2024학년도 컴퓨터구조 Lab Assignment #2
Single Cycle CPU

김도영, 선민수

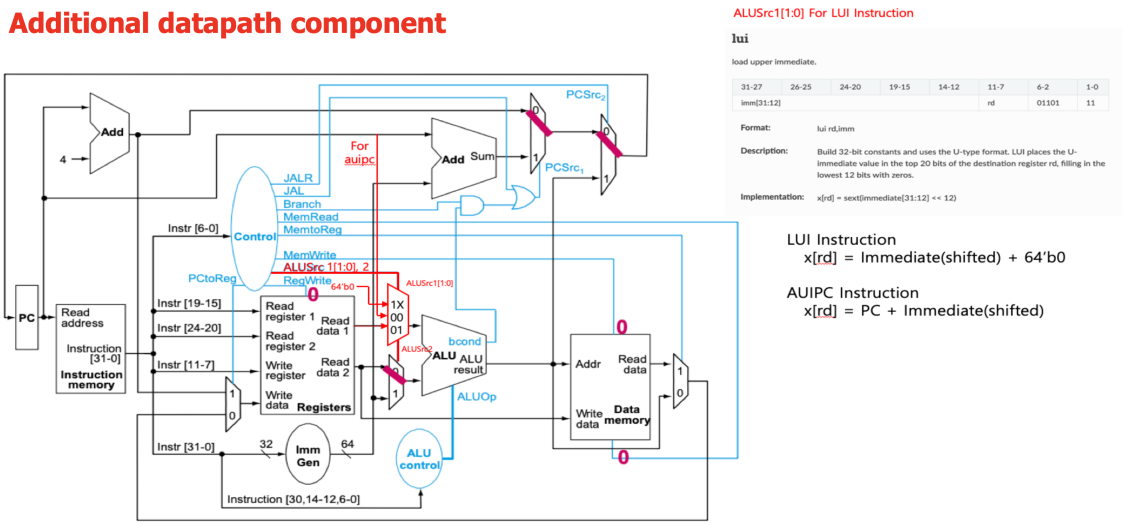
2024년 3월 25일

1 Introduction

이 과제에서는 Verilog를 이용하여 Single Cycle CPU를 구현하는 것을 목적으로 한다.

2 Design

본 과제에서 구현한 Single Cycle CPU는 수업 시간에 배운 Data Path를 기준으로 구현하였으며, 추가적인 LUI, AUIPC Instruction을 위한 Data Path 또한 구현하였다.



Design of Single-Cycle CPU

Single Cycle CPU를 구성하는 세부 모듈들과 각각의 역할은 아래와 같다.

- ALU Control Unit: Instruction에서 ALU가 필요한 OPCode, Funct3, Funct7을 추출하여 ALU에 alu_op로 전달한다.
- ALU: alu_op을 이용하여 현재 필요한 연산을 결정하고, 이에 대한 연산을 입력을 이용해 결과를 도출한다. BRANCH Instruction의 경우 bcond 출력을 이용해 BRANCH taken/not-taken을 출력한다.

- Control Unit: 주어진 Instruction을 이용하여 종류를 판단하고 필요한 Control Value를 출력한다.
- Data Memory: 프로그램에 필요한 데이터가 저장되는 메인 메모리로, 주어진 주소에 대한 데이터를 출력하거나, mem_write 신호에 따라 메모리에 데이터를 넣기도 한다. 이번 과제에서는 주어진 지시에 따라 Synchronous Data Write, Asynchronous Data Read의 메모리를 구현한다.
- Immediate Generator: Immediate가 사용되는 Instruction의 종류에 따라서 Immediate를 추출하여 imm_gen_out으로 출력한다.
- Instruction Memory: Instruction을 CPU에 불러오는 Memory이다. Data Memory와 달리 Instruction만이 저장되어 있으며, 새로운 데이터가 작성되는 일은 없다. Data Memory와 같이 주어진 주소에 대한 Instruction을 출력한다. 이번 과제에서는 주어진 지시에 따라 Asynchronous Instruction Read의 메모리를 구현한다.
- Next PC: Program Counter의 다음 값을 결정하는 모듈로, 위의 디자인 그림에서는 따로 구분되어 있지 않지만, Program Counter의 다음 값을 결정하는 Data Path를 모듈화한 모듈이다. JAL, JALR, BRANCH Instruction인지 판단하여 ALU의 결과값과 Immediate 값을 이용해 다음 Program Counter의 값을 결정한다.
- Program Counter: 주어진 next_pc 값으로 Program Counter를 변화시킨다.
- Register File: 주어진 Register 번호에 대한 Register의 데이터를 출력한다. write_enable 신호에 의해 대상 register에 값을 쓰기도 한다.
- CPU: 위의 모듈들을 관리하는 top module로, 전체 모듈들을 연결하며, 과제의 조건에 따라 instruction이 ecall인 경우 10번 레지스터의 값이 10이 될 때, 종료 신호를 출력한다.

Single Cycle CPU는 아래와 같은 단계를 통해 Instruction을 수행하게 된다.

- Instruction Fetch Stage: 이전 Instruction에 의해 결정된 next_pc를 이용하여 current_pc를 업데이트한 후 이를 통해서 Instruction Memory의 Instruction을 패치한다. Program Counter와 Next Program Counter Logic이 주축이 되어 해당 스테이지를 처리한다.
- Instruction Decode Stage: Instruction Fetch를 통해 얻은 Instruction을 필요한 데이터로 가공 및 추출한다. 동시에 Register File에서 필요한 레지스터를 읽어온다. Control Unit, ALU Control Unit, Register File을 주축으로 하여 해당 스테이지를 처리한다.
- Execution Stage: 주어진 Instruction과 추출된 데이터를 기반으로 연산을 진행한다. 동시에 BRANCH Instruction의 경우에는 bcond 신호를 결정한다. ALU가 해당 스테이지를 처리한다.
- Access Memory Stage: ALU에서 추출된 결과를 통해서 필요한 메모리의 주소를 계산한 후, Data Memory에서 해당 주소의 데이터를 읽는다. Instruction에 따라서 Write가 필요한 경우에는 mem_write를 통해 Memory에 데이터를 쓰도록 한다. Memory가 해당 스테이지를 처리한다.
- Write Back Stage: 연산 결과 혹은 메모리에서 읽은 데이터를 레지스터에 다시 쓰는 스테이지이다. Register File의 rd, din을 이용하여 write_enable에 따라 레지스터에 데이터를 쓰게 된다. Register File이 해당 스테이지를 관리한다.

3 Implementation

3.1 Program Counter

```
always @(posedge clk) begin
    if (reset) begin
        current_pc <= 0;
    end else begin
        current_pc <= next_pc;
    end
end
```

pc.v

3.2 Next Program Counter Logic

```
always @(*) begin
    if(is_jalr) begin
        next_pc = alu_result;
    end else begin
        if(branch && alu_bcond || is_jal)
            next_pc = current_pc + immediate;
        else
            next_pc = current_pc + 4;
    end
end
```

next_pc_logic.v

3.3 Control Unit

```
always @(*) begin
    is_jal = (opcode == `JAL);
    is_jalr = (opcode == `JALR);
    branch = (opcode == `BRANCH);
    mem_read = (opcode == `LOAD);
    mem_to_reg = (opcode == `LOAD);
    mem_write = (opcode == `STORE);
    alu_src_1[1] = (opcode == `LUI);
    alu_src_1[0] = (opcode == `AUIPC);
    alu_src_2 = ((opcode != `ARITHMETIC) && (opcode != `BRANCH));
    write_enable = ((opcode != `STORE) && (opcode != `BRANCH));
    pc_to_reg = (opcode == `JALR || opcode == `JAL);
    is_ecall = (opcode == `ECALL);
end
```

control_unit.v

3.4 ALU Control Unit

```
always @(*) begin
    alu_op = {funct7, funct3, opcode};
end
```

alu_ctrl_unit.v

3.5 Immediate Generator

```
always @(*) begin
    case(opcode)
    `ARITHMETIC_IMM: imm_gen_out = {{20{inst[31]}}, inst[31:20]};
    `LOAD          : imm_gen_out = {{20{inst[31]}}, inst[31:20]};
    `STORE         : imm_gen_out = {{20{inst[31]}}, inst[31:25], inst[11:7]};
    `BRANCH        : imm_gen_out = {{19{inst[31]}}, inst[31], inst[7], inst[30:25], inst[11:8], 1'b0};
    `JAL           : imm_gen_out = {{11{inst[31]}}, inst[31], inst[19:12], inst[20], inst[30:21], 1'b0};
    `LUI           : imm_gen_out = {inst[31:12], 12'b0};
    `AUIPC         : imm_gen_out = {inst[31:12], 12'b0};
    default        : imm_gen_out = {32{1'b0}};
    endcase
end
```

imm_gen.v

3.6 Register File

```
// Asynchronously read register file
always @(*) begin
    rs1_dout = (rs1 == 5'b0) ? 32'b0 : rf[rs1];
    rs2_dout = (rs2 == 5'b0) ? 32'b0 : rf[rs2];
end

// Synchronously write data to the register file
always @(posedge clk) begin
    if (write_enable)
        rf[rd] <= rd_din;
end
```

register_file.v

3.7 ALU

```
always @(*) begin
    case(opcode)
    `ARITHMETIC: begin
        alu_bcond = 1'b0;
        if(funct7 != `FUNCT7_SUB) begin
            case(funct3)
            `FUNCT3_ADD: alu_result = alu_in_1 + alu_in_2;
            `FUNCT3_SLL: alu_result = alu_in_1 << alu_in_2;
            `FUNCT3_XOR: alu_result = alu_in_1 ^ alu_in_2;
            `FUNCT3_SRL: alu_result = alu_in_1 >> alu_in_2;
            `FUNCT3_OR : alu_result = alu_in_1 | alu_in_2;
            `FUNCT3_AND: alu_result = alu_in_1 & alu_in_2;
            default    : alu_result = 32'b0;
            endcase
        end else begin
            alu_result = alu_in_1 - alu_in_2;
        end
    end

    `ARITHMETIC_IMM: begin
        alu_bcond = 1'b0;
        case(funct3)
        `FUNCT3_ADD: alu_result = alu_in_1 + alu_in_2;
        `FUNCT3_SLL: alu_result = alu_in_1 << alu_in_2;
        `FUNCT3_XOR: alu_result = alu_in_1 ^ alu_in_2;
        `FUNCT3_OR : alu_result = alu_in_1 | alu_in_2;
        `FUNCT3_AND: alu_result = alu_in_1 & alu_in_2;
        `FUNCT3_SRL: alu_result = alu_in_1 >> alu_in_2;
        default    : alu_result = 32'b0;
        endcase
    end
```

alu.v

```
    `LOAD: begin
        alu_bcond = 1'b0;
        alu_result = alu_in_1 + alu_in_2;
    end

    `JALR: begin
        alu_bcond = 1'b0;
        alu_result = alu_in_1 + alu_in_2;
    end

    `STORE: begin
        alu_bcond = 1'b0;
        alu_result = alu_in_1 + alu_in_2;
    end

    `BRANCH: begin
        alu_result = 32'b0;
        case(funct3)
        `FUNCT3_BEQ: alu_bcond = (alu_in_1 == alu_in_2);
        `FUNCT3_BNE: alu_bcond = (alu_in_1 != alu_in_2);
        `FUNCT3_BLT: alu_bcond = (alu_in_1 < alu_in_2);
        `FUNCT3_BGE: alu_bcond = (alu_in_1 >= alu_in_2);
        default    : alu_bcond = 1'b0;
        endcase
    end
```

alu.v

```
    `JAL: begin
        alu_result = 32'b0;
        alu_bcond = 1'b0;
    end

    `LUI: begin
        alu_result = alu_in_1 + alu_in_2;
        alu_bcond = 1'b0;
    end

    `AUIPC: begin
        alu_result = alu_in_1 + alu_in_2;
        alu_bcond = 1'b0;
    end

    default: begin
        alu_result = 32'b0;
        alu_bcond = 1'b0;
    end
    endcase
end
```

alu.v

3.8 Instruction/Data Memory

```
// Asynchronously read instruction from the memory
// (use imem_addr to access memory)
always @(*) begin
    dout = mem[imem_addr];
end
```

instruction_memory.v

```
// Asynchrnously read data from the memory
// Synchronously write data to the memory
// (use dmem_addr to access memory)
always @(*) begin
if(mem_read)
    dout = mem[dmem_addr];
else
    dout = 32'b0;
end
always @(posedge clk) begin
    if(mem_write) begin
        mem[dmem_addr] <= din;
    end
end
```

data_memory.v

3.9 CPU

```
always @(*) begin
if(pc_to_reg)
  rd_din = current_pc + 4;
else begin
  if(mem_to_reg)
    rd_din = data;
  else
    rd_din = alu_result;
end
end

always @(*) begin
if(alu_src_2)
  alu_in_2 = immediate;
else
  alu_in_2 = rs2_dout;
end

always @(*) begin
if(alu_src_1[1])
  alu_in_1 = 32'b0;
else if(alu_src_1[0])
  alu_in_1 = current_pc;
else
  alu_in_1 = rs1_dout;
end

always @(*) begin
if(is_ecall)
  rs1 = 5'b10001;
else
  rs1 = inst[19:15];
end

always @(*) begin
if(is_ecall && (rs1_dout == 10))
  is_halted = 1;
else
  is_halted = 0;
end

/* Omit Module Declarations ... */
```

cpu.v

4 Discussion

이번 과제에서 구현한 Single Cycle CPU의 세부 모듈들과 각각의 모듈의 Clock Synchronous/Asynchronous 여부는 아래와 같다.

- ALU Control Unit: Asynchronous
- ALU: Asynchronous
- Control Unit: Asynchronous
- Data Memory(Read Data): Asynchronous
- Data Memory(Write Data, Initializing, Reset): Synchronous
- Immediate Generator: Asynchronous
- Instruction Memory(Read Instruction): Asynchronous

- Instruction Memory(Initializing, Reset): Synchronous
- Next PC: Asynchronous
- Program Counter: Synchronous
- Register File(Register Read): Asynchronous
- Register File(Register Write, Initializing, Reset): Synchronous

5 Conclusion

이번 과제에서는 모든 Instruction이 1 Cycle에 완료되는 Single Cycle CPU를 구현하였다. Instruction을 기반으로 CPU의 동작을 제어할 수 있는 다양한 Control Flag을 생성하여 CPU의 Data Path를 조정할 수 있도록 하여 1 Cycle 내에 Instruction을 수행할 수 있도록 설계하였다.

이번 Single Cycle CPU를 구현하면서, 1 Cycle을 온전히 1개의 Instruction을 수행하는 데 사용할 수 있지만, 1 Cycle의 시간 동안 다른 세부 모듈들이 낭비되고 있음을 알 수 있었고, 이를 개선한 Multi-Cycle CPU의 필요성을 느낄 수 있었다.