课间实验报告封面示例：

**武汉大学计算机学院**

**本科生实验报告**

**计算机组成原理实验**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：计算机组成原理

指 导 教 师 ：计组组实验老师

学 生 学 号 ：2020326660020

学 生 姓 名 ：黄家宇（JoeWong JiaYu）

二○二二年4月

章节标题示例（根据实际实验内容修改章节标题）：

**1.实验内容**

本实验涉及并要求通过对计算机组成与原理课程的知识，要求掌握对知识的综合应用、 加深对cpu系统各模块的工作原理。试验主要是学习使用EDA技术与MIPS单周期cpu技术，以及了解cpu功能实现使用Mars将汇编语言转换成机器语言导入ModelSim进行Simulation & Wave。

内容：

1.增加cpu的功能指令：add/sub/and/or/slt/sltu/addu/subu/addi/ori/lw/sw/beq/j/jal/sll/nor/lui/slti/bne/andi/srl/jr

2.将指令汇编代码通过Mars得到机器码，再进行Simulation对应cpu加载的汇编文件运行正确

# 2实验步骤与分析

## 

## 2.1Verilog HDL：Verilog HDL是一种硬件描述语言，以[文本形式](https://baike.baidu.com/item/%E6%96%87%E6%9C%AC%E5%BD%A2%E5%BC%8F)来描述[数字系统](https://baike.baidu.com/item/%E6%95%B0%E5%AD%97%E7%B3%BB%E7%BB%9F)硬件的结构和行为的语言，用它可以表示逻辑电路图、[逻辑表达式](https://baike.baidu.com/item/%E9%80%BB%E8%BE%91%E8%A1%A8%E8%BE%BE%E5%BC%8F)，还可以表示数字逻辑系统所完成的逻辑功能。

## 2.2MARS：MARS是一个轻量级的交互式开发环境（IDE），用于使用MIPS汇编语言进行编程，编译成机器码，是 MIPS 汇编程序和运行时模拟器。Mars是一个用于 MIPS 汇编语言程序设计的IDE需要 Java 环境支持。

## 2.3ModelSim：Modelsim仿真工具是Model公司开发的。它支持Verilog、VHDL以及他们的混合仿真，它可以将整个程序分步执行，使设计者直接看到他的程序下一步要执行的语句，而且在程序执行的任何步骤任何时刻都可以查看任意变量的当前值，通过仿真结果来验证结果是否正确。

# 3实验结果与总结

## 3.1 总体设计

单周期 CPU：

指令：

• add/sub/and/or/slt/sltu/addu/subu

• addi/ori/lw/sw/beq

• j/jal

• sll/nor/lui/slti/bne/andi/srl/sllv/srlv/jr/jalr

模块组成及介绍：

| **模块** | **简单描述** |
| --- | --- |
| im.v | 存放指令的地方，从外面接受指令地址然后给出指令的具体二进制编码 |
| Ctrl\_encode.v | 定义控制信号(二进制数)的名字 |
| extend.v | 将取出的寄存器值扩展为32位立即数 |
| instruction\_def.v | 定义输入指令(二进制数)与其对应的字符串名字，也是方便辨认和使用 |
| npc.v | 根据控制信号更新pc的模块 |
| dm.v | 内存 |
| MIPS.v | 将所有模块的接口整合在一起的模块 |
| alu.v | 算数计算单位，用来做运算操作 |
| Mux.v | 多路选择器，根据控制信号选择通路，把几个多路选择器模块放到一个代码文件中是为了使文件结构更简单，也容易寻找同类型的模块 |
| PC.v | 根据npc给的值执行读指令操作 |
| ir.v | 让指令地址从0开始就是reset |
| gpr.v | 存放寄存器的值 |
| ctrl.v | 控制信号，所有模块的行为都由此模块控制 |

3.2 PC

**（1）功能描述**

存放指令地址，将pc中当前的地址送入指令寄存器im中读取具体的指令机器码。在pc写信号为1时，可以使用输入的新地址修改寄存器中的内容。

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| Newpc | 输入 | newpc就是下一个时钟周期要用的pc |
| Clk | 输入 | 时钟信号 |
| Reset | 输入 | 置位信号 |
| Oldpc | 输出 | 当前PC，程序寄存器值 |

3.3 gpr

**（1）功能描述**

使用寄存器存储少量的数据，可以通过指令中的字段选择需要读取的寄存器号，分别输出读取的两个寄存器中的数据。也可以在允许写信号为1时，将数据写入寄存器。

**（2）模块接口**

| **信号名** | **方向** | **描述** |
| --- | --- | --- |
| Rs | 输入 | 第一个源操作数寄存器号 |
| Rt | 输入 | 第二个源操作数寄存器号 |
| Rd | 输入 | 存放结果的目的寄存器号 |
| Wd | 输入 | 写入寄存器的操作结果 |
| We | 输入 | 写使能信号 |
| Outa | 输出 | 寄存器组输出一 |
| Outb | 输出 | 寄存器组输出二 |
| shift | 输出 | 寄存器的写使能信号 |

**3.4 ALU**

**（1）功能描述**

ALU主要功能是执行CPU的各种计算，完成对输入数据的进行加法、减法、与、或、左移、右移、乘法、除法运算等.

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| A | 输入 | 操作数1 |
| B | 输入 | 操作数2 |
| Zero | 输入 | ALU零信号 |
| Aluop | 输入 | ALU的控制信号，决定进行哪种计算 |
| C | 输出 | 运算结果 |

**3.5 Ctrl**

**（1）功能描述**

Ctrl是控制信号的单元。根据不同的对应的机器麻，确定该指令为哪种指令，对各种信号进行初始化。

**（2）模块接口**

| **信号名** | **方向** | **描述** |
| --- | --- | --- |
| Opcode | 输入 | Op码用来区分R、I、J型指令 |
| Func | 输入 | 指令功能字段 |
| Zero | 输入 | 判断分支条件是否成立 |
| RegSel [1:0] | 输出 | 写回寄存器号的选择型号 |
| ALUSrc | 输出 | A的ALU来源 |
| MemRead | 输出 | 存储器的可读信号 |
| RegWrite | 输出 | 寄存器的写使能信号 |
| MemWrite | 输出 | 存储器的写使能信号 |
| EXTOp | output | Extend的控制信号 |
| ALUCtrl[4:0] | output | ALU的操作控制信号 |

**3.6 Extend**

**（1）功能描述**

将十六位立即数进行扩展至三十二位。

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| In | 输入 | 16位立即数 |
| Extop | 输入 | 扩展选择信号 |
| Out | 输出 | 32位立即数 |

**3.7 dm**

**（1）功能描述**

存储数据，根据信号进行相应的读写操作

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| Addr | 输入 | 写入的地址 |
| Data\_in | 输入 | 来自寄存器的存入RAM的数据 |
| MemWrite | 输入 | 写入信号 |
| MemRead | 输入 | 从RAM读出来的数据 |
| data\_out | 输出 | 读出的数据 |

**3.8 im**

**（1）功能描述**

Im是指令寄存器，从外面接受指令地址然后给出指令的具体二进制编码。

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| Addr | 输入 | 要获取的指令地址 |
| Out | 输出 | 读取的指令 |

**3.9 NPC**

**（1）功能描述**

通过控制信号来更新pc的模块。

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| old\_PC | 输入 | PC地址 |
| Beq\_imm | 输入 | 跳转指令，执行条件EQ,跳到Imm |
| J\_addr | 输入 | 跳转地址 |
| Beq\_zero | 输入 | 跳转指令，执行条件EQ，给予0 |
| PC\_sel | 输入 | 跳转指令，执行条件EQ,跳到Imm |
| newPC | 输出 | 最终输出给PC的值（下一条指令的地址） |

3.10 IR

**（1）功能描述**

指令寄存器是用来存放指令的，存放当前正在执行的指令，包括指令的操作码，地址码，地址信息

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| Clk | 输入 | 时钟信号 |
| Rst | 输入 | 重置信号 |
| IRWr | 输入 | 编写寄存器 |
| Im\_dout | 输入 | 输入寄存器的数据 |
| Instr | 输出 | 输出寄存器指令 |
| Clk | 输入 | 时钟信号 |

3.11 MIPS

**（1）功能描述**

将其它模块连接组合成完整的CPU。

3.12 Mux

**（1）功能描述**

Mux 功能是在一条 TCP 连接上分发多个 TCP 连接的数据。实现细节详见Mux.Cool。Mux 是为了减少 TCP 的握手延迟而设计，而非提高连接的吞吐量。使用 Mux 看视频、下载或者测速通常都有反效果。Mux 只需要在客户端启用，服务器端自动适配。

**（2）模块接口**

| 信号名 | 方向 | 描述 |
| --- | --- | --- |
| RegDst | 输入 | 经过计算，R型指令送到rd中，I型指令送到rt中。 |
| Instrl\_rs | 输入 | 将指令发送到rs中 |
| Instrl\_rt | 输入 | 将指令发送到rt中 |
| Reg\_rd | 输出 | 输出寄存器rd |

3.13 instruction\_def

**（1）功能描述**

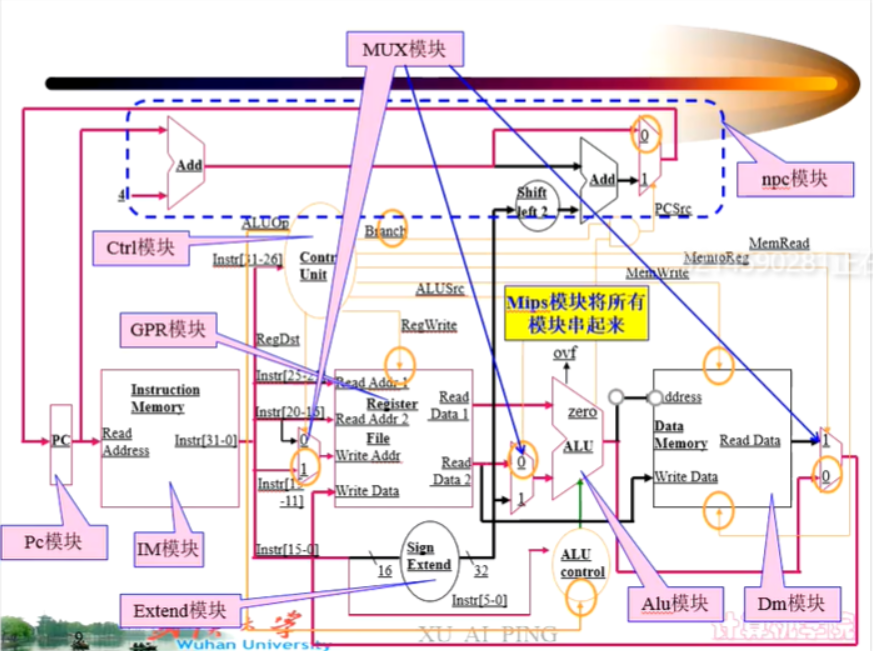
定义输入指令(二进制数)与其对应的字符串名字，也是方便辨认和使用

3.14 ctrl\_encode\_def

**（1）功能描述**

定义控制信号(二进制数)的名字，以方便辨认与使用

# 4 详细设计



4.2 PC

module pc(

input [31:0] newpc,

input clk,

input reset,

output [31:0] oldpc

);

reg [31:0] \_pc;

always@(posedge clk or posedge reset)begin

if(reset == 1'b1) begin

\_pc <= 32'h00003000;

end

else begin

\_pc <= newpc;

end

end

assign oldpc = \_pc;

endmodule

4.3 GPR

module gpr(

input clk,

input rst,

input [4:0] rs,

input [4:0] rt,

input [4:0] rd,

input [31:0] wd,

input we,

input memout,

output [31:0] outa,

output [31:0] outb

);

reg [31:0] Gpr[31:0];

integer i;

always @(posedge clk or posedge rst) begin

if(rst)

for(i = 0; i < 32; i = i+1)

Gpr[i] = 0;

else if(we)

begin

Gpr[rd] = (rd == 0)?0:wd;

$display("R[00-07]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", 0, Gpr[1], Gpr[2], Gpr[3], Gpr[4], Gpr[5], Gpr[6], Gpr[7]);

$display("R[08-15]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", Gpr[8], Gpr[9], Gpr[10], Gpr[11], Gpr[12], Gpr[13], Gpr[14], Gpr[15]);

$display("R[16-23]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", Gpr[16], Gpr[17], Gpr[18], Gpr[19], Gpr[20], Gpr[21], Gpr[22], Gpr[23]);

$display("R[24-31]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X", Gpr[24], Gpr[25], Gpr[26], Gpr[27], Gpr[28], Gpr[29], Gpr[30], Gpr[31]);

$display("R[%4X]=%8X", rd, Gpr[rd]);

end

end

assign outa = memout==1? ( rt == 0 ? 0: Gpr[rt]) :(rs == 0 ? 0: Gpr[rs]);

assign outb = memout==1? ( rs == 0 ? 0: Gpr[rs]) :(rt == 0 ? 0: Gpr[rt]);

endmodule

**4.4 ALU**

`include "ctrl\_encode\_def.v"

module alu(

input [31:0] A,

input [31:0] B,

input [4:0] ALUOp,

output reg [31:0] C,

output reg zero

);

initial begin

C = 32'b0;

zero = 0;

end

always @(A or B or ALUOp)begin

case(ALUOp)

`ALUOp\_ADD: C = A + B; //add

`ALUOp\_SUB: C = A - B; //sub

`ALUOp\_OR: C = A | B;

`ALUOp\_NOR: C = ~(A | B);

`ALUOp\_XOR: C = A ^ B;

`ALUOp\_EQL: zero=(A==B)?1'b1:1'b0;

`ALUOp\_BNE: zero=(A==B)?1'b0:1'b1;

`ALUOp\_AND: C = A & B;

`ALUOp\_SLL: C = A<<(B[10:6]);

`ALUOp\_SLT: C = (A < B) ? 32'd1 : 32'd0;

`ALUOp\_SRL: C = A>>(B[10:6]);

`ALUOp\_SRA: C = ({{31{B[31]}},1'b0} << (~A[4:0]))|(B >> A[4:0]);

`ALUOp\_LUI: C = (B<<16);

`ALUOp\_SLTU: C = ((A<B)?1:0);

`ALUOp\_NOP: C = A;

default: C = A;

endcase

end

endmodule

**4.5 Ctrl**

`include "ctrl\_encode\_def.v"

`include "instruction\_def.v"

module ctrl(

input [5:0] opcode,

input [5:0] func,

output reg [1:0] RegDst,

output reg ALUSrc,

output reg MemRead,

output reg RegWrite,

output reg MemWrite,

output reg [1:0] DatatoReg,

output reg [1:0] PC\_sel,

output reg ExtOp,

output reg [4:0] ALUCtrl,

output reg Memout

);

always @(opcode or func ) begin

case(opcode)

`INSTR\_RTYPE\_OP:begin // R type

case(func)

//addu

`INSTR\_ADDU\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

MemRead = 0;

RegWrite = 1;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_ADDU;

Memout=0;

end

//subu

`INSTR\_SUBU\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

MemRead = 0;

RegWrite = 1;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_SUBU;

Memout=0;

end

// add

`INSTR\_ADD\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_ADD;

Memout=0;

end

// sub

`INSTR\_SUB\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_SUB;

Memout=0;

end

// and

`INSTR\_AND\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_AND;

Memout=0;

end

// or

`INSTR\_OR\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_OR;

Memout=0;

end

//slt

`INSTR\_SLT\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_SLT;

Memout=0;

end

// sll

`INSTR\_SLL\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_SLL ;

Memout=1;

end

// srl

`INSTR\_SRL\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_SRL ;

Memout=1;

end

// sra

`INSTR\_SRA\_FUNCT:

begin

RegDst = 2'b01;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_SRA ;

Memout=1;

end

//jr

`INSTR\_JR\_FUNCT:

begin

RegDst = 2'b00;

ALUSrc = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b11;

ExtOp = 0;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

endcase

end

//lb

`INSTR\_LB\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b01;

PC\_sel = 2'b00;

ExtOp = 1;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

// lw

`INSTR\_LW\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b01;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

//sb

`INSTR\_SB\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 0;

MemRead = 1;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

// sw

`INSTR\_SW\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 0;

MemRead = 1;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 0;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

// addi

`INSTR\_ADDI\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 1;

ALUCtrl = `ALUOp\_ADD;

Memout=0;

end

//ori

`INSTR\_ORI\_OP: //6'b001101:

begin

RegDst = 2'b00;

ALUSrc = 1;

MemRead = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 1;

ALUCtrl = `ALUOp\_OR;

Memout=0;

end

// lui

`INSTR\_LUI\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 1;

ALUCtrl = `ALUOp\_LUI ;

Memout=0;

end

//slti

`INSTR\_SLTI\_OP:

begin

RegDst = 2'b00;

ALUSrc = 1;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b00;

ExtOp = 1;

ALUCtrl = `ALUOp\_NOP;

end

//beq

`INSTR\_BEQ\_OP:

begin

RegDst = 2'b00;

ALUSrc = 0;

MemRead = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b01;

ExtOp = 0;

ALUCtrl = `ALUOp\_EQL;

Memout=0;

end

// bne

`INSTR\_BNE\_OP:

begin

RegDst = 2'b00;

ALUSrc = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b01;

ExtOp = 0;

ALUCtrl = `ALUOp\_BNE ;

Memout=0;

end

// j

`INSTR\_J\_OP:

begin

RegDst = 2'b00;

ALUSrc = 0;

RegWrite = 0;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b00;

PC\_sel = 2'b10

ExtOp = 0;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

// jal

`INSTR\_JAL\_OP:

begin

RegDst = 2'b01;

ALUSrc = 0;

RegWrite = 1;

MemRead = 0;

MemWrite = 0;

DatatoReg = 2'b01;

PC\_sel = 2'b11;

ExtOp = 0;

ALUCtrl = `ALUOp\_NOP;

Memout=0;

end

endcase

end

endmodule

**4.6 Extend**

module extend(

input [15:0] in,

input ExtOp,

output reg[31:0] out

);

always@(in or ExtOp)begin

if(ExtOp == 1) out = {{16{in[15]}},in[15:0]};

else out = {{16{1'b0}},in[15:0]};

end

endmodule

**4.7 ctrl\_encode\_def**

// NPC control signal

`define NPC\_PLUS4 2'b00

`define NPC\_BRANCH 2'b01

`define NPC\_JUMP 2'b10

// EXT control signal

`define EXT\_ZERO 2'b00

`define EXT\_SIGNED 2'b01

`define EXT\_HIGHPOS 2'b10

`define shiftnum 32'h7fffffff

// ALU control signal

`define ALUOp\_NOP 5'b00000

`define ALUOp\_ADDU 5'b00001

`define ALUOp\_ADD 5'b00010

`define ALUOp\_SUBU 5'b00011

`define ALUOp\_SUB 5'b00100

`define ALUOp\_AND 5'b00101

`define ALUOp\_OR 5'b00110

`define ALUOp\_NOR 5'b00111

`define ALUOp\_XOR 5'b01000

`define ALUOp\_SLT 5'b01001

`define ALUOp\_SLTU 5'b01010

`define ALUOp\_EQL 5'b01011

`define ALUOp\_BNE 5'b01100

`define ALUOp\_GT0 5'b01101

`define ALUOp\_GE0 5'b01110

`define ALUOp\_LT0 5'b01111

`define ALUOp\_LE0 5'b10000

`define ALUOp\_SLL 5'b10001

`define ALUOp\_SRL 5'b10010

`define ALUOp\_SRA 5'b10011

`define ALUOp\_LUI 5'b10100

// GPR control signal

`define GPRSel\_RD 2'b00

`define GPRSel\_RT 2'b01

`define GPRSel\_31 2'b10

`define WDSel\_FromALU 2'b00

`define WDSel\_FromMEM 2'b01

`define WDSel\_FromPC 2'b10

// Memory control signal

`define BE\_SB 2'b00

`define BE\_SH 2'b01

`define BE\_SW 2'b10

`define ME\_LB 3'b000

`define ME\_LBU 3'b001

`define ME\_LH 3'b010

`define ME\_LHU 3'b011

`define ME\_LW 3'b100

**4.8 dm**

module dm(

input [31:0] addr,

input [31:0] data\_in,

input MemWrite,

input MemRead,

input clk,

input reset,

output [31:0] data\_out

);

reg [31:0] DMem[1023:0];

always@(posedge clk)

begin

if(MemWrite)

DMem[addr] <= data\_in;

$display("addr=%8X",addr);//addr to DM

$display("din=%8X",data\_in);//data to DM

$display("Mem[00-07]=%8X, %8X, %8X, %8X, %8X, %8X, %8X, %8X",DMem[0],DMem[1],DMem[2],DMem[3],DMem[4],DMem[5],DMem[6],DMem[7]);

end

assign data\_out = DMem[addr]

endmodule

**4.9 im**

module im(

input [11:2] Addr,

output [31:0] Out

);

reg [31:0] IMem[1023:0];

assign Out = IMem[Addr];

endmodule

**4.10 mux**

`include "ctrl\_encode\_def.v"

module alu(

input [31:0] A,

input [31:0] B,

input [4:0] ALUOp,

output reg [31:0] C,

output reg zero

);

initial begin

C = 32'b0;

zero = 0;

end

always @(A or B or ALUOp)begin

case(ALUOp)

`ALUOp\_ADD: C = A + B; //add

`ALUOp\_SUB: C = A - B; //sub

`ALUOp\_OR: C = A | B;

`ALUOp\_NOR: C = ~(A | B);

`ALUOp\_XOR: C = A ^ B;

`ALUOp\_EQL: zero=(A==B)?1'b1:1'b0;

`ALUOp\_BNE: zero=(A==B)?1'b0:1'b1;

`ALUOp\_AND: C = A & B;

`ALUOp\_SLL: C = A<<(B[10:6]);

`ALUOp\_SLT: C = (A < B) ? 32'd1 : 32'd0;

`ALUOp\_SRL: C = A>>(B[10:6]);

`ALUOp\_SRA: C = ({{31{B[31]}},1'b0} << (~A[4:0]))|(B >> A[4:0]);

`ALUOp\_LUI: C = (B<<16);

`ALUOp\_SLTU: C = ((A<B)?1:0);

`ALUOp\_NOP: C = A;

default: C = A;

endcase

end

endmodule

**4.11 NPC**

module npc(

input [31:0] oldPC,

input [15:0] beq\_imm,

input beq\_zero,

input [31:0] jr\_addr,

input [31:0] j\_addr,

input [1:0] PC\_sel,

output reg [31:0] newPC

);

always @(oldPC or beq\_imm or beq\_zero or PC\_sel) begin

case(PC\_sel)

2'b00: newPC = oldPC + 4;

2'b01: if(beq\_zero == 1) newPC = oldPC + 4 + {{14{beq\_imm[15]}},beq\_imm[15:0],2'b00};//beq

else newPC = oldPC + 4;

2'b10: newPC = (j\_addr<<2);

2'b11: newPC = jr\_addr;

endcase

end

endmodule

**4.12 instruction\_def**

// OP

`define INSTR\_RTYPE\_OP 6'b000000

`define INSTR\_LB\_OP 6'b100000

`define INSTR\_LH\_OP 6'b100001

`define INSTR\_LBU\_OP 6'b100100

`define INSTR\_LHU\_OP 6'b100101

`define INSTR\_LW\_OP 6'b100011

`define INSTR\_SB\_OP 6'b101000

`define INSTR\_SH\_OP 6'b101001

`define INSTR\_SW\_OP 6'b101011

`define INSTR\_ADDI\_OP 6'b001000

`define INSTR\_ADDIU\_OP 6'b001001

`define INSTR\_ANDI\_OP 6'b001100

`define INSTR\_ORI\_OP 6'b001101

`define INSTR\_XORI\_OP 6'b001110

`define INSTR\_LUI\_OP 6'b001111

`define INSTR\_SLTI\_OP 6'b001010

`define INSTR\_SLTIU\_OP 6'b001011

`define INSTR\_BEQ\_OP 6'b000100

`define INSTR\_BNE\_OP 6'b000101

`define INSTR\_BGEZ\_OP 6'b000001

`define INSTR\_BGTZ\_OP 6'b000111

`define INSTR\_BLEZ\_OP 6'b000110

`define INSTR\_BLTZ\_OP 6'b000001

`define INSTR\_J\_OP 6'b000010

`define INSTR\_JAL\_OP 6'b000011

// Funct

`define INSTR\_ADD\_FUNCT 6'b100000

`define INSTR\_ADDU\_FUNCT 6'b100001

`define INSTR\_SUB\_FUNCT 6'b100010

`define INSTR\_SUBU\_FUNCT 6'b100011

`define INSTR\_AND\_FUNCT 6'b100100

`define INSTR\_NOR\_FUNCT 6'b100111

`define INSTR\_OR\_FUNCT 6'b100101

`define INSTR\_XOR\_FUNCT 6'b100110

`define INSTR\_SLT\_FUNCT 6'b101010

`define INSTR\_SLTU\_FUNCT 6'b101011

`define INSTR\_SLL\_FUNCT 6'b000000

`define INSTR\_SRL\_FUNCT 6'b000010

`define INSTR\_SRA\_FUNCT 6'b000011

`define INSTR\_SLLV\_FUNCT 6'b000100

`define INSTR\_SRLV\_FUNCT 6'b000110

`define INSTR\_SRAV\_FUNCT 6'b000111

`define INSTR\_JR\_FUNCT 6'b001000

`define INSTR\_JALR\_FUNCT 6'b001001

`define INSTR\_BGEZ\_RT 5'b00001

`define INSTR\_BLTZ\_RT 5'b00000

**4.13 IR**

module IR (clk, rst, IRWr, im\_dout, instr);

input clk;

input rst;

input IRWr;

input [31:0] im\_dout;

output [31:0] instr;

reg [31:0] instr;

always @(posedge clk or posedge rst) begin

if ( rst )

instr <= 0;

else if (IRWr)

instr <= im\_dout;

end // end always

endmodule

**4.14 mips**

module mips( );

reg clk, reset;

initial begin

$readmemh( "code.txt", IM.IMem ) ;

$monitor("PC = 0x%8X, IR = 0x%8X", PC.oldpc, IM.Out );

clk = 1 ;

reset = 0 ;

#5 reset = 1 ;

#20 reset = 0 ;

end

always #50 clk = ~clk;

wire [31:0] old\_PC;

wire [31:0] new\_PC;

wire beq\_zero;

wire [1:0] PC\_sel;

wire [31:0] Instrl;

wire ExtOp;

wire [31:0] ext\_out;

wire [1:0] Data\_to\_Reg\_sel;

wire [31:0] Data\_to\_Reg;

wire [1:0] RegDst;

wire [4:0] Reg\_rd;

wire RegWrite;

wire [31:0] grf\_out\_A;

wire [31:0] grf\_out\_B;

wire ALUSrc;

wire [31:0] ALUSrc\_out;

wire [4:0] ALUCtr;

wire [31:0] ALU\_out;

wire [31:0] dm\_data\_out;

wire MemWrite;

wire MemRead;

im IM(old\_PC[11:2],Instrl);

npc NPC(old\_PC,Instrl[15:0],ext\_out,beq\_zero,PC\_sel,grf\_out\_A,new\_PC);

pc PC(new\_PC,clk,reset,old\_PC);

RegDst\_mux REGDST(RegDst,Instrl[20:16],Instrl[15:11],Reg\_rd);

DatatoReg\_muxDATATOREG(ALU\_out,dm\_data\_out,Data\_to\_Reg\_sel,Data\_to\_Reg);

gprGRF(clk,reset,Instrl[25:21],Instrl[20:16],Reg\_rd,Data\_to\_Reg,RegWrite,grf\_out\_A,grf\_out\_B);

extend EXTEND(Instrl[15:0],ExtOp,ext\_out);

ALUSrc\_mux ALUSRC(grf\_out\_B,ext\_out,ALUSrc,ALUSrc\_out);

alu ALU(grf\_out\_A,ALUSrc\_out,ALUCtr,ALU\_out,beq\_zero);

ctrlCTRL(Instrl[31:26],Instrl[5:0],RegDst,ALUSrc,MemRead,RegWrite,MemWrite,Data\_to\_Reg\_sel,PC\_sel,ExtOp,ALUCtr,memout);

dm DM(ALU\_out,grf\_out\_B,MemWrite,MemRead,clk,reset,dm\_data\_out);

endmodule

**设计结果分析**

# Test File for 7 Instruction, include:

# ADDU/SUBU/LW/SW/ORI/BEQ/JAL

Settings -> Memory Configuration -> Compact, Data at address 0

.text

L0:ori $29, $0, 12

ori $2, $0, 0x1234

ori $3, $0, 0x3456

addu $4, $2, $3

subu $6, $3, $4

sw $2, 0($0)

sw $3, 4($0)

sw $4, 4($29)

lw $5, 0($0)

beq $2, $5,L2

L1:lw $3, 4($29)

L2:lw $5, 4($0)

beq $3, $5, L1

subu $6, $6, $2

sw $6, -4($29)

beq $3, $3, L0

生成的机器代码：

341d000c

34021234

34033456

00432021

00643023

ac020000

ac030004

afa40004

8c050000

10450001

8fa30004

8c050004

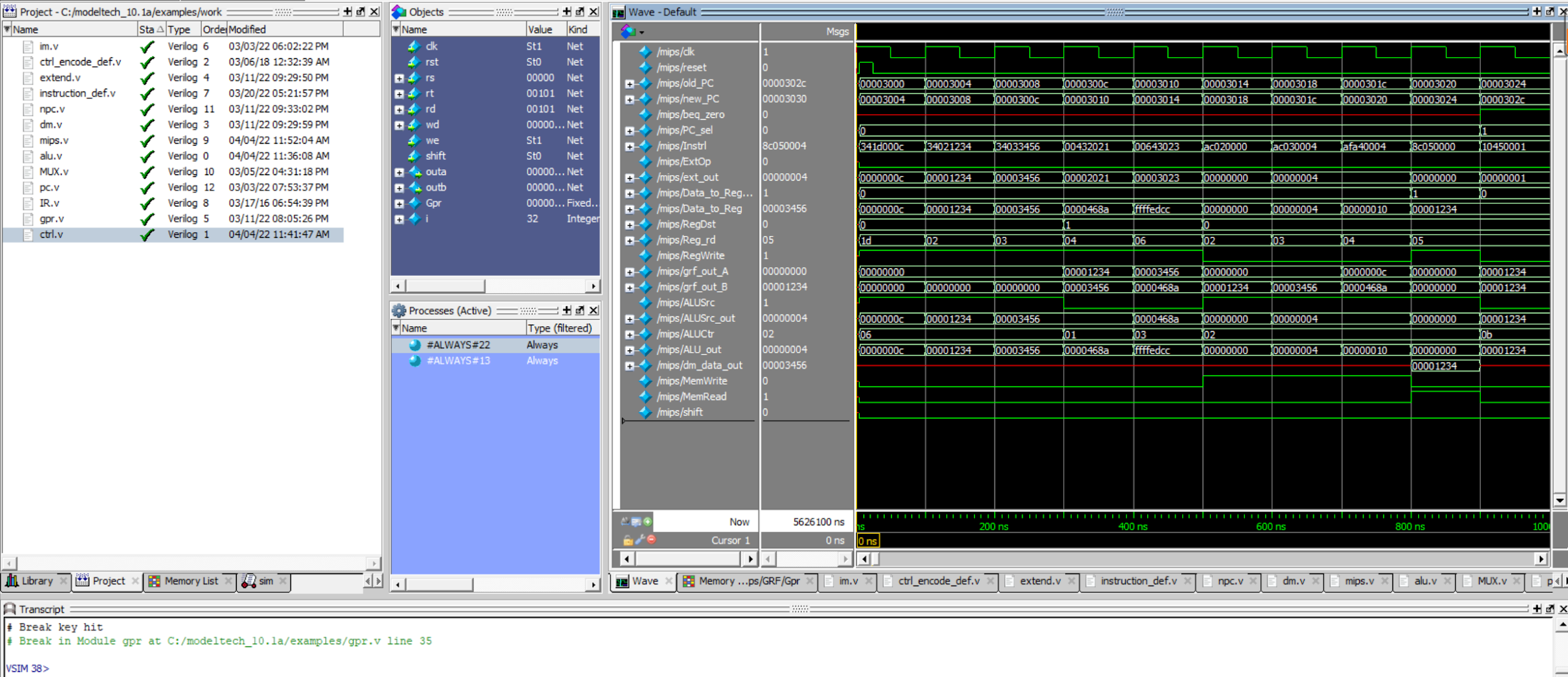
1065fffd

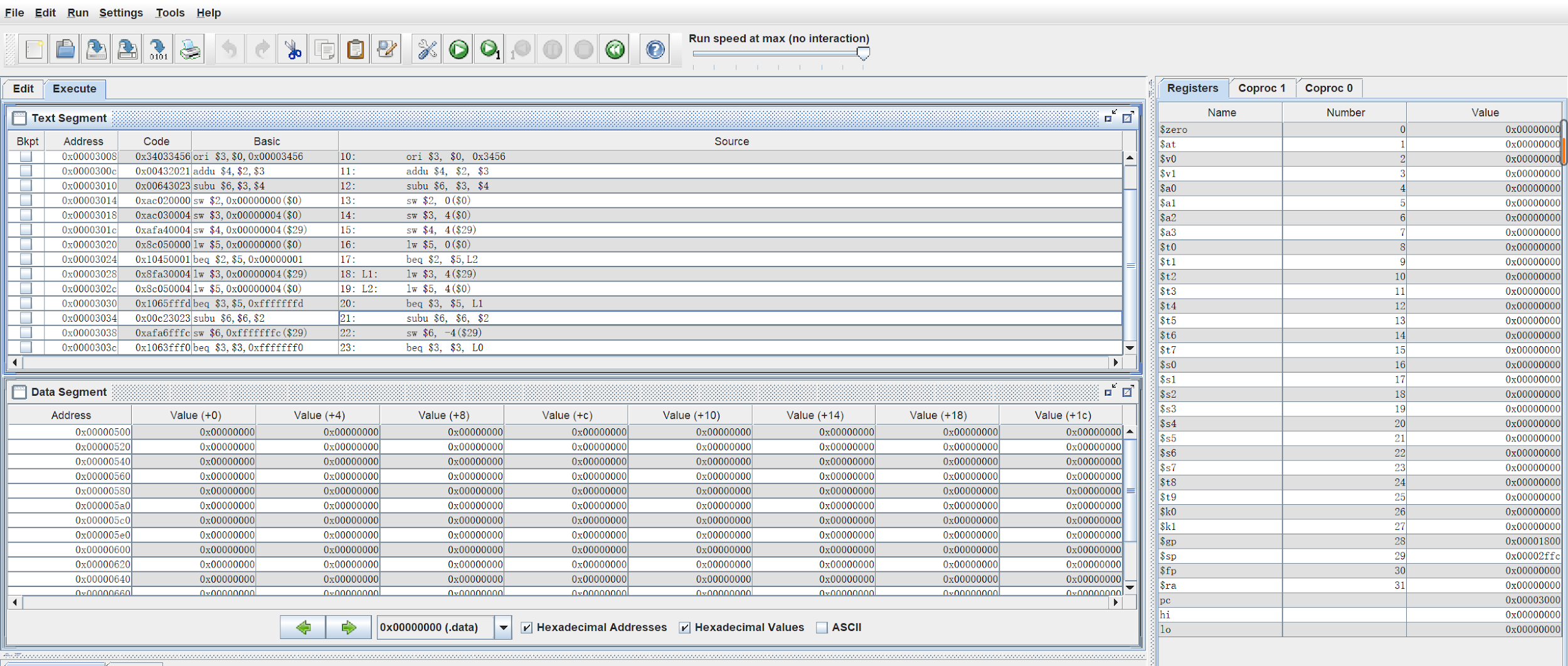
00c23023

afa6fffc

1063fff0

然后在mips.v这个文件里，用Mars生成的机器代码txt文件导入到IM里的IMem（memory）里，接着点击mips.v进行仿真

Modelsim仿真结果：****

Mars运行结果:****

**总 结**

对 Verilog HDL 语言的学习非常重要。这是我第一次接触这种设计语言。由于时间紧张，采用的是偏应用的学习方式， 没有一开始就全面详细地学习各个语法，而是，通过看老师发的样式代码，经过自己的修改，看看结果与假设是否符合来学习这门语言。通过本次实验，加深了我对cpu数据通路的了解，让我更加深刻地领悟了计算机组成与设计理论课程中所学习的CPU相关知识，同时调试过程相对庞大的工作量也锻炼了我查找bug的能力，仿真过程中遇到了特别多的问题，迫使自己不停去查阅各种资料，一次又一次地编译，本次实验的完成，让我获益良多

教师评语评分

评语：

评分：

评阅人：

年 月 日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）