2025.08.05

대한상공회의소
서울기술교육센터

RISC-V RV32I PipeLine

-Hazard Hardware Solve-

AI 시스템 반도체 설계 2기 | 최윤석 | 대한상공회의소 서울기술교육센터

CONTENTS

01 RISC-V PipeLine 개요

02 개발환경 및 언어

03 블록다이어그램

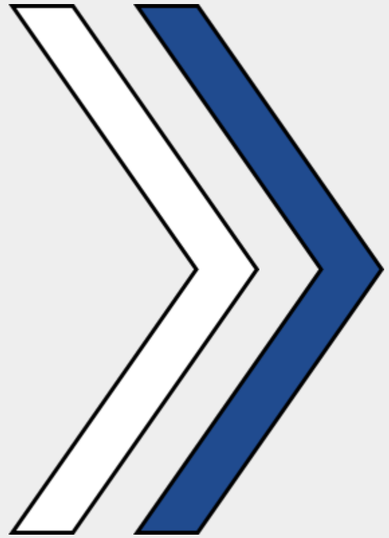
04 Hazard Hardware Solve

05 Simulation Result

06 결론 및 느낀점

01

RISC-V PipeLine 개요



- 개발 이유
- 설계 목표

RISC-V PipeLine 개요

개발 이유

- 멀티사이클 속도향상이 적음
- 멀티사이클 속도 X 스테이지수(5 stage) = 최대 5배 향상된 속도
- CPU 동작과 흐름 이해 및 하드웨어 구현 능력 UP

설계 목표

- Data Hazard, Control Hazard 해결
- 가장 작은 clk 손해 하드웨어 구현
- C언어, 어셈블리어 테스트

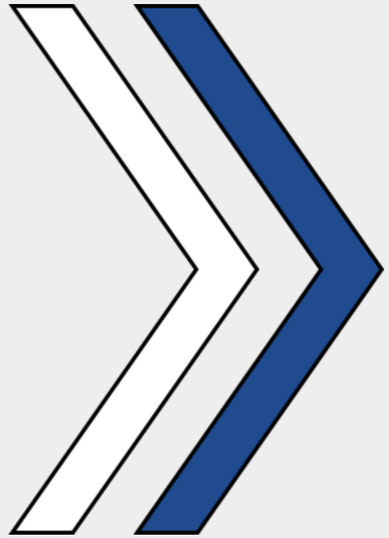
02
개발환경 및 언어

- SystemVerilog, Vivado, FPGA(baysys 3)

02 개발환경 및 언어

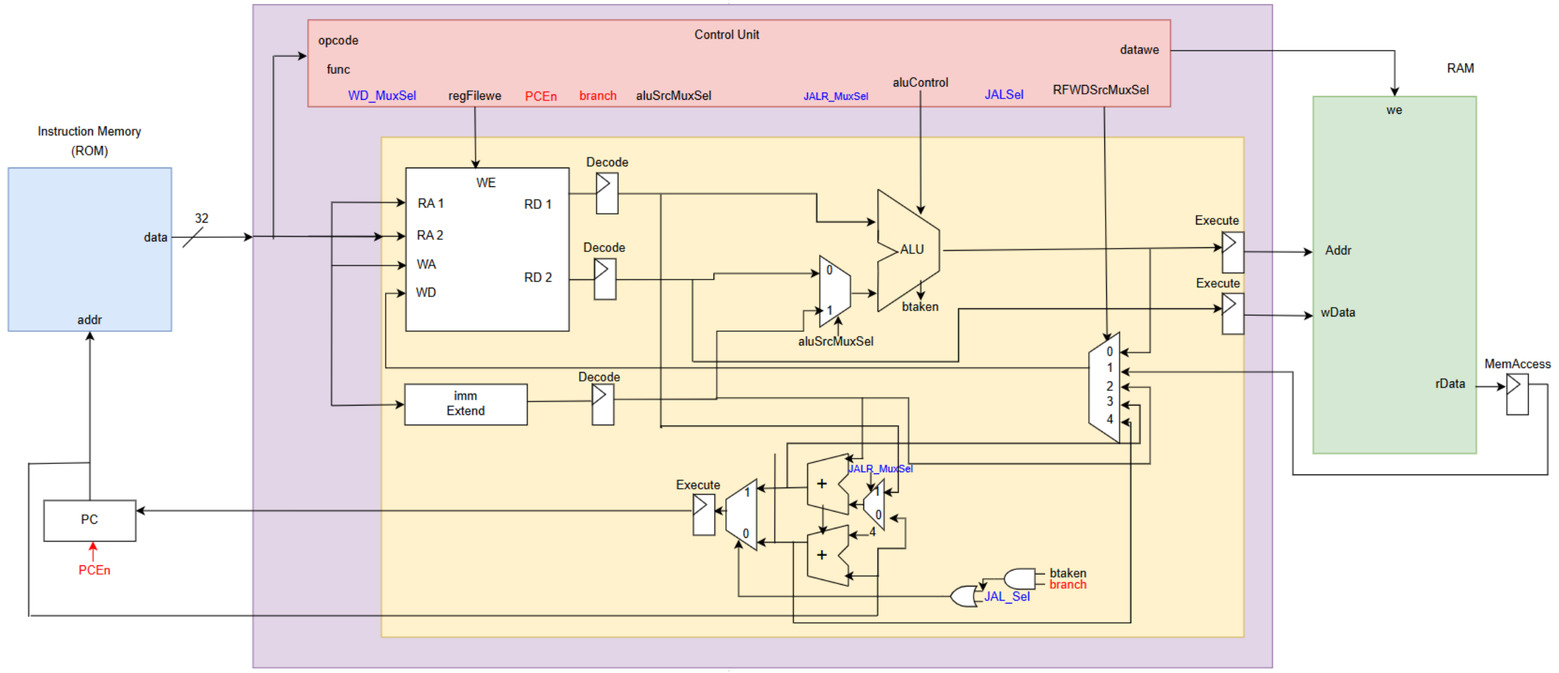
- 개발 언어 : SystemVerilog
- 시뮬레이션 환경 : Vivado
- FPGA : baysys 3

03
블록 다이어 그램



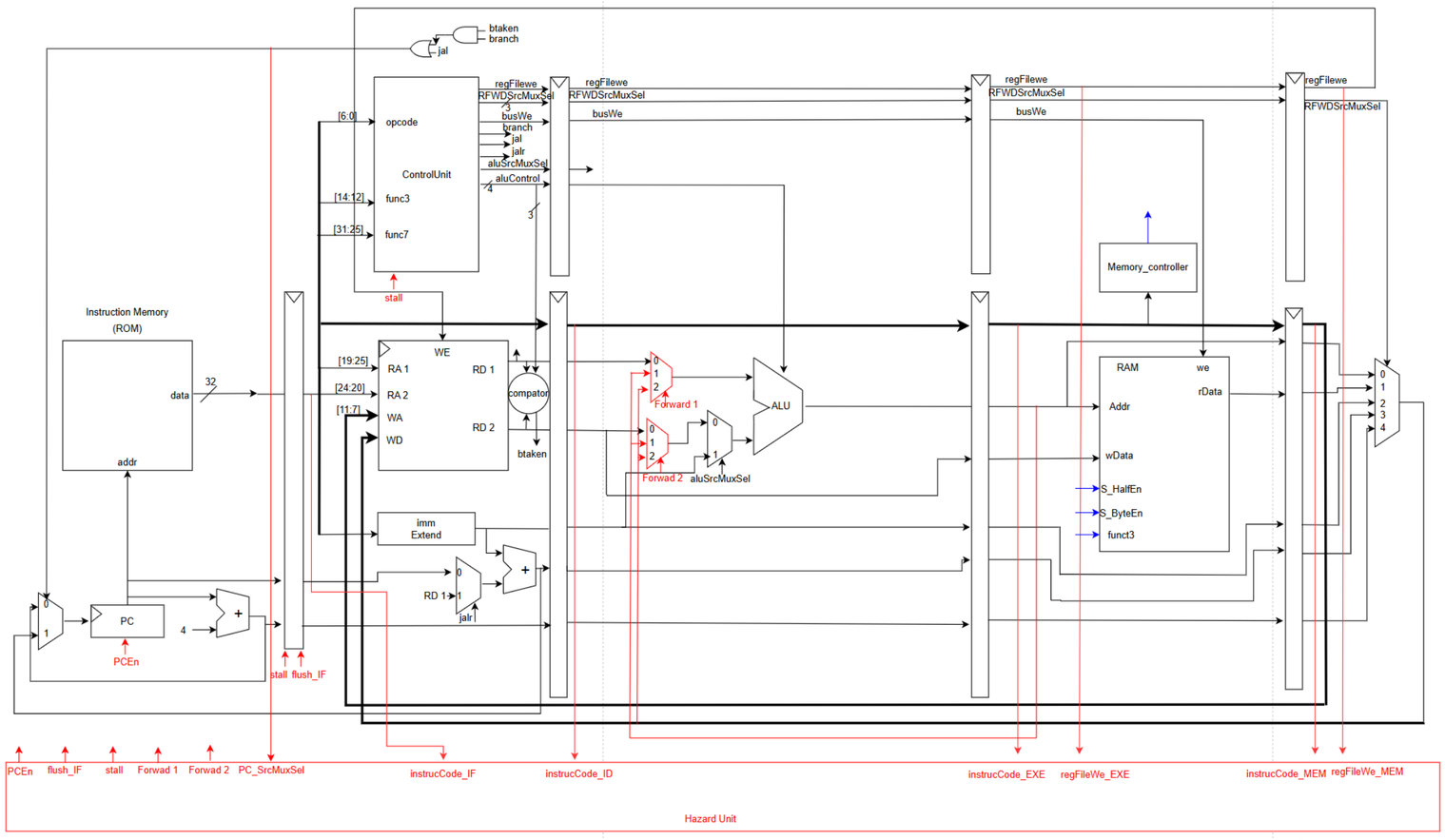
- block diagram, RTL 코드 분석
- RTL 시뮬레이션 결과

02 블록 다이어 그램(Multi Cycle)



- Control신호 한번에 처리됨
- stage 구간별 Data 처리가 힘듬
- 코드 가독성 떨어짐

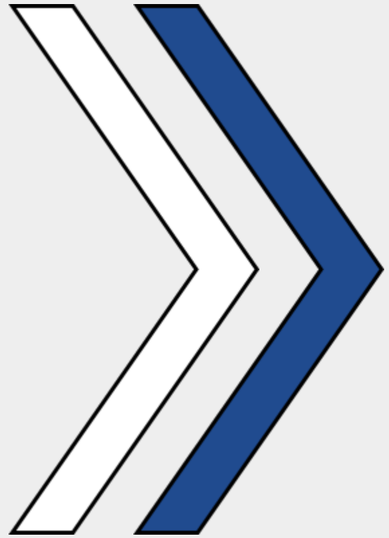
02 블록 다이어 그램(PipeLine)



- stage 구간별 Data/Control 신호 처리
- Hazard Unit 모든 stage Hazard 관리
- Forward Architecture
- DECODE stage: Jump, branch 처리
- stall, flush 방식으로 Hazard 해결

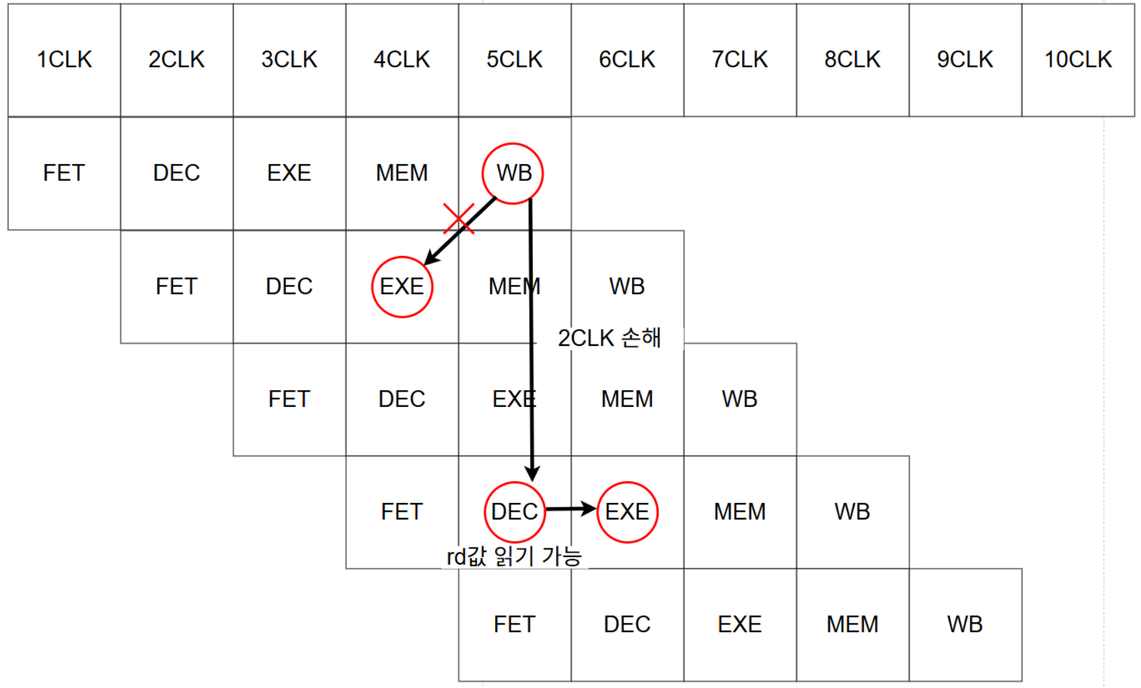
04

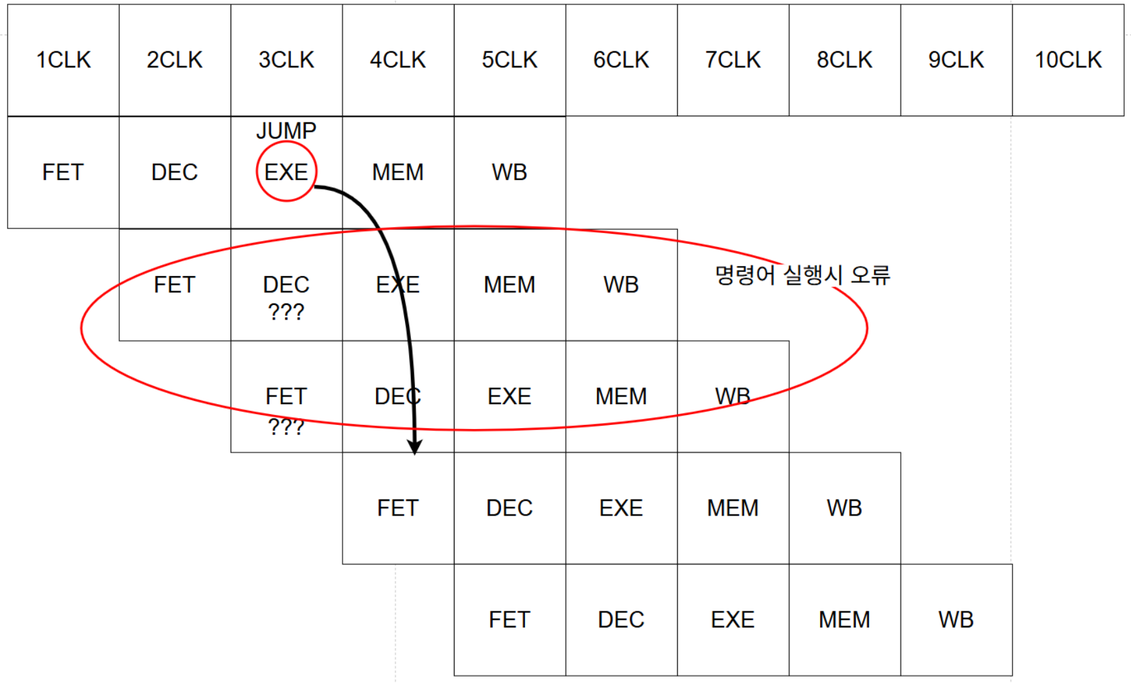
Hazard Hardware Solve



- DATA Hazard/Control Hazard
- 하드웨어적 해결방법

03 Hazard 개념





- 파이프라인 에서는 Execute구간에서 레지스터 파일에 바로 쓰기 불가능
- CLK 마다 계속 새로운 명령어 입력됨
- Hazard 소프트웨어(compiler) 해결 -> 많은 중간 nop명령 삽입 -> CLK 손해 증가

03 Hazard 종류

Data Hazard

개념 : 명령어가 앞선 명령어의 rd 레지스터를 사용할때 발생

- EXE_Hazard
  바로 앞의 명령어 값이 필요한 경우
- MEM_Hazard
  2번째 앞의 명령어 값이 필요한 경우
- Load-Use Data_Hazard
  앞선 명령어가 L타입 일때
  바로 뒤 명령어가 필요로 하는 경우

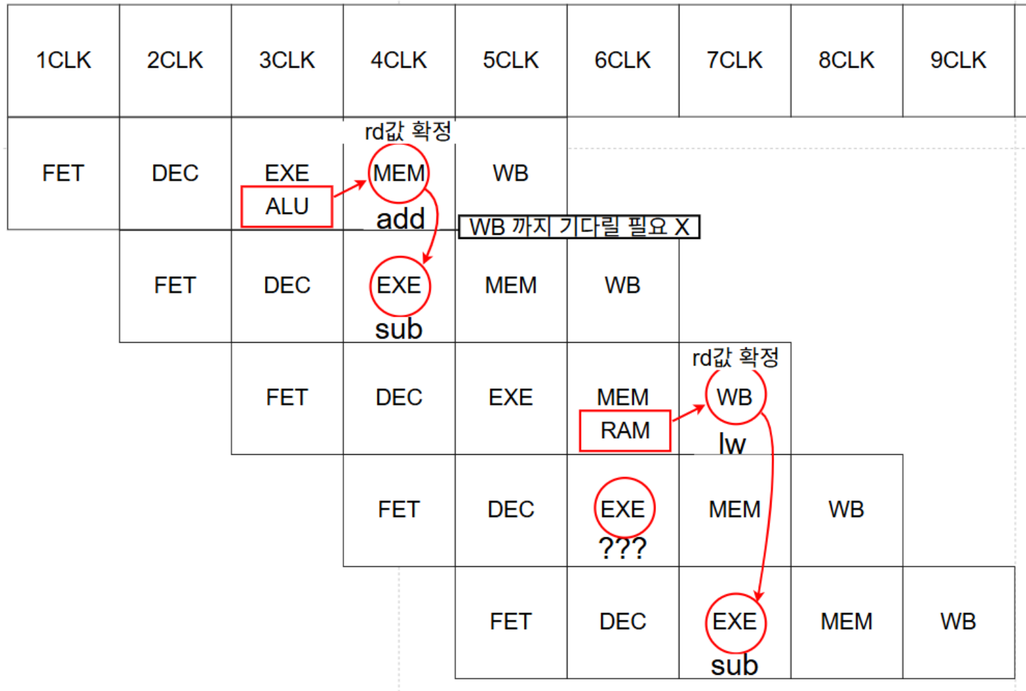
Control Hazard

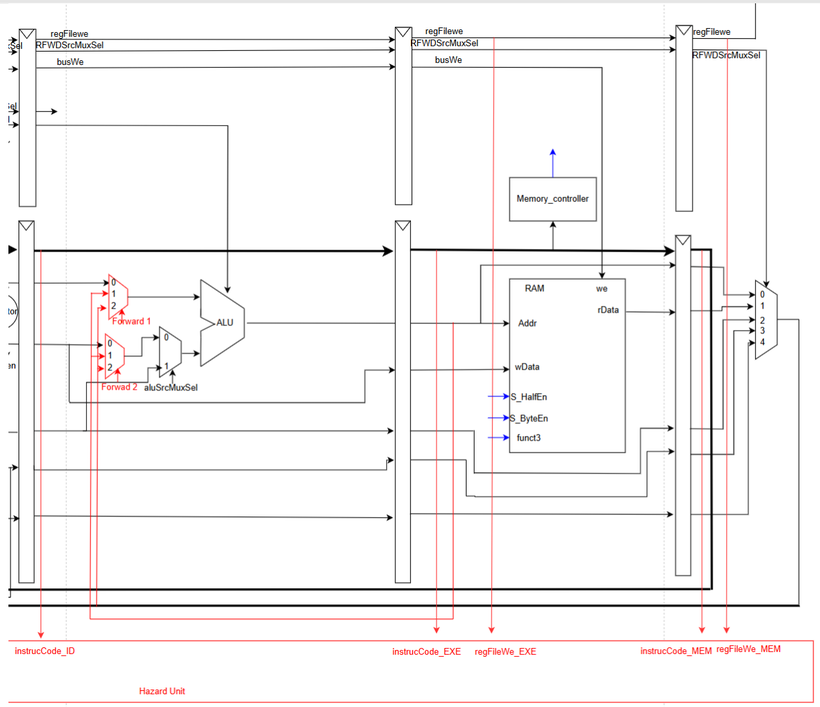
개념 : branch 또는 Jump 명령어로 PC값의 변할때 발생

해결방법 :

- flush 신호로 nop 명령어 삽입 (2 CLK 손해)
- DECODE stage에서 Jump, branch처리 (1CLK 손해)
- branch predict : branch가 뜨지 않을때는 flush X

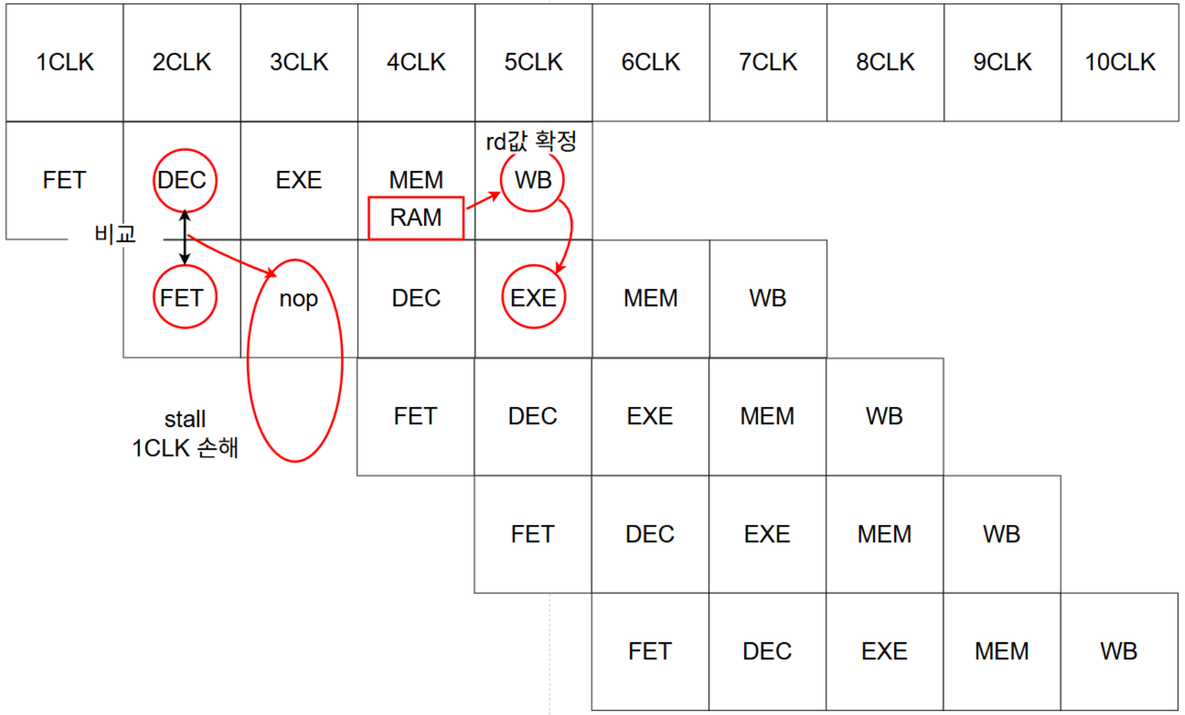
03 DATA Hazard

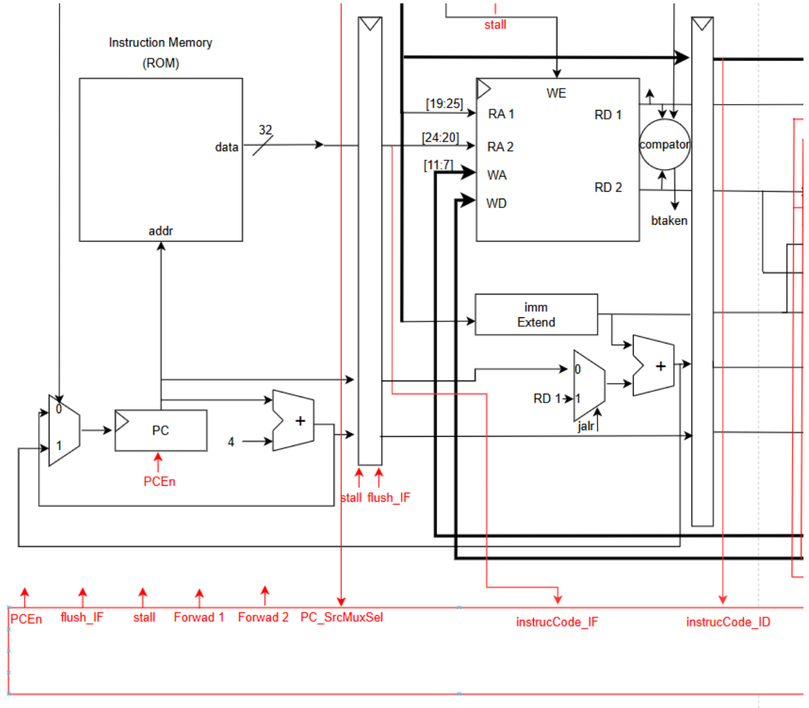




- add x3, x2, x1; sub x4, x3, x1 : (rd_MEM == rs1_EXE,rs2-EXE)&(regFileWe_MEM == 1)&(rd_MEM != 0) -> MEM_Forward
- lw x3, x1, 3; add x4, x2, x1; sub x5, x3, x1: (rd_WB == rs1_EXE,rs2-EXE)&(regFileWe_WB == 1)&(rd_WB != 0) -> WB_Forward
- lw x3, x1, 3; sub x5, x3, x1 : 4CLK때 sub EXE 연산때 lw rd값 확정 X -> Load-Use Data Hazard

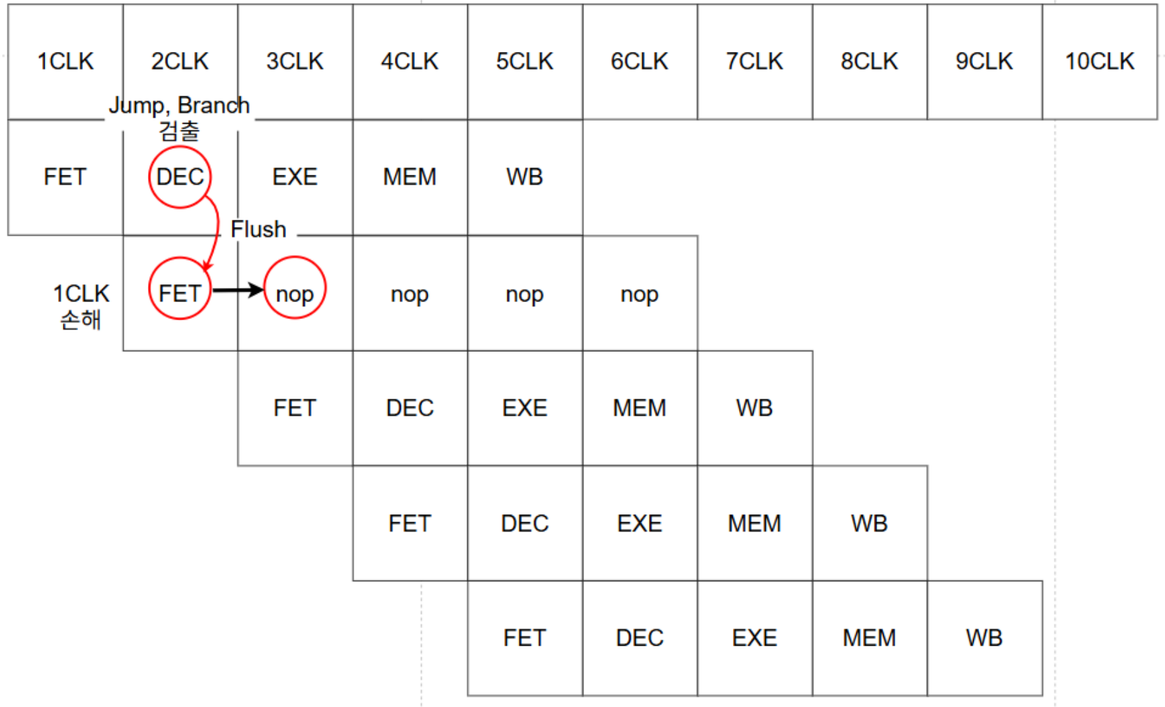
03 Load-Use Data Hazard

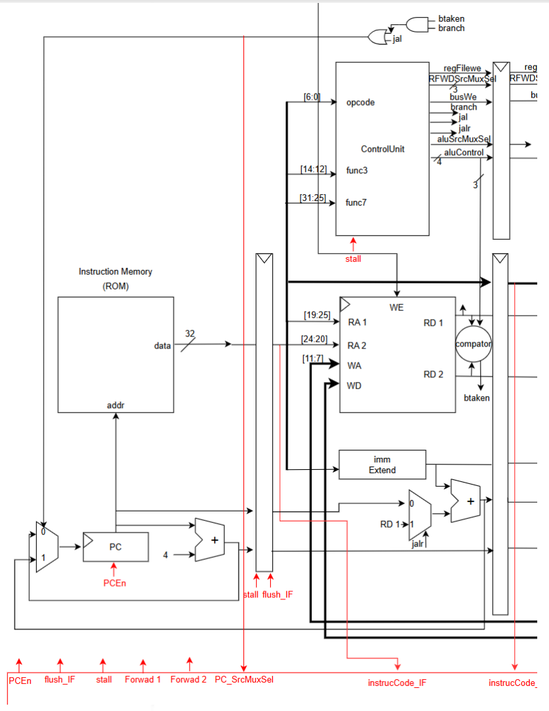




- DECODE에서 opcode == L타입 확인, FETCH와DECODE stage에서 rd_DEC == rs1_FET, rs2_FET 확인
- 조건 확인후 stall 신호 출력으로 1clk 명령어 입력을 멈추고 3CLK때 FETCH 명령어 유지,대신 nop명령어 출력
- stall로 인해 5CLK 구간에서 WB_Forward 가능

03 Control Hazard

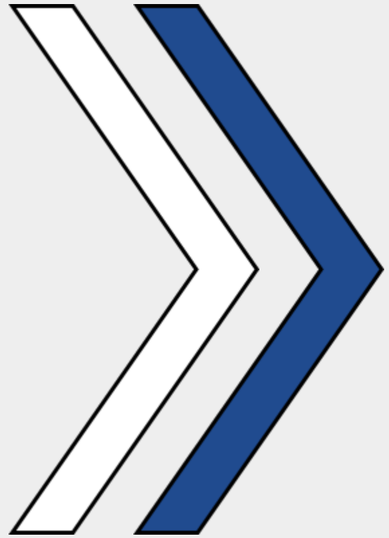




- DECODE-stage에서 비교기 와 점프 신호를 통해 분기 검출
- 분기시 전명령어 Flush -> nop(아무 실행X) 명령어로 변환

05

Simulation Result



- Hazard unit 유무 차이
- Store Byte, Half, Load Byte, Half

02 Hazard Test code

```
initial begin
    // $readmemh("code.mem", rom);
// DATA Hazard Test
rom[0] = 32'b0000000_00001_00010_000_00100_0110011; // add x4,x2,x1 -> 11+12=23
rom[1] = 32'b0100000_00001_00100_000_00101_0110011; // sub x5,x4,x1 -> 23-11=12 // EX hazard
rom[2] = 32'b0000000_00101_00100_000_00110_0110011; // add x6,x4,x5 -> 23+12=35 // EX + MEM hazard
// Load Use Data Hazard Test
rom[3] = 32'b0000000_00010_00000_010_01000_0100011; // sw x2,8(x0)
rom[4] = 32'b000000001000_00000_010_00111_0000011; // lw x7,8(x0)
rom[5] = 32'b0000000_00111_00001_000_01000_0110011; // add x8,x1,x7 -> 11+12=23 // Load Use Data hazard
rom[6] = 32'b0000000_00010_00010_000_01001_0110011; // add x9,x2,x2 -> 24 // MEM hazard
// Control Hazard Test
// Branch Not Taken
rom[7]  = 32'b0000000_00100_00101_000_01010_1100011; // beq x4, x5, +8   (23!=12 -> not taken)
rom[8]  = 32'b0000000_00001_00010_000_01010_0110011; // add x10, x2, x1   (12+11=23, 실행됨)
rom[9]  = 32'b0000000_00010_00001_000_01011_0110011; // add x11, x1, x2   (11+12=23, 실행됨)

// Branch Taken (flush 1 instruction)
rom[10] = 32'b0000000_00001_00001_000_01100_1100011; // beq x1, x1, +12  (taken, target=PC+12 → rom[13])
rom[11] = 32'b0000000_00001_00001_000_01100_0110011; // add x12, x1, x1   (flush, 실행X) // hazard
rom[12] = 32'b0000000_00010_00010_000_01101_0110011; // add x13, x2, x2   (jump, 실행X) // hazard
rom[13] = 32'b0000000_00001_00010_000_01110_0110011; // add x14, x2, x1   (정상 실행됨)

// JAL Test (flush 1 instruction)
rom[14] = 32'b0_0000000100_0_00000000_00011_1101111; // jal x3, +8 (target=PC+8 → rom[16])
rom[15] = 32'b0000000_00001_00001_000_01111_0110011; // add x15, x1, x1 (flush, 실행X) // hazard
rom[16] = 32'b0000000_00010_00010_000_10000_0110011; // add x16, x2, x2 (정상 실행) // hazard 2번 반복 예상
rom[17] = 32'b0000000_00010_00010_000_10001_0110011; // add x17, x2, x2 (정상 실행)
rom[18] = 32'b0000000_00001_00010_000_10010_0110011; // add x18, x2, x1 (정상 실행)

// JALR Test (flush 1 instruction)
rom[19] = 32'b000001011000_00000_000_00011_1100111; // jalr x3, x0, +88 (target=88=rom[22])
rom[20] = 32'b0000000_00001_00001_000_10011_0110011; // add x19, x1, x1 (flush, 실행X) // hazard
rom[21] = 32'b0000000_00010_00010_000_10100_0110011; // add x20, x2, x2 (jump, 실행X) // hazard
rom[22] = 32'b0000000_00001_00010_000_10101_0110011; // add x21, x2, x1 (정상 실행)
end
```

> “ 파이프라인 Hazard Unit X
> VS
> 파이프라인 Hazard Unit ”

- Test Code를 통해 Hazard발생 상황 시뮬레이션
- 실제C에서 생길수 있는 상황 재현
- 레지스터 값 오류, 실행 오류 비교

02 Hazard Test code

```
// DATA Hazard Test
rom[0] = 32'b0000000_00001_00010_000_00100_0110011; // add x4,x2,x1 -> 11+12=23
rom[1] = 32'b0100000_00001_00100_000_00101_0110011; // sub x5,x4,x1 -> 23-11=12 // EX hazard
rom[2] = 32'b0000000_00101_00100_000_00110_0110011; // add x6,x4,x5 -> 23+12=35 // EX + MEM hazard
// Load Use Data Hazard Test
rom[3] = 32'b0000000_00010_00000_010_01000_0100011; // sw x2,8(x0)
rom[4] = 32'b000000001000_00000_010_00111_0000011; // lw x7,8(x0)
rom[5] = 32'b0000000_00111_00001_000_01000_0110011; // add x8,x1,x7 -> 11+12=23 // Load Use Data hazard
rom[6] = 32'b0000000_00010_00010_000_01001_0110011; // add x9,x2,x2 -> 24 // MEM hazard
```

c언어 :

int x1 = 11; int x2 = 12; int x3 = 13; int x4 = 14; int x5 = 15; int x6 = 16;

Int x7 = 17;

x4 = x2 + x1; // 23

x5 = x4 - x1; // 12

x6 = x4 + x5; // 35

X7 = x2; // 12

X8 = x1 + x7 // 23

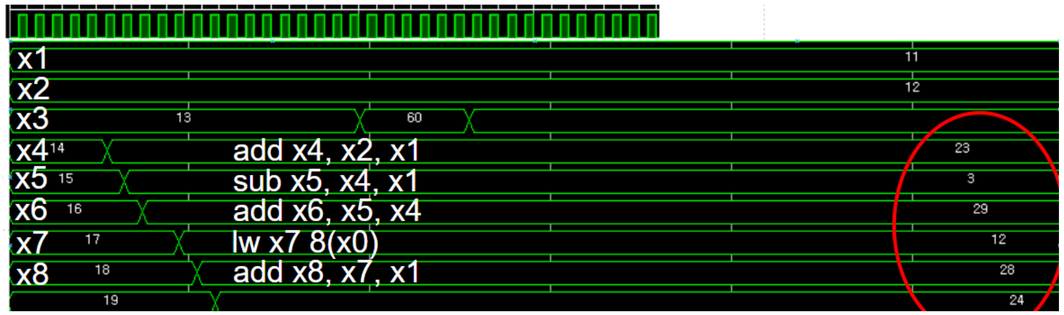
“ x5, x6, x8 = ? ”

- Data hazard 로 발생하는 오차(오류) 확인

- Load Use Data hazard로 발생하는 1clk 손실 확인

03 Hazard Test Simulation

Hazard Unit X



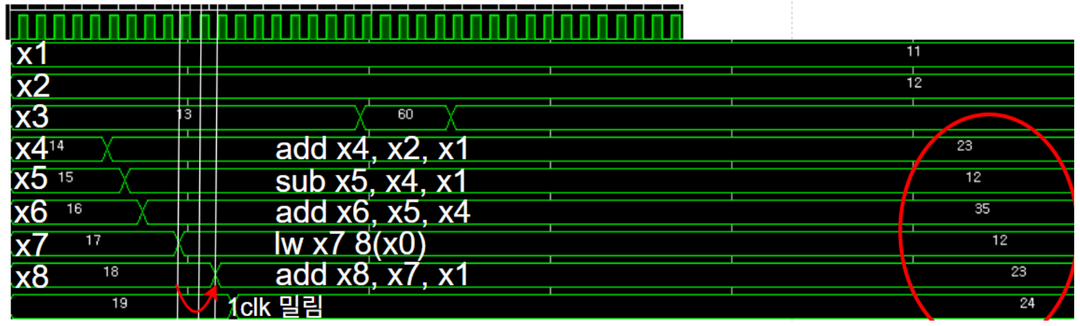
- x5,x6 x8 오차(오류) 발생

x4 초기값 14 -> 14 – 11 = 3 (x4 업데이트 전 연산)

x5 초기값 15 -> 15 + 14 = 29 (x4, x5 업데이트 전 연산)

lw 뒤의 add 명령어 또한 같은 방식으로 오류

Hazard Unit



- stall 신호로 add명령어 1clk 멈춤
- 이후 MEM hazard Forward로 정상값 출력
- Forward 와 stall 로 정상적으로 출력

02 Hazard Test code

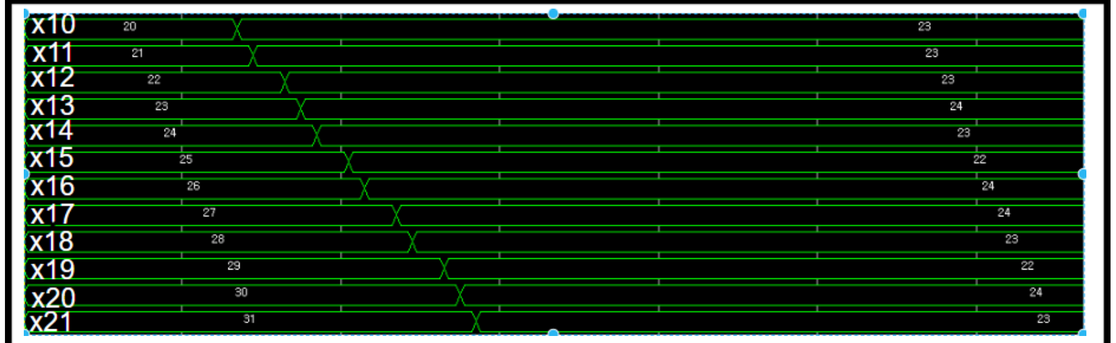
```
// Control Hazard Test
// Branch Not Taken
rom[7]  = 32'b0000000_00100_00101_000_01010_1100011; // beq x4, x5, +8   (23!=12 -> not taken)
rom[8]  = 32'b0000000_00001_00010_000_01010_0110011; // add x10, x2, x1   (12+11=23, 실행됨)
rom[9]  = 32'b0000000_00010_00001_000_01011_0110011; // add x11, x1, x2   (11+12=23, 실행됨)

// Branch Taken (flush 1 instruction)
rom[10] = 32'b0000000_00001_00001_000_01100_1100011; // beq x1, x1, +12  (taken, target=PC+12 → rom[13])
rom[11] = 32'b0000000_00001_00001_000_01100_0110011; // add x12, x1, x1   (flush, 실행X) // hazard
rom[12] = 32'b0000000_00010_00010_000_01101_0110011; // add x13, x2, x2   (jump, 실행X) // hazard
rom[13] = 32'b0000000_00001_00010_000_01110_0110011; // add x14, x2, x1   (정상 실행됨)

// JAL Test (flush 1 instruction)
rom[14] = 32'b0_0000000100_0_00000000_00011_1101111; // jal x3, +8 (target=PC+8 → rom[16])
rom[15] = 32'b0000000_00001_00001_000_01111_0110011; // add x15, x1, x1 (flush, 실행X) // hazard
rom[16] = 32'b0000000_00010_00010_000_10000_0110011; // add x16, x2, x2 (정상 실행) // hazard 2번 반복 예상
rom[17] = 32'b0000000_00010_00010_000_10001_0110011; // add x17, x2, x2 (정상 실행)
rom[18] = 32'b0000000_00001_00010_000_10010_0110011; // add x18, x2, x1 (정상 실행)

// JALR Test (flush 1 instruction)
rom[19] = 32'b000001011000_00000_000_00011_1100111; // jalr x3, x0, +88 (target=88=rom[22])
rom[20] = 32'b0000000_00001_00001_000_10011_0110011; // add x19, x1, x1 (flush, 실행X) // hazard
rom[21] = 32'b0000000_00010_00010_000_10100_0110011; // add x20, x2, x2 (jump, 실행X) // hazard
rom[22] = 32'b0000000_00001_00010_000_10101_0110011; // add x21, x2, x1 (정상 실행)
```

> “분기후 분기된 주소의 명령 수행”

- B(조건), J, JR type 분기 명령후
이전 PC +4의 명령이 수행되는 것은 오류
-파이프 라인 특성상 분기가 결정되기 전
명령어가 계속 입력됨
- 분기 후 PC 값이 바뀌어 다른명령어 입력 받아도
이미 들어온 명령어는 파이프를 돌아다님

03 Hazard Test Simulation

Hazard Unit X



- 분기와 관계 없이 모든 명령어가 수행 되어버림 (오류)
- C 언어 : if문,for문, 함수호출, 에서 조건과 관계없이 2개의 명령어를 수행하여 오류를 일으킴

Hazard Unit

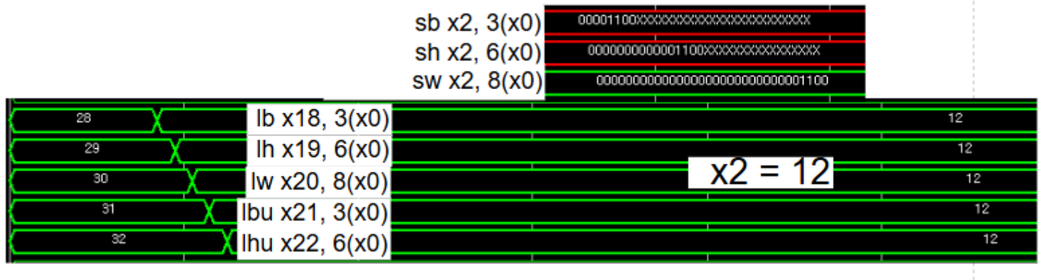
- 분기시 이전명령어 flush로 인해 사라짐 (정상작동)
- 점프된 주소 명령어 수행

02 byte, half Store/Load

```
always_ff @( posedge clk ) begin
    if (we) begin
        if (S_HalfEn) begin
            if (addr[31:1] % 2) mem[addr[31:2]][31:16] <= wData[15:0];
            else mem[addr[31:2]][15:0] <= wData[15:0];
        end
        else if (S_ByteEn) begin
            if(addr % 4 == 0) mem[addr[31:2]][7:0] <= wData[7:0];
            else if(addr % 4 == 1) mem[addr[31:2]][15:8] <= wData[7:0];
            else if(addr % 4 == 2) mem[addr[31:2]][23:16] <= wData[7:0];
            else mem[addr[31:2]][31:24] <= wData[7:0];
        end
        else mem[addr[31:2]] <= wData;
    end
end
```

```
always_comb begin
    rData = 32'bx; //Latch 방지
    case (funct3)
        3'b000: begin
            if (addr % 4 == 1) rData = {{24{mem[addr[31:2]][15]}}, mem[addr[31:2]][15:8]};
            if (addr % 4 == 2) rData = {{24{mem[addr[31:2]][23]}}, mem[addr[31:2]][23:16]};
            if (addr % 4 == 3) rData = {{24{mem[addr[31:2]][31]}}, mem[addr[31:2]][31:24]};
            else rData = {{24{mem[addr[31:2]][7]}}, mem[addr[31:2]][7:0]};
        end
        3'b001: begin
            if (addr[31:1] % 2) rData = {{16{mem[addr[31:2]][31]}}, mem[addr[31:2]][31:16]};
            else rData = {{16{mem[addr[31:2]][15]}}, mem[addr[31:2]][15:0]};
        end
        3'b100: begin
            if (addr % 4 == 1) rData = {{24{1'b0}}, mem[addr[31:2]][15:8]};
            if (addr % 4 == 2) rData = {{24{1'b0}}, mem[addr[31:2]][23:16]};
            if (addr % 4 == 3) rData = {{24{1'b0}}, mem[addr[31:2]][31:24]};
            else rData = {{24{1'b0}}, mem[addr[31:2]][7:0]};
        end
        3'b101: begin
            if (addr[31:1] % 2) rData = {{16{1'b0}}, mem[addr[31:2]][31:16]};
            else rData = {{16{1'b0}}, mem[addr[31:2]][15:0]};
        end
        3'b010: rData = mem[addr[31:2]];
    endcase
end
```

> “ Byte, Half단위 주소 사용 ”



- 명령어에 따라서 램에 주소를 4,2,1의 배수 사용
- Load시 기본적으로 32 bit signed extention,
  unsigned load시 32 bit zero extention

결론 및 느낀점

RESULT

- Load-Use Data hazard 1clk 손해로 마무리
- DECODE Stage 분기 검출
  FETCH-flush만으로 1clk 손해 마무리
  (EXECUTE Stage에서는 2clk 손해)
- Hazard 영향을 최대로 줄여 빠르게 동작 가능한
  RISC-V RV32I PipeLine CPU 구현 성공

FILLING

- 자료 분석력과 이해력 상승
- 아이디어 하드웨어 구현 능력 상승
- 스스로 해낸 결과의 자신감 얻음
- 설계 과정에서 생각할 것이 많았기에
  재미 와 성취감, 경험을 얻음