单周期 CPU

一、 数据通路设计

(一) IFU(Instruction Fetch Unit)

1. 基本描述

IFU 的主要功能是完成取指令的功能，内部包括 PC、IM(指令存储器)以及其他相关逻辑。IFU 除了能顺序执行指令以外，还能根据 BEQ 指令的执行情况决定顺序取指令还是转移取指令。

2. 基本部件端口定义

信号名	方向及位数	功能描述
PC_branch	input signed [31:0]	分支指令的地址偏移量
PC_jump	input [31:0]	j 指令跳转地址
PC_jr	input [31:0]	jr 指令跳转地址
Reset	input	复位信号
Clk	input	时钟信号
Zero	input	ALU 计算结果是否为 0 1：ALU 结果计算为 0 0：ALU 结果计算不为 0
nPC_sel	input [1:0]	下一个 pc 值的选择信号 2'b00:PC+4 2'b01:br 分支指令 2'b10:j 指令 2'b11:jr 指令
pc	output reg [31:0]	存储当前指令的地址
pc4	output [31:0]	输出 pc+4

3. nPC_sel 功能选择信号定义

功能编号	功能代号	功能描述
2'b00	IFU_add4	NPC = PC+4
2'b01	IFU_beq	NPC = PC+4 or PC+4+ PC_branch/4
2'b10	IFU_j	NPC = {pc[31:28],PC_jump[25:0],2'b0}
2'b11	IFU_jr	NPC = PC_jr

(二) IM(Instruction Memory 指令存储器)

1. 基本描述

IM 的功能是存储指令，首先从 code.txt 文件中读入 16 进制的指令，存入到 instructions 模块当中。可以接收地址信息并传出指令信息。

2. 端口定义及功能描述

信号名	方向及位数	功能描述
pc	input [31:0]	读取指令的地址
Instr	output [31:0]	取出 pc 对应的指令
寄存器 instructions[1023:0]	[31:0]	实际的指令地址从 0 开始存储 读入的 pc 要为(pc-0x00003000)/4

(三) GRF(General Register File 通用寄存器组)

1. 基本描述

GRF 主要功能是读写寄存器。GRF 内部包括 32 个寄存器以及其他相关逻辑。时钟上升沿到来时，写使能有效时，将数据写入相应的寄存器，并将相应寄存器的数据输出。

2. 端口定义及功能描述

信号名	方向	位数	功能描述
RA	in	[4:0]	读取寄存器 A 的下标
RB	in	[4:0]	读取寄存器 B 的下标
RW	in	[4:0]	写入寄存器的下标
WD	in	[31:0]	写入寄存器的数据
WE	in	1	写使能端口 在时钟上升沿到来时，如果 WE 为 1，则将 WD 的数据写入 RW 对应下标的寄存器 如果 WE 为 0，则数据无法写入
Clk	in	1	时钟信号
Reset	in	1	复位信号 1：复位有效，所有寄存器被置为 0x00000000 0：复位信号无效
busA	out	[31:0]	A 寄存器读出的数据
busB	out	[31:0]	B 寄存器读出的数据

(四) ALU(算数逻辑单元)

1. 基本描述

ALU 主要功能是将相应的数进行与控制信号相对应的运算。支持加减与或，右移 16 位等运算。

2. 端口定义

信号名	方向	位数	功能描述
A	in	[31:0]	读入的运算数 A
B	in	[31:0]	读入的运算数 B
Result	out	[31:0]	运算结果
Zero	out	1	运算结果是否为 0 1：为 0 0：不为 0
ALUCtr	in	[3:0]	运算功能选择信号

3. ALUCtr 信号功能定义

功能编号	功能代号	功能	功能描述
4’b0000	ALU_addu	无符号加	result = A + B
4’b0001	ALU_subu	无符号减	result = A - B
4’b0010	ALU_and	按位与	result = A & B
4’b0011	ALU_or	按位或	result = A | B
4’b0100	ALU_sll16	左移 16 位	result = B<<16

(五) DM(Data Memory 数据存储器)

1. 基本描述

DM 主要功能是储存数据，使用 RAM 实现，容量为 32bit * 32。

2. 端口定义

信号名	方向	位数	功能描述
DataIn	in	[31:0]	写入的数据
DataOut	out	[31:0]	读出的数据
Addr	in	[31:0]	数据的写入地址
WrEn	in	1	是否写入数据到 Addr
Reset	in	1	复位
Clk	in	[3:0]	时钟信号

(六) EXT

1. 基本描述

EXT 主要功能是将 16 位立即数扩展成 32 位。支持符号扩展和 0 扩展。

2. 端口定义

信号名	方向	位数	功能描述
Imm16	in	[15:0]	16 位立即数
Imm32	out	[31:0]	32 位拓展结果
ExtOp	in	1	控制立即数做拓展方式 0：0 拓展 1：符号拓展

二、 控制器设计

1. 基本描述

Controller 主要功能是通过指令的 opcode 和 funct 字段来判断指令的类型，从而决定各个部件的控制信号。

2. 端口定义

信号名	方向	位数	功能描述
OpCode	in	[5:0]	指令的 OpCode 字段
func	in	[5:0]	指令的 func 字段
ALUCtr	out	[3:0]	决定 ALU 的运算操作 4’b0000 加 4’b0001 减 4’b0010 按位与 4’b0011 按位或 4’b0100 左移 16 位
RegDst	out	[1:0]	2’b00：写入到 rd，即写入编号是指令中 11-15 位的寄存器(addu,subu) 2’b01：写入到 rt：写入编号为指令中 16-20 位的寄存器(ori,lui) 2’b10：写入到 31 号寄存器(用于 jal 指令)
RegWrite	out	1	0：寄存器写使能端口无效 1：寄存器写使能端口有效
ALUSrc	out	1	0：ALU B 口读入的是立即数 1：ALU B 口读入的是 GRF 的 busB
RegSrc	out	[1:0]	2’b00：从 ALU 运算结果读入到寄存器(addu,subu,ori,lui) 2’b01：从 DM 中读入值到寄存器(lw) 2’b10：从 PC4 读入值到寄存器(jal 指令)

MemWrite	out	1	0：DM 写使能无效 1：DM 写使能有效
nPC_sel	out	[1:0]	选择下一个 PC 地址信号 2’b00:PC+4 2’b01:br 分支指令 2’b10:j 指令 2’b11:jr 指令
ExtOp	out	1	0：进行 0 拓展 1：进行有符号拓展

3. 指令分析

见 excel 表格

指令	IFU			GRF				ALU		DM		EXT
	PC_branch	PC_jump	PC_jr	RA	RB	RW	WD	A	B	DataIn	MemAddr	imm16
addu				IM.Instr[25:21]	IM.Instr[20:16]	IM.Instr[15:11]	ALU.Result	GRF.busA	GRF.busB			
subu				IM.Instr[25:21]	IM.Instr[20:16]	IM.Instr[15:11]	ALU.Result	GRF.busA	GRF.busB			
ori				IM.Instr[25:21]		IM.Instr[20:16]	ALU.Result	GRF.busA	EXT.imm32			IM.Instr[15:0]
lw				IM.Instr[25:21]		IM.Instr[20:16]	DM.DataOut	GRF.busA	EXT.imm32	GRF.busB		IM.Instr[15:0]
sw				IM.Instr[25:21]	IM.Instr[20:16]			GRF.busA	EXT.imm32	GRF.busB	ALU.Result	IM.Instr[15:0]
beq	EXT.imm32			IM.Instr[25:21]	IM.Instr[20:16]			GRF.busA	GRF.busB			IM.Instr[15:0]
lui						IM.Instr[20:16]	ALU.Result		GRF.busB			IM.Instr[15:0]
jal		IM.Instr[25:0]				常量31	IFU.pc4					
jr			GRF.busA									
综合	EXT.imm32	IM.Instr[25:0]	GRF.busA	IM.Instr[25:21]	IM.Instr[20:16]	IM.Instr[15:11]	ALU.Result	GRF.busA	GRF.busB	GRF.busB	ALU.Result	IM.Instr[15:0]
						IM.Instr[20:16]	DM.DataOut		EXT.imm32			
						常量31	IFU.pc4					

4. 指令与控制信号真值表

见 excel 表格

	Input		Output							
指令	OPCode	Func	[1:0]nPC_sel	[3:0]ALUCtr	RegWrite	MemWrite	ExtOp	[1:0]RegDst(Mux_GRF_RW)	[1:0]RegSrc(Mux_GRF_WD)	ALUSrc(Mux_ALU_B)
addu	000000	100001	00(`IFU_add4)	0000(`ALU_addu)	1	0	x	00(`RD_15_11)	00(`RegS_ALU)	0(`ALUS_busB)
subu	000000	100011	00(`IFU_add4)	0001(`ALU_subu)	1	0	x	00(`RD_15_11)	00(`RegS_ALU)	0(`ALUS_busB)
ori	001101		00(`IFU_add4)	0011(ALU_or)	1	0	0`Ext_unsigned	01(`RD_20_16)	00(`RegS_ALU)	1(`ALUS_Imm)
lw	100011		00(`IFU_add4)	0000(`ALU_addu)	1	0	1`Ext_signed	01(`RD_20_16)	01(`RegS_DM)	1(`ALUS_Imm)
sw	101011		00(`IFU_add4)	0000(`ALU_addu)	0	1	1`Ext_signed	x	x	1(`ALUS_Imm)
beq	000100		01(`IFU_beq)	0001(`ALU_subu)	0	0	1`Ext_signed	x	x	0(`ALUS_busB)
lui	001111		00(`IFU_add4)	0100(ALU_sll16)	1	0	x	01(`RD_20_16)	00(`RegS_ALU)	0(`ALUS_busB)
jal	000011		10(`IFU_j)	x	1	0	x	10(`RD_31)	10(`RegS_PC4)	x
jr	000000	001000	11(`IFU_jr)	x	0	0	x	x	x	x
nop	000000	000000	00(`IFU_add4)	x	0	0	x	x	x	x

三、 测试程序

(一) MIPS 代码

```
.text
nop                 #空指令
ori $29 $0 0x0      #清空$sp
ori $28 $0 0x0      #清空$gp
#######算数类指令测试
lui $1 0x1234       #$1 = 0x12340000
```

```
ori $1 $1 0x8888      #$1 = 0x12348888
ori $2 $1 0x6666          #$2 = 0x1234eeee
lui $3 0xffff
ori $3 0xffff         #3 = 0xffffffff
######算数情况
addu $3 $3 $1         #$3 =    0x12348887
subu $4 $3 $1         #$4 =    0xffffffff
#####beq 不相等
beq $4,$3,aa          #$4!=$3  继续顺序执行
nop
#####sw
sw $4 4($0)       #结果：dm[4] = 0xffffffff
ori $5 $0 24      #$5 = 0x18(24)
sw $3 -12($5)         #dm[0xc] = 0x12345678
aa:
#####lw
lw $6 -12($5)         #$6 = 0x12348887
lw $7 4($0)       #$7 = 0xffffffff
lw $8 4($0)       #$8 = 0xffffffff

beq $7 $8 bb          #跳到 bb
nop
lw $6 0x10($0)        #结果：未执行

bb:
ori $7,$7,7       #$7 =$7 or 7                    结果：$7=7
jal cc            #跳转到 cc
nop
ori $8,$8,8       #$8 =$8 or 8                    结果：$8=8
```

```
jal end             #跳转到 end
nop
cc:
ori $9,$9,9         #$9 =$9 or 9                              结果：$9=9
jr $ra              #跳转到$ra
nop
end:
ori $10,$10,1           #$10 =$10 or 1          $10 为常数 1  结果：$10=1
ori $11,$11,1           #$11 =$11 or 1          $11 为循环变量 i 结果：$11=1
ori $12,$12,1           #$12 =$12 or 1        $12 为第 i 个斐波拉契数结果：$12=1
ori $13,$13,0           #$13 =$13 or 0          $13 为第 i-1 个斐波拉契数
结果：$13=0
ori $14,$14,20              #$14 =$14 or 20        $14 为 n=20
结果：$14=21
for_1:
jal calc            #jal  调用函数 calc 计算新的斐波拉契
nop
lui $ra,0           #清零$RA
beq $11,$14,for_end     #判断循环是否终止
nop
addu $11,$11,$10        #i=i+1
j for_1
nop
for_end:
j eeend
nop
calc:
    lui $15,0       #将$15 赋值为 0
    addu $15,$15,$12            #将$15 赋值为当前的 f[i]
    addu $12,$13,$12            #f[i]=f[i]+f[i-1]
```

```
    lui $13,0       #将$13 赋值为 0
    addu $13,$13,$15            #将$13 赋值为当前的 f[i-1]
    jr $ra                  #返回调用处
    nop
eeend:
addu $12,$12,$0                 #展示最后结果
```

(二) 指令 16 进制码

```
00000000
341d0000
341c0000
3c011234
34218888
34226666
3c03ffff
3463ffff
00611821
00612023
10830004
00000000
ac040004
34050018
aca3fff4
8ca6fff4
8c070004
8c080004
10e80002
00000000
8c060010
34e70007
```

0c000c1b

00000000

35080008

0c000c1e

00000000

35290009

03e00008

00000000

354a0001

356b0001

358c0001

35ad0000

35ce0014

0c000c2d

00000000

3c1f0000

116e0004

00000000

016a5821

08000c23

00000000

08000c34

00000000

3c0f0000

01ec7821

01ac6021

3c0d0000

01af6821

03e00008

00000000

01806021

(三) mars 运行结果

Name	Number	Value
$zero	0	0x00000000
$at	1	0x12348888
$v0	2	0x1234eeee
$v1	3	0x12348887
$a0	4	0xffffffff
$a1	5	0x00000018
$a2	6	0x12348887
$a3	7	0xffffffff
$t0	8	0xffffffff
$t1	9	0x00000009
$t2	10	0x00000001
$t3	11	0x00000014
$t4	12	0x00002ac2
$t5	13	0x00001a6d
$t6	14	0x00000014
$t7	15	0x00001a6d
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x00000000
$sp	29	0x00000000
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x000030d4
hi		0x00000000
lo		0x00000000

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000000	0xffffffff	0x00000000	0x12348887	0x00000000	0x00000000	0x00000000	0x00000000
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

(四) CPU 运行结果

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x0	00	00	00	00	FF	FF	FF	FF	00	00	00	00	87	88	34	12
0x20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x60																

	0	1	2	3	4	5	6	7
0x0	00000000	12348888	1234EEEE	12348887	FFFFFFFF	00000018	12348887	FFFFFFFF
0x8	FFFFFFFF	00000009	00000001	00000014	00002AC2	00001A6D	00000014	00001A6D
0x10	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
0x18	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

四、 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？

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

因为在 dm 里面存储的数据是以 32 位为单位的，即一个字，而读入的地址数据是以字节为单位的，需要舍掉最后两位，addr 信号是从 ALU 的 result 里来的，是一个 32 位数，被压缩成 10 位信号。

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 是针对 DM，PC 和 GRF 进行复位。当需要复位时，要将 cpu 恢复到最初的状态，pc 就回到 0x00003000 首地址，GRF 中的寄存器和 DM 都回到最初 0 的状态，这样其余的部件就相应的回到了最初的状态。

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.独热码

```
`define ALU_add      4'b0001
`define ALU_sub      4'b0010
`define ALU_or       4'b0100
```

`define ALU_and 4'b1000

2.格雷码

`define ALU_add 4'b0000

`define ALU_sub 4'b0001

`define ALU_or 4'b0011

`define ALU_and 4'b0010

3.二进制编码

`define ALU_add 4'b0000

`define ALU_subu 4'b0001

`define ALU_or 4'b0010

`define ALU_and 4'b0011

4. 根据你所列举的编码方式，说明他们的优缺点。

二进制编码、格雷码编码使用最少的触发器，消耗较多的组合逻辑，而独热码编码反之。

独热码编码的最大优势在于状态比较时仅仅需要比较一个位，从而一定程度上简化了译码逻辑。虽然在需要表示同样的状态数时，独热编码占用较多的位，也就是消耗较多的触发器，但这些额外触发器占用的面积可与译码电路省下来的面积相抵消。

格雷码每个相邻信号之间只有一位发生了变化，减小功耗，但是编码比较复杂。

5. 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 。

```
temp ← (GPR[rs]31‖GPR[rs]) + (GPR[rt]31‖GPR[rt])
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp31..0
endif
```

Add 的描述如上图。其中，他将两个数的第 31 位复制了一遍在每个数的第一位，用来判断是否溢出，而如果不考虑溢出，temp31_0 的值正是两个寄存器相加将溢出位舍去的值,便与 addu 相同。 而 addiu 中立即数采用的符号扩展因此和上面情况相同。

6. 根据自己的设计说明单周期处理器的优缺点。

优点：简单，每条指令一个时钟周期，指令之间不会冲突。
缺点：效率太低，单周期 cpu 中每个时钟周期要由执行时间最长的那条指令决定。

7. 简要说明 jal、jr 和堆栈的关系。

当需要调用函数时，需要 jal 来将当前的 pc+4 存入寄存器 31，而退出函数，时通过 jr 来返回调用前的位置。而当需要递归或多层调用时，$31 的值就会被覆盖掉。因此还需要在每次调用函数前将$31 存入堆栈中。