武汉轻工大学

计算机组成原理

课程设计报告

题目：单周期 CPU 及主机系统的设计与实现

专　　业：	计算机科学与技术
班　　级：	计算机 1501
学　　号：	1505110017
姓　　名：	冯帮尉
完成日期：	2018.01.09
指导教师：	蒋丽华

武汉轻工大学计算机组成原理课程设计报告

目录

武汉轻工大学计算机组成原理课程设计报告

1 课程设计概述

1.1 课设目的

通过原理图设计计算机组成的各个部件，深入了解并掌握可编程芯片 FPGA 的设计技术，加强学生对《计算机组成原理》课程所学知识综合利用的能力，培养学生创造性思维能力和独立解决实际问题的能力。

1.2 课设要求

(1)能够全面地应用课程中所学的基本理论和基本方法，完成从设计单个单元部件到设计计算机主机系统的过渡。

(2)能力独立思考、独立查阅资料，独立设计规定的系统。

(3)能够独立地完成实施过程，包括设计、布线、测试和排除故障。

1.3 课设任务

(1) 制定出详细设计方案；

(2) 通过原理图完成规定的设计任务，然后进行编译和仿真，保证设计的正确性；

(3) 生成 SOF 配置文件，下载到 DE2-115 开发板，通过实际线路进行验证；

(4) 对复杂系统的设计采取模块化、层次化的设计方法；

(5) 撰写设计报告，并对存在的问题进行分析、提出改进意见。

1.4 实验环境

开发环境 QuartusII:

QuartusII 平台是完全集成化、易学易用的可编程逻辑设计环境，主要用于设计新器件和中大规模 CPLD(复杂可编程逻辑器件)/FPGA(现场可编程逻辑器件)。该设计软件有原理图输入、硬件描述语言等多种设计方式，利用其所提供的标准门电路、芯片等逻辑器件，可以实现数字电路系统和计算机硬件系统从设计输入、编辑、编译、模拟仿真测试、封装到下载的全过程。QuartusII 平台可以保证所设计系统的可靠性、高效性和灵活性，其强大的图形界面和完整的帮助文档，使设计者能够轻松快速地掌握

和使用该 EDA 平台。在进行实验时，只需要在计算机上安装 QuartusII 设计软件即可进行计算机各功能部件及计算机主机系统的设计。

DE2-115 开发板：

DE2-115 系列平台一直位居于国内外 FPGA 教育开发平台的领先地位。因其拥有适应各种应用需求的丰富接口及工业等级的设计资源，成为全球 1000 所名校实验室中的首选。延续 DE2 系列开发平台之领先和成功，友晶科技推出最新款搭载 Cyclone IV E 芯片之 DE2-115 开发平台。此款开发平台不仅提供客户一个低功耗，丰富逻辑资源，大容量存储器以及 DSP 功能的选择，更因为我们了解客户对移动视频、语音、数据接入及高品质图像渴望，而搭配了丰富的外围接口，满足各类型开发之需求。

DE2-115 开发平台所配备之 Cyclone IV EP4CE115 为 Cyclone IV FPGA 系列之最大器件。此芯片具有 114,480 个逻辑单元（LEs）、高达 3.9 Mbps 的随机存储器、内嵌266个 乘法器，以及低功耗等特质，提供最强大的心脏。

接口部分除了承袭广受欢迎的 DE2 系列多媒体开发平台界面外, 还新增了接口以支持包括千兆以太网（GbE）在内的主流协议。HSMC 插槽提供通过 HSMC 子卡和电缆之附加功能和连接。亦可通过 HSMC 电缆将多片的 FPGA 板上透过 HSMC 插槽连接起来，以实现大型的 ASIC 原型开发之需求。

2 MIPS 指令和 CPU 设计思路

设计需求要操作计算机硬件，就必须使用该计算机的语言，即该计算机的指令。所有指令组成的指令表称为指令集。指令集是计算机程序员可见的指令集合，是计算机系统结构中的重要组成部分。指令集和硬件系统结构共同决定了计算机系统结构。

指令集定义了软件和硬件之间的接口，如图 2-1 所示。

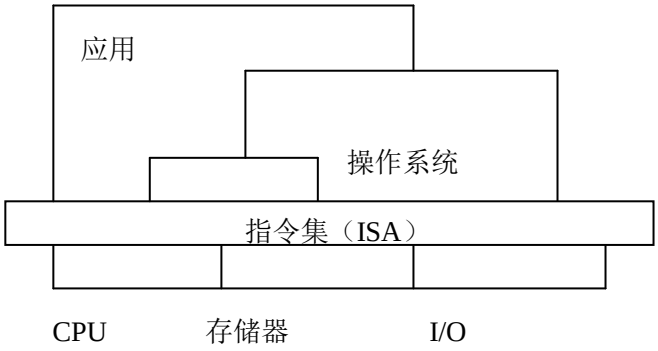


图 2-1　指令集在计算机系统中的位置

指令集定义如下内容：

(1) 寄存器堆的组织

(2) 数据类型和数据结构

(3) 指令格式

(4) 指令集合

(5) 数据的寻址和访问模式

(6) 外部操作：I/O 操作和中断等

MIPS 体系结构家族包含如下几代，每一代之间都有一些区别：

(1) MIPS I—32 位 CPU 使用的指令集。仍然被广泛使用着。

(2) MIPS II—为 R6000 机器定义，包含了一些细微的改进。

(3) MIPS III—R4xxx 的 64 位指令集。

(4) MIPS IV—MIPS III 的一个细微的升级。定义在 R10000 和 R5000 中。

(5) MIPS32—MIPS32 指令集基于 MIPS II 指令集，加入了 MIPS III、MIPS IV 和 MIPS V 中的一些指令来提高代码生成效率和数据转移效率。

在本书中，讲解主要基于 MIPS32 的指令集。本章简要地介绍 MIPS 指令集和单周期中央处理器 CPU 的逻辑设计。

2.1 MIPS 寄存器堆

MIPS 指令的操作数一般来自寄存器。寄存器是计算机结构的基本组成部分。在 MIPS32 体系结构中，共有 32 个大小为 32 位的寄存器，分别编号为 0 到 31。另外，MIPS32 体系结构中还有三个 32 位的寄存器 HI、LO 和 PC。HI 和 LO 用来保存乘法指令或者除法指令计算结果。PC 为程序计数器，用来保存当前指令地址。在编写程序时，通常使用助记符来表示这些寄存器，如表 3-1 所示。通常情况下，需要编译器来把高级语言中的变量分配到各个寄存器中。

表 3-1 MIPS 通用寄存器堆

寄存器名	寄存器编号	用途
$zero	0	常数 0
$at	1	保留给汇编器(assembler)，通常作为汇编器的临时变量
$v0-$v1	2~3	表达式计算或者过程调用返回结果
$a0-$a1	4~7	过程调用参数 1~3
$t0-$t7	8~15	临时变量，过程调用时不需要保存和恢复
$s0-$s7	16~23	过程寄存器变量。在过程返回之前，必须保存和恢复使用过的变量，从而保证调用过程的这些值没有变化
$t8-$t9	24~25	临时变量，过程调用时不需要保存和恢复
$k0-$k1	26~27	保留给操作系统核心，通常被中断或例外处理程序用来保存一些系统参数
$gp	28	全局指针，一些运行系统维护这个指针来更方便地存取“static”和“extern”变量
$sp	29	堆栈指针
$s8/$fp	30	第 9 个过程寄存器变量，过程调用时用做帧指针
$ra	31	过程的返回地址

例 3.1 将下面的表达式编译为 MIPS 指令：

a=b+c

变量 a，b，c 可以分配给寄存器 s0，t0，t1。

编译结果为：

add $s0，$t0，$t1

2.2 指令格式

2.2.1 指令格式

每一条 MIPS 指令是一个 32 位字。MIPS 指令集中共包括三种格式的指令，分别是立即数(immediate)类型(I 类型)指令、跳转(jump)类型(J 类型)指令和寄存器(register)类型(R 类型)指令，如图 2-2 所示。表 2-2 解释各指令域的意义。指令集的这种设计方法可以简化指令译码。

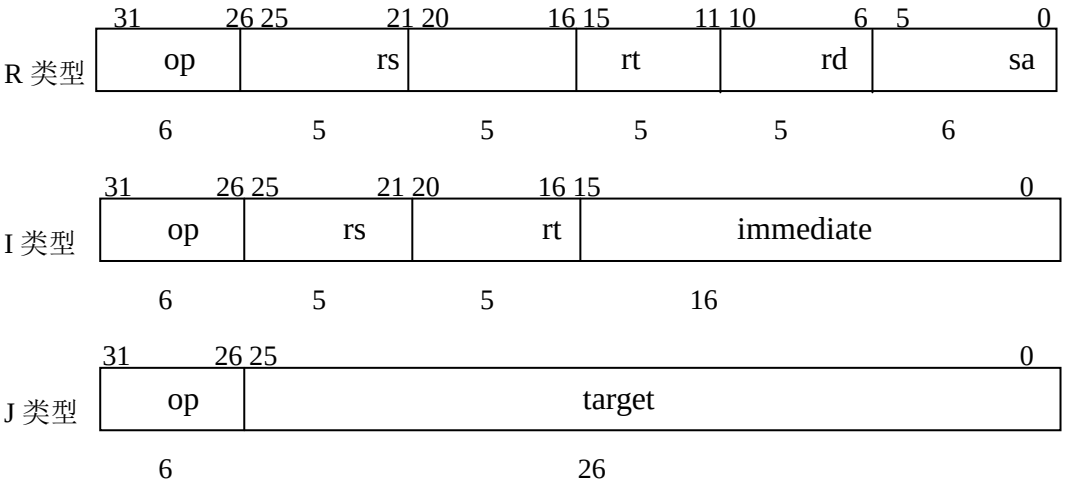


图 2-2　指令格式

表 2-2　CPU 指令域说明

域	描述
op	6 位指令主操作码
rd	5 位目的寄存器号
rs	5 位源寄存器号
rt	5 位目标(源/目的) 寄存器号，或在分支指令和其他一些指令中用来确定指令的功能
immediate	16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载(load)/数据保存(store)指令的数据地址字节偏移量和分支指令中相对程序计数器(PC)有符合偏移量
target	26 位，在向左移动 2 位后提供 jump 指令目标地址的低 28 位
sa	5 位偏移量
func	6 位功能码，在寄存器类型指令中用来指定指令的功能

2.2.2 主机系统指令

课程设计单周期 CPU 主机系统所涉及的 R 类型指令有 ADD、SUB、AND、OR、XOR、SLL、SRL、SRA、JR；I 指令有 ADDI、ANDI、ORI、XORI、LW、SW、BEQ、BNE、LUI；J 指令有 J、JAL，共有 20 条指令，各指令的格式编码如表 2-3 所示。

表 2-3　主机系统指令格式

域编号	[31..26]	[25..21]	[20..16]	[15..11]	[10..06]	[05..00]
R 形式	op	rs	rt	rd	sa	func
add	000000	rs	rt	rd	00000	100000
sub	000000	rs	rt	rd	00000	100010
and	000000	rs	rt	rd	00000	100100
or	000000	rs	rt	rd	00000	100101
xor	000000	rs	rt	rd	00000	100110
sll	000000	00000	rt	rd	sa	000000
srl	000000	00000	rt	rd	sa	000010
sra	000000	00000	rt	rd	sa	000011
jr	000000	rs	00000	00000	00000	001000
I 形式	op	rs	rt	Immediate(立即数)		
addi	001000	rs	rt	immediate		
andi	001100	rs	rt	immediate		
ori	001101	rs	rt	immediate		
xori	001110	rs	rt	immediate		
lw	100011	rs	rt	offset		
sw	101011	rs	rt	offset		
beq	000100	rs	rt	offset		
bne	000101	rs	rt	offset		
lui	001111	000000	rt	immediate		
J 形式	op	address				
j	000010	target				
jal	000011	target				

表 2-4 是这 20 条指令的详细说明。

表 2-4　指令说明

指令	格式	说明

add(加)	ADD rd,rs,rt	寄存器 rs 和寄存器 rt 中的整数相加，得到的 32 位结果保存到寄存器 rd 中，当结果产生整数溢出时，将产生整数溢出例外。
sub(减)	SUB rd,rs,rt	寄存器 rs 和寄存器 rt 中的整数相减，得到的 32 位结果保存到寄存器 rd 中，当结果产生整数溢出时，将产生整数溢出例外。
and(与)	AND rd,rs,rt	寄存器 rs 和寄存器 rt 中的整数相与，得到的 32 位结果保存到寄存器 rd 中。
or(或)	OR rd,rs,rt	寄存器 rs 和寄存器 rt 中的整数相或，得到的 32 位结果保存到寄存器 rd 中。
xor(异或)	XOR rd,rs,rt	寄存器 rs 和寄存器 rt 中的整数相异或，得到的 32 位结果保存到寄存器 rd 中。
sll(逻辑左移)	SLL rd,rt,sa	寄存器 rt 中的整数左移 sa 位，在低位中插入 0。得到的 32 位结果保存到寄存器 rd 中。
srl(逻辑右移)	SRL rd,rt,sa	寄存器 rt 中的整数右移 sa 位，在高位中插入 0。得到的 32 位结果保存到寄存器 rd 中。
sra(算术右移)	SRA rd,rt,sa	寄存器 rt 中的整数右移 sa 位，在高位中扩展原有整数的符号位。得到的 32 位结果保存到寄存器 rd 中。
jr(寄存器跳转)	JR rs	寄存器 rs 保存的是目标地址。在分支延迟槽中指令执行完成后转移到目标地址上。
addi(立即数加)	ADDI rt,rs,immediate	16 位的立即数为有符号数，经过符号扩展之后，与寄存器 rs 中的整数相加，产生一个 32 位的结果。然后，把该结果保存到寄存器 rt 中。当结果产生整数溢出时，将产生整数溢出例外。
andi(立即数与)	ANDI rt,rs,immediate	16 位的立即数为无符号数，经过 0 扩展之后，与寄存器 rs 中的整数相与，产生一个 32 位的结果。然后，把该结果保存到寄存器 rt 中。
ori(立即数或)	ORI rt,rs,immediate	16 位的立即数为无符号数，经过 0 扩展之后，与寄存器 rs 中的整数相或，产生一个 32 位的结果。然后，把该结果保存到寄存器 rt 中。
xori(立即数异或)	XORI rt,rs,immediate	16 位的立即数为无符号数，经过 0 扩展之后，与寄存器 rs 中的整数相异或，产生一个 32 位的结果。然后，把该结果保存到寄存器 rt 中。
lw(取字)	LW rt,offset(base)	offset 为有符号的 16 位数，符号扩展后与 base 寄存器中的内容相加，形成虚拟地址。主存储器中该虚拟地址指定的字保存到寄存器 rt 中。
sw(保存字)	SW rt,offset(base)	offset 为有符号的 16 位数，符号扩展后与 base 寄存器中的内容相加，形成虚拟地址。寄存器 rt 中字存入到主存储器中虚拟地址指定的位置中。
beq(相等即转移)	BEQ rs,rt,offset	16 位有符号数 offset 左移 2 位，与分支延迟槽中指令的 PC 相加，得到目标地址。如果寄存器 rs 中的整数和寄存器 rt 中的整数相等，则在分支延迟槽中指令执行完成后转移到目标地址上。
bne(不相等即转移)	BNE rs,rt,offset	16 位有符号数 offset 左移 2 位，与分支延迟槽中指令的 PC 相加，得到目标地址。如果寄存器 rs 中的整数和寄存器 rt 中的整数不相等，则在分支延迟槽中指令执行完成后转移到目标地址上。
lui(高位加载立即数)	LUI rt, immediate	16 位的立即数左移 16 位，并把低 16 位设置成 0.把产生的结果存入寄存器 rt 中。

j(跳转)	J target	26 位目标地址 target 左移 2 位，并使高 4 位与分支延迟槽中指令的地址高 4 位一致，形成目标地址。在分支延迟槽中指令执行完成后转移到目标地址上。
jal(跳转并链接)	JAL target	把分支延迟槽下一条指令的指令地址保存在寄存器 31 中，该地址作为过程调用返回地址链。26 位目标地址 target 左移 2 位，并使高 4 位与分支延迟槽中指令的地址高 4 位一致，形成目标地址。在分支延迟槽中指令执行完成后转移到目标地址上。

注意：分支指令之后的位置称为延迟槽。在Ⅰ类型指令中，16 位的立即数需要被扩展成 32 位数据。扩展的方法有两种：符号扩展和 0 扩展(表 2-5 所示)。符号扩展是把高 16 位置成与 16 位立即数最高位相同的值，即保持数据的符号不变。例如，16 位全 1 的立即数表示-1，符号扩展后仍是-1。因此，在有符号数据的运算中，比如 MIPS 算术运算指令，均对立即数进行符号扩展。0 扩展比较简单：高 16 位总是全 0。MIPS 逻辑运算指令对立即数进行 0 扩展。

表 2-5　立即数扩展

扩展	16 位立即数	扩展成 32 位数据
符号扩展	0xxxxxxxxxxxxxxx(正数)	0000000000000000 0xxxxxxxxxxxxxxx
	1xxxxxxxxxxxxxxx(负数)	1111111111111111 1xxxxxxxxxxxxxxx
0 扩展	xxxxxxxxxxxxxxxx	0000000000000000 xxxxxxxxxxxxxxxx

从表 3-3 和表 3-4 中可以看出以下几个问题：

(1)R 类型指令的操作码 op 为 000000，通过功能码 func 来区分不同的操作。而 I 类型和 J 类型指令的 op 各不相同，通过 op 来区分不同的指令。

(2)对于 R 类型指令，源数据一般为寄存器 rs 和寄存器 rt，目的寄存器为 rd。而对于除了分支指令的 I 类型指令，源数据一般为寄存器 rs 和立即数 immediate，目的寄存器为 rt。所以，需要注意寄存器 rt 的用法。分支指令的源数据为寄存器 rs 和寄存器 rt。

以下给出三类比较典型的 MIPS 指令的格式及其二进制编码的具体例子。

(1) add $17,$18,$19　　指令执行的操作是$17=$18+$19

6 位	5 位	5 位	5 位	5 位	6 位
op	rs	rt	rd	sa	func
0	18	19	17	0	32
000000	10010	10011	10001	00000	100000

(2) addi $17,$18,100　　指令执行的操作是$17=$18+100

6 位	5 位	5 位	16 位
op	rs	rt	immediate
8	18	17	100

001000　　10010　　10001　　0000 0000 0110 0100

(3) j 1000　　指令执行的操作是跳转至 1000(高 4 位与 PC 相同)

6 位	26 位
op	target
2	2500

00000　　00 0000 0000 0000 1001 1100 0100

有了以上类似指令的定义，就可将汇编程序转换成二进制代码程序，下面将讨论如何设计一个 CPU 使它能够执行该二进制代码程序，详细的逻辑电路设计将在以后各章给出。

2.3 CPU 设计思路

2.3.1 单周期 CPU 逻辑设计

单周期 CPU 的特点是每条指令的执行需要一个时钟周期，一条指令执行完再执行下一条指令。由于每个时钟周期的时间长短都是一样的，因此在确定时钟周期的时间长度时，要考虑指令集中最复杂的指令执行时所需时间。

现在要设计一个单周期 CPU，使它能执行由表 3-3 给出的主机系统指令组成的程序。通过对所有指令的分析，可知其中最复杂的指令是 lw。以 lw $4,80($1)为例，它从存储器中取来数据，放入寄存器堆的寄存器$4 中；存储器的地址由两个数相加得到，其中的一个数是寄存器$1 中的内容，另一个是指令中的立即数 80。它的指令格式及二进制码如下：

lw $4,80($1)　　指令执行的操作是　$4=Memory[$1+80]

6 位	5 位	5 位	16 位

op	rs	rt	immediate
35	1	4	80

100011 00001 00100 0000 0000 0100 0000

执行时需要以下 5 个步骤：

(1) 使用 PC 作为存储器地址，从存储器中把指令取来。

(2) 从寄存器堆中寄存器$1 读出 32 位数据；把立即数 80 符号扩展成 32 位。

(3) 把上述两个数相加。

(4) 使用相加的结果作为地址，从存储器中取来数据。

(5) 把取来的数据写入寄存器$4 中。

假设以上的每个步骤需要 1ns(1ns=10^{-9}s)，则共需 5ns。由此可以确定 CPU 的时钟(clock)频率为：F=1/T=1/(5*10^{-9})=200MHZ。

有些指令，比如 j 指令，比 lw 简单得多，并不需要 5ns。但是在单周期 CPU 中，简单指令也使用一个时钟周期。由于 lw 指令执行的 5 个步骤均在一个时钟周期内完成，并且有两种存储器访问(取指令和取数据)，这里需要有两个存储器模块：指令存储器和数据存储器。地址相加由算术逻辑单元 ALU(arithmetic logic unit)完成。ALU 是 CPU 中负责计算工作的器件，除了能够做加法之外，还要能够完成指令所需的其他操作。

由以上分析可以知道，能够执行 lw 指令的 CPU 需要的器件大致有：

(1) 一个 PC 寄存器，用于保存程序计数器 PC 值。

(2) 一个指令存储器模块，用于存放程序(指令)。

(3) 一个寄存器堆，其中有 32 个寄存器。

(4) 一个 ALU，用于完成指令所需的操作。

(5) 一个数据存储器模块，用于存放数据。

(6) 一个控制部件，用于产生控制信息。

(7) 一个加法器，用于产生下一条顺序指令的地址(PC+4)。把这些器件的输入和输出信号适当地连接(如图 2-3 所示)，一个能够执行 lw 指令的 CPU 的逻辑电路就设计完成了。图中的 ALUOP 是告诉 CPU 执行何种操作的控制信号，执行 lw 指令时，ALU 做加法。WRITEREG 为 1 时，在时钟的上升沿处，把从数据存储器取来的数据

写入寄存器堆。PC+4 写入 PC 也在时钟的上升沿处完成。

寄存器堆有两个读出端口：两个寄存器号分别从 N1 和 N2 送入，相应寄存器的内容分别从 Q1 和 Q2 送出。注意 lw 指令只需读一个寄存器数，源寄存器号是指令中的 rs。另外还有一个写端口：被写入的寄存器由 ND 指定，被写入的数据从 D1 送入。lw 指令中的 rt 是目的寄存器号。

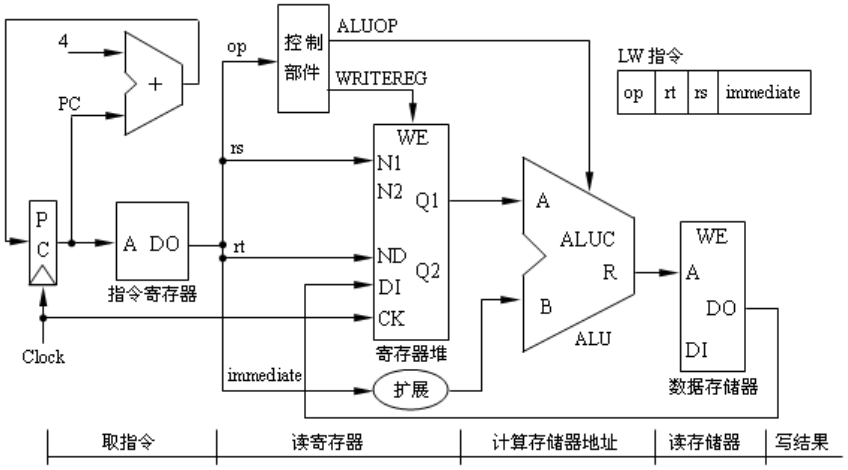


图 2-3　执行 lw 指令的 CPU 的逻辑电路

下面考虑 add 指令。以 add $3,$3,$4 为例。它把寄存器 3 和寄存器 4 的内容相加，结果放回寄存器 3 中。指令格式及二进制码如下：

add $3,$3,$4　　指令执行的操作是　$3=$3+$4

6 位	5 位	5 位	5 位	5 位	6 位
op	rs	rt	rd	sa	func
0	3	4	3	0	32

图 2-4 给出的是 CPU 执行 add 指令时所需的电路。它比 lw 电路要简单，因为 add 指令不访问数据存储器。add 指令的执行次序是：首先取指令，然后读两个源操作数，接下来由 ALU 做加法，最后把相加结果写入寄存器堆。这里有四个地方需要特别注意：第一个是两个寄存器数要同时读出，寄存器号分别是指令中的 rs 和 rt；第二个是寄存器输出端口 2 接 ALU B 输入端，而 lw 是扩展的立即数。再一个是目的寄存器号是 rd，而 lw 指令是 rt；最后一个地方是写入寄存器堆的数据来自于 ALU 的输出，而

lw 来自于存储器的输出。

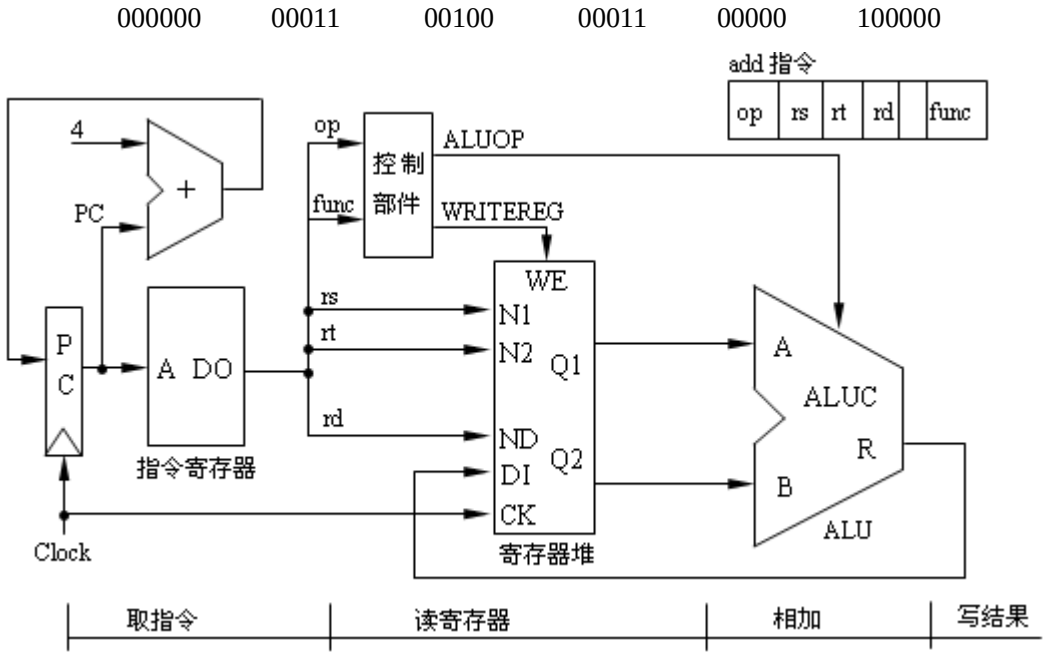


图 2-4　执行 add 指令的 CPU 的逻辑电路

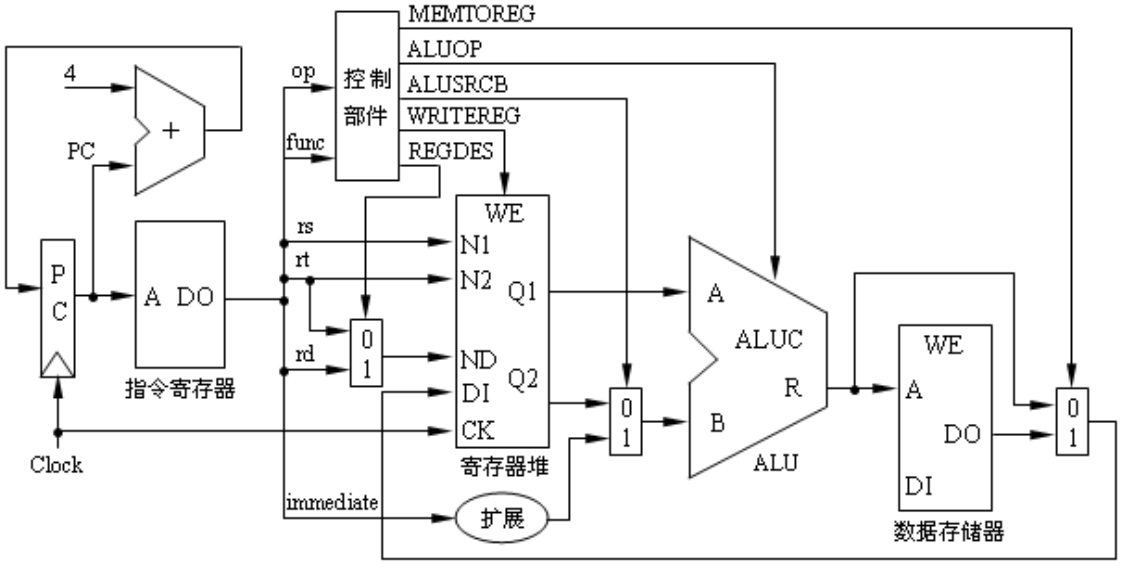


图 2-5　同时执行 lw 和 add 指令的 CPU 的逻辑电路

在这种一个输入端有多个数据来源的情况下，可以使用一个多路选择器(multiplexer)的电路，从多个来源中选择合适的一个。图 2-5 所示的 CPU 逻辑电路能执行 lw 和 add(实际上也能执行其他指令，例如 sub、and、or、addi、andi 和 ori 等，

前提是 ALU 提供了这些运算功能)。图中使用了三个多路选择器。与此相对应，控制部件提供了这些多路选择器的选择信号。这些选择信号的意义如下所述：

(1) ALUSRCB：为 1 时，选择扩展的立即数，为 0 时选择寄存器数据。

(2) MEMTOREG：为 1 时，选择存储器数据，为 0 时选择 ALU 输出的数据。

(3) REGDES：为 1 时，选择 rd，为 0 时选择 rt。

下面以 beq 为例讨论条件转移指令。其指令的格式为：beq rs,trt,label，属于 I 类型的指令。它首先比较寄存器 rs 和 rt 中的内容是否相等。若相等，则转移至 label 处，称其为目标地址。如果 PC 是这条指令的地址，则 label=PC+4(符号扩展)+immediate ×4，其中 immediate 是指令中的 16 位立即数。若不等，则执行 beq 的下面一条指令。比较两个数是否相等可由 ALU 做减法来实现。若相减结果为 0，则表示两个数相等。为此，在设计 ALU 时，可以输出一位 z：等于 1 时表示 ALU 的输出结果为 0。因此，转移的条件是：当前指令为 beq 并且 z=1。写成逻辑表达式为：BRANCH=beq・z。如果把 bne 指令也考虑进去，则：

$$BRANCH=beq \cdot z+bne \cdot \bar{z}$$

转移目标地址的计算由一个专用加法器完成。一个多路选择器选择 PC+4 或目标地址，它的选择信号为 BRANCH。即，BRANCH 为 1 时，选择目标地址，为 0 时选择 PC+4。CPU 的这部分电路如图 2-6 所示。

综合以上的电路，并加上 sw 和 j 指令，可以给出能够执行三种类型指令的单周期 CPU 电路图(如图 2-7 所示)。所有控制信号说明如下：

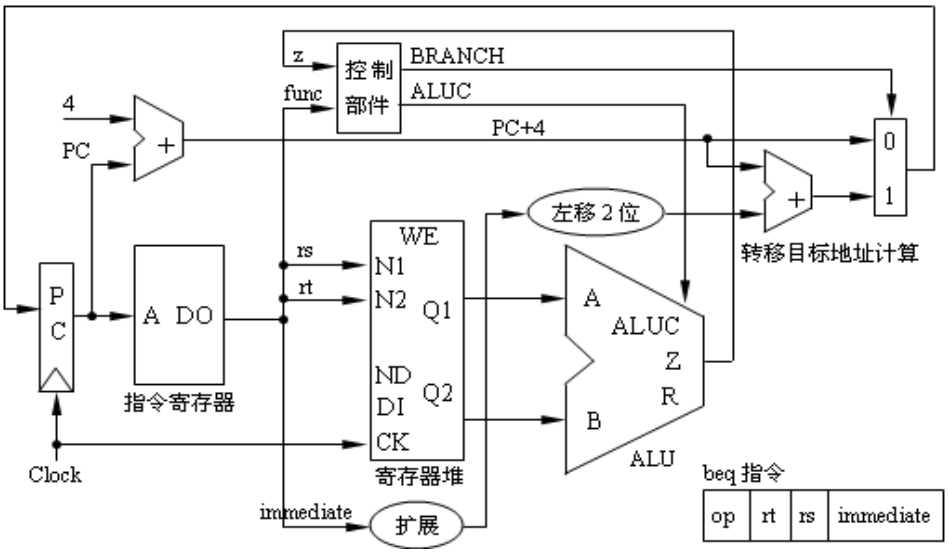


图 2-6　执行 beq 指令的 CPU 的逻辑电路

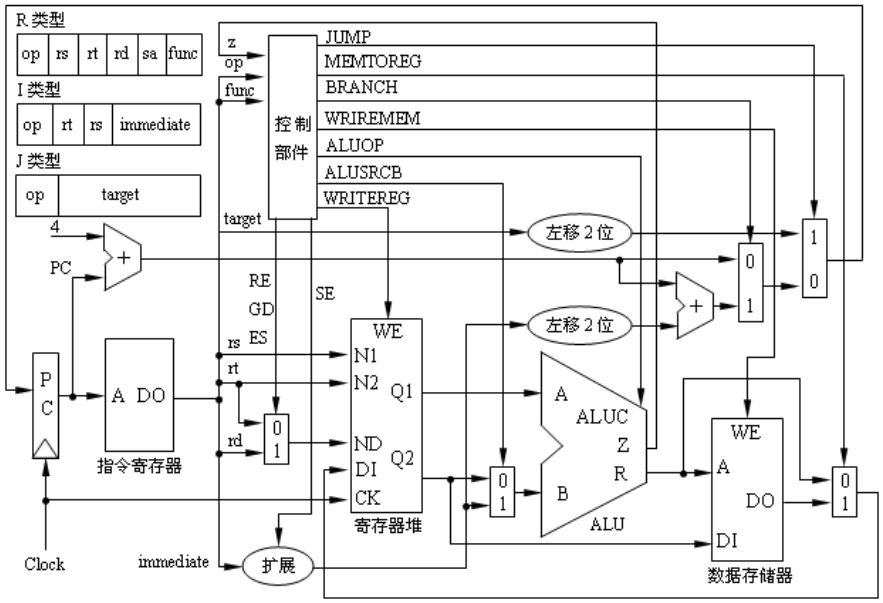


图 2-7　单周期 CPU 的逻辑电路

(1) ALUSRCB：为 1 时，选择扩展的立即数，为 0 时选择寄存器数据。

(2) ALUOP：ALU 操作的控制码。

(3) BRANCH：为 1 时，选择转移目标地址，为 0 时选择 PC+4(图中的 NPC)。

(4) JUMP：为 1 时，选择跳转目标地址，为 0 时选择由 BRANCH 选出的地址。

(5) WRITEMEM：为 1 时写入存储器，存储器地址由 ALU 的输出决定，写入数据为寄存器 rt 的内容。

(6) MEMTOREG：为 1 时，选择存储器数据，为 0 时选择 ALU 输出的数据。

(7) REGDES：为 1 时，选择 rd，为 0 时选择 rt。

(8) WRITEREG：为 1 时写入寄存器堆，目的寄存器号是由 REGDES 选出的 rt 或 rd，写入数据是由 MEMTOREG 选出的存储器数据或 ALU 的输出结果。

(9) SE：符号扩展。为 1 时，符号扩展，为 0 时，0 扩展。

从上面的介绍可以看出，设计 CPU 时一个重要的工作是理顺数据在 CPU 各个部分之间的数据通路，例如，寄存器堆和 ALU 之间的数据通路，寄存器堆和存储器之间的数据通道，ALU 和存储器之间的数据通道等。下面再根据不同类型的指令的处理过程回顾一下单周期 CPU 的设计。

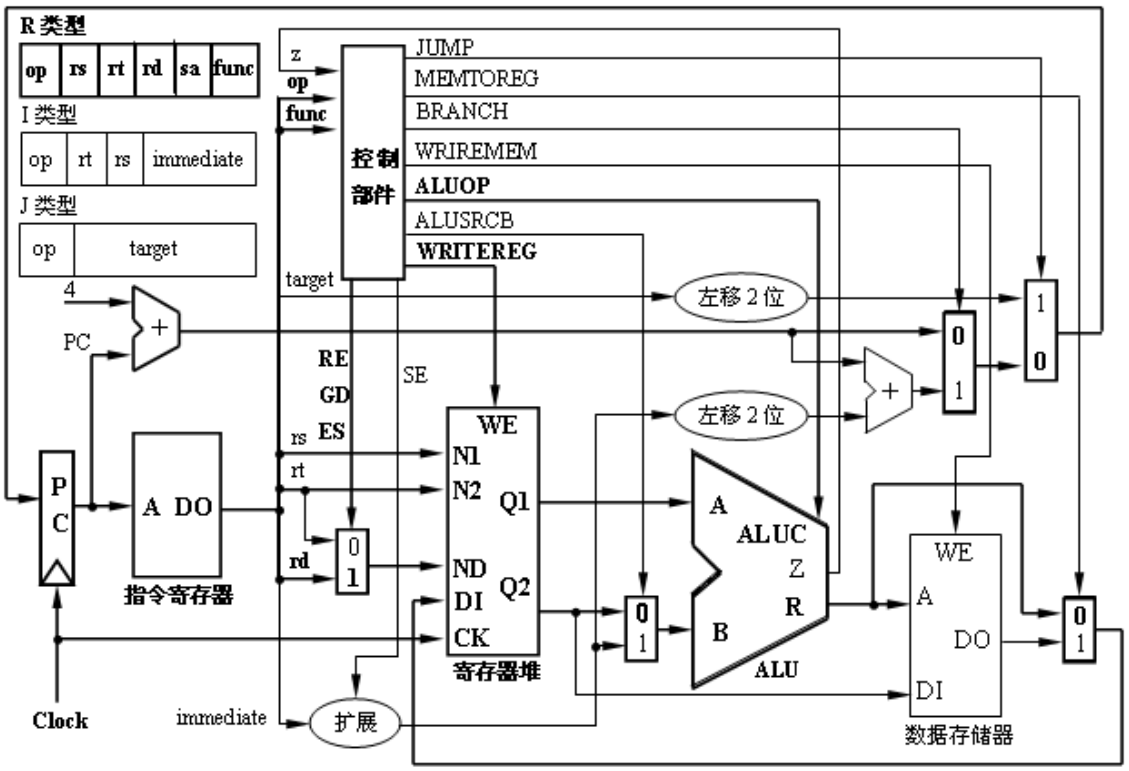


图 2-8　R 类型指令处理过程图

2.3.2 R 类型指令

R 类型指令使用寄存器 rs 和寄存器 rt 作为源数据寄存器，使用寄存器 rd 作为目的寄存器。该类型指令使用 ALU 对两个源数据进行某种运算之后，把 ALU 生成的结果写入到目的寄存器中。所以，在 R 类型指令处理过程中，存在如下几条数据通路，图 2-8 显示了这些数据通道的建立(图中粗线表示信号的传递，以下相同)：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 寄存器堆中的寄存器 rt 到 ALU 的输入数据 B；

(3) ALU 的计算结果 R 到寄存器堆中的寄存器 rd。

2.3.3 I 类型指令

I 类型指令存在着多种情况。包括运算指令、Load 指令、Store 指令、分支指令等。下面逐一介绍。

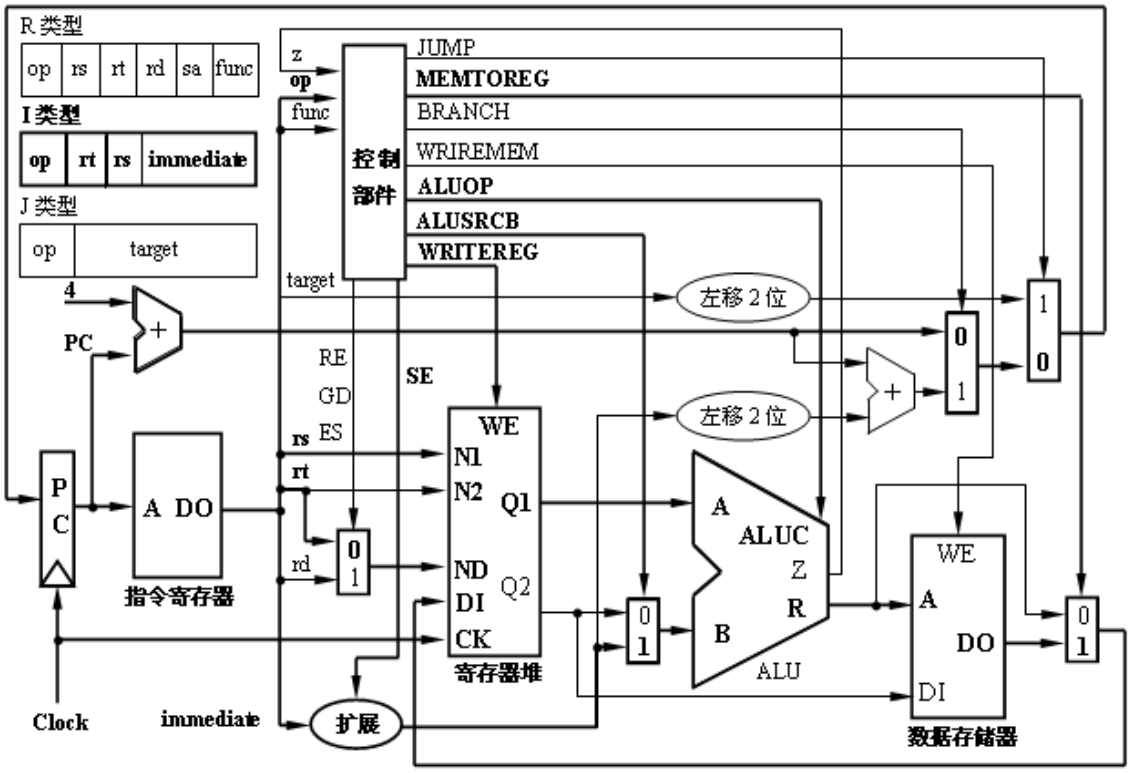


图 2-9　I 类型指令 lw 的处理过程图

1．运算指令

在 I 类型的运算指令中，使用寄存器 rs 和指令字中的立即数为源数据，使用寄存器 rt 作为目的寄存器。该类型指令使用 ALU 对两个源数据进行某种操作之后，把 ALU 产生的结果保存到目的寄存器中。所以，在 I 类型的运算指令中，需要建立如下几条数据通道：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 指令字中的立即数到 ALU 的输入数据 B；

(3) ALU 的运算结果 R 到寄存器堆中的寄存器 rt。

2．Load 指令

在 Load 指令中，使用寄存器 base(即寄存器 rs)和指令字中的立即数为源数据。使用 ALU 把这两个源数据相加，得到的结果作为输出给存储器的数据地址。然后，读入存储器按照数据地址给出的数据，把这个数据输入到目的寄存器 rt 中。所以，在 Load 指令中，需要建立如下几条数据通道，图 2-9 显示了这些数据通道的建立：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 指令字中的立即数到 ALU 的输入数据 B；

(3) ALU 的计算结果 R 到通往存储器的地址总线；

(4) 来自存储器的数据总线到寄存器堆的寄存器 rt。

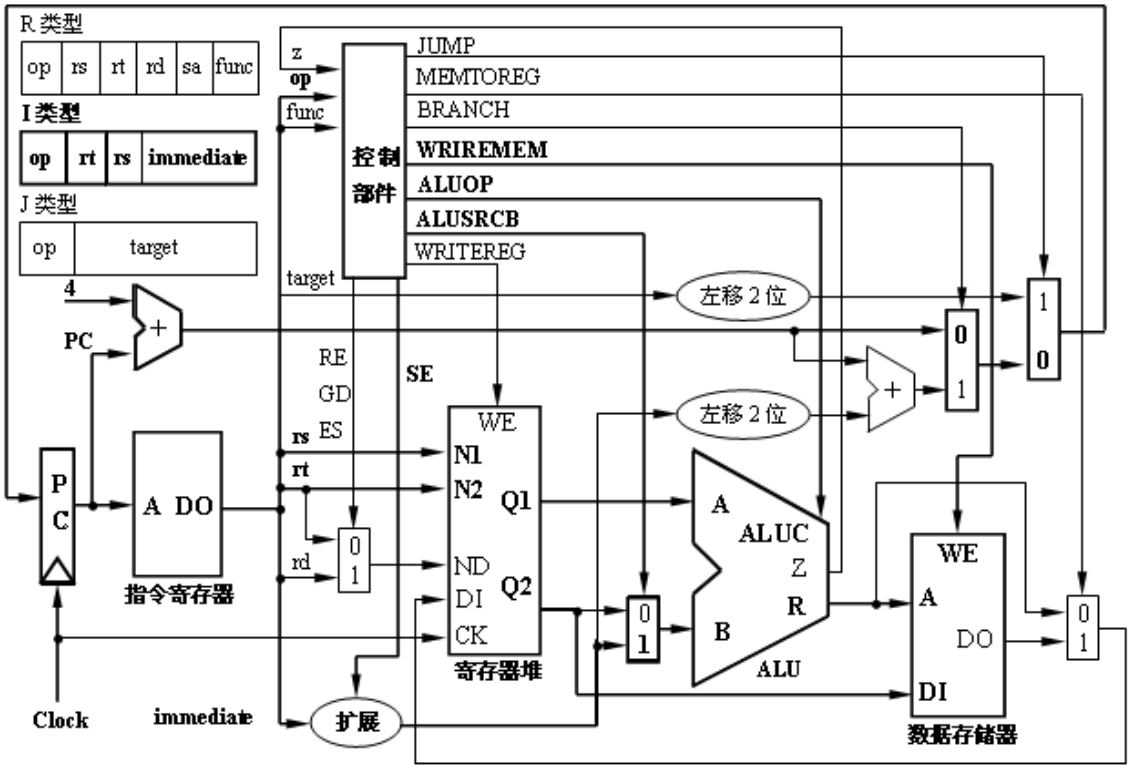


图 2-10　I 类型指令 sw 的处理过程图

3．Store 指令

在 Store 指令中，使用寄存器 base(即寄存器 rs)和指令字中的立即数为源数据。使用 ALU 把这两个源数据相加，得到的结果作为输出给存储器的数据地址。把寄存器 rt 中的数据作为输出给存储器的写入数据。在给存储器输出写入信号之后，使存储器按照数据地址把寄存器 rt 中的数据写入。所以，在 Store 指令中，需要建立如下几条数据通道，图 2-10 显示了这些数据通道的建立：

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 指令字中的立即数到 ALU 的输入数据 B；

(3) ALU 的计算结果 R 到通往存储器的地址总线；

(4) 寄存器堆的寄存器 rt 到通往存储器的数据总线。

4．分支指令

分支指令 beg 和 bne 使用寄存器 rs 和寄存器 rt 作为源数据寄存器，使用 ALU 对这两个源数据进行减法操作。然后，判断结果是否为 0，把判断的结果传送给控制部件 CU。CU 通过指令类别和是否为 0 的判断结果来确定是否需要进行指令跳转，给出跳转标志。同时，分支指令需要把程序计数器 PC 和 4 相加，得到一个新的 PC 值 A，作为下一个 PC 值的候选值。然后，再把 PC 值 A 和指令字中的立即数相加，得到一个新的 PC 值 B，作为另一个下一个 PC 值的候选值。最后，根据 CU 给出的跳转标志来确定是采用 PC 值 A 还是 PC 值 B 作为下一个 PC 值。这样，在分支指令的处理过程中，需要建立下面几条数据通道，图 2-11 显示了这些数据通道的建立：

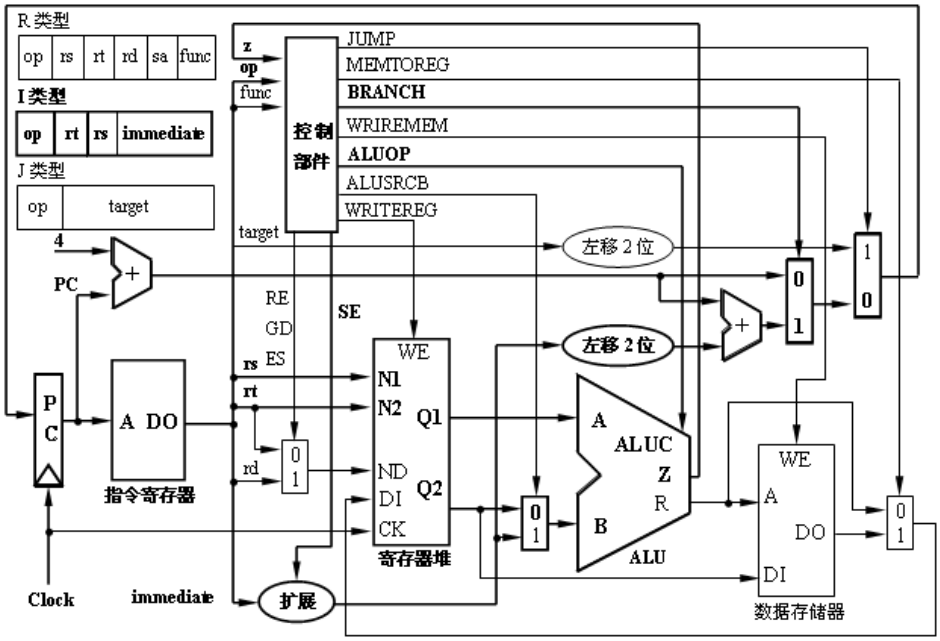


图 2-11　I 类型指令 beq 的处理过程图

(1) 寄存器堆中的寄存器 rs 到 ALU 的输入数据 A；

(2) 寄存器堆中的寄存器 rt 到 ALU 的输入数据 B；

(3) ALU 的输出结果 R 经判断是否为 0 后输出给控制部件 CU；

(4) 程序计数器 PC 加 4 后到数据选择器的一个输入；

(5) 程序计数器 PC 加 4 后再与指令字中的立即数相加，输出到数据选择器的另一个输入；

(6) 控制部件 CU 给出跳转标志到数据选择器的选择端；

(7) 数据选择器把选择结果输出给程序计数器 PC。

2.3.4 J 类型指令

跳转指令会引起程序控制流的无条件跳转。在跳转指令的处理过程中，需要把程序计数器 PC 加 4 后的高 4 位和指令字中的跳转地址合并。然后，把合并的结果作为下一个 PC 值，输出到程序计数器 PC 中。所以 jump 指令需要建立如下的数据通道，图 2-12 显示了这些数据通道的建立：

(1) 程序计数器 PC 加 4 后的高 4 位和指令字中的跳转地址合并；

(2) 合并结果输出到程序计数器 PC。

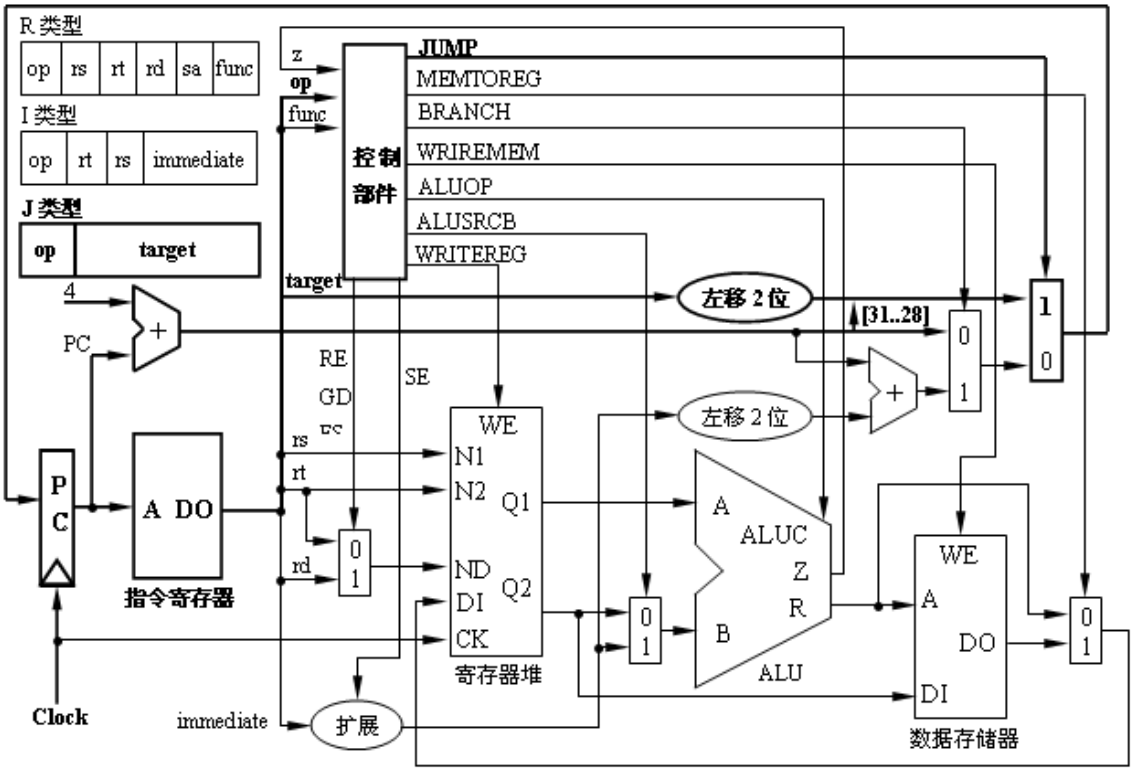


图 2-12　J 类型指令处理过程图

为了能够更好地进行计算机主机系统设计，并且以上 MIPS 指令能在设计出的机器上编程并运行，以下各节分别给出单周期 CPU 逻辑电路中的各个基本部件，使用原理图方式来实现。

3 多路选择器

多路选择器(Multiplexer)又称为数据选择器或多路开关，是一种多个输入一个输出的中规模器件，其功能是在选择控制码（或叫地址）电位的控制下，从几路数据输入中选择一路并将其送到一个公共输出端。也就是经过多路选择，把多个通道的数据传送到唯一的公共数据通道上去，因此实现数据选择功能的逻辑电路称为多路选择器。

多路选择器的功能类似一个多掷开关，如果它有两路数据 A0 和 A1，则通过选择控制信号 S，从两路数据中选中某一路数据送至输出端 Y。即指经过选择，把多个通道的数据传到唯一的公共数据通道上。所以说多路选择器相当于多个输入的单刀多掷开关。

3.1 1 位 2 选 1 多路选择器

1 位 2 选 1 多路选择器的电路符号如图 4-1 所示。输入信号：两个 1 位数据源 A0 和 A1；选择信号：1 位 S；输出信号：1 位 Y。其真值表如表 3-1 所示。

设计分析，根据 2 选 1 多路选择器描述和真值表，可得输入和输出逻辑关系：$Y=\overline{S}\cdot A0+S\cdot A1$。

表 3-1 1 位 2 选 1 多路选择器的真值表

输入(X 代表 0 或 1 任意数)			输出
S	A1	A0	Y
0	X	X	A0
1	X	X	A1

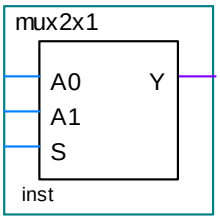


图 3-1 1 位 2 选 1 多路选择器电路

3.1.1 原理图设计 1 位 2 选 1 多路选择器

由逻辑关系式可知，当 S 为 0 时，Y 的值由 A0 确定；当 S 为 1 时，Y 的值由 A1 确定。因此可创建 1 位 2 选 1 多路选择器的原理图，如图 3-2 所示。

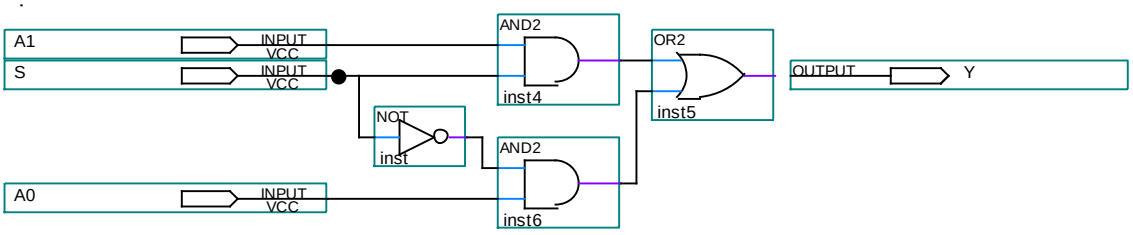


图 3-2　1 位 2 选 1 多路选择器的原理图

3.1.2 1 位 2 选 1 多路选择器的仿真验证

(1)新建项目 运行 Quartus Ⅱ软件，执行菜单“File”→“New Project Wizard”，在弹出的“New Project Wizard”对话框中选择项目和文件的保存路径→输入项目名称 cpu 及文件名称 cpu。

(2)创建原理图文件 执行菜单命令“File”→“New”，在“New”对话框的“Design Files” →“Block Diagram/Schematic File”，单击“OK”按钮。在原理图编辑窗口中按照如图 4-2 所示创建 1 位 2 选 1 多路选择器的原理图，保存为“mux2x1.bdf”。

(3)创建 VHDL 文件 在“New”对话框的“Design Files”→“VHDL File”，单击“OK”按钮。在 VHDL 程序编辑窗口中输入 1 位 2 选 1 多路选择器的 VHDL 程序，保存为“V_mux2x1.vhd”。

(4)编译原理图或 VHDL 文件 在项目导航窗口“Project Navigator”窗口中右键单击 mux2x1.bdf 文件或 V_mux2x1.vhd 文件，选择“Set as Top-Level Entity”单击，将文件设置为顶层文件。然后，在工具栏中单击▶图标进行编译。

(5)创建波形文件 编译成功后，执行菜单命令“File”→“New”，在“New”对话框的“Verification/Debugging Files”→“Vector Waveform File”，单击“OK”按钮，弹出波形图编辑窗口，在波形编辑窗口右边“Name”下的空白处单击鼠标右键，选择“Insert Node or Bus”命令，单击“Node Finder…”按钮。在弹出的窗口“Filter”栏中选择“Pins:all”，单击“List”按钮，单击窗口中间的 >> 按钮，单击“OK”按钮。这样将所有的端口引脚导入波形图编辑窗口中。

在波形编辑窗口中，对所有信号节点前面有 图标，通过下面的步骤一个一个依据如图 3-3 所示设置。右键要设置的节点信号，在单击“Value”→“Conut Value”弹出相应对话框，然后，进行值设置。最后保存该波形文件“mux2x1.vwf”和“V_mux2x1.vwf”。

(6)功能仿真 执行菜单命令“Processing”→“Start Simulation”，选择“Functional”功能仿真，单击对话框中的 ... 图标按钮，弹出“打开”对话框，找到保存的 mux2x1.vwf 或 V_mux2x1.vwf 波形文件打开。然后单击“Generate Functional Simulation Netlist”按钮，生成功能仿真网络表。最后，单击 Start 按钮，进行功能仿真。单击 Report 查看生成的功能仿真波形图，如图 3-3 所示。原理图和 VHDL 最后生成的功能仿真波形图应一样。

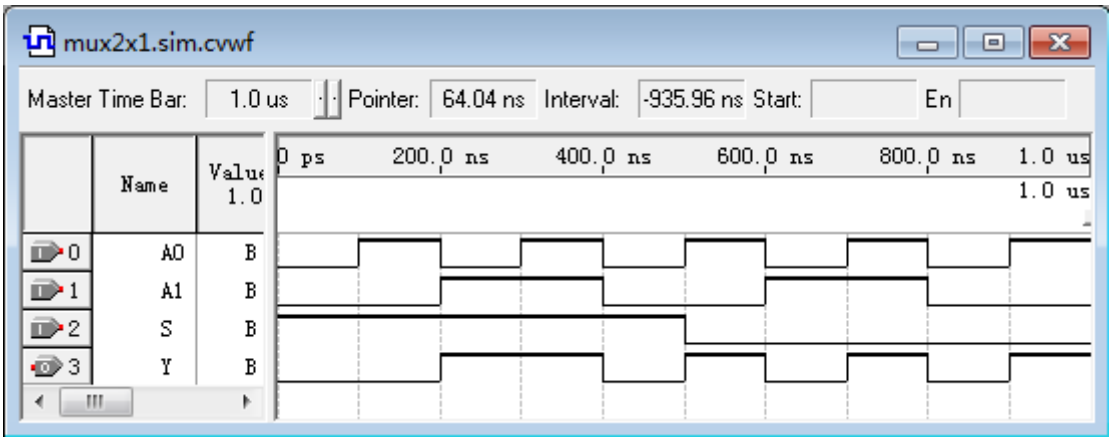


图 3-3 1 位 2 选 1 多路选择器的仿真波形图

(7)波形分析 由如图 3-3 所示 500.0ns 到 600.0ns 这一时段，A0 为 1，A1 为 0，S 为 0，故 Y 的值应该由 A0 来确定，为 1，图中也正好此时段 Y 的值为 0，因此得证。

(8)生成原理图元器件 执行菜单命令“File”→“Create/Update”→“Create Symbol Design Fors Current File”，保存文件，就生成了一个如图 3-1 所示的 1 位 2 选 1 多路选择器元器件 mux2x1.bsf（注意：当前文件一定要是 mux2x1.bdf）。

(9)生成对应的 VHDL 定义元件语句 执行菜单命令“File”→“Create/Update”→“Create VHDL Component Declaration Files For Current File”，保存文件，便于以后其他 VHDL 程序调用、例化（注意：当前文件一定要是 V_mux2x1.vhd）。

表 3-2 5 位 2 选 1 多路选择器的真值表

输入(X 代表 0 或 1 组成的任意数)			输出
S	A1	A0	Y
0	X	X	A0
1	X	X	A1

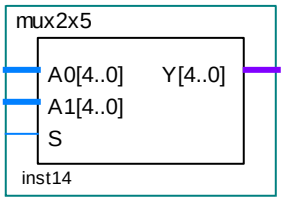


图 3-4 5 位 2 选 1 多路选择器电路符号

3.2 5 位 2 选 1 多路选择器

5 位 2 选 1 多路选择器的电路符号如图 3-4 所示。输入信号：两个 5 位数据源 A0 和 A1；选择信号：1 位 S；输出信号：5 位 Y。它可以由 5 个 1 位 2 选 1 多路选择器经过级联后得到。这 5 个 1 位 2 选 1 多路选择器的每个 A0 分别为 A0[0]到 A0[4]，A1 分别为 A1[0]到 A1[4]，Y 分别为 Y[0]到 Y[4]，选择信号都为 S。（注意：要相对应，如：第一个 1 位 2 选 1 多路选择器的 A0 为 A0[0]，则 A1 应为 A1[0]，Y 应为 Y[0]）。

真值表如表 3-2 所示。

3.2.1 原理图设计 5 位 2 选 1 多路选择器

由描述级联和真值表创建 5 位 2 选 1 多路选择器的原理图，如图 3-5 所示：

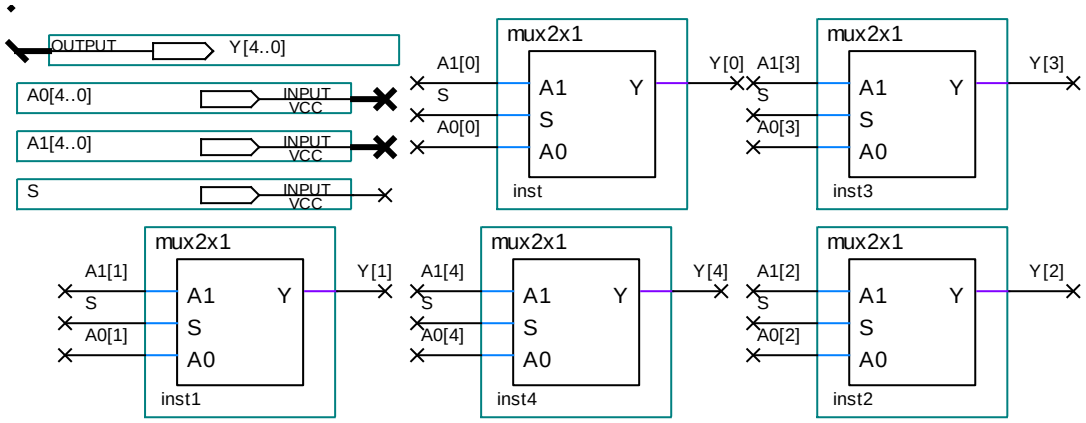


图 3-5 5 位 2 选 1 多路选择器的原理图

注意：由于 A0[4..0]是一个由 A0[4]、A0[3]、A0[2]、A0[1]、A0[0]按顺序组成，故输入信号 A0[4..0]拖出的线是加粗了的线，表示是总线。其它的也同理。

3.2.2 5 位 2 选 1 多路选择器的仿真验证

(1)打开 CPU 项目 执行菜单“File”→“Open Project…”，打开工程项目对话框，找到存放 CPU 项目的路径，双击 cpu.qpf 文件打开 CPU 项目。

(2)创建一个原理图文件 新建一个原理图文件，在原理图编辑窗口中按照图 3-5 创建 5 位 2 选 1 多路选择器的原理图，通过调用 mux2x1 元器件，保存为“mux2x5.bdf”。

(3)创建一个 VHDL 文件 新建一个 VHDL File 文件，在 VHDL 程序编辑窗口中输入 5 位 2 选 1 多路选择器的 VHDL 程序，保存为“V_mux2x5.vhd”。

(4)编译 将要编译的文件设置为顶层文件，编译。

(5)创建波形文件 通过编译后，新建 Vector Waveform File 文件，将所有的信号节点导入波形图编辑窗口中，在这里导入信号节点 A0[4..0]、A1[4..0]和 Y[4..0]时，只要导入 A0、A1、Y 即可，并且要将这三个信号节点的属性设置为 16 进制显示。对所有信号节点前面有 或 图标的，按如图 3-6 所示进行值设置。然后，保存该波形文件为“mux2x5.vwf”和“V_mux2x5.vwf”。

(6)功能仿真 原理图和 VHDL 最后生成的功能仿真波形图应一样，如图 3-6 所示。

(7)波形分析 由如图 4-6 所示 500.0ns 到 600.0ns 这一时段，A0 为 10，A1 为 06，

S 为 0，故 Y 的值应该由 A0 来确定，为 10，图中也正好此时段 Y 的值为 10，因此得证。

最后可以根据原理图，按照第 3 章 3.1.2 节方法生成 5 位 2 选 1 多路选择器元器件 mux2x5.bsf。

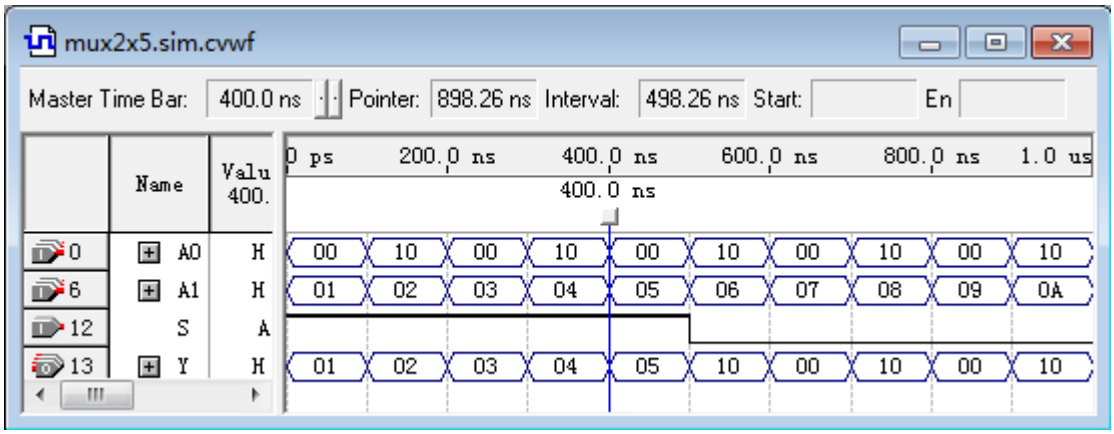


图 3-6 5 位 2 选 1 多路选择器的仿真波形图

3.3 32 位多路选择器

表 3-4 32 位 2 选 1 多路选择器的真值表

输入(X 代表 0 或 1 组成的任意数)			输出
S	A1	A0	Y
0	X	X	A0
1	X	X	A1

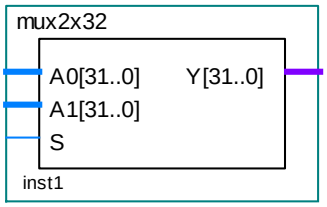


图 3-7　32 位 2 选 1 多路选择器电

3.3.1 32 位 2 选 1 多路选择器

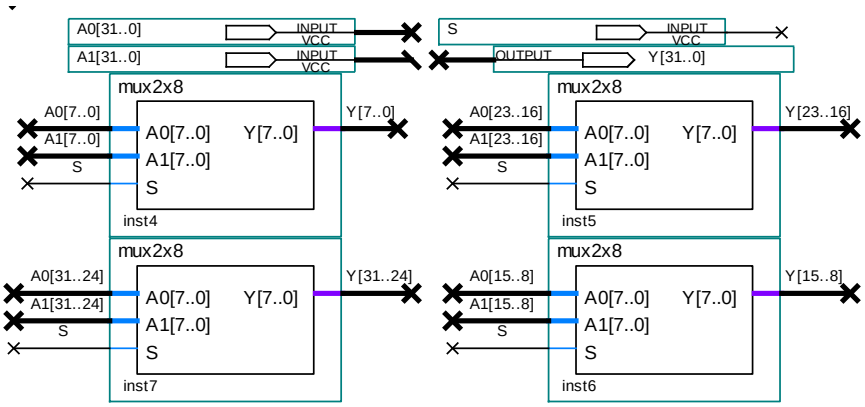


图 3-8 32 位 2 选 1 多路选择器的原理图

注意：A0、A1、Y 在不同的 mux2x8 中的下标。自己在命名输入和输出时，下标建议用逆序的，这是因为所用的软件中的自带的元件的下标就是逆序的。同时，保持一致也便于使用，在调用自带的元件和自己封装的元件时。

同样方法可以设计 8 位 2 选 1 多路选择器，由 8 位 2 选 1 多路选择器扩展为 32 位 2 选 1 多路选择器，32 位 2 选 1 多路选择器的电路符号如图 3-7 所示。输入信号：两个 32 位数据源 A0 和 A1；选择信号：1 位 S；输出信号：32 位 Y。它可以由 4 个 8 位 2 选 1 多路选择器经过级联后得到。这 4 个 8 位 2 选 1 多路选择器的每个 A0 分别为 A0[0]到 A0[31]，A1 分别为 A1[0]到 A1[31]，Y 分别为 Y[0]到 Y[31]，选择信号都为 S。真值表如表 3-4 所示。

由描述级联和真值表创建 32 位 2 选 1 多路选择器的原理图，如图 3-8 所示。

3.3.2 32 位 4 选 1 多路选择器

表 3-5 32 位 4 选 1 多路选择器的真值表

输入(X 代表 0 或 1 组成的任意数)						输出
S[1..0]		A[3..0]				Y
S1	S0	A3	A2	A1	A0	
0	0	X	X	X	X	A0
0	1	X	X	X	X	A1
1	0	X	X	X	X	A2
1	1	X	X	X	X	A3

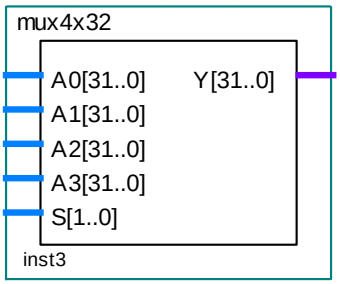


图 3-9 32 位 4 选 1 多路选择器电路符号

由描述级联和真值表创建 32 位 4 选 1 多路选择器的原理图，如图 3-10 所示。

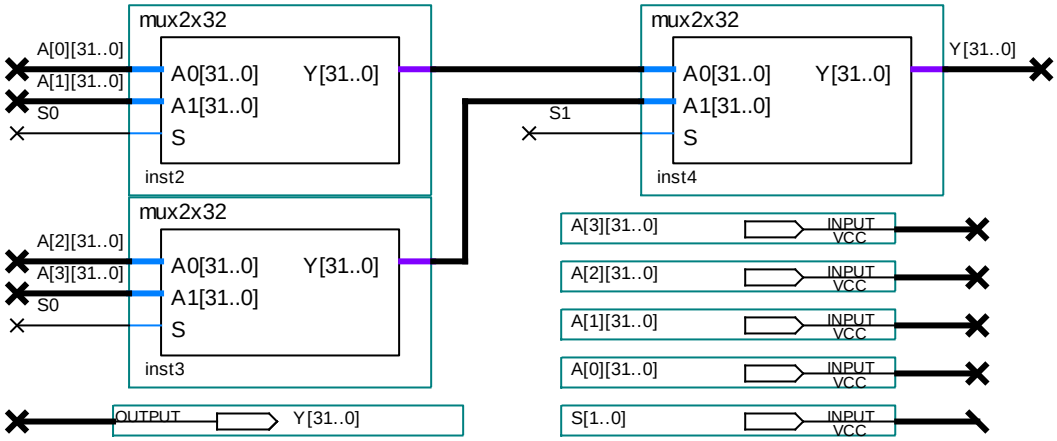


图 3-10 32 位 4 选 1 多路选择器的原理图

32 位 4 选 1 多路选择器的电路符号如图 3-9 所示。输入信号：四个 32 位数据源 A[0]、A[1]、A[2]、A[3]；选择信号：2 位 S[1..0]；输出信号：32 位 Y。它可以由 3 个 32 位 2 选 1 多路选择器经过级联后得到。首先，根据 S0 信号从 4 个中选出 2 个为 A[0][31..0]、A[2][31..0]或 A[1][31..0]、A[3][31..0]。然后，从选出的 2 个中再根据 S1 信号选出结果。这 3 个 32 位 2 选 1 多路选择器中有 2 个的 A0 分别为 A[0][31..0]和 A[2][31..0]，A1 分别为 A [1][31..0]和 A [3][31..0]，选择信号都为 S0；另外一个多路选择器的 A0 和 A1 分别为前两个多路选择器的结果，且前两个多路选择器中 A0 为 A[0][31..0]的选择器的结果为当前选择器的 A[0]，另一个为 A[1]，选择信号为 S1。Y 为第三个选择器的结果。真值表如表 3-5 所示。

3.3.3 32 位 32 选 1 多路选择器

32 位 32 选 1 多路选择器的电路符号如图 3-11 所示。输入信号：32 个 32 位数据源 A[31..0][31..0]。选择信号：5 位 S[4..0]；输出信号：32 位 Y。它可以由 10 个 32 位 4 选 1 多路选择器和 1 个 32 位 2 选 1 多路选择器经过级联后得到。首先，根据 S[1..0]信号用 8 个 32 位 4 选 1 多路选择器从 32 个输入信号中选出 8 个信号为 A[0][31..0]、A[4][31..0]、A[8][31..0]、A[12][31..0]、A[16][31..0]、A[20][31..0]、A[24][31..0]、A[28][31..0]或者其他几组信号也行。然后，用 2 个 32 位 4 选 1 多路选择器从选出的 8 个中再根据 S[3..2]信号选出 2 个信号，最后，用 1 个 32 位 2 选 1 多路选择器从选出的 2 个中再根据 S[4]选出最后结果。Y 为最后选出的结果。真值表如表 3-6 所示。

表 3-6 32 位 4 选 1 多路选择器的真值表

输入	S	S4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		S3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
		S2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1
		S1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
		S0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	A[31..0]		X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
输出	Y		A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19	A20	A21	A22	A23	A24	A25	A26	A27	A28	A29	A30	A31					

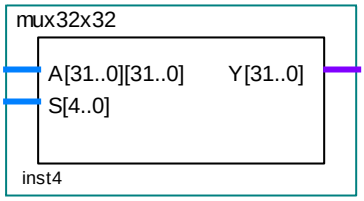


图 3-11 32 位 32 选 1 多路选择器电路符号

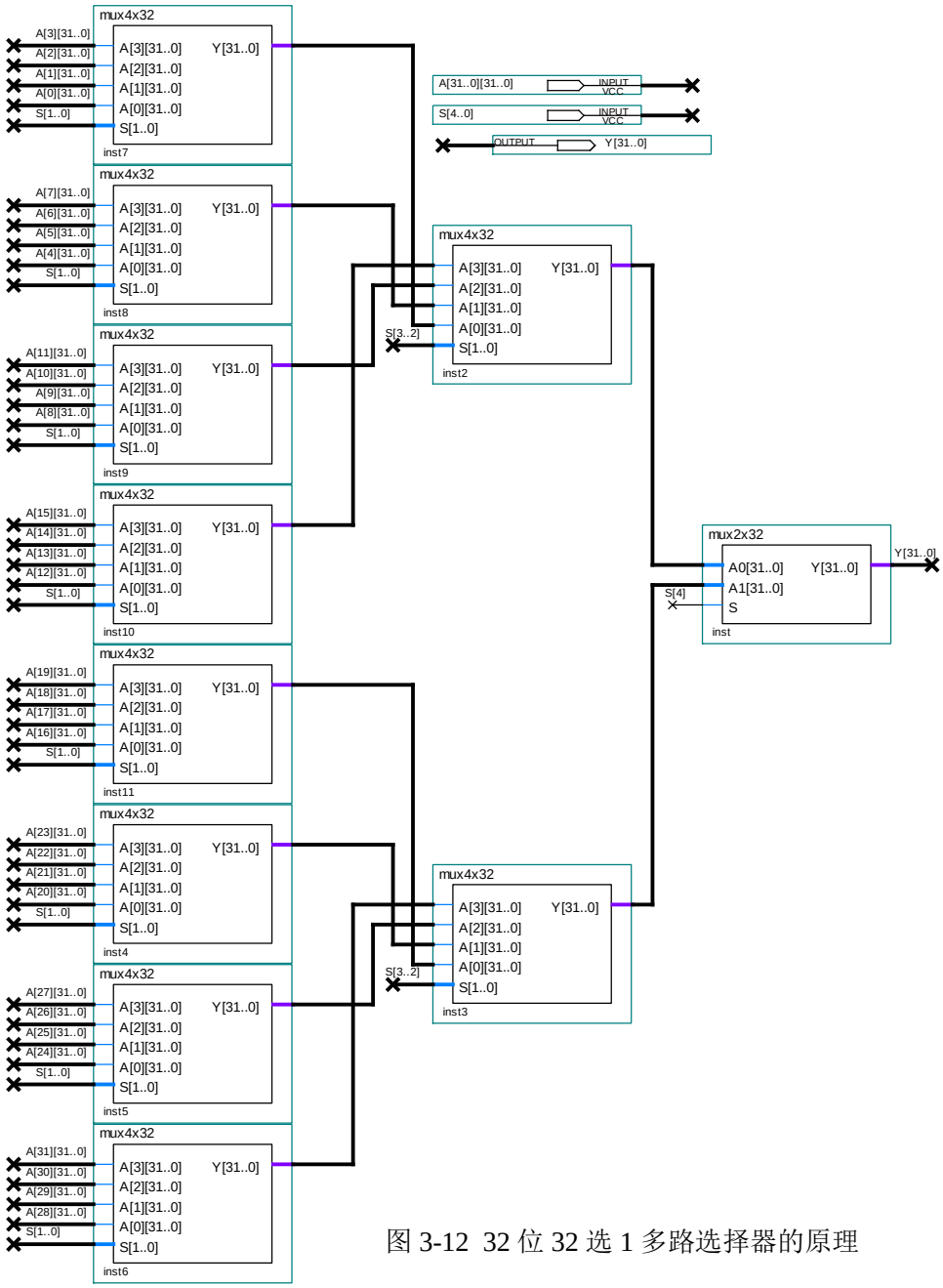


图 3-12 32 位 32 选 1 多路选择器的原理

由描述级联和真值表创建 32 位 32 选 1 多路选择器的原理图，如图 3-12 所示。

利用原理图和 VHDL 程序设计的 32 位 32 选 1 多路选择器，用到的设备（芯片）所需的引脚非常多，在现有的 FPGA 中找不到满足这么多引脚的设备可用，编译时在匹配引脚时会由于引脚不够而产生编译错误，故该原理图和 VHDL 程序不能通过编译，但可以把它封装成元件 mux4x32.bsf 和对应的 VHDL 定义元件语句，供其它原理图和 VHDL 程序调用。

4　加减器

加减器是以二进制方式进行数字的加法或减法运算，它能进行加法与减法运算，做减法运算时，是将减法转化为加法，然后做加法运算。它可以用全加器做成。

4.1 1 位加法器

在两个一位二进制数 A 和 B 相加时，如果考虑相邻低位的进位 C0，即将两个一位二进制数相加的同时，再加上来自低位的进位信号，这种运算电路称为全加器（1 位加法器）。1 位加法器有两个输出 S 和 C1，其中 S 为加法器的和，C1 为进位输出。1 位加法器的符号如图 4-1 所示，真值表如表 4-1 所示。

表 4-1 1 位加法器的真值表

输入	A	0	0	0	0	1	1	1	1
	B	0	0	1	1	0	0	1	1
	C0	0	1	0	1	0	1	0	1
输出	S	0	1	1	0	1	0	0	1
	C1	0	0	0	1	0	1	1	1

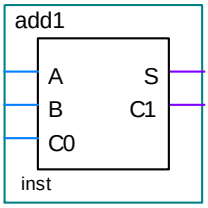


图 4-1 1 位加法器电路符号

4.1.1 原理图设计 1 位加法器

根据真值表，可得 1 位加法器的输入与输出逻辑关系：

S=（A⊕B）⊕C0

C1=（A▪B）+（（A⊕B）▪C0）=（A▪B）+（B▪C）+（A▪C）

由逻辑关系式，创建 1 位加法器的原理图，如图 4-2 所示。

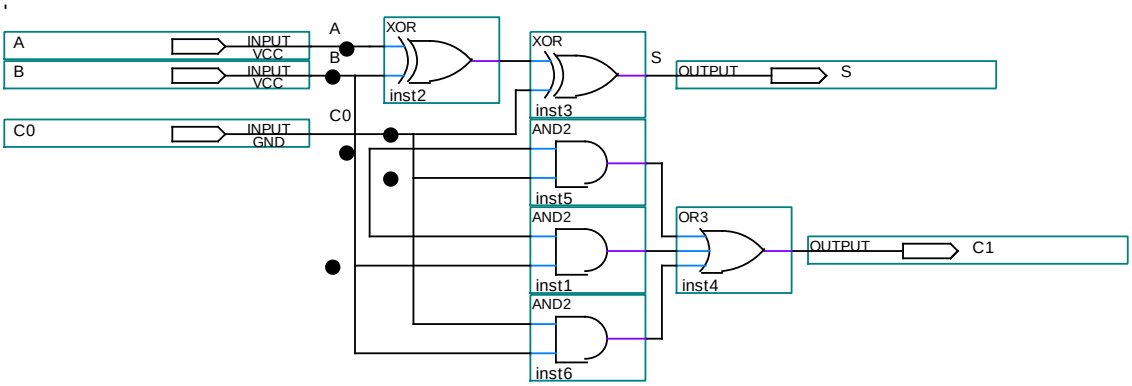


图 4-2 1 位加法器的原理图

4.1.2 1 位加法器的仿真验证

(1)打开 cpu 项目。

(2)创建一个原理图文件 新建一个原理图文件，在原理图编辑窗口中按照图 5-2 创建 1 位加法器的原理图，保存为“add1.bdf”。

(3)创建一个 VHDL 文件 新建一个 VHDL File 文件，在 VHDL 程序编辑窗口中输入 1 位加法器的 VHDL 程序，保存为“V_add1.vhd”。

(4)编译 将要编译的文件设置为顶层文件，编译。

(5)创建波形文件 通过编译后，新建 Vector Waveform File 文件，将所有的信号节点导入波形图编辑窗口中。对所有信号节点前面有 或 图标的，按如图 4-2 所示进行值设置。然后，保存该波形文件为“add1.vwf”和“V_add1.vwf”。

(6)功能仿真 原理图和 VHDL 最后生成的功能仿真波形图应一样，如图 4-2 所示。

(7)波形分析 由如图 4-2 所示 400.0ns 到 500.0ns 这一时段，A 为 0，B 为 0，C0 为 1，故 C1 和 S 的值应该分别为 0 和 1，图中正好此时段 C1 和 S 的值也分别为 0 和 1，因此得证。

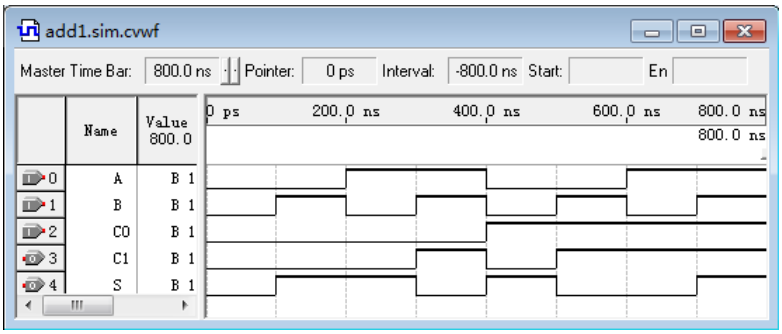


图 4-2　1 位加法器的仿真波形图

最后可以根据原理图，按照第 3 章 3.1.2 节方法生成 1 位加法器元器件 add1.bsf。

4.2　1 位加减器

1 位加减器是能进行 1 位的加法和减法运算。由于 A-B 可等价于 A+(-B)，所以可通过用补码运算，就可得到 A-B 的结果。1 位加减器有两个输出 S 和 C，其中 S 为加法器的和，C 用于判断做减法运算的结果是否为负数。有三个输入 A、B 和 SUB，其中 SUB 用于标记是否做减法运算。1 位加减器的电路符号如图 4-3 所示，真值表如表 4-2 所示。

表 4-2 1 位加减器的真值表

输入	A	0	0	1	1	0	0	1	1
	B	0	1	0	1	0	1	0	1
	SUB	0	0	0	0	1	1	1	1
输出	S	0	1	1	0	1	0	0	1
	C	0	0	0	1	1	0	1	1

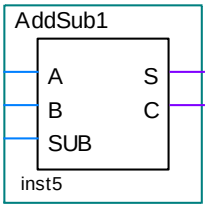


图 4-3 1 位加减器电路符号

由于是将减法运算转化为加法运算，则当 SUB=0 时，S=A+B；当 SUB=1 时，S=A-B=A+(-B)=A+$\overline{B}$+1，而 B XOR 0=B，B XOR 1=$\overline{B}$，故有 S=A+(B XOR SUB)+SUB。

由逻辑关系式，创建 1 位加减器的原理图，如图 4-4 所示。

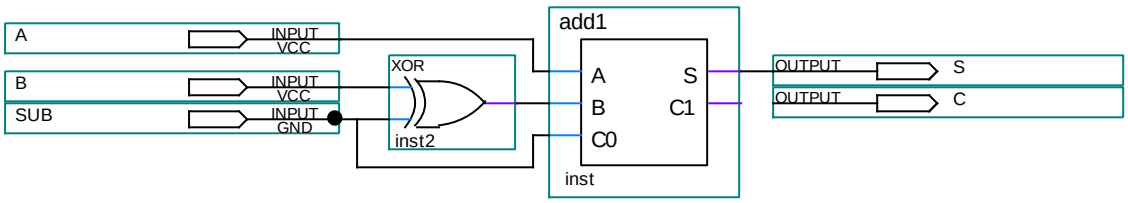


图 4-4 1 位加减器的原理图

4.3 8 位加法器

8 位加法器用于对两个 8 位二进制数进行加法运算，并产生进位。8 位加法器的电路符号如图 4-5 所示，真值表如表 4-3 所示。表中 A[7..0]表示 A 有 8 位输入端 A7、A6、A5、A4、A3、A2、A1、A0；B[7..0]表示 B 有 8 位输入端 B7、B6、B5、B4、B3、B2、B1、B0；S[7..0]表示 S 有 8 位输入端 S7、S6、S5、S4、S3、S2、S1、S0。8 位加法器的 A、B 都有 8 个输入端，加上进位 CarryIn，共有 17 个输入端。它有 9 个输出端，即 S7、S6、S5、S4、S3、S2、S1、S0 和 CarryOut，因此 8 位加法器可由 8 个 1 为加法器构成。

表 4-3 8 位加法器的真值表

输入			输出	
A[7..0]	B[7..0]	CarryIn	S[7..0]	CarryOut
A	B	进位输入	A+B+CarryIn	进位输出

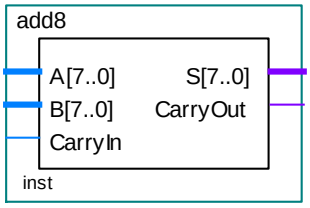


图 4-5 8 位加法器电路符号

由描述级联和真值表创建 8 位加法器的原理图，如图 4-6 所示：

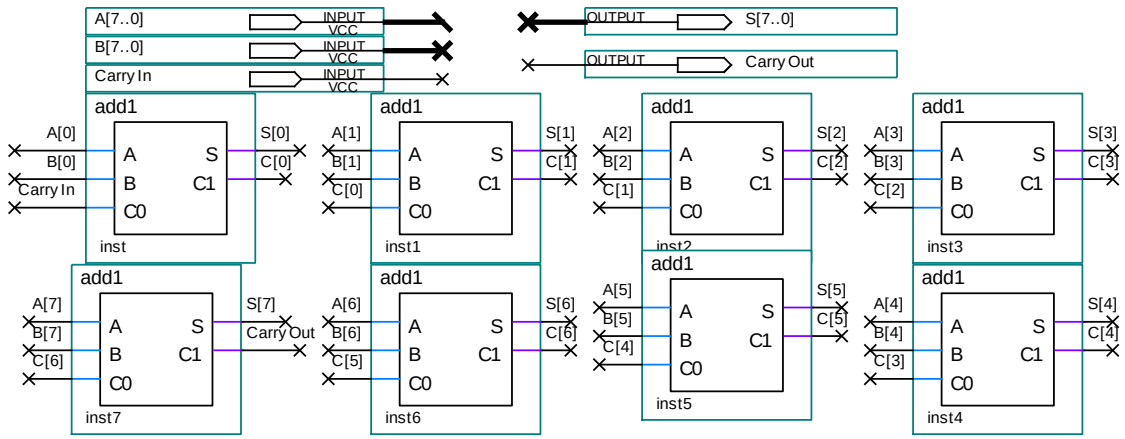


图 4-6 8 位加法器的原理图

4.4 32 位加减器

32 位加减器用于对两个 32 位二进制数进行加法或减法运算。它是先由 4 个 8 位的加法器级联成 32 位的加法器，然后像 1 位加法器变换成 1 位加减器那样，经过变换，最后实现的。

表 4-4 32 位加减器的真值表

输入			输出	
A[31..0]	B[31..0]	SUB	S[31..0]	CarryOut
A	B	符号标记	A+(B XOR SUB)+SUB	符号标记

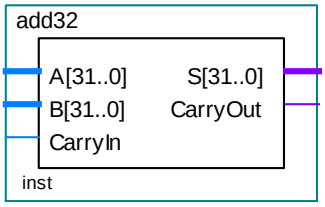


图 4-7 32 位加减器电路符号

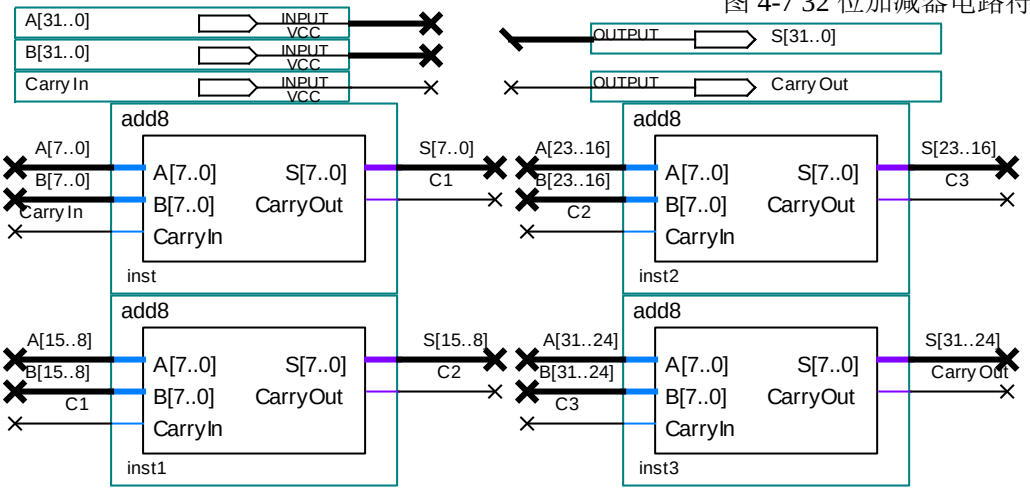


图 4-8 32 位加法器的原理图

32 位加减器有两个输出 S 和 CarryOut，其中 S 为加法器的和，CarryOut 用于判断做减法运算的结果是否为负数。有三个输入 A、B 和 SUB，其中 SUB 用于标记是

否做减法运算。32 位加减器的电路符号如图 4-7 所示，真值表如表 4-4 所示。（一般，做减法运算时，得到结果都为正，即通常都是在 A > B 的情况下）。

(1)首先由描述级联，创建 32 位加法器的原理图，如图 4-8 所示。

(2)然后由 32 位加法器和逻辑关系式，创建 32 位加减器的原理图，如图 4-9 所示。

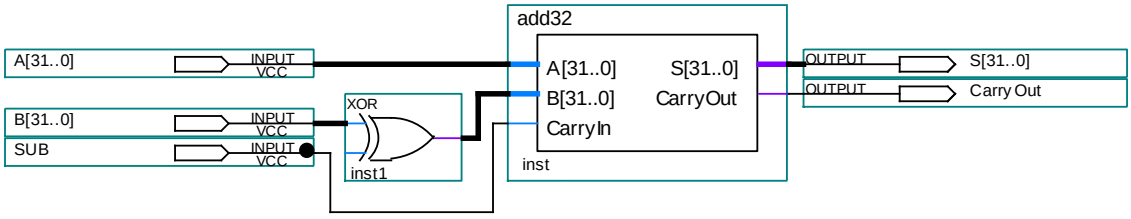


图 4-9 32 位加减器的原理图

5 移位运算器

移位运算器就是实现将二进制数向左边或向右边移动多少位。例如：二进制数 01001 它向左移一位后为 10010，右移一位后为 00100。通过这个例子是否感觉到右移时，可以在空出的位置根据情况补上 0 或 1。故它根据二进制数是有无符号分为逻辑移位运算和算术移位运算。这里的移位运算器要实现数据的逻辑左移 SLL、逻辑右移 SRL、算术右移 SRA（算术左移与逻辑左移一样）。

其中算术右移 SRA，把操作数看成带符号数。对操作数进行移位（要移动数的第 0 位（符号位）不变。右移后在空出的位置补与第 0 位相同的 1 或 0）。逻辑左移 SLL、逻辑右移 SRL，把操作数看成无符号数。对操作数进行移位（不管左右移，移后在空出的位置上补 0）。移位运算器的符号如图 5-1 所示，真值表如表 5-1 所示。

表 5-1 移位运算器的真值表

输入				输出
D[31..0]	SA[4..0]	Right	Arith	SH[31..0]
A	B	右移标志(否则左移)	算术运算标志(否则逻辑运算)	D 移位 SA 位

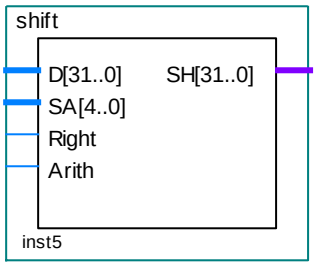


图 5-1 移位运算器电路符号

5.1 原理图设计移位运算

由真值表可得，当 ARITH=1，RIGHT=1 时，完成算术右移运算；当 ARITH=1，RIGHT=0 时，完成算术左移运算；当 ARITH=0，RIGHT=0 时，完成逻辑左移运算；当 ARITH=0，RIGHT=1 时，完成逻辑右移运算。但算术左移运算的效果与逻辑左移运算一样。

由于一个 32 位的数最多的移动位数为 32 位，所以移动位数 SA 设计为一个 5 位的数组。根据 SA[4..0]中 SA[4] 为 1 时，表示 D 需要移动的位数为 16 位，SA[3] 为 1 时，表示 D 需要移动的位数为 8 位，SA[2] 为 1 时，表示 D 需要移动的位数为 4 位，SA[1] 为 1 时，表示 D 需要移动的位数为 2 位，SA[0] 为 1 时，表示 D 需要移动的位

数为 1 位。例如，移动的位数为 9 位，则 SA[3]和 SA[0]为 1，其它的为 0，移位运算器进行移位时，首先由于 SA[3]为 1，将 SA[4]移位后的结果 S4 移动 8 位（由于 SA[4]为 0，SA[4]移位后的结果 S4 没有移动，即还为 D），得到结果 S3，然后由于 SA[0]为 1，将 SA[1]移位后的结果 S1 移动 8 位（同理，由于 SA[2]和 SA[1]为 0，SA[1]移位后的结果 S1 没有移动，即还为 S3）。每次移动都经过从 D 开始→S4→S3→S2→S1 到得到移位后的结果 SH。

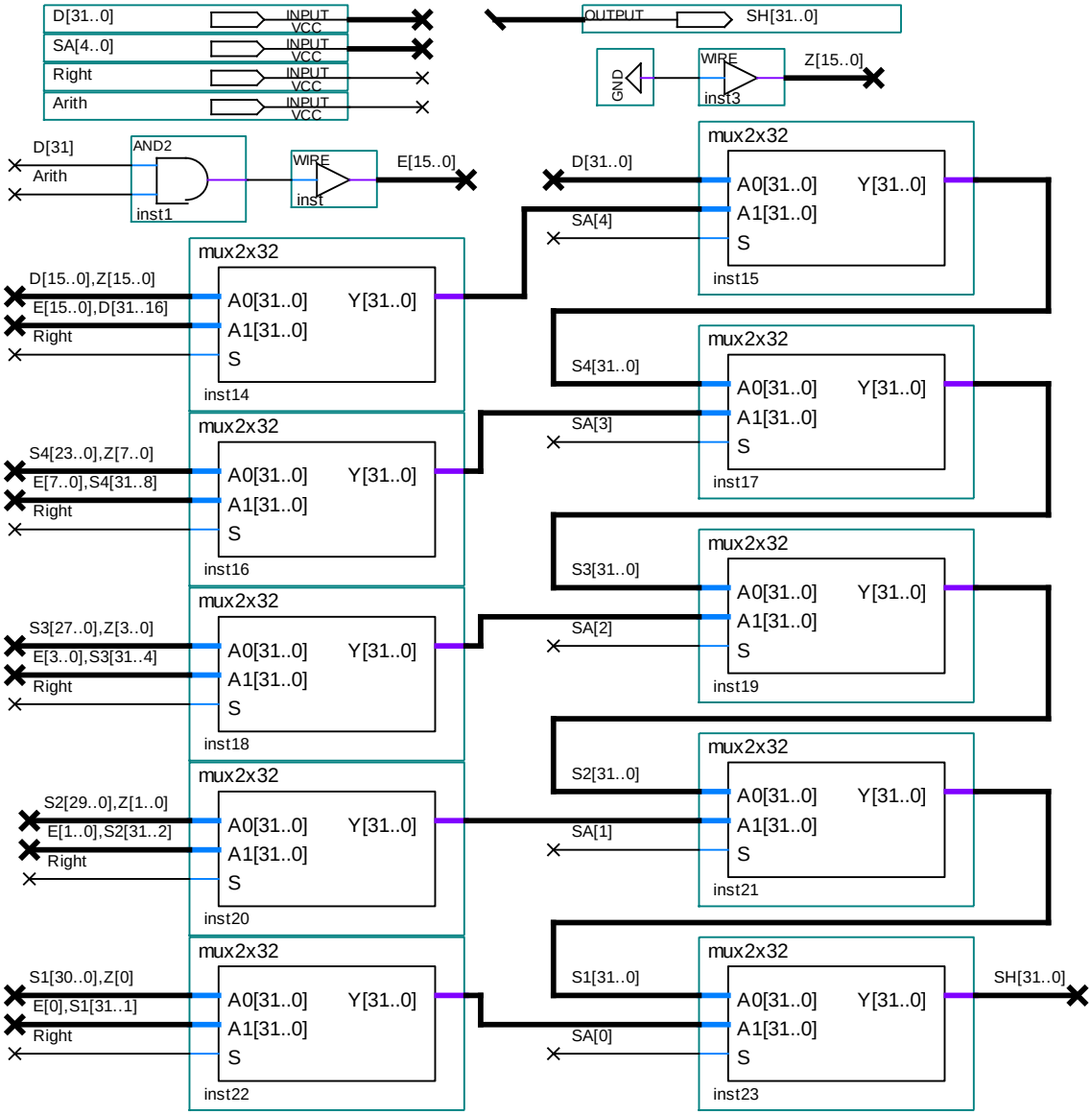


图 5-2 移位运算器的原理图

由描述和真值表可创建移位运算器的原理图，如图 5-2 所示。WIRE 元件是对位数进行任意位扩展，如：1 扩展为 11111。

5.2 移位运算的仿真验证

(1)打开 cpu 项目。

(2)创建一个原理图文件 新建一个原理图文件，在原理图编辑窗口中按照图 6-2 创建移位运算器的原理图，保存为“shift.bdf”。

(3)创建一个 VHDL 文件 新建一个 VHDL File 文件，在 VHDL 程序编辑窗口中输入移位运算器的 VHDL 程序，保存为“V_shift.vhd”。

(4)编译 将要编译的文件设置为顶层文件，编译。

(5)创建波形文件 通过编译后，新建 Vector Waveform File 文件，将所有的信号节点导入波形图编辑窗口中。对所有信号节点前面有 或 图标的，按如图 5-3 所示进行值设置。然后，保存该波形文件为“shitf.vwf”和“V_shift.vwf”。

(6)功能仿真 原理图和 VHDL 最后生成的功能仿真波形图应一样，如图 5-3 所示。

(7)波形分析 由如图 5-3 所示 0ns 到 100.0ns 这一时段，D 为 40000000，SA 为 10，Arith 为 1，Right 为 1，故 SH 的值应该为 80000000，图中也正好此时段 SH 的值为 80000000，因此得证。

00000002 的二进制　　0000 0000 0000 0000 0000 0000 0000 0010

01 二进制　　0000 1

算术右移 SA 位，即 1 位，得到 00000001　0000 0000 0000 0000 0000 0000 0000 0001

最后可以根据原理图生成移位运算器元器件 shift.bsf。

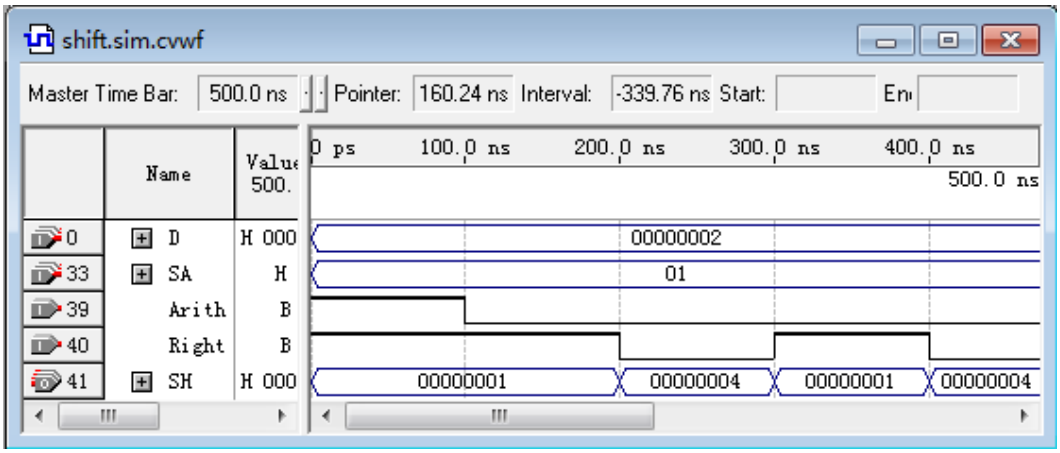


图 5-3　移位运算器的仿真波形图

6 算术逻辑运算器

运算器是计算机中执行各种算术和逻辑运算操作的部件。运算器的基本操作包括加、减、乘、除四则运算，与、或、非、异或等逻辑操作，以及移位和比较等操作，亦称算术逻辑部件 ALU(Arithmetic Logic Unit)。

这里将主要叙述算术逻辑运算器的加和减运算（ADD、SUB）、与和或逻辑操作（AND、OR）、异或和高位加载立即数（XOR、LUI）、移位运算（SLL、SRL、SRA）。高位加载立即数（LUI）是指把 16 位的立即数左移 16 位，并把低 16 位设置成 0。把这些运算操作编码如下表 6-1 所示（其中 X 表示为 0 或 1 都可）。

表 6-1 ALU 运算操作编码

运算	ADD	AND	SUB	OR	XOR	LUI	SLL	SRL	SRA
ALUC[3..0]	X000	X001	X100	X101	X010	X110	0011	0111	1111

通过表 6-1 所示的编码，可以将表中的 9 种运算操作用 ALUC 加以区别。首先通过 ALUC[2]控制，用 32 位 2 选 1 多路选择器选择出 ADD 和 SUB 中的一个。同样，AND 和 OR、XOR 和 LUI 中也可以分别选择出一个。而三种移位运算需要用 ALUC[3]和 ALU[2]加以控制，从三个中选择出一个，其中 ALUC[3]表示 ARITH 是否是算术移位运算，ALUC[2]表示 RIGTH 是否是右移移位运算。最后，通过一个 32 位 4 选 1 多路选择器从选择出的四个选择出一个作为运算器 ALU 的结果。同时，通过做减法运算判断结果是否为 0 来做比较运算。运算器的电路符号如图 6-1 所示。

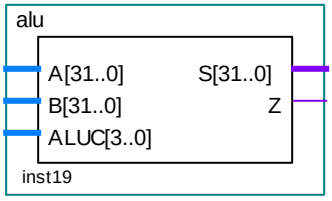


图 6-1 算术逻辑运算器的电路符号

6.1 0 操作数检测模块

由于比较操作是通过做减法运算判断结果是否为 0 来实现的，所以需要一个 0 操作数检测模块。它的功能就是判断输入的 32 位数的每一位都为 0。它的原理图如图 6-2 所示。

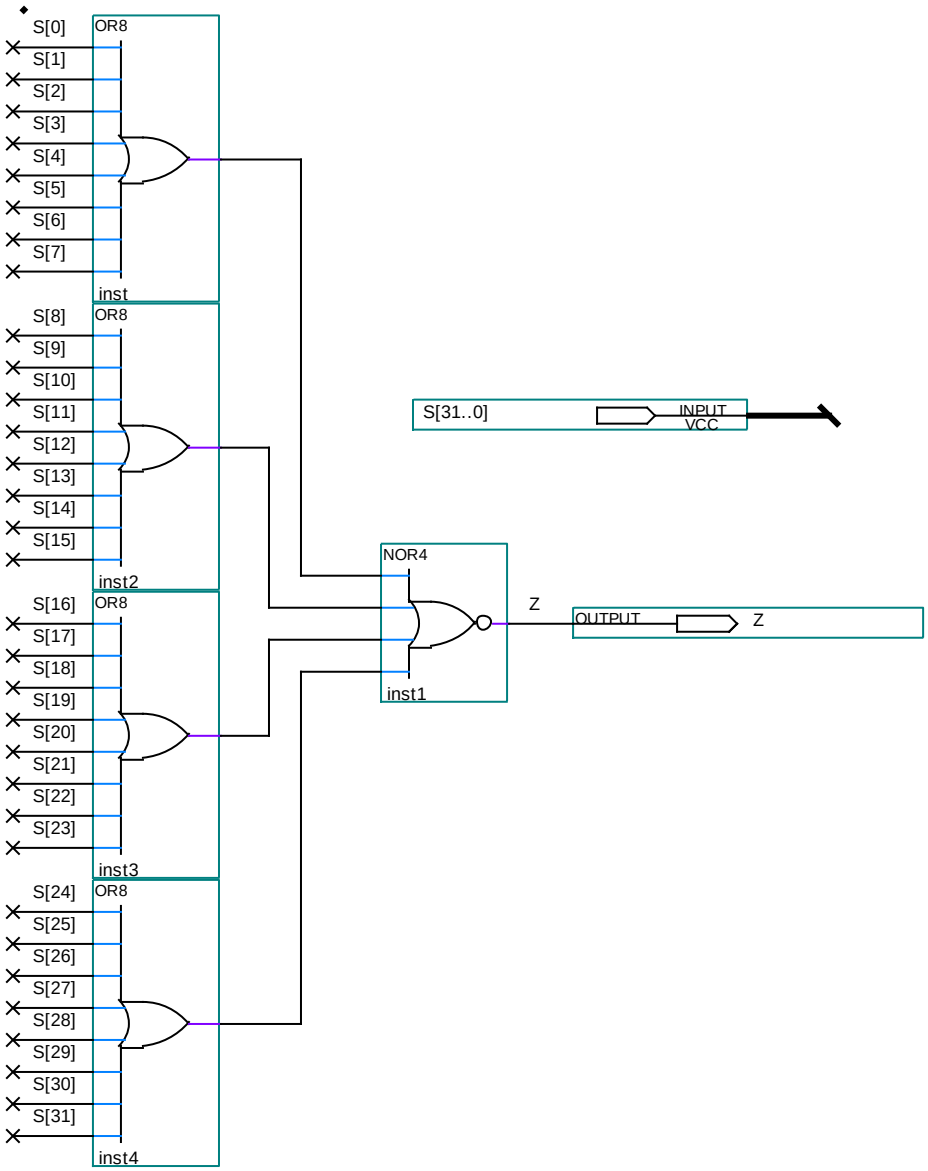


图 6-2 0 操作数检测模块的原理图

6.2 原理图设计算术逻辑运算器

如表 6-1 所示，通过输入 ALUC 的控制选择，调用前几章节的 32 位 2 选 1 多路选择器、32 位 4 选 1 多路选择器、32 位的加减器、移位运算器和 0 操作数检测模块，再加上一些与门、或门、异或门来实现 ALU 算术逻辑运算器。它的原理图如图 6-3 所示。

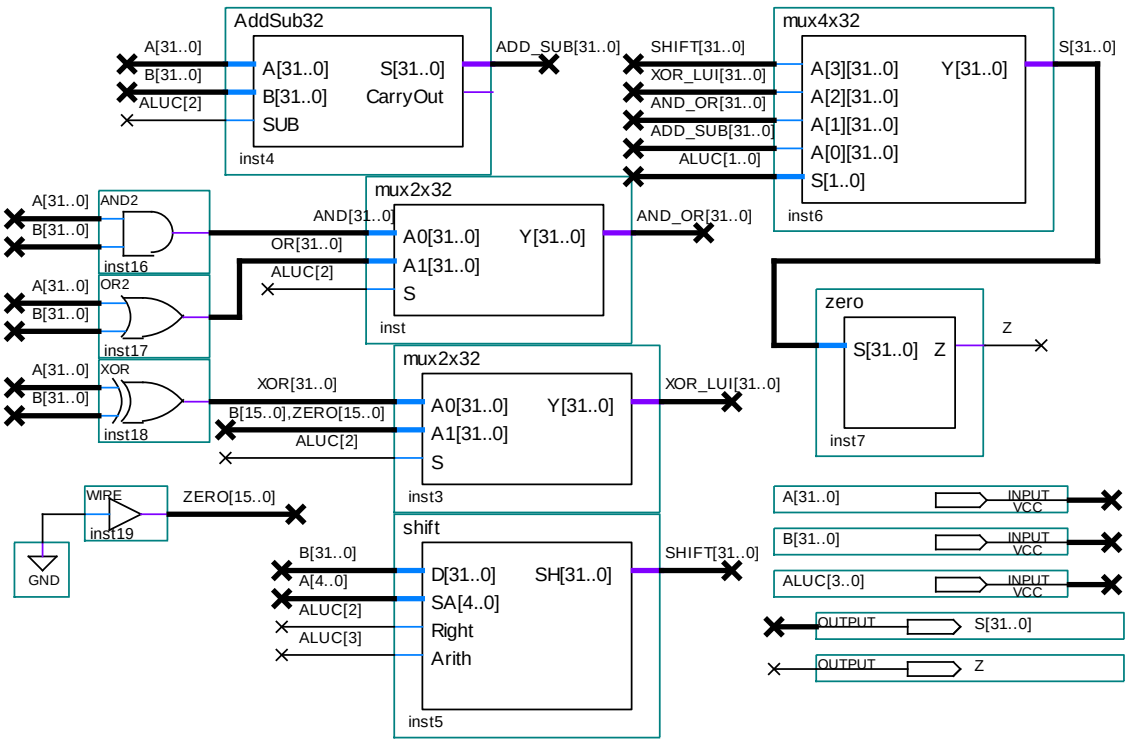


图 6-3 ALU 算术逻辑运算器的原理图

6.3 算术逻辑运算器的仿真验证

(1)打开 cpu 项目。

(2)创建一个原理图文件 新建一个原理图文件，在原理图编辑窗口中按照图 6-3 创建 ALU 算术逻辑运算器的原理图，保存为“alu.bdf”。

(3)创建一个 VHDL 文件 新建一个 VHDL File 文件，在 VHDL 程序编辑窗口中输入移位运算器的 VHDL 程序，保存为“V_alu.vhd”。

(4)编译 将要编译的文件设置为顶层文件，编译。

(5)创建波形文件 通过编译后，新建 Vector Waveform File 文件，将所有的信号节点导入波形图编辑窗口中。对所有信号节点前面有 或 图标的，按如图 6-4 所示进行值设置。然后，保存该波形文件为“alu.vwf”和“V_alu.vwf”。

(6)功能仿真 原理图和 VHDL 最后生成的功能仿真波形图应一样，如图 6-4 所示。

(7)波形分析 由如图 6-4 所示 200.0ns 到 300.0ns 这一时段，A 为 55555555，B 为 55555555，ALUC 为 0010，故做减法运算 S 的值应该为 00000000，图中也正好此时段 S 的值为 00000000，因此得证。

最后可以根据原理图生成算术逻辑运算器元器件 alu.bsf。

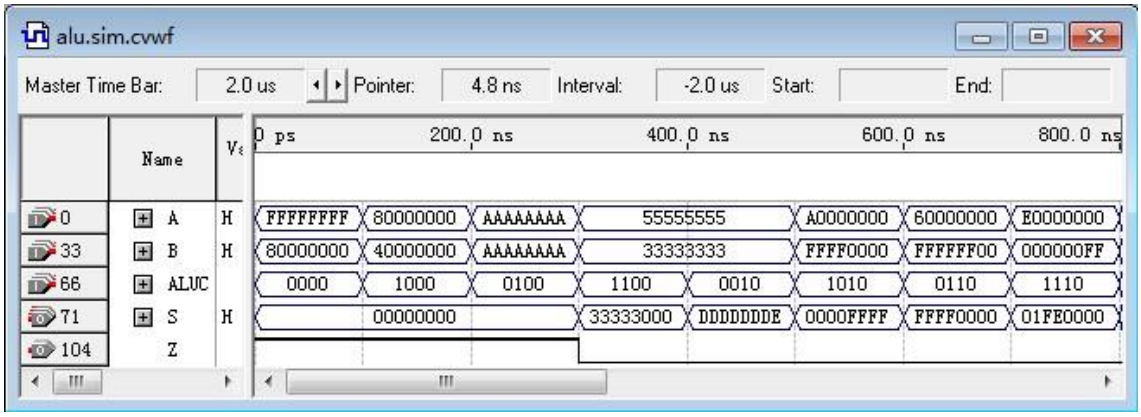


图 6-4 ALU 算术逻辑运算器的仿真波形图

7 寄存器堆

寄存器是数字电路中的基本模块，许多复杂的时序逻辑电路都是由它构成的。在数字系统中，寄存器是一种在某一特定信号的控制下用于存储一组二进制数据的时序逻辑电路。通常使用触发器构成寄存器，把多个 D 触发器的时钟端连接起来就可以构成一个存储多位二进制的寄存器。

在 CPU 设计中，寄存器堆是一个必不可少的可以保存指令和数据的器件，是 RISC 微处理器的核心，所有内部、外部数据读取都直接和它发生关系，它由一组寄存器组成，只要给出该寄存器堆中寄存器的编号，则其中的内容都可以读或者写。

这里将根据寄存器堆的特点，介绍了利用原理图和 VHDL 语言，进行 8 位、32 位的寄存器和 32 位的寄存器堆的设计，详细叙述了其工作原理及设计思想，并给出了实现部分的仿真波形。

表 7-1 寄存器号译码的真值表

输入	ENA	0	1	1	1	1	1	1	1	1
	N[4..0]	XX	00	01	02	03	04	05	06	07
输出	EN[31..0]	00000000	00000001	00000002	00000004	00000008	00000010	00000020	00000040	00000080
输入	N[4..0]	XX	08	09	0A	0B	0C	0D	0E	0F
输出	EN[31..0]	00000000	00000100	00000200	00000400	00000800	00001000	00002000	00004000	00008000
输入	N[4..0]	XX	10	11	12	13	14	15	16	17
输出	EN[31..0]	00000000	00010000	00020000	00040000	00080000	00100000	00200000	00400000	00800000
输入	N[4..0]	XX	18	19	1A	1B	1C	1D	1E	1F
输出	EN[31..0]	00000000	01000000	02000000	04000000	08000000	10000000	20000000	40000000	80000000

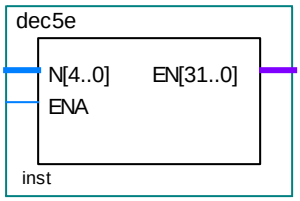


图 7-1 寄存器号译码的电路符号

7.1 寄存器号译码

MIPS 指令集的单周期 CPU 系统定义了 32 个 32 位的通用寄存器，这 32 个 32 位的通用寄存器也就是所要设计的寄存器堆，为了定位 32 个寄存器(R0~R31)，这里先

实现一个寄存器号译码，用来确定具体是哪一个寄存器，它是将一个5位的输入N[4..0]译码成 32 位的 EN[31..0]输出，例如，5 位输入 00010，则输出的 32 位 EN[31..0]的 EN[2]为 1，其它的位数都为 0，即选中了 R2 寄存器。同时，它还有一个使能端 ENA，当 ENA 为 0 时，EN[31..0]为 0。它的真值表如表 7-1 所示，电路符号如图 7-1 所示。

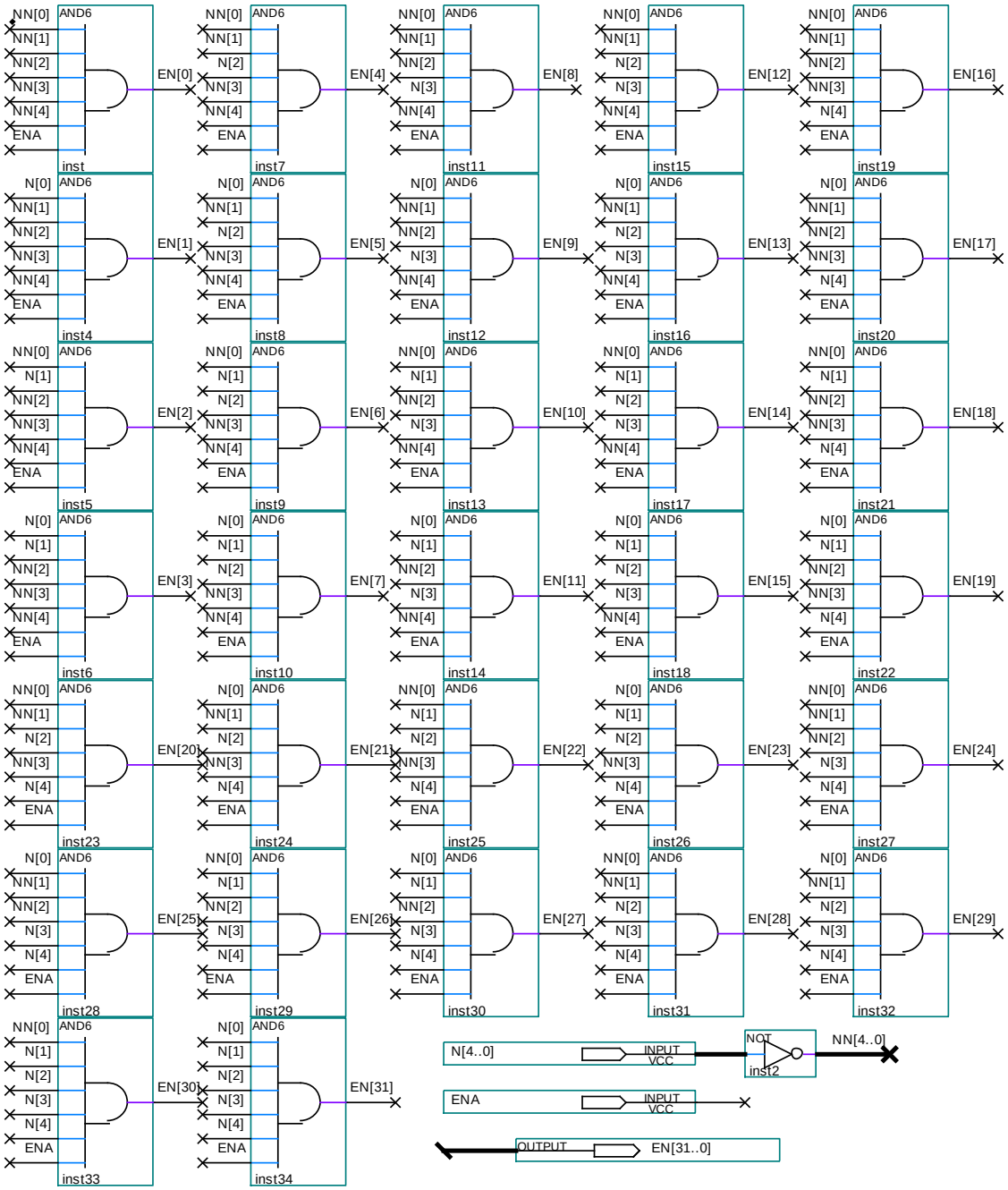


图 7-2 寄存器号译码的原理图

由描述和真值表创建寄存器号译码的原理图，如图 7-2 所示。

7.2 8 位触发器

8 位触发器的电路符号如图 7-3 所示。输入信号：一个 8 位数据源 D，1 位 CLRN 复位信号，1 位 EN 使能信号，1 位 CLK 时钟信号；输出信号：8 位 Q。它可以由 8 个 1 位 DFFE 触发器经过级联后得到。这 8 个 1 位 DFFE 触发器的每个 D 分别为 D[0]到 D[7]，Q 分别为 Q[0]到 Q[7]，时钟信号都为 CLK。（注意：要相对应，如：第一个 1 位 DFFE 触发器的 D 为 D[0]，则 Q 应为 Q[0]）。它的真值表如表 7-2 所示。

表 7-2 8 位触发器的真值表

输入				输出
D	CLK	EN	CLRN	Q
X	X	0	X	0
X	X	X	0	0
D	下降沿	1	1	D0
D	上升沿	1	1	D

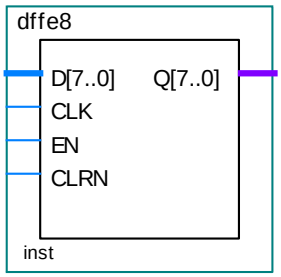


图 7-3 8 位触发器电路符号

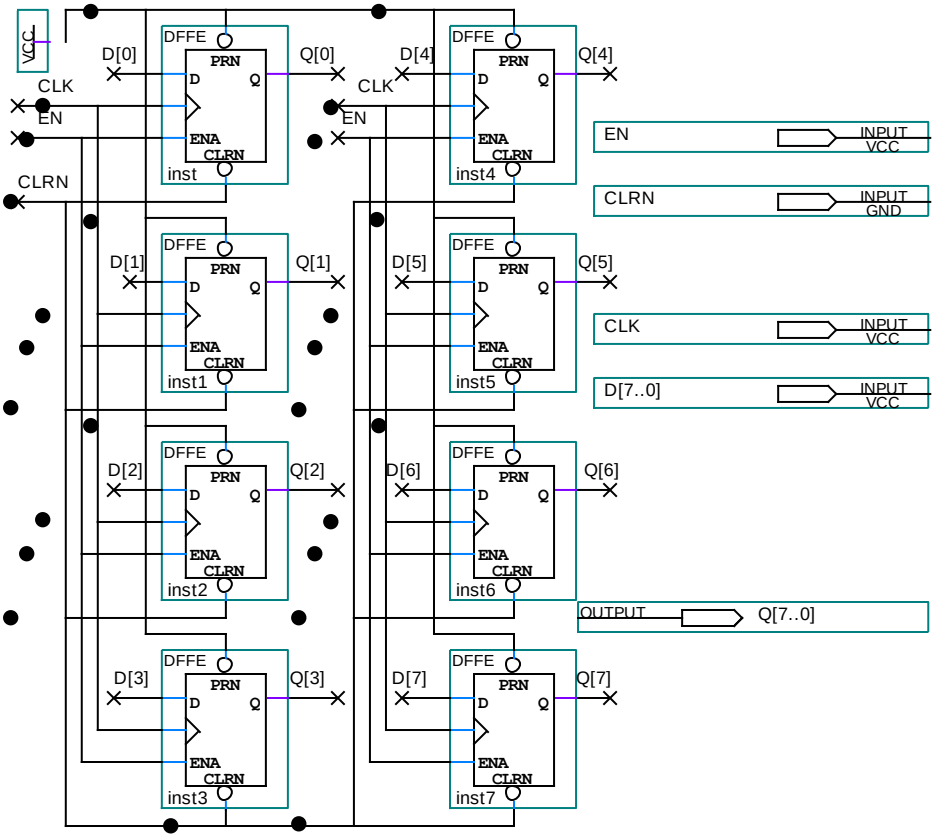


图 7-4 8 位触发器的原理图

由描述和真值表创建 8 位触发器的原理图，如图 7-4 所示。

7.3 32 位触发器

32 位触发器的电路符号如图 7-5 所示。输入信号：一个 32 位数据源 D，1 位 CLRN 复位信号，1 位 EN 使能信号，1 位 CLK 时钟信号；输出信号：32 位 Q。它可以由 4 个 8 位触发器经过级联后得到，时钟信号都为 CLK。它的真值表如表 7-3 所示。

表 7-3 32 位触发器的真值表

输入				输出
D	CLK	EN	CLRN	Q
X	X	0	1	D0
X	X	X	0	0
D	下降沿	1	1	D0
D	上升沿	1	1	D

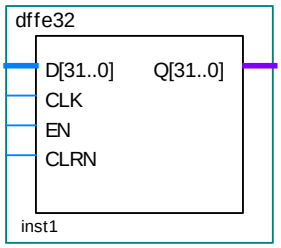


图 7-5 32 位触发器电路符号

由描述级联和真值表创建 32 位触发器的原理图，如图 7-6 所示。

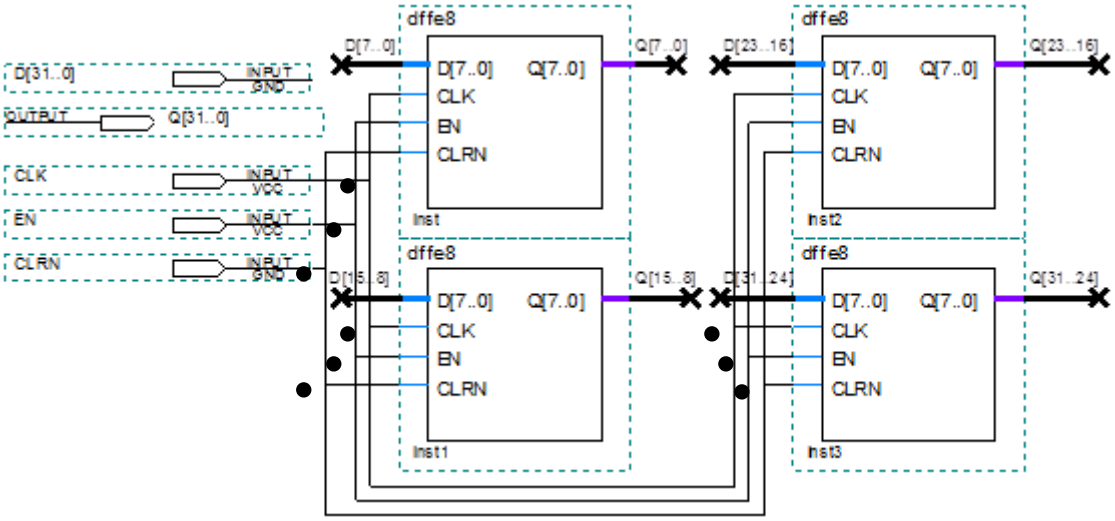


图 7-6 32 位触发器的原理图

7.4 32 位寄存器

32 位寄存器的电路符号如图 7-7 所示。输入信号：一个 32 位数据源 D，1 位 CLRN 复位信号，32 位 EN 使能信号，1 位 CLK 时钟信号；输出信号：Q[31..0][31..0]。它可以由 32 个 32 位触发器经过级联后得到，通过 EN[31..0]使能信号中的每一位来控制一个 32 位触发器，当 EN 中的有一位为 0 时，它控制的 32 位触发器的值保持原值不

变。其中，EN[0]没有控制 0 号的 32 位触发器，因为，0 号的 32 位触发器被硬接到了 0 上，当读它时，将始终得到 0；当写它时，将没有任何动作，没有任何影响，这样，如果某些指令需要丢弃运算结果，就可以把它作为保存结果的目的寄存器，如果某些指令需要以 0 作为操作数，就可以把它作为源数据寄存器。时钟信号都为 CLK。

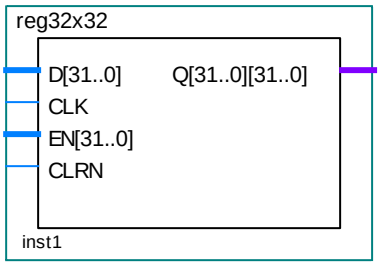


图 7-7 32 位触发器电路符号

由描述级联和真值表创建 32 位寄存器的原理图，如图 7-8 所示。

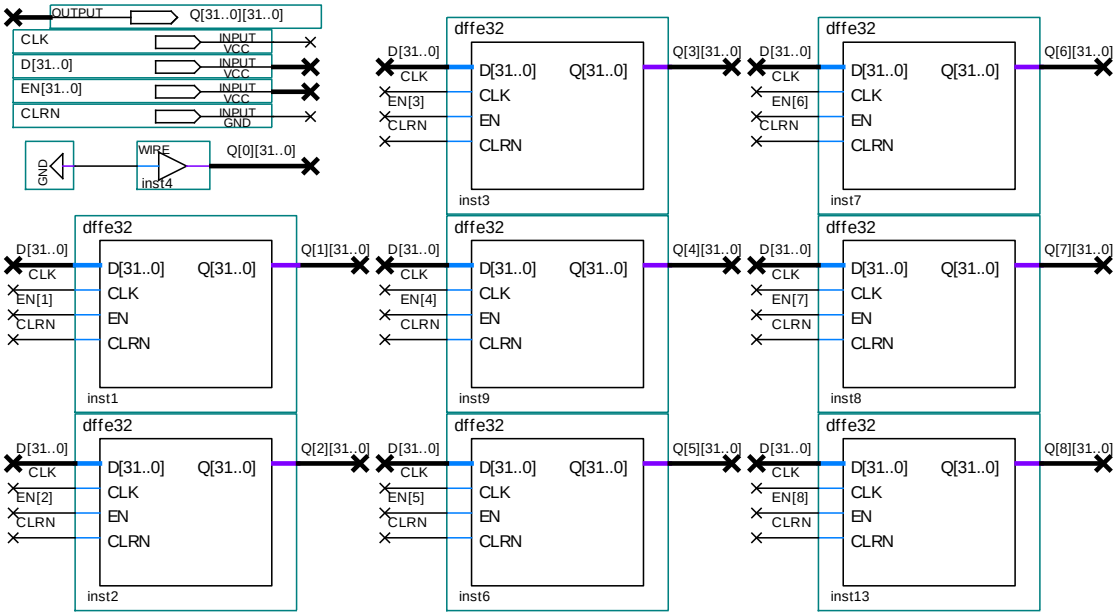


图 7-8 32 位寄存器的原理图

7.5 32 位寄存器堆

寄存器堆由一组寄存器组成，CPU 在运算过程中，将一些运算数据暂存在寄存器堆中。由于，MIPS 体系结构中 R 类型指令有 3 个操作数，其中两个操作数需要从寄存器堆中读出，作为算术逻辑运算器 ALU 的输入，另一个操作数是算术逻辑运算器 ALU 的输出，需要写入到寄存器堆中。这样，寄存器堆就需要 1 个写通道和 2 个读通道。在寄存器堆读操作中，需要给出要读出寄存器的寄存器号，然后，寄存器堆将该

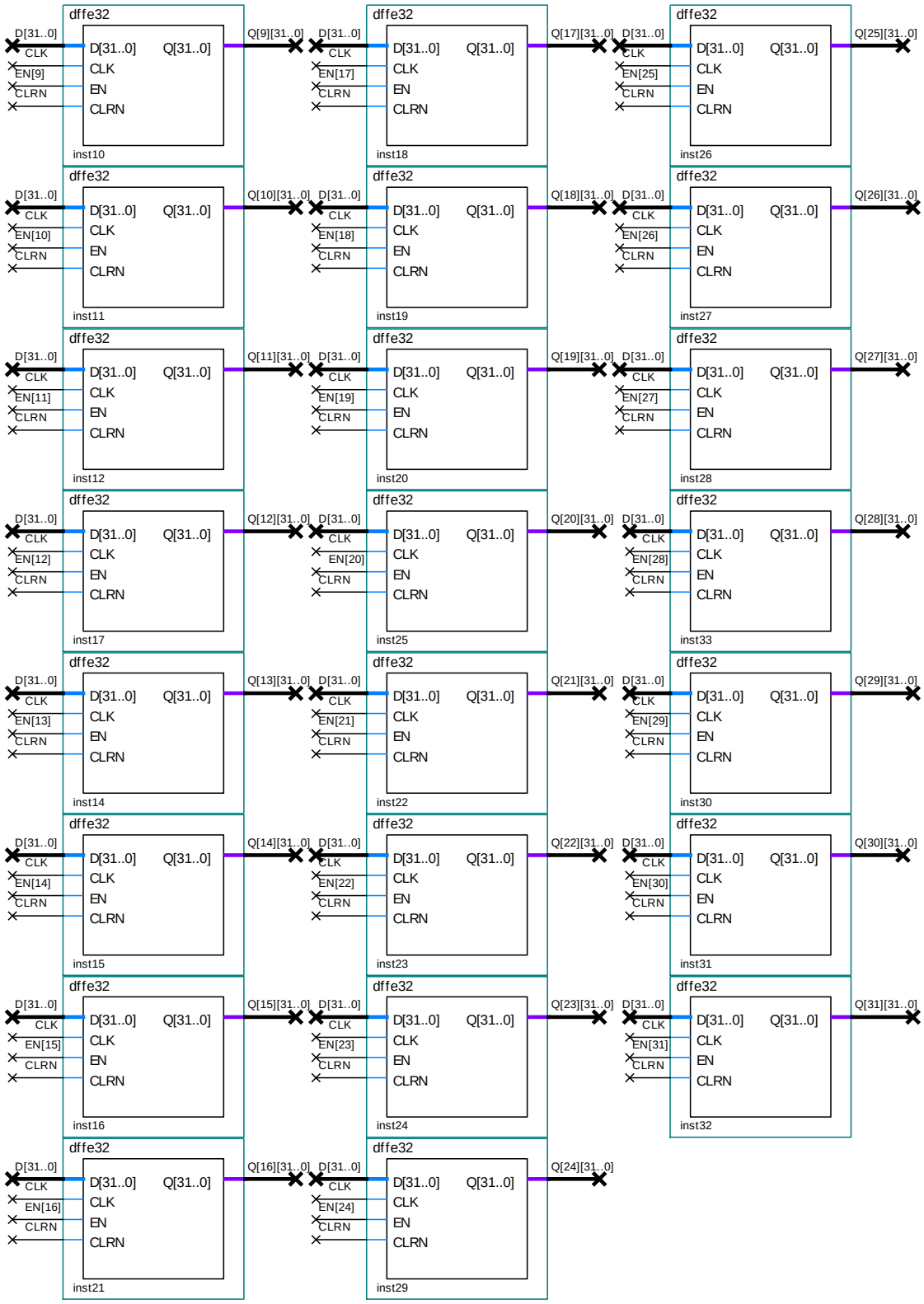


图 7-8 32 位寄存器的原理图（续）

寄存器号索引的寄存器中数据输出。在寄存器堆写操作中，需要给出要写入寄存器的

寄存器号和要写入的数据，然后，寄存器堆将把数据写入到该寄存器号索引的寄存器中。

寄存器堆的输入信号：N1[4..0]读取通道 0 的寄存器号，N2 读取通道 1 的寄存器号，N0 写通道的寄存器号，D 写通道的输入数据，WE 写使能信号。寄存器堆的输出信号：Q1 读通道 1 的输出结果，Q2 读通道 2 的输出结果。

由描述创建 32 位寄存器堆的原理图，如图 7-9 所示。

;

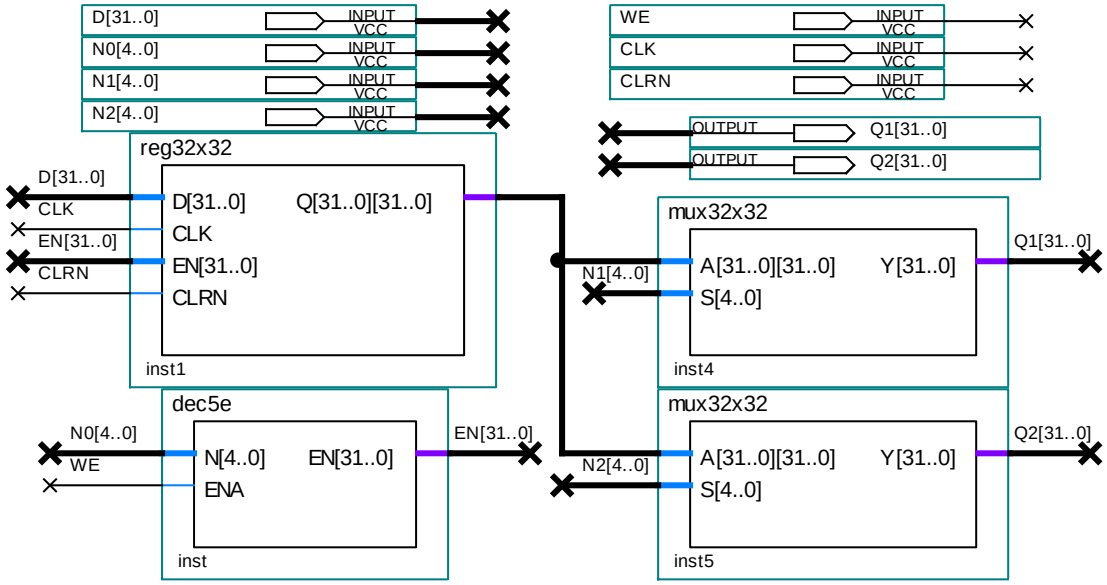


图 7-9 32 位寄存器堆的原理图

8 计算机主机系统设计

早期的计算机术语中，把运算器和控制器合在一起称为中央处理器，简称为 CPU。现在一般存储器也放入到 CPU 中，称为中央处理机，也就是主机系统。当然，这些器件之间的通信是通过总线来实现的。

运算器就好像是一个由电子线路构成的算盘，它的主要功能是进行加、减、乘、除等算术运算。除此之外，还可以进行逻辑运算，通常称为 ALU（算术逻辑运算部件）。存储器的功能是保存或记忆解题的原始数据和解题步骤。为此，在运算前需要把运算的数据和解题步骤通过输入设备送到存储器中保存起来。注意，不论是数据，还是解题步骤，在存放到存储器以前，全已变成 0 或 1 表示的计算机能识别的二进制代码。控制器是计算机中发号施令的部件，它控制计算机的各部件有条不紊地进行工作。具体的讲，控制器的任务是从内存中取出题解步骤加以分析，然后执行某种操作。

总线是构成计算机系统的骨架，是多个系统部件之间进行数据传送的公共通路。借助总线，计算机在各系统部件之间实现传送地址、数据和控制信息的操作。在 EDA 技术中，计算机主机系统设计是难度较大的实验，主要是进行一台完整计算机的数据通路、控制信号以及运算器等逻辑设计与详细设计，同时经过主机系统运行由第 3 章 MIPS 指令集构成的汇编程序，以测试该机器逻辑设计的正确性。本节讲述 CPU 核心部件控制器、主机系统的构成以及程序的运行。

8.1 跳转指令寄存器指定

跳转指令寄存器指定是从 32 位的寄存器堆中选择出一个 32 位的寄存器用于读或写数据。由于 JAL 指令要把分支延迟槽下一条指令的指令地址保存在寄存器 31 号中，故需要设置一个控制信号 CALL，当 CALL=1 时，选择 31 号寄存器，为 JAL 跳转指令提供寄存器，当 CALL=0 时，从 32 位的寄存器堆中根据地址选择出一个 32 位的寄存器用于读或写数据。它的真值表如表 8-1 所示，电路符号如图 8-1 所示。

表 8-1 跳转指令寄存器指定的真值表

输入	CALL	1	0
	REGN[4..0]	X	REGN[4..0]
输出	WN[4..0]	11111	REGN[4..0]

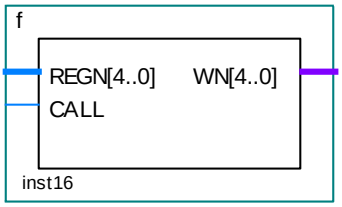


图 8-1 跳转指令寄存器指定电路符号

由描述和真值表创建跳转指令寄存器指定的原理图，如图 8-2 所示。

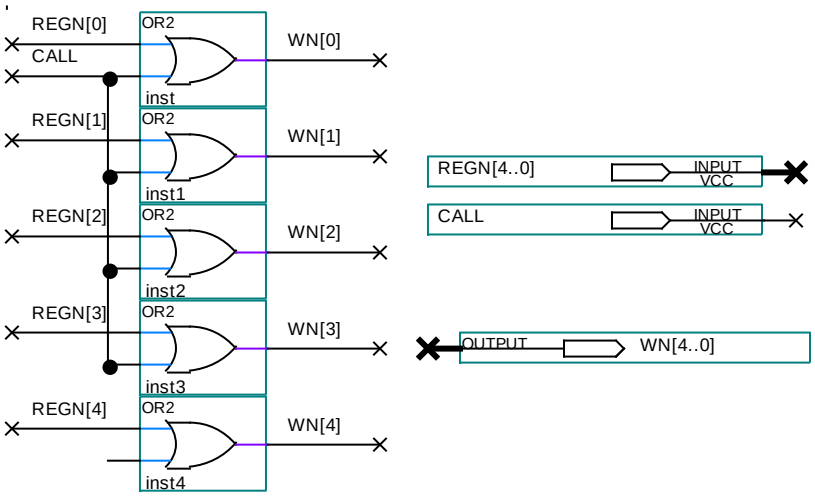


图 8-2 跳转指令寄存器指定的原理图

8.2 指令译码

指令译码是对取指令操作中得到的指令进行译码，确定这条指令需要完成的操作，从而产生相应的控制信号，驱动执行状态中的各种动作，它是控制器的一个重要组成部分。在 MIPS 指令集中，根据操作码 OP 和功能码 FUNC 来区别指令。根据前面列出的 20 条指令编码，其逻辑表达式如下，其中，OP[5..0]为操作码 OP，FUNC[5..0]为功能码 FUNC，电路符号如图 8-3 所示。

TYPE=$\overline{OP[5]}$▪$\overline{OP[4]}$▪$\overline{OP[3]}$▪$\overline{OP[2]}$▪$\overline{OP[1]}$▪$\overline{OP[0]}$

add=TYPE▪FUNC[5]▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪$\overline{FUNC[2]}$▪$\overline{FUNC[1]}$▪$\overline{FUNC[0]}$

sub=TYPE▪FUNC[5]▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪$\overline{FUNC[2]}$▪FUNC[1]▪$\overline{FUNC[0]}$

and=TYPE▪FUNC[5]▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪FUNC[2]▪$\overline{FUNC[1]}$▪$\overline{FUNC[0]}$

or=TYPE▪FUNC[5]▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪ FUNC[2]▪$\overline{FUNC[1]}$▪ FUNC[0]

xor=TYPE▪FUNC[5]▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪ FUNC[2]▪ FUNC[1]▪$\overline{FUNC[0]}$

sll=TYPE▪$\overline{FUNC[5]}$▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪$\overline{FUNC[2]}$▪$\overline{FUNC[1]}$▪$\overline{FUNC[0]}$

srl=TYPE▪$\overline{FUNC[5]}$▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪$\overline{FUNC[2]}$▪ FUNC[1]▪$\overline{FUNC[0]}$

sra=TYPE▪$\overline{FUNC[5]}$▪$\overline{FUNC[4]}$▪$\overline{FUNC[3]}$▪$\overline{FUNC[2]}$▪ FUNC[1]▪ FUNC[0]

jr=TYPE▪$\overline{FUNC[5]}$▪$\overline{FUNC[4]}$▪ FUNC[3]▪$\overline{FUNC[2]}$▪$\overline{FUNC[1]}$▪$\overline{FUNC[0]}$

addi=$\overline{OP[5]}$▪$\overline{OP[4]}$▪OP[3]▪$\overline{OP[2]}$▪$\overline{OP[1]}$▪$\overline{OP[0]}$

andi=$\overline{OP[5]}$▪$\overline{OP[4]}$▪ OP[3]▪ OP[2]▪$\overline{OP[1]}$▪$\overline{OP[0]}$

ori=$\overline{OP[5]}$▪$\overline{OP[4]}$▪ OP[3]▪ OP[2]▪$\overline{OP[1]}$▪ OP[0]

xori=$\overline{OP[5]}$▪$\overline{OP[4]}$▪ OP[3]▪ OP[2]▪ OP[1]▪$\overline{OP[0]}$

lw= OP[5]▪$\overline{OP[4]}$▪$\overline{OP[3]}$▪$\overline{OP[2]}$▪ OP[1]▪ OP[0]
sw= OP[5]▪$\overline{OP[4]}$▪ OP[3]▪$\overline{OP[2]}$▪ OP[1]▪ OP[0]
beq=$\overline{OP[5]}$▪$\overline{OP[4]}$▪$\overline{OP[3]}$▪ OP[2]▪$\overline{OP[1]}$▪$\overline{OP[0]}$
bne=$\overline{OP[5]}$▪$\overline{OP[4]}$▪$\overline{OP[3]}$▪ OP[2]▪$\overline{OP[1]}$▪ OP[0]
lui=$\overline{OP[5]}$▪$\overline{OP[4]}$▪ OP[3]▪ OP[2]▪ OP[1]▪ OP[0]
j=$\overline{OP[5]}$▪$\overline{OP[4]}$▪$\overline{OP[3]}$▪$\overline{OP[2]}$▪ OP[1]▪$\overline{OP[0]}$
jal=$\overline{OP[5]}$▪$\overline{OP[4]}$▪$\overline{OP[3]}$▪$\overline{OP[2]}$▪ OP[1]▪ OP[0]

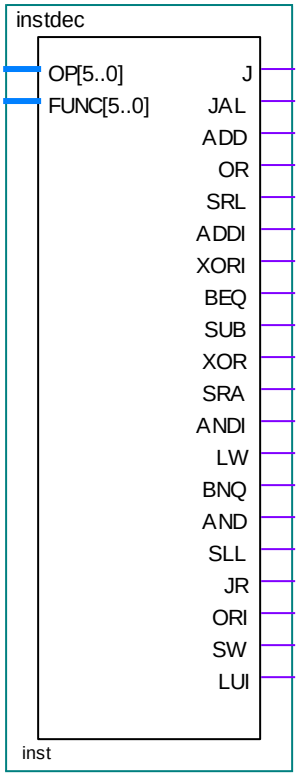


图 8-3 指令译码器的电路符号

由描述和逻辑关系式创建指令译码器的原理图，如图 8-4 所示，可以根据原理图生成指令译码器元器件 instdec.bsf。

8.3 控制部件设计

控制器是计算机中发号施令的部件，它控制计算机的各部件有条不紊地进行工作。具体的讲，控制器的任务是从内存中取出指令题解步骤加以分析，然后执行某种操作。PCSOURCE 是用于标记下一条指令的地址，通过计算得到的下一条指令有下列 4 种情况：为 0 时，下一条指令地址取 PC+4（在指令存储器中的地址是 A[7..2]）；为 1 时，下一条指令地址取分支跳转（BEQ、BNQ）指令的目标地址；为 2 时，下一

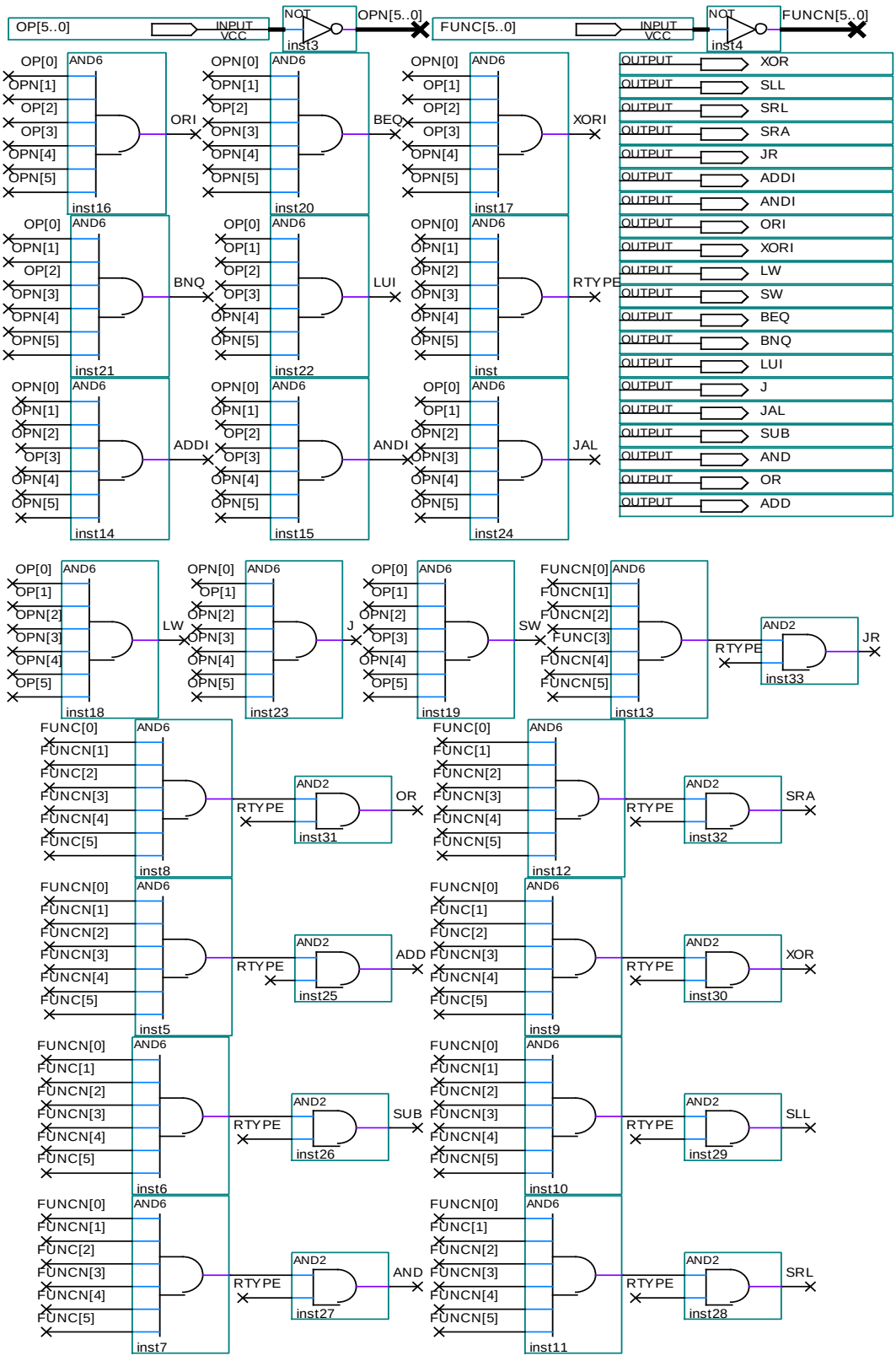


图 8-4 指令译码器的原理图

条指令地址取目的寄存器的值；为 3 时，下一条指令地址取跳转（J、JAL）指令的目标地址。CALL 标记（JAL 指令中要用）31 号寄存器，由于 JAL 指令要把分支延迟槽下一条指令的指令地址保存在寄存器 31 号中，故需要设置一个控制信号 CALL，当 CALL=1 时，选择 31 号寄存器，为 JAL 跳转指令提供寄存器，当 CALL=0 时，从 32 位的寄存器堆中根据地址选择出一个 32 位的寄存器用于读或写数据。M2REG（LW 指令）标记是将运算的结果直接写入寄存器堆，还是将数据存储器的数据取出后写入寄存器堆。ALUC 运算器的控制信号，用于选择做何种运算。WMEM 数据存储器的读写信号。SHIFT 标记移位运算，用于移位运算的运算数是否需要位数拓展。ALUIMM 标记立即数运算，用于有立即数参与运算前需要位数扩展。SEXT 标记参加运算的是有符号数。WREG 寄存器堆的读写信号。REGRT 标记指令中的[20..16]或[15..11]中的哪一种作为运算的目标寄存器。它的电路符号如图 8-5。根据第 2 章指令的执行分析，可以得出这 20 条指令对应的控制信号真值表如表 8-2 所示。

表 8-2 指令对应的控制信号真值表

	指令(R 类型)	add	sub	and	or	xor	sll	srl	sra	jr
入	op[5..0]	000000	000000	000000	000000	000000	000000	000000	000000	000000
	func[5..0]	100000	100010	100100	100101	100110	000000	000010	000011	001000
	z	x	x	x	x	x	x	x	x	x
出	pcsource[1..0]	00	00	00	00	00	00	00	00	10
	aluc[3..0]	x000	x100	x001	x101	x010	0011	0111	1111	x
	shift	0	0	0	0	0	1	1	1	x
	aluimm	0	0	0	0	0	0	0	0	x
	sext	x	x	x	x	x	x	x	x	x
	wmem	0	0	0	0	0	0	0	0	0
	wreg	1	1	1	1	1	1	1	1	0
	m2reg	0	0	0	0	0	0	0	0	x
	regrt	0	0	0	0	0	0	0	0	x
	call	0	0	0	0	0	0	0	0	x
	指令(I 类型)	addi	andi	ori	xori	lw	sw	beq	bne	lui
	op[5..0]	001	001	001	001	100	101	000	000	001

入		000	100	101	110	011	011	100	101	111
	func[5..0]	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx	xxxxxx
	z	x	x	x	x	x	x			x
出	pcsource[1..0]	00	00	00	00	00	00	01	10	00
	aluc[3..0]	x001	x001	x101	x010	x000	x000	x100	x100	x110
	shift	0	0	0	0	0	0	0	0	0
	aluimm	1	1	1	1	1	1	0	0	1
	sext	1	0	0	0	1	1	1	1	1
	wmem	0	0	0	0	0	1	0	0	0
	wreg	1	1	1	1	1	0	0	0	1
	m2reg	0	0	0	0	1	x	x	x	0
	regrt	1	1	1	1	1	x	x	x	1
	call	0	0	0	0	0	x	x	x	0

	指令(J 类型)	j	jal
入	op[5..0]	000010	000011
	func[5..0]	xxxxxx	xxxxxx
	z	x	x
出	pcsource[1..0]	11	11
	aluc[3..0]	x	x
	shift	x	x
	aluimm	x	x
	sext	x	x
	wmem	0	0
	wreg	0	1
	m2reg	x	x
	regrt	x	x
	call	x	1

根据表 8-2，可以得出这些控制信号的逻辑表达式如下。

pcsource[1]=j+jr+jal;

pcsource[0]= =beq z+bne $\bar{z}$+j+jal;
aluc[3]=sra;
aluc[2]=sub+or+srl+sra+ori+beq+bne+lui;
aluc[1]=xor+sll+srl+sra+xori+lui;
aluc[0]=and+or+sll+srl+sra+andi+ori;
shift=sll+srl+sra;
aluimm=addi+andi+ori+xori+lw+sw+lui;
sext=addi+lw+sw+beq+bne+lui;
wmem=sw;
wreg=add+sub+and+or+xor+sll+srl+sra+addi+andi+ori+xori+lw+lui+jal;
m2reg=lw;
regrt=addi+andi+ori+xori+lw+lui;
call=jal;

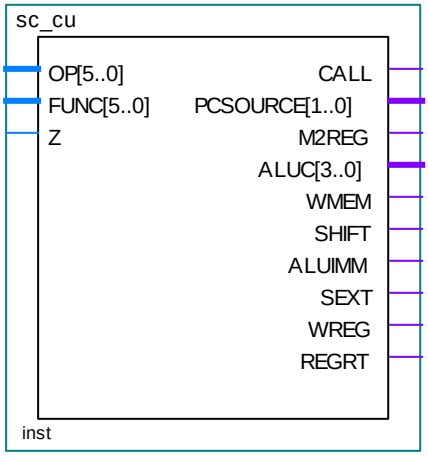


图 8-5 控制器的电路符号

由描述、调用 instdec 元件和逻辑关系式创建控制部件的原理图，如图 8-6 所示。最后可以根据原理图生成控制器元器件 sc_cu.bsf。

8.4 指令存储器

指令存储器是用于在计算机中保存指令。在本节中使用 LPM_ROM 来实现，它能保存 64 条指令字。如图 8-7 所示。指令存储器输入：A[31..0](指令地址)，实际使用的地址为 A[7..2]。指令的输出信号：DO[31..0]。指令存储器的元件符号如图 8-7 所示。真值表如表 8-3 所示，DO 表示在指令数据文件中地址 A 对应的指令数据。

表 8-3 指令存储器的真值表

输入	CLK	上降沿	下升沿
	A[31..0]	X	A
输出	DO[31..0]	保持不变	DO

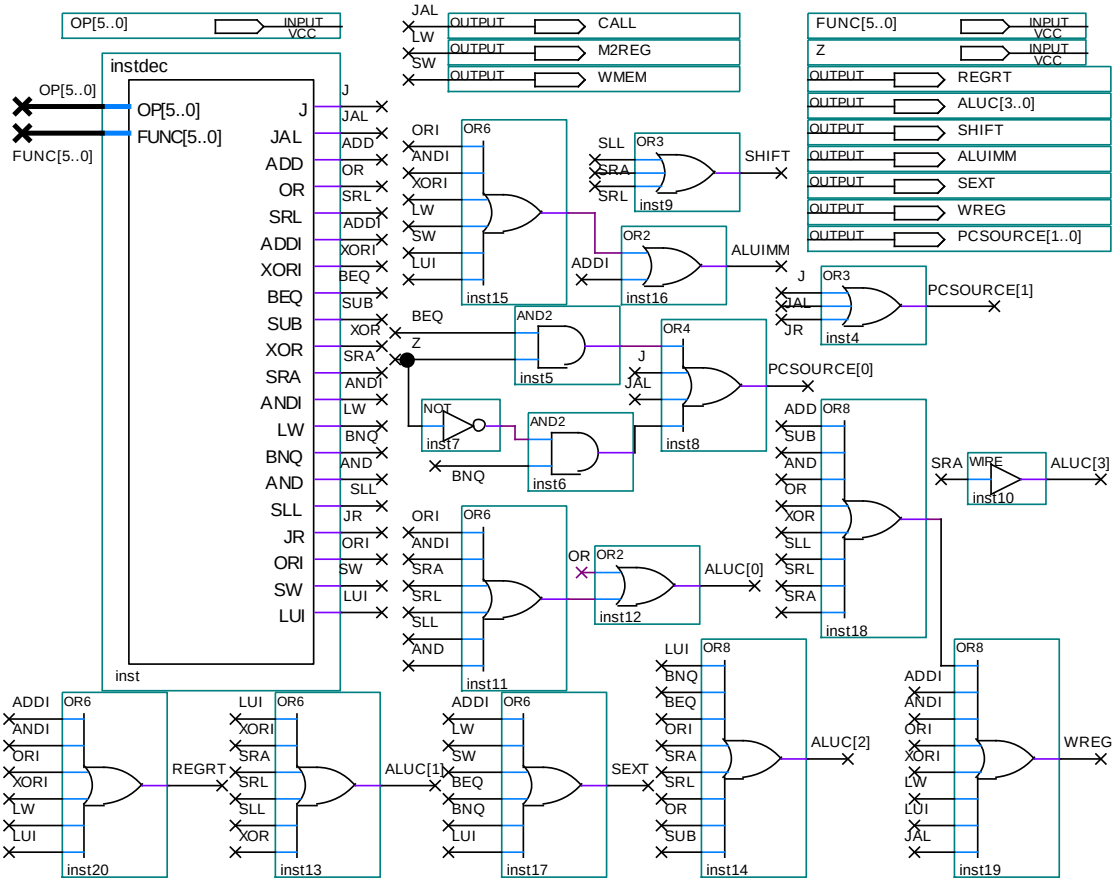


图 8-6 控制器的原理图

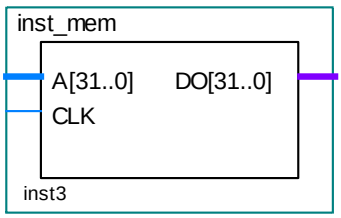


图 8-7 指令存储器的电路符号

8.5 数据存储器

数据存储器是用于在计算机中存放将要读出或写入的数据。在本节中使用 FPGA 中可调用的存储器结构 LPM_RAM_DQ 来实现，它能保存 32 条 32 位数据。数据存储器输入信号：WE 读写控制信号，A[31..0](指令地址)，实际使用的地址为 A[6..2]，DI 要写入的数据，CLK 读写控制信号，MEMCLK 时钟信号。数据的输出信号：DO[31..0]。数据存储器的元件符号如图 8-8 所示。真值表如表 8-4 所示，“↑”表示上升沿，“↓”表示下降沿。

表 8-4 数据存储器的真值表

输入	WE	0	0	1	1	X	X
	A[31..0]	A	A	A	A	A	A
	DI[31..0]	X	X	DI	DI	X	X
	CLK	X	X	0	0	1	1
	MEMCLK	↑	↓	↑	↓	↑	↓
输出	DO[31..0]	DO	不变	DI	不变	DO	不变

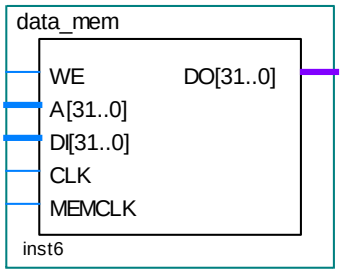


图 8-8 数据存储器的电路符号

8.6 单周期 CPU 设计

在本节中通过调用前面编写的一系列元器件来实现单周期 CPU，它能处理 MIPS 中本书列出的常用的 20 条指令。CPU 输入信号：CLOCK 为时钟信号，RESETN 置 0 信号，INSTR 为指令数据，MEM 为运算数据。CPU 输出信号：PC 是下一条要执行指令在指令存储器中的地址，WMEM 是数据存储器的读写信号，ALU 是数据读出或写入在数据存储器中的地址，DATA 是运算得到的要写入数据存储器的数据。单周期 CPU 的元件符号如图 8-9 示。

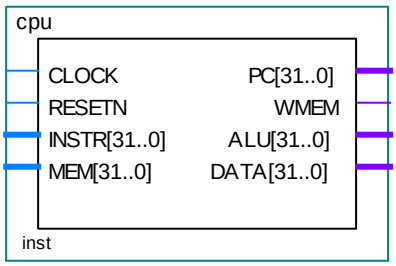


图 8-9 CPU 的电路符号

由控制器各控制信号的描述，以及 2.3 节 CPU 设计思路分析，可综合为完整的

CPU 逻辑设计，设计其 CPU 的原理图如图 8-10 所示。

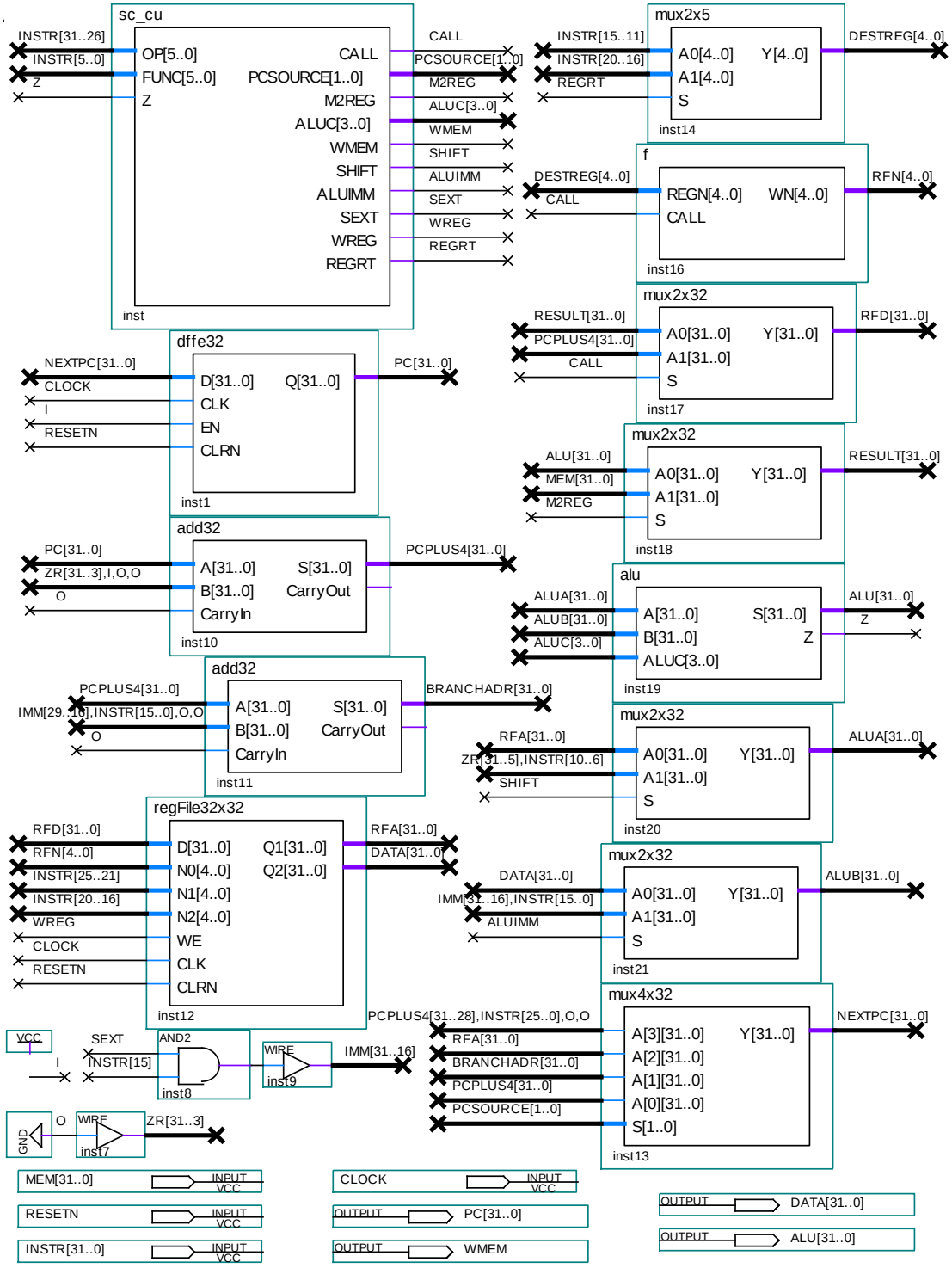


图 8-10 CPU 的原理图

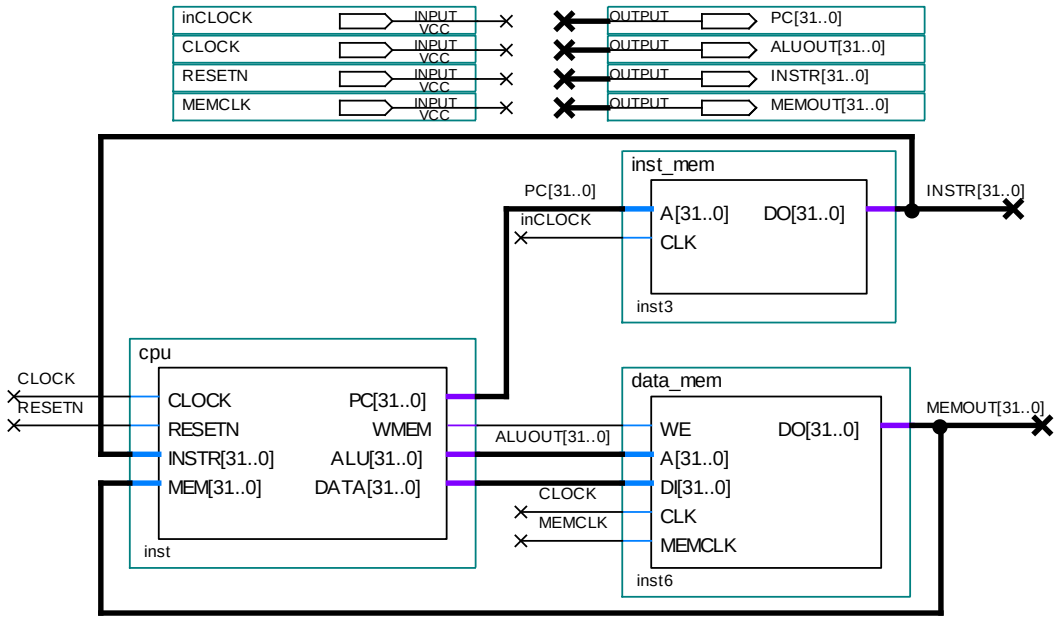


图 8-11 主机系统的原理图

8.7 计算机主机系统设计

主机系统的设计与实现是离不开存储器的，因为中央处理器 CPU 要从指令存储器中取得指令，分析解释后再从数据存储器中取得用于计算的数据，计算的结果可存放到数据存储器中。故本节将为 CPU 加上指令存储器和数据存储器，即主机系统。分析 CPU 的设计描述及各端口描述，不难得出如图 8-11 所示的主机系统的原理图。

8.8 加法程序调试仿真

(1)打开 cpu 项目。

(2)创建一个原理图文件 新建一个原理图文件，在原理图编辑窗口中按照图 8-11 创建主机系统的原理图，保存为“sc_comp.bdf”。

(3)创建一个 VHDL 文件 新建一个 VHDL File 文件，在 VHDL 程序编辑窗口中输入主机系统的 VHDL 程序，保存为“V_sc_comp.vhd”。

(4)编译 将要编译的文件设置为顶层文件，编译。

(5)创建波形文件 通过编译后，新建 Vector Waveform File 文件，将所有的信号节点导入波形图编辑窗口中。对所有信号节点前面有 或 图标的，按如图 8-12 所示进行值设置。然后，保存该波形文件为“sc_comp.vwf”和“V_sc_comp.vwf”。

(6)功能仿真 原理图和VHDL最后生成的功能仿真波形图应一样，如图8-12所示。最后仿真生成的数据存储文件要与图 8-12 所示一致，才表示仿真正确，且可按照如图 9-43 所示保存仿真生成的加法数据存储文件。

(7)波形分析 由如图 8-12，inCLOCK 在下降沿时，从指令存储器中取出指令。MEMCLK 在上升沿时，可对数据存储器进行读写操作。RESETN 为置 0 信号。CLOCK 为 CPU 的时钟信号，数据存储器的读写控制信号。所示 0.0ns 到 20.0ns 这一时段，经过 CPU 的运算得到 PC 指令存储器的输入地址为 00000000，ALUOUT 数据存储器的输入地址 00000000，则在指令存储器中取出的指令 INSTR 为 3C010000，读出的数据 MEMOUT 为 000000A3。图中正好也是这样。因此得证。最后仿真生成的数据存储文件要与图 8-13 所示一致，才表示仿真正确。

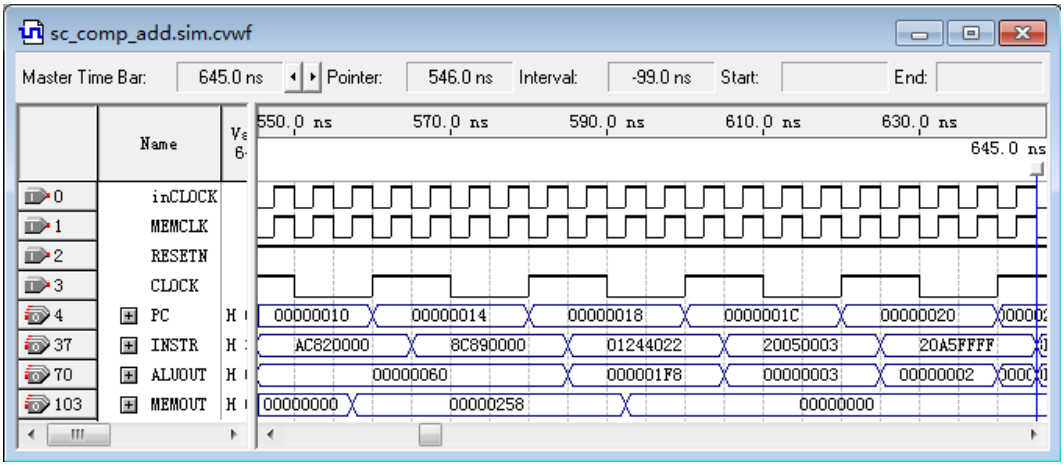
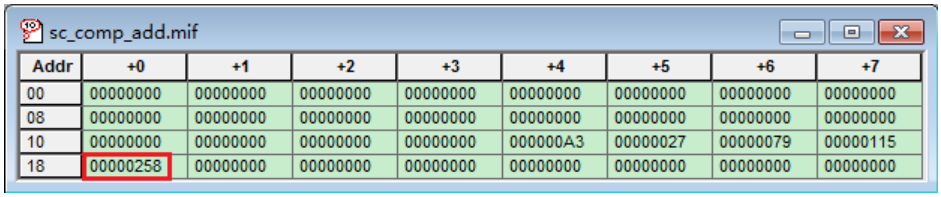


图 8-12 主机系统的加法仿真波形图



Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
08	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
10	00000000	00000000	00000000	00000000	000000A3	00000027	00000079	00000115
18	00000258	00000000	00000000	00000000	00000000	00000000	00000000	00000000

图 8-13 主机系统的加法仿真生成数据存储文件

8.9 乘法程序调试仿真

参照加法建立的指令存储文件和数据存储文件，在做乘法时，只需要将 lpm_rom_inst.vhdl 中的 (init_file => "inst_mem.mif",) 改为（init_file => "inst_mem_mul.mif",），保存即可。lpm_ram_dq_data.vhdl 中的 (init_file =>

"data_mem.mif",)改为（init_file => "data_mem_mul.mif",），保存即可。

(1)打开 cpu 项目。

(2)打开波形文件 “sc_comp.vwf”和“V_sc_comp.vwf”。

(3)重新功能仿真，原理图和 VHDL 最后生成的功能仿真波形图一样，如图 8-14 所示。

(4)波形分析 由如图 8-14，inCLOCK 在下降沿时，从指令存储器中取出指令。MEMCLK 在上升沿时，可对数据存储器进行读写操作。RESETN 为置 0 信号。CLOCK 为 CPU 的时钟信号，数据存储器的读写控制信号。所示 0.0ns 到 20.0ns 这一时段，经过 CPU 的运算得到 PC 指令存储器的输入地址为 00000000，ALUOUT 数据存储器的输入地址 00000000，则在指令存储器中取出的指令 INSTR 为 3C010000，读出的数据 MEMOUT 为 0000C9AE。图中正好也是这样。因此得证。最后仿真生成的数据存储文件要与图 8-15 所示一致，才表示仿真正确。且可按照如图 8-16 所示保存仿真生成的乘法数据存储文件。

注意仿真时间加长至 10μs，才能看到乘法运行的结果，因为乘法运行比加法运行复杂，运行所需时间也长些。

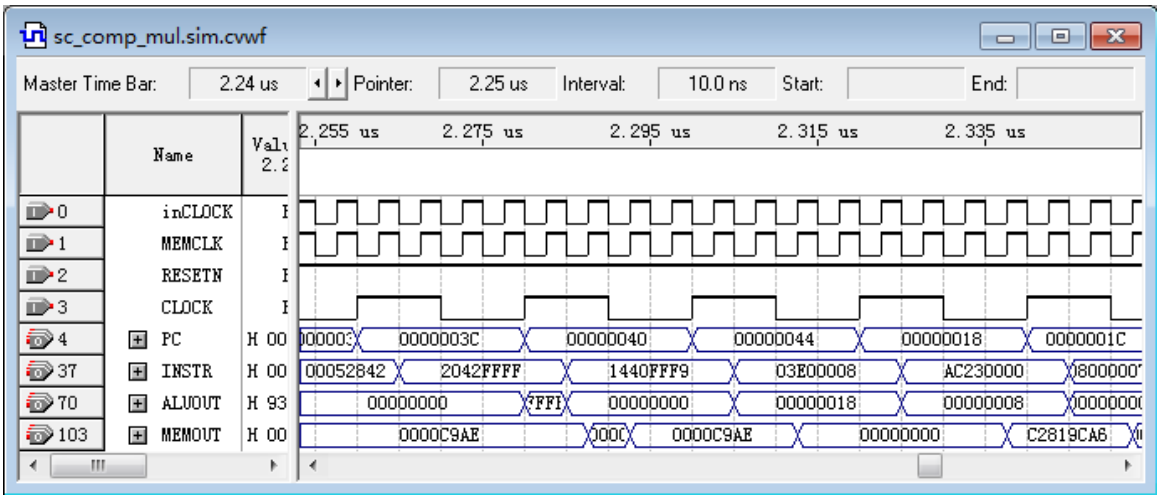


图 9-41 主机系统的乘法波形仿真图

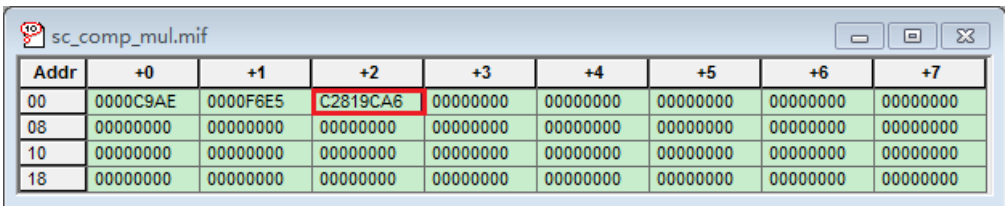


图 9-42 仿真生成的保存的数据文件

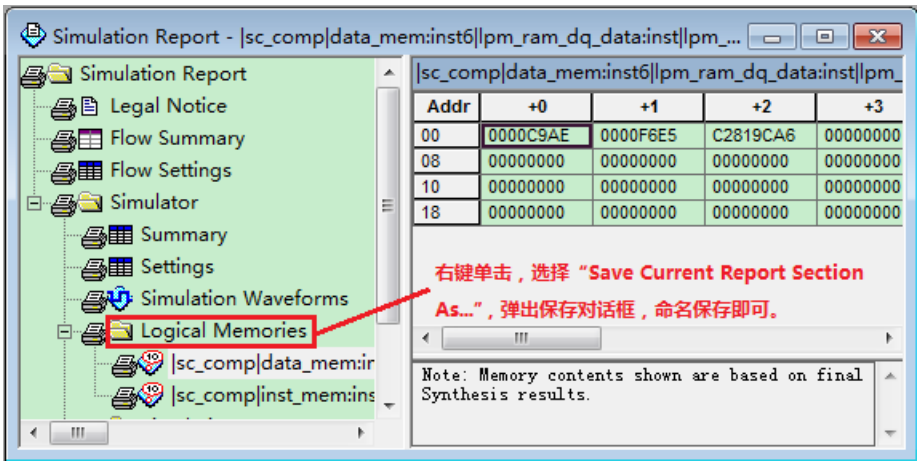


图 8-16 保存仿真生成的数据存储文件对话框

9 设计方案的改进意见

9.1 设计需求

新增几条 MIPS 指令，完成多个程序的仿真与下载，设计一个能运行的小型操作系统，执行简单指令。

9.2 改进方案

进行异或处理，改进加法和乘法程序，设计多种运行程序，进一步设计完成多周期中央处理器 CPU 设计。

10 总结与心得

首先，通过这次课程设计，我得到了很大的教训，知道软件初始设计对后来维护的影响之大。本次实验完成比较快，但是想要增加取消模块时，我撞得头破血流，因为本身的设计模式决定了这个功能不容易添加，除非大规模重新设计程序。因此一份完善的设计方案重要性不言而喻。

结合我原来设计方案与感悟，一份好的设计，要有完善的设计过程，从描述开始，再进行设计与实现，然后进行有效性验证，再进行进化（满足不同需求）。

有时候设计和实践相结合的，在尝试中修改，解决一个个 bug 后展现在我们眼前的才是完美的程序。问题是在研究中发现，并且在其间升华。

本次课程设计，培养了我们团队协作能力，每一项功能都和队友一起讨论，一起评估可行性，并将其细化，优势互补，最终得到可行的方案。大大减少了代码的编写时间，并且更不容易产生 bug。

提高了自己撰写工程报告的能力，不仅要做出来，而且还要清晰地表达出来，方便别人阅读和使用。程序是作品，而报告是说明书，没有说明书的作品，别人看不懂不会用，何谈优秀。

锻炼了自己利用 QuartusII 进行仿真的能力，锻炼了阅读仿真图的能力，以及将仿真结果对照源程序，分析仿真图，以及面对仿真图和预期不一致时进行分析；

观察了模块的内部图，对于用原理图实现实际电路部件有了一定的较为深刻的认识；

每次课程设计都会让自己感觉痛苦并快乐着，痛苦的是难与烦，快乐的是能够解决一个个问题，毕竟收获与付出成正比。

参考文献

[1] 朱子玉，李亚民，CPU 芯片逻辑设计技术，北京：清华大学出版社，2005

[2] 潘松，黄继业，EDA 技术实用教程—VHDL 版（第四版），北京：科学出版社，2010

[3] 任爱锋，初秀琴，基于 FPGA 的嵌入式系统设计，西安：西安电子科技大学出版社，2005

[1] 朱子玉，李亚民，CPU 芯片逻辑设计技术，北京：清华大学出版社，2005

[2] 潘松，黄继业，EDA 技术实用教程—VHDL 版（第四版），北京：科学出版社，2010

[3] 任爱锋，初秀琴，基于 FPGA 的嵌入式系统设计，西安：西安电子科技大学出版社，2005

[4] 21 嵌入式控制技术研究室.FPGA/SOPC 开发快速教程.www.21control.com

[5] 白中英，计算机组成原理（第五版），北京：科学出版社，2008

[6] 李亚民，计算机组成原理与设计—Verilog HDL 版，北京：清华大学出版社，2011

[7] Altera Corporation.Using DDR/DDR2 SDRAM With SOPC Builder. January 2006, ver1.0

[8] Altera Corporation. Cyclone II Device Handbook,Volume1 .

[9] Altera Corporation.Nios II Processor Reference Handbook, May 2009

[10] Altera Corporation.Quartus II Version 6.0 Handbook, May 2008

[12] Altera Corporation. Nios II Software Developer’s Handbook, May 2009

[13] Altera Corporation.Quartus II Installation & Licensing for PCs, May 2008