计算机组成原理实验报告

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 InstructionMemory、DataMemory、CentrolProcessingUnit、ProgramCounter、RegisterFile、ControlUnit、BitExtender、ArithmeticLogicalUnit，这些模块共可分成两级，第一级模块包括 InstructionMemory、DataMemory、CentrolProcessingUnit，CentrolProcessingUnit 中包含了第二级模块 ProgramCounter、RegisterFile、ControlUnit、BitExtender、ArithmeticLogicalUnit。

（二）关键模块定义

1. InstructionMemory

模块定义：

```
module InstructionMemory(
    input [31:0] currentCommandAddr,
    output [31:0] currentCommand
);
```

序号	功能名称	数据方向	功能描述
1.	输入地址	输入	32 位，输入的地址
2.	输出指令	输出	32 位，输出的指令

2. DataMemory

模块定义：

```
module DataMemory(
    input clk,
    input reset,
    input memWriteEnabled,
    input [2:0] loadWriteMood,
    input [31:0] addrOfDataInMem,
    input [31:0] dataWriteToMem,
    input [31:0] currentCommandAddr,
    output [31:0] dataGiveToReg
);
```

序号	功能名称	数据方向	功能描述
1.	时钟信号	输入	1 位，上跳沿更新内存的值
2.	同步复位信号	输入	1 位，信号为 1 时将此模块复位
3.	写使能信号	输入	1 位，信号为 1 时数据对 DM 的写入有效
4.	LS 类型选择	输入	3 位，对 LS 按 W/H/B 等进行选择
5.	LS 内存地址	输入	32 位，写入下述数据的地址
6.	写入数据	输入	32 位，写入到上述地址的数据
7.	输出数据	输出	32 位，上述地址对应的数据

注：

1.LS 类型选择信号的高两位选择 load 和 write 针对 word(信号的高两位为 0)half(信号的高两位为 1)还是 byte(信号的高两位为 2)，LS 类型选择信号的最低位仅对 load 有效，当 load half 或 load byte 时用以选择零扩展(0)或符号扩展(1)。

2.currentCommandAddr 用于$display，与模块实际功能无关，因此未在表中列出。

3. CentrolProcessingUnit

模块定义：

```
module CentrolProcessingUnit(
    input clk,
    input reset,
    input [31:0] currentCommand,
    output [31:0] currentCommandAddr,
    input [31:0] dataGiveToReg,
    output memWriteEnabled,
    output [2:0] loadWriteMood,
    output [31:0] addrOfDataInMem,
    output [31:0] dataWriteToMem
);
```

序号	功能名称	数据方向	功能描述
1.	时钟信号	输入	1 位，上跳沿更新其中 PC 和 GRF 的值
2.	同步复位信号	输入	1 位，信号为 1 时将其中的 PC 和 GRF 复位
3.	输入指令	输入	32 位，由 IM 给出的当前指令地址的指令
4.	输出指令地址	输出	32 位，下一条输入指令在 IM 中的地址
5.	输入数据	输入	32 位，由 DM 给出的当前数据地址的数据
6.	DM 写使能信号	输出	1 位，信号为 1 时数据对 DM 的写入有效
7.	DM 的 LS 类型选择	输出	32 位，对 LS 按 W/H/B 等进行选择
8.	DM 的 LS 内存地址	输出	32 位，DM 写入下述数据的地址
9.	DM 的写入数据	输出	32 位，DM 上述地址对应的数据

4. ProgramCounter

模块定义：

```
module ProgramCounter(
    input clk,
    input reset,
    input [31:0] branchAddr,
```

```
    input [31:0] jumpAddrFromImm,
    input [31:0] jumpAddrFormReg,
    input selectOfBranchOrNext,
    input [1:0] selectOfJumpSource,
    output reg [31:0] currentCommandAddr,
    output [31:0] nextCommandAddr
);
```

序号	功能名称	数据方向	功能描述
1.	时钟信号	输入	1 位，上跳沿更新当前指令地址为下一条指令地址
2.	同步复位信号	输入	1 位，信号为 1 时将当前指令恢复到初始化时的值
3.	立即数分支地址	输入	32 位，处理 b 类指令时得到的已处理的跳转地址，配合分支选择信号实现分支
4.	立即数跳转地址	输入	32 位，处理 j 和 jal 指令时得到的未处理的跳转地址，配合跳转选择信号实现跳转
5.	寄存器跳转地址	输入	32 位，处理 jr 和 jalr 指令时得到的来自寄存器的跳转地址，配合跳转选择信号实现跳转
6.	分支选择信号	输入	1 位，用来选择下一条指令，如果是 0 就选择当前指令加四，否则选择立即数分支地址
7.	跳转选择信号	输入	2 位，用来选择下一条指令，如果是 0 就选择经过分支选择信号选择的信号，如果是 1 就选择经过处理的立即数跳转地址，如果是 2 就选择寄存器跳转地址
8.	当前指令地址	输出	32 位，当前指令的地址
9.	下一条指令地址	输出	32 位，当前指令加四，即没有分支或跳转的情况下，下一条指令应该的地址

注：

1.处理 B 类指令时得到的已处理的跳转地址是指已经对立即数进行了零扩展，左移两位和加下一条地址。

2.处理 j 和 jal 指令时得到的未处理的跳转地址是指来自当前指令 0 到 25 位的立即数。

3.经过处理的立即数跳转地址是指对 4 中立即数零扩展到 28 位，左移两位作为低 28 位，下一条地址的高 4 位作为高 4 位得到的地址。

4.若跳转选择信号非法(3)，下一条指令的地址为 0xffffffff。

5. RegisterFile

模块定义：

```
module RegisterFile(
    input clk,
    input reset,
    input regWriteEnabled,
    input writeToReg31,
    input [4:0] regReadAddr1,
    input [4:0] regReadAddr2,
    input [4:0] regWriteAddr3,
    input [31:0] dataWriteToReg,
    input [31:0] currentCommandAddr,
    output [31:0] regData1,
    output [31:0] regData2
);
```

序号	功能名称	数据方向	功能描述
1.	时钟信号	输入	1 位，上跳沿更新内存的值
2.	同步复位信号	输入	1 位，信号为 1 时将此模块复位
3.	写使能信号	输入	1 位，信号为 1 时数据对 GRF 的写入有效
4.	写 31 号寄存器	输入	1 位，信号为 1 时如果写入数据，数据写入到$31
5.	读取地址 1	输入	5 位，读取到下述输出数据 1 的数据的地址
6.	读取地址 2	输入	5 位，读取到下述输出数据 2 的数据的地址
7.	写入地址 3	输入	5 位，写入下述输入数据的地址

8.	输入数据	输入	32 位，上述写入地址 3 需要写入的数据
9.	输出数据 1	输出	32 位，上述读取地址 1 对应的数据
10.	输出数据 2	输出	32 位，上述读取地址 2 对应的数据

注：

currentCommandAddr 用于$display，与模块实际功能无关，因此未在表中列出。

6. ControlUnit

模块定义：

```
module ControlUnit(
    input [31:0] currentCommand,
    output reg memOrALUToReg,
    output reg memWriteEnabled,
    output reg branchControl,
    output reg [4:0] ALUOperation,
    output reg [1:0] ALUInputSelect,
    output reg regWriteAddrSelect,
    output reg regWriteEnabled,
    output reg [2:0] loadWriteMood,
    output reg [1:0] extendMood,
    output reg writeToReg31,
    output reg [1:0] selectOfJumpSource,
    output reg linkWhenJump
);
```

序号	功能名称	数据方向	功能描述
1.	输入指令	输入	32 位，当前指令
2.	寄存器写入选择	输出	1 位，选择将 ALU 的结果写入寄存器(0)还是从 DM 中取出数据写入寄存器(1)
3.	内存写入使能	输出	1 位，信号为 1 时数据对 DM 的写入有效

4.	分支控制信号	输出	1 位，信号为 1 时若 ALU 输出为 0 则进行分支跳转
5.	ALU 功能选择信号	输出	5 位，控制 ALU 进行不同的运算操作，详细定义见 ALU 部分
6.	ALU 输入选择信号	输出	2 位，选择 ALU 的输入，高位用以选择第一个操作数来自 GRF 第一个输出(0)还是立即数(1)，低位用以选择第二个操作数来自 GRF 第二个输出(0)还是立即数(1)
7.	寄存器写入地址选择	输出	1 位，选择 GRF 的写入地址来自指令[20:16](0)还是[15:11](1)
8.	寄存器使能信号	输出	1 位，信号为 1 时数据对 GRF 的写入有效
9.	LS 类型选择信号	输出	3 位，在 DM 对 LS 按 W/H/B 等进行选择
10.	位扩展选择信号	输出	2 位，控制对立即数的扩展方式，分为符号扩展(0)，零扩展(1)和部分位零扩展(2)
11.	31 号寄存器写入	输出	1 位，信号为 1 时如果写入数据到 GRF，数据写入到 31 号寄存器
12.	PC 跳转选择信号	输出	2 位，用来选择下一条指令，详见 PC 部分的说明
13.	PC 跳转链接信号	输出	1 位，用以选择写入 GRF 的数据，为 1 时将下一条指令地址写入 GRF 实现跳转并链接

注：

部分位零扩展指处理 sll、srl、sra 的时候将[10:6]进行零扩展。

7. BitExtender

模块定义：

```
module BitExtender(
    input [15:0] immToExtend,
    input [1:0] extendMood,
    output [31:0] immAfterExtend
```

```
);
```

序号	功能名称	数据方向	功能描述
1.	待扩展数据	输入	16 位，来自当前指令的低 16 位
2.	扩展模式选择	输入	2 位，符号扩展(0)，零扩展(1)和用于 sll、srl、sra 指令的部分位零扩展(2)
3.	扩展后数据	输出	32 位，按选择信号要求扩展后的数据

注：

若选择信号非法(3)，输出为 0xffffffff。

8. ArithmeticLogicalUnit

模块定义：

```
module ArithmeticLogicalUnit(
    input [31:0] ALUInputA,
    input [31:0] ALUInputB,
    input [4:0] ALUOperation,
    output ALUIsZero,
    output reg [31:0] ALUOutput
);
```

序号	功能名称	数据方向	功能描述
1.	操作数 1	输入	32 位，传入 ALU 的第一个操作数
2.	操作数 2	输入	32 位，传入 ALU 的第二个操作数
3.	操作类型信号	输入	5 位，ALU 操作类型选择信号
4.	结果为零信号	输出	1 位，ALU 的计算结果为 0 时此信号为 1
5.	运算结果	输出	32 位，两个操作数运算的结果

注：

1.最多支持 32 种不同的运算，现已定义了前 17 种运算，如果进行未定义的运算，得到的结果是 0xffffffff。

2.ALU 支持的运算定义表：

序号	输入信号	执行的操作
1.	0	两个操作数按位与
2.	1	两个操作数按位或
3.	2	两个操作数做加法
4.	3	两个操作数按位异或
5.	4	第一个操作数大于 0 时输出 0
6.	5	第一个操作数小于 0 时输出 0
7.	6	两个操作数做减法
8.	7	两个操作数符号比较，第一个操作数更小时输出 1
9.	8	第一个操作数大于等于 0 时输出 0
10.	9	第二个操作数逻辑左移第一个操作数位
11.	10	第二个操作数逻辑右移第一个操作数位
12.	11	第二个操作数算术右移第一个操作数位
13.	12	两个操作数按位或非
14.	13	两个操作数不相等时输出 0
15.	14	两个操作数无符号比较，第一个操作数更小时输出 1
16.	15	第一个操作数小于等于 0 时输出 0
17.	16	将第二个操作数的低 16 位作为输出的高 16 位，其余位输出 0

（三）控制模块设计

1.控制模块真值表

为便于表格呈现，将表格分为四部分，将确定 bltz、bgez 的 20-16 位填在 4-0 位的位置，并以前导/进行区分。

第一部分：

name	31	30	29	28	27	26	5	4	3	2	1	0	*MemtoReg*	*MemWrite*	*Branch*	*ALU*[4]	*ALU*[3]	*ALU*[2]
lw	1	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	1	0	0	0	0	0
sw	1	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	1	0	0	0	0

lb	1	0	0	0	0	0	X	X	X	X	X	X	1	0	0	0	0	0
lbu	1	0	0	1	0	0	X	X	X	X	X	X	1	0	0	0	0	0
sb	1	0	1	0	0	0	X	X	X	X	X	X	X	1	0	0	0	0
lh	1	0	0	0	0	1	X	X	X	X	X	X	1	0	0	0	0	0
lhu	1	0	0	1	0	1	X	X	X	X	X	X	1	0	0	0	0	0
sh	1	0	1	0	0	1	X	X	X	X	X	X	X	1	0	0	0	0
add	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
$addu$	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0
sub	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	1
$subu$	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	1
and	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
or	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0
slt	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1
nor	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	1
xor	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	0	0
sll	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
srl	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
sra	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0
$sllv$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
$srlv$	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0
$srav$	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	0
$sltu$	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0	0	1	1
beq	0	0	0	1	0	0	X	X	X	X	X	X	X	0	1	0	0	0
bne	0	0	0	1	0	1	X	X	X	X	X	X	X	0	1	0	1	1
$bgtz$	0	0	0	1	1	1	X	X	X	X	X	X	X	0	1	0	0	1
$blez$	0	0	0	1	1	0	X	X	X	X	X	X	X	0	1	0	1	1
$bltz$	0	0	0	0	0	1	$\N$	$\0$	$\0$	$\0$	$\0$	$\0$	X	0	1	0	0	1

bgez	0	0	0	0	0	1	\N	\0	\0	\0	\0	\1	X	0	1	0	1	0
addi	0	0	1	0	0	0	X	X	X	X	X	X	0	0	0	0	0	0
addiu	0	0	1	0	0	1	X	X	X	X	X	X	0	0	0	0	0	0
andi	0	0	1	1	0	0	X	X	X	X	X	X	0	0	0	0	0	0
ori	0	0	1	1	0	1	X	X	X	X	X	X	0	0	0	0	0	0
xori	0	0	1	1	1	0	X	X	X	X	X	X	0	0	0	0	0	0
lui	0	0	1	1	1	1	X	X	X	X	X	X	0	0	0	1	0	0
slti	0	0	1	0	1	0	X	X	X	X	X	X	0	0	0	0	0	1
sltiu	0	0	1	0	1	1	X	X	X	X	X	X	0	0	0	0	1	1
j	0	0	0	0	1	0	X	X	X	X	X	X	X	0	X	X	X	X
jr	0	0	0	0	0	0	0	0	1	0	0	0	X	0	X	X	X	X
jal	0	0	0	0	1	1	X	X	X	X	X	X	0	0	X	X	X	X
jalr	0	0	0	0	0	0	0	0	1	0	0	1	0	0	X	X	X	X

第二部分：

name	31	30	29	28	27	26	5	4	3	2	1	0	*ALU*[1]	*ALU*[0]	*ALUSrc*[1]	*ALUSrc*[0]	*RegDst*	*RegWrite*
lw	1	0	0	0	1	1	X	X	X	X	X	X	1	0	0	1	0	1
sw	1	0	1	0	1	1	X	X	X	X	X	X	1	0	0	1	X	0
lb	1	0	0	0	0	0	X	X	X	X	X	X	1	0	0	1	0	1
lbu	1	0	0	1	0	0	X	X	X	X	X	X	1	0	0	1	0	1
sb	1	0	1	0	0	0	X	X	X	X	X	X	1	0	0	1	X	0
lh	1	0	0	0	0	1	X	X	X	X	X	X	1	0	0	1	0	1
lhu	1	0	0	1	0	1	X	X	X	X	X	X	1	0	0	1	0	1
sh	1	0	1	0	0	1	X	X	X	X	X	X	1	0	0	1	X	0
add	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	1
addu	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	0	1	1
sub	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	1
subu	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	1	1

and	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
or	0	0	0	0	0	0	1	0	0	1	0	1	0	1	0	0	1	1
slt	0	0	0	0	0	0	1	0	1	0	1	0	1	1	0	0	1	1
nor	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	1
xor	0	0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1	1
sll	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1
srl	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	1
sra	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	1
sllv	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1
srlv	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0	1	1
srav	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1
sltu	0	0	0	0	0	0	1	0	1	0	1	1	1	0	0	0	1	1
beq	0	0	0	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	1	1	0	0	*X*	0
bne	0	0	0	1	0	1	*X*	*X*	*X*	*X*	*X*	*X*	0	1	0	0	*X*	0
bgtz	0	0	0	1	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0	0	*X*	0
blez	0	0	0	1	1	0	*X*	*X*	*X*	*X*	*X*	*X*	1	1	0	0	*X*	0
bltz	0	0	0	0	0	1	*N*	\0	\0	\0	\0	\0	0	1	0	0	*X*	0
bgez	0	0	0	0	0	1	*N*	\0	\0	\0	\0	\1	0	0	0	0	*X*	0
addi	0	0	1	0	0	0	*X*	*X*	*X*	*X*	*X*	*X*	1	0	0	1	0	1
addiu	0	0	1	0	0	1	*X*	*X*	*X*	*X*	*X*	*X*	1	0	0	1	0	1
andi	0	0	1	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0	1	0	1
ori	0	0	1	1	0	1	*X*	*X*	*X*	*X*	*X*	*X*	0	1	0	1	0	1
xori	0	0	1	1	1	0	*X*	*X*	*X*	*X*	*X*	*X*	1	1	0	1	0	1
lui	0	0	1	1	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0	1	0	1
slti	0	0	1	0	1	0	*X*	*X*	*X*	*X*	*X*	*X*	1	1	0	1	0	1
sltiu	0	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	1	0	0	1	0	1
j	0	0	0	0	1	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0

jr	0	0	0	0	0	0	0	0	1	0	0	0	*X*	*X*	*X*	*X*	*X*	0
jal	0	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	1
jalr	0	0	0	0	0	0	0	0	1	0	0	1	*X*	*X*	*X*	*X*	1	1

第三部分：

name	31	30	29	28	27	26	5	4	3	2	1	0	*HAndD*[2]	*HAndD*[1]	*HAndD*[0]	*Extend*[1]	*Extend*[0]	*JL*[1]
lw	1	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	*X*	0	0	0
sw	1	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	*X*	0	0	0
lb	1	0	0	0	0	0	*X*	*X*	*X*	*X*	*X*	*X*	1	0	1	0	0	0
lbu	1	0	0	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	1	0	0	0	0	0
sb	1	0	1	0	0	0	*X*	*X*	*X*	*X*	*X*	*X*	1	0	*X*	0	0	0
lh	1	0	0	0	0	1	*X*	*X*	*X*	*X*	*X*	*X*	0	1	1	0	0	0
lhu	1	0	0	1	0	1	*X*	*X*	*X*	*X*	*X*	*X*	0	1	0	0	0	0
sh	1	0	1	0	0	1	*X*	*X*	*X*	*X*	*X*	*X*	0	1	*X*	0	0	0
add	0	0	0	0	0	0	1	0	0	0	0	0	*X*	*X*	*X*	*X*	*X*	0
add	0	0	0	0	0	0	1	0	0	0	0	1	*X*	*X*	*X*	*X*	*X*	0
sub	0	0	0	0	0	0	1	0	0	0	1	0	*X*	*X*	*X*	*X*	*X*	0
subu	0	0	0	0	0	0	1	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	0
and	0	0	0	0	0	0	1	0	0	1	0	0	*X*	*X*	*X*	*X*	*X*	0
or	0	0	0	0	0	0	1	0	0	1	0	1	*X*	*X*	*X*	*X*	*X*	0
slt	0	0	0	0	0	0	1	0	1	0	1	0	*X*	*X*	*X*	*X*	*X*	0
nor	0	0	0	0	0	0	1	0	0	1	1	1	*X*	*X*	*X*	*X*	*X*	0
xor	0	0	0	0	0	0	1	0	0	1	1	0	*X*	*X*	*X*	*X*	*X*	0
sll	0	0	0	0	0	0	0	0	0	0	0	0	*X*	*X*	*X*	1	0	0
srl	0	0	0	0	0	0	0	0	0	0	1	0	*X*	*X*	*X*	1	0	0
sra	0	0	0	0	0	0	0	0	0	0	1	1	*X*	*X*	*X*	1	0	0
sllv	0	0	0	0	0	0	0	0	0	1	0	0	*X*	*X*	*X*	*X*	*X*	0
srlv	0	0	0	0	0	0	0	0	0	1	1	0	*X*	*X*	*X*	*X*	*X*	0

srav	0	0	0	0	0	0	0	0	0	1	1	1	*X*	*X*	*X*	*X*	*X*	0
sltu	0	0	0	0	0	0	1	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	0
beq	0	0	0	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
bne	0	0	0	1	0	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
bgtz	0	0	0	1	1	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
blez	0	0	0	1	1	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
bltz	0	0	0	0	0	1	*N*	\0	\0	\0	\0	\0	*X*	*X*	*X*	0	0	0
bgez	0	0	0	0	0	1	*N*	\0	\0	\0	\0	\1	*X*	*X*	*X*	0	0	0
addi	0	0	1	0	0	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
addiu	0	0	1	0	0	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
andi	0	0	1	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	1	0
ori	0	0	1	1	0	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	1	0
xori	0	0	1	1	1	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	1	0
lui	0	0	1	1	1	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
slti	0	0	1	0	1	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
sltiu	0	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
j	0	0	0	0	1	0	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0
jr	0	0	0	0	0	0	0	0	1	0	0	0	*X*	*X*	*X*	*X*	*X*	1
jal	0	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	*X*	0
jalr	0	0	0	0	0	0	0	0	1	0	0	1	*X*	*X*	*X*	*X*	*X*	1

第四部分：

name	31	30	29	28	27	26	5	4	3	2	1	0	*JL*[0]	*JR*	*To*31
lw	1	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
sw	1	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	*X*	*X*
lb	1	0	0	0	0	0	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
lbu	1	0	0	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
sb	1	0	1	0	0	0	*X*	*X*	*X*	*X*	*X*	*X*	0	*X*	*X*

lh	1	0	0	0	0	1	X	X	X	X	X	X	0	0	0
lhu	1	0	0	1	0	1	X	X	X	X	X	X	0	0	0
sh	1	0	1	0	0	1	X	X	X	X	X	X	0	X	X
add	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
$addu$	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
sub	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
$subu$	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0
and	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
or	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0
slt	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0
nor	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0
xor	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0
sll	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
srl	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
sra	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
$sllv$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$srlv$	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
$srav$	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
$sltu$	0	0	0	0	0	0	1	0	1	0	1	1	0	0	0
beq	0	0	0	1	0	0	X	X	X	X	X	X	0	X	X
bne	0	0	0	1	0	1	X	X	X	X	X	X	0	X	X
$bgtz$	0	0	0	1	1	1	X	X	X	X	X	X	0	X	X
$blez$	0	0	0	1	1	0	X	X	X	X	X	X	0	X	X
$bltz$	0	0	0	0	0	1	$\N$	$\0$	$\0$	$\0$	$\0$	$\0$	0	X	X
$bgez$	0	0	0	0	0	1	$\N$	$\0$	$\0$	$\0$	$\0$	$\1$	0	X	X
$addi$	0	0	1	0	0	0	X	X	X	X	X	X	0	0	0
$addiu$	0	0	1	0	0	1	X	X	X	X	X	X	0	0	0

andi	0	0	1	1	0	0	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
ori	0	0	1	1	0	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
xori	0	0	1	1	1	0	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
lui	0	0	1	1	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
slti	0	0	1	0	1	0	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
sltiu	0	0	1	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	0	0	0
j	0	0	0	0	1	0	*X*	*X*	*X*	*X*	*X*	*X*	1	*X*	*X*
jr	0	0	0	0	0	0	0	0	1	0	0	0	0	*X*	*X*
jal	0	0	0	0	1	1	*X*	*X*	*X*	*X*	*X*	*X*	1	1	1
jalr	0	0	0	0	0	0	0	0	1	0	0	1	0	1	0

2.对真值表的处理

考虑到以上指令分三类，lw、sw、lb、lbu、sb、lh、lhu、sh、beq、bne、bgtz、blez、addi、andi、ori、xori、lui、slti、sltiu、j、jal 以[31:26]进行编码，add、sub、and、or、slt、nor、xor、sll、srl、sra、sllv、srlv、srav、sltu、jr、jalr[31:26]全为 0，以[5:0]编码，bltz、bgez[31:26]仅第 26 位为 1，以[20:16]编码，因此可将指令分为上述三类分别进行处理以使控制器简洁且有条理。下给出三个模块实际有效的真值表(为上述真值表的子集，为便于辅助生成代码的程序生成用于 ControlUnit 的代码，对真值表进行了简化)。

```
31 26
lw 1 0 0 0 1 1 -> 1 0 0 0 0 0 1 0 0 1 0 1 0 0 X 0 0 0 0 0 0
sw 1 0 1 0 1 1 -> X 1 0 0 0 0 1 0 0 1 X 0 0 0 X 0 0 0 0 X X
lb 1 0 0 0 0 0 -> 1 0 0 0 0 0 1 0 0 1 0 1 1 0 1 0 0 0 0 0 0
lbu 1 0 0 1 0 0 -> 1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0
sb 1 0 1 0 0 0 -> X 1 0 0 0 0 1 0 0 1 X 0 1 0 X 0 0 0 0 X X
lh 1 0 0 0 0 1 -> 1 0 0 0 0 0 1 0 0 1 0 1 0 1 1 0 0 0 0 0 0
lhu 1 0 0 1 0 1 -> 1 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0
sh 1 0 1 0 0 1 -> X 1 0 0 0 0 1 0 0 1 X 0 0 1 X 0 0 0 0 X X
beq 0 0 0 1 0 0 -> X 0 1 0 0 0 1 1 0 0 X 0 X X X 0 0 0 0 X X
bne 0 0 0 1 0 1 -> X 0 1 0 1 1 0 1 0 0 X 0 X X X 0 0 0 0 X X
```

```
bgtz 0 0 0 1 1 1 -> X 0 1 0 0 1 0 0 0 0 X 0 X X X 0 0 0 0 X X
blez 0 0 0 1 1 0 -> X 0 1 0 1 1 1 1 0 0 X 0 X X X 0 0 0 0 X X
addi 0 0 1 0 0 0 -> 0 0 0 0 0 0 1 0 0 1 0 1 X X X 0 0 0 0 0 0
addiu 0 0 1 0 0 1 -> 0 0 0 0 0 0 1 0 0 1 0 1 X X X 0 0 0 0 0 0
andi 0 0 1 1 0 0 -> 0 0 0 0 0 0 0 0 0 1 0 1 X X X 0 1 0 0 0 0
ori 0 0 1 1 0 1 -> 0 0 0 0 0 0 0 1 0 1 0 1 X X X 0 1 0 0 0 0
xori 0 0 1 1 1 0 -> 0 0 0 0 0 0 1 1 0 1 0 1 X X X 0 1 0 0 0 0
lui 0 0 1 1 1 1 -> 0 0 0 1 0 0 0 0 0 1 0 1 X X X 0 0 0 0 0 0
slti 0 0 1 0 1 0 -> 0 0 0 0 0 1 1 1 0 1 0 1 X X X 0 0 0 0 0 0
sltiu 0 0 1 0 1 1 -> 0 0 0 0 1 1 1 0 0 1 0 1 X X X 0 0 0 0 0 0
j 0 0 0 0 1 0 -> X 0 X X X X X X X X 0 X X X X X 0 1 X X
jal 0 0 0 0 1 1 -> 0 0 X X X X X X X X 1 X X X X X 0 1 1 1

5 0
add 1 0 0 0 0 0 -> 0 0 0 0 0 0 1 0 0 0 1 1 X X X X X 0 0 0 0
addu 1 0 0 0 0 1 -> 0 0 0 0 0 0 1 0 0 0 1 1 X X X X X 0 0 0 0
sub 1 0 0 0 1 0 -> 0 0 0 0 0 1 1 0 0 0 1 1 X X X X X 0 0 0 0
subu 1 0 0 0 1 1 -> 0 0 0 0 0 1 1 0 0 0 1 1 X X X X X 0 0 0 0
and 1 0 0 1 0 0 -> 0 0 0 0 0 0 0 0 0 0 1 1 X X X X X 0 0 0 0
or 1 0 0 1 0 1 -> 0 0 0 0 0 0 0 1 0 0 1 1 X X X X X 0 0 0 0
slt 1 0 1 0 1 0 -> 0 0 0 0 0 1 1 1 0 0 1 1 X X X X X 0 0 0 0
nor 1 0 0 1 1 1 -> 0 0 0 0 0 1 1 0 0 0 0 1 1 X X X X X 0 0 0 0
xor 1 0 0 1 1 0 -> 0 0 0 0 0 0 1 1 0 0 1 1 X X X X X 0 0 0 0
sll 0 0 0 0 0 0 -> 0 0 0 0 0 1 0 0 1 1 0 1 1 X X X 1 0 0 0 0 0
srl 0 0 0 0 0 1 0 -> 0 0 0 0 0 1 0 1 0 1 0 1 1 X X X 1 0 0 0 0 0
sra 0 0 0 0 0 1 1 -> 0 0 0 0 0 1 0 1 1 1 0 1 1 X X X 1 0 0 0 0 0
sllv 0 0 0 1 0 0 -> 0 0 0 0 0 1 0 0 1 0 0 1 1 X X X X X 0 0 0 0
srlv 0 0 0 1 1 0 -> 0 0 0 0 0 1 0 1 0 0 0 1 1 X X X X X 0 0 0 0
```

```
srav 0 0 0 1 1 1 -> 0 0 0 0 1 0 1 1 0 0 1 1 X X X X X 0 0 0 0
sltu 1 0 1 0 1 1 -> 0 0 0 0 1 1 1 0 0 0 1 1 X X X X X 0 0 0 0
jr 0 0 1 0 0 0 -> X 0 X X X X X X X X 0 X X X X X 1 0 X X
jalr 0 0 1 0 0 1 -> 0 0 X X X X X X X X 1 1 X X X X X 1 0 1 0

20 16
bltz 0 0 0 0 0 0 -> X 0 1 0 0 1 0 1 0 0 X 0 X X X 0 0 0 0 X X
bgez 0 0 0 0 0 1 -> X 0 1 0 1 0 0 0 0 0 X 0 X X X 0 0 0 0 X X
```

3.辅助搭建电路的程序

我们需要将上述三个真值表转换为 ControlUnit 中的代码，手工编码是一件痛苦的事情，且容易出错，因此考虑使用 C 语言根据真值表直接生成这部分的代码，程序代码如下：

```
#include <stdio.h>
#define GETDEFINE 0

char aludef[17][20]=
{
    "`AandB",
    "`AorB",
    "`AaddB",
    "`AxorB",
    "`AgtzSet0",
    "`AltzSet0",
    "`AsubB",
    "`AsignedltB",
    "`AgezSet0",
    "`BlogicleftA",
    "`BlogicrightA",
    "`BsignedrightA",
```

```
        "`AnorB",
        "`AnotBSet0",
        "`AunsignedltB",
        "`AlezSet0",
        "`luiB"
};

bool read1(char name[],int num,int in[])
{
    char s[3]; int tmp;
    if (scanf("%s",name)==-1) return false;
    for (int i=0;i<num;i++)
        if (scanf("%d",&tmp)==-1) return false;
    getchar(),getchar(),getchar();
    for (int i=0;i<21;i++)
        if (scanf("%s",s)==-1) return false;
        else in[i]=s[0]=='1';
    return true;
}

bool read2(char name[],int num,int in[])
{
    char s[111]; int tmp,tot=0;
    if (scanf("%s",name)==-1) return false;
    for (int i=0;i<num;i++)
        if (scanf("%d",&tmp)==-1) return false;
        else in[tot++]=tmp;
    gets(s); return true;
```

```
}

int solve(int in[])
{
    int ans=0;
    for (int i=3;i<=7;i++)
        ans<<=1,ans|=in[i];
    return ans;
}

void getcode()
{
    char name[10];
    int in[21],fst=1,sx,sy;
    scanf("%d %d",&sx,&sy);
    while (read1(name,sx-sy+1,in))
    {
        if (fst) printf("if (cmd[%d:%d]==`%s) begin\n",sx,sy,name),fst--;
        else printf("\t\t\telse if (cmd[%d:%d]==`%s) begin\n",sx,sy,name);
        printf("\t\t\t\tmemOrALUToReg<=%d; memWriteEnabled<=%d;
        branchControl<=%d;\n",in[0],in[1],in[2]);
        printf("\t\t\t\tALUOperation<=%s; ALUInputSelect<=2'b%d%d;
        regWriteDataSelect<=%d;\n",aludef[solve(in)],in[8],in[9],in[10]);
        printf("\t\t\t\tregWriteEnabled<=%d; loadWriteMood<=3'b%d%d%d;
        extendMood<=3'b%d%d;\n",in[11],in[12],in[13],in[14],in[15],in[16]);
        printf("\t\t\t\tselectOfJumpSource<=3'b%d%d; linkWhenJump<=%d;
        writeToReg31<=%d;\n\t\t\tend\n",in[17],in[18],in[19],in[20]);
    }
```

```
        fputs("\t\t\telse begin end",stdout);
        return;
    }

    void getdefine()
    {
        char name[10];
        int in[21],sx,sy;
        scanf("%d %d",&sx,&sy);
        for (;read2(name,sx-sy+1,in);puts(""))
        {
            printf("\t`define %s\t%d'b",name,sx-sy+1);
            for (int i=0;i<sx-sy+1;i++) printf("%d",in[i]);
        }
        return;
    }

    int main()
    {
        freopen("out.txt","w",stdout);
        if (GETDEFINE) getdefine();
        else getcode(); return 0;
    }
```

将上面给出的真值表作为输入粘贴进本程序，即可在 out.txt 中得到 verilog 中可用的代码，重复三次运行此程序即可得到三部分的 verilog 程序，需要说明的是，程序中操作码以宏定义的形式给出，因此我们需要把 GETDEFINE 改成 1 再次运行程序，以获得宏定义。

二、测试方案

（一）典型测试样例

根据指令的类型，主要构造了九个测试点，如下：

testpoint1：主要测试四个跳转指令

包含的指令：add,addi,beq,j,jal,jalr,jr,slti,sw

testpoint2：主要测试六个条件跳转指令

包含的指令：addi,addiu,beq,bgez,bgtz,blez,bltz,bne,j,slti,sw

testpoint3：主要测试若干 R 类型的指令，有点多所以分了两个测试点

包含的指令：addi,addiu,addu,and,lw,nor,or,sllv,slt,sltu,srav,srlv,subu,sw,xor

testpoint4：主要测试若干 R 类型的指令，有点多所以分了两个测试点

包含的指令：add,addi,addiu,addu,and,lw,nor,or,sllv,slt,sltu,srav,srlv,sub,subu,sw,xor

testpoint5：主要测试八个内存读写指令

包含的指令：addi,addiu,beq,j,lb,lbu,lh,lhu,lw,sb,sh,sw

testpoint6：主要测试若干 I 类型的指令，有点多所以分了两个测试点

包含的指令：addi,addiu,andi,ori,sll,srl,sw

testpoint7：主要测试若干 I 类型的指令，有点多所以分了两个测试点

包含的指令：addi,lui,slti,sltiu,sra,sw,xori

testpoint8：实质是跑水仙花数，找个比较长的有意义的程序跑下试试

包含的指令：add,addi,beq,j,jal,jr,slt,slti,sw

注意：10ns 一周期，要 20000us

testpoint9：实质是跑 n=100 的快排，找个有递归的有意义的程序跑下试试

包含的指令：add,addi,addiu,beq,bgez,bgtz,blez,bltz,bne,j,jal,jr,lw,sll,srl,sub,sw

注意：有递归，DM 要开到 0x2ffc

注：

测试点较长，为方便阅读，文中不给出具体测试点内容，具体测试点内容已在“CPU 自动测试或辅助工具提交窗口”中打包提交。

（二）数据构造方法

覆盖所有测试功能正常进行的样例，极端情况的样例，以 R 指令的测试为例，我们对每个 R 指令，测试在两个输入分别为正正，负负，正负，负正，零零，正零，负零，正数负数分别包含如 1、-1、2147483647,、-2147483648 在内的数据和随机数，同时我们会测试 ARB、BRA、ARA、BRB，其它测试点的构造类似，具体见 MIPS 汇编程序，考虑到汇编程序较长，为方便阅读，本文档中不给出各个测试点的内容，各个测试点的 asm 代码已在“CPU 自动测试或辅助工具提交窗口”中打包提交。

（三）自动测试工具

1.通过修改 Mars 使 Mars 在仿真过程中输出符合 OJ 要求的结果，核心代码如下：

Memory.java 中

```
public int set(int address, int value, int length) throws AddressErrorException {
    int oldValue = 0;
    if (Globals.debug) System.out.println("memory["+address+"] set to
    "+value+"("+length+" bytes)");
    int relativeByteAddress;
    if (inDataSegment(address)) {
        oldValue = storeBytesInTable(dataBlockTable, relativeByteAddress, length,
        value);
    }
    else if (address > stackLimitAddress && address <= stackBaseAddress) {
        relativeByteAddress = stackBaseAddress - address;
        oldValue = storeBytesInTable(stackBlockTable, relativeByteAddress, length,
        value);
    }
    else if (inTextSegment(address)) {
        if
        (Globals.getSettings().getBooleanSetting(Settings.SELF_MODIFYING_CODE_ENABLED)
```

```
        ) {
            ProgramStatement oldStatement = getStatementNoNotify(address);
            if (oldStatement != null) {
                oldValue = oldStatement.getBinaryStatement();
            }
            setStatement(address, new ProgramStatement(value, address));
        }
        else {
            throw new AddressErrorException(
                "Cannot write directly to text segment!",
                Exceptions.ADDRESS_EXCEPTION_STORE, address);
        }
    }
    else if (address >= memoryMapBaseAddress && address < memoryMapLimitAddress) {
        relativeByteAddress = address - memoryMapBaseAddress;
        oldValue = storeBytesInTable(memoryMapBlockTable, relativeByteAddress, length,
        value);
    }
    else if (inKernelDataSegment(address)) {
        oldValue = storeBytesInTable(kernelDataBlockTable, relativeByteAddress,
        length, value);
    }
    else if (inKernelTextSegment(address)) {
        throw new AddressErrorException(
            "DEVELOPER: You must use setStatement() to write to kernel text segment!",
            Exceptions.ADDRESS_EXCEPTION_STORE, address);
    }
    else {
```

```
            throw new AddressErrorException("address out of range ",
                Exceptions.ADDRESS_EXCEPTION_STORE, address);
        }
        notifyAnyObservers(AccessNotice.WRITE, address, length, value);
        int offset = 8 - (length << 1);
        String buffer=String.format("@%08x: *%08x <= %08x\n",
            RegisterFile.getProgramCounter()-4,address,getWord((address>>2)<<2));
        SystemIO.printString(buffer);
        return oldValue;
    }
```

RegisterFile.java 中

```
public static int updateRegister(int num, int val) {
    int old = 0;
    if (num == 0) {}
    else {
        for (int i=0; i< regFile.length; i++) {
            if(regFile[i].getNumber()== num) {
                old = (Globals.getSettings().getBackSteppingEnabled())
                ?
                Globals.program.getBackStepper().addRegisterFileRestore(num,regFile[i]
                .setValue(val))
                : regFile[i].setValue(val);
                break;
            }
        }
    }
    if (num== 33) {
        old = (Globals.getSettings().getBackSteppingEnabled())
```

```
            ?
            Globals.program.getBackStepper().addRegisterFileRestore(num,hi.setValue(val))
            : hi.setValue(val);
        }
        else if(num== 34) {
            old = (Globals.getSettings().getBackSteppingEnabled())
            ?
            Globals.program.getBackStepper().addRegisterFileRestore(num,lo.setValue(val))
            : lo.setValue(val);
        }
        if (num < 32) {
            String buffer=String.format("@%08x: $%2d <= %08x\n",getProgramCounter()-
            4,num,val);
            SystemIO.printString(buffer);
        }
        return old;
    }
```

MemoryConfigurations.java 中

```
private static int[] dataBasedCompactConfigurationItemValues = {
        0x00003000, // .text Base Address
        0x00000000, // Data Segment base address
        0x00001000, // .extern Base Address
        //0x00001800, // Global Pointer $gp backup
        0x00000000, // Global Pointer $gp
        0x00000000, // .data base Address
        0x00002000, // heap base address
        //0x00002ffc, // stack pointer $sp backup
        0x00000000, // stack pointer $sp
```

```
    0x00002ffc, // stack base address
    0x00003fff, // highest address in user space
    0x00004000, // lowest address in kernel space
    0x00004000, // .ktext base address
    0x00004180, // exception handler address
    0x00005000, // .kdata base address
    0x00007f00, // MMIO base address
    0x00007fff, // highest address in kernel (and memory)
    0x00002fff, // data segment limit address
    0x00003ffc, // text limit address
    0x00007eff, // kernel data segment limit address
    0x00004ffc, // kernel text limit address
    0x00002000, // stack limit address
    0x00007fff  // memory map limit address
};
```

2.基于 C 语言的自动测试工具，给出的比较函数仅供参考，可根据需要修改比较函数：

```
#pragma GCC optimize(2)
#pragma GCC optimize(3,"Ofast","inline")

#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

const int time=20000;
const int pointnum=9;
const char isedir[105]="D:\\XilinxISE\\14.7\\ISE_DS\\ISE";
char filedir[105],testpoint[105],buffer1[505],buffer2[505];
```

```
inline void gettcl()
{
    FILE* fpr=fopen("mips.tcl","w");
    fprintf(fpr,"run %dus;\nexit\n",time);
    fclose(fpr); return;
}

void dfs(char* dir,FILE* fpr)
{
    long handle;
    char dirbuffer[105];
    _finddata_t findData;
    strcpy(dirbuffer,dir),strcat(dirbuffer,"\\*.*");
    handle=_findfirst(dirbuffer,&findData);
    if (handle==-1) return;
    do
    {
        if (findData.attrib&_A_SUBDIR)
        {
            if (!strcmp(findData.name,".") || !strcmp(findData.name,"..")) continue;
            int len=strlen(buffer1);
            sprintf(buffer1,"%s\\%s",buffer1,findData.name);
            strcpy(dirbuffer,dir),strcat(dirbuffer,"\\");
            strcat(dirbuffer,findData.name);
            dfs(dirbuffer,fpr),buffer1[len]=0;
        }
        else
```

```
        {
            int len=strlen(findData.name);
            if (len>1 && findData.name[len-1]=='v' && findData.name[len-2]=='.')
            fprintf(fpr,"verilog work \"%s\\%s\"\n",buffer1,findData.name);
        }
    }
    while (!_findnext(handle,&findData));
    _findclose(handle); return;
}

inline void getprj()
{
    FILE* fpr=fopen("mips.prj","w");
    strcpy(buffer1,filedir);
    dfs(filedir,fpr); return;
}

inline void filecompare(const char* s)
{
    int cnt=0;
    FILE* ans=fopen("ans.txt","r");
    FILE* out=fopen("out.txt","r");
    for (int i=0;i<5;i++) fgets(buffer1,512,out);
    while (fgets(buffer2,512,ans))
    {
        cnt++;
        if (strlen(buffer2)<3) break;
        if (!fgets(buffer1,512,out))
```

```
            {
                printf("test point %s wrong at line %d\n",s,cnt);
                puts("your answer is fewer than expected answer");
                return;
            }
            else if (strcmp(buffer1,buffer2))
            {
                printf("test point %s wrong at line %d\n",s,cnt);
                printf("expected answer is %s\n",buffer2);
                printf("your answer is %s\n",buffer1);
                return;
            }
        }
        printf("test point %s accepted\n",s);
        return;
    }

    inline void solve(const char* s)
    {
        printf("\ntesting %s\n",s);
        sprintf(buffer1,"java -jar mars.jar a nc mc CompactDataAtZero dump .text HexText
code.txt %s",s),system(buffer1);
        sprintf(buffer1,"java -jar mars.jar nc mc CompactDataAtZero dump .text HexText
code.txt >ans.txt %s",s),system(buffer1);
        sprintf(buffer1,"%s\\bin\\nt64\\fuse --nodebug  --prj mips.prj -o mips.exe
mips_tb >log.txt",isedir),system(buffer1);
        system("mips.exe -nolog -tclbatch mips.tcl >out.txt");
        filecompare(s); return;
```

```
}


int main(int argc,char *argv[])
{
    strcpy(filedir,argv[0]);
    for (int i=0;filedir[i];i++)
        if (filedir[i]=='/') filedir[i]=0;
    sprintf(buffer1,"XILINX=%s",isedir);
    putenv(buffer1),gettcl(),getprj();
    puts("initial secceed");
    for (int i=0;i<pointnum;i++)
    {
        sprintf(testpoint,"testpoint%d.asm",i+1);
        solve(testpoint);
    }
    return 0;
}
```

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？

因为一个 word 是四个 byte，所以输入的地址信号的低两位不用于确定 word，在 sw，lw 过程中不起作用，在 sb、lb、lbu、sh、lh、lhu 才会起作用，本次的 CPU 要求支持的指令集中无 sb、lb、lbu、sh、lh、lhu 这六个指令，因此地址信号 addr 位数是[11:2]而不是[9:0]，这个 addr 信号来自 ALU 的运算结果。

（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。

第一种方式：从指令出发，记录下每条指令对应的控制信号如何取值，优点在于可以方便的扩展指令，每次扩展指令后只需要增加一条判断即可，而不必做过多其它的修改；但在添加控制信号时，每条判断中都需要增加这条控制信号，当指令数量较多时手动进行这一操作是繁琐的(不过考虑到可以用高级语言根据真值表直接产生控制信号译码的代码，这一点无伤大雅)。

第二种方式：从控制信号出发，记录下控制信号每种取值所对应的指令，优点在于可以方便的添加控制信号，尤其对于 B、J 指令相关的部分数据通路，往往只有少数指令需要用到，通过记录下控制信号每种取值所对应的指令来添加控制信号十分方便快捷；但在添加新指令的过程中，尤其对于控制信号中 1 较多的指令，需要对很多控制信号增加判断，同样费时费力。

考虑到第一种方式的代码主要是纵向的延伸，第二种方式的代码主要是横向的延伸，第一种方式的阅读体验更佳；第一种方式的代码的格式性更好，更便于从高级语言生成控制信号译码的代码；信号通路的数量上限不会太高(ALU，CMP 等部件可预先留出足够的位用于选择不同的操作)，但可以扩展的指令相对较多，因此从这一点上考虑，从指令出发，记录下每条指令对应的控制信号如何取值相比于从控制信号出发，记录下控制信号每种取值所对应的指令更有优势，本人也采用了从指令出发，记录下每条指令对应的控制信号如何取值的方式进行控制信号的译码。

Verilog 语言设计控制器的代码示例如下：

```
if (|currentCommand==0) begin
    memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
    regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[31:26]==0) begin
    if (currentCommand[5:0]==`add) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AaddB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
```

```
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    if (currentCommand[5:0]==`addu) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AaddB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`sub) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AsubB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`subu) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AsubB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`and) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`or) begin
```

```
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AorB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`slt) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AsignedltB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`nor) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AnorB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`xor) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AxorB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[5:0]==`sll) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`BlogicleftA; ALUInputSelect<=2'b10; regWriteAddrSelect<=1;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b10;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
```

```
end
else if (currentCommand[5:0]==`srl) begin
    memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`BlogicrightA; ALUInputSelect<=2'b10; regWriteAddrSelect<=1;
    regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b10;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[5:0]==`sra) begin
    memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`BsignedrightA; ALUInputSelect<=2'b10; regWriteAddrSelect<=1;
    regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b10;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[5:0]==`sllv) begin
    memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`BlogicleftA; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
    regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[5:0]==`srlv) begin
    memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`BlogicrightA; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
    regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[5:0]==`srav) begin
    memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`BsignedrightA; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
```

```
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[5:0]==`sltu) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AunsignedltB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[5:0]==`jr) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
            regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b10; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[5:0]==`jalr) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=1;
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b10; linkWhenJump<=1; writeToReg31<=0;
        end
        else begin end
    end
    else if (currentCommand[31:26]==1) begin
        if (currentCommand[20:16]==`bltz) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=1;
            ALUOperation<=`AltzSet0; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
            regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
```

```
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[20:16]==`bgez) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=1;
            ALUOperation<=`AgezSet0; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
            regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else begin end
    end
    else begin
        if (currentCommand[31:26]==`lw) begin
            memOrALUToReg<=1; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[31:26]==`sw) begin
            memOrALUToReg<=0; memWriteEnabled<=1; branchControl<=0;
            ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
            regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[31:26]==`lb) begin
            memOrALUToReg<=1; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
            regWriteEnabled<=1; loadWriteMood<=3'b101; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
```

```
end
else if (currentCommand[31:26]==`lbu) begin
    memOrALUToReg<=1; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
    regWriteEnabled<=1; loadWriteMood<=3'b100; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[31:26]==`sb) begin
    memOrALUToReg<=0; memWriteEnabled<=1; branchControl<=0;
    ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
    regWriteEnabled<=0; loadWriteMood<=3'b100; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[31:26]==`lh) begin
    memOrALUToReg<=1; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
    regWriteEnabled<=1; loadWriteMood<=3'b011; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[31:26]==`lhu) begin
    memOrALUToReg<=1; memWriteEnabled<=0; branchControl<=0;
    ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
    regWriteEnabled<=1; loadWriteMood<=3'b010; extendMood<=3'b00;
    selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
end
else if (currentCommand[31:26]==`sh) begin
    memOrALUToReg<=0; memWriteEnabled<=1; branchControl<=0;
    ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
```

```
        regWriteEnabled<=0; loadWriteMood<=3'b010; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`beq) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=1;
        ALUOperation<=`AxorB; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
        regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`bne) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=1;
        ALUOperation<=`AnotBSet0; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
        regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`bgtz) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=1;
        ALUOperation<=`AgtzSet0; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
        regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`blez) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=1;
        ALUOperation<=`AlezSet0; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
        regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`addi) begin
```

```
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`addiu) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AaddB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`andi) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AandB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b01;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`ori) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AorB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b01;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
    end
    else if (currentCommand[31:26]==`xori) begin
        memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
        ALUOperation<=`AxorB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
        regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b01;
        selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
```

```
        end
        else if (currentCommand[31:26]==`lui) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`luiB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[31:26]==`slti) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AsignedltB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[31:26]==`sltiu) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AunsignedltB; ALUInputSelect<=2'b01; regWriteAddrSelect<=0;
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[31:26]==`j) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
            regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b01; linkWhenJump<=0; writeToReg31<=0;
        end
        else if (currentCommand[31:26]==`jal) begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
```

```
            regWriteEnabled<=1; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b01; linkWhenJump<=1; writeToReg31<=1;
        end
        else begin
            memOrALUToReg<=0; memWriteEnabled<=0; branchControl<=0;
            ALUOperation<=`AandB; ALUInputSelect<=2'b00; regWriteAddrSelect<=0;
            regWriteEnabled<=0; loadWriteMood<=3'b000; extendMood<=3'b00;
            selectOfJumpSource<=3'b00; linkWhenJump<=0; writeToReg31<=0;
        end
    end
```

（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？

清零信号 reset 所驱动的部件包括 PC、GRF 和 DM，复位后，PC 指向 0x3000，GRF 和 DM 中的所有数据清零，清零信号 reset 所驱动的部件的共同点包括：都是时序部件，使用前都需要初始化(reset)其中的内容以保证后续行为的正确性。

（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 addi 与 addiu 为例分析，add 与 addu 完全同理，在《MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set》中，有“Format: ADDI rt, rs, immediate. To add a constant to a 32-bit integer. If overflow occurs, then trap.”和“Format: ADDIU rt, rs, immediate. To add a constant to a 32-bit integer”，即 addi 是加立即数指令，支持溢出检测，具体表现为遇到溢出时，

溢出错误标志变为高电平，传送到控制器中，导致此时的寄存器写使能信号无效，最终结果为不将运算结果写入目的寄存器；addiu 是加立即数指令，不受溢出限制，具体表现为遇到溢出时，对溢出的结果进行 32bit 求模，将求模结果写入目的寄存器中，因而不受溢出限制。考虑到 C 语言中不对计算结果溢出进行处理，MIPS 指令的所有计算指令均可以忽略溢出，在忽略溢出的前提下，addi 和 addiu 都能计算出溢出后的结果，即进行 32bit 求模，并且将结果写入目的寄存器中，因此二者是等价的。

（五）

根据自己的设计说明单周期处理器的优缺点。

单周期处理器的优点在于实现简单，相比于多周期流水线拥有更高的性能，相比于流水线处理器使用更少的部件，且无需考虑指令间的冲突和延迟槽等问题。单周期处理器的缺点在于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；需要三个加法器，一个在 ALU 中，另两个用于 PC 的逻辑，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的情况；采用独立的指令存储器和数据存储器，而这在实际系统中是不现实的，大多数计算机采用一个单独的大容量存储器来存储指令和数据，并且同时支持读和写操作。