# Lab2 Report

**20230642 이채영, 20230683 박한비**

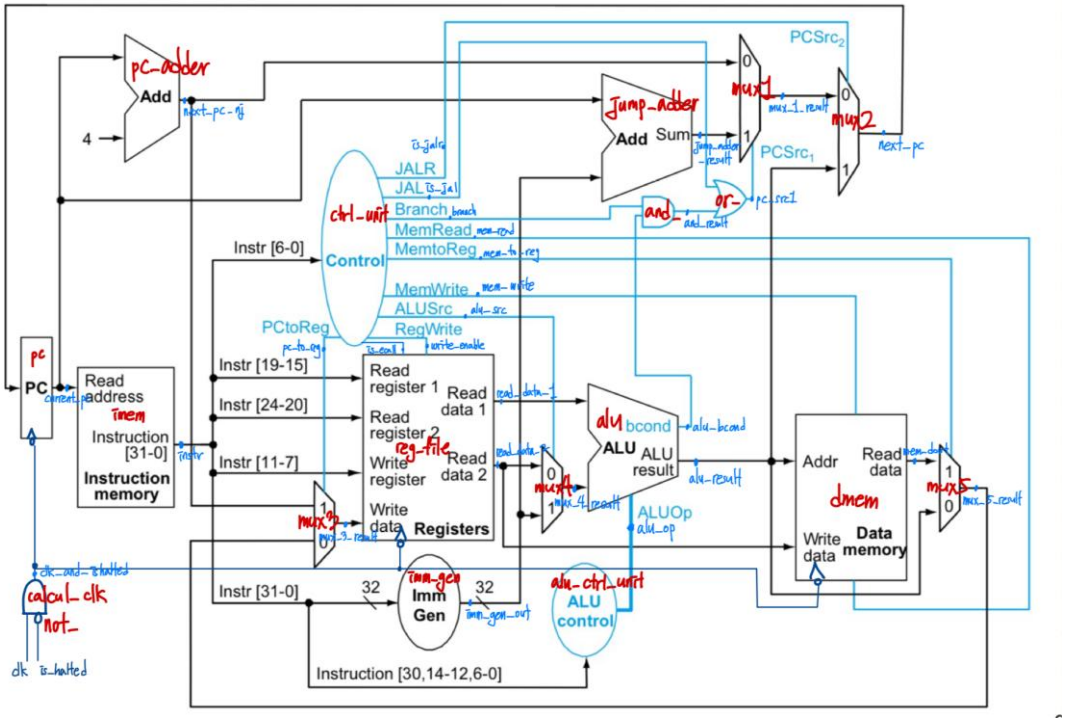
## 1. Introduction

Lab2에서는 RISC-V instruction을 기반으로 하는 Single-Cycle CPU를 구현하였다. Single-Cycle CPU는 하나의 사이클 동안 하나의 명령어를 실행하며, 이는 모든 명령어에 똑같이 적용된다.

## 2. Design

기본적으로 Lec5 single-cycle-cpu 교안의 34쪽에 제시된 single-cycle-cpu의 최종 구조를 바탕으로 구현하였다. PC, Instruction memory, Control unit, Register file, ALU, Data memory, Immediate Generator, ALU control unit, Adder 2개와 2:1 Mux 5개, and, or 및 not gate로 구성하였다.

clk은 positive edge를 기준으로 동작하며, is_halted 신호와 통합한 clk_and_ishalted를 사용하였다. clk을 받는 모듈은 PC, Register file, Data memory 뿐이며 나머지는 asynchornous하게 작동한다. 추가적으로 control_unit에서 is_ecall 시그널을 활성화하면 Register file에서 신호를 받아 x17 레지스터의 값을 검사하고, is_halted를 활성화하여 CPU의 작동을 멈추도록 구현하였다. 아래 그림에 코드에서 사용한 모듈 이름을 빨간색 글씨로, wire의 이름은 파란색 글씨로 표기하였다.




Single-Cycle CPU의 5가지 stage는 아래와 같다.

1) IF (Instruction Fetch)

PC 값을 기반으로 instruction을 불러오는 과정이다. PC, instruction memory 모듈을 통해 구현하였다.

2) ID (Instruction Decode)

Instruction을 해석하여 실행할 동작을 결정하는 과정이다. Register file을 통해 구현하였다.

3) EX (Execution)

ID stage에서 해석한 명령어를 수행하는 과정이다. ALU 모듈에서 산술 또는 논리 연산을 하는 과정이 이 단계에 해당된다.

4) MEM (Memory Access)

Store, Load의 경우에만 진행되는 단계로, Data memory에 접근하고 쓰는 과정이다. Control unit에서 결정되는 mem_write, mem_read 신호에 따라 동작한다.

5) WB (Write Back)

연산 결과를 레지스터에 쓰거나 다음 단계로 전달하는 과정이다. ALU와 Data memory의 결과를 이전 모듈로 전달함으로써 구현하였다.

## 3. Implementation

cpu.v의 cpu 모듈을 가장 상위 모듈로 하고 cpu 내부에 각 기능에 맞는 하위 모듈 include, 모듈 간을 연결하는 회로를 설계 및 작성항여서 cpu를 구현해냈다.

### 1) cpu.v

single cycle cpu를 구현한 모듈로, 내부에 여러 모듈들을 include하고 wire로 적절히 연결하며 cpu의 회로를 구현하였다. 위의 design에서 제시한 설계도와 동일한 구조이다.

### 2) pc.v

pc(program counter)의 값을 업데이트하며 cpu가 현재(그리고 다음) 수행할 instruction line의 메모리를 가리키도록 하여 프로세스의 흐름을 통제한다.

- clock synchronous : clk이 들어올 때마다(positive edge) current_pc를 next_pc로 업데이트하여 cpu가 다음 instruction line을 수행할 수 있게 해준다. reset 신호도 들어왔을 경우 next_pc 대신 0으로 pc를 초기화시켜 cpu가 다음 process를 실행할 수 있도록 해준다.

### 3) instruction_memory.v

instruction line을 저장한 magic memory를 지니고 있으며 input으로 들어오는 pc에 따라 메모리에서 pc 주소에 해당하는 instruction을 출력하여 cpu가 해당 instruction을 해석하고 실행할 수 있게 한다.

- clock asynchronous(do not touch 제외, 우리가 구현해야 할 부분) : clk이 들어오는 것과 관계없이 비동기식으로 메모리에서 pc가 가리키는 addr 인덱스(기존의 skeleton code에 따라 addr 중 [11:2]만 참고) 위치에 저장된 32bit 값을 output으로 출력한다.

### 4) register_file.v

32개의 register를 지니고 있으며 input으로 들어오는 인덱스에 따라 register의 값을 출력하거나 입력(이 경우 write 값도 필요)한다. 메모리에서 pc 주소에 해당하는 instruction을 출력하여 cpu가 해당 instruction을 해석하고 실행할 수 있게 한다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 input으로 받은 rs1, rs2의 인덱스 위치에 해당하는 레지스터 값을 output으로 출력한다. 또한 is_halted에 기본적으로 0을 할당하나 is_ecall 신호가 들어왔을 경우 비동기식으

로 1을 할당해준다.

- clock synchronous : clk이 들어올 때마다(positive edge) input으로 받은 rd의 인덱스 위치에 해당하는 레지스터 값을 input으로 받은 rd_din 값으로 변경한다. 단, 이는 is_ecall 신호가 들어왔고 또 해당 레지스터가 인덱스가 0인 x0이 아닐 경우에만 수행한다.

5) control_unit.v

instruction의 타입에 따라 각각의 모듈이 특정 일(write 등)을 수행할 지 여부를 결정하고 mux 등을 통해 cpu.v 회로 상의 흐름을 조절하며 또 instruction에 맞는 input(alu에 input 2로 immediate 값이 들어갈 지 register 값이 들어갈 지 등)이 각 모듈에 들어가도록 도와주는 control 신호들을 설정해준다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 instruction 값의 일부인 opcode를 input으로 받아 그 값에 따라 적절한 control 신호들은 on 되도록, 다른 신호들은 off 되도록 설정해준다. 해당 연산은 opcodes.v에 define되어있는 값들과 Lec5 single-cycle-cpu 교안의 35-36쪽 control values를 참고하여 작성해냈다.

6) immediate_generator.v

instruction에서 이용되는 immediate 값이 instruction format 상에선 부분부분 나누어져 저장되어있다. 이를 instruction 타입을 참고해 완전한 immediate 값으로 합쳐서 output으로 출력해준다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 instruction 값을 input으로 받아 opcode를 추출하고 opcode에 따라 부분부분 저장된 immediate part를 적절하게 합쳐서 완전한 immediate를 만든다. 해당 연산은 opcodes.v에 define되어있는 값들과 Lab2_single_cycle_cpu 교안의 6쪽에 나타나 있는 opcode에 따른 immediate 값의 위치를 참고하여 작성해냈다.

7) alu_control_unit.v

instruction의 타입에 따라 alu.v에서 적절한 arithmetic 연산을 수행하도록 alu의 operation code를 설정해준다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 instruction 값의 일부인 opcode와 funct3, funct7_5를 input으로 받아 그 값에 따라 적절하게 alu의 operation code(alu_op)를 설정해준다. alu_op의 인덱스상 [5:2] 위치는

arithmetic operation의 타입을 정해주고 [1:0] 위치는 instruction이 branch 타입일 경우 BEQ/BNE/BGE/BLT 중 어느 instruction인지 알려주는 역할(btype)이다. 해당 연산은 opcodes.v과 alu_func.v에 define되어있는 값들과 Lab2_single_cycle_cpu 교안의 6쪽에 나타나있는 instruction에 따른 funct3, funct7_5 값을 참고하여 작성해냈다.

8) alu.v

cpu 상에서 arithmetic operation instruction과 그 외 instruction을 위한 arithmetic 연산을 수행한다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 alu_control_unit.v에서 설정한 alu_op를 input으로 받아 일부([5:2])를 function code로 설정, 일부([1:0])를 branch type으로 설정하고 function code에 따라 적절한 arithmetic 연산을 input으로 들어온 두 피연산자 A, B에 수행하여 나온 결과값을 output으로 출력한다. 또한 branch type에 따라 arithmetic 연산의 결과값에 다른 비교 연산자를 써서 조건문을 만들고 조건문을 통과할 경우 branch의 조건을 통과했다는 의미인 alu_bcond 신호를 on으로 해 output으로 출력한다. 여기서 arithmetic 연산은 instruction이 branch type일 경우 자동으로 sub(-)가 되기 때문에  두 피연산자 A, B의 값을 비교한 것과 같게 된다. 해당 모듈은 이전에 lab1 과제로 제출했던 alu 모듈을 기반으로 하였다.

9) data_memory.v

data를 저장한 magic memory를 지니고 있으며 input으로 들어오는 인덱스에 따라 따라 메모리에서 인덱스 주소에 해당하는 값을 출력하거나 입력(이 경우 write 값도 필요)한다. 메모리에서 인덱스 주소에 해당하는 값을 출력하여 레지스터에 저장하는 load instruction을 수행하거나 레지스터 값을 반대로 메모리에 입력, 저장하는 store instruction을 수행하도록 해 RISC의 load&store 특징을 실현시킨다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 input으로 받은 addr의 인덱스(기존의 skeleton code에 따라 addr 중 [15:2]만 참고) 위치에 해당하는 메모리에서의 값을 output으로 출력한다. 단, 이는 mem_read 신호가 들어왔고 또 mem_write 신호는 들어오지 않았을 경우에만 수행하고 이가 아닐 경우 output 값을 0으로 설정한다.
- clock synchronous : clk이 들어올 때마다(positive edge) input으로 addr의 인덱스(기존의 skeleton code에 따라 addr 중 [15:2]만 참고) 위치에 해당하는 메모리에서의 값을 input으로 받은 din 값으로 변경한다. 단, 이는 mem_write 신호가 들

어왔고 또 mem_read 신호는 들어오지 않았을 경우에만 수행한다.

10) add.v, mux.v, and.v, or.v

각각 다음 pc 계산(add) 혹은 cpu.v 회로 상 흐름 조절(mux) 및 signal들을을 and/or 연산으로 합쳐주는 역할 등을 한다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 input으로 받은 두 피연산자(mux의 경우 ctrl 신호도 input으로 필요)를 모듈의 목적에 맞게 연산 혹은 조건식 구성 후 계산이 완료된 result를 output으로 출력한다.

## 4. Discussion

Verilog를 이용해 vending machine을 구현하며 latch, clk의 올바른 사용에 관련된 에러를 주로 겪었다.

Single cycle cpu를 구현하면서, cpu의 작동 원리/규칙을 누락하여 생기는 에러들이 있었다.

1) x0(=0) 변환 금지

x0 레지스터는 ISA 규칙에 의하여 항상 0의 값을 갖는다. 해당 규칙은 알고 있었으나 x0 레지스터에 write를 하지 않을 것이라고 예상하고 register_file.v에서 x0 레지스터에 write를 하려고 하는 경우 막는 조치를 생략하였다. 그 결과로, loop_mem testbench에서 JAL/JALR을 할 때 rd를 x0으로 설정함으로써 원래 돌아와야 할 address인 'PC + 4'가 저장되지 않도록 하는 것이 프로그램의 의도인데 의도와 다르게 x0 레지스터에 값이 저장되어 결과적으로 result가 달라지는 문제가 발생하였다. 이러한 의도와 규칙을 loop_mem testbench를 돌려보면서 알게 되었고 register_file.v에서 rd에 write를 할 때 인덱스가 0이 아닌지(x0이 아닌지) 확인하는 조건을 추가하였다.

2) alu 모듈의 차이

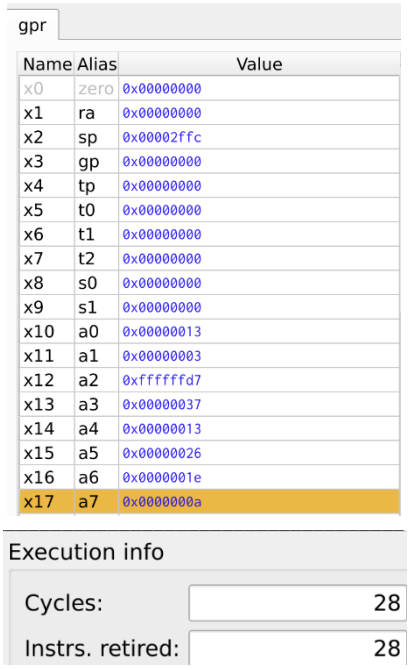
implementation에서 이전에 lab1 과제로 구현했던 alu 모듈을 기반으로 하였다고 했듯이 FuncCode에 따라 다른 arithmetic 연산을 수행하는 부분은 이전에 구현했던 alu 모듈을 거의 그대로 가져왔는데 알고 보니 이전의 alu 모듈에서는 shift 연산을 할 경우 오른쪽 피연산자는 1로 고정되었는데 여기서 구현해야 할 shift 연산은 오른쪽 피연산자에 두번째 피연산자로 받았던 input B를 넣어야 했다. 이러한 다른 점으로 인해 문제가 발생해 result로 나온 레지스터 값이 기댓값과 달랐고, testbench를 돌려보면서 이 점을 알게 되

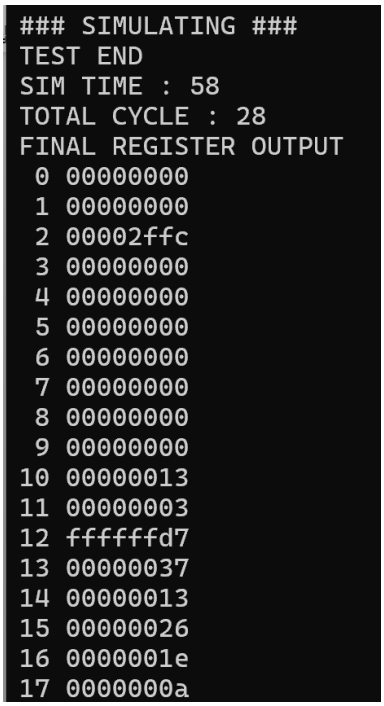
어 이후 shift 연산 부분을 수정하여 문제를 해결했다.

## 5. Conclusion

주어진 asm 코드를 ripes에서 실행한 결과와, 구현한 Single-Cycle CPU를 토대로 테스트벤치를 실행한 결과는 아래와 같다. 테스트벤치의 instruction은 레지스터를 x17까지만 사용하므로 x0~x17까지만 나타내었다.

<basic_mem>




<loop_mem>

| Name | Alias | Value |
|---|---|---|
| x0 | zero | 0x00000000 |
| x1 | ra | 0x00000000 |
| x2 | sp | 0x00002ffc |
| x3 | gp | 0x00000000 |
| x4 | tp | 0x00000000 |
| x5 | t0 | 0x00000000 |
| x6 | t1 | 0x00000000 |
| x7 | t2 | 0x00000000 |
| x8 | s0 | 0x00000000 |
| x9 | s1 | 0x00000000 |
| x10 | a0 | 0x00000000 |
| x11 | a1 | 0x00000000 |
| x12 | a2 | 0x00000000 |
| x13 | a3 | 0x00000000 |
| x14 | a4 | 0x0000000a |
| x15 | a5 | 0x00000009 |
| x16 | a6 | 0x0000005a |
| x17 | a7 | 0x0000000a |

Execution info

| | |
|---|---|
| Cycles: | 222 |
| Instrs. retired: | 222 |

```
### SIMULATING ###
TEST END
SIM TIME : 446
TOTAL CYCLE : 222
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
```

<non-controlflow_mem>

| Name | Alias | Value |
|---|---|---|
| x0 | zero | 0x00000000 |
| x1 | ra | 0x00000000 |
| x2 | sp | 0x00002ffc |
| x3 | gp | 0x00000000 |
| x4 | tp | 0x00000000 |
| x5 | t0 | 0x00000000 |
| x6 | t1 | 0x00000000 |
| x7 | t2 | 0x00000000 |
| x8 | s0 | 0x00000000 |
| x9 | s1 | 0x00000000 |
| x10 | a0 | 0x0000000a |
| x11 | a1 | 0x0000003f |
| x12 | a2 | 0xfffffff1 |
| x13 | a3 | 0x0000002f |
| x14 | a4 | 0x0000000e |
| x15 | a5 | 0x00000021 |
| x16 | a6 | 0x0000000a |
| x17 | a7 | 0x0000000a |

Execution info

| | |
|---|---|
| Cycles: | 39 |
| Instrs. retired: | 39 |

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 0000000a
11 0000003f
12 fffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
```

3개의 테스트벤치에 대해 ripes에서 실행한 결과와 구현한 CPU의 결과에서 cycle 수, 최종 레지스터 값이 모두 일치함을 볼 수 있다. 이로써 Single-Cycle CPU를 성공적으로 구현하였다.

Lab2를 통해 Single-Cycle CPU의 구조와 각 stage의 과정을 이해할 수 있게 되었다. Single-Cycle CPU는 이후에 배운 Multi-Cycle CPU나 Pipelined CPU에 비해 효율성이 떨어지지만, 직접 구현하는 과정을 통해 Verilog를 활용한 Processor 설계에 익숙해질 수 있었다.