# 计算机组成原理P4单周期CPU实验报告

20373944 何天然

## 一、CPU设计方案综述

### （一）总体设计概述

本CPU为Verilog实现的单周期MIPS-CPU，支持的指令集包含{ lw、sw、lb、lbu、sb、lh、lhu、sh、add、addu、sub、subu、and、or、slt、nor、xor、sll、srl、sra、sllv、srlv、srav、sltu、beq、bne、bgtz、blez、bltz、bgez、addi、addiu、andi、ori、xori、lui、slti、sltiu、j、jr、jal、jalr }共42条，其中add和sub不支持溢出中断，其行为与addu和subu完全一致。为了实现这些功能，CPU主要包含了Controller、Datapath、ALU、DM、EXT、GRF、IM、NPC、PC，这些模块共可分成两层，第一层模块为Controller和Datapath。第二层模块为ALU、DM、EXT、GRF、IM、NPC、PC，都包含在Datapath中

### 关键模块定义

### 1. PC（Program Counter）

模块定义：

module PC(

input [31:0] nextPC,

input clk,

input reset,

output reg [31:0] PC

);

表1 PC端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | nextPC | I | 32 | 下一条指令的地址 |
| 2 | clk | I | 1 | 时钟信号 |
| 3 | reset | I | 1 | 同步复位 |
| 4 | PC | O | 32 | 当前指令地址 |

内部逻辑说明：

用一个32位寄存器存储当前指令地址，当时钟上升沿到来时，如果同步复位信号有效，则将寄存器复位为起始地址0x00003000，否则将新的指令地址写入寄存器。

### 2. NPC（Next PC）

模块定义：

module NPC(

input [31:0] PC,

input [31:0] offset,

input [25:0] index,

input [31:0] register,

input [2:0] ctrl,

output [31:0] PCAdd4,

output [31:0] nextPC

);

表2 NPC端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | PC | I | 32 | 当前指令地址 |
| 2 | offset | I | 32 | branch跳转的偏移量 |
| 3 | index | I | 26 | j或jal跳转地址的2-27位 |
| 4 | register | I | 32 | jr或jalr指令的跳转地址 |
| 5 | ctrl | I | 3 | 选择下条指令的地址 |
| 6 | PCAdd4 | O | 32 | 当前指令地址加4 |
| 7 | nextPC | O | 32 | 下一条指令地址 |

内部逻辑说明：

分别计算出PC+4、branch跳转地址、j或jal跳转地址、jr或jalr跳转地址，然后根据Ctrl信号选择要输出的下条指令地址nextPC。PCAdd4用于jal和jalr指令将PC+4的值存入相应寄存器。

### 3. IM（Instruction Memory）

模块定义：

module IM(

input [9:0] A,

output [31:0] RD

);

表3 IM端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | A | I | 10 | 所取指令的地址 |
| 2 | RD | O | 32 | 32位指令 |

内部逻辑说明：

IM存储所有指令。根据地址取出相应指令。

### 4. GRF（General Register File）

模块定义：

module GRF(

input [4:0] A1,

input [4:0] A2,

input [4:0] A3,

input [31:0] WD,

input WE,

input clk,

input reset,

input [31:0] PC,

output [31:0] RD1,

output [31:0] RD2

);

表4 GRF端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | A1 | I | 5 | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD1 |
| 2 | A2 | I | 5 | 5位地址输入信号，指定32个寄存器中的一个，将其中存储的数据读出到RD2 |
| 3 | A3 | I | 5 | 5位地址输入信号，指定32个寄存器中的一个作为写入的目标寄存器 |
| 4 | WD | I | 32 | 32位数据输入信号 |
| 5 | WE | I | 1 | 写使能信号 |
| 6 | clk | I | 1 | 时钟信号 |
| 7 | reset | I | 1 | 同步复位 |
| 8 | PC | I | 32 | 当前指令地址 |
| 9 | RD1 | O | 32 | 输出A1指定的寄存器中的32位数据 |
| 10 | RD2 | O | 32 | 输出A2指定的寄存器中的32位数据 |

内部逻辑说明：

通过多路分解器进行输入数据WD和使能信号WE的分配，A3为选择信号；通过多路选择器进行输出数据RD1和RD2的选择，A1和A2分别为选择信号。当时钟上升沿到来时，如果同步复位信号有效，则将所有寄存器的值清零，否则将WD写入A3对应的寄存器。PC信号仅用于格式化输出。

### 5. ALU（Arithmetic Logical Unit）

模块定义：

module ALU(

input [31:0] srcA,

input [31:0] srcB,

input [4:0] shamt,

input [4:0] ctrl,

output [31:0] out,

output flag

);

表5 ALU端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | srcA | I | 32 | 操作数A |
| 2 | srcB | I | 32 | 操作数B |
| 3 | shamt | I | 5 | 移位操作数 |
| 4 | ctrl | I | 5 | 控制信号 |
| 5 | out | O | 32 | 计算结果 |
| 6 | flag | O | 1 | 逻辑运算的结果 |

内部逻辑说明：

支持加、减、与、或、异或、移位、比较等运算。分开进行每种计算，最后通过多路选择器和ctrl信号来选择输出结果。flag信号为out的最低位，表示是否满足跳转条件，传入控制器用于控制信号的生成。

### 6. DM（Data Memory）

模块定义：

module DM(

input [31:0] A,

input [1:0] byteSel,

input [2:0] ctrl,

input [31:0] WD,

input WE,

input clk,

input reset,

input [31:0] PC,

output [31:0] RD

);

表6 DM端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | A | I | 32 | 32位地址输入信号 |
| 2 | ctrl | I | 3 | 控制信号 |
| 3 | WD | I | 32 | 32位数据输入信号 |
| 4 | WE | I | 1 | 写使能信号 |
| 5 | clk | I | 1 | 时钟信号 |
| 6 | reset | I | 1 | 同步复位 |
| 7 | PC | I | 32 | 当前指令地址 |
| 8 | RD | O | 32 | 32位数据输出信号 |

内部逻辑说明：

用一常量ADDRBITS记录地址位数，根据地址位数确定内存的大小，并从32位地址信号A中取出 [ADDRBITS-1:2] 位作为字地址，A的[1:0]位用于字节选择。当时钟上升沿到来时，如果同步复位信号有效，则清空内存，否则进行数据写入。根据控制信号可对字、半字、字节进行读写。

### 7. EXT（Extender）

模块定义：

module EXT(

input [15:0] imm16,

input [2:0] ctrl,

output [31:0] imm32

);

表7 EXT端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | imm16 | I | 16 | 16位立即数输入信号 |
| 2 | ctrl | I | 3 | 控制信号 |
| 3 | imm32 | O | 32 | 32位立即数输出信号 |

内部逻辑说明：

根据控制信号，可将16位立即数零扩展、符号扩展、加载到高位、左移两位并进行符号扩展，最后输出32位立即数。

### 8.Datapath

模块定义：

module Datapath(

    input regWrite,

    input [1:0] regDst,

    input [1:0] regSrc,

    input memWrite,

    input ALUSrc,

    input [4:0] ALUCtrl,

    input [2:0] EXTCtrl,

    input [2:0] NPCCtrl,

    input [2:0] DMCtrl,

    input clk,

    input reset,

    output [5:0] op,

    output [5:0] funct,

    output [4:0] rt,

    output ALUFlag

    );

表8 Datapath端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | regWrite | I | 2 | GRF写使能信号 |
| 2 | regDst | I | 2 | GRF写入地址选择 |
| 3 | regSrc | I | 2 | GRF写入数据选择 |
| 4 | memWrite | I | 1 | DM写使能信号 |
| 5 | ALUSrc | I | 1 | ALU操作数选择 |
| 6 | ALUCtrl | I | 5 | ALU控制信号 |
| 7 | EXTCtrl | I | 3 | EXT控制信号 |
| 8 | NPCCtrl | I | 3 | NPC控制信号 |
| 9 | DMCtrl | I | 3 | DM控制信号 |
| 10 | clk | I | 1 | 时钟信号 |
| 11 | reset | I | 1 | 同步复位 |
| 12 | op | O | 6 | 指令中op字段 |
| 13 | funct | O | 6 | 指令中funct字段 |
| 14 | rt | O | 5 | 指令中rt字段 |
| 15 | ALUFlag | O | 1 | ALU逻辑运算结果 |

内部逻辑说明：

数据通路，连接了各个子模块。输入信号中除了时钟和复位信号，其余都为控制器输出的控制信号，而输出信号则都传入控制器，用于指令的识别和控制信号的生成。

### 9.Controller

模块定义：

module Controller(

input [5:0] op,

input [5:0] funct,

input [4:0] rt,

input ALUFlag,

output regWrite,

output [1:0] regDst,

output [1:0] regSrc,

output memWrite,

output ALUSrc,

output [4:0] ALUCtrl,

output [2:0] EXTCtrl,

output [2:0] NPCCtrl,

output [2:0] DMCtrl

);

表9 Controller端口定义

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 信号名 | 方向 | 位数 | 描述 |
| 1 | op | I | 6 | 指令中op字段 |
| 2 | funct | I | 6 | 指令中funct字段 |
| 3 | rt | I | 5 | 指令中rt字段 |
| 4 | ALUFlag | I | 1 | ALU逻辑运算结果 |
| 5 | regWrite | O | 2 | GRF写使能信号 |
| 6 | regDst | O | 2 | GRF写入地址选择 |
| 7 | regSrc | O | 2 | GRF写入数据选择 |
| 8 | memWrite | O | 1 | DM写使能信号 |
| 9 | ALUSrc | O | 1 | ALU操作数选择 |
| 10 | ALUCtrl | O | 5 | ALU控制信号 |
| 11 | EXTCtrl | O | 3 | EXT控制信号 |
| 12 | NPCCtrl | O | 3 | NPC控制信号 |
| 13 | DMCtrl | O | 3 | DM控制信号 |

内部逻辑说明：

先根据op、funct、rt字段进行指令的识别，再通过或逻辑实现从指令到控制信号的映射。为了简化识别电路，可以先将R指令识别出来，之后对R指令再单独进行识别。

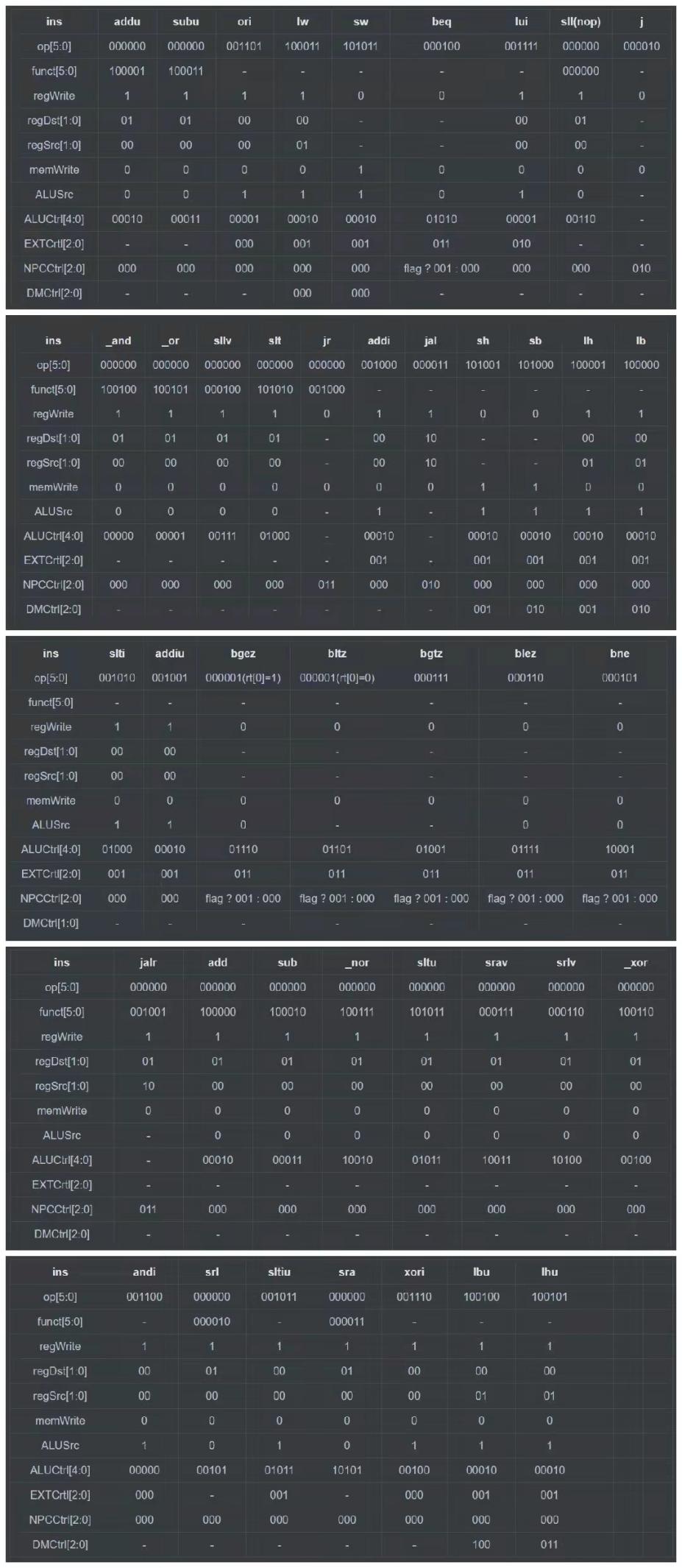


图1 指令与控制信号对照表

### （三）重要机制实现方法

#### 1. 跳转

NPC模块、EXT模块、ALU模块协同工作支持指令beq的跳转机制。

NPC模块内置了判定单元和计算单元来独立支持指令j、jal、jr 的跳转机制。

#### 2. 半字、字节存取

通过控制信号DMCtrl来判断是对字、半字还是字节进行操作。对于写指令，直接根据地址信号，找到相应位置的字、半字或字节进行写入。对于读指令，先根据地址取出相应位置的字、半字或字节，再根据控制信号进行零扩展或符号扩展后输出。

#### 3. 控制信号

通过异或与缩减运算（判等）来识别指令，对于R型指令，先通过op识别出该类型是否为R型指令，再通过funct具体识别是哪一条R型指令。对于begz或bltz指令，则要通过op和rt识别。通过或逻辑产生控制信号，对于多位的控制信号，分别对每一位用一个或运算控制。

## 二、测试方案

### （一）典型测试样例

#### 1. 计算指令测试

ori $a0, $0, 0

ori $a1, $0, 1

ori $a2, $0, 65535

ori $a3, $a0, 123

lui $a0, 65535

lui $a1, 0

lui $a2, 2

lui $a3, 456

addu $t0, $a0, $a1

subu $t1, $a1, $a2

sll $t2, $a3, 2

sllv $t3, $a3, $a1

slt $t4, $a0, $a1

#### 2. 跳转指令测试

ori $t0, $0, 0

ori $s0, $0, 5

for\_begin:

beq $t0, $s0, for\_end

jal func

addi $t0, $t0, 1

j for\_begin

for\_end:

beq $0, $0, end

func:

jr $ra

end:

#### 3. 存取指令测试

ori $a0, 1

ori $a1, 2

ori $a2, 3

ori $a3, 4

sw $a0, 0($0)

sw $a1, 0($a3)

sh $a2, 8($0)

sh $a3, 10($0)

sb $a0, 12($0)

sb $a1, 13($0)

sb $a2, 14($0)

sb $a3, 15($0)

lw $t0, 0($0)

lw $t1, 0($a3)

lh $t2, 8($0)

lh $t3, 10($0)

lb $t4, 12($0)

lb $t5, 13($0)

lb $t8, 14($0)

lb $t7, 15($0)

### （二）自动测试工具

#### 1. 测试样例生成器

代码来源于讨论区

#### mips_gnrt

图2 测试样例生成器部分代码

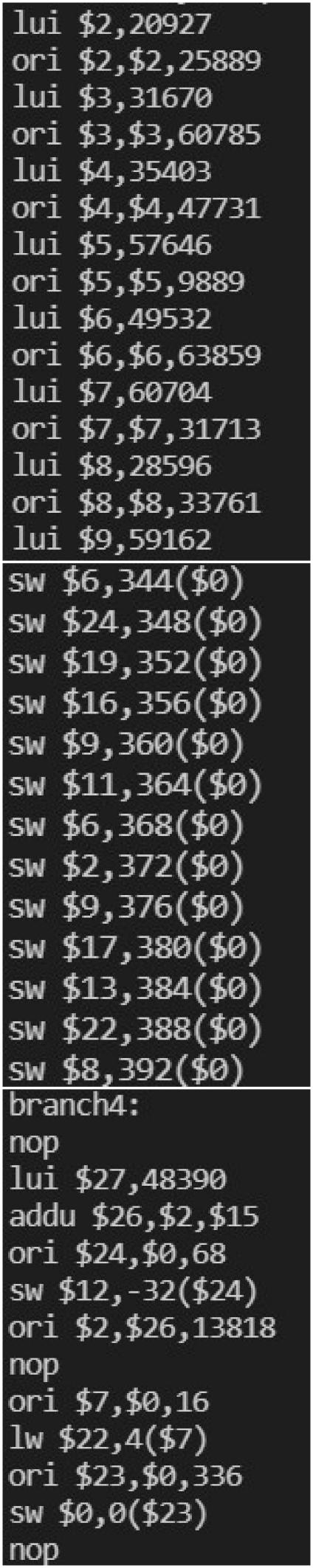


图3 生成的测试样例（部分）

#### 自动执行脚本

运行环境：win10 64位 python 3.9.6

步骤1：爆改mars，加入格式化输出，并把$gp和$sp的初始值改为0。

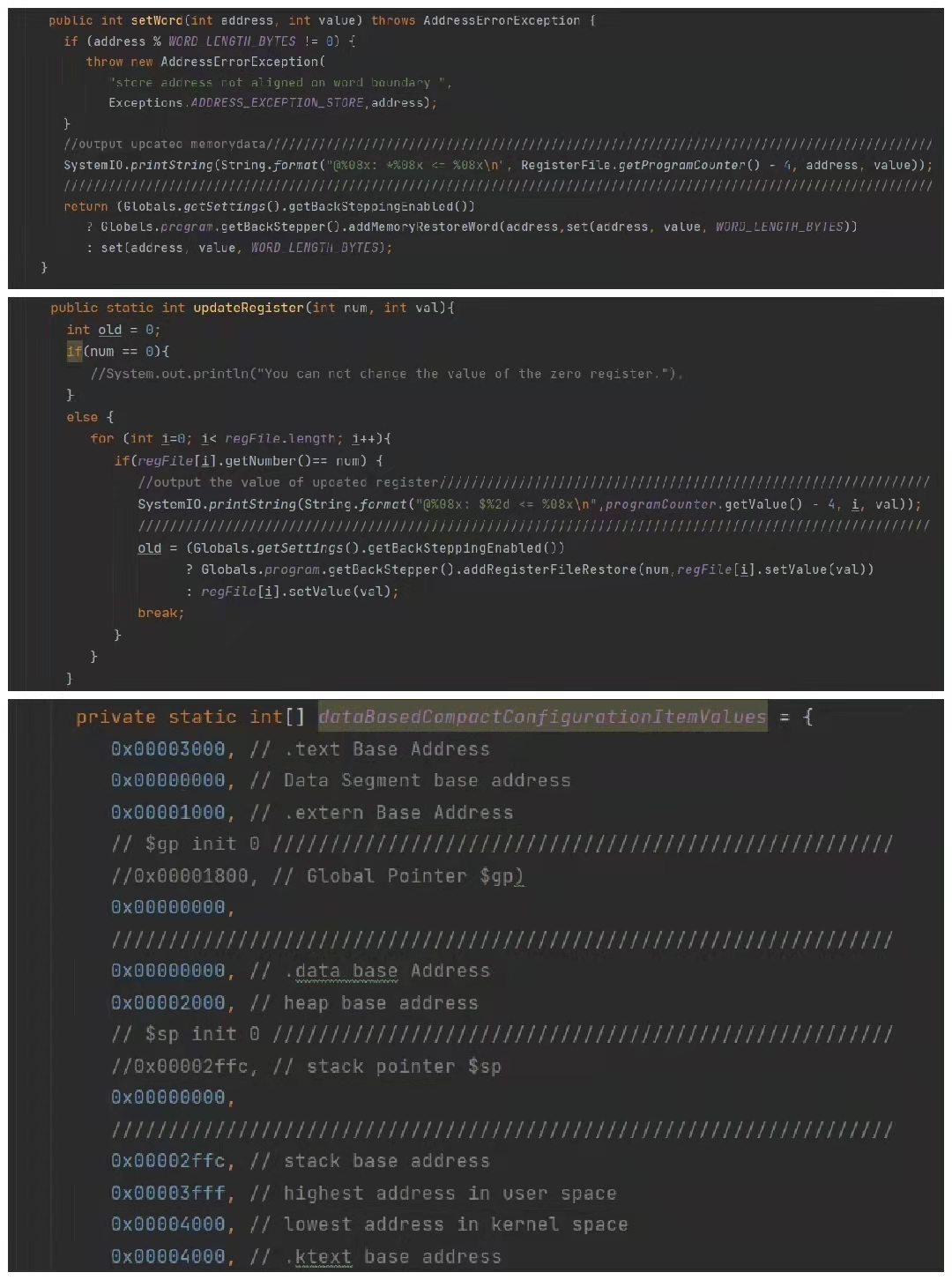
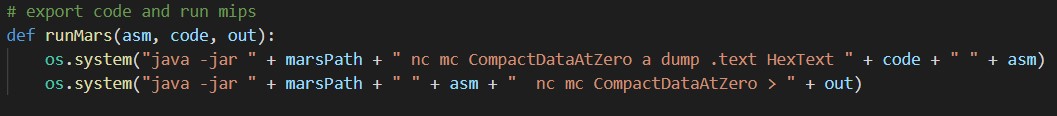


图4 修改的mars代码

步骤2：运行mars，将结果输出到文件，并导出指令

图5 运行mars和导出指令的脚本

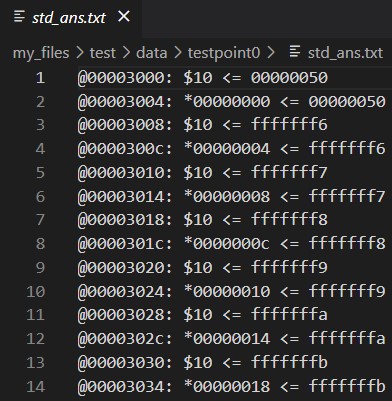
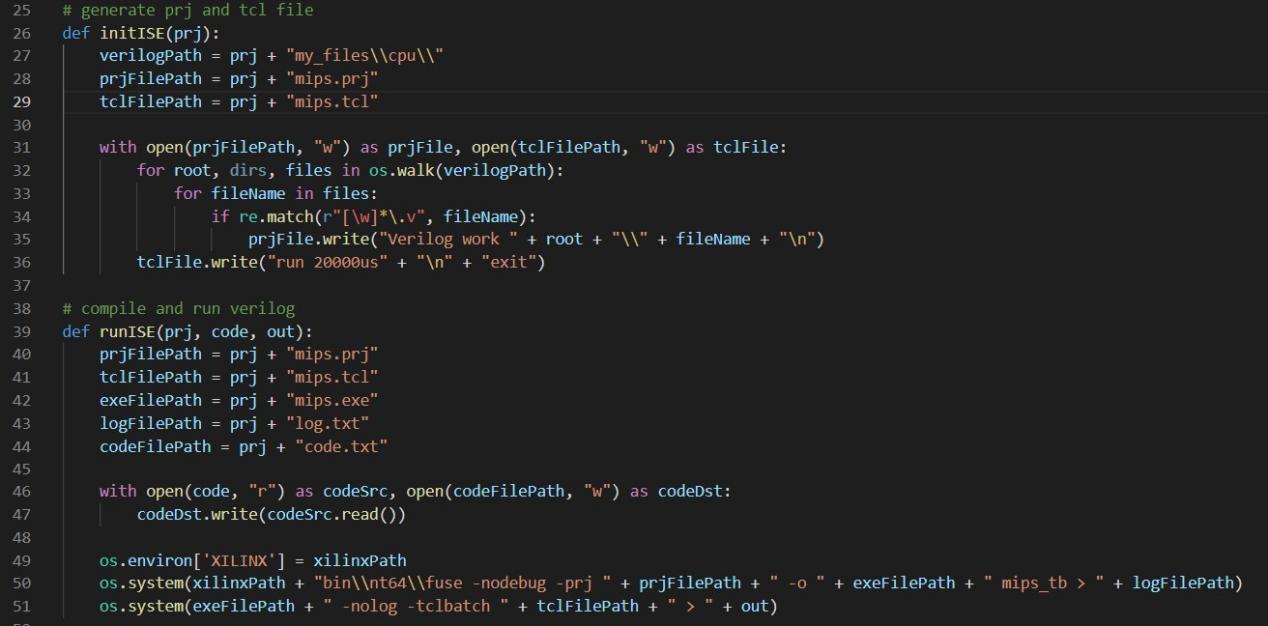
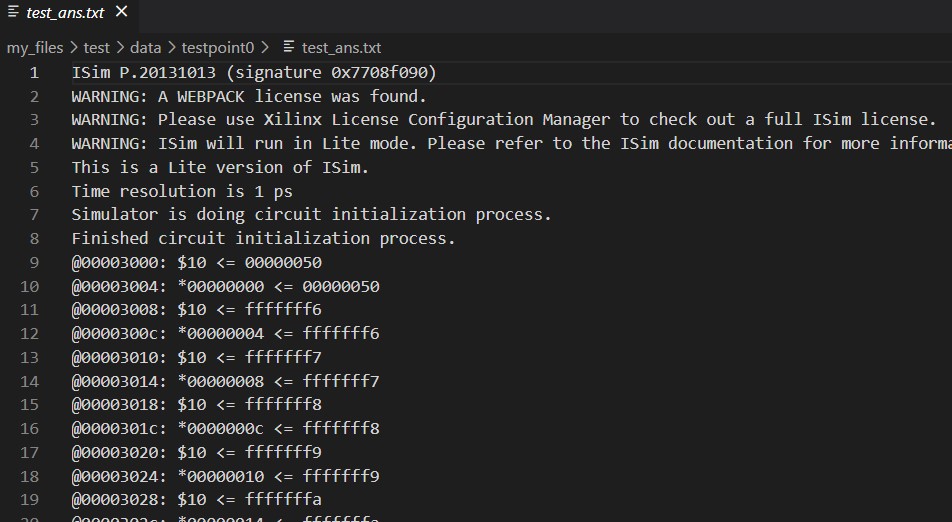


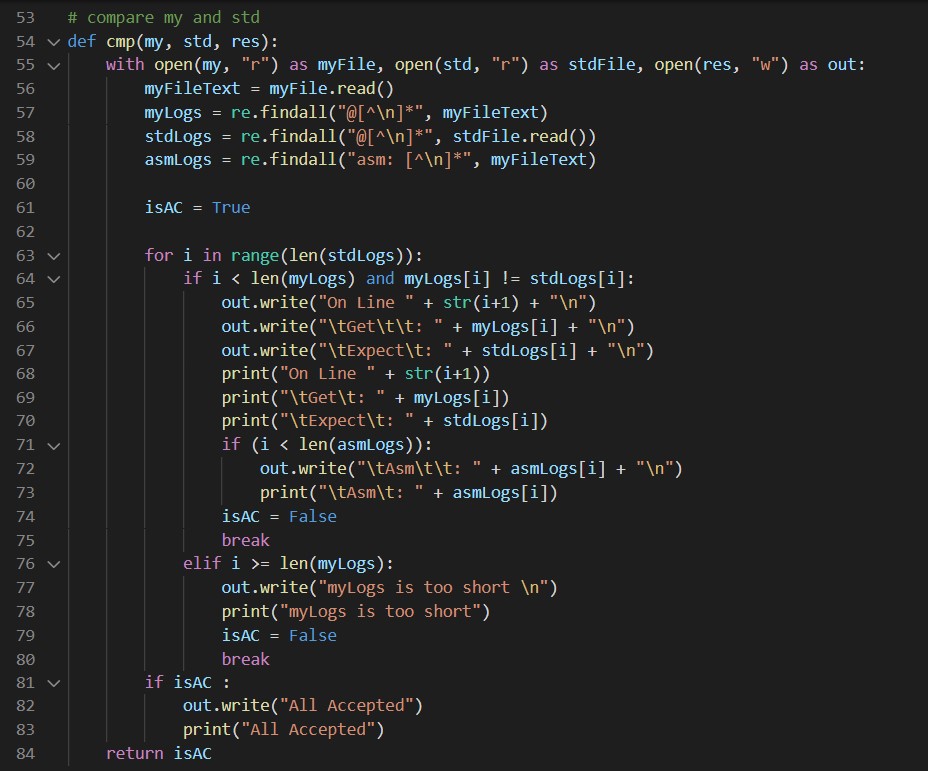
图6 mars的运行结果

步骤3：生成prj和tcl文件，编译verilog文件，进行仿真并将结果输出到文件

图7 ISE运行脚本

图8 ISE的运行结果

步骤4：将mars和ISE的运行结果进行文本比对，如果出错则给出错误信息

图9 文本比对程序

## 三、思考题

### （一）根据你的理解，在下面给出的DM的输入示例中，地址信号addr位数为什么是[11:2]而不是[9:0]？这个addr信号又是从哪里来的？question1

因为对内存的读写通常是以字为单位的（不考虑lh,sh等），而对字进行操作时地址必须是4的倍数，所以可以忽视32位字节地址的最低两位（不考虑地址检查），取 [11:2] 这10位字地址就行。

addr的信号来源是ALU的运算结果的 [11:2] 位。

### （二）思考Verilog语言设计控制器的译码方式，给出代码示例，并尝试对比各方式的优劣。

法1：每种控制信号匹配所有相应的指令。

代码示例：

assign R = ~|(op ^ 6'b000000);

assign ori = ~|(op ^ 6'b001101);

assign lw = ~|(op ^ 6'b100011);

assign addu = R & ~|(funct ^ 6'b100001);

...

assign regWrite = |{ addu, subu, ori, lw, lui };

assign memWrite = |{ sw, sh, sb };

...

优势：在对一种控制信号进行检查时更容易。

劣势：在对一条指令进行检查时更困难。

法2：每条指令匹配所有相应的控制信号。

代码示例：

`define R 6'b000000

`define addu 6'b100001

...

always @(\*)

if (op == `R && funct == `addu) begin

regWrite = 1'b1;

regDst = 2'b01;

ALUCtrl = 5'b00010;

end else if (...)

...

优势：在对一条指令进行检查时更容易。

劣势：在对一种控制信号进行检查时更困难。

### （三）在相应的部件中，reset的优先级比其他控制信号（不包括clk信号）都要高，且相应的设计都是同步复位。清零信号reset所驱动的部件具有什么共同特点？

① 内部都含有寄存器部件

② 具有记忆功能（时序逻辑电路），存储CPU当前的状态

③ 在设定上都是可读写的、在时钟上升沿到来时进行数据的写入

④ 都有固定的初始值，复位时会写入初始值

### （四）C语言是一种弱类型程序设计语言。C语言中不对计算结果溢出进行处理，这意味着C语言要求程序员必须很清楚计算结果是否会导致溢出。因此，如果仅仅支持C语言，MIPS指令的所有计算指令均可以忽略溢出。 请说明为什么在忽略溢出的前提下，addi与addiu是等价的，add与addu是等价的。提示：阅读《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中相关指令的Operation部分。

以add和addu为例（addi和addiu同理）。在遇到溢出时，add会发出整数溢出（IntegerOverflow）的异常信号，不会将运算结果写入寄存器中；而addu则不受溢出限制，在做加法后直接丢弃超过32位的部分，取低32位作为结果写入目的寄存器。在忽略溢出的情况下，add和addu一样也能计算出结果并将低32位写入目的寄存器，所以两者等价。

### 根据自己的设计说明单周期处理器的优缺点。

优点：设计简单，不需要额外寄存器来存储中间状态，控制信号的变化也更简单，相比流水线也不需要考虑冒险、延迟槽等。

缺点：第一，它需要足够长的周期来完成最慢的指令（lw），即使大部分指令的速度都非常快。第二，它需要多个加法器，而加法器是相对占用芯片面积的电路，尤其是如果它们的速度比较快。第三，它采用独立的指令存储器和数据存储器，而这在实际的系统中是不现实的。大多数计算机有一个单独的大容量存储器来存储指令和数据，并且支持读和写操作。