2019《FPGA应用实验》实验报告

实验编号： 大作业 实验时间： 2020.6.20

实验名称： 简单RISC CPU设计

班级： F1703603 学号： 517021911065 姓名： 刘浩文

1、实验平台

采用Xilinx 公司的FPGA集成开发环境 Xilinx ISE Design Suite 10.1 sp3，实验开发板为Xilinx Spartan-3E FPGA Starter Kit。

由于使用的物理机是macOS系统，Xilinx ISE Design Suite 10.1 sp3得在Windows虚拟机中使用，很不方便。因此先在macOS系统中使用Sublime Text v3.2.2完成源文件的编写，使用命令行工具Icarus Verilog version 10.3和波形查看工具Scansion v1.12完成编译、测试与仿真工作，再转到Windows7虚拟机的Xilinx ISE Design Suite 10.1 sp3完成仿真与综合工作。

2、实验设计要求：

设计一个RISC CPU，其能够完成有限Fibonacci数列的计算。

3、模块设计框图

CPU与外围电路（PDF上的电路图有小错误，CPU的ADDRESS端口应为输出端口）。

手机屏幕截图

描述已自动生成

CPU内部电路与模块（由于改为**同步复位**，所以PC需要额外的CLK\_CTRL输入端口）。

手机屏幕截图

描述已自动生成

4、实验原理：

4.1．简化 RISC CPU 的设计

确定数据总线的宽度：8 位，一个字节的宽度，一次传送一个字节的数据。

寻址方式：采用最简单的方式 —— 直接寻址。

直接寻址：数据总是放在存储器中，寻址单元的地址由指令中的操作数给出。

4.2．指令系统

指令宽度：指令寄存器的宽度；16位，两个字节。

指令：操作码(opcode，3 位，占据 [15:13])+ 操作数(operand，13 位，占据[12:0])；在这个CPU中，操作数只是地址。

操作码的位宽：决定CPU的指令的数目。

操作数的位宽：在直接寻址方式下，决定了寻址空间的大小。

寻址空间：ROM 和 RAM 的大小，以字节为单位。

图片包含 游戏机, 钟表, 画

描述已自动生成

指令：

 HALT:

 停机。该指令将空一个指令周期，什么都不作。

 JRZ:

 先判断当前 ALU 中的运算结果是否为 0

 若为 0，则跳过下一条语句，否则继续执行

 If ALU(data) = 0

 (PC+1)+1

 ADD:

 将累加器(ACC)中的数据与指令中地址所指的存储器单元中的数据相加

 结果仍送回累加器中

 ACC<=ACC(data) + MEMORY[address](data)

 MEMORY

 ROM/RAM

 AND:

 将累加器(ACC)中的数据与指令中地址所指的存储器单元中的数据相与

 结果仍送回累加器中

 ACC<=ACC(data) & MEMORY[address](data)

 XOR:

 异或。将累加器(ACC)中的数据与指令中给出的地址的存储器单元中的数据异或

 结果仍送回累加器中

 ACC<=ACC(data) ^ MEMORY[address](data)

 LOAD:

 读数据。将指令中给出的地址指定的存储单元中的数据装入累加器

 ACC<=MEMORY[address](data)

 STORE:

 写数据:

 将累加器中的数据写入指令中给出的地址的存储单元

 MEMORY[address]<=ACC(data)

 JUMP:

 无条件跳转

 跳转到指令中给出的目的地址，继续执行

指令编码:

parameter HALT = 3'b000,

JRZ = 3'b001,

ADD = 3'b010,

AND = 3'b011,

XOR = 3'b100,

LOAD = 3'b101,

STORE = 3'b110,

JUMP = 3'b111;

4.3．RISC CPU 的组成

CPU内部电路与模块见第三节模块设计框图。

时钟周期和指令周期：

 时钟周期

 计算机中最基本、最小的时间单位

 在一个时钟周期内，CPU 仅完成一个最基本的动作

 所有的时序逻辑电路都是由基本存储单元(触发器)构成

 触发器的状态通常只能在时钟脉冲边沿到来时，才能转换

 通常，一个时钟周期转换一次

 指令周期

 执行一条指令所需的时间

 通常，CPU 执行一条指令，需要完成若干个基本操作(动作)

 一个指令周期需要若干个时钟周期

控制单元：

手机屏幕截图

描述已自动生成

程序计数器：

手机屏幕截图

描述已自动生成

指令寄存器：

手机屏幕截图

描述已自动生成手机屏幕截图

描述已自动生成

累加器：

手机屏幕截图

描述已自动生成

算术逻辑单元：

手机屏幕截图

描述已自动生成

总线控制器：

手机屏幕截图

描述已自动生成

地址多路器：

手机屏幕截图

描述已自动生成

时钟发生器：

手机屏幕截图

描述已自动生成

4.4．RISC CPU及外围电路

存储器地址分配:寻址空间 8 K:0000H ~ 1FFFH

 采用程序和数据分开存放方式

 程序存放在 ROM 中，程序运行时产生的数据和运算结果存放在 RAM 中

 ROM:0FFFH —— 0000H, 大小 4K bytes

 RAM:1FFFH —— 1000H, 大小 4K bytes

CPU与外围电路见第三节模块设计框图。

5、Verilog 模块设计

**状态机控制器：**

/\* File: machine\_ctrl.v \*/

`timescale 1ns / 100ps

module machine\_ctrl (

output enable,

input clk\_fetch, clk\_ctrl,

input reset

);

reg enable1, enable2;

//always @(posedge clk\_fetch or posedge reset) begin : proc\_enable

always @(posedge clk\_fetch) begin

enable1 <= 1'b1;

end

always @(negedge clk\_ctrl) // 引入时钟信号，使能够同步复位

if(reset) begin

enable2 <= 1'b0;

end else enable2 <= 1'b1;

assign enable = enable1 & enable2;

/\*

always @(enable)

$display("enable-------------------------------------------------- %b", enable);

\*/

endmodule

**状态机：**

/\* File: machine.v \*/

`timescale 1ns / 100ps

module machine (

output reg halt,

output reg inc\_pc,

output reg load\_pc,

output reg load\_ir,

output reg load\_acc,

output reg bus\_enable,

output reg mem\_rd,

output reg mem\_wr,

input enable,

input clk\_ctrl,

input alu\_zero\_flag,

input [2:0] opcode

);

parameter HALT = 3'b000,

JRZ = 3'b001,

ADD = 3'b010,

AND = 3'b011,

XOR = 3'b100,

LOAD = 3'b101,

STORE = 3'b110,

JUMP = 3'b111;

parameter CYCLE0 = 3'b000,

CYCLE1 = 3'b001,

CYCLE2 = 3'b010,

CYCLE3 = 3'b011,

CYCLE4 = 3'b100,

CYCLE5 = 3'b101,

CYCLE6 = 3'b110,

CYCLE7 = 3'b111;

reg [2:0] state;

always @(negedge clk\_ctrl)

if (enable != 1'b1)

begin

{load\_acc, load\_ir, load\_pc, inc\_pc} <= 4'b0000;

{halt, bus\_enable, mem\_wr, mem\_rd} <= 4'b0000;

state <= CYCLE0;

end

else if(enable == 1'b1) instruction\_cycle;

task instruction\_cycle;

casex (state)

CYCLE0:

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0100;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0001;

//$display("取指高8位, %b", opcode);

state <= CYCLE1;

end

CYCLE1:

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0101;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0001;

//$display("取指低8位, %b", opcode);

state <= CYCLE2;

end

CYCLE2:

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("空一个时钟周期, %b", opcode);

state <= CYCLE3;

end

CYCLE3:

begin

if (opcode == HALT) begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0001;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b1000;

//$display("halt下一条指令的地址已准备好, %b", opcode);

end else begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0001;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("下一条指令的地址已准备好, %b", opcode);

end

state <= CYCLE4;

end

CYCLE4:

begin

if (opcode == JUMP) begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0010;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("loadpc准备跳转, %b", opcode);

end else

if (( opcode == ADD) || (opcode == AND) || ( opcode == XOR) || (opcode == LOAD))

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0010;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0001;

//$display("使用IR中的地址读操作数, %b", opcode);

end

else

if (opcode == STORE) begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0100;

//$display("写数据已准备好, %b", opcode);

end else begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("CYCLE4空转, %b", opcode);

end

state <= CYCLE5;

end

CYCLE5:

begin

if ((opcode == ADD) || (opcode == AND) || (opcode == XOR) || (opcode == LOAD))

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b1000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0001;

//$display("使用IR中的地址读操作数 acc装入数据, %b", opcode);

end

else

if ((opcode == JRZ) && (alu\_zero\_flag))

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0001;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("JRZ判断准备跳过下一条语句, %b", opcode);

end

else

if (opcode == JUMP)

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0011;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("IR地址送入PC中准备跳转, %b", opcode);

end

else

if (opcode == STORE)

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0110;

//$display("写数据已准备好, %b", opcode);

end

else

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("CYCLE5空转, %b", opcode);

end

state <= CYCLE6;

end

CYCLE6:

begin

if (opcode == STORE)

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0100;

//$display("acc数据写到数据总线上, %b", opcode);

end

else

if ((opcode == ADD) || (opcode == AND) || (opcode == XOR) || (opcode == LOAD))

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0001;

//$display("读出存储器的数据, %b", opcode);

end

else

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("CYCLE6空转, %b", opcode);

end

state <= CYCLE7;

end

CYCLE7:

begin

if ((opcode == JRZ) && (alu\_zero\_flag == 1))

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0001;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("JRZ跳过一条指令, %b", opcode);

end

else

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("CYCLE7空转, %b", opcode);

end

state <= CYCLE0;

end

default:

begin

{ load\_acc, load\_ir, load\_pc, inc\_pc } <= 4'b0000;

{ halt, bus\_enable, mem\_wr, mem\_rd } <= 4'b0000;

//$display("default空转, %b", opcode);

state <= CYCLE0;

end

endcase

endtask

endmodule

**程序计数器：**

/\* File: PC.v \*/

`timescale 1ns / 100ps

module PC (

output reg [12:0] addr\_pc,

input inc\_pc,

input clk\_ctrl,

input reset,

input load\_pc,

input [12:0] addr\_ir

);

reg flag;

//always @(posedge inc\_pc or posedge reset) begin : proc\_addr\_pc

always @(posedge inc\_pc or negedge flag) begin

if (!flag) addr\_pc <= 13'b0;

else

if (load\_pc) begin // load\_pc信号为真，载入指定的地址（JUMP）

addr\_pc <= addr\_ir;

//$display("pc %b", addr\_ir);

end

else begin // 一般情况下pc自增

addr\_pc <= addr\_pc + 1'b1;

//$display("pc %b", addr\_pc + 1'b1);

end

end

always @(negedge clk\_ctrl) // 引入时钟信号，使其能同步复位

if(reset) begin

flag <= 1'b0;

end else flag <= 1'b1;

/\*

always @(addr\_pc)

$display("pc %b", addr\_pc);

\*/

endmodule

**指令寄存器：**

/\* File: IR.v \*/

`timescale 1ns / 100ps

module IR (

output reg [2:0] opcode,

output reg [12:0] addr\_ir,

input clk\_ctrl,

input reset,

input load\_ir,

input [7:0] data\_bus

);

reg state;

always @(posedge clk\_ctrl) begin : proc\_state

if(reset) begin

opcode <= 3'b0;

addr\_ir <= 13'b0;

state <= 1'b0;

end

else

if (load\_ir)

casex(state)

1'b0: begin // 取指令高8位

opcode <= data\_bus[7:5];

addr\_ir[12:8] <= data\_bus[4:0];

state <= 1'b1;

//$display("ir state0 %b %b", data\_bus[7:5], data\_bus[4:0]);

end

1'b1: begin // 取指令低8位

addr\_ir[7:0] <= data\_bus;

state <= 1'b0;

//$display("ir state1 %b", data\_bus);

end

default:

begin

opcode <= 3'bx;

addr\_ir <= 13'bx;

state <= 1'bx;

//$display("ir statex %b %b", 3'bx, 13'bx);

end

endcase

else state <= 1'b0;

end

endmodule

**累加器：**

/\* File: ACC.v \*/

`timescale 1ns / 100ps

module ACC (

output reg [7:0] acc\_out,

input clk\_ctrl,

input reset,

input load\_acc,

input [7:0] data\_bus

);

//always @(posedge clk\_ctrl or posedge reset)

always @(posedge clk\_ctrl) // 中间数据寄存器，暂存要计算的数据

if (reset) acc\_out <= 8'b0;

else

if (load\_acc) acc\_out <= data\_bus;

endmodule

**算术逻辑单元：**

/\* File: ALU.v \*/

`timescale 1ns / 100ps

module ALU (

output reg [7:0] alu\_out,

output alu\_zero\_flag,

input clk\_alu,

input [2:0] opcode,

inout [7:0] acc\_out,

input [7:0] data\_bus

);

parameter HALT = 3'b000,

JRZ = 3'b001,

ADD = 3'b010,

AND = 3'b011,

XOR = 3'b100,

LOAD = 3'b101,

STORE = 3'b110,

JUMP = 3'b111;

assign alu\_zero\_flag = !alu\_out; // JRZ判断用

always @(posedge clk\_alu)

casex (opcode) // 依据指令进行相应计算

HALT: alu\_out <= acc\_out;

JRZ: alu\_out <= acc\_out;

ADD: alu\_out <= acc\_out + data\_bus;

AND: alu\_out <= acc\_out & data\_bus;

XOR: alu\_out <= acc\_out ^ data\_bus;

LOAD: alu\_out <= data\_bus;

STORE: alu\_out <= acc\_out;

JUMP: alu\_out <= acc\_out;

default:

alu\_out <= 8'bx;

endcase

endmodule

**总线控制器：**

/\* File: bus\_controller.v \*/

`timescale 1ns / 100ps

module bus\_controller (

output [7:0] data\_bus,

input [7:0] acc\_out,

input bus\_enable

);

assign data\_bus = (bus\_enable) ? acc\_out : 8'bz;

endmodule

**地址多路器：**

/\* File: address\_mux.v \*/

`timescale 1ns / 100ps

module address\_mux (

output [12:0] address,

input clk\_fetch,

input [12:0] addr\_pc, addr\_ir

);

assign address = (clk\_fetch) ? addr\_pc : addr\_ir;

endmodule

**时钟发生器：**

/\* File: clock\_generator.v \*/

`timescale 1ns / 100ps

module clock\_generator (

output clk\_ctrl,

output reg clk\_fetch, clk\_alu,

input clock, reset

);

reg [7:0] state;

parameter idle = 8'b0000\_0000,

cycle0 = 8'b0000\_0001,

cycle1 = 8'b0000\_0010,

cycle2 = 8'b0000\_0100,

cycle3 = 8'b0000\_1000,

cycle4 = 8'b0001\_0000,

cycle5 = 8'b0010\_0000,

cycle6 = 8'b0100\_0000,

cycle7 = 8'b1000\_0000;

assign clk\_ctrl = ~clock; // 总控时钟

always @(negedge clock) begin : proc\_state

if(reset) begin

clk\_fetch <= 0; // 控制CPU启动，取指或取数据

clk\_alu <= 0; // 控制

state <= idle;

end else

case (state)

idle : state <= cycle0;

cycle0 : begin clk\_alu <= ~clk\_alu; state <= cycle1; end

cycle1 : begin clk\_alu <= ~clk\_alu; state <= cycle2; end

cycle2 : state <= cycle3;

cycle3 : begin clk\_fetch <= ~clk\_fetch; state <= cycle4; end

cycle4 : state <= cycle5;

cycle5 : state <= cycle6;

cycle6 : state <= cycle7;

cycle7 : begin clk\_fetch <= ~clk\_fetch; state <= cycle0; end

default : state <= idle;

endcase

end

endmodule

**RISC CPU模块：**

/\* File: CPU.v \*/

`timescale 1ns / 100ps

`include "ACC.v"

`include "address\_mux.v"

`include "ALU.v"

`include "bus\_controller.v"

`include "clock\_generator.v"

`include "IR.v"

`include "machine.v"

`include "machine\_ctrl.v"

`include "PC.v"

module CPU (

output mem\_rd, mem\_wr,

output [12:0] address,

output halt,

inout [7:0] data\_bus,

input clock, reset

);

/\*

always @(\*) begin

$display("addr\_pc %b", addr\_pc);

end

always @(\*) begin

$display("instruction %b", { opcode, addr\_ir });

end

\*/

wire [2:0] opcode;

wire clk\_ctrl, clk\_fetch, clk\_alu;

wire sm\_enable, inc\_pc, load\_pc, load\_acc, load\_ir, bus\_enable, alu\_zero\_flag;

wire [12:0] addr\_pc, addr\_ir;

wire [7:0] acc\_out, alu\_out;

clock\_generator m\_clkgen(.clk\_ctrl(clk\_ctrl), .clk\_fetch(clk\_fetch), .clk\_alu(clk\_alu), .clock(clock), .reset(reset));

machine\_ctrl m\_mc(.enable(sm\_enable), .clk\_fetch(clk\_fetch), .clk\_ctrl(clk\_ctrl), .reset(reset));

machine m\_sm(.halt(halt), .inc\_pc(inc\_pc), .load\_pc(load\_pc), .load\_ir(load\_ir), .load\_acc(load\_acc), .bus\_enable(bus\_enable), .mem\_rd(mem\_rd), .mem\_wr(mem\_wr), .clk\_ctrl(clk\_ctrl), .enable(sm\_enable), .alu\_zero\_flag(alu\_zero\_flag), .opcode(opcode));

PC m\_pc(.addr\_pc(addr\_pc), .inc\_pc(inc\_pc), .clk\_ctrl(clk\_ctrl), .reset(reset), .load\_pc(load\_pc), .addr\_ir(addr\_ir));

IR m\_ir(.opcode(opcode), .addr\_ir(addr\_ir), .clk\_ctrl(clk\_ctrl), .reset(reset), .load\_ir(load\_ir), .data\_bus(data\_bus));

ACC m\_acc(.acc\_out(acc\_out), .clk\_ctrl(clk\_ctrl), .reset(reset), .load\_acc(load\_acc), .data\_bus(alu\_out));

ALU m\_alu(.alu\_out(alu\_out), .alu\_zero\_flag(alu\_zero\_flag), .clk\_alu(clk\_alu), .opcode(opcode), .acc\_out(acc\_out), .data\_bus(data\_bus));

address\_mux m\_addrmux(.address(address), .clk\_fetch(clk\_fetch), .addr\_pc(addr\_pc), .addr\_ir(addr\_ir));

bus\_controller m\_bus\_ctrl(.data\_bus(data\_bus), .bus\_enable(bus\_enable), .acc\_out(acc\_out));

endmodule

**地址译码器：**

/\* File: address\_decoder.v \*/

`timescale 1ns / 100ps

module address\_decoder (

output reg sel\_rom, sel\_ram,

input [12:0] address

);

always @(address)

casex (address)

13'b1\_xxxx\_xxxx\_xxxx: { sel\_rom, sel\_ram } <= 2'b01;

13'b0\_xxxx\_xxxx\_xxxx: { sel\_rom, sel\_ram } <= 2'b10;

default: { sel\_rom, sel\_ram } <= 2'b00;

endcase

endmodule

**ROM设计：**

/\* File: ROM.v \*/

`timescale 1ns / 100ps

module ROM (

output [7:0] data\_bus,

input [12:0] address,

input sel\_rom, mem\_rd

);

reg [7:0] memory[13'b0\_1111\_1111\_1111:0];

assign data\_bus = (mem\_rd && sel\_rom) ? memory[address] : 8'bz;

endmodule

**RAM设计：**

/\* File: RAM.v \*/

`timescale 1ns / 100ps

module RAM (

inout [7:0] data\_bus,

input [12:0] address,

input sel\_ram, mem\_rd, mem\_wr

);

reg [7:0] memory[13'b0\_1111\_1111\_1111:0];

integer offset = 13'b1\_0000\_0000\_0000;

assign data\_bus = (mem\_rd && sel\_ram) ? memory[address - offset] : 8'bz;

always @(posedge mem\_wr)

memory[address - offset] <= data\_bus;

endmodule

**测试台设计：**

/\* File: test\_bench.v \*/

`timescale 1ns / 100ps

`include "CPU.v"

`include "address\_decoder.v"

`include "RAM.v"

`include "ROM.v"

module test\_bench;

reg clock, reset;

wire halt;

wire [12:0] address;

wire [7:0] data\_bus;

wire mem\_wr, mem\_rd;

wire sel\_rom, sel\_ram;

reg [7:0] n;

CPU m\_cpu(

.mem\_rd (mem\_rd),

.mem\_wr (mem\_wr),

.address(address),

.halt (halt),

.data\_bus(data\_bus),

.clock (clock),

.reset (reset)

);

address\_decoder m\_ad(

.sel\_rom(sel\_rom),

.sel\_ram(sel\_ram),

.address(address)

);

ROM m\_rom(

.data\_bus(data\_bus),

.address (address),

.sel\_rom (sel\_rom),

.mem\_rd (mem\_rd)

);

RAM m\_ram(

.data\_bus(data\_bus),

.address (address),

.sel\_ram (sel\_ram),

.mem\_rd (mem\_rd),

.mem\_wr (mem\_wr)

);

/\*

always @(address)

$display("address %b", address);

always @(data\_bus)

$display("data\_bus %b", data\_bus);

\*/

initial begin

$dumpfile("RISC\_CPU.vcd");

$dumpvars(0, test\_bench);

$readmemb("Fibonacci.asm", m\_rom.memory);

$display("ROM loaded successfully!");

#1 clock = 1'b0;

forever begin

#10 clock = ~clock;

end

end

initial begin

//reset = 1'b0;

reset = 1'b1;

#100 reset = 1'b0;

#22000

for(n=0; n<=34; n=n+1) $display("ROM: %b; RAM: %b.", m\_rom.memory[n], m\_ram.memory[n]);

$stop;

end

initial begin

$monitor("At time %tns, reset = %b, temp = %b, fn1 = %b, fn2 = %b.", $time, reset, m\_ram.memory[0], m\_ram.memory[1], m\_ram.memory[2]);

end

endmodule

**Fibonacci汇编程序（Fibonacci.asm）：**

**汇编程序（每行16个二进制数字）：**

0000000000000001 // FN1、FN2初始值

LOAD 0000000000000 // 载入FN1值

STORE 1000000000001 // 存入FN1位置

LOAD 0000000000001 // 载入FN2值

STORE 1000000000010 // 存入FN2位置

LOAD 1000000000010 // LOOP: LOAD FN2

STORE 1000000000000 // STORE TEMP

ADD 1000000000001 // ADD FN1

STORE 1000000000010 // STORE FN2

LOAD 1000000000000 // LOAD TEMP

STORE 1000000000001 // STORE FN1

XOR 0000000011101 // XOR LIMIT (0000000011101)

JRZ 0000000000000 // JRZ

JUMP 0000000001010 // JUMP LOOP

HALT 0000011101001 // OVER: HALT (后8位存有LIMIT值)

JUMP 0000000011100 // JUMP OVER

**机器码：**

00000000

00000001

10100000

00000000

11010000

00000001

10100000

00000001

11010000

00000010

10110000

00000010

11010000

00000000

01010000

00000001

11010000

00000010

10110000

00000000

11010000

00000001

10000000

00011101

00100000

00000000

11100000

00001010

00000000

11101001

11100000

00011100

6、试验仿真结果和分析

使用macOS下命令行工具Icarus Verilog version 10.3编译test\_bench.v，编译成功，无ERROR或WARNING：

手机屏幕截图

描述已自动生成

运行上一步编译生成的文件cpu.vpp，得到屏幕输出结果：

电脑萤幕

描述已自动生成电脑屏幕的截图

描述已自动生成

第一张图是CPU对Fibonacci数列的计算过程，和Fibonacci.asm程序设计一致，第二张图是打印出的最后ROM和RAM的部分数据，ROM中是程序机器码，RAM中是计算得到的最终的值：11101001，和LIMIT相同，说明仿真正确。

用波形查看工具Scansion v1.12查看仿真生成的波形文件RISC\_CPU.vcd：

电脑萤幕画面

描述已自动生成

由于CPU内部模块被封装在CPU中，因此CPU内部信号没有在波形文件中存储。为了能对运行过程进行分析，重写另一个测试台文件tb\_analys.v，将CPU内部暴露在测试台中。

编译并运行tb\_analys.v，用波形查看工具Scansion v1.12查看仿真生成的波形文件analys.vcd：

电脑萤幕画面

描述已自动生成

图中标示线指向第一个CLK\_FETCH的上升沿，此时CPU（状态机）开始工作，开始执行第一条指令HALT（其实是FN1的初始值00000000存在ROM的开头）。波形图中指令周期、读写信号符合设计。

在Windows7虚拟机的Xilinx ISE Design Suite 10.1 sp3中综合成功：

电脑萤幕画面

描述已自动生成

电脑萤幕画面

描述已自动生成

电脑萤幕画面

描述已自动生成