HDL Final Project Report

Ethan Partidas

April 2023

1 Processor Specifications

This processor is an 8 bit processor with a single-cycle datapath written in Verilog. Each register stores an 8 bit value, and all operations are 8 bit. Instructions are 16 bit, and therefore are accessed by two consecutive locations in memory.

The processor has 3 inputs and 4 outputs:

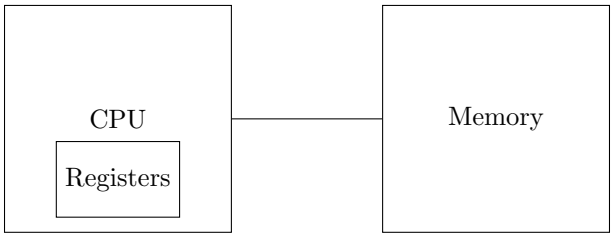
- clock: clock signal
- reset: indicates that the internal CPU variables should be reset to 0.
- memory: gives the CPU access to the main memory
- write_address: where the CPU wants to write to in memory
- write_value: the value the CPU wants to write at write_address
- write: whether the CPU wants to write to memory
- halt: whether the CPU has finished execution

The ISA of the CPU has 16 instructions. Each instruction is split into 3 or 4 fields with 4 or 8 bits per field. The following is a summary of those instructions.

Instruction	15:12	11:8	7:4	3:0
HALT	0	X	X	X
LOAD	1	RD	RS	X
STORE	2	RD	RS	X
LI	3	RD	IM	
ADD	4	RD	RS1	RS2
SUB	5	RD	RS1	RS2
MUL	6	RD	RS1	RS2
DIV	7	RD	RS1	RS2
NOT	8	RD	RS	X
AND	9	RD	RS1	RS2
OR	A	RD	RS1	RS2
XOR	B	RD	RS1	RS2
JUMP	C	X	IM	
BEQ	D	RD	RS1	RS2
BGT	E	RD	RS1	RS2
BLT	F	RD	RS1	RS2

They all do what you expect. For an exact specification of what each instruction does, see the HDL Code section.

2 Block Diagram



The registers are contained within the CPU module, and the memory exists as an array of bytes outside of the CPU.

3 Test Program

The program we will use to test the CPU will generate 10 Fibonacci numbers. This involves using two registers X and Y to store pairs of consecutive Fibonacci numbers, and adding them repeatedly to get the desired output. The output is written into memory, which the test bench will then print out.

3.1 Registers

Register	Value
0	Counter
1	0A
2	Register X
3	Register Y
4	01
5	Branch Destination
6	Write Address
7	Temp
8	00

3.2 Assembly Code

Address	Instruction	Byte Code
00	LI 1,0A	310A
02	LI 2,01	3201
04	LI 3,01	3301
06	LI 4,01	3401
08	LI 5,1E	351E
0A	LI 6,80	3680
0C	BEQ 5,0,1	D501
0E	STORE 6,2	2620
10	ADD 7,2,3	4723
12	OR 2,3,8	A238
14	OR 3,7,8	A378
16	ADD 0,0,4	4004
18	ADD 6,6,4	4664
1A	JUMP 0C	C00C
1E	HALT	0000

4 HDL Code

4.1 CPU

```
module xcpu (
  input clock,
  input reset,
  input reg [7:0] memory [0:255],
  output reg [7:0] write_address,
  output reg [7:0] write_value,
  output reg write,
  output reg halt
);

  reg [7:0] program_counter = 0;

  reg [7:0] register_file [0:15];

  reg [3:0] operation_code = 0;
  reg [3:0] destination_register = 0;
  reg [3:0] source_register_1 = 0;
  reg [3:0] source_register_2 = 0;
  reg [7:0] immediate = 0;

  integer i;

  always @(posedge clock) begin
    if (reset) begin
      write_address = 0;
      write_value = 0;
      write = 0;
      halt = 0;
      program_counter = 0;

      for (i = 0; i < 16; i = i + 1) begin
        register_file[i] = 0;
      end

    end else if (~halt) begin
      operation_code = memory[program_counter][7:4];
      destination_register = memory[program_counter][3:0];
      source_register_1 = memory[program_counter+1][7:4];
      source_register_2 = memory[program_counter+1][3:0];
      immediate = memory[program_counter+1];

      write = 0;

      case (operation_code)
        4'b0000: begin // HALT
          halt = 1;
        end
        4'b0001: begin // LOAD
          register_file[destination_register] = memory[register_file[source_register_1]];
        end
        4'b0010: begin // STORE
          write_address = register_file[destination_register];
          write_value = register_file[source_register_1];
          write = 1;
        end
```

```
        4'b0011: begin // LI
          register_file[destination_register] = immediate;
        end
        4'b0100: begin // ADD
          register_file[destination_register] = register_file[source_register_1] +
              register_file[source_register_2];
        end
        4'b0101: begin // SUB
          register_file[destination_register] = register_file[source_register_1] -
              register_file[source_register_2];
        end
        4'b0110: begin // MUL
          register_file[destination_register] = register_file[source_register_1] *
              register_file[source_register_2];
        end
        4'b0111: begin // DIV
          register_file[destination_register] = register_file[source_register_1] /
              register_file[source_register_2];
        end
        4'b1000: begin // NOT
          register_file[destination_register] = ~register_file[source_register_1];
        end
        4'b1001: begin // AND
          register_file[destination_register] = register_file[source_register_1] &
              register_file[source_register_2];
        end
        4'b1010: begin // OR
          register_file[destination_register] = register_file[source_register_1] |
              register_file[source_register_2];
        end
        4'b1011: begin // XOR
          register_file[destination_register] = register_file[source_register_1] ^
              register_file[source_register_2];
        end
        4'b1100: begin // JUMP
          program_counter = immediate - 2;
        end
        4'b1101: begin // BEQ
          if (register_file[source_register_1] == register_file[source_register_2]) begin
            program_counter = register_file[destination_register] - 2;
          end
        end
        4'b1110: begin // BGT
          if (register_file[source_register_1] > register_file[source_register_2]) begin
            program_counter = register_file[destination_register] - 2;
          end
        end
        4'b1111: begin // BLT
          if (register_file[source_register_1] < register_file[source_register_2]) begin
            program_counter = register_file[destination_register] - 2;
          end
        end
      endcase
      program_counter = program_counter + 2;
    end
  end
endmodule
```

4.2 Test Bench

```
// Testbench
module test;

  reg clock;
  reg reset;
  reg [7:0] memory [0:255];
  reg [7:0] write_address;
  reg [7:0] write_value;
  reg write;
  reg halt;

  integer i;

  // Instantiate design under test
  xcpu XCPU(
    .clock(clock),
    .reset(reset),
    .memory(memory),
    .write_address(write_address),
    .write_value(write_value),
    .write(write),
    .halt(halt)
  );

  initial begin
    // Dump waves
    $dumpfile("dump.vcd");
    $dumpvars(1);

    clock = 0;
    for (i = 0; i < 256; i = i + 1) begin
      memory[i] <= 0;
    end

    memory[0] <= 'h31;
    memory[1] <= 'h0A;
    memory[2] <= 'h32;
    memory[3] <= 'h01;
    memory[4] <= 'h33;
    memory[5] <= 'h01;
    memory[6] <= 'h34;
    memory[7] <= 'h01;
    memory[8] <= 'h35;
    memory[9] <= 'h1E;
    memory[10] <= 'h36;
    memory[11] <= 'h80;
    memory[12] <= 'hD5;
    memory[13] <= 'h01;
    memory[14] <= 'h26;
    memory[15] <= 'h20;
    memory[16] <= 'h47;
    memory[17] <= 'h23;
    memory[18] <= 'hA2;
    memory[19] <= 'h38;
    memory[20] <= 'hA3;
    memory[21] <= 'h78;
    memory[22] <= 'h40;
```

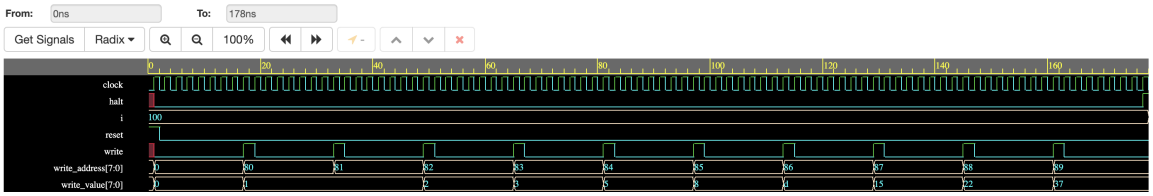
```
    memory[23] <= 'h04;
    memory[24] <= 'h46;
    memory[25] <= 'h64;
    memory[26] <= 'hC0;
    memory[27] <= 'h0C;

    reset = 1;
    cycle;
    reset = 0;

    while (~halt) begin
      cycle;
      if (write) begin
        memory[write_address] <= write_value;
      end
    end

    $display("Results.");
    for (i = 0; i < 10; i = i + 1) begin
      $display("%d", memory['h80+i]);
    end
  end

  task cycle;
    #1 clock = 1;
    #1 clock = 0;
  endtask

endmodule
```

5 Simulation Results

5.1 Text Output

```
# KERNEL: Reset.
# KERNEL: Results.
# KERNEL:  1
# KERNEL:  1
# KERNEL:  2
# KERNEL:  3
# KERNEL:  5
# KERNEL:  8
# KERNEL: 13
# KERNEL: 21
# KERNEL: 34
# KERNEL: 55
# KERNEL: Simulation has finished. There are no more test vectors to simulate.
# VSIM: Simulation has finished.
Done
```

5.2 Waveform Output



6 Conclusion

I've always wondered what it would be like to design a CPU from scratch, and with Verilog, it is surprisingly easy! It does all the hard work of implementing the state machines, allowing the programmer to focus on designing their ISA.