RISC-V multi cycle processor

Mohammad Vatankhah Ardakani

Arman Ghafarnia

Computer Architecture final project

Spring 2022

Overview

در این پروژه ما قصد داریم یک پردازنده چند سیکل RISC-V را به کمک زبان Systemverilog پیاده سازی کنیم.

Control Unit

واحد کنترل پردازنده‌ی چند سیکل $RISC\text{-}V$ شامل بخش‌های زیر می‌باشد:

- $Main\ FSM$
- $ALU\ Decoder$
- $Instruction\ Decoder$

Main FSM

کد مربوط به $Main\ FSM$ به صورت زیر می‌باشد.
ورودی‌ها و خروجی‌ها:

```
module main_FSM(input  logic       clk,
                input  logic       rst,
                input  logic [6:0] op,
                output logic       branch,
                output logic       pcupdate,
                output logic       regwrite,
                output logic       memwrite,
                output logic       irwrite,
                output logic [1:0] resultsrc,
                output logic [1:0] alusrca,
                output logic [1:0] alusrcb,
                output logic       adrsrc,
                output logic [1:0] aluop);
```

برای مشخص کردن $state$ها، $current\_state,\ next\_state$ را به‌صورت زیر تعریف کرده‌ایم که هر کدام به شکل یک عدد 4 بیتی تعریف شده‌اند.

```
logic [3:0] current_state, next_state;
```

در یک بلوک $always$ با توجه به $reset$، استیت بعدی را مشخص می‌کنیم.

```
always_ff @(posedge clk, posedge rst)
   if(rst)
      current_state <= 0;
   else
      current_state <= next_state;
```

با توجه به شکل 7.45 مرجع هریس، استیت‌های بعدی را با توجه به $op\ code$ مشخص میکنیم.

```
always_comb
   begin
      case(current_state)
         0:
         begin
            next_state = 1;
         end
         1:
         begin
            if     (op == 7'b0000011 | op == 7'b0100011) next_state = 2;
            else if(op == 7'b0110011)                    next_state = 6;
            else if(op == 7'b0010011)                    next_state = 8;
            else if(op == 7'b1101111)                    next_state = 9;
            else if(op == 7'b1100011)                    next_state = 10;
            else                                         next_state = 2;
         end
         2:
         begin
            if     (op == 7'b0000011) next_state = 3;
            else if(op == 7'b0100011) next_state = 5;
            else                      next_state = 2;
         end
         3:
         begin
            next_state = 4;
         end
         4:
         begin
            next_state = 0;
         end
         5:
         begin
            next_state = 0;
         end
         6:
         begin
            next_state = 7;
         end
         7:
         begin
            next_state = 0;
         end
         8:
         begin
            next_state = 7;
         end
         9:
         begin
            next_state = 7;
         end
         10:
         begin
            next_state = 0;
         end
         default:
         begin
            next_state = 0;
         end
      endcase
   end
```

در ادامه نیز با توجه به شکل 7.45 مرجع هریس، مقادیر سیگنال‌های کنترلی در هر استیت را مشخص می‌کنیم.

```
always_comb
begin
   case(current_state)
0:
   begin
      adrsrc = 0;
      irwrite = 1;
      alusrca = 2'b00;
      alusrcb = 2'b10;
      aluop = 2'b00;
      resultsrc = 2'b10;
      pcupdate = 1;
      regwrite = 0;
      memwrite = 0;
      branch = 0;
   end
1:
   begin
      alusrca = 2'b01;
      alusrcb = 2'b01;
      aluop = 2'b00;
      regwrite = 0;
      memwrite = 0;
      irwrite = 0;
      resultsrc = 0;
      adrsrc = 0;
      pcupdate = 0;
      branch = 0;
   end
2:
   begin
      alusrca = 2'b10;
      alusrcb = 2'b01;
      aluop = 2'b00;
      regwrite = 0;
      memwrite = 0;
      irwrite = 0;
     resultsrc = 0;
     adrsrc = 0;
     pcupdate = 0;
     branch = 0;
   end
3:
   begin
      resultsrc = 2'b00;
      adrsrc = 1'b1;
      regwrite = 0;
      memwrite = 0;
      irwrite = 0;
      alusrca = 0;
      alusrcb = 0;
      aluop = 0;
      pcupdate = 0;
      branch = 0;
   end
4:
   begin
      resultsrc = 2'b01;
      regwrite = 1;
      memwrite = 0;
      irwrite = 0;
      alusrca = 0;
      alusrcb = 0;
      adrsrc = 0;
      aluop = 0;
      pcupdate = 0;
      branch = 0;
   end
5:
   begin
      resultsrc = 2'b00;
      adrsrc = 1;
      memwrite = 1;
```

```
    regwrite = 0;
    irwrite = 0;
    alusrca = 0;
    alusrcb = 0;
    aluop = 0;
    pcupdate = 0;
    branch = 0;
end
6:
    begin
        alusrca = 2'b10;
        alusrcb = 2'b00;
        aluop = 2'b10;
        regwrite = 0;
        memwrite = 0;
        irwrite = 0;
        resultsrc = 0;
        adrsrc = 0;
        pcupdate = 0;
        branch = 0;
    end
7:
    begin
        resultsrc = 2'b00;
        regwrite = 1;
        memwrite = 0;
        irwrite = 0;
        alusrca = 0;
        alusrcb = 0;
        adrsrc = 0;
        aluop = 0;
        pcupdate = 0;
        branch = 0;
    end
8:
    begin
        alusrca = 2'b10;
        alusrcb = 2'b01;
        aluop = 2'b10;
        regwrite = 0;
        memwrite = 0;
        irwrite = 0;
        resultsrc = 0;
        adrsrc = 0;
        pcupdate = 0;
        branch = 0;
    end
9:
    begin
        alusrca = 2'b01;
        alusrcb = 2'b10;
        aluop = 2'b00;
        resultsrc = 2'b00;
        pcupdate = 1;
        regwrite = 0;
        memwrite = 0;
        irwrite = 0;
        adrsrc = 0;
        branch = 0;
    end
10:
    begin
        alusrca = 2'b10;
        alusrcb = 2'b00;
        aluop = 2'b01;
        resultsrc = 2'b00;
        branch = 1;
        regwrite = 0;
        memwrite = 0;
        irwrite = 0;
        adrsrc = 0;
        pcupdate = 0;
        branch = 1;
    end
```

ALU Decoder

کد مربوط به $ALU\ Decoder$ به صورت زیر می‌باشد.

```
module alu_dec(input  logic [1:0] aluop,
               input  logic       op5,
               input  logic [2:0] func3,
               input  logic       func7b5,
               output logic [2:0] alucontrol);
   logic RtypeSub;
   assign RtypeSub = func7b5 & op5; // TRUE for R-type subtract
   always_comb
      case(aluop)
         2'b00:
         alucontrol = 3'b000; // addition
         2'b01:
         alucontrol = 3'b001; // subtraction
         default:
            case(func3) // R-type or I-type ALU
               3'b000:
                  if (RtypeSub)
                     alucontrol = 3'b001; // sub
                  else
                     alucontrol = 3'b000; // add, addi
               3'b010:
                  alucontrol = 3'b101; // slt, slti
               3'b110:
                  alucontrol = 3'b011; // or, ori
               3'b111:
                  alucontrol = 3'b010; // and, andi
               default:
                  alucontrol = 3'bxxx;
            endcase
      endcase
   endmodule
```

در این ماژول، عملیاتی که باید توسط ALU انجام بشود، با توجه به $ALU\ op,\ op[5],\ func3$ $func7[5]$ و جدول 2 ارائه شده در صورت پروژه تعیین می‌شود.

Instruction Decoder

کد مربوط به $Instruction\ Decoder$ به صورت زیر می‌باشد.

```
module instr_dec(input  logic [6:0] op,
                 output logic [1:0] immsrc);

   always_comb
      case(op)
         7'b0110011:
            immsrc = 2'bxx; // R-Type
         7'b0010011:
            immsrc = 2'b00; // I-Type
         7'b0000011:
            immsrc = 2'b00; // lw
         7'b0100011:
            immsrc = 2'b01; // sw
         7'b1100011:
            immsrc = 2'b10; // beq
         7'b1101111:
            immsrc = 2'b11; // jal
         default:
            immsrc = 2'bxx;
      endcase
endmodule
```

در این ماژول با توجه به جدول 3 ارائه شده در صورت پروژه، 2بیت کنترل کننده‌ی واحد $extend$ با توجه به $op\ code$ تعیین می‌شوند.
حالا باید بخش های بالا که برای واحد کنترل طراحی شده‌بود را به یکدیگر متصل کنیم.

Controller

کد مربوط به ماژول $controller$ به صورت زیر است.

```
odule controller(input  logic       clk,
                 input  logic       reset,
                 input  logic [6:0] op,
                 input  logic [2:0] func3,
                 input  logic       func7b5,
                 input  logic       zero,
                 output logic [1:0] immsrc,
                 output logic [1:0] alusrca,
                 output logic [1:0] alusrcb,
                 output logic [1:0] resultsrc,
                 output logic       adrsrc,
                 output logic [2:0] alucontrol,
                 output logic       irwrite,
                 output logic       pcwrite,
                 output logic       regwrite,
                 output logic       memwrite);

    logic pcupdate;
    logic branch;
    logic [1:0] aluop ;
    assign pcwrite = ((zero&branch)|pcupdate);
    alu_dec aludec(aluop, op[5], func3, func7b5, alucontrol);
    main_FSM fsm(clk, reset, op, branch, pcupdate, regwrite, memwrite, irwrite, resultsrc, alusrca, alusrcb, adrsrc, aluop);
    instr_dec instrdec(op, immsrc);
ndmodule
```

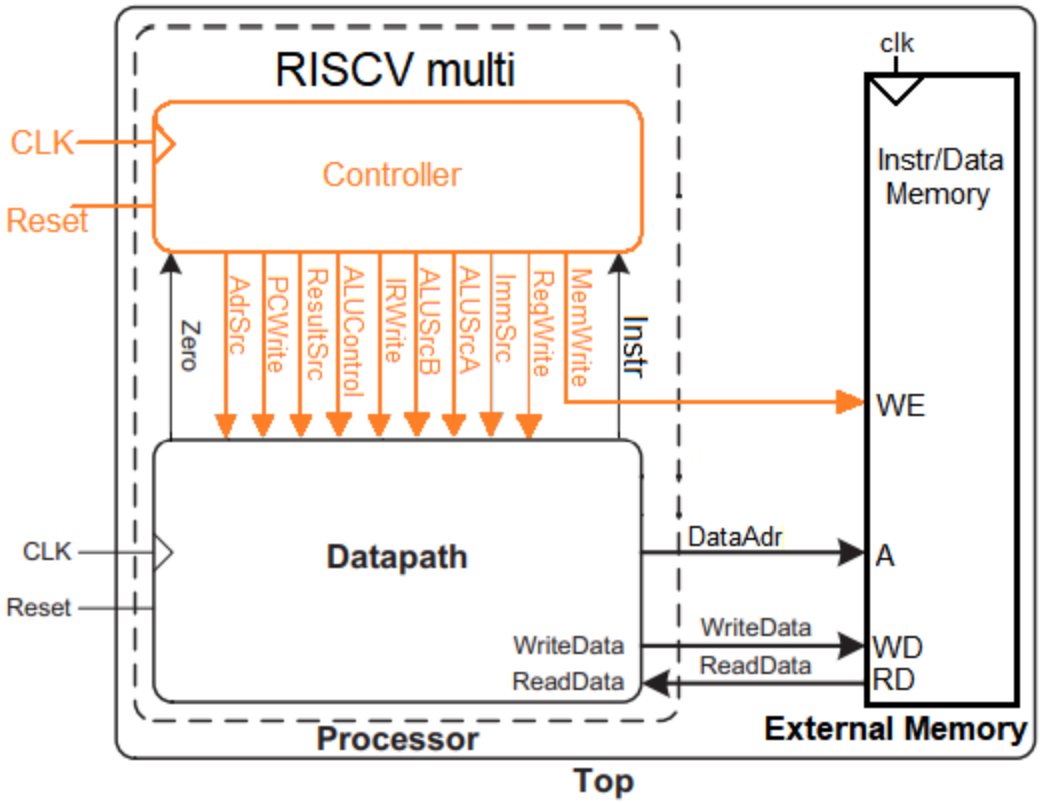
در این ماژول از بخش‌های مختلف واحد کنترل $instance$ گرفته شده‌است و ورودی‌ها و خروجی‌های مورد نیاز تولید شده‌است.
همانطور که در کد مشخص است، $PCWrite$ با توجه به منطق زیر مقدار دهی شده‌است.

در تصویر زیر مشاهده می‌شود که تست بنچ مربوط به کنترلر با موفقیت انجام شده است.

```
#         47 tests completed with          0 errors
# hash = 39
# ** Note: $stop    : F:/SBU/term4/CA/project/p/controller.sv(135)
#    Time: 485 ps  Iteration: 1  Instance: /testbench
# Break in Module testbench at F:/SBU/term4/CA/project/p/controller.sv line 135
```

Multi-cycle processor interfaced to external memory

در تصویر زیر سیگنال‌هایی که باید توسط واحد کنترل و $data\ path$ و $external\ memory$ به یکدیگر داده شوند، مشخص شده‌اند.



همانطور که در شکل بالا مشخص شده است، ما باید در ماژول $risc\ v\ multi$ از دو ماژول $controller, Datapath$ یک $instance$ گرفته شود و اتصالات مناسب را ایجاد کنیم وسیگنال‌های کنترلی تولید شده در ماژول کنترلر را به $Datapath$ ارسال کنیم.

RISC-V multi module

کد ماژول $risc\ v\ multi$ به صورت زیر می‌باشد.

```
module riscvmulti(input  logic        clk, reset,
                  output logic        MemWrite,
                  output logic [31:0] Adr, WriteData,
                  input  logic [31:0] ReadData);
      logic RegWrite, IRWrite, PCWrite, AdrSrc, Zero;
      logic [1:0] ALUSrcA, ALUSrcB, ImmSrc, ResultSrc;
      logic [2:0] ALUControl;
      logic [31:0] instr;

      controller c (clk, reset, instr[6:0], instr[14:12], instr[30], Zero, ImmSrc,
      ALUSrcA, ALUSrcB, ResultSrc, AdrSrc, ALUControl, IRWrite, PCWrite, RegWrite, MemWrite);

      datapath   dp(clk, reset, MemWrite, RegWrite, IRWrite, PCWrite, AdrSrc, ALUSrcA,
      ALUSrcB, ImmSrc, ResultSrc, ALUControl, ReadData, Zero, instr, Adr, WriteData);
endmodule
```

در ابتدای کد، اتصالات بین دو واحد کنترل و $datapath$ را با استفاده از $logic$ تعریف کرده‌ایم.

Top module

در این ماژول ما باید اتصال بین 2 ماژول $riscvmulti$ که شامل 2 ماژول $controller$, $datapath$ می‌باشد و ماژول mem را برقرار کنیم.

کد ماژول top به صورت زیر است.

```
module top(input  logic        clk, reset,
           output logic [31:0] WriteData, DataAdr,
           output logic        MemWrite);

  logic [31:0] ReadData;

  // instantiate processor and memories
  riscvmulti rvmulti(clk, reset, MemWrite, DataAdr,
                     WriteData, ReadData);
  mem mem(clk, MemWrite, DataAdr, WriteData, ReadData);
endmodule
```

در $datapath$ باید بلوک‌های منطقی موجود در پردازنده چند سیکل و اتصالات بین آنها مشخص شود. در پردازنده چند سیکل $RISC\text{-}V$ به بلوک‌های منطقی مختلفی نیاز داریم. در پایین کد توصیف رفتاری آنها و توضیحی کوتاه درباره نحوه کار کردن آنها ارائه شده است.

ALU

```
module alu(input  logic [31:0] a,
           input  logic [31:0] b,
           input  logic [2:0]  control,
           output logic [31:0] out,
           output logic zero);
      logic [31:0]tmp;
      always @(a, b, control)
        begin
           if (control == 3'b010)     // And
              out = a & b;
           else if( control == 3'b011)    // Or
              out = a | b;
           else if( control == 3'b000)    // Add
              out = a + b;
           else if( control == 3'b001)    // SUB
              out = a - b;
           else if( control == 3'b101)    // SLT
           begin
              tmp = a - b;
              out[31:1] = 31'h0;
              out[0] = (tmp[31] ==  1'b1);
           end
           if (out == 32'h00000000)
              zero = 1;
           else
              zero = 0;
      end
endmodule
```

ALU دو عدد 32 بیتی a, b و یک عدد 3 بیتی به عنوان کنترلر دریافت می‌کند و عملیات مورد نظر که با 3 بیت کنترلی مشخص می‌شود را روی a, b انجام می‌دهد. همچنین اگر حاصل عملیات انجام شده روی a, b برابر 0 باشد، خروجی $Zero$ موجود در این ماژول، برابر 1 خواهد شد.

Extend

```
module extend(input logic [31:7] instr,
              input logic [1:0] immsrc,
              output logic [31:0] immext);

   always_comb

      case(immsrc)
         // I-type
         3'b00: immext = {{20{instr[31]}}, instr[31:20]};
         // S-type (stores)
         3'b01: immext = {{20{instr[31]}}, instr[31:25], instr[11:7]};
         // B-type (branches)
         3'b10: immext = {{20{instr[31]}}, instr[7], instr[30:25], instr[11:8], 1'b0};
         // J-type (jal)
         3'b11: immext = {{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0};
         default: immext = 32'bx; // undefined
      endcase
endmodule
```

در واحد $Extend$ نحوه‌ی استخراج imm از $instruction$ با توجه به سیگنال کنترلی $immsrc$ مشخص می‌شود.

Flopr

```
module flopr #(parameter WIDTH = 8)
              (input  logic             clk,
               input  logic             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
```

$flopr$ یک رجیستر است که یک عدد را در خود ذخیره می‌کند. همچنین یک بیت $reset$ وجود دارد که اگر برابر 1 باشد، عدد ذخیره شده در آن پاک می‌شود.

Flopenr

```
module flopenr #(parameter WIDTH = 8)
                (input  logic                clk,
                 input  logic                reset,
                 input  logic                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
```

مشابه $flopr$ است با این تفاوت که اگر بیت en برابر 1 باشد مقدار در رجیستر ذخیره می‌شود.

Mux2

```
module mux2(input  logic [31:0] d0,
            input  logic [31:0] d1,
            input  logic        s,
            output logic [31:0] y);
      assign y = s ? d1 : d0;
endmodule
```

دو ورودی دارد و با توجه به بیت کنترلی، یکی از آنها را انتخاب می‌کند و در خروجی قرار می‌دهد.

Mux3

```
module mux3(input  logic [31:0] d0,
            input  logic [31:0] d1,
            input  logic [31:0] d2,
            input  logic [1:0]  s,
            output logic [31:0] y);
      assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule
```

مانند $mux2$ است با این تفاوت که 3 ورودی دارد و با 2 بیت کنترلی خروجی آن انتخاب می‌شود.

RegFile

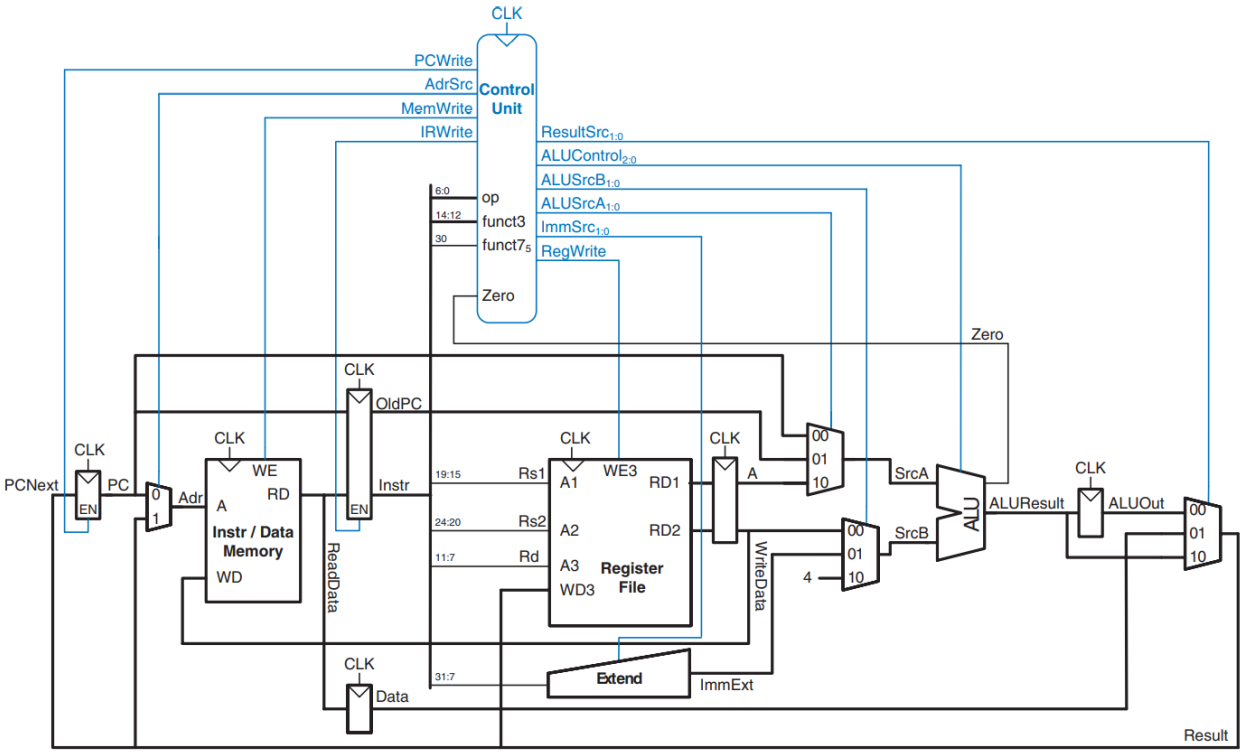
```
module reg_file(input  logic        clk,
                input  logic        we3,
                input  logic [4:0]  a1, a2, a3,
                input  logic [31:0] wd3,
                output logic [31:0] rd1, rd2);

   logic [31:0] rf[31:0];
   always_ff @(posedge clk)
      if (we3) rf[a3] <= wd3;
         assign rd1 = (a1 != 0) ? rf[a1] : 0;
         assign rd2 = (a2 != 0) ? rf[a2] : 0;
endmodule
```

رجیستر فایل شامل 32 رجیسترِ 32 بیتی است. ما می‌توانیم هربار مقدار 2 تا از رجیسترها را بخوانیم یا در یکی از رجیسترها مقداری بنویسیم.

Datapath

حالا باید با توجه به شکل 7.27 مرجع هریس باید این بلوک های منطقی را در ماژول $datapath$ به یکدیگر متصل کنیم.



همانطور که در شکل بالا مشاهده می‌شود، در پردازنده‌ی چند سیکل $RISC\text{-}V$، یک عددALU، 3عدد $flopr$، 2عدد $flopenr$، یک عدد$regfile$، 3عدد $mux3$، یک عدد $mux2$ و یک عدد $extend$ وجود دارد.

کد $datapath$ به صورت زیر می‌باشد.

ورودی‌ها و خروجی‌ها:

```
module datapath(input  logic        clk,
                input  logic        reset,
                input  logic        MemWrite,
                input  logic        RegWrite,
                input  logic        IRWrite,
                input  logic        PCWrite,
                input  logic        AdrSrc,
                input  logic [1:0]  ALUSrcA,
                input  logic [1:0]  ALUSrcB,
                input  logic [1:0]  ImmSrc,
                input  logic [1:0]  ResultSrc,
                input  logic [2:0]  ALUControl,
                input  logic [31:0] rd,
                output logic        Zero,
                output logic [31:0] instr,
                output logic [31:0] Adr,
                output logic [31:0] WriteData);
```

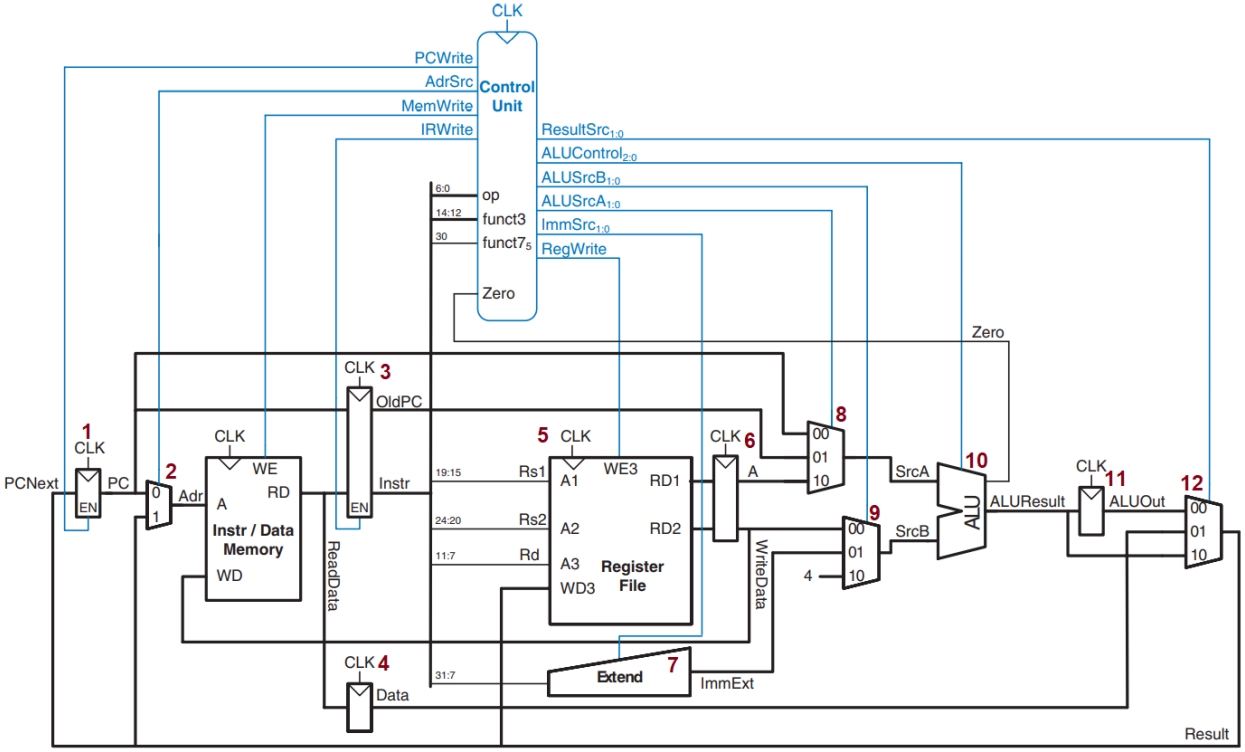
همانطور که مشاهده می‌شود، 13 ورودی داریم. 2 ورودی $clock, reset$ هستند. 10 ورودی سیگنال‌های کنترلی هستند که از واحد کنترل به $datapath$ می‌آیند. و یک ورودی، خروجی $memory$ می‌باشد.

حالا باید سیم های موجود شکل 7. 27 را با استفاده از $logic$ تعریف کنیم.

```
logic [31:0] Result;
logic [31:0] SrcA, SrcB;
logic [31:0] ALUOut;
logic [31:0] OldPC, A;
logic [31:0] PC;
logic [31:0] ImmExt;
logic [31:0] data;
logic [31:0] rd1, rd2;
logic [31:0] ALUResult;
```

حالا باید با استفاده از سیم های بالا، بلوک‌های منطقی که در قسمت قبل تعریف کردیم را به هم متصل کنیم.

با توجه به شکل زیر بلوک‌ها را تعریف می‌کنیم.



شماره‌های نسبت داده شده به هر یک از بلوک های بالا، به کد متناظر آنها در تصویر زیر نیز نسبت داده شده‌است.

```
1  flopenr #(32) PCReg(clk, reset, PCWrite,Result, PC);
2  mux2          PCmux(PC, Result, AdrSrc, Adr);
3  flopenr #(64) pc_instr(clk, reset, IRWrite, {rd, PC}, {instr, OldPC});
4  flopr   #(32) Data(clk, reset, rd, data);
5  reg_file      regfile(clk, RegWrite, instr[19:15], instr[24:20],
                         instr[11:7], Result, rd1, rd2);
6  flopr   #(64) wrA(clk, reset, {rd1, rd2}, {A, WriteData});
7  extend        ext(instr[31:7], ImmSrc, ImmExt);
8  mux3          a(PC, OldPC, A, ALUSrcA, SrcA);
9  mux3          b(WriteData, ImmExt, 32'd4, ALUSrcB, SrcB);
10 alu           ALU1(SrcA, SrcB, ALUControl, ALUResult, Zero);
11 flopr   #(32) aluflop(clk, reset, ALUResult, ALUOut);
12 mux3          resmux(ALUOut, data, ALUResult, ResultSrc, Result);
```

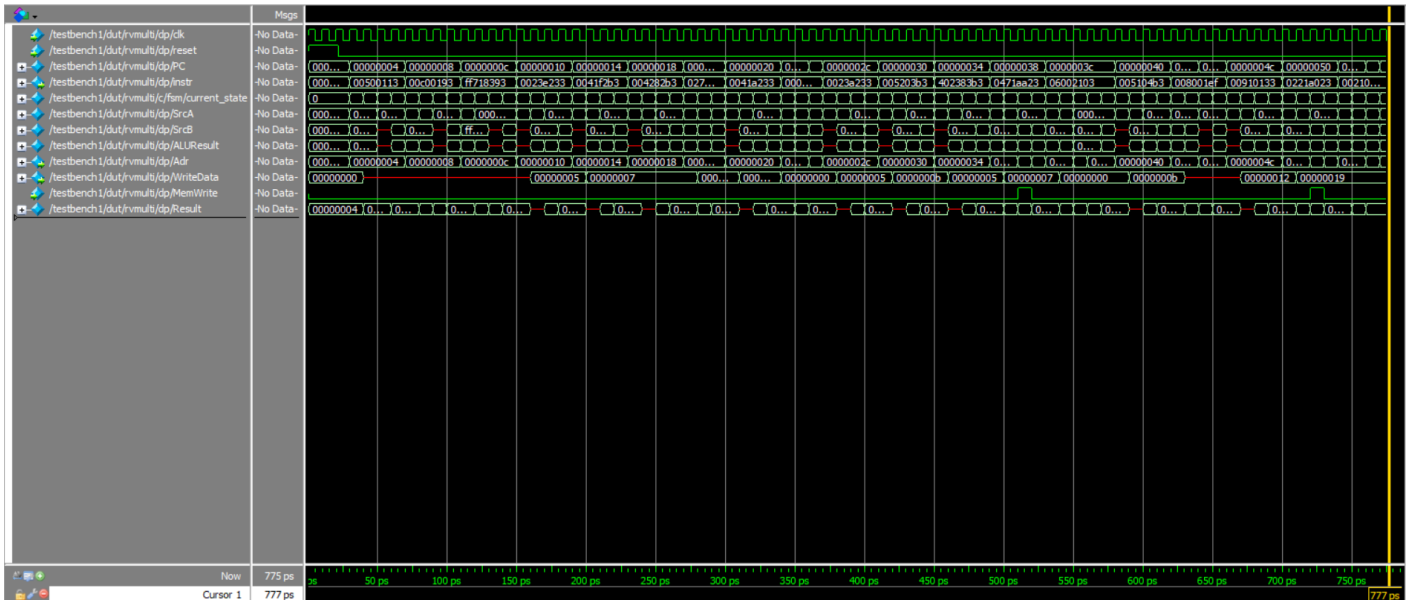
در کد بالا، بلوک‌های منطقی موجود در شکل 7. 27 تعریف شده‌اند و ورودی‌ها به آنها داده می‌شود و خروجی‌های مورد نیاز را به ما می‌دهند.

تصویر زیر نشان می‌دهد که تست بنچ مربوط به پردازنده با موفقیت انجام شده است.

```
# Simulation succeeded
# hash = ffe0b9e2
# ** Note: $stop    : F:/SBU/term4/CA/project/project final/top.sv(46)
#    Time: 725 ps  Iteration: 1  Instance: /testbench1
# Break in Module testbench1 at F:/SBU/term4/CA/project/project final/top.sv line 46
```

Signal analysis

نتیجه کلی تست بنچ وقتی مقدار همه‌ی سیگنال‌ها به صورت hex باشد به صورت زیر است.



حالا سیگنال‌های خروجی را تحلیل می‌کنیم.

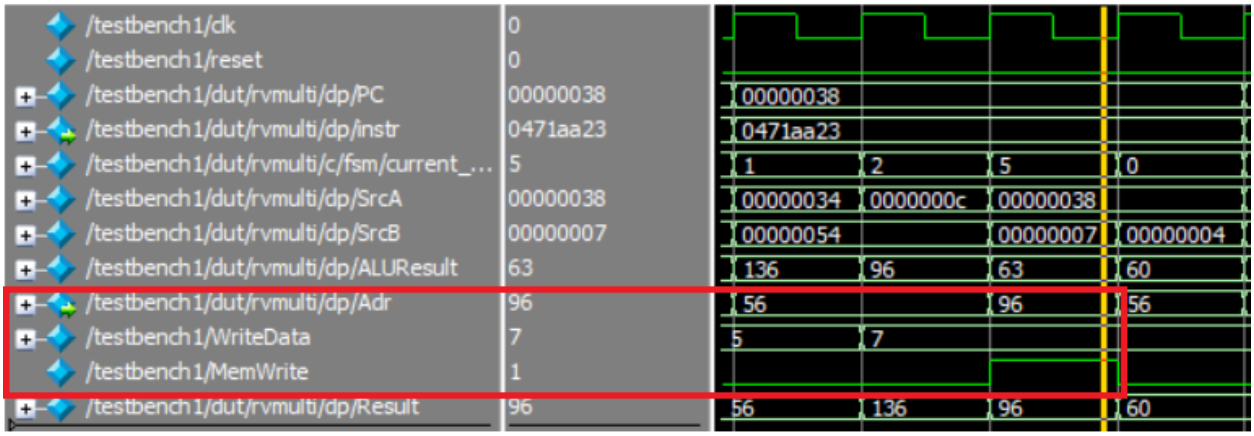
سیگنال‌هایی که نشان میدهند مقادیر درست در آدرس‌های درست نوشته شده‌است در شکل های زیر مشخص شده‌اند.

با اجرای کد خط زیر باید مقدار 7 در آدرس 96 موری ذخیره شود.

```
sw    x7, 84(x3)          # [96] = 7          34          0471AA23
```

در شکل زیر مشاهده می‌شود که $WriteData = 7$ است که یعنی مقداری که در موری نوشته خواهد شد، مقدار 7 است. همچنین در زمانی که $MemWrite = 1$ است، مقدار $Adr = 96$ است.

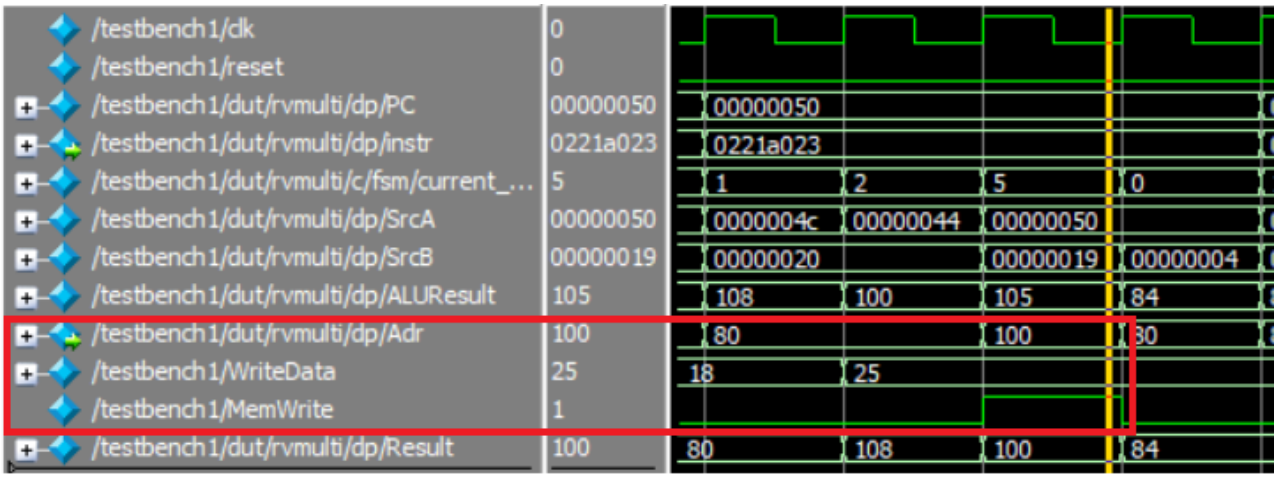
پس مقدار 7 در آدرس 96 موری نوشته خواهد شد.



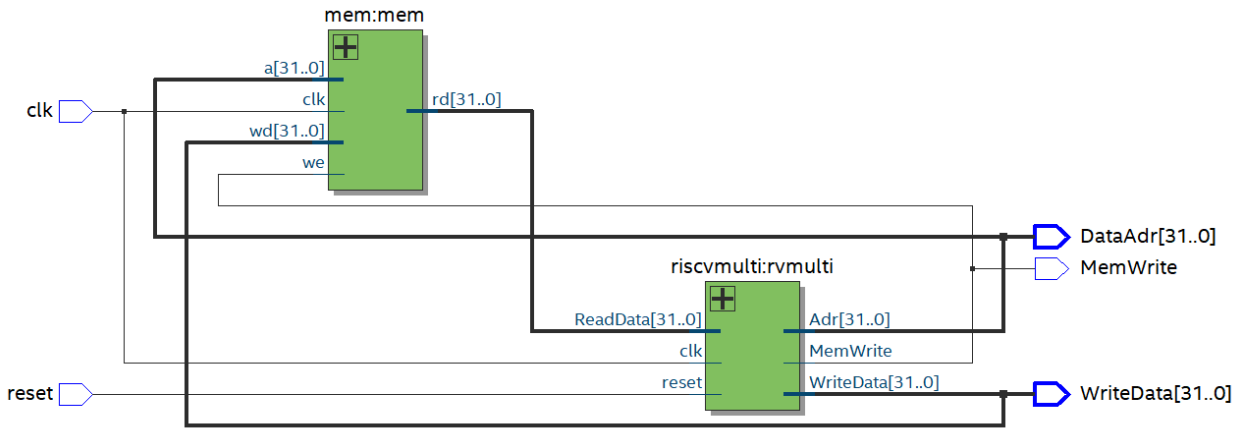
با اجرای کد خط زیر باید مقدار 25 در آدرس 100 موری ذخیره شود.

```
sw    x2, 0x20(x3)          # write mem[100] = 25    50          0221A023
```

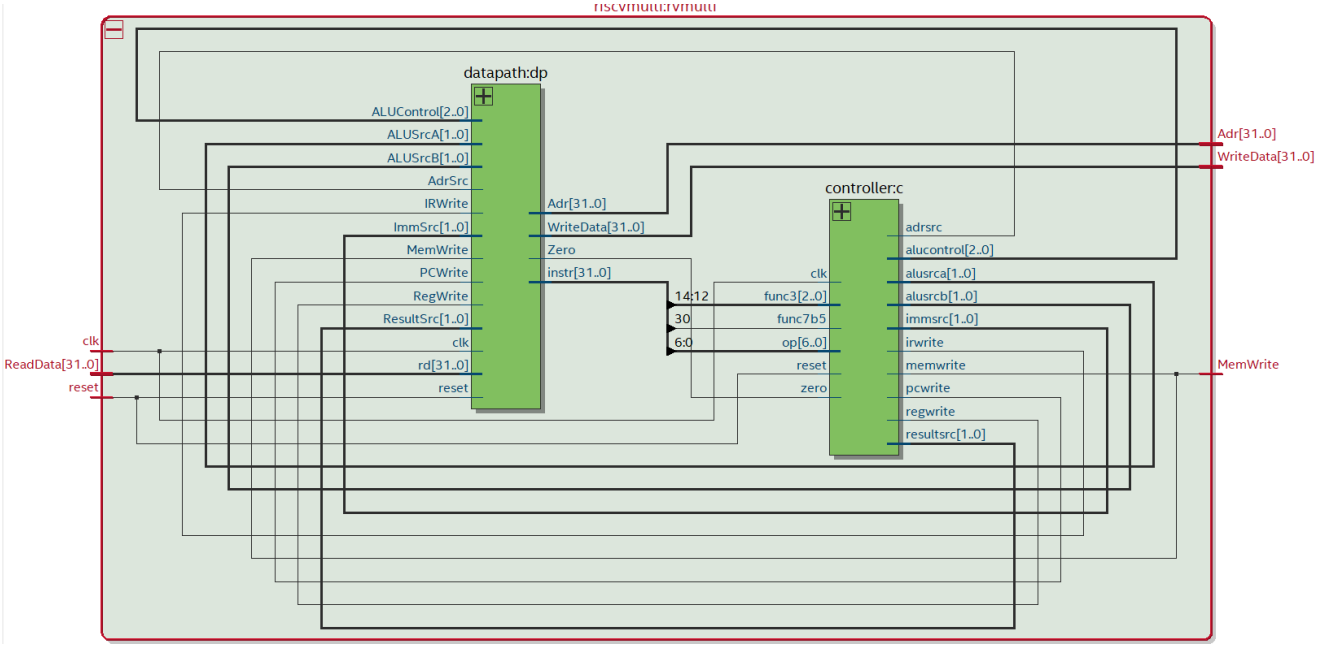
در شکل زیر مشاهده می‌شود که $WriteData = 25$ است که یعنی مقداری که در موری نوشته خواهد شد، مقدار 25 است. همچنین در زمانی که $MemWrite = 1$ است، مقدار $Adr = 100$ است. پس مقدار 25 در آدرس 100 موری نوشته خواهد شد.



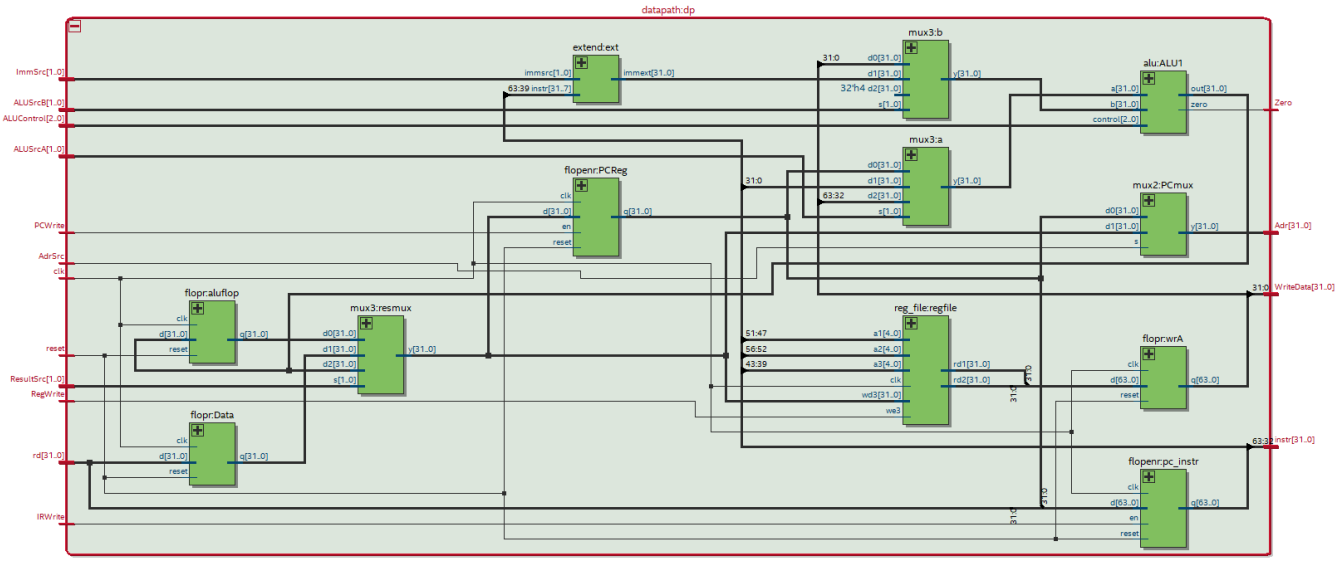
RTL Viewer



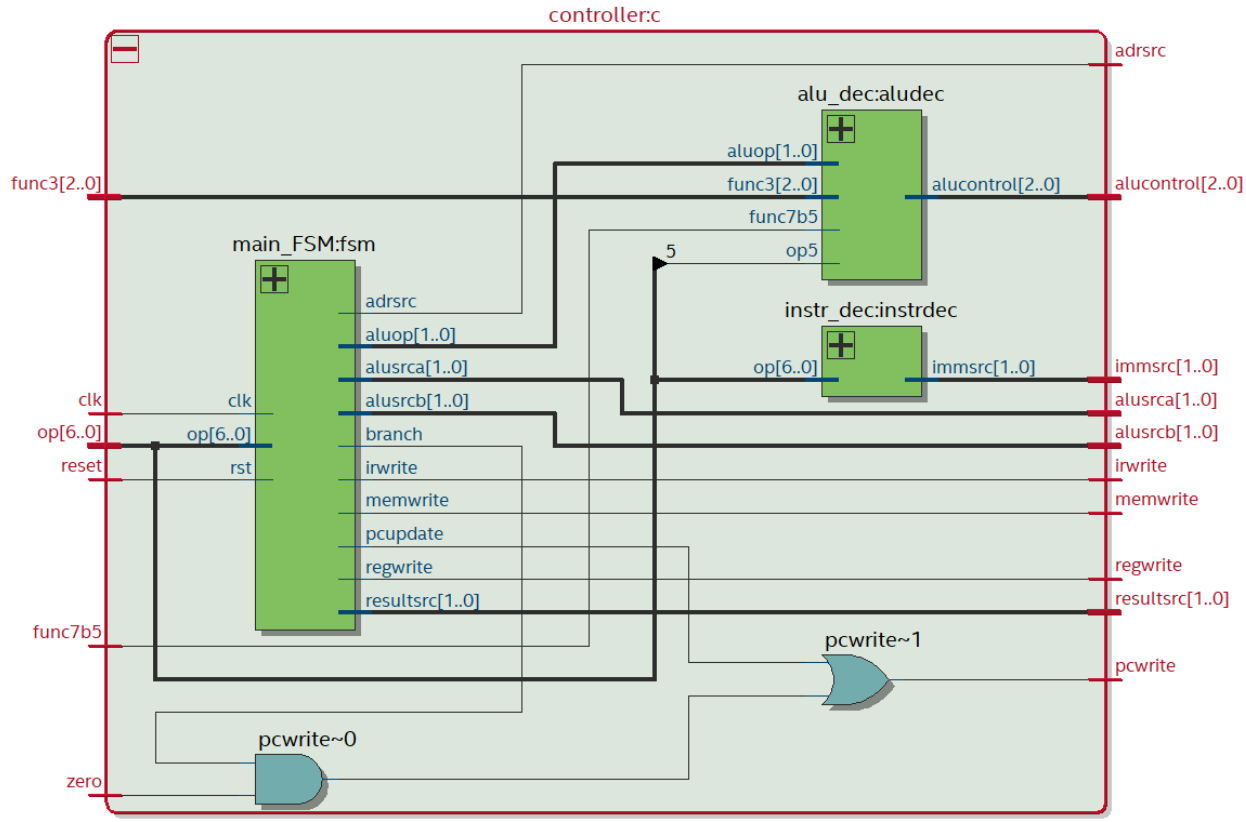
RTL Viewer : riscvmulti



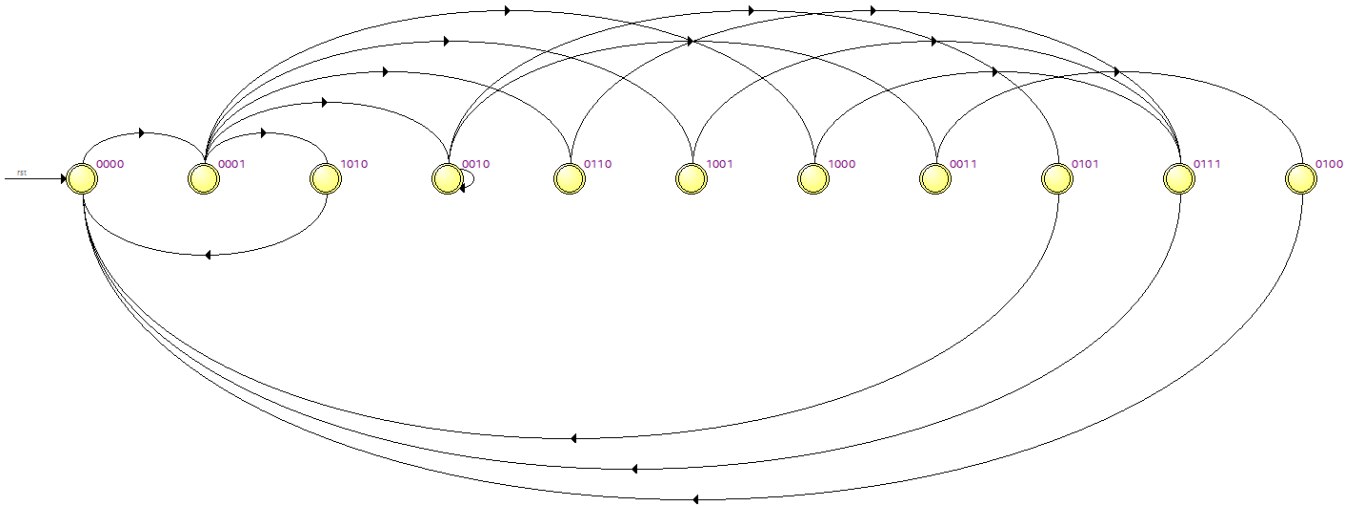
RTL Viewer : riscvmulti : datapath



RTL Viewer : riscvmulti : controller



State machine viewer



Flow summary

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sun Jul 03 13:16:36 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,156 / 114,480 (3 %)
Total registers	2435
Total pins	67 / 529 (13 %)
Total virtual pins	0
Total memory bits	2,048 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Timing report

Path Summary | Statistics | Data Path

	Property	Value
1	From Node	riscvmulti:rvmulti|controller:c|main_FSM:fsm|current_state.1010
2	To Node	riscvmulti:rvmulti|datapath:dp|alu:ALU1|out[20]
3	Delay	13.361

Table 1

Step	PC (hex)	Instr (hex)	State	Result	Result Notes
3	00	n/a	S0:fetch	4	PC+4
4	04	00500113	S1:Decode	X	OldPC+Imm
5	04	00500113	S8:ExecuteI	X	ALUResult = x0(0) + 5
6	04	00500113	S7:ALUWB	5	Result = ALUOUT
7	04	00500113	S0:Fetch	8	PC + 4
8	08	00C00193	S1:Decode	X	OldPC + Imm
9	08	00C00193	S8:ExecuteI	X	ALUResult = x0(0) + 12
10	08	00C00193	S7:ALUWB	12	Result = ALUOUT
11	08	00c00193	S0:Fetch	0x0C	PC + 4
12	0c	ff718393	S1:Decode	X	OldPC + Imm
13	0c	ff718393	S8:ExecuteI	X	ALUResult = x3(12) + (-9)
14	0c	ff718393	S7:ALUWB	3	Result = ALUOUT
15	0c	ff718393	S0:Fetch	0x10	PC + 4
16	10	0023e233	S1:Decode	X	OldPC + Imm
17	10	0023e233	S6:ExecuteR	X	ALUResult = x7(3) | x2(5)
18	10	0023e233	S7:ALUWB	7	Result = ALUOUT
19	10	0023e233	S0:Fetch	0x14	PC + 4
20	14	0041f2b3	S1:Decode	X	OldPC + Imm
21	14	0041f2b3	S6:ExecuteR	X	ALUResult = x3(12)&x4(7)

22	14	0041f2b3	S7:ALUWB	4	Result = ALUOUT
23	14	0041f2b3	S0:Fetch	0x18	PC + 4
24	18	004282b3	S1:Decode	X	OldPC + Imm
25	18	004282b3	S6:ExecuteR	X	ALUResult = x5(4)+x4(7)
26	18	004282b3	S7:ALUWB	11	Result = ALUOUT
27	18	004282b3	S0:Fetch	0x1C	PC + 4
28	1c	02728863	S1:Decode	X	OldPC + Imm
29	1c	02728863	S10:BEQ	X	ALUResult = x5(11)-x7(3) ≠ 0
30	1c	02728863	S0:Fetch	0x20	PC + 4
31	20	0041a233	S1:Decode	X	OldPC + Imm
32	20	0041a233	S6:ExecuteR	X	ALUResult = x3(12) < x4(7)
33	20	0041a233	S7:ALUWB	0	Result = ALUOUT
34	20	0041a233	S0:Fetch	0x24	PC + 4
35	24	00020463	S1:Decode	X	OldPC + Imm
36	24	00020463	S10:BEQ	X	ALUResult = x4(0) - x0(0) = 0
37	28	00020463	S0:Fetch	0x2C	PC + 4
38	2c	0023a233	S1:Decode	X	OldPC + Imm
39	2c	0023a233	S6:ExecuteR	X	ALUResult = (x7(3) < x2(5)) = 1
40	2c	0023a233	S7:ALUWB	1	Result = ALUOUT

41	2c	0023a233	S0:Fetch	0x30	PC+4
42	30	005203b3	S1:Decode	X	OldPC+Imm
43	30	005203b3	S6:ExecuteR	X	ALUResult = x4(1) + x5(11)
44	30	005203b3	S7:ALUWB	12	Result = ALUOUT
45	30	005203b3	S0:Fetch	0x34	PC + 4
46	34	402383b3	S1:Decode	X	OldPC + Imm
47	34	402383b3	S6:ExecuteR	X	ALUResult = x7(12) - x2(5)
48	34	402383b3	S7:ALUWB	7	Result = ALUOUT
49	34	402383b3	S0:Fetch	0x38	PC + 4
50	38	0471aa23	S1:Decode	X	OldPC + Imm
51	38	0471aa23	S2:MemAdr	X	ALUResult= x3(12) + 84
52	38	0471aa23	S5:MemWrite	96	Result = ALUOUT
53	38	0471aa23	S0:Fetch	0x3C	PC + 4
54	3c	06002103	S1:Decode	X	OldPC + Imm
55	3c	06002103	S2:MemAdr	X	ALUResult = x0(0) + 96
56	3c	06002103	S3:MemRead	96	Result = ALUOUT
57	3c	06002103	S4:MemWB	7	Result = mem[96] = 7
58	3c	06002103	S0:Fetch	0x40	PC + 4
59	40	005104b3	S1:Decode	X	OldPC + Imm
60	40	005104b3	S6:ExecuteR	X	ALUResult = x2(7)+x5(11)

61	40	005104b3	S7:ALUWB	18	Result = ALUOUT
62	40	005104b3	S0:Fetch	0x44	PC + 4
63	44	008001ef	S1:Decode	X	OldPC + Imm
64	44	008001ef	S9:JAL	0x48	OldPC + 4
65	48	008001ef	S7:ALUWB	0x44	Result = ALUOUT
66	48	008001ef	S0:Fetch	0x4c	PC + Imm
67	4c	00910133	S1:Decode	X	OldPC + Imm
68	4c	00910133	S6:ExecuteR	X	ALUResult = x2(7)+x9(18)
69	4c	00910133	S7:ALUWB	25	Result = ALUOUT
70	4c	00910133	S0:Fetch	0x50	PC + 4
71	50	0221a023	S1:Decode	X	OldPC + Imm
72	50	0221a023	S2:MemAdr	X	ALUResult = x3(68)+32
73	50	0221a023	S5:MemWrite	100	Result = ALUOUT
74	50	0221a023	S0:Fetch	0x54	PC + 4
75	54	00210063	S1:Decode	X	OldPC + Imm
76	54	00210063	S10:BEQ	X	ALUResult = x2(25) - x2(25)
77	50	00210063	S0:Fetch	0x54	PC + 4

سیگنال‌های خروجی با مقادیر پیش بینی شده بالا مطابقت دارد. پس پردازنده به طور درست کار می‌کند.