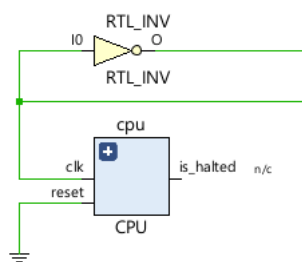
2.7. 다음은 각각의 모듈에 대한 **synchronous/asynchronous 디자인의** 포괄적인 설명이다.

Cpu	Top module로서 sub module들의 인스턴스 선언과 wire 변수들의 연결이 이루어지는 곳이다.
ALU	ALU 연산을 한다. ALU operation의 종류와, ALU 연산을 할 input 2개를 asynchronous하게 읽어와서, 연산을 할 result와, branch operation일 경우에는 branch condition에 해당하는 지를 리턴한다.
ALUControlUnit	Instruction를 해독하여 ALU operation을 내보내주는 역할을 한다. Instruction을 asynchronous하게 읽어온다.
ControlUnit	Asynchronous하게 Instruction을 읽어와, 이에 해당하는 CPU의 동작을 확인하여, 각 module에 내보내준다.
Inst Memory	Instruction memory에서는 asynchronous하게 instruction을 읽어오며 CPU Reset시 synchronous하게 초기화가 이루어진다.
Cache	우선 synchronous하게 메모리를 초기화해주고, asynchronous하게 cache status (state)를 설정해준다. 그에 따라 해당 state에서 Valid bit, Dirty bit, Data Bank, Tag bank, dmem_addr, dmem_read, dmem_write 등을 설정해주며 각 state의 역할을 asynchronous하게 수행한다.
Data Memory	Data memory에서는 asynchronous하게 memory에서 data(dout)를 읽어오며 synchronous하게 data(din)를 write한다. 마찬가지로 CPU Reset시 synchronous하게 초기화가 이루어진다. is_output_valid 및 request_arrived도 assign문을 통해 asynchronous하게 설정된다.
Mux	selector가 0이면 output으로 mux1을 assign하고, 아니면 output으로 mux2를 assign한다. Dataflow modeling이다.
Mux4	Selector가 00이면 output으로 mux1을 assign하고, 01이면 mux2를 assign하고, 10이면 mux3를 assign한다. Dataflow modeling이다.
Mux5bit	Selector가 0이면 output으로 5비트 mux1을 assign하고, 아니면 mux2를 assign한다. Dataflow modeling이다.
Opcodes	연산들의 funct3과 opcode를 선언해 놓았다.
PC	Synchronous하게 매 clk마다 current_pc를 next_pc(pc+4)의 값으로 업데이트 해준다.
RegisterFile	register file의 rs1 rs2 값을 asynchronous하게 읽어오고 write_enable = 1이고 rd가 0이 아니라면 synchronous하게 rd에 write한다. CPU Reset시 synchronous하게 초기화가 이루어진다.
hazardDetection	hazard detection 조건을 만족한다면 Asynchronous하게 PCwrite, IF_ID_write, hazard_out signal을 업데이트해준다.
forwardingUnit	input인 rs1EX, rs2EX, rdMEM, regWriteMEM, rdWB, regWriteWB가

	바뀔에 따라 asynchronous하게 forwarding이 어떻게 되는지를 내보낸다.
Top	CPU instance를 선언하고, 처음에 reset을 한 뒤, clk를 특정 시간 간격마다 -clk로 바꾸어주어 synchronous한 동작에 도움을 준다. 프로그램이 끝나면, total_cycle과 register의 상황을 출력해 준다.

3. Implementation

3.1. <cpu.v> = top module



Input : clk, reset

Output : is_halted

사용한 wire/Reg 변수들의 설명은 다음과 같다.

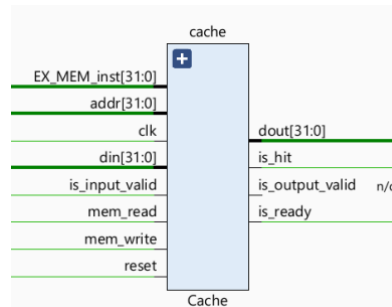
PC	nextPC : 다음 clock의 PC currentPC : 현재 PC real_currentPC : mux를 거치고 난 현재 PC beforeInst : Instruction memory를 나온 dout
ALU	alu_op : 결정된 ALU operation real_ALU_inA : ALU에 들어가는 최종 첫 번째 operand real_ALU_inB : ALU에 들어가는 최종 두 번째 operand alu_bcond : 여기서 항상 0 = alu_zero ALU_out : ALU output
register	WriteRegister : RD RegReadData1 : RS1 RegReadData2 : RS2 MemData : memory data rf17 : Register file [17] <- 레지스터파일 사용X
memory	real_DataOutput : mux를 거치고 난 data output DataReadData : mux를 거치기 전 data output real_real_DataReadData : 2번째 mux를 거치고 난 data output
Immediate generator	Imm_gen_out : immediate generator output

& mux step	
Control unit	<p>isJal : Jal instruction인지 여부 output</p> <p>isJalr : Jalr instruction인지 여부 output</p> <p>isBranch Branch instruction인지 여부 output</p> <p>RegWrite : 레지스터에 쓰는지 여부 output</p> <p>MemRead : 메모리를 읽는지 여부 output</p> <p>MemToReg : 메모리의 값을 레지스터로 가져오는지 여부 output</p> <p>MemWrite : 메모리에 쓰는지 여부 output</p> <p>ALUSrc : ALU 연산에 immediate가 활용되는지 여부 output</p> <p>PCtoReg : PC가 register에 저장되는지 여부 output(여기서는 항상 0)</p> <p>Is_ecall : ecall instruction인지 여부 output</p>
Forwarding unit	<p>forwardingA : A를(rs1) 어디서 가져오는지를 나타냄</p> <p>forwardingB : B를(rs2) 어디서 가져오는지를 나타냄</p>
Hazard detection	<p>IF_ID_write : IF_ID 파이프라인 레지스터에 write 가능 signal</p> <p>PCWrite : 다음 PC 가져오기 가능 signal</p> <p>Hazard_out : hazard detection 여부 signal</p> <p>afterhaltingMuxrs1 : (ecall 연산일 경우 rs1을 17로, 아닐 경우 기존의 instruction에서 rs1을 가져오게 된다.) CPU에서 다룰 최종적인 rs1의 값이라 할 수 있다.</p>
Branch predictor	<p>branchPC : taken했을 때 뛰어야하는 PC값을 저장한다.</p> <p>isTaken : taken 여부를 설정하는 핵심 파라미터이다.</p>
Cache	<p>is_input_valid : Cache가 사용되는지 여부를 나타낸다.</p> <p>is_output_valid : cache output값이 valid한지 여부를 나타낸다.</p> <p>is_hit : cache가 mem stage에서 hit 되었는지를 나타낸다.</p> <p>is_ready : cache가 state를 다 지나서 나올 준비가 되었는지를 나타낸다.</p> <p>Cachestall : cache가 stall되어야 하는지 여부를 나타낸다.</p> <p>Cachebeforeinst : cache에 들어가기 전의 inst를 나타낸다.</p>
Pipeline (파이프라인 레지스터 / 설명은 생략한다)	<p>IF_ID_inst :</p> <p>IF_ID_currentPC:</p> <p>ID_EX_alu_op :</p> <p>ID_EX_alu_src :</p> <p>ID_EX_mem_write</p> <p>ID_EX_mem_read</p>

	ID_EX_mem_to_reg
	ID_EX_reg_write
	ID_EX_rs1_data
	ID_EX_rs2_data
	ID_EX_rs1
	ID_EX_rs2
	ID_EX_imm
	ID_EX_ALU_ctrl_unit_input
	ID_EX_rd
	ID_EX_currentPC
	ID_EX_isHalted
	ID_EX_isEcall
	ID_EX_isJal
	ID_EX_isJalr
	ID_EX_isBranch
	ID_EX_inst
	EX_MEM_mem_write
	EX_MEM_mem_read
	EX_MEM_is_branch
	EX_MEM_mem_to_reg
	EX_MEM_reg_write
	EX_MEM_alu_out
	EX_MEM_dmem_data
	EX_MEM_rd
	EX_MEM_bcond
	EX_MEM_SUM_out
	EX_MEM_isHalted
	EX_MEM_jump_rdin
	EX_MEM_inst
	MEM_WB_mem_to_reg
	MEM_WB_reg_write
	MEM_WB_mem_to_reg_src_1
	MEM_WB_mem_to_reg_src_2
	MEM_WB_isHalted
	MEM_WB_isJal
	MEM_WB_isJalr

3.2. <Cache.v>

- cache



Input : reset, clk, addr, mem_read, mem_write, din, EX_MEM_inst

Output : is_ready, is_output_valid, dout, is_hit

- Asynchronous logic : is_input_valid가 1이 되면 asynchronous하게, next_cache_status가 변화하고, 이에 따라 cache_status가 변화하면 그에 맞게 asynchronous logic이 실행된다.
- (Positive clk) Synchronous logic : reset버튼이 눌러졌을 때 Tag_Bank, Data_Bank, Valid, Dirty와 같은 캐시 초기화가 이루어지고, 이후 cache_status에 next_cache_status를 대입하는 부분은 synchronous하게 실행된다.
- Implementation :
캐시의 status는 크게 CACHE_STOP, CACHE_END, TAG_COMPARE, READ_MISS, READ_MISS_WRITE, READ_MISS_2, WRITE_HIT_CACHE_CONFLICT, WRITE_MISS_ALLOCATE, WRITE_MISS_ALLOCATE_WRITE로 나뉜다.

[CACHE_STOP] 캐시가 중지된 상태로, is_input_valid일 때, TAG_COMPARE state로 넘어간다.

[CACHE_END] 캐시가 끝난 상태로, is_ready를 1로 바꾸어준다.
(CACHE_STOP로 넘어가기 전 마무리 단계)

[TAG_COMPARE] set_index에 맞는 tag와 valid bit를 확인하여 캐시의 hit, miss 여부를 확인한다.

이때, read_hit인 경우, dout는 캐시의 Data_Bank에서 가져올 수 있고, CACHE_END state로 넘어간다.

read_miss인 경우, 해당하는 블록의 dirty bit가 1이면 값이 있는 것이므로, 그 값을 지키기 위해 dmem에 써 주는 별도의 state인 READ_MISS_WRITE로 넘어간다. 아닌 경우에는 READ_MISS state로 넘어간다.

write_hit인 경우, 해당하는 블록의 dirty bit가 0이면, din된 값을 블록의 올바른 위치에 써 주고, dirty bit를 1로 바꾸어준다. CACHE_END state로 넘어간다. dirty bit가 1이면, cache conflict가 일어난 것으로, WRITE_HIT_CACHE_CONFLICT state로 넘어간다.

write_miss인 경우, 해당 위치에 써 주어야 하는데, dirty bit가 1이면 값이 있는 것으로 그 값을 지키기 위해 dmem에 써 주는 별도의 state인 WRITE_MISS_ALLOCATE_WRITE state로 넘어간다. dirty bit가 0이면 WRITE_MISS_ALLOCATE state로 넘어간다.

[READ_MISS_WRITE]

read_miss인데, dirty인 경우, 원래 캐시에 있던 블록을 dmem에 넣어주고, READ_MISS_2 state로 이동한다.

[READ_MISS]

dmem_ready가 1이면, dmem에서 값을 읽어온다. 값을 읽어왔으면, 그 값을 캐시에 할당하고, CACHE_END state로 넘어간다. 이때, Dirty bit는 0이다.

[READ_MISS_2]

READ_MISS와 마찬가지로 값을 캐시에 할당한다. 이때 Dirty bit는 1이다. 왜냐하면, 앞의 READ_MISS_WRITE state에서 다른 값을 넣어줬기 때문에 메모리와 캐시의 값이 다르기 때문이다.

[WRITE_HIT_CACHE_CONFLICT]

dmem에 캐시의 원래 블록의 값을 써 준다. 그리고, 원래 써 주어야 할 din 값을 Data_Bank에 써 준다. CACHE_END state로 넘어간다.

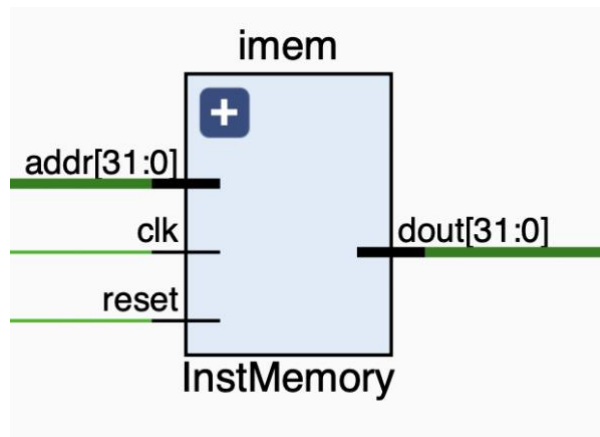
[WRITE_MISS_ALLOCATE_WRITE]

dmem에 캐시의 원래 블록의 값을 써 준다. WRITE_MISS_ALLOCATE state로 넘어간다.

[WRITE_MISS_ALLOCATE]

dmem의 데이터를 읽어오고, 그곳에 써 주어야 할 값을 써 준다.

3.3. <InstMemory.v>

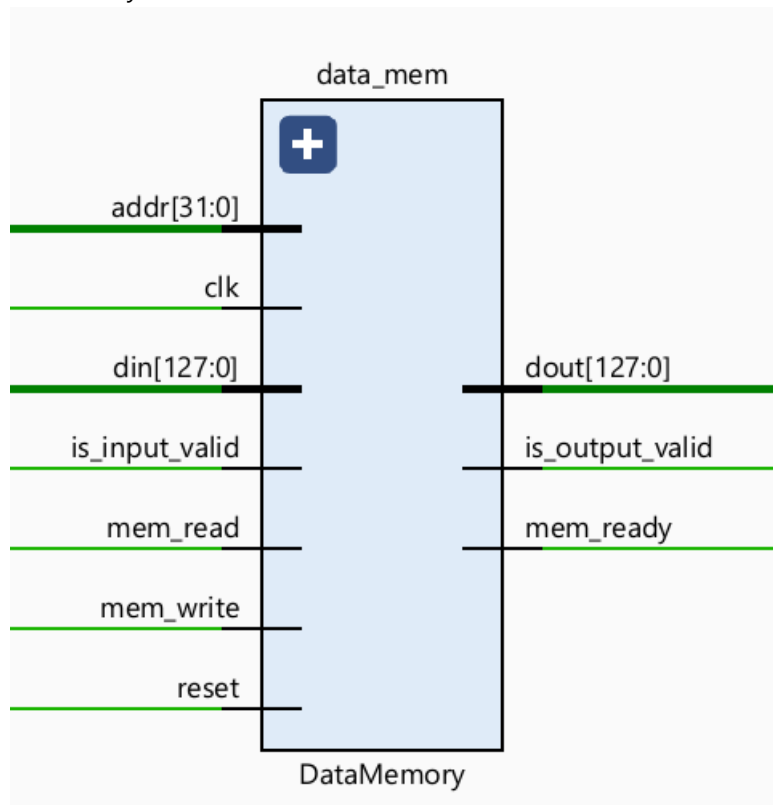


Input : reset, clk, addr

Output : dout

- Asynchronous logic : imm_addr를 이용해 memory에 access해 dout에 담는 과정을 addr가 변할 때마다 asynchronous하게 진행하였다.
- (Positive clk) Synchronous logic : reset버튼이 눌러졌을 때 instruction memory를 0으로 초기화하는 과정은 clk이 positive일 때마다 synchronous하게 진행하였으며, pc값이 돌아올때마다 각각의 테스트 코드에 있는 memory 경로의 값을 읽어오는 과정도 clk에 따라 진행되었다

3.4. <DataMemory.v>

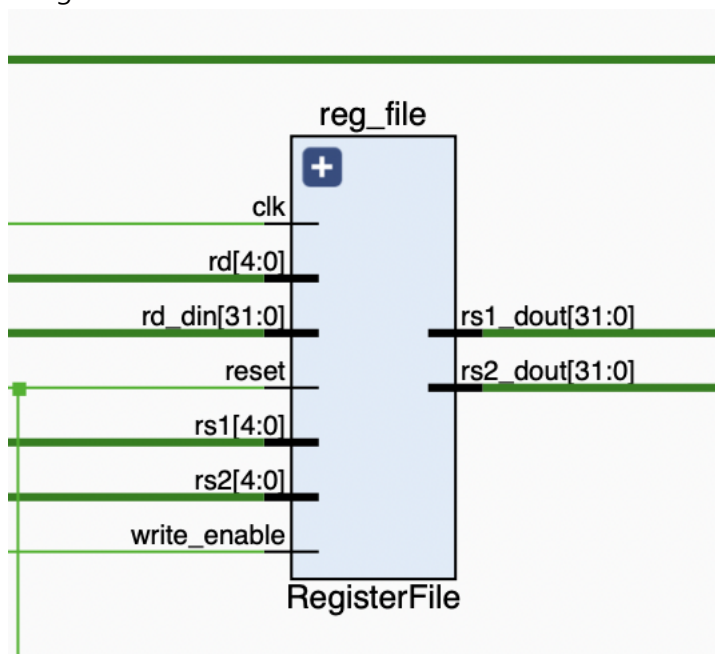


Input : reset, clk, is_input_valid, addr, din, mem_read, mem_write

Output : dout, is_output_valid, mem_ready

- Asynchronous logic : mem_read signal이 1이면 dout에 mem[dmem_addr]를 대입하는 과정은 assign의 오른쪽 값이 바뀌면 대입되는 방식을 이용했으므로 asynchronous하게 진행된다고 볼 수 있다. 마찬가지로 request_arrived, is_output_valid 값을 세팅하는 과정도 asynchronous하게 진행되었다.
- (Positive clk) Synchronous logic : positive clk일 때 mem_write signal이 1인지 확인하고 din값을 memory에 write하는 과정과 reset이 1일때 data memory를 초기화하는 과정은 synchronous하게 진행된다. delay counter를 1씩 감소시키는 과정도 synchronous하게 진행되었다.

3.5. <RegisterFile.v >



Input : reset, clk, rs1, rs2, rd, rd_din, write_enable

Output : rs1_dout, rs2_dout

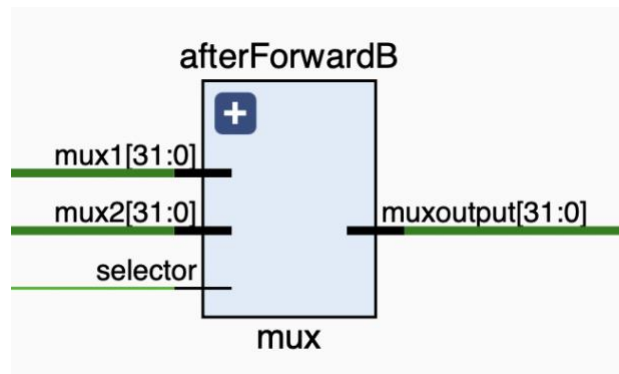
- Asynchronous logic : rf[rs1], rf[rs2]값이 변할 때마다 rs1_dout, rs2_dout에 assign문을 통해 대입해주는 과정은 asynchronous하다.
- (Positive clk) Synchronous logic : positive clk일 때마다 rf[0]의 값은 변하면 안되므로 rd가 0이 아니고 write enable이 1인지를 확인해 rd_din을 레지스터 파일에 대입해주는 과정은 synchronous하다. 마찬가지로 reset이 1일 때 레지스터 파일을 0으로 초기화해주는 과정또한 synchronous하다.

3.6. <mux.v>

mux1과 mux2와 selector를 input으로 받아서 selector가 0이면 mux1을, 1이면

mux2를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분들은 다음과 같다.

- AfterForwardB

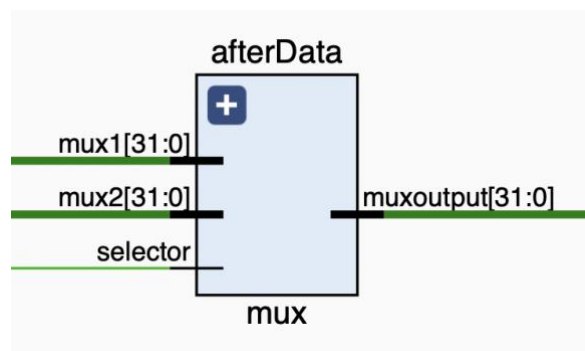


Input : mux1 (real_ID_EX_rs2_before), mux2 (ID_EX_imm), selector(ID_EX_alu_src)

Output : muxoutput(real_ALU_inB)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. Real_ID_EX_rs2_before와 ID_EX_imm 사이에서 real_ALU_inB signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

- afterData



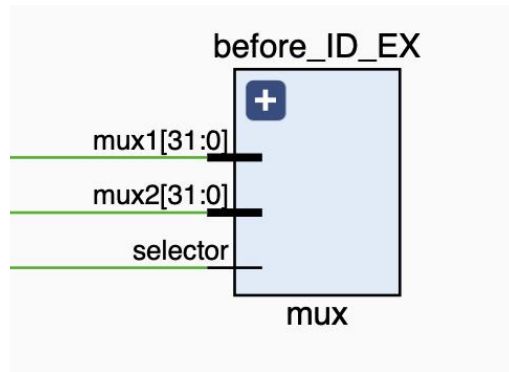
Input : mux1 (MEM_WB_mem_to_reg_src1), mux2 (MEM_WB_mem_to_reg_src_2),

selector(MEM_WB_mem_to_reg)

Output : muxoutput(real_DataReadData)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다.
MEM_WB_mem_to_reg_src1과 MEM_WB_mem_to_reg_src_2 사이에서
MEM_WB_mem_to_reg signal을 통해 한 값만 내보내주는 조건문 역할을 해준다.

- Before_ID_EX

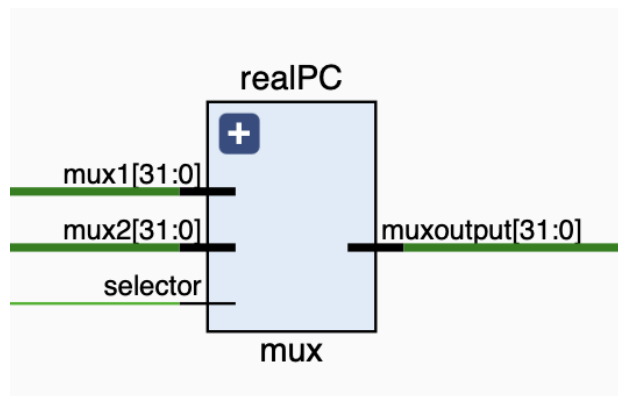


Input : mux1 (control_out), mux2 (0), selector(hazard_out)

Output : muxoutput(real_pipeline_signal)

- Asynchronous logic : clk에 의존적인 Logic이나 initial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. Control_out과 0 사이에서 hazard_out selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

- realPC

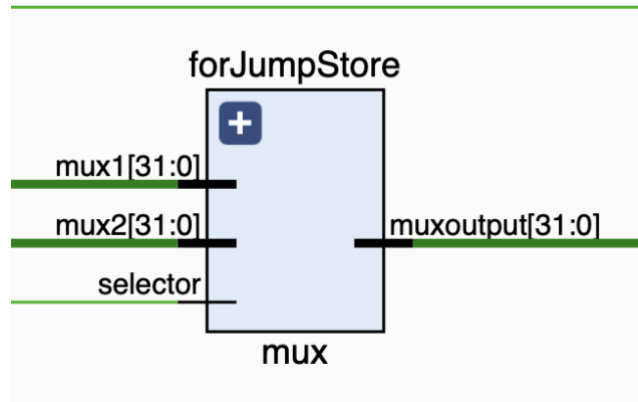


Input : mux1 (currentPC+4), mux2(branchPC), selector(isTaken)

Output : muxoutput (nextPC)

- Asynchronous logic : clk에 의존적인 Logic이나 initial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. current+4과 branchPC사이에서 isTaken selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

- forJumpStore



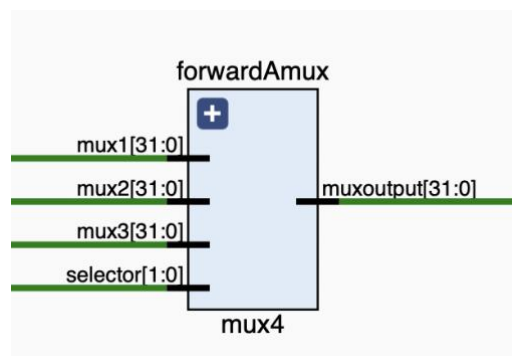
Input : mux1 (real_DataReadData), mux2(EX_MEM_jump_rdin),
 selector(MEM_WB_isJal || MEM_WB_isJalr)
 Output : muxoutput (real_real_DataReadData)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. Real_DataReadData과 EX_MEM_jump_rdin사이에서 MEM_WB_isJal || MEM_WB_isJalr selector를 통해 한 값만 내보내주는 조건문 역할을 한다.

3.7. <mux4.v>

mux1과 mux2와 mux3과 selector를 input으로 받아서 selector가 00이면 mux1을, 01이면 mux2를 10이면 mux3를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분들은 다음과 같다.

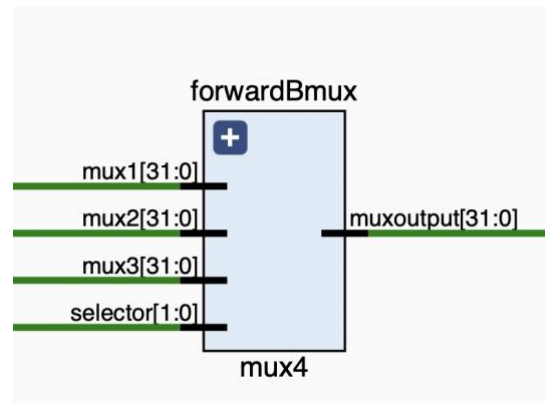
● forwardAmux



Input : mux1 (ID_EX_rs1_data), mux2 (EX_MEM_alu_out), mux3(real_DataReadData),
 selector(forwardingA)
 Output : muxoutput(real_ALU_inA)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ID_EX_rs1_data와 4와 EX_MEM_alu_out와 real_DataReadData중에서 forwardingA signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

- forwardBmux



Input : mux1 (ID_EX_rs2_data), mux2 (EX_MEM_alu_out), mux3(real_DataReadData), selector(forwardingB)

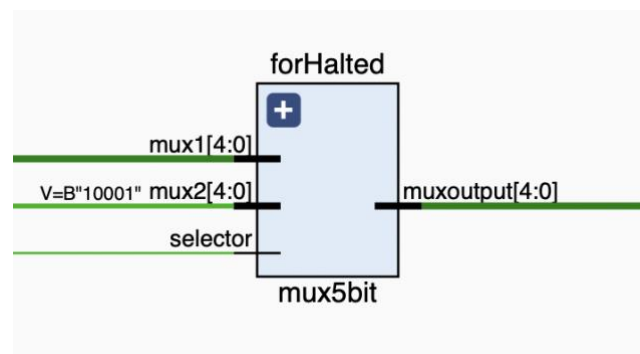
Output : muxoutput(real_ID_EX_rs2_before)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. ID_EX_rs2_data와 4와 EX_MEM_alu_out와 real_DataReadData중에서 forwardingB signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

3.8. <mux5bit.v>

5 bit mux1과 5 bit mux2와 selector를 input으로 받아서 selector가 0이면 mux1을, 1이면 mux2를 출력해주는 모듈이다. mux module을 사용하여 instance를 만든 부분은 다음과 같다.

- forHalted



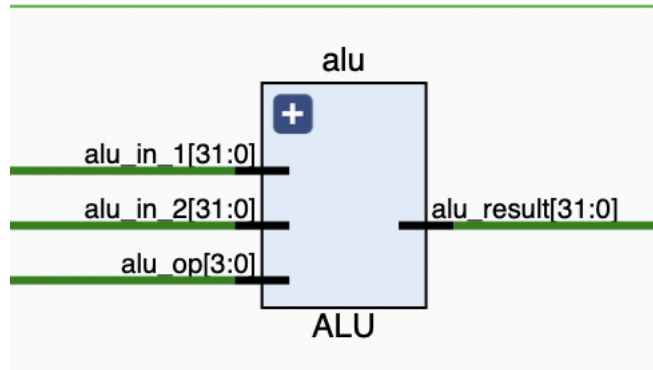
Input : mux1 (IF_ID_inst[19:15]), mux2 (5'b10001), selector(isEcall)

Output : muxoutput(aftergaltingMuxrs1)

- Asynchronous logic : clk에 의존적인 Logic이나 intial을 이용하는 구간이 없고 assign문을 이용했으므로 asynchronous한 로직만 있다. IF_ID_inst[19:15]와 17 중에서 isEcall signal에 따라 한 값만 내보내주는 조건문 역할을 한다.

3.9. <ALU.v>

● ALU



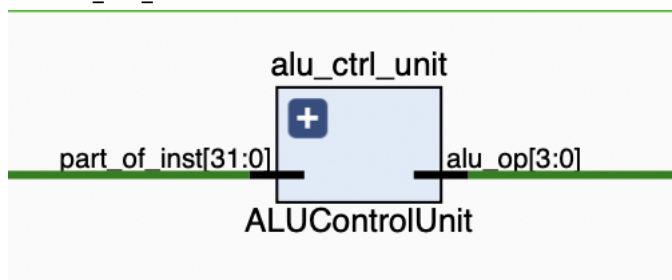
input : alu_in_1, alu_in_2, alu_op

output : alu_bcond, alu_result

- Asynchronous Logic : switch-case문을 이용해 alu_op 4비트에 따라 alu_result의 연산이 달라지도록 구현하였다. 특히 branch instruction의 경우 op 4비트 MSB가 1인 특징을 지니며, BEQ BNE BLT, BGE 의 조건에 따라 alu_bcond가 설정된다. 이들은 asynchronous하게 연산된다.

3.10. <ALUControlUnit.v>

● alu_ctrl_unit



input : part_of_inst

output : alu_op

- Asynchronous Logic : input인 part_of_inst가 바뀔 때마다 asynchronous하게 동작하여, 각 part_of_inst에 맞는 alu_op를 대입한다.
- Implementation : part_of_inst를 가공해서 input으로 넣을 수도 있었겠지만, 가공하지 않고 32bit의 whole instruction을 input으로 넣었다.
우선, opcode에 해당하는 part_of_inst[6:0]의 값에 따라 연산의 종류에 대한 case를 구분했다.
ARITHMETIC 연산일 때는, ALU 연산에 해당하는 부분인 part_of_inst[14:12]와,

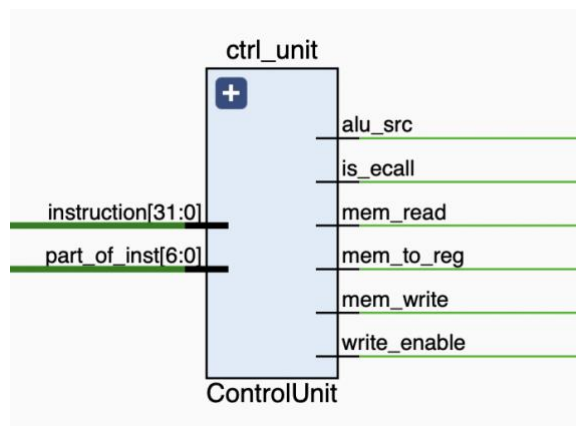
part_of_inst[30]을 추가로 검토해 주었다. add와 sub는 funct3이 일치하고, sub 연산만 part_of_inst[30]이 1이기 때문에 따로 구분했다.

ARITHMETIC_IMM 연산일 때도, ALU 연산 부분은 part_of_inst[14:12]를 사용하여 alu operation을 구분해 주었다.

LOAD, JALR, STORE 연산일 때의 ALU 연산은 add이다. 왜냐하면 각각 ALU unit에서 immediate value와의 연산 부분이 있기 때문이다. (JAL은 별도의 Add unit을 활용하여 ALU 연산이 존재하지 않는다.)

BRANCH 연산일 때는 part_of_inst[14:12]를 사용하여 별도로 alu_operation을 구분해 주어야 했다. 그 이유는, ALU unit에서 rs1과 rs2에 따른 branch condition도 계산해야 하기 때문이다.

3.11. <ControlUnit.v>



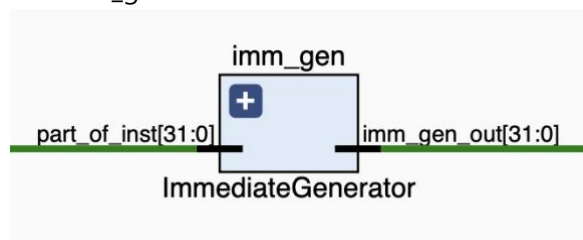
input : instruction, part_of_inst

output : alu_src, is_ecall, mem_read, mem_to_reg, mem_write, write_enable

- Asynchronous logic : input인 instruction이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction에 맞는 control signal을 blocking assignment로 대입한다. 자세한 구현은 lab2의 single-cycle CPU에서의 구현과 일치한다.

3.12. <ImmediateGenerator.v>

- imm_gen



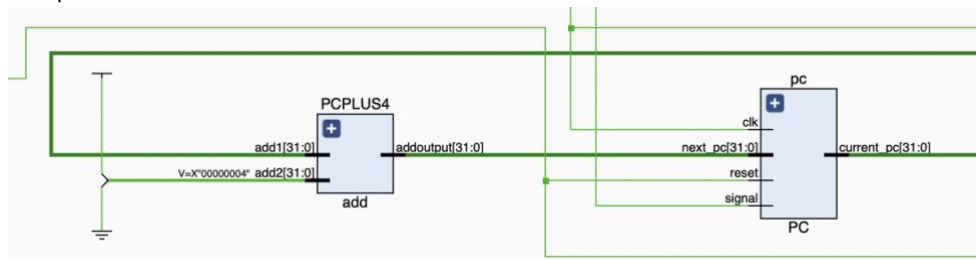
input : part_of_inst

output : imm_gen_out

- Asynchronous Logic : input인 `part_of_inst`이 바뀔 때마다 asynchronous하게 동작하며, 각 instruction을 해독하여 내부에 있는 immediate value를 찾아낸다.
- Implementation : 우선, opcode인 `part_of_inst[6:0]`을 통해 명령어의 종류를 구분했다. 각 명령어들에 대하여 RISC-V 규정에 따라 immediate value를 찾을 수 있다. 자세한 값은 Introduction에 있는 1.5의 표를 참조하였다. STORE와 BRANCH, JAL 명령의 경우에는 sign-extension을 해 주었다.

3.13. <PC.v>

- pc



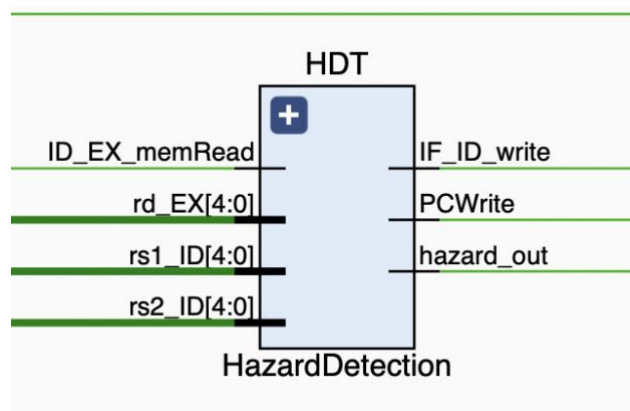
input : `clk`, `next_pc`, `reset`, `signal`

output : `current_pc`

- (Positive `clk`) Synchronous Logic : positive `clk`일 때마다 우선 `reset`이 1일 경우 `current_pc`를 0으로 초기화해준다. 아닐 경우 `next_pc(pc+4)`를 `current_pc`에 넣어 준다. 단, 이 과정은 `signal` (`PCWrite`)이 1일 때만 시행한다. (여기서 첫번째 `clk`에도 작동하도록 `current_pc == 0` 일 때도 `pc+4`값을 받도록 임의로 설정했다. 이는 Non-controlflow lab이기에 가능한 것이며, 다음 랩에서 수정할 예정이다.)

3.14. <HazardDetection.v>

- HDT

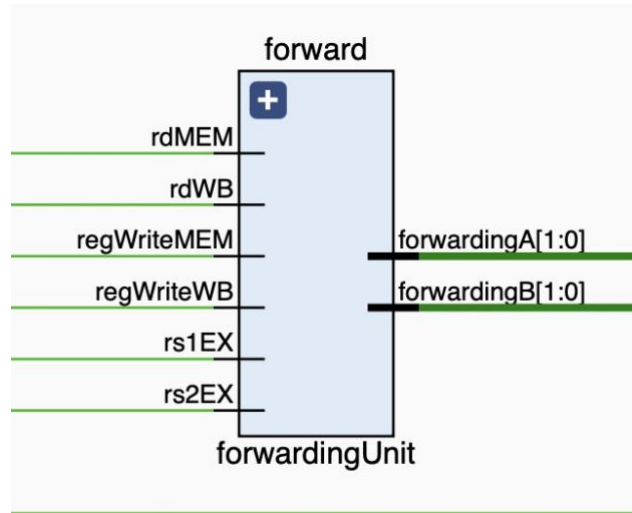


Input : `ID_EX_memRead`, `rd_EX`, `rs2_ID`, `rs2_ID`

Output : `IF_ID_write`, `PCWrite`, `hazard_out`

- Asynchronous logic : hazard detection 조건을 만족하는 경우, PCWrite 를 0으로, IF_ID_write를 0으로, hazard out을 1로 설정해 업데이트해준다.

- <ForwardingUnit.v>



input : rdMEM, rdWB, regWriteMEM, regWriteWB, rs1EX, rs2EX

output : forwardingA, forwardingB

- Asynchronous logic : input 값들의 변화에 따라 asynchronous하게 forwardingA, forwardingB에 값을 넣어준다. 기본적으로 forwardingA,B의 값은 0이다. 자세한 구현은 본 보고서의 2.4에 있다.

4. Discussion

- 4.1. hazard_out과 cache stall의 우선순위에 관한 문제
hazard_out와 IsTaken의 우선순위는 cache stall보다 뒤에 있어야 하는데, reset과 같은 위치에 hazard_out을 두는 바람에 엉뚱한 곳에서 forwarding되는 문제가 생겼었다.
- 4.2. Internal forwarding Register file 사용에 의한 forwarding의 문제
Lab5의 registerfile은 internal forwarding을 구현하고 있어 기존에 올바르게 forwarding되던 instruction의 rs1이 다른 수로 포워딩되는 문제가 발생해 EX_MEM_alu_out 이 잘못 기재되는 문제가 생겼다. 이는 Lab4의 registerfile을 이용함으로써 해결할 수 있었다.
- 4.3. Cache의 state 구분에 관한 문제
이 캐시는 write-allocate, write-back policy를 가진다. 따라서, write miss시 메인 메모리의 블록을 캐시 메모리에 할당하여 캐시에 write하고, write될 때는 캐시에만 write되었다가, dirty가 1인데 write를 해야 할 때는, 캐시 메모리의 값을 메인 메모리에 write하고, 다시 값을 써 주어야 한다. 이러한 policy를 지키기 위해 캐시의 state를 구분하여 쓰는 데 어려움이 있었다.
- 4.4. READ_MISS, WRITE_MISS 처리 시, dmem에 write를 해 주어야 하는 문제

miss일 때는 dmem의 값을 cache에 가져와 주어야 한다. 이때, dirty bit를 활용하여 cache에 dmem과 다른 값이 존재하고 있었다면, 이를 잃어버리지 않도록 이 값을 dmem에 잘 저장해 주어야 한다. 또한, 이후에 dirty bit setting에도 주의해야 한다.

5. Conclusion

- 5.1. 수업시간에 배운 5-stage pipelined CPU를 직접 구현하면서 CPU의 동작을 이해하고, 컴퓨터처럼 사고하는 능력을 배울 수 있었다.
- 5.2. Cache를 이용해서 Write-back / Write-through 방식을 이용해 hit/miss를 구현하였는데, dirty bit에 대한 이해와 언제 업데이트 되는지에 대한 이론적 모호함이 캐시를 구현하면서 사라질 수 있었다.
- 5.3. cache를 사용했을 때 어떤 locality의 코드를 가지고 있느냐에 따라 hit / miss rate의 여부가 차이가 나게 되고, cycle 및 latency의 차이가 남을 직접 두 가지의 테스트 벤치 코드를 돌려보며 알 수 있었다.