



**系统硬件综合设计报告**

**专 业 班 级 物联网工程2018级1班**

**学 生 学 号 2018217870**

**学 生 姓 名 王亚杰**

**指 导 教 师 陈田、刘军、丁贤庆、安鑫、阙夏、李建华**

**2021年7月**

**目录**

[1. 实验目的 3](#_Toc77459303)

[2. 实验内容 3](#_Toc77459304)

[2.1 MIPS32架构的指令说明及本文设计 3](#_Toc77459305)

[2.1.1 MIPS32指令 3](#_Toc77459306)

[2.1.2 本文设计 4](#_Toc77459307)

[2.2 MIPS32架构寻址 5](#_Toc77459308)

[3. 具体实现 5](#_Toc77459309)

[3.1整体架构设计 5](#_Toc77459310)

[3.2 分模块设计 6](#_Toc77459311)

[3.2.1 Instructions 6](#_Toc77459312)

[3.2.2 Regfile 7](#_Toc77459313)

[3.2.3 ALU 7](#_Toc77459314)

[3.2.4 Data Memory 8](#_Toc77459315)

[3.2.5 control 8](#_Toc77459316)

[3.3 数据流介绍展示 9](#_Toc77459317)

[3.3.1 addu指令 9](#_Toc77459318)

[3.3.2 sw指令 10](#_Toc77459319)

[3.3.2 beq指令 11](#_Toc77459320)

[4. 调试及结果展示 12](#_Toc77459321)

[附录 18](#_Toc77459322)

**说明**

此次课设我们组共两人，依次为王亚杰、孙国龙，**课设从设计、编写代码实现到后期调试、报告撰写全部由我们两人独立完成**,共实现8条指令：addu subu ori lw sw beq j lui。其中王亚杰主要负责整体结构设计及lw sw beq j lui指令的实现，孙国龙负责addu subu ori指令的实现及后期调试。这份报告由我们两个人分别撰写对应任务部分结合而成。

**单周期MIPS架构CPU的设计与实现**

# 1. 实验目的

通过设计并实现支持8条指令的单周期MIPS32架构的CPU，进一步理解和掌握CPU设计的基本原理和步骤。

# 2. 实验内容

## 2.1 MIPS32架构的指令说明及本文设计

### 2.1.1 MIPS32指令

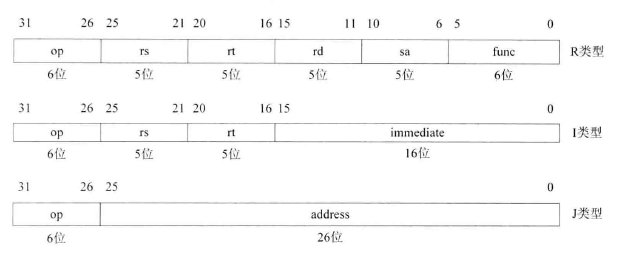


图2-1 MIPS32架构中的三种指令格式

MIPS32架构中的所有指令都是32位，也就是32个0、1编码连在一起表示一条指令，有三种指令格式。如图所示：其中op是指令码、func是功能码。

(1)**R类型**：具体操作由op、func结合指定，rs和rt是源寄存器的编号，rd是目的寄存器的编号，比如假设目的寄存器是S3,那么对应的rd就是00011(此处是二进制数。MIPS32架构中有32个通用寄存器，使用5位编码就可以全部表示，所以rs、rt、rd的宽度都是5位。sa只有在移位指令中使用，用来指定移位位数。

(2)**I类型：**具体操作由op指定，指令的低16位是立即数，运算时要将其扩展至32位，然后作为其中一个源操作数参与运算。

(3)**J类型：**具体操作由op指定，一般是跳转指令，低26位是字地址，用于产生跳转的目标地址。

### 2.1.2 本文设计

（1）**R类型**：

① addu：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 100001 |

进行的操作：R[rd]←R[rs]+R[rt]; PC←PC+4

② subu：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 000000 | rs | rt | rd | 00000 | 100011 |

进行的操作：R[rd]←R[rs]-R[rt]; PC←PC+4

（2）**I类型**

① ori：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 001101 | | rs | rt | | imm16 | |

进行的操作：R[rt]←R[rs]+zero\_ext(imm16); PC←PC+4

② lw：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 100011 | | rs | rt | | imm16 | |

进行的操作：R[rt]←Mem(R[rs]+sign\_ext(imm16)); PC←PC+4

③ sw：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 101011 | | rs | rt | | imm16 | |

进行的操作：Mem(R[rs]+sign\_ext(imm16))←R[rt]; PC←PC+4

④ beq：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 000100 | | rs | rt | | imm16 | |

进行的操作：if(R[rs] == R[rt]) then PC←PC+4+[sign\_ext(imm16)]

else PC←PC+4

⑤ lui：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | | | 20 16 | | 15 0 |
| 001111 | | rs | rt | | imm16 | |

进行的操作：R[rt][31:16]←imm16 , R[rt][15:0]←0; PC←PC+4

（3）**J类型**

① j：

|  |  |  |
| --- | --- | --- |
| 31 26 | 25 0 | |
| 001101 | | imm25 |

进行的操作：PC←{PC[31:28], imm25, 2’b0}

## 2.2 MIPS32架构寻址

MIPS32架构的寻址模式有四种：寄存器寻址、立即数寻址、寄存器相对寻址和PC相对寻址。在我们的指令中，R类型的addu、subu指令采用寄存器寻址；I类型的ori、lw、sw指令采用寄存器相对寻址，beq指令除此之外还采用了PC相对寻址；J指令采用了PC相对寻址。

# 3. 具体实现

## 3.1整体架构设计

（1）设计架构如图所示：

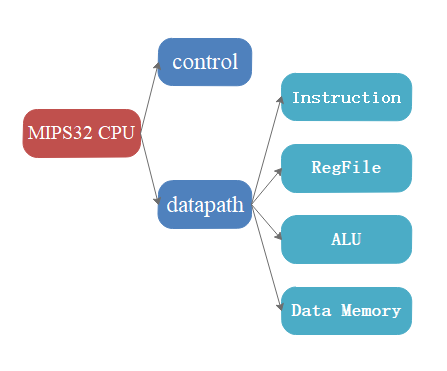


图3-1 设计架构

整体分为控制模块与数据流模块两部分；控制模块为数据流模块提供控制信号；数据流模块分为指令模块、寄存器模块、运算器模块、存储器模块四大部分，分别实现了不同功能。

（2）整体架构：

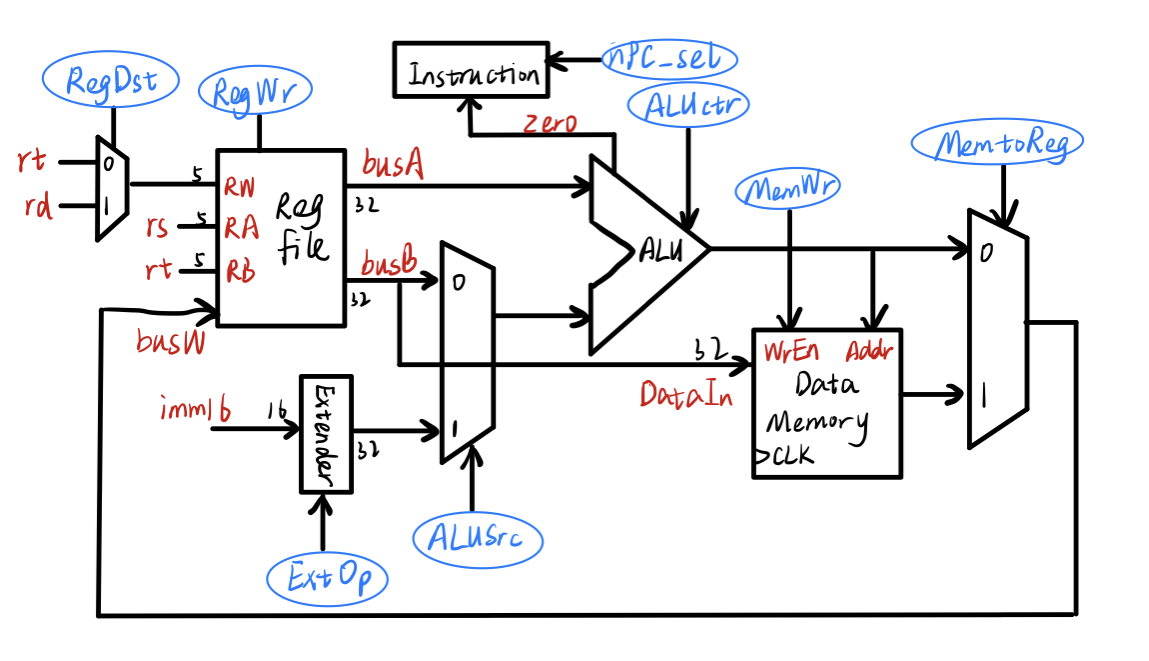


图3-2 整体架构

整体架构图如图所示，为了查看清晰，此处并未将控制模块列入其中，而仅以蓝色椭圆框及内含的文字表示一个控制信号，此处的控制信号全部由控制模块产生。

## 3.2 分模块设计

### 3.2.1 Instructions

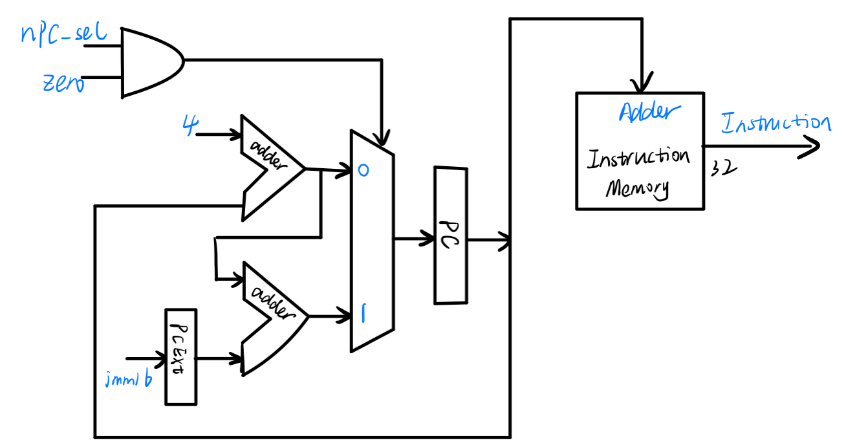


图3-3 Instructions模块

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| nPC\_sel | 2 | 输入 | 控制PC跳转 |
| zero | 1 | 输入 | beq指令中控制PC跳转 |
| Instructions | 32 | 输出 | 取出的指令 |

表3-1 Instructions模块的接口描述

Instructions模块主要实现取指令功能，模块中有两个adder，分别产生PC+4和PC+sign\_ext(imm16)，这两个adder的输出又通过二选一数据选择器选择赋值给PC，其中nPC\_sel控制信号来自control模块，zero来自ALU模块。

### 3.2.2 Regfile

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| RegWr | 1 | 输入 | 写使能信号 |
| RA | 5 | 输入 | 读入的第一个寄存器编号 |
| RB | 5 | 输入 | 读入的第二个寄存器编号 |
| RW | 5 | 输入 | 写入的寄存器编号 |
| busW | 32 | 输入 | 写入的数据 |
| busA | 32 | 输出 | 从第一个寄存器中读出的数据 |
| busB | 32 | 输出 | 从第二个寄存器中读出的数据 |

表3-2 Regfile模块的接口描述

Regfile模块是寄存器相关模块，实现寄存器的读写功能。因为共有32个寄存器，需要5位来对寄存器编号进行编码，故有关编号的输入都是5位的。其中，RA、RB寄存器的信息分别由busA、busB端口输出，因有些指令会利用到立即数，故busB端口与立即数通过控制信号进行二选一的选择。RW、RegWr、busW是与写有关的端口，RegWr控制是否进行写操作；RegWr端口的输入是一个二选一选择器的输出，该二选一选择器是通过控制信号判断rt为需要输入的寄存器编号还是rs为需要输入的寄存器编号。

### 3.2.3 ALU

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| ALUctr | 2 | 输入 | 控制ALU中进行的操作 |
| busA | 32 | 输入 | 操作数1 |
| busB | 32 | 输入 | 操作数2 |
| ans | 32 | 输出 | 运算结果 |
| zero | 32 | 输出 | 运算结果 |

表3-3 ALU模块的接口描述

ALU模块是运算相关的模块，由ALUctr控制信号控制两个操作数进行的数据操作，ans与zero均为答案输出，只是继续传递的方向不同，zero是向Instruction模块传递，主要是为在beq指令时根据运算结果控制PC的变化情况。

### 3.2.4 Data Memory

|  |  |  |  |
| --- | --- | --- | --- |
| **接口名** | **宽度(bit)** | **输入\输出** | **作用** |
| WrEn | 1 | 输入 | 写使能信号 |
| Addr | 32 | 输入 | 需要访问的地址 |
| Data In | 32 | 输入 | 需要写入的数据 |
| Data Out | 32 | 输出 | 读出的数据 |

表3-4 Data Memory模块的接口描述

Data Memory是与存储器相关的模块。在进行读操作时WrEn信号置0，Addr端口输入需要访问的地址，由Data Out端口输出访问的数据；在进行写操作时WrEn信号置1，数据由Data In端口读入并写入到Addr端口输入的地址中。

### 3.2.5 control

|  |  |
| --- | --- |
| **指令** | **控制信号** |
| addu | ALUsrc=RegB, ALUctr=“ADD”, RegDst=rd,  RegWr, nPC\_sel=“+4” |
| subu | ALUsrc=RegB, ALUctr=“SUB”, RegDst=rd,  RegWr, nPC\_sel=“+4” |
| ori | ALUsrc=Imm, ALUctr=“OR”, RegDst=rt,  RegWr, ExtOp=“Zero”, nPC\_sel=“+4” |
| lw | ALUsrc=Imm, ALUctr=“ADD”, RegDst=rt,  RegWr, ExtOp=“Sign”, MemtoReg, nPC\_sel=“+4” |
| sw | ALUsrc=Imm, ALUctr=“ADD”, MemWr,  ExtOp=“Sign”, nPC\_sel=“+4” |
| beq | ALUsrc=RegB, ALUctr=“SUB”, nPC\_sel=“Br” |

表3-5 control模块指令与控制信号对应关系

control模块是整个CPU运行的核心，表格中列出了指令与相关控制信号的对应关系，具体对应关系是如何推导出来的将在下一部分—数据流中详细介绍。

## 3.3 数据流介绍展示

为了避免冗余，此处只结合代码展示具有代表性的addu、sw、beq三个指令的数据流。其中addu重点展示regfile实现读写寄存器、ALU实现数据操作等相关功能实现及控制信号产生依据；sw展示Data Memory进行写入等相关功能实现及控制信号产生依据；beq展示Instructions模块对于指令读取等相关功能的实现及控制信号产生依据。

### 3.3.1 addu指令

1. 利用Instructions模块从code.txt中读取出指令代码并赋值给instruction。
2. 根据instruction接下来调用控制模块生成中间需要的控制信号，控制信号的产生原理将在后续数据流的推导过程中介绍。

*if(instruction[31:26]==6'b000000)*

*begin*

*//ADDU*

*if(instruction[5:0]==6'b100001)*

*begin*

*nPC\_sel=2'b00;*

*RegDst=1'b1;*

*RegWr=1'b1;*

*ExtOp=2'b00;*

*ALUSrc=1'b0;*

*ALUctr=2'b00;*

*MemWr=1'b0;*

*MemtoReg=1'b0;*

*j\_sel=1'b0;*

*end*

1. 接下来调用Regfile模块。

① 控制信号产生：

a. 由于需要进行写入寄存器操作，故RegWr控制信号为1。

b. 根据第二部分中给出的我们设计的指令格式可知，写入的寄存器为rd，故RegDst控制信号为1。

② 在调用模块时即为Regfile的RA、RB端赋初始值，关键代码如下：

*gpr GPR(.RegWr(RegWr),.ra(instruction[25:21]),.rb(instruction[20:16]),.rw(rw),.busW(busW),.clk(clk),.rst(rst),.busA(busA),.busB(busB),.Data\_in(Data\_in));*

1. 调用ALU模块进行数据处理。

①控制信号产生：

a. 由于需要使用的是从Regfile中读出的数据，故二选一数据选择器的控制信号ALUSrc应为0.

b. 应进行加法操作，故ALUctr应控制ALU进行加法操作，即为00。

②根据ALUctr进行加法操作，关键代码如下：

*ADD:begin*

*Alu\_out=busA+busB;*

1. 后续处理

①控制信号产生：

a. 由于存储器不进行写入操作，故控制信号MemWr=0

b. 由于不涉及存储器操作，故MemtoReg控制信号为0选择ALU的输出

②后续数据流：运算结果送入busW，因已将写使能信号RegWr置1，故可直接进行写入操作。

### 3.3.2 sw指令

1. 利用Instructions模块从code.txt中读取出指令代码并赋值给instruction。
2. 根据instruction接下来调用控制模块生成中间需要的控制信号，控制信号的产生原理将在后续数据流的推导过程中介绍。

*else if(instruction[31:26]==6'b101011)*

*begin*

*nPC\_sel=2'b00;*

*RegDst=1'b0;*

*RegWr=1'b0;*

*ExtOp=2'b01;*

*ALUSrc=1'b1;*

*ALUctr=2'b00;*

*MemWr=1'b1;*

*MemtoReg=1'b0;*

*j\_sel=1'b0;*

*end*

1. 接下来调用Regfile模块。

① 控制信号产生：

a. 由于不需要进行写入寄存器操作，故RegWr控制信号为0。

② 输出信号：根据第二部分中的指令格式可知，寄存器rt中的数据为我们应写入的数据，经busB端口输出；寄存器rs中的数据是我们应写入的地址中的一部分，经busA输出。

1. ALU模块输入端的选择与运算。

①控制信号产生：

a. 访存时需要的地址数据由两部分组成，busA的输出还应与立即数进行加法运算最后得出最终地址。故需经ALU进行加法运算的两个输入端口数据应为busA与立即数，故ALUsrc控制信号应为1。

b. 应进行加法操作，故ALUctr应控制ALU进行加法操作，即为00。

②根据ALUctr进行加法操作，关键代码如下：

*ADD:begin*

*Alu\_out=busA+busB;*

1. Data Memory模块的处理

①控制信号产生：

a. 由于存储器进行写入操作，故控制信号MemWr=1。

②数据流：ALU运算出的最终地址经Addr端口输入存储器作为写入地址，寄存器读出的数据经Regfile的busB端口输出并经Data Memory的Data In端口输入并写入到Addr输入的地址中。

### 3.3.2 beq指令

1. 利用Instructions模块从code.txt中读取出指令代码并赋值给instruction。
2. 根据instruction接下来调用控制模块生成中间需要的控制信号，控制信号的产生原理将在后续数据流的推导过程中介绍。

*else if(instruction[31:26]==6'b000100)*

*begin*

*nPC\_sel=2'b10;*

*RegDst=1'b0;*

*RegWr=1'b0;*

*ExtOp=2'b01;*

*ALUSrc=1'b0;*

*ALUctr=2'b01;*

*MemWr=1'b0;*

*MemtoReg=1'b0;*

*j\_sel=1'b0;*

*end*

1. 接下来调用Regfile模块。

① 控制信号产生：

a. 由于不需要进行写入寄存器操作，故RegWr控制信号为0。

② 输出信号：根据第二部分中的指令格式可知，寄存器rt和寄存器rs为我们需要比较的数值，分别由busA、busB端口输出。

1. ALU模块输入端的选择与运算。

①控制信号产生：

a. 由于我们比较的数据来自busA与busB端口，故ALUsrc控制信号应为0。

b. 应进行减法操作进行比较两个数据是否相等，故ALUctr应控制ALU进行加法操作，即为01。若结果为0则表示数据相等需要跳转，否则为不需要跳转，此时运算结果通过zero端口传递到Instructions模块控制下一步取指令操作。

②根据ALUctr进行减法操作，关键代码如下：

*SUB:begin*

*Alu\_out=busA-busB;*

1. Instructions模块的处理

①从ALU模块传递过来的zero控制信号为0，故二选一数据选择器的控制信号为1，即为将现在的PC值加上扩展的16位立即数生成最新的PC值，再根据此PC值进行取值操作，即实现指令跳转功能。

# 4. 调试及结果展示

1. 使用MARS编写汇编代码

其中使用的指令有：ori, addu, subu, sw, bw, beq, lui, j，将CPU拥有的所有指令进行使用，源代码如下。

***ori*** *$3,$0,0x93*

***ori*** *$6,$0,0xae*

***addu*** *$8,$3,$6*

***subu*** *$9,$3,$6*

***addu*** *$0,$9,$10*

***sw*** *$9,16($0)*

***lw*** *$10,16($0)*

*l3:****beq*** *$9,$10,l1*

***lui*** *$11,0xcdcd*

***j*** *end*

*l1:****ori*** *$11,$0,0xefef*

***lui*** *$9,0x4567*

***j*** *l3*

*end:*

1. 导出程序的八进制机器码

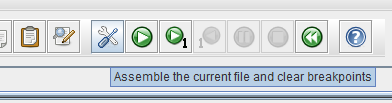


图4-1 编译汇编程序

①首先点击编译按钮，使MARS编译汇编程序生成机器码。

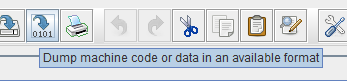


图4-2 导出机器码

②点击Dump按钮将机器码导出。

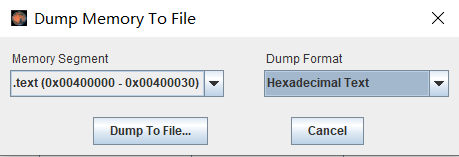


图4-3 导出八进制机器码

③注意设置Dump Format为八进制文本格式。

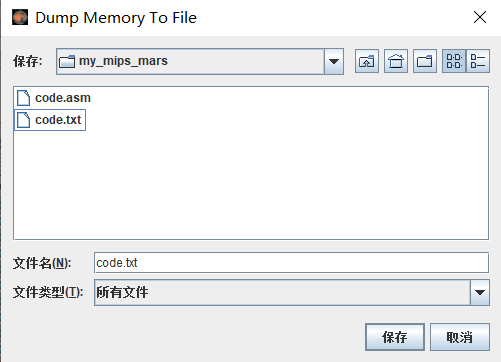


图4-4 保存机器码文件

④选择保存机器码的文件路径。

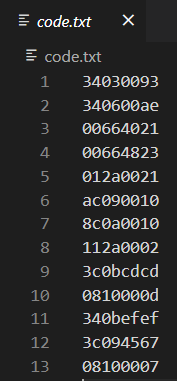


图4-5 生成的机器码文件

⑤得到生成的机器码后，两个数字一行将其分割开，用于使ModelSim仿真时读取。

1. 在ModelSim中将机器码读入到程序存储器中

在test.v中编写代码将机器码读入到程序存储器中，源代码：

*module test;*

*reg clk,rst;*

*mips launch(.clk(clk),.rst(rst));*

*initial begin*

*rst=0;*

*clk=1;*

*#1 rst=1;*

*#2 rst=0;*

***$readmemh("code.txt",launch.MAIN.IFU.im);***

*end*

*always*

*#30 clk=~clk;*

*endmodule*

其中， ***$readmemh("code.txt",launch.MAIN.IFU.im);*** 语句是从文件中读取机器码，并将其装载到程序存储器im中。

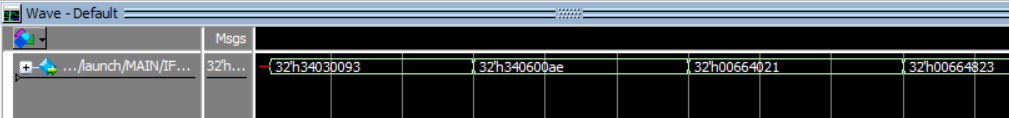


图4-6 ModelSim中指令的值

利用ModelSim仿真，观察指令的值是否与code.txt文件中的值一致。

**code.txt中的值:**

*34030093*

*340600ae*

观察后易知，指令的值与code.txt文件中的值一致。

4) 仿真运行并比照寄存器的值是否一致

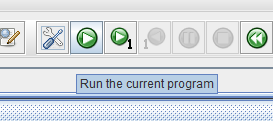


图4-7 MARS中进行编译

①首先在MARS中对汇编程序进行编译。

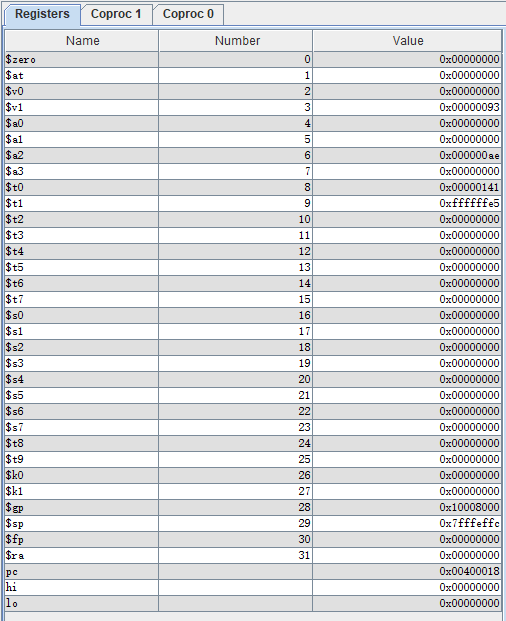


图4-8 MARS中寄存器组的值

②然后运行程序，得到运行结束后寄存器组的值。

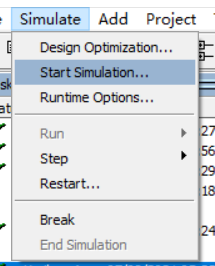


图4-9 开始仿真

③在ModelSim中点击Start Simulation开始仿真。

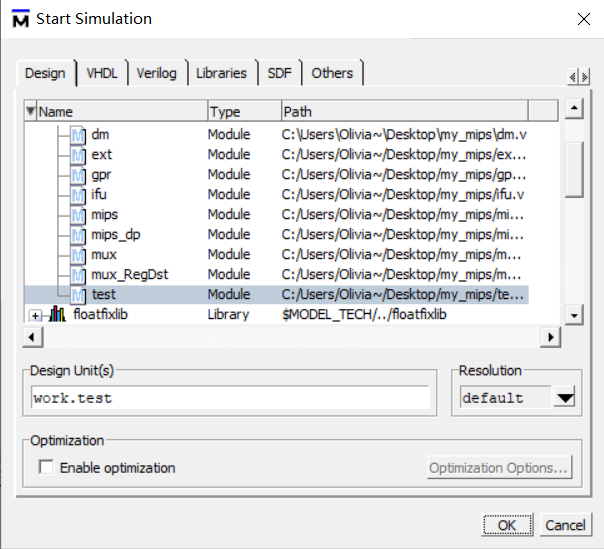


图4-10 选择仿真模块

④选择待仿真的模块，注意不要勾选Enable optimization。

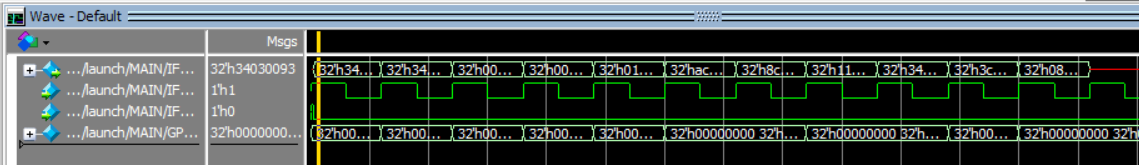


图4-11 将全部指令运行后的波形

⑤将指令全部运行，通过观察波形可知现在已经将所有指令运行完毕。

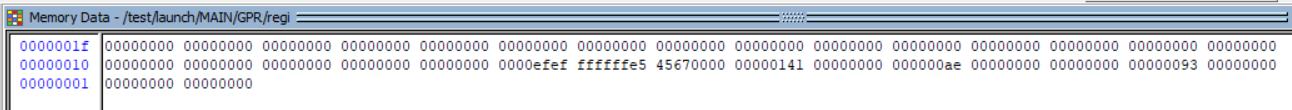


图4-12 MemoryList中寄存器组的值

⑥从MemoryList中观察寄存器组的值，并于MARS中运行出的寄存器组的值进行比照，一直两个软件运行得到的结果一致，即：基于mips的cpu仿真结果正确。

# 附录

(1) mip.v

module **mips**(clk,rst);

  input clk,rst;

  wire [1:0]ExtOp,ALUctr,nPC\_sel;

  wire ALUSrc,MemWr,MemtoReg,RegDst,RegWr,j\_sel;

  wire [31:0]instruction;

  ctrl CU(.instruction(instruction),.RegDst(RegDst),.RegWr(RegWr),.ExtOp(ExtOp),.nPC\_sel(nPC\_sel),.ALUctr(ALUctr),.MemtoReg(MemtoReg),.MemWr(MemWr),.ALUSrc(ALUSrc),.j\_sel(j\_sel));

  mips\_dp MAIN(.clk(clk),.rst(rst),.RegDst(RegDst),.RegWr(RegWr),.ExtOp(ExtOp),.nPC\_sel(nPC\_sel),.ALUctr(ALUctr),.MemtoReg(MemtoReg),.MemWr(MemWr),.ALUSrc(ALUSrc),.j\_sel(j\_sel),.Instruction(instruction));

endmodule

(2) mips\_dp.v

module **mips\_dp**(clk,rst,RegDst,RegWr,ExtOp,nPC\_sel,ALUctr,MemtoReg,MemWr,ALUSrc,j\_sel,Instruction);

  input clk,rst;

  input [1:0]ExtOp,ALUctr,nPC\_sel;

  input ALUSrc,MemWr,MemtoReg,RegDst,RegWr,j\_sel;

  wire [31:0]instruction;

  wire [31:0]busA,busB,busW,Mux\_ALUSrc\_out,imm32,Alu\_out,Addr,Data\_in,Data\_out,jValue;

  wire [31:0]zero;

  wire [4:0]rw;

  output [31:0]Instruction;

  assign Instruction[31:0]=instruction[31:0];

*//connect all compoenet*

  ifu IFU(.nPC\_sel(nPC\_sel),.zero(zero),.clk(clk),.rst(rst),.instruction(instruction),.j\_sel(j\_sel),.jValue(jValue));

  ext EXT(.imm16(instruction[15:0]),.imm32(imm32),.ExtOp(ExtOp));

  alu ALU(.busA(busA),.busB(Mux\_ALUSrc\_out),.ALUctr(ALUctr),.zero(zero),.Alu\_out(Alu\_out),.Addr(Addr));

  mux\_RegDst MUX\_RegDst(.a0(instruction[20:16]),.a1(instruction[15:11]),.rw(rw),.RegDst(RegDst));

  mux MUX\_ALUSrc(.a0(busB),.a1(imm32),.op(ALUSrc),.out(Mux\_ALUSrc\_out));

  mux MUX\_MemtoReg(.a0(Alu\_out),.a1(Data\_out),.op(MemtoReg),.out(busW));

  gpr GPR(.RegWr(RegWr),.ra(instruction[25:21]),.rb(instruction[20:16]),.rw(rw),.busW(busW),.clk(clk),.rst(rst),.busA(busA),.busB(busB),.Data\_in(Data\_in));

  dm DM(.Data\_in(Data\_in),.MemWr(MemWr),.Addr(Addr),.clk(clk),.rst(rst),.Data\_out(Data\_out));

endmodule

(3) ctrl.v

module **ctrl**(instruction,RegDst,RegWr,ExtOp,nPC\_sel,ALUctr,MemtoReg,MemWr,ALUSrc,j\_sel);

  input [31:0]instruction;

  output reg [1:0]ExtOp,ALUctr,nPC\_sel;

  output reg RegDst,RegWr,MemtoReg,MemWr,ALUSrc,j\_sel;

  initial begin

    nPC\_sel=0;

    RegDst=0;

    RegWr=0;

    ExtOp=0;

    nPC\_sel=0;

    ALUctr=0;

    MemtoReg=0;

    MemWr=0;

    ALUSrc=0;

    j\_sel=0;

end

always@(\*)begin

*//R-type*

  if(instruction[31:26]==6'b000000)

    begin

*//ADDU*

      if(instruction[5:0]==6'b100001)

        begin

          nPC\_sel=2'b00;

          RegDst=1'b1;

          RegWr=1'b1;

          ExtOp=2'b00;

          ALUSrc=1'b0;

          ALUctr=2'b00;

          MemWr=1'b0;

          MemtoReg=1'b0;

          j\_sel=1'b0;

      end

*//SUBU*

      else if(instruction[5:0]==6'b100011)

        begin

          nPC\_sel=2'b00;

          RegDst=1'b1;

          RegWr=1'b1;

          ExtOp=2'b00;

          ALUSrc=1'b0;

          ALUctr=2'b01;

          MemWr=1'b0;

          MemtoReg=1'b0;

          j\_sel=1'b0;

        end

      end

*//ORI*

    else if(instruction[31:26]==6'b001101)

      begin

        nPC\_sel=2'b00;

          RegDst=1'b0;

          RegWr=1'b1;

          ExtOp=2'b00;

          ALUSrc=1'b1;

          ALUctr=2'b10;

          MemWr=1'b0;

          MemtoReg=1'b0;

          j\_sel=1'b0;

        end

*//LW*

    else if(instruction[31:26]==6'b100011)

      begin

        nPC\_sel=2'b00;

          RegDst=1'b0;

          RegWr=1'b1;

          ExtOp=2'b01;

          ALUSrc=1'b1;

          ALUctr=2'b00;

          MemWr=1'b0;

          MemtoReg=1'b1;

          j\_sel=1'b0;

        end

*//SW*

      else if(instruction[31:26]==6'b101011)

        begin

          nPC\_sel=2'b00;

          RegDst=1'b0;

          RegWr=1'b0;

          ExtOp=2'b01;

          ALUSrc=1'b1;

          ALUctr=2'b00;

          MemWr=1'b1;

          MemtoReg=1'b0;

          j\_sel=1'b0;

        end

*//BEQ*

      else if(instruction[31:26]==6'b000100)

        begin

          nPC\_sel=2'b10;

          RegDst=1'b0;

          RegWr=1'b0;

          ExtOp=2'b01;

          ALUSrc=1'b0;

          ALUctr=2'b01;

          MemWr=1'b0;

          MemtoReg=1'b0;

          j\_sel=1'b0;

        end

*//LUI*

      else if(instruction[31:26]==6'b001111)

        begin

          nPC\_sel=2'b00;

          RegDst=1'b0;

          RegWr=1'b1;

          ExtOp=2'b10;

          ALUSrc=1'b1;

          ALUctr=2'b10;

          MemWr=1'b0;

          MemtoReg=1'b0;

          j\_sel=1'b0;

      end

*//J*

     else if(instruction[31:26]==6'b000010)

       begin

          nPC\_sel=2'b01;

          RegDst=1'b0;

          RegWr=1'b0;

          ExtOp=2'b01;

          ALUSrc=1'b0;

          ALUctr=2'b01;

          MemWr=1'b0;

          MemtoReg=1'b1;

          j\_sel=1'b1;

    end

  end

endmodule

(4) ifu.v

module **ifu**(nPC\_sel,zero,clk,rst,instruction,j\_sel,jValue);

  input clk,rst;

  input [1:0]nPC\_sel;

  input [31:0]zero;

  input j\_sel;

  input [25:0]jValue;

  output [31:0]instruction;

  reg [31:0]pc;

  reg [7:0]im[1023:0];

  reg [31:0]pcnew;

  wire [31:0]temp,t0,t1;

  wire [15:0]imm16;

  reg [31:0]extout;

*//give instruction a value*

  assign instruction={im[pc[9:0]],im[pc[9:0]+1],im[pc[9:0]+2],im[pc[9:0]+3]};

  assign imm16=instruction[15:0];

*//set extout value*

  assign temp={{16{imm16[15]}},imm16};

*//j condition*

  always@(\*)begin

    if(j\_sel==1)begin

      extout={pc[31:28],jValue[25:0],2'b0};

    end

      if(j\_sel==0)begin

      extout=temp[31:0]<<2;

    end

  end

*//set pcnew*

  assign t0=pc+4;

  assign t1=t0+extout;

  always@(\*)

  begin

    if(nPC\_sel==2'b00)begin

      pcnew=t0;

    end

    else if(nPC\_sel==2'b01)begin

      pcnew=t1;

    end

    else if(nPC\_sel==2'b10)begin

      if(zero==0)begin

        pcnew=t1;

      end

    else begin

      pcnew=t0;

    end

    end

  end

*//reset*

  always@(posedge clk,posedge rst)

  begin

    if(rst) pc=32'h0000\_3000;

      else if(j\_sel==0)pc=pcnew;

      else if(j\_sel==1)pc=extout;

  end

endmodule

(5) gpr.v

module **gpr**(RegWr,ra,rb,rw,busW,clk,rst,busA,busB,Data\_in);

  input clk,rst,RegWr;

  input [31:0]busW;

  input [4:0]ra,rb,rw;

  output [31:0]busA,busB,Data\_in;

  reg [31:0]regi[31:0];

*//reset*

  integer i;

  always@(posedge rst)

  begin

    if(rst)

      for(i=0;i<32;i=i+1)

      regi[i]=0;

    end

*//set busA & busB*

  assign busA=regi[ra];

  assign busB=regi[rb];

  assign Data\_in=busB;

*//Register write in*

  always@(posedge clk)begin

    if(RegWr)begin

      regi[rw]<=busW;

      regi[0]<=0;

    end

  end

endmodule

(6) alu.v

module **alu**(busA,busB,ALUctr,zero,Alu\_out,Addr);

  input [31:0]busA,busB;

  input [1:0]ALUctr;

  output [31:0]zero,Addr;

  output reg[31:0]Alu\_out;

*//set ADD,SUB,OR*

  parameter ADD=2'b00;

  parameter SUB=2'b01;

  parameter OR=2'b10;

*//three conditions*

  always@(\*)begin

    case(ALUctr)

      ADD:begin

        Alu\_out=busA+busB;

      end

      SUB:begin

        Alu\_out=busA-busB;

      end

      OR:begin

        Alu\_out=busA|busB;

      end

    endcase

  end

  assign zero=Alu\_out;

  assign Addr=Alu\_out;

endmodule

(7) dm.v

module **dm**(Data\_in,MemWr,Addr,clk,rst,Data\_out);

  input [31:0]Data\_in,Addr;

  input clk,rst,MemWr;

  output reg[31:0]Data\_out;

  reg [7:0]DataMem[1023:0];

  wire [9:0]pointer;

  assign pointer=Addr[9:0];

*//reset*

  integer i;

  always@(negedge rst)begin

    for(i=0;i<1024;i=i+1)

    DataMem[i]=0;

  end

  always@(posedge clk)begin

*//store word*

    if(MemWr==1)begin

      DataMem[pointer]<=Data\_in[31:24];

      DataMem[pointer+1]<=Data\_in[23:16];

      DataMem[pointer+2]<=Data\_in[15:8];

      DataMem[pointer+3]<=Data\_in[7:0];

    end

  end

  always@(negedge clk)begin

*//load word*

  if(MemWr==0)begin

    Data\_out<={DataMem[pointer],DataMem[pointer+1],DataMem[pointer+2],DataMem[pointer+3]};

  end

end

endmodule

(8) ext.v

module **ext**(imm16,imm32,ExtOp);

  input [15:0]imm16;

  input [1:0]ExtOp;

  output reg[31:0]imm32;

*//set ZERO,SIGN*

  parameter ZERO=2'b00;

  parameter SIGN=2'b01;

  parameter LUI=2'b10;

*//two conditions*

  always@(\*)begin

    case(ExtOp)

      ZERO:imm32={16'b0,imm16};

      SIGN:imm32={{16{imm16[15]}},imm16};

      LUI:imm32={imm16,16'b0};

    endcase

  end

endmodule

(9) mux.v

module **mux**(a0,a1,op,out);

  input op;

  input [31:0]a0,a1;

  output reg[31:0]out;

  always@(\*)begin

    if(op) out=a1;

    else out=a0;

    end

  endmodule

(10) mux\_RegDst.v

module **mux\_RegDst**(a1,a0,rw,RegDst);

  input[4:0]a1,a0;

  input RegDst;

  output reg[4:0]rw;

always@(\*)begin

  if(RegDst)rw=a1;

  else rw=a0;

  end

endmodule

(11) code.asm

**ori** $3,$0,0x93

**ori** $6,$0,0xae

**addu** $8,$3,$6

**subu** $9,$3,$6

**addu** $0,$9,$10

**sw** $9,16($0)

**lw** $10,16($0)

**l3**:**beq** $9,$10,l1

**lui** $11,0xcdcd

**j** end

**l1**:**ori** $11,$0,0xefef

**lui** $9,0x4567

**j** l3

**end**:

(12)code.txt

34030093

340600ae

00664021

00664823

012a0021

ac090010

8c0a0010

112a0002

3c0bcdcd

0810000d

340befef

3c094567

08100007

(13)test.v

module **test**;

  reg clk,rst;

  mips launch(.clk(clk),.rst(rst));

  initial begin

    rst=0;

    clk=1;

    #1 rst=1;

    #2 rst=0;

**$readmemh**("code.txt",launch.MAIN.IFU.im);

  end

  always

  #30 clk=~clk;

endmodule