[Lab04] Single-cycle CPU

전기·정보공학부 김선우 (2019-16380)

Introduction

Single-cycle로 작동하는 CPU를 설계한다. TSC ISA 중 일부를 실행할 수 있는 마이크로아키텍처를 Verilog로 작성한다. 명령어 형식 및 CPU 모듈 인터페이스는 다음과 같다.

Figure 1 ISA formats

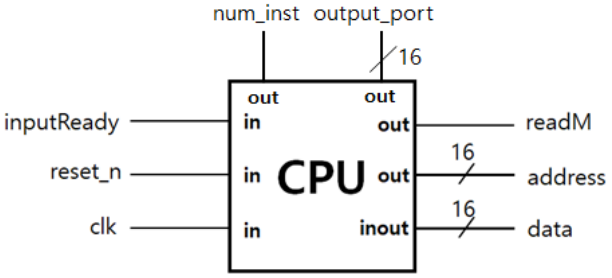

Figure 2 CPU module interface

모듈 인터페이스의 포트는 다음과 같다. reset_n과 clk는 각각 리셋과 클락이고, inputReady, readM, address, data 4개의 포트는 메모리와 연결된다. readM이 High이고 address에 메모리 주소를 출력했을 때, 일정 딜레이 이후 data 포트로 address 위치의 값이, inputReady가 High로 일정 시간 동안 입력된다.

num_inst, output_port는 테스트 및 디버깅을 위한 출력 포트로, 각각 현재까지 실행한 명령어 수와 WWD 명령의 결과로 출력할 데이터 값을 출력한다.

구현할 명령어는 총 5개로, 그 형식 및 기능은 다음과 같다.

Table 1 구현 대상 명령어

Assembler Format	Operation
• ADD $rd $rs $rt	• $rd <- $rs + $rt • pc <- pc + 1
• ADI $rt $rs imm	• $rt <- $rs + (sign-extend imm) • pc <- pc + 1
• LHI $rt, imm	• $rt <- { imm, 00000000 }
• JMP target	• pc <- { pc[15:12], target }
• WWD $rs	• output_port <- $rs • pc <- pc + 1

Design

구현해야 할 전체 회로는 다음과 같다. (source: 강의자료)

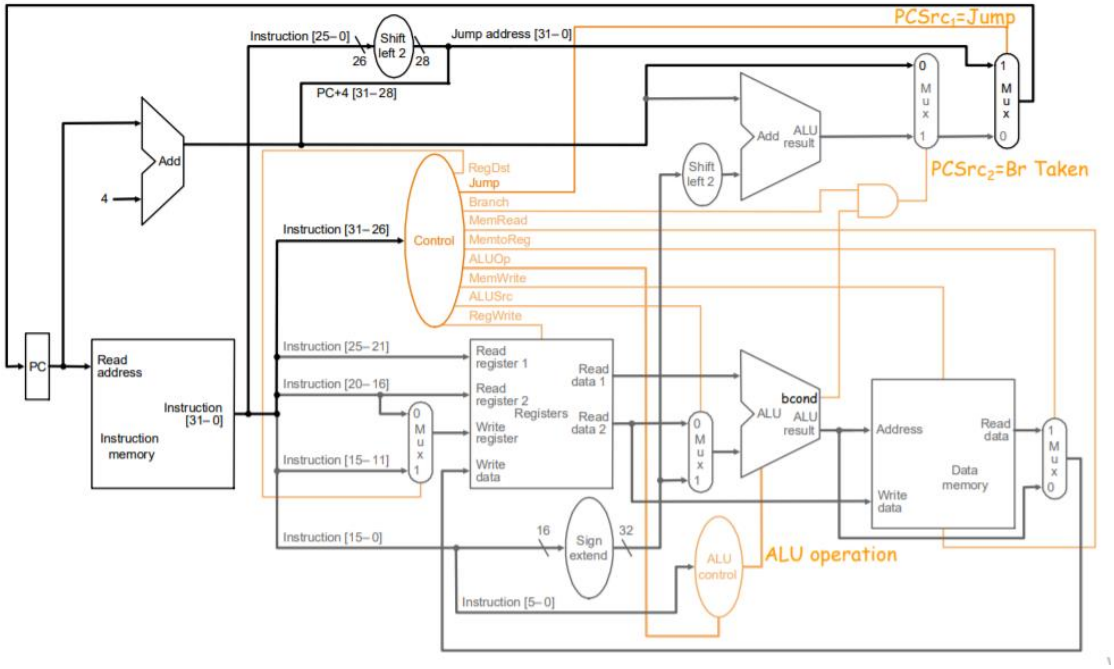

Figure 3 Circuit Diagram

이번 실습에서 구현해야 하는 명령어는 ADD, ADI, LHI, JMP, WWD의 5개이므로, 위 회로도에서 branch condition을 확인하고 branch address를 구하는 부분은 구현하지 않는다. 또, 데이터 메모리를 사용하는 회로도 구현하지 않는다. 그러나 output_port, num_inst를 출력할 회로와 관련 control 신호는 추가로 구현해야 한다. MemRead는 instruction을 읽기 위해 외부에 출력하는 신호로, Control 모듈이 아닌 PC에서 구현한다.

모든 명령이 한 클락 사이클 동안 실행되므로, PC에 새로운 명령어 주소를 업데이트하는 타이밍을 클락의 상승 에지로 해 실행 주기를 조절한다. 구현해야 할 Control과 Datapath 서브모듈들의 구체적인 인터페이스 및 기능은 다음과 같다.

1. Controls

1) Control (Main control)

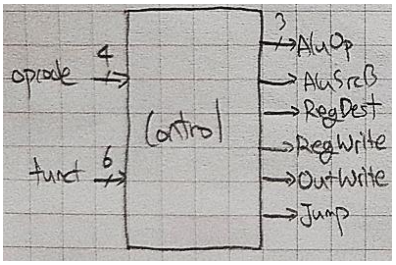

Figure 4 Control (Main) submodule interface

: Instruction이 업데이트된 이후에, opcode와 funct 비트 값에 따라 datapath의 각 모듈을 제어하는 신호를 Combinational logic으로 출력한다. Opcode, funct 값이 업데이트될 때마다 신호가 업데이트된다. (funct는 WWD를 구별하기 위해 사용한다)

본 실습에서는 PC의 주소로 명령어를 읽어올 때만 메모리의 값을 읽어오므로, MemRead 신호는 PC 모듈에서 출력하고, Control은 Jump, RegDst, RegWrite, ALUSrc, ALUOp과 output_port에 값을 쓰기 위한 OutWrite 신호, 총 6개 신호를 출력한다.

2) ALU control

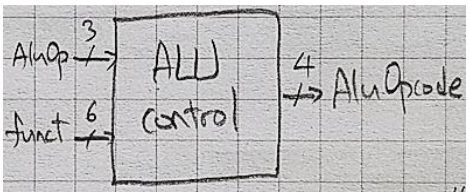

Figure 5 ALU Control submodule interface

: Instruction이 업데이트된 이후에, Main Control 모듈에서 출력한 AluOp 신호와 funct 비트에 따라 ALU에서 출력할 연산 결과를 결정하는 신호를 Combinational logic으로 출력한다. Opcode 값이 15 (R-type)일 때는 funct 값에 따라, opcode 값이 15가 아닐 때에는 opcode 값에 따라 신호를 출력하도록 AluOp 신호가 opcode에 대한 정보를 갖고 있고, 이 AluOp과 funct로 ALU Control의 출력 신호를 조합한다. Opcode, funct 값이 업데이트될 때마다 신호가 업데이트된다.

2. Datapath

1) PC (Program Counter)

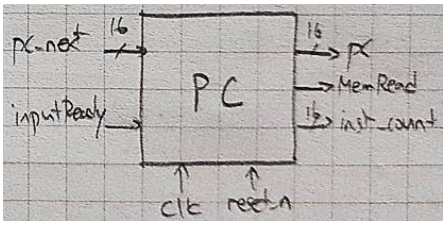

Figure 6 Program Counter submodule interface

: 클락 clk, 초기화 reset_n 입력이 있다. Multi-cycle CPU처럼 PVSWrite(Program Visible State Write) 신호가 있는 경우가 아니므로, Control이 클락을 받아 사이클을 개시하는 신호를 출력하게 하는 것보다 PC에서 클락에 따라 사이클이 시작할 때마다 메모리에 명령어 주소 하나를 전달해 읽어오게 하는 것이 간결하다. 메모리에서 명령어를 읽기 위해 전달해야 하는 MemRead 신호도 PC 모듈에서 출력한다. Inst_count는 테스트 및 디버깅을 위해 현재 실행하는 명령어의 번호를 나타내는 값이다.

2) Instruction

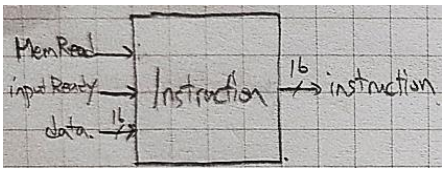

Figure 7 Instruction submodule interface

: 사이클마다 메모리에서 읽어온 명령어 값을 저장하고 업데이트하는 모듈이다. inputReady 신호와 readM 값을 고려해, 사이클 당 한번 유효한 명령어 값을 읽어온다.

3) Register File

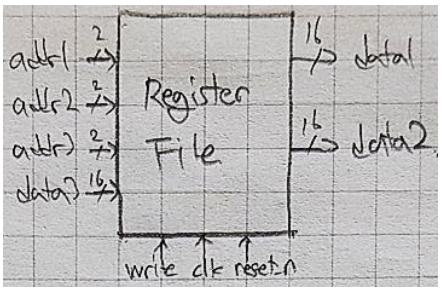

Figure 8 Register File submodule interface

: 클락의 posedge에 동기적으로 값을 쓰고, 비동기적으로 값을 읽어오는 Register File 모듈이다. Reset_n이 Low일 때 비동기적으로 모든 주소의 값을 0으로 초기화한다.

4) ALU

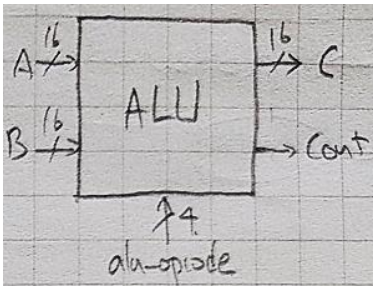

Figure 9 ALU submodule interface

: 16-비트 A, B 입력에 대해 산술/논리 연산을 수행하고, 그 값을 출력하는 ALU 모듈이다. Alu_opcode 값에 따라 출력할 연산 값이 정해지며, 총 16 종류의 연산을 구현할 수 있다. 조합논리회로로 구현되며, 입력 값이 바뀔 때마다 업데이트된다.

서브모듈들을 알맞게 wire로 연결해 전체 회로를 설계한다.

Implementation

각 서브모듈은 다음과 같이 구현했다.

1. Controls

: Control (Main), ALU Control 모두 명령어에 따라 출력 신호를 조합적으로 구성한다. Control 신호는, 명령어 집합이 확장될 수 있게 모든 명령어를 지원하도록 구현했다.

1) Control (Main)

: initial begin으로 Jump에 Low를 대입해 첫 posedge 클락에서 주소 0의 명령어 값을 읽어오게 한다.

- AluOp: R-type 명령어는 funct로 결정하도록, 나머지는 연산을 직접 지정
- AluSrcB: immediate 연산이 필요한 (SWD, Branch, ADI, ORI, LHI, LWD)이면 Low, 나머지 High
- RegDest: R-type에서 High, 나머지 Low
- RegWrite: WWD를 제외한 R-type, ADI, ORI, LHI, LWD에서 High, 나머지 Low
- OutWrite: WWD에서 High, 나머지 Low
- Jump: JMP에서 High, 나머지 Low

2) ALU Control

: AluOp값에 따라, 0인 경우 funct에 따라 case 문으로 연산을 선택하고 0이 아닌 경우 AluOp값에 따라 case 문으로 각각 지정된 연산을 선택한다. 구체적인 내용은 "controls.v" 파일과 "opcodes.v" 파일에 작성했다.

2. Datapath

1) PC

: 클락이 posedge이거나 reset_n이 negedge일 때, 매 사이클을 시작한다. Reset인 경우, pc 값을 -1, MemRead는 Low, inst_count는 0으로 초기화한다.

매 사이클(posedge 클락) 명령어를 읽어오는 PC의 동작은 다음과 같다.

① Next_pc 입력을 pc로 출력, MemRead로 High 출력, inst_count의 값 1 증가 (출력)

② inputReady가 Low->High->Low가 되면, MemRead로 Low 출력

InputReady가 negedge일 때마다 MemRead 출력을 High에서 Low으로 내려, 사이클 당 한번, 메모리 출력이 유효할 때만 명령어 값을 읽어오게 한다.

모듈 밖 메인 CPU 모듈에서, MemRead 신호를 readM 포트로, pc를 address 포트로 출력한다. Pc_next 값은 Jump 일 때 target address로, 아닐 때 pc+1을 갖는 wire로 선언해 PC 모듈에 입력한다.

2) Instruction

: 테스트벤치에 정의된 메모리의 동작을 보면, inout 포트 data를 사용해 readM(여기서는 MemRead) 신호가 High일 때는 메모리의 값을 출력하고, 아닐 때는 입력을 받아오도록 tri state buffer로 구현되어 있다. 또 readM이 High일 때, 일정 읽기 딜레이 이후 출력 값이 data 포트로 전달되고 inputReady로 High가 출력되며, 일정 시간 동안 유지했다가 각각 Z(High-impedance), Low 값을 출력한다. PC에서 inputReady가 다시 Low가 되는 지점, 즉 출력 값이 무효해지는 지점까지 readM에 High를 전달하다가 이때에 0을 인가하므로, 이 순간으로부터 메모리 읽기 딜레이가 지난 지점에서 inputReady 신호가 다시 High가 된다. 이때도 아직 같은 사이클 안이지만, readM이 Low라 data에 메모리 값이 출력되지 않는다.

내부 reg로 instruction을 저장하고 있다가, data의 값이 변하는 순간에 inputReady와 MemRead 신호 모두 High이면 instruction을 data 값으로 업데이트한다

3) Register File

: 지난 주차 실습에서 구현한 것을 사용했다. 비동기적 읽기와 초기화, 클락의 posedge에 동기화된 쓰기를 지원한다. CPU 모듈의 clk, Control 모듈의 RegWrite 신호를 입력한다. 읽기 주소는 명령어에서 해당하는 부분을 wire로 연결한다. 쓰기 주소는 RegDest 신호로 결정하고, 쓰기 데이터는 ALU 모듈의 출력을 연결한다.

4) ALU

: 지난 주차 실습에서 구현한 것을 바탕으로, Cin 포트와 불필요한 연산을 생략했다. 구체적인 연산 구현은 "controls.v" 파일과 "opcodes.v" 파일에 작성했다. Cout 신호는 bcond wire로 연결했으나, 이번 실습에서는 사용하지 않았다. A, B 입력은 Register File 모듈의 출력과 연결한다.

Output_port 의 경우, OutWrite 신호가 High 일 때만 addr1 으로 입력된 $rs 위치의 값이 출력되고 Low 인 경우 Z 값이 출력되도록 작성했다.

Discussion

테스트벤치 메모리 모듈에서 값을 읽어올 때, 딜레이와 readM, inputReady 신호를 고려해 유효한 값의 타이밍을 확인하는 부분을 작성하면서 모듈 간 신호를 주고받는 상황에서 타이밍을 고려해야 하고, 이에 따라 클락 속도를 결정하고 조율할 수 있음을 생각해보았다. 이렇게 모듈 간 신호를 주고받을 때, 회로를 구현하는 일반적인 방법을 추가로 학습해보겠다.

여러 서브모듈을 사용하고 각 모듈 간에 Control 및 Datapath 신호를 연결하는 상황에서, 각 모듈 단위로 기능을 테스트하며 설계하는 법을 익혔다. 신호의 값이 업데이트되는 타이밍을 잘 확인하고, 전체 회로도와 비교하며 설계를 점검하고 디버깅하는 연습이 되었다.

ADI $0, $0, 1과 같이 Register File에서 값을 읽고 쓰는 것을 한 명령어에 하는 상황이 있는데, 이때 읽기와 쓰기의 타이밍 역시 잘 고려할 필요가 있다. Single-cycle 방식의 경우 읽고 쓰는 것이 같은 사이클에 실행되므로 이런 문제가 발생한다. Multi-cycle 방식으로 각 단계를 구분한다면 타이밍 처리가 더 정확하겠다.

이렇게 모듈 간 신호를 주고받을 때 타이밍 문제가 생기므로, 한 클락 사이클에 명령어 전체를 시행하는 것보다 각 명령어의 실행 단계(stage)에 맞추어 클락 사이클을 지정하는 Multi-cycle 방식이 타이밍 조절의 측면에서도 유리하겠다.

Conclusion

제공된 testbench에 대해 시뮬레이션 결과 모두 Pass했고, 추가로 작성해본 테스트 경우에도 잘 작동했다.

CPU 회로를 잘 설계하기 위해서는, 이론상 딜레이를 고려하지 않은 것과 달리 출력 신호 타이밍을 정확하게 계산해야 한다는 것을 알 수 있었다.