编号： 08-021830122-贾志军

南京航空航天大学

实验报告

题 目 单周期 MIPS CPU 的设计（带溢出判断36 条）、

学生姓名 贾志军

学 号 021830122

学 院 计算机科学与技术/人工智能学院

专 业 计算机科学与技术

班 级 1618101

指导教师 施慧彬

目录

[摘 要 3](#_Toc45028583)

[第一章 引 言 4](#_Toc45028584)

[1.1 单周期 CPU 的理论结构 4](#_Toc45028585)

[1.1.1 PC 模块定义 4](#_Toc45028586)

[1.1.2 NPC 模块定义 5](#_Toc45028587)

[1.1.3 ALU 模块定义 6](#_Toc45028588)

[1.1.4 dm 模块定义 6](#_Toc45028589)

[1.1.5 im 模块定义 6](#_Toc45028590)

[1.1.6 regist 模块定义 7](#_Toc45028591)

[1.1.7 ctrl 模块定义 8](#_Toc45028592)

[1.1.8 ext 模块定义 8](#_Toc45028593)

[1.1 各个指令与控制信号之间的关系 10](#_Toc45028594)

[第二章 单周期 MIPS CPU 的具体设计与调试 12](#_Toc45028595)

[2.1 具体 Verilog 代码实现 12](#_Toc45028596)

[2.2 ModelSim 模拟及测试 24](#_Toc45028597)

[第三章 总结与展望 25](#_Toc45028598)

单周期 MIPS CPU 的设计

# 摘 要

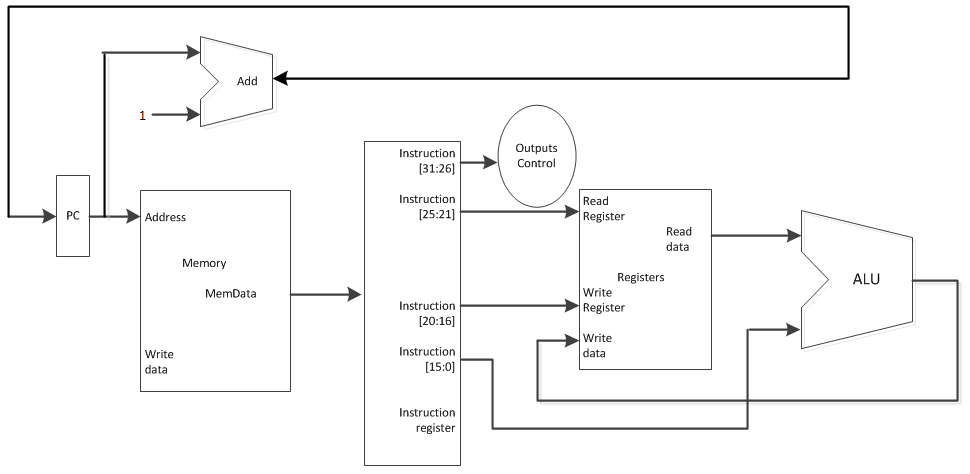
为了提升对单周期 MIPS CPU 的理解能力和动手能力，以个人为单位，完成一个单周期 CPU 的开发。研究的结果与主要结论：能写出一个能执行 add，sub，and，or，slt，lw，sw，beq 和jump 等 36 条指令，支持溢出的单周期 MIPS CPU。

关键词：单周期，处理器，汇编语言，模拟

# 第一章 引 言

## 1.1 单周期 CPU 的理论结构

单周期处理器的数据通路结构如图所示。



### 1.1.1 PC 模块定义

（1）基本描述

PC 主要功能是完成输出当前指令地址。复位后，PC 指向 0x0000\_3000，此处为第

一条指令的地址。

（2）模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.1 PC接口的模块 |  |
| 信号名 | 方向 | 描述 |
| [31:0] NPC | I | 输入的指令地址 |
| clk | I | 时钟信号 |
| rst | I | 复原信号 |
| [31:0] PC | O | 输出当前指令地址 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | 表1.2 PC接口的功能 | |  | |
| 序号 | | | | 功能名称 | | 功能描述 | |
| 1 | | | | 复位 | | rst=1时地址复位为0x0000 0300 | |
| 2 | | | | 输出指令地址 | | 时钟到来是，将NPC赋给PC | |
|  |  | 1.1.2 NPC 模块定义 | |  | |  | |  |
|  |  | （1）基本描述 | |  | |  | |  |

NPC 主要功能是根据当前指令是否为跳转分支指令，输出下一条指令的地址。该模块

调用了 signext 模块。

（2）模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.3 NPC接口的模块 |  |
| 信号名 | 方向 | 描述 |
| zero | I | ALU运算结果是否为0 |
| [31:0] pc | I | 当前地址 |
| [31:0]busA | I | Rt寄存器的值 |
| [25:0]Target | I | J型指令的地址段 |
| [5:0]op | I | 操作码 |
| [5:0]func | I | R型指令标识段 |
| jump | I | 是否为J型指令 |
| [15:0]imm16 | I | 立即数 |
| [4:0]rt | I | Rt寄存器的值 |
| [31:0]NPC | O | 下条指令的地址 |
| Branch | I | 是否需要跳转Branch地址 |
|  | 表1.4 NPC接口的功能 |  |
| 序号 | 功能名称 |  |
| 1 | 输出指令地址 | 根据zero和branch输出下条地址 |
| 2 | 输出指令地址 | 根据jump和target输出下条地址 |

### 1.1.3 ALU 模块定义

1. 基本描述

实现addu，subu，slt，and，nor，or，xor，sll，srl，sltu ，sllv，sra，srav，srlv，lui，slti，sltiu 等基本操作

（2）模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.5 ALU的接口模块 |  |
| 信号名 | 方向 | 描述 |
| [4:0]ALUctr | I | ALU控制信号 |
| [31:0]busA | I | ALU运算值 |
| [31:0] tempBus | I | ALU运算值 |
| [4:0] shamt | I | shamt段 |
| [31:0] Result | O | ALU运算结果 |
| Zero | O | ALU运算结果是否为0 |

### 1.1.4 dm 模块定义

数据内存大小为 4K，根据输入的地址读出数据内存中的数据，并根据数据写信号，将输入的数据选择写入数据内存中。

1. 模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.7dm的接口模块 |  |
| 信号名 | 方向 | 描述 |
| [31:0] addr | I | 地址信号 |
| we | I | 写使能 |
| clk | I | 时钟 |
| [31:0] din | I | 写入地址的值 |
| [5:0] op | I | 操作码 |
| [31:0] dout | O | 读取内存地址的值 |

1. 功能模块

|  |  |  |
| --- | --- | --- |
|  | 表1.8dm的功能模块 |  |
| 序号 | 功能名称 | 功能描述 |
| 1 | 读数据主存数据 | 根据地址，读取内存值 |
| 2 | 写入主存 | 根据写使能和写地址主存写值 |

### 1.1.5 im 模块定义

(1）基本描述

指令内存大小为 4K，初始化从 code.txt 载入指令。根据输入的指令地址，输出当前位置存储的指令。

（2）模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.9im的接口模块 |  |
| 信号名 | 方向 | 描述 |
| [31:0]addr | I | 指令地址 |
| [31:0]dout | O | 指令地址 |

（3）功能模块

|  |  |  |
| --- | --- | --- |
|  | 表1.10im的功能模块 |  |
| 序号 | 功能名称 | 功能描述 |
| 1 | 载入指令 | 初始化载入 code.txt 中的指令 |
| 2 | 输出指令 | 根据输入指令地址，输出当前指令 |

### 1.1.6 regist 模块定义

（1）基本描述

根据输入的两个寄存器地址，输出相应寄存器的值，根据寄存器写信号和寄存器地址，将输入的数据选择写入寄存器。

(2）模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.12register的功能模块 |  |
| 序号 | 功能名称 | 功能描述 |
| 1 | 读寄存器数据 | 读 rs、rt，rd 寄存器的数据 |
| 2 | 向寄存器写入数据 | 根据写信号向寄存器选择写数据 |

|  |  |  |
| --- | --- | --- |
|  | 表1.11regist的接口模块 |  |
| 信号名 | 方向 | 描述 |
| [31:0]addr | I | 指令地址 |
| [4:0] Rw | O | 指令地址 |
| [4:0] Ra | I | Rs寄存器 |
| [4:0] Rb | I | Rt寄存器 |
| [4:0] rt | I | rt段 |
| [5:0]op | I | 操作码 |
| [5:0]func | I | func段 |
| [31:0] PC | I | 当前指令地址 |
| [31:0] busW | I | 写入寄存器的值 |
| [31:0] busA | O | Rs寄存器的值 |
| [31:0] busB | O | Rt寄存器的值 |

（3）功能接口

### 1.1.7 ctrl 模块定义

（1） 基本描述根据输入的指令高 6 位(op 和 func 字段)，利用真值表化简，输出

Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp,Rtype,ALUop,funcop,AL

Uctr 控制信号。真值表采用书上 200 页图 4-22，再加上 jump 指令的输入输出。

（2）模块接口

|  |  |  |
| --- | --- | --- |
|  | 表1.13ctrl的接口模块 |  |
| 信号名 | 方向 | 描述 |
| [5:0] op | I | 操作码 |
| [5:0] func | I | func段 |
| Branch | O | 是否满足分支 |
| Jump | O | 是否为跳转指令 |
| RegDst | O | 主存是否写 |
| ALUSrc | O | ALUSrc值 |
| MemtoReg | O | 主存写寄存器 |
| RegWr | O | 是否写寄存器 |
| MemWr | O | 是否写主存 |
| Rtype | O | 是否为R指令 |
| ExtOp | O | 是否需要符号扩展 |
| [4:0] ALUop | O | ALUop控制信号 |
| [4:0] funcop | O | funcop控制信号 |
| [4:0] ALUctrl | O | ALU控制信号 |

(3)功能模块

|  |  |  |
| --- | --- | --- |
|  | 表1.14ctrl的功能模块 |  |
| 序号 | 功能名称 | 功能描述 |
| 1 | 输出各种控制信号 | 输出各个控制信号 |

### 1.1.8 ext 模块定义

（1） 根据ExtOp是否需要符号扩展对立即数扩展

（2）模块接口

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 表1.15ext的接口模块 | |  | |
| 信号名 | 方向 | | 描述 | |
| [15:0] imm16 | I | | 操作码 | |
| [31:0] imm32 | O | | func段 | |
| ExtOp | I | | 是否满足分支 | |
| （3）功能模块 | 表1.16ext的功能模块 | |  | |
| 序号 | 功能名称 | | 功能描述 | |
| 1 | 立即数扩展 | | 根据Extop对立即数扩展 | |
| 2 | 向寄存器写入数据 | | 根据写信号向寄存器选择写数据 | |
| 1.1.9 instruction 模块定义  （1） 更具指令输出译码信息  （2）模块接口  表1.17instruction的接口模块 | |  | |
| 信号名 | 方向 | | 描述 | |
| [31:0] Instruction | I | | 指令 | |
| [31:26] op | O | | 操作码 | |
| [25:21] rs | O | | rs寄存器 | |
| [20:16] rt | O | | rt寄存器 | |
| [15:11] rd | O | | rd寄存器 | |
| [10:6] shamt | O | | shamt段 | |
| [5:0] func | O | | func段 | |
| [15:0] imm16 | O | | 立即数 | |
| [25:0] Target | O | | Target段 | |

（3）功能模块

|  |  |  |
| --- | --- | --- |
|  | 表1.18instruction的功能模块 |  |
| 序号 | 功能名称 | 功能描述 |
| 1 | 指令译码 | 根据指令译码 |
| 1.1.10 datapath 模块定义  （1） 更具指令输出译码信息  （2）模块接口  、  信号名 | 表1.19datapath的接口模块  方向 | 描述 |
| clk | I | 时钟 |
| rst | I | 复原信号 |
| Branch | I | 是否分支跳转 |
| Jump | I | 是否跳转指令 |
| RegDst | I | 数据选择器信号 |
| ALUSrc | I | ALUSrc段 |
| MemtoReg | I | 主存写寄存器 |
| RegWr | I | 是否写寄存器 |
| MemWr | I | 是否写主存 |
| ExtOp | I | 是否要符号扩展 |
| [4:0] ALUctr | I | ALU控制信号 |
| [5:0] op | O | 操作码 |
| [5:0] func | O | func段 |
| [4:0] Rt | O | Rt寄存器 |
| （3）功能模块 | 表1.20Datapath的功能模块 |  |
| 序号 | 功能名称 | 功能描述 |
| 1 | 数据通路 | 连接数据通路 |

* 1. 各个指令与控制信号之间的关系

#### R-type 指令与控制信号

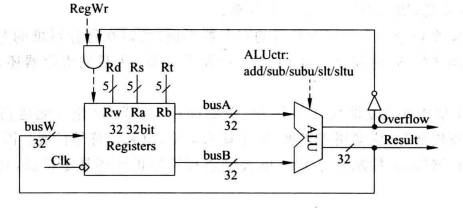
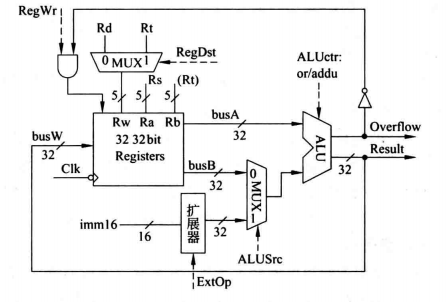


图 1.2 R-type 指令与控制信号

#### I 指令与控制信号



# 第二章 单周期 MIPS CPU 的具体设计与调试

## 2.1 具体 Verilog 代码实现

module PC(input [31:0]NPC,output reg [31:0]pc,input rst,input clk);

always@(posedge clk)

begin

if(rst)

pc<=32'h0000\_0000;

else

pc<=NPC;

end

endmodule

module ALU(ALUctr,busA,tempBus,shamt,Result,Zero

);

input[4:0] ALUctr;

input [31:0] busA,tempBus;

input [4:0] shamt;

output reg[31:0] Result;

output Zero;

assign Zero=(Result==0);

always @(busA or tempBus or ALUctr)

begin

case(ALUctr)

//R Type

5'b00000 : Result <= busA + tempBus; // addu

5'b00001 : Result <= busA - tempBus; // subu

5'b00010 : Result <= busA < tempBus;// slt

5'b00011 : Result <= busA & tempBus; // and

5'b00100 : Result <= ~(busA | tempBus); // nor

5'b00101 : Result <= busA | tempBus; // or

5'b00110 : Result <= busA ^ tempBus; // xor

5'b00111 : Result <= tempBus << shamt; //sll

5'b01000 : Result <= tempBus >> shamt;// srl

5'b01001 : Result <= (busA <tempBus)?1:0; // sltu return 0

//5'b01010 : ; jalr:there's no pc

//5'b01011 : ; jr:there's no pc

5'b01100 : Result <= tempBus << busA; //sllv

5'b01101 : Result <= ($signed(tempBus)) >>> shamt; //sra

5'b01110 : Result <=($signed(tempBus)) >>> busA; //srav

5'b01111 : Result <= tempBus >> busA; //srlv

5'b10000 : Result <= (tempBus << 5'b10000); //lui

endcase

end

// always @(A,B,ALUctr)

// begin

// SUBctr=ALUctr[2];

// OVctr=!ALUctr[1]&ALUctr[0];

// SIGctr=ALUctr[0];

// OPctr[1]=ALUctr[2]&ALUctr[1];

// OPctr[0]=!ALUctr[2]&ALUctr[1]&!ALUctr[0];

// EXT={32{SUBctr}};

// EXT\_A=B^EXT;

// {A\_Carry2,A\_Result[30:0]}=A[30:0]+EXT\_A[30:0]+SUBctr;

// {A\_Carry,A\_Result[31]}=A[31]+EXT\_A[31]+A\_Carry2;

// A\_Overflow=A\_Carry^A\_Carry2;

// A\_Sign=A\_Result[31];

// less=(SIGctr==1)?(A\_Sign^A\_Overflow):(A\_Carry^SUBctr);

// if(OPctr==2'b00)

// Result=A\_Result;

// else if(OPctr==2'b01)

// Result=A|B;

// else if(OPctr==2'b10)

// begin

// if(less==1)

// Result={32{1'b1}};

// else Result={32{1'b0}};

// end

// Zero=(Result==0)?1'b1:1'b0;

// Overflow=A\_Overflow&OVctr;

// end

Endmodule

module ctrl(op,func,Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp,Rtype,ALUop,funcop,ALUctr);

input [5:0] op,func;

output Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp,Rtype;

output[4:0] ALUop,funcop,ALUctr;

assign Branch=(!op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&!op[3]&op[2]&!op[1]&op[0])|(!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&op[0])|(!op[5]&!op[4]&!op[3]&op[ 2]&op[1]&op[0])|(!op[5]&!op[4]&!op[3]&op[2]&op[1]&!op[0]);

assign Jump=(!op[5]&!op[4]&!op[3]&!op[2]&op[1]&!op[0])|(!op[5]&!op[4]&!op[3]&!op[2]&op[ 1]&op[0]);

assign RegDst=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0];

assign ALUSrc=(!op[5]&!op[4]&op[3]&!op[2]&!op[1]&op[0])|(op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])|(op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])|(!op[5]&!op[4]&op[3]&op[2]&op[1]&op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[1]&!op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])|(op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0])|(op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0])|(op[5]&!op[4]&op[3]&!op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0])|(!op[5]&!op[4]&op[3]&op[2]&op[1]&!op[0]);

assign MemtoReg=(op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])|(op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0])|(op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0]);

assign RegWr=(!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&!op[2]&!op[1]&op[0])|(op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])|(!op[5]&!op[4]&op[3]&op[2]&op[1]&op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[1]&!op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])|(op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0])|(op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0])|(!op[5]&!op[4]&op[3]&op[2]&op[1]&!op[0])|(!op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0]);

assign MemWr=(op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])|(op[5]&!op[4]&op[3]&!op[2]&!op[1]& !op[0]);

assign ExtOp=(op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0])|(op[5]&!op[4]&op[3]&!op[2]&op[1]& op[0])|(!op[5]&!op[4]&op[3]&!op[2]&!op[1]&op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[ 1]&!op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0]);

assign Rtype=!op[5]&!op[4]&!op[3]&!op[2]&!op[1]&!op[0];

assign ALUop[0]=(!op[5]&!op[4]&!op[3]&op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&!op[3]&op[2]& !op[1]&op[0])|(!op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])|(!op[5]&!op[4]&op[3]&op[2 ]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0]);

assign ALUop[1]=(!op[5]&!op[4]&op[3]&!op[2]&op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&!op[1]&!op[0])|(!op[5]&!op[4]&op[3]&op[2]&op[1]&!op[0])|(!op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0]);

assign ALUop[2]=(!op[5]&!op[4]&op[3]&op[2]&!op[1]&op[0])|(!op[5]&!op[4]&op[3]&op[2]&op[1]&!op[0]);

assign ALUop[3]=(!op[5]&!op[4]&op[3]&!op[2]&op[1]&op[0])|(!op[5]&!op[4]&!op[3]&!op[2]&op[1]&op[0]);

assign ALUop[4]=(!op[5]&!op[4]&op[3]&op[2]&op[1]&op[0]);

assign funcop[0]=(func[5]&!func[4]&!func[3]&!func[2]&func[1]&func[0])|(func[5]&!func[4]&!func[3]&func[2]&!func[1]&!func[0])|(func[5]&!func[4]&!func[3]&func[2]&!func[1]&func[0])|(!func[5]&!func[4]&!func[3]&!func[2]&!func[1]&!func[0])|(func[5]&!func[4]&func[3]&!func[2]&func[1]&func[0])|(!func[5]&!func[4]&func[3]&!func[2]&!func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&!func[2]&func[1]&func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[1]&!func[0]);

assign funcop[1]=(func[5]&!func[4]&func[3]&!func[2]&func[1]&!func[0])|(func[5]&!func[4] &!func[3]&func[2]&!func[1]&!func[0])|(func[5]&!func[4]&!func[3]&func[2]&func[1]& !func[0])|(!func[5]&!func[4]&!func[3]&!func[2]&!func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[1]&func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[ 1]&!func[0])|(!func[5]&!func[4]&func[3]&!func[2]&!func[1]&func[0])|(!func[5]&!func[4]&func[3]&!func[2]&!func[1]&!func[0]);

assign funcop[2]=(func[5]&!func[4]&!func[3]&func[2]&func[1]&func[0])|(func[5]&!func[4]&!func[3]&func[2]&!func[1]&func[0])|(func[5]&!func[4]&!func[3]&func[2]&func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&!func[2]&!func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&func[2]&!func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&!func[2]&func[1]&func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[1]&func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[1]&!func[0]);

assign funcop[3]=(!func[5]&!func[4]&!func[3]&!func[2]&func[1]&!func[0])|(func[5]&!func[4]&func[3]&!func[2]&func[1]&func[0])|(!func[5]&!func[4]&func[3]&!func[2]&!func[1]&func[0])|(!func[5]&!func[4]&func[3]&!func[2]&!func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&func[2]&!func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&!func[2]&func[1]&func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[1]&!func[0])|(!func[5]&!func[4]&!func[3]&func[2]&func[1]&func[0]);

assign funcop[4]=0;

assign ALUctr=(Rtype)?funcop:ALUop;

endmodule

module DataPath(clk,rst,Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp,ALUctr,op,func,Rt);

input clk,rst,Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp;

input [4:0] ALUctr;

output [5:0] op,func;

output [4:0] Rt;

wire [31:0] pc,NPC;

wire[31:0] instructions;

PC PC(NPC,pc,rst,clk);

im\_4k im(pc[11:2],instructions);

wire [4:0] Rs,Rt,Rd,shamt;

wire[15:0] imm16;

wire[25:0] Target;

instruction instruction(instructions,op,Rs,Rt,Rd,shamt,func,imm16,Target);

wire [31:0] Result;

wire[31:0] busW,busA,busB;

wire[4:0] Rw,Ra,Rb;

assign Rw=RegDst?Rd:Rt;

assign Ra=Rs;

assign Rb=Rt;

register register(Result,RegWr,Rw,Ra,Rb,busW,clk,busA,busB,op,func,Rd,pc);

wire[31:0] imm32;

ext ext(imm16,imm32,ExtOp);//ext模块还没写

wire zero;

wire[31:0] tempBus;

assign tempBus=ALUSrc?imm32:busB;

ALU alu(ALUctr,busA,tempBus,shamt,Result,zero);

wire[31:0] dout;

dm\_4k dm(Result,busB,MemWr,op,clk,dout);

assign busW=MemtoReg?dout:Result;

npc npc(pc,NPC,Target,Jump,Branch,zero,imm16,op,func,busA,Rt);

endmodule

module dm\_4k( //

addr,// Address In

din,//Data in

we,//Write Enable

// clock

op,

clk,

dout//Data Out

);

input [31:0] addr;

input wire we,clk;

input [31:0] din;

input [5:0] op;

output [31:0] dout;

reg [31:0] dm [1023:0];//1024\*4=4096

integer i;

initial

begin

for(i=0;i<1024;i=i+1)

dm[i]=32'b0;

end

assign dout=dm[addr[11:2]];

always @ (posedge clk)

begin

if(we)

begin

if(op==6'b101000) //sb 主存存字节

begin

if(addr[1:0]==2'b00)

dm[addr[11:2]][7:0]<=din[7:0];

if(addr[1:0]==2'b01)

dm[addr[11:2]][15:8]<=din[7:0];

if(addr[1:0]==2'b10)

dm[addr[11:2]][23:16]<=din[7:0];

if(addr[1:0]==2'b11)

dm[addr[11:2]][31:24]<=din[7:0];

end

else

dm[addr[11:2]]<=din;

end

end

endmodule

module ext(imm16,imm32,ExtOp);

input [15:0] imm16;

output [31:0] imm32;

input ExtOp;

assign imm32=(ExtOp==1)?{{16{imm16[15]}},imm16}:{16'b0,imm16};

endmodule

module im\_4k(

addr,//Address in

dout//Data Out

);

input [11:2] addr;

output [31:0] dout;

reg[31:0] im[1023:0];//1024\*4=4096

initial begin

$readmemh("code.txt",im); //主存去指令

end

assign dout=im[addr];

endmodule

module instruction(Instruction,op,rs,rt,rd,shamt,func,imm16,Target);

input [31:0] Instruction;

output [31:26] op;

output [25:21] rs;

output [20:16] rt;

output [15:11] rd;

output [10:6] shamt;

output [5:0] func;

output [15:0] imm16;

output [25:0] Target;

assign op=Instruction[31:26];

assign rs=Instruction[25:21];

assign rt=Instruction[20:16];

assign rd=Instruction[15:11];

assign shamt=Instruction[10:6];

assign func=Instruction[5:0];

assign imm16=Instruction[15:0];

assign Target=Instruction[25:0];

endmodule

module mips(

clk,

rst

);

input clk;

input rst;

wire [5:0] op,func;

wire [4:0] ALUctr,funcop,ALUop,Rt;

wire RegWr,ALUSrc,RegDst,MemtoReg,MemWr,Branch,Jump,ExtOp,Rtype;

DataPath datap(clk,rst,Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp,ALUctr,op,func);

ctrl ctrl(op,func,Branch,Jump,RegDst,ALUSrc,MemtoReg,RegWr,MemWr,ExtOp,Rtype,ALUop,funcop,ALUctr);

Endmodule

module npc(

pc,

NPC,

Target,

jump,

Branch,

zero,

imm16,

op,

func,

busA,

rt

);

input [31:0] pc,busA;

input [25:0] Target;

input jump,Branch,zero;

input[5:0] op,func;

input[4:0] rt;

input [15:0] imm16;

output [31:0] NPC;

wire [31:0]pc,busA;

reg [31:0] NPC;

wire [29:0] imm30,func30;

wire[25:0] Target;

wire jump,Branch,zero;

wire [5:0] op,func;

wire [4:0] rt;

wire[15:0] imm16;

wire[31:0] imm32,func32;

signext #(16,30) signext(imm16,imm30);

assign imm32={imm30,2'b00};

wire[31:0] N\_NPC=pc+4;

wire[31:0] B\_NPC=N\_NPC-4+imm32;

wire[31:0] J\_NPC={pc[31:28],Target[25:0],2'b00};

wire[31:0] BGEZ\_NPC=pc+func32;

always@(\*)

begin

case(op)

6'b000100:NPC=(zero&Branch)?B\_NPC:N\_NPC; //

6'b000101:NPC=(!zero&Branch)?B\_NPC:N\_NPC;

6'b000110:NPC=(Branch&&(busA==0||busA[31]==1))?B\_NPC:N\_NPC;

6'b000111:NPC=(Branch&&busA!=0&&busA[31]==0)?B\_NPC:N\_NPC;

6'b000010:NPC=(jump)?J\_NPC:N\_NPC;

6'b000011:NPC=(jump)?J\_NPC:N\_NPC;//jal

6'B000001:

begin

if(Branch&&rt==0) //bgez 和

NPC=(busA[31]==1)?B\_NPC:N\_NPC;

else

if(Branch&&rt==1)

NPC=(busA==0|busA[31]==0)?B\_NPC:N\_NPC;

end

default:

begin

if(op==6'b000000&&((func==6'b001000)|(func==6'b001001)))

begin

NPC=busA;

end

else

NPC=N\_NPC;

end

endcase

end

endmodule

module register(addr,RegWr,Rw,Ra,Rb,busW,clk,busA,busB,op,func,rd,PC);

input clk,RegWr;

input[31:0] addr;

input [4:0] Rw,Ra,Rb,rd;

input [5:0] op,func;

input [31:0] busW,PC;

output[31:0] busA,busB;

reg[31:0] regist[31:0]; //寄存器

integer i;

initial // 注意！！！！！

begin

i=0;

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

regist[i]=0;

end

always@(posedge clk)

begin

if(RegWr)

case(op)

6'b100011://lw

begin

regist[Rw]<=busW;

end

6'b100000:

begin

if(addr[1:0]==2'b00)

regist[Rw]<={{24{busW[7]}},busW[7:0]};

if(addr[1:0]==2'b01)

regist[Rw]<={{24{busW[15]}},busW[15:8]};

if(addr[1:0]==2'b10)

regist[Rw]<={{24{busW[23]}},busW[23:16]};

if(addr[1:0]==2'b11)

regist[Rw]<={{24{busW[31]}},busW[31:24]};

end

6'b100100://lbu

begin

if(addr[1:0]==2'b00)

regist[Rw]<={{24'b0},busW[7:0]};

if(addr[1:0]==2'b01)

regist[Rw]<={{24'b0},busW[15:8]};

if(addr[1:0]==2'b10)

regist[Rw]<={{24'b0},busW[23:16]};

if(addr[1:0]==2'b11)

regist[Rw]<={{24'b0},busW[31:24]};

end

6'b000011:regist[31]<=PC+4;//jal 注意下！

default://jalr

begin

if(func==6'b001001&&rd==5'b11111)

regist[31]<=PC+4;

else

regist[Rw]<=busW; //lw Rtype

end

endcase

end

assign busA=(Ra!=0)?regist[Ra]:0;

assign busB=(Rb!=0)?regist[Rb]:0;

endmodule

module testbanch\_single;

reg clk;

reg rst;

mips mips(clk,rst);

initial

begin

rst=1;

clk=0;

#10 rst=0;

end

always

#5 clk=~clk;

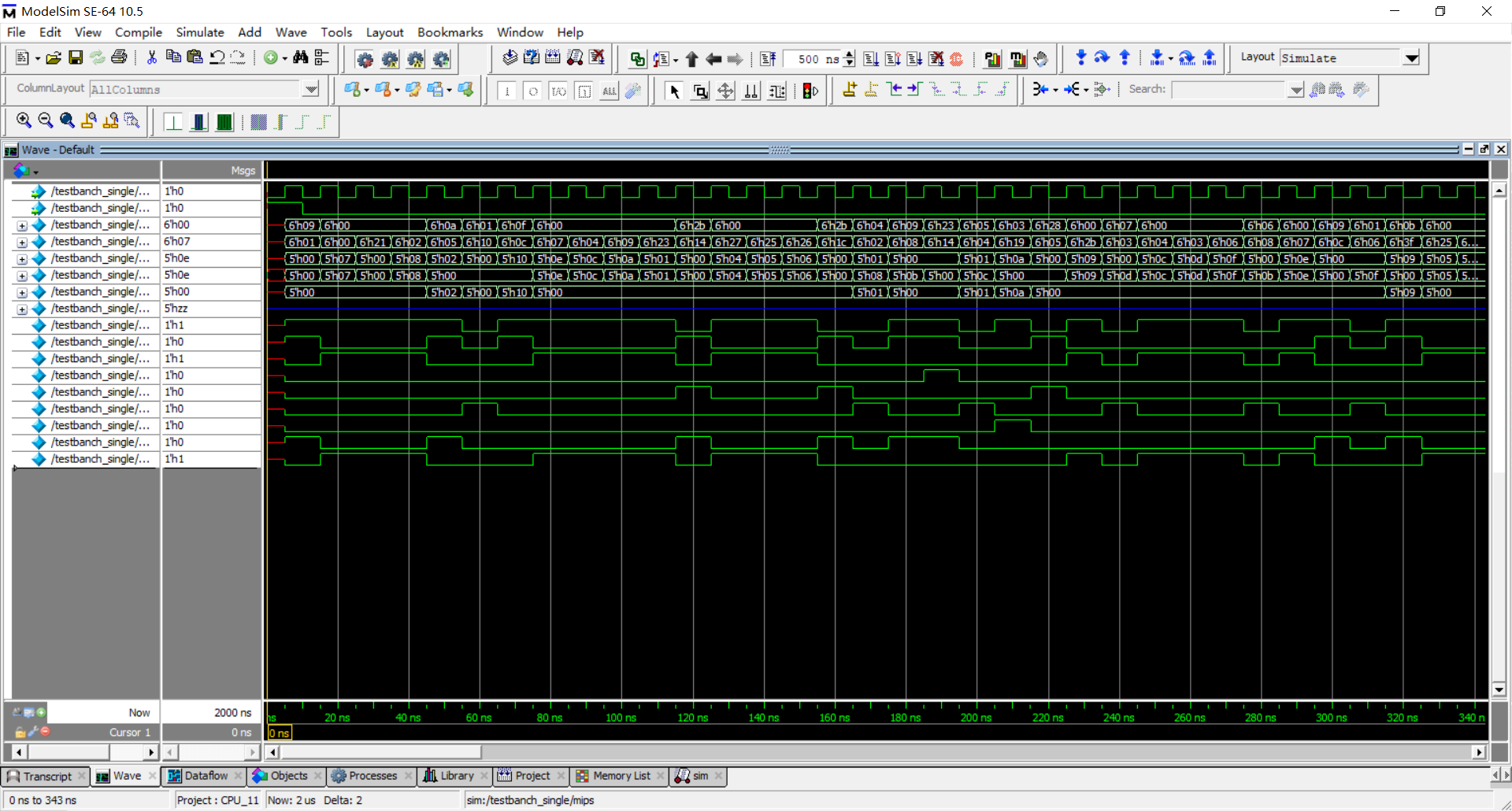
## 2.2 ModelSim 模拟及测试

2.2.1 综合测试用代码

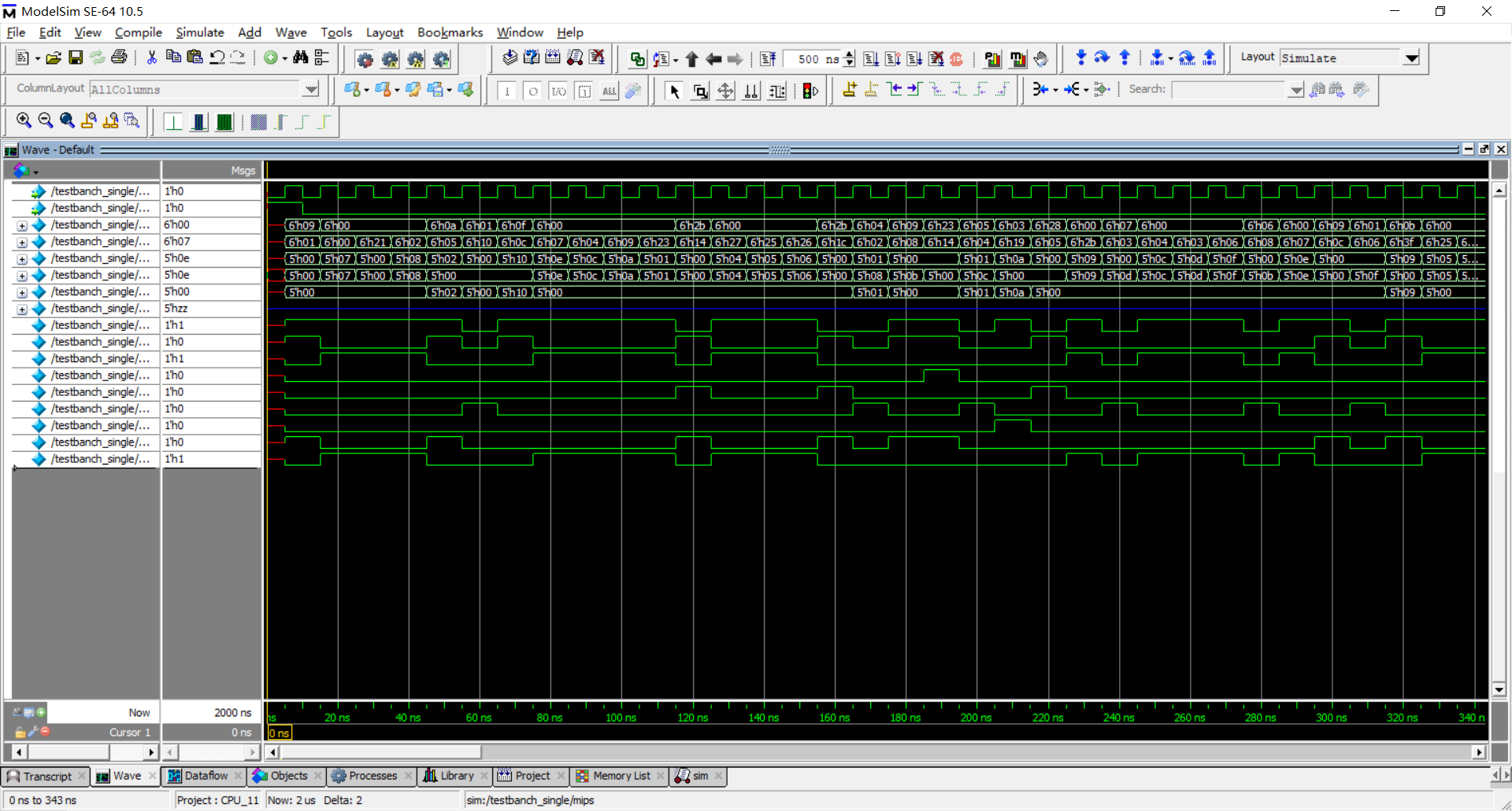
（1） 汇编代码

（2） MARS 软件处理后的 DUMP 文件（HEX）

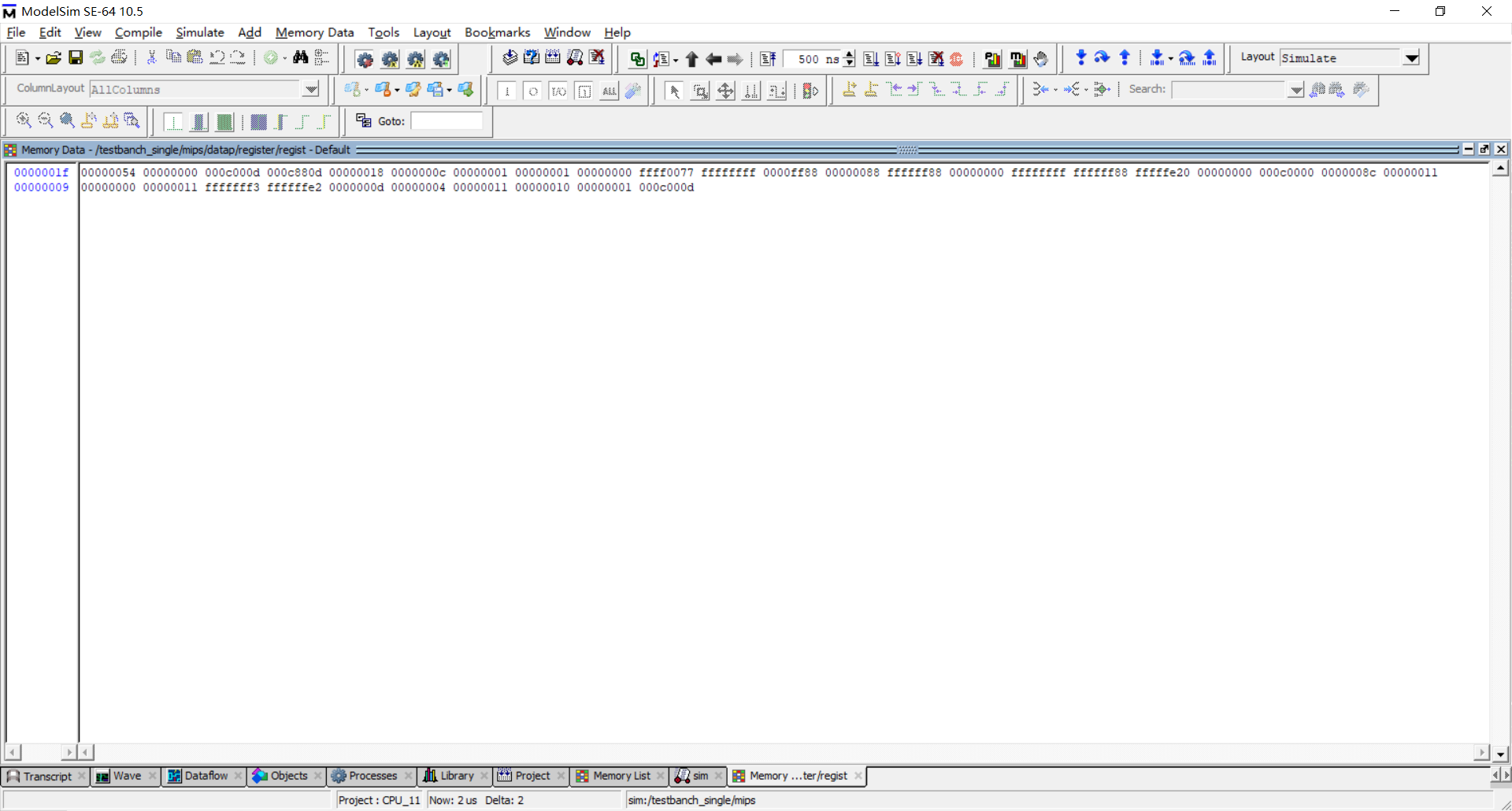
（3） 运行结果如下



。



1. 寄存器如下



# 第三章 总结与展望

实验中遇到很多问题，

（1）PC子模块中最开始的时候没有使用非阻塞赋值，导致PC一直出错。非阻塞赋值使得在结构块结束的时候才进行赋值，而不是立即赋值，后来使用非阻塞赋值后解决了这个错误。

（2）if（PCSrc==1) assign NextPC = PC + 4；将这样的if语句放在过程块中一直出错，一直弄不懂原因，后来请教同学，发现将其放在always结构块中并且检查NextPC的类型就可以解决问题。

（3）若将assign用在过程块initial和always中，此时只支持reg类型，但是对于非过程块中连续赋值，assign支持net型。

（4）对于Extend子模块，我将sa的符号位扩展桶16为立即数的扩展放在一起，使得在执行sll指令的时候能够保证sa扩展为32位。

（5）对于sll,实现起来遇到的麻烦比较大，因为需要同时控制ALUSrcA和ALUSrcB，最终是在CPU主模块中添加了ALUSrcA来作为判断，当执行的ALUSrcA为1的时候，送入到ALU中的data1不再是rs中的数据而是扩展后的sa.

（6） 最开始的时候只写了if语句而对else语句不做处理，本来觉着没什么问题，但是执行后却发现当sll执行过后ALUSrcA就一直为1，不改变，最后添加了else语句使问题得到解决。

（7）最开始的时候对于仿真后出现XXX和ZZZ的结果完全不知所措，因为代码也没有报错。后来发现是因为在CPU中各个子模块中的参数的传递顺序不一致

（8）一些小细节的错误，比如单词拼写错误或者是input后面的“，”写成了“；”之类的，或者是output最后不能有“，”或者是“；”。这种错误报错的时候很不明确，所以检查的时候很麻烦。

关键是理解数据通路图，对每条指令，通过数据通路图完整地走几遍能够有效帮助理解。

写代码花了几天时间，调试花了两倍的时间，幸好最终问题都解决了

，感谢老师的付出。

参 考 文 献

1. MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set[M]. Revision 2.50,

Mountain View, CA: MIPS Technologies Inc., July 1, 2005.

1. 袁春风，等. 计算机组成与系统结构[M]. 第 2 版，北京：清华大学出版社，2010.