Computer Architecture (CSED 311), Spring 2025
Lab 2: Single-Cycle CPU

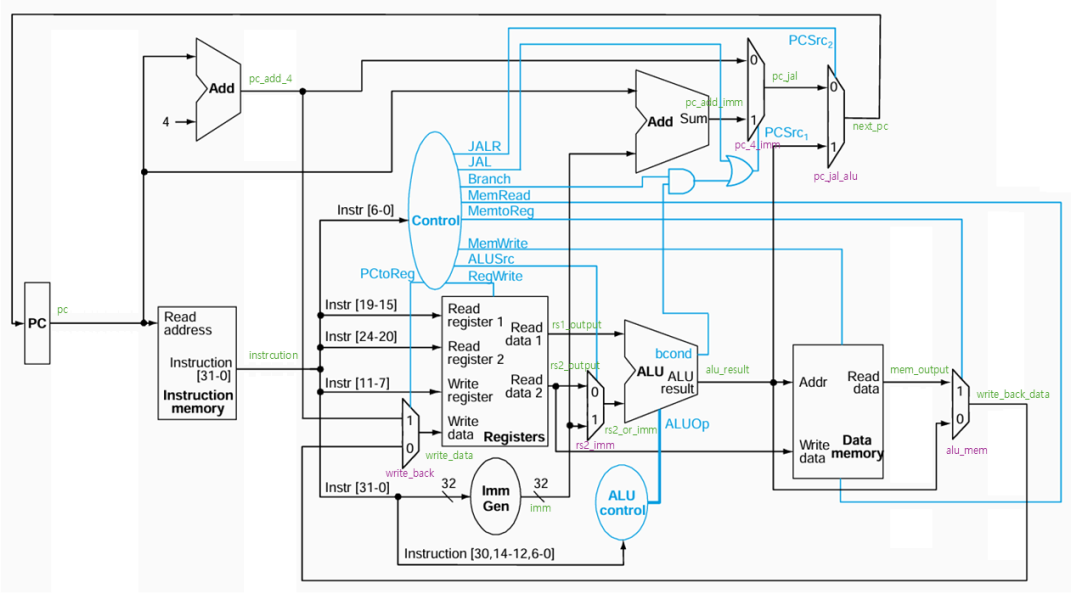
구현웅 (20210940), 김민서 (20220826)
3/23, 2025

1. Introduction

본 랩에서는 RV32I 명령어 집합을 구현하는 single-cycle CPU를 설계한다. CPU를 구성하는 각 모듈을 설계 및 구현하고, clock synchronous 및 asynchronous 여부를 구분한다. 또한, 구현한 Single-cycle CPU의 작동 과정을 IF, ID, EX, MEM, WB의 5단계로 나누어 설명하겠다.

2. Design

본 랩의 cpu는 강의에서 제시한 single-cycle의 설계를 기반으로 한다. 디자인을 다섯 단계로 나누어 설명하고 사용되는 모듈의 기능을 synchronous에 따라 나누어 설명한다.



2.1. Instruction process

한 사이클의 인스트럭션 처리 과정은 크게 IF, ID, EX, MEM, WB 5단계로 나눌 수 있다. IF는 instruction fetch 의미하며 현재 pc의 값을 통해 메모리에서 불러오는 과정이다. 디자인 상에서는 pc의 값을 instruction memory 모듈의 전달을 하고 memory 상의 instruction을 불러오는 것을 확인할 수 있다.

ID는 instruction decode을 의미하며 bit로 이루어진 instruction을 의미에 맞게 파싱해 각각의 모듈의 전달하는 역할이다. 특히 control 모듈에서는 instruction를 해석해 적절한 control bits를 생성하고 있다. 그외의 register, immediate generator, alu control 모듈로 instruction의 일부가 전달됨을 확인할 수 있다.

EX는 execution을 의미하며 alu로 계산을 수행하는 부분이다. Decode 단계에서 생성된 register data, immediate data를 통해 연산을 수행하고 결과를 다른 단계로 전달한다. Control flow의 경우에도 ALU의 계산이 필요한 경우에는 값을 만들어 bcond라는 signal로 전달한다.

MEM은 memory 접근을 의미하며 data memory module에 접근해 데이터를 쓰거나(load), 저장하는(store) 작업을 수행한다. 이때 읽고 쓰는 control은 instruction decode 단계에서 설정되어 MemWrite, MemRead 등의 wire로 전달됨을 확인할 수 있다.

마지막 WB은 write back를 의미하며 ALU혹은 data memory에서 생성된 결과를 다시 register에 쓰는 작업이다. 설계 상에서 생성된 데이터를 다시 register의 입력으로 넣는 것을 확인할 수 있다. 또한 register의 값으로 생성된 결과가 다시 register의 입력으로 연결되는 sequential logic이므로 synchronous의 필요성도 재차 확인할 수 있다.

Instruction을 기준으로 동작 흐름은 이상이지만, jump와 branch instruction을 지원하기 위해 후보가 되는 pc 값을 만들어주고 이를 instruction decode에서 만든 signal로 제어하고 있음을 이미지의 상단에서 확인할 수 있다.

2.2. Clock synchronous / asynchronous

CPU는 pc와 pc에 의해 로드되는 instruction에 따라 동작이 결정된다. 내부 동작에 따라 pc+4 혹은 다른 값이 다시 pc의 입력으로 제공되어 새로운 instruction 동작을 수행한다. 따라서 본 설계는 출력이 입력으로 돌아오는 sequential logic이다. 그 외에도 register의 출력 값에 따라 alu 혹은 data memory의 출력이 write back되어 register의 write 입력으로 들어온다. 이 부분에서도 sequential logic의 특성을 확인할 수 있다. 그리고 sequential logic에 의해 state를 업데이트를 하는 기준인 clock의 필요성이 생기며 clock에 의존해 값의 변경하느냐에 따라 synchronous와 asynchronous가 나뉘게 된다.

본 설계는 write는 synchronous하게 read는 asynchronous하도록 규칙을 따른다. pc, register, data memory 모듈에서는 데이터를 쓰고 이 데이터는 다음 instruction process에 사용될 수 있다. 만약 write가 asynchronous하게 데이터를 사용한다면 read를 하는 시점에서 데이터가 완성이 되었는지 모듈이 보장을 해줄 수 없을 것이다. 따라서 clock 단위로 write가 수행되고 cycle 동안에는 read에 대해 항상 일정한 값을 제공하도록 설계된 것이다.

synchronous data는 pc, register data, memory data 세 가지 종류가 있다. Instruction control를 담당하는 pc는 한 사이클 동안 일정한 값을 가지며 다음 사이클에 pc+4 혹은 jump에 의해 다른 값을 가지게 된다. Register data는 write back에 의해 alu, memory data의 결과가 data write가 이루어지고 이렇게 써진 data는 다음 사이클부터 읽어 사용된다. memory data도 register와 비슷하게 write가 제공되며 memory data의 출력이 register의 write 입력으로 제공되니깐 한 사이클 동안은 계속 같은 값을 가질 수 있도록 write는 동기화 되어 있다.

그외의 instruction memory에서 data를 읽거나 register값으로 alu의 결과를 만드는 과정은 결과가 cycle이 끝나기 전까지만 제공되면 될 뿐이지 데이터가 생성되는 시점은 중요하지 않다. 이런 의미에서 asynchronous라고 볼 수 있다.

3. Implementation

본 랩에서 구현한 single-cycle CPU는 제공된 skeleton code 내 각 모듈의 인터페이스를 특별히 수정하지 않아도 구현하는데 문제가 없었다. 그 외에 mux와 같이 추가적으로 정의하여 사용한 모듈도 있다.

3.1. mux32

mux32는 32bit wire에 대한 selector이며 bit logic으로 구현되었다.

```
assign dout = ({32{select}} & w0) | ({32{!select}} & w1);
```

3.2. pc

pc의 구현은 다음과 같이 next_pc를 입력 받아 다음 clock에 동기적으로 current_pc를 업데이트한다.

```
input             reset;
input             clk;
input      [31:0] next_pc;
output reg [31:0] current_pc;

initial begin
  current_pc = 0;
end

always @(posedge clk) begin
  current_pc <= reset ? 0 : next_pc;
end
```

3.3. instruction_memory

Instruction memory 모듈에서는 memory 접근이지만 read만 이루어지므로 다음과 같은 간단한 combinational logic으로 구현된다.

```
assign dout = mem[imem_addr];
```

3.4. register_file

register file 모듈의 read는 asynchronous 이며, 다음과 같은 간단한 combinational logic으로 이루어진다.

```
assign rs1_dout  = rf[rs1];
assign rs2_dout  = rf[rs2];
```

그에 반해, write에 대해서는 clk에 동기화로 업데이트를 하는 것을 확인할 수 있고, 불필요한 wrtie를 막기위해 write_enable 신호가 있어야한 write 수행한다. 또한 reset 신호에 의해 register의 memory를 초기화하므로 reset 중일 때는 write가 비활성화된다.

```
always @(posedge clk) begin
  if (!reset) begin
    if (write_enable && rd != 0) begin
      rf[rd] <= rd_din;
    end
  end
end
```

3.5. control_unit

control unit은 순전히 combinational logic으로 instruction에 대한 control signal를 생성한다. 다음은 opcode의 종류로 결정되는 신호다.

```
  is_jal     = opcode == `JAL;
  is_jalr    = opcode == `JALR;
  branch     = opcode == `BRANCH;
  mem_read   = opcode == `LOAD;
  mem_to_reg = opcode == `LOAD;
  mem_write  = opcode == `STORE;
  is_ecall   = opcode == `ECALL;
```

그외의 여러 opcode가 같은 신호를 생성하는 경우에는 case 구문으로 구현되어 있다.

```
 //alu_src : zero -> src is register; one -> imm gen;
 case (opcode)
    `BRANCH, `ARITHMETIC: alu_src = 0;
    default:              alu_src = 1;
 endcase

 //write_enable
 case (opcode)
   `BRANCH, `STORE, `ECALL
          :  write_enable = 0;
   default:  write_enable = 1;
 endcase

 //pc_to_reg : zero -> normal , one -> PC+4 to register write input
 case (opcode)
   `JAL, `JALR: pc_to_reg = 1;
   default:     pc_to_reg = 0;
 endcase
```

3.6. immediate_generator

immediate generator는 alu 등에서 사용되는 immediate value 생성하며 instruction에 의해서만 의존하기 때문에 다음과 같이 instruction을 파싱해서 제작한다. 특히 연산에 대해 sign-extend를 지원해야하기 때문에 비는 상위비트를 채워주는 것을 확인할 수 있다.

```
always @(*) begin
  case (opcode)
    `ARITHMETIC_IMM, `LOAD, `JALR: begin
      imm_gen_out[11:0]  = part_of_inst[31:20];
      imm_gen_out[31:12] = {20{imm_gen_out[11]}};  //sign-extend
    end

    `STORE: begin
      imm_gen_out[4:0]   = part_of_inst[11:7];
      imm_gen_out[11:5]  = part_of_inst[31:25];
      imm_gen_out[31:12] = {20{imm_gen_out[11]}};  //sign-extend
    end

    `BRANCH: begin
      imm_gen_out[0]    = 1'b0;
      imm_gen_out[11]   = part_of_inst[7];
      imm_gen_out[4:1]  = part_of_inst[11:8];
      imm_gen_out[10:5] = part_of_inst[30:25];
      imm_gen_out[12]   = part_of_inst[31];
      imm_gen_out[31:13] = {19{imm_gen_out[12]}};  //sign-extend
    end

    `JAL: begin
      imm_gen_out[0]     = 1'b0;
      imm_gen_out[19:12] = part_of_inst[19:12];
      imm_gen_out[11]    = part_of_inst[20];
      imm_gen_out[10:1]  = part_of_inst[30:21];
      imm_gen_out[20]    = part_of_inst[31];
      imm_gen_out[31:21] = {11{imm_gen_out[20]}};  //sign-extend
    end

    `ECALL: imm_gen_out = 10;

    default: imm_gen_out = 0;

  endcase
end
```

3.7. alu_control_unit

alu_control_unit은 {funct7[5], funct3, opcode}로 구성된 part_of_inst를 입력받아 alu_op를 출력한다. alu_op는 alu 모듈의 input으로 연결되어 ALU가 어떤 연산을 해야하는지 선택하는데, 그 값은 opcode, funct3, funct7 값에 의하여 결정된다. alu_op가 가질 수 있는 값은 alu_def.v 파일에 정의되어 있다.

SUB의 경우는 ADD와 funct3이 겹치는데, funct7 값을 추가로 비교하여 alu_op를 결정한다. 다만, opcode가 ARITHMETIC_IMM인 경우는 무조건 ADD가 선택되도록 하였다. ECALL 명령어의 경우 ALU가 어떤 연산을 해야하는지 명시된 바는 없지만, BEQ에 해당하는 alu_op를 갖도록 구현하여 x17 레지스터의 값이 10과 일치하는지 비교하도록 하였다.

```
input      [10:0] part_of_inst;
output reg [ 3:0] alu_op;

wire [6:0] funct7;
wire [2:0] funct3;
wire [6:0] opcode;

assign {funct7[5], funct3, opcode} = part_of_inst;
assign {funct7[6], funct7[4:0]} = 0;

always @(*) begin
  case (opcode)
    `ARITHMETIC, `ARITHMETIC_IMM: begin
      case (funct3)
        `FUNCT3_ADD: alu_op = (opcode == `ARITHMETIC && funct7 ==
`FUNCT7_SUB)
                             ? `SUB : `ADD;
        `FUNCT3_SLL: alu_op = `SLL;
        `FUNCT3_XOR: alu_op = `XOR;
        `FUNCT3_OR:  alu_op = `OR;
        `FUNCT3_AND: alu_op = `AND;
        `FUNCT3_SRL: alu_op = `SRL;
        default:     alu_op = 0;  // never reaches
      endcase
    end

    `BRANCH: begin
      case (funct3)
        `FUNCT3_BEQ: alu_op = `BEQ;
        `FUNCT3_BNE: alu_op = `BNE;
        `FUNCT3_BLT: alu_op = `BLT;
        `FUNCT3_BGE: alu_op = `BGE;
        default:     alu_op = 0;  // never reaches
      endcase
    end
```

```
    `LOAD, `STORE, `JAL, `JALR
          :         alu_op = `ADD;

    `ECALL :        alu_op = `BEQ;

    default:        alu_op = 0;  // never reaches
  endcase
end
```

3.8. alu

alu 모듈은 alu_op의 값에 따라 alu_in_1과 alu_in_2의 값으로부터 연산 결과인 alu_result와 alu_bcond를 출력한다. alu_bcond는 opcode가 branch인 명령어들에 대하여 branch condition이 만족되었는지를 나타낸다.

```
input      [ 3:0] alu_op;
input      [31:0] alu_in_1;
input      [31:0] alu_in_2;
output reg [31:0] alu_result;
output reg        alu_bcond;

always @(*) begin
  case (alu_op)
    `BEQ, `BNE, `BLT, `BGE, `SUB
        :    alu_result = alu_in_1 - alu_in_2;
    `ADD:    alu_result = alu_in_1 + alu_in_2;
    `XOR:    alu_result = alu_in_1 ^ alu_in_2;
    `OR :    alu_result = alu_in_1 | alu_in_2;
    `AND:    alu_result = alu_in_1 & alu_in_2;
    `SLL:    alu_result = alu_in_1 << alu_in_2;
    `SRL:    alu_result = alu_in_1 >> alu_in_2;
    default: alu_result = 0;  // never reaches
  endcase
end
```

alu_bcond의 계산에는 alu_result를 활용하였다. opcode가 branch인 경우 alu_op가 SUB이기 때문에, alu_result와 0을 비교하면 branch condition의 만족 여부를 판단할 수 있다. 특히, BLT와 BGE의 경우 부호를 비교해야 하므로 verilog HDL의 $signed 함수를 사용하였다.

```
always @(*) begin
  case (alu_op)
    `BEQ:    alu_bcond = alu_result == 0;
    `BNE:    alu_bcond = alu_result != 0;
    `BLT:    alu_bcond = $signed(alu_result) < 0;
    `BGE:    alu_bcond = $signed(alu_result) >= 0;
    default: alu_bcond = 0;
  endcase
end
```

3.9. data_memory

data memory는 register와 거의 동일한 동작을 수행한다. Combinational logic으로 asynchronous하게 dout을 생성한다. 이때 mem_read 비활성화 되어 있으면 명시적으로 0의 값을 갖도록 설계되었다.

```
  always @(*) begin
    dout = mem_read ? mem[dmem_addr] : 0;
  end
```

write는 clk에 대해서 동기화를 진행하며 mem_write signal에 의존한다. register와 아주 유사한 동작을 수행한다.

```
  always @(posedge clk) begin
    if (mem_write) begin
      mem[dmem_addr] <= din;
    end
  end
```

3.10. is_halted (on cpu.v)

cpu.v 파일을 보면, cpu 모듈은 ECALL 명령어 실행시 특정 조건을 만족하면 is_halted 플래그를 설정하도록 명시되어 있다. is_halted의 구현은 다음과 같이 구성하였다.

우선, reg_file의 입력 rs1은 원래 instruction[19:15]에 해당하는데, 이를 opcode가 ecall인 경우는 17이 입력되도록 구성한다. 그리고 imm_gen에서 opcode가 ECALL인 경우 상수 10을 출력하도록 구현한다. 이는 추후 x17 레지스터의 값과 상수 10을 비교하기 위함이다. 두 값의 비교는 alu에서 일어난다. alu_op는 opcode가 ecall인 경우 BEQ가 출력되도록 하여, opcode가 ecall이고 alu_bcond 플래그가 참이면 is_halted 플래그를 설정하도록 구현했다.

4. Discussion

이번에 구현한 RV32I 명령어 집합에는 곱셈 및 나눗셈 연산이 빠져있다. 추후 더 확장된 ISA를 구현하면서 이들을 ALU에 추가해볼 수 있을 것이다.

Branch instruction의 경우 ALU의 동작이 뺄셈으로 지정되어 있다. 뺄셈으로 구현할 경우, branch condition을 계산하는데 ALU의 output을 활용할 수 있어서 효과적인 것으로 보인다. 예를 들어, BEQ의 경우 alu에서 두 입력의 차가 0이면 bcond를 1로 두도록 할 수 있다.

제공된 skeleton code의 메모리 reset 로직은 동기적으로 구현되어있는데, sequential logic에서 메모리 역할을 하는 레지스터에 대해 non-blocking이 아닌 blocking assignment가 사용되었다. 이 과정에서 linter error를 무시하기 위한 directive comment까지 사용된 것을 확인할 수 있었는데, 이렇게 구현되어야만 하는 이유에 대해 추후 논의할 만 하다.

5. Conclusion

본 랩에서는 RV32I 명령어 집합을 구현한 single-cycle CPU를 설계하였다. RISC-V single-cycle CPU의 구성 요소를 구현하고 서로 조합하는 과정에서 각 요소의 상호작용과 CPU 작동 과정상 ID, IF, EX, MEM, WB의 5가지 단계를 잘 이해할 수 있었다. Single-cycle CPU을 성공적으로 구현해본 경험은 추후 multi-cycle 및 pipelined CPU의 설계를 이해하고 구현하는데 효과적으로 기여할 것이다.