单周期 CPU（Verilog 实现）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，支持的指令集包括{addu,subu,ori,beq,lw,sw,lui,jal,jr,nop}，为了实现这些功能，CPU 主要包括了IM,GRF,ALU,EXT,DM,Controller,DM,PC,Decode,MUX。

（二）关键模块定义

1. GRF

寄存器堆内部核心是 32 个寄存器，本模块包含一个数据写入端口和两个数据输出端口以满足运算。0 号寄存器始终为 0。

表格 1 GRF 模块端口定义

信号名	方向	描述
Clk	I	时钟信号
Reset	I	复位信号，将 32 个寄存器的值全部清零 1：复位 0：无效
RegWrite	I	写使能信号 1：可向 GRF 中写入数据 0：不能向 GRF 中写入数据
A1[5:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中储存的数据读到 RD1
A2[5:0]	I	5 位地址输入信号，指定 32 个寄存器中的一个，将其中储存的数据读到 RD2
A3[5:0]	I	写寄存器地址
WD[31:0]	I	32 位写入数据
RD1[31:0]	O	输出 A2 指定的寄存器中的 32 位数据 A
RD2[31:0]	O	输出 A3 指定的寄存器中的 32 位数据 B
PC[31:0]	I	PC 值

2. IM

模块的具体定义与功能见下表。

表格 2　IFU 模块端口定义

信号名	方向	描述
toNPC[31:0]	I	当前 PC 地址加 4
Instr[31:0]	O	输出当前要执行的 32 位指令

3. ALU

提供 32 位加、减、或运算以及比较等功能。

表格 3　ALU 模块端口定义

信号名	方向	描述
Src1[31:0]	I	32 位输入 A
Src2[31:0]	I	32 位输入 B
Aluop[2:0]	I	控制信号 000:加 001:减 010:或运算 011:(lui)加载到高位
Zero	O	当 Src1 与 Src2 传入数据相等时输出 1，否则为 0
Result[31:0]	O	32 位输出数据

4. EXT

将 16 位立即数扩展为 32 位。

表格 4　EXT 模块端口定义

信号名	方向	描述
Imm[15:0]	I	待扩展 16 位输入
Extop	I	控制信号 0:无符号扩展 1:有符号扩展
Result[31:0]	O	32 位数据输出

5. DM

表格 5　DM 模块端口定义

信号名	方向	描述
MemAddr[5:0]	I	5 位写入地址
Data[31:0]	I	32 位输入数据
Clk	I	时钟信号
Reset	I	（异步）复位信号
MemWrite	I	写使能信号 1：可以向 DM 写入数据 0：无效
RD[31:0]	O	32 位输出数据
PC[31:0]	I	PC 值

6. PC

表格 6　PC 模块端口定义

信号名	方向	描述
NPC[31:0]	I	32 位输入数据
Clk	I	时钟信号
Reset	I	（异步）复位信号
ToNPC[31:0]	O	32 位输出数据

7. NPC

表格 7　NPC 模块端口定义

信号名	方向	描述
Imm26[25:0]	I	Instr[25:0]
ra[31:0]	I	32 位输入数据
zero	I	beq 判断信号
Npcop	I	NPC 选择信号
Pc_plus_4	O	PC+4
NPC[31:0]	O	32 位输出数据
PC[31:0]	I	PC 值

8. Controller

表格 8　Controller 模块端口定义

信号名	方向	描述
Op[5:0]	I	Instr[31:26]
Func[5:0]	I	Instr[5:0]
RegWrite	O	写使能信号 1：可向 GRF 中写入数据 0：不能向 GRF 中写入数据
ALUop[2:0]	O	控制信号 000:加 001:减 010:或运算 011:比较运算 100:取决于 func
RegDst	O	RW 选择信号 0：GRF 写入的寄存器地址为(Instr[16:20]) 1：GRF 写入的寄存器地址为(Instr[25:21])
ALUsrc	O	ALUsrcB 输入选择信号 0：SrcB 输入来自 GRF 的 RD2 输出 1：SrcB 输入来自 16 位 imm 扩展后的 32 位数
Extop	O	EXT 扩展类型选择信号 0：无符号扩展 1：有符号扩展
MemWrite	O	写使能信号 1：可以向 DM 写入数据 0：无效
MemtoReg	O	DM 读出控制信号、GRF 写入选择信号 0：GRF 的 WD 写入数据为 ALU_result，DM 输出使能无效 1：GRF 的 WD 写入数据为从 DM 输出 RD
NPCop[1:0]	O	IM 地址更新选择信号 0：PC <= PC + 4 1：beq 指令，PC <= PC + 4 + sign_extend(offset||00)
Lui_if	O	判断是否为 lui

9.　Decode

表格 9　Decode 模块端口定义

信号名	方向	描述
Instr[31:0]	I	32 位输入数据
Op[5:0]	O	Instr[31:26]
Func[5:0]	O	Instr[5:0]

Rs[4:0]	O	Instr[25:21]
Rt[4:0]	O	Instr[20:16]
Rd[4:0]	O	Instr[15:11]
Imm16[15:0]	O	Instr[15:0]
Imm26[26:0]	O	Instr[26:0]

10. MUX_A3

表格 10 MUX_A3 模块端口定义

信号名	方向	描述
rt[4:0]	I	Insts[20:16]
rd[4:0]	I	Instr[15:11]
RegDst	I	选择信号
A3[4:0]	O	5 位地址

11. MUX_lui

表格 11 MUX_lui 模块端口定义

信号名	方向	描述
ALU_result[31:0]	I	32 位 ALU 结果输入
Sel_lui[31:0]	I	32 位 lui 输入
Lui_if	I	选择信号
Out[31:0]	O	32 位输出信号

12. MUX_WD

表格 12 MUX_WD 模块端口定义

信号名	方向	描述
ALU_out[31:0]	I	32 位 ALU 结果输入
Mem_out[31:0]	I	32 位 lui 输入
MemtpReg[1:0]	I	选择信号
Pc_plus_4	I	Pc+4
WD_in[31:0]	O	32 位输出信号

13. MUX_ALU

表格 13　MUX_ALU 模块端口定义

信号名	方向	描述
RD2[31:0]	I	32 位 RD2 输入
EXT_out[31:0]	I	32 位 EXT 输入
ALUSrc	I	选择信号
srcB[31:0]	O	32 位输出信号

二、CPU 测试

1. ori 测试

MIPS:

```
.text
ori $t0, $zero, 123
ori $t1, $t0, 246
```

机器码:

```
3408007b
350900f6
```

实际输出：

```
@00003000: $ 8 <= 0000007b
@00003004: $ 9 <= 000000ff
```

2. 综合测试

MIPS:

```
.text
ori $t0, $t0, 123
addu $t1, $t1, $t0
subu $t2, $t1, $t0
beq $t2, $zero, if
addu $t3, $t1, $t1
if:
addu $t3, $t2, $t2
```

机器码:

```
3508007b
01284821
01285023
11400001
01295821
014a5821
```

实际输出：

@00003000: $ 8 <= 0000007b
@00003004: $ 9 <= 0000007b
@00003008: $10 <= 00000000
@00003014: $11 <= 00000000

3. 综合测试

MIPS:

```
.text
ori $t0, $t0, 123
addu $t1, $t1, $t0
subu $t2, $t1, $t0
jal if
beq $t2, $zero, end
addu $t3, $t1, $t1
if:
addu $t3, $t1, $t2
jr $31
end:
```

机器码:

```
3508007b
01284821
01285023
0c000c06
11400003
01295821
012a5821
03e00008
```

实际输出：

@00003000: $ 8 <= 0000007b
@00003004: $ 9 <= 0000007b
@00003008: $10 <= 00000000
@0000300c: $31 <= 00003010
@00003018: $11 <= 0000007b

4. Lw+sw 测试

MIPS:

```
.text
ori $t0, $t0, 123
sw $t0, 0($zero)
lw $t1, 0($zero)
```

机器码:

```
3508007b
```

ac080000
8c090000
实际输出：

```
@00003000: $ 8 <= 0000007b
@00003004: *00000000 <= 0000007b
@00003008: $ 9 <= 0000007b
```

三、思考题

1、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？
DM 的寻址方式是字节寻址，PC 的变化以 4 为颗粒度，输入的地址应当是 4 的整数倍（，因而在二进制表示中低两位为零）。在本设计中 DM 中的储存单
元定义为 reg [31:0] DataMemory [1023:0];这表明本设计实现时最终对储存单元的寻址以 1 为颗粒度，因此传入的地址需要逻辑右移两位，也即可以只取[11:2]位。在现有指令集中，只有 sw 和 lw 需要 DM 参与，其指令分别为：

```
//sw
DM[MemWriteAddr]      GRF_Reg[rt],
MemWriteAddr      GRF_Reg[rs]+sign_ext(Imm16)
( DM[ GRF_Reg[rs] + sign_ext(Imm16) ]      GRF_Reg[rt] )
//lw
GRF_Reg[rt]      DM[MemReadAddr],
MemReadAddr      GRF_Reg[rs]+sign_ext(Imm16)
( GRF_Reg[rt]      DM[ GRF_Reg[rs] + sign_ext(Imm16) ] )
```

故地址均从 ALU 的 result 来。
2、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 是针对哪些部件进行清零复位操作？
这些部件为什么需要清零？
Reset 信号是针对寄存器堆、指令储存器、数据储存器这三个部件的。原因是这三个部件存放的数据有储存性，声明周期甚至可以比当前运行程序还要长。想要达到清零效果，就必须对三个部件同时在某个上升沿清零。3、列举出用 Verilog 语言设计控制器的几种编码方式（至少三种），并给出代码
示例。
1）利用 case 或 if-else 语句，如：

```
if(Op==6'b0) begin
case(Funct)
6'b100001:addu
……
endcase
end
```

else begin
case(Op)
6'b001101:……
……
endcase
end
2）利用宏定义，如：
`define state1 4'b0001
`define state2 4'b0010
`define state3 4'b0100
`define state4 4'b1000
3）assign 语句，如：
assign addu = !op[5] & !op[4] & !op[3] & !op[2] & !op[1]
& !op[0] & func[5] & !func[4] & !func[3] & !func[2]
& !func[1] & func[0] ;
……
assign RegDst=addu | subu ;
……2、根据你所列举的编码方式，说明他们的优缺点。
1）case 语句：
优点：逻辑清晰容易理解，
缺点：容易出现错误，语法的精确掌握并不容易。
2）宏定义：
优点：易于修改和增加指令，
缺点：代码冗长，而且容易混淆。
3）assign 语句：
优点：代码不容易出现语法和逻辑上的错误，
缺点：有些语句过长，不很清晰。
4、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 。
addi 与 addiu 是涉及 16 位立即数的假发，add 和 addu 是寄存器之间的加法。这两组中的前者和后者之间差别即为是否考虑溢出。考虑溢出的指令在发生溢出时会报告 Overflow exception（通过是否改变标志位来判断）。
5、根据自己的设计说明单周期处理器的优缺点。
我在设计时使用了分布式译码与独热信号总线传输，使得每个模块可以在内部产生控制信号，避免了控制信号与实际需求脱节的情况，但是更改控制信号时要更改所有模块，因此修改起来略显繁琐。