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

（二）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。

ins	addu	subu	ori	lw	sw	beq	lui	sll(nop)	j
op[5:0]	000000	000000	001101	100011	101011	000100	001111	000000	000010
funct[5:0]	100001	100011	-	-	-	-	-	000000	-
regWrite	1	1	1	1	0	0	1	1	0
regDst[1:0]	01	01	00	00	-	-	00	01	-
regSrc[1:0]	00	00	00	01	-	-	00	00	-
memWrite	0	0	0	0	1	0	0	0	0
ALUSrc	0	0	1	1	1	0	1	0	-
ALUCtrl[4:0]	00010	00011	00001	00010	00010	01010	00001	00110	-
EXTCtrl[2:0]	-	-	000	001	001	011	010	-	-
NPCCtrl[2:0]	000	000	000	000	000	flag ? 001 : 000	000	000	010
DMCtrl[2:0]	-	-	-	000	000	-	-	-	-

ins	_and	_or	sllv	slt	jr	addi	jal	sh	sb	lh	lb
op[5:0]	000000	000000	000000	000000	000000	001000	000011	101001	101000	100001	100000
funct[5:0]	100100	100101	000100	101010	001000	-	-	-	-	-	-
regWrite	1	1	1	1	0	1	1	0	0	1	1
regDst[1:0]	01	01	01	01	-	00	10	-	-	00	00
regSrc[1:0]	00	00	00	00	-	00	10	-	-	01	01
memWrite	0	0	0	0	0	0	0	1	1	0	0
ALUSrc	0	0	0	0	-	1	-	1	1	1	1
ALUCtrl[4:0]	00000	00001	00111	01000	-	00010	-	00010	00010	00010	00010
EXTCtrl[2:0]	-	-	-	-	-	001	-	001	001	001	001
NPCCtrl[2:0]	000	000	000	000	011	000	010	000	000	000	000
DMCtrl[2:0]	-	-	-	-	-	-	-	001	010	001	010

ins	slti	addiu	bgez	bltz	bgtz	blez	bne
op[5:0]	001010	001001	000001(rt[0]=1)	000001(rt[0]=0)	000111	000110	000101
funct[5:0]	-	-	-	-	-	-	-
regWrite	1	1	0	0	0	0	0
regDst[1:0]	00	00	-	-	-	-	-
regSrc[1:0]	00	00	-	-	-	-	-
memWrite	0	0	0	0	0	0	0
ALUSrc	1	1	0	-	-	0	0
ALUCtrl[4:0]	01000	00010	01110	01101	01001	01111	10001
EXTCtrl[2:0]	001	001	011	011	011	011	011
NPCCtrl[2:0]	000	000	flag ? 001 : 000	flag ? 001 : 000	flag ? 001 : 000	flag ? 001 : 000	flag ? 001 : 000
DMCtrl[1:0]	-	-	-	-	-	-	-

ins	jalr	add	sub	_nor	sltu	srav	srlv	_xor
op[5:0]	000000	000000	000000	000000	000000	000000	000000	000000
funct[5:0]	001001	100000	100010	100111	101011	000111	000110	100110
regWrite	1	1	1	1	1	1	1	1
regDst[1:0]	01	01	01	01	01	01	01	01
regSrc[1:0]	10	00	00	00	00	00	00	00
memWrite	0	0	0	0	0	0	0	0
ALUSrc	-	0	0	0	0	0	0	0
ALUCtrl[4:0]	-	00010	00011	10010	01011	10011	10100	00100
EXTCtrl[2:0]	-	-	-	-	-	-	-	-
NPCCtrl[2:0]	011	000	000	000	000	000	000	000
DMCtrl[2:0]	-	-	-	-	-	-	-	-

ins	andi	srl	sltiu	sra	xori	lbu	lhu
op[5:0]	001100	000000	001011	000000	001110	100100	100101
funct[5:0]	-	000010	-	000011	-	-	-
regWrite	1	1	1	1	1	1	1
regDst[1:0]	00	01	00	01	00	00	00
regSrc[1:0]	00	00	00	00	00	01	01
memWrite	0	0	0	0	0	0	0
ALUSrc	1	0	1	0	1	1	1
ALUCtrl[4:0]	00000	00101	01011	10101	00100	00010	00010
EXTCtrl[2:0]	000	-	001	-	000	001	001
NPCCtrl[2:0]	000	000	000	000	000	000	000
DMCtrl[2:0]	-	-	-	-	-	100	011

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

```
35  void print_instr(int x)
36  {
37      int y;
38      if (data1[x].type == 1)
39          printf("addu $%d,$%d,$%d\n", data1[x].r1, data1[x].r2, data1[x].r3);
40      else if (data1[x].type == 2)
41          printf("subu $%d,$%d,$%d\n", data1[x].r1, data1[x].r2, data1[x].r3);
42      else if (data1[x].type == 3)
43          printf("ori $%d,$%d,%d\n", data1[x].r1, data1[x].r2, data1[x].r3);
44      else if (data1[x].type == 4)
45          printf("lui $%d,%d\n", data1[x].r1, data1[x].r2);
46      else if (data1[x].type == 5)
47          printf("nop\n");
48      else if (data1[x].type == 6)
49      {
50          y = data1[x].aux;
51          printf("ori $%d,$0,%d\n", helper[y].r1, helper[y].r2);
52          printf("lw $%d,%d($%d)\n", data1[x].r1, data1[x].r3, data1[x].r2);
53      }
54      else if (data1[x].type == 7)
55      {
56          y = data1[x].aux;
57          printf("ori $%d,$0,%d\n", helper[y].r1, helper[y].r2);
58          printf("sw $%d,%d($%d)\n", data1[x].r1, data1[x].r3, data1[x].r2);
59      }
60  }
```

图 2 测试样例生成器部分代码

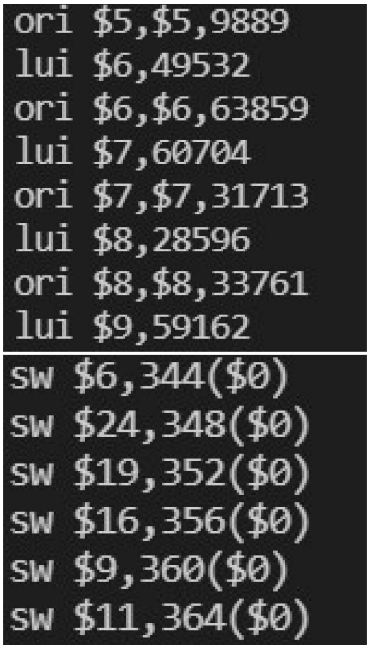
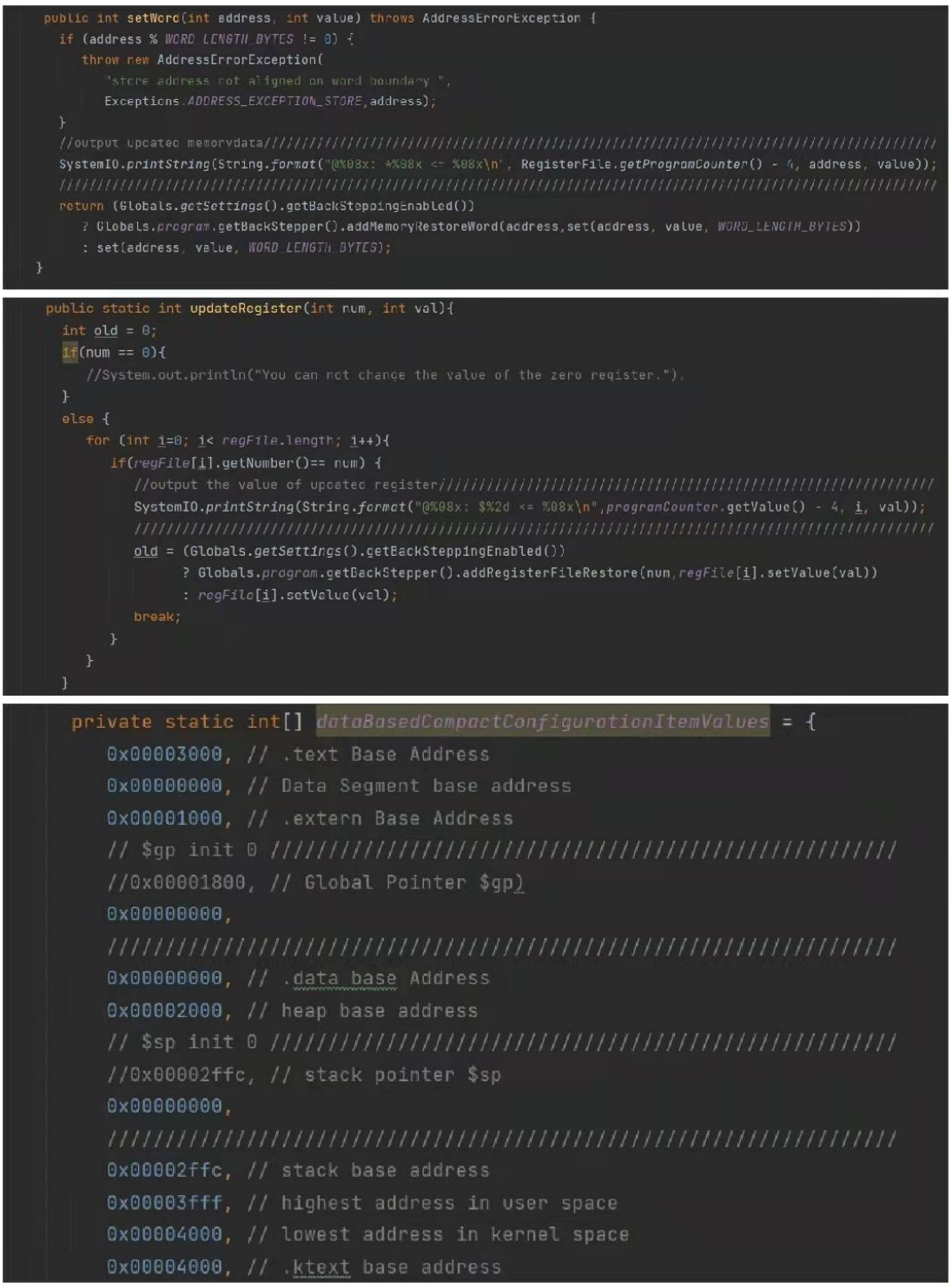


图 3 生成的测试样例（部分）

2. 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。



```
public int setWord(int address, int value) throws AddressErrorException {
  if (address % WORD_LENGTH_BYTES != 0) {
     throw new AddressErrorException(
        "store address not aligned on word boundary ",
        Exceptions.ADDRESS_EXCEPTION_STORE,address);
  }
  //output updated memorydata//////////////////////////////////////////////////////////////////////////////////
  SystemIO.printString(String.format("@%08x: *%08x <= %08x\n", RegisterFile.getProgramCounter() - 4, address, value));
  ////////////////////////////////////////////////////////////////////////////////////////////////////////////
  return (Globals.getSettings().getBackSteppingEnabled())
     ? Globals.program.getBackStepper().addMemoryRestoreWord(address,set(address, value, WORD_LENGTH_BYTES))
     : set(address, value, WORD_LENGTH_BYTES);
}
```

```
public static int updateRegister(int num, int val){
  int old = 0;
  if(num == 0){
     //System.out.println("You can not change the value of the zero register.");
  }
  else {
     for (int i=0; i< regFile.length; i++){
        if(regFile[i].getNumber()== num) {
           //output the value of updated register//////////////////////////////////////////////////////////
           SystemIO.printString(String.format("@%08x: $%2d <= %08x\n",programCounter.getValue() - 4, i, val));
           ////////////////////////////////////////////////////////////////////////////////////////////////
           old = (Globals.getSettings().getBackSteppingEnabled())
                 ? Globals.program.getBackStepper().addRegisterFileRestore(num,regFile[i].setValue(val))
                 : regFile[i].setValue(val);
           break;
        }
     }
  }
```

```
private static int[] dataBasedCompactConfigurationItemValues = {
    0x00003000, // .text Base Address
    0x00000000, // Data Segment base address
    0x00001000, // .extern Base Address
    // $gp init 0 ////////////////////////////////////////////////////
    //0x00001800, // Global Pointer $gp)
    0x00000000,
    ////////////////////////////////////////////////////////////////////
    0x00000000, // .data base Address
    0x00002000, // heap base address
    // $sp init 0 ////////////////////////////////////////////////////
    //0x00002ffc, // stack pointer $sp
    0x00000000,
    ////////////////////////////////////////////////////////////////////
    0x00002ffc, // stack base address
    0x00003fff, // highest address in user space
    0x00004000, // lowest address in kernel space
    0x00004000, // .ktext base address
```

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

```
# export code and run mips
def runMars(asm, code, out):
    os.system("java -jar " + marsPath + " nc mc CompactDataAtZero a dump .text HexText " + code + " " + asm)
    os.system("java -jar " + marsPath + " " + asm + "  nc mc CompactDataAtZero > " + out)
```

图 5 运行 mars 和导出指令的脚本

std_ans.txt

my_files > test > data > testpoint0 > std_ans.txt

```
@00003000: $10 <= 00000050
@00003004: *00000000 <= 00000050
@00003008: $10 <= fffffff6
@0000300c: *00000004 <= fffffff6
@00003010: $10 <= fffffff7
@00003014: *00000008 <= fffffff7
@00003018: $10 <= fffffff8
@0000301c: *0000000c <= fffffff8
@00003020: $10 <= fffffff9
@00003024: *00000010 <= fffffff9
@00003028: $10 <= fffffffa
@0000302c: *00000014 <= fffffffa
@00003030: $10 <= fffffffb
@00003034: *00000018 <= fffffffb
```

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

```python
# generate prj and tcl file
def initISE(prj):
    verilogPath = prj + "my_files\\cpu\\"
    prjFilePath = prj + "mips.prj"
    tclFilePath = prj + "mips.tcl"

    with open(prjFilePath, "w") as prjFile, open(tclFilePath, "w") as tclFile:
        for root, dirs, files in os.walk(verilogPath):
            for fileName in files:
                if re.match(r"[\w]*\.v", fileName):
                    prjFile.write("Verilog work " + root + "\\" + fileName + "\n")
        tclFile.write("run 20000us" + "\n" + "exit")

# compile and run verilog
def runISE(prj, code, out):
    prjFilePath = prj + "mips.prj"
    tclFilePath = prj + "mips.tcl"
    exeFilePath = prj + "mips.exe"
    logFilePath = prj + "log.txt"
    codeFilePath = prj + "code.txt"

    with open(code, "r") as codeSrc, open(codeFilePath, "w") as codeDst:
        codeDst.write(codeSrc.read())

    os.environ['XILINX'] = xilinxPath
    os.system(xilinxPath + "bin\\nt64\\fuse -nodebug -prj " + prjFilePath + " -o " + exeFilePath + " mips_tb > " + logFilePath)
    os.system(exeFilePath + " -nolog -tclbatch " + tclFilePath + " > " + out)
```

图 7 ISE 运行脚本

test_ans.txt

my_files > test > data > testpoint0 > test_ans.txt

```
ISim P.20131013 (signature 0x7708f090)
WARNING: A WEBPACK license was found.
WARNING: Please use Xilinx License Configuration Manager to check out a full ISim license.
WARNING: ISim will run in Lite mode. Please refer to the ISim documentation for more informa
This is a Lite version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
@00003000: $10 <= 00000050
@00003004: *00000000 <= 00000050
@00003008: $10 <= fffffff6
@0000300c: *00000004 <= fffffff6
@00003010: $10 <= fffffff7
@00003014: *00000008 <= fffffff7
@00003018: $10 <= fffffff8
@0000301c: *0000000c <= fffffff8
@00003020: $10 <= fffffff9
@00003024: *00000010 <= fffffff9
@00003028: $10 <= fffffffa
```

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

```python
# compare my and std
def cmp(my, std, res):
    with open(my, "r") as myFile, open(std, "r") as stdFile, open(res, "w") as out:
        myFileText = myFile.read()
        myLogs = re.findall("@[^\n]*", myFileText)
        stdLogs = re.findall("@[^\n]*", stdFile.read())
        asmLogs = re.findall("asm: [^\n]*", myFileText)

        isAC = True

        for i in range(len(stdLogs)):
            if i < len(myLogs) and myLogs[i] != stdLogs[i]:
                out.write("On Line " + str(i+1) + "\n")
                out.write("\tGet\t\t: " + myLogs[i] + "\n")
                out.write("\tExpect\t: " + stdLogs[i] + "\n")
                print("On Line " + str(i+1))
                print("\tGet\t: " + myLogs[i])
                print("\tExpect\t: " + stdLogs[i])
                if (i < len(asmLogs)):
                    out.write("\tAsm\t\t: " + asmLogs[i] + "\n")
                    print("\tAsm\t: " + asmLogs[i])
                isAC = False
                break
            elif i >= len(myLogs):
                out.write("myLogs is too short \n")
                print("myLogs is too short")
                isAC = False
                break
        if isAC :
            out.write("All Accepted")
            print("All Accepted")
    return isAC
```

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？

文件	模块接口定义

dm.v	dm(clk,reset,MemWrite,addr,din,dout);
	input clk; //clock
	input reset; //reset
	input MemWrite; //memory write enable
	input [11:2] addr; //memory's address for write
	input [31:0] din; //write data
	output [31:0] dout; //read data

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。

（五）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。