Computer Architecture (CSED 311), Spring 2025
Lab 3: Multi-Cycle CPU

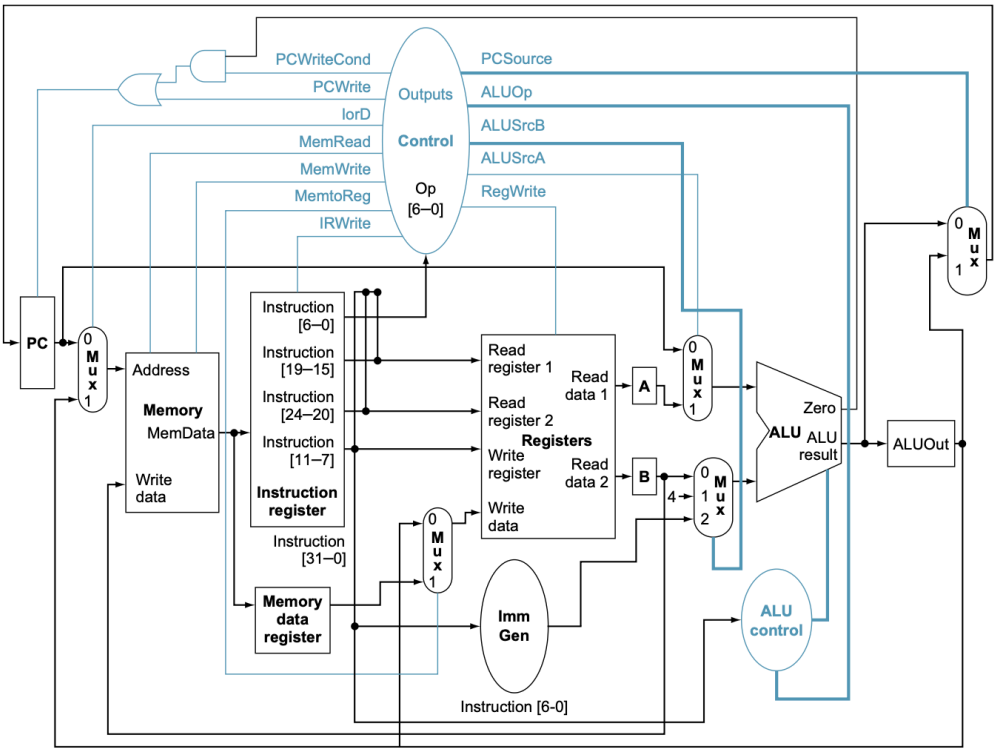
구현웅 (20210940), 김민서 (20220826)
4/14, 2025

1. Introduction

본 랩에서는 RV32I 명령어 집합을 구현하는 multi-cycle CPU를 설계한다. Single-cycle CPU와 multi-cycle CPU를 비교하고, multi-cycle CPU의 설계 및 구현, CPU를 구성하는 각 모듈의 clock synchronous 및 asynchronous 여부를 설명하겠다. 또한 microcode controller의 설계를 다루어 보겠다.

2. Design

본 랩의 cpu는 강의에서 제시한 multi-cycle CPU의 설계를 기반으로 한다. Single-cycle CPU와의 차이를 다루고 각 모듈의 기능을 synchronous/asynchronous 여부에 따라 설명하겠다.



2.1. Multi-Cycle vs. Single-Cycle CPU

이전 Lab 2에서 구현한 single-cycle CPU는 IF, ID, EX, MEM, WB의 5단계가 모두 하나의 cycle 내에서 실행되었다. 이와 대조적으로, 이번 Lab 3에서 구현한 multi-cycle CPU는 각 step이 하나에서 수 개의 micro-cycle을 요구한다. 예를 들어 IF이 2-cycle, ID가 1-cycle, EX는 2-cycle, MEM은 5-cycle, WB가 1-cycle이라면 가장 긴 경로 기준으로 IF1–IF2–ID–EX1–EX2–MEM1–MEM2–MEM3–MEM4–MEM5–WB의 순서로 실행된다. 또한, Branch 명령어의 경우 MEM과 WB를 거치지 않으므로 IF1-IF2-ID-EX1-EX2의 순서로 실행된다.

이처럼 multi-cycle CPU는 각 명령어마다 요구하는 (micro-)cycle의 수가 다를 수 있다. Single-cycle CPU에서는 모든 명령어가 하나의 cycle 내에서 실행되어야 하므로 clock speed를 가장 느린 명령어를 기준으로 설정할 수밖에 없다. 그러나 multi-cycle CPU에서는 각 명령어가 서로 다른 길이의 cycle 수를 가질 수 있어서 명령어 실행 시간이 다를 수 있으므로 single-cycle CPU에 비해 wall-clock time을 더 줄일 수 있다.

2.2. Clock Synchronous / Asynchronous

저번 Lab 2에서 설계한 single-cycle CPU는 pc, register file, data memory의 세 모듈이 synchronous하였고, 나머지는 모두 combinational logic으로만 구성된 asynchronous 모듈이었다. 이번에 설계한 multi-cycle CPU는 microcode controller 모듈이 추가적으로 synchronous하게 구현되어 있다. microcode controller는 multi-cycle CPU의 설계에서 도입된 모듈이다. CPU의 pc에 대응되는 micro program counter (upc)를 가지며, 현재의 upc에 따라 적절한 control output을 출력한다.

2.3. Microcode controller

Microcode controller는 horizontal 또는 vertical하게 구현할 수 있다. horizontal 구현에서는 하나의 ROM을 사용하여 각 microcode가 ROM의 row 하나에 대응한다. microcode는 control output의 집합으로 구성된다. vertical 구현에서는 두 개의 ROM을 사용하여, 첫 번째 ROM은 각 row에 microcode를 저장하며 두 번째 ROM에서는 해당 microcode를 decode한 최종 control output을 저장한다.

이번 Lab에서는 horizontal하게 microcode controller를 구현하였다. 따라서 microcode의 각 비트는 control output 하나에 대응된다. CPU의 State는 IF1, IF2, ID, EX1, EX2, MEM1, MEM2, WB로 구성하였고, 각 state에 microcode를 하나씩 대응시켜 총 8개의 microcode가 ROM에 저장되도록 하였다.

2.4. Resource reuse

Single-cycle CPU는 모든 step이 한 cycle 내에 실행되므로 각 step에서 사용하고자 하는 모듈이 중복될 경우 이를 여러 개 둘 수밖에 없었다. 그러나, multi-cycle CPU에서는 각 step이 서로 다른 cycle에서 실행되므로 중복되는 모듈 사용을 줄일 수 있다. 예를 들어, single-cycle CPU 구현에서 JALR 명령어는 점프 할 명령어의 위치를 계산하기 위해 ALU와 다른 별도의 가산기를 썼다. 하지만 multi-cycle CPU에서는 이를 ALU에서 계산되도록 하여 별도의 가산기를 둘 필요를 없앨 수 있다. 또한, JALR 외 다른 명령어에서 다음 명령어의 위치를 얻기 위해 pc+4를 계산하는 부분도 별도의 원래 별도의 가산기를 뒀으나, 이 역시 ALU에서 처리되도록 하여 공간 사용 측면에서 효율적으로 구현할 수 있다.

3. Implementation

본 랩은 각각의 모듈의 사양은 single-cycle cpu와 거의 동일하며, 이를 instruction에 대해 적절하게 조절하는 microcode가 핵심이다. Microcode의 구현을 중점적으로 설명한다. Microcode는 state로 구성된 FSM 설계해 instruction에 따라 다른 루틴을 따른다. 기본적으로는 micro pc가 state 자체를 의미하며, 기본적으로는 1씩 증가를 하지만 opcode가 무엇이냐에 따라 종종 다른 state로 점프를 뛴다. Microcode 내부에는 state에 대한 control signal를 포함하고 있으며, state와 opcode에 동시에 의존하는 경우에는 추가적인 combinational logic으로 control signal를 생성하도록 설계되었다.

본 단락에서는 register의 사용 방식, state의 설계, microcode의 구성, signal 생성, time difference for pc에 대한 순으로 multi-cycle cpu 구현에 핵심을 짚어나간다.

3.1. Register

본 랩에서는 여러 사이클에 걸쳐 CPU 내부 모듈을 사용하면서 register에 값을 임시로 저장하게 되었다. 이를 위해 각각에 register에 데이터를 쓰라는 신호를 write_~의 형식으로 정의하였으며 매 사이클 마다 write 신호가 있다면 register를 업데이트한다. alu_bcond의 경우에도 bcond라는 register에 데이터를 쓰고 pc를 업데이트할 때 이 신호를 반영해 PC의 값을 선택한다.

```
always @(posedge clk) begin
  if(reset) begin
    IR <= 0;
    MDR <= 0;
    A <= 0;
    B <= 0;
    ALUOut <= 0;
    bcond <= 0;
  end
  else begin
    if(write_IR)
      IR <= mem_dout;
    if(write_MDR)
      MDR <= mem_dout;
    if(write_AB) begin
      A <= rs1_dout;
      B <= rs2_dout;
    end
    if(write_ALUOut)
      ALUOut <= alu_result;
    if(write_bcond)
      bcond <= alu_bcond;
  end
end
```

3.2. State

CPU의 내부의 유닛을 instruction마다 다르게 사용하기 위한 FSM의 상태는 다음과 같이 8개로 정의하였다. 수업에서 논의한 FSM의 설계와는 조금 다르며 랩 환경에 맞게 더욱 축소하였다.

```
`define IF1     3'b000
`define IF2     3'b001
`define ID      3'b010
```

```
`define EX1     3'b011
`define EX2     3'b100
`define MEM1    3'b101
`define MEM2    3'b110
`define WB      3'b111
```

IF1에서는 현재 pc의 값을 업데이트하며, IF2에서는 memory에서 instruction을 불러와 IF에 저장한다. ID에서는 register의 데이터를 읽어 A, B에 저장한다. EX1에서는 ALUOut의 데이터를 만들어 쓰며, EX2에서는 bcond를 생성한다. MEM1, MEM2에서는 메모리 액세스를 수행한다. 마지막으로 WB에서는 register의 데이터를 쓴다.

제공된 설계와 다른 점은 IF1이다. 시스템의 일관된 동작을 위해 instruction의 마지막 단계가 끝나면 pc를 업데이트하는 방식이 아닌 IF1에서 pc의 업데이트를 하도록 했으며, pc의 값 변경과 IF에 데이터 쓰기를 포함해 2cycle이 걸린다. 또한, IF의 새로운 값은 ID 때부터 읽을 수 있어 JAL에 대해 ID를 스킵하고 EX1으로 넘어가는 동작을 허용하지 않는다.

3.3. Microcode

Microcode는 다음과 같이 state에 대한 control signal의 묶음으로 표현이 되어 있으며, 각 bit가 의미하는 것은 상위 비트 부터 `i_or_d, write_pc, write_IR, write_AB, writeALUOut, write_reg, write_bcond, write_MDR, mem_if_read, mem_excute` 이다. write_ 로 시작하는 bit는 활성화 되면 대상에 write signal를 보내 레지스터를 업데이트한다. mem_if_read 같은 경우에는 instruction를 memory에서 불러올 때 보내는 read signal이다. mem_excute의 경우 mem state에서 read, write에 대해 허용하는 신호이다. 또한 이 신호는 현재 state가 어디인지를 나타내는 의미로 다른 신호를 생성하는데 사용되기도 한다. 즉, microcode_unit에서 microcode의 역할과 signal unit의 역할이 모두 포함되어 있다.

```
    microcode[0] = 10'b0100000000; // IF1
    microcode[1] = 10'b0010000010; // IF2
    microcode[2] = 10'b0001000000; // ID
    microcode[3] = 10'b0000100000; // EX1
    microcode[4] = 10'b0000001000; // EX2
    microcode[5] = 10'b1000000101; // MEM1
    microcode[6] = 10'b0000000000; // MEM2
    microcode[7] = 10'b0000010000; // WB
```

micro pc의 다음 값은 기본적으로 1씩 증가하지만 instuction 마다 필요없는 stage는 건너뛰도록 구성되어있다. 다음이 그 코드를 나타내며 instruction 마다 서로 다른 micro pc를 지정하는 것을 확인할 수 있다.

```
  always @(*) begin
    case (current_upc)
      `IF2: begin
        if(opcode == `JAL)
```

```
          next_upc = current_upc + 1;
        else
          next_upc = current_upc + 1;
      end
      `EX2: begin
        if(opcode == `BRANCH || opcode == `ECALL)
          next_upc = `IF1;
        else if(opcode == `ARITHMETIC_IMM ||opcode == `ARITHMETIC || opcode
== `JALR ||opcode == `JAL)
          next_upc = `WB;
        else
          next_upc = current_upc + 1;
      end
      `MEM1: begin
        if(opcode == `STORE)
          next_upc = `IF1;
        else
          next_upc = current_upc + 2;
      end
      default:
        next_upc = current_upc + 1;
    endcase
  end
```

3.4. Signals

microcode의 bit가 signal 자체를 의미하는 경우 직접 연결되어 있으나 특정 state에 대해 특정 opcode에 작용하는 신호의 경우에는 combinal logic으로 구성되어 있다. 예를 들어 mem_read나 mem_write는 기존에 opcode에 의존에 추가로 state에도 의존하도록 구성되어 있다.

```
  assign i_or_d = microcode[current_upc][9];
  assign write_pc = microcode[current_upc][8];
  assign write_IR = microcode[current_upc][7];
  assign write_AB = microcode[current_upc][6];
  assign write_ALUOut = microcode[current_upc][5];
  assign write_reg = microcode[current_upc][4];
  assign write_bcond = microcode[current_upc][3];
  assign write_MDR = microcode[current_upc][2];
  assign mem_if_read = microcode[current_upc][1];
  assign mem_excute = microcode[current_upc][0];
  assign is_ecall = opcode == `ECALL;

  assign mem_read = mem_if_read ? 1 : ( mem_excute && opcode == `LOAD);
  assign mem_write = ( mem_excute && opcode == `STORE);
  assign alu_scr_A = ( write_ALUOut && !(opcode == `BRANCH || opcode == `JAL
|| opcode == `JALR) ) || (write_pc && opcode == `JALR) || write_bcond;
```

```
  assign alu_scr_imm = !( (write_ALUOut && opcode == `ARITHMETIC) ||
write_bcond );
  assign alu_scr_4 = write_pc ^ (opcode == `JAL || opcode == `JALR);

  assign control_add = (write_ALUOut && (opcode == `BRANCH)) || write_pc;
```

3.5. Time difference for PC

PC의 값을 계산하는 경우는 PC+4, PC+imm, A+imm 세 가지가 있으며 모두 더하기 연산을 필요로한다. 한 instruction이 수행할 때 ALU는 본래 instruction이 요구하는 A op B, A op imm 등을 수행하면서 동시에 PC의 값도 계산해야한다. 이를 구현하기 위해 EX1에서는 instruction 요구하는 A op B(imm)를 수행해 ALUOut에 저장하며 IF1에서 PC의 값을 생성하도록 구성했다.

(a) JAL, JALR

JAL과 JALR의 경우에는 PC+4를 레지스터에 write back한다. 이를 위해 EX1 단계에서 PC+4의 값을 계산해 ALUOut에 저장하고 WB 단계를 수행한다. 그리고 각각 다음 IF1 단계에서 PC+imm, A+imm를 계산하고 PC를 업데이트한다.

(b) BRANCH

Branch의 경우 PC+imm의 값을 EX1에서 생성해 ALUOut 저장하며, EX2에서 A-B를 연산해 bcond를 생성한다. bcond는 따로 레지스터에 저장해두어 다음 PC의 값을 업데이트하는 IF1 단계에서 ALUOut의 PC+imm를 사용할지 PC+4를 사용할지 결정한다. 또한 EX1에서는 더하기 연산을 EX2에서는 원래 요구되는 빼기 연산을 요구하기 때문에 control_add라는 signal로 alu에 더하기 연산을 강제한다.

```
  // ---------- ALU ----------
  alu alu(
    .alu_op( control_add ? `ADD : alu_op),      // input
    .alu_in_1(alu_in_1),    // input
    .alu_in_2(alu_in_2),    // input
    .alu_result(alu_result),  // output
    .alu_bcond(alu_bcond)     // output
  );
```

Jump instruction의 요구사항을 기반으로 alu의 입력 결정 신호를 생성한다. alu_scr_A의 의미는 0이면 PC, 1이면 A이다. alu_scr_imm은 0이면 A를 활성화 되면 imm를 선택한다. alu_scr_4의 경우 활성화 되면 alu_scr_imm의 무관하게 4를 alu의 입력으로 넣는다. 신호의 구성은 다음과 같다.

```
  assign alu_scr_A = ( write_ALUOut && !(opcode == `BRANCH || opcode == `JAL
|| opcode == `JALR) ) || (write_pc && opcode == `JALR) || write_bcond;
  assign alu_scr_imm = !( (write_ALUOut && opcode == `ARITHMETIC) ||
write_bcond );
```

```
assign alu_scr_4 = write_pc ^ (opcode == `JAL || opcode == `JALR);
```

예를 들어, JARL의 경우 write_ALUOut이 활성화되어 있는 EX1에서는 alu_scr_A는 0, alu_scr_4는 1로 PC+4의 연산을 수행하고, write_pc가 활성화되어 있는 IF1 단계서는 A+imm 연산을 수행한다.

4. Discussion

주어진 testbench의 cycle 수를 세어보니 basic_ripes는 184개, loop_ripes는 1544개로 나왔다. 답안과 비교하면 사이클 수가 조금 더 많은데, 답안의 cycle 수는 IF, ID, EX, MEM, WB의 5단계로 구성했을 시 도출 가능하다. 현재 구현의 경우 IF1, IF2, ID, EX1, EX2, MEM1, MEM2, WB의 8단계로 cycle 수가 더 많이 소요된다.

Microcode controller는 horizontal과 vertical 방식으로 구분된다. Horizontal한 구현은 하나의 큰 ROM에 각 micro-pc에 해당하는 microcode를 작성하고, vertical한 구현은 이를 더 작은 ROM 두 개를 사용하여 구현한다. 우리의 구현은 horizontal한 구현인데, 추후 vertical한 구현으로 변경하여 ROM 용량의 차이를 비교하여 효율성이 개선된 정도를 비교해볼 수 있을 것이다.

5. Conclusion

본 랩에서는 RV32I 명령어 집합을 구현한 multi-cycle CPU를 설계하였다. RISC-V multi-cycle CPU의 구성 요소를 구현하고 서로 조합하는 과정에서 microcode controller의 설계를 비롯하여 각 요소의 상호작용을 잘 이해할 수 있었다. Multi-cycle CPU을 성공적으로 구현해본 경험은 추후 pipelined CPU의 설계를 이해하고 구현하는데 효과적으로 기여할 것이다.