2024학년도 컴퓨터구조 Lab Assignment #3
Multicycle CPU

김도영, 선민수

2024년 4월 16일

1 Introduction

이 과제에서는 Verilog를 이용하여, 이전에 구현한 singlecycle CPU에서 발전한 multicycle CPU를 만드는 것을 목적으로 한다. Multicycle CPU와 singlecycle CPU의 가장 큰 차이점은 한 명령어를 실행하는 데 여러 cycle을 사용한다는 점이다. 이러한 차이점 때문에, 한 명령어를 실행하는 데 한 clock cycle만을 이용하는 singlecycle CPU에 비해 다음과 같은 장점이 있다.

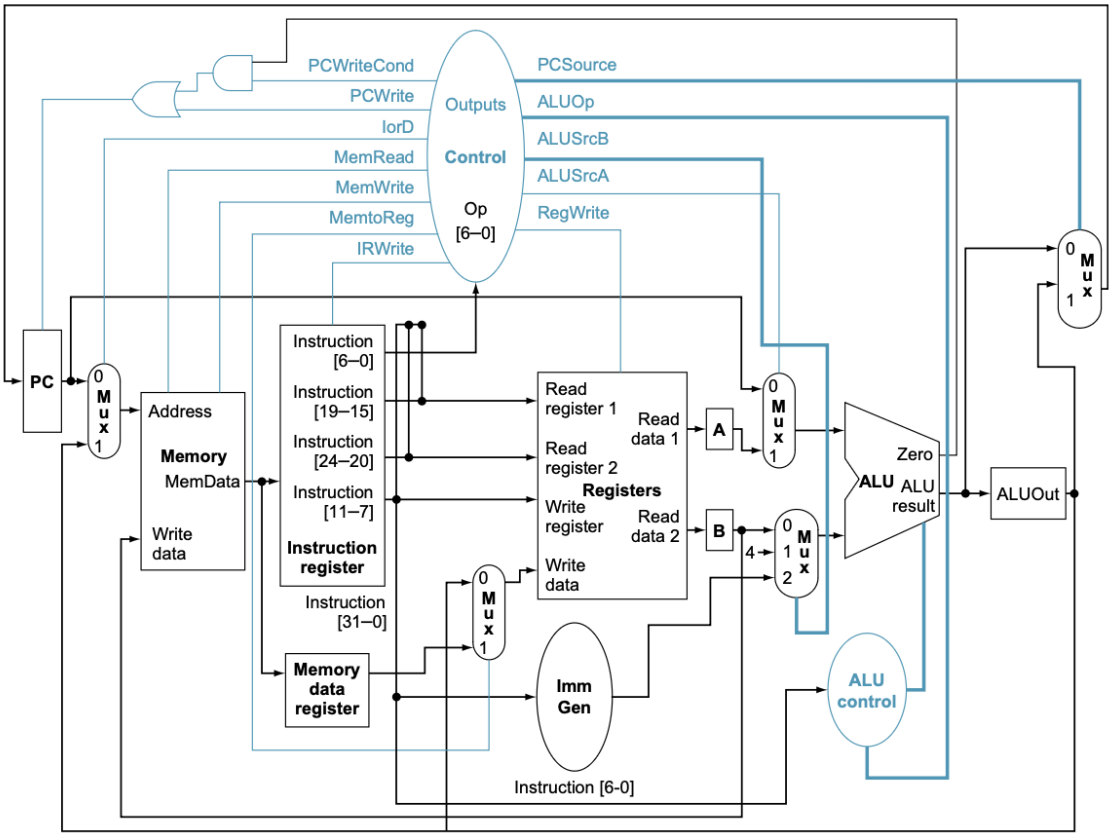
- 가장 느린 명령어의 실행 속도에 다른 명령어의 실행 속도를 맞출 필요가 없으므로 전체적인 실행 속도가 빠르다.
- 하나의 자원(ALU, memory, register file... etc)을 여러 cycle에 나누어 사용할 수 있으므로 singlecycle CPU의 중복된 부분을 제거할 수 있다.

2 Design

본 과제에서 구현한 multicycle CPU는 P&H 4판 Ch 4.5.에서 설명하는 multicycle CPU를 기준으로 강의 교안의 설계를 일부 반영하여 구현하였다. 교과서에서는 JAL, JALR, ECALL, BEQ를 제외한 나머지 Branch 명령어들을 구현하지 않았으며, 이러한 부분들에 대하여 강의 교안의 설계를 반영하였다.

Multicycle CPU를 구현하는데 쓰이는 각각의 세부 모듈과 그 구현은 다음과 같다.

- PC: 현재의 program counter 값을 저장하는 모듈로, clock의 positive edge마다 `next_pc` 신호를 받아 `pc_update` 신호가 `1`일 때 program counter를 업데이트하는 동기 회로이다.
- Memory: 실제 CPU에 연결된 memory의 역할을 하는 모듈로, singlecycle CPU의 구현과 다르게 명령어와 데이터를 모두 저장하며, clock의 positive edge마다 입력에 따라 메모리 값을 변경하는 동기 회로이다.
- Instruction register: Memory에서 불러온 명령어를 해당 명령어가 실행되는 동안 저장해두는 레지스터로, clock의 positive edge에만 업데이트되는 동기 회로이다.
- Memory data register: Memory에서 불러온 데이터를 다음 명령어가 실행되기 전까지 저장해두는 레지스터로, clock의 positive edge에만 업데이트되는 동기 회로이다.
- Registers: CPU의 programmer visible state 중 하나인 레지스터이다. `x0`부터 `x31`까지 총 32개를 가지고 있으며, clock의 positive edge에만 업데이트되는 동기 회로이다.

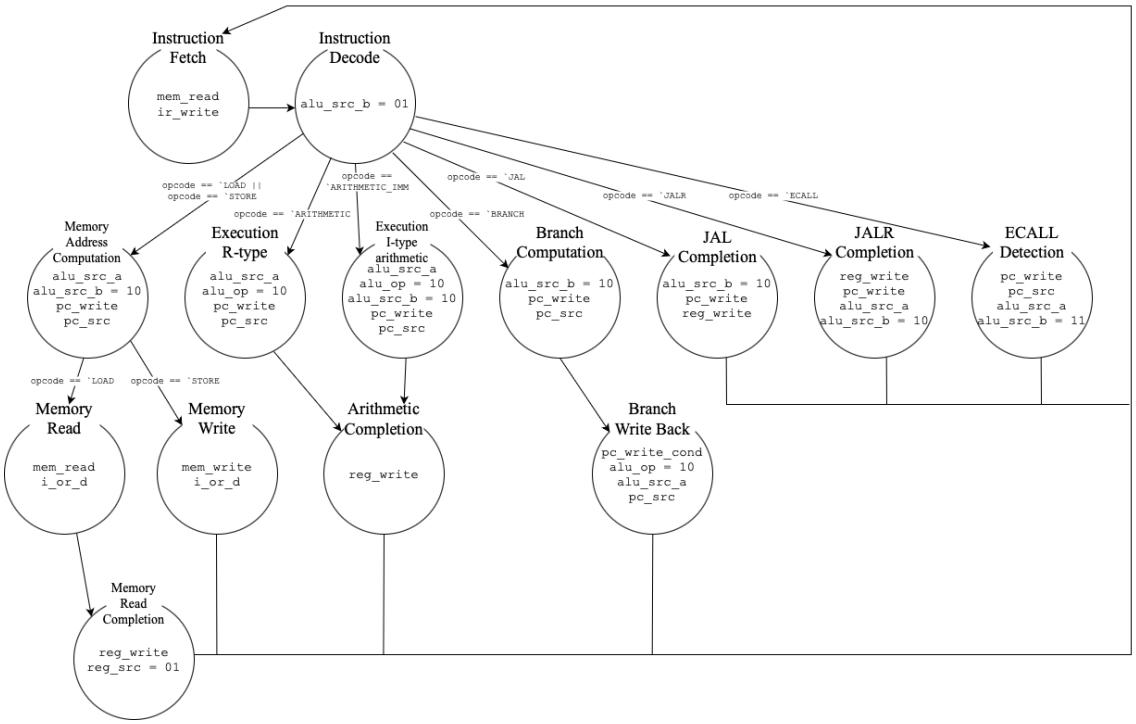


Multicycle CPU의 전반적인 모듈 설계.

- Control: 현재 명령어의 opcode를 받아 명령어의 실행 과정에 따라 해당하는 control 신호를 계산하는 모듈이다. 이때, singlecycle 구현과 다르게 여러 cycle에 걸쳐 다른 control 신호를 출력해야 하므로, clock의 positive edge에만 업데이트되는 동기 회로로 구현되었다.
- A, B: 레지스터의 값을 다음 명령어의 instruction decode stage 전 까지 저장하는 레지스터로, clock에 맞추어 업데이트되는 동기 회로이다.
- ALU: 두 입력값과 ALU contorl 신호를 받아 해당하는 연산을 하는 모듈로, 입력이 바뀌면 출력도 곧바로 바뀌는 비동기 회로이다.
- ImmGen: 명령어를 받아 명령어에 따른 immediate 값을 계산하는 모듈로, 비동기 회로이다.
- ALU control: ALU가 수행해야 할 연산을 지정해주는 ALU control 신호를 계산하는 비동기 회로 모듈이다.
- ALUOut: ALU의 결과값을 다음 ALU 연산 전까지 저장해두는 레지스터로, clock의 postivie edge에 맞추어 업데이트되는 동기 회로이다.

Multicycle CPU는 한 명령어가 여러 clock cycle에 걸쳐서 실행될 수 있다. 따라서, singlecycle CPU에서 `ControlUnit`이 현재 명령어에 의해서만 출력이 결정되는 조합 논리 회로였던 것과 달리, multicycle CPU의 `ControlUnit`은 내부 상태와 현재 입력에 따라서 출력이 결정되는 순차 논리 회로,

그 중에서도 유한 상태 기계로 구현된다. ControlUnit의 각 상태와 그에 따른 출력, 상태 전이를 명시한 다이어그램은 다음과 같다. 이때, 각 상태에 따른 출력이 명시되지 않은 신호들은 암묵적으로 0으로 간주하며, 출력의 이름이 명시되었으나 값이 명시되지 않은 신호들은 1-bit `1'b1`이다. 예를 들어, Instruction Decode stage에서 `alu_op`는 `2'b00`이며, Branch Computation stage에서 `pc_write`는 `1'b1`이다.



ControlUnit 모듈의 FSM 다이어그램.

- Instruction Fetch state: Memory에서 명령어를 가져오는 stage이다.
- Instruction Decode stage: 명령어에 따라 해당하는 레지스터 값들을 A, B 레지스터에 저장하는 stage이다. 이때, ALU에서 `PC + 4`값 또한 같이 계산하여 ALUOut 레지스터에 저장한다.
- Memory Address Computation stage: LOAD, STORE 명령어를 실행하는데 필요한 주소값을 계산하는 stage로, program counter 또한 동시에 ALUOut에 저장된 `PC + 4`로 업데이트한다.
- Execution R-type stage: R-type 산술 명령어를 실행하여 그 결과값을 ALUOut에 저장하는 stage로, 위와 같이 PC 또한 업데이트한다.
- Execution I-type arithmetic stage: I-type 산술 명령어를 실행하여 그 결과값을 ALUOut에 저장하는 stage로, 위와 같이 PC 또한 업데이트한다.
- Branch Computation stage: BRANCH 명령어의 결과 주소값을 계산하는 stage로, 분기가 taken 되지 않을 때를 대비하여 PC를 `PC + 4`로 업데이트한다.
- JAL Completion stage: JAL 명령어의 결과 주소값을 계산하고 이로 PC를 업데이트하며, `rd` 레지스터를 `PC + 4`로 업데이트하는 stage이다.
- JALR Completion stage: JALR 명령어의 결과 주소값을 계산하고 이로 PC를 업데이트하며, `rd` 레지스터를 `PC + 4`로 업데이트하는 stage이다.

- ECALL Detection stage: ECALL 명령어일 때 `GPR[x17]`의 값과 10을 비교하는 연산을 수행하는 stage이다.
- Memory Read stage: Memory Address Computation stage에서 계산한 주소를 기반으로 memory 값을 읽어 이를 Memory Data Register에 저장하는 stage이다.
- Memory Write stage: Memory Address Computation stage에서 계산한 주소를 기반으로 memory의 해당 주소에 B 레지스터의 값을 저장하는 stage이다.
- Arithmetic Completion: R-type, I-type 산술 연산의 결과를 레지스터에 저장하는 stage이다.
- Branch Write Back stage: 분기 조건을 검사하여 만약 분기가 taken되었을 경우 PC를 분기 목표 주소로 업데이트하는 stage이다.
- Memory Read Completion stage: Memory Read stage에서 저장한 memory 값을 레지스터에 다시 저장하는 stage이다.

3 Implementation

ControlUnit과 ALUControl을 제외한 다른 모듈들은 singlecycle 구현과 대동소이하므로, 이 두 모듈의 구현에 집중하여 설명한다.

3.1 ControlUnit

```
...
// Calculate the next state.
always @(*) begin
    case(state)
    `IF: next_state = `ID;

    `ID: begin
        case(opcode)
        `ARITHMETIC:     next_state = `EXR;
        `ARITHMETIC_IMM: next_state = `EXI;
        `LOAD:           next_state = `MAC;
        `JALR:           next_state = `JALRC;
        `STORE:          next_state = `MAC;
        `BRANCH:         next_state = `BC;
        `JAL:            next_state = `JALC;
        `ECALL:          next_state = `ED;
        default:         next_state = state;
        endcase
    end

    `MAC: begin
        if(opcode == `LOAD)
            next_state = `MR;
        else if(opcode == `STORE)
            next_state = `MW;
        else
            next_state = state;
    end

    `EXR:    next_state = `AC;
```

```
31      `EXI:    next_state = `AC;
32      `BC:     next_state = `BWB;
33      `JALC:   next_state = `IF;
34      `JALRC:  next_state = `IF;
35      `MR:     next_state = `MRC;
36      `MW:     next_state = `IF;
37      `AC:     next_state = `IF;
38      `MRC:    next_state = `IF;
39      `ED:     next_state = `IF;
40      `BWB:    next_state = `IF;
41      default: next_state = state;
42      endcase
43  end
44
45  // Compute output signal with respect to the current state.
46  always @(*) begin
47      pc_write_cond = (state == `BWB);
48
49      pc_write      = (state == `MAC)  || (state == `EXR)  ||
50                      (state == `EXI)  || (state == `JALC) ||
51                      (state == `JALRC) || (state == `BC) ||
52                      (state == `ED);
53
54      i_or_d        = (state == `MR) || (state == `MW);
55      mem_read      = (state == `IF) || (state == `MR);
56      mem_write     = (state == `MW);
57
58      reg_src = (state == `MRC);
59      ir_write = (state == `IF);
60
61      pc_src = (state == `MAC) || (state == `EXR) || (state == `ED) ||
62               (state == `EXI) || (state == `BWB) || (state == `BC);
63
64      alu_op = {
65          (state == `EXR) || (state == `EXI) || (state == `BWB) || (state == `ED),
66          1'b0
67      };
68
69      alu_src_a = (state == `MAC) || (state == `EXR) || (state == `EXI) ||
70                  (state == `BWB) || (state == `JALRC) || (state == `ED);
71
72      alu_src_b = {
73          (state == `MAC) || (state == `EXI) || (state == `BC) ||
74          (state == `JALC) || (state == `JALRC) || (state == `ED),
75          (state == `ID) || (state == `ED)
76      };
77
78      reg_write = (state == `JALC) || (state == `JALRC) || (state == `AC) || (state == `MRC);
79  end
80  ...
```

ControlUnit.v

ControlUnit은 CPU 전체의 control 신호를 출력하는 모듈로, 각 cycle에 따라 다른 신호를 내보내어 CPU를 구성하는 모듈이 cycle에 따라 다른 기능을 수행하도록 하고, 때문에 하나의 자원을 재사용할 수 있도록 한다. 예를 들어, Instruction Decode에서는 `alu_src_b`를 `2'b01`로 설정하며 ALU가 `PC + 4`를 계산하도록 하는데, 이러한 ALU 출력은 ALUOut 레지스터에 저장되고, 이는 다시 Execution R-type과 같은 stage에서 `alu_op`를 `2'b10`으로 설정하여 ALU를 재사용할 수 있게 한다.

3.2 ALUControl

```
```
always @(*) begin
    case(aluOp)
        2'b00: alu_op_o = `ALU_ADD;
        2'b01: alu_op_o = `ALU_SUB;
        default: begin
            case(instruction[6:0])
                `ARITHMETIC: begin
                    case(instruction[14:12])
                        `FUNCT3_ADD: alu_op_o = (instruction[30]) ? `ALU_SUB : `ALU_ADD;
                        `FUNCT3_SLL: alu_op_o = `ALU_SLL;
                        `FUNCT3_SRL: alu_op_o = `ALU_SLR;
                        `FUNCT3_AND: alu_op_o = `ALU_AND;
                        `FUNCT3_OR:  alu_op_o = `ALU_OR;
                        `FUNCT3_XOR: alu_op_o = `ALU_XOR;
                        default: alu_op_o = 4'b0;
                    endcase
                end
                `ARITHMETIC_IMM: begin
                    case(instruction[14:12])
                        `FUNCT3_ADD: alu_op_o = `ALU_ADD;
                        `FUNCT3_SLL: alu_op_o = `ALU_SLL;
                        `FUNCT3_SRL: alu_op_o = `ALU_SLR;
                        `FUNCT3_AND: alu_op_o = `ALU_AND;
                        `FUNCT3_OR:  alu_op_o = `ALU_OR;
                        `FUNCT3_XOR: alu_op_o = `ALU_XOR;
                        default: alu_op_o = 4'b0;
                    endcase
                end
                `BRANCH: begin
                    case(instruction[14:12])
                        `FUNCT3_BEQ: alu_op_o = `ALU_BEQ;
                        `FUNCT3_BNE: alu_op_o = `ALU_BNE;
                        `FUNCT3_BLT: alu_op_o = `ALU_BLT;
                        `FUNCT3_BGE: alu_op_o = `ALU_BGE;
                        default: alu_op_o = 4'b0;
                    endcase
                end
                `LOAD: alu_op_o = `ALU_ADD;
                `STORE: alu_op_o = `ALU_ADD;
                `JAL: alu_op_o = `ALU_ADD;
                `JALR: alu_op_o = `ALU_ADD;
                `ECALL: alu_op_o = `ALU_BEQ;
                default: alu_op_o = 4'b0;
            endcase
        end
    endcase
```
```

```
end
```

ALUControl.v

ALU에게 alu_op_o로 control signal을 넘겨주는 ALUControl은 주어진 aluOp에 따라서 필요한 control signal을 생성한다. 2'b00은 ALU_ADD를, 2'b01은 ALU_SUB를 ALU에게 넘겨준다. 이외의 aluOp는 주어진 instruction의 FUCNT3, FUNCT7에 의해서 alu_op_o를 결정한다. FUNCT7이 ARITHMETIC, ARITHMETIC_IMM, BRANCH인 경우에는 FUNCT3에 의해서 최종적인 alu_op_o가 결정된다. 이외의 FUNCT7에 경우에는 ECALL을 제외하고는 모두 ALU에서 ADD를 수행하기 때문에, alu_op_o = 'ALU_ADD'로 넘겨준다. ECALL의 경우에는 cpu.v에서 is_halted 조건을 검사할 때, ALU에 의한 bcond를 통해 x17 register의 값을 검사 하기 때문에, 이를 위해 'ALU_BEQ'를 통해서 조건 검사를 할 수 있도록 하였다.

4 Discussion

4.1 각 테스트벤치의 실행 cycle

```
### SIMULATING ###
TEST END
SIM TIME : 236
TOTAL CYCLE : 117 (Answer : 116)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000013
11 00000003
12 fffffd7
13 00000037
14 00000013
15 00000026
16 0000001e
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
```

```
31 00000000
Correct output : 32/32
```

basic_mem.txt 실행 결과

basic_mem.txt 실행 결과는 다음과 같으며, 실행에 걸린 총 cycle 수는 117 cycle이다.

```
### SIMULATING ###
TEST END
SIM TIME : 1962
TOTAL CYCLE : 980 (Answer : 977)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000000
11 00000000
12 00000000
13 00000000
14 0000000a
15 00000009
16 0000005a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

loop_mem.txt 실행 결과

loop_mem.txt 실행 결과는 다음과 같으며, 실행에 걸린 총 cycle 수는 980 cycle이다.

5 Conclusion

이번에는 지난 Single Cycle CPU에서 발전한 Multi Cycle CPU를 구현하였다. Single Cycle과는 달리 스테이지별로 필요한 clock만큼만 소요하여, Single Cycle CPU처럼 가장 실행이 오래 걸리는

instruction에 맞춘 clock speed를 가지지 않는다는 점이 특징이다. Multi Cycle CPU는 Single Cycle CPU에서 FSM을 통한 flow control을 통하여 구현할 수 있었으며, total cycle에서는 증가할 수 있지만, clock frequency의 증가로 인해 결국 기존 Single Cycle CPU에서 발전한 것을 확인할 수 있었다. 다만 Multi Cycle CPU 또한 Single Cycle CPU에서 거론되었던, 기존 instruction이 수행되는 동안 사용되지 않는 stage의 모듈들이 모두 낭비되고 있는 점을 보완하지는 못하였고, 궁극적으로 이를 개선할 수 있는 Pipelined CPU의 필요성을 확인할 수 있었다.