计算机组成原理
实验讲义

南京大学计算机科学与技术系

张泽生

2009 年 6 月

目录

第一章 集成电路发展与 CPU 设计

1.1 集成电路的发展

在集成电路设计与生产发展到今天，人们设计数字电路系统的方法也发生了巨大的变化，以前设计数字电路均使用小规模（SSI）、中规模（MSI）电路。现在设计数字电路用大规模（LSI）或超大规模（VLSI）电路。原 Intel 公司的总裁戈登·摩尔(Gordon Moore)先生，大约在三十年前就观察到，集成电路技术正在以单个芯片上集成的晶体管数量每一年半或两年就翻一番的惊人速度发展着。这种现象，俗称摩尔定律，直到今天仍在延续。无疑，这种技术将对人们的生活各方面产生巨大的影响。对大多数的硬件产品来说，设计者有二种选择，一种是设计能在单个芯片上实现的电路，一种是设计安装在一块印制电路板(PCB)上由多个芯片实现的电路。实现这样的电路常用到的芯片主要有三种：标准芯片、可编程逻辑器件、定制芯片。下面将分别介绍这三种芯片。

1.1.1 标准芯片

我们将一些常用逻辑电路集成在某一个芯片上去实现某种逻辑功能，它们通常在功能和规格上均符合公认的标准。这类芯片称为标准芯片或称固定逻辑功能器件。为了构造逻辑电路，设计者选择能完成所需功能的芯片，然后确定这些芯片应如何连接，以实现更大规模的逻辑电路。然而标准芯片的缺点就是每个芯片功能是固定的，无法更改。

1.1.2 可编程逻辑器件

可编程逻辑器件与功能固定的标准芯片不同，内含可由用户配置的电路，可

在更大范围内实现不同的逻辑电路。这些芯片具有通用化的结构，包括一个可编程开关集合，允许用户以多种方式修改芯片内部的电路。设计者可以通过适当选择开关的配置来实现在特定应用中所需的功能。开关并不是在制造的时候就进行编程，而是由最终用户编程。这些芯片被称为可编程逻辑器件(Programmable Logic Device ,简称为 PLD)。是 20 世纪 70 年代发展起来的一种新型逻辑器件，是目前数字系统设计的主要硬件基础。目前生产和使用的 PLD 产品主要有 PROM、现场可编程逻辑阵列 FPLA（Field Programmable Logic Array）、可编程阵列逻辑 PAL（ Programmable Array Logic）、通用阵列逻辑 GAL（Generic Array Logic）、可擦除的可编程逻辑器件 EPLD（Erasable Programmable Logic Device）、复杂可编程逻辑器件 CPLD（Complex Programmable Logic Device）、现场可编程门阵列 FPGA（Field Programmable Gate Array）等几种类型。其中 CPLD、 FPGA 运用前景最被看好。这两类可编程逻辑器件的结构不同。

与 CPLD 相比，FPGA 可提供更高的逻辑密度，可以构建更多功能的电路。最新的 FPGA 器件可提供诸如内建硬核处理器、大容量存储器、时钟管理系统、专用的 DSP 等特性，并支持多种最新的高速接口技术。

与 FPGA 相比，CPLD 提供的逻辑资源少得多。但 CPLD 可预测性好，它没有上电加载问题，对于关键的控制应用非常理想。而且 CPLD 器件功耗相对较低且价格低廉，使其在成本敏感的、电池供电的便携式应用。

由于可编程逻辑器件的出现，人们设计数字电路系统的方法也发生了巨大的改变，原先都是通过设计，画出逻辑电路图，然后做 PCB 板，进行焊接调试。而现在人们用 Verilog HDL 语言或 VHDL 语言来编写硬件逻辑功能下载到可编程逻辑器件上实现。大多数型号的可编程逻辑器件都可以进行多次编程。这个功能非常有用，用户可以根据可编程逻辑器件标准成品部件所提供各种逻辑能力、速度和电压特性来选择适合自己所要的器件，在进行硬件部板设计（实验板也可以是最终的生产板）的同时，设计人员可先将要设计的功能，用 Verilog HDL 语言或 VHDL 语言编写设计出来，利用软件工具仿真和测试其设计，一旦实验板做好，就可以将设计下载到器件中，并立即在实际运行的电路中对设计进行全面测试。完全没有非重发性工程成本，最终的设计也比采用定制固定逻辑器件完成得更快。

1.1.3 定制芯片

可编程逻辑器件(PLD)是芯片生产厂家所做的定型产品，可以从各供应厂商采购到。人们可以利用 PLD 可编程性来实现数字硬件中的大多数逻辑电路。然而可编程逻辑器件不是万能的，它也有不足，一般情况下，用户电路的设计是不可能将芯片资源用尽，从而造成宝贵的芯片资源的浪费。并且构成的电路实际运行速度也受到限制。因此在某些情况下，可编程逻辑器件也许不能达到设计预期的性能或成本目标。在这种情况下，就要与芯片生产厂家合作，是选择重新设计，还是选择适当的技术，将原有的设计逻辑进行优化来制造芯片，最终生产出所需的芯片。这个过程即所谓定制设计或半定制设计，这类芯片也称为定制芯片或半定制芯片。此类芯片是专为特殊应用所生产的，也被称为专用集成电路(ASIC)。

定制芯片最大的优点在于；可以针对特定任务做最优化设计。因此常常能够达到更高的性能，而且定制芯片中有可能比其他类型芯片集成更多的逻辑电路（它没有浪费）。虽然这种芯片的生产成本很高，但是如果用在销售量大的产品中，将成本平均分摊至每个芯片，则每个芯片的成本可能低于功能相同的可编程逻辑器件供应芯片。此外，如果可以用单个芯片来替代多个芯片，则最终产品的印制电路板所需的芯片装配空间将可以进一步降低，从而降低了产品成本。

定制设计芯片的解决方案美中不足之处在于：制造定制芯片通常耗时不菲，需要数月才能完成。相比之下，如果使用可编程逻辑器件，芯片可由最终用户编程，无需耽搁制造时间。

在 CPU 的设计中同样可以用这三种芯片来设计，用第一种芯片设计，既使用现有固定逻辑芯片来搭建自己的电路，它需要较大的印制线路板，而且一旦设计成功就不易更改，若要更改，需要花费更多的时间和金钱。用第二种芯片设计，它所需要的印制线路板较小，设计成功后，一旦发现问题可随时进行修改，设计时间和成本较低，在大批量生产时，成本相对第三种芯片设计较高。用第三种芯片设计，它实用于大批量的生产，所需要的印制线路板较小，但设计成功后，不易修改、设计时间较长、大批量时成本低。在现代的 CPU 设计中，一般在初始设计时多数采用 FPGA 来设计，一旦定型后，可通过生产产家根据在 FPGA 芯片上的设计移植到定制的芯片上，进行芯片的大批量生产。这种设计方法既节省了设计时间，又为大批量生产节省了资金。

1.2 简述如何进行 CPU 设计

在计算机设计中，CPU 的设计是十分重要的部分，但如何设计是摆在每一个设计者面前着重要考虑的问题。一般来讲，要设计 CPU，首先要确定是设计通用 CPU，还是设计专用 CPU。一旦被确定，接下来就要考虑是要 CPU 与编译软件除指令编译外相对独立（软硬件可相对独立设计，软件在 CPU 上运行效率低一些），还是要编译软件除指令编译外，还要协助完成 CPU 对指令在运行中的一些优化工作（软件在 CPU 上运行效率提高，软硬件设计相互牵制较大）。Intel 8086CPU 属于前者，MIPS CPU 属于后者。指令系统的设计，可根据实际任务的要求来选择所需的指令，确定使用何种指令结构、指令格式和操作码。确定后，再对每一条指令进行仔细分析，把每条指令执行过程分成若干个状态或段，把所有指令的状态或段进行综合，并归纳成最终需要的若干个状态或段，以便后面的设计。

在单周期 CPU 设计中，因是单周期设计，所以不考虑把每条指令分成若干个状态或段，而只关注在 1 个时钟周期内能不能完成就可以了。画出带控制信号的单周期数据通路图，列出指令译码表，画出指令译码图，并与带控制信号的单周期数据通路图的信号对应相连。单周期 CPU 在实际的计算机设计中基本不使用。

在多周期 CPU 设计中，首先用把所有指令的状态进行综合，并归纳成最终需要的若干个状态，画出指令流程的状态图（这里我们给出一个实际状态图的例子见图 1.2.1），根据状态图设计状态机。然后画出带控制信号的多周期数据通路图，列出指令译码表，最后设计控制译码器，将控制译码器信号与多周期数据通路图上的控制信号对应相连。注意在多周期 CPU 设计中，有可能会出现不同状态下，做的是同一种操作。

在流水线CUP设计中，首先对每一条指令进行仔细分析，把每条指令执行过程分成若干个段，把所有指令的段进行综合，并归纳成最终需要的若干个段。例如：

取指(IF)：从指令存储器取指令并进行顺序 PC 下地址计算

取数和译码（ID)：寄存器取数，同时对指令进行译码

执行（EX)：指令执行或计算内存单元地址

读写存储器(MEM)：从数据存储器中读取数据或将数据存入存储器中

写寄存器（WB)：将数据写到寄存器中

由于有五个段，它可以构成一个五级指令流水线。如再把某些段分细，它可以构成一个更多级指令流水线。当然指令流水线级数也不是越多越好，它与硬件成本和时钟速度有关，需综合考虑。

根据若干个段画出带控制信号的若干级指令流水线数据通路图(包括流水线寄存器)，列出指令译码表，画出指令译码电路图，列出冒险检测信号表，画出冒险检测单元电路图，按实际需要画出转发单元电路图。

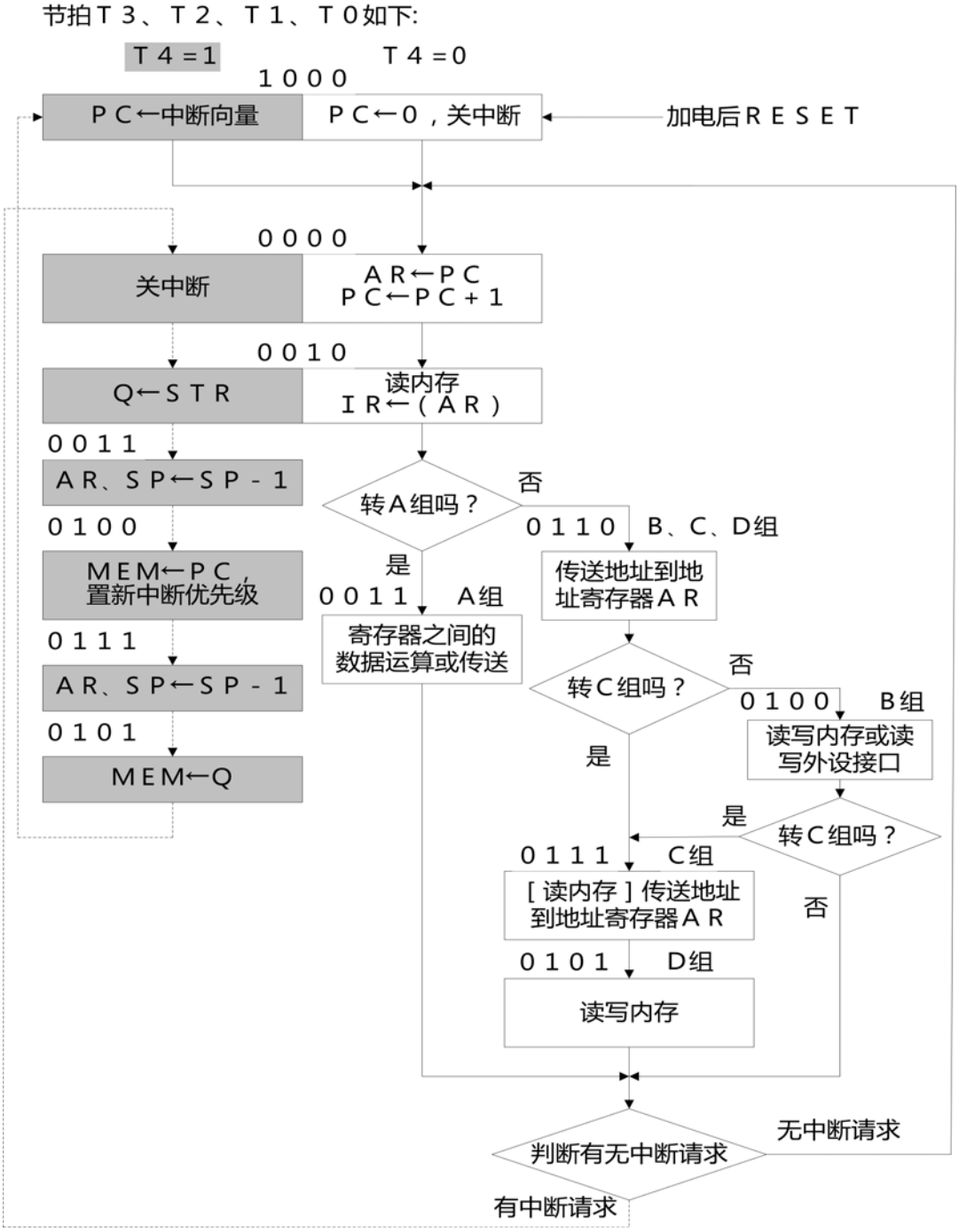
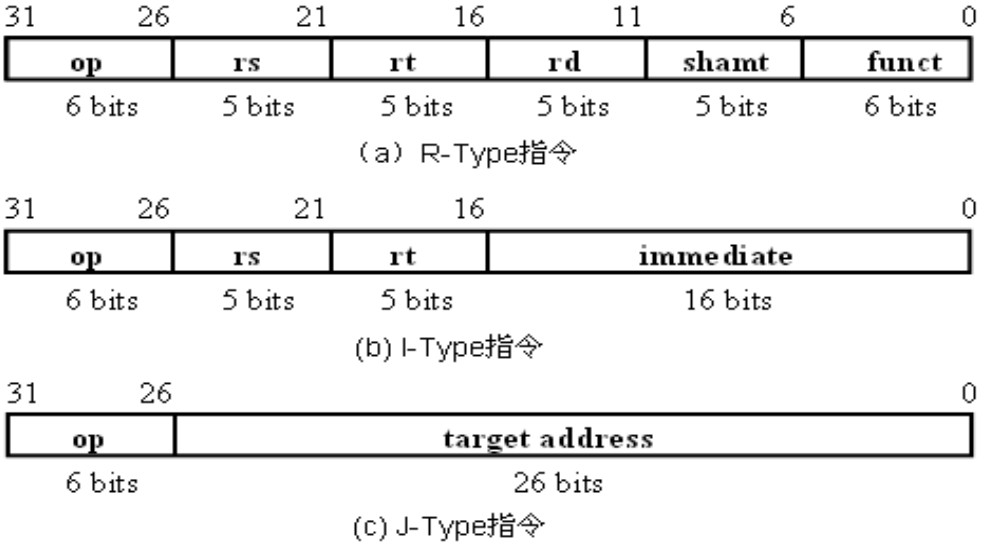


图 1.2.1

第二章 MIPS 指令描述

MIPS 指令系统的设计是按照规整、简单和一致等特性来设计的，它有利于指令的流水线执行，MIPS 是典型的 RISC 处理器，对于 32 位 MIPS 指令系统，它采用了 32 位定长指令字，操作码字段也是固定长度，没有专门的寻址方式字段，由指令格式确定各操作数的寻址方式。

指令格式只有以下三种：



下面是针对 32 位 MIPS 指令的特点：

1.指令长度一致 ,有利于简化取指令和指令译码操作 ,MIPS 指令都是 32 位；

2.指令格式少，且源寄存器位置相同，有利于在指令未知时就可取操作数，MIPS 指令的 rs 和 rt 位置一定，在指令译码时就可读 rs 和 rt 的值；

3.只有 lw/sw 指令才能访问存储器，有利于减少操作步骤，规整流水线，MIPS 可以把 lw/sw 指令的地址计算和运算指令的执行步骤规整在同一个周期；

4.数据和指令在内存中要“对齐”存放，有利于减少访存次数和流水线的规整；

5.表 2.1-2.8 所示指令集与 The MIPS32® Instruction Set Revision 2.62 兼容，标记星号的为基本指令，未标记星号的为扩展指令，一共包括 21 条算术

运算指令，13 条跳转指令，14 条存取指令，8 条逻辑运算指令，6 条移位指令，6 条数据移动指令，以及 14 条自陷指令。

表 2.1 算术运算指令

add	加法（带溢出位）	addi	立即数加法（带溢出位）
addiu	立即数加法(不带溢出位）	addu	加法（不带溢出位）
clo	计算前导一	clz	计算前导零
div	除法（有符号）	divu	除法（无符号）
madd	乘加（有符号）	maddu	乘加（无符号）
msub	乘减（有符号）	msubu	乘减（无符号）
mul	乘法（结果写到通用寄存器）	mult	乘法（有符号）
multu	乘法（有符号）	slt	小于置一（有符号）
slti	立即数小于置一(有符号）	sltiu	立即数小于置一(无符号）
sltu	小于置一（无符号）	sub	减法（有符号）
subu	减法（无符号）		
*seb	符号扩展字节	*seh	符号扩展半字

表 2.2 逻辑运算指令

and	与	andi	立即数与
lui	取立即数的高 16 位	nor	或非
or	或	ori	立即数或
xor	异或	xori	立即数异或

表 2.3 移位指令

sll	逻辑左移	sllv	逻辑左移变量
sra	算术右移	srav	算术右移变量
srl	逻辑右移	srlv	逻辑右移变量
*rotr	循环右移	*rotrv	循环右移变量

表 2.4 分支跳转指令

bal	转移并链接	beq	相等转移
bgez	大于等于零转移	bgezal	大于等于零转移并链接
bgtz	大于零转移	blez	小于等于零转移
bltz	小于零转移	bltzal	小于等于零转移并链接
bne	不相等转移	j	无条件跳转
jal	无条件跳转并链接	jalr	无条件跳转并链接寄存器
jr	.寄存器跳转		
*B	无条件转移	*jalr.hb	无条件跳转并链接到冒险阻塞寄存器
*jr.hb	冒险阻塞寄存器跳转		

表 2.5 存取控制指令

lb	取字节（有符号）	lbu	取字节（无符号）
lh	取半字（有符号）	lhu	取半字（无符号）
ll	取链接字	lw	取字
lwl	取左半字	lwr	取右半字
sb	保存字节	sc	保存条件字
sh	保存半字	sw	保存字
swl	保存左半字	swr	保存右半字
*pref	预取	*sync	同步访存
*synci	同步缓存		

表 2.6 数据移动指令

mfhi	从高位移	mflo	从低位移
movn	非零条件移动	movz	零条件移动
mthi	移到高位寄存器	mtlo	移到低位寄存器
*movf	浮点假条件移动	*movt	浮点真条件移动
*rdhwr	读硬件寄存器		

表 2.7 指令控制指令

nop	空指令		
*ehb	执行冒险阻塞	*pause	等待 LL 位来清除
*ssnop	超标量空指令		

表 2.8 自陷指令

break	跳出点	syscall	系统跳用
teq	相等自陷	teqi	立即数相等自陷
tge	大于等于自陷	tgei	立即数大于等于自陷
tgeiu	无符号立即数大于等于自陷	tgeu	无符号大于等于自陷
tlt	小于自陷	tlti	立即数小于自陷
tltiu	无符号立即数小于自陷	tltu	无符号小于自陷
tne	不等自陷	tnei	立即数不等自陷

在下面解释指令的操作时，采用了类似于 C 语言的描述语言。我们给出以下这些符号的含义如下：

．用下标表示字段中具体的位。对于指令和数据，按从最低位到最高位(即从右到左)的顺序依次进行编号，最低位为第 0 位，次低位为第 1 位，依此类推，下标可以是一个数字，也可以是一个范围，例如：$GPR[rs]_0$ 表示寄存器 rs 的最低位，$GPR[rs]_{3..0}$ 表示寄存器 rs 的最低四位。

．上标用于表示对字段进行复制的次数。例如，0^2 表示 2 位二进制 0。.

．符号||用于两个字段的拼接，并且可以出现在数据传送的任何一边。如下面例子 :$GPR[rd]31..0 \leftarrow GPR[rs]_{31..2}||0^2$ 表示将 rs 寄存器的高 30 位拼上最低 2 位 0，构成 1 个 32 位数送 rd 寄存器。

．vAdrr 为虚拟地址。

．pAdrr 为物理地址。

2.1 算术运算指令

2.1.1． add

0	rs	rt	rd	0	0x20 （100000）
6	5	5	5	5	6

指令格式：

add rd, rs, rt

功能描述：

如果没有溢出就将寄存器 rs 与 rt 的和存入寄存器 rd 中，有溢出则 rs 与 rt 的和不存入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] + GPR[rt]

temp ← (GPR[rs]$_{31}$||GPR[rs]$_{31..0}$) + (GPR[rt]$_{31}$||GPR[rt]$_{31..0}$)

if temp$_{32}$≠temp$_{31}$ then

SignalException(IntegerOverflow)

else

GPR[rd] $_{31..0}$ ← temp$_{31..0}$

endif

2.1.2 addi

8	rs	rt	immediate
6	5	5	16

指令格式：

addi rt, rs, immediate

功能描述：

如果没有溢出就将寄存器 rs 与有符号立即数的和存入寄存器 rt 中，有溢出

则寄存器 rs 与有符号立即数的和不存入寄存器 rt 中。

操作描述：

GPR[rt] ← GPR[rs] + immediate

temp ← (GPR[rs]$_{31}$||GPR[rs]$_{31..0}$)+ sign_extend(immediate)
if temp$_{32}$≠temp$_{31}$ then
SignalException(IntegerOverflow)
else
GPR[rd] $_{31..0}$ ← temp$_{31..0}$
endif

2.1.3 addiu

9	rs	rt	immediate
6	5	5	16

指令格式：

addiu rt, rs, immediate

功能描述：

将寄存器 rs 与有符号立即数的和存入寄存器 rt 中，不产生溢出。

操作描述：

GPR[rt] ← GPR[rs] + immediate

temp ← GPR[rs] + sign_extend(immediate)
GPR[rt] ← temp

2.1.4 addu

0	rs	rt	rd	0	0x21（100001）
6	5	5	5	5	6

指令格式：

addu rd, rs, rt

功能描述：

将寄存器 rs 与 rt 的和存入寄存器 rd 中，不产生溢出。

操作描述：

```
GPR[rd] ← GPR[rs] + GPR[rt]

temp ← GPR[rs]+GPR[rt]
        GPR[rd] ← temp
```

2.1.5 clo

0x1c	rs	0	rd	0	0x21（100001）
6	5	5	5	5	6

指令格式：

```
clo rd,  rs
```

功能描述：

将寄存器 rs 中数据起始为 1 的个数存入 rd 中 ,如果字中都是 1 ,结果为 32。

操作描述：

```
GPR[rd] ← count_leading_ones GPR[rs]

temp ← 32
for i in 31 .. 0
    if GPR[rs]i = 0 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

2.1.6 clz

0x1c	rs	0	rd	0	0x20（100000）
6	5	5	5	5	6

指令格式：

clz rd, rs

功能描述：

将寄存器 rs 中数据起始为 0 的个数存入 rd 中 ,如果字中都是 0 ,结果为 32。

操作描述：

```
GPR[rd] ← count_leading_zeros GPR[rs]

temp ← 32
for i in 31 .. 0
    if GPR[rs]i = 1 then
        temp ← 31 - i
        break
    endif
endfor
GPR[rd] ← temp
```

2.1.7 div

0	rs	rt	0	0x1a（011010）
6	5	5	10	6

指令格式：

div rs, rt

功能描述：

寄存器 rs 被寄存器 rt 除，将商存入寄存器 LO，余数存入寄存器 HI。

操作描述：

```
(HI, LO) ← GPR[rs] / GPR[rt]

q ← GPR[rs]31..0 div GPR[rt]31..0
LO ← q
r ← GPR[rs]31..0 mod GPR[rt]31..0
HI ← r
```

2.1.8 divu

0	rs	rt	0	0x1b（011011）
6	5	5	10	6

指令格式：

divu rs, rt

功能描述：

rs 被 rt 除，将商存入寄存器 LO，余数存入寄存器 HI。

操作描述：

(HI, LO) ← GPR[rs] / GPR[rt]

q ← (0 || GPR[rs]$_{31..0}$) div (0 || GPR[rt]$_{31..0}$)

r ← (0 || GPR[rs]$_{31..0}$) mod (0 || GPR[rt]$_{31..0}$)

LO ← sign_extend(q$_{31..0}$)

HI ← sign_extend(r$_{31..0}$)

2.1.9 madd

0x1c	rs	rt	0	0x0（000000）
6	5	5	10	6

指令格式：

madd rs, rt

功能描述：

将 rs 和 rt 的乘积所得的 64 位结果与链接寄存器 LO 和 HI 中的 64 位值相加。

操作描述：

(HI,LO) ← (HI,LO) + (GPR[rs] × GPR[rt])

temp ← (HI || LO) + (GPR[rs] × GPR[rt])

HI ← temp$_{63..32}$

LO ← temp$_{31..0}$

2.1.10 maddu

0x1c	rs	rt	0	0x1（000001）
6	5	5	10	6

指令格式：

maddu rs, rt

功能描述：

将 rs 和 rt 的乘积所得的 64 位结果与链接寄存器 LO 和 HI 中的 64 位值相加

操作描述：

(HI,LO) ← (HI,LO) + (GPR[rs] × GPR[rt])

temp ← (HI || LO) + (GPR[rs] × GPR[rt])

HI ← $temp_{63..32}$

LO ← $temp_{31..0}$

2.1.11 msub

0x1c	rs	rt	0	0x4（000100）
6	5	5	10	6

指令格式：

msub rs, rt

功能描述：

将链接寄存器 LO 和 HI 中的 64 位值与 rs 和 rt 的乘积所得的 64 位结果相减。

操作描述：

(HI,LO) ← (HI,LO) - (GPR[rs] × GPR[rt])

temp ← (HI || LO) - (GPR[rs] × GPR[rt])

HI ← $temp_{63..32}$

LO ← $temp_{31..0}$

2.1.12 msubu

0x1c	rs	rt	0	0x4（000101）
6	5	5	10	6

指令格式：

msub rs, rt

功能描述：

将链接寄存器 LO 和 HI 中的 64 位值与 s 和 rt 的乘积所得的 64 位结果相减。

操作描述：

(HI,LO) ← (HI,LO) - (GPR[rs] × GPR[rt])

temp ← (HI || LO) - (GPR[rs] × GPR[rt])

HI ← $temp_{63..32}$

LO ← $temp_{31..0}$

2.1.13 mul

0	rs	rt	rd	0	0x2（000010）
6	5	5	5	5	6

指令格式：

mul rd, rs, rt

功能描述：

将 rs 与 rt 乘积的低 32 位存入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] × GPR[rt]

temp ← GPR[rs] × GPR[rt]

GPR[rd] ← $temp_{31..0}$

HI ← UNPREDICTABLE

LO ← UNPREDICTABLE

2.1.14 mult

0	rs	rt	0	0x18（011000）
6	5	5	10	6

指令格式：

mult rs, rt

功能描述：

寄存器 rs 和 rt 的数据相乘 ,乘积的低位和高位数分别存入寄存器 LO 和 HI。

操作描述：

(HI, LO) ← GPR[rs] × GPR[rt]

prod ← $GPR[rs]_{31..0}$ × $GPR[rt]_{31..0}$

LO ← $prod_{31..0}$

HI ← $prod_{63..32}$

2.1.15 multu

0	rs	rt	0	0x19（011001）
6	5	5	10	6

指令格式：

multu rs, rt

功能描述：

寄存器 rs 和 rt 的数据相乘 ,乘积的低位和高位数分别存入寄存器 LO 和 HI。

操作描述：

(HI, LO) ← GPR[rs] × GPR[rt]

prod ← (0 || $GPR[rs]_{31..0}$) × (0 || $GPR[rt]_{31..0}$)

LO ← $prod_{31..0}$

HI ← $prod_{63..32}$

2.1.16 slt

0	rs	rt	rd	0	0x2a（101010）
6	5	5	5	5	6

指令格式：

slt rd, rs, rt

功能描述：

若寄存器 rs 比 rt 小，寄存器 rd 置 1；否则，rd 置 0。

操作描述：

GPR[rd] ← (GPR[rs] < GPR[rt])

if GPR[rs] < GPR[rt] then
    GPR[rd] ← $0^{GPRLEN-1}$ || 1
else
    GPR[rd] ← 0^{GPRLEN}
endif

2.1.17 slti

0xa	rs	rt	immediate
6	5	5	16

指令格式：

slti rt, rs, immediate

功能描述：

寄存器 rs 与有符号扩展立即数进行有符号数比较若小，寄存器 rt 置 1；否则，rt 置 0。

操作描述：

GPR[rt] ← (GPR[rs] < immediate)

if GPR[rs] < sign_extend(immediate) then
    GPR[rt] ← $0^{GPRLEN-1}$|| 1

else

GPR[rt] ← 0^{GPRLEN}

endif

2.1.18 sltiu

0xb	rs	rt	immediate
6	5	5	16

指令格式：

sltiu rt, rs, immediate

功能描述：

寄存器 rs 与有符号扩展立即数进行无符号数比较若小，寄存器 rt 置 1；否则，rt 置 0。

操作描述：

GPR[rt] ← (GPR[rs] < immediate)

if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then

GPR[rt] ← $0^{GPRLEN-1}$ || 1

else

GPR[rt] ← 0^{GPRLEN}

endif

2.1.19 sltu

0	rs	rt	rd	0	0x2b（101011）
6	5	5	5	5	6

指令格式：

sltu rd, rs, rt

功能描述：

若寄存器 rs 比 rt 小(无符号数比较)，寄存器 rd 置 1；否则，rd 置 0，不产生溢出。

操作描述：

PR[rd] ← (GPR[rs] < GPR[rt])

if (0 || GPR[rs]) < (0 || GPR[rt]) then

GPR[rd] ← $0^{GPRLEN-1}$ || 1

else

GPR[rd] ← 0^{GPRLEN}

endif

2.1.20 sub

0	rs	rt	rd	0	0x22（100010）
6	5	5	5	5	6

指令格式：

sub rd, rs, rt

功能描述：

如果没有溢出就将寄存器 rs 与 rt 的差存入寄存器 rd 中，有溢出则 rs 与 rt 的差不存入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] − GPR[rt]

temp ← (GPR[rs]31||$GPR[rs]_{31..0}$) − (GPR[rt]31||$GPR[rt]_{31..0}$)

if $temp_{32} \neq temp_{31}$ then

SignalException(IntegerOverflow)

else

$GPR[rd]_{31..0} \leftarrow temp_{31..0}$

Endif

2.1.21 subu

0	rs	rt	rd	0	0x23（100011）
6	5	5	5	5	6

指令格式：

subu rd, rs, rt

功能描述：

将寄存器 rs 与 rt 的差（无符号数的减法）存入寄存器 rd 中，不产生溢出。

操作描述：

GPR[rd] ← GPR[rs] − GPR[rt]

temp ← GPR[rs] − GPR[rt]

GPR[rd] ← temp

2.1.22 seb*

0x31	0	rt	rd	0x32	0x32（100000）
6	5	5	5	5	6

指令格式：

seb rd, rt

功能描述：

将寄存器 rt 的低八位进行带符号扩展后，将结果存入寄存器 rd。

操作描述：

GPR[rd] ← SignExtend(GPR[rt]$_{7..0}$)

2.1.23 seh*

0x31	0	rt	rd	0x48	0x32（100000）
6	5	5	5	5	6

指令格式：

seh rd, rt

功能描述：

将寄存器 rt 的低 16 位进行带符号扩展后，将结果存入寄存器 rd。

操作描述：

GPR[rd] ← SignExtend(GPR[rt]$_{16..0}$)

2．2 逻辑运算指令

2.2.1 and

0	rs	rt	rd	0	0x24（100100）
6	5	5	5	5	6

指令格式：

and rd, rs, rt

功能描述：

将寄存器 rs 与 rt 进行逐位与后入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] and GPR[rt]

2.2.2 andi

0xc	rs	rt	immediate
6	5	5	16

指令格式：

andi rt, rs, immediate

功能描述：

将寄存器 rs 与 0 扩展立即数的逐位逻辑与结果存入寄存器 rt 中。

操作描述：

GPR[rt] ← GPR[rs] and zero_extend(immediate)

2.2.3 lui

0xf	0	rt	immediate
6	5	5	16

指令格式：

lui rt, immediate

功能描述：

将立即数 immediate 的低位值存入寄存器 rt 的高位地址，将寄存器的低位值置 0。

操作描述：

GPR[rt] ← immediate || 0^{16}

2.2.4 nor

0	rs	rt	rd	0	0x27（100111）
6	5	5	5	5	6

指令格式：

nor rd, rs, rt

功能描述：

将寄存器 rs 与 rt 按位逻辑或非的结果存入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] nor GPR[rt]

2.2.5 or

0	rs	rt	rd	0	0x25（100101）
6	5	5	5	5	6

指令格式：

or rd, rs, rt

功能描述：

将寄存器 rs 与 rt 按位逻辑或的结果存入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] or GPR[rt]

2.2.6 ori

0xd	rs	rt	immediate
6	5	5	16

指令格式：

ori rt, rs, immediate

功能描述：

将寄存器 rs 与 0 扩展立即数的逐位逻辑或结果存入存入寄存器 rt 中。

操作描述：

GPR[rt] ← GPR[rs] or zero_extend(immediate)

2.2.7 xor

0	rs	rt	rd	0	0x26（100110）
6	5	5	5	5	6

指令格式：

xor rd, rs, rt

功能描述：

将寄存器 rs 与 rt 按位逻辑异或的结果存入寄存器 rd 中。

操作描述：

GPR[rd] ← GPR[rs] xor GPR[rt]

2.2.8 xori

0xe	rs	rt	immediate
6	5	5	16

指令格式：

xori rt, rs, immediate

功能描述：

将寄存器 rs 与 0 扩展立即数的逐位逻辑异或结果存入寄存器 rt 中。

操作描述：

GPR[rt] ← GPR[rs] xor zero_extend(immediate)

2.3 移位指令

2.3.1 sll

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

指令格式：

sll rd, rt, shamt

功能描述：

由立即数 shamt 指定寄存器 rt 的左移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] << shamt

s ←shamt

temp ← $GPR[rt]_{(31-s)..0}$ || 0^s

GPR[rd] ← temp

2.3.2 sllv

0	rs	rt	rd	0	0x4（000100）
6	5	5	5	5	6

指令格式：

sllv rd, rt, rs

功能描述：

由 rs 指定寄存器 rt 的左移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] << rs

s ← $GPR[rs]_{4..0}$

temp ← $GPR[rt]_{(31-s)..0}$ || 0^s

GPR[rd] ← temp

2.3.3 sra

0	rs	rt	rd	shamt	0x3(000011)
6	5	5	5	5	6

指令格式：

sra rd, rt, shamt

功能描述：

由立即数 shamt 指定寄存器 rt 的算术右移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] >> shamt (arithmetic)

s ←shamt

temp ← (GPR[rt]31)s || GPR[rt]31..s

GPR[rd] ← temp

2.3.4 srav

0	rs	rt	rd	0	0x7（000111）
6	5	5	5	5	6

指令格式：

srav rd, rt, rs

功能描述：

由 rs 指定寄存器 rt 的算术右移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] >> rs (arithmetic)

s ← $GPR[rs]_{4..0}$

temp ← $(GPR[rt]_{31})s$ || $GPR[rt]_{31..s}$

GPR[rd] ← temp

2.3.5 srl

0	rs	rt	rd	shamt	0x2(000010)
6	5	5	5	5	6

指令格式：

srl rd, rt, shamt

功能描述：

由立即数 shamt 指定寄存器 rt 的左移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] >> shamt (logical)

s ←shamt

temp ← 0^s || $GPR[rt]_{31..s}$

GPR[rd] ← temp

2.3.6 srlv

0	rs	rt	rd	0	0x6（000110）
6	5	5	5	5	6

指令格式：

srlv rd, rt, rs

功能描述：

由 rs 指定寄存器 rt 的右移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] >> GPR[rs] (logical)

s ← $GPR[rs]_{4..0}$

temp ← 0^s || $GPR[rt]_{31..s}$

GPR[rd] ← temp

2.3.7 rotr*

0	0	1	rt	rd	shamt	0x2（000010）
6	4	1	5	5	5	6

指令格式：

rotr rd, rt, shamt

功能描述：

由立即数 shamt 指定寄存器 rt 的右移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] ↔(right) shamt

```
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
    UNPREDICTABLE
endif
s ← shamt
temp ← GPR[rt]s-1..0 || GPR[rt]31..s
GPR[rd] ← temp
```

2.3.8 rotrv*

指令格式：

0	rs	rt	rd	0	1	0x6（000110）
6	5	5	5	4	1	6

指令格式：

rotrv rd, rt, rs

功能描述：

由 rs 指定寄存器 rt 的右移位数，并将结果存入寄存器 rd。

操作描述：

GPR[rd] ← GPR[rt] ↔(right) GPR[rs]

```
if ((ArchitectureRevision() < 2) and (Config3SM = 0)) then
```

UNPREDICTABLE

endif

s ← GPR[rs]4..0

temp ← $GPR[rt]_{s-1..0}$ || $GPR[rt]_{31..s}$

GPR[rd] ← temp

2.4 分支跳转指令

2.4.1 bal

0x1	0	0x33	offset
6	5	5	16

指令格式：

bal offset

功能描述：

将返回地址存入 31 号寄存器，同时将 offset 符号扩展两位和与当前 PC 相加，作为新的 PC。

操作描述：

procedure_call

I: target_offset ← sign_extend(offset || 0^2)

GPR[31] ← PC + 8

I+1: PC ← PC + target_offset

2.4.2 beq

0x4	rs	rt	Offset
6	5	5	16

指令格式：

beq rs, rt, label

功能描述：

若寄存器 rs 和 rt 相等，转移的指令数由有符号偏移量左移 2 位来决定。

操作描述：

```
if GPR[rs] = GPR[rt] then branch

I:      target_offset ← sign_extend(offset || 02)
        condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

2.4.3 bgez

0x1	rs	1	offset
6	5	5	16

指令格式：

```
bgez    rs,    offset
```

功能描述：

若寄存器 rs 大于等于 0，转移的指令数由有符号偏移量左移 2 位来决定。

操作描述：

```
if GPR[rs] ≥ 0 then branch

I:      target_offset ← sign_extend(offset || 0^2)
        condition ← GPR[rs] ≥ 0^GPRLEN
I+1:    if condition then
            PC ← PC + target_offset
        endif
```

2.4.4 bgezal

0x1	rs	0x11	offset
6	5	5	16

指令格式：

bgezal rs, offset

功能描述：

若寄存器 rs 大于等于 0，转移的指令数由有符号偏移量左移 2 位来决定。将下一条指令地址保存在寄存器 31 中。

操作描述：

if GPR[rs] ≥ 0 then procedure_call

```
I:    target_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] ≥ 0^GPRLEN
      GPR[31] ← PC + 8
I+1:  if condition then
          PC ← PC + target_offset
      endif
```

2.4.5 bgtz

0x7	rs	0	offset
6	5	5	16

指令格式：

bgtz rs, offset

功能描述：

若寄存器 rs 大于 0，转移的指令数由有符号偏移量左移 2 位来决定。

操作描述：

if GPR[rs] > 0 then branch

```
I:    target_offset ← sign_extend(offset || 0^2)
      condition ← GPR[rs] > 0^GPRLEN
I+1:  if condition then
          PC ← PC + target_offset
      endif
```

2.4.6 blez

0x6	rs	0	offset
6	5	5	16

指令格式：

blez rs, offset

功能描述：

若寄存器 rs 小于等于 0，转移的指令数由有符号偏移量左移 2 位来决定。

操作描述：

if GPR[rs] ≤ 0 then branch

I: target_offset ← sign_extend(offset || 0^2)
condition ← GPR[rs] ≤ 0^{GPRLEN}
I+1: if condition then
PC ← PC + target_offset
endif

2.4.7 bltz

0x1	rs	0	offset
6	5	5	16

指令格式：

bltz rs, offset

功能描述：

若寄存器 rs 小于 0，转移的指令数由有符号偏移量左移 2 位来决定。

操作描述：

if GPR[rs] < 0 then branch

I: target_offset ← sign_extend(offset || 0^2)
condition ← GPR[rs] < 0^{GPRLEN}
I+1: if condition then

PC ← PC + target_offset

endif

2.4.8 bltzal

0x1	rs	0x10	offset
6	5	5	16

指令格式：

bltzal rs, offset

功能描述：

若寄存器 rs 小于 0，转移的指令数由有符号偏移量左移 2 位来决定。将下一条指令地址保存在寄存器 31 中。

操作描述：

if GPR[rs] < 0 then procedure_call

I: target_offset ← sign_extend(offset || 0^2)

condition ← GPR[rs] < 0^{GPRLEN}

GPR[31] ← PC + 8

I+1: if condition then

PC ← PC + target_offset

endif

2.4.9 bne

0x5	rs	rt	offset
6	5	5	16

指令格式：

bne rs, rt, offset

功能描述：

若寄存器 rs 与 rt 不相等，转移的指令数由有符号偏移量左移 2 位来决定。

操作描述：

if GPR[rs] ≠ GPR[rt] then branch

I: target_offset ← sign_extend(offset || 0^2)
condition ← (GPR[rs] ≠ GPR[rt])
I+1: if condition then
PC ← PC + target_offset
endif

2.4.10 j

0x2	target

6 26

指令格式：

j target

功能描述：

无条件跳转到目标指令。

操作描述：

I:
I+1: PC ← $PC_{GPRLEN-1..28}$ || instr_index || 0^2

2.4.11 jal

0x3	target

6 26

指令格式：

jal target

功能描述：

无条件跳转到目标指令。并将下一条指令的地址保存到寄存器 31 中

操作描述：

I: GPR[31] ← PC + 8
I+1: PC ← $PC_{GPRLEN-1..28}$ || instr_index || 0^2

2.4.12 jalr

0	rs	0	rd	0	0x9（001001）
6	5	5	5	5	6

指令格式：

jalr rs (rd = 31 implied)

jalr rd, rs

功能描述：

无条件跳转到由寄存器 rs 指定的指令 ,并将下一条指令的地址保存到寄存器 rd（默认为 31）中。

操作描述：

GPR[rd] ← return_addr, PC ← GPR[rs]

```
I:      temp ← GPR[rs]
        GPR[rd] ← PC + 8
I+1:    if Config1CA = 0 then
            PC ← temp
        else
            PC ← tempGPRLEN-1..1 || 0
            ISAMode ← temp0
        endif
```

2.4.13 jr

0	rs	0	hints	0x8（001000）
6	5	10	5	6

指令格式：

jr rs

功能描述：

无条件跳转到由寄存器 rs 指定的指令。

操作描述：

PC ← GPR[rs]

I: temp ← GPR[rs]
I+1: if $Config1_{CA}$ = 0 then
PC ← temp
else
PC ← $temp_{GPRLEN-1..1}$ || 0
ISAMode ← $temp_0$
endif

2.4.14 b*

0x4	0	0	offset
6	5	5	16

指令格式：

b offset

功能描述：

无条件跳转，转移的指令数由有符号偏移量左移 2 位来决定。相当于 beq r0, r0, offset。

操作描述：

I: target_offset ← sign_extend(offset || 0^2)
I+1: PC ← PC + target_offset

2.4.15 jalr.hb*

0	rs	0	rd	1	hint	0x9
6	5	5	16	1	4	6

指令格式：

jalr.hb rs (rd = 31 implied)
jalr.hb rd, rs

功能描述：

无条件跳转到由寄存器 rs 指定的指令，并将下一条指令的地址保存到寄存器

rd（默认为 31）中，同时清除所有的冒险阻塞单元。

操作描述：

```
GPR[rd] ← return_addr,
PC ← GPR[rs],
clear execution and instruction hazards

I:      temp ← GPR[rs]
        GPR[rd] ← PC + 8
I+1:    if Config1CA = 0 then
            PC ← temp
        else
            PC ← temp^(GPRLEN-1..1) || 0
            ISAMode ← temp^0
        endif
        ClearHazards()
```

2.4.16 jr.hb*

0	rs	0	rd	1	hint	0x8
6	5	5	16	1	4	6

指令格式：

```
jr.hb    rs
```

功能描述：

无条件跳转到由寄存器 rs 指定的指令,同时清除所有的冒险阻塞单元。

操作描述：

```
PC ← GPR[rs],
clear execution and instruction hazards

I:      temp ← GPR[rs]
I+1:    if Config1CA = 0 then
            PC ← temp
```

```
else
    PC ← temp_{GPRLEN-1..1} || 0
    ISAMode ← temp_0
endif
ClearHazards()
```

2.5 存取控制指令

2.5.1 lb

0x20	base	rt	offset
6	5	5	16

指令格式：

lb rt, offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址中的字节内容存入寄存器 rt 中，字节由 lb 扩展符号。

操作描述：

```
GPR[rt] ← memory[GPR[base] + offset]

vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_{PSIZE-1..2} || (pAddr_{1..0} xor ReverseEndian^2)
memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)
byte ← vAddr_{1..0} xor BigEndianCPU^2
GPR[rt] ← sign_extend(memword_{7+8*byte..8*byte})
```

2.5.2 lbu

0x24	base	rt	offset
6	5	5	16

指令格式：

lbu　　rt,　offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址中的字节内容存入寄存器 rt 中。

操作描述：

GPR[rt] ← memory[GPR[base] + offset]

vAddr ← sign_extend(offset) + GPR[base]

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)

memword ← LoadMemory (CCA, BYTE, pAddr, vAddr, DATA)

byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$

GPR[rt] ← zero_extend($memword_{7+8*byte..8*byte}$)

2.5.3 lh

0x21	base	rt	offset
6	5	5	16

指令格式：

lh　rt,　offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址中的 16 位数值（半字）存入寄存器 rt 中，半字由 lb 扩展符号。

操作描述：

GPR[rt] ← memory[GPR[base] + offset]

vAddr ← sign_extend(offset) + GPR[base]

if vAddr0 ≠ 0 then

　　SignalException(AddressError)

endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))

memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)

byte ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)

GPR[rt] ← sign_extend($memword_{15+8*byte..8*byte}$)

2.5.4 lhu

0x25	base	rt	offset
6	5	5	16

指令格式：

lhu rt, offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址中的 16 位数值（半字）存入寄存器 rt 中。

操作描述：

GPR[rt] ← memory[GPR[base] + offset]

vAddr ← sign_extend(offset) + GPR[base]

if vAddr0 ≠ 0 then

SignalException(AddressError)

endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)

pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))

memword ← LoadMemory (CCA, HALFWORD, pAddr, vAddr, DATA)

byte ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)

GPR[rt] ← zero_extend($memword_{15+8*byte..8*byte}$)

2.5.5 ll

0x30	base	rt	offset
6	5	5	16

指令格式：

ll rt, offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址对应的 32 位数值存入寄存器 rt 中，并且启动原子读-修改-写操作。这个操作通过条件存指令来完成，如果其他处理器含有取字的块进行写操作时 sc 将失败。由于 SPIM 没有模拟多处理器，可以成功进行条件存操作。

操作描述：

```
GPR[rt] ← memory[GPR[base] + offset]

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
LLbit ← 1
```

2.5.6 lw

0x23	base	rt	offset
6	5	5	16

指令格式：

lw rt, offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址中的 32 为数值（字）存入寄存器 rt 中。

操作描述：

```
GPR[rt] ← memory[GPR[base] + offset]

vAddr ← sign_extend(offset) + GPR[base]
```

```
if vAddr1..0 ≠ 0^2 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

2.5.7 lwl

0x22	base	rt	offset
6	5	5	16

指令格式：

lwl rt, offset(base)

功能描述：

在小端方式下存储器的初始内容:

高位字节			低位字节	
a	b	c	d	
3	2	1	0	←字节地址

寄存器 24 的初始内容：

w	x	y	z

在小端方式下	寄存器 24 的内容				vAddr
执行 LWL $24,0($0)后	d	x	y	z	0
执行 LWL $24,1($0)后	c	d	y	z	1
执行 LWL $24,2($0)后	b	c	d	z	2
执行 LWL $24,3($0)后	a	b	c	d	3

在大端方式下存储器的初始内容:

低位字节			高位字节	
a	b	c	d	
0	1	2	3	←字节地址

寄存器 24 的初始内容：

w	x	y	z

在大端方式下	寄存器 24 的内容				vAddr
执行 LWL $24,0($0)后	a	b	c	d	0
执行 LWL $24,1($0)后	b	c	d	z	1
执行 LWL $24,2($0)后	c	d	y	z	2
执行 LWL $24,3($0)后	d	x	y	z	3

操作描述：

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
    pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword_7+8*byte..0 || GPR[rt]_23-8*byte..0
GPR[rt] ← temp
```

2.5.8 lwr

0x26	base	rt	offset
6	5	5	16

指令格式：

```
lwr     rt,   offset(base)
```

功能描述：

在小端方式下存储器的初始内容:

高位字节			低为字节	
a	b	c	d	
3	2	1	0	←字节地址

寄存器 24 的初始内容：

w	x	y	z

在小端方式下	寄存器 24 的内容				vAddr
执行 LWR $24,0($0)后	a	b	c	d	0
执行 LWR $24,1($0)后	w	a	b	c	1
执行 LWR $24,2($0)后	w	x	a	b	2
执行 LWR $24,3($0)后	w	x	y	a	3

在大端方式下存储器的初始内容:

低位字节			高位字节	
a	b	c	d	
0	1	2	3	←字节地址

寄存器 24 的初始内容：

w	x	y	z

在大端方式下	寄存器 24 的内容				vAddr
执行 LWR $24,0($0)后	w	x	y	a	0
执行 LWR $24,1($0)后	w	x	a	b	1
执行 LWR $24,2($0)后	w	a	b	c	2
执行 LWR $24,3($0)后	a	b	c	d	3

操作描述：

```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← pAddr_PSIZE-1..2 || (pAddr_1..0 xor ReverseEndian^2)
if BigEndianMem = 0 then
    pAddr ← pAddr_PSIZE-1..2 || 0^2
endif
byte ← vAddr_1..0 xor BigEndianCPU^2
memword ← LoadMemory (CCA, byte, pAddr, vAddr, DATA)
temp ← memword_31..32-8*byte || GPR[rt]_31–8*byte..0
GPR[rt] ← temp
```

2.5.9 sb

0x28	base	rt	offset
6	5	5	16

指令格式：

sb rt, offset(base)

功能描述：

将寄存器 rt 的低字节保存到 base 寄存器的值加上 offset 得到的地址中。

操作描述：

memory[GPR[base] + offset] ← GPR[rt]

vAddr ← sign_extend(offset) + GPR[base]

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)

pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)

bytesel ← $vAddr_{1..0}$ xor $BigEndianCPU^2$

dataword ← $GPR[rt]_{31-8*bytesel..0}$ || $0^{8*bytesel}$

StoreMemory (CCA, BYTE, dataword, pAddr, vAddr, DATA)

2.5.10 sc

0x38	base	rt	offset
6	5	5	16

指令格式：

sc rt, offset(base)

功能描述：

将寄存器 rt 的内容保存到 base 寄存器的值加上 offset 得到的地址中，并且启动原子读-修改-写操作。如果这个原子操作成功，内存字被修改，寄存器 rt 被置为 1。如果其他处理器向含有取字的块进行写操作时，这个原子操作将失败，指令没有修改内存，寄存器 rt 被置为 0。由于 SPIM 没有模拟多处理器，可以成功进行条件存操作。

操作描述：

```
if atomic_update
then memory[GPR[base] + offset] ← GPR[rt], GPR[rt] ← 1
else GPR[rt] ← 0

vAddr ← sign_extend(offset) + GPR[base]
if vAddr1..0 ≠ 02 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
if LLbit then
    StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
endif
GPR[rt] ← 031 || LLbit
```

2.5.11 sh

0x29	base	rt	offset
6	5	5	16

指令格式：

```
sh   rt,   offset(base)
```

功能描述：

将寄存器 rt 的低 16 位值保存到 base 寄存器的值加上 offset 得到的地址中。

操作描述：

```
memory[GPR[base] + offset] ← GPR[rt]

vAddr ← sign_extend(offset) + GPR[base]
if vAddr0 ≠ 0 then
    SignalException(AddressError)
endif
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
```

pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor (ReverseEndian || 0))

bytesel ← $vAddr_{1..0}$ xor (BigEndianCPU || 0)

dataword ← $GPR[rt]_{31-8*bytesel..0}$ || $0^{8*bytesel}$

StoreMemory (CCA, HALFWORD, dataword, pAddr, vAddr, DATA)

2.5.12 sw

0x2b	base	rt	offset
6	5	5	16

指令格式：

sw rt, offset(base)

功能描述：

将寄存器 rt 的值保存到 base 寄存器的值加上 offset 得到的地址中。

操作描述：

memory[GPR[base] + offset] ← GPR[rt]

vAddr ← sign_extend(offset) + GPR[base]

if $vAddr_{1..0} \neq 0^2$ then

SignalException(AddressError)

endif

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)

dataword ← GPR[rt]

StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)

2.5.13 swl

0x2a	base	rt	offset
6	5	5	16

指令格式：

swl rt, offset(base)

功能描述：

在小端方式下存储器的初始内容:

高位字节　　低位字节

a	b	c	d
3	2	1	0

←字节地址

寄存器 24 的初始内容：

w	x	y	z

在小端方式下	存储器的内容				vAddr
执行 SWL $24,0($0)后	a	b	c	w	0
执行 SWL $24,1($0)后	a	b	w	x	1
执行 SWL $24,2($0)后	a	w	x	y	2
执行 SWL $24,3($0)后	w	x	y	z	3

在大端方式下存储器的初始内容:

低位字节　　高位字节

a	b	c	d
0	1	2	3

←字节地址

寄存器 24 的初始内容：

w	x	y	z

在大端方式下	寄存器 24 的内容				vAddr
执行 SWL $24,0($0)后	w	x	y	z	0
执行 SWL $24,1($0)后	a	w	x	y	1
执行 SWL $24,2($0)后	a	b	w	x	2
执行 SWL $24,3($0)后	a	b	c	w	3

操作描述：

memory[GPR[base] + offset] ← GPR[rt]

vAddr ← sign_extend(offset) + GPR[base]

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)

$pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, (pAddr_{1..0}$ xor $ReverseEndian^2)$

if BigEndianMem = 0 then

$pAddr \leftarrow pAddr_{PSIZE-1..2} \,||\, 0^2$

endif

byte ← $vAddr_{1..0}$ xor $BigEndianCPU^2$

dataword ← $0^{24-8*byte}$ || $GPR[rt]_{31..24-8*byte}$

StoreMemory(CCA, byte, dataword, pAddr, vAddr, DATA)

2.5.14 swr

0x2e	base	rt	offset
6	5	5	16

指令格式：

swr rt, offset(base)

功能描述：

在小端方式下存储器的初始内容:

高位字节			低位字节	
a	b	c	d	
3	2	1	0	←字节地址

寄存器 24 的初始内容：

w	x	y	z

在小端方式下	存储器的内容				vAddr
执行 SWR $24,0($0)后	w	x	y	z	0
执行 SWR $24,1($0)后	x	y	z	d	1
执行 SWR $24,2($0)后	y	z	c	d	2
执行 SWR $24,3($0)后	z	b	c	d	3

在大端方式下存储器的初始内容:

低位字节			高位字节	
a	b	c	d	
0	1	2	3	←字节地址

寄存器 24 的初始内容：

w	x	y	z

在大端方式下	寄存器 24 的内容				vAddr
执行 SWR $24,0($0)后	z	b	c	d	0

执行 SWR $24,1($0)后	y	z	c	d	1
执行 SWR $24,2($0)后	x	y	z	d	2
执行 SWR $24,3($0)后	w	x	y	z	3

操作描述：

vAddr ← sign_extend(offset) + GPR[base]

(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)

pAddr ← $pAddr_{PSIZE-1..2}$ || ($pAddr_{1..0}$ xor $ReverseEndian^2$)

if BigEndianMem = 0 then

pAddr ← $pAddr_{PSIZE-1..2}$ || 0^2

endif

byte ← vAddr1..0 xor $BigEndianCPU^2$

dataword ← $GPR[rt]_{31-8*byte}$ || 0^{8*byte}

StoreMemory(CCA, WORD-byte, dataword, pAddr, vAddr, DATA)

2.5.15 pref*

0x33	base	hint	offset
6	5	5	16

指令格式：

pref hint, offset(base)

功能描述：

将 base 寄存器的值加上 offset 得到的地址中的内容预取到主存和 cache 之间。

操作描述：

prefetch_memory(GPR[base] + offset)

vAddr ← GPR[base] + sign_extend(offset)

(pAddr, CCA) ← AddressTranslation(vAddr, DATA, LOAD)

Prefetch(CCA, pAddr, vAddr, DATA, hint)

2.5.16 sync*

0	0	stype	0xf
6	15	5	6

指令格式：

sync (stype = 0 implied)

sync stype

功能描述：

同步访存。

操作描述：

SyncOperation(stype)

2.5.17 synci*

0x1	base	0x1f	offset
6	5	5	16

指令格式：

synci offset(base)

功能描述：

同步缓存。

操作描述：

vaddr ← GPR[base] + sign_extend(offset)

SynchronizeCacheLines(vaddr) /* Operate on all caches */

2.6 数据移动指令

2.6.1 mfhi

0	0	rd	0	0x10（010000）
6	10	5	5	6

指令格式：

mfhi rd

功能描述：

乘除操作把得到的结果存入 HI 和 LO 两个额外的寄存器中 ,这些操作向(从) 这些寄存器中移数据。乘、除、余伪指令使用这些单元与使用其他寄存器单元一样，在计算机结束后移动结果。本指令将高位寄存器 HI 中的数值移到寄存器 rd 中。

操作描述：

GPR[rd] ← HI

2.6.2 mflo

0	0	rd	0	0x12（001100）
6	10	5	5	6

指令格式：

mflo rd

功能描述：

将低位寄存器 LO 中的数值移到寄存器 rd 中。

操作描述：

GPR[rd] ← LO

2.6.3 movn

0	rs	rt	rd	0xb
6	5	5	5	11

指令格式：

movn rd, rs, rt

功能描述：

如果寄存器 rt 不为 0，将寄存器 rs 中的数值移到寄存器 rd 中。

操作描述：

```
if GPR[rt] ≠ 0 then
    GPR[rd] ← GPR[rs]
endif
```

2.6.4 movz

0	rs	rt	rd	0xa
6	5	5	5	11

指令格式：

movz rd, rs, rt

功能描述：

如果寄存器 rt 为 0，将寄存器 rs 中的数值移到寄存器 rd 中。

操作描述：

```
if GPR[rt] = 0 then
    GPR[rd] ← GPR[rs]
Endif
```

2.6.5 mthi

0	rs	0	0x11（001011）
6	5	15	6

指令格式：

mthi rs

功能描述：

将寄存器 rs 的值移到高位寄存器 HI 中。

操作描述：

```
HI ← GPR[rs]
```

2.6.6 mtlo

0	rs	0	0x13（001101）
6	5	15	6

指令格式：

mtlo rs

功能描述：

将寄存器 rs 的值移到低位寄存器 LO 中。

操作描述：

LO ← GPR[rs]

2.6.7 movf*

0	rs	cc	0	rd	0	0x1（000001）
6	5	3	2	5	5	6

指令格式：

movf rd, rs, cc

功能描述：

如果 FPU 寄存器条件代码标记 cc 为 0，将 CPU 寄存器 rs 中的数值移到寄存器 rd 中。如果 cc 被指令忽略，令条件代码标记为 0。

操作描述：

```
if FPConditionCode(cc) = 0 then
    GPR[rd] ← GPR[rs]
endif
```

2.6.8 movt*

0	rs	cc	1	rd	0	0x1（000001）
6	5	3	2	5	5	6

指令格式：

movt rd, rs, cc

功能描述：

如果 FPU 寄存器条件代码标记 cc 为 1，将 CPU 寄存器 rs 中的数值移到寄存器 rd 中。如果 cc 被指令忽略，令条件代码标记为 0。

操作描述：

```
if FPConditionCode(cc) = 1 then
    GPR[rd] ← GPR[rs]
endif
```

2.6.9 rdhwr*

0x1f	0	rt	rd	0	0x3b（111011）
6	5	5	5	5	6

指令格式：

rdhwr rt, rd

功能描述：

将硬件寄存器 rd 中内容放入寄存器 rt 中。

操作描述：

GPR[rt] ← HWR[rd]

case rd

0: temp ← EBaseCPUNum

1: temp ← SYNCI_StepSize()

2: temp ← Count

3: temp ← CountResolution()

29: temp ← UserLocal

30: temp ← Implementation-Dependent-Value

31: temp ← Implementation-Dependent-Value

otherwise: SignalException(ReservedInstruction)

endcase

GPR[rt] ← temp

2.7.指令控制指令

2.7.1 nop

0	0	0	0	0	0
6	5	5	5	5	6

指令格式：

nop

功能描述：

实际上相当于sll r0, r0, 0.

操作描述：

无

2.7.2 ehb*

0	0	0	0	0x3	0
6	5	5	5	5	6

指令格式：

ehb

功能描述：

停止所有指令的执行，知道所有阻塞单元都被清除。

操作描述：

ClearExecutionHazards()

2.7.3 pause*

0	0	0	0	0x5	0
6	5	5	5	5	6

指令格式：

pause

功能描述：

等待 ll 比特位来清除

操作描述：

```
if LLBit ≠ 0 then
    EPC ← PC + 4          /* Resume at the following instruction */
    DescheduleInstructionStream()
Endif
```

2.7.4 ssnop*

0	0	0	0	0x1	0
6	5	5	5	5	6

指令格式：

ssnop

功能描述：

跳出超标量处理器

操作描述：

无

2.8 自陷指令

2.8.1 break

0	code	0xd
6	20	6

指令格式：

break

功能描述：

产生断点异常

操作描述：

SignalException(Breakpoint)

2.8.2 syscall

0	code	0xc
6	20	6

指令格式：

syscall

功能描述：

产生系统调用异常

操作描述：

SignalException(SystemCall)

2.8.3 teq

0	rs	rd	code	0x34
6	5	5	10	6

指令格式：

teq rs, rt

功能描述：

若寄存器 rs 与寄存器 rt 中的内容相等，则自陷。

操作描述：

```
if GPR[rs] = GPR[rt] then
    SignalException(Trap)
endif
```

2.8.4 teqi

1	rs	0xc	immediate	
6	5	5	10	6

指令格式：

teqi rs, immediate

功能描述：

若寄存器 rs 与立即数相等，则自陷。

操作描述：

```
if GPR[rs] = sign_extend(immediate) then
    SignalException(Trap)
endif
```

2.8.5 tge

0	rs	rd	code	0x30
6	5	5	10	6

指令格式：

tge rs, rt

功能描述：

若寄存器 rs 的内容大于或等于寄存器 rt 中的内容，则自陷。

操作描述：

```
if GPR[rs] ≥ GPR[rt] then
    SignalException(Trap)
endif
```

2.8.6 tgei

1	rs	0x8	immediate	
6	5	5	10	6

指令格式：

tgei rs, immediate

功能描述：

若寄存器 rs 的内容大于或等于 immediate，则自陷。

操作描述：

```
if GPR[rs] ≥ sign_extend(immediate) then
    SignalException(Trap)
endif
```

2.8.7 tgeiu

1	rs	0x9	immediate	
6	5	5	10	6

指令格式：

tgeiu rs, immediate

功能描述：

若寄存器 rs 的内容大于或等于 immediate，则自陷。

操作描述：

```
if (0 || GPR[rs]) ≥ (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

2.8.8 tgeu

0	rs	rd	code	0x31
6	5	5	10	6

指令格式：

```
tgeu    rs,    rt
```

功能描述：

若寄存器 rs 的内容大于或等于寄存器 rt 中的内容，则自陷。

操作描述：

```
if (0 || GPR[rs]) ≥ (0 || GPR[rt]) then
    SignalException(Trap)
endif
```

2.8.9 tlt

0	rs	rd	code	0x32
6	5	5	10	6

指令格式：

```
tlt    rs,    rt
```

功能描述：

若寄存器 rs 的内容小于寄存器 rt 中的内容，则自陷。

操作描述：

```
if GPR[rs] < GPR[rt] then
    SignalException(Trap)
endif
```

2.8.10 tlti

1	rs	0xa	immediate	
6	5	5	10	6

指令格式：

tlti rs, immediate

功能描述：

若寄存器 rs 的内容小于 immediate，则自陷。

操作描述：

```
if GPR[rs] < sign_extend(immediate) then
    SignalException(Trap)
endif
```

2.8.11 tltiu

1	rs	0xb	immediate	
6	5	5	10	6

指令格式：

tltiu rs, immediate

功能描述：

若寄存器 rs 的内容小于 immediate，则自陷。

操作描述：

```
if (0 || GPR[rs]) < (0 || sign_extend(immediate)) then
    SignalException(Trap)
endif
```

2.8.12 tltu

0	rs	rd	code	0x33
6	5	5	10	6

指令格式：

tltu rs, rt

功能描述：

若寄存器 rs 的内容小于寄存器 rt 中的内容，则自陷。

操作描述：

```
if (0 || GPR[rs]) < (0 || GPR[rt]) then
    SignalException(Trap)
Endif
```

2.8.13 tne

0	rs	rd	code	0x36
6	5	5	10	6

指令格式：

```
tne     rs,     rt
```

功能描述：

若寄存器 rs 的内容不等于于寄存器 rt 中的内容，则自陷。

操作描述：

```
if GPR[rs] ≠ GPR[rt] then
    SignalException(Trap)
endif
```

2.8.14 tnei

1	rs	0xe	immediate	
6	5	5	10	6

指令格式：

```
tnei     rs,     immediate
```

功能描述：

若寄存器 rs 的内容不等于于 immediate，则自陷。

操作描述：

```
if GPR[rs] ≠ sign_extend(immediate) then
    SignalException(Trap)
Endif
```

第三章　实验篇

3.1 实验一　寄存器组设计实验

3.1.1 实验目的：

熟悉并掌握基本 MIPS 计算机组件的工作原理与设计方法。

掌握用 Verilog HDL 语言或 VHDL 语言设计一个由 32 个寄存器组成的字长为 32 位的寄存器组。

3.1.2 实验设备：

1．装有 Quartus II 的计算机一台。

2．Altera DE2-70 开发板一块。

3.1.3 实验原理框图：

32 位字长的 32 个寄存器组成的寄存器组的原理框图如图 3.1.3.1、图 3.1.3.2：


源寄存器1地址[4:0]
5
源寄存器2地址[4:0]
5
目的寄存器地址[4:0]
5
写数据[31:0]
32
寄存器写（低电平有效）
最高8位写使能（低电平有效）
次高8位写使能（低电平有效）
次低8位写使能（低电平有效）
最低8位写使能（低电平有效）
时钟
寄存器组
源寄存器1数据输出[31:0]
32
源寄存器2数据输出[31:0]
32

图 3.1.3.1

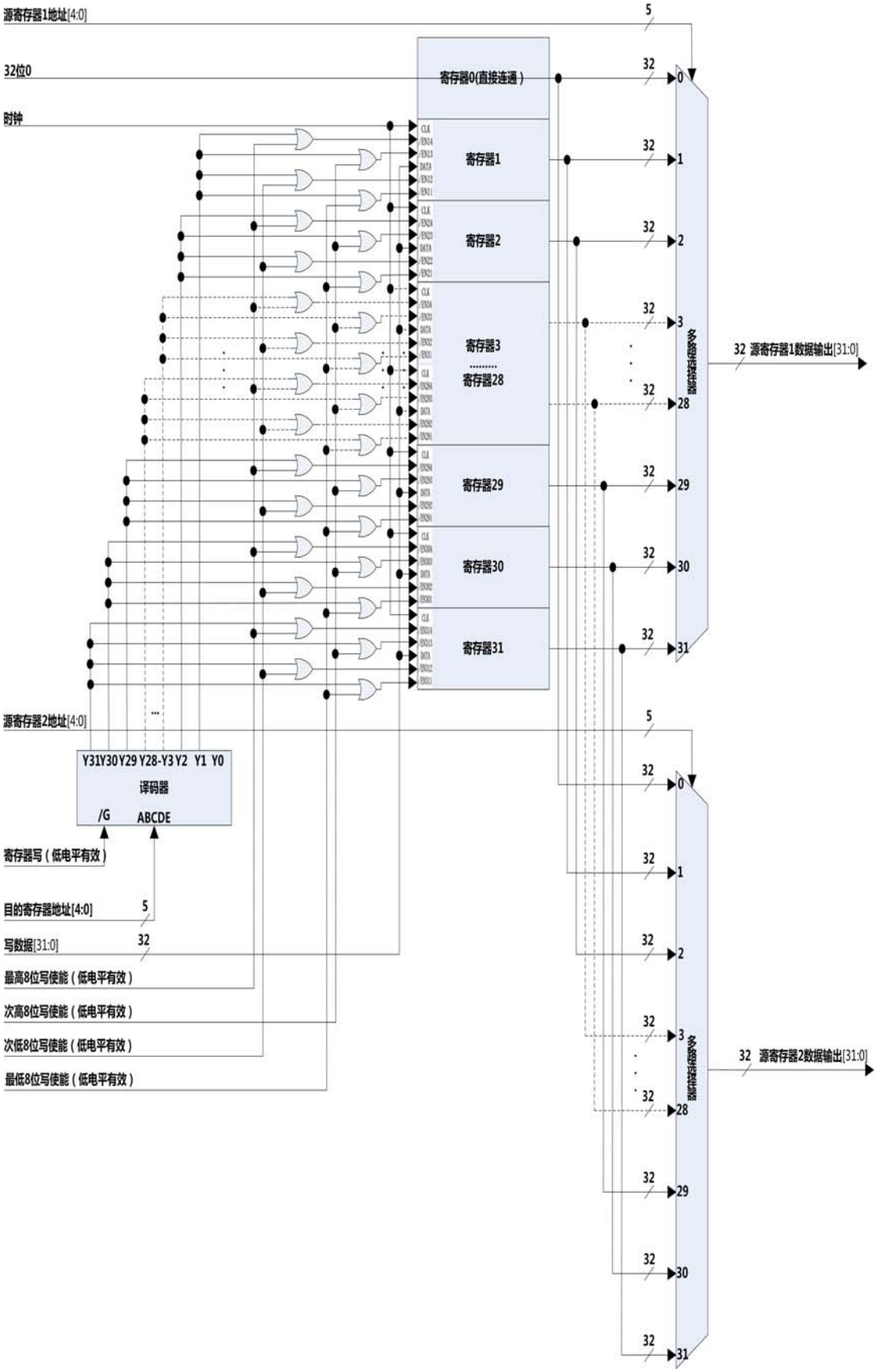

源寄存器1地址[4:0]
32位0
时钟
寄存器0(直接连通)
寄存器1
寄存器2
寄存器3
寄存器28
寄存器29
寄存器30
寄存器31
32 源寄存器1数据输出[31:0]
源寄存器2地址[4:0]
Y31Y30Y29 Y28-Y3 Y2 Y1 Y0
译码器
/G
ABCDE
寄存器写(低电平有效)
目的寄存器地址[4:0]
写数据[31:0]
最高8位写使能(低电平有效)
次高8位写使能(低电平有效)
次低8位写使能(低电平有效)
最低8位写使能(低电平有效)
32 源寄存器2数据输出[31:0]

图 3.1.3.2

3.1.4 实验任务：

根据实验原理框图完成一个由 32 个寄存器组成的字长为 32 位的寄存器组设计，并在 Quartus II 上模拟实现。

根据图 3.1.4.1 的实验原理框图完成一个由 16 个寄存器组成的字长为 4 位的寄存器组设计，并在 Altera DE2-70 开发板实现。

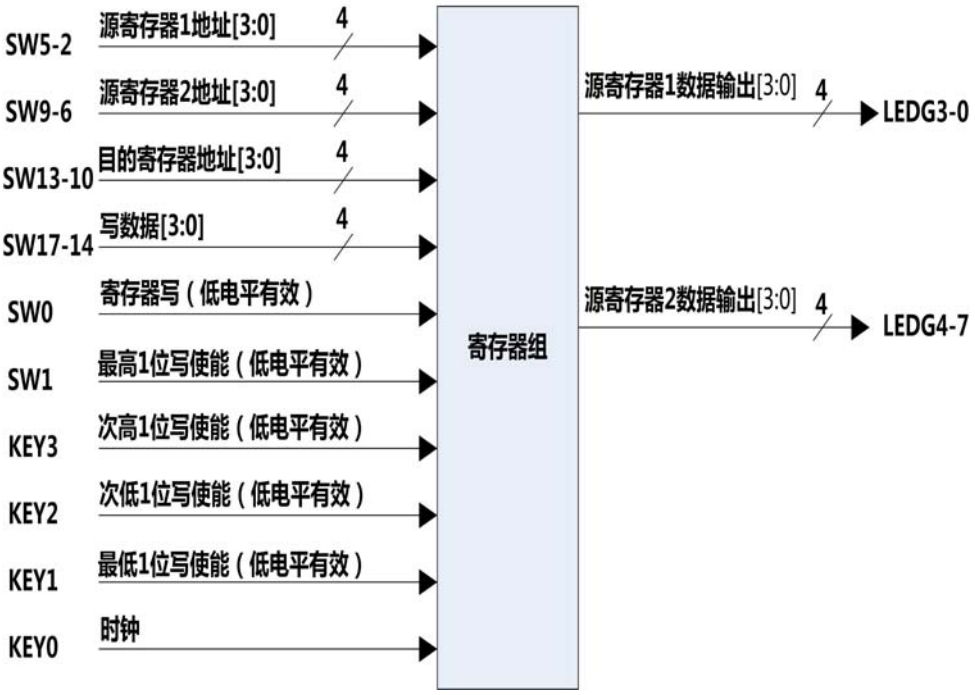


图 3.1.4.1

3.1.5 实验步骤：

打开计算机电源运行 Quartus II

Altera DE2-70 加电，并将下载电缆与计算机相连

用 Verilog HDL 语言或 VHDL 语言编写一个由 32 个寄存器组成的字长为 32 位的寄存器组的程序

编译调试仿真，记录数据

用 Verilog HDL 语言或 VHDL 语言编写一个由 16 个寄存器组成的字长为 4 位的寄存器组的程序

编译调试仿真下载到 Altera DE2-70，实验并记录数据

3.1.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.1.7 思考题：

1.MIPS 机器的运算器与普通运算器有哪些不同？
2.用 Microsoft Office Visio 画出用不带使能端得 8D 触发器构成一个 8 位的时钟上升沿有效并带使能端（低电平有效）的寄存器。

3.2 实验二 ALU 与 ALU 控制器设计实验

3.2.1 实验目的

1. 掌握 ALU 的工作原理和逻辑功能。
2．掌握 ALU 控制器的工作原理和作用

3.2.2 实验设备

1．装有 Quartus II 的计算机一台。
2．Altera DE2-70 开发板一块。

3.2.3 实验任务

1　实现 32 位的 ALU，使其能够支持基本的指令。
2．用 Verilog HDL 语言或 VHDL 语言来编写，实现 ALU 及 ALU 的控制器。

3.2.4 实验原理与电路图

ALU 的原理

在 MIPS 中，ALU 可执行的功能与操作见表 3.2.4.1，需三位控制信号。

除输出运算结果 Result，ALU 还输出信号 zero、Less、Overflow 分别表示运算结果是否为 0、两数比较是大还是小、是否有溢出，以用于某些判断指令。

为提高 ALU 的控制效率，ALU 采用两级控制，即通过 ALU 控制器实现对 ALU 的控制，而不是直接控制 ALU。图 3.2.4.1 为 ALU 原理示意图和逻辑图。图 3.2.4.2 为 ALU 控制器逻辑图。

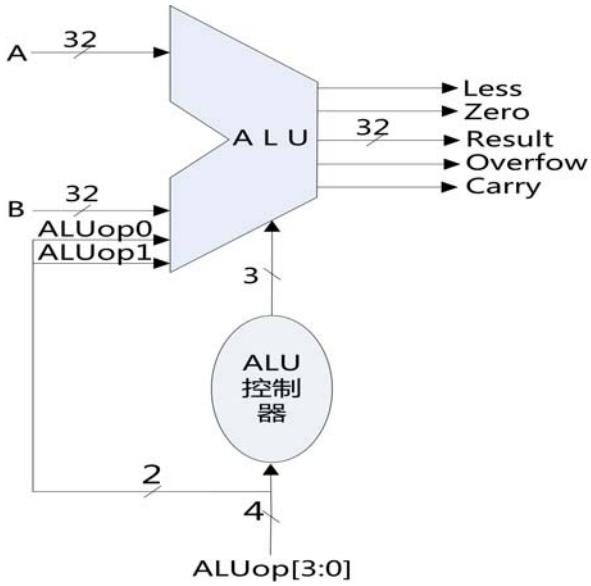
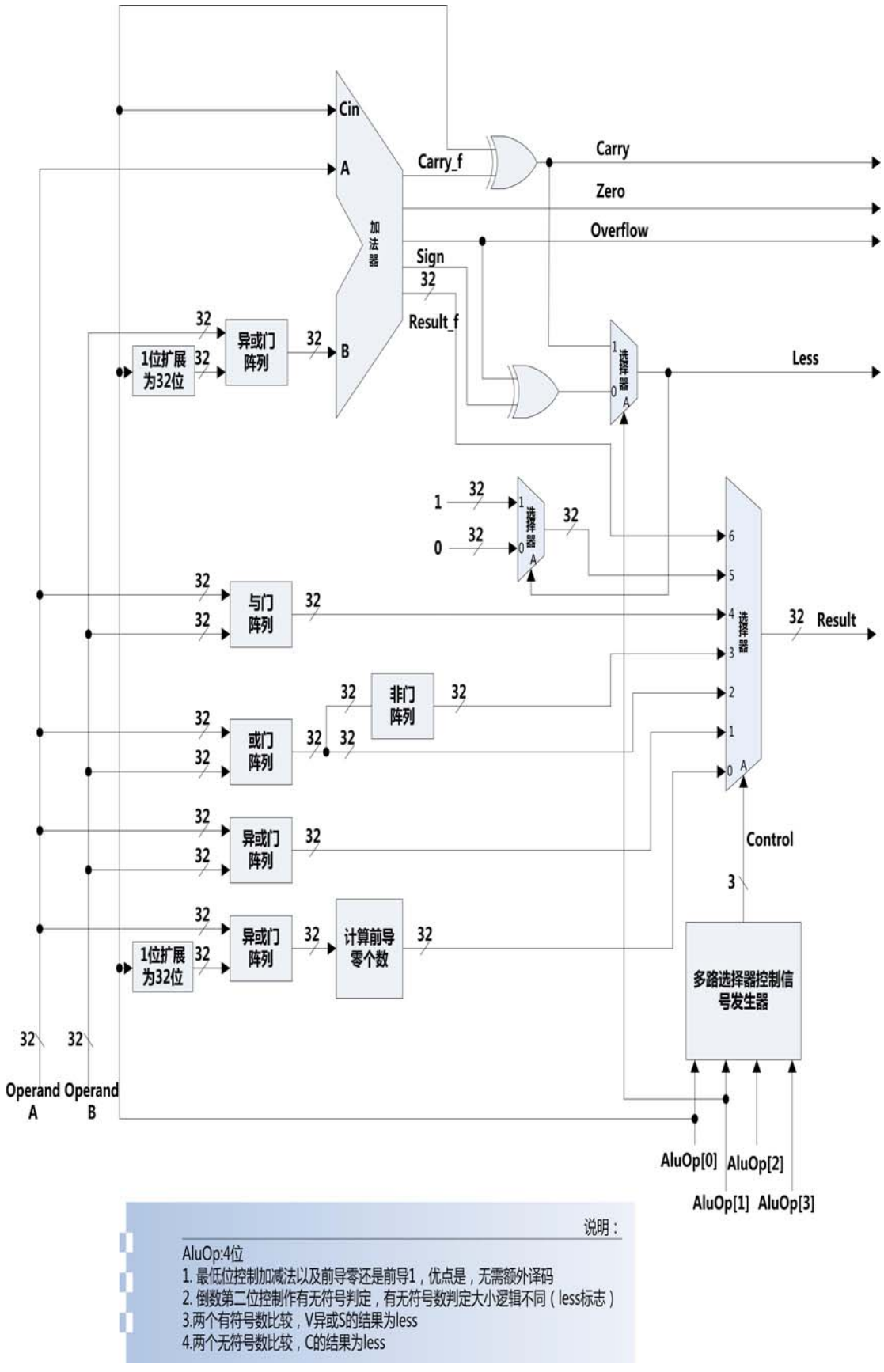


图 3.2.4.1



说明：

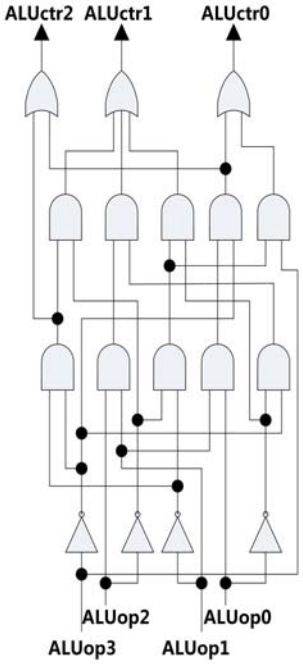
AluOp:4位

1. 最低位控制加减法以及前导零还是前导1，优点是，无需额外译码
2. 倒数第二位控制作有无符号判定，有无符号数判定大小逻辑不同（less标志）
3. 两个有符号数比较，V异或S的结果为less
4. 两个无符号数比较，C的结果为less

图 3.2.4.2

表 3.2.4.1：

ALUop	ALUctr	功能
0 0 0 0	1 1 0	加法
0 0 0 1	1 1 0	有符号减法
0 0 1 0	0 0 0	前导零
0 0 1 1	0 0 0	前导一
0 1 0 0	1 0 0	与
0 1 0 1	1 0 1	slt/slti
0 1 1 0	0 1 0	或
0 1 1 1	1 0 1	sltu/sltiu
1 0 0 0	0 1 1	或非
1 0 0 1	0 0 1	异或



说明：

$$ALUctr2 = \overline{ALUop3}\,\overline{ALUop1} + \overline{ALUop3}\,ALUop2\,ALUop0$$

$$ALUctr1 = \overline{ALUop3}\,\overline{ALUop2}\,\overline{ALUop1} + \overline{ALUop3}\,ALUop2\,ALUop1\,\overline{ALUop0} + \overline{ALUop2}\,\overline{ALUop1}\,\overline{ALUop0}$$

$$ALUctr0 = ALUop3\,\overline{ALUop2}\,\overline{ALUop1} + \overline{ALUop3}\,ALUop2\,ALUop0$$

图 3.2.4.3

AluOp:4位

1. 最低位控制加减法以及前导零还是前导1，优点是，无需额外译码
2. 倒数第二位控制作有无符号判定，有无符号数判定大小逻辑不同（less标志）
3. 两个有符号数比较，V异或S的结果为less
4. 两个无符号数比较，C的结果为less

3.2.5 实验步骤：

略（要求实验者自己写出）

3.2.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.2.7 思考题：

1.说明一级译码和二级译码各自的特点。

2.分析我们在实验中所用的ALU控制器与教课书上所用的ALU控制器编码为什么不一样？

3.3 实验三 32 位桶形移位器设计实验

3.3.1 实验目的

1. 学习掌握桶形移位器的工作原理。
2. 掌握双向桶形移位器的设计方法。

3.3.2 实验设备

1．装有 Quartus II 的计算机一台。

2．Altera DE2-70 开发板一块。

3.3.3 实验原理与电路图

图 3.3.3.1、图 3.3.3.2 给出一个 8 位的桶型移位器的逻辑图

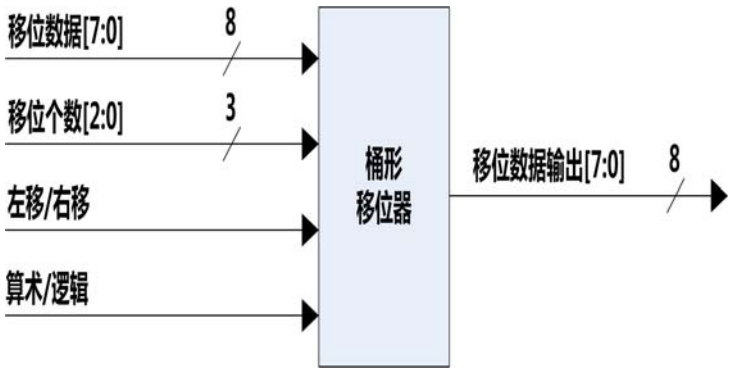


图 3.3.3.1

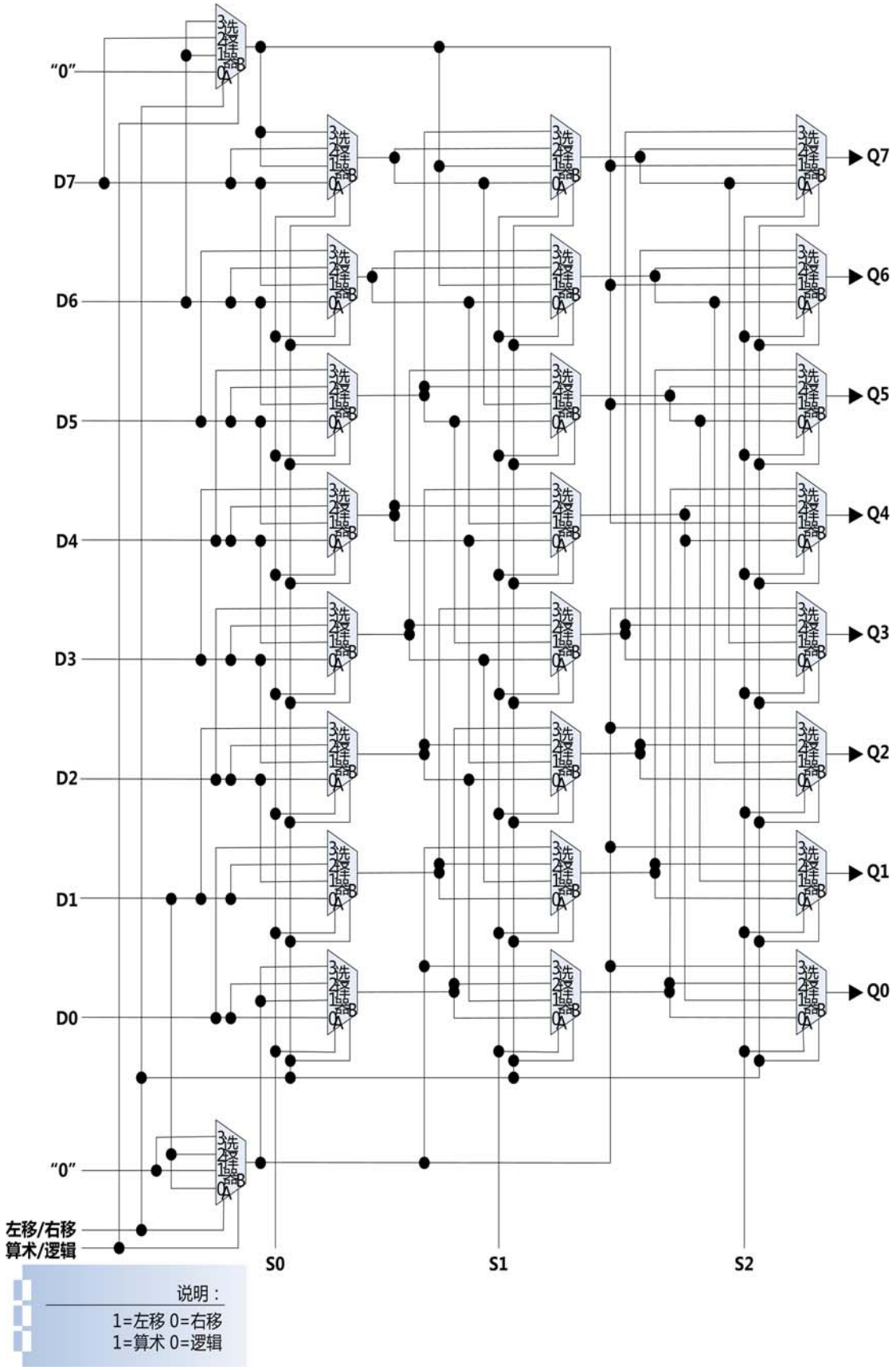

"0"
D7
D6
D5
D4
D3
D2
D1
D0
"0"
左移/右移
算术/逻辑
S0
S1
S2
Q7
Q6
Q5
Q4
Q3
Q2
Q1
Q0
说明：
1=左移 0=右移
1=算术 0=逻辑

图 3.3.3.2

3.3.4 实验任务

1　用 Verilog HDL 语言或 VHDL 语言来编写，实现 32 位的桶形移位器。并在 Quartus II 上实现模拟仿真。

2．用 Verilog HDL 语言或 VHDL 语言来编写，在 Altera DE2-70 开发板实现 8 位的桶形移位器（如图 3.3.4.1），使其能够正常工作。

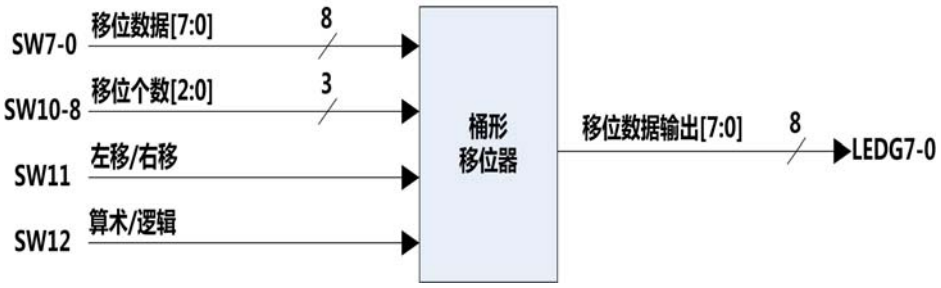


图 3.3.4.1

3.3.5 实验步骤：

略（要求实验者自己写出）

3.3.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.3.7 思考题：

1.说明桶形移位器与普通移位寄存器各自有什么特点？

2.试将 32 位的桶形移位器算术左移改成循环左移，其它功能不变，并用 Microsoft Office Visio 画出相同功能的 8 位桶形移位器。

3.4 实验四 单时钟周期 CPU 的设计实验

3.4.1 实验目的

1. 理解 MIPS 常用的指令系统并掌握单周期 CPU 的工作原理与逻辑功能实现。

2．通过对单周期 CPU 的运行状况进行观察和分析，进一步加深理解。

3.4.2 实验设备

1．装有 Quartus II 的计算机一台。

2．Altera DE2-70 开发板一块。

3.4.3 实验任务

用 Verilog HDL 语言或 VHDL 语言来编写，实现单周期 CPU 设计。 能够完成以下十六条指令：

add rd,rs,rt

addu rd,rs,rt

addi rt,rs,imm

addiu rt,rs,imm

sub rd,rs,rt

subu rd,rs,rt

nor rd,rs,rt

xori rt,rs,imm

clo

clz

slt rd,rs,rt

sltu rd,rs,rt

slti rt,rs,imm

sltiu rt,rs imm

blez rs,imm

j target

3.4.4 实验原理与电路图

略（要求实验者自己画出）

3.4.5 实验步骤：

略（要求实验者自己写出）

3.4.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.4.7 思考题：

单时钟周期 CPU 有什么特点？在设计单时钟周期 CPU 应注意些什么？

3.5 实验五 多时钟周期 CPU 的设计实验

3.5.1 实验目的

1. 深入理解 MIPS 指令系统并掌握在多时钟周期 CPU 的设计中，状态机是如何设计的。
2. 掌握多时钟周期 CPU 的工作原理与逻辑功能实现。
3. 通过对多时钟周期 CPU 的运行状况进行观察和分析，进一步加深理解。

3.5.2 实验设备

1．装有 Quartus II 的计算机一台。
2．Altera DE2-70 开发板一块。

3.5.3 实验任务

用 Verilog HDL 语言或 VHDL 语言来编写，实现多时钟周期 CPU 设计。 能够完成以下二十二条指令（均不考虑虚拟地址和 Cache，并且默认为小端方式）：

add rd,rs,rt
addu rd,rs,rt
addi rt,rs,imm

addiu rt,rs,imm

sub rd,rs,rt

subu rd,rs,rt

nor rd,rs,rt

xori rt,rs,imm

clo

clz

slt rd,rs,rt

sltu rd,rs,rt

slti rt,rs,imm

sltiu rt,rs imm

sllv rd,rt,rs

sra rd,rt,shamt

blez rs,imm

j target

lwl rt,offset(base)

lwr rt,offset(base)

lw rt, imm(rs)

sw rt,imm(rs)

3.5.4 实验原理与电路图

略（要求实验者自己画出）

3.5.5 实验步骤：

略（要求实验者自己写出）

3.5.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.5.7 思考题：

多时钟周期 CPU 有什么特点？设计多时钟周期 CPU 与单时钟周期 CPU 有什么不同？应注意哪些问题？

3.6 实验六 整数乘法器的设计实验

3.6.1 实验目的

1. 学习并掌握整数乘法器的算法。
2. 掌握用 Verilog HDL 语言或 VHDL 语言来编写整数乘法器。

3.6.2 实验设备

1．装有 Quartus II 的计算机一台。

2．Altera DE2-70 开发板一块。

3.6.3 实验任务

用 Verilog HDL 语言或 VHDL 语言来编写，设计并实现一种 32 位整数乘法器。

3.6.4 写出算法并画出流程图

略（要求实验者自己写出 32 整数乘法器的算法并画出流程图）

3.6.5 实验步骤：

略（要求实验者自己写出）

3.6.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.6.7 思考题：

试写出除本次实验所用的算法外，自己已知的各种整数乘法器的算法。

3.7 实验七 整数除法器的设计实验

3.7.1 实验目的

1. 学习并掌握整数除法器的算法。
2. 掌握用 Verilog HDL 语言或 VHDL 语言来编写整数除法器。

3.7.2 实验设备

1．装有 Quartus II 的计算机一台。

2．Altera DE2-70 开发板一块

3.7.3 实验任务

用 Verilog HDL 语言或 VHDL 语言来编写，设计并实现一种 32 位整数除法器。

3.7.4 写出算法并画出流程图

略（要求实验者自己写出 32 整数除法器的算法并画出流程图）

3.7.5 实验步骤：

略（要求实验者自己写出）

3.7.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.7.7 思考题：

试写出除本次实验所用的算法外，自己已知的各种整数除法器的算法。

3.8 实验八 流水线 CPU 的设计实验

3.8.1 实验目的

1. 学习并掌握流水线 CPU 的工作原理和设计方法。
2. 通过对流水线 CPU 的运行状况进行观察和分析，进一步加深理解。

3.8.2 实验设备

1．装有 Quartus II 的计算机一台。

2．Altera DE2-70 开发板一块。

3.8.3 实验任务

用 Verilog HDL 语言或 VHDL 语言来编写，实现流水线 CPU 设计。 能够完成以下二十二条指令（均不考虑虚拟地址和 Cache，并且默认为小端方式）：

add rd,rs,rt

addu rd,rs,rt

addi rt,rs,imm

addiu rt,rs,imm

sub rd,rs,rt

subu rd,rs,rt

nor rd,rs,rt

xori rt,rs,imm

clo

clz

slt rd,rs,rt

sltu rd,rs,rt

slti rt,rs,imm

sltiu rt,rs imm

sllv rd,rt,rs

sra rd,rt,shamt

blez rs,imm

j target

lwl rt,offset(base)

lwr rt,offset(base)

lw rt, imm(rs)

sw rt,imm(rs)

3.8.4 实验原理与电路图

略（要求实验者自己画出）

3.8.5 实验步骤：

略（要求实验者自己写出）

3.8.6 实验报告的要求：

认真记录实验数据，并在实验报告中要将以上实验的数据进行分析与总结。对实验中所遇到的问题，以及如何解决要给予说明。

3.8.7 思考题：

流水线 CPU 有什么特点？设计流水线 CPU 与多时钟周期 CPU 有什么不同？应注意哪些问题？

参考书及文献

1、教材：袁春风编著《计算机组成与系统结构》清华大学出版社

2、 （美）David A.Patterson John L.Hhennessy 著《计算机组成与设计硬件/软件接口》

3、《FPGA 与 SOPC 设计教程-DE2 实践》张志刚编著 ,西安电子科技大学出版社 ,2007。

4、《计算机系统结构》张晨曦、王志英等著，2008。

5、《计算机组成课程设计》楼学庆、平玲娣 著，浙江大学出版社，2007。

6、《Verilog 数字系统设计教程》夏宇闻编著，北京航空航天大学出版出版社，2008。

7、《VHDL 与数字电路设计》卢毅、赖杰编著，科学出版社，2002。

8、MIPS32® Architecture For Programmers Volume II: The MIPS32® Instruction Set. Revision 2.62. January 2, 2009

附录 A：

DE2-70 上 EP2C70F896C6 的引脚分配表：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
oENET_CLK	PIN_D27	DRAM_DQ[20]	PIN_V3
iCLK_28	PIN_E16	DRAM_DQ[21]	PIN_W1
iCLK_50	PIN_AD15	DRAM_DQ[22]	PIN_W2
iCLK_50_2	PIN_D16	DRAM_DQ[23]	PIN_W3
iCLK_50_3	PIN_R28	DRAM_DQ[24]	PIN_Y1
iCLK_50_4	PIN_R3	DRAM_DQ[25]	PIN_Y2
iAUD_ADCDAT	PIN_E19	DRAM_DQ[26]	PIN_Y3
AUD_ADCLRCK	PIN_F19	DRAM_DQ[27]	PIN_AA1
AUD_BCLK	PIN_E17	DRAM_DQ[28]	PIN_AA2
oAUD_DACDAT	PIN_F18	DRAM_DQ[29]	PIN_AA3
AUD_DACLRCK	PIN_G18	DRAM_DQ[3]	PIN_AD1
oAUD_XCK	PIN_D17	DRAM_DQ[30]	PIN_AB1
DRAM_DQ[0]	PIN_AC1	DRAM_DQ[31]	PIN_AB2
DRAM_DQ[1]	PIN_AC2	DRAM_DQ[4]	PIN_AD2
DRAM_DQ[10]	PIN_AF2	DRAM_DQ[5]	PIN_AD3
DRAM_DQ[11]	PIN_AF3	DRAM_DQ[6]	PIN_AE1
DRAM_DQ[12]	PIN_AG2	DRAM_DQ[7]	PIN_AE2
DRAM_DQ[13]	PIN_AG3	DRAM_DQ[8]	PIN_AE3
DRAM_DQ[14]	PIN_AH1	DRAM_DQ[9]	PIN_AF1
DRAM_DQ[15]	PIN_AH2	oDRAM0_A[0]	PIN_AA4
DRAM_DQ[16]	PIN_U1	oDRAM0_A[1]	PIN_AA5
DRAM_DQ[17]	PIN_U2	oDRAM0_A[10]	PIN_Y8
DRAM_DQ[18]	PIN_U3	oDRAM0_A[11]	PIN_AE4
DRAM_DQ[19]	PIN_V2	oDRAM0_A[12]	PIN_AF4
DRAM_DQ[2]	PIN_AC3	oDRAM0_A[2]	PIN_AA6

DE2-70 上 EP2C70F896C6 的引脚分配表续 1：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
oDRAM0_A[3]	PIN_AB5	oDRAM1_A[7]	PIN_W4
oDRAM0_A[4]	PIN_AB7	oDRAM1_A[8]	PIN_W7
oDRAM0_A[5]	PIN_AC4	oDRAM1_A[9]	PIN_W8
oDRAM0_A[6]	PIN_AC5	oDRAM1_BA[0]	PIN_T7
oDRAM0_A[7]	PIN_AC6	oDRAM1_BA[1]	PIN_T8
oDRAM0_A[8]	PIN_AD4	oDRAM1_CAS_N	PIN_N8
oDRAM0_A[9]	PIN_AC7	oDRAM1_CKE	PIN_L10
oDRAM0_BA[0]	PIN_AA9	oDRAM1_CLK	PIN_G5
oDRAM0_BA[1]	PIN_AA10	oDRAM1_CS_N	PIN_P9
oDRAM0_CAS_N	PIN_W10	oDRAM1_LDQM0	PIN_M10
oDRAM0_CKE	PIN_AA8	oDRAM1_RAS_N	PIN_N9
oDRAM0_CLK	PIN_AD6	oDRAM1_UDQM1	PIN_U8
oDRAM0_CS_N	PIN_Y10	oDRAM1_WE_N	PIN_M9
oDRAM0_LDQM0	PIN_V9	oENET_CMD	PIN_B27
oDRAM0_RAS_N	PIN_Y9	oENET_CS_N	PIN_C28
oDRAM0_UDQM1	PIN_AB6	ENET_D[0]	PIN_A23
oDRAM0_WE_N	PIN_W9	ENET_D[1]	PIN_C22
oDRAM1_A[0]	PIN_T5	ENET_D[10]	PIN_B25
oDRAM1_A[1]	PIN_T6	ENET_D[11]	PIN_A25
oDRAM1_A[10]	PIN_T4	ENET_D[12]	PIN_C24
oDRAM1_A[11]	PIN_Y4	ENET_D[13]	PIN_B24
oDRAM1_A[12]	PIN_Y7	ENET_D[14]	PIN_A24
oDRAM1_A[2]	PIN_U4	ENET_D[15]	PIN_B23
oDRAM1_A[3]	PIN_U6	ENET_D[2]	PIN_B22
oDRAM1_A[4]	PIN_U7	ENET_D[3]	PIN_A22
oDRAM1_A[5]	PIN_V7	ENET_D[4]	PIN_B21
oDRAM1_A[6]	PIN_V8	ENET_D[5]	PIN_A21

DE2-70 上 EP2C70F896C6 的引脚分配表续 2：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
ENET_D[6]	PIN_B20	oFLASH_A[6]	PIN_AH22
ENET_D[7]	PIN_A20	oFLASH_A[7]	PIN_AF22
ENET_D[8]	PIN_B26	oFLASH_A[8]	PIN_AH27
ENET_D[9]	PIN_A26	oFLASH_A[9]	PIN_AJ27
iENET_INT	PIN_C27	oFLASH_BYTE_N	PIN_Y29
oENET_IOR_N	PIN_A28	oFLASH_CE_N	PIN_AG28
oENET_IOW_N	PIN_B28	FLASH_DQ[0]	PIN_AF29
oENET_RESET_N	PIN_B29	FLASH_DQ[1]	PIN_AE28
iEXT_CLOCK	PIN_R29	FLASH_DQ[10]	PIN_AD29
oFLASH_A[0]	PIN_AF24	FLASH_DQ[11]	PIN_AC28
oFLASH_A[1]	PIN_AG24	FLASH_DQ[12]	PIN_AC30
oFLASH_A[10]	PIN_AH26	FLASH_DQ[13]	PIN_AB30
oFLASH_A[11]	PIN_AJ26	FLASH_DQ[14]	PIN_AA30
oFLASH_A[12]	PIN_AK26	FLASH_DQ15_AM1	PIN_AE24
oFLASH_A[13]	PIN_AJ25	FLASH_DQ[2]	PIN_AE30
oFLASH_A[14]	PIN_AK25	FLASH_DQ[3]	PIN_AD30
oFLASH_A[15]	PIN_AH24	FLASH_DQ[4]	PIN_AC29
oFLASH_A[16]	PIN_AG25	FLASH_DQ[5]	PIN_AB29
oFLASH_A[17]	PIN_AF21	FLASH_DQ[6]	PIN_AA29
oFLASH_A[18]	PIN_AD21	FLASH_DQ[7]	PIN_Y28
oFLASH_A[19]	PIN_AK28	FLASH_DQ[8]	PIN_AF30
oFLASH_A[2]	PIN_AE23	FLASH_DQ[9]	PIN_AE29
oFLASH_A[20]	PIN_AJ28	oFLASH_OE_N	PIN_AG29
oFLASH_A[21]	PIN_AE20	oFLASH_RST_N	PIN_AH28
oFLASH_A[3]	PIN_AG23	iFLASH_RY_N	PIN_AH30
oFLASH_A[4]	PIN_AF23	oFLASH_WE_N	PIN_AJ29
oFLASH_A[5]	PIN_AG22	oFLASH_WP_N	PIN_AH29

DE2-70 上 EP2C70F896C6 的引脚分配表续 3：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
GPIO_CLKIN_N0	PIN_T25	GPIO_0[26]	PIN_L21
GPIO_CLKIN_N1	PIN_AH14	GPIO_0[27]	PIN_M22
GPIO_CLKIN_P0	PIN_T24	GPIO_0[28]	PIN_N22
GPIO_CLKIN_P1	PIN_AG15	GPIO_0[29]	PIN_N25
GPIO_CLKOUT_N0	PIN_H23	GPIO_0[3]	PIN_D29
GPIO_CLKOUT_N1	PIN_AF27	GPIO_0[30]	PIN_N21
GPIO_CLKOUT_P0	PIN_G24	GPIO_0[31]	PIN_N24
GPIO_CLKOUT_P1	PIN_AF28	GPIO_1[0]	PIN_G27
GPIO_0[0]	PIN_C30	GPIO_1[1]	PIN_G28
GPIO_0[1]	PIN_C29	GPIO_1[2]	PIN_H27
GPIO_0[10]	PIN_F29	GPIO_1[3]	PIN_L24
GPIO_0[11]	PIN_G29	GPIO_1[4]	PIN_H28
GPIO_0[12]	PIN_F30	GPIO_1[5]	PIN_L25
GPIO_0[13]	PIN_G30	GPIO_1[6]	PIN_K27
GPIO_0[14]	PIN_H29	GPIO_1[7]	PIN_L28
GPIO_0[15]	PIN_H30	GPIO_0[4]	PIN_E27
GPIO_0[16]	PIN_J29	GPIO_1[8]	PIN_K28
GPIO_0[17]	PIN_H25	GPIO_1[9]	PIN_L27
GPIO_0[18]	PIN_J30	GPIO_1[10]	PIN_K29
GPIO_0[19]	PIN_H24	GPIO_1[11]	PIN_M25
GPIO_0[2]	PIN_E28	GPIO_1[12]	PIN_K30
GPIO_0[20]	PIN_J25	GPIO_1[13]	PIN_M24
GPIO_0[21]	PIN_K24	GPIO_1[14]	PIN_L29
GPIO_0[22]	PIN_J24	GPIO_1[15]	PIN_L30
GPIO_0[23]	PIN_K25	GPIO_1[16]	PIN_P26
GPIO_0[24]	PIN_L22	GPIO_1[17]	PIN_P28
GPIO_0[25]	PIN_M21	GPIO_0[5]	PIN_D28

DE2-70 上 EP2C70F896C6 的引脚分配表续 4：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
GPIO_1[18]	PIN_P25	oHEX1_D[1]	PIN_AE16
GPIO_1[19]	PIN_P27	oHEX1_D[2]	PIN_AF16
GPIO_1[20]	PIN_M29	oHEX1_D[3]	PIN_AG16
GPIO_1[21]	PIN_R26	oHEX1_D[4]	PIN_AE17
GPIO_1[22]	PIN_M30	oHEX1_D[5]	PIN_AF17
GPIO_1[23]	PIN_R27	oHEX1_D[6]	PIN_AD17
GPIO_1[24]	PIN_P24	oHEX1_DP	PIN_AC17
GPIO_1[25]	PIN_N28	oHEX2_D[0]	PIN_AE7
GPIO_1[26]	PIN_P23	oHEX2_D[1]	PIN_AF7
GPIO_1[27]	PIN_N29	oHEX2_D[2]	PIN_AH5
GPIO_0[6]	PIN_E29	oHEX2_D[3]	PIN_AG4
GPIO_1[28]	PIN_R23	oHEX2_D[4]	PIN_AB18
GPIO_1[29]	PIN_P29	oHEX2_D[5]	PIN_AB19
GPIO_1[30]	PIN_R22	oHEX2_D[6]	PIN_AE19
GPIO_1[31]	PIN_P30	oHEX2_DP	PIN_AC19
GPIO_0[7]	PIN_G25	oHEX3_D[0]	PIN_P6
GPIO_0[8]	PIN_E30	oHEX3_D[1]	PIN_P4
GPIO_0[9]	PIN_G26	oHEX3_D[2]	PIN_N10
oHEX0_D[0]	PIN_AE8	oHEX3_D[3]	PIN_N7
oHEX0_D[1]	PIN_AF9	oHEX3_D[4]	PIN_M8
oHEX0_D[2]	PIN_AH9	oHEX3_D[5]	PIN_M7
oHEX0_D[3]	PIN_AD10	oHEX3_D[6]	PIN_M6
oHEX0_D[4]	PIN_AF10	oHEX3_DP	PIN_M4
oHEX0_D[5]	PIN_AD11	oHEX4_D[0]	PIN_P1
oHEX0_D[6]	PIN_AD12	oHEX4_D[1]	PIN_P2
oHEX0_DP	PIN_AF12	oHEX4_D[2]	PIN_P3
oHEX1_D[0]	PIN_AG13	oHEX4_D[3]	PIN_N2

DE2-70 上 EP2C70F896C6 的引脚分配表续 5：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
oHEX4_D[4]	PIN_N3	oHEX7_DP	PIN_G2
oHEX4_D[5]	PIN_M1	oI2C_SCLK	PIN_J18
oHEX4_D[6]	PIN_M2	I2C_SDAT	PIN_H18
oHEX4_DP	PIN_L6	iIRDA_RXD	PIN_W22
oHEX5_D[0]	PIN_M3	oIRDA_TXD	PIN_W21
oHEX5_D[1]	PIN_L1	iKEY[0]	PIN_T29
oHEX5_D[2]	PIN_L2	iKEY[1]	PIN_T28
oHEX5_D[3]	PIN_L3	iKEY[2]	PIN_U30
oHEX5_D[4]	PIN_K1	iKEY[3]	PIN_U29
oHEX5_D[5]	PIN_K4	oLCD_BLON	PIN_G3
oHEX5_D[6]	PIN_K5	LCD_D[0]	PIN_E1
oHEX5_DP	PIN_K6	LCD_D[1]	PIN_E3
oHEX6_D[0]	PIN_H6	LCD_D[2]	PIN_D2
oHEX6_D[1]	PIN_H4	LCD_D[3]	PIN_D3
oHEX6_D[2]	PIN_H7	LCD_D[4]	PIN_C1
oHEX6_D[3]	PIN_H8	LCD_D[5]	PIN_C2
oHEX6_D[4]	PIN_G4	LCD_D[6]	PIN_C3
oHEX6_D[5]	PIN_F4	LCD_D[7]	PIN_B2
oHEX6_D[6]	PIN_E4	oLCD_EN	PIN_E2
oHEX6_DP	PIN_K2	oLCD_ON	PIN_F1
oHEX7_D[0]	PIN_K3	oLCD_RS	PIN_F2
oHEX7_D[1]	PIN_J1	oLCD_RW	PIN_F3
oHEX7_D[2]	PIN_J2	oLEDR[0]	PIN_AJ6
oHEX7_D[3]	PIN_H1	oLEDR[1]	PIN_AK5
oHEX7_D[4]	PIN_H2	oLEDR[10]	PIN_AC13
oHEX7_D[5]	PIN_H3	oLEDR[11]	PIN_AB13
oHEX7_D[6]	PIN_G1	oLEDR[12]	PIN_AC12

DE2-70 上 EP2C70F896C6 的引脚分配表续 6：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
oLEDR[13]	PIN_AB12	OTG_D[10]	PIN_D6
oLEDR[14]	PIN_AC11	OTG_D[11]	PIN_E7
oLEDR[15]	PIN_AD9	OTG_D[12]	PIN_D7
oLEDR[16]	PIN_AD8	OTG_D[13]	PIN_E8
oLEDR[17]	PIN_AJ7	OTG_D[14]	PIN_D9
oLEDG[8]	PIN_AC14	OTG_D[15]	PIN_G10
oLEDG[0]	PIN_W27	OTG_D[2]	PIN_G11
oLEDR[2]	PIN_AJ5	OTG_D[3]	PIN_F11
oLEDG[1]	PIN_W25	OTG_D[4]	PIN_J12
oLEDG[2]	PIN_W23	OTG_D[5]	PIN_H12
oLEDG[3]	PIN_Y27	OTG_D[6]	PIN_H13
oLEDG[4]	PIN_Y24	OTG_D[7]	PIN_G13
oLEDG[5]	PIN_Y23	OTG_D[8]	PIN_D4
oLEDG[6]	PIN_AA27	OTG_D[9]	PIN_D5
oLEDG[7]	PIN_AA24	oOTG_DACK0_N	PIN_D12
oLEDR[3]	PIN_AJ4	oOTG_DACK1_N	PIN_E12
oLEDR[4]	PIN_AK3	iOTG_DREQ0	PIN_G12
oLEDR[5]	PIN_AH4	iOTG_DREQ1	PIN_F12
oLEDR[6]	PIN_AJ3	OTG_FSPEED	PIN_F7
oLEDR[7]	PIN_AJ2	iOTG_INT0	PIN_F13
oLEDR[8]	PIN_AH3	iOTG_INT1	PIN_J13
oLEDR[9]	PIN_AD14	OTG_LSPEED	PIN_F8
oOTG_A[0]	PIN_E9	oOTG_OE_N	PIN_D10
oOTG_A[1]	PIN_D8	oOTG_RESET_N	PIN_H14
oOTG_CS_N	PIN_E10	oOTG_WE_N	PIN_E11
OTG_D[0]	PIN_H10	PS2_KBCLK	PIN_F24
OTG_D[1]	PIN_G9	PS2_KBDAT	PIN_E24

DE2-70 上 EP2C70F896C6 的引脚分配表续 7：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
PS2_MSCLK	PIN_D26	oSRAM_ADV_N	PIN_AD16
PS2_MSDAT	PIN_D25	oSRAM_BE_N[0]	PIN_AC21
oSD_CLK	PIN_T26	oSRAM_BE_N[1]	PIN_AC20
SD_CMD	PIN_W28	oSRAM_BE_N[2]	PIN_AD20
SD_DAT	PIN_W29	oSRAM_BE_N[3]	PIN_AH20
SD_DAT3	PIN_Y30	oSRAM_CE1_N	PIN_AH19
oSRAM_A[0]	PIN_AG8	oSRAM_CE2	PIN_AG19
oSRAM_A[1]	PIN_AF8	oSRAM_CE3_N	PIN_AD22
oSRAM_A[10]	PIN_AF14	oSRAM_CLK	PIN_AD7
oSRAM_A[11]	PIN_AG14	SRAM_DPA[0]	PIN_AK9
oSRAM_A[12]	PIN_AE15	SRAM_DPA[1]	PIN_AJ23
oSRAM_A[13]	PIN_AF15	SRAM_DPA[2]	PIN_AK20
oSRAM_A[14]	PIN_AC16	SRAM_DPA[3]	PIN_AJ9
oSRAM_A[15]	PIN_AF20	SRAM_DQ[0]	PIN_AH10
oSRAM_A[16]	PIN_AG20	SRAM_DQ[1]	PIN_AJ10
oSRAM_A[17]	PIN_AE11	SRAM_DQ[10]	PIN_AH17
oSRAM_A[18]	PIN_AF11	SRAM_DQ[11]	PIN_AJ18
oSRAM_A[2]	PIN_AH7	SRAM_DQ[12]	PIN_AH18
oSRAM_A[3]	PIN_AG7	SRAM_DQ[13]	PIN_AK19
oSRAM_A[4]	PIN_AG6	SRAM_DQ[14]	PIN_AJ19
oSRAM_A[5]	PIN_AG5	SRAM_DQ[15]	PIN_AK23
oSRAM_A[6]	PIN_AE12	SRAM_DQ[16]	PIN_AJ20
oSRAM_A[7]	PIN_AG12	SRAM_DQ[17]	PIN_AK21
oSRAM_A[8]	PIN_AD13	SRAM_DQ[18]	PIN_AJ21
oSRAM_A[9]	PIN_AE13	SRAM_DQ[19]	PIN_AK22
oSRAM_ADSC_N	PIN_AG17	SRAM_DQ[2]	PIN_AK10
oSRAM_ADSP_N	PIN_AC18	SRAM_DQ[20]	PIN_AJ22

DE2-70 上 EP2C70F896C6 的引脚分配表续 8：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
SRAM_DQ[21]	PIN_AH15	iSW[14]	PIN_L5
SRAM_DQ[22]	PIN_AJ15	iSW[15]	PIN_L4
SRAM_DQ[23]	PIN_AJ16	iSW[16]	PIN_L7
SRAM_DQ[24]	PIN_AK14	iSW[17]	PIN_L8
SRAM_DQ[25]	PIN_AJ14	iSW[2]	PIN_AB25
SRAM_DQ[26]	PIN_AJ13	iSW[3]	PIN_AC27
SRAM_DQ[27]	PIN_AH13	iSW[4]	PIN_AC26
SRAM_DQ[28]	PIN_AK12	iSW[5]	PIN_AC24
SRAM_DQ[29]	PIN_AK7	iSW[6]	PIN_AC23
SRAM_DQ[3]	PIN_AJ11	iSW[7]	PIN_AD25
SRAM_DQ[30]	PIN_AJ8	iSW[8]	PIN_AD24
SRAM_DQ[31]	PIN_AK8	iSW[9]	PIN_AE27
SRAM_DQ[4]	PIN_AK11	iTD1_CLK27	PIN_G15
SRAM_DQ[5]	PIN_AH12	iTD1_D[0]	PIN_A6
SRAM_DQ[6]	PIN_AJ12	iTD1_D[1]	PIN_B6
SRAM_DQ[7]	PIN_AH16	iTD1_D[2]	PIN_A5
SRAM_DQ[8]	PIN_AK17	iTD1_D[3]	PIN_B5
SRAM_DQ[9]	PIN_AJ17	iTD1_D[4]	PIN_B4
oSRAM_GW_N	PIN_AG18	iTD1_D[5]	PIN_C4
oSRAM_OE_N	PIN_AD18	iTD1_D[6]	PIN_A3
oSRAM_WE_N	PIN_AF18	iTD1_D[7]	PIN_B3
iSW[0]	PIN_AA23	iTD1_HS	PIN_E13
iSW[1]	PIN_AB26	oTD1_RESET_N	PIN_D14
iSW[10]	PIN_W5	iTD1_VS	PIN_E14
iSW[11]	PIN_V10	iTD2_CLK27	PIN_H15
iSW[12]	PIN_U9	iTD2_D[0]	PIN_C10
iSW[13]	PIN_T9	iTD2_D[1]	PIN_A9

DE2-70 上 EP2C70F896C6 的引脚分配表续 9：

信号名称	FPGA 引脚	信号名称	FPGA 引脚
iTD2_D[2]	PIN_B9	oVGA_G[2]	PIN_A11
iTD2_D[3]	PIN_C9	oVGA_G[3]	PIN_C12
iTD2_D[4]	PIN_A8	oVGA_G[4]	PIN_B12
iTD2_D[5]	PIN_B8	oVGA_G[5]	PIN_A12
iTD2_D[6]	PIN_A7	oVGA_G[6]	PIN_C13
iTD2_D[7]	PIN_B7	oVGA_G[7]	PIN_B13
iTD2_HS	PIN_E15	oVGA_G[8]	PIN_B14
oTD2_RESET_N	PIN_B10	oVGA_G[9]	PIN_A14
iTD2_VS	PIN_D15	oVGA_HS	PIN_J19
oUART_CTS	PIN_G22	oVGA_R[0]	PIN_D23
iUART_RTS	PIN_F23	oVGA_R[1]	PIN_E23
iUART_RXD	PIN_D21	oVGA_R[2]	PIN_E22
oUART_TXD	PIN_E21	oVGA_R[3]	PIN_D22
oVGA_B[0]	PIN_B16	oVGA_R[4]	PIN_H21
oVGA_B[1]	PIN_C16	oVGA_R[5]	PIN_G21
oVGA_B[2]	PIN_A17	oVGA_R[6]	PIN_H20
oVGA_B[3]	PIN_B17	oVGA_R[7]	PIN_F20
oVGA_B[4]	PIN_C18	oVGA_R[8]	PIN_E20
oVGA_B[5]	PIN_B18	oVGA_R[9]	PIN_G20
oVGA_B[6]	PIN_B19	oVGA_SYNC_N	PIN_B15
oVGA_B[7]	PIN_A19	oVGA_VS	PIN_H19
oVGA_B[8]	PIN_C19		
oVGA_B[9]	PIN_D19		
oVGA_BLANK_N	PIN_C15		
oVGA_CLOCK	PIN_D24		
oVGA_G[0]	PIN_A10		
oVGA_G[1]	PIN_B11		

附录 B:

DE2-70 的开发板简介

图 B.1 所示为 DE2-70 开发板的照片 ,图中标出了开发板的布局以及板上关键组件的位置。

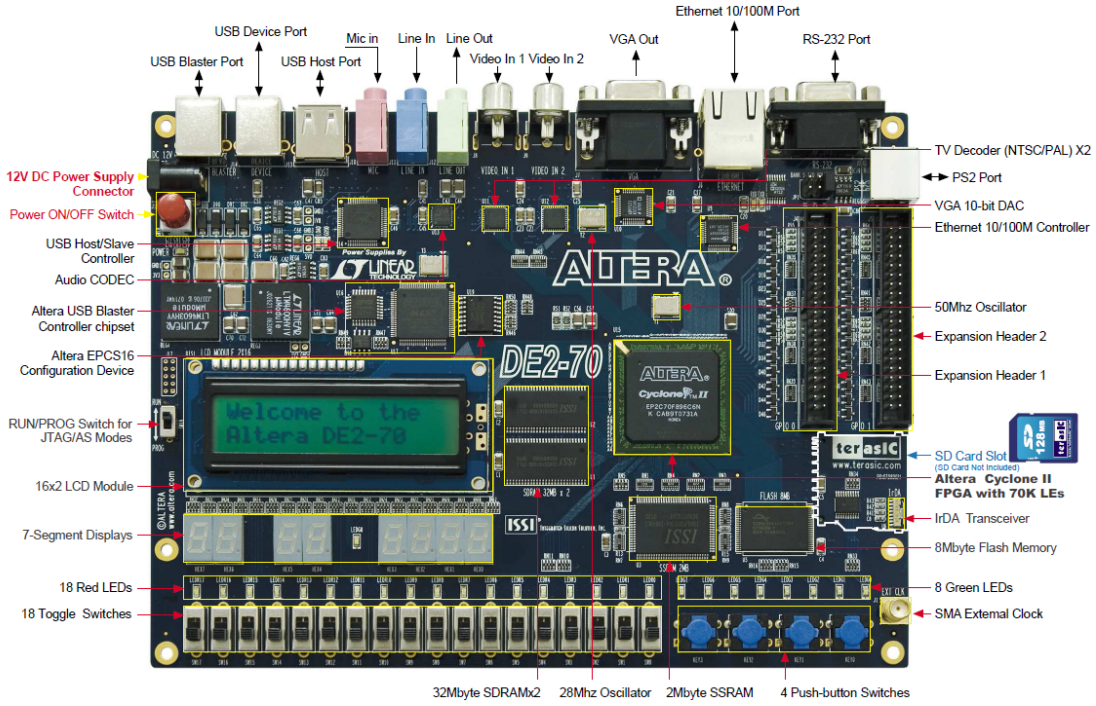


图 **B.1 DE2-70** 开发板

DE2-70 开发板为用户提供了大量组件 ,使得开发者可以轻松的实现多种多媒体设计项目。

（1）DE2-70 开发板提供了如下硬件配置：

（1）Altera Cyclone II 2C70 FPGA 芯片。

（2） Altera 系列配置芯片 EPCS16。

（3）用于编程和 API 控制的 USB Blaster(在板上);支持 JTGA 和 Active Serial（AS）两种编程模式。

（4） 2Mbyte SSRAM。

（5） 2 个 32-Mbyte SDRAM。

（6） 8-Mbyte Flash memory。

（7） SD Card 槽。

（8） 4 个按钮开关。

（9） 18 个拨动开关。

（10） 18 个红色 LEDs。

（11） 9 个绿色 LEDs。

（12） 50-MHz 振荡器和 28.63-MHz 振荡器提供时钟源。

（13） 24-bit CD 音频 CODEC（编码/解码器） 并带有 line-in, line-out, 和 microphone-in 接口。

（14） VGA DAC (数模转换器)，并带有 VGA-out 接口。

（15） 2 个 TV 解码器 (NTSC/PAL/SECAM) 和 TV-in 接口。

（16） 10/100 网络控制器并带有接口。

（17） USB Host/Slave 控制器，并带有 USB A 型和 B 型接口。

（18） RS-232 接口。

（19） PS/2 鼠标/键盘接口。

（20） IrDA 转换器。

（21） 1 个 SMA 接口。

（22） 2 个带有二极管保护的 40-pin 外部扩展接口。

（23） DE2-70 的结构框图

（24） 图 B.2 显示了 DE2-70 的结构框图，为了便于使用，所有的外围部件都通过 FPGA 连接起来，这样，用户就可以通过配置 FPGA 来完成所有系统的控制。

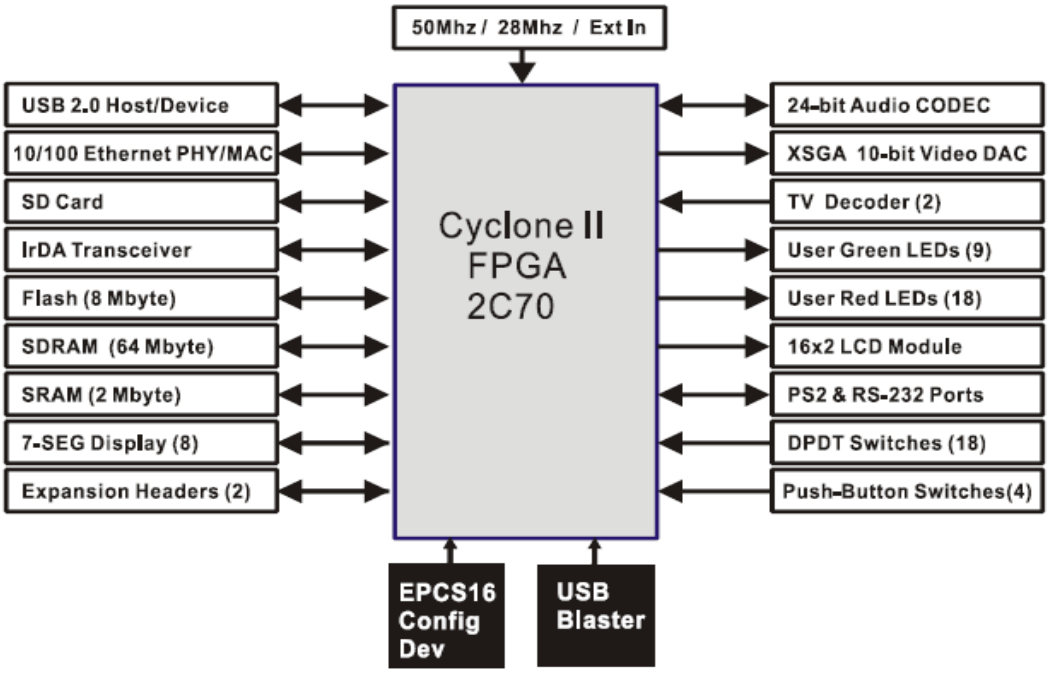


图 B.2 显示了 DE2-70 的结构框图

图中各部件的简单特性介绍如下：

Cyclone II 2C70 FPGA： 68,416 个 LE， 250 个 M4K RAM 块，总计 1,152,000 RAM bits，150 个片内乘法器，4 个 4 PLL， 622 个用户 I/O 引脚，896 引脚 FineLine BGA 封装。

配置芯片和 USB Blaster：配置芯片为 EPCS16，用于编程和 API 控制的 USB Blaster，支持 JTAG 和 AS 两种编程模式。

SSRAM： 2-Mbyte （512K x 36 bits）标准同步 SRAM，可用于 Nios II 处理器的存储器或直接由 FPGA 控制使用。

SDRAM：有 2 个 32-Mbyte（4M x 16 bits x 4 banks） 单数据率同步动态 RAM，可用于 Nios II 处理器的存储器或直接由 FPGA 控制使用。

Flash memory：· 8-Mbyte NOR Flash 存储器，支持字节和字两种模式，可用于 Nios II 处理器的存储器或直接由 FPGA 控制使用。

SD 卡插座 :支持 SPI 和 1-bit SD 模式 ,可用于 Nios II 处理器的存储器。

按钮开关：4 个按钮开关，以由施密特触发器消抖，通常为高，按下时输出一个低电平。

拨动开关：18 个用于输入的拨动开关，向上拨为‘1’，向下拨为‘0’。

Clock inputs：有 50-MHz 和 28.63-MHz 振荡器和 SMA 外部时钟输入。

Audio CODEC： Wolfson WM8731 24-bit sigma-delta 音频 CODEC，带有 line-in, line-out, 和 microphone-in 接口 ，采样频率 8 to 96 KHz，用于各种音频输入/输出。

VGA output： ADV7123 140-MHz triple 10-bit high-speed video DAC， 15 脚高密 D-sub 接口，支持 1600 x 1200 at 100-Hz 刷新频率，可作 FPGA 的输出显示用。

NTSC/PAL/ SECAM TV 解码电路：用了 2 个 ADV7180 Multi-format SDTV Video Decoders，用于各种视频输入/输出。

10/100 以太网控制器：带有通用处理器接口的集成 MAC 和 PHY ，支持 100Base-T 和 10Base-T 应用，带有 auto-MDIX，支持 10 Mb/s 和 100 Mb/s 全双工操作，完全兼容 IEEE 802.3u 规范，支持 IP/TCP/UDP 求和校验半双工模式背压流控。

USB 主/从 控制器：满足国际串行总线标准 Rev. 2.0，支持全速和低速数

据传送。支持主/从两个 USB 口（A 口是主口，B 口是从口），支持大部分处理器的高速并行接口，支持 PIO 和 DMA 操作。

串行接口：一个 RS-232（DB-9 串行接口） 接口，一个用于接鼠标或键盘的 PS/2 接口。

IrDA 转换器：含有一个 115.2-kb/s 的红外转换器，32 mA LED 驱动电流，集成 EMI 保护，满足 IEC825-1 标准，边缘输入检测。

2 个 40 引脚的扩展头：Cyclone II 的 72 个 I/O 引脚，和 8 个电源端和地端接到两 40 引脚的扩展头上，有二极管和电阻保护。