Lab05 multicycle cpu report

2019-12172 이재원

1. Introduction

Multicycle cpu의 거의 모든 명령을 구현하며, 5 stage로 나눈 multicycle을 구현하는 것이 목표이다.

2. Design

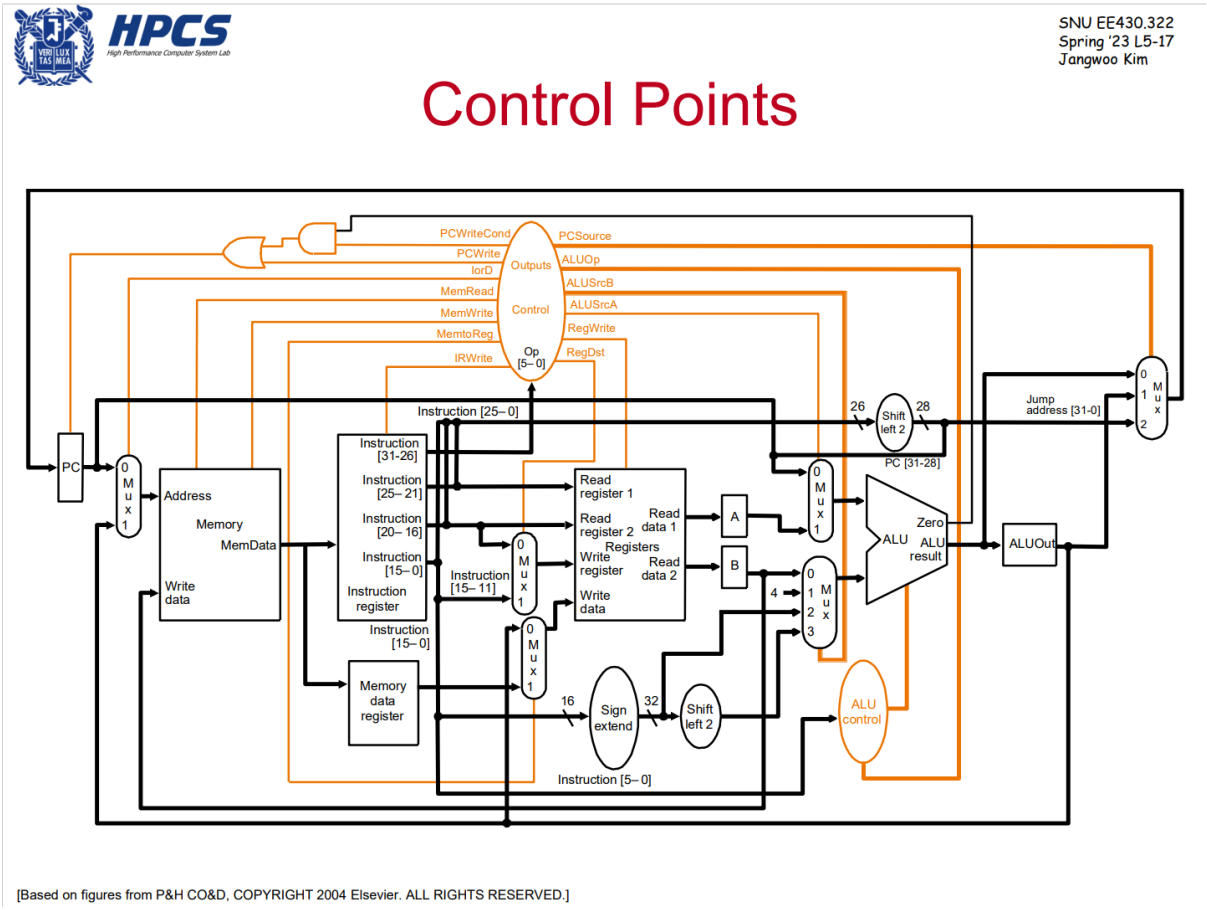

Figure 1. multicycle cpu lecture slide p.17

Figure 1을 바탕으로 design할 예정이며, cpu.v module 안에 MDR, IR, control_unit, datapath module을 만들 것이다. 또한, datapath에서 ALUOut과 ALU result는 서로 1cycle 차이 나는데 이는 branch 연산시 필수적이므로 ALUOut 또한, 새로운 module로 만들 것이다. Figure 1에서는 Read data 1, Read data 2의 결과를 각각 A, B라는 register에 저장하는데 이는 control signal을 stage에 맞게 출력하면 불필요한 요소라고 생각하여 구현하지 않을 예정이다.

또한, 기본적으로 single_cycle의 code를 재사용할 예정이다.

3. Implementation

① cpu.v

하위 module에서 instruction fetch 등을 전부 하므로 num_inst를 계산하는 것 외에 특별한 역할은 없다. 여러가지 하위 module의 signal을 연결한다.

A. IR.v

inputReady일 때 instruction을 fetch해서 새로운 명령을 fetch할 때까지 instruction을 저장한다. Reg를 이용하여 쉽게 구현할 수 있다.

B. MDR.v

IR과 비슷하게 memory data를 저장하는 것이다. Control signal이 없기 때문에 1 cycle 뒤에 출력하는 flip flop이라고 볼 수 있다.

C. Datapath.v

기본적으로 figure 1의 datapath를 바탕으로 만들어서 signal 이름도 같게 만들었으나 몇 가지 다른 점이 있다. 먼저, RF를 write하는 부분이 달라졌다. JAL 같은 명령을 수행하기 위해서 2번 register에 write해야 하기 때문에 write register address를 선택하는 MUX(RegDst)가 1개 늘어났고, write data를 선택하는 MUX(MemtoReg)가 pc + 1을 선택하는 signal이 추가되었다. code에서 MUX 관련된 부분은 따로 주석으로 MUX라고 쓰여있어서 쉽게 이해할 수 있다.

또한, PC source도 MUX에서 register (2번)의 값을 가져오는 부분이 추가되었다.

다음은 datapath의 하위 module이다.

i. ALU.v

Single_cycle일 때와 비슷하나 singl_cycle에서 모든 명령을 구현하지 않아서 추가하지 않는 연산을 구현했다. 이름은 TSC assembly와 동일하게 하여 이해하는 데 어려움이 없다.

ii. ALUOut.v

Branch 연산시 ID 단계에서 구한 PC + imm + 1의 값을 저장하기 위해 만들었다. 이 결과는 EX단계에서 사용하기 때문에 1cycle 동안만 값을 저장하는 flip flop이다.

iii. RF.v

Single_cylcle일 때와 달라진 점이 없다.

iv. PC.v

Figure 1에서 PC는 단순히 명령을 저장하는 register로만 표현되어있기 때문에 pc.v에서는 어떠한 연산을 수행하지 않고 저장만 하는 register이다. 오직 write하는 signal(pc_en)이 들어오면 clock에 맞춰 쓰기만 한다. 여기서 나온 결과가 IorD MUX를 거쳐 memory에 가고, memory write하지 않는 이상 memory address가 바뀌도 전혀 지장 없기 때문에 PC.v는 항상 결과를 출력한다. 따라서 waveform을 보면,

D. Control_unit.v

Control_unit의 경우 stage가 state이기 때문에 각 stage를 update하는 부분, next state를 계산하는 부분, 현재 state에 따라 output을 계산하는 부분으로 나눌 수 있다.

```
204     // sequential logic for state(stage)
205     always @ (posedge clk) begin
206         if (!reset_n) begin next_stage <= `STAGE_IF; stage <= 0; end
207         else stage <= next_stage;
208     end
209
210     // combinational logic for next state (next_stage)
211     always @ (*) begin
212         case (stage)
213                 `STAGE_IF : next_stage = `STAGE_ID;
214
215                 `STAGE_ID: if (isJtype_Jump || isRtype_Special) next_stage <= `STAGE_IF;
216                     else if ( (opcode == `typeR) && (func_code == `FUNC_HLT) ) next_stage = `STAGE_ID;
217                     else next_stage = `STAGE_EX;
218
219                 `STAGE_EX: if (isItype_Branch) next_stage = `STAGE_IF;
220                 else if (isItype_Memory) next_stage = `STAGE_MEM;
221                 else next_stage = `STAGE_WB;
222
223                 `STAGE_MEM: if (opcode == `OPCODE_LWD) next_stage = `STAGE_WB;
224                 else next_stage = `STAGE_IF;
225
226                 `STAGE_WB: next_stage = `STAGE_IF;
227         endcase
228     end
```

Figure 2. sequential logic for state & combinational logic for next state in control_unit.v

Control_unit.v code의 마지막에 stage를 update하는 부분이 있으며, 여기서 present state는 stage, next state는 next stage로 구현했다. 이 부분은 description에 나온 부분을 그대로 구현하여 특별한 설명은 필요 없을 것이라고 생각된다.

현재 stage에 따라 output을 출력하는 부분은 앞부분이며 각 signal별로 output을

구현했다. 주석으로 어떤 signal인지 명시했으며, PC_en 신호의 경우 figure 1에서 PC_write, PC_writeCondition, ALU_zero를 합친 signal이다.

4. Discussion

JAL, JR같은 명령이 figure 1에 없어서 따로 datapath를 수정해야 했다. 또한, MUX의 input과 control signal도 늘어나게 되었다. PC의 경우 MIPS는 left shift 2번 하는 signal이 있는데 TSC에서는 없어 ALUSrcB MUX input이 줄어들었다.

또한, figure 1에 나와있는 대로 구현하다보니 PVS enable을 사용하지 않았는데 이로 인해 address가 매 stage마다 바뀌는 것이 waveform에서 보여 debugging하는데 어려움을 겪었다. 하지만 PVS en이 없더라도 기능에는 문제가 없어 따로 추가하지 않았다.

5. Conclusion

Multicycle cpu를 오류 없이 구현하여 all pass했다.