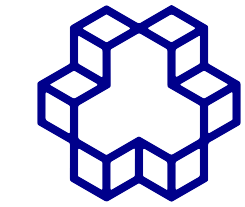


CPU Design Project Report

Fatemeh Amirabadi

Prof. Atena Abdi

HOMEWORK N°

Introduction

This report presents the design and implementation of a simple CPU using Verilog. The CPU is capable of executing basic arithmetic and logic operations, loading and storing data from/to memory, and fetching instructions from an instruction memory.
Size of each part:
Momory size: 128 byte Instruction memory and data memory: 128 bytes

Opcod size: 8 bit

Bus size (based on bus-output): 8-bit
the bus can transfer 8-bit data between the CPU registers and memory modules

Size of each cpu register:
Alu-op1 : 8 bit
alu-op2:8 bit
Pc :4bit
AR :4bit
TR:8bit
AC:8bit

DR:8bit
IR:8bit
Data :8bit
address: 4bit

Modules and Their Functions

The CPU design is divided into several modules, each responsible for a specific function. The key modules are:

- **cpu**: The main module that coordinates the operation of other modules.
- **instruction_memory**: Stores and fetches instructions.
- **data_memory**: Stores and fetches data.
- **alu**: Performs arithmetic and logic operations.
- **counter**: Generates sequential states.
- **decoder**: Decodes the instruction into control signals.

CPU Module

The `cpu` module coordinates the activities of the other modules based on the clock and the current state. It fetches instructions, decodes them, fetches operands, executes the instruction, and writes back the results. The sequence of operations is controlled by the `seq_out` signal from the `counter` module.

```
module cpu(init_ins, init_data, clk);
    input [127:0] init_ins;
    input [127:0] init_data;
    input wire clk;

    wire [2:0] seq_out;
    wire [7:0] decoder_out;

    reg enable;
    reg imem_enable, dmem_enable, alu_enable, decoder_enable;
    reg seq_reset, initial_load, memory_write;
    reg [2:0] alu_mode, decoder_in;
    reg [7:0] IR, DR, AC, TR, alu_op1, alu_op2;
    wire [7:0] alu_out, imem_out, dmem_out;
    reg [3:0] AR, PC;
```

```
16      wire signed [7:0] bus_output;
17      reg [3:0] bus_selector;
18
19      instruction_memory ins_memory(imem_enable, imem_out, AR, initial_load, init_ins, c
20      data_memory data_memory(dmem_enable, dmem_out, AR, TR, memory_write, initial_load,
21      counter sequencer(seq_out, clk, seq_reset);
22      decoder decoder(decoder_enable, decoder_out, decoder_in);
23      alu alu(alu_out, alu_enable, alu_mode, alu_op1, alu_op2);
24      bus bus(bus_output, PC, AC, IR, DR, TR, AR, imem_out, dmem_out, bus_selector);
25
26      initial begin
27          enable = 1;
28          initial_load = 1;
29          imem_enable = 0;
30          dmem_enable = 0;
31          alu_enable = 0;
32          decoder_enable = 0;
33          seq_reset = 1;
34          PC = 0;
35          AC = 0;
36      end
37
38      always @(seq_out) begin
39          if (enable == 1) begin
40              initial_load = 0;
41              case(seq_out)
42                  3'b000: // ins fetch
43                  begin
44                      seq_reset = 0;
45                      bus_selector = 4'b0111; // Select PC
46                      AR = bus_output;
47                      imem_enable = 1;
48                  end
49                  3'b001: // decode
50                  begin
51                      imem_enable = 0;
52                      IR = imem_out;
53                      decoder_enable = 1;
54                      decoder_in = IR[6:4];
55                  end
56                  3'b010: // op fetch
57                  begin
58                      if (IR[7] == 1)
59                          enable = 0;
60                      AR = IR[3:0];
61                      #2
62                      dmem_enable = 1;
63                      bus_selector = 4'b0110; //data memory
64                      memory_write = 0;
65                      decoder_enable = 0;
66
67                  end
68                  3'b011: // execute
69                  begin
70
```

```
                        dmem_enable = 0;
                        DR = dmem_out;
                        if (decoder_out[0]) // Add
                        begin
                            alu_enable = 1;
                            bus_selector = 4'b0001; // Select AC
                            #3
                            AC = bus_output;
                            alu_op1 = DR;

                            bus_selector = 4'b0011; // Select DR
                            alu_op2 = bus_output;
                            alu_mode = 3'b000;
                        end
                        else if (decoder_out[3]) // Load
                        begin
                            alu_enable = 1;
                            bus_selector = 4'b0011; // Select DR
                            alu_op1 = bus_output;
                            alu_op2 = bus_output;
                            alu_mode = 3'b000;
                        end
                        else if (decoder_out[4]) // Arithmetic shift left
                        begin
                            alu_enable = 1;
                            bus_selector = 4'b0011; // Select DR
                            alu_op1 = bus_output;
                            alu_op2 = bus_output;
                            alu_mode = 3'b011;
                        end
                        else if (decoder_out[5]) // Store
                        begin
                            bus_selector = 4'b0001; // Select AC
                            TR = bus_output;
                        end
                        else if (decoder_out[7]) // Root
                        begin
                            alu_enable = 1;
                            bus_selector = 4'b0011; // Select DR
                            alu_op1 = bus_output;
                            alu_op2 = bus_output;
                            alu_mode = 3'b111;
                        end
                        else if (decoder_out[6]) // XNOR
                        begin
                            alu_enable = 1;
                            bus_selector = 4'b0001; // Select AC
                            alu_op1 = bus_output;

bus_selector = 4'b0011; // Select DR
                            alu_op2 = bus_output;
                            alu_mode = 3'b100;
                        end
                        else if (decoder_out[2]) // Arithmetic shift right
                        begin
```

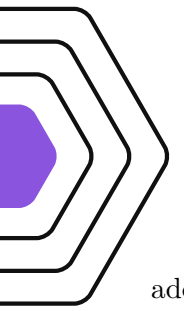
```
                        alu_enable = 1;
                        bus_selector = 4'b0011; // Select DR
                        alu_op1 = bus_output;
                        alu_op2 = bus_output;
                        alu_mode = 3'b101;
                    end
                    else if (decoder_out[1]) // 2's complement
                    begin
                        alu_enable = 1;
                        bus_selector = 4'b0011; // Select DR
                        alu_op1 = bus_output;
                        alu_op2 = bus_output;
                        alu_mode = 3'b110;
                    end
                end
                3'b100: // writeback
                begin
                    alu_enable = 0;
                    AC = alu_out;
                    if (~decoder_out[5]) begin
                        bus_selector = 4'b0011; // Select DR
                        TR = bus_output;
                    end
                    memory_write = 1;
                    dmem_enable = 1;
                end
                3'b101: // program counter
                begin
                    memory_write = 0;
                    dmem_enable = 0;
                    PC = PC + 1;
                end
                3'b110: // reset
                begin
                    seq_reset = 1;
                end
            endcase
        end
    end
endmodule
```

Bus Module

The `instruction_memory` module stores a set of instructions and outputs the instruction at the address specified by the `AR` register.

```
module bus(bus_out, PC, AC, IR, DR, TR, AR, ins_mem, data_mem, bus_selector);
input [7:0] PC, AC, IR, DR, TR, AR, ins_mem, data_mem;
input [3:0] bus_selector;
output reg signed [7:0] bus_out;
```

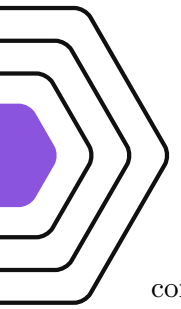
```
5
6  always @(*) begin
7       case (bus_selector)
8           4'b0000: bus_out = AR;
9           4'b0001: bus_out = AC;
10          4'b0010: bus_out = IR;
11          4'b0011: bus_out = DR;
12          4'b0100: bus_out = TR;
13          4'b0101: bus_out = ins_mem;
14          4'b0110: bus_out = data_mem;
15          4'b0111: bus_out = PC;
16          default: bus_out = 8'b00000000;
17      endcase
18 end
19 endmodule
```



Instruction Memory Module

The `instruction_memory` module stores a set of instructions and outputs the instruction at the address specified by the `AR` register.

```
1  module instruction_memory(enable, ins_out, AR, first_load, init_ins, clk);
2      input [3:0] AR;
3      input first_load, clk, enable;
4      input [127:0] init_ins;
5      output reg [7:0] ins_out;
6      reg [7:0] memory [15:0];
7
8      integer i;
9
10     always @(posedge clk) begin
11         if (first_load == 1) begin
12             for (i = 0; i < 16; i = i + 1)
13                 memory[i] = init_ins[i*8+:8];
14         end else if (enable == 1)
15             ins_out = memory[AR];
16     end
17 endmodule
```



Data Memory Module

The `data_memory` module stores data and allows reading and writing operations based on the control signals and the address specified.

```
module data_memory(enable, data_out, AR, data_in, write, first_load, init_data, clk);
    input [3:0] AR;
    input write, first_load, clk, enable;
    input [7:0] data_in;
    input [127:0] init_data;
    output reg [7:0] data_out;
    reg [7:0] memory [15:0];

    integer i;

    always @(posedge clk) begin
        if (first_load == 1) begin
            for (i = 0; i < 16; i = i + 1)
                memory[i] = init_data[i*8+:8];
        end else if (enable == 1 && write == 1)
            memory[AR] = data_in;
        else if (enable == 1)
            data_out = memory[AR];
    end
endmodule
```

ALU Module

The `alu` module performs arithmetic and logic operations based on the control signals.

```
module alu(out, enable, mode, op1, op2);
    input enable;
    input [2:0] mode;
    input signed [7:0] op1, op2;
    output reg signed [7:0] out;

    reg signed [7:0] root_temp;

always @ (mode, op1, op2, enable) begin
        if (enable == 1'b1) begin
            case(mode)
                3'b000: out = op1 + op2; // sum
                3'b011: out = op1 <<< 1; // arithmetic shift left
                3'b111: begin // root
                    root_temp = 0;
                    for (; root_temp * root_temp < op1;)
                        root_temp = root_temp + 1;
                    out = root_temp;
                end
                3'b100: out = ~(op1 ^ op2); // xnor
                3'b101: out = op1 >>> 1; // arithmetic shift right (division by 2)
                3'b110: out = -op1; // 2's complement
                default: out = op1;
            endcase
        end
```

```
    end
endmodule
```

Counter Module

The `counter` module generates a sequence of states that control the sequence of operations in the CPU.

```
module counter(out, clk, reset);
    input clk, reset;
    output reg [2:0] out;
    always @ (posedge clk) begin
        out = out + 1;
        if (reset==1)
            out = 3'b000;
    end
endmodule
```

Decoder Module

The `decoder` module decodes the instruction into control signals that dictate the operation to be performed by the CPU.

```
module decoder(en, out, in);
    input [2:3] in;
    input en;
    output reg [7:0] out;

    always @ (in) begin
        if (en == 1) begin
            case (in)
                3'b000: out = 8'b00000001;
                3'b001: out = 8'b00000010; // 2's complement
                3'b010: out = 8'b00000100; // Arithmetic shift right
                3'b011: out = 8'b00001000;
                3'b100: out = 8'b00010000;
                3'b101: out = 8'b00100000;
                3'b110: out = 8'b01000000; // XNOR
                3'b111: out = 8'b10000000;
            endcase
        end
    end
endmodule
```

Simulation and Testing

The design was tested using several testbench modules to verify its functionality.

1.0.1 Counter Test

The `counter_test` module verifies the operation of the `counter` module.

```
module counter_test3;
    reg clk, reset;
    wire [2:0] out;
    counter count(out, clk, reset);
    always begin
        clk = ~clk;
        #10;
    end
    initial begin
        clk = 1; reset = 1;
        #15;
        reset = 0;
        #110;
        reset = 1;
        #20;
    end
endmodule
```

1.0.2 Decoder Test

The `decoder_test` module verifies the operation of the `decoder` module.

```
module decoder_test3;
    reg [2:0] in;
    wire [7:0] out;
    reg en;
    decoder deco(en, out, in);
    initial begin
        en = 1;
        in = 3'b000;
        #100;
        in = 3'b001; // 2's complement
        #100;
        in = 3'b010; // Arithmetic shift right
        #100;
        in = 3'b011;
        #100;
        in = 3'b100;
        #100;
        in = 3'b101;
        #100;
        in = 3'b110; // XNOR
        #100;
        in = 3'b111;
        #100;
    end
```

```
endmodule
```

1.0.3 ALU Test

The `alu_test` module verifies the operation of the `alu` module.

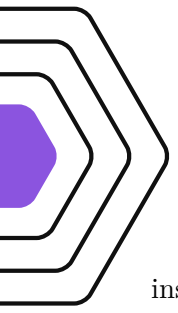
```
module alu_test3;
    reg [7:0] op1, op2;
    reg enable;
    reg [2:0] mode;
    wire [7:0] out;
    alu alu(out, enable, mode, op1, op2);
    initial begin
        enable = 1;
        mode = 3'b000;
        op1 = 8'b00001001;
        op2 = 8'b00000001;
        #10;
        mode = 3'b011;
        #10;
        mode = 3'b111;
        #10;
        mode = 3'b100; // XNOR
        #10;
        mode = 3'b101; // Arithmetic shift right
        #10;
        mode = 3'b110; // 2's complement
        #10;
    end
endmodule
```

1.0.4 CPU Test

The `cpu_test` module verifies the operation of the entire CPU.

```
module cpu_test3;
    reg [127:0] ins_mem, data_mem;
    reg clk;
    cpu cpu(ins_mem, data_mem, clk);
    always begin
        clk = ~clk;
        #10;
    end

    initial begin
        clk = 0;
        ins_mem = 0;
        data_mem = 0;

        // Load operand from memory location 0 to AC
        ins_mem[7-:8]   = 8'b01000000; // Load mem_0 into AC

        // Check if AC is zero
        ins_mem[15-:8]  = 8'b00000001; // Subtract 1 (no-op to use skip next instructi

```

```
21          // If AC is zero, jump to the end
22          ins_mem[23-:8]  = 8'b11111111; // Exit (jump to end)
23
24          // Otherwise, find the highest set bit
25          ins_mem[31-:8]  = 8'b01100110; // 2's complement of AC
26          ins_mem[39-:8]  = 8'b01010001; // Store 2's complement in mem_1 (used as count
27
28          // Loop: Shift right until AC becomes zero
29          ins_mem[47-:8]  = 8'b01000100; // Arithmetic shift right (division by 2) on AC
30          ins_mem[55-:8]  = 8'b00000001; // Subtract 1 (no-op to use skip next instructi
31          ins_mem[63-:8]  = 8'b11110111; // Jump to loop if not zero
32
33  // At this point, mem_1 contains the count of shifts
34          // Compute 2^(mem_1 + 1) by shifting left
35          ins_mem[71-:8]  = 8'b01010010; // Load mem_1 into AC
36          ins_mem[79-:8]  = 8'b00000010; // Add 1 to AC (increment the count)
37          ins_mem[87-:8]  = 8'b01010001; // Store back to mem_1 (now it contains the exp
38
39          // Initialize result to 1
40          ins_mem[95-:8]  = 8'b01110101; // Load 1 into AC (root of mem_1 is 1)
41          ins_mem[103-:8] = 8'b01010010; // Store 1 in mem_1
42
43          // Compute 2^(count + 1)
44          ins_mem[111-:8] = 8'b01000001; // Load mem_1 into AC
45          ins_mem[119-:8] = 8'b00110000; // Multiply AC by 2
46          ins_mem[127-:8] = 8'b11111110; // Jump to next step if count + 1 shifts comple
47          ins_mem[127-:8] = 8'b01010000; // Store result back to mem_0
48
49          // Exit
50          ins_mem[127-:8] = 8'b11111111; // Exit
51
52          // Initial data: operand to round up
53          data_mem[7-:8]  = 8'b00001010; // Initial data at mem_0 (10)
54      end
55  endmodule
```



Conclusion

The CPU design successfully demonstrates the fundamental principles of a simple CPU, including instruction fetch, decode, execute, and writeback stages. Further enhancements could include support for additional instructions, pipelining, and improved memory management.