计算机组成 MidTerm

1. 概述

ENIAC——世界上第一台电子计算机；1946年2月14日，美国宾夕法尼亚大学；采用十进制；约翰莫克里，约翰埃克特

EDVAC——存储程序计算机，冯诺依曼结构；1949年交付；五部分组成，采用二进制，实现存储程序的概念；约翰冯诺依曼

UNIVAC——埃克特-谟克利计算机公司（EMCC）；第一台交付美国人口普查局；面向商业和行政管理用户，量产计算机，开启商用计算机时代

计算机主要类别：

大型机（IBM701），面向大容量数据处理为主，高速输入输出，储存空间大，并行事务处理强，兼顾科学计算；

超级计算机（CDC6600，超级计算机之父，西摩克雷），面向科学计算，运算速度最快，性能最高；

小型计算机（PDP-8，小型机之父，肯·奥尔森，DEC公司），介于大型机和微机之间，Minicoputer逐渐被Server取代；

微型计算机（Altair 8800，爱德华·罗伯茨，MITS），个人或家庭使用，小型轻便

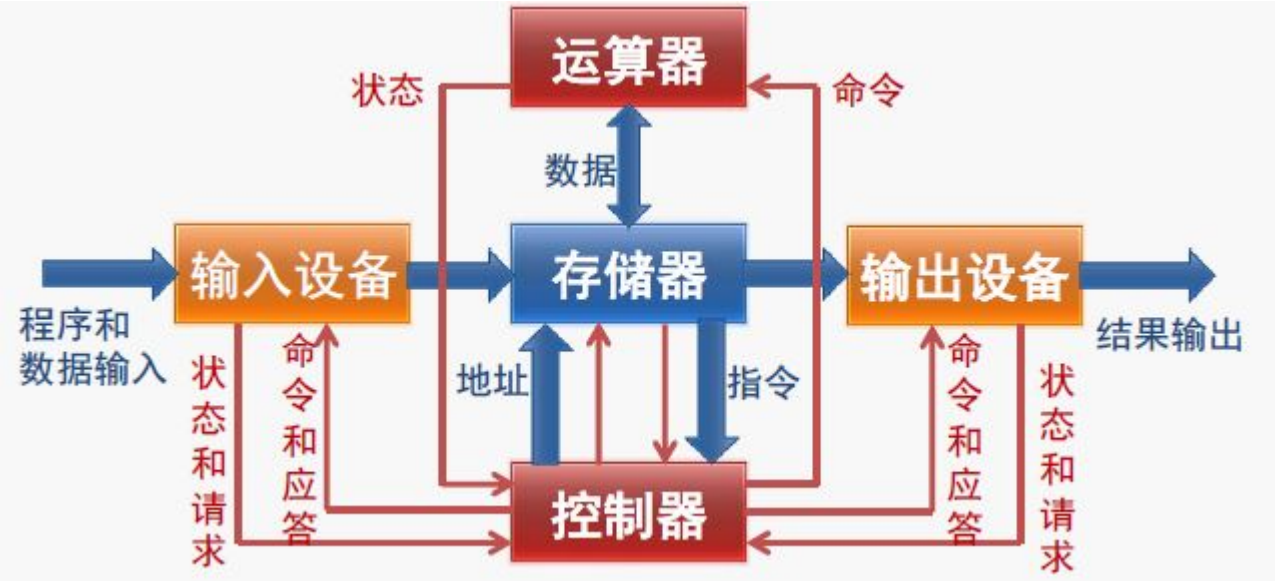
Apple I型，1975年，史蒂夫·乔布斯，史蒂夫·沃兹尼亚克，木质外壳，手工制作

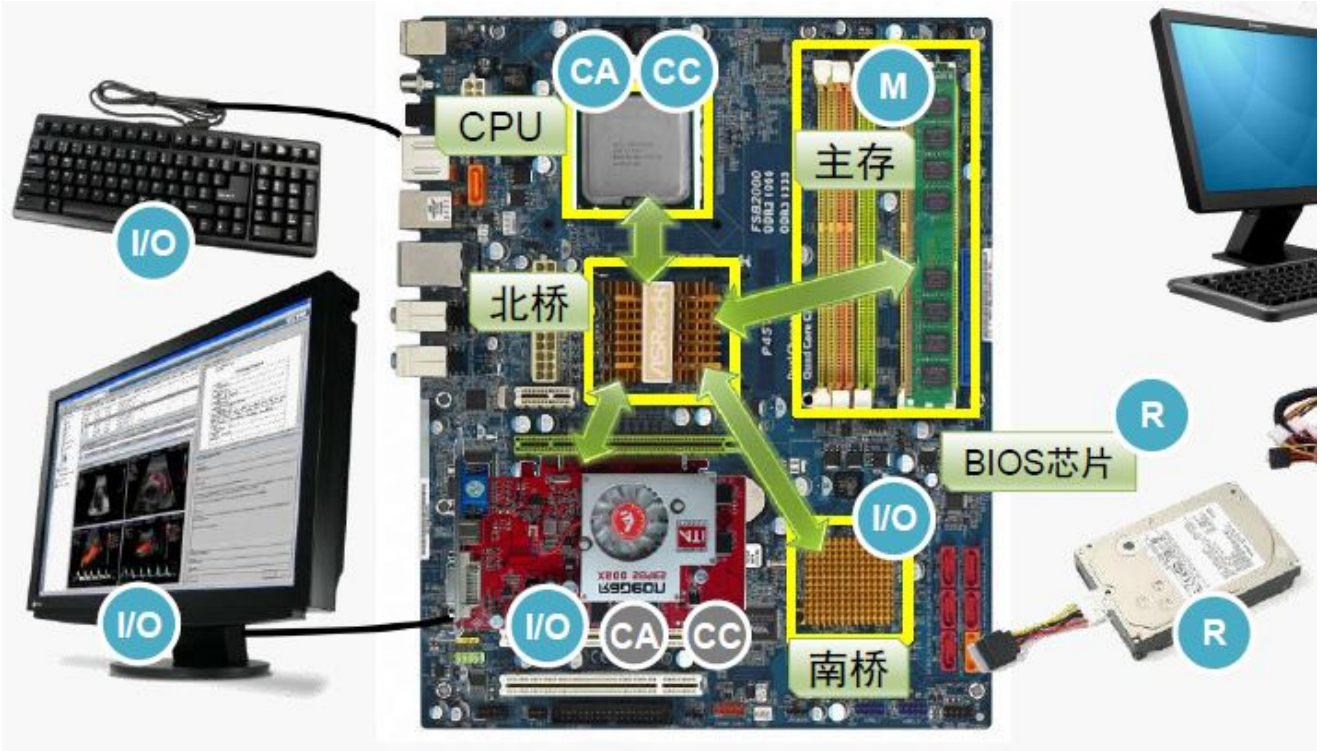
Apple II型，1977年，廉价轻便，1980-1990年代美国教育系统电脑的实际标准

IBM PC 5150，1981年，开放性构架，成为个人计算机的制造标准

2. 计算机的基本结构

冯诺依曼结构：CA运算器 + CC控制器 + M储存器 + I输入设备 + O输出设备 （+ R外部记录设备）





ROM：只读(Read Only)，非易失性(Non-Volatile)；用于存放不需要经常修改的程序/数据，比如BIOS。

Mask ROM——掩膜式ROM，厂家制造时一次性写入，之后不可修改；

PROM——可编程ROM，储存内容由用户写入，但只能写入一次不可修改；

EPROM——可擦除可编程ROM，可以多次编程，比较灵活，但修改时需要将芯片拔下使用紫外线擦除之后重写，主要用于早期的微控制器；

EEPROM——电可擦除可编程ROM，EPROM，指示不需要紫外线，而是用高电压进行擦除，并且支持按字节擦除，但擦除和写入的编程电压高于工作电压；

Flash Memory——快速闪存存储器，在线擦除和编程，正常工作电压下可实现编程。

Intel汇编格式：Intel制定，x86相关的文档手册使用，主要用于MS-DOS和Windows系统。

寄存器和立即数没有前缀，十六进制使用后缀h，二进制使用后缀b；

第一个数是目的操作数，第二个数是源操作数；

基寄存器使用"[]"标明；

间接寻址方式为$segreg:[base+index*scale+disp]$；

内存单元操作数带前缀（dword ptr；word ptr；byte ptr）以指出操作数的大小

AT&T汇编格式：AT&T制定，起源于贝尔实验室研发的Unix，主要用于Unix和Linex等系统。

寄存器使用前缀%，立即数使用前缀$，十六进制立即数使用前缀0x；

第一个数是源操作数，第二个数是目的操作数；

基寄存器用"()"标明；

间接寻址方式为$\%segreg:disp(base,index,scale)$；

操作码带后缀（l，32位长整数；w，16位字；b，8位字节）以指出操作数的大小

汇编语言程序：分段最多四段（代码，数据，堆栈，附加），每一段以“Name SEGMENT”开始，以“Name ENDS”结束；每段中有若干语句行（指令/伪指令/宏指令）组成。

```
DATA SEGMENT
        ...;    # 数据段语句
DATA ENDS

STACK SEGMENT
        ...;    # 堆栈段语句
STACK ENDS

CODE SEGMENT
        ...;    # 代码段语句
CODE ENDS
```

指令语句（可执行语句）：[标号:] 指令操作助记符 [操作数表达式1 [, 操作数表达式2]][; 注释]　汇编器计算指令语句中表达式的值，产生机器代码，CPU按照机器代码的要求完成各种运算或操作

```
L: ADD  AD,     BX ;    这是一条指令语句
```

伪指令语句（说明性语句）：[名字] 伪指令指示符 [操作数表达式1 [, 操作数表达式2 [, ...]]][; 注释]
汇编器计算伪指令语句中表达式的值，不产生机器代码，汇编器会“解释”伪指令语句的含义并遵照“执行”

```
A       DB       20H,   30H ;   声明了字节变量的伪指令语句
```

数据定义：通常定义在DS和ES中，变量长度由说明符DB\DW\DD定义

符号定义：定义常量，用EQU和=定义，EQU左边符号名不可重复定义，=左边的符号名可以重复定义

段定义：SEGMENT/ENDS定义段开始和结束，ASSUME指定段寄存器

```
ASSUME CS:CODE, DS:DATA
        BEGIN:
        ...
        END BEGIN;      模块结束
```

指定段内偏移地址：ORG制定当前可用的存储单元的便宜地址为常数表达式的值；EVEN将当前可用的储存单元的便宜地址调整为最近的偶数值

过程定义：类似函数定义，过程名 PROC 类型属性名 / 过程名 ENDP；类型属性名可选择NEAR（近过程）和FAR（远过程），默认NEAR

宏定义：源程序中的一端有独立功能的程序代码，只定义一次，可以多次调用。宏定义被调用时，用宏定义体取代源程序中的宏指令名，用实在参数取代宏定义中的形式参数，称为宏展开。

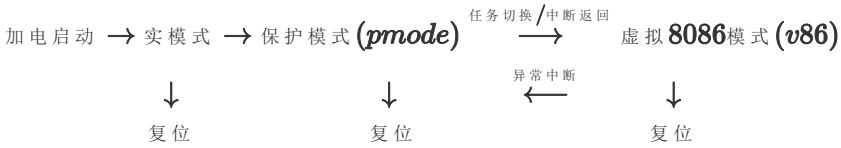
```
宏指令名 MACRO [ 形式参数，形式参数，... ]
                ...;    “宏定义体”
                ENDM
```

3. CISC和x86指令

Intel 8086——1978年，16位通用寄存器，16位数据线，20位地址线，对外可寻址内存空间2^{20}（1MB），物理地址采用“段+偏移”的方式形成。

Intel 80286——1982年，24位地址总线，对外可寻址内存空间2^{24}（16MB），具有“保护模式”和8086兼容的“实模式”。

Intel 80386——1985年，x86系列第一款32位CPU，地址总线32位，对外可寻址内存空间2^{32}（4G），改进了保护模式，增加了虚拟8086模式，可以同时模拟多个8086。

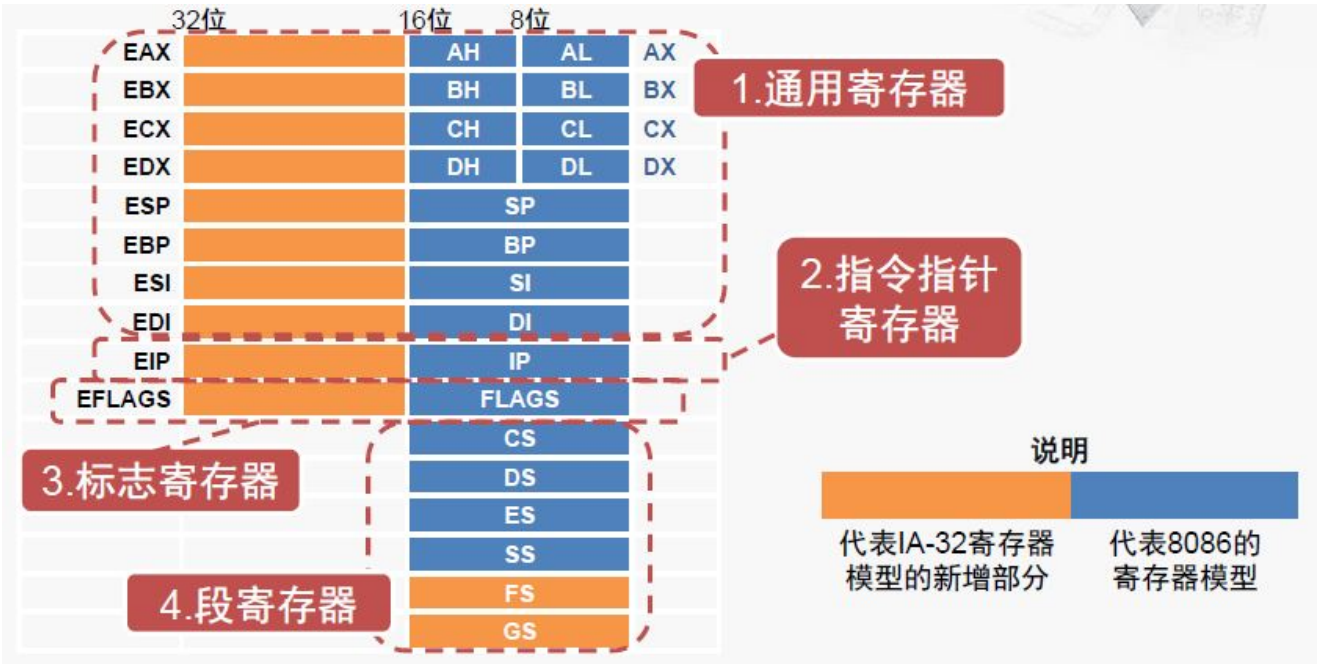


pmode：80386及以上CPU的主要工作模式，支持多任务、检查访问权限、设置执行特权级，可以访问4GB物理存储空间，引入续存概念。

V86模式：保护模式下的特殊工作状态，模拟8086，对于中断/异常会有相应处理。

AMD Opteron——2003，x86扩展到64位的第一款CPU，兼容32位x86且不降低性能。

IA-32和8086的寄存器模型



指令指针寄存器IP——保存一个内存地址指向当前需要取出的指令，每取出一个指令，IP自动增加指向下一指令，转移指令/过程调用/返回指令会改变IP。IP只有16位寻址能力，单独使用IP寻址不能满足8086的20位对外寻址要求，需要用段加偏移的方式。

段寄存器——代码段寄存器CS，数据段寄存器DS，附加段寄存器ES，堆栈段寄存器SS，附加段寄存器FS和GS（80386起新加，功能和ES相同）。

8086的物理地址生成：$\text{物理地址}(20bit) = \text{段基址}(16bit, \text{来自段寄存器}) \times 16 + \text{偏移值}(16bit)$。过程需要移位器（乘16等同于左移4位）和地址加法器参与。

段跨越前缀：如果数据存放在数据段以外的其他段，比如ES，则应在指令中给出

```
MOV     AX,     ES: [3000H]
ES: MOV AX,     [3000H]
```

IA32保护模式IP寻址：GDTR指向全局描述符表（GDT）初始地址，通过段寄存器在GDT中找到某个描述符，取出其中的基地址（32bit），检查权限等，之后用该基地址直接加上偏移量（32bit）即得到物理地址（32bit）。一个描述符如下。

字节7	字节6	字节5	字节4	字节3	字节2	字节1	字节0
基地址	其他	权限	基地址	基地址	基地址	段界限	段界限

x86-64的描述符中没有了段基址和段界限，只有访问权限字节和若干控制位（字节5、6），其他都是0，即所有代码段都从地址0开始。

x86传送指令：把数据或地址传送到寄存器或存储器单元中

分组	助记符	功能	操作数类型
通用数据传送指令	MOV	传送	字节/字
	PUSH	压栈	字
	POP	弹栈	字
	XCHG	交换	字节/字
累加器专用传送指令	XLAT	换码	字节
	IN	输入	字节/字
	OUT	输出	字节/字
地址传送指令	LEA	装入有效地址	字
	LDS	把指针装入寄存器和DS	4个字节
	LES	把指针装入寄存器和ES	4个字节
标志传送指令	LAHF	把标志装入AH	字节
	SAHF	把AH送标志寄存器	字节
	PUSHF	标志压栈	字
	POPF	标志弹栈	字

MOV DST, SRC——操作数默认1字节，通过长度标记更换（BYTE PTR/ WORD PTR/ DWORD PTR）；立即数不能作为目的操作数，立即数不能直接传送到段寄存器，存储单元之间不能直接传送，段寄存器之间不能直接传送，CS不能作为目的寄存器。

XCHG OPR1, OPR2——交换两个操作数，要求位宽相同，类型相同，不允许存储单元之间交换，不允许使用段寄存器。

XLAT——从BX中取得数据表起始地址的偏移量，从AL中取得数据表索引值，在数据表中查得表项内容（0-index），并将表项内容存入AL。

x86运算指令

分组	助记符	功能	操作数类型
加法	ADD	加	字节/字
	ADC	加（带进位）	字节/字
	INC	加1	字节/字
减法	SUB	减	字节/字
	SBB	减（带借位）	字节/字
	DEC	减1	字节/字
	NEG	取补	字节/字
	CMP	比较	字节/字
乘法	MUL	乘（不带符号）	字节/字
	IMUL	乘（带符号）	字节/字
除法	DIV	除（不带符号）	字节/字
	IDIV	除（带符号）	字节/字

分组	助记符	功能	操作数类型
符号扩展	CBW	将字节扩展为字	/
	CWD	将字扩展为双字	/
十进制调整	AAA	加法的ASCII调整	/
	DAA	加法的十进制调整	/
	AAS	减法的ASCII调整	/
	DAS	减法的十进制调整	/
	AAM	乘法的ASCII调整	/
	AAD	除法的ASCII调整	/

ADD DST, SRC——DST←DST+SRC

ADC DST, SRC——DST←DST+SRC+CF

INC OPR——OPR←OPR+1

DAA——加法十进制调整指令，跟在在二进制加法之后，将AL中的“和”数调整为压缩BCD数格式，结果送回AL。

BCD(Binary-Coded Decimal)使用二进制编码，但看起来具有十进制的形式。9502（十进制）→25H/1EH（十六进制，2byte）→95H/02H（压缩BCD）或09H/05H/00H/02H（非压缩BCD）

x86转移指令：改变指令执行的顺序（无条件转移+条件转移 / 直接转移+间接转移）

分组	格式	功能	测试条件
无条件转移指令	JMP LABEL	无条件转移	
	CALL LABEL	过程调用	
	RET	过程返回	

分组		格式		功能	测试条件
条件转移指令	根据某一状态标志转移	JC	LABEL	有进位时转移	CF=1
		JNC	LABEL	无进位时转移	CF=0
		JP/JPE	LABEL	奇偶位为1时转移	PF=1
		JNP/JPO	LABEL	奇偶位为0时转移	PF=0
		JZ/JE	LABEL	为零/相等时转移	ZF=1
		JNZ/JNE	LABEL	不为零/不相等时转移	ZF=0
		JS	LABEL	负数时转移	SF=1
		JNS	LABEL	正数时转移	SF=0
		JO	LABEL	溢出时转移	OF=1
		JNO	LABEL	无溢出时转移	OF=0

分组		格式		功能	测试条件
条件转移指令	对无符号数	JB/JNAE	LABEL	低于/不高于等于时转移	CF=1
		JNB/JAE	LABEL	不低于/高于等于时转移	CF=0
		JA/JNBE	LABEL	高于/不低于等于时转移	CF=0且ZF=0
		JNA/JBE	LABEL	不高于/低于等于时转移	CF=1或ZF=1
	对有符号数	JL/JNGE	LABEL	小于/不大于等于时转移	SF≠OF
		JNL/JGE	LABEL	不小于/大于等于时转移	SF=OF
		JG/JNLE	LABEL	大于/不小于等于时转移	ZF=0且SF=OF
		JNG/JLE	LABEL	不大于/小于等于时转移	ZF=1或SF≠OF

分组	格式		功能	测试条件
循环控制指令	LOOP	LABEL	循环	CX≠0
	LOOPZ/LOOPE	LABEL	为零/相等时循环	CX≠0且ZF=1
	LOOPNZ/LOOPNE	LABEL	不为零/不相等时循环	CX≠0且ZF=0
	JCXZ	LABEL	CX值为零时循环	CX=0

JMP SHORT LABEL——短转移IP←IP+8bit位移量

JMP NEAR PTR LABEL——近转移IP←IP+16bit位移量

JMP FAR PTR LABEL——远转移IP←LABEL的偏移地址 CS←LABEL的段基址

段内直接转移：JMP SHORT/NEWAR PTR LABEL

段间直接转移：JMP FAR PTR LABEL

JMP DWORD PTR OPR——寻址到OPR指定的存储单元的双字，将该双字中的低字送到IP，高字送到CS

段间间接转移：JMP DWORD PTR OPR

8086中所有条件转移都是短转移（-128到127字节范围内）

LOOPNE/ LOOPNZ LABEL——不为0/不相等时循环，CX←CX-1；若$CS \neq 0$且ZF=0，则转移至LABEL，否则结束循环，顺序执行下一条

x86控制指令：控制处理器状态

分组	格式	功能
标志操作指令	STC	把进位标志CF置1
	CLC	把进位标志CF清0
	CMC	把进位标志CF取反
	STD	把方向标志DF置1
	CLD	把方向标志DF清0
	STI	把中断标志IF置1
	CLI	把中断标志IF清0
外同步指令	HLT	暂停
	WAIT	等待
	ESC	交权
	LOCK	封锁总线（指令前缀）
空操作	NOP	空操作

4. RISC和MIPS指令

约翰·亨尼西领导RISC研究，创立MIPS计算机系统公司。

MIPS诞生即为32位，1992年扩展至64位。

MIP的基本原则为"A simpler CPU is a faster CPU"，主要关注点为减少指令类型和降低指令复杂度。32个通用寄存器，每个都是32位宽。

MIPS指令：固定指令长度32位（1字）；只有LOAD和STORE指令可以访存；寻址方式简单；指令数量少，功能简单。

R型指令：(高位) opcode(6bit)+rs(5bit)+rt(5bit)+rd(5bit)+shamt(5bit)+funct(6bit) (低位) 所有R型指令opcode均为0；funct域指定了R型指令的类型；rs为第一个源操作数，rt为第二个源操作数，rd为目的操作数，5bit恰好足够表示所有32个寄存器；shamt用于指定移位指令进行移位操作的位数，5bit恰好32位，对于非移位指令，该域设为0。

add rd, rs, rt——R[rd]=R[rs]+R[rt]

sll rd, rs, shamt——R[rd]=R[rs]<<shamt

I型指令：(高位) opcode(6bit)+rs(5bit)+rt(5bit)+immediate(16bit) (低位) 没有funct域，用opcode指定操作类型；rs为（第一个）源操作数，rt为目的操作数（某些指令下为第二个源操作数）；immediate为16bit立即数，访存指令(lw rt, imm(rs))通常可以满足地址偏移需求，运算指令大多数情况下可以满足需求。

addi rt, rs, imme——R[rt]=R[rs]+imme

lw rt, imme(rs)——R[rt]=Mem[R[rs]+imme]

slti rt, rs, imme——R[rt]=(R[rs]<imme) ? 1 : 0

分支指令：改变控制流

条件分支指令（I型）：beq rs, rt, imm / bne rs, rt, imm

计算目标地址的方法：next instruction = PC + 4

若分支条件不成立，PC = next instruction

若分支条件成立，PC = next instruction + (immediate * 4)

非条件分支指令（J型）：j addr (高位) opcode(6bit)+address(26bit) (低位)

计算目标地址的方法：New PC = { (PC+4)[31,30,29,28], address, 0, 0 }

非条件分支指令（R型）：jr rs　直接可以跨越32bit

MIPS汇编器提供的伪指令

寄存器传送：move dst, src——addi dst, src, 0

装载地址：la dst, label——add any_reg, $0, $0　　lw dst, label(any_reg)

装载32位立即数：li dst, imm——addi dst, dst, imm高16位　sll dst, dst, 16　　addi dst, dst, imm低16位

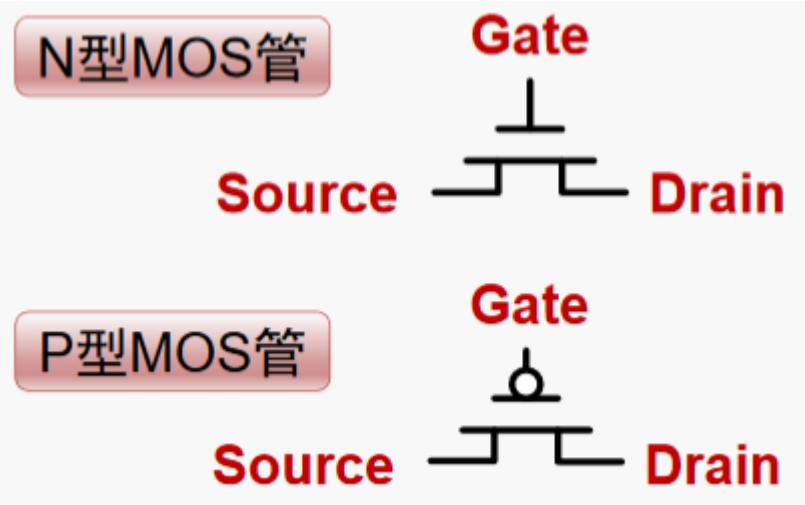
5. 数字电路设计

莱布尼兹 & 二进制

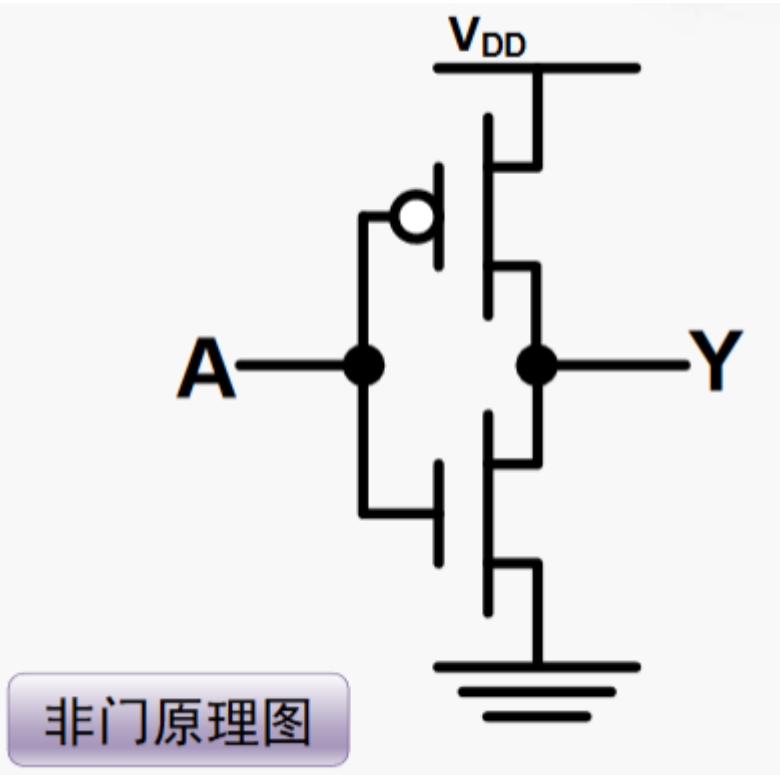
布尔 & 布尔代数

香农 & 开关电路

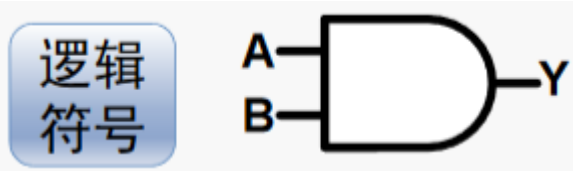
MOS管：NMOS栅极高电平时导通，PMOS栅极低电平时导通。

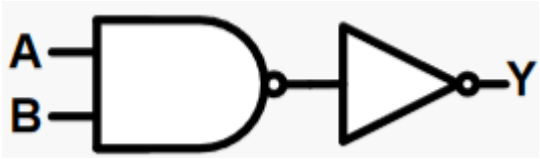


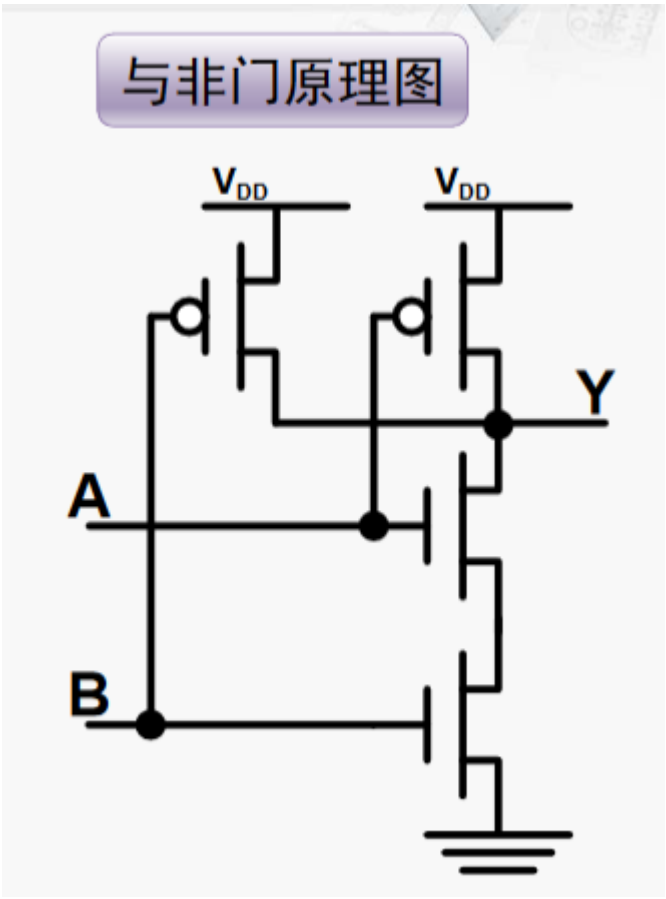
非门：$Y = \bar{A}$



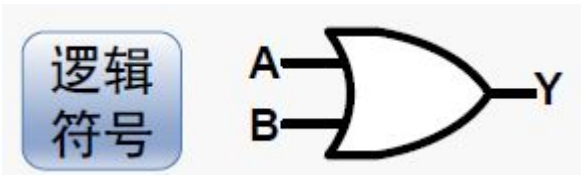
与门：实际上用与非门连接一个非门实现；$Y = A \cdot B$

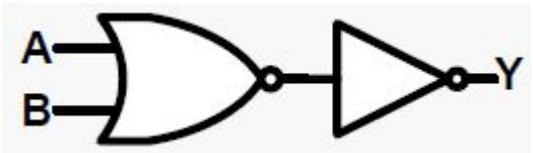


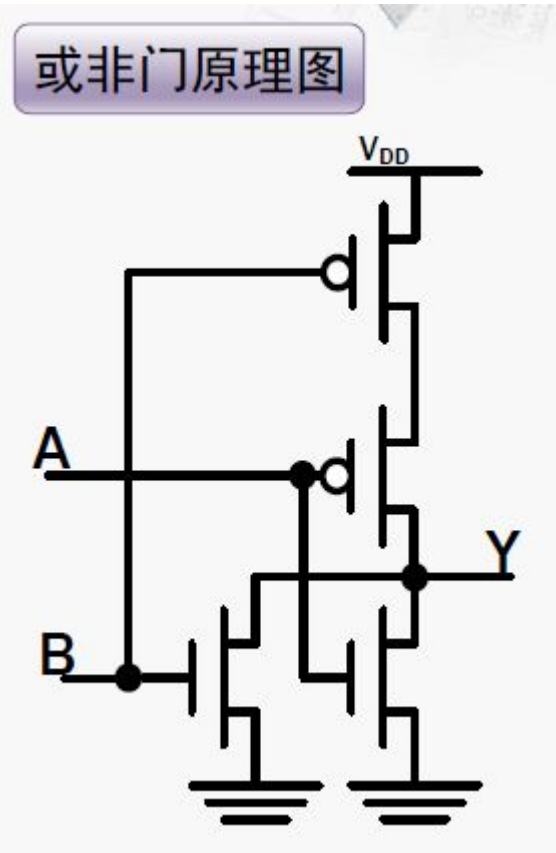




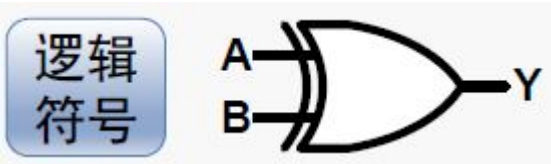
或门：实际上用或非门连接一个非门实现；$Y = A + B$



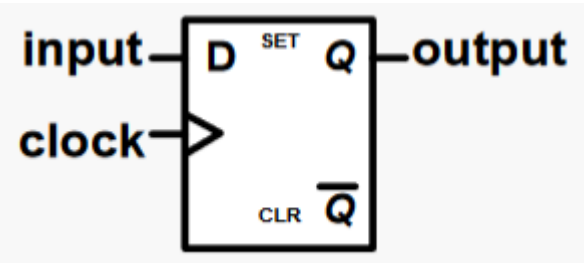




异或门：$Y = A \oplus B = (\bar{A} \cdot B) + (A \cdot \bar{B})$ 或 $Y = A \hat{} B$；用符号XOR表示

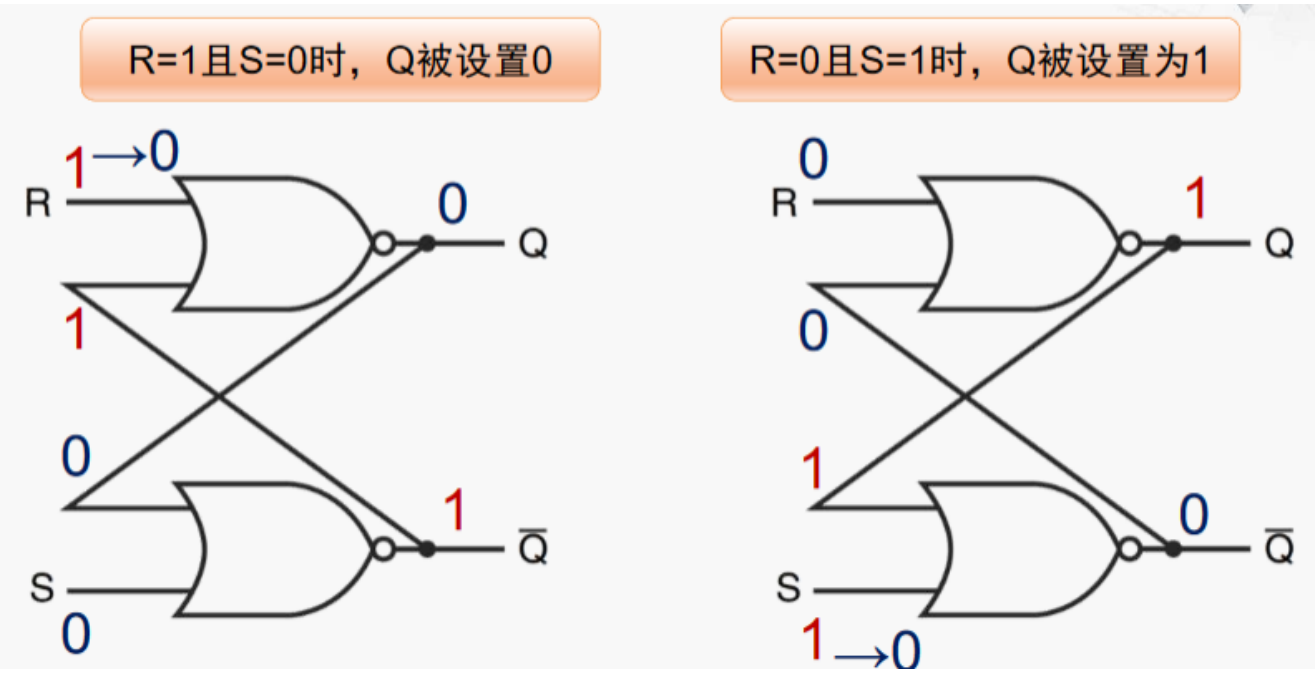


D触发器（D Flip-Flop，DFF）：具有储存信息能力的基本单元，有一个数据输入Input，一个数据输出Ouput(Q/$\bar{Q}$)，一个时钟输入Clock；在时钟上升沿（0→1）时，采样输入D的值，传送到输出Q，其余时间输出Q的值不变。

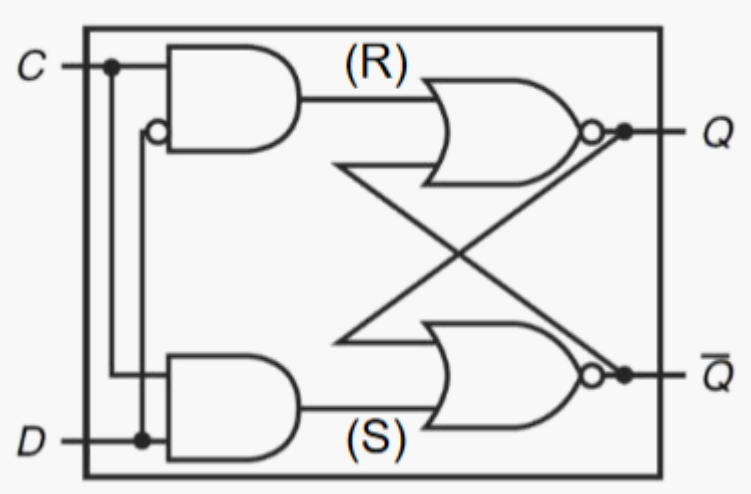


利用D触发器便可以制作出通用寄存器、移位寄存器等。

具有存储功能的电路：具有一个0-1数据输入S，一个重置输入R（为0不重置，为1重置），一个输出Q/$\bar{Q}$。当R=1且S=0时，Q被设置为0；当R=0且S=1时，Q倍设置为1。其实这个电路是对称的，并没有绝对的S,R和Q,$\bar{Q}$之说

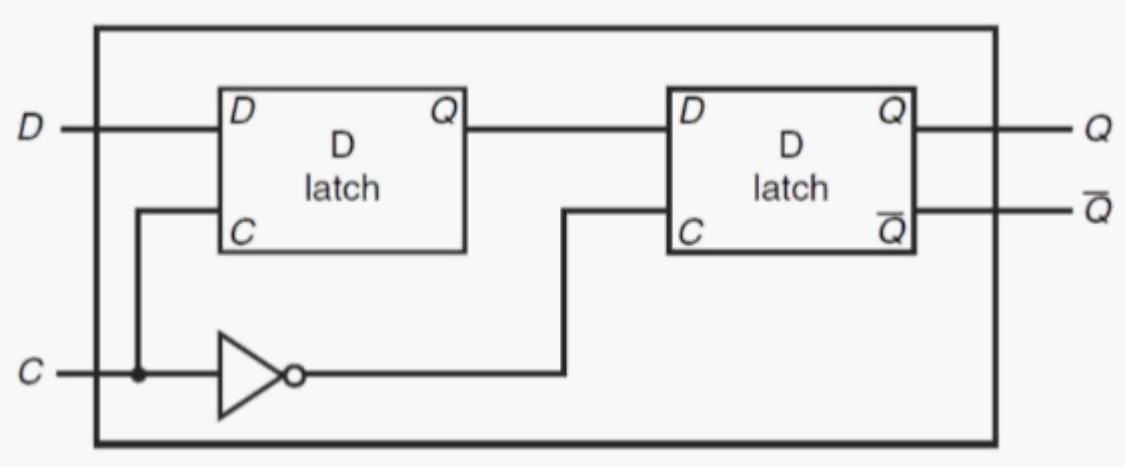


D锁存器（D latch）：在时钟C为高电平时锁存数据D，当C为低电平时输出Q不随D的变化而变化。

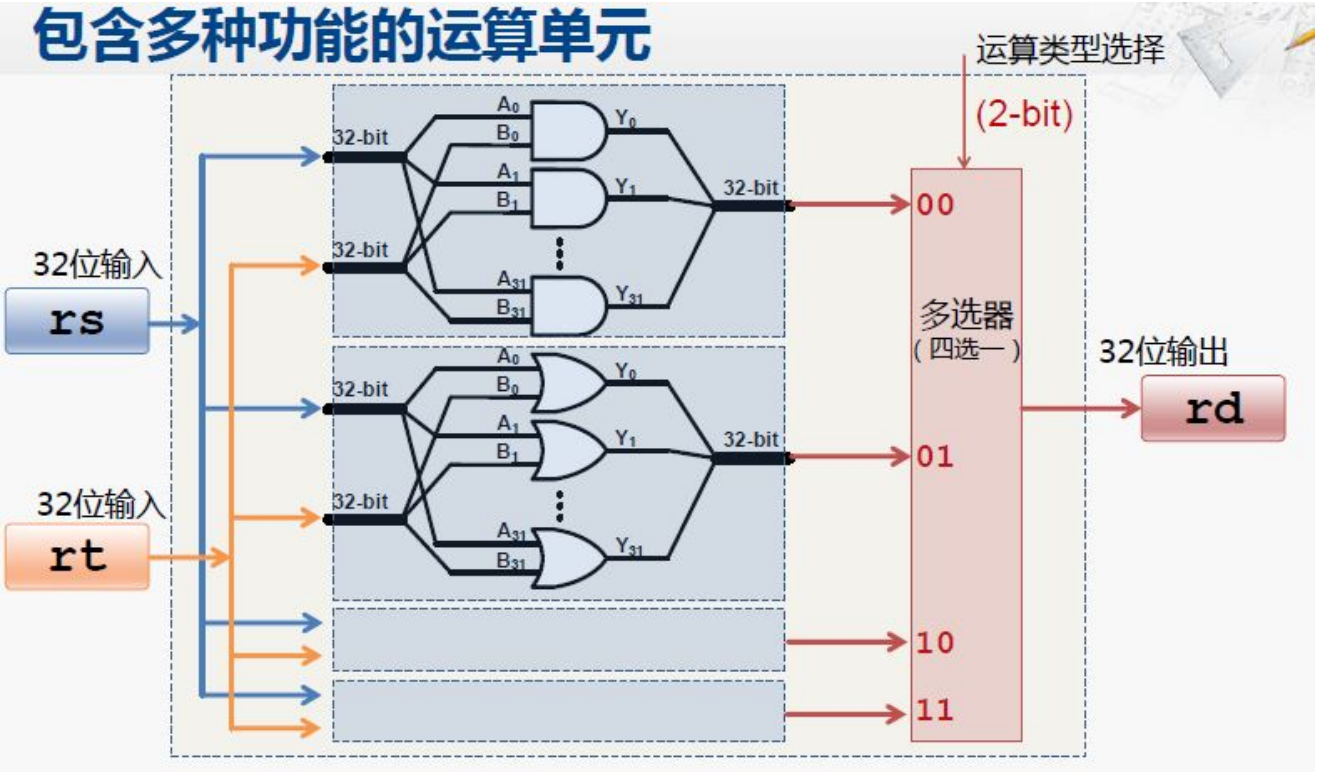


C	D	(R)	(S)	Q	$\bar{Q}$
0	0	0	0	没有变化	没有变化
0	1	0	0	没有变化	没有变化
1	0	1	0	0	1
1	1	0	1	1	0

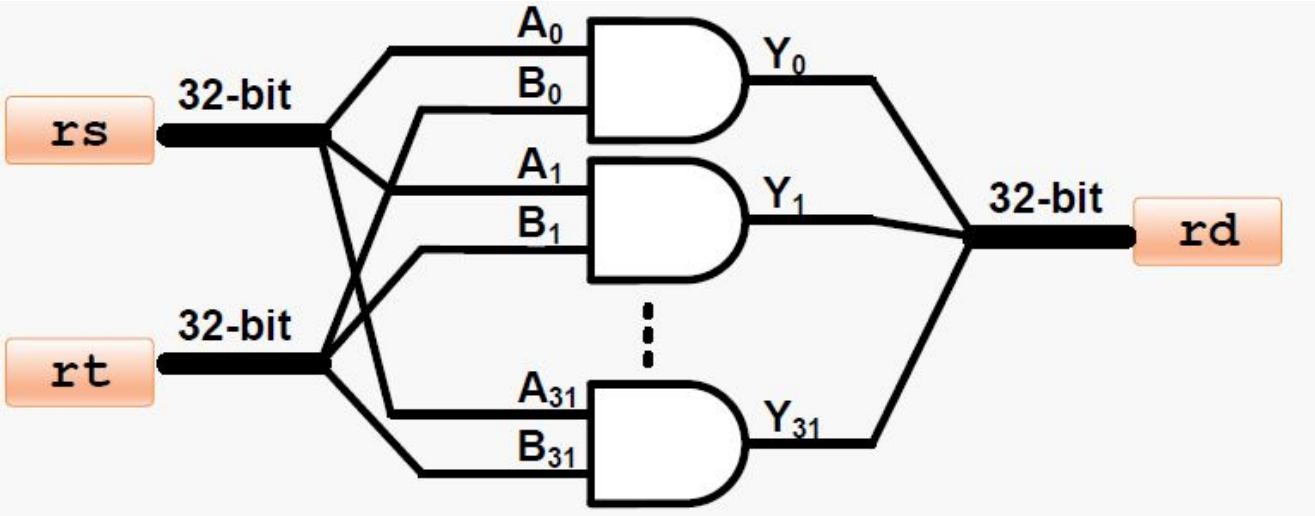
下降沿锁存DFF电路：用两个D latch组合，实现在时钟下降沿时锁存数据D的功能；为了使之在时钟上升沿所存数据D，在时钟上加一个非门即可。

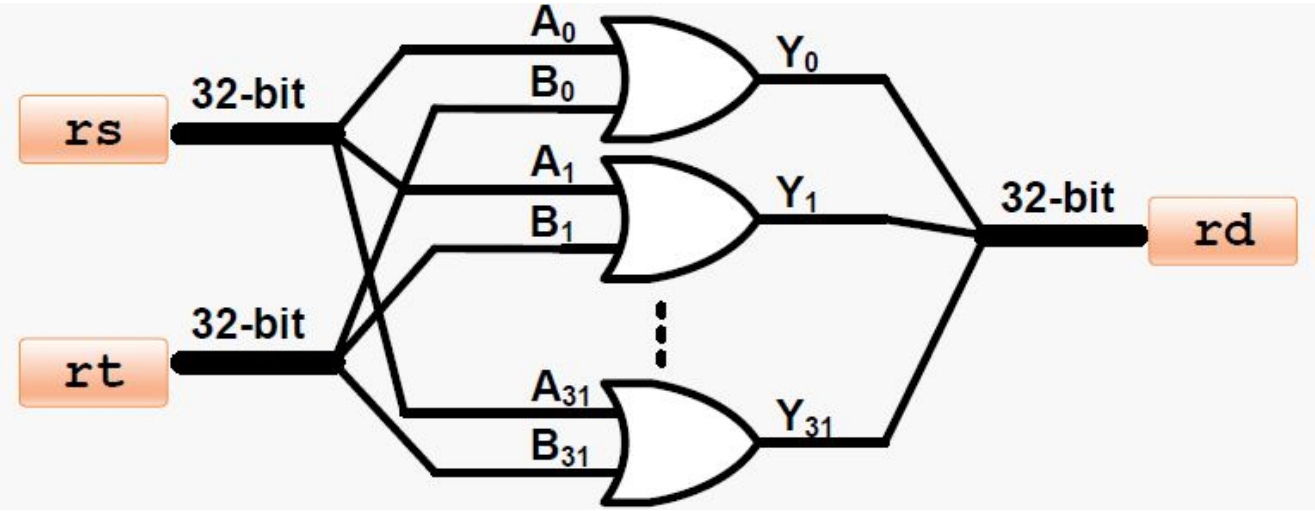


ALU实现的基本想法：输入两个32位数和一个运算类型的编码，经过ALU运算，输出一个32位的结果，因此ALU需要两个32-bit的数据输入接口，针对不同的运算设计不同的运算器，分别对这两个32位的数产生一个结果，之后通过一个多选器选择所需的某个运算的结果输出（多选器的输入为各种运算的运算结果，和运算类型的编码）。



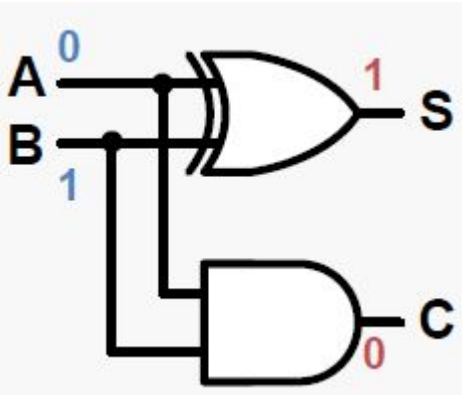
与/或运算的实现：对输入的数的各位分别通过一个与/或门进行运算，将各个门的结果输出作为位运算的结果即可。



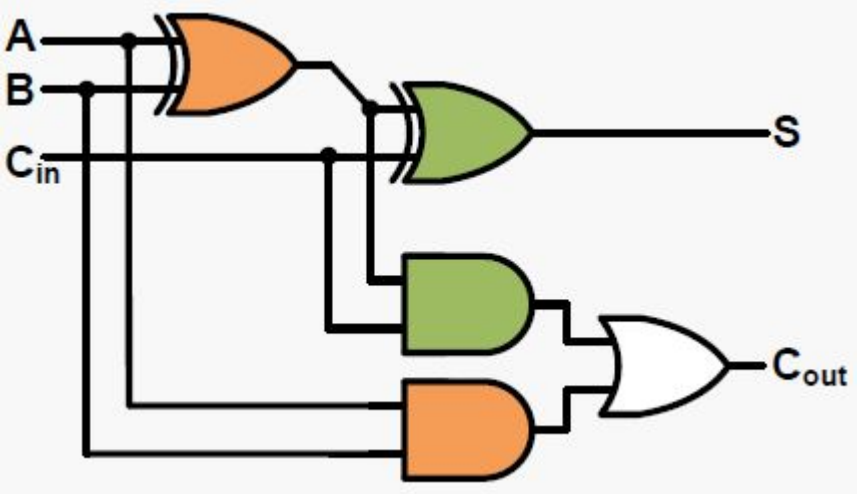


行波加法器（RCA）的实现：采用全加器作为基本单元，每一位一个全加器计算这一位加法的结果；全加器有三个输入A、B、C_{in}（输入进位），两个输出S和C_{out}（输出进位）。

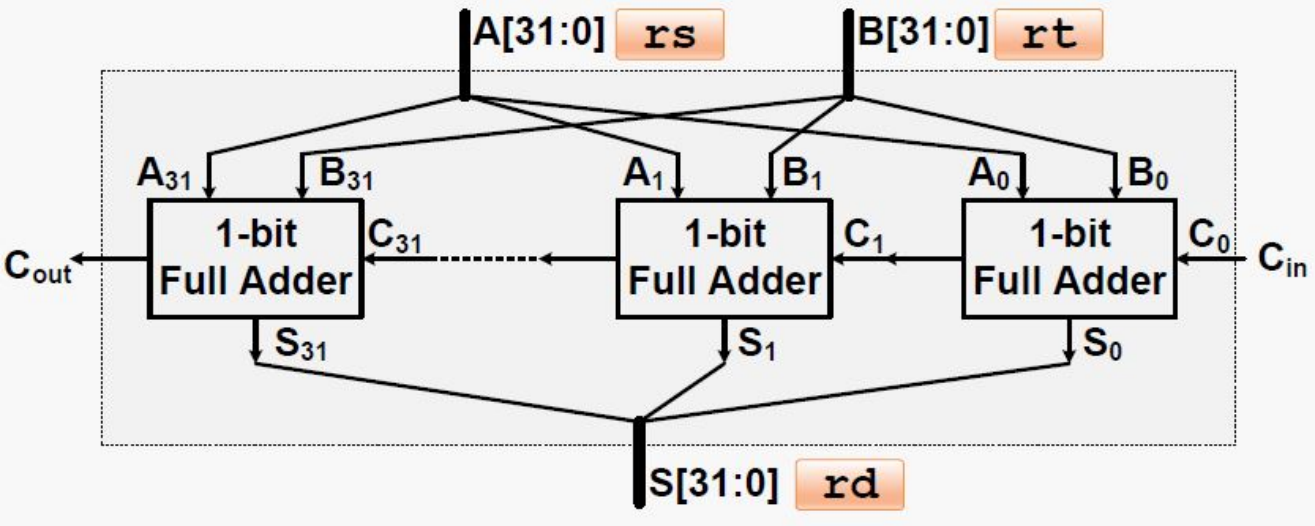
半加器(Half Adder)：将两个一位的二进制数A和B相加，输出S（和）和C（进位）



全加器(Full Adder)



32位行波加法器：从第0位到第31位依次计算下去。



溢出：运算结果超出了正常的表示范围，仅针对有符号数的运算（整数相加得到负数；复数相加得到正数）；检查方法是“最高位的进位输入不等于最高位的进位输出”则发生溢出（用异或门检查C_{out}和C_{31}即可）

MIPS提供两类加法指令：add和addi将操作数视为有符号数，发生溢出时会产生异常；addu和addiu将操作数视为无符号数，不会处理溢出

x86通过标志位OF处理溢出：11号标志位为溢出标志位OF，如果发生溢出，则将OF设置为1，否则OF=0；无视操作数类型，都认为是有符号数，自动设置。

减法器的实现：A-B=A+(~B+1)，如果是减法，则将B取反（非门）之后，进行加法，在加法中将C_0设置为1（可以将减法控制信号连接在C_0上，自动完成这一设置），完成减法的工作。

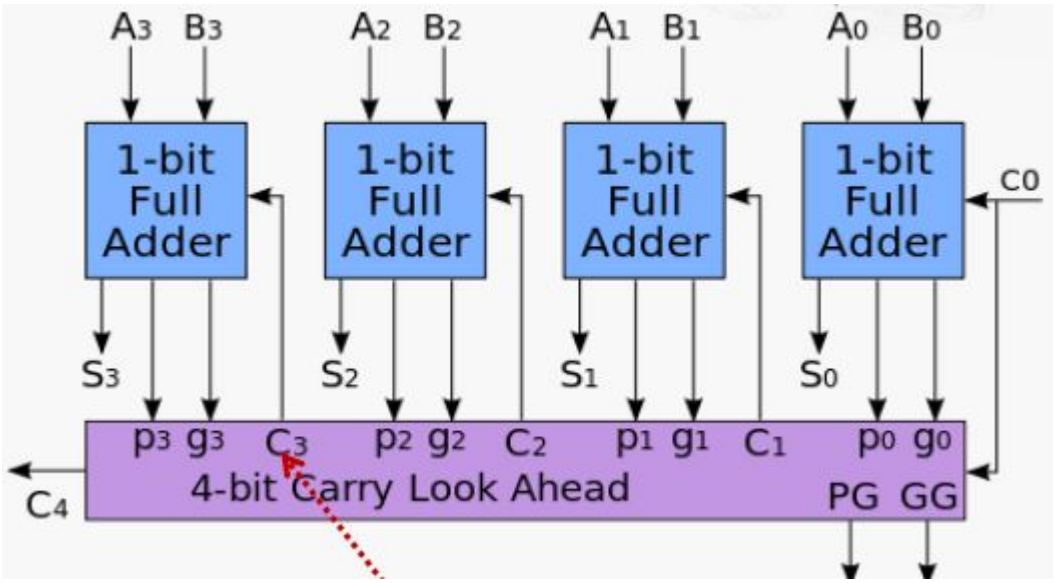
超前进位加法器（CLA）：提前计算出各级进位输出信号，而非等待其算出。根据如下公式递推地计算出各个进位输出即可。

$$C_{i+1} = G_i + P_i \cdot C_i$$

其中

$$G_i = A_i \cdot B_i$$

$$P_i = A_i + B_i$$

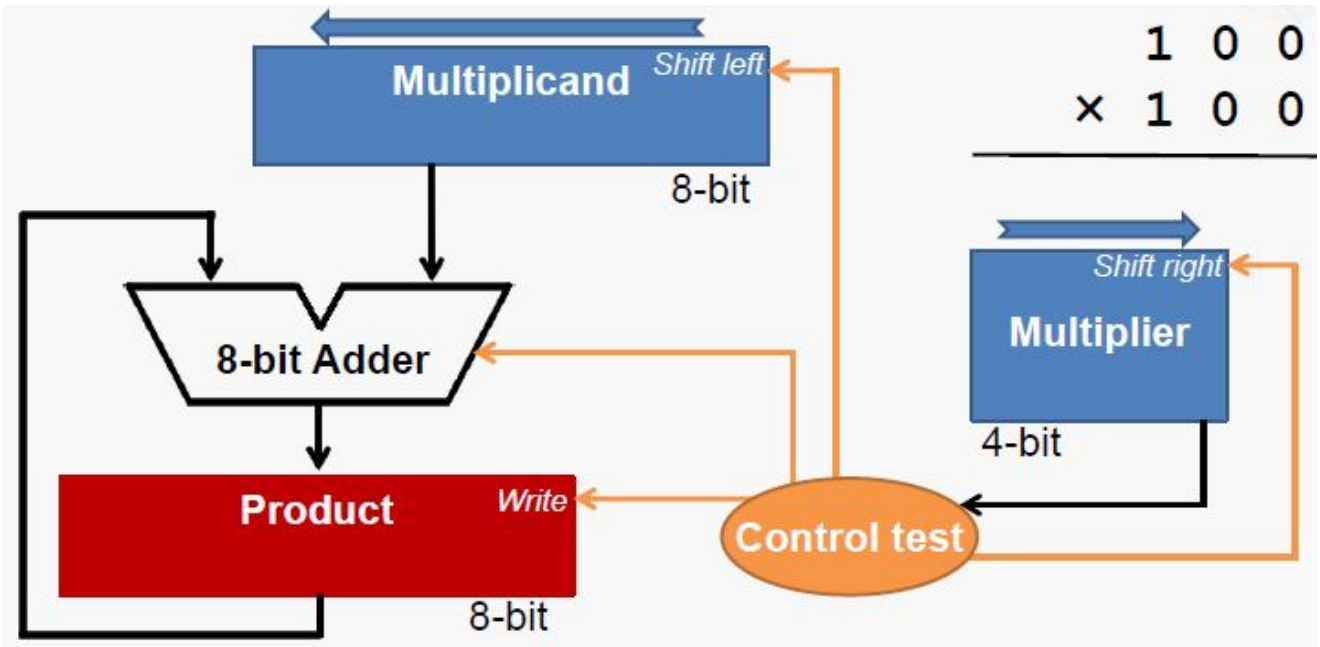


计算延迟时，算出最后一个C的门延迟，再加上最后一级全加器的门延迟（1级），即可得到总的门延迟时间。

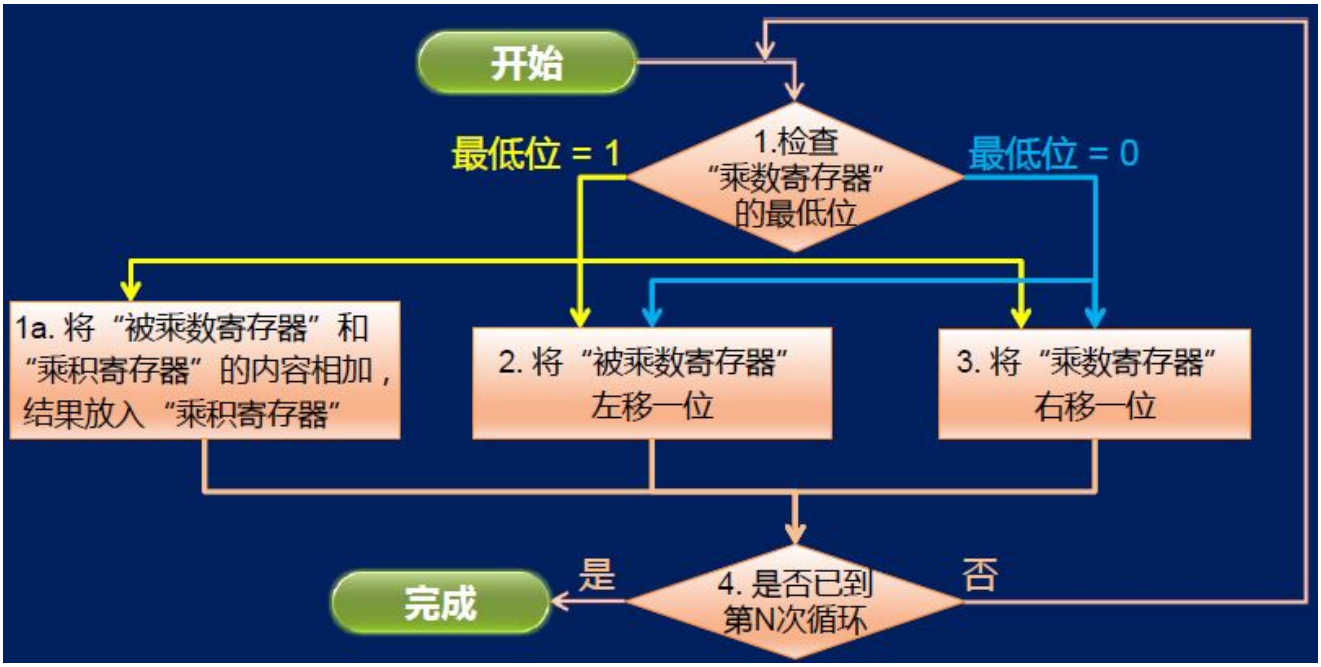
通常情况下，32-bit加法器采用多个小规模的超前进位加法器按照行波加法器的方式拼接而成。

乘法器的实现：Product = Multiplicand × Multiplier

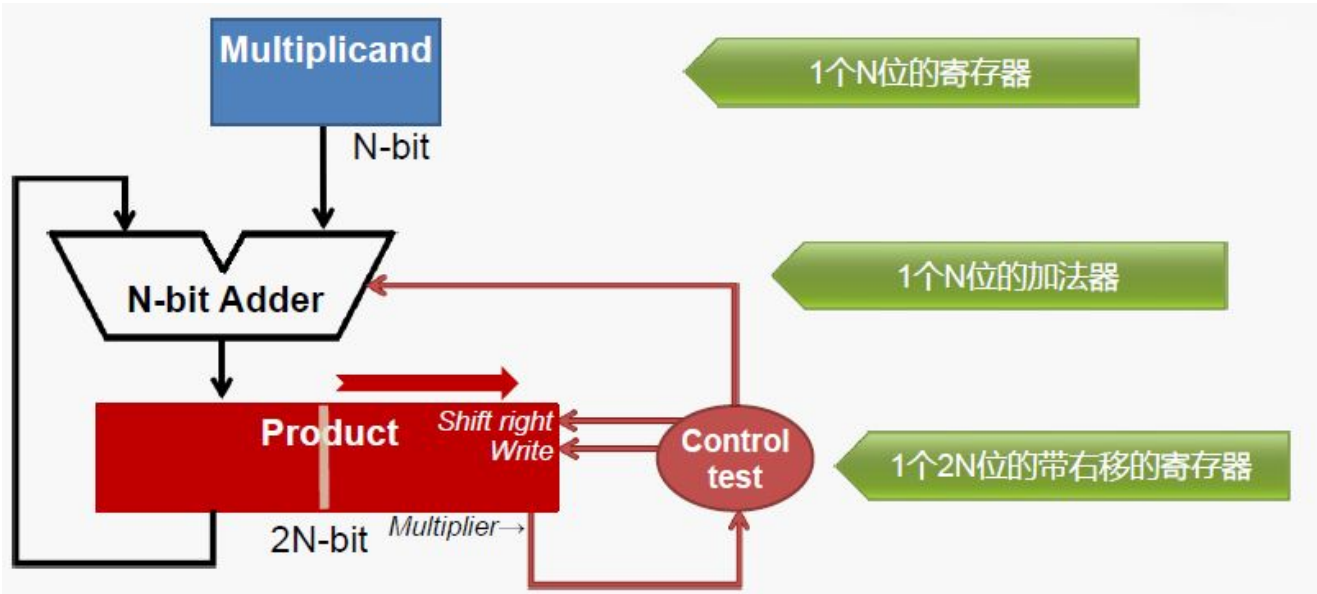
Version 1：n位乘法器，需要2n位的被乘数寄存器、2n位的积寄存器和n位的乘数寄存器，一个2n位的加法器，以及控制电路；要求被乘数寄存器可以实现左移，乘数寄存器可以实现右移。



工作流程为：检查乘数寄存器最低位，若为1，则将被乘数寄存器与乘积寄存器相加，结果放入乘积寄存器，被乘数寄存器左移一位，乘数寄存器右移一位；若为0，则只讲被乘数寄存器左移，乘数寄存器右移；完成后判断当前是否已达到第n次循环，若达到则终止，否则继续循环。



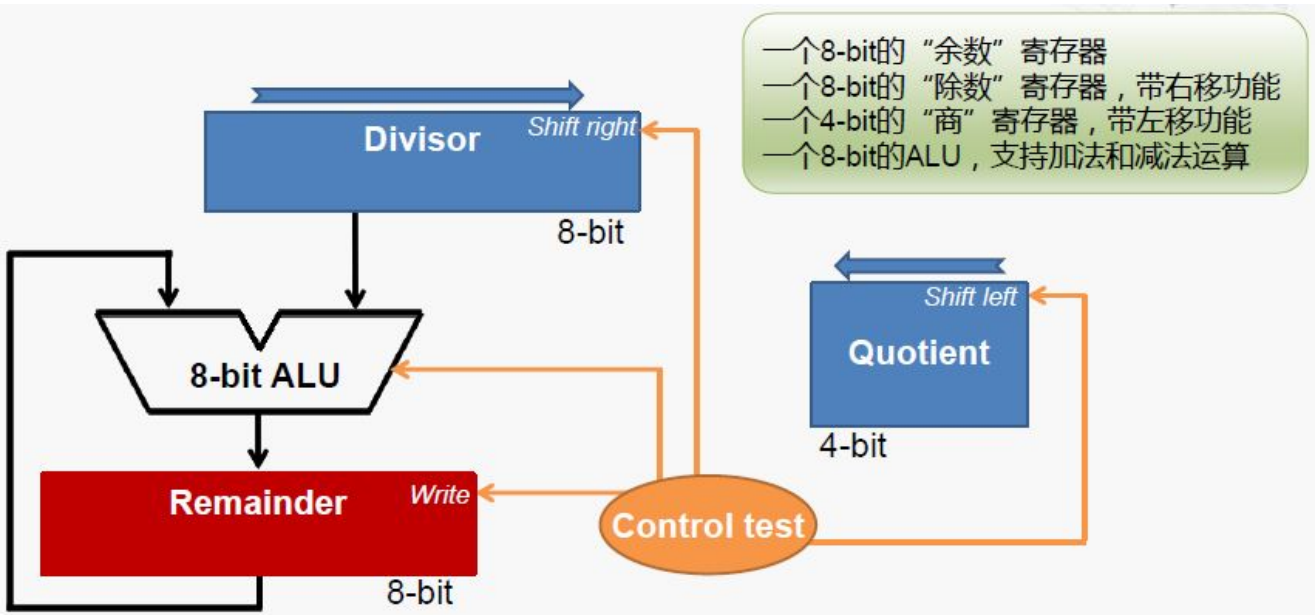
Version 2：面积优化，为了减少不必要的硬件资源，使用如下硬件。n位被乘数寄存器，无左移功能；2n位乘积寄存器，可以右移，初始时高n位为0，低n位为乘数，只有高n位会参与运算；n位加法器；控制电路



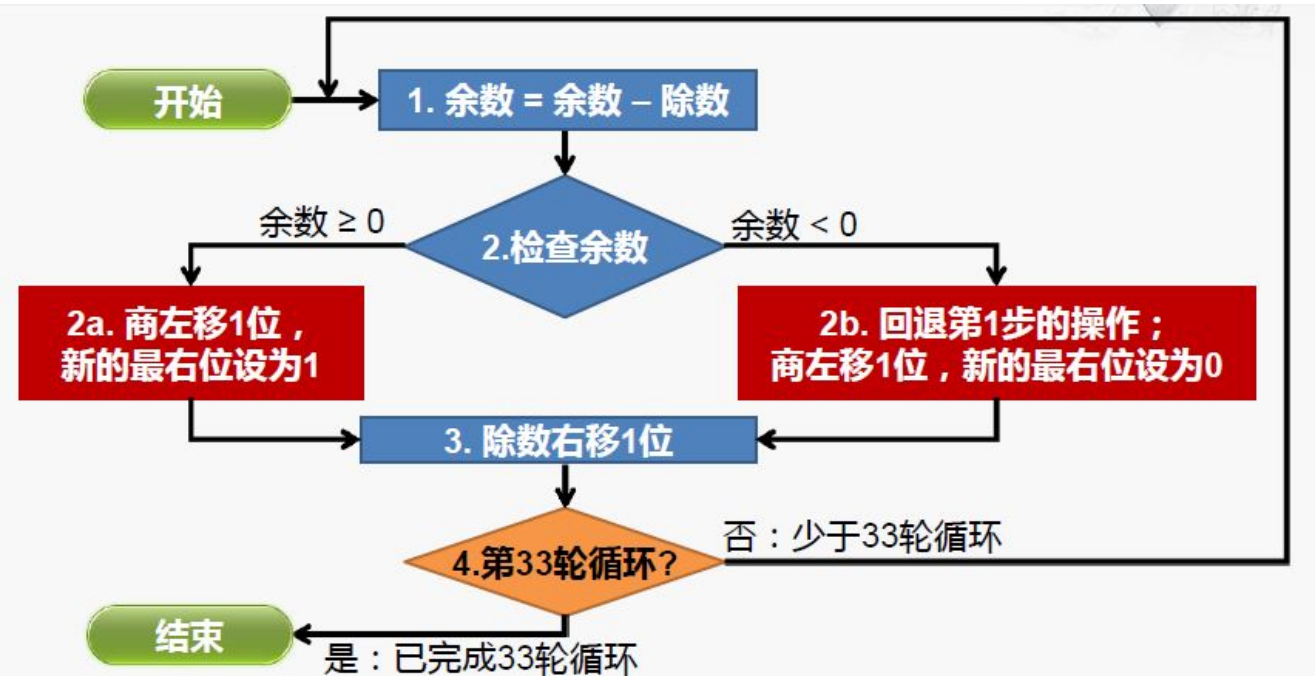
工作流程为：检查乘积寄存器最低位，若为1，则将被乘数寄存器与乘积寄存器高n位相加，结果放入乘积寄存器高n位，乘积寄存器右移一位；若为0，则只将乘积寄存器右移一位；完成后判断当前是否已达到第n次循环，若达到则终止，否则继续循环。

除法器的实现：被除数÷除数=商……余数

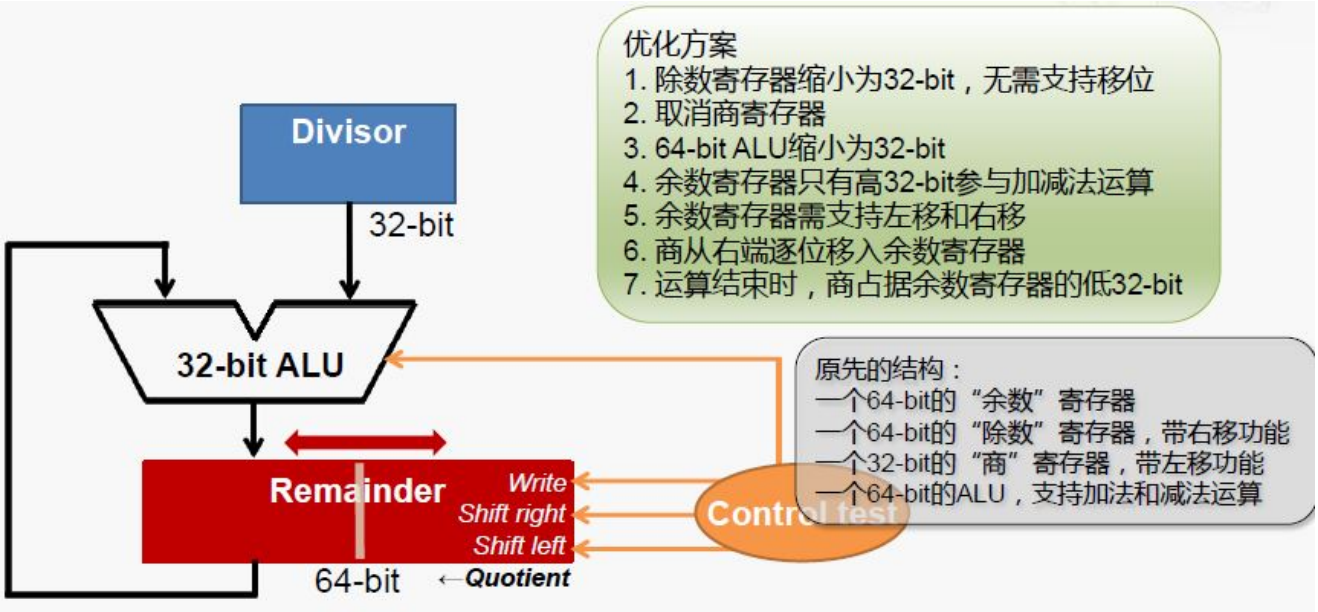
Version 1：n位除法器由如下部件组成。2n位余数寄存器，初始值为被除数；2n位除数寄存器，具有右移功能，初始时高n位为除数，低n位为0；n位商寄存器，具有左移功能，初始值为0；2n位ALU，支持加减法运算。



工作流程为：首先用余数寄存器减去除数寄存器，结果放在余数寄存器中；之后检查余数寄存器最高位，若为0，说明余数≥0，则商左移1位，新的低位设为1；否则寿命余数<0，则用余数加上除数，结果放在余数寄存器中（回退操作），商左移1位，新的低位设为0；之后除数右移一位；最后判断是否是第n+1轮循环，若是则终止，否则继续循环。



Version 2：面积优化。n位除数寄存器，不需要移位；取消商寄存器；n位ALU，支持加减运算；2n位余数寄存器，只有高32位参与运算，支持左移，初始值为被除数，运算过程中商从右端进入余数寄存器，运算结束时商占据余数寄存器低n位，余数占据余数寄存器高n位。



发现这样的一个除法器与乘法器的组件完全相同，仅仅是2n位寄存器的左移和右移的需求不同，因此可以乘法器和除法器组合起来，用一套组件完成，通过不同的控制逻辑即可完成乘法或者除法。

6. 单周期处理器设计

设计一个单周期处理器：确定指令系统，分析指令得到数据通路的需求；选择合适的组件；连接组件建立数据通路；分析每条指令以确定控制信号；集成控制信号形成控制逻辑；根据控制逻辑实现控制器

考虑一个简化的MIPS指令系统（只有R型和I型指令）：无符号加减法（addu rd, rs, rt；subu rd, rs, rt），立即数逻辑或（ori rt, rs, imm16），装载和储存一个字（lw rt, imm16(rs)；sw rt, imm16(rs)），条件分支（beq rs, rt, imm16）。我们需要如下的组件：

ALU，支持加、减、或和比较相等的运算，操作数为两个32位数，来自寄存器或者扩展的立即数，输出运算后的32位数；

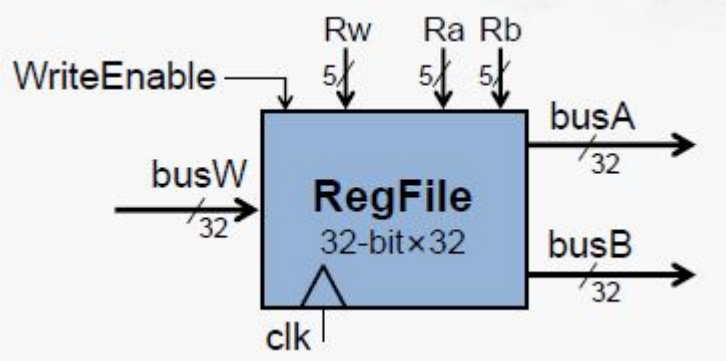
立即数扩展元件，支持零扩展和符号扩展，将一个16位立即数扩展为32位；

程序计数器PC，32位寄存器，指向当前待执行的指令，需要支持+4操作和加一个立即数的操作。

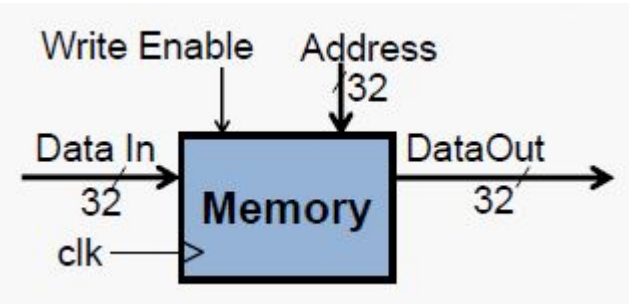
寄存器堆，32个32位宽的寄存器，支持两个读操作rs和rt，支持一个写操作rt或rd（“两读一写”寄存器堆）

两个储存器（对应cache），一个只读的指令存储器，地址和数据均为32位（指令cache），一个可读可写的数据存储器，地址和数据均为32位（数据cache）

寄存器堆：内部包含32个32位寄存器；具有三个数据接口，busA和busB为两组32位的数据输出接口，busW为一组32位的数据输入接口；具有三个读写控制接口，Ra和Rb分别5bit，选中对应寄存器将内容放到busA和busB上，Rw 5bit，选中对应寄存器，在时钟信号上升沿如果写使能信号有效，则将busW的内容写入该寄存器；具有一个时钟信号输入clk；具有一个控制信号接口WriteEnable写使能信号，为1时表示Rw可写。

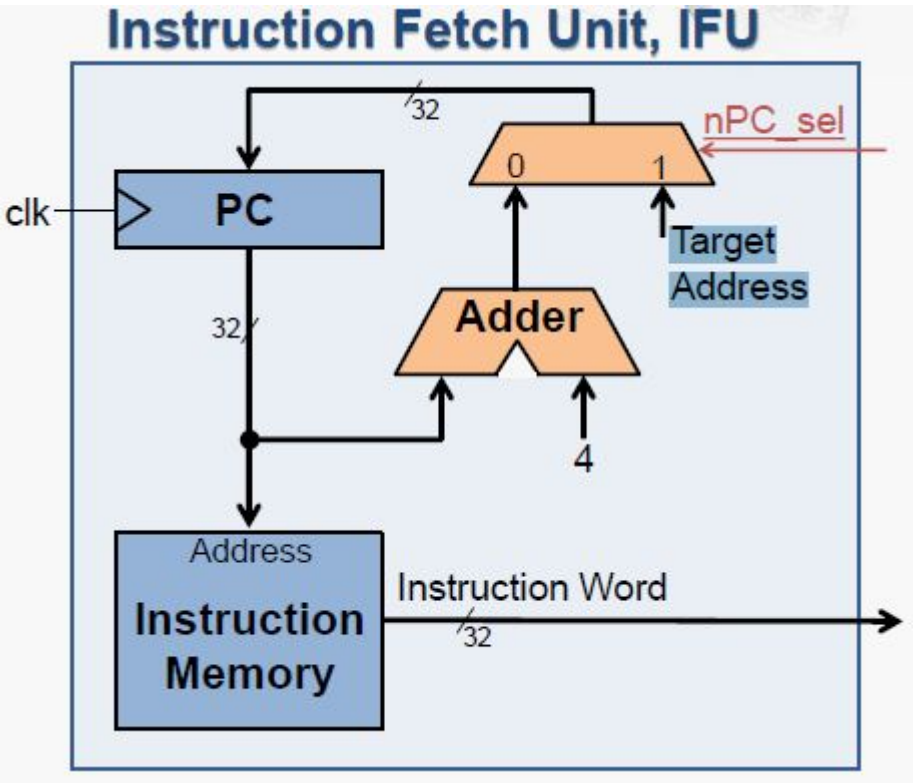


存储器：具有两个数据接口，Data_In和Data_Out分别为32bit的数据输入和数据输出接口；一个读写接口，Address为32位的地址信号，指定了一个存储单元，将其内容送至Data_Out；一个控制信号WriteEnable写使能信号，在时钟信号的上升沿如果写使能有效则将Data_In写入Address对应单元。



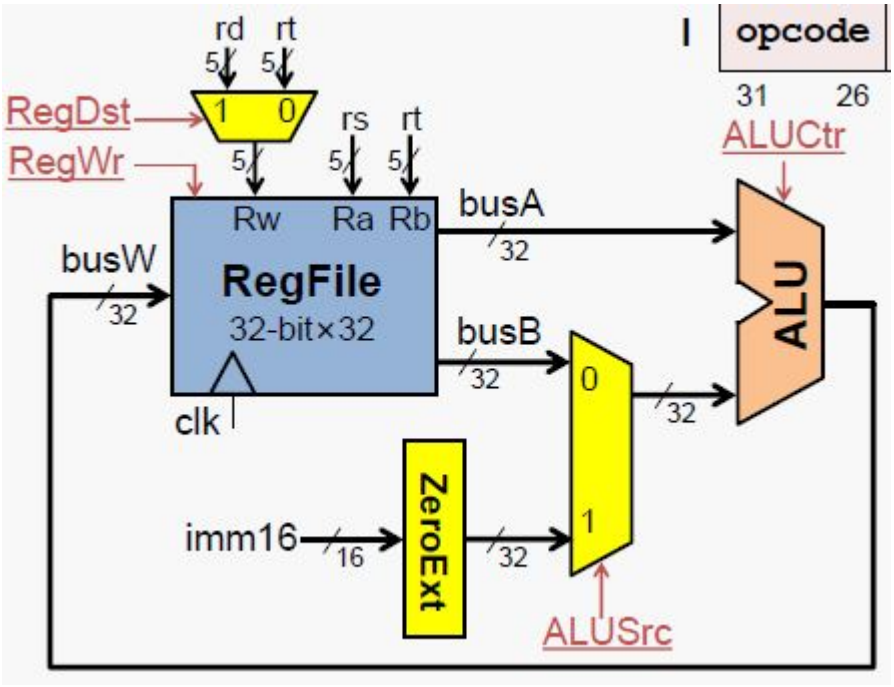
建立数据通路：需要根据指令共同的需求和不同指令的不同需求，来连接各个组件，建立数据通路。

对于取指令，这是对于所有指令都相同的，一个取指单元IFU需要包含一个PC，还需要有更新PC的部件。为了满足分支指令，还需要有一个选择信号来控制是+4操作还是跳转到分支目标地址。



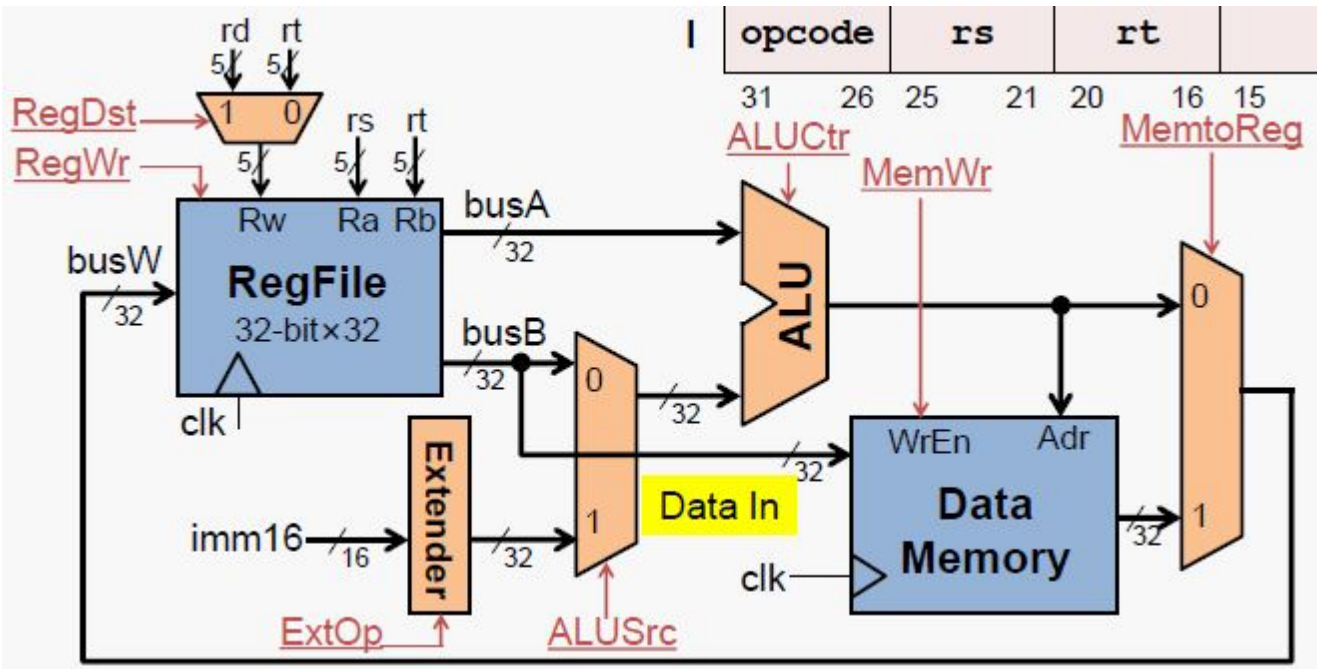
加减法指令：addu/subu rd, rs, rt。R[rd]=R[rs] op R[rt]　　Ra和Rb分别为rs和rt，Rw为rd，需要寄存器堆写使能信号RegWr和ALU运算类型选择的信号ALUCtr。

逻辑运算指令：ori rt, rs, imm16。R[rt]=R[rs] op ZeroExt[imm16]　Ra为rs，Rw为rt，此处需要在Rw处加入多选器（自然而然地需要加入RegDst控制信号来进行选择）来选择rd或者rt作为Rw；ALU的输入为busA和一个32位的立即数，为了能够输入一个立即数，在ALU输入的B接口加入多选器（同时再多选器上加入ALUSrc控制信号进行选择）来选择是使用立即数还是使用busB，同时还需要将imm16零扩展为32位立即数，因而在多选器的立即数接口上再加入一个零扩展元件。

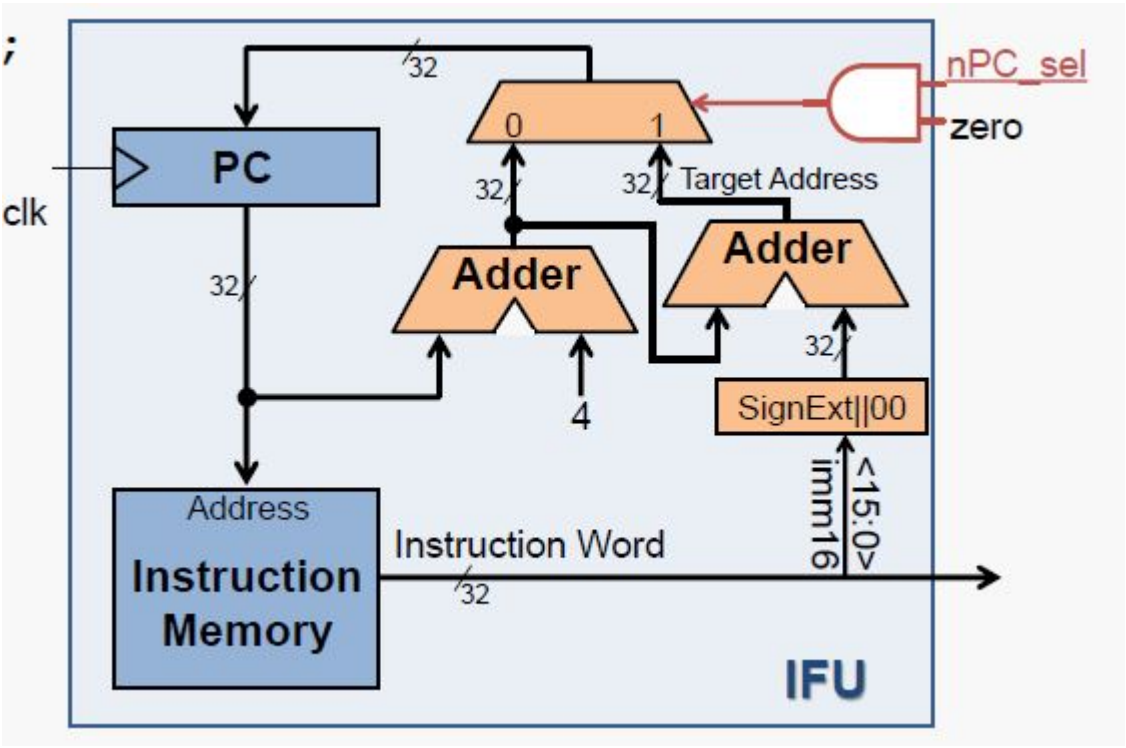


装载访存指令：lw rt, imm16(rs)。R[rt]=Mem[R[rs]+SignExt[imm16]]　需要计算busA（来自rs寄存器）和一个有符号扩展的32位立即数，因此将零扩展原件换为扩展元件，支持零扩展或有符号扩展（需要控制信号ExtOp进行选择）；ALU计算出的值为存储元件中的地址，从存储元件中读出值之后需要输出至busW，因而需要增加一个存储元件，并在存储元件之后增加一个多选器（加入MemtoReg信号来选择）来选择busW是ALU输出还是存储元件输出。

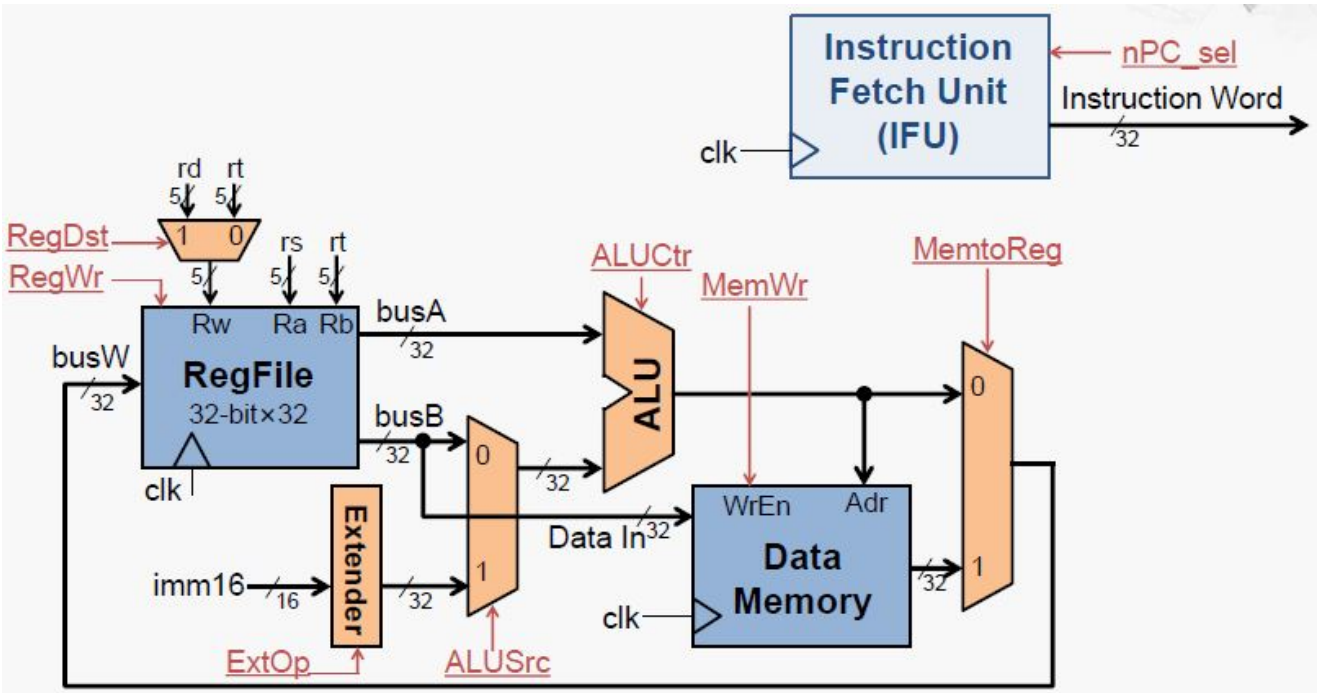
储存访存指令：sw rt, imm16(rs)。Mem[R[rs]+SignExt[imm16]]=R[rt]　需要将busB（来自rt寄存器）的值输入到存储单元的Data_In接口，新增一条连线，同时也需要加入存储单元的写使能信号MemWr。



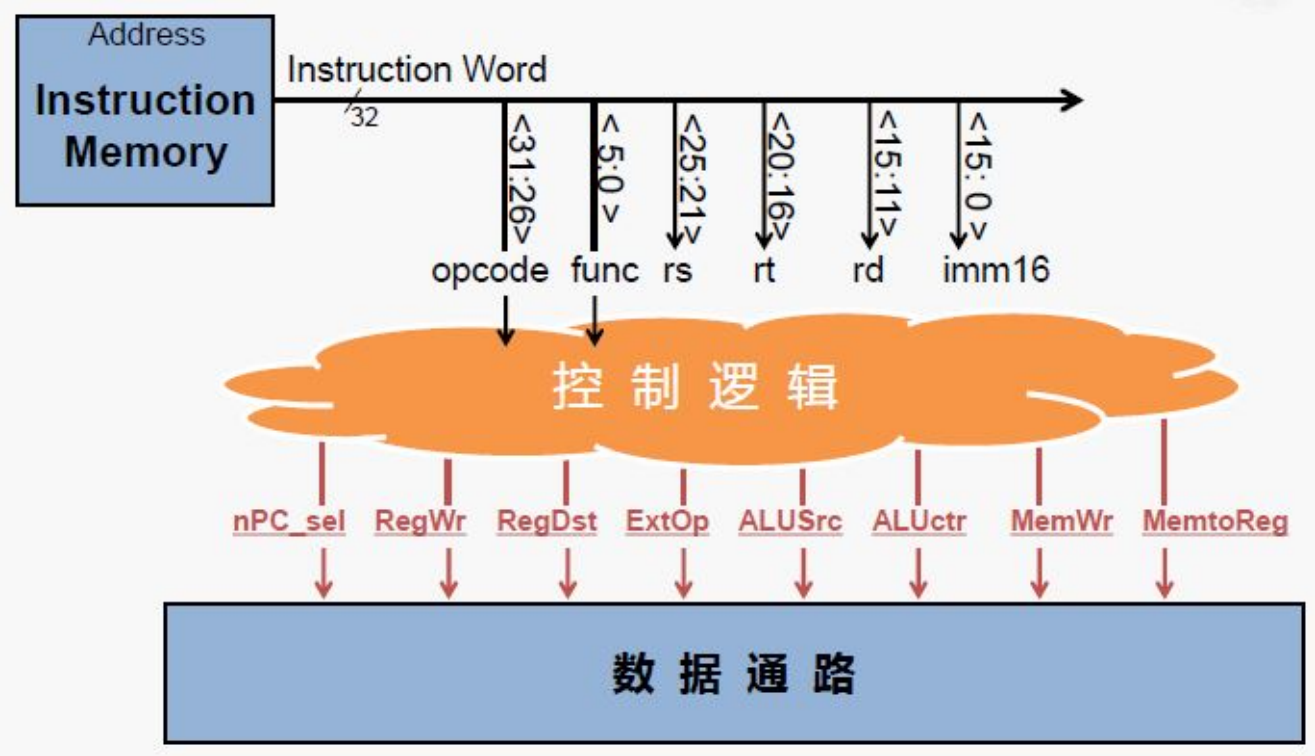
分支指令：beq rs, rt, imm16。PC = ((R[rs] - R[rt] == 0) ? PC + 4 + SignExt[imm16] * 4 : PC + 4) 之前的IFU尽管考虑到了nPC_sel信号的问题，但没有考虑到具体的目标地址计算的问题，因而IFU需要有如下的修改。



最终的数据通路如下图（还没有控制逻辑）



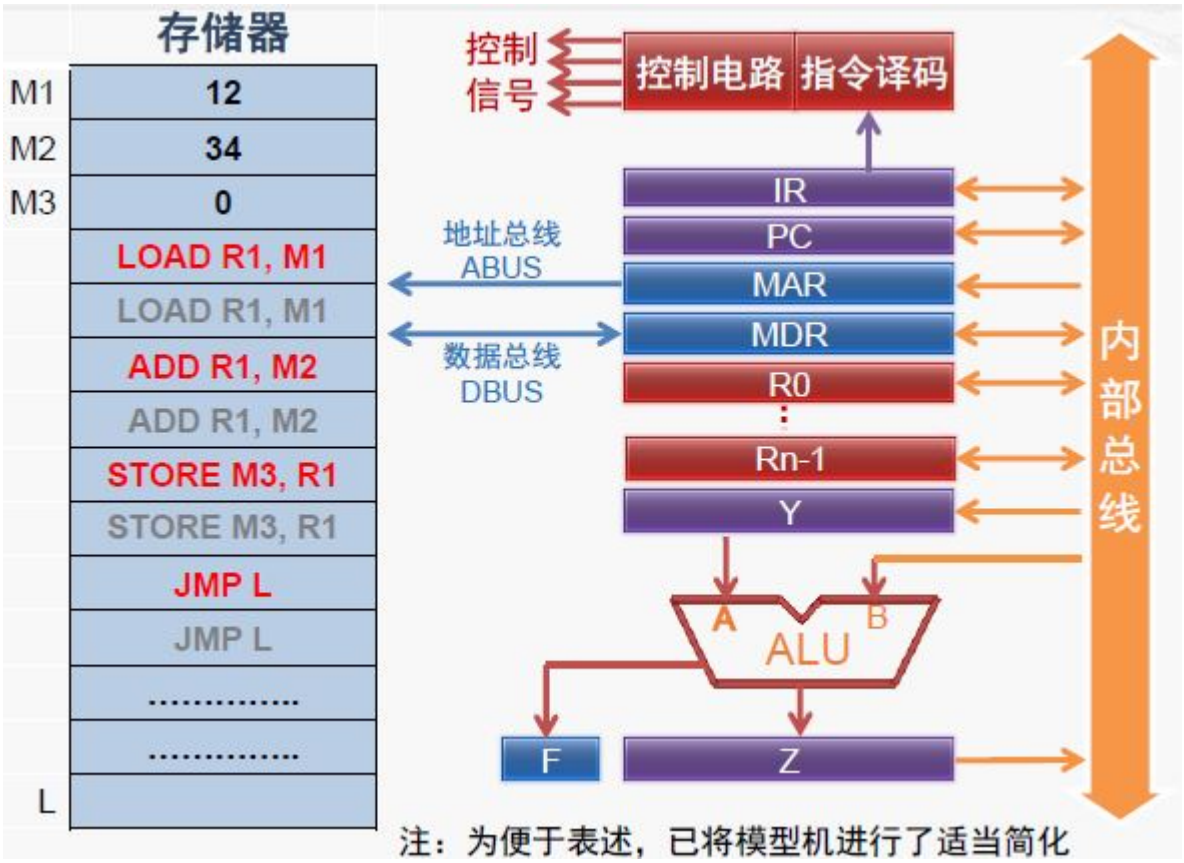
上述数据通路中有8个控制信号，对于不同的指令，这些控制信号也是不同的，并且是映射的，因而需要设计一套控制逻辑，从指令“运算出”信号，传至数据通路中。由于指令编码中只有31:26的opcode域和5:0的funct域是和编码相关的，因而根据这两个域进行设计即可。

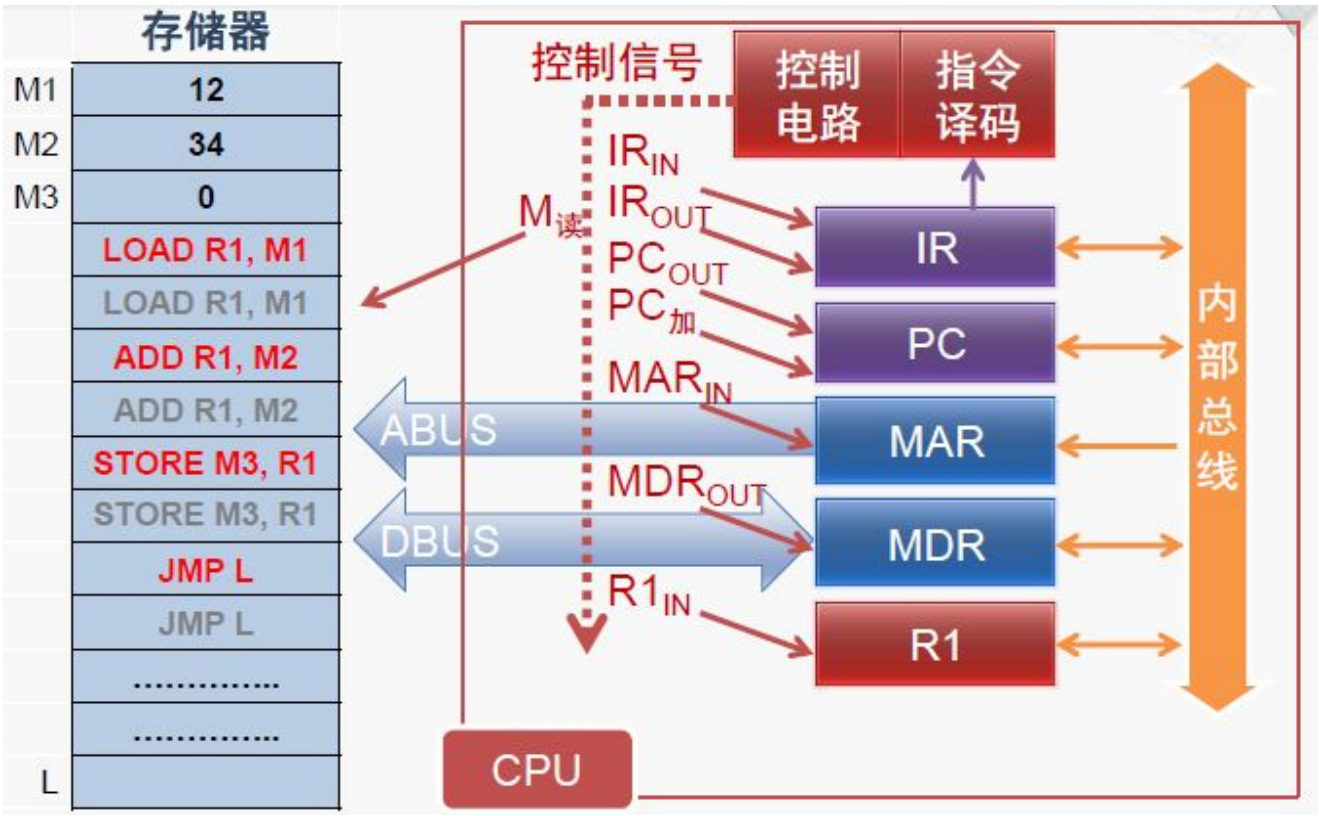


控制信号的真值表如下图，其中0和1均表示确定的值，x表示0/1均可的值。根据真值表直接通过逻辑电路实现即可。

func	100000	100010	/			
opcode(op)	000000	000000	001101	100011	101011	000100
	add	sub	ori	lw	sw	beq
RegDst	1	1	0	0	x	x
ALUSrc	0	0	1	1	1	0
MemtoReg	0	0	0	1	x	x
RegWr	1	1	1	1	0	0
MemWr	0	0	0	0	1	0
nPC_sel	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x
ALUctr<1:0>	00(ADD)	01(SUB)	10(OR)	00(ADD)	00(ADD)	01(SUB)

以上均为MIPS的单周期处理器设计，下面考虑一个x86系统（CISC），简化的模型机和控制信号系统如下。





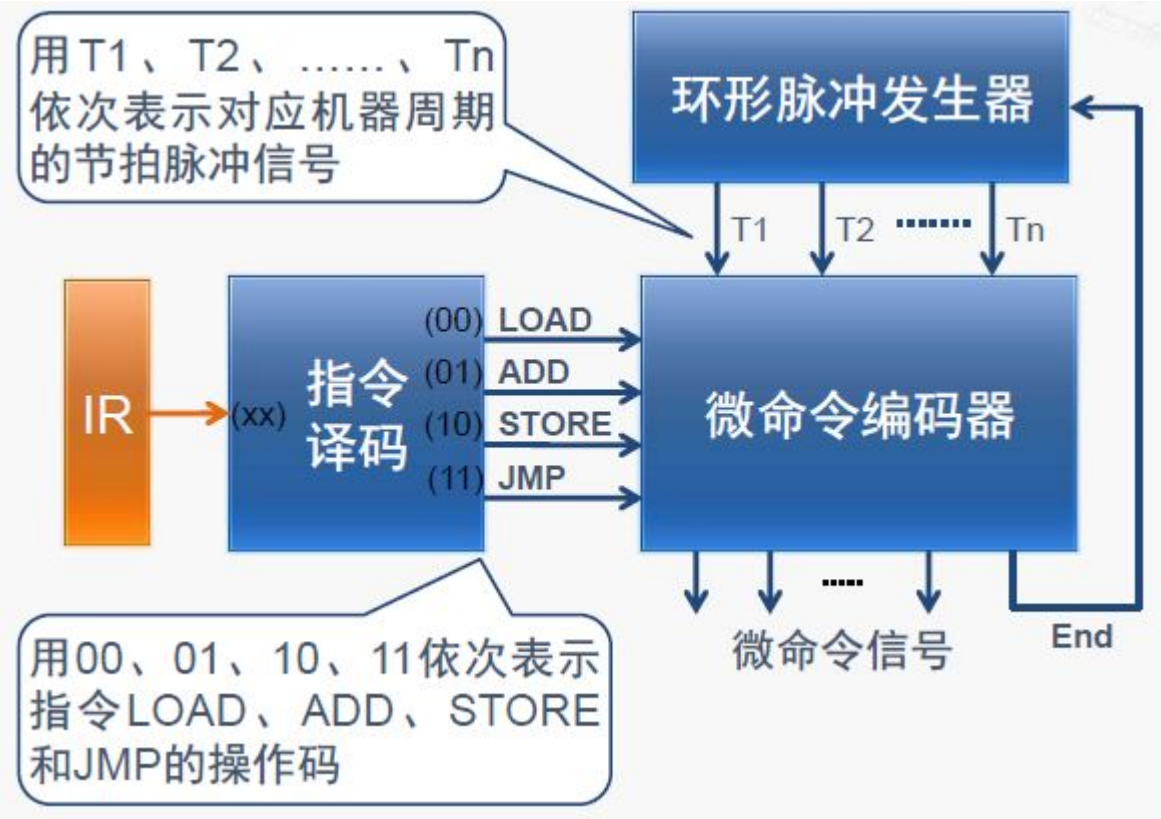
对于一条x86指令，需要如下的控制逻辑：

取指令，并形成下一条指令的地址——控制器发出PCout和MARin信号，使得MAR←PC；控制器发出Mread信号，使得MDR←存储器中内容（指令）；控制器发出PCadd信号，使得PC←PC+n（n为指令长度）；控制器发出MDRout和IRin信号，使得IR←MDR。这一段的控制逻辑各个指令都是相同的。

根据指令的不同，发出不同的信号以执行指令。

在设计CISC的控制器时，既可以像MIPS一样设计一个完全由硬件控制的控制器（硬布线控制器），也可以设计一个软硬件共同控制的控制器（微程序控制器）。

硬布线控制器：环形脉冲发生器（循环地产生节拍脉冲信号）+指令译码器（确定IR中存放的是哪一条指令）+微命令编码器（在不同的节拍脉冲信号的同步下产生相应的微命令信号/控制信号）



对于LOAD，ADD，STORE，JMP四条指令来说，微命令编码器需要产生的信号表如下表。这是从输入（四个指令输入，七个时钟输入）的角度来看的，对不同的输入组合需要那些信号输出；但在设计电路时，需要从输出的角度来看，考虑不同的输出会有怎样的输入组合，如此便可以得到表格下的各个信号输出的逻辑表达式。

指令名	T1	T2	T3	T4	T5	T6	T7
LOAD (00)	PC_{OUT} MAR_{IN} $M_{读}$ $PC_{加}$	MDR_{OUT} IR_{IN}	IR_{OUT} MAR_{IN} $M_{读}$	MDR_{OUT} $R1_{IN}$	End		
ADD (01)	同上	同上	IR_{OUT} MAR_{IN} $M_{读}$	MDR_{OUT} Y_{IN}	$R1_{OUT}$ add	Z_{OUT} $R1_{IN}$	End
STORE (10)	同上	同上	IR_{OUT} MAR_{IN}	$R1_{OUT}$ MDR_{IN} $M_{写}$	End		
JMP (11)	同上	同上	IR_{OUT} PC_{IN}	End			

PC_{OUT}=T1　　MAR_{IN}=T1+T3 (LOAD+ADD+STORE)

$PC_{加}$=T1　　$M_{读}$=T1+T3 (LOAD+ADD)

IR_{IN}=T2　　MDR_{OUT}=T2+T4 (LOAD+ADD)

IR_{OUT}=T3　　PC_{IN}=T3 JMP

Y_{IN}=T4 ADD　　$R1_{IN}$=T4 LOAD+T6 ADD

MDR_{IN}=T4 STORE　　$R1_{OUT}$=T4 STORE+T5 ADD

add=T5 ADD　　$M_{写}$=T4 STORE

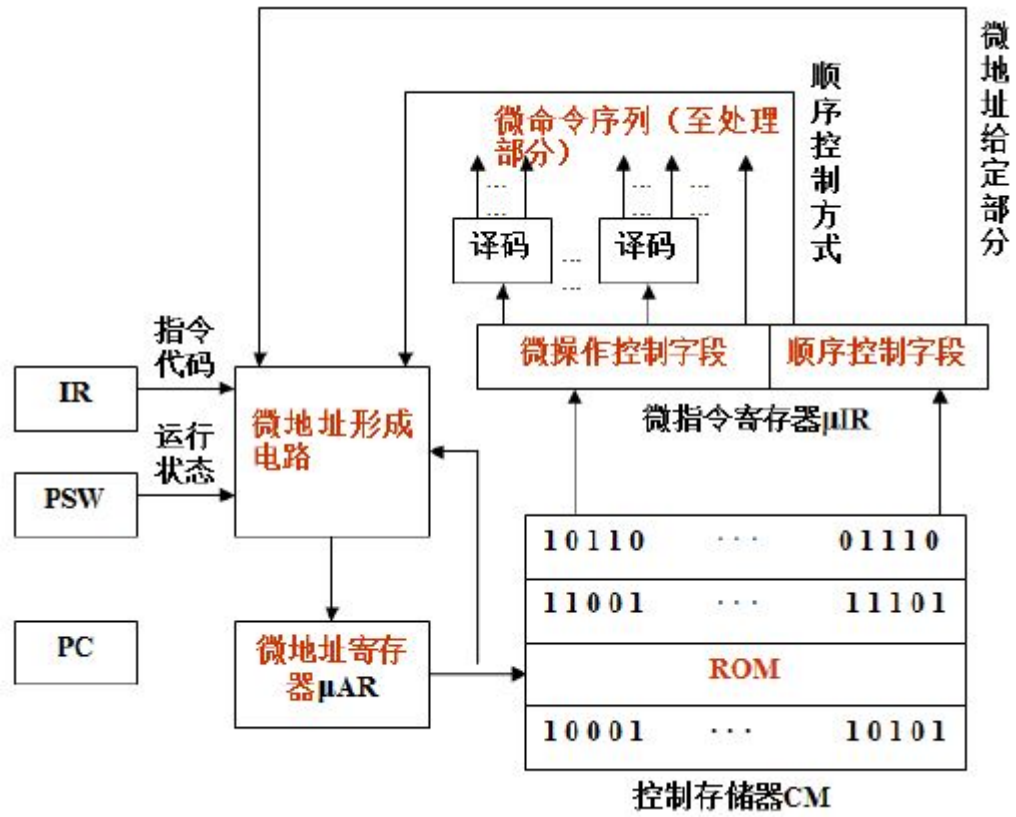
Z_{OUT}=T6 ADD

End=T5 (LOAD+STORE)+T7 ADD+T4 JMP

硬布线控制器的关键就在于直接使用组合逻辑电路来产生微操作信号作为控制信号。其优点是：硬件电路实现所有逻辑，指令执行速度很快；缺点是：控制逻辑电路非常复杂，设计和验证的难度大，扩充指令集和修改指令集会很困难。

微程序控制器将每一条机器指令分解为一系列更加基本的微操作（类比于硬布线控制其中的一个“周期”）；控制器只需要将需要产生的信号以微代码的形式编为微指令即可，在制造CPU的时候，将这些微命令储存在CPU内部的一个ROM里，在CPU执行程序时，从ROM中取微指令来控制相关部件即可。

微指令即为控制信号序列，是一个0-1序列，每一位对应某一个特定的信号，1表示发出该信号，0表示不发出；若干的微指令组合起来形成一段微程序，用来解释执行某一条机器指令。



控制器存储器CM：ROM，存放微程序，每个存储单元存放一个微指令。

微指令寄存器μIR：存放CM中读出的微指令，分为两部分，“微操作控制字段”和“顺序控制字段”。

微地址形成电路：形成下一条微指令的微地址，形成的依据为（微地址给定部分+现行微指令中的顺序控制方式+机器指令的相关代码+机器运行状态）

微地址寄存器μAR：保存微指令对应的微地址，指向CM单元；当读出一条微指令后，微地址形成电路将下一个微指令的地址存入μAR中。

微指令寄存器中的微指令有如下的编码方式：

直接表示法：微操作控制字段直接存放的是微命令序列，之后存放顺序控制字段；如此编码简单直观，但编码效率很低。

编码表示法：将微指令代码分组编码，将不能同时出现的相斥的信号放在一组之中，然后将其编码为较短的编码；如此编码可以减少CM的存储压力，但微指令代码必须要译码，会增加硬件开销和控制信号的延迟，从而影响微程序执行速度。

混合表示法：对于速度要求高，或者与其他控制信号都相斥的控制信号直接表示；对于其他的控制信号编码表示。

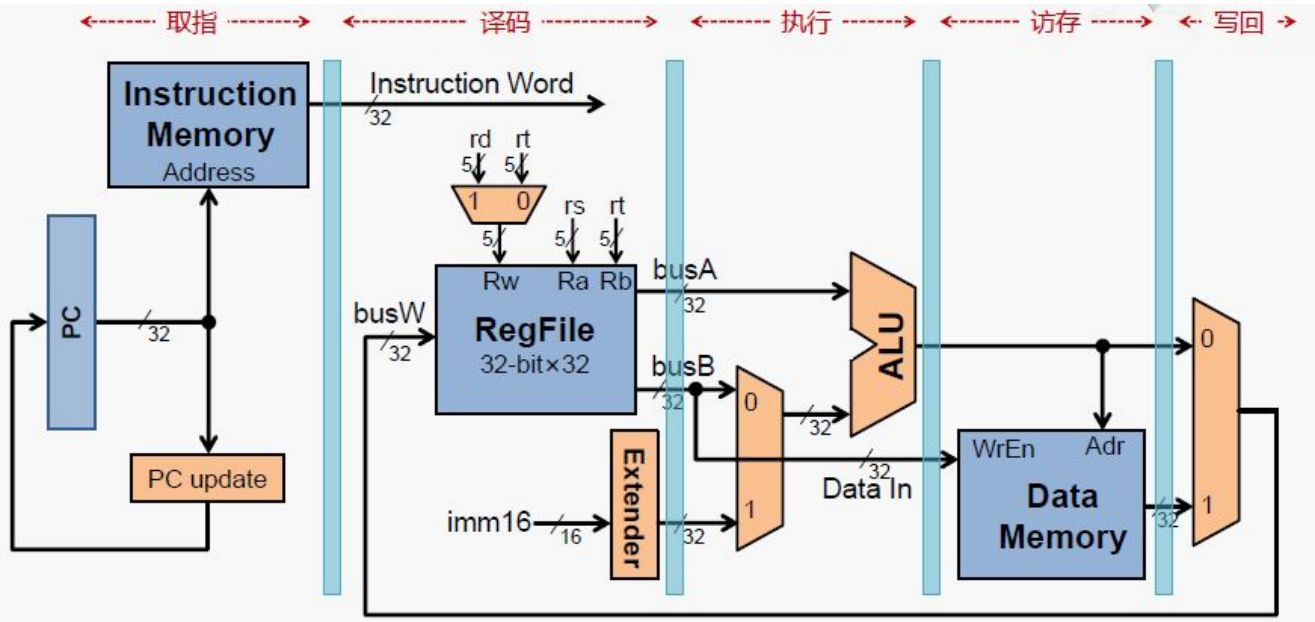
顺序控制字段可以有很多表示方式，在这里针对这个四指令的指令体系给出一种简单的5bit字段表示。abcde，其中a代表是否发生分支，bcde为下一条指令的微地址，分支在这一体系下只会在取指之后发生，发生分支后，bc为取得指令的操作码，de为10（此时bcde恰好为取得指令的第一条微代码的微地址）；若没有分支，则bcde为顺序执行的下一条微代码的微地址。

微地址	指令	机器周期	微操作字段	下址字段
0000	取指令	T1	1111 0000 0000 0000	0 0001
0001		T2	0000 1100 0000 0000	1 xx10
0010	LOAD(00)	T3	0110 0010 0000 0000	0 0011
0011		T4	0000 1000 1000 0000	0 1001
0110	ADD(01)	T3	0110 0010 0000 0000	0 0100
0100		T4	0000 1000 0100 0000	0 0101
0101		T5	0000 0000 0010 0100	0 0111
0111		T6	0000 0000 1000 0010	0 1001
1010	STORE(10)	T3	0100 0010 0000 0000	0 1000
1000		T4	0000 0000 0011 1000	0 1001
1110	JMP(11)	T3	0000 0011 0000 0000	0 1001
1001	End	Tn	0000 0000 0000 0001	0 0000

微程序控制器的缺点是执行速度较慢，因为其控制逻辑由微程序的软件实现。优点在于控制器规整，将程序技术和储存逻辑引入CPU中；灵活性强，可以方便地修改和扩充指令。

7. 流水线处理器

流水线技术将单周期处理器的一条指令的工作切分为若干部分，按照流水方式进行，从而达到增加吞吐量的目的（在流水线充满的情况下），但这会让一条指令的执行时间增加（需要在原指令的基础上加入流水线寄存器的时间）。



流水线技术的优化：平衡划分，因为流水线的时钟周期由最长的步骤决定；超级流水线，增加流水线深度（级数），使得流水线吞吐率和时钟频率进一步提高，但这不是无限制的（目前一般的流水线深度都在15级左右）；超标量流水线，两条或以上流水线结构并行工作。

MIPS执行指令的主要步骤为5步：取指（Fetch），译码（Decode），执行（Execute），访存（Memory），回写（Write-back）。因而可以按照这五步来划分流水线。

1.取指 **Fetch**	从存储器取指，更新**PC**
2. 译码 **Decode**	指令译码，从寄存器堆读出寄存器的值
3. 执行 **Execute**	运算指令：进行算术逻辑运算　　访存指令：计算存储器的地址
4. 访存 **Memory**	Load指令：从存储器读数据　　Store指令：将数据写入存储器
5. 回写 **Write-back**	将数据写入寄存器堆

486是第一款x86流水线CPU，五级流水线，包括取指，译码，地址生成，执行，回写五个步骤。

Pentium是第一款超标量的x86 CPU，双发射5级流水线（U流水和V流水）

1964年的第一台超算，CDC6600被认为是最早采用超标量技术的计算机，但CDC6600没有采用流水线技术，CDC7600采用了超标量流水线技术。

现代多核CPU中一个核中有一整套超标量流水线，和其Cache等。

冒险Hazard：阻止下一条指令在下一个时钟周期开始执行的情况。解决冒险的通用的最简单的方法就是使流水线停顿（stall），产生一个空泡（bubble），但这会使得流水线无法填充满，降低流水线的性能。

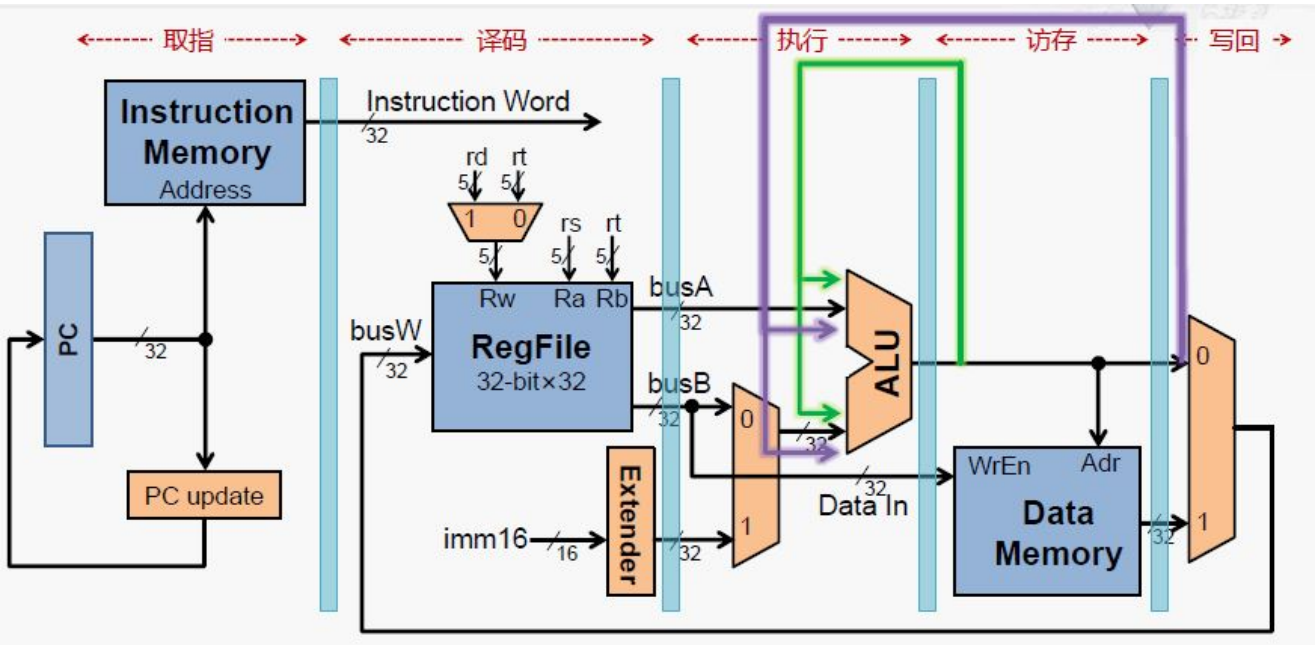
结构冒险：下一条指令所需要的硬件部件正在为此前的指令工作。

比如如果指令和数据存放在同一个存储器之中，会出现一条指令要写数据，一条指令要取指的情况；为了解决这一问题，可以选择将下一条指令stall，也可以选择将指令和数据放在不同的存储器之中（I-cache和D-cache），读写就不再矛盾。

再比如两条指令中一条要写寄存器，另一条需要读这个寄存器，也会出现冒险；解决方法就是将读写口分开设为不同的独立接口，并且前半个时钟周期写，后半个时钟周期读。

数据冒险：一条指令需要使用之前的指令的运算结果，但这个结果并没有写回。

比如前一条指令是一条运算指令，正在执行阶段，写回阶段将在两个周期之后，而下一条指令需要读运算指令要写回的寄存器，就发生了数据冒险；解决方法可以是停顿下一条指令两个周期，以等待写回完成，但也可以选择数据前递，将上一条指令执行阶段计算出的值直接传给下一条指令作为读寄存器取出的值，这种前递Forwarding也叫做旁路Bypass。



数据前递可以解决所有由运算产生且和寄存器写回相关的数据冒险，并且不停顿。但对于读存储器并向寄存器写回引起的数据冒险，前递无法在读内存之前就得到该值，因而必须停顿一个周期之后再前递。

控制冒险：需要根据之前的指令的结果来决定下一步的行为，具体表现在分支指令上。由于分支指令会有两种不同的控制流，因而会产生冒险。

转移指令会对流水线性能产生很大的影响——因为平均每隔4到7条指令就会有一条转移指令，大致占程序的15%到25%；并且随着流水线深度增加，超标量数的增加，转移指令的损失也越来越大（一旦转移错误，之前取出的所有指令都将作废，重新沿着正确的控制流执行）。

转移开销：转移指令被执行并确实发生转移时，产生的开销，包括将按顺序预取的指令废除（排空流水线）的开销，和从转移目标地址重新取指令的开销。转移开销的构成——转移条件判定引起的开销+生成目标地址引起的开销。

延迟转移技术：如果转移条件在取下一或二条指令时并未计算完成，那么可以在转移指令之后插入若干条合适的指令，比如可以将指令的顺序调换，将不会影响转移条件的指令放在转移指令之后，当被调度的指令执行完成后，转移指令的目标地址和转移条件都已经计算完成。

转移预测技术：依据指令过去的行为来预测将来的行为，需要预测转移条件的判定和转移的目标地址。

硬件固定预测部转移：总是顺序地取下一条指令。实现简单但预测效果不好

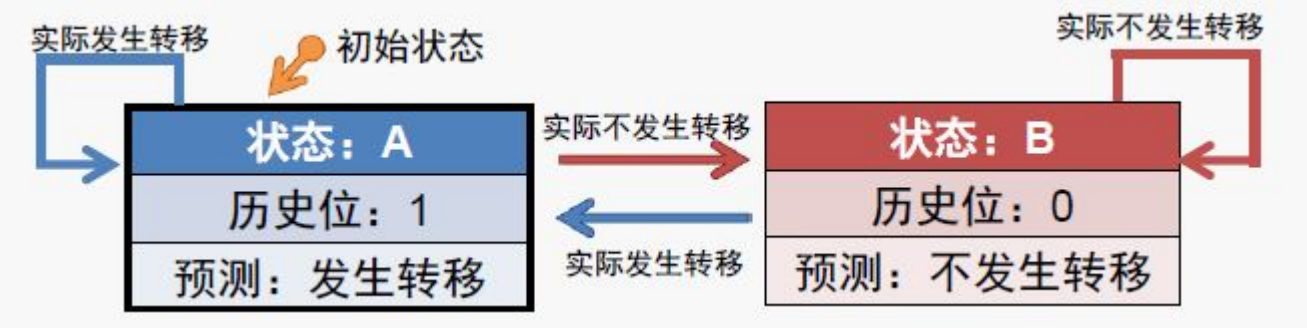
编译制导的预测（Motorola M88110）：在转移指令的编码中加一位，由编译器设置该位来告诉硬件是预测转移还是预测不转移。可以使软件on各国指令类型和历史信息对不同指令进行不同的预测，但需要修改ISA，无法移植。

基于偏移的预测（IBM RS/6000首先使用）：如果转移指令地址和转移目标地址的相对偏移为负，则预测转移（有可能是循环），否则预测不转移。BTFN（Back Taken, Forward Not-taken）

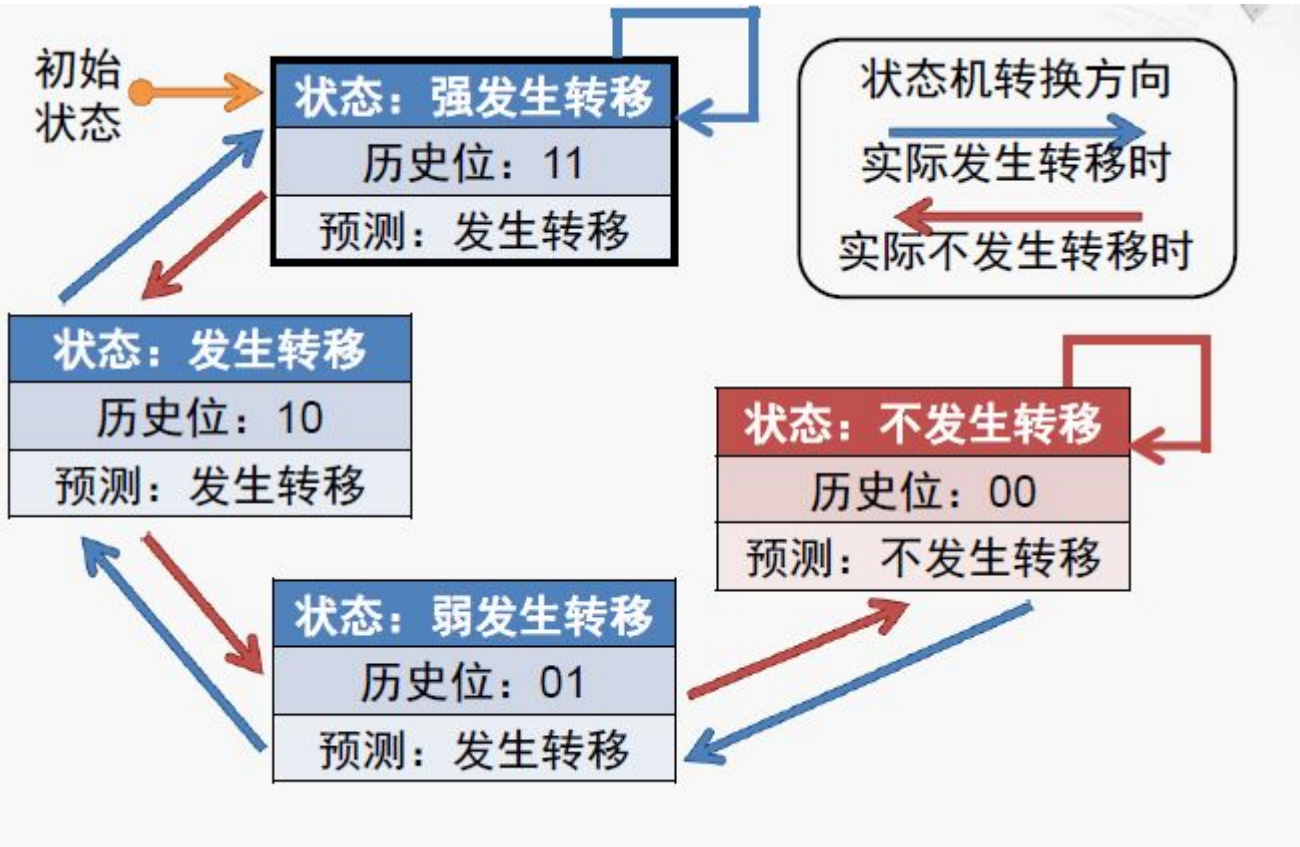
基于历史信息的预测（目前普遍使用）：转移与否取决于之前的指令的执行情况。

历史位信息预测：通过之前的转移指令是否发生转移来预测这一条转移指令是否会发生转移。

取指部件中，使用1位历史信息进行预测。历史位为1表示最近一次转移指令实际发生了转移，为0表示实际不发生转移。对于每一条出现过的转移指令，都有对应的历史信息（如果历史信息表足够大的话）。一位历史信息的初始状态



Pentium两位历史信息转移预测器，11，10，01都预测发生转移，分别为强发生转移状态，发生转移状态，弱发生转移状态。Pentium倾向于转移，其初始状态设置为强发生转移状态。



除了Pentium的倾向转移的历史位的设定，还有其他的方式，比如均衡地设置。一般情况，历史信息位越高，预测越正确，但三位之后提升很微弱了，采用2-3位居多。

转移目标缓冲器：BHT实现了“要不要转移”的预测，BTB用来预测“转移到哪里”

BTB

转移目标地址	转移条件判定	转移指令地址
BTA_1	T/NT	BIA_1
BTA_2	T/NT	BIA_2
BTA_3	T/NT	BIA_3
……	……	……

*BIA: Branch Instruction Address
*BTA: Branch Target Address
*T: Taken
*NT: Not Taken

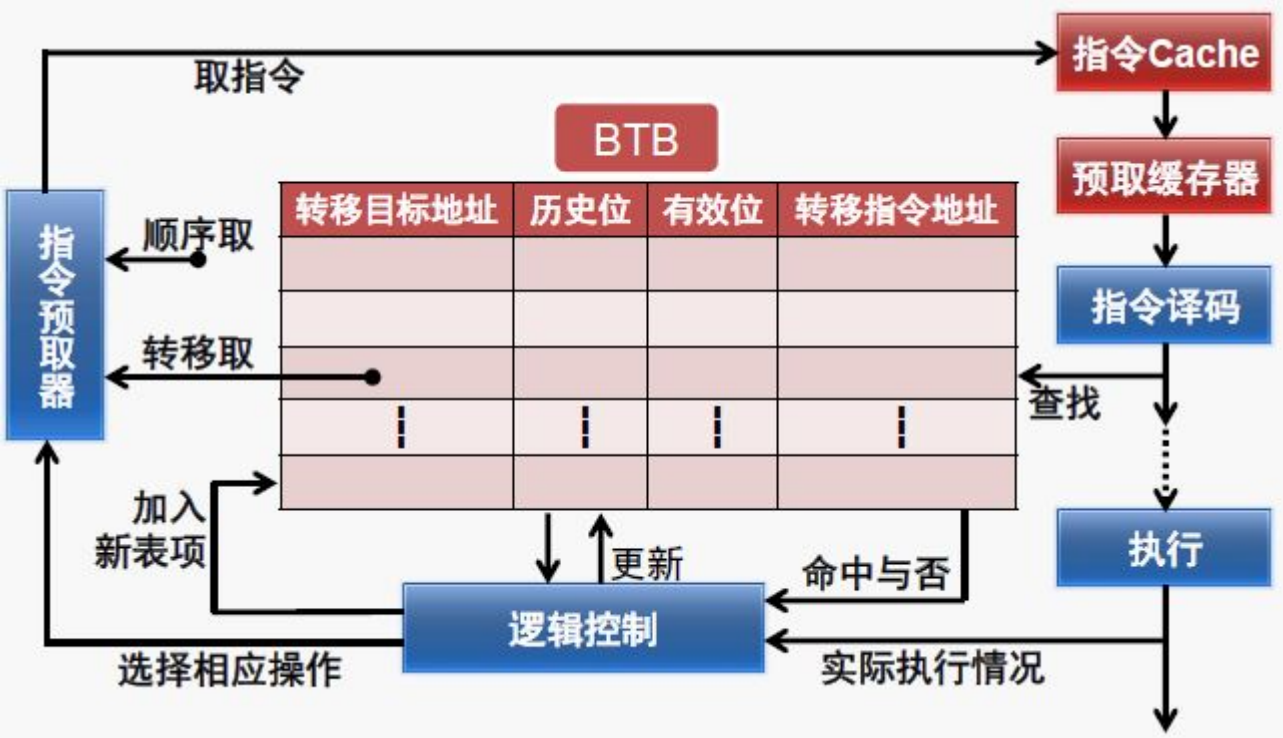
BTB的基本操作有：分配BTB表项——当某条转移指令第一次执行时在BTB中分配一个表项，指令地址保存在转移指令地址域，转移目标地址保存在转移目标地址域；

BTB表项比较——将需要预测的指令地址与BTB的转移指令地址域比较，如果有某一项匹配，则BTB命中；

产生转移目标地址——如果BTB命中，且转移条件判定为T，则将转移目标地址域作为下一条指令地址；

更新BTB——转移指令最终得到的目标地址和条件判定结果与预测结果进行比较，如果不一致，则根据实际结果更新BTB。

Pentium BTB工作机制如下图。在译码阶段检查指令是否为转移指令，如果是则在BTB中进行查找；如果BTB命中则根据历史位预测是否发生转移，如果预测转移，则将转移目标地址提交给指令预取器，“转移取”，如果预测不发生转移，则顺序取下一条指令提交给指令预取器，“顺序取”；如果BTB未命中则预测不转移，顺序取，假如指令实际没有转移则无其他操作，否则指令实际发生转移，预测错误，在BTB中新建表项并将历史位设定为11。



RET指令无需判定转移条件，其转移目标地址已经生成，并且RET指令必须与CALL成对出现，因而是可以预期的；转移目标地址会在流水线晚期的访存阶段才能被获得，这就导致流水线必须停顿。

为了解决RET指令的停顿问题，引入返回地址栈RAS，RAS在CPU中，CALL压栈，RET弹栈，弹栈内容提交给取指部件，直接完成下一条指令的取指，避免停顿。