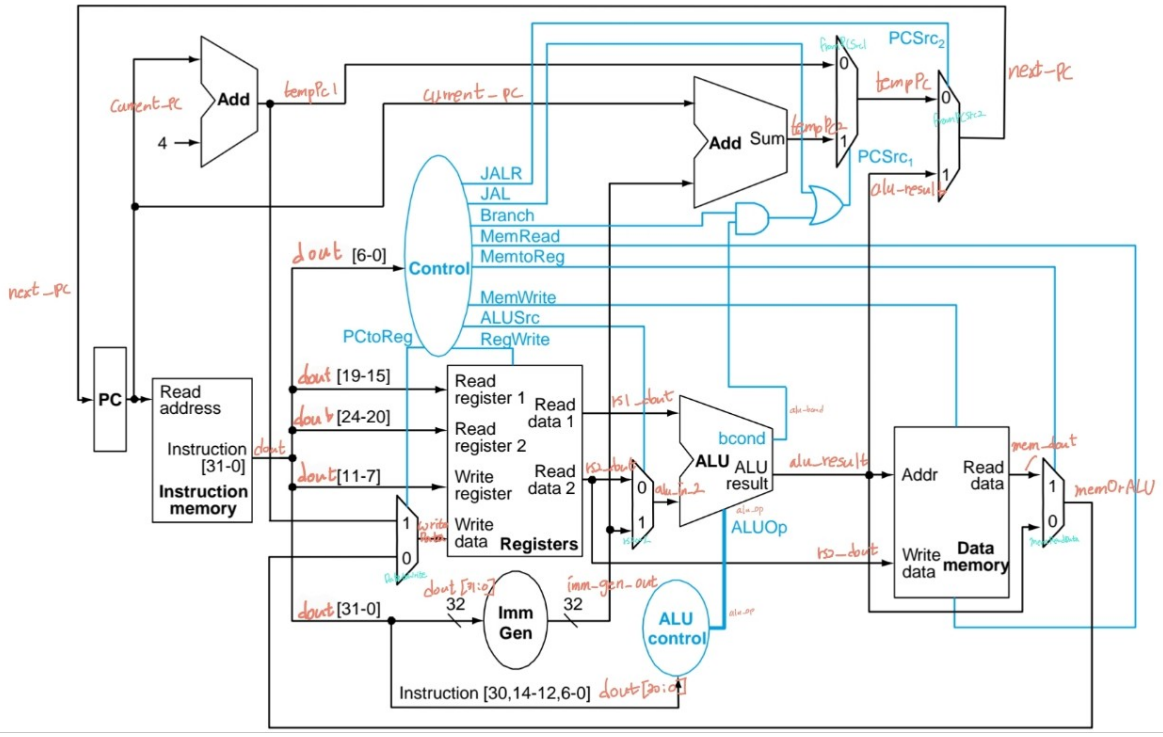
Single cycle CPU

1) Introduction

Single cycle CPU를 구현하는 실습이다. cpu.v에 주어진 스켈레톤 코드를 바탕으로 수업시간에 배운 single cycle CPU를 구현한다. 배열을 이용해 가상의 메모리를 만들어 이를 읽어오는 방식으로 구현한다.

이를 구현하기 위해 주어진 cpu, Memory, opcodes, RegisterFile 외에 adder, ALU, ALUControlUnit, ALUOps, ControlUnit, immediategenerator, mux, pc 모듈을 추가했다.

2) Design



Single cycle CPU는 instruction fetch 단계에서 현재 PC 값에 해당하는 instruction을 instruction memory로부터 읽어온다. 또한 PC register에서는 다음 명령어 주소를 업데이트 하여 다음 instruction fetch를 준비한다.

Instruction decode 및 operand fetch 단계에서는 register file에서 필요한 레지스터 값을 읽어오고 instruction을 해독하여 필요한 control signal을 발생시키며, 명령어 별 필요한 immediate value를 발생시킨다.

Execute 단계에서는 ALU 연산이 진행된다.

Memory access 단계는 load, store instruction만 동작하며 메모리로부터 값을 읽어온다.

Write back 단계에서는 메모리에서 읽어온 값을 레지스터에 저장하거나 alu 연산 결과를 메모리에 저장하는 등 write 과정이 일어난다.

clock synchronous module: InstMemory의 초기화 부분, DataMemory의 초기화 및 write 부분, pc, RegisterFile 초기화 및 write 부분
asynchronous module: adder, ALU, ALUControlUnit, ControlUnit, ImmediateGenerator, InstMemory의 read 부분, DataMemory의 read 부분, mux, RegisterFile의 read 부분

Top module> cpu.v
Reset, clk를 입력으로 받고 프로그램 종료 시 is_halted를 set하여 출력한다.
위의 그림과 같이 각 모듈과 멀티플랙서, adder 사이 wire를 연결했다. 또한 각 모듈을 제어하는 control signal도 wire를 이용해 적절히 연결했다. 프로그램의 종료를 알리는 operation이 읽히면 ControlUnit 모듈에서 is_halted를 set하여 cpu 모듈을 종료한다.

Submodules>

먼저 datapath에 포함되는 모듈들을 소개한다.

1) PC : 매 클럭마다 pc를 업데이트 한다. Clock synchronous 모듈이다. Instruction fetch 과정에서 동작한다.
2) InstMemory: 처음 cpu가 reset 입력을 받았을 때, 주어진 경로의 파일을 읽어서 instruction을 저장하는 mem 배열에 입력한다. 이후 현재 pc 값이 들어올 때마다 해당 위치의 명령어를 asynchronously read한다. Instruction fetch를 위해 메모리로부터 명령어를 읽어온다.
3) RegisterFile: cpu가 reset될 때, positive edge clokc에서 clock synchronous하게 32개의 레지스터 값을 초기화한다. 이후 rs1, rs2와 is_halted 판단을 위한 register 17의 값을 asynchronous하게 읽는다. Rd에 저장할 입력값이 들어오면 clock synchronous하게 해당 레지스터에 값을 저장한다. Instruction decode를 위해 레지스터에서 값을 읽어온다.
4) ImmediateGenerator: I-type, load, store, JAL, JALR, branch instruction에 필요한 상수를 만드는 모듈이다. Instruction decode를 위해 적절한 상수를 만들어 execute stage로 넘긴다.
5) ALU: register file을 비롯한 다양한 source에서 들어온 rs1 data와 rs2 data를 이용해 입력된 operation에 해당하는 연산을 수행하고 결과를 출력한다. 이 모듈은 Asynchronous하게 동작한다. Execute stage에서 연산을 담당한다.
6) DataMemory: cpu가 reset 될 때, 전체 메모리를 초기화하고 이후, data를 clock synchronously write하고 asynchronously read한다. MEM stage에 해당하는 모듈로, 필요한 메모리 값을 읽어온다.
7) Adder: 주로 새로운 pc를 계산하는 데 사용하며 두 개의 adder가 사용된다. Adder는 pcplus4, pcplusImm으로 다음 instruction으로 넘어가기 위한 pc+4연산을 하는 adder와 JAL, JALR, branch 등을 위한 pc+immediate value 연산을 위한 adder가 있다.
8) Mux: 두 입력이 들어왔을 때 적절한 control signal에 따라 둘 중 한 입력을 선택해 출력한다. 본 실습에서는 DatatoWrite, memReadData, fromPCSrc1, fromPCSrc2, rs2orI의 5개 멀티플랙서(mux)를 사용한다.

A. DatatoWrite: register file에 적을 데이터를 PCtoReg 값에 따라서 pc+4와 memReadData에서 보낸 값 중 하나를 선택한다.
B. memReadData: MemtoReg 값에 따라서 메모리에서 읽은 값 또는 alu 계산 결과 중 하나를 선택한다.
C. fromPCSrc1: PCSrc1 값에 따라서 pc+4 또는 pc+imm 중 하나를 선택한다.
D. fromPCSrc2: is_jalr 값에 따라서 fromPCSrc1에서 보낸 값과 alu 연산 결과 중 하나를 선택한다.
E. rs2orI: ALUSrc 값에 따라 rs2에서 읽은 값과 immediate 값 중 하나를 선택한다.

이제 control과 관련된 모듈을 소개한다.

9) ALUControlUnit: instruction fetch 이후, instruction 전체를 가져와서 opcode와 funct3 부분을 통해 ALU에서 어떤 연산을 수행할지를 alu_op wire를 통해 출력한다. alu_op에 들어가는 값은 AluOps.v에서 정의한 연산 별 상수 값을 이용한다. Instruction decode 단계에서 제어 신호를 발생시킨다.
10) ControlUnit: opcode를 받아서, instruction 종류에 따라 필요한 control signal을 발생시킨다. Control signal에는 입력 받은 instruction에 따라 다음의 control signal을 set한다. Instruction decode 단계에서 제어 신호를 발생시킨다.
    A. instruction이 jal인지 판단하는 is_jal
    B. jalr인지 판단하는 is_jalr
    C. branch 명령어인 branch
    D. 메모리 읽기가 가능한지 명시하는 mem_read
    E. Register 읽기가 가능한지 명시하는 mem_to_reg
    F. 메모리 쓰기가 가능한지 명시하는 mem_write
    G. Register 쓰기가 가능한지 명시하는 reg_write
    H. Mux를 위한 signal: alu_src, pc_to_reg
    I. 마지막 instruction인지 명시하는 is_halted
    J. is_ecall

나머지 파일에는 instruction이나 ALU에서 실행되는 operation 종류를 가리키는 상수를 정의해두었다.

- AluOps.v: ALU에서 수행할 연산의 종류에 대해 상수 값을 정의함
- opcodes.v: RISC-V instrction들의 opcode, funct3 부분에 대해 상수 값을 정의함

3) Implementation

Top module> cpu.v
Input: reset, clk

Logic: 위에서 첨부한 그림과 같이 wire를 선언하고 주어진 스켈레톤 코드에서 멀티플랙

서 5개, adder 2개를 추가한다. 또한, mux fromPCSrc1에서 사용할 PCSrc1을 발생시키기 위해 pc_src_1에 control signal branch, is_jal과 alu_bcond를 이용해 적절한 값을 assign 한다.

Output: is_halted
CPU에서 돌린 프로그램이 종료되었음을 나타내는 신호를 출력한다.

Submodules>

1) PC
   Input: reset, clk, next_pc

   Logic: 매 positive clock edge마다 current_pc를 next_pc로 업데이트 한다. 이때 reset이 1이면 current_pc를 0으로 초기화한다.

   Output: current_pc

2) InstMemory
   Input: reset, clk, addr(current_pc)

   Logic: positive clock edge 동안 reset이 1이면 주어진 경로에 존재하는 파일로부터 명령어를 읽어오고 각각을 mem에 저장한다. Reset이 0이면 addr이 들어오면 asynchronous하게 해당 위치의 명령어를 읽는다.

   Output: dout에 읽은 명령어를 저장해서 내보낸다.

3) RegisterFile
   Input: reset, rs1, rs2, rd, rd_din, reg_write

   Logic: Asynchronous하게 일어나는 동작은 register에 있는 값을 읽어가는 동작이다. rs1, rs2이 입력되면 rs1_dout, rs2_dout에 해당 레지스터 값을 읽어서 대입한다. 또한 ECALL 명령어가 프로그램 종료를 의미하는지 판단하는데 쓰일 17번 레지스터 값을 읽는다.
   positive clock edge에 synchronous하게 일어나는 동작은 레지스터에 어떤 값을 쓰는 동작이다. Positive clock edge에 reset이 1이면 모든 레지스터 값을 0으로 초기화하고 pc 값은 0x2ffc로 설정한다. Reset이 1이 아니면 rd 레지스터에 rd_din 값을 저장한다.

   Output: rs1 레지스터의 데이터 rs1_dout, rs2 레지스터의 데이터 rs2_dout, 17번 레지스터 값 rf17

4) ImmediateGenerator

Input: inst

Logic: InstMemory에서 읽어온 instruction 전체를 받아서 opcode 부분을 이용해 각 instruction 종류에 맞는 상수를 생성한다.
Instruction이 상수 연산이나 Load 연산인 경우, store 연산, branch 연산, JAL 또는 JALR인 경우로 나눠 각 inst에서 상수 부분인 비트를 32비트 수로 signed-extend 한다. 만들어진 상수는 imm_gen_out에 담아 출력한다.

Output: imm_gen_out

5) ALU

Input: ALU 연산 종류를 나타내는 alu_op, rs1 data인 rs1_dout, rs2 data 또는 상수 값인 alu_in_2

Logic: 이 모듈은 alu 연산 결과인 alu_result와 branch 연산에서 branch의 taken 여부를 판단하는 alu_bcond 신호를 발생시킨다. input 중 하나라도 들어오면 아래 로직이 동작한다.
Input 중 하나라도 변하면 alu_result, alu_bcond를 0으로 초기화한다. 이후 alu_op에 따라 그에 맞는 연산을 진행한다. Branch instruction의 경우, 각 명령어마다 두 입력 데이터가 같은 지, 다른 지, 보다 작은 지, 보다 크거나 같은 지를 판단해 alu_bcond를 set하고 alu_result는 0으로 출력한다.
산술 연산의 경우 add, sub, sll, xor, or, and, srl에 따라 두 데이터에 해당하는 산술 연산을 진행하고 alu_result에 연산 결과를 저장해 출력한다.

Output: alu_result, alu_bcond

6) DataMemory

Input: reset, clk, addr(alu_result), din(rs2_dout), mem_read(memory 읽기 가능 여부), mem_write(memory 쓰기 가능 여부)

Logic: 이 모듈은 32비트의 reg를 MEM_DEPTH개 만큼 배열로 만들어 memory mem으로 사용한다. Memory write 동작은 positive clock edge에 일어나며, reset이 1이면 mem 전체를 0으로 초기화하고, reset이 0이면 mem_write가 1일 때 din의 데이터를 mem의 addr 위치에 쓴다. Memory read 동작은 asynchronous하게 일어나며 mem_read가 1일 때 dout에 addr 위치의 메모리 값을 저장해 출력한다.

Output: dout(특정 위치의 메모리 값)

7) Adder: pcplus4, pcplusImm의 adder가 존재
Input: add1, add2

Logic: addout에 add1+add2한 값을 assign한다.

Output: addout

A. pcplus4
input: current_pc, constant 4 / output: tempPc1

B. pcplusImm
input: current_pc, imm_gen_out / tempPc2

8) Mux
Input: mux_in1, mux_in2, control

Logic: mux_out에 control이 0이면 mux_in1을, control이 1이면 mux_in2를 assign한다.

Output: mux_out

A. DatatoWrite
input: memOrALU(memReadData의 결과값), tempPc1, pc_to_reg
output: writeData

B. memReadData
input: alu_result, mem_dout(메모리에서 읽어온 값), mem_to_reg
output: memOrALU

C. fromPCSrc1
input: tempPc1, tempPc2, pc_src_1
output: tempPc

D. fromPCSrc2
input: tempPc, alu_result, is_jalr
output: next_pc

E. rs2orI
input: rs2_dout, imm_gen_out, alu_src
output: alu_in_2

9) ALUControlUnit
Input: part_of_inst

Logic: 받아온 instruction 전체인 part_of_inst에서 opcode 부분과 funct3 부분, sub instruction을 나타내는 30번 비트를 reg에 저장한다. 이후 opcode와 funct3에 따라 case를 나눠 alu_op에 적절한 alu 연산 종류를 입력한다.
Branch instruction의 경우 funct3에 따라 각각 BEQ, BNE, BLT, BGE의 상수를 alu_op에

입력한다.
Arithmetic instruction의 경우 각 funct3와 is_sub 값에 따라 ADD, SUB, SLL, XOR, OR, AND, SRL을 alu_op에 입력한다.
Immediate arithmetic instruction의 경우 funct3에 따라 ADD, SLL, XOR, OR, AND, SRL을 alu_op에 입력한다.
JALR, Load, Store의 경우에는 alu에서 덧셈 연산만 하므로 alu_op에 ADD를 입력한다.

Output: alu_op

10) ControlUnit
Input: reset, part_of_inst(opcode), rf17(17번 레지스터 값)

Logic: asynchronous하게 동작하며 opcode에 따라 다른 control signal을 출력한다. Always 문 안에서 모든 control signal is_jal, is_jalr, branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write, pc_to_reg, is_ecall을 0으로 초기화한다. 이후 opcode인 part_of_inst 값을 이용해 arithmetic 연산이면 reg_write를 1로 대입한다.
Part_of_inst가 immediate arithmetic 연산이면 reg_write, alu_src를 1로,
Load 연산이면 reg_write, mem_read, mem_to_reg, alu_src를 1로,
JALR 연산이면 reg_write, is_jalr, alu_src, pc_to_reg를 1로,
Store 연산이면 mem_write, alu_src를 1로,
Branch는 branch를 1로
JAL은 is_jal, pc_to_reg를 1로
ECALL이면 is_ecall을 1로 대입하고, rf17이 10을 저장하는지 확인 후 is_halted를 1로 대입한다.

Output: is_jal, is_jalr, branch, mem_read, mem_to_reg, mem_write, alu_src, reg_write, pc_to_reg, is_ecall, is_halted

11) Discussion

cpu.v에서 같은 이름을 여러 wire에서 사용해 data가 엉뚱한 곳으로 가는 문제가 있었다. 이를 해결하기 위해 기존에 같은 이름으로 작성했던 wire의 이름을 모두 다르게 수정했다. 또한, cpu.v 파일에서 모듈 간 포트와 wire를 연결하는 방식을 이해할 수 있었다.
Immediategenerator.v에서 상수를 생성할 때, 32비트 상수로 sign-extend 해주지 않아 엉뚱한 결과값이 나왔다. 그러나 sign-extend를 위해 $signed를 사용하자 올바른 결과값을 얻을 수 있었다.
정의한 상수 이름을 사용하기 위해서는 각 상수 이름 앞에 `를 붙여야 하지만 처음에 이를 붙이지 않아 상당한 컴파일 오류가 발생했다. 또한, 상수를 정의한 파일을 `include하지 않아 발생한 컴파일 오류를 상수 사용 형식을 알게 됨으로써 모두 수정했다.

12) Conclusion

수업시간에 배운 single cycle CPU를 직접 구현함으로써 datapath와 control의 구체적인 동작을 자세히 이해할 수 있었다. 특히, data를 쓰는 동작은 PVS인 pc, register file, memory 부분을 변화시키는 동작이므로 clock synchronous하게 동작해야 함을 이해했다.