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 实现的简单 MIPS 单周期处理器，支持的指令集包含{addu, subu, ori, lw, sw, beq, lui, jal, jr, nop}。为了实现这些功能，CPU 主要包含了 ALU、CU、DM、EXT、GRF、IM、NPC、PC 等模块。数据通路如下表所示。

表 1 数据通路表

部件	PC	NPC			IM	GRF				EXT	ALU		DM	
输入信号	PCI	PC	Imm	Ra	A	A1	A2	A3	DI	DI	A	B	A	DI
addu	NPC.NPC	PC.PCO			PC.PCO	IM.S[25:21]	IM.S[20:16]	IM.S[15:11]	ALU.C		GRF.DO1	GRF.DO2		
subu	NPC.NPC	PC.PCO			PC.PCO	IM.S[25:21]	IM.S[20:16]	IM.S[15:11]	ALU.C		GRF.DO1	GRF.DO2		
ori	NPC.NPC	PC.PCO			PC.PCO	IM.S[25:21]		IM.S[20:16]	ALU.C	IM.S[15:0]	GRF.DO1	EXT.DO		
lw	NPC.NPC	PC.PCO			PC.PCO	IM.S[25:21]		IM.S[20:16]	DM.DO	IM.S[15:0]	GRF.DO1	EXT.DO	ALU.C	
sw	NPC.NPC	PC.PCO			PC.PCO	IM.S[25:21]	IM.S[20:16]			IM.S[15:0]	GRF.DO1	EXT.DO	ALU.C	GRF.DO2
beq	NPC.NPC	PC.PCO	IM.S[15:0]		PC.PCO	IM.S[25:21]	IM.S[20:16]				GRF.DO1	GRF.DO2		
lui	NPC.NPC	PC.PCO			PC.PCO	IM.S[25:21]		IM.S[20:16]	ALU.C	IM.S[15:0]		EXT.DO		
jal	NPC.NPC	PC.PCO	IM.S[25:0]		PC.PCO			0x1f	NPC.PC4					
jr	NPC.NPC	PC.PCO		GRF.DO1	PC.PCO	IM.S[25:21]								
完整	NPC.NPC	PC.PCO	IM.S[15:0] IM.S[25:0]	GRF.DO1	PC.PCO	IM.S[25:21]	IM.S[20:16]	IM.S[15:11] IM.S[20:16] 0x1f	ALU.C DM.DO NPC.PC4	IM.S[15:0]	GRF.DO1	GRF.DO2 EXT.DO	ALU.C	GRF.DO2

（二）关键模块定义

1. GRF

GRF 负责读写 32 个寄存器，编号为 0~31，其中 0 号寄存器的值始终保持为 0，其它寄存器的初始值为 0。GRF 可以随时读出 2 个不同寄存器的内容，并在信号的控制下在时钟上升沿将指定内容写入指定寄存器，具有同步复位的功能。其端口设置如表 2 所示。

表 2 GRF 端口描述

端口名称	方向	功能描述
clk	I	时钟信号
reset	I	同步复位信号
GRFWrite	I	写使能信号
A1[4:0]	I	指定 32 个寄存器中的一个，将其中存储的数据读出到 DO1

A2[4:0]	I	指定 32 个寄存器中的一个，将其中存储的数据读出到 DO2
A3[4:0]	I	指定 32 个寄存器中的一个，作为写入的目标寄存器
DI[31:0]	I	需要写入 A3 指定的寄存器的数据
DO1[31:0]	O	32 位数据输出信号，表示 A1 指定的寄存器中的数据
DO2[31:0]	O	32 位数据输出信号，表示 A2 指定的寄存器中的数据

2. DM

DM 负责在 32bit*1024 大小的空间中读写数据，所有数据初始值为 0，起始地址为 0x00000000。具有同步复位功能。其端口设置如表 3 所示。

表 3 DM 端口描述

端口名称	方向	功能描述
clk	I	时钟信号
reset	I	同步复位信号
DMWrite	I	写使能信号
A[31:0]	I	要读入或输出的数据地址（以字节为单位）
DI[31:0]	I	需要写入的数据
DO[31:0]	O	A 指定的地址的数据（字）

3. ALU

ALU 提供 32 位加、减、或、左移 16 位（lui）运算功能（也可进行大小比较，即两数相减判断正负以及 0）。目前没有支持判断溢出的功能。其端口设置如表 4 所示。

表 4 ALU 端口描述

端口名称	方向	功能描述
ALUOp[2:0]	I	代表需要进行的运算操作 0：A+B 1：A-B 2：A|B 3：B<<16
A[31:0]	I	进行运算的第一个数

B[31:0]	I	进行运算的第二个数
C[31:0]	O	运算结果

4.CU

CU 根据机器码里的 Op 和 Funct 两部分判断当前 CPU 需要进行什么指令，根据不同指令的需求通过输出信号控制 CPU 各部分的运行。可以根据信号的名字和表 1 判断其用途。每个指令对应控制信号的值如表 5 所示。

表 5 CU 中每个指令对应的控制信号

输出信号	指令								
	addu	subu	ori	lw	sw	beq	lui	jal	jr
NPCOp[1:0]	00	00	00	00	00	01	00	10	11
GRFWrite	1	1	1	1	0	0	1	1	0
GRF_A3_Op[1:0]	00	00	01	01	XX	XX	01	10	XX
GRF_DI_Op[1:0]	00	00	00	01	XX	XX	00	10	XX
EXTOp	X	X	0	1	1	X	0	X	X
ALUOp[2:0]	000	001	010	000	000	001	011	XXX	XXX
ALU_B_Op	0	0	1	1	1	0	1	X	X
DMWrite	0	0	0	0	1	0	0	0	0

5. PC

PC 为寄存器，存储当前指令地址，具有同步复位功能，起始地址为 0x00003000。其端口设置如表 6 所示。

表 6 PC 端口描述

端口名称	方向	功能描述
clk	I	时钟信号
reset	I	同步复位信号
PCI[31:0]	I	下一步的指令地址
PCO[31:0]	I	当前输出的指令地址

6.NPC

NPC 根据当前指令的内容计算下一条指令的地址。其端口设置如表 7 所示。

表 7 PC 端口描述

端口名称	方向	功能描述
NPCOp[1:0]	I	控制信号
eq	I	ALU 计算结果是否为 0。用于判断 beq。
PC[31:0]	I	当前的指令地址
Imm[25:0]	I	（可能）需要跳转的立即数
Ra[31:0]	I	（可能）需要跳转的寄存器值
NPC[31:0]	O	下一步指令地址
PC4[31:0]	O	当前 PC+4

7. IM

IM 为一个 1024*32bit 大小的存储空间，根据当前 PC 读出指令。其端口设置如表 8 所示。

表 8 PC 端口描述

端口名称	方向	功能描述
A[31:0]	I	指令地址
S[31:0]	O	读出的指令

8.EXT

分别实现立即数的 0 扩展以及符号扩展。随后输出作为 ALU 的运算数或 PC 偏移量。其端口设置如表 9 所示。

表 9 PC 端口描述

端口名称	方向	功能描述
EXTSign	I	扩展方式控制信号
DI[15:0]	I	需要进行扩展的数
DO[31:0]	O	扩展后的数

（三）重要机制实现方法

1. 跳转

CPU 支持 beq、jal、jr 跳转指令。

本质上就是在 NPC 模块中加入一个控制信号 NPCOp，然后判断下一步的地址。

对于 beq，需要在 ALU 中判断两个寄存器的数值是否相等，若相等则将立即数进行符号扩展、左移的处理，作为偏移量加到 PC 值上。

对于 jal 和 jr，就可以直接赋值成对应的地址。

二、测试方案

（一）典型测试样例

1. ALU 与 GRF 功能测试

将只涉及 ALU 和 GRF 的指令多次修改寄存器和立即数，重复测试。特别注意了寄存器 0 和立即数大小为边界的例子。例如：

```
.text
   ori $0,$0,100
   lui $1,0xffff
   ori $2,$2,0x0f0f
   ori $3,$2,0xf0f0
   ori $4,$5,100
   ori $5,$4,200
   addu $6,$4,$5
   subu $7,$5,$4
   lui $8,0
```

2. DM 功能测试

主要测试 lw 与 sw 指令，和上面的指令相结合。例如：

```
.data
   A: .space 1000
.text
   ori $0,$0,100
   lui $1,0xffff
   ori $2,$2,0x0f0f
   ori $3,$2,0xf0f0
```

```
ori $4,$5,100
ori $5,$4,200
addu $6,$4,$5
subu $7,$5,$4
lui $8,0
ori $9,$0,0
ori $10,$0,4
lw $11,A($9)
sw $1,A($9)
sw $5,A($10)
lw $11,A($9)
lw $12,A($10)
```

3. 跳转指令测试

用跳转指令多构造一些向前、向后的情况，同时也可以检查其他指令的操作是否正确。例如：

```
.data
   A: .space 1000
.text
   jal f3
   jal f3
   f1:
   lui $1,100
   ori $2,$0,100
   ori $3,$0,200
   addu $4,$4,$2
   sw $4,A
   lw $5,A
   beq $3,$4,f2
   beq $4,$5,f1
   f2:
   nop
   jal f4
   f5:
   jr $ra
   f3:
   jr $ra
   f4:
   jal f5
```

4. 同步复位测试

在上面的过程中时不时使用 reset，检查是否复位。

（二）自动测试工具

1. 测试样例生成器

无

2. 自动执行脚本

使用 python，设置环境变量，通过命令行生成、运行仿真程序，将输出重定向到一个 txt 中。

```
xilinx_path='C:\\Xilinx\\14.7\\ISE_DS\\ISE'
os.environ['XILINX'] = xilinx_path
os.system(xilinx_path + '\\bin\\nt64\\fuse -nodebug -prj mips.prj -o mips.exe test>log.txt')
os.system('mips.exe -nolog -tclbatch mips.tcl> out2.txt')
```

3. 正确性判定脚本

我使用 cpp 编写了一个模拟 CPU 的程序，可以将模拟结果输出到另一个文件，然后用命令行的 diff 语句判断。

```
diff out1.txt out2.txt
```

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

答： 可以看做 addr 右移了两位，这样就相当于按字标号，每次读出对应一字数据，但位数又是和整体地址对应的，比较整齐。Addr 信号是从 ALU 中运算得出的。

（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。

答：第一种是指令对应控制信号取值。优点是可以方便对于某个确定的指令 debug，也方便增加指令，缺点是找到对应的信号比较麻烦。

代码片段如下：

```
reg [12:0] res;
assign {NPCOp,GRFWrite,GRF_A3_Op,GRF_DI_Op,EXTSign,ALUOp,ALU_B_Op,DMWrite}=res;

always @(*)
begin
   case (Opcode)
      6'b000000:
      begin
         case (Funct)
            6'b100001://addu
               res<=13'b0_0100_0000_0000;
            6'b100011://subu
               res<=13'b0_0100_0000_0100;
            6'b001000://jr
               res<=13'b1_1000_0000_0000;
            default: res<=0;
         endcase
      end

      6'b001101://ori
         res<=13'b0_0101_0000_1010;
      6'b100011://lw
         res<=13'b0_0101_0110_0010;
      6'b101011://sw
         res<=13'b0_0000_0010_0011;
      6'b000100://beq
         res<=13'b0_1000_0000_0100;
      6'b001111://lui
         res<=13'b0_0101_0000_1110;
      6'b000011://jal
         res<=13'b1_0110_1000_0000;
      default: res<=0;
   endcase
end
```

第二种是控制信号每种取值对应指令，类似于 losigim 搭建单周期 cpu 时的与或门阵列。优点是比较直观，可以对于某个确定的控制信号错误 debug，缺点是代码量略大，比第一种直接用位拼接赋值的形式麻烦。

代码片段如下：

```
assign Addu=(Opcode==6'b000000 && Funct==6'b100001);
......
assign GRFWrite=Addu|Subu|Ori|Lw|Lui|Jal|Jr;
......
```

（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？

答： reset 驱动的部件有 PC、DM、GRF，他们都是有存储功能的部件，都在时钟上升沿触发，复位之后整个程序能够完全回到运行之前的状态。

（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 。

答：MIPS 中的 addu 与 addiu 实际上是指模 2^32 意义下的加法，因此不会产生溢出。而这种意义下的加法可以适用于无符号整数，所以被命名为 unsigned。同样，也可以适用于忽略溢出的情况，就像 C 语言里的加法一样。

MIPS 中的 addi 与 add 在溢出的时候会产生错误信号，在忽略溢出的时候便不需要，只需执行正常加法即可。

因此，在忽略溢出的前提下，addi 与 addiu 等价，add 与 addu 等价。

（五）根据自己的设计说明单周期处理器的优缺点。

答：

优点：实现简单，结构清晰，搭建方便，适合入门理解 CPU 的结构。

缺点：相比流水线结构，单周期 CPU 部件经常闲置，不能有效利用，效率低。