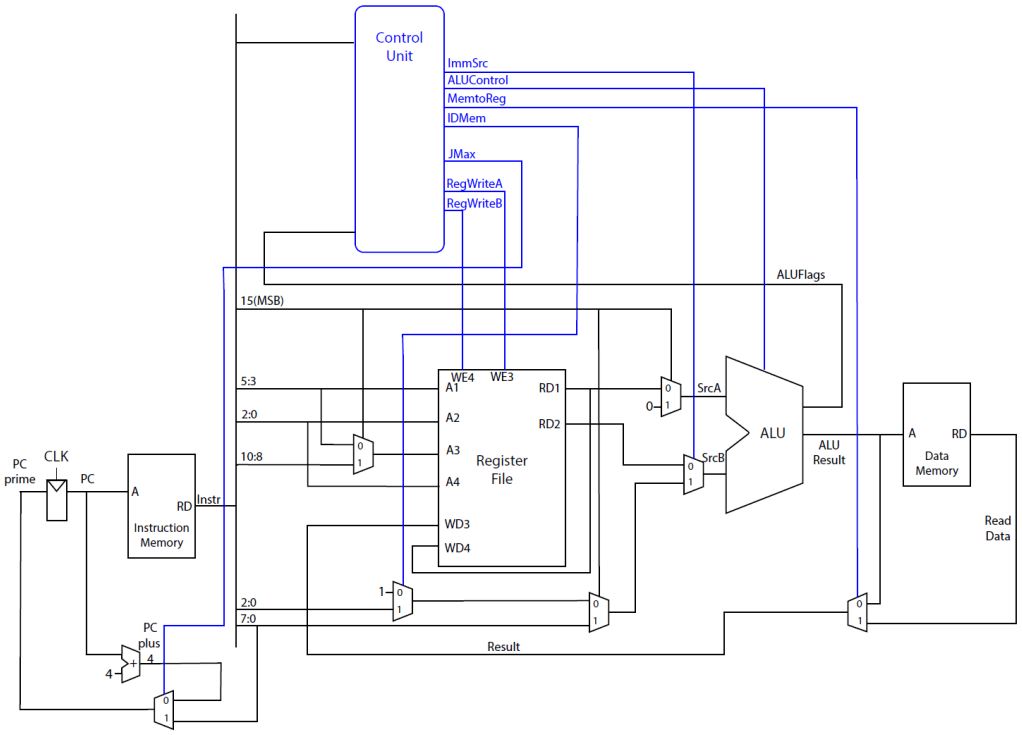
شرح آزمایش

در این آزمایش بنا به آن است که برای دستورات صورت پروژه مذکور پردازنده‌ای با ساختار تک‌چرخه‌ای پیاده‌سازی کنیم.

به بررسی ساختار تک تک اجزای این پردازنده می‌پردازیم:

-



ماژول top

```
module top (
        input logic clk, reset,
        output logic [15:0] Instr);

  logic [7:0] PC, DataAdr, ReadData, ShowData;
  logic [1:0] ShowEn;

  cpu cpu(clk, reset, PC, Instr,
        DataAdr, ReadData,
        ShowEn, ShowData);

  imem imem(PC, Instr);
  dmem dmem(clk, DataAdr, ReadData);

  BCD bcd(ShowEn[0], ShowData);
  SevenSeg ss(ShowEn[1], ShowData);

endmodule
```

عملکرد:

این ماژول ریشه پردازنده و بطور مستقیم شامل ماژول cpu و imem و dmem (مموری‌های خارج از دیتا پت درنظر گرفته شده‌اند.) و همچنین bcd و sevenseg برای نمایش مناسب اعداد می‌باشد.

ورودی‌ها:

clk : کلاک اصلی سیستم
reset: بیت اصلی ریست

خروجی‌ها:

Instr : دستورالعمل‌های 16 بیتی پردازنده

ماژول imem

```
module imem(input logic [7:0]   a,
            output logic [15:0] rd);
  logic   [15:0]  RAM[255:0];

  initial
    $readmemh("memfile.dat",RAM);

  assign rd = RAM[a[7:2]]; // word alligned
endmodule
```

عملکرد:

این ماژول وظیفه واکشی دستورات را دارد و از فایل دستورات (memfile.dat) با تابع readmemh آدرس 16 بیتی که درون فایل به صورت HEX نوشته شده است را میخواند و درون اینستراکشن مموری می‌ریزد.

اینستراکشن دارای 256 خانه 16 بیتی می‌باشد.

و آدرس هر کدام از این خانه‌ها 8 بیتی می‌باشد که 2 بیت آن word offset می‌باشد و برای دسترسی به هر کدام از خانه‌ها فقط به 6 بیت دیگر احتیاج داریم.

ورودی‌ها:

a : آدرس یک دستور در اینستراکشن مموری

خروجی‌ها:

rd : دستور 16 بیتی خوانده شده از حافظه

ماژول dmem

```
module dmem(input logic clk,

            input logic [7:0] a,
            output logic [7:0] rd);
   logic [7:0] RAM[255:0];

   assign rd = RAM[a[7:2]]; //word alligned
endmodule
```

عملکرد:

این ماژول وظیفه واکشی داده‌ها را دارد بدین شکل که دوباره یک آدرس 8 بیتی را می‌گیرد و داده‌ی متناظر با آدرس مورد نظر را به عنوان خروجی می‌دهد.

تمامی نکات درباره آدرس حافظه که قبلا گفته‌شد اینجا نیز برقرار است.

ورودی‌ها:

a : آدرس یک دستور در دیتا مموری

خروجی‌ها:

rd : داده 8 بیتی خوانده شده از حافظه

ماژول cpu

```
module cpu( input logic             clk, reset,
            output logic    [7:0]   PC,
            input logic     [15:0]  Instr,
            output logic    [7:0]   ALUResult,
            input logic     [7:0]   ReadData,
            output logic    [1:0]   Show,
```

```
        output logic   [7:0]  ShowData);

  logic [3:0] ALUFlags, ALUControl;
  logic RegWriteA, RegWriteB, ImmSrc,
        IDmux, JMux,
        MemtoReg;

  controller c(   clk, reset, Instr, ALUFlags,
                  RegWriteA, RegWriteB, ImmSrc,
                  IDmux, JMux,
                  MemtoReg, Show, ALUControl);
  datapath dp(    clk, reset, Instr, ALUFlags,
                  RegWriteA, RegWriteB, ImmSrc,
                  IDmux, JMux,
                  MemtoReg, ShowData,
                  ReadData, PC, ALUResult,
                  ALUControl);
endmodule
```

عملکرد:

این ماژول دربرگیرنده بخش‌های اصلی پردازنده یعنی datapath و controller می‌باشد.

ماژول datapath

```
module datapath(    input logic          clk, reset,
                    input logic [15:0]  Instr,
                    output logic [3:0]  ALUFlags,
                    input logic          RegWriteA, RegWriteB, ImmSrc,
                                         IDmux, JMux,
                                         MemtoReg,
                    output logic [7:0]  ShowData,
                    input logic [7:0]       ReadData,
                    output logic [7:0]  PC, ALUResult,
                    input logic [3:0]       ALUControl);

  logic [7:0] PCPlus4, PCNext;
  logic [2:0] RA3;
  logic [7:0] RD2;
  logic [7:0] Result, SrcA, SrcB;
  logic [7:0] idtemp, Imm;

  //assign ShowData = RD1;

  //  next PC logic
  mux2 #(8)   pcmux(PCPlus4, Instr[7:0], JMux, PCNext);
  flopr #(8)  pcreg(clk, reset, PCNext, PC);
  adder #(8)  pcadd(PC, 8'b100, PCPlus4);

  // register file logic
  mux2 #(3)   ra3mux(Instr[5:3], Instr[10:8], Instr[15], RA3);
  regfile     rf(clk, RegWriteA, RegWriteB,
                 Instr[5:3], Instr[2:0],
                 RA3, Instr[2:0],
                 Result,
                 ShowData, RD2);
  mux2 #(8)   resmux(ALUResult, ReadData, MemtoReg, Result);

  // ALU Logic
  mux2 #(8)   srcamux(ShowData, 8'b0, Instr[15], SrcA);
  mux2 #(8)   idmux({5'b0,Instr[2:0]}, 8'b1, IDmux, idtemp);
  mux2 #(8)   immmux(idtemp, Instr[7:0], Instr[15], Imm);
  mux2 #(8)   srcbmux(RD2, Imm, ImmSrc, SrcB);
  alu          alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);

endmodule
```

عملکرد:

این بخش وظیفه پردازش داده‌ها و انجام عملیات مربوطه با توجه به دستورات برنامه را دارد.

لازم به ذکر است از 2 مولتی‌پلکسر برای INC DEC استفاده شده است.

ماژول regfile

```
module regfile(input logic     clk,
               input logic       we3, we4,
               input logic [2:0]   ra1, ra2,
                                   wa3, wa4,
               input logic [7:0]   wd3,
               output logic [7:0]  rd1, rd2);
 logic [7:0] rf[7:0];

// four ported register file
// read two ports combinationally
// write third & fourth port on rising edge of clock

 always_ff @(posedge clk) begin
    if (we3) rf[wa3] <= wd3;
     if (we4) rf[wa4] <= rd1;
  end

 assign rd1 = rf[ra1];
 assign rd2 = rf[ra2];

endmodule
```

عملکرد:

این بخش برای نگهداری داده‌های ریجستری می‌باشد.

ماژول alu

```
module alu( input logic [7:0]   SrcA, SrcB,
            input logic [3:0]   ALUControl,
            output logic [7:0]  ALUResult,
```

```
            output logic [3:0]  ALUFlags);

    logic [7:0] res[1:0];
    logic   [3:0]   flags[1:0];
    initial begin
        flags[0] = 4'b0;
        flags[1] = 4'b0;
    end

    mainAlu ma(SrcA, SrcB, ALUControl[2:0],
                res[0], flags[0]);
    ShiftRotate sr(SrcA, SrcB, ALUControl[2:0],
                res[1], flags[1]);

    mux2 #(4)   FlagsMux(flags[0], flags[1], ALUControl[3], ALUFlags);
    mux2 #(8)   ResMux(res[0], res[1], ALUControl[3], ALUResult);

endmodule

module mainAlu(input logic [7:0]    A, B,
                input logic [2:0]   control,
                output logic [7:0]  res,
                output logic [3:0]  flags);

    logic cf, zf, sf, of;

    always_comb
        casex(control)
            3'b000: {cf, res} = {1'b0, A} + {1'b0, B};
            3'b001: {cf, res} = {1'b0, A & B};
            3'b010: {cf, res} = {1'b0, A} - {1'b0, B};
            3'b011: {cf, res} = {1'b0, A | B};
            3'b100: {cf, res} = {1'b0, A ^ B};
            3'b101: {cf, res} = {1'bx, B};
            3'b110: {cf, res} = {1'bx, B};
            3'b111: {cf, res} = {1'bx, ~A};
            default: {cf, res} = 9'bx;
        endcase
    assign of = (~A[7] & ~B[7] & res[7]) | (A[7] & B[7] & ~res[7]);
    assign zf = (res === 0);
    assign sf = res[7];

    assign flags = {cf, zf, sf, of};

endmodule
```

عملکرد:

تمامی عملیات منطقی مانند ضرب و جمع و شیفت در این بخش محاسبه می شوند.

از 4 بیت برای کنترل آن استفاده می‌شود.

ماژول ShiftRotate

```
module ShiftRotate( input logic [7:0]   A, B,
                    input logic [2:0]   control,
                    output logic [7:0]  SRRes,
                    output logic [3:0]  SRFlags);

   logic Select;
   logic [7:0] res[1:0];
   logic [3:0] flags[1:0];

   assign Select = control[2];

   Shift   s(A, B, control[1:0], res[0], flags[0]);
   Rotate r(A, B, control, res[1], flags[1]);

   mux2 #(4)   FlagsMux(flags[0], flags[1], Select, SRFlags);
   mux2 #(8)   ResMux(res[0], res[1], Select, SRRes);

endmodule

module Shift(   input logic [7:0]   A, B,
                input logic [1:0] control,
                output logic [7:0] res,
                output logic [3:0] flags);

   logic cf, zf, sf, of;

   always_comb
      casex(control)
         2'b00: {res, cf} = {A, 1'b0} >>> B;
         2'b01: {res, cf} = {A, 1'b0} >> B;
         2'b10: {cf, res} = {1'b0, A} <<< B;
         2'b11: {cf, res} = {1'b0, A} << B;
         default: {cf, res} = 9'bx;
      endcase

   assign of = ~(A[7] === res[7]);
   assign zf = (res === 0);
   assign sf = res[7];

   assign flags = {cf, zf, sf, of};
endmodule

module Rotate(input logic [7:0] A, B,
```

```
            input logic [2:0] control,
            output logic [7:0] res,
            output logic [3:0] flags);

  logic cf, zf, sf, of;

  always_comb
    if(control[0]) begin
      casex(B[2:0])
        3'b000: res = A;
        3'b001: res = {A[0], A[7:1]};
        3'b010: res = {A[1:0], A[7:2]};
        3'b011: res = {A[2:0], A[7:3]};
        3'b100: res = {A[3:0], A[7:4]};
        3'b101: res = {A[4:0], A[7:5]};
        3'b110: res = {A[5:0], A[7:6]};
        3'b111: res = {A[6:0], A[7]};
        default: res = 8'bx;
      endcase
    end else begin
      casex(B[2:0])
        3'b000: res = A;
        3'b001: res = {A[6:0], A[7]};
        3'b010: res = {A[5:0], A[7:6]};
        3'b011: res = {A[4:0], A[7:5]};
        3'b100: res = {A[3:0], A[7:4]};
        3'b101: res = {A[2:0], A[7:3]};
        3'b110: res = {A[1:0], A[7:2]};
        3'b111: res = {A[0], A[7:1]};
        default: res = 8'bx;
      endcase
    end

  assign cf = 1'b0;
  assign of = 1'b0;
  assign zf = (res === 0);
  assign sf = res[7];

  assign flags = {cf, zf, sf, of};
endmodule
```

عملکرد:

ماژول شیفت همان‌طور که از اسمش پیداست وظیفه شیفت دادن را برعهده دارد.

از متغیر control هم برای تعیین نوع شیفت استفاده می‌شود.

نهایتا داده های جدید و فلگ های جدید را خروجی می دهند.

ماژول روتیت هم به مانند ماژول شیفت است با این تفاوت که عملیات مخصوص به خود را انجام می دهد.

ماژول شیفت روتیت هم ترکیبی از هر دو است.

ماژول mux2

```
module mux2 #(parameter WIDTH = 8)

          (  input logic [WIDTH-1:0] d0, d1,
               input logic s,
               output logic [WIDTH-1:0] y);

   assign y = s ? d1 : d0;

endmodule
```

ماژول adder

```
module adder #(parameter WIDTH=8)
          (input logic [WIDTH-1:0] a, b,
           output logic [WIDTH-1:0] y);
  assign y = a + b;
endmodule
```

ماژول flopr, flopenr

```
           module flopr #(parameter WIDTH = 8)

          (input logic clk, reset,
           input logic [WIDTH-1:0] d,
           output logic [WIDTH-1:0] q);
always_ff @(posedge clk, posedge reset)
   if (reset) q <= 0;
   else q <= d;
endmodule
```

```
module flopenr #(parameter WIDTH = 8)
            (input logic   clk, reset, en,
             input logic [WIDTH-1:0] d,
             output logic [WIDTH-1:0] q);
always_ff @(posedge clk, posedge reset)
   if (reset) q <= 0;
   else if (en) q <= d;
endmodule
```

عملکرد:

ورودی‌ها:

خروجی‌ها:

ماژول controller

```
module controller(  input logic          clk, reset,
                    input logic [15:0]  Instr,
                    input logic [3:0]     ALUFlags,
                    output logic          RegWriteA, RegWriteB, ImmSrc,
                                          IDmux, JMux,
                                          MemtoReg,
                    output logic [1:0]  Show,
                    output logic [3:0]  ALUControl);
  logic      FlagW;
  logic      RegWA, RegWB, JM;

  decoder dec(    Instr,
                  RegWA, RegWB, ImmSrc,
                  IDmux, JM,
                  MemtoReg, Show, FlagW,
                  ALUControl);
  condlogic cl(   clk, reset,
                  Instr, ALUFlags, FlagW,
                  RegWA, RegWB, JM,
                  RegWriteA, RegWriteB, JMux);
endmodule
```

عملکرد:

بخش کنترل کننده سیگنالهایی که به datapath وارد می‌شوند.

ماژول decoder

```
module decoder( input logic [15:0]  Instr,
                output logic        RegWA, RegWB, ImmSrc,
                                    IDMux, JM, MemtoReg,
                output logic [1:0]  Show,
                output logic        FlagW,
                output logic [3:0]  ALUControl);

  logic [7:0] controls;

  // Main Decoder
  always_comb
    casex(Instr[15:6])
      10'b0000000001: controls = 8'b10000000;
      10'b0000000010: controls = 8'b10000000;
      10'b0000000011: controls = 8'b10000000;
      10'b0000000100: controls = 8'b10000000;
      10'b0000000101: controls = 8'b10000000;
      10'b0000000110: controls = 8'b10000000;
      10'b0000000111: controls = 8'b11000000;
      10'b0000001000: controls = 8'b10000000;
      10'b0000001001: controls = 8'b10100000;
      10'b0000001010: controls = 8'b10100000;
      10'b0000001011: controls = 8'b10100000;
      10'b0000001100: controls = 8'b10100000;
      10'b0000001101: controls = 8'b10100000;
      10'b0000001110: controls = 8'b10100000;
      10'b0000001111: controls = 8'b10110000;
      10'b0000010000: controls = 8'b10110000;
      10'b0000000000: controls = 8'b00000000;
      10'b0000010010: controls = 8'b00000001;
      10'b0000010011: controls = 8'b00000010;
      10'b0000010100: controls = 8'b00000000;
      10'b10000?????: controls = 8'b00100100;
      10'b10001?????: controls = 8'b00100100;
      10'b10010?????: controls = 8'b00100100;
      10'b10011?????: controls = 8'b00100100;
      10'b10100?????: controls = 8'b00100100;
      10'b10101?????: controls = 8'b00100100;
      10'b10110?????: controls = 8'b10100000;
      10'b10111?????: controls = 8'b10101000;
      default:        controls = 8'bx;
    endcase
```

```
assign {    RegWA, RegWB, ImmSrc, IDMux,
          MemtoReg,JM,Show} = controls;

always_comb
   if(Instr[15]) begin
      ALUControl = 4'b0101;
   end else begin
      casex(Instr[10:6])
         5'b00000: ALUControl = 4'b0000;
         5'b00001: ALUControl = 4'b0000;
         5'b00010: ALUControl = 4'b0001;
         5'b00011: ALUControl = 4'b0010;
         5'b00100: ALUControl = 4'b0011;
         5'b00101: ALUControl = 4'b0100;
         5'b00110: ALUControl = 4'b0101;
         5'b00111: ALUControl = 4'b0110;
         5'b01000: ALUControl = 4'b0111;
         5'b01001: ALUControl = 4'b1000;
         5'b01010: ALUControl = 4'b1001;
         5'b01011: ALUControl = 4'b1010;
         5'b01100: ALUControl = 4'b1011;
         5'b01101: ALUControl = 4'b1100;
         5'b01110: ALUControl = 4'b1101;
         5'b01111: ALUControl = 4'b0000;
         5'b10000: ALUControl = 4'b0010;
         5'b10100: ALUControl = 4'b0010;
      default:    ALUControl = 4'b0000;
      endcase
   end

always_comb
   casex(Instr[15:6])
      10'b0000000001: FlagW = 1'b1;
      10'b0000000010: FlagW = 1'b1;
      10'b0000000011: FlagW = 1'b1;
      10'b0000000100: FlagW = 1'b1;
      10'b0000000101: FlagW = 1'b1;
      10'b0000000110: FlagW = 1'b0;
      10'b0000000111: FlagW = 1'b0;
      10'b0000001000: FlagW = 1'b0;
      10'b0000001001: FlagW = 1'b1;
      10'b0000001010: FlagW = 1'b1;
      10'b0000001011: FlagW = 1'b1;
      10'b0000001100: FlagW = 1'b1;
      10'b0000001101: FlagW = 1'b1;
```

```
        10'b0000001110: FlagW = 1'b1;
        10'b0000001111: FlagW = 1'b1;
        10'b0000010000: FlagW = 1'b1;
        10'b0000000000: FlagW = 1'b0;
        10'b0000010010: FlagW = 1'b0;
        10'b0000010011: FlagW = 1'b0;
        10'b0000010100: FlagW = 1'b1;
        default:        FlagW = 1'bx;
    endcase
endmodule
```

ماژول condlogic

```
module condlogic(  input              clk, reset,
                   input logic [15:0]  Instr,
                   input logic [3:0]   ALUFlags,
                   input logic         FlagW,
                   input logic         RegWA, RegWB, JM,
                   output logic        RegWriteA, RegWriteB, JMux);

  logic      FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(4)flagreg1(clk, reset, FlagWrite,
                       ALUFlags, Flags);

  condcheck cc(Instr, Flags, CondEx);

  assign FlagWrite = FlagW & CondEx;
  assign RegWriteA = RegWA & CondEx;
  assign RegWriteB = RegWB & CondEx;
  assign JMux = JM & CondEx;
endmodule
```

ماژول condcheck

```
module condcheck(   input logic [15:0]  Instr,
                    input logic [3:0]     Flags,
                    output logic          CondEx);

   logic CF, ZF, SF, OF;

   assign {CF, ZF, SF, OF} = Flags;

   always_comb
      casex(Instr[15:11])
         5'b0????: CondEx = 1'b1;
         5'b10000: CondEx = ZF;
         5'b10001: CondEx = CF;
         5'b10010: CondEx = ~(CF | ZF);
         5'b10011: CondEx = (SF !== OF);
         5'b10100: CondEx = (SF === OF) & ~ZF;
         5'b10101: CondEx = 1'b1;
         5'b10110: CondEx = 1'b1;
         5'b10111: CondEx = 1'b1;
         default:        CondEx = 1'bx;
      endcase
endmodule
```

ماژول BCD

```
module BCD( input      en,
            input [7:0] ShowData);
   always@(en)
      $display("%b",ShowData);
endmodule
```

ماژول SevenSeg

```
module SevenSeg(    input      en,
                    input [7:0] ShowData);
   always@ (en)
      $display("%d",ShowData);
endmodule
```

تست

```
LI R0 #3 //R0 = 3
LI R1 #2 //R1 = 2
LI R2 #1 //R2 = 1
LI R3 #4 //R3 = 4
OR R1 R0 //R1 = 3
XOR R2 R3 //R2 = 5
MOV R5 R2 //R5 = 5
ADD R1 R2 //R1 = 8
SUB R2 R1 //R2 = -3
XCHG R5 R0 // R0 = 5 R5 = 3
NOT R2 // R2 = 11111101 R2 = 00000010 = 2
JL to start //not accept
SAR R0 #2 //R0 = 1
SLR R3 #1 //R3 = 2
SAL R3 #4 //R3 = 32
SLL R1 #3 //R1 = 64
DEC R0 #1 // R0 = 0
DEC R0 #1 // R0 = -1
INC R0 #1 // R0 = 0
NOP
CMP R0 R1 //R0-R1 = 0-64=-64
LI R4 #5 // R4 = 5
LI R6 #15 // R6 = 15
ShowR R4
JA to start //not accept
ShowR R6
AND R6 R4 // R6 = 5
ShowRseg R4
ShowRseg R6
CMP R6 R4
JE to JMP0 //accept so R7 doesn't affect
LI R7 #7
JMP to 0
```

اینستراکشن‌ها

```
B003
B102
B201
B304
0108
0113
```

```
01AA
004A
00D1
01E8
0217
9D00
0242
0299
02DC
030B
0405
0403
03C7
0033
0501
B405
B60F
9700
04A0
04B0
00B4
04E0
04F7
0534
8680
B707
AF00
```

همگی دستورات تست شده‌اند و درستی آن در ویدیو قابل مشاهده است.

نتیجه سیمیولیشن

آزمایشگاه معماری و مدارهای منطقی – پروژه چهارم و پنجم
متین زیودار – محمد خدام