



401243095

آراس ولیزاده

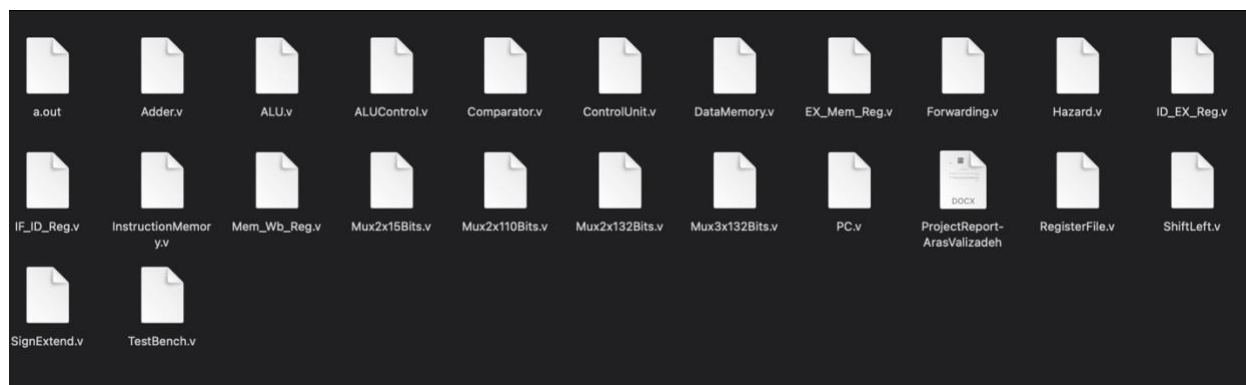
به نام خدا

گزارش پروژه معماری کامپیوتر

در این پروژه با ساختار پایپ لاین که در طول تدریس آشنا شده بودیم با استفاده از وریلاگ آن را پیاده سازی می‌کنیم .


ساختار کلی بدین صورت است که تمامی کامپوننت های تعریف شده با وریلاگ در یک فایل جمع شده و تمامی کامپوننت ها در فایل testbench گرده هم می‌آوریم و پردازنده رو در آن قسمت سر هم می‌کنیم .

ساختار کلی :



برای بخش ابتدایی پروژه ، باید دستور های مورد نظر را در حافظه دستورات نگهداری کنیم برای این کار در حافظه دستورات به ازای pc+4 که به ایندکس های مختلف مموری اشاره می‌کند دستورات را بر اساس opcode های پیش فرض تعریف کرده و درون حافظه دستورات نگهداری می‌کنیم .

در ابتدا دو دستور load ، برای اینکه مقادیر پایه $f(0)$ و $f(1)$ را به رجیستر های موجود بیاوریم ، سپس تا زمانی که به پنجمین جمله تابع فیبوناچی برسیم و آن را محاسبه می‌کنیم .



```

module InstructionMemory(
    input clock,
    input [31:0] pc ,
    output reg [31:0] readData
);
reg [31:0] instructionMemory [0:1060];
initial begin
    instructionMemory[0] = 32'h8C000000 ; //lw
    instructionMemory[4] = 32'h8C010001 ; //lw
    instructionMemory[8] = 32'h03FEF020 ; // i++
    instructionMemory[12] = 32'h00011020 ; // add 2
    instructionMemory[16] = 32'h00221820 ; // add 3
    instructionMemory[20] = 32'h03FEF020 ; // i++
    instructionMemory[24] = 32'h00622020; // add 4
    instructionMemory[28] = 32'h03FEF020 ; // i++
    instructionMemory[32] = 32'h00832820; // add 5
    instructionMemory[36] = 32'h03FEF020 ; // i++
    instructionMemory[40] = 32'h28AA0006 ; //lsti
    // 0010 1000 1010 1010 0000 0000 0000 0110
    instructionMemory[44] = 32'h13C50002 ; // bedq
    instructionMemory[48] = 32'hAC010018 ; // sw
    // 1010 1100 0010 0000 0000 0000 0001 1000
end
always @ (pc)begin
    readData <= instructionMemory[pc];
end
endmodule

```

دستورات طبق اپ کد هایی که دارند به بخش کنترل یونیت فرستاده شده و در نهایت سیگنال های کنترلی متناسب با هر دستور ساخته می شود .

```
module ControlUnit (
    input [5:0] opCode ,
    output reg registerDestination,
    output reg branch,
    output reg memoryRead,
    output reg memoryToRegister,
    output reg [3:0] ALUop,
    output reg memoryWrite,
    output reg AluSrc,
    output reg registerWrite,
    input reset
);
always @(posedge reset)begin
    registerDestination <= 1'b0;
    branch <= 1'b0;
    memoryRead <= 1'b0;
    memoryToRegister <= 1'b0;
    ALUop <= 4'b0000;
    memoryWrite <= 1'b0;
    AluSrc <= 1'b0;
    registerWrite <= 1'b0;
end
always@(opCode) begin
    case (opCode)
        6'b000000: // Rtype
        begin
            registerDestination<=1 ;
            branch<=0 ;
            memoryRead<=0 ;
            memoryToRegister<=0 ;
            memoryWrite<=0 ;
            AluSrc<=0 ;
            registerWrite<=1 ;
            ALUop<=4'b0010 ;
        end
        6'b001010: //slti
        begin
            registerDestination<=0 ;
            branch<=0 ;
            memoryRead<=0 ;
            memoryToRegister<=0 ;
            memoryWrite<=0 ;
            AluSrc<=1 ;
            registerWrite<=1 ;
            ALUop<=4'b0101 ;
        end
        6'b100011: begin // lw
            registerDestination<=0 ;
            branch<=0 ;
            memoryRead<=1 ;
            memoryToRegister<=1 ;
            memoryWrite<=0 ;
            AluSrc<=1 ;
            registerWrite<=1 ;
            ALUop<=4'b0000 ;
        end
        6'b101011: begin //sw
            $display("Store detected");
            branch<=0 ;
            memoryRead<=0 ;
            memoryToRegister<=0 ;
            memoryWrite<=1 ;
            AluSrc<=1 ;
            registerWrite<=0 ;
            ALUop<=4'b0000 ;
        end
        6'b000100: //beq
        begin
            branch<= 1;
            memoryRead<=0 ;
            memoryToRegister<=0 ;
            memoryWrite<=0 ;
            AluSrc<=0 ;
            registerWrite<=0 ;
            ALUop<=4'b0001 ;
        end
    endcase
end
endmodule
```

سیگنال های کنترلی ساخته شده متناسب با هر دستور در به صورت متوالی درون رجیستر های بین استیج ها هدایت می شوند . سینگال کنترلی واحد ای ال یو به واحد کنترلی آن پاس می دهیم و برای ساخت سینگال کنترلی خود ای ال یو متناسب با نوع دستور و ۶ بیت انتهایی دستور که فانکشن می باشد ، سینگال کنترلی را برای ای ال یو میسازیم .

```
1 module ALUControl(
2     input [3:0] ALUOp,
3     input [5:0] funct,
4     output reg [3:0] ALUControl
5 );
6 always @(*)begin
7     case (ALUOp)
8         4'b0000: begin
9             ALUControl = 4'b0000;
10        end
11        4'b0001: begin
12            ALUControl = 4'b0001;
13        end
14        4'b0101: begin
15            ALUControl = 4'b1000;
16        end
17        4'b0010: begin
18            case (funct)
19                6'b100000: begin
20                    ALUControl = 4'b0000;
21                end
22                6'b100010: begin
23                    ALUControl = 4'b0001;
24                end
25                6'b101010: begin
26                    ALUControl = 4'b1000;
27                end
28                default: ALUControl = 4'bxxxx;
29            endcase
30        end
31        default: ALUControl = 4'bxxxx;
32    endcase
33 end
34 endmodule
35
```

در واحد کنترلی ای ال یو با توجه به سیگنال ورودی عملی که ال ای یو باید انجام دهد را مشخص میکنیم دو دستور برنچ و کم کردن دو عدد از یکدیگر با هم همپوشانی دارند در نتیجه با یک سینگال یکسان که در ۴ بیت ساخته می شود این دو دستور را هندل میکنیم ، ای ال یو به شکل زیر پیاده سازی شده است :

```
1 module ALU32Bit(
2     input wire signed [31:0] data1,
3     input wire signed [31:0] data2,
4     input wire [3:0] ALUControl,
5     input wire [4:0] shiftAmount,
6     input wire reset,
7     output reg overflow,
8     output reg zero,
9     output reg signed [31:0] result
10 );
11 wire [31:0] neg_data2;
12 assign neg_data2 = -data2;
13 always @(posedge reset)begin
14     zero <= 1'b0;
15 end
16 always @(ALUControl, data1, data2) begin
17     case (ALUControl)
18         4'b0000: begin // add
19             $display("data 1 : %d , data2: %d "
20                 , data1,data2);
21             result <= data1 + data2;
22             if (data1[31] == data2[31] && result[31] == ~
23                 data1[31]) begin
24                 overflow <= 1'b1;
25             end
26             else begin
27                 overflow <= 1'b0;
28             end
29         end
30         4'b0001: begin // sub also used for branch
31             result <= data1 + neg_data2;
32             #50
33             if (result == 0 )begin
34                 $display("beq taken");
35             end
36             else begin
37                 $display("beq notTaken %d %d"
38                     , data1 ,data2);
39             end
40         end
41         4'b1000: begin // less
42             if (data1 < data2)begin
43                 $display("is less than");
44                 result <= 1;
45             end
46             else begin
47                 result <= 0;
48             end
49         end
50     endcase
51     if (data1 == data2) begin
52         zero <= 1'b1;
53     end
54     else
55         zero <= 1'b0;
56 end
57 endmodule
```

به طور کلی باید تمامی هازارد های موجود در پایپ لاین باید رفع شوند برای همین منظور دو واحد هازارد و فورواردینگ درون پردازنده قرار داده شده اند تا سیگنال های متناسب با دستور و نوع هازارد (۱ یا ۲) را ایجاد کند و تمامی ورودی های ماکس های مختلف از جمله قبل از ای ال یو را کنترل کند. دو نوع هازارد را با بخش زیر کنترل میکنیم :

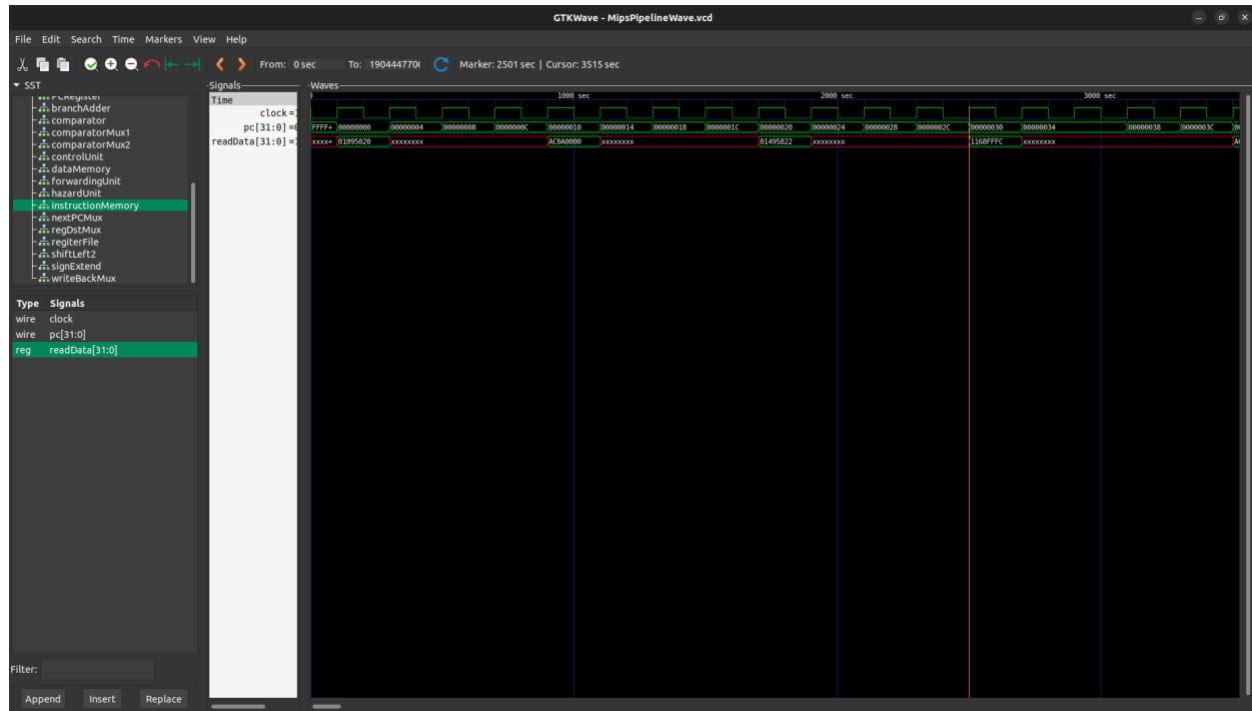
```
always@(ID_ExMemRead or ID_Ex_Rt or IF_ID_Instr) begin
    if (ID_ExMemRead && (holdPC == 1'b0) && (holdIF_ID == 1'b0)) begin
        if (ID_Ex_Rt == IF_ID_Instr[25:21] || ID_Ex_Rt == IF_ID_Instr[20:15]) begin
            holdPC <= 1;
            holdIF_ID <= 1;
            muxSelector <= 1;
        end
    end
    // beq opcode
    else if ((IF_ID_Instr[31:26] == 6'b000100) && (holdPC == 1'b0) && (holdIF_ID == 1'b0)) begin
        holdPC <= 1;
        holdIF_ID <= 1;
        muxSelector <= 1;
    end
    else begin
        holdPC <= 0;
        holdIF_ID <= 0;
        muxSelector <= 0;
    end
end
```

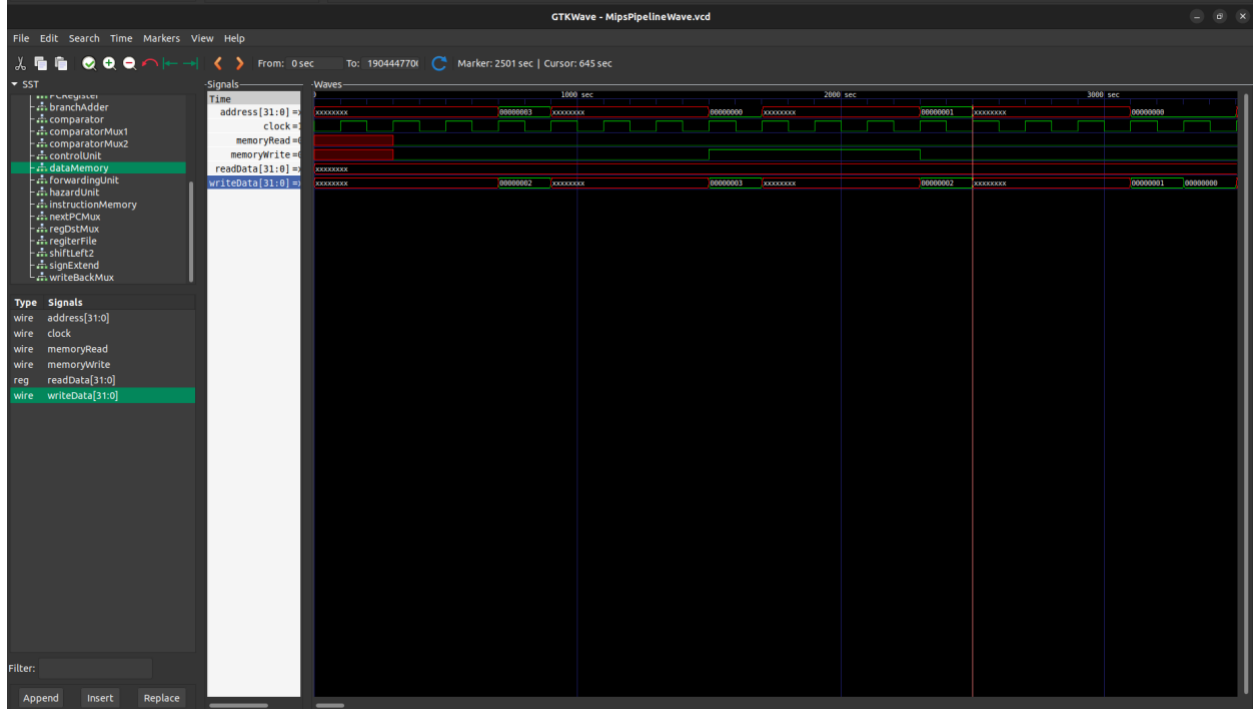
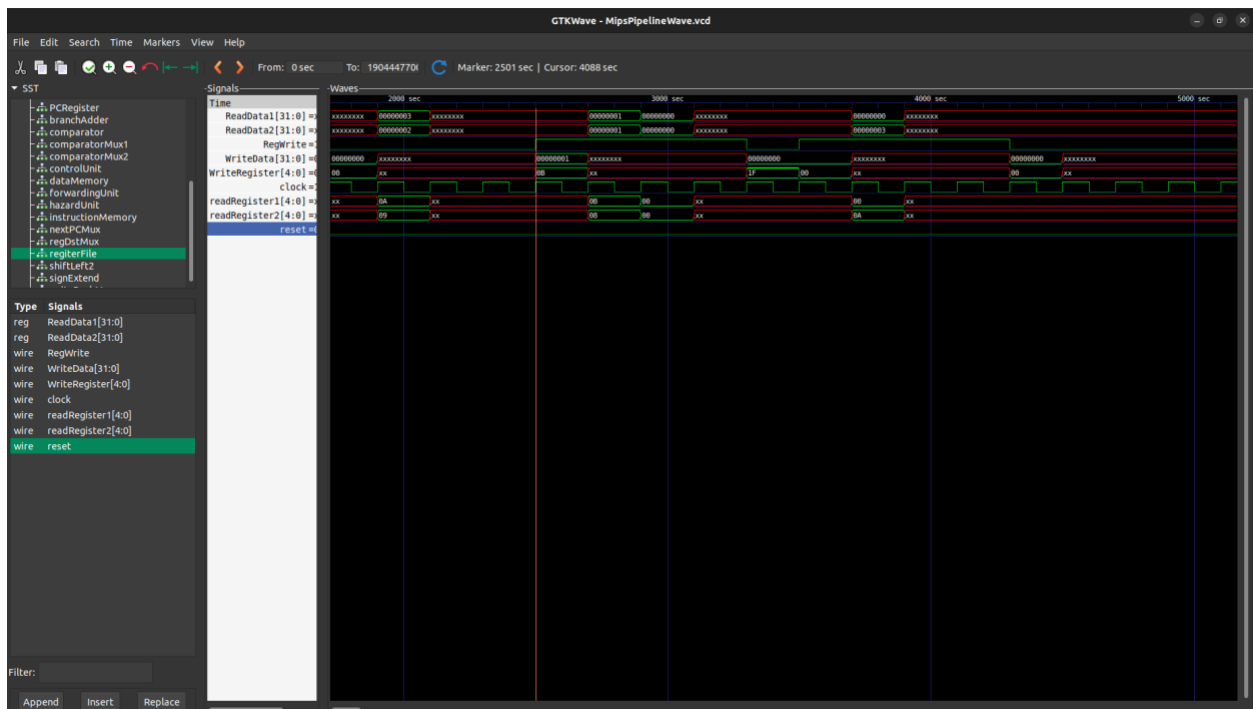
و همچنین در قسمت فورواردینگ داریم :

```
1  always@(EX_MemRegwrite or EX_MemWriteReg or
    Mem_WbRegwrite or Mem_WbWriteReg or ID_Ex_Rs or ID_Ex_Rt)
    begin
2      //forwarding from ALU to ALU & from ALU to ID stage
3      if(EX_MemRegwrite && EX_MemWriteReg) begin
4          if (EX_MemWriteReg==ID_Ex_Rs)begin
5              upperMux_sel<=2'b10;
6              comparatorMux1Selector<=2'b01;
7          end
8          else //no forwarding
9              begin
10                 upperMux_sel<=2'b00;
11                 comparatorMux1Selector<=2'b00;
12             end
13             if(EX_MemWriteReg==ID_Ex_Rt)begin
14                 lowerMux_sel<=2'b10;
15                 comparatorMux2Selector<=2'b01;
16             end
17             else //no forwarding
18                 begin
19                     lowerMux_sel<=2'b00;
20                     comparatorMux2Selector<=2'b00;
21                 end
22             end
23         else if (Mem_WbRegwrite && Mem_WbWriteReg)
            //forwarding from Memorystage to ALU & from Memorystage to
            ID stage
24             begin
25                 if ((Mem_WbWriteReg==ID_Ex_Rs) &&
                    (EX_MemWriteReg!=ID_Ex_Rs))
26                     begin
27                         upperMux_sel<=2'b01;
28                         comparatorMux1Selector<=2'b10;
29                     end
30                     else //no forwarding
31                         begin
32                             upperMux_sel<=2'b00;
33                             comparatorMux1Selector<=2'b00;
34                         end
35                     if((Mem_WbWriteReg==ID_Ex_Rt) && (EX_MemWriteReg
                        ==ID_Ex_Rt) )
36                         begin
37                             lowerMux_sel<=2'b01;
38                             comparatorMux2Selector<=2'b10;
39                         end
40                     else //no forwarding
41                         begin
42                             lowerMux_sel<=2'b00;
43                             comparatorMux2Selector<=2'b00;
44                         end
45                     end
46                 else begin
47                     //No forwarding
48                     upperMux_sel<=2'b00;
49                     lowerMux_sel<=2'b00;
50                     comparatorMux1Selector<=2'b00;
51                     comparatorMux2Selector<=2'b00;
52                 end
53             end
54         end
55     end
```

به طور کلی پروژه شامل قسمت های مختلفی دیگر از جمله رجیستر های بین دو استیج و می باشد . حال برای تست برنامه فیبوناچی ما در ابتدا دو دستور لود را امتحان میکنیم بدین شکل که در مموری دو مقدار پایه فیبوناچی ۰ و ۱ را قرار می دهیم و در ادامه با دستور اد مقادیر بالاتر برنامه را حساب میکنیم و در نهایت با رسیدن به پنجمین جمله فیبوناچی برنچ میکنیم . برای تست دستور SW هم حاصل نتیجه مقایسه دو رجیستر را درون یک رجیستر سوم ریخته و مقدار آن را درون مموری ذخیره می کنیم .

که خروجی برنامه و طول موج های آن به شکل زیر می باشد :






```

register 0 :      0
register 1 :      1
register 2 :      1
register 3 :      2
register 4 :      x
register 5 :      x
i :            3
register 10 (lst) :      x
address :      x
writeData :      x
data 1 :      x , data2:      x
data 1 :      1 , data2:      3
data 1 :      x , data2:      x
memory 1 :      1
address :      4
writeData :      3
register 0 :      0
register 1 :      1
register 2 :      1
register 3 :      2
register 4 :      3
register 5 :      x
i :            4
is less than
beq taken
memory 1 :      1
address :      0
writeData :      x
register 0 :      0
register 1 :      1
register 2 :      1
register 3 :      2
register 4 :      3
register 5 :      5
i :            5
register 10 (lst) :      1
Stroe detected
data 1 :      1 , data2:      0
memory 1 :      1
address :      0

```

بخش ب پروژه بدین صورت می باشد که باید دستور ذخیره در حافظه را در طول ۲ کلاک انجام بدیم . برای بخش ب پروژه با توجه به خواسته سوال از این روش استفاده شده که در کلاک اول ۱۶ بیت کم ارزش و در کلاک دوم ۱۶ بیت پر ارزش دیتا را درون مموری سیو کنیم . در استفاده از کامپوننت های قبلی تغییری ایجاد نمیکنیم و خواسته سوال را در بخش دیتا مموری هندل میکنیم اما چون با این کار ماهیت پایپ لاین به ۶ استیج تغییر می یابد نیاز به تعریف یک سری از رجیستر ها سنکرون با کلاک برای انتقال دیتا از استیج قبلی به استیج بعدی می باشند .

برای این کار دیتا مموری به شکل زیر در می آید :

```
1
2  always@(negedge clk) begin
3      $display("memory: %b \n", memory[address2]);
4      if(MemWrite1==1)
5          memory[address1][31:16]<=writeData1;
6      if(MemWrite2==1)
7          memory[address2][15:0]<=writeData2;
8  end
9
10 always@(address1 or address2 or Memread1 or Memread2)
    begin
11     if(Memread1==1)
12         readData1=memory[address1][31:16];
13     if(Memread2==1)
14         readData2=memory[address2][15:0];
15
```

[illegible]