P4 设计文档

——Verilog 单周期

一、指令集

1. ADDU：不支持溢出的加法

编码	Opcode 000000	rs	rt	rd	Shamt 00000	Func 100001
	6	5	5	5	5	6
描述	GPR[rd] ← GPR[rs] + GPR[rt]					

2. SUBU：不支持溢出的减法

编码	Opcode 000000	rs	rt	rd	Shamt 00000	Func 100011
	6	5	5	5	5	6
描述	GPR[rd] ← GPR[rs] - GPR[rt]					

3. ORI：或立即数

编码	Opcode 001101	rs	rt	immediate
	6	5	5	16
描述	GPR[rt] ← GPR[rs] OR zero_extend(immediate)			

4. LW：加载字

编码	Opcode 100011	base	rt	offset
	6	5	5	16
描述	Addr ← GPR[base] + sign_extend(offset) GPR[rt] ← memory[Addr]			

5. SW：存储字

编码	Opcode 101011	base	rt	offset
	6	5	5	16
描述	Addr ← GPR[base] + sign_extend(offset) memory[Addr] ← GPR[rt]			

6. BEQ：相等时跳转

编码	Opcode 000100	rs	rt	offset
	6	5	5	16

描述	If (GPR[rs] == GPR[rt])
	PC ← PC + 4 + sign_extend(offset||0^2)
	Else
	PC ← PC + 4			

7. LUI：立即数加载至高位

编码	Opcode 001111	0 00000	rt	immediate
	6	5	5	16
描述	GPR[rt] ← immediate||0^16			

8. JAL：跳转并链接

编码	Opcode 000011	index
	6	26
描述	PC ← PC[31:28]||index||00 GPR[31] ← PC+4	

9. JR：跳转至寄存器

编码	Opcode 000000	rs	0	0	Func 001000
	6	5	10	5	6
描述	PC ← GPR[rs]				

二、模块规格

1. PC：指令地址寄存器

模块端口定义如下：

信号名	位数	方向	描述
clk		I	内置时钟信号
reset		I	同步复位信号
PCI	[31:0]	I	下一条指令的地址
PC	[31:0]	O	当前指令的地址

模块功能说明如下：

序号	功能名	描述
1	读指令地址	通过 PC 端口输出当前指令地址
2	写指令地址	时钟上升沿到来时更新指令地址
3	同步复位	时钟上升沿到来时若 reset 信号为 1，则复位指令地址至初始状态 0x00003000

2. NPC：指令地址计算器

模块端口定义如下：

信号名	位数	方向	描述
NPCI	[31:0]	I	当前指令地址信号
NPCIMM	[31:0]	I	参与下一条指令地址计算的立即数信号
NPCOp	[1:0]	I	下一条指令地址计算的选择信号
NPC	[31:0]	O	下一条指令的地址信号

模块功能说明如下：

序号	功能名	描述
1	计算指令地址	若 NPCOp==00，计算 NPCI+4 并通过 NPC 端口输出 若 NPCOp==01，计算 NPCI+4+NPCIMM 并通过 NPC 端口输出 若 NPCOp==10，将 NPCIMM 通过 NPC 端口输出

3. IM：指令存储器

模块端口定义如下：

信号名	位数	方向	描述
IMI	[31:0]	I	当前指令地址信号
IM	[31:0]	O	当前指令信号

模块功能说明如下：

序号	功能名	描述
1	读指令	读出指令存储器中 IMI 地址对应的指令并通过 IM 端口输出
2	初始化写指令	初始时向指令存储器中读入所有指令

4. GRF： 寄存器堆

模块端口定义如下：

信号名	位数	方向	描述
clk		I	内置时钟信号
reset		I	同步复位信号
WE		I	写使能信号
A1	[4:0]	I	第一个地址输入信号
A2	[4:0]	I	第二个地址输入信号
A3	[4:0]	I	第三个地址输入信号
WD	[31:0]	I	写入数据信号
PC	[31:0]	I	当前指令的地址信号
RD1	[31:0]	O	A1 所对应的寄存器的数据信号
RD2	[31:0]	O	A2 所对应的的寄存器的数据信号

模块功能说明如下：

序号	功能名	描述
1	读数据	读出 A1，A2 地址对应寄存器数据并通过 RD1，RD2 端口输出
2	写数据	时钟上升沿到来时，若 WE 信号为 1，则向 A3 地址对应寄存器写入数据 WD（0 号寄存器不能被写入）并输出一条与 PC 有关的信息
3	同步复位	时钟上升沿到来时，若 reset 信号为 1，则复位所有寄存器至初始状态 0x00000000

5. ALU：计算模块

模块端口定义如下：

信号名	位数	方向	描述
ALUA	[31:0]	I	参与运算的第一个数据信号
ALUB	[31:0]	I	参与运算的第二个数据信号
ALUOp	[2:0]	I	运算方式的选择信号
ALU	[31:0]	O	运算结果的数据信号

模块功能说明如下：

序号	功能名	描述
1	不支持溢出加法	若 ALUOp==001，计算 A+B 并通过 ALU 端口输出
2	不支持溢出减法	若 ALUOp==010，计算 A-B 并通过 ALU 端口输出
3	按位或运算	若 ALUOp==011，计算 A|B 并通过 ALU 端口输出

6. EXT：立即数拓展模块

模块端口定义如下：

信号名	位数	方向	描述
EXTIMM	[15:0]	I	参与拓展的立即数信号
EXTOp	[2:0]	I	拓展方式的选择信号
EXT	[31:0]	O	拓展完成后的数据信号

模块功能说明如下：

序号	功能名	描述
1	零拓展	若 EXTOp==001，将立即数加载至输出信号低位并用 0 填充输出信号的高 16 位
2	符号拓展	若 EXTOp==010，将立即数加载至输出信号低位并用其最高位填充输出信号的高 16 位
3	补 00 符号拓展	若 EXTOp==011，将立即数末尾补两个 0 后进行符号拓展
4	加载至高位	若 EXTOp==100，将立即数加载至输出信号高位并用 0 填充输出信号的低 16 位

7. DM：数据存储器

模块端口定义如下：

信号名	位数	方向	描述
clk		I	内置时钟信号
reset		I	同步复位信号
DMA	[31:0]	I	存取的地址信号
DMD	[31:0]	I	存取的数据信号
DMWE		I	存数据使能信号
PC	[31:0]	I	当前指令的地址信号
DM	[31:0]	O	取出的数据信号

模块功能说明如下：

序号	功能名	描述
1	存数据	时钟上升沿到来时，若 DMWE 信号为 1，则向数据存储器 DMA 对应地址中写入 DMD 数据并输出一条与 PC 有关的信息
2	取数据	将数据存储器 DMA 地址对应的数据通过 DM 端口输出
3	同步复位	时钟上升沿到来时，若 reset 信号为 1，则复位数据存储器至初始状态 0x00000000

三、控制器设计

1. 控制信号说明

序号	信号名	位数	描述
1	NPCOp	[1:0]	作为计算 NPC 时的选择信号
2	WE		作为 GRF 的写使能信号
3	ALUOp	[2:0]	作为 ALU 的计算方式选择信号
4	EXTOp	[2:0]	作为 EXT 的拓展方式选择信号
5	DMWE		作为 DM 的写使能信号
6	GRFA3_MUXOp	[1:0]	作为 GRF 的 A3 端口输入信号的选择信号
7	GRFWD_MUXOp	[1:0]	作为 GRF 的 WD 端口输入信号的选择信号
8	ALUB_MUXOp		作为 ALU 的 B 端口输入信号的选择信号
9	NPCIMM_MUXOp	[1:0]	作为 NPC 的 IMM 端口输入信号的选择信号

2. 指令与控制信号真值表

func	100001	100011							001000	000000
opcode	000000	000000	001101	100011	101011	000100	001111	000011	000000	000000
指令名	addu	subu	ori	lw	sw	beq	lui	jal	jr	nop
NPCop	00	00	00	00	00	01	00	10	10	0
WE	1	1	1	1	0	0	1	1	0	0
ALUOp	001	010	011	001	001	010	000	000	000	000
EXTOp	000	000	001	010	010	011	100	000	000	000
DMWE	0	0	0	0	1	0	0	0	0	0
GRFA3_MUXOp	01	01	00	00	00	00	00	10	00	0
GRFWD_MUXOp	00	00	00	01	00	00	10	11	00	00
ALUB_MUXOp	1	1	0	0	0	1	0	0	0	0
NPCIMM_MUXOp	00	00	00	00	00	00	00	01	10	00

四、数据通路

1. 信号连接表

	NPC	GRF			
输入信号	NPCIMM	A1	A2	A3	WD
addu		IM.IM[25:21]	IM.IM[20:16]	IM.IM[15:11]	ALU.ALU
subu		IM.IM[25:21]	IM.IM[20:16]	IM.IM[15:11]	ALU.ALU
ori		IM.IM[25:21]		IM.IM[20:16]	ALU.ALU
lw		IM.IM[25:21]		IM.IM[20:16]	DM.DM
sw		IM.IM[25:21]	IM.IM[20:16]		
beq	EXT.EXT	IM.IM[25:21]	IM.IM[20:16]		
lui				IM.IM[20:16]	EXT.EXT
jal	PC||index			31	PC
jr	GRF.RD1	IM.IM[25:21]			
nop					
ALL	EXT.EXT PC||index GRF.RD1	IM.IM[25:21]	IM.IM[20:16]	IM.IM[15:11] IM.IM[20:16]	ALU.ALU DM.DM EXT.EXT PC
	EXT	ALU		DM	
输入信号	EXTIMM	ALUA	ALUB	DMA	DMD
addu		GRF.RD1	GRF.RD2		
subu		GRF.RD1	GRF.RD2		
ori	IM.IM[15:0]	GRF.RD1	EXT.EXT		
lw	IM.IM[15:0]	GRF.RD1	EXT.EXT	ALU.ALU	
sw	IM.IM[15:0]	GRF.RD1	EXT.EXT	ALU.ALU	GRF.RD2
beq		GRF.RD1	GRF.RD2		
lui	IM.IM[15:0]	GRF.RD1			
jal					
jr					
nop					
ALL	IM.IM[15:0]	GRF.RD1	GRF.RD2 EXT.EXT	ALU.ALU	GRF.RD2

2. 冲突选择表

GRFA3			
选择信号 GRFA3_MUXOp	00	01	10
连接信号	IM.IM[20:16]	IM.IM[15:11]	0x0000001f

GRFWD				
选择信号 GRFWD_MUXOp	00	01	10	11
连接信号	ALU.ALU	DM.DM	EXT.EXT	PC.PC+4

ALUB		
选择信号 ALUB_MUXOp	0	1
连接信号	EXT.EXT	GRF.RD2

NPCIMM			
选择信号 NPCIMM_MUXOp	00	01	10
连接信号	EXT.EXT	PC||index	GRF.RD1

五、测试样例

样例 1:　　期望结果：

```
ori $t1,$0,1          # $9  <= 0x00000001
lui $t2,1             # $10 <= 0x00010000
ori $t3,$0,0xffff     # $11 <= 0x0000ffff
lui $t4,0xffff        # $12 <= 0xffff0000
beq $t1,$t2,end
nop
addu $s1,$t1,$t2      # $17 <= 0x00010001
addu $s2,$t2,$t4      # $18 <= 0x00000000
subu $s3,$t2,$t1      # $19 <= 0x0000ffff
subu $s4,$t1,$t3      # $20 <= 0xffff0002
ori $t6,$0,4          # $14 <= 0x00000004
sw $s3,0($t6)         # *00000004 <= 0x0000ffff
sw $s4,4($t6)         # *00000008 <= 0xffff0002
ori $t5,$0,8          # $13 <= 0x00000008
lw $s5,0($t5)         # $21 <= 0xffff0002
addu $t7,$t6,$0       # $15 <= 0x00000004
addu $t8,$t1,$0       # $24 <= 0x00000001
begin:
addu $t7,$t7,$t1      # $15 <= $15 +1
beq $t7,$t5,end       # if ($15 == 0x00000008) jump to end
addu $t8,$t8,$t8      # $24 <= $24 + $24
beq $0,$0,begin       # jump to begin
end:
nop
```

运行结果：

```
This is a Full version of ISim.
Time resolution is 1 ps
Simulator is doing circuit initialization process.
Finished circuit initialization process.
@00003000: $ 9 <= 00000001
@00003004: $10 <= 00010000
@00003008: $11 <= 0000ffff
@0000300c: $12 <= ffff0000
@00003018: $17 <= 00010001
@0000301c: $18 <= 00000000
@00003020: $19 <= 0000ffff
@00003024: $20 <= ffff0002
@00003028: $14 <= 00000004
@0000302c: *00000004 <= 0000ffff
@00003030: *00000008 <= ffff0002
@00003034: $13 <= 00000008
@00003038: $21 <= ffff0002
@0000303c: $15 <= 00000004
@00003040: $24 <= 00000001
@00003044: $15 <= 00000005
@0000304c: $24 <= 00000002
@00003044: $15 <= 00000006
@0000304c: $24 <= 00000004
@00003044: $15 <= 00000007
@0000304c: $24 <= 00000008
@00003044: $15 <= 00000008
ISim>
```

样例 2：　　期望结果：

```
ori $sp,$0,0x00002ffc   # $29 <= 0x00002ffc
ori $s0,$0,10           # $16 <= 0x0000000a
ori $t1,$0,1            # $9  <= 0x00000001
ori $t2,$0,8            # $10 <= 0x00000008
ori $a0,$0,10           # $4  <= 0x0000000a
jal ans                 # $31 <= pc+4 and jump and link to ans
sw $v0,0($0)            # *00000000 <= 0x00000037
ori $s1,$0,55           # $17 <= 0x00000037
beq $v0,$s1,end         # jump to end
ans:
beq $a0,$t1,if_end      # if ($4  == 0x00000001) jump to if_end
subu $sp,$sp,$t2        # $29 <= $29 - 8
sw $a0,0($sp)           # *($29) <= $4
sw $ra,4($sp)           # *($29+4) <= $31
subu $a0,$a0,$t1        # $4  <= $4 - 1
jal ans                 # $31 <= pc + 4 and junp and link to ans
lw $a0,0($sp)           # $4  <= *($29)
lw $ra,4($sp)           # $31 <= *($29+4)
addu $sp,$sp,$t2        # $29 <= $29 + 8
addu $v0,$v0,$a0        # $2  <= $2 + $4
jr $ra                  # return
if_end:
ori $v0,$0,1            # $2  <= 0x00000001
jr $ra                  # return
```

```
end:
Nop
```

运行结果：

```
Simulator is doing circuit initialization process.
Finished circuit initialization process.
@00003000: $29 <= 00002ffc
@00003004: $16 <= 0000000a
@00003008: $ 9 <= 00000001
@0000300c: $10 <= 00000008
@00003010: $ 4 <= 0000000a
@00003014: $31 <= 00003018
@00003028: $29 <= 00002ff4
@0000302c: *00002ff4 <= 0000000a
@00003030: *00002ff8 <= 00003018
@00003034: $ 4 <= 00000009
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fec
@0000302c: *00002fec <= 00000009
@00003030: *00002ff0 <= 0000303c
@00003034: $ 4 <= 00000008
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fe4
@0000302c: *00002fe4 <= 00000008
@00003030: *00002fe8 <= 0000303c
@00003034: $ 4 <= 00000007
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fdc
@0000302c: *00002fdc <= 00000007
@00003030: *00002fe0 <= 0000303c
@00003034: $ 4 <= 00000006
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fd4
@0000302c: *00002fd4 <= 00000006
@00003030: *00002fd8 <= 0000303c
@00003034: $ 4 <= 00000005
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fcc
@0000302c: *00002fcc <= 00000005
@00003030: *00002fd0 <= 0000303c
@00003034: $ 4 <= 00000004
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fc4
@0000302c: *00002fc4 <= 00000004
@00003030: *00002fc8 <= 0000303c
@00003034: $ 4 <= 00000003
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fbc
```

```
@0000302c: *00002fbc <= 00000003
@00003030: *00002fc0 <= 0000303c
@00003034: $ 4 <= 00000002
@00003038: $31 <= 0000303c
@00003028: $29 <= 00002fb4
@0000302c: *00002fb4 <= 00000002
@00003030: *00002fb8 <= 0000303c
@00003034: $ 4 <= 00000001
@00003038: $31 <= 0000303c
@00003050: $ 2 <= 00000001
@0000303c: $ 4 <= 00000002
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fbc
@00003048: $ 2 <= 00000003
@0000303c: $ 4 <= 00000003
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fc4
@00003048: $ 2 <= 00000006
@0000303c: $ 4 <= 00000004
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fcc
@00003048: $ 2 <= 0000000a
@0000303c: $ 4 <= 00000005
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fd4
@00003048: $ 2 <= 0000000f
@0000303c: $ 4 <= 00000006
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fdc
@00003048: $ 2 <= 00000015
@0000303c: $ 4 <= 00000007
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fe4
@00003048: $ 2 <= 0000001c
@0000303c: $ 4 <= 00000008
@00003040: $31 <= 0000303c
@00003044: $29 <= 00002fec
@00003048: $ 2 <= 00000024
@0000303c: $ 4 <= 00000009
@00003040: $31 <= 0000303c
ISim>
# run 1.00us
@00003044: $29 <= 00002ff4
@00003048: $ 2 <= 0000002d
@0000303c: $ 4 <= 0000000a
@00003040: $31 <= 00003018

@00003044: $29 <= 00002ffc
@00003048: $ 2 <= 00000037
@00003018: *00000000 <= 00000037
@0000301c: $17 <= 00000037
ISim>
```

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 采用的是字节寻址的方式，每个字的第一个字节的地址即为其字地址，因此写入字时 addr 总为 4 的整数倍，其末两位可以省略，又因此设计中 DM 要求为 1024 字，则地址应为[11:2]。此 addr 信号由 ALU 计算得来。

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 会复位 PC，GRF 与 DM，其中 PC 复位至指令地址初值 0x00003000，GRF 中所有寄存器清零，DM 全部清零，旨在使整个系统回到开始工作前的状态。

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）采用 always 和 case 语句相结合的方式

代码示例：

```
always @( * ) begin
    case （opcode）
        6’b000000:
        ......
    endcase
end
```

（2）采用 assign 语句

代码示例：

```
wire addu,subu,......
assign addu = op[0] && ......
......
```

（3）利用宏定义

代码示例：

```
`define addu 6’b100001
......
```

4. 根据你所列举的编码方式，说明他们的优缺点。

第一种方式代码冗长且不清晰，指令与控制信号的对应关系不够直观，但是增加新的指令时不需要改动原有代码。第二种方式可以直接由真值表生成，但是代码不够直观，容易写错看错。第三种方式综合了前两种方式的优点，且可以跨模块使用。

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。

在忽略溢出的前提下，addi 和 addiu 都是将立即数符号拓展至 32 位后与 rs 寄存器中值相加并将结果后 32 位存入 rt 寄存器中，二者等价。Add 与 addu 都是将 rs 寄存器与 rt 寄存器中值相加并将结果的后 32 位存入 rd 寄存器，二者等价。

6. 根据自己的设计说明单周期处理器的优缺点。

优点：单周期处理器设计与结构较为简单

缺点：由于统一时钟周期的缘故，所有指令的运行时间均以最长时间为准，造成处理器执行指令较慢。

7. 简要说明 jal、jr 和堆栈的关系。

Jal 与 jr 指令配套使用，jal 用于调用并链接函数，将下一条指令的地址存入$ra（$31）中，函数执行完毕后通过 jr 指令返回，将 PC 置为$ra 的值，以此完成函数的调用与返回。栈用来存储 GRF 无法保存或冲突的局部变量，通过栈顶指针$sp 及偏移量访问。在函数调用前后，需要根据需要将需要保护的寄存器的值写入与读出，以正确运行程序。