计算机组成原理笔记

陈鸿峥

2018.12 *

目录

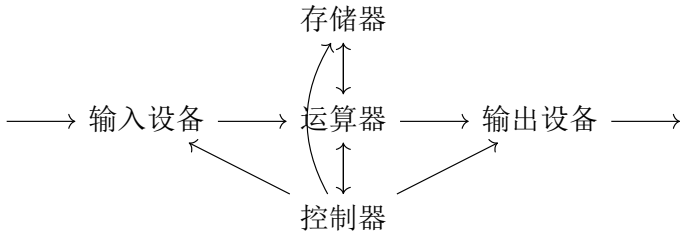
*Build 20181223

本课程使用的教材为David A.Patterson (UCB), John L.Hennessy (Stanford), 《计算机组成与设计（硬件软件接口）》.

1 计算机历史

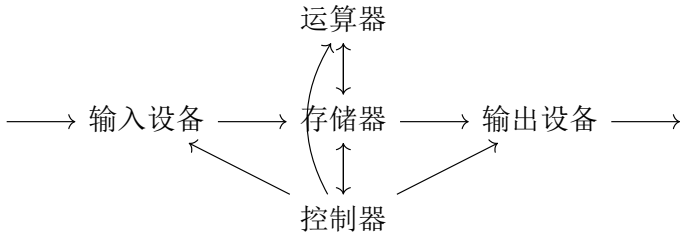
1.1 计算模型

- 图灵机(1936)
- 冯诺依曼体系结构(1945)[1] — 存储程序原理（**运算器**为中心）
  计算机采用二进制表示机器指令和数据，按照程序指令**顺序**执行



而现在由于计算不是瓶颈，存储访问成为了瓶颈，故现代微机以**存储器**为中心

[1]非冯诺依曼体系结构：并行计算、量子计算、生物计算



1.2 物理器件与大规模集成

1904	弗莱明(Fleming)	二极管	
1907	德福雷斯特(De Forest)	三极管	
1938	香农(Shannon)	布尔代数与二值电子器件	奠定数字电路基石
1946		第一台通用计算机ENIAC	
1947	布莱顿(Brattain) 巴丁(Bardeen)	点接触晶体管	
1949	肖克利(Shockley)	结型晶体管(1949)	1956诺贝尔奖
1950		二进制和存储程序EDVAC	实现冯诺依曼设想
1958	Jack Kilby	集成电路	2000诺贝尔奖
1965	Moore	摩尔定律	在价格不变的情况下，每18个月芯片上晶体管数目翻倍，性能也提升一倍
1971	Intel Co	第一款微处理器4004	

1.3 计算机的发展

1.3.1 单处理器(1971-2002)

命题 1 (安迪-比尔定律). *Andy gives, Bill takes away.* 安迪是原*Intel CEO*，比尔是原微软*CEO*，硬件厂商靠软件开发商用光自己提供的硬件资源得以生存

性能提升主要手段

- 提升工作主频(KHz→GHz)：生产工艺不断进步，流水线技术
- 发掘指令级并行(ILP)

但遇到频率墙和功耗墙

$$\text{功耗(power)} \propto 1/2 \times \text{CMOS电容} \times \text{电压}^2 \times \text{转换(01)频率}$$

2004年，Intel放弃4GHz Pentium4芯片开发，因无法解决散热问题，通过加快主频提升处理器性能的路走到尽头

1.3.2 多核处理器

采用多核处理器不过是将硬件的问题丢到软件[2]

定理 1 (阿姆达尔(Amdahl)定律).

改进后的执行时间 = 受改进影响部分的执行时间/改进提高的倍数 + 不受影响的执行时间

$$S_A = \frac{1}{s + (1-s)/N},$$

对计算机系统的某个部分采用并行优化措施后所获得的计算机性能的提高是有上限的，上限由串行部分所占的比例决定

定理 2 (古斯塔夫森(Gustafson)定律).

$$S_G = (s' + p' \times N)/(s' + p') = N + (1-N) \times s',$$

其中，s'和p'为程序串行部分与可并行化部分在并行系统上执行的时间占总时间的比例，N为处理器数量，简便起见设总时间$s' + p' = 1$

打破Amdahl定律**问题规模不变**的假设，任何足够大的任务都可以被有效地并行化，只要问题规模可扩展，并行所带来的加速比就可以扩展

2 导论

2.1 基本概念

表示计算机通信带宽时

KB(yte)	MB	GB	TB	PB	EB	ZB
10^3	10^6	10^9	10^{12}	10^{15}	10^{18}	10^{21}

表示计算机存储二进制时

KiB(yte)	MiB	GiB	TiB
2^{10}	2^{20}	2^{30}	2^{40}

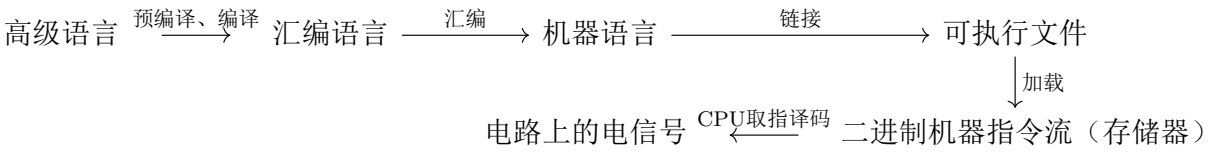
- 位(bit/b)：计算机处理、存储、传输信息的最小单位
- 字节(Byte/B) 1 Byte = 8 bit：现代计算机主存按字节编制，字节是最小可寻址单位
- 字(Word)：表示被处理信息的单位，用来度量数据类型的宽度[3]

一台32位的电脑，一个字等于4个字节，字长为32位；若字长为16位，则一个字等于2字节.

[2] “向多核的转变并不是因为我们在软件或体系结构技术上取得了中大突破而带来的。相反，这种转变是当单处理器体系结构发展遇到了难以克服的巨大障碍时，我们被迫作出的一种选择。” —Kurt Keutzer (UCB), *The Landscape of Parallel Computing Research: A View from Berkeley*

[3]字长是指CPU中数据通路的宽度，等于CPU内部总线的宽度或运算器的位数或通用寄存器的宽度；字和字长的宽度可以一样，也可以不同，通常是字节的整数倍

2.2 程序到电子信号



计算机内部工作过程：逐条执行加载到内存中的二进制机器指令流的过程指令执行分为两个阶段，周期性重复性进行：

- 取指阶段：CPU从内存中读取指令，程序计数器(PC)保存要被要被取出的**下一条**指令的地址，除非遇跳转指令，否则都加一个增量[4]
- 执行阶段：对取出的指令译码后执行

2.3 功能部件

2.3.1 中央处理器(CPU)

核心组成：

- 控制单元(Control unit)：对指令进行译码，产生控制信号
- 数据通路(Datapath)：完成指令执行，ALU+寄存器

寄存器分类：

- 通用寄存器(GRS, General Register Set)：存放操作数和中间结果
- 程序计数器(PC, Program Counter)：存放下条要执行的指令
- 指令寄存器(IR, Instruction Register)：存放当前指令

2.3.2 存储器

层次化结构(Hierarchies)

- 内存(Primary)：高速缓存(Cache)、主存(MM, Main Memory)
- 外存(Secondary)：磁盘、光盘、闪存等

2.4 计算机结构的八个想法

1. 摩尔(Moore)定律：集成电路资源每18 – 24个月翻倍
2. 抽象(abstraction)：简化设计
3. 加速常用操作(Make common case fast)：见定理1
4. 并行(parallelism)
5. 流水线(pipelining)
6. 预测(prediction)

[4]程序计数器（Program Counter）是一个实际存在的寄存器吗？ - Belleve的回答 - 知乎 https://www.zhihu.com/question/22609253/answer/21965180 PC每次增加**一条指令的长度/寻址粒度**，在MIPS中一条指令长4字节，寻址粒度1字节，故每次PC加4；而x86体系指令长度不定，每次增加量会变化

7. 内存等级制(hierarchy)
8. 冗余实现可靠性(redundancy)：检测故障及解决

2.5 性能评价

$$\text{计算机的性能(Performance)} = 1/\text{执行时间(Execution time)}$$

按照单位（量纲）进行换算即可

$$\begin{aligned}\text{CPU执行时间(s)} &= \text{执行程序所需CPU时钟周期(cyc)} \times \text{时钟周期s/cyc)} \\ &= \text{指令数目(ins)} \times \text{CPI(cyc/ins)} \times \text{时钟周期(s/cyc)}\end{aligned}$$

程序性能对执行事件的影响：

- 算法、编程语言、编译器都对指令数和CPI产生直接影响
- 指令集体系结构则同时对指令数、CPI、时钟频率产生影响

3 指令系统

3.1 概述

复杂指令集计算机	精简指令集计算机
CISC, Complex Instruction Set Computer	RISC, Reduce Instruction Set Computer
出现较早，大而全	小而精
指令周期长，专用寄存器，微程序控制，难编译优化生成高效目标代码，效率低(二八定律，简单指令使用频率高)	指令周期短，大量通用寄存器，组合逻辑电路控制，优化编译系统
变长指令字	定长指令字
借鉴思想	现在多用
x86	ARM, MIPS,SPARC

3.2 指令格式

指令一般由**操作码**和**地址码**（包括操作数和寻址方式）决定

3.2.1 操作码设计

- 关注程序代码长度时：变长指令字、变长操作码
- 关注性能时：定长指令字、定长操作码

扩展操作码：使用频率高的指令用短的操作码，频率低的指令用长的操作码

- 零地址指令：空操作、停机、堆栈

- 一地址指令：取反、取负、累加器
- 二地址指令（最常用）：分别存放双目运算中两个源操作数地址，并将其中一个地址作为结果地址
- 三地址指令（RISC）：双目运算中两个源操作数地址和一个结果地址

例 1. *36*位长的指令系统，设计一个扩展操作码，使之能表示以下指令

- *7*条两个*14*位地址和一个*5*位地址的指令
- *600*条一个*14*位地址和一个*5*位地址的指令
- *100*条无地址指令

分析. 各指令的范围如下

- 三地址指令：$36 = 3(op) + 14(ad1) + 14(ad2) + 5(ad3)$
  $000 \sim 110\,((0)_{10} \sim (6)_{10})$共*7*条，111为扩展码
- 二地址指令：$36 = 3 + 14(op) + 14(ad1) + 5(ad2)$
  $00,0000,0000,0000 \sim 00,0010,0101,0111\,((0)_{10} \sim (599)_{10})$共*600*条，$111,00,0010,0101,1000$为拓展码
- 零地址指令：$36 = 3 + 14(op) + 12 + 7$
  最后*7*位$000,0000 \sim 110,0011\,((0)_{10} \sim (99)_{10})$共*100*条

3.2.2 寻址方式

通常特指操作数寻址（对应的是指令寻址，PC增值和跳转），目的是扩大访存范围，提高访问数据的灵活性和有效性

立即数寻址	直接给出操作数本身，无需访存快速，操作数大小受地址字段长度限制，大量使用	MOV AX, 1000H
存储器直接寻址	操作数在存储器中，直接给出操作数在存储器中的地址，寻址空间受指令地址字段长度限制，较少使用	MOV AX, [1000H]
存储器间接寻址	存储器中的内容是操作数的地址，需二次寻址	
寄存器直接寻址	直接给出寄存器编号，无需访存速度快，地址范围有限，可用通用寄存器较少，使用最多，提高性能常用手段	MOV AX, BX
寄存器间接寻址	寄存器中的内容是操作数的地址，二次寻址	MOV AX, [BX]
相对寻址（偏移）	相对当前指令(PC)位移量为A的单元	EA=(PC)+A
基址寻址（偏移）	相对基址(B)位移量为A的单元	EA=(B)+A
变址寻址（偏移）	相对形式地址A（数组基址）位移量为(I)的单元	EA=(I)+A, I=(I)±X
堆栈寻址	从寄存器到堆栈或反过来，指令短	EA=栈顶
复合寻址	间接寻址+相对/变址寻址	间接相对EA=(PC)+A, 相对间接EA=((PC)+A)

寻址方式的确定

- 操作码中给定寻址方式：MIPS
- 专门寻址方式：x86（0-1字节）

3.3 数据表示

3.3.1 进制(system)

二进制(binary)、八进制(octonary)、十进制(decimal)、十六进制(hexadecimal)

- 十进制转二进制：整数部分除以2取余，小数部分乘2取整
- 二进制转十六进制：四位四位统计

3.3.2 符号数

1. 二进制
   - 符号数值(sign-magnitude)形式（原码）：首位0为正数，1为负数，将符号位一起考虑有以下表示

$$A=\begin{cases}A & A\in[0,2^{n-1})\\ 2^{n-1}-A & A\in(-2^{n-1},0]\end{cases}$$

- 反码(1's complement)$\sim A$：除符号位不变，其他位取反

$$\sim A = \begin{cases} A & A \in [0, 2^{n-1}) \\ (2^n - 1) + A & A \in (-2^{n-1}, 0] \end{cases}$$

分析. 反码是全*1*的补数

$$\sim A = (2^n - 1) - A = (11\ldots1)_2 - A_2$$

即在mod $2^n - 1$意义下的运算

- 补码(2's complement)$[A]_c$：反码+1，按照原来十进制转二进制方法即可得对应有符号十进制数，由于没有正负0，故表示的数多了一位，补码的补码为原码

$$[A]_c = \begin{cases} A & A \in [0, 2^{n-1}) \\ 2^n + A & A \in [-2^{n-1}, 0) \end{cases}$$

分析. 补码的设计非常关键，理解补码的由来对于后面的四则运算有着很大帮助。之所以要有补码，是因为希望能做到**减去一个数等于加上某个数**，而这在模2^n的意义下即可实现。
那么就有

$$[A]_c = 2^n - A = ((2^n - 1) - A) + 1 = \sim A + 1$$

即在mod 2^n意义下的运算，以*4*位二进制为例

$$(5)_{10} = (0101)_2 \implies (5)_c = 2^4 - 5 = 11 = (10000)_2 - (0101)_2 = (1011)_2$$

- 移码(bias)$[A]_b$：补码的符号位取反，引入目的是保证浮点数的机器零

$$[A]_b = A + 2^{n-1}, A \in (-2^{n-1}, 2^{n-1})$$

分析. 相当于把正数移到负数的部分，负数移到正数的部分

2. 十进制：用ASCII码或BCD码表示

3.3.3 小数表示

- 定点数，首位符号位
  - 定点整数：小数点固定在最低位右边，$0 \le |x| \le 2^n - 1$
  - 定点小数：小数点固定在数值部分最高位的左边，$0 \le |x| \le 1 - 2^{-n}$
- 浮点数(IEEE 754)

符号S,1	阶码E,8,移码	尾数F,23,原码

移码偏置常数为127（单精度）、1023（双精度）

$$(-1)^S \times 1.F \times 2^{E-127}$$

例 2.

$$1\ 0110\ 1001\ 0001 = 1.0110\ 1001\ 0001 \times 2^{(12)_{10}}$$

指数：$12 + 127 = 139 \rightarrow 1000\ 1011$

尾数：011 0100 1000 1000 0000 0000 左对齐，因为有小数点

符号S	指数E(exponent)	尾数F(mantissa)
0	1000 1011	011 0100 1000 1000 0000 0000
1位	8位	23位

特殊值表示

阶码（移码）	尾数	数据类型
$1 \sim 254$	任何值	规格化数
0	0	0
0	非零数	非规格化数
255	0	$+\infty / -\infty$
255	非零数	NAN, Not A Number

单精度可表示范围$[10^{-38}, 10^{+38}]$，双精度$[10^{-308}, 10^{+308}]$

3.3.4 C语言数据类型

C语言中数据类型大小以字节为单位

声明	数据长度(32位机，Byte)
char	1
short	2
int	4
long	4
float	4
double	8

3.4 数据存储

从80年代开始，几乎所有机器都采用字节编址（4字节32位二进制8位十六进制）

- 大端方式(Big Endian)：最高有效位(MSB)所在地址为数的地址，MIPS，Photoshop、JPEG
- 小端方式(Little Endian)：最低有效位(LSB)所在地址为数的地址，x86，GIF、RTF

字节交换：大端小端互换

数据边界对齐：减少访存次数，按字地址对齐（4的倍数，二进制后两位为0）

3.5 数据纠错

冗余校验：奇偶校验码

3.6 MIPS指令系统

所有指令都是32位宽，按字地址对齐

表 1: 三种指令格式

R-Type	用于寄存器 sub rd,rs,rt	寄存器寻址	$32 = 6(op) + 5(rs) + 5(rt) + 5(rd) + 5(shamt) + 6(funct)$
I-Type	运算指令： ori rt,rs,imm16 存储指令： lw rt,rs,imm16 条件分支： beq rs,rt,imm16	立即数、基址(PC)寻址	$32 = 6(op) + 5(rs) + 5(rt) + 16(\text{immediate})$
J-Type	无条件跳转 j target	按字对齐的伪直接寻址	$32 = 6(op) + 26(\text{target address})$

注：J-Type中伪直接寻址含义：PC高4位拼上26位直接目标地址，最后添2个0（相当于乘4）为32位目标地址

指令字段含义

- 操作码(op)
- 第一个源操作数寄存器(rs)，5位32个
- 第二个源操作数寄存器(rt)
- 结果寄存器(rd)
- 移位指令的位移量(shamt)

操作码的不同编码定义了不同的含义，若操作码相同时，再用不同功能码区分

3.7 MIPS语法表

三个特殊寄存器：HI、LO、PC

表 2: 通用寄存器

寄存器	名称	用途
$0	$zero	常量0
$1	$at	保留给汇编器
$2-$3	$v0-$v1	函数调用返回值
$4-$7	$a0-$a3	函数调用参数
$8-$15	$t0-$t7	临时变量
$16-$23	$s0-$s7	保存(saved)
$24-$25	$t8-$t9	其他临时变量
$26-$27	$k0-$k1	为OS保留
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	函数调用返回地址(return address)

存储器数据指定：

- 32位机→可访问空间 $2^{32}bytes = 4GB$
- 大端方式
- 访存地址通过一个32位寄存器内容加16位偏移量得到
- 16位偏移量为有符号整数（补码表示）
- 按边界对齐

表 3: MIPS指令

算术	加法	add $s1,$s2,$s3	$s1=$s2+$s3
	立即数加法	addi $s1,$s2,20	$s1=$s2+20
	乘法	mult $s2,$s3	HI,LO=$s2 $s3
	除法	div $s2,$s3	LO=$s2/$s3,HI=$s2 mod $s3
	从高位移出	mfhi $s1	$s1=HI
	从低位移出	mflo $s1	$s1=LO
逻辑	与	and $s1,$s2,$s3	$s1=$s2&$s3
	立即与	andi $s1,$s2,20	$s1=$s2&20
	左移	sll $s1,$s2,20	$s1=$s2<<20
传输	存字	sw $s1,500($s2)	($s2+500)=$s1
	存半字	sh $s1,500($s2)	($s2+500)=$s2
	存位	sb $s1,500($s2)	($s2+500)=$s3
	读字	lw $s1,500($s2)	$s1=($s2+500)
条件分支	分支等于	beq $s1,$s2,25	if ($s1==$s2) goto PC+4+100
	分支不等	bne $s1,$s2,25	if ($s1!=$s2) goto PC+4+100
	分支小于	slt $s1,$s2,$s3	if ($s2<$s3) $s1=1;else $s1=0
跳转	跳转	j 10000	goto 10000
	switch/函数返回	jr $ra	goto $ra
	函数调用	jal 10000	$ra=PC+4;goto 10000

3.8 标志位

NF(SF)	negative	符号
OF(VF)	overflow	溢出
CF	carry	进借位
ZF	zero	零

- 进/借位CF：**无符号数**运算结果是否超出范围，即使超出结果仍对
- 溢出OF：**有符号数**运算结果是否超出范围，若超出则结果不对

4 计算机的运算

4.1 基本运算

4.1.1 位运算

位运算针对二进制数，逻辑运算针对表达式的值

无符号数	逻辑左移	高位移出，低位补0
	逻辑右移	低位移出，高位补0
有符号数	算术左移	高位移出，低位补0
	算术右移	低位移出，高位补**符号位**

移位符号<<和>>不区分算术还是逻辑移位，只由参与运算的数值决定
算术左移溢出判断：若移出的位不等于新的符号位，即$CF \oplus SF = 1$，则溢出

4.1.2 位扩展和位截断

- 位扩展：如`float`变`double`；数据存入寄存器时也要扩展，`lb`
  - 无符号数：0扩展，即前面补0
  - 有符号整数：符号扩展，即前面补符号
- 位截断：如`double`变`int`；强行丢弃长数的高位，可能溢出或数据不正确

4.1.3 加减法

原码	补码	移码
符号与数值单独运算	符号与数值一起运算	符号与数值一起运算
同号相加进位溢出	变形两位补码 01正溢出，10负溢出	两加数和和数符号都相同则溢出
$A \pm B = A \pm B$	$[A+B]_c = [A]_c + [B]_c$ $[A-B]_c = [A]_c + [-B]_c$	$[A]_b + [B]_b = [A+B]_c$ $[A]_b - [B]_b = [A]_b + [-[B]_b]_c = [A-B]_c$
浮点数尾数	定点数	浮点数阶码

注意进位和溢出的区别，在模2^n意义下，加负数进位相当于回到原点；而只有同号运算才可能溢出
C不考虑溢出的异常处理

- unsigned整型溢出：模运算
- signed整型溢出：undefined behavior
- MIPS上的C编译器会选用无符号的算术运算指令，如`addu`、`addiu`、`subu`

全加器实现

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$
$$C_i = (A_i \oplus B_i)C_{i-1} + A_iB_i$$

并行加法器：

- 若$A_iB_i = 1$，则$C_{\text{out}} = 1$与C_{in}无关（进位生成）
- 若$A_i + B_i = 1$，则$C_{\text{out}} = 1$与C_{in}相同（进位传递）

组内并行，组间串或并行
BCLA(成组先行进位)芯片

4.1.4 乘法

4.1.5 除法

5 处理器

5.1 处理器概述

计算机五大组成部分：[控制器+数据通路（运算器）]处理器、存储器、输入、输出

- 操作元件：组合逻辑电路，所有操作元件都必须从状态元件接受输入，并并将输出写入状态元件
- 状态元件：时序逻辑电路，只有状态元件可以存储信息

定义 1 (寄存器组(Register File))**.** 包含

1. 两个读端口（组合逻辑）：*busA*和*busB*读入地址，经过一个取数时间*(AccessTime)*后，两条线有效
2. 一个写端口（时序逻辑）：写使能为*1*且时钟边沿到达

要经过一个*clk-to-Q(*门闩延迟*)*，输入信号在寄存器的输出端才有效

一个时钟周期就是一个节拍

5.2 数据通路的建立

CPU执行指令主要分为两个阶段

1. 取指阶段（公共操作）：
   - 取指令
   - $PC \leftarrow PC + \Delta = PC + 4$
   - 译码
2. 执行阶段：
   - 主存地址运算
   - 取操作数
   - 算术逻辑运算
   - 存结果
   - 判断检测异常事件
   - 若有异常，则自动切换到异常处理程序
   - 检测是否有中断请求，有则转中断处理

MIPS中三类基本指令：R-type、I-type、J-type
七条指令

1. 加减 `add/sub rd,rs,rt`
2. 或立即数 `ori rt,rs,imm16`：零扩展 立即数需要零扩展(ZeroExt)为32位
3. 存 `lw rt,rs,imm16`：符号扩展
4. 取 `sw rt,rs,imm16`：符号扩展

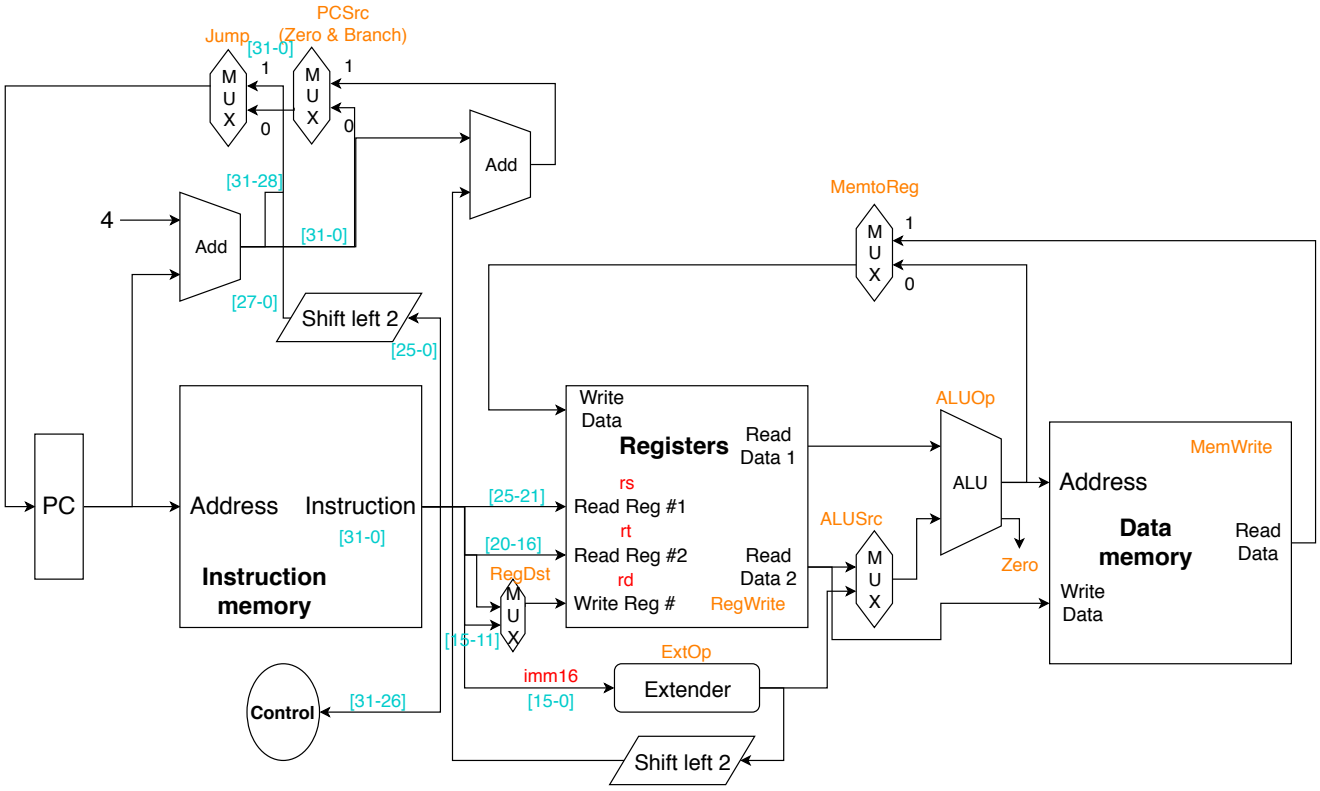


图 1: MIPS基本数据通路

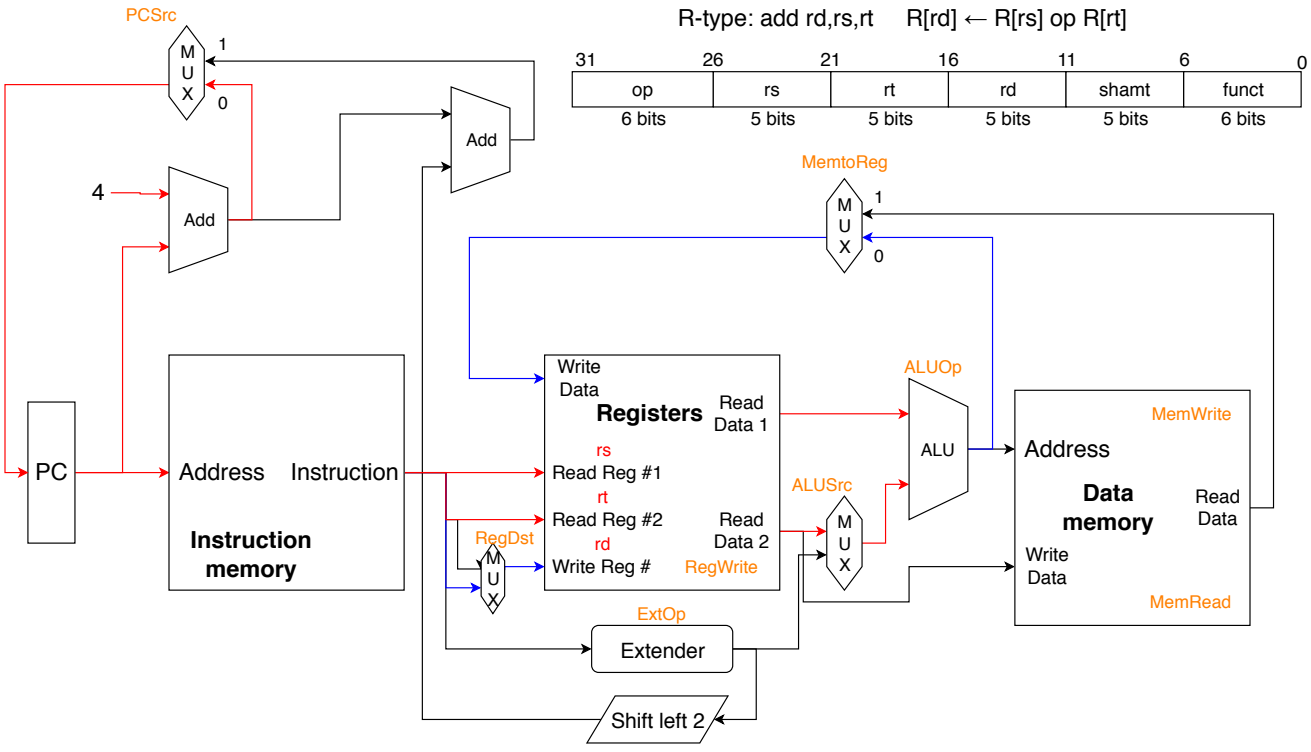


图 2: add/sub通路

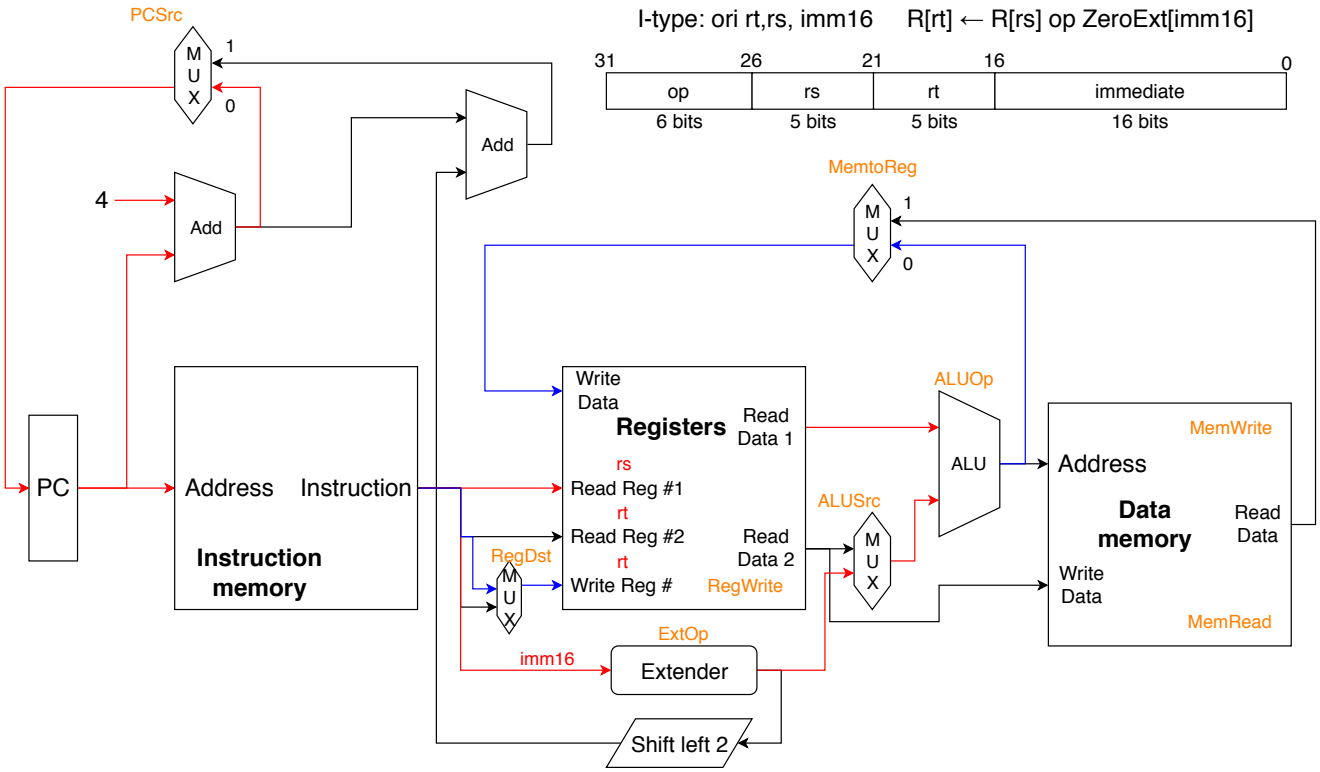


图 3: ori通路

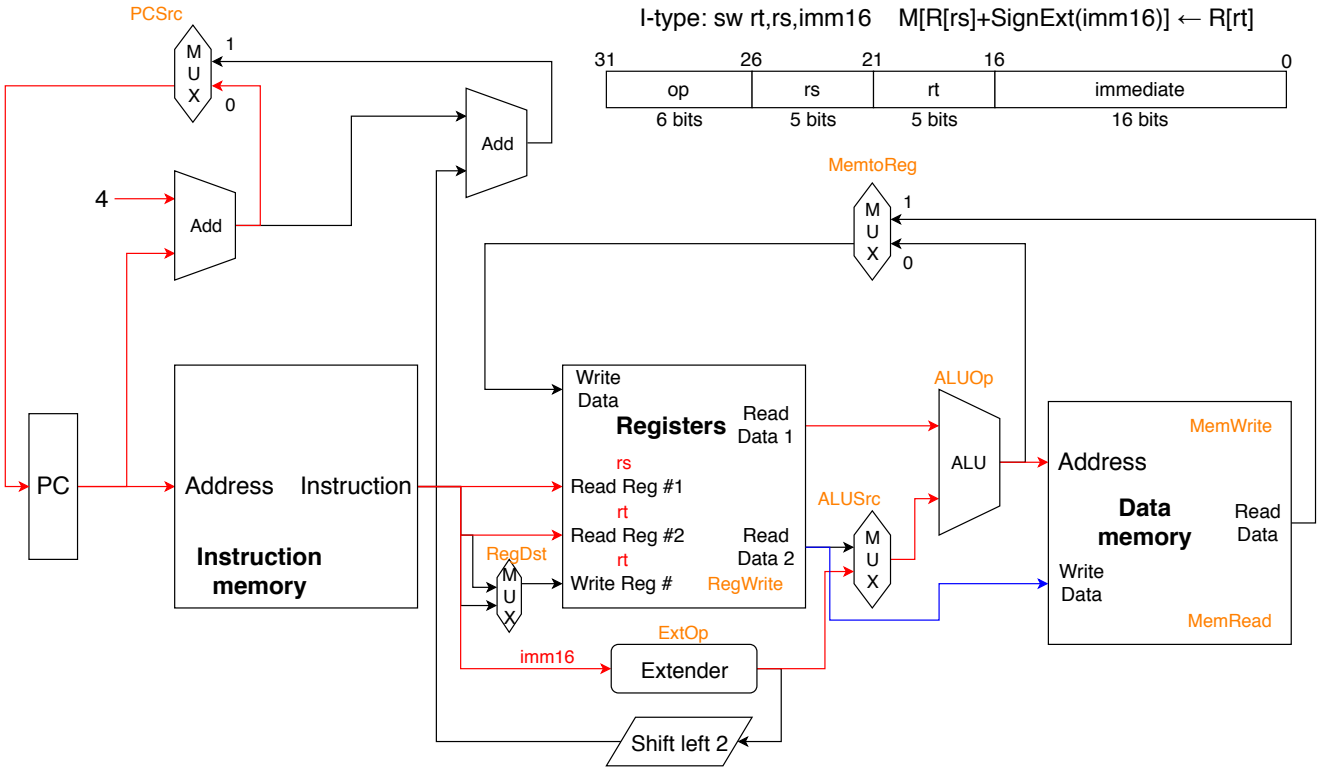


图 4: sw通路

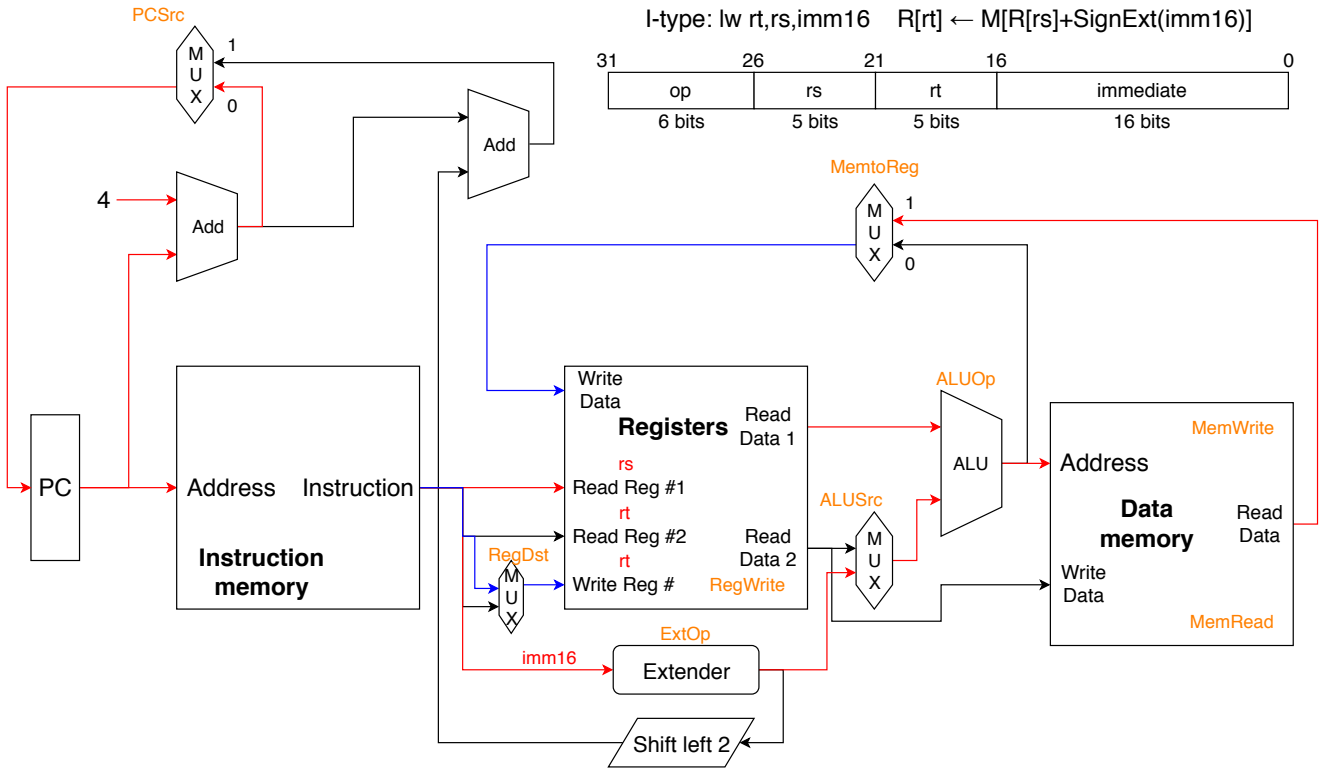


图 5: lw通路

5. 分支 `beq rs,rt,imm16`：PC只需30位，因每次加4
6. 跳转 `j target`：PC+4的高四位串接26位立即数然后左移2位

5.3 多周期

五个阶段：取指(IF)、译码(ID)、执行(EXE)、访存(MEM)、写回(WB)

- 有限状态机：通过组合逻辑硬连线(PLA)实现
- 微程序：用ROM存放微程序实现

5.4 流水线

提升工作主频：
减少每个流水级执行时间→减少每个流水级的任务量→任务再分解→增加流水线级数
副作用：

- 寄存器开销(overhead)：收益下降
- 非均匀延迟(Nonuniform delays)：吞吐率受限于最慢栈的时间，但很难将ALU和存储器划分成更小的栈

单个任务执行时间没有缩短，但是总的吞吐率增加了
时钟周期等于最长阶段花费时间t，N条指令执行时间$(5+N-1)\times t$

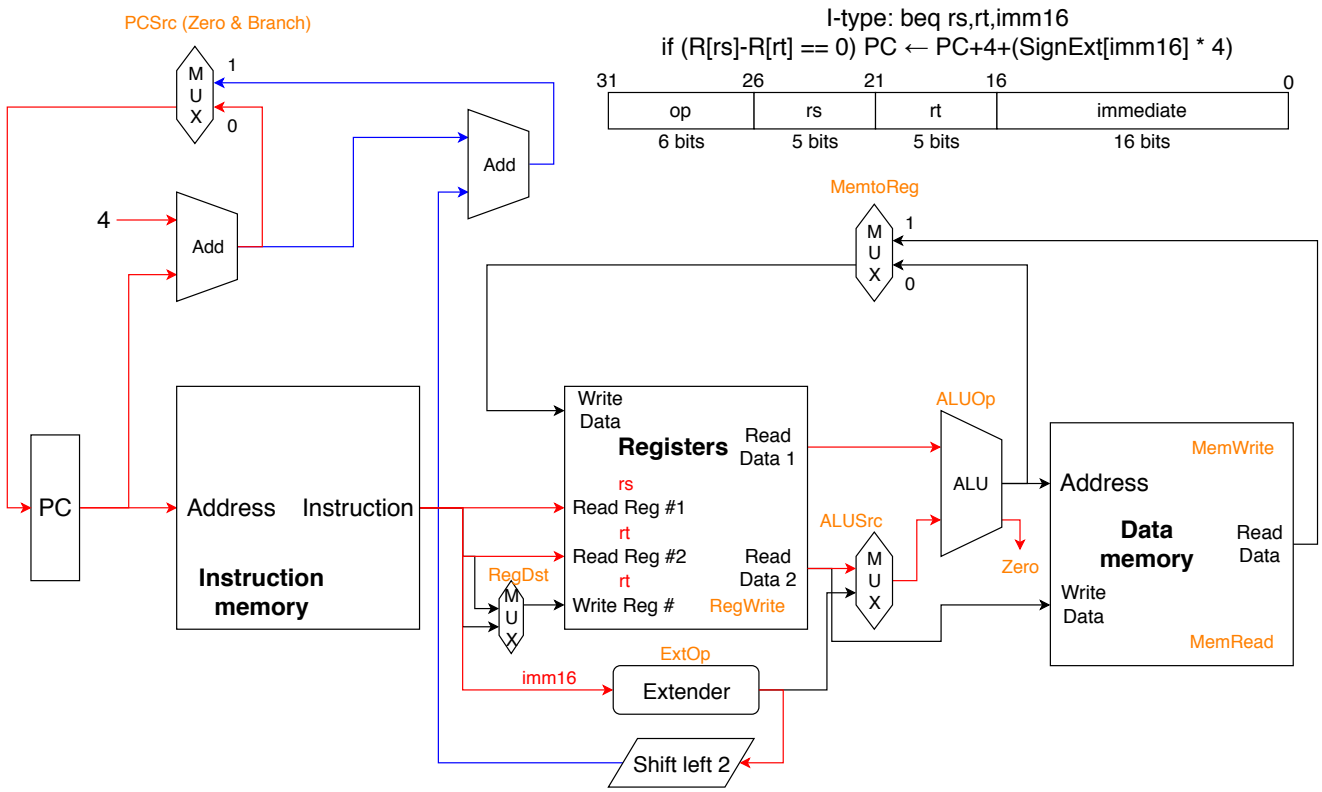


图 6: beq通路

利于流水线执行的指令集

- 指令长度一致：简化取指和指令译码
- 指令格式少，且源寄存器位置相同：利于在指令未知时预取操作数
- 只有load/store指令才能访问存储器，利于减少操作步骤，规整流水线
- 数据和指令在内存中**对齐**存放，利于减少访存次数和流水线规整

5.4.1 冒险(Harzard)

- 结构冒险/资源冲突：一个功能部件同时被多条指令使用产生，如Load和R-type同时要写回
  - 通过加空操作(NOP)延迟写操作（每条指令都有五个阶段）
  - 设置多个部件（比如多个端口、寄存器读写口分开），避免冲突
- 控制冒险/分支冒险/转移冒险：在jump/beq之前已有几条指令被取出
  - 静态分支预测：总是预测条件不满足，或加启发式规则
  - 动态分支预测：根据历史情况进行调整（微型强化学习）
  - 指令静态调度：编译优化指令顺序，实现分支延迟
- 数据冒险/数据相关：写后读

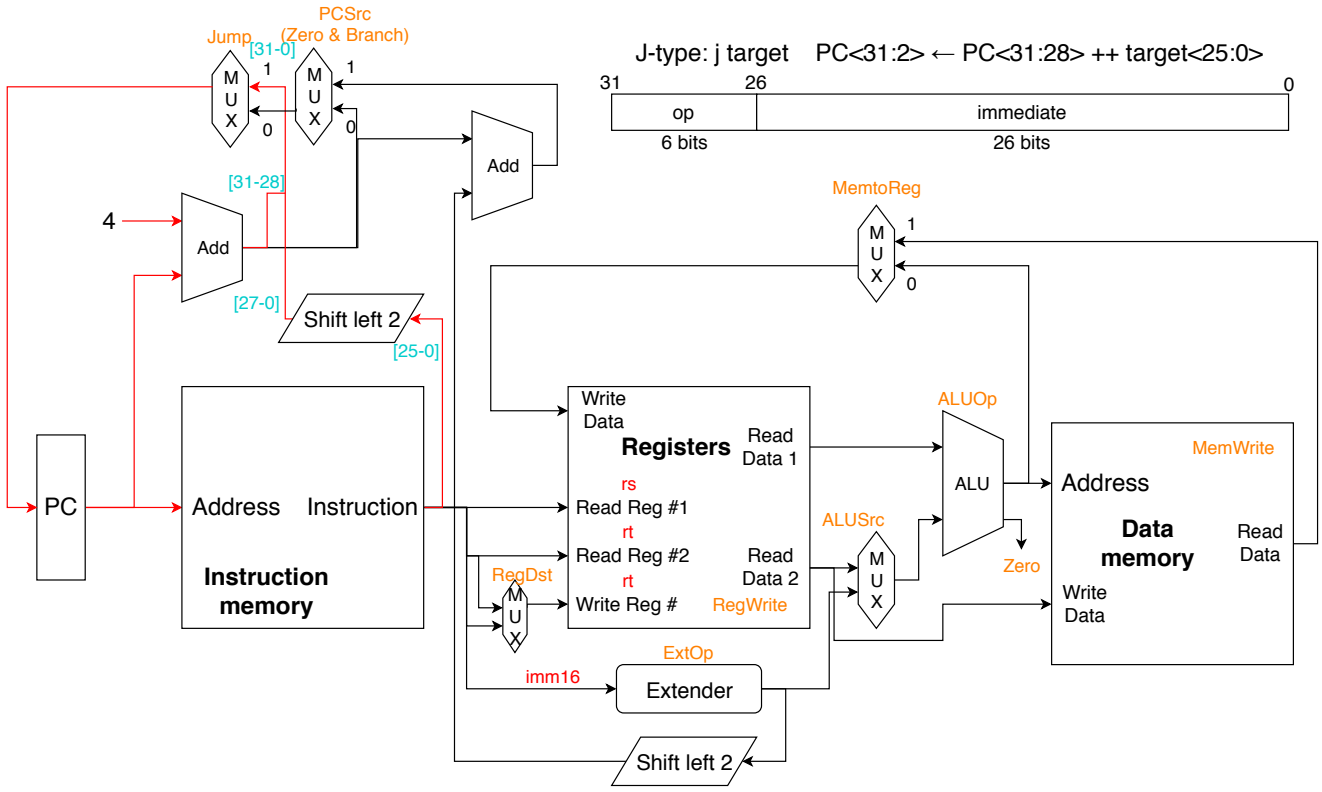


图 7: jump通路

– 转发(forwarding/bypassing)：将数据从流水段寄存器中直接渠道ALU的输入端，如果在Data Memory读出则无法转发(Load-use数据冒险)

– 阻塞(stall)：插入Bubble或插入NOP

– 静态指令调度：编译优化指令顺序，拉大具有数据冒险指令的距离，减少流水线可能产生的停顿（可以解决load-use）；也即先把后面无关的操作调到前面来执行；或者说利用闲置资源先干后面的事情（乱序执行）

6 存储器的层次结构

6.1 概述

存储器按照存储方式可分为以下几种：

1. 随机访问存储器(RAM)
   存储器任意单元可随时访问且访问所需时间相同

   - 静态(SRAM)：cache
     触发器；只要加电源，信息就能一直保持；集成度低，引脚多，速度快
   - 动态(DRAM)：主存
     电容；每隔一段时间必须刷新；现在一般用DDR3 SDRAM(Double Data-Rate Synchronous)

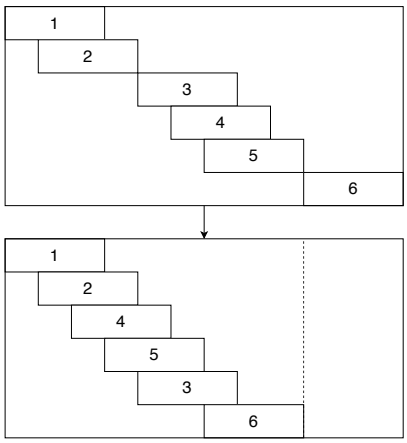


图 8: 编译器调度

2. 只读存储器(ROM)：BIOS(Basic Input Output System)
   正常工作时只读，能随机读出，不能随机写入
   - MROM：只读
   - PROM：一次写
   - EPROM/EEPROM：多次改写
3. 相联存储器(Content Addressed Memory, CAM)：快表(TLB)
   按内容检索到存储位置进行读写
4. 直接存取存储器(DAS)：磁盘
   可以直接定位到要读写的数据块，存取时间的长短与数据所在位置有关
5. 顺序存储器(SAS)：磁带
   数据按顺序从存储载体的始端读出或写入，存取时间的长短与数据所在位置有关

按照信息的可保存性分为：

- 断电后数据是否丢失
  - 挥发性(volatile)/易失存储器：SRAM、DRAM
  - 非挥发性/非易失存储器(NVM)：ROM、磁盘、闪存
- 读出后是否保存数据
  - 破坏性存储器（读出原信息被破坏，需重写）：DRAM
  - 非破坏性存储器：SRAM

地址译码两种方式

1. 线选法（一位地址译码）
   SRAM，只在单方向译码，同时读出一条字线上的所有位，k位地址对应2^k地址驱动线
2. 位片式（二维双译码）
   k位地址对应$2^{\frac{k}{2}}+2^{\frac{k}{2}}$条地址驱动线

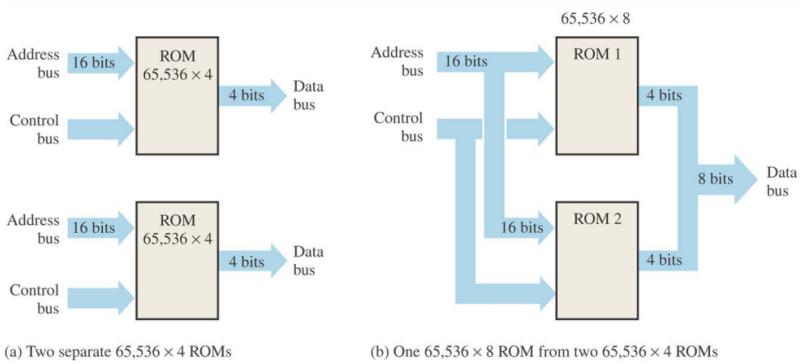
主存空间的划分

1. ROM区：存放系统程序、标准子程序
2. RAM区：存放用户程序

6.2 存储容量扩展

6.2.1 位扩展

存储芯片($mk \times n$位/片)构成存储器($mk \times N$位)，需要$\lceil N/n \rceil$片
字数不变（存储单元个数不变），位数扩展（字长加长）
地址线及读写控制线相连接，数据线单独引出，不需外部译码器

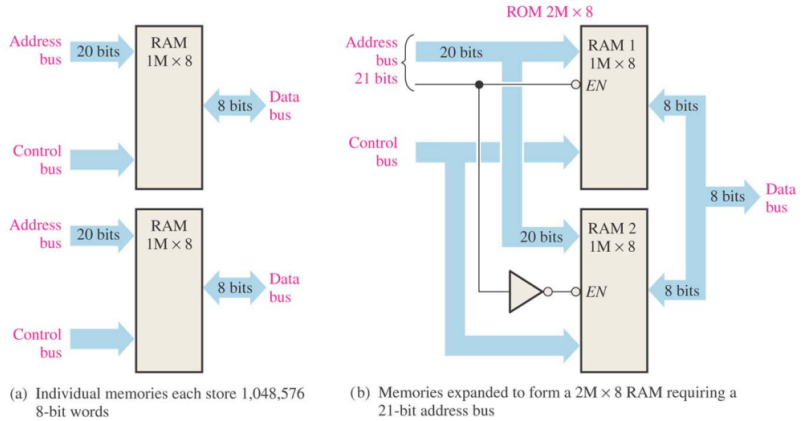


(a) Two separate 65,536 × 4 ROMs

(b) One 65,536 × 8 ROM from two 65,536 × 4 ROMs

6.2.2 字扩展

存储芯片($mk \times n$位/片)构成存储器($Mk \times n$位)，需要$\lceil M/m \rceil$片
位数不变（字长不变），扩充容量（存储单元个数增加）
地址线、读写控制线对应相接，片选信号分别与外部译码器各输出端相连



(a) Individual memories each store 1,048,576 8-bit words

(b) Memories expanded to form a 2M × 8 RAM requiring a 21-bit address bus

6.3 Cache概述

6.3.1 计算机存储层次结构

CPU与cache之间以字为单位传送，cache与主存之间以块为单位传送
程序访问局部性

- 时间(temporal)局部性
- 空间(spatial)局部性

cache对程序员是透明的，即程序员在编写程序时无需了解cache是否存在或如何设置

6.3.2 命中与失效

- 命中(hit)：要访问的信息在cache中
- 失效(miss)：不在cache中

平均访问时间

$$\bar{T} = pT_h + (1-p)(T_h + T_m) = T_h + (1-p)T_m$$

6.4 cache与主存的映射

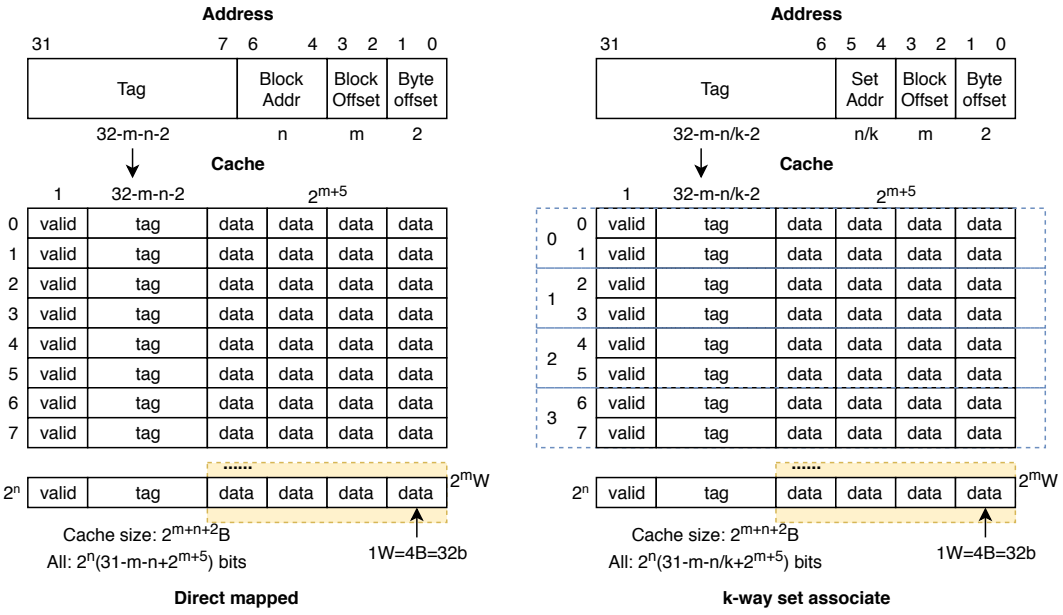
cache存放一个主存块的对应单位为行(line)或块(block)或槽(slock)或项(entry)。
位宽即为cache一个块的大小，如传送单位为512B，则cache每个块大小为512B 频繁的cache替换称为cache抖动

6.4.1 直接(direct)映射

cache内存放的内容是主存的一个子集，因此给出一个主存地址，要在cache中找到这个地址，应该将这个32位地址全部用上。末几位用于确定在cache内存储的位置，高为则用来确定是否为该地址。注意区别块地址、字地址、字节地址等。按字索引，后两位不需要。先计算出块地址（因而先模块内数目），再计算块内地址。

6.4.2 组相联(set associative)映射

直接模组的数目。相联度高，缺失率低，但会增加命中时间只要命中了tag则对应的主存块一定在cache里



6.5 Cache替换算法

- FIFO
- Least Recently Used, LRU
- Random

6.6 Cache一致性

- 写命中(hit)
  - 写直达(write through)：同时写cache和主存
    加入写缓冲(write buffer)，先存入写缓冲，当写主存操作结束后再将写缓冲数据释放
  - 写回(write back)：只写cache不写主存
    每个cache行置一个脏位(dirty bit)，若cache行中的主存块被修改，则置为1；只有当脏位为1的块从cache中替换出去时才将其写回主存
- 写不命中(miss)
  - 写分配(allocate-on-miss)：更新主存块相应单元，再将该主存块装入cache（空间局部性）
  - 写不分配(no-allocate-on-write)：直接写主存，不放回

通常写直达+写分配/写不分配，写回+写分配

6.7 多级Cache

一般L1 Cache为分立Cache（数据指令分开放），减少命中时间获得较短时钟周期
L2 Cache为联合Cache，降低缺失率以减少主存缺失损失
Intel Core i7采用三级Cache

L1	L2	L3
32KB I/ 32 KB D	256KB	2MB/core
4-way I/8-way D	8-way	16-way
Pseudo-LRU	Pseudo-LRU	Pseudo-LRU +ordered selection algorithm

6.8 虚拟存储器

6.8.1 概述

- 将内外存统一管理的存储管理机制，按需调页(demand paging)
- 虚存是主存和磁盘的抽象，OS使每个进程看到的存储空间都一致
- 虚存为每个进程提供一个假象，好像每个进程都独占主存，且主存空间极大

分页(paging)基本思想

- 将内存分为固定长且较小的存储块，每个进程也划分为固定长度的程序块
- 程序块(页/page)可装到存储器可用的存储块(页框/page frame)中

- 无需用连续页框来存放一个进程
- 操作系统为每个程序/进程生成一个页表(page table)
- 通过页表实现逻辑地址到物理地址的转换(address mapping)

逻辑地址与物理地址区别

- 逻辑地址：程序中指令所用的地址
- 物理地址：存放指令或数据实际内存地址

6.8.2 组织方式

页式虚拟存储器，存在主存中

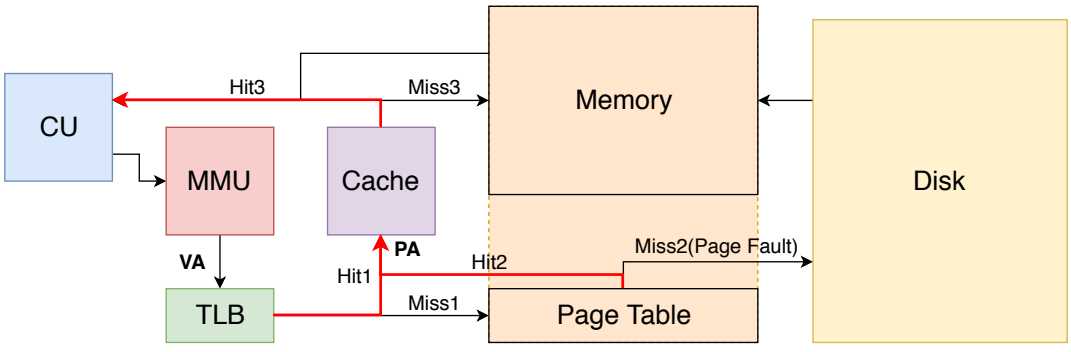
- 指令给出虚拟地址
- 每个页表记录对应的虚页情况
- valid为0说明缺页(page fault)，代价读磁盘，软件处理
- CPU执行指令时，先由MMU将逻辑地址转为物理地址
- 页大小比cache的block大得多，全相联映射
- 写回策略

页表结构

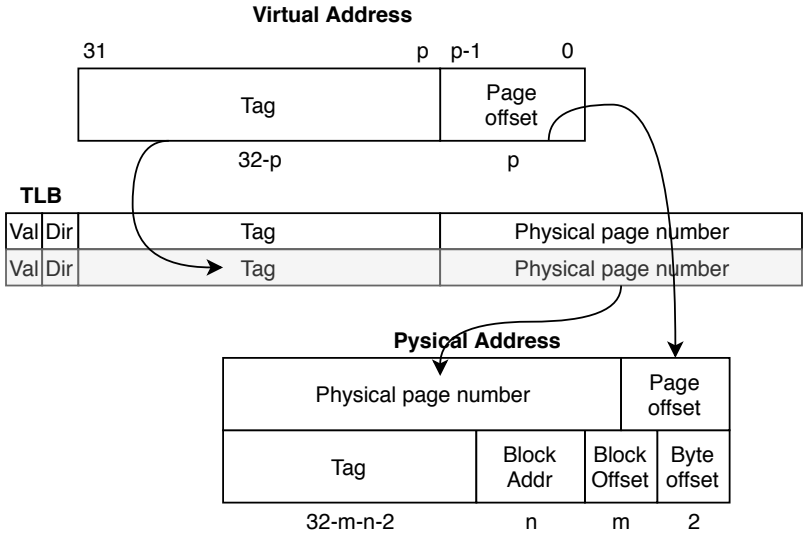
- 每个进程一个页表
- 页表项数由进程大小决定
- 页表在主存的首地址记录在页表

6.8.3 快表(TLB)

转换后备缓冲器(Translation-Lookaside Buffer)，存在cache中
每次读写操作都至少带来两次存储器访问，一次访问页表，一次访问所需的数据或指令
使用cache来存储页表项，它包含了最近使用的那些页表项
不可能页表miss了，快表hit了，因为页表缺失，信息一定不在主存页不在cache



TLB全相联，有有效位、脏位



7 输入输出系统

7.1 IO接口

IO接口是主机与IO设备之间数据交换的界面，它屏蔽了IO设备的差异，提供了一致的访问界面。功能如下

1. 数据格式（串并）转换和电平变换
2. 数据缓存（速度不匹配）

IO端口：IO接口中的各类寄存器
寄存器编址

- 统一编址，与主存统一编址，可以用访存指令去访问外设中的存储器
- 独立编址，对IO端口单独编号，需要专门的IO指令

7.2 磁盘存储器

磁盘组织：磁道、扇区、柱面
道(track)密度：垂直于磁道方向上（半径方向）单位长度磁介质所容纳的磁道数
位(bit)密度：单位长度磁道上所能记录的二进制信息位数面(surface)密度：单位面积上记录的二进制信息位数=道密度×位密度
存取时间t=寻道时间t_s+旋转等待时间t_w+数据传输时间(t_{WR})
最大旋转延迟=1/磁盘转速*60s/1min
平均旋转延迟=最大旋转延迟/2

7.3 闪存存储器

电可擦写、可编程只读存储器(EEPROM)

- NOR：随机，可以直接按字节访问，主要用于存储程序代码(code)
- NAND：块级IO访问，主要用于存储数据(data)

读快、写慢（块擦）
通过损耗均衡(wear leveling)减少块的磨损
固态盘SSD即为闪存

7.4 IO控制方式

7.4.1 程序查询方式

IO完全由CPU指令控制，数据传输再CPU的寄存器与外设及其接口的数据缓冲寄存器之间进行，IO不直接访问内存

完全串行工作

7.4.2 程序中断/中断驱动方式

由外设主动通知CPU，可以处理异常事件

1. 当外设准备好时，向CPU发中断请求
2. CPU响应后，中止线性程序执行，转入“中断服务程序”进行输入/输出操作
3. 中断服务程序执行完，CPU返回程序断点继续执行，外设和CPU并行工作

中断响应的条件

- CPU处于开中断状态
- 在一条指令**执行完**（区别“异常”是在指令执行过程中）
- 至少要有一个未被屏蔽的中断请求

中断响应过程

- 关中断
- 保护断点和程序状态
- 识别中断源

中断判优：在同时出现的若干个中断请求中找出级别最高的，以便进行相应的中断服务

- 软件判优：轮询法
- 硬件判优
  - 串行判优：链式查询
  - 并行判优：独立请求

优先级

- 中断响应的优先级由硬件排队线路决定
- 中断处理优先级由软件设置屏蔽码决定

缺点

- 对IO请求相应慢
- 数据传送速度慢

7.4.3 直接存储器(DMA)访问方式

直接存储器存取(Direct Memory Access)：

- 独立于处理器、能在高速外设和主存之间直接传送数据
- 由专门硬件（DMA控制器）控制总线进行传输
- 高速设备（磁盘光盘等），成批数据交换，且单位数据间的时间间隔较短
- 采用“请求–响应”方式
  - 每当高速设备准备好数据，就进行一次“DMA请求”，DMA控制器接收到DMA请求后，申请总线使用权
  - DMA控制器的总线使用优先级比CPU高
- 与中断控制方式结合使用
  - DMA传送前，“寻道”“旋转” 等操作结束时，通过“中断” 告知CPU
  - DMA控制器控制总线传送数据时， CPU执行其他程序
  - DMA传送结束时，要通过“DMA结束中断” 告知CPU

CPU对DMA请求的响应时间可以发生在每个总线周期结束时
DMA请求的优先级高于中断请求 IO数据一致性：OS维护

- IO读操作：cache置为无效
- IO写操作：cache flush（刷新），强迫cache中被更新的数据写回内存