گزارشکار پروژه معماری کامپیوتر

غزل عربعلی (۹۷۵۲۱۳۹۶)

بهاره کاوسی نژاد (۹۹۴۳۱۲۱۷)

عنوان پروژه : " طراحی و پیاده سازی یک پردازنده Pipeline RISC-V در gem5 "

هدف کلی پروژه :

هدف ما در اجرای این پروژه مقایسه زمان اجرای سلسله دستورات (برنامه یا Program)برابر در هر ۲ حالت پیاده سازی پردازنده و مشاهده تاثیر اجرای دستور العمل ها در پردازنده Pipelined با ۵ stage است .

روند پیاده سازی :

در ابتدای عمل ما جهت طراحی و پیاده سازی (شبیه سازی پردازنده از طریق gem5) باید تمامی المان های (Components) موجود در این پردازنده را از طریق زبان Verilog که یک زبان توصیف سخت افزار هست طراحی کنیم. در ادامه به توضیحات مختصری درباره ۲ نوع پیاده سازی میپردازیم .

طراحی پردازنده به زبان Verilog :

۱. طراحی پردازنده به صورت " SingleCycle "

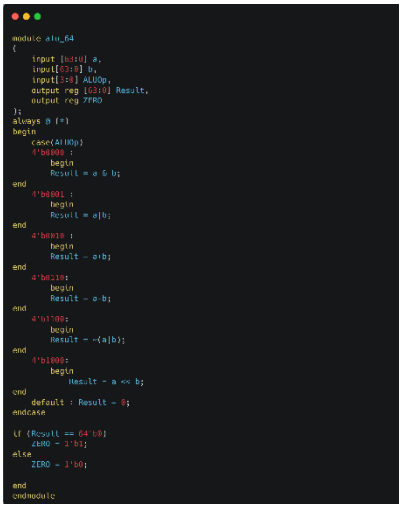
در این مدل پیاده سازی دستورالعمل ها(Instructions) به صورت Sequential اجرا میشوند یعنی بعد از اجرای کامل یک دستور العمل (اتمام سیکل های fetch و Decode و Execute برای یک دستورالعمل)دستورالعمل بعدی وارد سیکل های fetch و decode و .. میشود .

پی نوشت : کد Verilog موجود در فایل " RISC_V_Processor_SingleCycle.v " (در فایل های موجود نام اصلاح شود)

فایل ذکر شده شامل Module های متفاوتی است که به آن ها میپردازیم :

module alu_64 :

این ماژول یک واحد (Arithmetic & Logic Unit) ۶۴ بیتی را پیاده سازی میکند که عملیات مختلفی را بر روی دو عملوند (Operand) ورودی بر اساس سیگنال ALUOp انجام میدهد و یک نتیجه و یک خروجی flag صفر تولید میکند . عملیات های پشتیبانی شده شامل AND bitwise ، OR bitwise ، جمع ، تفریق ، NOR bitwise و شیفت به چپ است.



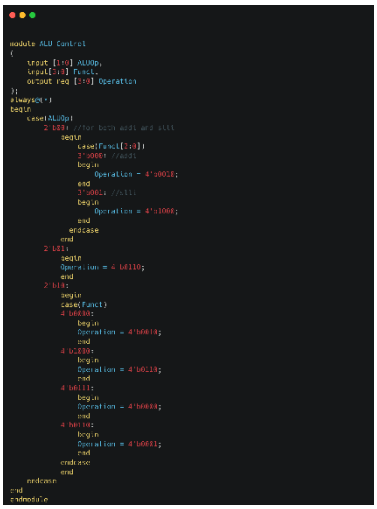
```
module alu_64
(
    input [63:0] a,
    input[63:0] b,
    input[3:0] ALUOp,
    output reg [63:0] Result,
    output reg ZERO
);
always @ (*)
begin
    case(ALUOp)
    4'b0000 :
        begin
        Result = a & b;
end
    4'b0001 :
        begin
        Result = a|b;
end
    4'b0010 :
        begin
        Result = a+b;
end
    4'b0110:
        begin
        Result = a-b;
end
    4'b1100:
        begin
        Result = ~(a|b);
end
    4'b1000:
        begin
        Result = a << b;
end
    default : Result = 0;
endcase

if (Result == 64'b0)
    ZERO = 1'b1;
else
    ZERO = 1'b0;

end
endmodule
```

```
module alu_64
(
    input [63:0] a,
    input[63:0] b,
    input[3:0] ALUOp,
    output reg [63:0] Result,
    output reg ZERO
);
```

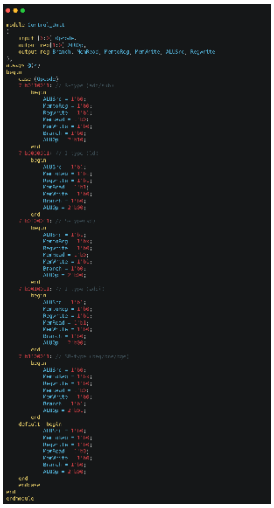
module ALU_Control :



این ماژول یک واحد کنترل ALU را پیاده سازی میکند که یک سیگنال کنترلی ۴بیتی برای ALU بر اساس ورودی های ۲بیتی ALUOp و ۴بیتی Funct تولید میکند. این ماژول عملیات ALU مناسب را برای انواع دستورالعمل های مختلف، از جمله دستورالعمل های ADDI، SLLI، BEQ، BNE، نوع R و نوع SB انتخاب میکند.

```
module ALU_Control
(
    input [1:0] ALUOp,
    input[3:0] Funct,
    output reg [3:0] Operation
);
```

module Control_Unit :



این ماژول یک واحد کنترل را برای یک پردازنده پیاده‌سازی میکند که سیگنال‌های کنترلی را بر اساس opcode دستورالعمل (Instruction) تولید میکند. سیگنال‌های کنترلی شامل انتخاب عملیات ALU ، دسترسی به حافظه، فعال سازی نوشتن در رجیستر و ارزیابی وضعیت branch میشوند.

```
module Control_Unit
(
    input [6:0] Opcode,
    output reg[1:0] ALUOp,
    output reg Branch, MemRead, MemtoReg, MemWrite, ALUSrc, Regwrite
);
```

module data_generator:

این ماژول یک دستورالعمل ۳۲ بیتی را به عنوان ورودی میگیرد و immediate data متناظر را برای انواع مختلف دستورالعمل ها تولید میکند. این ماژول immediate data را به عنوان یک سیگنال خروجی ۶۴ بیتی ارائه میدهد.

```
module data_generator
(
input [31:0] instruction,
output reg[63:0] imm_data

);
wire[6:0] opcode;
assign opcode = instruction[6:0];
always @(*)
begin
    case (opcode)
        7'b0000011: imm_data =  {{52{instruction[31]}}, instruction [31:20]};
        7'b0100011: imm_data = {{52{instruction[31]}}, instruction [31:25],
instruction [11:7]};
        7'b1100011: imm_data = {{52{instruction[31]}}, instruction [31] , instruction
[7], instruction [30:25], instruction [11:8]};
        7'b0010011: imm_data = {{52{instruction[31]}}, instruction[31:20]};
        default : imm_data = 64'd0;
    endcase
end
endmodule
```

immediate data تولید شده به فیلد opcode دستورالعمل ورودی بستگی دارد.

```
module data_generator
(
input [31:0] instruction,
output reg[63:0] imm_data

);
```

module Adder :

این ماژول یک جمع کننده ۶۴ بیتی است که دو ورودی a و b را میگیرد و مجموع ورودی ها را به عنوان سیگنال خروجی ارائه میدهد. عملیات جمع با استفاده از یک بلوک همیشه انجام میشود که با تغییر هر یک از سیگنال های ورودی فعال میشود .

```
module Adder(
    input[63:0] a, b,
    output reg [63:0] out
);
always@(*)
    out = a + b;
endmodule
```

module Data_Memory :

این ماژول یک آدرس حافظه ۶۴ بیتی، داده ۶۴بیتی و سیگنال های کنترلی برای عملیات نوشتن در حافظه و خواندن از حافظه را به عنوان ورودی میگیرد. این ماژول یک داده خواندنی ۶۴ بیتی و هشت عنصر خروجی ۶۴بیتی را به عنوان خروجی ارائه میدهد.
داده ها را از حافظه میخواند و درون حافظه مینویسد و داده ها را در یک آرایه۶۴*۸بیتی ذخیره میکند.

```
module Data_Memory
(
  input [63:0] mem_addr,
  input[63:0] write_data,
  input clk, mem_write, mem_read,
  output reg [63:0] read_data,
  output[63:0] element1,
  output[63:0] element2 ,
  output[63:0] element3,
  output[63:0] element4,
  output[63:0] element5,
  output[63:0] element6,
  output[63:0] element7,
  output[63:0] element8);
reg[0:7] data_mem[63:0];
initial
begin
    data_mem[0] = 64'd0;
end
always @(negedge clk)
begin
    if (mem_write)
        begin
            data_mem[mem_addr] = write_data[7:0];
data_mem[mem_addr + 1] = write_data[15:8];
data_mem[mem_addr + 2] = write_data[23:16];
data_mem[mem_addr + 3] = write_data[31:24];
data_mem[mem_addr + 4] = write_data[39:32];
data_mem[mem_addr + 5] = write_data[47:40];
data_mem[mem_addr + 6] = write_data[55:48];
data_mem[mem_addr + 7] = write_data[63:56];
end
end
  always @(*)
    begin
        if(mem_read)
        begin
            read_data = { data_mem[mem_addr + 7], data_mem[mem_addr + 6],
data_mem[mem_addr + 5], data_mem[mem_addr + 4], data_mem[mem_addr + 3],
data_mem[mem_addr + 2], data_mem[mem_addr + 1], data_mem[mem_addr] };
end
end
  assign element1= {data_mem[7],data_mem[6],data_mem[5], data_mem[4], data_mem[3],
data_mem[2], data_mem[1], data_mem[0]};
assign element2 = { data_mem[15], data_mem[14], data_mem[13], data_mem[12],
data_mem[11], data_mem[10], data_mem[9], data_mem[8] };
assign element3 = { data_mem[23], data_mem[22], data_mem[21], data_mem[20],
data_mem[19], data_mem[18], data_mem[17], data_mem[16] };
assign element4 = { data_mem[31], data_mem[30], data_mem[29], data_mem[28],
data_mem[27], data_mem[26], data_mem[25], data_mem[24] };
assign element5 = { data_mem[39], data_mem[38], data_mem[37], data_mem[36],
data_mem[35], data_mem[34], data_mem[33], data_mem[32] };
assign element6 = { data_mem[47], data_mem[46], data_mem[45], data_mem[44],
data_mem[43], data_mem[42], data_mem[41], data_mem[40] };
assign element7 = { data_mem[55], data_mem[54], data_mem[53], data_mem[52],
data_mem[51], data_mem[50], data_mem[49], data_mem[48] };
assign element8 = { data_mem[63], data_mem[62], data_mem[61], data_mem[60],
data_mem[59], data_mem[58], data_mem[57], data_mem[56] };
endmodule
```

```
module Data_Memory
(
  input [63:0] mem_addr,
  input[63:0] write_data,
  input clk, mem_write, mem_read,
  output reg [63:0] read_data,
  output[63:0] element1,
  output[63:0] element2 ,
  output[63:0] element3,
  output[63:0] element4,
  output[63:0] element5,
  output[63:0] element6,
  output[63:0] element7,
  output[63:0] element8);
reg[0:7] data_mem[63:0];
initial
begin
    data_mem[0] = 64'd0;
```

module Instruction_Memory :

این ماژول دستورالعمل های ۳۲بیتی را در یک آرایه ۸ بیتی ذخیره میکند و دستورالعمل ها را بر اساس یک
instruction address ۶۴ بیتی میخواند.
ماژول یک دستورالعمل خروجی ۳۲بیتی را بر اساس
instruction address ورودی ارائه میدهد.

پی نوشت : کد کامل Verilog این module در فایل
"Instruction_Memory.v" موجود است .

```
module Instruction_Memory(
    input [63:0] Inst_Address,
    output reg[31:0] Instruction
);
reg[7:0] inst_memory[131:0];
```

module Instruction_Parser :

این ماژول یک دستورالعمل ۳۲ بیتی را به عنوان ورودی دریافت میکند و فیلدهای مختلف دستورالعمل مانند opcode، function code ، رجیستر مقصد و رجیسترهای منبع را استخراج میکند. این ماژول این فیلدهای استخراج شده را به عنوان سیگنال های خروجی ارائه میدهد.

```
module Instruction_Parser
(
    input [31:0] instruction,
    output[6:0] opcode, funct7,
    output[4:0] rd , rs1, rs2,
    output[2:0] funct3

);
assign opcode = instruction[6:0];
assign rd = instruction[11:7];
assign funct3 = instruction[14:12];
assign rs1 = instruction[19:15];
assign rs2 = instruction[24:20];
assign funct7 = instruction[31:25];
endmodule
```

module mux2x1:

```
module mux2x1
(
    input [63:0] a,b,
    input s ,
    output[63:0] data_out
);
assign data_out = s ? b : a;
endmodule
```

این ماژول یک مالتی پلکسر ۲ به ۱ است که دو ورودی ۶۴ بیتی و یک سیگنال select تک بیتی را به عنوان ورودی میگیرد. این ماژول ورودی انتخاب شده با توجه سیگنال select را به عنوان سیگنال خروجی ارائه میکند.

module Program_Counter :

این ماژول شمارنده برنامه است که یک سیگنال Clock، یک سیگنال reset و یک سیگنال ورودی ۶۴ بیتی را به عنوان ورودی میگیرد. این ماژول یک سیگنال خروجی ۶۴بیتی را به عنوان مقدار شمارنده برنامه فعلی ارائه میدهد.
این ماژول هنگام فعال شدن سیگنال reset ، شمارنده برنامه را روی صفر مقداردهی میکند و حتی در حین کار میتواند شمارنده برنامه را به صفر برساند (reset_force) .

```
module Program_Counter
(
    input clk, reset,
    input[63:0] PC_In,
    output reg [63:0] PC_Out
);
reg reset_force;
initial
PC_Out <= 64'd0;
always @(posedge clk or posedge reset)
begin
    if (reset || reset_force) begin
        PC_Out = 64'd0;
        reset_force <= 0;
end
    else
    PC_Out = PC_In;
end
always @(negedge reset) reset_force <= 1;
endmodule
```

module selector :

این ماژول دو ورودی ۶۴ بیتی a و b، یک سیگنال branch، یک سیگنال صفر و یک سیگنال funct3 را به عنوان ورودی میگیرد. این ماژول یک سیگنال select را به عنوان خروجی بر اساس مقادیر سیگنال های ورودی ارائه میدهد. سیگنال بسته به opcode موجود در instruction و شرایط مشخص شده توسط سیگنال‌های ورودی، تعیین میکند که آیا باید یک branch بگیرد یا نه .

```
module selector
(
    input branch, ZERO,
    input[63:0] a, b,
    input[2:0] funct3,
    output reg sel
);
```

module registerFile :

این ماژول مقادیر داده های ۶۴ بیتی را ذخیره میکند. این ماژول دارای دو پورت خواندن و یک پورت نوشتن است و ورودی هایی را برای فعال کردن نوشتن، نوشتن داده ها و آدرس های خواندن میگیرد. سیگنال های خروجی مقادیر داده ای هستند که از آدرس های مشخص شده خوانده میشوند. این ماژول با Clock عمل میکند و قابل ریست شدن است و رجیسترها را صفر میکند.

```
module registerFile
(
    input [63:0] WriteData,
    input[4:0] RS1,
    input[4:0] RS2,
    input[4:0] RD,
    input RegWrite, clk, reset,
    output reg [63:0] ReadData1,
    output reg[63:0] ReadData2
);
reg[63:0] Registers[31:0];
initial
    begin
    Registers[0] = 64'd 0;
    .
    .
    .
    .
    .
    end
  always @(posedge clk)
    if (RegWrite)
    begin
    Registers[RD] = WriteData;
end
always @(*)
    if (reset)
    begin
    ReadData1 = 64'b0;
        ReadData2 = 64'b0;
        end
    else
        begin
        ReadData1 = Registers[RS1] ;
ReadData2 = Registers[RS2];
end
endmodule
```

```
module registerFile
(
    input [63:0] WriteData,
    input[4:0] RS1,
    input[4:0] RS2,
    input[4:0] RD,
    input RegWrite, clk, reset,
    output reg [63:0] ReadData1,
    output reg[63:0] ReadData2
);
```

پی نوشت : کد کامل Verilog این module در فایل " registerFile.v " موجود است .

module RISC_V_Processor :

این ماژول، ماژول اصلی ما بوده و از این ماژول برای تبدیل کد Verilog به کد زبان C++ استفاده میکنیم . پردازنده RISC-V شامل اجزای مختلفی مانند program counter ، instruction memory ، control unit، register file ، data memory و ALU میباشد که یک Clock و سیگنال reset را به عنوان ورودی میگیرد و خروجی های مختلفی مانند نتیجه عملیات ALU و داده های خوانده شده از حافظه را ارائه میدهد .

```
module RISC_V_Processor
(
    input clk, reset
);
wire[63:0] PC_In_from_mux;
wire[63:0] PC_Out;
wire[63:0] a1_out;
wire[63:0] a2_out;
wire[63:0] b_in = 64'd4;
wire[31:0] Instruction;
wire[4:0] rd;
wire[4:0] rs1;
wire[4:0] rs2;
wire[6:0] opcode;
wire[6:0] funct7;
wire[2:0] funct3;
wire[63:0] ReadData1;
wire[63:0] ReadData2;
wire[1:0] ALUOp;
wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
wire[3:0] Operation;
wire[3:0] Funct;
assign Funct = { Instruction[30], Instruction[14:12]};
wire[63:0] Result_from_alu;
wire zero_output;
wire[63:0] imm_data;
wire[63:0] out_from_mux2;
wire sel;
wire[63:0] out_from_DM;
wire[63:0] out_from_mux3;
wire[63:0] b_adder2;
assign b_adder2 = imm_data << 1;
.
.
.
.
.
endmodule
```

پی نوشت : کد کامل Verilog این module در فایل
"RISC_V_processor_SingleCycle.v" موجود است .

۲. طراحی پردازنده به صورت " Pipelined "

در این مدل پیاده سازی دستورالعمل ها (Instructions) با همپوشانی و در ۵ گام (Stage)
(Fetch - Decode - Execute - Memory - WriteBack) اجرا میشوند . در این مدل پیاده سازی پس از آنکه سیکل Fetch برای دستور العمل ابتدایی تمام شد و وارد سیکل Decode شد دستورالعمل بعدی وارد سیکل Fetch میشود و بدین صورت دستورات با همپوشانی اجرا خواهند شد .

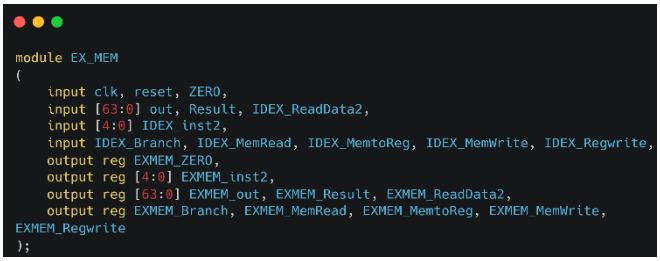
پی نوشت : کد Verilog موجود در فایل " RISC_V_Processor_Pipelined.v " (اسم درون فایل ها اصلاح شود)

علاوه بر module های مطرح شده در حالت اول پیاده سازی به دلیل وجود گام های مختلف و نیز حضور یکسری رجیستر ها میانی بینابین این گام ها به توضیح مختصر module های دیگر میپردازیم :

module EX_MEM :

یک register file است که خروجی execution stage پایپ لاین پردازشگر را که شامل نتیجه ALU، سیگنال های کنترلی و branch target را ذخیره میکند تا به مرحله بعدی منتقل شود. این رجستر توسط لبه Clock و سیگنال reset راه اندازی میشود.

```
module EX_MEM
(
    input clk, reset, ZERO,
    input [63:0] out, Result, IDEX_ReadData2,
    input [4:0] IDEX_inst2,
    input IDEX_Branch, IDEX_MemRead, IDEX_MemtoReg, IDEX_MemWrite, IDEX_Regwrite,
    output reg EXMEM_ZERO,
    output reg [4:0] EXMEM_inst2,
    output reg [63:0] EXMEM_out, EXMEM_Result, EXMEM_ReadData2,
    output reg EXMEM_Branch, EXMEM_MemRead, EXMEM_MemtoReg, EXMEM_MemWrite,
EXMEM_Regwrite
);
always @(posedge clk or reset)
begin
    if(clk)
        begin
            EXMEM_out = out;
            EXMEM_ZERO = ZERO;
            EXMEM_Result = Result;
            EXMEM_ReadData2 = IDEX_ReadData2;
            EXMEM_inst2 = IDEX_inst2;
            EXMEM_Branch = IDEX_Branch;
            EXMEM_MemRead = IDEX_MemRead;
            EXMEM_MemtoReg= IDEX_MemtoReg;
            EXMEM_MemWrite= IDEX_MemWrite;
            EXMEM_Regwrite= IDEX_Regwrite;
        end
    else
        begin
            EXMEM_out = 0;
            EXMEM_ZERO = 0;
            EXMEM_Result = 0;
            EXMEM_ReadData2 = 0;
            EXMEM_inst2 = 0;
            EXMEM_Branch = 0;
            EXMEM_MemRead = 0;
            EXMEM_MemtoReg= 0;
            EXMEM_MemWrite= 0;
            EXMEM_Regwrite=0;
        end
    end
endmodule
```



module Forwarding_Unit :

یک واحد منطقی combinational است که بر اساس سیگنال های کنترلی و مقادیر رجیستر ها تعیین میکند که داده ها را از خروجی execution stage یا memory stage پایپ لاین پردازشگر به ورودی execution stage ارسال کند .
این ماژول دو سیگنال forwarding ۲بیتی را برای استفاده در مرحله بعدی پایپ لاین تولید میکند .

```
module Forwarding_Unit
(
    input EXMEM_ReadData2, MEMWB_read_data,
    input rs1, rs2,
    input EXMEM_Regwrite,
    input MEMWB_RegWrite,

    output reg [1:0] fwd_A,
    output reg [1:0] fwd_B
);
```

module ID_EX :

یک register file است که خروجی instruction decode stage را در پایپ لاین پردازشگر، شامل فیلدهای instruction، مقادیر رجسترها ، immediate values و سیگنال های کنترلی ذخیره میکند تا به گام (Stage) بعدی منتقل شود.

```
module ID_EX
(
    input clk, reset,
    input [3:0] inst1,
    input [4:0] inst2,
    input [63:0] ReadData1, ReadData2, PC_Out, imm_data,
    input [1:0] ALUOp,
    input Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite,
    output reg [3:0] IDEX_inst1,
    output reg [4:0] IDEX_inst2,
    output reg [63:0] IDEX_PC_Out, IDEX_ReadData1, IDEX_ReadData2, IDEX_imm_data,
    output reg [1:0] IDEX_ALUOp,
    output reg IDEX_Branch, IDEX_MemRead, IDEX_MemtoReg, IDEX_MemWrite, IDEX_ALUSrc,
IDEX_Regwrite
);
always @(posedge clk or reset)
    begin
        if(clk)
            begin
                IDEX_PC_Out = PC_Out;
                IDEX_ReadData1 = ReadData1;
                IDEX_ReadData2 = ReadData2;
                IDEX_imm_data = imm_data;
                IDEX_inst1 = inst1;
                IDEX_inst2 = inst2;
                IDEX_Branch = Branch;
                IDEX_MemRead = MemRead;
                IDEX_MemWrite = MemWrite;
                IDEX_ALUSrc = ALUSrc;
                IDEX_Regwrite = RegWrite;
                IDEX_ALUOp = ALUOp;
                IDEX_MemtoReg = MemtoReg;

            end
        else
            begin
                IDEX_PC_Out = 0;
                IDEX_ReadData1 = 0;
                IDEX_ReadData2 = 0;
                IDEX_imm_data = 0;
                IDEX_inst1 = 0;
                IDEX_inst2 = 0;
                IDEX_Branch = 0;
                IDEX_MemRead = 0;
                IDEX_MemWrite = 0;
                IDEX_ALUSrc = 0;
                IDEX_Regwrite = 0;
                IDEX_ALUOp = 0;
                IDEX_MemtoReg = 0;
            end
    end
endmodule
```

module IF_ID :

یک register file است که خروجی instruction fetch stage را در پایپ لاین پردازنده، شامل مقادیر instruction و شمارنده برنامه (PC) ذخیره میکند تا به مرحله بعدی منتقل شود .
این رجستر توسط لبه Clock و سیگنال reset راه اندازی میشود .
این ماژول همچنین مقادیر پیش‌فرض سیگنال‌های خروجی را هنگامی که سیگنال reset اعلام میشود، ارائه میکند.

```
module IF_ID
(
    input clk, reset,
    input [31:0] instruction,
    input [63:0] PC_Out,
    output reg [31:0] IFID_instruction,
    output reg [63:0] IFID_PC_Out
);
always @(posedge clk or reset)
begin
    if(clk)
        begin
            IFID_instruction = instruction;
            IFID_PC_Out = PC_Out;
        end
    else
        begin
            IFID_instruction = 0;
            IFID_PC_Out = 0;
        end
end
endmodule
```

module MEM_WB :

یک register file است که خروجی مرحله memory stage را در پایپ لاین پردازنده ذخیره میکند، از جمله داده های خواندنی برای حافظه یا نتیجه ALU، و سیگنال های کنترلی مربوط به نوشتن در رجیستر و memory-to-register forwarding ، تا به مرحله بعدی منتقل شود.
این رجستر توسط لبه Clock و سیگنال reset راه اندازی میشود .
این ماژول همچنین مقادیر پیش‌فرض سیگنال‌های خروجی را هنگامی که سیگنال reset اعلام میشود، ارائه میکند.

```
module MEM_WB
(
    input clk, reset,
    input [63:0] read data, Result,
    input EXMEM_MemtoReg, EXMEM_RegWrite,
    input [4:0] EXMEM_inst2,
    output reg [63:0] MEMWB_read_data, MEMWB_Result,
    output reg MEMWB_MemtoReg, MEMWB_RegWrite,
    output reg [4:0] MEMWB_inst2
);
    always @(posedge clk or reset)
    begin
        if(clk)
            begin
                MEMWB_read_data = read data;
                MEMWB_Result = Result;
                MEMWB_MemtoReg = EXMEM_MemtoReg;
                MEMWB_RegWrite = EXMEM_RegWrite;
                MEMWB_inst2 = EXMEM_inst2;

            end
        else
            begin
                MEMWB_read_data = 0;
                MEMWB_Result = 0;
                MEMWB_MemtoReg = 0;
                MEMWB_RegWrite = 0;
                MEMWB_inst2 = 0;
            end
    end
endmodule
```

module MUX_Triple :

این ماژول پیاده سازی یک مالتی پلکسر ۳ به ۱ با سه ورودی ۶۴ بیتی و یک ورودی select ۲ بیتی است.
خروجی یکی از ورودی ها را بر اساس ورودی select انتخاب میکند و نتیجه ۶۴ بیتی مربوطه را خروجی میدهد.

```verilog
module MUX_Triple
(
    input [63:0] a, b, c,
    input [1:0] sel,
    output reg [63:0] Res
);
always@(*)
begin
    case (sel)
    2'b00: Res = a;
    2'b01: Res = b;
    2'b10: Res = c;
        default: Res = 2'bX;
    endcase
end
endmodule
```

module RISC_V_Processor_Pipelined :

```verilog
module RISC_V_Processor_Pipelined
(
    input clk, reset
);
wire [63:0] PC_In from mux;
wire [63:0] PC_Out;
wire [63:0] a1_out;
wire [63:0] a2_out;
wire [63:0] b_in = 64'd4;
wire [31:0] Instruction;
wire [63:0] ReadData1;
wire [63:0] ReadData2;
wire [1:0] ALUOp;
wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
wire [3:0] Operation;
wire [3:0] Funct;
assign Funct = {Instruction[30], Instruction[14:12]};
wire [63:0] Result_from_alu;
wire zero_output;
wire [63:0] imm_data;
wire [63:0] out_from_mux2;
wire [63:0] out_from_DM;
wire [63:0] out from mux3;
wire [31:0] IFID_instruction;
wire [63:0] IFID_PC_Out;
wire [4:0] rd;
wire [4:0] rs1;
wire [4:0] rs2;
wire [6:0] opcode;
wire [6:0] funct7;
wire [2:0] funct3;
wire [63:0] IDEX_PC_Out;
wire [63:0] IDEX_ReadData1;
wire [63:0] IDEX ReadData2;
wire [63:0] IDEX_imm_data;
wire [3:0] IDEX_inst1;
wire [4:0] IDEX_inst2;
wire [1:0] IDEX_ALUOp;
wire IDEX_Branch, IDEX_MemRead, IDEX_MemtoReg, IDEX_MemWrite, IDEX_ALUSrc,
IDEX_Regwrite;
wire [63:0] b_adder2;
assign b_adder2 = IDEX_imm_data << 1;
wire [63:0] EXMEM_out;
wire EXMEM_ZERO;
wire [63:0] EXMEM Result;
wire [63:0] EXMEM_ReadData2;
wire [4:0] EXMEM_inst2;
wire EXMEM Branch, EXMEM MemRead, EXMEM MemtoReg, EXMEM MemWrite, EXMEM Reg
wire [63:0] MEMWB_read_data;
wire [63:0] MEMWB_Result;
wire MEMWB_MemtoReg, MEMWB_RegWrite;
wire [4:0] MEMWB inst2;
wire PC_src;
wire [1:0] FU_fwdA;
wire [1:0] FU_fwdB;
wire [63:0] Res;
wire [63:0] Resa;
wire [63:0] Resb;
.
.
.
endmodule
```

این ماژول، ماژول اصلی ما بوده و از این ماژول برای تبدیل کد Verilog به کد زبان C++ استفاده میکنیم . یک پردازنده RISC-V پایپ لاین شده را نشان میدهد که دستورالعمل ها را با عبور از مراحل مختلف پایپ لاین اجرا میکند. این ماژول شامل اجزایی برای Fetch دستورالعمل، Decode ، Execute و WriteBack و MemoryAccess و Forwarding برای اجرای کارآمد و جلوگیری از Hazard است.

پی نوشت : کد کامل Verilog این module در فایل "RISC_V_Processor_Pipelined.v" موجود است .

تا به حال پردازنده مد نظر را به ۲ صورت (SingleCycle و Pipelined)به زبان توصیف سخت افزار Verilog پیاده سازی کردیم .

ما جهت شبیه سازی پردازنده خواسته شده در "gem5" باید فایل قابل اجرا (executable) کد C++ آن پردازنده را داشته باشیم حال به تبدیل کد Verilog به زبان C++ میپردازیم .

نصب Verilator :

در ابتدا برای تبدیل کد Verilog به زبان C++ نیاز به ابزاری به نام Verilator داریم که برای شبیه سازی و صحت سنجی مدار های دیجیتالی توصیف شده به زبان توصیف سخت افزار Verilog یا SystemVerilog استفاده میشود . که برای نصب این ابزار از Command زیر استفاده کردیم :

```
$ sudo apt-get install verilator
```

ذخیره Module ها در فایل های جداگانه (با دامنه v ← .v) :

در تبدیل کد Verilog موجود به کد به زبان C++ باید هر module موجود در کد Verilog اصلی را در فایل های ".v" جداگانه قرار دهیم و module اصلی را که در فایل RISC_V_processor.v (در حالتSingleCycle) و در فایل RISC_V_Processor_Pipelined.v (در حالت Pipelined) قرار دارد را با استفاده از ابزار نصب شده (Verilator) و با استفاده از فایل کمکی CppWrapper که مخصوص برای module اصلی ما هست را به کد به زبان C++ تبدیل میکنیم .

ماژول های پردازنده به صورت " SingleCycle " (موجود در فولدر SingleCycle)

- Instruction_Memory.v - Data_Memory.v - data_generator.v - Control_Unit.v - ALU_Control.v - alu_64.v - Adder.v
RISC_V_processor.v - selector.v - registerFile.v - Program_Counter.v - mux2x1.v - Instruction_Parser.v

ماژول های پردازنده به صورت " Pipelined " (موجود در فولدر 5Stage)

- EX_MEM.v - Data_Memory.v - data_generator.v - Control_Unit.v - ALU_Control.v - alu_64.v - Adder.v
- MEM_WB.v - Instruction_Parser.v - Instruction_Memory.v - IF_ID.v - ID_EX.v - Forwarding_Unit.v
RISC_V_Processor_Pipelined.v - selector.v - registerFile.v - Program_Counter.v - mux2x1.v - MUX_Triple.v

ساخت فایل کمکی CppWrapper برای module اصلی :

(RISC_V_Processor_Pipelined.v و RISC_V_processor.v)

ما هنگام تبدیل کد Verilog به کد C++ از CppWrapper استفاده میکنیم تا یک interface بین طراحی Verilog و محیط شبیه‌سازی مبتنی بر نرم‌افزار که در آن کد C++ اجرا میشود، فراهم کنیم. Wrapper به عنوان یک پل بین دو محیط عمل میکند و به سیگنال ها اجازه میدهد بین آنها منتقل شود و شبیه سازی را قادر میسازد تا به طور دقیق رفتار طرح Verilog را به زبان C++ مدل کند.

پی نوشت : پس از تکمیل فایل RISC_V_Processor_Pipelined.v و RISC_V_processor.v که module های اصلی ما در ۲ حالت پیاده سازی هستند و همچنین ساخت فایل Wrapper جداگانه برای هر کدام باید Command زیر را در terminal اجرا کنیم .

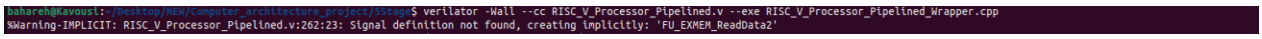
فایل کمکی CppWrapper پردازنده " SingleCycle " ← RISC_V_processorWrapper.cpp

فایل کمکی CppWrapper پردازنده " Pipelined " ← RISC_V_Processor_Pipelined_Wrapper.cpp

```
$ verilator -Wall --cc RISC_V_processor.v --exe RISC_V_processorWrapper.cpp
```

```
bahareh@Kavoust:~/Desktop/NEW/Computer_architecture_project/SingleCycle$ verilator -Wall --cc RISC_V_Processor_SingleCycle.v --exe RISC_V_Processor_SingleCycleWrapper.cpp
%Warning-DECLFILENAME: RISC_V_Processor_SingleCycle.v:4:8: Filename 'RISC_V_Processor_SingleCycle' does not match MODULE name: 'RISC_V_Processor'
```

```
$ verilator -Wall --cc RISC_V_Processor_Pipelined.v --exe RISC_V_Processor_Pipelined_Wrapper.cpp
```



پس از اجرای این دستور فولدری با نام " obj_dir " ایجاد میشود که شامل فایلی با دامنه mk. خواهد بود .

ساخت فایل قابل اجرا (Executable) از طریق دستور make :

چالش میانی : در تبدیل کد Verilog موجود به کد به زبان C++ که با استفاده از ابزار Verilator و کمک فایل Wrapper انجام میشود در سیستم عامل Linux نیاز به automake1.13 برای ساخت makefiles داشتیم (فایلی با دامنه mk ← mk.) در نتیجه automake به روز رسانی شده 1.16.3 را به 1.13 تبدیل کردیم .

```
$ make -j -f VRISC_V_processor.mk VRISC_V_processor
```

```
$ make -j -f VRISC_V_Processor_Pipelined.mk VRISC_V_Processor_Pipelined
```

پس از اجرای این دستور فایل قابل اجرای (Compiled cpp) برای استفاده در شبیه ساز gem5 خواهیم داشت .

فایل executable پردازنده " SingleCycle " ← VRISC_V_processor

فایل executable پردازنده " Pipelined " ← VRISC_V_Processor_Pipelined

شبیه سازی پردازنده در gem5 :

" توضیحات فایل اسکریپت بر اساس gem5 Documentation موجود در سایت "

فایل اسکریپت پردازنده " SingleCycle " ← 5Stage

فایل اسکریپت پردازنده " SingleCycle " ← SingleCycle

```
import m5
from m5.objects import *

system = System()

system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'

system.clk_domain.voltage_domain = VoltageDomain()
```

برای نوشتن این اسکریپت در ابتدا نیازمندیم کتابخانه لازم را ادد کنیم . و سپس Simobject های کامپایل شده موجود در این کتابخانه را نیز ادد میکنیم . (Simobject ها اشیایی از کلاس های C++ هستند که main interface را به تمام اشیا gem5 میفرستند)

```
import m5
from m5.objects import *
```

بعد از import , library های موردنیاز مشخص کنیم از چه پارامترها و قطعاتی جهت شبیه سازی استفاده میکنیم .

یک شی از سیستمی که قصد شبیه سازی آن را داریم میسازیم که من جمله simobject های افزوده شده در مرحله قبل هست. System والد تمام اشیاء دیگر در سیستم شبیه سازی شده ما خواهد بود. شی System اطلاعات کاربری زیادی مثل محدوده های حافظه فیزیکی، دامنه clock ، دامنه voltage ، هسته (در شبیه سازی تمام سیستم)، و غیره را شامل میشود .

```
system = System()
```

حال clock سیستم را تنظیم میکنیم (این پارامتر باید یک مقدار اعشاری میباشد).برای تنظیم کلاک سیستم ابتدا دامنه clock را روی سیستم ایجاد میکنیم و سپس در دامنه ایجاد شده clock را تنظیم میکنیم .تنظیم این مقادیر برای simobject مشابه تنظیم اعضای یک شی (attribute) است . فرکانس clock به صورت پیش فرض روی 1 GHZ تنظیم شده .یک دامنه برای ولتاژ در دامنه clock تعیین میکنیم که به صورت پیش فرض قرار داده شده .

```
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()
```

```
system.mem_mode = 'timing'
system.mem_ranges = [AddrRange('512MB')]

system.cpu = X86MinorCPU()

system.membus = SystemXBar()

system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
```

حال باید حافظه سیستم را تنظیم کنیم(از حالت timing استفاده میکنیم) محدوده حافظه را ۵۱۲ مگابایت قرار میدهیم .

```
system.mem_mode = 'timing'
system.mem_ranges = [AddrRange('512MB')]
```

سپس cpu را میسازیم (در این فعالیت این پارامتر مقدار TimingSimple را اخذ میکند که از ISAX86 (Instruction Set Architecture) تبعیت میکند) برای ایجاد این cpu یک شی از آن میسازیم .

```
system.cpu = X86MinorCPU()
```

سپس memory bus سراسری سیستم را ساخته و portهای cache روی cpu را به آن وصل میکنیم. در این سیستم شبیه سازی شده cache نداریم بنابراین باید port های I-cache و D-cache را به صورت مستقیم به memory bus وصل شوند .

```
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
```

حال برای اطمینان از کارکرد درست سیستم چند port دیگر را متصل میکنیم. باید یک I/O controller روی cpu ایجاد کنیم و آن را به memory bus وصل کنیم. همچنین به دلیل اینکه از ISA X86 استفاده می کنیم باید port های pio و interrupt را به memory bus وصل کنیم .

```
system.cpu.createInterruptController()

system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports
```

```
system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports
```

در این مرحله یک memory controller ایجاد میکنیم و آن را به memory bus متصل کنیم. مقدار controller به صورت پیش فرض DDR3_1600_x64 قرار گرفته است که برای محدوده ی در نظر گرفته شده برای memory ما استفاده میشود .

```
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
system.system_port = system.membus.cpu_side_ports
```

```
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_x64()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
system.system_port = system.membus.cpu_side_ports
```

آدرس binary ما برابر آدرس فایل کامپایل شده ی برنامه ای است که قصد اجرای آن را روی پردازنده(کامپیوتر) شبیه سازی شده داریم .

آدرس binary پردازنده" SingleCycle " :

```
thispath = os.path.dirname(os.path.realpath(__file__))
binary = os.path.join(thispath,"../../../","tests/test-
progs/hello/bin/x86/linux/Project/VRISC_V_processor",)
```

آدرس binary پردازنده" Pipelined " :

```
thispath = os.path.dirname(os.path.realpath(__file__))
binary = os.path.join(thispath,"../../../", "tests/test-
progs/hello/bin/x86/linux/Project/VRISC_V_Processor_Pipelined",)
```

در آخر یک شی از root میسازیم و با متد instantiate تمام پارامترها در C++ معادلسازی میشوند و شبیه سازی شروع میشود .

```
system.workload = SEWorkload.init_compatible(binary)

process = Process()
process.cmd = [binary]
system.cpu.workload = process
system.cpu.createThreads()

root = Root(full_system = False, system = system)
m5.instantiate()

print("Beginning simulation!")
exit_event = m5.simulate()

print('Exiting @ tick {} because {}'
      .format(m5.curTick(), exit_event.getCause()))
```

```
system.workload = SEWorkload.init_compatible(binary)
process = Process()
process.cmd = [binary]
system.cpu.workload = process
system.cpu.createThreads()
root = Root(full_system = False, system = system)
m5.instantiate()
print("Beginning simulation!")
exit_event = m5.simulate()
```

- با اتمام شبیه سازی تعداد تیک ها نمایش داده میشود .

```
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

فایل قابل اجرا که در مرحله قبل بدست آوردیم را درون اسکریپ پایتون قرار دادیم حال Command زیر را اجرا میکنیم .

```
$ build/X86/gem5.opt configs/tutorial/Project/SingleCycle.py
```

```
bahareh@Kavousi:~/Desktop/NEW/Computer_architecture_project/gem5$ build/X86/gem5.opt configs/tutorial/Project/SingleCycle.py
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 22.1.0.0
gem5 compiled Mar 23 2023 16:45:48
gem5 started Jul  8 2023 00:56:00
gem5 executing on Kavousi, pid 19702
command line: build/X86/gem5.opt configs/tutorial/Project/SingleCycle.py

Global frequency set at 1000000000000 ticks per second
```

```
$ build/X86/gem5.opt configs/tutorial/Project/5Stgage.py
```

```
bahareh@Kavousi:~/Desktop/NEW/Computer_architecture_project/gem5$ build/X86/gem5.opt configs/tutorial/Project/5Stage.py
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 22.1.0.0
gem5 compiled Mar 23 2023 16:45:48
gem5 started Jul  8 2023 01:44:25
gem5 executing on Kavousi, pid 23871
command line: build/X86/gem5.opt configs/tutorial/Project/5Stage.py

Global frequency set at 1000000000000 ticks per second
```

این دستور را برای هر دو پردازنده پیاده سازی شده و برای ۵ حالت مختلف اجرای دستورالعمل ها اجرا میکنیم .

الف/ اجرای ۱ دستورالعمل (Instruction)

پردازنده" SingleCycle "

```
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISCVProcessorWrapper riscvProcessorWrapper;

    // Reset the module
    riscvProcessorWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscvProcessorWrapper.step(10);

    // Print the output signal

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);


    return 0;
}
```

پردازنده" Pipelined "

```
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISC_V_Processor_PipelinedModuleWrapper riscpModuleWrapper;

    // Reset the module
    riscpModuleWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscpModuleWrapper.step(10);

    // Print the output signals

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);

    return 0;
}
```

پردازنده " SingleCycle "

```
---------- Begin Simulation Statistics ----------
simSeconds                          0.291014     # Number of seconds simulated (Second)
simTicks                        291013647000     # Number of ticks simulated (Tick)
finalTick                       291013647000     # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                        1000000000000     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                            13.03     # Real time elapsed on the host (Second)
hostTickRate                     22329954616     # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                            760340     # Number of bytes of host memory used (Byte)
simInsts                             3464736     # Number of instructions simulated (Count)
simOps                               6195585     # Number of ops (including micro ops) simulated (Count)
hostInstRate                          265853     # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                            475393     # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

پردازنده " Pipelined "

```
---------- Begin Simulation Statistics ----------
simSeconds                          0.237889     # Number of seconds simulated (Second)
simTicks                        237888666000     # Number of ticks simulated (Tick)
finalTick                       237888666000     # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                        1000000000000     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                            10.63     # Real time elapsed on the host (Second)
hostTickRate                     22373124450     # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                            760344     # Number of bytes of host memory used (Byte)
simInsts                             2826825     # Number of instructions simulated (Count)
simOps                               4996089     # Number of ops (including micro ops) simulated (Count)
hostInstRate                          265856     # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                            469870     # Simulator op (including micro ops) rate (op/s) ((Count/Second))
system.clk_domain.clock                 1000     # Clock period in ticks (Tick)
```

```
The value of a + b is 5
```

ب/ اجرای ۵ دستورالعمل (Instruction)

پردازنده" SingleCycle " پردازنده" Pipelined "

```cpp
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISC_V_Processor_PipelinedModuleWrapper riscpModuleWrapper;

    // Reset the module
    riscpModuleWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscpModuleWrapper.step(10);

    // Print the output signals

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);

    return 0;
}
```

```cpp
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISCVProcessorWrapper riscvProcessorWrapper;

    // Reset the module
    riscvProcessorWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscvProcessorWrapper.step(10);

    // Print the output signal

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);

    return 0;
}
```

پردازنده" SingleCycle "

```
2 ---------- Begin Simulation Statistics ----------
3 simSeconds                                  0.291023                       # Number of seconds simulated (Second)
4 simTicks                                291022678000                       # Number of ticks simulated (Tick)
5 finalTick                               291022678000                       # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
6 simFreq                                 1000000000000                       # The number of ticks per simulated second ((Tick/Second))
7 hostSeconds                                   13.37                       # Real time elapsed on the host (Second)
8 hostTickRate                            21773613296                       # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
9 hostMemory                                    760340                       # Number of bytes of host memory used (Byte)
10 simInsts                                    3458584                       # Number of instructions simulated (Count)
11 simOps                                      6183876                       # Number of ops (including micro ops) simulated (Count)
12 hostInstRate                                 258759                       # Simulator instruction rate (inst/s) ((Count/Second))
13 hostOpRate                                   462655                       # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

پردازنده" Pipelined "

```
2 ---------- Begin Simulation Statistics ----------
3 simSeconds                                  0.238137                       # Number of seconds simulated (Second)
4 simTicks                                238136798000                       # Number of ticks simulated (Tick)
5 finalTick                               238136798000                       # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
6 simFreq                                 1000000000000                       # The number of ticks per simulated second ((Tick/Second))
7 hostSeconds                                   11.05                       # Real time elapsed on the host (Second)
8 hostTickRate                            21556013012                       # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
9 hostMemory                                    761368                       # Number of bytes of host memory used (Byte)
10 simInsts                                    2829555                       # Number of instructions simulated (Count)
11 simOps                                      5001286                       # Number of ops (including micro ops) simulated (Count)
12 hostInstRate                                 256126                       # Simulator instruction rate (inst/s) ((Count/Second))
13 hostOpRate                                   452707                       # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

```
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
```

پ/ اجرای ۱۰ دستورالعمل (Instruction)

پردازنده" SingleCycle " پردازنده" Pipelined "

```cpp
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISC_V_Processor_PipelinedModuleWrapper riscpModuleWrapper;

    // Reset the module
    riscpModuleWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscpModuleWrapper.step(10);

    // Print the output signals

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);

    return 0;
}
```

```cpp
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISCVProcessorWrapper riscvProcessorWrapper;

    // Reset the module
    riscvProcessorWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscvProcessorWrapper.step(10);

    // Print the output signal

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);


    return 0;
}
```

پردازنده “ SingleCycle ”

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.291121    # Number of seconds simulated (Second)
simTicks                                 291120634000    # Number of ticks simulated (Tick)
finalTick                                291120634000    # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                 1000000000000     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                                     13.61    # Real time elapsed on the host (Second)
hostTickRate                              21395011876    # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                                     760344    # Number of bytes of host memory used (Byte)
simInsts                                      3468145    # Number of instructions simulated (Count)
simOps                                        6202071    # Number of ops (including micro ops) simulated (Count)
hostInstRate                                   254877    # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                                     455795    # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

پردازنده “ Pipelined ”

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.238217    # Number of seconds simulated (Second)
simTicks                                 238217185000    # Number of ticks simulated (Tick)
finalTick                                238217185000    # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                 1000000000000     # The number of ticks per simulated second ((Tick/Second))
stSeconds                                       10.92    # Real time elapsed on the host (Second)
stTickRate                                21814473221    # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                                     760344    # Number of bytes of host memory used (Byte)
simInsts                                      2832977    # Number of instructions simulated (Count)
simOps                                        5007798    # Number of ops (including micro ops) simulated (Count)
hostInstRate                                   259423    # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                                     458575    # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

```
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
```

ت/ اجرای ۱۵ دستورالعمل (Instruction)

پردازنده “ SingleCycle ”

```
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);
    RISCVProcessorWrapper riscvProcessorWrapper;
    riscvProcessorWrapper.reset();
    riscvProcessorWrapper.step(10);
    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    return 0;
}
```

پردازنده “ Pipelined ”

```
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);

    // Instantiate the wrapper
    RISC_V_Processor_PipelinedModuleWrapper riscpModuleWrapper;

    // Reset the module
    riscpModuleWrapper.reset();

    // Run the simulation for 10 clock cycles
    riscpModuleWrapper.step(10);

    // Print the output signals

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);

    return 0;
}
```

پردازنده “ SingleCycle ”

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.291289    # Number of seconds simulated (Second)
simTicks                                 291289434000    # Number of ticks simulated (Tick)
finalTick                                291289434000    # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                 1000000000000     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                                     13.15    # Real time elapsed on the host (Second)
hostTickRate                              22158068985    # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                                     760344    # Number of bytes of host memory used (Byte)
simInsts                                      3461314    # Number of instructions simulated (Count)
simOps                                        6189073    # Number of ops (including micro ops) simulated (Count)
hostInstRate                                   263296    # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                                     470791    # Simulator op (including micro ops) rate (op/s) ((Count/Second))
system.clk_domain.clock                          1000    # Clock period in ticks (Tick)
```

پردازنده “ Pipelined ”

```
---------- Begin Simulation Statistics ----------
simSeconds                                   0.238356    # Number of seconds simulated (Second)
simTicks                                 238355562000    # Number of ticks simulated (Tick)
finalTick                                238355562000    # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
simFreq                                 1000000000000     # The number of ticks per simulated second ((Tick/Second))
hostSeconds                                     10.88    # Real time elapsed on the host (Second)
hostTickRate                              21904208419    # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
hostMemory                                     761372    # Number of bytes of host memory used (Byte)
simInsts                                      2836386    # Number of instructions simulated (Count)
simOps                                        5014264    # Number of ops (including micro ops) simulated (Count)
hostInstRate                                   260652    # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate                                     460790    # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

```
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
```

ث/ اجرای ۲۰ دستورالعمل (Instruction)

پردازنده" SingleCycle "

```
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);
    RISC_V_Processor_PipelinedModuleWrapper riscpMod
    riscpModuleWrapper.reset();
    riscpModuleWrapper.step(10);

    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;

    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);

    return 0;
}
```

پردازنده" Pipelined "

```
int main(int argc, char** argv) {
    Verilated::commandArgs(argc, argv);
    RISCVProcessorWrapper riscvProcessorWrapper;
    riscvProcessorWrapper.reset();
    riscvProcessorWrapper.step(10);
    int a = 2;
    int b = 3;
    int c = 4;
    int d = 5;
    int e = 6;
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    printf("The value of a + b is %d\n", a + b);
    printf("The value of b * c is %d\n", b * c);
    printf("The value of c / a is %d\n", c / a);
    printf("The value of d - a is %d\n", d - a);
    printf("The value of e - a is %d\n", e - a);
    return 0;
}
```

پردازنده " SingleCycle "

```
 2 ---------- Begin Simulation Statistics ----------
 3 simSeconds                     0.291380        # Number of seconds simulated (Second)
 4 simTicks                   291380161000        # Number of ticks simulated (Tick)
 5 finalTick                  291380161000        # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
 6 simFreq                   1000000000000         # The number of ticks per simulated second ((Tick/Second))
 7 hostSeconds                       13.31        # Real time elapsed on the host (Second)
 8 hostTickRate                21897990916        # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
 9 hostMemory                       760348        # Number of bytes of host memory used (Byte)
10 simInsts                        3471546        # Number of instructions simulated (Count)
11 simOps                          6208516        # Number of ops (including micro ops) simulated (Count)
12 hostInstRate                     260892        # Simulator instruction rate (inst/s) ((Count/Second))
13 hostOpRate                       466579        # Simulator op (including micro ops) rate (op/s) ((Count/Second))
14 system.clk_domain.clock            1000        # Clock period in ticks (Tick)
```

پردازنده " Pipelined "

```
 2 ---------- Begin Simulation Statistics ----------
 3 simSeconds                     0.238792        # Number of seconds simulated (Second)
 4 simTicks                   238791592000        # Number of ticks simulated (Tick)
 5 finalTick                  238791592000        # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
 6 simFreq                   1000000000000         # The number of ticks per simulated second ((Tick/Second))
 7 hostSeconds                       10.97        # Real time elapsed on the host (Second)
 8 hostTickRate                21762696399        # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
 9 hostMemory                       760348        # Number of bytes of host memory used (Byte)
10 simInsts                        2839787        # Number of instructions simulated (Count)
11 simOps                          5020729        # Number of ops (including micro ops) simulated (Count)
12 hostInstRate                     258807        # Simulator instruction rate (inst/s) ((Count/Second))
13 hostOpRate                       457568        # Simulator op (including micro ops) rate (op/s) ((Count/Second))
```

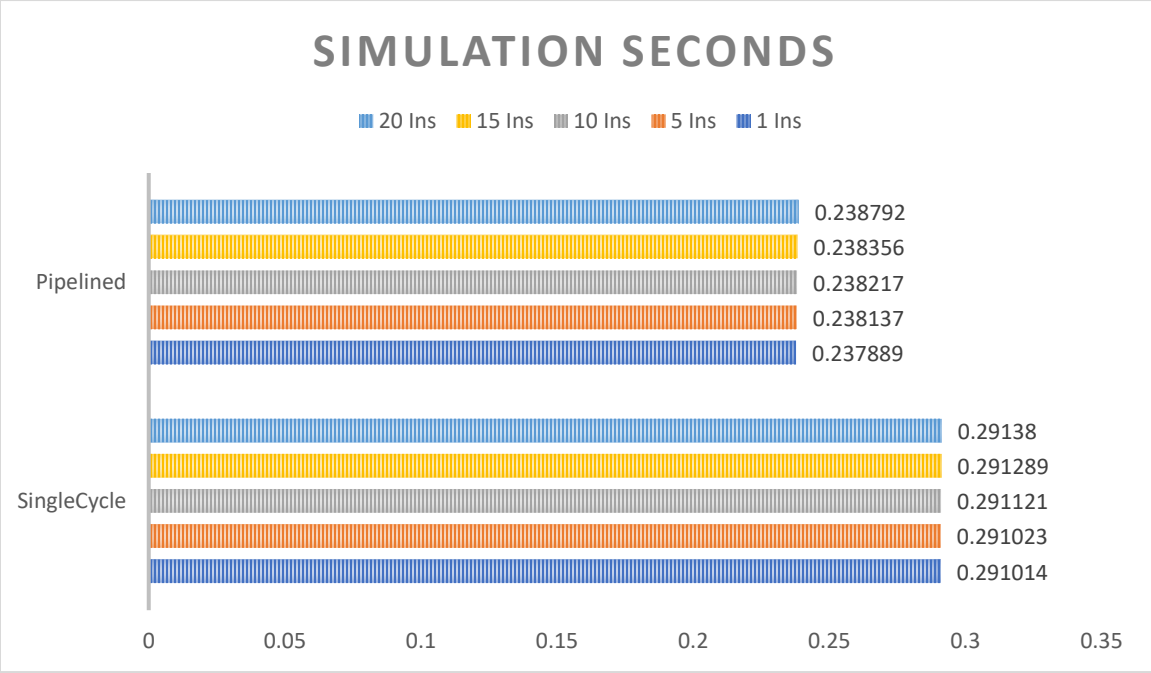
```
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
The value of a + b is 5
The value of b * c is 12
The value of c / a is 2
The value of d - a is 3
The value of e - a is 4
```

نتایج شبیه سازی برای پردازنده RISC-V به صورت SingleCycle

SIM Model(Single Cycle)	simSeconds	simTicks	finalTick	simFreq	hostseconds	hostTickRate
X86TimingSimple-DDR3_1600_8x8(1 instruction)	0.291014	231013647000	231013647000	1000000000000	13.03	22329954616
X86TimingSimple-DDR3_1600_8x8(5 instruction)	0.291023	291022678000	291022678000	1000000000000	13.37	21773613296
X86TimingSimple-DDR3_1600_8x8(10 instruction)	0.291121	291120634000	291120634000	1000000000000	13.61	21395011876
X86TimingSimple-DDR3_1600_8x8(15 instruction)	0.291289	291289434000	291289434000	1000000000000	13.15	22158068985
X86TimingSimple-DDR3_1600_8x8(20 instruction)	0.291380	291380161000	291380161000	1000000000000	32.31	21897990916

نتایج شبیه سازی برای پردازنده RISC-V به صورت Pipelined

SIM Model(Single Cycle)	simSeconds	simTicks	finalTick	simFreq	hostseconds	hostTickRate
X86TimingSimple-DDR3_1600_8x8(1 instruction)	0.237889	237888666000	237888666000	1000000000000	10.63	22373124450
X86TimingSimple-DDR3_1600_8x8(5 instruction)	0.238137	238136798000	238136798000	1000000000000	11.05	21556013012
X86TimingSimple-DDR3_1600_8x8(10 instruction)	0.238217	238217185000	238217185000	1000000000000	10.92	21814473221
X86TimingSimple-DDR3_1600_8x8(15 instruction)	0.238356	238355562000	238355562000	1000000000000	10.88	21904208419
X86TimingSimple-DDR3_1600_8x8(20 instruction)	0.238792	238791592000	238791592000	1000000000000	10.97	21762696399



" تحلیل داده ها با استفاده از نمودار و جداول "

در این نوع پردازنده که به ۲صورت پیاده سازی شده یعنی"SingleCycle" و " Pipelined " زمانی را میتوانیم جهت نشان دادن تاثیر پایپ لاین شدن این پردازنده استفاده کنیم به نام زمان SimSeconds(Simulation Seconds) .

- SimSeconds : زمان شبیه سازی را نشان میدهد .

با توجه به جداول موجود که از خروجی gem5 بدست آمده میتوان فهمید که در تمامی حالات اجرای برنامه یعنی از اجرای ۱ دستورالعمل تا اجرای ۲۰ دستورالعمل زمان شبیه سازی به صورت چشم گیری برای اجرای دستورات در حالت پایپ لاین شده کاهش میابد .

پی نوشت : تعداد دستورات در هر اجرا و نیز نوع دستورات از لحاظ منطقی و ریاضی بودن به دلخواه انتخاب شده است .

پی نوشت : تعداد دستورات برای ۵ حالت ۱, ۵, ۱۰, ۱۵و ۲۰ بررسی شده و دستورات شامل عملیات های ضرب و تقسیم و جمع و تفریق است .

ممکن است در اجرای یک دستورالعمل در ۲ حالت "SingleCycle" و "Pipelined " تفاوت چندانی دیده نشود و حتی در حالت پایپ لاین نشده یعنی همان SingleCycle ما زمان شبیه سازی کمتری نسبت به حالت پایپ لاین شده یعنی همان Pipelined داشته باشیم چرا که ما برای پایپ لاین کردن پردازنده بین گام های مختلف یکسری رجیستر ها یا همان registerfile جهت ذخیره داده های

میانی قرار دادیم که خواندن از آنها و بازنویسی در آنها میتواند برای یک دستور زمان زیادی را نسبت به حالتی که دستورات در پردازنده پایپ لاین نشده اجرا میشود برای ما اضافه کند

در حالت کلی در پردازنده پایپ لاین شده پس از پر شدن پایپ لاین ما به ازای زمان اجرای طولانی ترین بخش یک دستورالعمل که Clock پایپ لاین بر مبنای آن تنظیم میشود یک دستور خروجی خواهیم داشت و دستورات با همپوشانی اجرا شده و زمان اجرای کلی یک برنامه که شامل چندین Instruction هست به طور چشم گیری کاهش میابد .