Lab 4 - Multi Cycle CPU

20160595 수학과 최규민
20160169 물리학과 최수민

1. Introduction

이번 과제에서는 Multi Cycle TSC CPU를 구현한다. Multi Cycle CPU는 한 사이클에 모든 instruction을 실행하는 Single Cycle CPU와는 달리 한 instruction을 여러 step으로 나누어 instruction마다 소요되는 시간을 다르게 함으로써 효율을 높인다. 이를 위해 control unit을 finite state machine으로 만들어 각 state마다 적절한 control signal을 내도록 하며, single cycle CPU에서 여러 개가 사용되었던 ALU 등의 Resource를 다시 사용할 수 있도록 구현한다.

이번 과제를 통해 배워야 하는 것은 다음의 두 가지로 요약할 수 있다.

1. Multi Cycle CPU 구현을 통해 해당 구현이 Single Cycle CPU보다 성능이 좋은지 이해한다.
2. Control unit을 Finite State Machine으로 구현하며 Micro-code controller의 작동방식을 이해한다.

2. Design

Multi Cycle CPU가 Single Cycle CPU와 다른 가장 큰 차이점은 control 이 finite state machine으로 구현되어 각 state에 맞는 control 시그널을 datapath에 전달해주고, 현재 state에서 조건에 따라 다음 state로 진행하면서 control을 변경한다는 점이다. 따라서 Single Cycle CPU에서 Control Unit은 Combinational Logic으로 구현되었지만, 이번 Multi Cycle CPU의 구현에서는 clock-synchronous한 finite state machine으로 구현되었다.

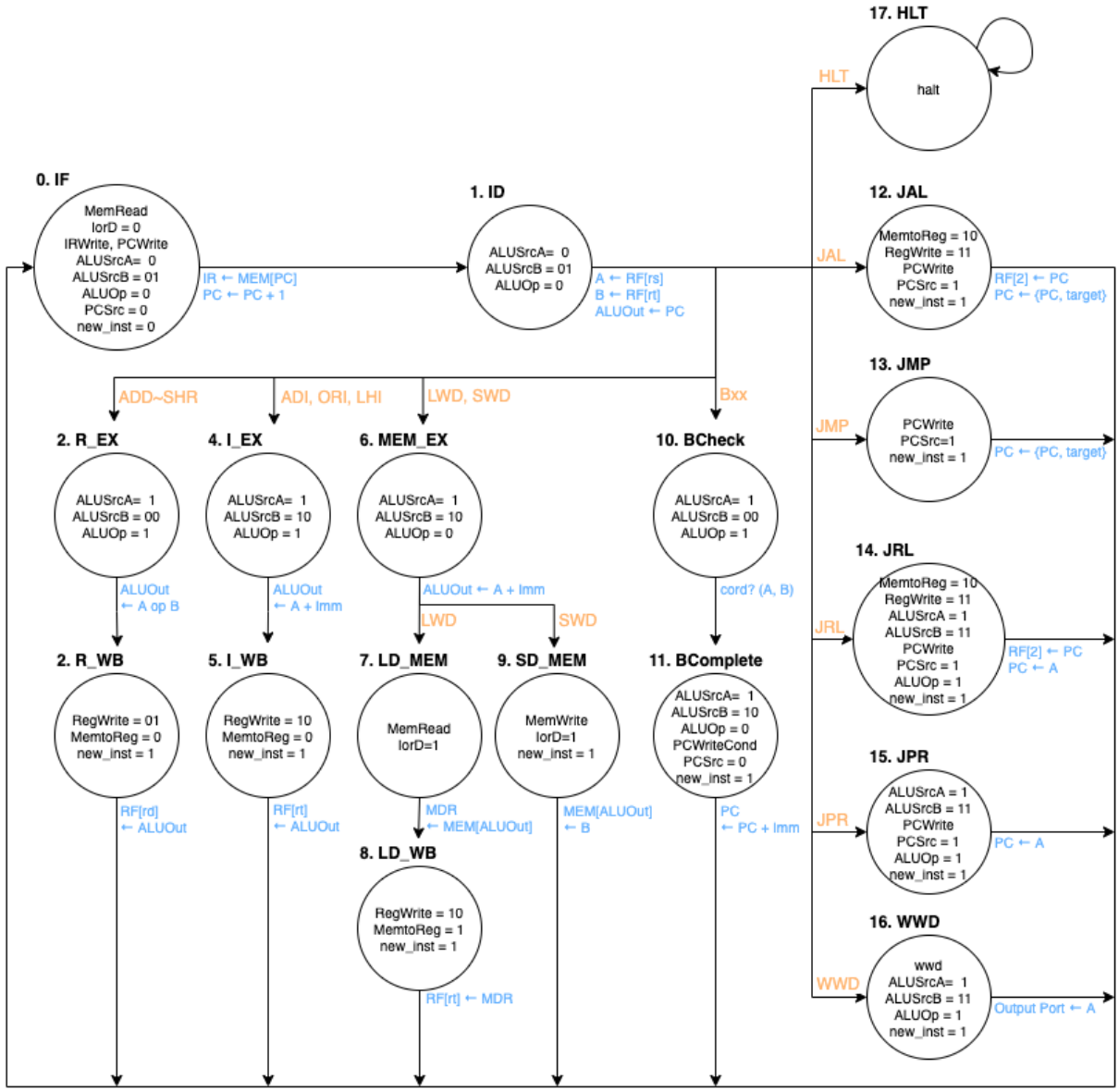
Finite state machine은 두개 파트로 구성되는데, 하나는 현재 state와 다른 input(opcode 등 instruction)에 따라서 다음 state를 결정하는 sequential logic이며, 또 다른 하나는 datapath에 가해줄 control signal을 현재 state에 따라 적절하게 결정해주는 combinational logic이다.

Single Cycle CPU에서 없으나 새로 추가된, 혹은 수정된 각 control signal들의 기능은 아래 [표1] 에서 나타내었다. 한편, control unit finite state machine의 동작을 나타낸 diagram은 아래 [그림 1]에서 나타내었고, datapath 설계 구조는 [그림 2]에서 나타내었다. Finite state Diagram에서 HLT state를 제외한 다른 leaf node state(다른 state로 진행방향이 나타나지 않은 state들)은 상태가 끝나면 다시 초기 상태인 IF 상태로 돌아간다. 모든 state들은 IF로 돌아가면서 new_inst 시그널을 output시켜 CPU에서 num_inst의 값을 증가시킨다. HLT state의 경우 자신의 상태를 계속 반복하게 되고, halt 시그널로 CPU의 is_halted 시그널을 설정한다. 파란색 글씨는 state의 동작을 묘사했고, 각 state는 번호와 이름으로 표시하였다. 주황색 글씨는 next state를 결정하는 조건이다.

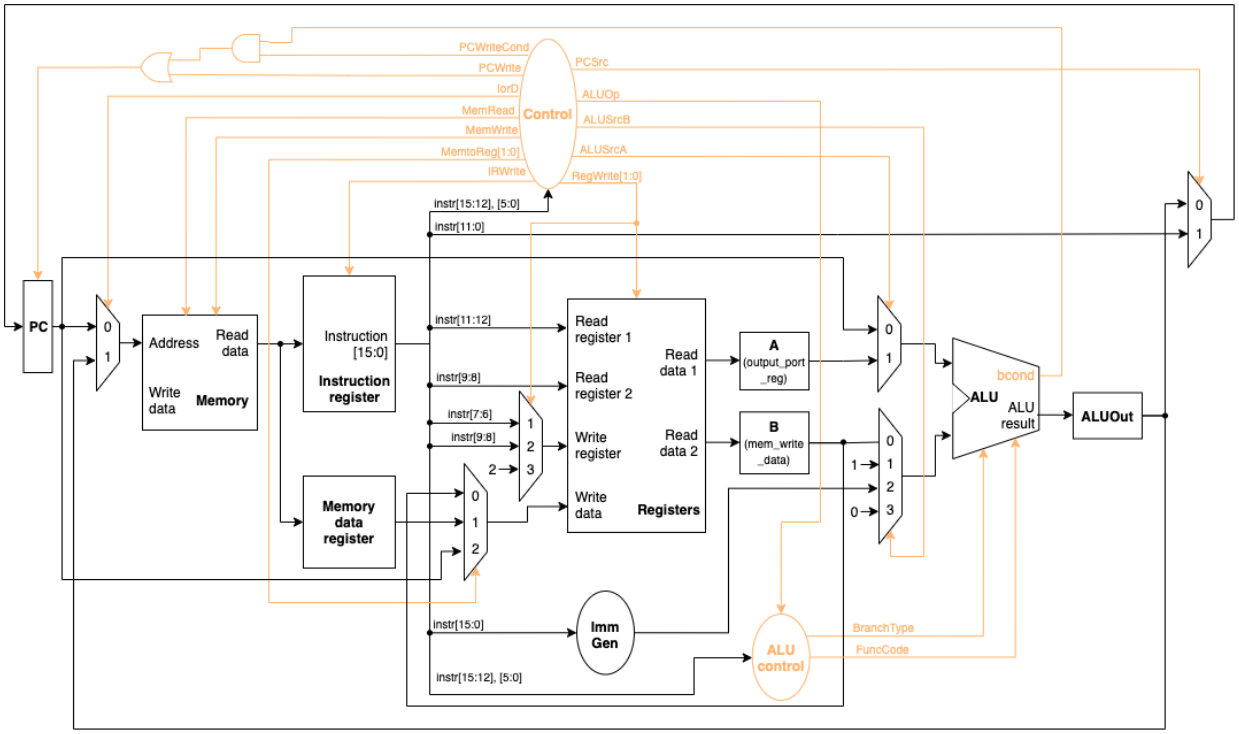
Signal	비인가(0)	인가(1)
pc_write_cond	아무 효과 없음	Branch condition이 맞으면 PC latching
pc_write	아무 효과 없음	무조건 PC latching
i_or_d	PC에서 instruction 주소를 가져옴	ALUOut에서 데이터 주소를 가져옴
Ir_write	Instruction latching 불가	Instruction latching
pc_src	ALU에서 PC가져옴	{PC[15:12, Imm[11:0]}에서 PC 가져옴
halt	아무 효과 없음	프로그램이 halt되었음을 알림
wwd	아무 효과 없음	output_port에 레지스터 출력
new_inst	아무 효과 없음	num_inst += 1
alu_op	공통 연산 실행(ADD)	instruction별 연산 실행
alu_src_A	PC값이 ALU의 첫번째 operand	RegisterFile read_out1(A)이 ALU의 첫번째 operand

Signal		효과
alu_src_B [1:0]	00	RegisterFile read_out2(B)이 ALU의 두번째 operand
	01	1이 ALU의 두번째 operand (PC 증가용)
	10	Immediate값이 ALU의 두번째 operand
reg_write [1:0]	00	RF값 변경 불가
	01	RF[rd] 값 변경
	10	RF[rt] 값 변경
	11	RF[2] 값 변경
mem_to_reg [1:0]	00	ALUOut값을 레지스터에 입력
	01	MDR 값을 레지스터에 입력
	10	PC값을 레지스터에 입력

[표1] Control unit에서 사용되는 각 control signal의 기능. 2비트 control 중 사용되지 않는 시그널은 표시하지 않았다.



[그림1] Finite-state diagram for multicycle control unit. 각 state를 이름과 번호를 표시하고, 각 state에서 발생하는 control signal을 표시하였다. 주황색 글씨는 state 변경 조건에 해당하고, 파란색 글씨는 각 state에서의 동작에 해당한다.



[그림2] Multi Cycle CPU의 datapath. Halt, wwd 등의 control unit 내부 시그널과 testing 용도로 사용되는 new_inst 시그널은 그림에 표시하지 않았다. Control unit, register file, memory 등의 모듈은 모두 clock-synchronous하게 구현되었다.

Multi-Cycle CPU의 설계는 Single Cycle CPU의 설계와 상당부분 유사하지만, control unit이 clock synchronous한 finite state machine으로 구성된다는 점과 한 instruction에 같은 resource module(ALU 등)을 여러번 사용한다는 점에서 차이가 있다. 메인 모듈은 CPU 모듈이며, 서브 모듈에는 control을 구성하는 control_unit, alu_control_unit 모듈과 datapath를 구성하는 register_file, alu, memory, immediate_generator, mux4_1, mux2_1이 있다.

2.1. CPU 모듈

CPU 모듈은 input으로 reset signal(reset_n), clock signal(clk)을 받고, inout port로 사용하는 data가 있다. output으로는 메모리에 쓰기 및 읽기 신호를 주는 read_m과 write_m, 메모리 주소 address와 하나의 instruction이 끝날 때마다 num_inst를 증가시켜 출력시켜주고, WWD instruction에는 레지스터값을 output_port를 통해 출력해준다. HLT instruction에는 is_halted를 출력하여 프로그램이 halt되었음을 알린다. 또한 pc, instruction, memory data 등을 register 변수에 저장하고, 다른 서브 모듈들에 적절한 input과 output을 연결한다.

2.2. control_unit 모듈

control_unit 모듈은 [그림1]에서 나타난 것처럼 finite state machine으로 동작한다. 현재 state를 저장하고 있으면서, 현재 state와 외부 input에 따라 적절하게 계산된 next state로 clock에

synchronous하게 업데이트된다. 현재 state에 맞는 적절한 signal을 출력하는 logic과 next state를 계산하는 logic은 combinational logic으로 구현한다. 각 signal의 기능들은 single cycle CPU와 유사하고, 변경되거나 새로 추가된 signal의 기능들은 [표1]에 나타나 있다.

2.3. alu_control_unit 모듈

alu_control_unit 모듈은 instruction과 control(ALUOp)에 따라 ALU에 가하는 시그널인 funcCode와 branchType을 생성한다. [표1]에서 나타났듯이 ALUOp 비인가시에는 공통연산인 ADD를, 인가시에는 instruction별로 각각 알맞은 연산을 실행하도록 signal을 생성해준다.

2.4. alu 모듈

alu 모듈은 input으로 연산을 할 두 수(A, B)와 alu_control_unit 모듈의 output인 func_code, branch_type을 받아 output으로 적절한 연산의 결과(C)와 오버플로 여부를 판단하는 overflow_flag, 그리고 branch 조건이 성립함을 나타내는 bcond를 내보내는 combinational logic으로 구성된다.

2.5. 기타 모듈

Register_file, immediate_generator, mux2_1, mux4_1, memory 모듈은 single cycle CPU에서의 Design과 동일하다. Register_file은 clock synchronous하게 업데이트시킨다. Memory 모듈은 CPU에서 주는 read_m과 write_m에 따라 clock synchronous하게 memory를 읽고 쓸수 있게 한다.

3. Implementation

3.1. CPU 모듈

```
else begin
        // update pc
        if (pc_write || (pc_write_cond && alu_bcond)) begin
                pc <= pc_nxt;
        end

        // fetch instruction
        if (ir_write) begin
                instruction <= data;
        end
        // or get data from memory
        else if (read_m) begin
                mem_read_data <= data;
        end

        // update num_inst
        if (new_inst) begin
                num_inst <= num_inst + 1;
        end

        // export data to output_port
        if (wwd) begin
                output_port_reg <= reg_read_out1;
        end

        // keep alu_output to reg
        alu_output_reg <= alu_output;
        mem_write_data <= reg_read_out2;
end
```

[그림 3] CPU 모듈의 코드 중 일부. Posedge마다 시그널에 맞게 pc, data를 업데이트 해주고 new_inst 가 인가되면 저장하고 있는 num_inst값을 업데이트, wwd가 인가되면 레지스터값을 output port로 출력해주며, 각종 data register도 가지고 있다.

CPU 모듈에는 각 서브 모듈의 input과 output을 전달할 수 있는 각종 wire 변수, 메모리에서 읽어온 instruction 또는 데이터를 저장하는 두 reg 변수 instruction, mem_read_data와 각종 연산값 (alu_output_reg, mem_write_data 등)을 reg 변수로 가지고 있으며 각 서브 모듈의 인스턴스들이 선언되어 있다. CPU 모듈에서는 테스트와 출력을 위해 num_inst를 업데이트시켜주고, output_port에 레지스터 값을 출력해주며, 메모리에서 읽어온 데이터를 저장하고, 여러 서브 모듈간의 data를 연결하는 datapath 전반을 구현하였다.

3.2. control_unit 모듈

```
// update state
always @(posedge clk) begin
  if (!reset_n)
    current_state <= 0;
  else begin
    current_state <= next_state;
  end
end
```

[그림 4] control_unit 모듈 synchronous logic part의 코드 중 일부. Posedge마다 state를 업데이트 시킨다.

```
// logic for next state
always @(*) begin
  case(current_state)
      IF: next_state = ID;
      ID: begin
        case(opcode)
            4'd15: begin // R-Type
              case(func_code)
                `INST_FUNC_JPR: next_state = JPR;
                `INST_FUNC_JRL: next_state = JRL;
                `INST_FUNC_WWD: next_state = WWD;
                `INST_FUNC_HLT: next_state = HLT;
                default: next_state = R_EX;
              endcase
            end
```

```
case (current_state)

  IF: begin
    mem_read = 1;
    i_or_d = 0;
    ir_write = 1;
    pc_write = 1;
    alu_src_A = 2'b00;
    alu_src_B = 2'b01;
    alu_op = 0;
    new_inst = 0;
  end
```

[그림 5] control_unit 모듈 combinational logic part의 코드 중 일부. 현재 state와 instruction에 맞게 next state와 출력할 signal들을 결정한다.

control_unit 모듈은 clock에 synchronous하게 state를 업데이트 시키는 synchronous logic 부분과, 현재 state와 input에 맞게 시그널과 next state를 정하는 combinational logic으로 구현하였다. 예시를 위하여 IF, ID state일 때의 next state logic과 signal 결정 logic만을 나타내었다. 그 외 동작들은 Design 단계에서 결정한 동작과 거의 동일한데, 이는 [그림 1] 에서 나타내었다.

3.3. alu_control_unit 모듈

```
always @(*) begin
    if(!ALUOp) begin // common operation
        funcCode = `FUNC_ADD;
        branchType = 0;
    end
    else begin // individual operations
       case (opcode)
            `ALU_OP: begin
                case(funct)
                    `INST_FUNC_JPR: funcCode = `FUNC_ID1; // pc <- rs
                    `INST_FUNC_JRL: funcCode = `FUNC_ID1; // pc <- rs
                    `INST_FUNC_WWD: funcCode = `FUNC_ID1; // outputport <- rs
                    default: funcCode = funct[3:0];
                endcase
            end
```

[그림 6] alu_control_unit 모듈의 일부. Instruction의 opcode와 funct, ALUOp 시그널에 따라 적절한 funcCode, branchType을 생성해 출력한다.

alu_control_unit은 control signal에 따라 적절한 alu control line을 생성하는 부가 control unit으로서, instruction의 opcode와 funct, control signal인 ALUOp에 따라 적절한 funcCode, branchType을 생성하는 alu_control_unit 모듈의 combinational logic으로 구현하였다. ALUOp 비인가시에는 공통연산인 ADD를 실행하고, 인가시에는 instruction별로 각각 알맞은 연산을 실행하도록 한다.

3.4. alu 모듈

```
always @(*) begin

   // reset output
   C = 0;
   overflow_flag = 0;

   case(func_code)
      `FUNC_ADD: begin
         C = A + B;
         // NOTE : From Soomin's labl overflow detection
         if((A[`NumBits - 1] === B[`NumBits - 1])
         && (A[`NumBits - 1] !== C[`NumBits -1]))
            overflow_flag = 1;
         else
            overflow_flag = 0;
      end
      `FUNC_SUB: begin
         C = A - B;
         if((A[`NumBits - 1] !== B[`NumBits - 1])
         && (A[`NumBits - 1] !== C[`NumBits -1]))
            overflow_flag = 1;
         else
            overflow_flag = 0;
      end
```

[그림 7] alu 모듈의 일부. func_code에 따라 두 input에 대해 적절한 연산을 수행하여 그 결과를 반환한다. Overflow를 체크하고, branch condition이 맞는 경우에는 bcond를 인가시킨다.

alu 모듈은 alu_control_unit에서 생성한 func_code와 branch_type에 따라 적절한 연산을 수행하여 output해주도록 구현하였다. ADD, SUB연산시에는 overflow 여부를 체크하여 overflow_flag를 인가시키고, branch type에 따라 조건이 맞는 경우 bcond를 인가시킨다. alu 모듈은 alu_control_unit 시그널에 따라 한 instruction의 다른 cycle에서 여러번 재사용된다.

3.5. 기타 모듈

Register_file, immediate_generator, mux2_1, mux4_1 모듈을 구현하여 [그림2]와 같이 연결해주었다. 이 모듈들은 control signal에 의존하여 적절한 동작을 수행한다.

4. Discussion

- 기존에는 control unit의 state는 물론 control signal 까지 posedge clk을 업데이트 하려고 하였으나 state 변경 및 next state계산, control signal output이 한번에 변경되지 않는 문제가 있었다. 이로 인해 state는 clock synchronous하게 바꾸고, next state 계산 및 control signal output은 combinational logic으로 변경, 구현하여 문제를 해결하였다. 실제 micro-code 구현도 output에는 PLA, ROM 등 combinational logic을 사용하기 때문에 위 구현이 더 적절한 것으로 판단하였다.

- 가장 오래 걸리는 instruction에 맞추지 않고 개별 instruction의 clock cycle이 달라도 되는 점, Resource를 최소화하면서 한 instruction에 같은 resource를 여러번 재사용할 수 있는 점 등, Single Cycle CPU와 비교하여 Multi Cycle CPU가 가지는 장점들을 확인할 수 있었다.

5. Conclusion

Multi Cycle CPU 구현을 통해 Single Cycle CPU와는 달리 한 instruction을 여러 step으로 나누어 효율을 높였으며 Resource Reuse 면에서 좋은 구조임을 이해하였고, 이를 위해 control unit을 finite state machine으로 구현하며 Micro code 작동방식을 이해할 수 있었다.