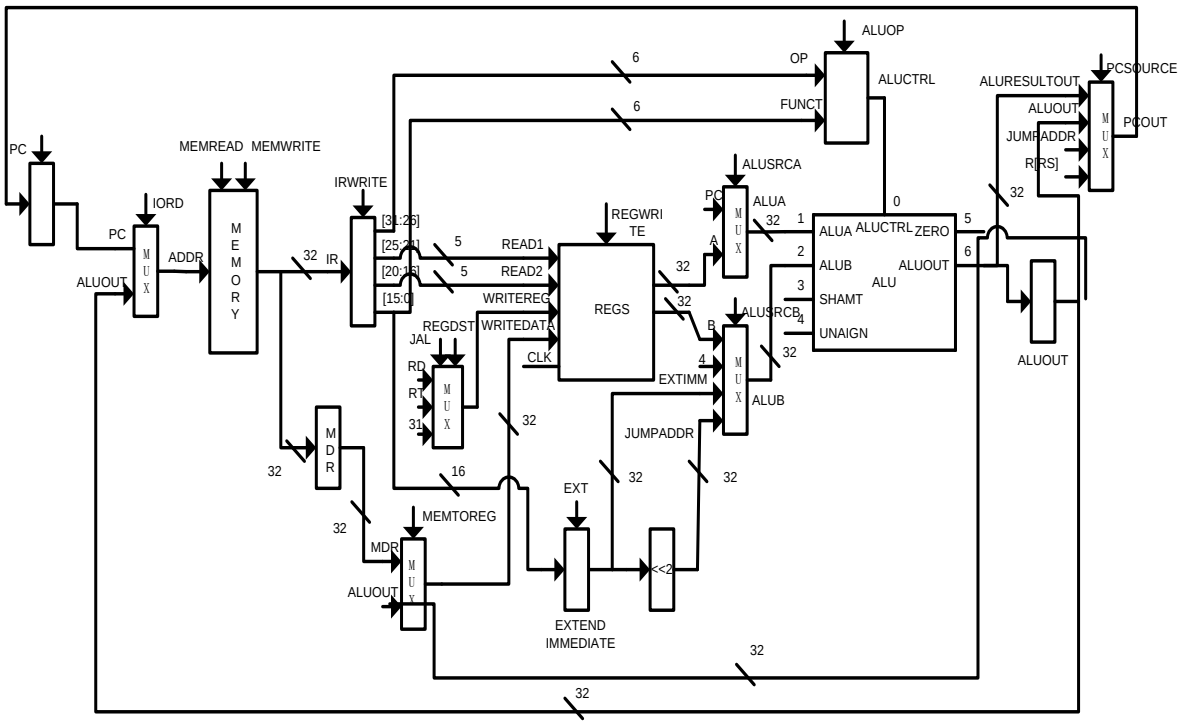
NJU CS

MIPS 运算器设计开发综合报告

第 02 组 顾天晓 陈智勇 陈丽渊 曹锋

一 系统整体结构

1.1 数据通路:



1.2 MyMIPS 指令集

1.2.1 MyMIPS 字长:

32 位

1.2.2 MyMIPS 的操作数

寄存器 32 个
存储字 2^{30} 个 (4GB)

1.2.3 MyMIPS 寻址方式:

1. 立即数寻址

2. 寄存器寻址
3. 基址寻址
4. PC 相对寻址
5. 伪直接寻址

1.2.4 指令:

1.2.4.1 算数指令和逻辑指令

add

R[RD] = R[RS]+R[RT];

0	RS	RT	RD	0	0x20

addu

R[RD] = R[RS]+R[RT];

0	RS	RT	RD	0	0x21

addi

R[RT] = R[RS]+SignExtImm;

0x8	RS	RT	ADDR/IMM

addiu

R[RT] = R[RS]+ZeroExtImm;

0x9	RS	RT	ADDR/IMM

and

R[RD] = R[RS]&R[Rt];

0	RS	RT	RD	SHAMT	0x24

andi

R[RT] = R[RS]&ZeroExtImm;

0xc	RS	RT	ADDR/IMM

nor

R[RD] = ~(R[RS] | R[RT]);

0	RS	RT	RD	SHAMT	0x27

or

R[RD] = R[RS] | R[RT];

0	RS	RT	RD	SHAMT	0x25

ori

R[RT] = R[RS] | ZeroExtImm;

0xd	RS	RT	ADDR/IMM

sll　SHAMT 字段为移位的位数，srl 亦同。

R[RD] = R[RS]<<SHAMT;

0	RS	RT	RD	SHAMT	0x00

srl

R[RD] = R[RS]>>SHAMT;

0	RS	RT	RD	SHAMT	0x02

sub

R[RD] = R[RS]-R[RT];

0	RS	RT	RD	SHAMT	0x22

subu

R[RD] = R[RS]-R[RT];

0	RS	RT	RD	SHAMT	0x23

1.2.4.2 常数乘法指令

lui

R[RT] = {IMM, 16'b0 };

0xf	RS	RT	ADDR/IMM

1.2.4.3 比较指令

slt

R[RD]= (R[RS]<R[RT])?1:0;

0	RS	RT	RD	SHAMT	0x2a

sltu

R[RD]= (R[RS]<R[RT])?1:0;

0	RS	RT	RD	SHAMT	0x2b

slti

R[RD]= (R[RS]<ZeroExtImm)?1:0;

0xa	RS	RT	ADDR/IMM

sltiu

R[RD]= (R[RS]<SignExtImm)?1:0;

0xb	RS	RT	ADDR/IMM

1.2.4.4 转移指令

beq

if(R[RS]==R[RT])PC=PC+4+BranchAddr

0x4	RS	RT	ADDR/IMM

bne

if(R[RS]!=R[RT])PC=PC+4+BranchAddr

0x5	RS	RT	ADDR/IMM

1.2.4.5 跳转指令

j

PC=JumpAddr

0x2	ADDR

jal

R[31]=PC+4;PC=JumpAddr

0x3	ADDR

jr

PC=R[RS]

0	RS	RT	RD	SHAMT	0x08

1.2.4.6 取数指令

lbu

R[RT]={24{1'b0},M[R[RS]+SignExtImm](7:0)}

0x24	RS	RT	ADDR/IMM

lhu

R[RT]={16{1'b0},M[R[RS]+SignExtImm](15:0)}

0x25	RS	RT	ADDR/IMM

lw

R[RT] = M[R[RS]+SignExtImm]

0x23	RS	RT	ADDR/IMM

1.2.4.7 保存指令

sb

M[R[RS]+SignExtImm](7:0)=R[RT](7:0)

0x28	RS	RT	ADDR/IMM

sh

M[R[RS]+SignExtImm](15:0) = R[RT](15:0)

0x29	RS	RT	ADDR/IMM

sw

M[R[RS]+SignExtImm]=R[RT]

0x2b	RS	RT	ADDR/IMM

1.3 控制信号

链接:
[各类型指令控制信号状态图]
[全部指令控制信号状态图]

1.3.1 ALUA 操作数来源

信号名	值	作用
ALUSRCA	1	ALU 第一个操作数来自寄存器 A
	0	ALU 第一个操作数是 PC

1.3.2 ALUB 操作数来源

信号名	值	作用
ALUSRCB	00	ALU 的第二个输入来自寄存器 B
	01	ALU 的第二个输入是常数 4
	10	ALU 的第二个输入是经过扩展的数
	11	ALU 的第二个输入是经过扩展的 16 位数左移 2 位的值

1.3.3 写如寄存器的地址选择信号

信号名	值	作用
REGDST	1	写入的是 R[RD]
	0	写入的是 R[RT]

1.3.4 写寄存器数据来源

信号名	值	作用
MEMTOREG	1	数据来自 MDR
	0	数据来自 ALUOUT

1.3.5 ALUOP 的控制信号

信号名	值	作用
ALUOP	00	ALU 执行加法操作
	01	ALU 执行减法操作
	10	指令的功能字段决定 ALU 操作

1.3.6 PC 来源

信号名	值	作用
PCSOURCE	00	ALU 的输出(PC+4)
	01	ALUOUT 寄存器中的值
	11	计算 JumpAddr 的部件提供的值

1.3.7 写操作相关控制信号

信号名	值	作用
REGWRITE	1	写寄存器
MEMREAD	1	读存储器
MEMWRITE	1	写存储器
IRWRITE	1	写指令寄存器
PCWRITE	1	写程序计数器
PCWRITECON	1	依据 ZERO，然后写程序计数器

1.3.8 存储器地址来源

信号名	值	作用
IORD	1	来自 ALUOUT
	0	来自 PC

1.3.9 无符号数信号

信号名	值	作用
UNSIGN	1	无符号数扩展
	0	有符号数扩展

1.3.10 特殊指令控制信号

信号名	值	作用
JR	1	使得 R 型指令在执行到第三步的时候增加一个写 PC 的操作
JAL	1	使得写如寄存器号为 31，即$ra
LUI	1	使得立即数扩充字段取得的值为{IR[15:0],{16{1'b0}}
BNE	1	使得分支指令在执行到最后时当 zero 为 0 时跳转
	0	使得分支指令在执行到最后时当 zero 为 1 时候跳转

链接:
各类型指令控制信号状态图
全部指令控制信号状态图

二 系统的设计思路和过程

包括以下内容:

1. 设计思想:
2. 用 Top-Down 方式设计系统
3. 用 Bottom-Up 方式考虑 ALU 模块的门级实现

2.1 设计思想

下面引用一张图开始我们说明的问题:

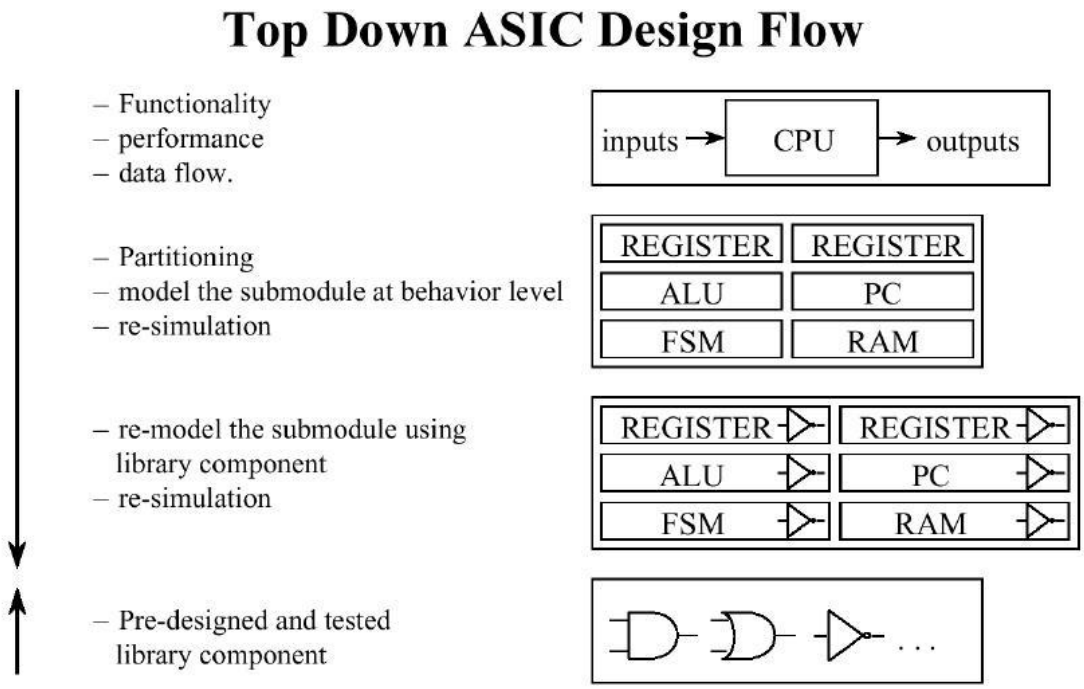


图 2.1 一般设计模式

上图中，前三步是按照 Top-Down 的方式设计的，而最后一步至第三步则是按照 Bottom-Up 的方式设计。

为什么按照这样的思想流程设计？这是由于在分析问题的时候，我们往往根据需求做一些顶层抽象描述，然后逐渐细化，当我们描述到一定抽象层次的时候，这时候我们要面对真正意义上的实现过程时，采用自底层逐渐扩充的方式，这样我们先解决简单问题然后逐渐复杂化，可以慢慢把繁琐复杂的细节问题一一实现。

2.2 Top-Down 设计系统

在这个阶段， 我们将自顶向下设计整个系统， 其中 ALU 部分我们将用行为级的描述， 因

此将我们的注意力集中在对指令和数据通路的设计上。

这个部分主要包括以下内容:

1. 指令系统的设计
2. 指令的多周期实现和控制信号的设计
3. 数据通路的设计和控制信号电路的实现

在开始图一所描述的设计步骤之前我们得先完成一些先前准备工作。

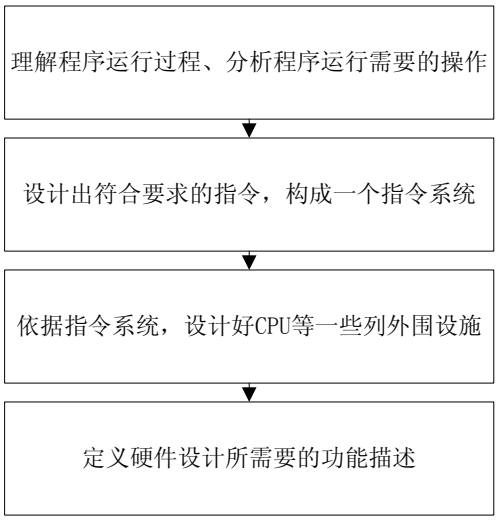


图 2.2 前期设计流程

如上图描述，实现一个系统的过程是相当复杂和漫长的，本次组成原理实验大作业的目的是完成以上完整系统的一个部分的功能。而设计实现这个部分的时候不得不考虑其他部分的协作，这有助于设计出更好的作品，更有实际运用的意义。

2.2.1 指令系统的设计:

主体思路: 参考 MIPS 指令系统和另外一个所谓的 TinyMIPS 指令系统, 设计出我们自己的指令系统 MyMIPS, 然后再由此分析得出我们的指令格式.

我们知道一个完整的运算器操作主要涉及如下几个地方，算术逻辑单元（ALU）、寄存器组（Register）、内存（RAM）、程序计数器（PC），此外还有一些控制电路，这些控制电路大部分由有限状态机（FSM）实现。

关于 CPU 的功能，我们需要首先设计好指令系统，而指令系统，则和以上提到的部件密切相关。因为有时候，一条指令的功能可能不仅仅只有一种方式实现，比如 MOVE 指令，我们可以设计有 SR 到 DR 的直接复制数据通路，也可以设计为 SR+0->DR 等等。

本次大作业的要求是实现 MIPS 中描述的部分功能，首先一点比较明确的是机器字长是 32 位的。在开始 VerilogHDL 设计之前我们先把指令系统进行分析。

2.2.1.1 MIPS 指令系统

类似于软件工程的需求分析，我们对于一个 CPU 的功能进行类似的分析。

CPU 的功能是完成我们设计好的每一条指令，而我们设计好的指令和 CPU 的设计密切相关。

首先引入完整的 MIPS 的指令操作码位图，如下所示：

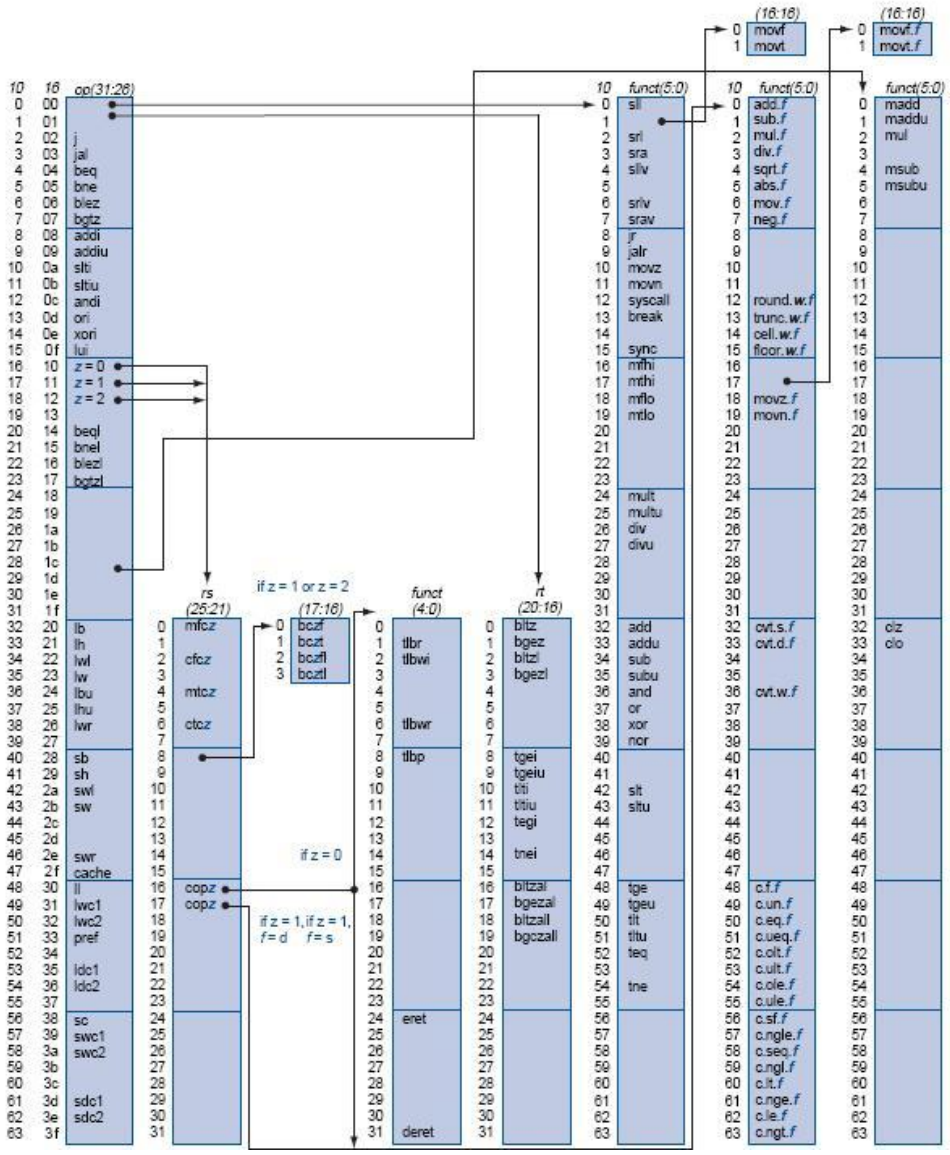


图 2.3　MIPS 的指令操作码位图

关于指令系统的设计，这需要引入更多的知识，在此我们仅仅是引用 MIPS 来帮助我们设计 CPU。所以，我们整个设计活动是建立在一个已经设计好的指令系统下开始的。如果想要往上追溯，由于与本课程关系不紧密，在此忽略。

MIPS 是一个庞大的指令系统，我们可以选择一些具有代表性的指令出来首先完成其实现，初步实现 CPU 的框架，而其余指令则会在我们完成初步设计之后，进行讨论决定实现

还是不予以实现。

2.2.1.2 TinyMIPS 指令系统：

Move	ALU	Jump
lui	addu	jump
lw	addui	jr
sw	subu	jal
mfhi	and	jalr
mthi	andi	beq
mflo	xor	bne
mtlo	xori	blez
	slt	bgtz
	sltu	bltz
	slti	bgez
	sltiu	

图 2.4　TinyMIPS

TinyMIPS 没有浮点数指令，没有乘除指令，没有 I/O 或系统调用指令，没有处理异常的指令，没有或操作而仅有异或操作，只有右移一位的操作，在内存中存取一整个字的操作。

2.2.1.3 MyMIPS 指令系统:

提出 MIPS 和 TinyMIPS 的目的是为了明确我们当前所做的工作的重要性。我们借助他们的设计思想，提出自己的 MyMIPS，然后定义出所需的硬件。

MyMIPS 提出的目的是完善硬件设施，就是不去关心指令本身的功能以及内容，仅仅关注操作那些硬件设施。

事实上，我们的 MyMIPS 实现的指令集是 MIPS 完整指令集中的核心指令集，这个指令集共有 29 条指令，关于该指令系统的具体内容如下所示：

MIPS 字长

32 位

MIPS 的操作数

寄存器 32 个

存储字 2^{30} 个

MIPS 寻址方式：

6. 立即数寻址
7. 寄存器寻址
8. 基址寻址
9. PC 相对寻址

10. 伪直接寻址

MyMIPS

1.算数指令和逻辑指令

add addu addi addiu
and andi
nor or ori
sll srl
sub subu

2.常数乘法指令

lui

3.比较指令

slt sltu slti sltiu

4.转移指令

beq
bne

5.跳转指令

j
jal
jr

6.取数指令

lbu
lhu
lw

7.保存指令

sb
sh
sw

2.2.1.4 MyMIPS 指令格式

指令的格式首先和一条指令的长度密切相关，此外还和指令系统的的指令个数有关。

1. 按照指令的类型定义指令格式

MyMIPS 指令为 32 位，一个字也是 32 位，指令分段，其中不同的指令分段不同，但是每条指令的最高六位为必有的段，为操作码字段。

指令分段，每条指令的前

6 位	5 位	5 位	5 位	5 位	6 位

R 型指令

操作码为全部为 0，分类依据是指令涉及三个寄存器，主要由 FUNCT 字段区分功能。

OP	RS	RT	RD	SHAMT	FUNCT

I 型指令

操作码不完全相同，分类依据是涉及两个寄存器和一个 16 位数的操作。这个 16 位的数可能是代表地址，也可能就是一个立即数。

OP	RS	RT	ADDR/IMM

J 型指令

OP	ADDR

PS: ALU 运算的都是 32 位数，为此需要把一些不满足 32 位的立即数字段的值扩展为 32 位。分下面三种情况：

1. SignExtImm = { 16{ immediate(15)}, immediate};
2. ZeorExtImm = { 16{ 1b'0}, immediate };
3. BranchAddr = { 14{immediate[15]}, immediate,2b'0};
4. JumpAddr = { PC[31:28], address, 2b'0 };

2. 下按上面的指令格式定义出针对每个具体指令的指令格式：

NAME/MNEMONIC: //指令的缩写名称

OPERATION(in verilog): //用 Verilog 描述功能

//下面的表格所示，指令各字段的名称，以及 OP 和 FUNCT 字段的值

FORMAT:

6 位	5 位	5 位	5 位	5 位	6 位

1.算数指令和逻辑指令

add

R[RD] = R[RS]+R[RT];

0	RS	RT	RD	0	0x20

addu

R[RD] = R[RS]+R[RT];

0	RS	RT	RD	0	0x21

addi

R[RT] = R[RS]+SignExtImm;

0x8	RS	RT	ADDR/IMM

addiu

R[RT] = R[RS]+ZeroExtImm;

0x9	RS	RT	ADDR/IMM

and

R[RD] = R[RS]&R[Rt];

0	RS	RT	RD	SHAMT	0x24

andi

R[RT] = R[RS]&ZeroExtImm;

0xc	RS	RT	ADDR/IMM

nor

R[RD] = ~(R[RS] | R[RT]);

0	RS	RT	RD	SHAMT	0x27

or

R[RD] = R[RS] | R[RT];

0	RS	RT	RD	SHAMT	0x25

ori

R[RT] = R[RS] | ZeroExtImm;

0xd	RS	RT	ADDR/IMM

sll　SHAMT 字段为移位的位数，srl 亦同。

R[RD] = R[RS]<<SHAMT;

0	RS	RT	RD	SHAMT	0x00

srl

R[RD] = R[RS]>>SHAMT;

0	RS	RT	RD	SHAMT	0x02

sub

R[RD] = R[RS]-R[RT];

0	RS	RT	RD	SHAMT	0x22

subu

R[RD] = R[RS]-R[RT];

0	RS	RT	RD	SHAMT	0x23

2.常数乘法指令

lui

R[RT] = {IMM, 16’b0 };

0xf	RS	RT	ADDR/IMM

3.比较指令

slt

R[RD]

0	RS	RT	RD	SHAMT	0x2a

sltu

0	RS	RT	RD	SHAMT	0x2b

slti

0xa	RS	RT	ADDR/IMM

sltiu

0xb	RS	RT	ADDR/IMM

4.转移指令

beq

0x4	RS	RT	ADDR/IMM

bne

0x5	RS	RT	ADDR/IMM

5.跳转指令

j

0x2	ADDR

jal

0x3	ADDR

jr

0	RS	RT	RD	SHAMT	0x08

6.取数指令

lbu

0x24	RS	RT	ADDR/IMM

lhu

0x25	RS	RT	ADDR/IMM

lw

0x23	RS	RT	ADDR/IMM

7.保存指令

sb

0x28	RS	RT	ADDR/IMM

sh

0x29	RS	RT	ADDR/IMM

sw

0x2b	RS	RT	ADDR/IMM

2.2.2 指令的多周期实现和控制信号的设计:

定义出来我们的指令系统之后，我们开始考虑到指令的实现。
但是在讨论指令的实现之前， 我们有必要先来看一下我们的实现需要的硬件和控制信号。

2.2.2.1 控制信号和硬件

首先必须的硬件清单：

- 程序运行时需要的程序计数器 PC 寄存器；
- 程序存储器，用于存放程序；
- 指令寄存器，用于存放取出的指令；
- 寄存器堆，ALU 操作数的存放；
- ALU；
- 控制电路。
- ……

以上只是能设计到这一步的时候能想到的硬件，下面的设计过程中还会继续添加硬件设施。
在下面提出的新硬件我们会用

-

符号标出。

为方便叙述，重复指令的位码如下

6 位 OP	5 位 RS	5 位 RT	5 位 RD	5 位 SHAM	6 位 FUNCT
31 26	25 21	20 16	15 11	10 6	5 0

关于控制信号大致可以分为如下几类：

1. 关于写操作或者和写地址选择的控制信号，这些信号要在一个时钟控制信号控制下改变，而且在不同的时钟周期内的值很容易影响整个系统的操作的实现。
2. 一些比较特殊的控制信号，由于对写入和写入地址的控制无关，或者只是辅助第一类控制信号的辅助信号。这些信号很容易用组合逻辑实现。

接下来， 我们来考虑指令的实现：
由于在单周期内实现不同指令的控制电路设计比较复杂，而且很多指令都有共同的操作，从设计以及描述上简便来看，我们选择了多周期 CPU。

2.2.2.2 指令的多周期实现的设计（结合部分需要的控制信号的设计）：

分析一条指令的实现如下：

1. 取指

   从存储器中取出指令存入指令寄存器之中，并计算下条指令的地址。

   IR <= MEMORY[PC];

   PC <=PC + 4;

   在这一步中我们需要访问存储器，而且需要 ALU 计算下一条 PC 的地址并且写入 PC 寄存器中。

2. 指令译码和读寄存器

   A <= REG[IR[25:21]];//即取 RS 字段为 A 操作数；

   B <= REG[IR[20:16]];//即取 RT 字段为 B 操作数；

   此时硬件清单中还需添加 A 与 B 寄存器，

   - A 与 B 寄存器。

   这是 ALU 在需要寄存器操作数时指令译码期间必做的操作，此外还有一点可以并行处理的是分支指令的跳转地址的计算，这个地址是下条指令，或者说在经历过取指后当前 PC 寄存器中的值加上 BranchAddr = { 14{immediate[15]}, immediate,2b'0}的值，这个值用于下个周期了根据 ALU 的输出判断是否把这一步的 ALUOUT 写入 PC 寄存器中。

   所以硬件还学一个寄存器用于记录 ALUOUT。

   - ALUOUT，ALU 输出寄存器。

   ALUOUT <= PC + BranchAddr = { 14{immediate[15]}, immediate,2b'0};

3. 指令的执行，存储地址的计算或分支的完成。

   也就是 ALU 运算结束，并有 ALUOUT 输出。

   有四种情况：

   a) 存储器访问

      典型指令为取字 lw，

      ALUOUT <= A + BranchAddr = { 14{immediate[15]}, immediate,2b'0}

   b) 算术逻辑指令

      典型指令太多，如 add，sub，and，

      ALUOUT <= A OP B;

   c) 分支

      分支指令为 bne 与 beq，此时分别为不等跳转和相等跳转，ALU 进行的操作是 A – B，通过 ALU 的 zero 输出判断是否跳转，bne 指令时 ALU 的 zero 为 0 时跳转，而 beq 指令时 zero 为 1 的时候跳转，这个时候我们要区分开了，详细设计后面会解决。

      需要值得一提的是，此时 ALUOUT 寄存器中的值仍是上一步计算出来的地址，因为采用了边缘触发，所以寄存器的值在下个周期内才会写入更新。而此时 ALU 的输出可能为 0 ，也可能为其他任何一个数，但这些都不重要。

   d) 跳转

      跳转与分支有很大不同是跳转无需判断，直接计算地址写入 PC 然后转向执行下条指令。

      J 指令

PC <= JumpAddr = { PC[31:28], address, 2b'0 };

JAL 指令

PC = PC +4;

PC <= JumpAddr = { PC[31:28], address, 2b'0 };

相比 J 指令 JAL 仅仅多做的操作是并行完成一个取指时候要做的把 PC+4 写入 PC 的过程，那么这个时候移位的操作就需要一个专门移两位且把 PC 的高四位拼接的部件来做了，这是为了提高效率。

- 计算 JumpAddr = { PC[31:28], address, 2b'0 }的部件

JR 比较复杂，因为我们还没有提出从寄存器堆到 PC 的数据通路，但是有个变通的做法是，此时做的操作是把 R[RS]中的值与 0 相加，然后写入 PC 中，因为 ALU 的输出有与 PC 的数据通路。

4. 存储器的访问或 R 型指令的完成
   a) 存储器的访问

   此时 ALUOUT 是上一步 ALU 计算出的地址，如果是写如存储器则为

   MEMORY[ALUOUT] <= B;

   如 sw 指令，此时 RT 也就是 B 寄存器中的值为要写入寄存器的值；

   如果是读取存储器则为

   MDR <=MEMORY[ALUOUT];

   这里使用新的 MDR 数据寄存器则是因为我们要写入不通的寄存器中，而且访问寄存器堆需要一个完整的时钟周期。

   - MDR 数据寄存器

   b) R 型指令的完成

   REG[IR[15:11]] <= ALUOUT;//即写入数据给 RD
5. 读取存储器完成

   REG[IR[20:16]] <= MDR;

分析上述指令的功能与编码，可得以下结论：

1. ALU 有两个操作数，一个为 A，另一个为 B。
2. A 操作数的来源有两个，一个为 A 寄存器中的值，另一个是在计算 PC+4 时候的 PC，用一个一位信号 ALUSRCA 来控制。

信号名	值	作用
ALUSRCA	1	ALU 第一个操作数来自寄存器 A
	0	ALU 第一个操作数是 PC

3. B 操作数的来源有 4 个，
   a) 一个为 B 寄存器中的值；
   b) 一个为常数 4，用来计算 PC+4；
   c) 一个为扩展指令中的低 16 位数得到的 32 位数，为
      i. SignExtImm = { 16{ immediate(15)}, immediate};

ii. ZeorExtImm = { 16{ 1b'0}, immediate };

d) 一个为扩展指令中低 16 位且左移 2 位后的 32 位数，为
BranchAddr = { 14{immediate[15]}, immediate,2b'0};

那么必须用一个两位信号来控制，定位 ALUSRCB

信号名	值	作用
ALUSRCB	00	ALU 的第二个输入来自寄存器 B
	01	ALU 的第二个输入是常数 4
	10	ALU 的第二个输入是经过扩展的数
	11	ALU 的第二个输入是经过扩展的 16 位数左移 2 位的值

4. 写回寄存器的时候不是写回 R[RD]就是 R[RT]，所以用一位信号就可以控制，这个信号命名为 RegDst。

信号名	值	作用
REGDST	1	写入的是 R[RD]
	0	写入的是 R[RT]

5. ALU 执行的操作大部分是加法与减法，而其他操作，都是一两条特定算术指令才会执行。而仅有的 R 型指令才会用 FUNCT 字段控制 ALU 的操作，其余的指令只需根据操作码即可确定 ALU 的操作是加还是减，所以 ALU 控制的译码分为两段，一部分是识别出是 R 型指令还是其他做加法操作(beq,jal)或是做减法操作的指令(beq,bne)，这样译码的好处是简便。

用一个两位的信号，控制这三种结果，设这个信号为 ALUOP。

信号名	值	作用
ALUOP	00	ALU 执行加法操作
	01	ALU 执行减法操作
	10	指令的功能字段决定 ALU 操作

6. PC 的来源一是 ALU 的输出(PC+4)，或者是 ALUOUT 寄存器中的值，或者为计算 JumpAddr = { PC[31:28], address, 2b'0 }的部件提供的值。
同样，用一个两位的信号，控制为 PCSOURCE

信号名	值	作用
PCSOURCE	00	ALU 的输出(PC+4)
	01	ALUOUT 寄存器中的值
	11	计算 JumpAddr 的部件提供的值

7. 此外，还需要的控制信号是寄存器堆的写(REGWRITE)、存储器的读(MEMREAD)、存储器的写(REGWRITE)、指令寄存器的写(IRWRITE)、PC 的写(PCWRITE)，以及 ZERO 输出控制 PC 的写(PCWRITECOND)。

信号名	值	作用
REGWRITE	1	写寄存器
MEMREAD	1	读存储器
MEMWRITE	1	写存储器

IRWRITE	1	写指令寄存器
PCWRITE	1	写程序计数器
PCWRITECON	1	依据 ZERO，然后写程序计数器

前面留下一个细节问题是如何同时实现 beq 与 bne，我们有如下两种选择：

a) 简化 CPU 的控制设计，前面知道，简化了 ALUOP 后，beq 与 bne 指令提供给 ALU 的命令只能是执行减法操作。那么，当指令为 beq 时，结果为 0，zero 为 1 跳转，当指令为 bne 时，结果不为 0，zero 为 0 跳转。PCWRITE 此时无能为力参与区分，因为我们设计为使能信号，这也是简化的结果，稍后分析。
这个时候我们应该把 PCWRITE 设为两个信号，分别控制为宜。

b) 简化控制信号的设计，此时要复杂化 ALU 的控制设计，即传递一个消息给 ALU，当指令为 bne 时，算出来结果不为 0 是 zero 为 1。

c) 设计一条专门的信号控制 bne 和 beq，这中想法简单易于实现。

现在解释为何要设计 PCWRITE 和 PCWRITECOND，这是因为大多数指令是不需要判断 zero 的值来决定是否写 PC，而 PCWRITECOND 要依据 zero 的值决定判断是否写入 PC，就等价于把这两个信号组合成了一个 2 位信号。大多数指令需要写入 PC，如果设计不写入 PC 的指令，理论上可以实现重复执行一条指令的功能，此外，如果想要停止循环，还可以通过另一个使能来决定是否写寄存器，这个使能还是通过 ALU 的计算来决定的。

8. 存储器单元的地址来源

一是来自 PC，取指令；一是来自 ALUOUT，取字指令。

INSTRUCTION OR DATA

信号名	值	作用
IORD	1	来自 ALUOUT
	0	来自 PC

9. 由于考虑到有符号的数，所以还要提供一位信号给符号扩展单元

信号名	值	作用
UNSIGN	1	无符号数扩展
	0	有符号数扩展

10. 其余控制信号的分析

有几条指令比较特别，比如 jr，jal，lui，bne。这些指令和其余同类型指令在大多数周期内操作控制信号一致，但有些特殊的信号仍然需要实现，比如

1. jr

jr 是跳转到 R[RS]中指向的地址，这个地址可以用 R[RS]+$zero 来实现，而在 jr 指令的机器码中，RT 与 RD 字段刚好都是零，所以$zero 的选择可以直接顺其自然的实现。跳转的地址就是 ALU 运算出来的结果。

R 型指令的第四步应该是把 ALU 运算出来的值，存入 ALUOUT，且在这个周期内把值写入 PC 中，其余的 R 型指令是不能在第四个周期写值给 PC 的。所以，我们就要在 JR 的第三个或者第四个周期内写入 PC。

2. jal

jal 和 j 有相同的状态，不同的操作是 jal 需要把 jal 这条指令的下条指令的地址写入 $ra 中。jal 的实现我们可以在 j 实现的过程中添加一步。但是除此之外，由于在 jal 指令的字段中不存在提供写入地址的字段，所以我们不得不添加一条控制信号来实现向地址为 31 号寄存器中写入值。

3. lui

lui 指令的作用是把 I 型指令的 16 位的立即数字段置于一个 32 位数的高 16 位，然后把这个值写入到 RT 指向的寄存器中。这个过程大多数和立即数的算数指令基本一致，唯一不同的是，扩展的立即数既不是 SignExtImm 也不是 ZeroExtImm。

lui 指令和 jr 指令有一个相同之处，lui 指令的 RS 字段为 0，也就是说 ALU 的 A 操作数来自$zero，也就是 0，那么只要立即数字段的扩充为所需的值就能得出结果。

所以我们加一位控制信号来使得 16 位立即数扩充的时候有三种选择。

4. bne

bne 指令的实现使得我们不得不修改 beq 指令的是实现，其实很简单，添加一位控制信号区分 beq 与 bne，然后把这个信号和 zero 取异或操作，然后再与 PCWRITECOND 相与。

2.2.2.3 各类指令需要用的功能单元分类：

指令类型	指令类型用到的功能单元				
R 型	取指令	访问寄存器	ALU	访问寄存器	
取字	取指令	访问寄存器	ALU	访问存储器	访问寄存器
存字	取指令	访问寄存器	ALU	访问存储器	
转移	取指令	访问寄存器	ALU		

2.2.3 数据通路的设计和控制信号的设计与实现

1. 首先根据上面给出的控制信号详细分析各类核心指令的执行过程中的状态改变
2. 然后给出一个总体上的数据通路的设计图，具体分析各个模块的内容
3. 最后给出一个全部控制信号的状态转换图， 我们的控制信号将根据这张图来设计实现。

链接: 各种控制信号定义

2.2.3.1 各类指令的控制信号的状态图

有了控制信号和电路之后，我们可以详细分析每条核心指令的执行过程中的状态改变。

1. R 型指令

ADDU R
AND R
ADD R
JR R
NOR R

OR	R
SLT	R
SLTU	R
SLL	R
SRL	R
SUB	R
SUBU	R

R 型指令由其 funct 段区分，格式基本都是
R[RD] = R[RS] OP R[RT];
控制状态图如下：

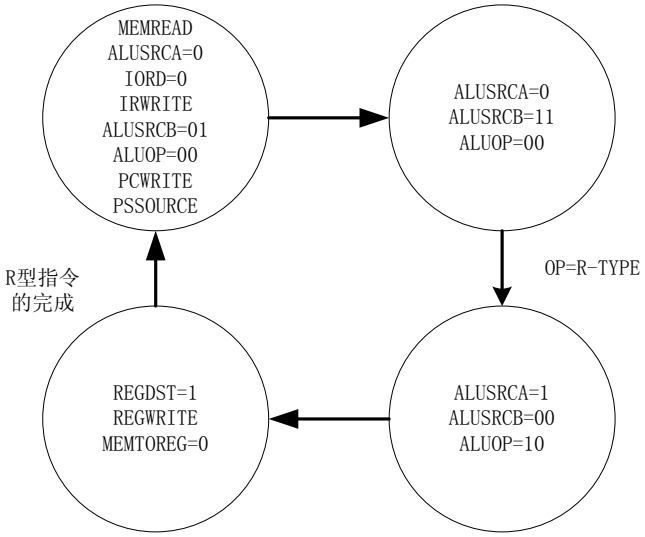


图 2.5　R 型指令控制状态图
(关于图中控制信号的定义请点击这里)

但是有几条指令需要特别说明：

a) JR
详细过程如见前面解释。

b) SLT SLTU
这两条指令从描述上来看和一般的算数逻辑运算不同，但是 ALU 设计上可以很容易实现，具体参见下面的 ALU 详细设计部分

c) SLL SRL
移位指令需要获取存储在 shamt 中的信息来决定移位的个数，如果把移位的工作交给 ALU，那么我们需要把这个信息传递给 ALU，这样 ALU 的操作数 B 看似是唯一的选择。我们可以在 B 的 mux 上在多一位选择，这样的控制信号也得增加一位。此外我们可以采用串联扩展的方法，或者在 16 位扩展单元之前加一些部件使得传递到扩展单元的为处理好的 32 位 shamt 中的信息，或者在其余三个数据入口增加选择信号。但是为什么不把移位放在立即数字段或者放在寄存器中，这是由于

i. 移位最多只需移动 32 位，为此，5 位的字段足以记录要移动的位数。
ii. 这样设计，可以减少指令执行的状态图，我们可以设计一条取 shamt 字段给 ALU 的数据通路。

但是实际上我们可以提供一个专门用来让 ALU 获取 shamt 信息的通道，这个在后卖

面的 ALU 详细设计中会体现。

2. J 型指令

J 型指令只有两条，需要做的都是计算 JumpAddr，不同的是有一项要写 PC 进$ra，状态比较简单。

J

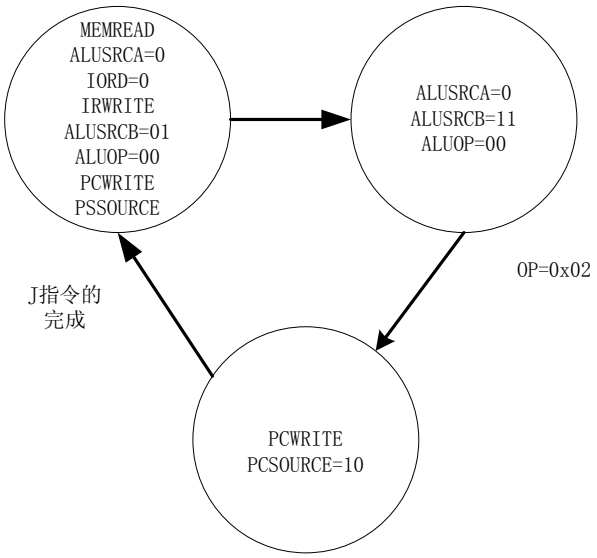


图 2.6　J 指令控制状态图

(关于图中控制信号的定义请点击这里)

JAL

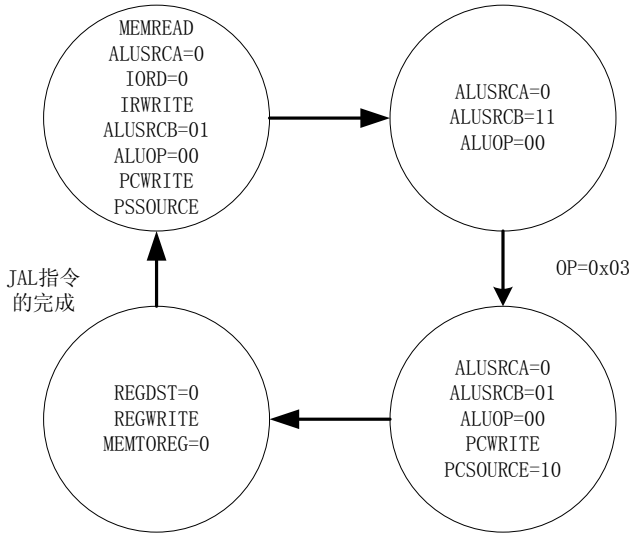


图 2.7　JAL 指令控制状态图

(关于图中控制信号的定义请点击这里)

3. I 型指令

I 型指令又可分为如下几类。

1. 立即数算数运算类

addi, addiu, andi, ori, slti, sltiu

ADDI	I
ADDIU	I
ANDI	I
ORI	I
SLTI	I
SLTIU	I

为了处理这些立即数，ALU 需要知道具体的操作，但是这些指令没有 FUNCT 段，所以前面讨论的 ALU 控制设计就必须改变了。

总结一下 ALU 的操作

加法操作

ALUOUT <= A + B;

减法操作

ALUOUT <=A – B

与操作

ALUOUT <=A & B;

或操作

ALUOUT <=A | B;

异或操作

ALUOUT <= ~(A | B);

左移操作

ALUOUT <= A<<SHAMT;

右移操作

ALUOUT <= A>> SHAMT;

比较操作

ALUOUT <=(A < B)?1:0;

JR 指令用到的，类似与 MOVE 的操作。

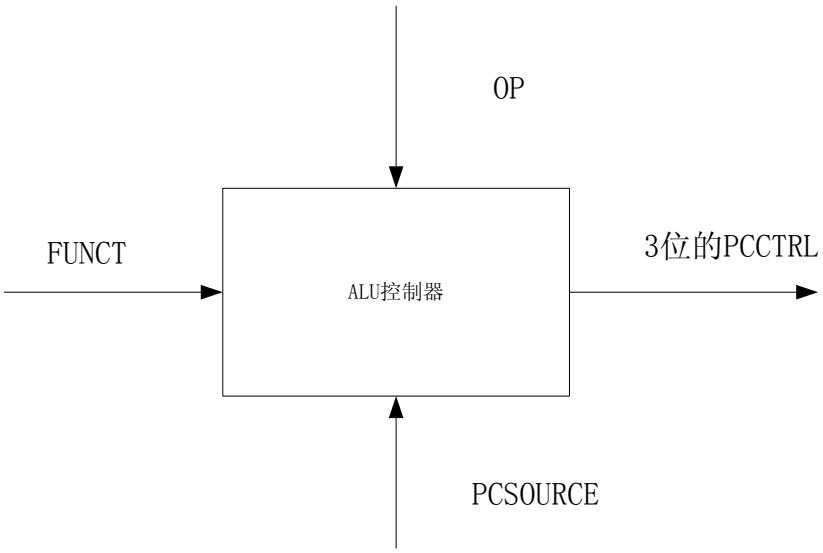
ALUOUT <= A + 0;

一共九种，需要至少 4 位控制码。我们可以发现最后一种可以添加 B 的数据来源来实现，然后只要把两种扩展的立即数合并为一种就可以了。

ALUCTRL 信号的编码

信号名	功能	值
ALUCTRL	ALUOUT <= A + B;	000
	ALUOUT <=A – B	001
	ALUOUT <=A & B;	010
	ALUOUT <=A | B;	011
	ALUOUT <= ~(A | B);	100
	ALUOUT <= A<<SHAMT;	101
	ALUOUT <=A>> SHAMT;	110
	ALUOUT <=(A < B)?1:0;	111

所以我们采用 3 位编码，其中解码要用到 6 位 OP 字段以及 6 位 FUNCT 字段。



为什么采用重复解码的控制这主要由于参与有限状态机的信号越少越容易设计，而且，我们修改 ALUOP 如下

信号名	值	作用
ALUOP	00	ALU 执行加法操作
	01	ALU 执行减法操作
	10	操作由 OP 和 FUNCT 共同决定

这样设计的原因主要因为，在一条指令的不同周期内 ALU 参与了多次计算，其中有一条在一开始就决定必须要做的就是和这条指令意义密切相关的操作，这个从指令的功能描述上可以很容易的看出，其余的都是辅助性的 ALU 操作，是动态的。
这样立即数的加减就得到处理的，接下来 ALU 还要处理无符号数和有符号数。

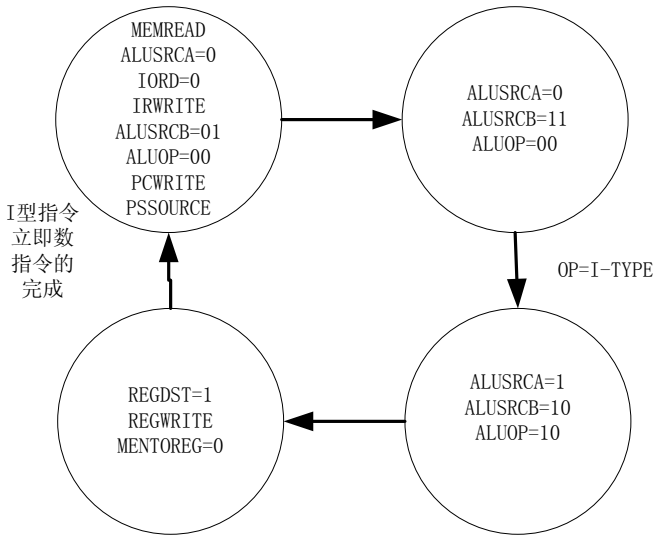


图 2.8　I 型指令控制状态图

(关于图中控制信号的定义请点击这里)

2. 分支指令

BEQ	I
BNE	I

这两条指令是前面已经讨论过了，现在就是决定如何在 zero 为 0 的时候确定 PC 的写信号。

PCWRITECOND 是显然不够用的，那么我们只好再添加一位 PCWRITECOND2，但是这无疑为有限状态机多了一位输出，增加了难度，现在还有个处理的方式，就是在 ALU 中再添加一位控制，使得 zero 输出反值。但是这样又使得 ALU 变得复杂，我们最终选择的是添加一位控制信号区分 beq 与 bne，然后把这个信号和 zero 取异或操作，然后再与 PCWRITECON 相与。

3.存储器访问指令

LBU	I
LHU	I
LUI	I
LW	I
SB	I
SH	I
SW	I

SB、SH 和 SW 的实现是在写入 Memory 的时候对 OP 进行判断，然后分别实现

LBU、LHU 与 LW 的不同之处只是在写入寄存器的时候对数据取适当的位数即可，用两位信号控制。

2.2.3.2 数据通路：

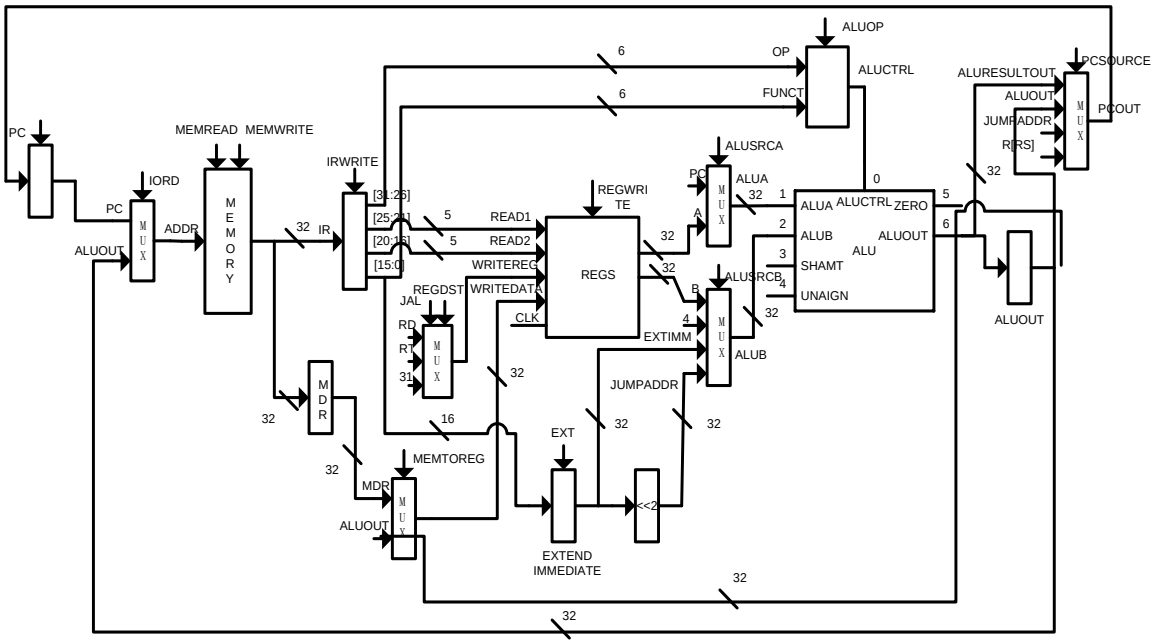


图 2.9　数据通路

各模块定义如下：

- 多路选择器 MUX

  通过控制信号选择输入源。

- 寄存器

  如 IR、MDR、A、B、ALUOUT、PC

  这些都是单个且类似的寄存器，通过时钟信号控制写入。

- 寄存器堆

  有两个读地址，分别为 READ1、READ2，这两个地址的输出数据为 READ1 DATA、READ DATA2。有一个写入地址为 WRITE，此外写入的数据输入为 WRITE DATA。

- 立即数扩展单元

  通过扩展信号确定是有符号还是无符号扩展。

输入为 immdiate	Sign=1	Sign=0
输出	SignExtImm = { 16{ immediate(15)}, immediate};	ZeorExtImm = { 16{ 1b'0}, immediate };

- 存储器取值指令控制

  分别实现的指令为 lbu，lhu，lui，lw。

输入为从存储器读取的值	输出
2'b00	取一个字节
2'b01	取半个字
2'b10	取一个字
2'b11	

- ALU 控制器

有输入 OP，FUNCT 还有 ALUOP；

输出则为 3 位的 ALUCTRL。

信号名	功能	值
ALUCTRL	ALUOUT <= A + B;	000
	ALUOUT <=A – B	001
	ALUOUT <=A & B;	010
	ALUOUT <=A | B;	011
	ALUOUT <= ~(A | B);	100
	ALUOUT <= A<<SHAMT;	101
	ALUOUT <=A>> SHAMT;	110
	ALUOUT <=(A < B)?1:0;	111

ALUOP	ALUCTRL	功能
00	000	ALUOUT <= A + B;
01	001	ALUOUT <=A – B
10	由 OP 和 FUNCT 的译码决定	

zero 是指当 ALU 结果出现 0 的时候 zero 信号输出 1 还是 0，这个主要解决 bne 与 beq 的问题。

u 是指 ALU 采用有符号运算还是无符号运算。为 1 表示无符号运算。

OP	FUNCT	ALUCTL	U
0	0x20,6'b100000	000	0
	0x21,6'b100001	000	1
	0x24,6'b100100	010	1
	0x27,6'b100111	100	1
	0x25,6'b100101	011	1
	0x2a,6'b101010	111	0
	0x2b,6'b101011	111	1
	0x00,6b000000	101	1
	0x02,6'b000010	110	1
	0x22,6'b100010	001	0
	0x23,6'b100011	001	1
0x08,6'b001000	x	000	0
0x09,6'b001001	x	000	1
0x0c,6'b001100	x	010	1
0x04,6'b000100	x	001	0
0x05,6'b000101	x	001	0
0x02	x	xxx	x
0x03,6'b000011	x	000	1
0x24,6'b100100	x	000	0
0x25,6'b100101	x	000	0
0x0f,6'b001111			
0x23,6'b100011	x	000	0

0x0d,6'b001101	x	011	1
0x0a,6'b001010	x	111	0
0x0b,6'b001011	x	111	1
0x28,6'b101000	x	000	0
0x29,6'b101001	x	000	0
0x2b,6'b101011	x	000	0

在执行 lui 指令的时候我们再一次面临是否要添加一条从寄存器组到 aluout 的数据通路，添加数据通路的后果就是需要添加至少一条控制信号。

添加信号 R2R。

- ALU

ALU 的输入有 A 与 B，移位计数器输入 SHAMT，控制信号 ALUCRL，控制信号 U。

ALU 的输出有 zero，以及给 ALUOUT 输入数据。

具体功能见上表。

- 控制信号

控制信号是最目前本实验中最难描述的部分。

状态图如下。

其中有些关键的控制信号，我们用状态机来改变，一些在指令执行过程中不参与读写控制的非关键信号我们用另一种简单的译码来实现。

比如实现 beq 与 bnq

比如实现 jr 与 lui 指令，在实现这两条指令的时候我们采用了添加数据通路的方式来实现。

不超过 16 个状态，我们用 4 位来描述不同的状态。

控制单元逻辑图如下

输出控制信号	当前状态
PCWRITE	STATE0+STATE6
PCWRITECOND	STATE8
IORD	STATE3+STATE5
MEMREAD	STATE0+STATE3
MEMWRITE	STATE5
IRWRITE	STATE0
MEMTOREG	STATE4
PCSOURCE1	STATE6
PCSOURCE0	STATE8
ALUOP1	STATE2+STATE9
ALUOP0	STATE8
ALUSRCB1	STATE1+STATE2
ALUSRCB0	STATE0+STATE1+STATE6
ALUSRCA	STATE2+STATE8+STATE9
REGWRITE	STATE4+STATE7+STATE10
REGDST	STATE10

我们用

STATEN 来表示状态 N。

NEXTSTATEN 表示输出的状态为 N。

输出	输入	条件
NEXTSTATE0	STATE4+STATE5+STATE6+STATE7+STATE8+STATE10	
NEXTSTATE1	STATE0	
NEXTSTATE2	STATE1	OP 为 I 型（除分支指令之外的）指令
NEXTSTATE3	STATE2	OP 为取值
NEXTSTATE4	STATE3	
NEXTSTATE5	STATE2	OP 为存值
NEXTSTATE6	STATE1	OP 为 J 型
NEXTSTATE7	STATE2+STATE6	为立即数加减指令为 jal 指令
NEXTSTATE8	STATE1	为条件分支
NEXTSTATE9	STATE1	OP 为 R 型
NEXTSTATE10	STATE9	
NEXTSTATE11		待定
NEXTSTATE12		待定

- 非关键信号的控制及其他

  这些信号主要有比如 beq 与 bne 指令的控制，在实现 jr 与 lui 指令的时候的信号控制等等。

  这些信号会在之后的实现中具体描述。

  还有移位指令的计数器的传递值

2.2.3.3 全部控制信号的有限状态机图

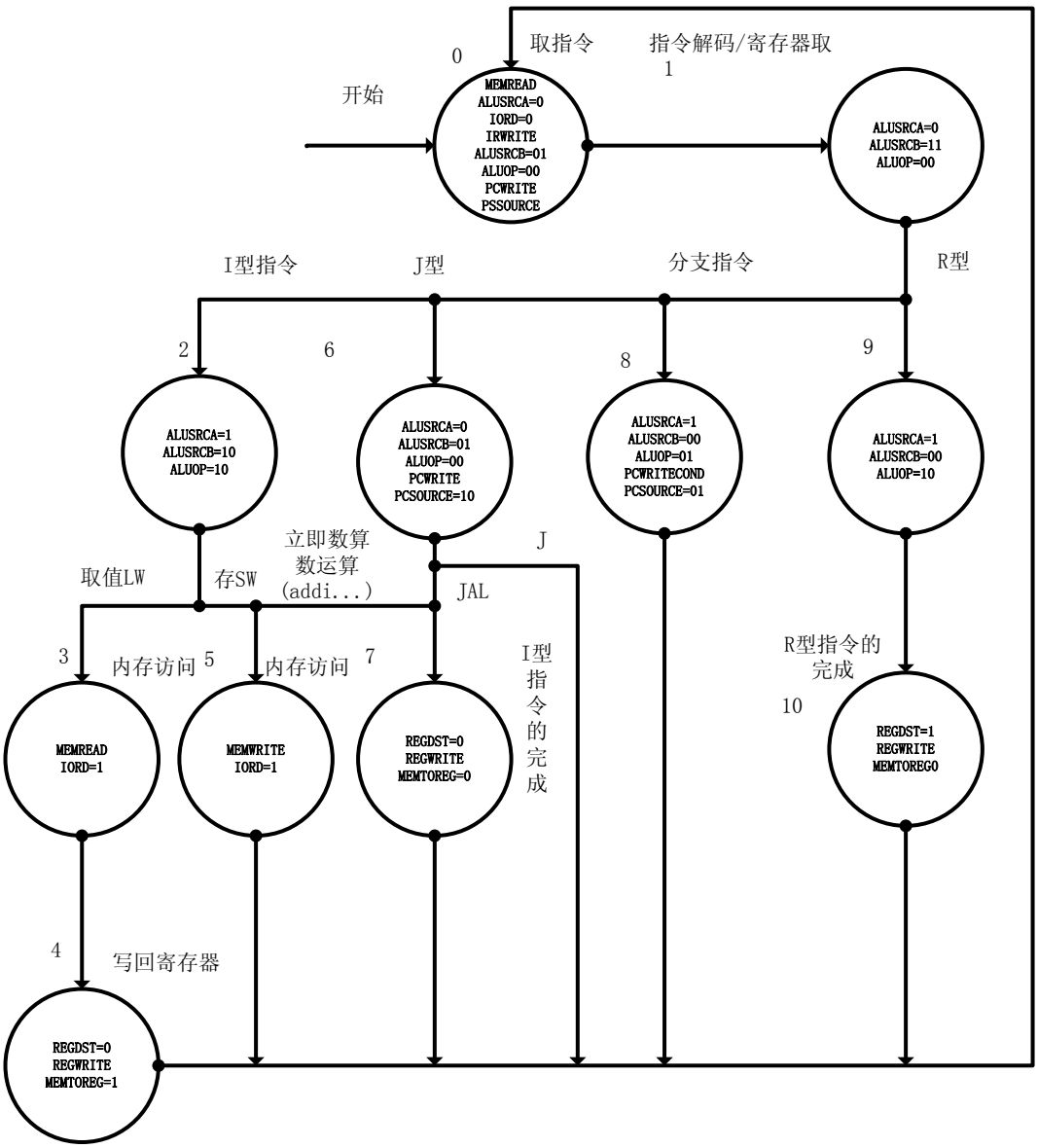


图 2.10　全部控制信号状态图

(关于图中控制信号的定义请点击这里)

2.3 Bottom-Up 从门级实现 ALU

到目前为止， 我们的精力主要集中在数据通路以及控制信号电路的实现上，却只从行为级定义了 ALU 的各种操作，因此这个阶段我们将具体从门级对 ALU 进行实现。

2.3.1 关于 ALU 处理单元的构建

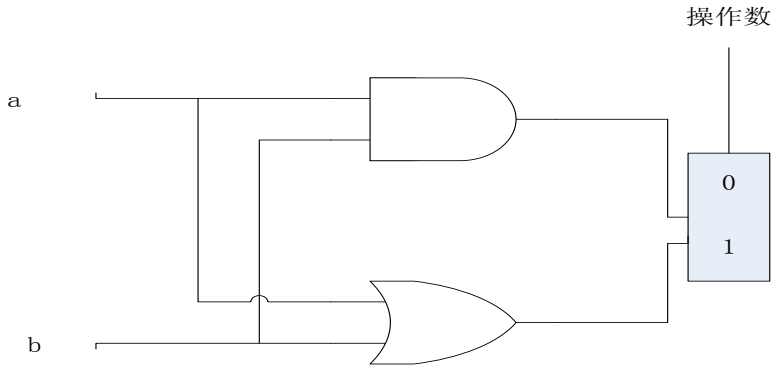
对于 ALU 的操作，我们前面定义了八种，我们把移位操作的两种和其余六种分开设计。

2.3.1.1 设计非移位部分

我们采用自下而上的设计方法：

考虑 1 位的 ALU:

逻辑操作最简单，因为它们可以直接映射到如果所示的硬件组件：



其中右边是一个二选一的选择器(mutex)。

ALU 的下一个功能是加法器。加法器必须有两个操作数输入以及一个和的 1 位输出，这里还必须有第二个输出用于进位，成为进位输出，相应的还要有进位输入。

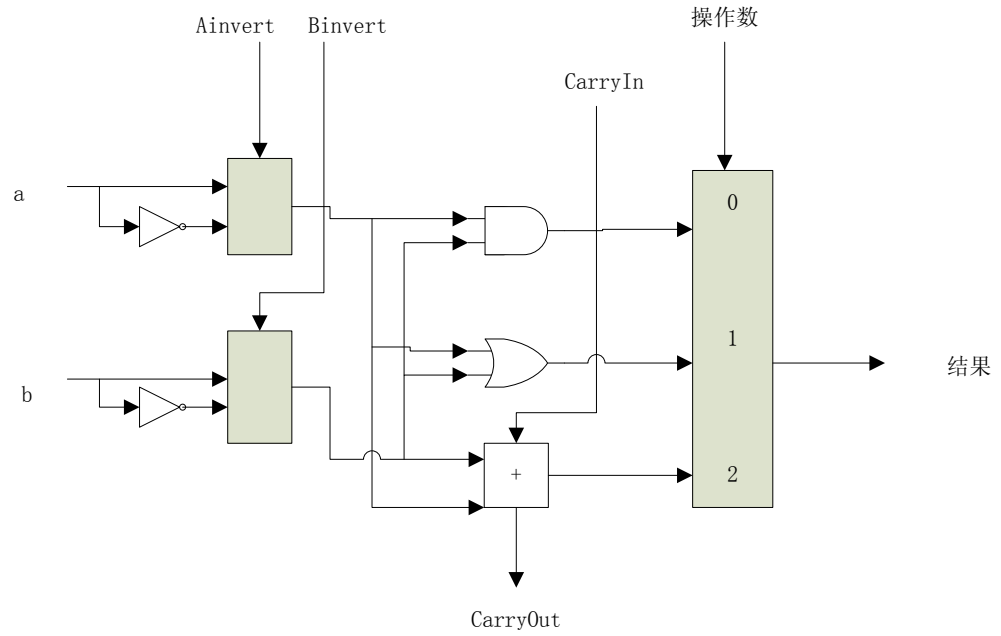
根据加法的真值表达式，我们可以得到：

$$\text{CarryOut} = \text{a} \cdot \text{b} + \text{a} \cdot \text{CarryIn} + \text{b} \cdot \text{CarryIn}$$

$$\text{Sum} = (\text{a} \cdot\sim \text{b} \cdot\sim \text{CarryIn}) + (\sim \text{a} \cdot \text{b} \cdot\sim \text{CarryIn}) + (\sim \text{a} \cdot\sim \text{b} \cdot \text{CarrIn}) + (a \cdot b \cdot CarryIn)$$

于是，加法操作完成，减法相当于加一个操作数的负数，就这样我们就可以用加法器执行减法。二进制补码取负的方法就是取反加 1，于是 b 输入也需要用一个 mutex，通过选择信号判断是 b 还是 b 的取反。巧合的是，只要我们将仅为输入 CarryIn 设置为 1，这样就实现了减法！

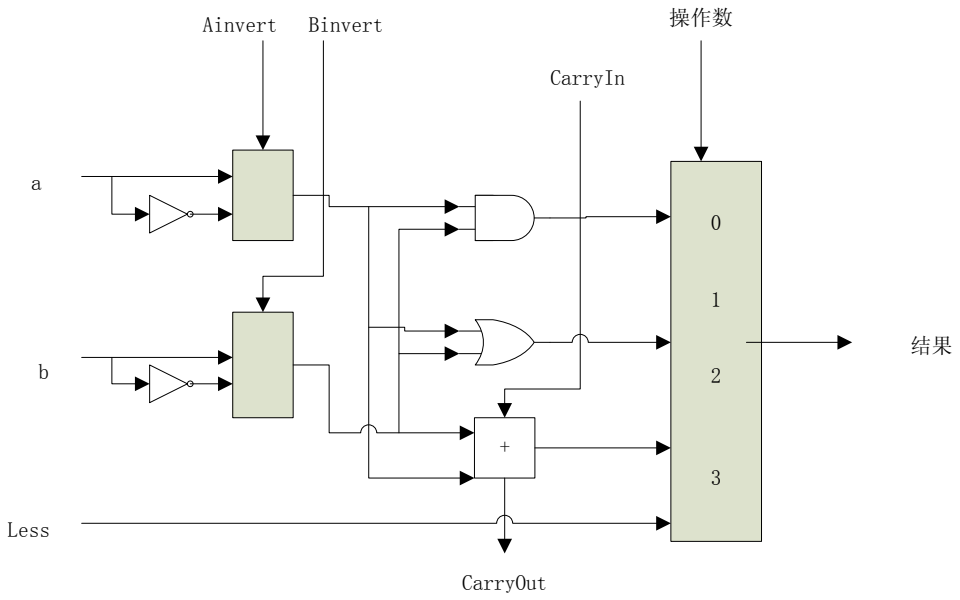
为了扩展运算，我们预先也加入关于 a 和 a 的取反的 mutex。于是有了这样的设计图：



从 1 位的 ALU 扩展到 32 位的 ALU:

我们采用直接将 1 位加法器连接起来构成所谓的行波进位加法器，级联比较简单，就是将低位加法器的进位输出作为高 1 位加法器的进位输入。

现在我们需要修改 ALU，使得它还支持另外一条指令，小于置位指令 slt，如果 rs<rt，这个操作的结果是 1，否则是 0.因此，32 位输出中，除了最低位，其他位输出都为 0，最低位根据比较的结果置位，为结果增加一个输入，专门为 slt 设计，称为 Less。于是 1 位的 ALU 扩展成为：



这里我们讨论这条指令，我们应该使用减法：a-b.首先考虑有符号数的 sltu:如果差是负数，则 a<b，进行小于比较操作时如果 a<b，最低位需要被置为 1，也就是说，如果 a-b 是负数最低位置为 1，如果 a-b 是正数最低位置为 0.需要的结果恰好与值的符号位完全相同；1 意味着负数，0 意味着正数。根据这个结果，只需要将加法器的输出的符号位（也就是最高位）连接到最低位的结果输出不是加法器的输出；对于 slt 操作的 ALU 输出显然是 Less 的输入值。我们这里命名为 set.

针对符号数的 slt,set 的方式比较复杂,比如考虑正数和负数的比较,如果还是采用上面的方式,显然会出错(正数最高位 0,负数最高位 1,减法会使最高位置为 1,正数小于负数,判断出错).

我们上网查了资料,采取一个比较统一的方法,对于最高位的 1 位 ALU,当然此时这里的 a 和 b 决定了符号位.经过整理,采取了如下对 set 的判断:

Flow=(ax^bx==0)?(ax!=AddOut)　　set=flow?(ax?1:0):AddOut

其中,AddOut 是加法器的

另外，需要注意的是每次使用 ALU 进行减法运算时，将 CarryIn 和 Binvert 都置为 1.对于加法和逻辑运算，两条控制线都置为 0.因此通过将 CarrIn 和 Binvert 合并成一条线，可以节省 ALU 的控制。

进一步要让 ALU 支持 BNE 指令，这里方法很简单，只要做减法就可以了。因此，增加硬件测试结果是否为 0，就可以测试相对性。最简单就是可以将所有 ALUout 进行或运算然后通过反相器发送信号。现在，我们就初步形成了 1 位的 ALU 和 32 位的 ALU。

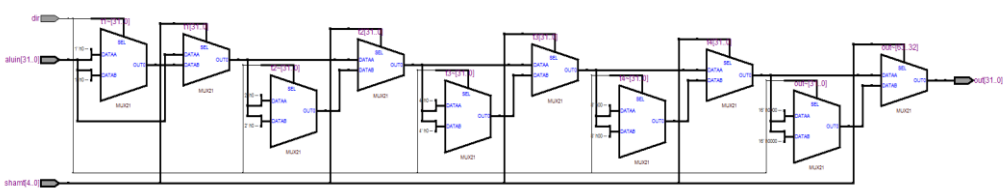
示意图另附在图片库里。

ALU 控制线：

ALU 控制线	方法
0000	与
0100	或
1000	加
1010	减
1110	小于比较设置
0011	或非

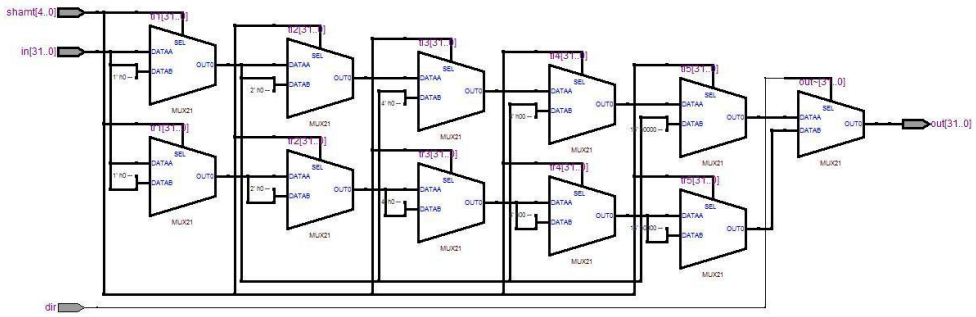
我们会根据需要再扩展这里的控制线。

2.3.1.2 移位操作的设计



移位过程可以看作是选择数据的过程，即左移一位的移与不移可以看作是选择左边一位的数据和选择当前一位的数据。同理可得移 2 位、移 4 位、移 8 位、移 16 位的设计。
一个 32 位的数，在 MIPS 指令中，移位只有移动 1~31 位有意义，移超过 31 位和移动零位的结果分别是零和数据本身。根据二进制编码的机理，一个五位的 SHAMT 从高位到低位每一位的意思分别代表移动了 16 位，移动了 8 位，移动了 4 位，移动了 2 位，移动了 1 位。把这些串联起来控制就可以得到任意 1~31 位的操作。

后来，我们对上图经过改进得到如下图。



至此， 我们的整个系统已经设计完成， 而穿插在其中的代码实现也基本完成， 下面就是对我们的整个系统的测试部分。事实上， 我们还做了一个界面和一个简易的汇编代码到机器码的翻译器来简化我们的测试过程。由于这个过程与本实验内容没有直接的关系，因此在这里省略。

三 系统测试

MIPS 指令的语法这里不多提，还有一些寻址方式前面已经提到，下面提供一段汇编程序来实现

Offset	MIPS	Machine
00000000	ADDI $S0, $ZERO, 1	20100001
00000004	ADDI $S1, $ZERO, 1	20110001
00000008	ADDI $S2, $ZERO, 100	20120064
0000000C	ADD $S0, $S0, $S1	02118020
00000010	SW $S0, 0($S2)	AE500000
00000014	ADDI $S2, $S2, 4	22520004
00000018	ADD $S1, $S0, $S1	02118820
0000001C	SW $S1, 0($S2)	AE510000
00000020	ADDI $S2, $S2, 4	22520004
00000024	J 3	08000003

这是一段计算斐波那契数列的 MIPS 汇编指令程序，左边是在内存中的地址，由于只是测试，我们使用了 PC=0 开始执行程序，中间的 MIPS 是按照格式书写的一条条指令，最右边是十六进制的机器码。

把上述机器码通过$readmemh 系统任务导入 ModelSim 中，经过若干单步运行之后，把内存中以 100 起始的数据导出部分如下：

d7: 00000000 00010100 10001010 11011101
d3: 00000000 00001100 10110010 00101000
cf: 00000000 00000111 11011000 10110101
cb: 00000000 00000100 11011001 01110011
c7: 00000000 00000010 11111111 01000010
c3: 00000000 00000001 11011010 00110001
bf: 00000000 00000001 00100101 00010001
bb: 00000000 00000000 10110101 00100000
b7: 00000000 00000000 01101111 11110001
b3: 00000000 00000000 01000101 00101111
af: 00000000 00000000 00101010 11000010
ab: 00000000 00000000 00011010 01101101
a7: 00000000 00000000 00010000 01010101
a3: 00000000 00000000 00001010 00011000
9f: 00000000 00000000 00000110 00111101
9b: 00000000 00000000 00000011 11011011
97: 00000000 00000000 00000010 01100010
93: 00000000 00000000 00000001 01111001
8f: 00000000 00000000 00000000 11101001

8b: 00000000 00000000 00000000 10010000

87: 00000000 00000000 00000000 01011001

83: 00000000 00000000 00000000 00110111

7f: 00000000 00000000 00000000 00100010

7b: 00000000 00000000 00000000 00010101

77: 00000000 00000000 00000000 00001101

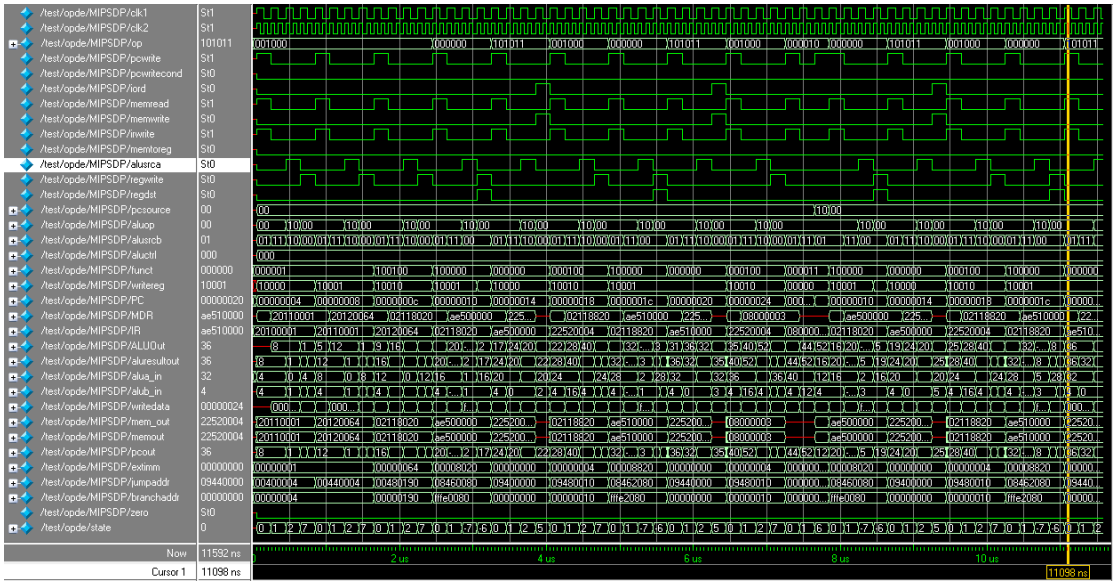
73: 00000000 00000000 00000000 00001000

6f: 00000000 00000000 00000000 00000101

6b: 00000000 00000000 00000000 00000011

67: 00000000 00000000 00000000 00000010

部分执行的波形如下：



经过对信号和数据的校对，证实，求解斐波那契数列结果正确。