گزارش آزمایش دهم

دانشگاه صنعتی شریف – **بهار ۱۴۰۰**

آزمایشگاه طراحی سیستم‌های دیجیتال – دکتر اجلالی

نویسندگان:
سروش جهان‌زاد – ۹۸۱۰۰۳۸۹
علی حاتمی تاجیک – ۹۸۱۰۱۳۸۵

۱ مقدمه

در سند پیش رو مراحل انجام آزمایش دهم درس آزمایشگاه طراحی سیستم‌های دیجیتال (طراحی یک پردازنده پشته‌ای) شرح داده شده است. خروجی کار یک پردازنده بسیار ساده با روش پایپ‌لاین شده. از صبر و بردباری شما در مطالعه‌این مستند سپاسگزاریم.

با احترام

سروش جهان‌زاد، علی حاتمی تاجیک –بهار ۱۴۰۰

۲ توضیحات کلی

در این آزمایش باید یک پردازنده مبتنی بر پشته می‌ساختیم که ورودی و خروجی آن به صورت Memory Mapped باشد. برای این ماشین ۷ ورودی و یک خروجی در نظر گرفته‌ایم که خروجی در آدرس F8 و ورودی‌ها در آدرس‌های F9 تا FF جای گرفته‌اند. طراحی پردازنده به صورت مالتی سایکل است و بعضا به اینستراکشن بعدی به صورت پایپلاین فچ می‌شود. (این ترتیب برای قسمت مالتی سایکل است و برای قسمت سینگل سایکل قدری تفاوت دارد)

ساختار کلی به این صورت است که یک مادر برد در بالاترین لایه داریم. این مادربرد شامل یک پردازنده، یک پشته، یک رم و یک انکودر است که خروجی memory mapped را به چراغ‌های سون سگمنت (در اینجا از یک بیسیدی انکودر استفاده شده است) است.

۳ Stack

برای استک از کد آمایش های قبل استفاده شده است.

۴ Memory

برای کلیت مموری از کد آمایش‌های قبل استفاده شده است با این تفاوت که مموری مپ به این صورت پیاده سازی شده است که ورودی‌ها به رم وارد می‌شوند و اگر کاربر درخواست نوشن روی محتویات ورودی مپ‌شده داشت این اتفاق نادیده گرفته می‌شود. هنگامی که سیگنال ریست به رم می‌رسد کدی که خواسته شده است درون حافظه قرار می‌گیرد و تنها خاصیت این سیگنال همین است و برای تست از آن استفاده شده است. کد حافظه در ادامه آمده است:

```
module memory (
  clk,
  readwriteN,
  address,
  data_in,
  data_out,
  indata1,
  indata2,
  indata3,
  indata4,
  indata5,
  indata6,
  indata7,
  outdataio
);
  input wire [7:0] address;
  input wire clk;
  input wire [7:0] data_in;
  input wire readwriteN;
  output reg [7:0] data_out;
  input wire [7:0] indata1,
    indata2,
    indata3,
    indata4,
    indata5,
    indata6,
    indata7;

  output wire [7:0] outdataio;

  reg signed [7:0] ram [255:0];

  // F8 OUT DATA IS MEMORY MAPPED OUTPUT
  assign outdataio = ram[8'hF8];
```

```
  always @(negedge clk or negedge resetN) begin
    if (~resetN) begin
      // Code goes here and when we reset code harcoded in the ram
      ram[0] <= 8'h00; ram[1] <= 8'd12;       // PUSHC 12     # push 12
      ram[2] <= 8'h00; ram[3] <= 8'd23;       // PUSHC 23     # push 23
      ram[3] <= 8'h01; ram[4] <= 8'hF8;       // PUSH h'F8    # Push the X to stack from i/o
      ram[5] <= 8'h06;                        // ADD          # X + 23
      ram[6] <= 8'h02; ram[7] <= 8'hF0;       // POP h'F0     # F0 <= X + 23
      ram[8] <= 8'h01; ram[9] <= 8'hF0;       // PUSH h'F0    # X + 23 in stack
      ram[10] <= 8'h01; ram[11] <= 8'hF0;     // PUSH h'F0    # 2 (X + 23) s in stack
      ram[12] <= 8'h06;                       // ADD          # (X + 23)*2
      ram[13] <= 8'h07;                       // SUB          # (X + 23)*2 - 12
      ram[14] <= 8'h0F;                       // FINISH
    end else if (~readwriteN && (address < 8'hF9)) begin
      ram [address] <= data_in;
    end else begin
      data_out <= ram [address];
    end
  end

  // Memory mapped inputs
  always @(*) begin
    ram[8'hF9] = indata1;
    ram[8'hFA] = indata2;
    ram[8'hFB] = indata3;
    ram[8'hFC] = indata4;
    ram[8'hFD] = indata5;
    ram[8'hFE] = indata6;
    ram[8'hFF] = indata7;
  end

endmodule
```

۵ Seven Segment Encoder [1]

از قطعه کد زیر برای تبدیل عدد باینری به رقم‌های بیسیدی استفاده شده است:

```
module seven_seg_encoder (
  number,
  sign,
  hun,
  ten,
  one
);
  input signed [7:0] number;
  output reg [3:0]sign,
         one,
         ten,
         hun;

  wire [6:0] abs_number;

  // Get the abslout value to decode
```

1 Source [link]

```
  assign abs_number = (number < 0)? ~number + 1 : number;

  // encodong binary to BCD
  integer i;
  always @(abs_number) begin
    sign = (number < 0)? 4'd15 : 4'd14;
    hun = 4'd0;
    ten = 4'd0;
    one = 4'd0;

    for (i = 7; i >= 0; i = i-1) begin
      // add 3 to columns >= 5
      if (hun >= 5)
        hun = hun + 3;
      if (ten >= 5)
        ten = ten + 3;
      if (one >= 5)
        one = one + 3;

      // shifts
      hun = hun << 1;
      hun[0] = ten[3];

      ten = ten <<1 ;
      ten[0] = ten[0];

      one = one << 1;
      one[0] = abs_number[i];

    end
  end
endmodule
```

ALU

برای انجام محاسبات از یک ALU استفاده شده است که عملیات‌های جمع و ضرب را انجام می‌دهد و فلگ‌های صفر و علامتدار و اورفلو را بـه ما برمیگرداند:

```
module alu (
  first,
  second,
  addSubN,
  result,
  z,
  s,
  v   // Overflow Flag
);

input signed [7:0] first, second;
input addSubN;
output signed [7:0] result;
output z,s,v;
wire signed [8:0] first_extended, second_extended, result_extended;

assign first_extended = {first[7], first};
```

```
assign second_extended = {second[7], second};

assign result_extended = addSubN? first_extended + second_extended
                  : first_extended - second_extended;



assign result = result_extended[7:0];

assign z = (result == 0);
assign s = (result[7]==1);
assign v = (result_extended > 127 || result_extended < -128);

endmodule
```

۶ مادربرد

در مادربرد صرفا بخش‌های مختلف را کنار هم می‌گذاریم تا عملکرد پردازنده کامل شود و در واقع این بخش، ماژول بالایی طراحی ماست.

```
module multiCycle (
  indata1,
  indata2,
  indata3,
  indata4,
  indata5,
  indata6,
  indata7,
  error,
  clk,
  resetN,
  haltN,
  seven_seg_sign,
  seven_seg_digit_1,
  seven_seg_digit_2,
  seven_seg_digit_3,
  finish_flag
);
  // PORTS
  // I/O Mapped Inputs
  input [7:0] indata1,
  indata2,indata3,
  indata4,indata5,
  indata6,indata7;

  // Clock and control signals
  input clk,
  resetN, haltN;

  // Error flag (If overflowed or inputs were negative)
  output error, finish_flag;

  // Seven Segment LEDs digits and sign
  // We assumed that negative sign will be shown by input 15 (4'b1111)
  // and 14 for off
```

```
output [3:0] seven_seg_sign,
seven_seg_digit_1,
seven_seg_digit_2,
seven_seg_digit_3;

// Stack
wire stack_full,stack_empty, stack_push, stack_pop;
wire [7:0] stack_data_in, stack_data_out;
stack stack1(
  stack_data_out,
  stack_full,stack_empty,
  stack_push,stack_pop,
  stack_data_in,
  clk,resetN);

// RAM
wire [7:0] outmapped;
wire ram_readWriteN;
wire [7:0] ram_address;
wire [7:0] ram_data_in;
wire [7:0] ram_data_out;
memory memory1(
  clk,ram_readWriteN,
  ram_address,ram_data_in,ram_data_out,
  indata1,indata2,indata3,indata4,
  indata5,indata6,indata7,
  outmapped, resetN);

// PROCESSOR
wire overflow, stack_error;
multicycle_processor processor(
  clk,
  haltN,
  resetN,
  ram_readWriteN,
  ram_address,
  ram_data_out,
  ram_data_in,
  stack_data_in,
  stack_data_out,
  stack_full,
  stack_empty,
  stack_push,
  stack_pop,
  overflow,
  stack_error,
  finish_flag);

// ERROR
wire inputs_sign;
assign inputs_sign = indata1[7] |
            indata2[7] |
            indata3[7] |
            indata4[7] |
            indata5[7] |
            indata6[7] |
            indata7[7]; // If one of them is 1 then error signal should turn on.
```

```
  assign error =  overflow ||     // If overflowed over procedures
           stack_error ||  // If stack error occured (Pop from empty stack or push into full stack)
           inputs_sign ||
           (outmapped > 127);    // If inputs were negative
  // 7-segment
  seven_seg_encoder encoder(
    outmapped,
    seven_seg_sign,
    seven_seg_digit_3,
    seven_seg_digit_2,
    seven_seg_digit_1);
endmodule
```

۷ پردازنده

مالتی سایکل

```
module multicycle_processor (
  clk,
  haltN,
  resetN,
  ram_readWriteN,
  ram_address,
  ram_data_in,
  ram_data_out,
  stack_data_out,
  stack_data_in,
  stack_full,
  stack_empty,
  stack_push,
  stack_pop,
  overflow,
  stack_error,
  finish_flag
);
// ports
input wire clk, haltN, resetN, stack_full, stack_empty;
input wire [7:0] ram_data_in, stack_data_in;
output reg [7:0] ram_address, ram_data_out, stack_data_out;
output reg ram_readWriteN, stack_push, stack_pop;
output reg overflow = 0, stack_error = 0, finish_flag = 0;

// States
localparam HALTED =     4'd15;// FINISH
localparam INITS =      4'd14;
localparam FETCH1 =     4'd12;
localparam FETCH2 =     4'd13;
localparam PUSHC =      4'd0; // pushc opcode
localparam PUSHM_READ = 4'd1; // push  opcode
localparam POP =        4'd2; // pop   opcode
localparam JUMP =       4'd3; // jump  opcode
localparam JZ =         4'd4; // z     opcode
localparam JS =         4'd5; // js    opcode
localparam ADD_FIRST =  4'd6; // add   opcode
localparam ADD_SECOND = 4'd10;
localparam SUB =        4'd7; // sub   opcode
```

```
localparam POP_WRITE =  4'd8;

// Local used registers
reg [3:0] current_state = 4'd14;
reg [3:0] opcode;
reg [7:0] oprand;

// PROGRAM COUNTER
reg [7:0] pc;

// ALU
reg [7:0] first;
reg addSubN;
wire [7:0] result;
wire z,s,v;

alu alu_instance(first, stack_data_in, addSubN, result, z, s);

// Flags
reg z_flag, s_flag;

always @(posedge clk) begin
  if (~resetN) begin
    current_state <= INITS;
  end else if (haltN || finish_flag) begin   // When the processor is halted we are changing inputs a
nd so
    case (current_state)
      INITS: begin // program counter will be set, read signal will be set
        pc <= 16'b0;
        ram_readWriteN <= 1'b1;
        stack_pop <= 1'b0;
        stack_push <= 1'b0;
        ram_address <=  1'b0;
        ram_readWriteN <= 1'b1;
        current_state <= FETCH1;
      end

      FETCH1: begin // opcode be fetched, the address must be updated in the last state/clock
        opcode <= ram_data_in[3:0];
        ram_readWriteN <= 1'b1;
        if (ram_data_in < 4 ) begin // if opcode is pushc/push/pop
          ram_address <= pc + 1;
          pc <= pc + 2;
        end else begin
          pc <= pc + 1;
        end

        current_state <= FETCH2;
      end

      FETCH2: begin // oprand will be fetched: this state is important even if
      // no oprand is fetched because it  will decide where we go next
      // oprand will be fetched if opcode need it and the oprand will be loaded where it is nedded
      // based on the opcode

        // make pop signal ready
```

```
if (opcode == POP ||
  opcode == JUMP ||
  (opcode == JZ && z_flag) ||
  (opcode == JS && s_flag) ||
  opcode == ADD_FIRST ||
  opcode == SUB) stack_pop <= 1'b1;
else stack_pop <= 1'b0;
// make stack push signal ready.
if (opcode == PUSHC) stack_push <= 1'b1;
else stack_push <= 1'b0;

// reset ram mode to read
ram_readWriteN <= 1'b1;

case (opcode)
  PUSHC: begin    // data to push prepared here
    // prepare data to be pushed
    stack_push <= 1'b1;
    stack_data_out <= ram_data_in;
    current_state = PUSHC;

    // if the stack is full the overflow flag turn 1
    if (stack_full) stack_error <= 1'b1;

    // prepare to fetch next instruction
    ram_address <= pc;
  end

  PUSHM_READ: begin // prepare data to be read
    ram_address <= ram_data_in; // fetch oprand as address indirect
    current_state <= PUSHM_READ;

    // if the stack is full the overflow flag turn 1
    if (stack_full) stack_error <= 1'b1;
  end

  POP: begin
    // prepare to get the data
    oprand <= ram_data_in;
    stack_pop <= 1'b1;
    current_state <= POP;

    // prepare to fetch next instruction
    ram_address <= pc;

    // if the stack is empty the overflow flag turn 1
    if (stack_empty) stack_error <= 1'b1;
  end

  JUMP:
    current_state <= JUMP;

  JZ: begin
    if (z_flag) begin // if we have the flag 1 then we should pop the address and
      // store it in pc
      current_state <= JUMP;
```

```
        end else begin // else the next instruction must be fetched
          current_state <= FETCH1;
          ram_address <= pc;
        end
      end

      JS: begin
        if (s_flag) begin // if we have the flag 1 then we should pop the address and
          // store it in pc
          current_state <= JUMP;
        end else begin // else the next instruction must be fetched
          current_state <= FETCH1;
          ram_address <= pc;
        end
      end

      ADD_FIRST: begin
        addSubN <= 1;
        current_state <= ADD_FIRST;
      end

      SUB: begin
        addSubN <= 0;
        current_state <= ADD_FIRST;
      end

      HALTED: begin
        finish_flag <= 1;
      end
    endcase
  end

  PUSHC: begin // the push signal is one from last state
  // this state do the job, change the signal, and fetch the next opcode
    stack_push <= 1'b0;

    // fetch1 procedures
    opcode <= ram_data_in[3:0];
    if (ram_data_in < 4 ) begin // if opcode is pushc/push/pop
      ram_address <= pc + 1;
      pc <= pc + 2;
    end else begin
      pc <= pc + 1;
    end
    current_state <= FETCH2;
  end

  PUSHM_READ: begin   // read signal is one from last state - it will store the data
  // in stack in-data reg to be pushed next clock, push signal should become one here
    // prepare read data to be pushed
    stack_data_out <= ram_data_in;
    stack_push <= 1'b1;
    current_state <= PUSHC; // the works that would be done is completly the same
                  // as in the PUSHC state
    // prepare for fetch instruction
    ram_address <= pc;
```

```
end

POP: begin  // pop signal is 1 from last state, so the data will be stored in the ram_data_out
// pop signal become 0 and next opcode should be fetched
  stack_pop <= 1'b0;
  // prepare data to be stored
  ram_data_out <= stack_data_in;
  ram_readWriteN <= 1'b0;
  ram_address <= oprand;

  // catch the next instruction
  // fetch1 procedures
  opcode <= ram_data_in[3:0];
  current_state <= POP_WRITE;
end

POP_WRITE: begin
  ram_readWriteN <= 1'b1;
  if (opcode < 4 ) begin // if opcode is pushc/push/pop and need operand
    ram_address <= pc + 1;
    pc <= pc + 2;
  end else begin
    pc <= pc + 1;
  end
  current_state <= FETCH2;
end

JUMP: begin // here pop flag is enabled from the last state, so the data will be popped to the
// pc.  then we go to the FETCH1 state
// we use this state for JZ and JS too, becuase the condition is checked in fetch2 state
  stack_pop <= 1'b0;
  pc <= stack_data_in;
  ram_address <= stack_data_in;
  current_state <= FETCH1;
end


// Same procedures goes for the addition and subtraction! so we just change the addSubN
// in fetch2 and use same states for the addition and subtraction
ADD_FIRST: begin // first data that is poped will be caught here
  first <= stack_data_in;

  // prepare to fetch next instruction
  ram_address <= pc;
  current_state <= ADD_SECOND;
end

ADD_SECOND: begin // second number is poped and is on stack data in. this wire goes
// to ALU directly to prepare the result faster
  stack_data_out <= result;
  stack_pop <= 1'b0;
  // push this result to the stack in next clock
  stack_push <= 1'b1;

  // update flags
  z_flag <= z;
  s_flag <= s;
```

```
        if (v) overflow <= 1'b1;

        // catch the next instruction
        // fetch1 procedures
        opcode <= ram_data_in[3:0];
        if (ram_data_in < 4 ) begin // if opcode is pushc/push/pop and need operand
          ram_address <= pc + 1;
          pc <= pc + 2;
        end else begin
          pc <= pc + 1;
        end
        current_state <= FETCH2;
      end

    endcase
  end
end

endmodule
```

سینگل سایکل

در این بخش، پردازنده‌ای را در نظر می‌گیریم که همه‌ی واحدهای گفته شده را درون خود دارد و از طریق حافظه‌ی داخلی‌اش با فضای بیرون ارتباط برقرار می‌کند. در این پردازنده همانند پردازنده‌ی قبل هنگامی که سیگنال haltN برابر صفر باشد، پردازنده می‌ایستد و می‌توانیم با سیگنال‌های direct، دستورهایی را داخل حافظه بنویسیم. پس از اتمام این کار، می‌توانیم سیگنال haltN را برابر ۱ قرار دهیم تا پردازنده شروع به کار کند و دستوری که PC به آن اشاره می‌کند را اجرا کند. در هر کلاک، این پردازنده PC را بسته به اینکه دستور اجرا شده عملوند داشته یا نه، یک یا دو واحد زیاد می‌کند و به این ترتیب روند برنامه ادامه پیدا می‌کند.

```
module processor (

  output signed [7:0] direct_read_data,
  input [7:0] direct_read_address,
  input [7:0] direct_write_address,
  input signed [7:0] direct_write_data,
  input direct_memory_write,
  input clk,
  input resetN,
  input haltN
);
  localparam PUSHC = 4'b0000;
  localparam PUSH = 4'b0001;
  localparam POP = 4'b0010;
  localparam JUMP = 4'b0011;
  localparam JZ = 4'b0100;
  localparam JS = 4'b0101;
  localparam ADD = 4'b0110;
  localparam SUB = 4'b0111;

  reg [7:0] pc, next_pc;
  reg z_flag, s_flag, push, pop, mem_write;

  // stack
  reg [3:0] stack_pointer; // empty position at the top of stack
  reg signed [7:0] stack_mem [0:7];
  reg signed [7:0] data_to_memory, data_to_stack;
  wire stack_full = (stack_pointer == 8);
```

```
wire stack_empty = (stack_pointer == 0);
wire [7:0] top_of_stack = stack_empty ? 0 : stack_mem[stack_pointer - 1];

// instruction memory
// reg [11:0] instruction_memory [0:255];
// wire [11:0] instruction = instruction_memory[pc];
wire [11:0] instruction = {data_memory[pc][7:4], data_memory[pc + 1][7:0]};
wire [3:0] opcode = instruction[11:8];
wire [7:0] operand = instruction[7:0];

// data memory
reg signed [7:0] data_memory [0:255];
wire [7:0] memory_read_address = operand;
wire [7:0] memory_write_address = operand;
wire [7:0] data_from_memory = data_memory[memory_read_address];
assign direct_read_data = data_memory[direct_read_address];

always @(*) begin
  next_pc = pc + 1;
  push = 0;
  pop = 0;
  mem_write = 0;
  case (opcode)
    PUSHC: begin
      next_pc = pc + 2;
      push = 1;
      data_to_stack = operand;
    end
    PUSH: begin
      next_pc = pc + 2;
      push = 1;
      data_to_stack = data_from_memory;
    end
    POP: begin
      next_pc = pc + 2;
      pop = 1;
      mem_write = 1;
      data_to_memory = top_of_stack;
    end
    JUMP: begin
      pop = 1;
      next_pc = top_of_stack;
    end
    JZ: begin
      if (z_flag) begin
        pop = 1;
        next_pc = top_of_stack;
      end
    end
    JS: begin
      if (s_flag) begin
        pop = 1;
        next_pc = top_of_stack;
      end
    end
  endcase
```

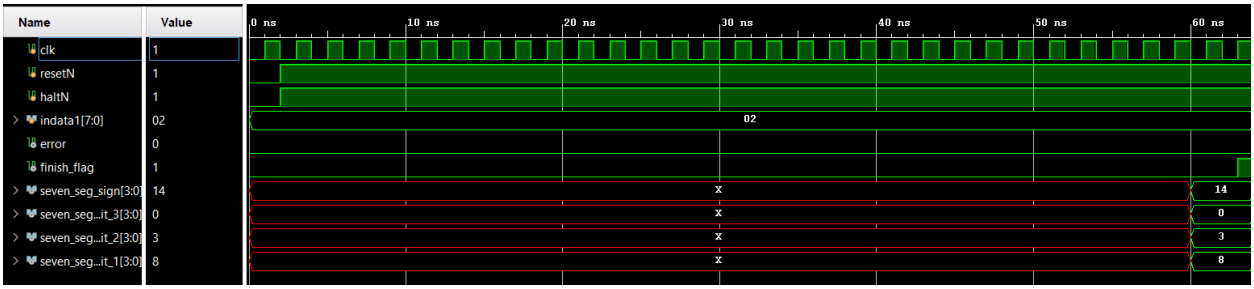
```
  end

  integer i;
  always @(posedge clk or negedge resetN) begin
    if (~resetN) begin
      pc <= 0;
      z_flag <= 0;
      s_flag <= 0;
      stack_pointer <= 0;
      for (i = 0; i < 256; i = i + 1) begin
        data_memory[i] <= 8'b0;
      end
    end
    else if (haltN) begin
      pc <= next_pc;
      case (opcode)
        ADD: begin
          stack_mem[stack_pointer - 2] <= stack_mem[stack_pointer - 1] + stack_mem[stack_pointer -
2];
          stack_pointer <= stack_pointer - 1;
          z_flag <= (stack_mem[stack_pointer - 1] + stack_mem[stack_pointer - 2] == 0);
          s_flag <= (stack_mem[stack_pointer - 1] + stack_mem[stack_pointer - 2] < 0);
        end
        SUB: begin
          stack_mem[stack_pointer - 2] <= stack_mem[stack_pointer - 2] - stack_mem[stack_pointer -
1];
          stack_pointer <= stack_pointer - 1;
          z_flag <= (stack_mem[stack_pointer - 2] - stack_mem[stack_pointer - 1] == 0);
          s_flag <= (stack_mem[stack_pointer - 2] - stack_mem[stack_pointer - 1] < 0);
        end
        default: begin
          if (pop & ~stack_empty) begin
            stack_pointer <= stack_pointer - 1;
          end
          if (push & ~stack_full) begin
            stack_mem[stack_pointer] <= data_to_stack;
            stack_pointer <= stack_pointer + 1;
          end
          if (mem_write) begin
            data_memory[memory_write_address] <= data_to_memory;
          end
        end
      endcase
    end
    else begin
      if (direct_memory_write) begin
        data_memory[direct_write_address] <= direct_write_data;
      end
    end
  end
endmodule
```

۸ تست

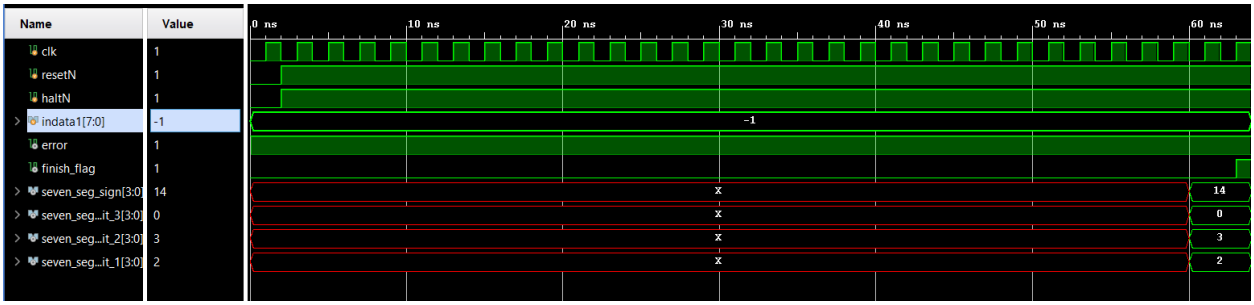
تست مالتی سایکل

```
`timescale 1ns / 1ps

module cpu_tb;
  reg [7:0] indata1,
  indata2,indata3,
  indata4,indata5,
  indata6,indata7;

  // Clock and control signals
  reg clk = 0,
  resetN = 0, haltN = 0;

  // Error flag (If overflowed or inputs were negative)
  wire error, finish_flag;

  // Seven Segment LEDs digits and sign
  // We assumed that negative sign will be shown by input 15 (4'b1111)
  // and 14 for off
  wire [3:0] seven_seg_sign,
  seven_seg_digit_1,
  seven_seg_digit_2,
  seven_seg_digit_3;

  multiCycle machine(
  indata1,indata2,indata3,indata4,
  indata5,indata6,indata7,
  error,clk,resetN,haltN,
  seven_seg_sign,
  seven_seg_digit_1,
  seven_seg_digit_2,
  seven_seg_digit_3,
  finish_flag);

  initial begin
    forever
      #1 clk = ~clk;
  end

  initial begin
    indata1 = 2;
    indata2 = 0;
    indata3 = 0;
    indata4 = 0;
    indata5 = 0;
    indata6 = 0;
    indata7 = 0;

    #2 haltN = 1;
    resetN = 1;

    #62 $finish;
  end
endmodule
```

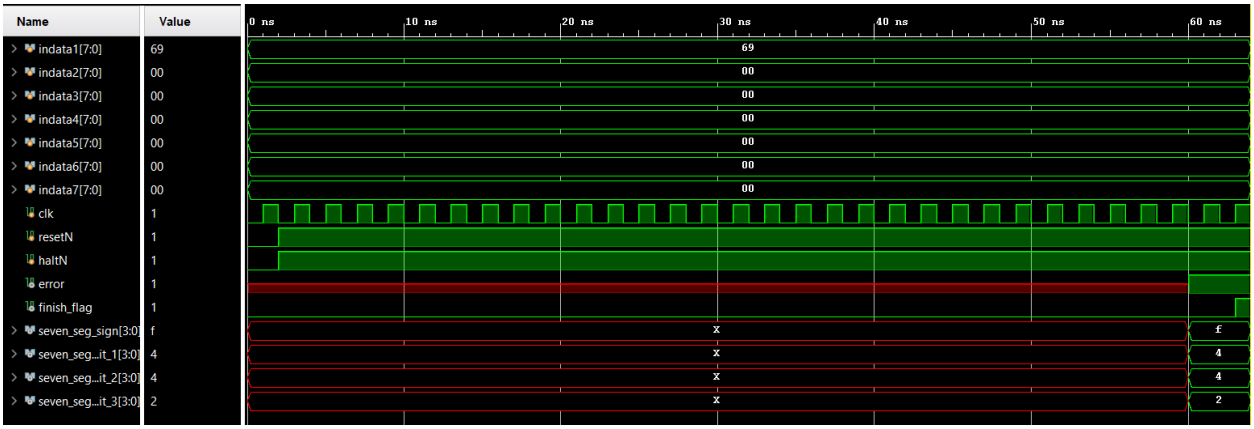
در این تست عدد ورودی X را ۲ میدهیم و قدری صبر می‌کنیم تا عمیات به پایان برسد. نتیجه همانطور که انتظار داشتیم ۳۸ بدست آمده است:



یکبار هم برای برسی کردن اورها یکبار عدد منفی و ورودی میدهیم و یکبار هم عددی که نتیجه را بزرگتر از ۱۲۷ کند:



زمانی که عدد منفی وارد کرده ایم با اینکه جواب درست را پردازنده به دست آورده است ولی از همان ابتدا ارور داریم.



اینجا هم که پاسخ بزرگتر از ۱۲۷ شده است ارور دریافت کرده ایم.

تست سینگل سایکل

```
module processor_tb ();
  wire [7:0] direct_read_data;
  reg [7:0] direct_read_address;
  reg [7:0] direct_write_address;
  reg [7:0] direct_write_data;
  reg direct_memory_write;
  reg clk;
  reg resetN;
  reg haltN;

  processor p(
    .direct_read_data(direct_read_data),
    .direct_read_address(direct_read_address),
    .direct_write_address(direct_write_address),
    .direct_write_data(direct_write_data),
    .direct_memory_write(direct_memory_write),
    .clk(clk),
    .resetN(resetN),
    .haltN(haltN)
  );

  initial begin
    clk = 0;
    forever #5 clk = ~clk;
  end

  reg [11:0] instructions [0:127];
  integer i;
  initial begin
    $dumpfile("processor.vcd");
    $dumpvars(0, processor_tb);
    $dumpvars(0, p);

    instructions[0] = 12'b000111111111;
    instructions[1] = 12'b000000010111;
    instructions[2] = 12'b011000000000;
    instructions[3] = 12'b001011111111;
    instructions[4] = 12'b000111111111;
    instructions[5] = 12'b000111111111;
    instructions[6] = 12'b011000000000;
    instructions[7] = 12'b000000001100;
    instructions[8] = 12'b011100000000;
    instructions[9] = 12'b001011111111;

    haltN = 0;
    resetN = 0;
    direct_read_address = 255;
    direct_memory_write = 0;
    #10 resetN = 1;
    direct_memory_write = 1;
    direct_write_address = 0;
    for (i = 0; i < 10; i = i + 1) begin
      direct_write_data = {instructions[i][11:8], 4'b0000};
      #10;
      direct_write_address = direct_write_address + 1;
```

```
        if (instructions[i][11:8] < 3) begin
          direct_write_data = instructions[i][7:0];
          #10;
          direct_write_address = direct_write_address + 1;
        end
      end
      direct_write_address = 255;
      direct_write_data = 17;
      #10;
      direct_memory_write = 0;
      haltN = 1;
      #100;
      haltN = 0;
      #10;
      $finish;
    end
endmodule
```

داده ابتدایی ۱۷ بوده است که ۶۸ جواب درست بدست آمده است: