CSED311: Lab 2 (due Mar. 26)

김태연(20220140), 손량(20220323)

Last compiled on: Sunday 24th March, 2024, 18:30

1 Introduction

RISC-V architecture를 바탕으로 구성된 Single cycle CPU가 작동하는 방식을 알아보고, 각 Instruction type에 따라 거치는 Datapath에 유의하며 CPU를 구성하는 각 module을 설계하고 연결하여 Single cycle CPU를 구현한다.

2 Design

이번에 구현한 RISC-V Single cycle CPU는 다음과 같은 주요 기능을 가진 Submodule로 나누었으며, 해당 값들을 생성하고 선택하기 위해 추가적으로 Adder와 Multiplexer를 구현하여 사용했다.

2.1 Program Counter

PC, 즉 프로그램 카운터는 주어진 Clock signal에 synchronous하게 신호에 따라 다음 PC address의 값을 현재 PC address에 대입한다. 출력된 PC address는 CPU 내에서 구현된 분기(`Bxx`), 점프(`JAL, JALR`) 등의 Instruction에 따라 (현재 PC + 4) 혹은 (임의의 Instruction address)로 값이 변화하게 된다.

2.2 ALU

Register에 저장되어있는 32bit integer 또는 Immediate generator을 통해 Instruction으로부터 생성된 32bit integer의 값을 읽어들인 후 정의된 연산의 ID에 따라 연산을 진행하며, `always @(*)`를 사용하여 주어진 opcode에 해당하는 연산 결과를 CPU clock cycle과 asynchronous하게 `alu_result`로 반환했다. Textbook에 등장하는 `bcond`를 32bit 연산 결과의 LSB에 할당하여, 논리 연산의 경우 LSB 1bit만 사용하여 연산을 진행한다.

2.3 Register file

RV32I Architecture의 32bit Register 32개를 구현하였다. 레지스터의 값을 작성하는 것과 읽어오는 것은 서로 다르게 설계하였다.

- Write – `posedge clk` 에 따라 Clock signal에 synchronous하게 레지스터를 수정한다.
- Read – `always @` 문 밖에서 Clock signal에 asynchronous하게 레지스터의 값을 입력으로 들어온 `rs1` 및 `rs2`에 해당하는 레지스터를 참조하여 출력하도록 연결을 구성했다.

2.4 Data memory

주어진 "Magic memory"는 32bit integer `MEM_DEPTH = 16384`개로 구성되어있는 64KiB 메모리이다. 메모리에 값을 작성하는 것과 읽어오는 것은 레지스터와 마찬가지의 로직으로 설계하였다.

- Write – `posedge clk` 에 맞추어 `mem_write` 신호가 주어진 경우에만 synchronous하게 memory file을 수정한다.
- Read – `always @` 문 밖에서 Clock signal과 asynchronous하게, `dmem_addr`로 주어진 메모리 주소에 `mem_read` signal이 참인 경우에 `dout`에 assign하였다.

2.5 Instruction memory

Instruction memory는 32bit instruction `MEM_DEPTH = 1024`개로 구성되어있는 4KiB 메모리이다. Testbench (혹은 RV32I로 컴파일된 프로그램)의 각 bytecode를 읽어서 Lab 2에서 구현한 Single-cycle CPU가 실행할 수 있도록 저장한다. 32bit PC를 입력받아, 실제로 구현된 1024개 instruction을 저장할 수 있는 LSB를 제외한 하위 10bit에 해당하는 instruction index로 변환하여 해당 index의 instruction을 Clock 신호와 asynchronous하게 `dout` 으로 반환한다.

2.6 Control unit

어떤 Instruction type이던, LSB 7bit는 RV32I에서 opcode로 정의가 되어 있는 부분이다. 구현한 Control unit은 Clock에 asynchronous하게 Instruction의 LSB 7bit를 입력받아 해당 opcode에 대응되는 CPU control signal을 생성한다.

2.7 Immediate generator

RV32I instruction 중 immediate value를 가지는 instruction의 타입이 존재하며, 각 타입별 Immediate value가 생성되는 부분들 또한 다르다. 본 Single cycle CPU에서는 32bit instruction을 입력받고, 주어진 ISA에서 구현해야할 type에 해당하는 I-type, SB-type, UJ-type, S-type의 Instruction의 Immediate generation에 해당하는 output으로 타입을 구분하여 Clock signal과 asynchronous하게 Immediate value를 출력하였다.

2.8 Ecall unit

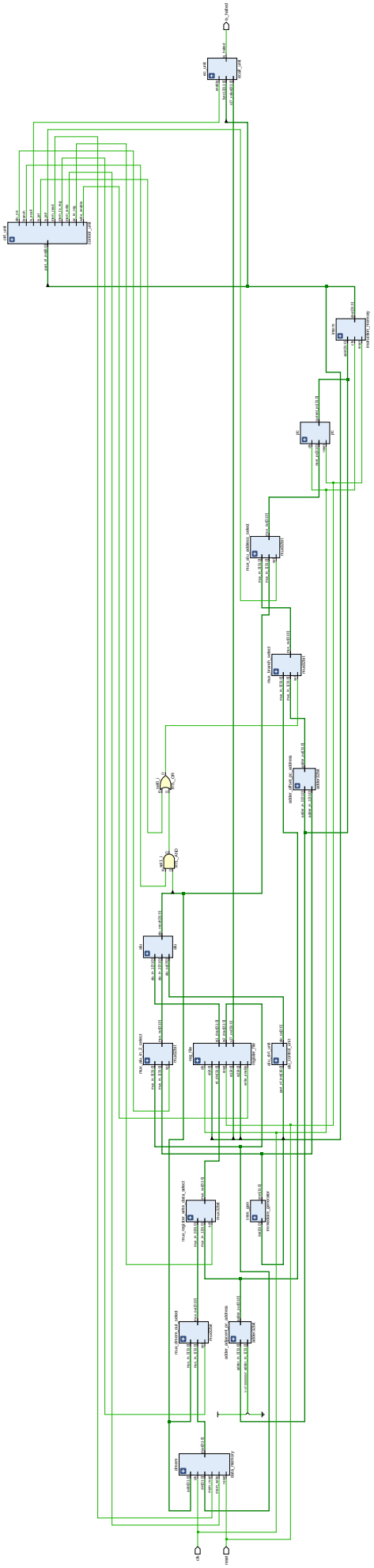
프로그램의 종료를 판별하기 위한 Ecall의 구조를 정의하였으며, Clock signal과 asynchronous하게 수행된다.

3 Implementation

각 베릴로그 파일에 대한 설명은 다음과 같다.

3.1 `cpu.v` – 내부적인 Module을 연결하여 CPU 구성

하위 모듈들을 모두 조합하여 Single cycle CPU의 전체적인 Schematic을 구성하였다. CPU의 전체적인 구성을 출력하기 위해 Vivado Xilinix를 통해 만들어진 Schematic을 첨부하였다.



`cpu.v`는 다음과 같은 submodule을 연결하여 구성했다.

- `pc` – Program counter를 구현하였다.
- `imem` – Instruction memory를 구현하였다.
- `reg_file` – Register를 구현하였다.
- `ctrl_unit` – CPU Control unit을 구현하였다.
- `ec_unit` – `ecall` 을 처리하기 위한 unit을 구현하였다.
- `imm_gen` – 각 Instruction type으로부터 immediate value를 획득한다.
- `alu_ctrl_unit` – Instruction으로부터 ALU의 opcode를 획득한다.
- `alu` – ALU를 구현하였다.
- `dmem` – Data memory를 구현하였다.
- `mux_register_write_data_select` – Multiplexer로, (PC + 4) 혹은 Data memory의 출력을 선택하여 register file에 작성한다.
- `mux_alu_in_2_select` – Multiplexer로, ALU에 들어갈 입력으로 immediate generator에서 생성한 immediate value 혹은 register의 값을 선택한다.
- `mux_dmem_out_select` – Multiplexer로, Write Back 단계에 사용할 출력으로 ALU의 출력 혹은 Data memory의 출력을 선택한다.
- `adder_adjacent_pc_address` – 인접한 다음 PC (PC + 4)를 계산하기 위해 존재하는 32bit adder이다. 두 번째 입력은 4로 고정되어있다.
- `adder_offset_pc_address` – 현재 PC의 주소에서 immediate generator에서 생성한 immediate value만큼 떨어진 address를 계산하는 32bit adder이다.
- `mux_branch_select` – Branch conditon과 Branch instruction인지 판단하여 다음 PC address를 결정하는 Multiplexer이다.
- `mux_alu_address_select` – `JAL` 및 `Bxx`, `JALR` operation에 따라 다음 PC address를 결정하는 Multiplexer이다.

3.2 `cpu_def.v` – CPU control unit을 위한 상수 정의

`cpu.v`의 Contol unit 디자인 부분에 있어서, Contol line을 배열로 생성하여 사용하였다. 해당 배열의 index와 control line의 각 signal을 대응시키기 위한 상수를 정의했다.

3.3 `opcodes.v` – CPU instruction set에 대응되는 상수 정의

Instruction의 `opcode` 부분을 판단하여 operation을 특정하기 위한 상수이다. 또한, R-type 혹은 I-type instruction의 경우 `func3` 및 `func7` 부분이 존재하며, 이 경우 해당 부분에 해당하는 operation을 정의하여 사용하였다.

3.4 `control_unit.v` – Instruction으로부터 CPU control signal 생성

`opcode.v`를 통해 instruction에 부여된 타입에 따라 해당 instruction에 의해 생성되는 signal을 출력한다. 다음과 같은 출력이 존재한다.

- `is_jal` – 해당 instruction이 `JAL`일 경우이다.
- `is_jalr` – 해당 instruction이 `JALR`일 경우이다.
- `branch` – 해당 instruction이 Branch instruction (`Bxx`)일 경우이다.
- `mem_read` – 해당 instruction이 `lw`와 같이 메모리로부터 값을 읽는 경우이다.
- `mem_to_reg` – 해당 instruction이 `lw`와 같이 읽은 메모리의 값을 레지스터로 옮기는 경우이다.

  `mem_read`는 Data memory unit에 read를 선택하기 위한 신호이며, `mem_to_reg`는 레지스터의 입력 Source를 결정하기 위한 Multiplexer의 선택 signal이다.
- `mem_write` – 해당 instruction이 `sw`와 같이 메모리에 값을 쓰는 경우이다.
- `alu_src` – 해당 instruction이 ALU를 사용할 때 immediate generator로부터 생성되는 입력을 사용하는 경우이다.
- `write_enable` – 해당 instruction이 레지스터에 값을 작성하는 경우이다.
- `pc_to_reg` – `JAL` 및 `JALR` instruction에서 기존 PC를 register에 옮기기 위한 신호이다.
- `is_ecall` – 프로그램의 종료를 위한 `ecall`인 경우이다.

3.5 `register_file.v` – Register file 구현

- Hard-wired zero Register인 `rf[0]` (`$x0`) 에 작성 시 값이 변조되지 않도록 추가적인 조건을 추가하였다.
- `$x17` Register는 `ecall` 처리 후 `is_halted`을 설정하는데 사용되게 되며, 이를 위해 추가적으로 해당 `$x17` 레지스터만을 출력하는 wire를 가지고 있다.

3.6 `instruction_memory.v` – Instruction memory 구현

Instruction memory를 구현한 모듈이다. 이 lab에서는 instruction memory가 read only라고 가정했기 때문에, `addr` 핀에 주소를 입력하면 `dout` 핀으 로 32비트 명령어를 출력하도록 하였다.

3.7 `data_memory.v` – Data memory 구현

Data memory를 구현한 모듈이다. 이 lab에서는 data memory는 write도 가능하기 때문에, `mem_read`, `mem_write` 핀을 따로 두어 read와 write를 제어할 수 있도록 구현하였다. `mem_read`가 1일 때에는 `dout` 핀으로 주어 진 주소에서 word를 읽어 출력하고, `mem_write`가 1일 때에는 `din` 핀으로 주어진 주소에 word를 쓴다.

3.8 `alu.v` – ALU 동작 구현

연산을 선택하기 위한 `alu_op`에 따라 두 개의 32bit 입력에 해당하는 연산 결과를 `alu_result`에 반환한다. 산술 연산의 경우 32bit를 모두 사용하여 결과 값을 전달하였으며, 논리 연산의 경우 LSB 1bit에 해당 논리 연산의 결과를 반환하였다.

3.9 `alu_def.v` – ALU 구성을 위한 상수 정의

`alu` 모듈의 `alu_op`에 해당하는 입력이다. 해당 파일에서는 ALU의 `alu_op` 에 들어갈 operation에 들어갈 상수를 정의하였으며, 다음과 같다.

Opcode	Definition	Description
`4'b0000`	`ALU_ADD`	Add
`4'b0001`	`ALU_SUB`	Subtract
`4'b0010`	`ALU_SLL`	Logical shift left
`4'b0011`	`ALU_SLT`	Signed less than
`4'b0100`	`ALU_SLTU`	Unsigned less than
`4'b0101`	`ALU_XOR`	Bitwise XOR
`4'b0110`	`ALU_SRL`	Logical shift right
`4'b0111`	`ALU_SRA`	Arithmetic shift right
`4'b1000`	`ALU_OR`	Bitwise OR
`4'b1001`	`ALU_AND`	Bitwise AND
`4'b1010`	`ALU_EQ`	Equal
`4'b1011`	`ALU_NE`	Not equal
`4'b1011`	`ALU_GE`	Signed greater than or equal
`4'b1011`	`ALU_GEU`	Unsigned greater than or equal
`4'b1111`	`ALU_ERR`	Error signal (For debug)

3.10 `alu_control_unit.v` – Instruction과 ALU opcode 대응

입력받은 Instruction에서 요하는 ALU의 기능에 해당하는 ALU opcode를 반환하기 위한 ALU Control Unit이며, 각 Instruction 타입에 따라 ALU에 입력되는 `opcode`가 달라지도록 설계했다.

- Arithmetic instruction – 각 Instruction에 해당하는 Arithmetic operation으로 `alu_op`를 결정한다.
- Branch – Branch condition에 해당하는 논리 연산을 진행하도록 `alu_op`를 결정한다.
- Load / Store – `ALU_ADD`를 사용하여 주어진 Offset으로
- 이외의 경우 – `ALU_ERR`을 사용해 Debugging을 진행하였다.

3.11 `immediate_generator.v` – Instruction type 별 immediate value 생성

Instruction이 `inst`로 주어질 때 opcode, func3 값을 통해 immediate value 를 생성해 `imm` 핀으로 출력하도록 구현하였다.

3.12 `pc.v` – Program Counter 동작 구현

PC 값을 저장하고 관리하기 위한 모듈이다. 이 모듈은 각 positive clock edge마다 `next_pc` 핀으로 다음 PC 값을 받아 `current_pc` 레지스터를 업데 이트해준다.

3.13 `adder32bit.v` – 내부적인 Adder 동작 구현

32bit integer 2개를 입력받아 더한 값을 반환한다. 다음 PC를 계산하기 위해 사용된다.

3.14 `mux32bit.v` – 내부적인 Multiplexer 동작 구현

32bit integer 2개 중 하나의 값을 `sel` 신호에 따라 선택하여 반환하는 Multiplexer이다.

3.15 `top.v` – Top module

`cpu.v`에서 구현한 `cpu` 모듈을 사용하여 최종적으로 Single cycle CPU를 구현하였다.

4 Discussion

이로써 RV32I base instruction set의 대부분을 구현한 single-cycle CPU를 완성할 수 있었다. 이 single-cycle CPU는 single cycle 안에 instruction/data fetch가 일어난 다는 비현실적인 가정 하에서 구현되었는데, DDR3 AXI interconnect 등을 구현해 실제 FPGA 상에서 CPU를 구동해 보는 것도 좋을 것이라 생각한다.

5 Conclusion

주어진 testbench 파일들이 테스트를 통과함을 확인할 수 있었다.
`basic_mem.txt`에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 58
TOTAL CYCLE : 28 (Answer : 28)
FINAL REGISTER OUTPUT
 0 00000000
 1 00000000
 2 00002ffc
 3 00000000
 4 00000000
 5 00000000
 6 00000000
 7 00000000
 8 00000000
 9 00000000
10 00000013
11 00000003
12 fffffd7
13 00000037
14 00000013
15 00000026
16 0000001e
17 0000000a
18 00000000
19 00000000
```

```
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

loop_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 446
TOTAL CYCLE : 222 (Answer : 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
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
```

```
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

non-controlflow_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 80
TOTAL CYCLE : 39 (Answer : 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
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```