CPU设计文档

基本部件

IFU（取指令单元，从存储指令的 ROM 模块中取出下一条指令的32位二进制码）

- 内部包括 **PC**（程序计数器）、**IM**（指令存储器）及相关逻辑。
- PC 用**寄存器**实现，应具有**异步复位**功能，复位值为起始地址。
- **起始地址：0x00003000。**
- 地址范围：0x00003000 ~ 0x00006FFF。
- IM 用 **ROM** 实现，容量为 4096 × 32bit。
- ROM 内部的起始地址是从 0 开始的，即 ROM 的 0 位置存储的是 PC 为 0x00003000 的指令，每条指令是一个 32bit 常数。
- 经过以上分析，不难发现 ROM 实际地址宽度仅需 12 位，请使用恰当的方法**将 PC 中储存的地址同 IM 联系起来。**

Splitter（将每条指令的 32 位二进制码分成分别表示寄存器号、funct 码等的二进制码段）

GRF（通用寄存器组，也称为寄存器文件、寄存器堆）

- 用具有写使能的寄存器实现，寄存器总数为 32 个，应具有**异步复位**功能。
- **0 号寄存器**的值始终保持为 0。其他寄存器**初始值（复位后）均为 0**，无需专门设置

ALU（算术逻辑单元，实现指令需要的数学运算）

- 提供 32 位加、减、或运算及大小比较功能。
- 加减法按无符号处理（不考虑溢出）。

DM（数据存储器）

- 使用 RAM 实现，容量为 3072 × 32bit，应具有**异步复位**功能，复位值为 0x00000000。
- **起始地址：0x00000000。**
- 地址范围：0x00000000 ~ 0x00002FFF。
- RAM 应使用双端口模式，即设置 RAM 的 **Data Interface** 属性为 **Separate load and store ports。**

EXT（扩展单元，根据需要进行相应的位扩展）

- 可以使用 Logisim 内置的 Bit Extender

Controller（控制器，根据 splitter 得到的 6 位 funct 码和 6 位 instr_index 码确定指令的类型并输出对应的控制信号）

- 控制器的设计，从最基本的层面来说，是一个**译码**的过程，将每一条机器指令中包含的信息，转化为给 CPU 各部分的控制信号
- 解码逻辑分解为**和逻辑**和**或逻辑**两部分
  - 和逻辑
    - 功能是**识别**，将输入的机器码识别为相应的指令；
    - 在“和逻辑”的这一部分电路中，我们根据 op1 到 op6 这 6 个信号，通过将恰当的将原信号和通过非门后的信号连接到同一个**与门**中，来使得“输入指令所对应的与门”能够产生“真”，而其他的与门产生“假”值。 —— 因为我们通过是哪个与门输出真来判断输入的是哪种指令，而同一时刻我们只可能输入同一种指令。
  - 或逻辑
    - 功能是**生成**，根据输入的指令的不同，产生不同的控制信号。
    - 在“或逻辑”的这一部分电路中，我们已经拥有了与门的输出结果，现在只需要考虑：当输入了某种指令，需要使哪些控制信号输出“真”？ 由于同一时刻有且只可能有一个与门输出“真”，只需要将每个“指令对应的与门输出”连接到“该指令所产生的控制信号的或门输入”即可。
- 指令的opcode和funct

funct	**100000**	**100010**	**---------**	**---------**	**---------**	**---------**	**---------**	**----------**
opcode	000000	000000	001101	100011	101011	000100	001111	----------
	add	sub	ori	lw	sw	beq	lui	nop
Wreg_sel (RegDst)	1	1	0	0	0	x	1	x
Wdata_sel (MemtoReg)	00	00	00	01	01	x	10	x
ALUin_sel (ALUsrc)	0	0	1	1	1	0	x	x
ALUop_sel (ALUOp)	10	10	11	00	00	01	x	x
l/s_sel (MemWrite)	0	0	0	0	1	0	0	x
W_en (RegWrite)	1	1	1	1	0	0	1	0
DM_en	0	0	0	1	1	0	0	0
Branch(Branch)	0	0	0	0	0	1	0	x
EXT_sel	x	x	1	0	0	0	0	x
Shift_sel	x	x	x	x	x	0	1	x

  - ALUOP编码表

ALUOP	含义
00	加法
01	减法
10	依赖于func
11	或

- ALU译码器

ALUop	funct	ALU_sel
00	x	010(加)
01	x	110(减)
10	100000	010(加)
10	100010	110(减)
10	100100	000(与)
10	100101	001(或)
10	101010	111(小于置位)
11	x	001(或)

- Wdata

Wdata_sel	对应
00	ALU
01	DM
10	EXT

结构设计

模块

- IFU:指令处理
  - Adder
  - PC
  - IM
- GRF：寄存器相关
  - Reg1,Reg2
  - Wreg
  - Wdata
- ALU：运算

- DM：内存
- EXT：位扩展
- Shift：移位
- Nadd：分支跳转加法器
- Controller：控制器

指令	Adder		PC	IM	GRF				ALU		DM		EXT(Sign-ext)	Shift	Nadd	
	A	B			Reg1	Reg2	Wreg	Wdata	A	B	Add	Wdata			A	B
R型指令	PC	4	Adder	PC	Rs	Rt	Rd	ALU	Rdata1	Rdata2						
Ori	PC	4	Adder	PC	Rs		Rt	ALU	Rdata1	EXT			imm16			
Lw	PC	4	Adder	PC	Rs		Rt	DM	Rdata1	Sign-ext	ALU		imm16			
Sw	PC	4	Adder	PC	RS	Rt			Rdata1	Sign-ext	ALU	Rdata2	imm16			
Beq	PC	4	Adder/Nadd	PC	Rs	Rt			Rdata1	Rdata2			imm16	EXT	Adder	Shift
Lui	PC	4	Adder	PC			Rt	Shift					imm16	EXT		

结构图

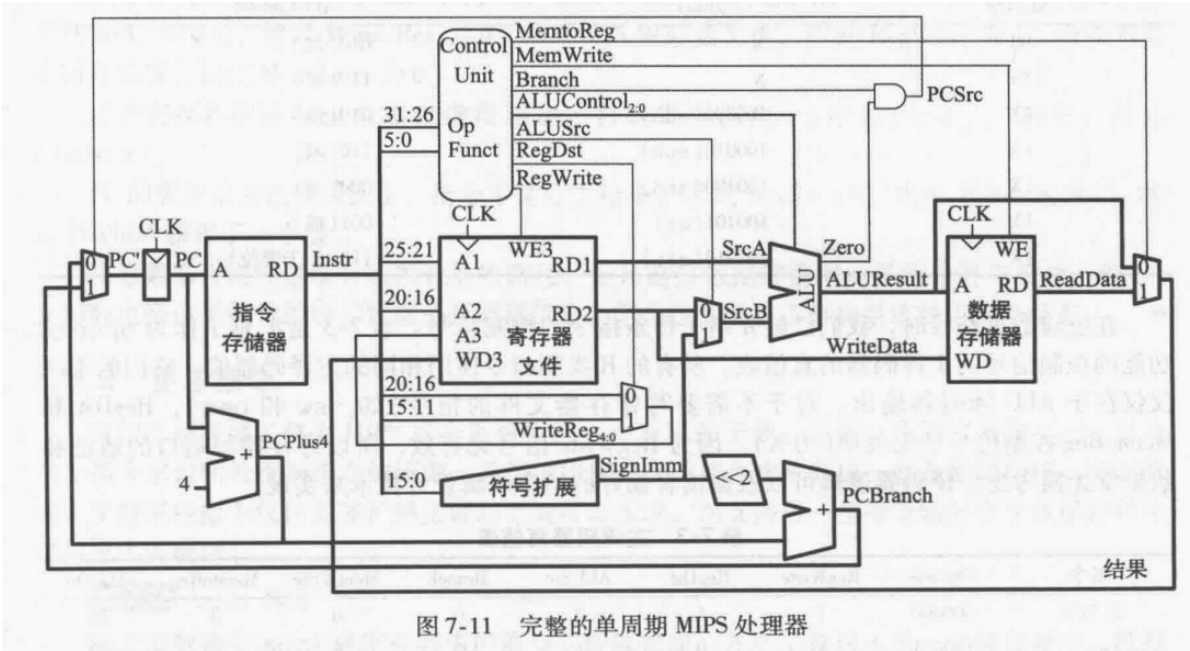

图 7-11　完整的单周期 MIPS 处理器

测试方案

#初始化所有寄存器
ori $0,$0,0
ori $1,$0,1
ori $2,$0,2
ori $3,$0,3
ori $4,$0,4
ori $5,$0,5
ori $6,$0,6
ori $7,$0,7
ori $8,$0,8

```
ori $9,$0,9
ori $10,$0,10
ori $11,$0,11
ori $12,$0,12
ori $13,$0,13
ori $14,$0,14
ori $15,$0,15
ori $16,$0,16
ori $17,$0,17
ori $18,$0,18
ori $19,$0,19
ori $20,$0,20
ori $21,$0,21
ori $22,$0,22
ori $23,$0,23
ori $24,$0,24
ori $25,$0,25
ori $26,$0,26
ori $27,$0,27
ori $28,$0,28
ori $29,$0,29
ori $30,$0,30
ori $31,$0,31

#存储指针
ori $t0,$0,0
ori $s0,$0,4

ori $a0, $0, 123
sw $a0, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

ori $a1, $a0, 456
sw $a1, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

ori $0, $0, 123        #测试目标寄存器是$0的情况
sw $0, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

ori $t1, $0, 0         #0 及附近的数
sw $t1, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

ori $t1, $0, 1
sw $t1, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

#ori $t1, $0, -1
#sw $t1,52($0)
ori $t1, $0, 65533 #16位数边界附近的数
sw $t1, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针
```

```
ori $t1, $0, 25779
sw $t1, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

lui $a2, 123          # 符号位为 0
sw $a2, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0     # 维护 index 指针

lui $a3, 0xffff        # 符号位为 1
sw $a3, 0($t0)      # 将 result 保存到内存位置
add $t0, $t0, $s0    # 维护 index 指针

ori $a3, $a3, 0xffff    # $a3 = -1
sw $a3, 0($t0)
add $t0, $t0, $s0

add $s1, $a0, $a2     # 正正
add $s2, $a0, $a3     # 正负
add $s3, $a3, $a3     # 负负
sub $s1, $a0, $a2     # 正正
sub $s2, $a0, $a3     # 正负
sub $s3, $a3, $a3     # 负负
nop

sw $a0, 0($t0)
sw $a1, 4($t0)
sw $a2, 8($t0)
sw $a3, 12($t0)
sw $s1, 16($t0)
sw $s2, 20($t0)
sw $s3, 24($t0)
lw $a0, 0($t0)
lw $a1, 12($t0)
sw $a0, 28($t0)
sw $a1, 32($t0)

ori $s0,$0,32
add $t0,$s0,$t0
ori $s0,$0,4

add $t0,$s0,$t0         #$base为正，offset为负
ori $a0, $0, 0x1234
sw $a0, -4($t0)
lui $t0, 0xffff        # 符号位为 1
ori $t0, $t0, 0xffff    # $a3 = -1
ori $a0, $0, 0x5678     #$base为负，offset为正
sw $a0, 69($t0)

ori $t0, $0, 0
ori $a0, $0, 1
ori $a1, $0, 2
ori $a2, $0, 1
```

```
beq $a0, $a1, loop1    # 不相等
beq $a0, $a2, loop2    # 相等

loop1:sw $a0, 36($t0)
loop2:sw $a1, 40($t0)

loop5:
beq $0,$0,loop5
```

思考题

1. 上面我们介绍了通过 FSM 理解单周期 CPU 的基本方法。请大家指出单周期 CPU 所用到的模块中，哪些发挥状态存储功能，哪些发挥状态转移功能。

   存储：IFU,GRF,DM

   状态转移：Controller,Splitter,ALU

2. 现在我们的模块中 IM 使用 ROM， DM 使用 RAM， GRF 使用 Register，这种做法合理吗？ 请给出分析，若有改进意见也请一并给出。

   合理的。ROM只有读的功能，能防止指令被修改；RAM可以读和写，方便对内存进行管理修改；而GRF在每条指令中都被使用，用Register速度最快，便于响应。

3. 在上述提示的模块之外，你是否在实际实现时设计了其他的模块？如果是的话，请给出介绍和设计的思路。

   还设计了移位，分支等模块，具体见上文

4. 事实上，实现 `nop` 空指令，我们并不需要将它加入控制信号真值表，为什么？

   因为nop空指令全为0，使得所有的控制信号输出都为0，（寄存器写入信号RegWrite、数据存储器读写信号MemWrite、MemToReg均为0），不会对寄存器或内存值产生影响

5. 阅读 Pre 的 “MIPS 指令集及汇编语言”一节中给出的测试样例，评价其强度（可从各个指令的覆盖情况，单一指令各种行为的覆盖情况等方面分析），并指出具体的不足之处

   较为周全，但仍有不足。

   - sub指令未测试（本次CPU的条件下）
   - 边界值测量较少
   - 存取指令时相关参数未考虑负数情况
   - 跳转指令时未测试跳转位置在指令前或者本身的情况