

VERILOG LAB2 RISC-V



AUGUST 17, 2023

OMNIA MOHAMED SELIM IBRAHIM

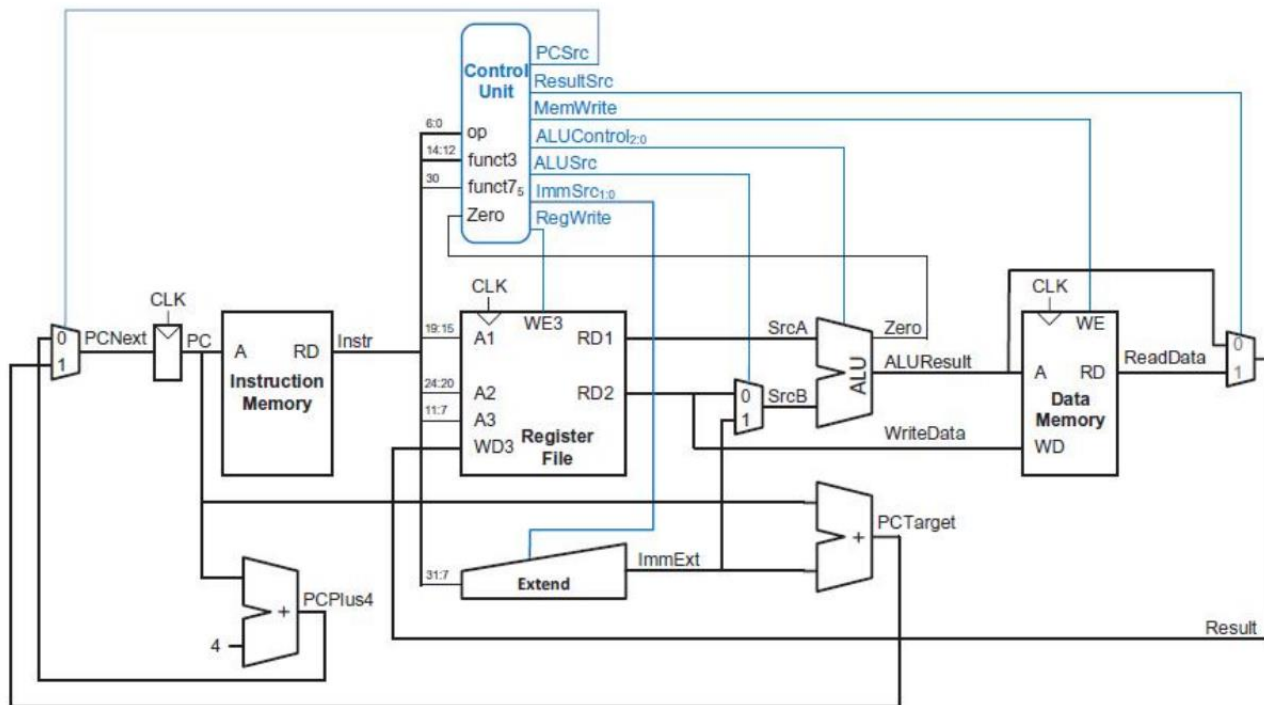
omniamselim@gmail.com

Contents

Architecture	2
ALU	2
Truth Table:	2
Screenshot of the code:	3
Data_Memory	3
Screenshot of the code:	3
Instruction_Memory	4
Screenshot of the code:	4
Program_Counter	4
Screenshot of the code:	4
Register_File	4
Screenshot of the code:	5
Sign_extend	5
Screenshot of the code:	5
Control_Unit	5
Alu decoder truth table	6
Main decoder truth table	6
Screenshot of the code:	7
RISCV_ITI	11
Screenshot of the code:	11
TB_RISCV	12
Machine code	12
Simulation Results	14

Architecture

Implementation of a 32-bit single-cycle microarchitecture RISC-V processor is based on Harvard Architecture.



ALU

An Arithmetic/Logical Unit (ALU) combines a variety of mathematical and logical operations into a single unit.

Truth Table:

ALU_control	Function
000	$ALU_result = SrcA + SrcB$
001	$ALU_result = SrcA \ll SrcB$
010	$ALU_result = SrcA - SrcB$
100	$ALU_result = SrcA \wedge SrcB$
101	$ALU_result = SrcA \gg SrcB$
110	$ALU_result = SrcA SrcB$
111	$ALU_result = SrcA \& SrcB$

Screenshot of the code:

```
module ALU(input [31:0]SrcA,input [31:0]SrcB,[2:0]ALU_control,
           output reg [31:0] ALU_result ,output reg zero,output reg sign);

always@(*) begin
case (ALU_control)
    3'b000: ALU_result=SrcA+SrcB;
    3'b001:ALU_result=SrcA<<SrcB;
    3'b010:ALU_result=SrcA-SrcB;
    3'b100:ALU_result=SrcA^SrcB;
    3'b101:ALU_result=SrcA>>SrcB;
    3'b110:ALU_result=SrcA|SrcB;
    3'b111:ALU_result=SrcA&SrcB;
    default :ALU_result=0;
endcase

if(ALU_result==0) begin
    zero=1;
end
else begin
    zero=0;
end
sign=ALU_result[31];

end
endmodule
```

Data_Memory

It has a single read/write port, and when its write enable, WE, is asserted, it writes data WD into address A on the rising edge of the clock.

Screenshot of the code:

```
module Data_Memory(input [31:0]A,input [31:0]WD,input WE,input clk,output reg [31:0]RD) ;
reg [31:0]data_memory[0:63];
always@(posedge clk) begin
if (WE) begin
    data_memory[A[31:2]]<=WD;
end
end
always@(*)begin
    RD=data_memory[A];
end

endmodule
```

Instruction_Memory

Each instruction memory has a single read port.

It reads the 32-bit data (i.e., instruction) from the 32-bit instruction address input, A, onto the read data output, RD.

Screenshot of the code:

```
module Instruction_Memory(input [31:0]pc,output reg [31:0] instruction );
reg [31:0]instr_mem[0:63];
initial begin
    $readmemh("rom_riscv.txt",instr_mem,0,20);
end
always@(*) begin
    instruction=instr_mem[pc/4];
end
endmodule
```

Program_Counter

Screenshot of the code:

```
module Program_Counter(input clk,input load,input areset,input PCSrc,input [31:0]ImmExt,output reg[31:0]pc_out );
reg [31:0] nextpc;
always@(posedge clk or negedge areset)begin
    if(!areset) begin
        pc_out<=0;
    end
    else begin
        if(load) begin
            pc_out<=nextpc;
        end
        else pc_out<=pc_out;
    end
end
always@(*)begin
    case(PCSrc)
        1'b1:nextpc=pc_out+ImmExt;
        1'b0:nextpc=pc_out+32'h00000004;
        default: nextpc=pc_out+32'h00000004;
    endcase
end
endmodule
```

Register_File

The 32-bit registers are stored in the register file.

The register file has one input write port (WD3), two read output ports (RD1 and RD2), and RD1 and RD2 are read without respect to the clock edge.

At the clock's rising edge, the register file is read asynchronously and written synchronously.

Screenshot of the code:

```
module Register_File (input [4:0]A1,input [4:0]A2,input [4:0]A3, input [31:0]WD3,input WE3,input clk,
                     output reg [31:0] RD1,output reg [31:0]RD2);
    reg [31:0]regfile[0:31];
    integer i;
    initial begin
    for (i=0;i<32;i=i+1) begin
        regfile[i]=0;
    end
    end
    always @(posedge clk )
    begin
        if (WE3) begin
            regfile[A3] <= WD3;
        end
    end
    always@(*)begin
        RD1 <= regfile[A1];
        RD2 <= regfile[A2];
    end
endmodule
```

Sign_extend

Screenshot of the code:

```
module Sign_extend(input[1:0]ImmSrc,input[31:7]Instr,output reg[31:0]ImmExt);
always@(*) begin
    case(ImmSrc)
        2'b00:ImmExt={{20{Instr[31]}},Instr[31:20]};
        2'b01:ImmExt={{20{Instr[31]}},Instr[31:25],Instr[11:7]};
        2'b10:ImmExt={{20{Instr[31]}},Instr[7],Instr[30:25],Instr[11:8],1'b0};
        default: ImmExt=0;
    endcase
end
endmodule
```

Control_Unit

Based on the opcode, funct3, and funct7 fields of the corresponding instructions (Instr14:12 and Instr30), the control unit computes the control signals. The source of most of the control information is the opcode, as well as R-type and I-type instructions.the ALU op is determined by using the funct3 and funct7 fields.

Alu decoder truth table:

ALUOp	Funct3	{OpCode5,funct7}	ALUControl
2'b00	xx	xx	3'b000
2'b01	3'b000	xx	3'b010
	3'b001	xx	3'b010
	3'b100	xx	3'b010
	default	xx	3'b010
2'b10	3'b000	2'b11	3'b010
		2'b00	3'b000
		2'b01	
		2'b10	
	3'b001	xx	3'b001
	3'b100	xx	3'b100
	3'b101	xx	3'b101
	3'b110	xx	3'b110
	3'b111	xx	3'b111
	default	xx	3'b000
default	xxx	xx	3'b000

Main decoder truth table

OpCode	RegWrite	ImmSrc	ALUSrc	MemWrite
7'b000_0011	1'b1	2'b00	1'b1	1'b0
7'b010_0011	1'b0	2'b01	1'b1	1'b1
7'b011_0011	1'b1	xx	1'b0	1'b0
7'b001_0011	1'b1	2'b00	1'b1	1'b0
7'b110_0011	1'b0	2'b10	1'b0	1'b0
default	1'b0	2'b00	1'b0	1'b0

OpCode	ResultSrc	Branch	ALUOp
7'b000_0011	1'b1	1'b0	2'b00
7'b010_0011	1'b1	1'b0	2'b00
7'b011_0011	1'b0	1'b0	2'b10
7'b001_0011	1'b0	1'b0	2'b10
7'b110_0011	1'b0	1'b1	2'b01
default	1'b0	1'b0	2'b00

Screenshot of the code:

```

module Control_Unit ( input zero,sign,[31:0]instr,output load,
    output reg PCSrc,ALUSrc,ResultSrc,MemWrite,RegWrite,[2:0]ALUControl,[1:0]ImmSrc );
    reg [6:0]OpCode;
    reg [2:0]funct3;
    reg [1:0]ALUOp;
    reg funct7,Branch;

    always @(*) begin
        OpCode=instr[6:0];
        funct3=instr[14:12];
        funct7=instr[30];
    end

```



```

// alu decoder
] always @(*) begin
]     case (ALUOp)
]         2'b00:begin
]             ALUControl=3'b000;//add
]         end
]         2'b01:begin
]             case (funct3)
]                 3'b000: ALUControl = 3'b010;
]                 3'b001: ALUControl = 3'b010;
]                 3'b100: ALUControl = 3'b010;
]                 default: ALUControl = 3'b000;
]             endcase
]         end
]         2'b10:begin
]             case (funct3)
]                 3'b000:begin
]                     if ({OpCode[5],funct7}==2'b11)
]                         ALUControl=3'b010;//sub
]                     else
]                         ALUControl=3'b000;//add
]                     end
]                 3'b001:begin
]                     ALUControl=3'b001;//Shift Left
]                 end
]                 3'b100:begin
]                     ALUControl=3'b100;//xor
]                 end
]                 3'b101:begin
]                     ALUControl=3'b101;//Shift right
]                 end
]                 3'b110:begin
]                     ALUControl=3'b110;//or
]                 end
]                 3'b111:begin
]                     ALUControl=3'b111;//and
]                 end
]                 default: begin
]                     ALUControl=3'b000;
]                 end
]             endcase
]         end
]         default: begin
]             ALUControl=3'b000;
]         end
]     endcase
] end
end

```

```

// main decoder
always @(*) begin
    case (OpCode)
        7'b000_0011:begin //loadWord
            RegWrite=1'b1;
            ImmSrc=2'b00;
            ALUSrc=1'b1;
            MemWrite=1'b0;
            ResultSrc=1'b1;
            Branch=1'b0;
            ALUOp=2'b00;

        end
        7'b010_0011:begin //storeWord
            RegWrite=1'b0;
            ImmSrc=2'b01;
            ALUSrc=1'b1;
            MemWrite=1'b1;
            Branch=1'b0;
            ALUOp=2'b00;

        end
        7'b011_0011:begin //R-Type
            RegWrite=1'b1;
            ALUSrc=1'b0;
            MemWrite=1'b0;
            ResultSrc=1'b0;
            Branch=1'b0;
            ALUOp=2'b10;

        end
        ---
        7'b001_0011:begin //I-Type
            RegWrite=1'b1;
            ImmSrc=2'b00;
            ALUSrc=1'b1;
            MemWrite=1'b0;
            ResultSrc=1'b0;
            Branch=1'b0;
            ALUOp=2'b10;

        end
        7'b110_0011:begin //Branch Instructions
            RegWrite=1'b0;
            ImmSrc=2'b10;
            ALUSrc=1'b0;
            MemWrite=1'b0;
            Branch=1'b1;
            ALUOp=2'b01;

        end
        default: begin
            RegWrite=1'b0;
            ImmSrc=2'b00;
            ALUSrc=1'b0;
            MemWrite=1'b0;
            ResultSrc=1'b0;
            Branch=1'b0;
            ALUOp=2'b00;

        end
    endcase
end

```

```

//PCSrc
always @(*) begin
    if (Branch==1'b1)
    begin
        case (funct3)
        3'b000:begin //Beq
            PCSrc=Branch & zero;
        end
        3'b001:begin //Bnq
            PCSrc=Branch & ~(zero);
        end
        3'b100:begin //Blt
            PCSrc= Branch & sign;
        end
        default: begin
            PCSrc=1'b0;
        end
        endcase
    end
    else
    begin
        PCSrc=1'b0;
    end
end

assign load=1'b1;

endmodule

```

RISCV_ITI

This is the top module.

Screenshot of the code:

```
module RISCV_ITI (  
    input clk, areset  
);  
  
    reg [31:0] SrcB, Result;  
    wire [31:0] SrcA, ALUResult;  
    wire [2:0] ALUControl;  
    wire zero, sign, PCSrc, load, RegWrite, WE, ALUSrc, ResultSrc;  
    wire [31:0] ImmExt, PC, Instr, WD, RD;  
    wire [1:0] ImmSrc;  
  
    Instruction_Memory InstrMem1 (PC, Instr);  
    Sign_extend Sign1 (ImmSrc, Instr[31:7], ImmExt);  
    ALU alu1 (SrcA, SrcB, ALUControl, ALUResult, zero, sign);  
    Program_Counter pc1 (clk, load, areset, PCSrc, ImmExt, PC);  
    Data_Memory DataMemory1 (ALUResult, WD, WE, clk, RD);  
    Control_Unit CU1 (zero, sign, Instr, load, PCSrc, ALUSrc, ResultSrc, WE, RegWrite, ALUControl, ImmSrc);  
    Register_File RF1 (Instr[19:15], Instr[24:20], Instr[11:7], Result, RegWrite, clk, SrcA, WD);  
  
    always @(*) begin  
        if (ALUSrc==0)  
            SrcB=WD;  
        else  
            SrcB=ImmExt;  
    end  
  
    always @(*) begin  
        if (ResultSrc==1)  
            Result=RD;  
        else  
            Result=ALUResult;  
    end  
  
endmodule
```

TB_RISCV

```
module tb_RISCV();

reg clk, areset;
integer i;

RISCV_ITI riscv(.clk(clk), .areset(areset));

always
begin
#10; clk=~clk;
end

initial
begin
for(i=0; i<21; i=i+1)
    $display("%h", riscv.InstrMem1.instr_mem[i]);
clk=0;
areset=0;
#5;
areset=1;
#3000;


$stop;
end
endmodule
```

Machine code

```
00004033
00000093
00100113
00100193
00100213
00000293
00a00313
00000393
00418c63
00110133
404181b3
00229393
0023a023
00420a63
002080b3
004181b3
00229393
0013a023
00128293
fc62cae3
00000000
```

Code :

```
#include <iostream>
using namespace std;
int main()
{ int x = 0;
  int y = 1;
  int sel = 1;
  for (int i = 0; i < 10; i++)
  {
    if (sel == 1)
    { x = x + y;
      sel = 2;
      cout << x << endl; }
    else
    { y = x + y;
      sel = 1;
      cout << y << endl;}
  }
}
```

 assembly code of FIBONACCI series - Notepad

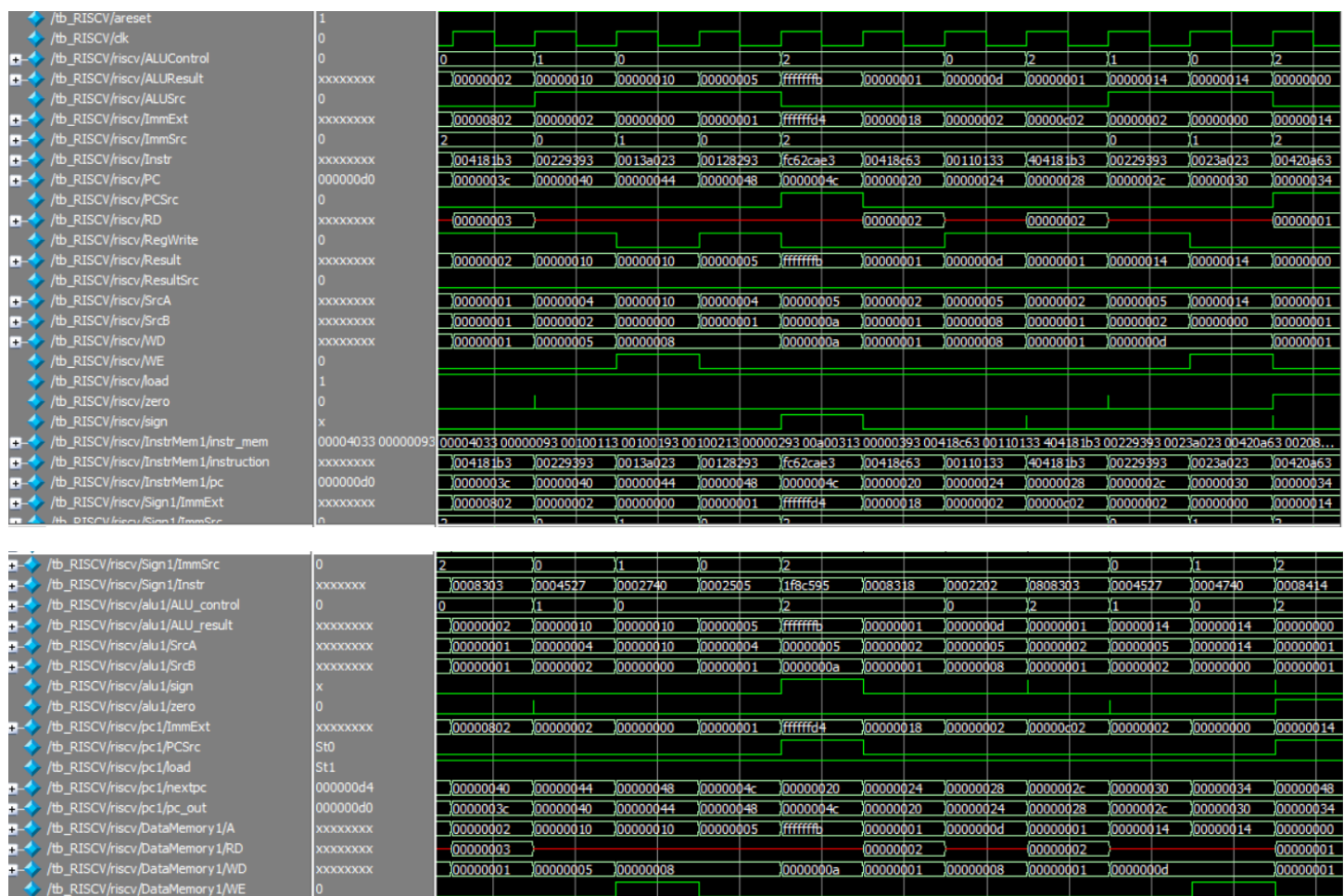
File Edit Format View Help

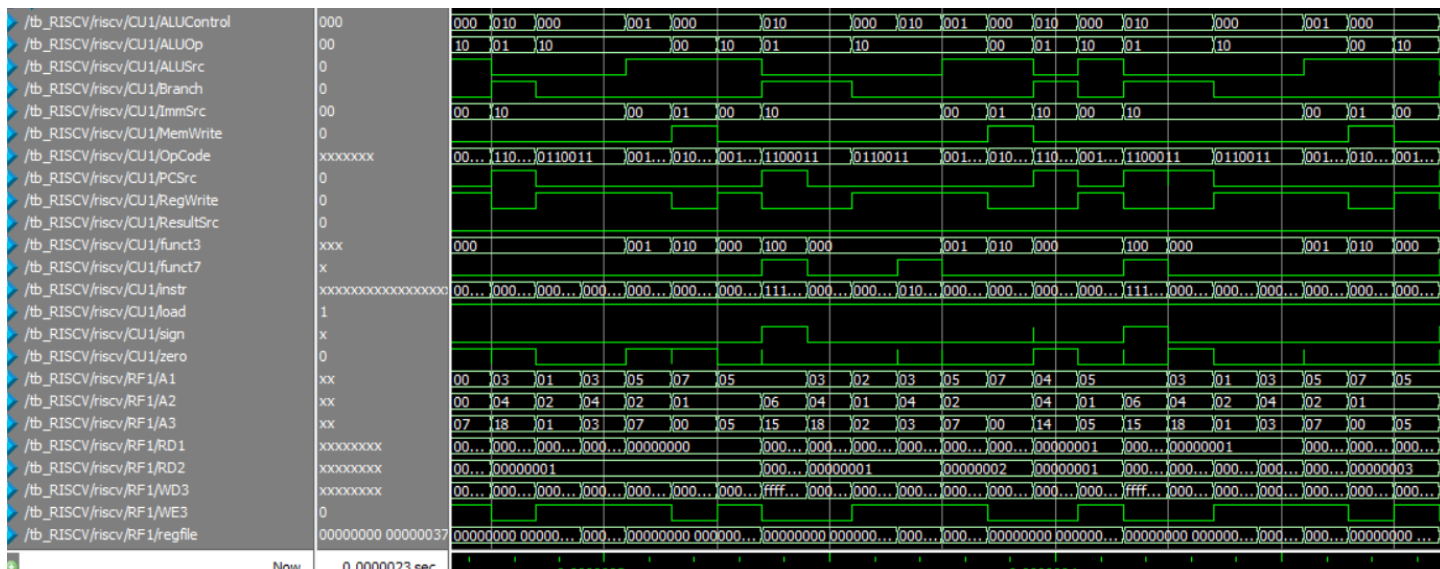
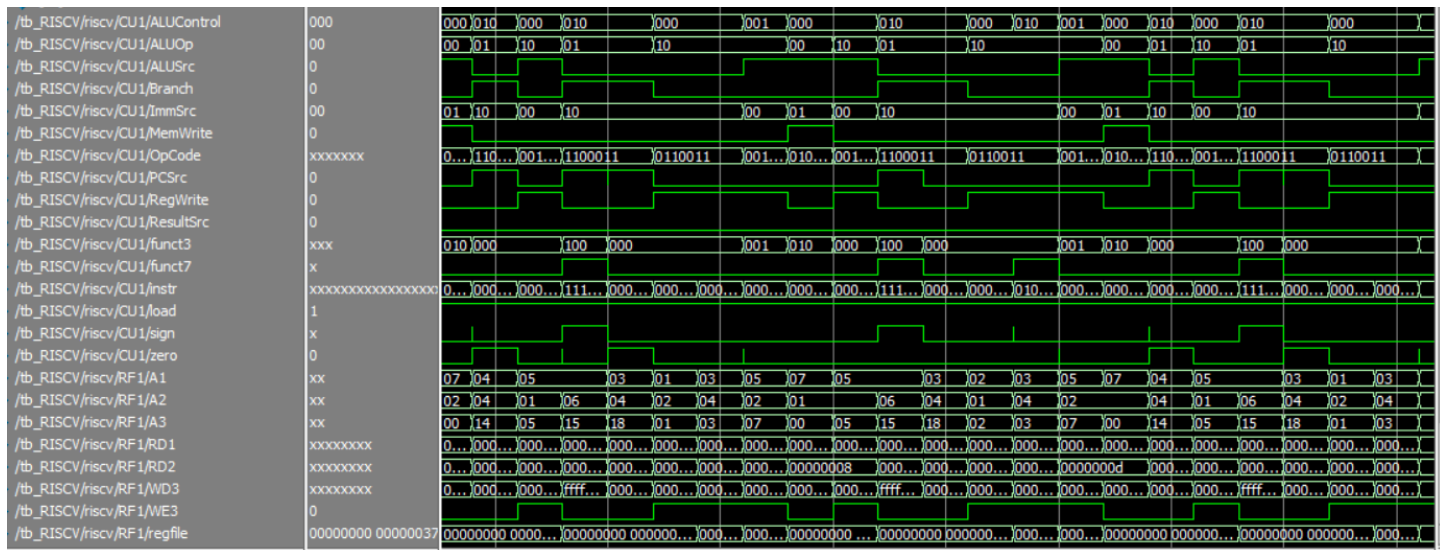
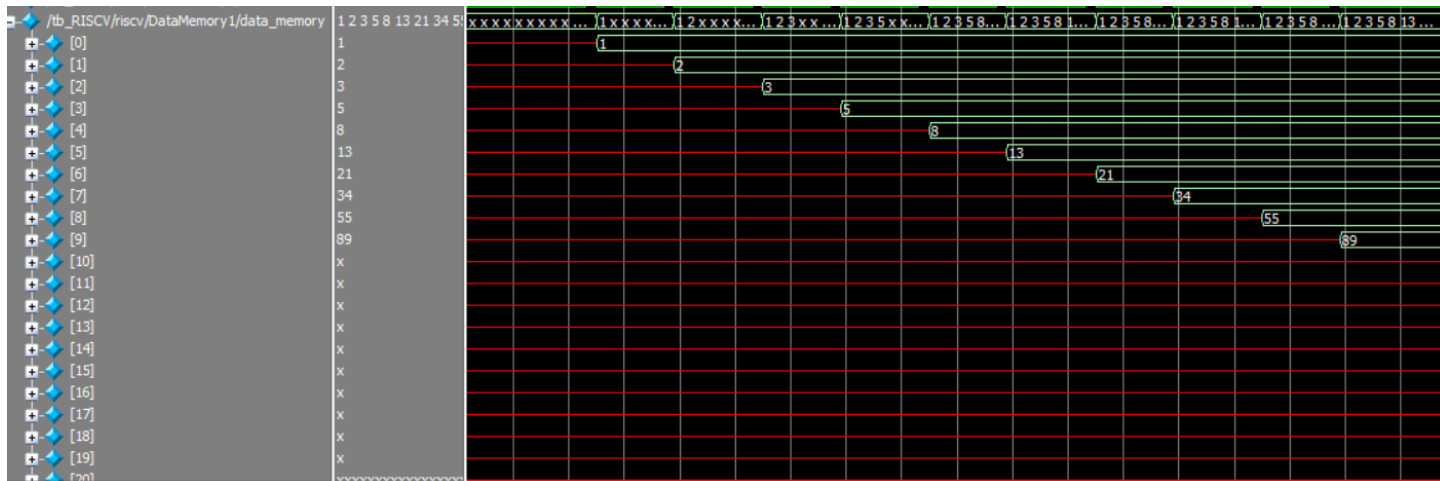
```
0: xor x0,x0,x0
4: addi x1,x0,0
8: addi x2,x0,1
12: addi x3,x0,1
16: addi x4,x0,1
20: addi x5,x0,0
24: addi x6,x0,10
28: addi x7,x0,0 |
loop:
32: beq x3,x4,eq
36: add x2,x2,x1
40: sub x3,x3,x4
44: slli x7,x5,2
48: sw x2,0(x7)
52: beq x4,x4,endloop
eq:
56: add x1,x1,x2
60: add x3,x3,x4
64: slli x7,x5,2
68: sw x1,0(x7)
endloop:
72: addi x5,x5,1
76: blt x5,x6,loop
80: halt
```

Simulation Results

```
VSIM 40> run
```

```
# 00004033
# 00000093
# 00100113
# 00100193
# 00100213
# 00000293
# 00a00313
# 00000393
# 00418c63
# 00110133
# 404181b3
# 00229393
# 0023a023
# 00420a63
# 002080b3
# 004181b3
# 00229393
# 0013a023
# 00128293
# fc62cae3
# 00000000
```





Now 0.0000023 sec

