Computer Organization Notes

经12-计18 张诗颖

1 RISC-V 指令系统

> **Levels of Representation**
> High Level Language Program 高级语言层
> $\xrightarrow{Compiler}$ Assembly Language Program 汇编语言层
> $\xrightarrow{Assembler}$ Machine Language Program 操作系统层 / 指令系统层
> $\xrightarrow{Machine\ Interpretation}$ Control Signal Specification 微体系结构层 / 数字逻辑层

- **ISA (Instruction Set Architecture) / 指令集架构**

  指令按功能分类：
  数据运算指令
  数据传输指令
  控制指令
  输入输出指令
  其它指令

  指令相关 Terminology：
  指令格式：`操作码 + 操作数地址` 的二进制分配方案（操作数：立即数/寄存器/内存）
  指令字：完整的一条指令的二进制表示
  指令字长：指令字中二进制代码的位数
  机器字长：计算机能够直接处理的二进制数据的位数
  指令字长（字节倍数）：0.5, 1, 2, ... 个机器字长
  对于THINPAD RISC-V来说，机器字长是4字节，指令字长也是4字节
  * 定长指令字结构（e.g., RISC-V）/变长指令字结构，定长操作码/扩展操作码

  指令集分类：
  CISC / Complex Instruction Set Computing 复杂指令集：x86 (Intel, AMD)
  RISC / Reduced Instruction Set Computing 精简指令集：ARM (ARM Holdings)

  * RISC-V不同的部分以模块化的方式组织在一起：ARM架构分为 A (Application)，R (Real-Time) 和 E (Embedded) 三个领域，彼此不兼容

- **RISC-V Introduction**

  > **Developers:** Krste Asanovic, Andrew Waterman, Yunsup Lee from UCB in 2010
  > **Documents:** RISC-V spec 指令集, RISC-V privileged spec 特权级编程, of which RV32I（32位基础整数指令集，也是唯一强制要求实现的指令集）is the basic instruction set

Name	Type	Descrtion
RV32I	基础指令	整数指令，包含算术、分支、逻辑、访存指令，有32个32位寄存器，能寻址32位地址空间
RV32E	基础指令	与RV32I一样，只不过只能使用前16个32位寄存器
RV64I	基础指令	整数指令，包含算术、分支、逻辑、访存指令，有32个64位寄存器，能寻址64位地址空间
RV128I	基础指令	整数指令，包含算术、分支、逻辑、访存指令，有32个128位寄存器，能寻址128位地址空间
M	扩展指令	包含乘法、除法、求模取余指令
F	扩展指令	单精度(32bit)浮点指令
D	扩展指令	双精度(32bit)浮点指令,必须要同时支持F扩展指令
Q	扩展指令	四倍精度浮点指令
A	扩展指令	存储器原子操作指令，比如比较并交换，读-改-写等指令
C	扩展指令	压缩指令,指令长度为16位，主要用于改善程序大小
P	扩展指令	单指令多数据（Packed-SIMD)指令
B	扩展指令	位操作指令
H	扩展指令	支持 Hypervisor 管理指令
J	扩展指令	动态翻译语言的指令
L	扩展指令	十进制浮点指令
N	扩展指令	用户中断指令
G	通用指令	包含 I、M、A、F、D指令

RISC-V 的**指令长度**固定为32位。

RISC-V 架构定义了**三种工作模式**（又称特权模式 / Privileged Mode）：

Machine Mode 机器模式，简称 M Mode

Supervisor Mode 监督模式，简称 S Mode

User Mode 用户模式，简称 U Mode

RISC-V 架构定义 M Mode 为必选模式，另外两种为可选模式。通过不同的模式组合可以实现不同的系统。

- **（大实验）监控程序的地址空间划分**

虚地址区间	说明
0x80000000-0x800FFFFF	Kernel代码空间
0x80100000-0x803FFFFF	用户代码空间
0x80400000-0x807EFFFF	用户数据空间
0x807F0000-0x807FFFFF	Kernel数据空间
0x10000000-0x10000008	串口数据及状态

串口寄存器位定义

地址	位	说明
0x10000000（数据寄存器）	[7:0]	串口数据，读、写地址分别表示串口接收、发送一个字节
0x10000005（状态寄存器）	[5]	状态位，只读，为1时表示串口空闲，可发送数据
0x10000005（状态寄存器）	[0]	状态位，只读，为1时表示串口收到数据

串口通信设计思想：MMIO（Memory-Mapped IO）

1.1 RV32I 指令格式

> RV32I: Base Integer Instruction Set, Version 2.1

所有指令都是32bit字长；有6种指令格式（核心指令格式：R/寄存器型，I/立即数型，S/存储型，U/大立即数；变种指令格式：B/分支指令，J/跳转指令）

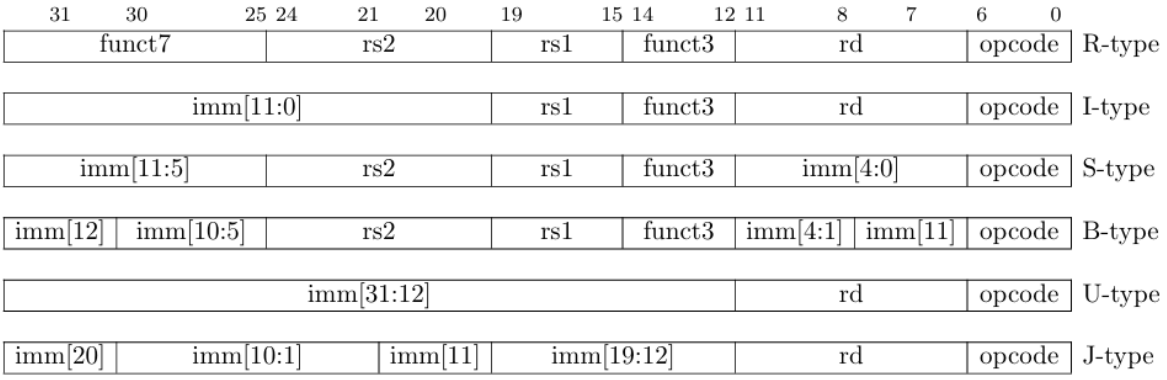
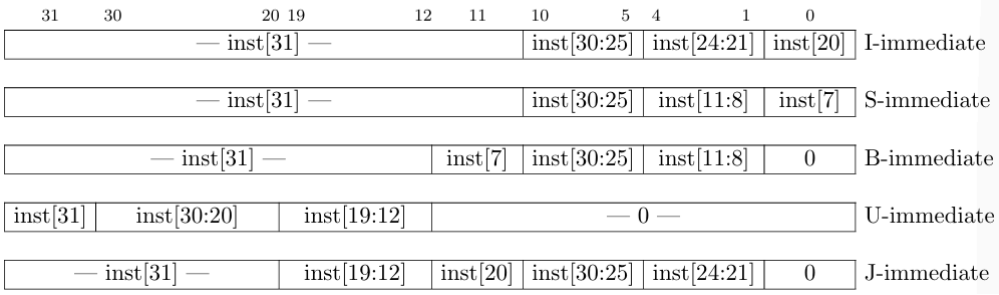


Figure 2.3: RISC-V base instruction formats showing immediate variants.

- `rd`：目标寄存器
  `rs`：源寄存器
  `imm`：立即数（immediate）
  `funct3`, `funct7`：minor opcode 或指令的其他数据
- **多少位用于立即数取决于指令中存在的寄存器号个数**
  R-type：存在一个目标寄存器和两个源寄存器的指令时没有立即数。例如两个寄存器相加（`ADD`）
  I-type：存在一个目标寄存器和一个源寄存器的指令拥有12bits的立即数，例如一个寄存器相加（`ADDI`）
  S-type：存在两个源寄存器且没有目标寄存器的指令，例如`STORE`指令，也有12bits的立即数，但因为各个寄存器位于不同的位置，立即数也必须位于不同的位置。
  U-type：只存在一个目标寄存器，没有minor opcode的指令，例如`LUI`指令，拥有20bits的立即数（major opcode和目标寄存器号需要12bits）
- **`S`和`B`、`J`和`U`指令格式的区别**
  `S`型指令将12位立即数分为“低5位”和“高7位”两个部分。
  `B`型指令的立即数也有12位，但它被重新编码为跳转偏移地址。
  - 由于分支指令的目标地址是相对于当前指令地址的偏移量，并且这个偏移量总是偶数（因为指令地址是 4 字节对齐的），立即数的最低位永远是 0。因此，在 B 型指令中，RISC-V 选择不在指令中存储该位，而是将立即数左移一位。
  - 移位后的12位立即数被分为四个部分放在不同的区域，但需要注意立即数中大多数位和在`S`型中没有变化
  - 立即数的最高位`imm[12]`是一个符号位，它用于表示这个立即数是正数还是负数，即对应跳转分支目标地址相对于当前地址为正或负偏移量。

  `J`型指令的立即数字段左移了1位，`U`型指令的立即数字段左移了12位。

Figure 2.4 shows the immediates produced by each of the base instruction formats, and is labeled to show which instruction bit (inst[y]) produces each bit of the immediate value.

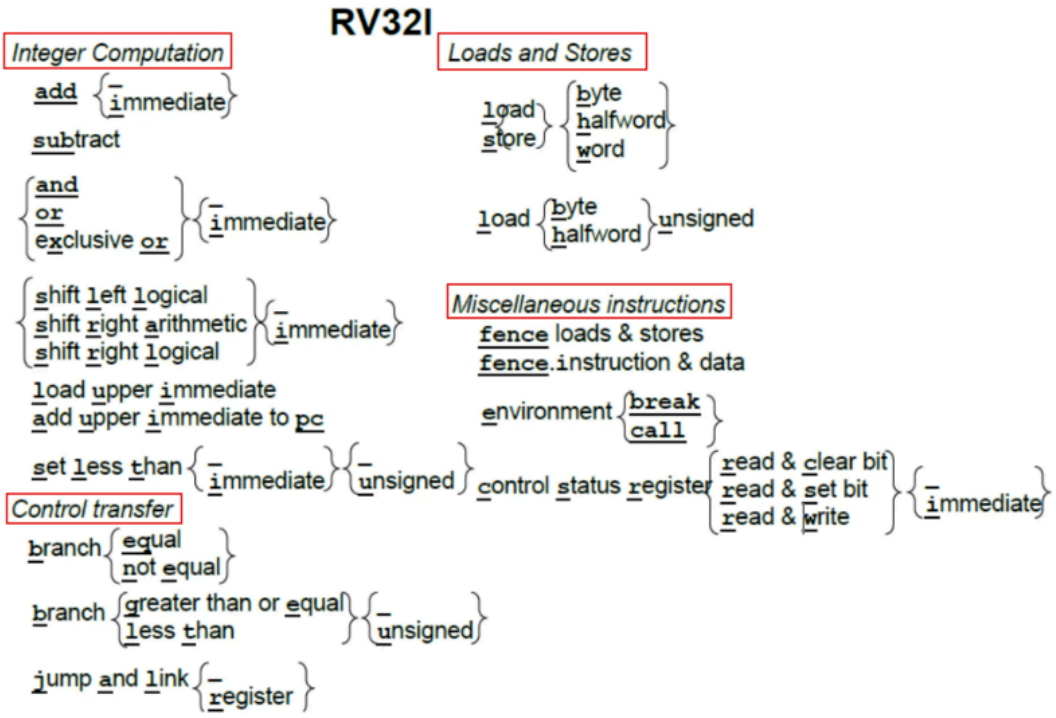


- **RISC-V 指令格式基本特点**
  - 源寄存器和目标寄存器都设计固定在所有 RISC-V 指令同样的位置上，指令译码相对简单，所以指令在 CPU 流水线中执行时，可以先开始访问寄存器，然后再完成指令解码。
  - 所有立即数的符号位总是在指令的最高位。这么做的好处是，有可能成为**关键路径**的立即数符号扩展（Sign extension）可以在指令解码前进行，有利于 CPU 流水线的时序优化。
- **监控程序基础版本（19条）：** `add`, `addi`, `and`, `andi`, `auipc`, `beq`, `bne`, `jal`, `jalr`, `lb`, `lui`, `lw`, `or`, `ori`, `sb`, `slli`, `srli`, `sw`, `xor`

  监控程序支持中断版本：+ `csrrc`, `csrrs`, `csrrw`, `ebreak`, `ecall`, `mret`

  监控程序支持TLB版本：+ `sfence.VMA`

以下内容参考 [RISC-V base architecture - 翻车鱼 (shi1011.cn)]



Instruction	Format	Meaning
`add rd, rs1, rs2`	R	Add registers
`sub rd, rs1, rs2`	R	Subtract registers
`sll rd, rs1, rs2`	R	Shift left logical by register
`srl rd, rs1, rs2`	R	Shift right logical by register
`sra rd, rs1, rs2`	R	Shift right arithmetic by register
`and rd, rs1, rs2`	R	Bitwise AND with register
`or rd, rs1, rs2`	R	Bitwise OR with register
`xor rd, rs1, rs2`	R	Bitwise XOR with register
`slt rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu rd, rs1, rs2`	R	Set if less than register, unsigned
`addi rd, rs1, imm[11:0]`	I	Add immediate
`slli rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui rd, imm[31:12]`	U	Load upper immediate
`auipc rd, imm[31:12]`	U	Add upper immediate to `pc`

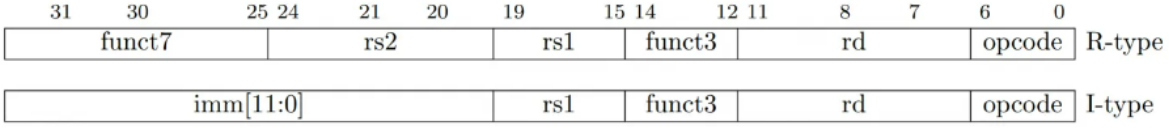
Table 3.2: Listing of RV32I computational instructions.

Instruction	Format	Meaning
`lb rd, imm[11:0](rs1)`	I	Load byte, signed
`lbu rd, imm[11:0](rs1)`	I	Load byte, unsigned
`lh rd, imm[11:0](rs1)`	I	Load half-word, signed
`lhu rd, imm[11:0](rs1)`	I	Load half-word, unsigned
`lw rd, imm[11:0](rs1)`	I	Load word
`sb rs2, imm[11:0](rs1)`	S	Store byte
`sh rs2, imm[11:0](rs1)`	S	Store half-word
`sw rs2, imm[11:0](rs1)`	S	Store word
`fence pred, succ`	I	Memory ordering fence
`fence.i`	I	Instruction memory ordering fence

Table 3.3: Listing of RV32I memory access instructions.

1. 整数计算 Integer Computation

> 算术指令：`add`, `sub`
> 逻辑指令：`and`, `or`, `xor`
> 移位指令：`sll`, `srl`, `sra`



R型：Register - Register

	inst[31:25]	inst[24:20]	inst[19:15]	inst[14:12]	inst[11:7]	inst[6:0]	PC = PC + 4
add 指令	0000000	rs2	rs1	000	rd	0110011	**reg[rd] = reg[rs1] + reg[rs2]**
sub 指令	0100000	rs2	rs1	000	rd	0110011	**reg[rd] = reg[rs1] - reg[rs2]**
sll 指令	0000000	rs2	rs1	001	rd	0110011	reg[rd] = reg[rs1] << reg[rs2]
slt 指令	0000000	rs2	rs1	010	rd	0110011	reg[rd] = reg[rs1] < reg[rs2]
sltu 指令	0000000	rs2	rs1	011	rd	0110011	reg[rd] = reg[rs1] < reg[rs2]
xor 指令	0000000	rs2	rs1	100	rd	0110011	**reg[rd] = reg[rs1] ^ reg[rs2]**
srl 指令	0000000	rs2	rs1	101	rd	0110011	reg[rd] = reg[rs1] >> reg[rs2]
sra 指令	0100000	rs2	rs1	101	rd	0110011	reg[rd] = reg[rs1] >> reg[rs2]
or 指令	0000000	rs2	rs1	110	rd	0110011	**reg[rd] = reg[rs1] | reg[rs2]**
and 指令	0000000	rs2	rs1	111	rd	0110011	**reg[rd] = reg[rs1] & reg[rs2]**

shift left logical
set less than (signed)
set less than unsigned
shift right logical
shift right arithmetic

- shift logical: 补0
  shift arithmetic: 根据最高符号位补0或1
- set less than unsigned: 决定是否符号拓展；把源寄存器内的整数视为 unsigned integer

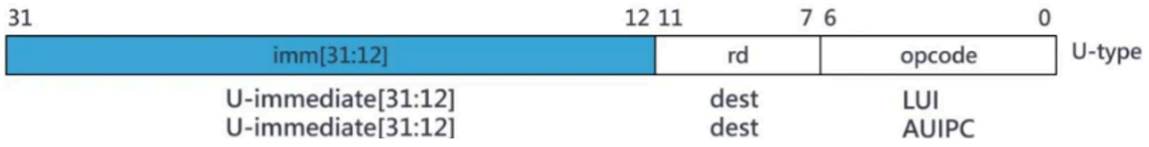
I型：Immediate

	inst[31:20]		inst[19:15]	inst[14:12]	inst[11:7]	inst[6:0]	PC = PC + 4
addi 指令	imm[11:0]		rs1	000	rd	0010011	**reg[rd] = reg[rs1] + imm[11:0]**
slti 指令	imm[11:0]		rs1	010	rd	0010011	reg[rd] = reg[rs1] < imm[11:0]
sltiu 指令	imm[11:0]		rs1	011	rd	0010011	reg[rd] = reg[rs1] < imm[11:0]
xori 指令	imm[11:0]		rs1	100	rd	0010011	**reg[rd] = reg[rs1] ^ imm[11:0]**
ori 指令	imm[11:0]		rs1	110	rd	0010011	**reg[rd] = reg[rs1] | imm[11:0]**
andi 指令	imm[11:0]		rs1	111	rd	0010011	**reg[rd] = reg[rs1] & imm[11:0]**
slli 指令	0000000	shamt	rs1	001	rd	0010011	reg[rd] = reg[rs1] << shamt[4:0]
srli 指令	0000000	shamt	rs1	101	rd	0010011	reg[rd] = reg[rs1] >> shamt[4:0]
srai 指令	0100000	shamt	rs1	101	rd	0010011	reg[rd] = reg[rs1] >> shamt[4:0]

set less than immediate (signed)
set less than immediate unsigned
shift left logical immediate
shift right logical immediate
shift right arithmetic immediate

- 在立即数算术指令中，没有减法运算
- 其中的`imm[11:0]`表示无符号整数

U型：Upper Immediate



lui (load upper immediate): opcode = 0110111

```
immU = inst[31:12]
reg[rd] <= {immU, 000000000000}
pc = pc + 4
```

auipc (add upper immediate to PC)

```
immU = inst[31:12]
Reg[rd] <= {immU, 000000000000} + pc
Pc = pc +4
```

- `LUI`：将20bits的立即数存储到`rd`位置的前20位，后12位用0补齐
  `AUIPC`：将`LUI`后的立即数加上`PC`储存到`rd`位置

- `lui` 配合 `addi`（设置低12bits）可以将寄存器设置为任意32bits的立即数（但注意这个12bits的立即数需要是正数，否则符号拓展之后结果可能会不同）
  解决方法：`lui` 高20位加1；或者可以使用伪指令 `li x10, 0xDEADBEEF`

□ 例2：设置0xDEADBEEF

```
lui  x10,0xDEADB       # x10=0xDEADB000
addi x10,x10,0xEEF     # x10=0xDEADAEEF
```

解决办法：lui 高20位 加1

伪指令li x10,0xDEADBEEF：自动生成两条指令

2. 存取 Load / Store

> 32bits：`lw`, `sw`
> 16bits：`lh`, `lhu`, `sh`
> 8bits：`lb`, `lbu`, `sb`

I型：Load

	inst[31:20]	inst[19:15]	inst[14:12]	inst[11:7]	inst[6:0]	
lb 指令 load byte	imm[11:0]	rs1	000	rd	0000011	reg[rd] = memory[[reg[rs1] + imm[11:0]]
lbu 指令 load byte unsigned	imm[11:0]	rs1	100	rd	0000011	reg[rd] = memory[[reg[rs1] + imm[11:0]]
lh 指令 load halfword	imm[11:0]	rs1	001	rd	0000011	reg[rd] = memory[[reg[rs1] + imm[11:0]]
lhu 指令 load halfword unsigned	imm[11:0]	rs1	101	rd	0000011	reg[rd] = memory[[reg[rs1] + imm[11:0]]
lw 指令 load word	imm[11:0]	rs1	010	rd	0000011	reg[rd] = memory[[reg[rs1] + imm[11:0]]

- 从内存中加载数据，并将其（符号拓展）到目标寄存器 `rd` 的**低位**部分，（如果有）符号扩展的结果会填充到寄存器的高位部分。

S型：Store

	inst[31:25]	inst[24:20]	inst[19:15]	inst[14:12]	inst[11:7]	inst[6:0]	
sb 指令 store byte	imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	memory[[reg[rs1] + imm[11:0]] = reg[rs2][7:0]
sh 指令 store halfword	imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	memory[[reg[rs1] + imm[11:0]] = reg[rs2][15:0]
sw 指令 store word	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	memory[[reg[rs1] + imm[11:0]] = reg[rs2][31:0]

STORE 指令

- 将寄存器的对应字节部分存储到内存中的**低位**对应位置，高位位置对应的数据不会被修改。

3. 控制与跳转 Control Transfer

J型：直接跳转

	inst[31]	inst[30:21]	inst[20]	inst[19:12]	inst[11:7]	inst[6:0]	
jal 指令 jump and link	imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	1101111	reg[rd] = pc + 4 pc = pc + sign_ext(imm[20:1], 1'b0)

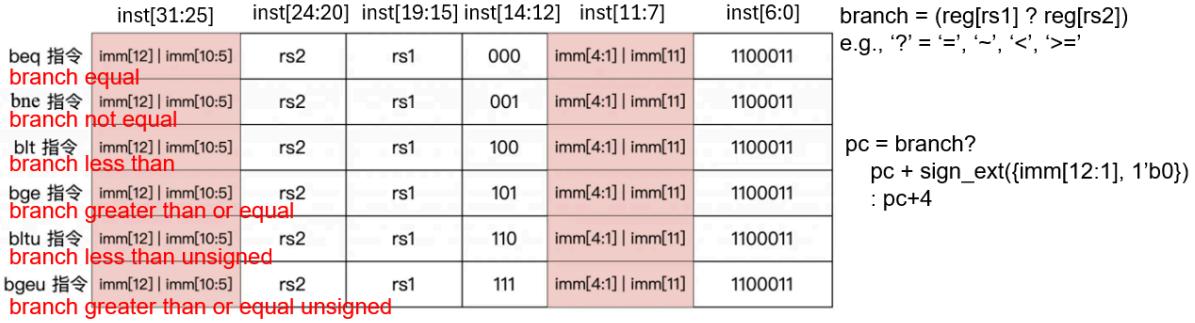
- 将20bits的立即数作符号拓展并左移一位，产生一个32bits的符号数
- 将该32bits的符号数和 `pc` 相加得到跳转地址（这样 `jal` 可以作为短跳转指令，跳转至 `pc±1MB` 的范围内）

- 同时，`jal`会把紧随其后的那条指令的地址，即`pc+4`，存入目标寄存器（通常为`ra`）中。如果目标寄存器为0，则`jal`相当于`goto`指令；否则`jal`可以实现函数调用的功能。

I型：相对跳转

- `jalr`指令会把12bits的立即数和源寄存器相加（通常`ra`作为源寄存器，`x0`作为目标寄存器），并把相加的结果末位清零，作为新的跳转地址。和`jal`指令一样，`jalr`也会把紧随其后的那条指令的地址，存入目标寄存器中

B型：条件分支



- 现对两个寄存器`rs1`和`rs2`的带/无符号值比较判断条件是否成立
- 若条件成立，跳转的目标地址为当前`pc`地址，加立即数符号拓展后左移的偏移量

4. 杂项指令 Control Status

控制状态（Control Status）寄存器的相关指令。控制状态寄存器用来进行处理器的控制，包括处理器的优先级配置，中断相关处理以及虚拟地址的相关处理。

- `csrrc`, `csrrs`, `csrrw`, `csrrci`, `csrrsi`, `csrrwi` 可以用来访问控制状态寄存器。
- `ecall` 指令用于向运行时环境发出调用请求，一般用于实现系统调用。
- `ebreak` 指令将控制转移到调试环境。
- `fence` 指令对外部可见的访存请求，例如设备 IO 和内存访问等进行串行化。
- `fence.i` 同步指令和数据流。

5. 伪指令 Pseudo Instruction

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Pseudoinstruction	Base Instruction(s)	Meaning
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags

6. 汇编指导语句 Assembler Directives

```
.text        # 进入代码段
.align 2     # 后续代码按照 2 字节对齐
.global main # 声明全局符号 main

.section .rodata       # 进入只读数据段
.balign 4              # 数据段按照 4 字节对齐
.string "hello, %s!\n" # 创建空字符结尾的字符串
```

更多汇编器指导语句见下表：

Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.
`.option rvc`	Compress subsequent instructions (see Chapter 7).
`.option norvc`	Don't compress subsequent instructions.
`.option relax`	Allow linker relaxations for subsequent instructions.
`.option norelax`	Don't allow linker relaxations for subsequent instructions.
`.option pic`	Subsequent instructions are position-independent code.
`.option nopic`	Subsequent instructions are position-dependent code.
`.option push`	Push the current setting of all `.options` to a stack, so that a subsequent `.option pop` will restore their value.
`.option pop`	Pop the option stack, restoring all `.options` to their setting at the time of the last `.option push`.

1.2 ThinPAD RISC-V 指令系统

- **采用与RV32I兼容的指令格式**
  32位固定字长
  操作码位置及长度固定
  寻址方式简单
- **共设计有19+3+6+1条指令**
  19条基础指令：可以用于执行监控程序1（不包括异常与中断，不包括虚拟地址）
  3条运行RV64监控程序需要的指令
  6条是支持优先级，支持中断与异常的指令
  1条是支持虚拟地址的指令（没有TLB的话可实现为Nop）

所需要实现的基础指令19条，后续只统计这19条的情况

监控程序基础版本	ADD, ADDI, AND, ANDI, AUIPC, BEQ, BNE, JAL, JALR, LB, LUI, LW, OR, ORI, SB, SLLI, SRLI, SW, XOR
监控程序支持中断版本	基础版本所有指令+CSRRC, CSRRS, CSRRW, EBREAK, ECALL, MRET
监控程序支持TLB版本	上一行所有指令+SFENCE.VMA

- **RISC-V 指令系统**
  - **算术指令**：RISC-V忽略溢出问题（高位被截断，低位写入）；有4个乘法指令（`mul` / 得到整数低32位乘积，`mulh` / 得到高32位且操作数都是有符号数，`mulhu` / 操作数都是无符号数，`mulhsu` / 操作数一个有符号一个没符号）；除法指令（e.g., `div`，`rem` 表示取余）
    如果乘法需要获得完整的64位值，需要使用指令 `mulh[[s]u] rdh, rs1, rs2`，`mul rdl, rs1, rs2`
  - **访存指令（数据传输指令）**：专用内存到寄存器之间传输数据的指令（`load` 和 `store`），其它指令只能操作寄存器

    ```
    memop reg, off(bAddr) # 操作码 / 源寄存器或目标寄存器 / 基地址寄存器 / 地址偏移立即数
    ```

    - 内存是按照地址进行编址的，不是按照字进行编址的
    - 字（word = 4 bytes）地址之间有4个字节的距离（字的地址为最低位的字节的地址）
    - 小端机；字节数据传输指令（`load` 高位符号拓展，`store` 高位直接忽略）
      有无符号的`load`符号拓展：`lb`, `lh` vs `lbu`, `lhu`

      ```
      let *(x1) = 0x0000_0180:
      lb x11, 1(x1) # x11 = 0x0000_0001
      lb x12, 0(x1) # x12 = 0xFFFF_FF80
      sb x12, 2(x1) # *(x1) = 0x0080_0180
      ```

  - **分支与跳转指令**
    - 比较：`slt`（`slti`）/ set less than，`sltu`（`sltiu`）/ set less than unsigned
    - 分支：`blt`（`bge`），`bltu`（`bgeu`）
    - 条件跳转：`beq` / branch if equal, `bne` / branch if not equal
      无条件跳转：`jal` / jump and link / 将某一条指令的地址放在寄存器`ra`（`jal rd offset`），`jalr` / jump and link register（`jal rd offset(rs1)`）

1. 寻址方式小结（x86 + RISCV）

- 立即数寻址：操作数是指令的地址字段部分直接给出的**值**（e.g., I型 `addi`）
- 直接寻址：操作数是指令的地址字段给出的值作为**地址**直接（而不是偏移量）对应的存储器中的**值**
- 寄存器寻址（操作数是寄存器中的**值**，e.g., R型 `ADD`） + 间接寻址（操作数是寄存器中的值作为**地址**对应的寄存器的**值**）
- 变址寻址：操作数**的地址**是变址寄存器中的值作为**地址**对应的寄存器的**值**（间接寻址），和指令中的变址偏移量**立即数**相加得到（e.g, I/S型 `Load`, `Store`）
- 相对寻址：操作数**的地址**是程序计数器PC的**值**（即当前执行指令的地址），和指令中的相对寻址偏移量**立即数**相加得到（e.g., J型 `JAL`，B型 `BEQ`）
- 间接寻址：指令的地址码字段给出的是操作数地址的地址（多一次读内存储器操作）
- 基址寻址：操作数是计算机中设置的一个专用的基址寄存器的**值**，和指令中的地址码**立即数**相加得到
- 堆栈寻址：堆栈指针SP给出的栈顶地址（返回地址默认在`ra`）

2. 寄存器

- **通用寄存器（General Purpose Registers）**：RISC-V 有32个32bit（`XLEN=32`）通用寄存器，编号从`x0`到`x31`，用于存储操作数和计算结果。`x0` is hardwired with all bits equal to 0. General purpose registers `x1-x31` hold values that are Boolean values, two's complement signed binary integers, or unsigned binary integers. 还有另一个通用寄存器：program counter / `pc` 存储有当前指令的地址。

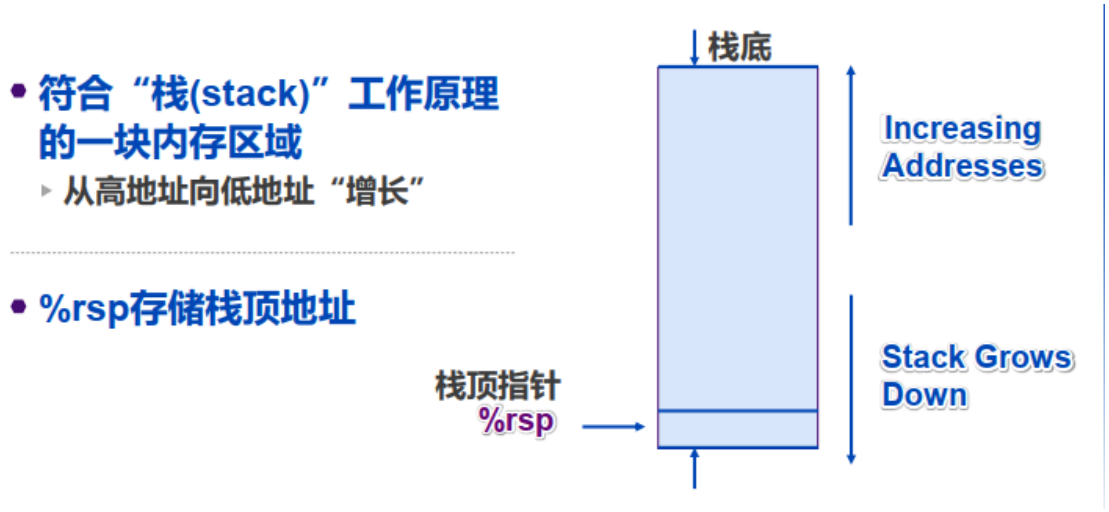
寄存器	ABI名字	描述	保存者Saver
x0	zero	Hard-wired zero	--
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5	t0	Temporary/alternative link register	Caller
x6-7	t1-2	temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	temporaries	Caller

- **控制状态寄存器（CSR / Control and Status Registers）**：用于配置或记录一些运行的状态（e.g., 异常和中断处理），是处理器核内部的寄存器，使用专有的12位地址码空间

3. 函数调用

寄存器：`x10 - x17` 用来传递参数或返回值；`x1` 是返回地址寄存器，用于返回到起始点；`x2` 是sp栈指针
跳转指令"link"：在调到对应函数内部之前，将下一条指令的地址放置在寄存器 `x1`（`ra` / 返回地址寄存器）中

- 寄存器使用惯例：`caller`（调用者函数）调用者保存的寄存器在函数调用前后可能会被改变，`callee`（被调用函数）被调用者保存的寄存器在调用前后不会被改变
- 数据的内存排布情况——栈帧结构（stack frame）



2 数据表示及检错纠错

2.1 数据表示

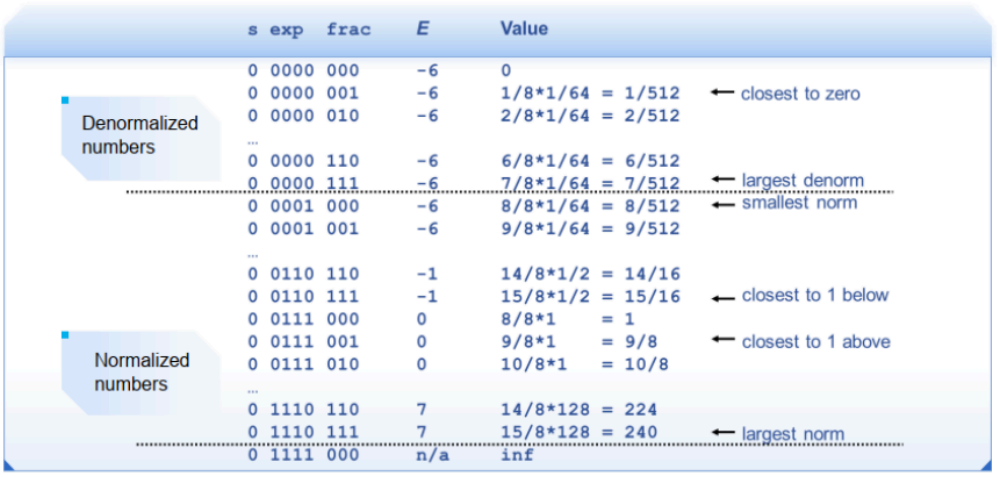
> 这部分详见《计算机系统概论》笔记

- **逻辑型数据**：1和0
- **字符型数据**：三个字符集编码标准
  - ASCII (American Standard Code for Information Interchange): 7位二进制编码，占用一个字节，可以表示128个西文字符
  - UNICODE (Universal Multiple-Octet Coded Chareacter Set)：ISO10646标准，为每种语言的每个字符设定了统一并且唯一的二进制编码；若使用16位表示一个字符，可以表示65535个字符；兼容ASCII
  - UTF-8编码：变长字符编码，字符长度由首字节确定；字符首字节外均以"10"开始，可自同步

字符位数	字节1	字节2	字节3	字节4	字节5	字节6
7	0ddddddd					
11	110ddddd	10dddddd				
16	1110dddd	10dddddd	10dddddd			
21	11110ddd	10dddddd	10dddddd	10dddddd		
26	111110dd	10dddddd	10dddddd	10dddddd	10dddddd	
31	1111110d	10dddddd	10dddddd	10dddddd	10dddddd	10dddddd

- **点阵字体**：本质是单色位图（文字编码 => 查找文字文件 => 找到点阵 => 显示）
  矢量字体（True Type）：一个字可以用多条曲线来表示，每条曲线保存其关键点
- **数值型数据**：
  - 小数：定点数，浮点数（注意点：数值范围和数据精度）
    - 注意 `exp=0111` 对应 `E=0`

      8位浮点数表示：exp域宽度为4 bits，frac域宽度为3 bits，bias=7

    - 浮点数算术运算

      加减：转化为浮点数 => 计算阶差并统一到高次幂 => 尾数求和（注意符号）=> 规格化处理（位数最高位为0则要进行左移和阶码减1的操作）=> 舍入处理 => 检查溢出

      乘除：阶数相加减，尾数乘除 => 规格化 => 舍入（并可能再次规格化）=> 溢出检查
  - 整数（带符号位）：原码（Signed-Magnitude）/ 补码（Two's Complement）/ 反码（One's Complement），符号拓展，大端/小端，加减乘除
    - 反码是原码按位取反，补码是原码按位取反加一
    - 零的原码和反码均有两个编码，补码只有一个

2.2 检错纠错码

码距（最小码距）：任意两个合法码之间至少有几个二进制位不同。
三种常用的检错纠错码：奇偶校验码（奇校验/偶校验）、海明校验码、循环冗余校验码

1. 奇偶校验码

> 用于并行码检错

原理：在k位数据码之外增加1位校验位，使K+1位码字中取值为1的位数总保持为偶数（偶 校验）或奇数（奇校验）。

2. 海明校验码

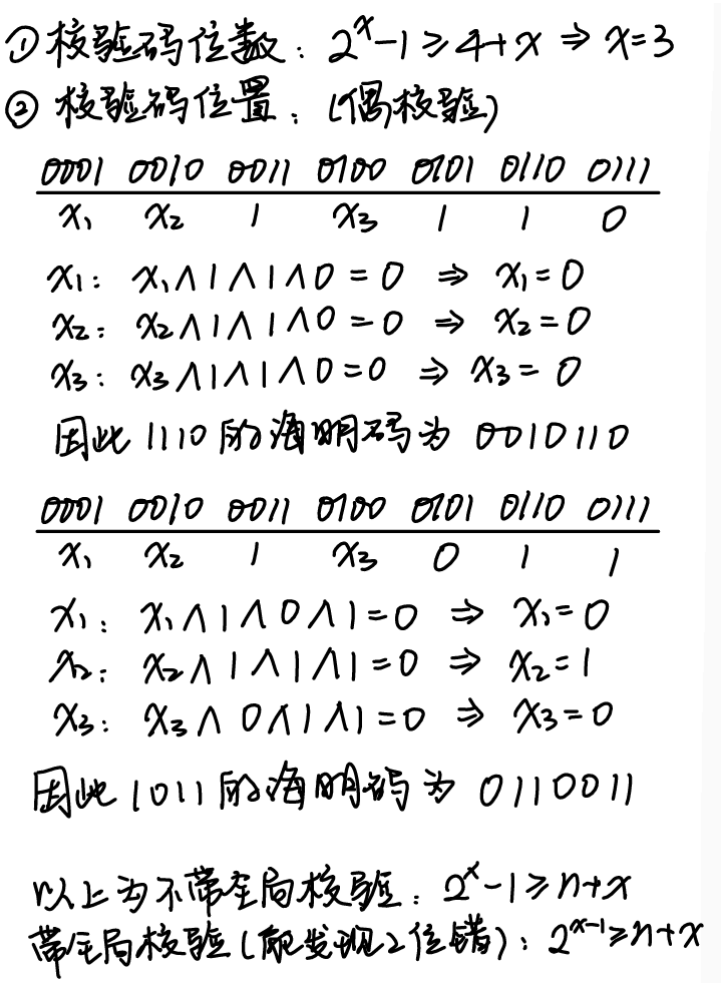
> 用于多位并行数据检错纠错处理

原理：为**k**个**数据位**设立**r**个**校验位**，使k+r位的码字同时具有这样两个特性；a) 能发现并改正k+r位中任何一位出错，b) 能发现k+r位中任何二位同时出错，但已无法改正。

- 不带全局校验：$2^r - 1 \geq k + r$，即用2^{r}个编码分别表示k个数据位，r个校验位中哪一位出错，都不错
- 带全局校验（能发现2位错）：$2^{r-1} \geq k + r$，用r-1位校验码为出错位编码，再单独设一位用以区分1位还是2位同时出错（偶校验），更实用

编码示例

> 若采用不带全局校验的海明码的方式，请给出1110和1011两个4位数据的海明码编 码。（异或 == 使得有偶数个1）



解码示例

海明码原理海 - 掘金 (juejin.cn)

2.3 ALU 的实现

1. 加减乘除的硬件实现

加减：超前进位加法器
乘除：原码乘法、布斯算法、恢复余数法、加减交替法

- 原码乘法
  若乘数的当前位==1，将被乘数和部分积求和
  若乘数的当前位==0，则跳过
  部分积移位
  所有位都乘完后，部分积即为最终结果
- （补码）布斯算法

  5. 请采用布斯乘法计算（-3）* 10。（用 6 位二进制表示）

  $[-3]_{补}=111101$，$[3]_{补}=000011$
  $[10]_{补}=001010$

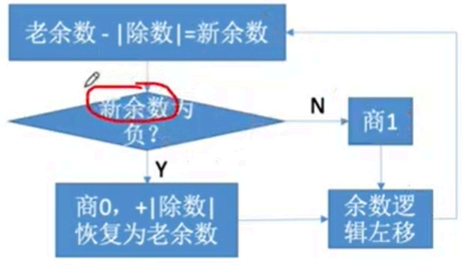
步骤	操作	部分积	附加位	
0	初始值	000000 001010	0	← 6x2+1 = 13位乘积结果
1	右移	000000 000101	0	
2	−X	000110 000101	0	
	右移	000001 100010	1	
3	+X	111110 100010	1	Rule：00：nop
	右移	111111 010001	0	01：+X
4	−X	000010 010001	0	10：−X
	右移	000001 001000	1	11：nop
5	+X	111110 001000	1	（其中X为被乘数，即第一个数）
	右移	111111 000100	0	
6	右移	111111 000010		

$(111111000010)_2=[-30]_{补}$
因此结果为-30.

- 原码一位除运算（恢复余数法）
  $[X/Y]_{原}=(X\bigoplus Y)(|X|/|Y|)$
  计算机会先默认上商1，如果搞错了再置位0；直接用求得的负余数求下一位商

设机器字长为5位（含1位符号位，n=4），x=0.1011，y=0.1101，采用原码恢复余数法求x/y
$|x|$=0.1011，$|y|$=0.1101，$[|y|]_{补}$=0.1101，$[-|y|]_{补}$=1.0011

被除数/余数			商
		0.1011	
	+[−|y|]补	1.0011	
余数为负，就要商0，并恢复余数		1.1110	0
	+[|y|]补	0.1101	
逻辑左移		0.1011	
	左移	1.0110	
	+[−|y|]补	1.0011	
余数为正，就要商1，不用恢复余数		0.1001	01
	左移	1.0010	
	+[−|y|]补	1.0011	
		0.0101	011
		...	...



- 加减交替除法（不恢复余数法）

  加减交替法(不恢复余数法)-CSDN博客

补码除法：加减交替法

设机器字长为5位（含1位符号位，n=4），x=+0.1000，y=-0.1011，采用补码加减交替除法求x/y

$[x]_{补}$=00.1000，$[y]_{补}$=11.0101，$[-y]_{补}$=00.1011　　$[x/y]_{补}$=1.0101，余0.0111×2^{-4}

	被除数/余数	ACC	MQ
	00.1000	001000	00000
$+[y]_{补}$	11.0101		
	11.1101	111101	00001
左移	11.1010	111010	00010
$+[-y]_{补}$	00.1011		
	00.0101	000101	00010
左移	00.1010	001010	00100
$+[y]_{补}$	11.0101		
	11.1111	111111	00101
左移	11.1110	111110	01010
$+[-y]_{补}$	00.1011		
	00.1001	001001	01010
左移	01.0010	010010	10100
$+[y]_{补}$	11.0101		
	00.0111	000111	10101

逻辑左移

精度误差不超过 2^{-n}

末位商恒置1

补码除法：
- 符号位参与运算
- 被除数/余数、除数采用双符号位

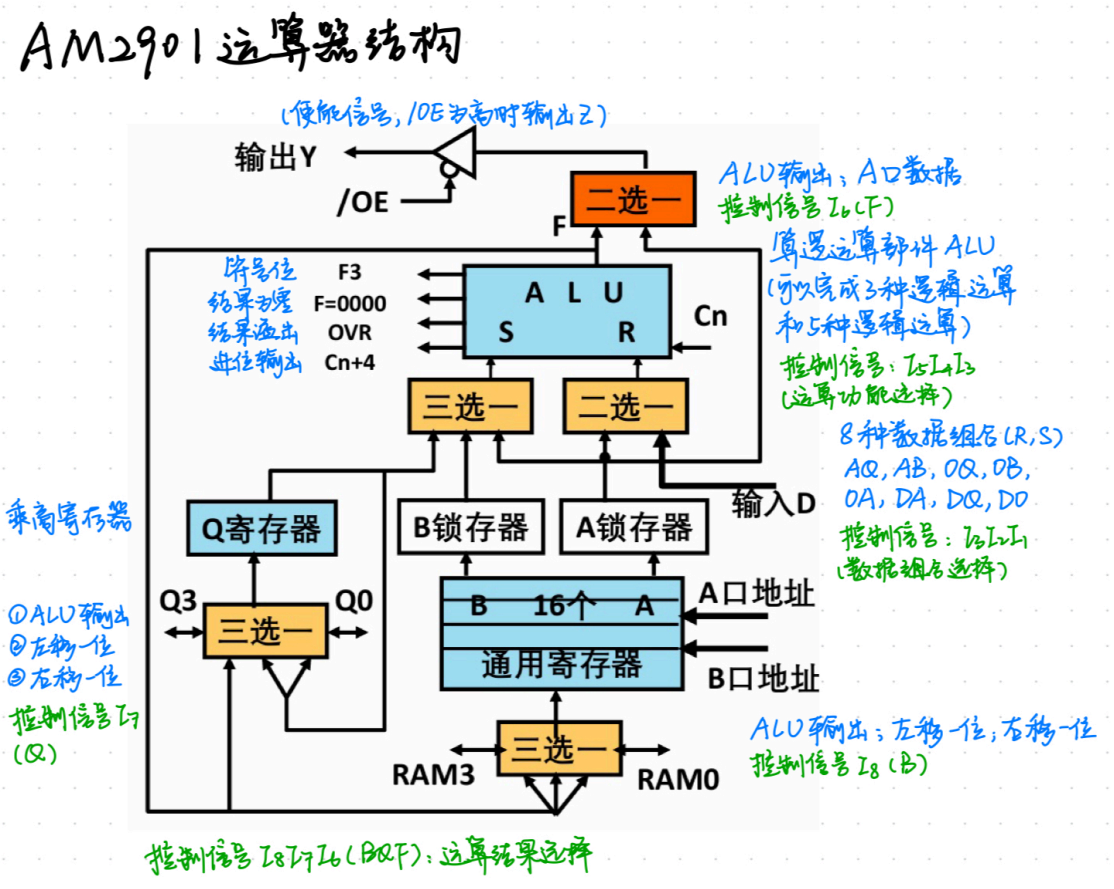
被除数和除数同号，则被除数减去除数；
异号则被除数加上除数。

余数和除数同号，商1，余数左移一位减去除数；
余数和除数异号，商0，余数左移一位加上除数。
重复n次

2. 运算器

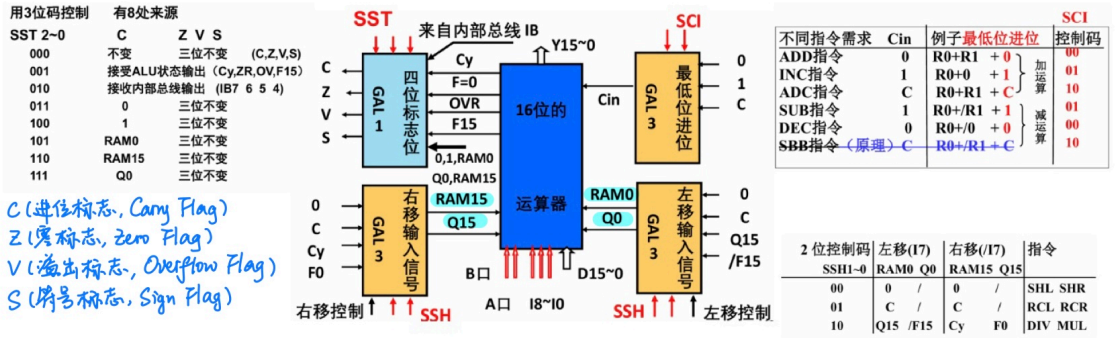
> 分类：定点运算器，浮点运算器

- **AM2901芯片**：4位的位片结构运算器器件



- **16位运算器的完整组成**
  多片AM2901芯片可以组合成16位运算器部件

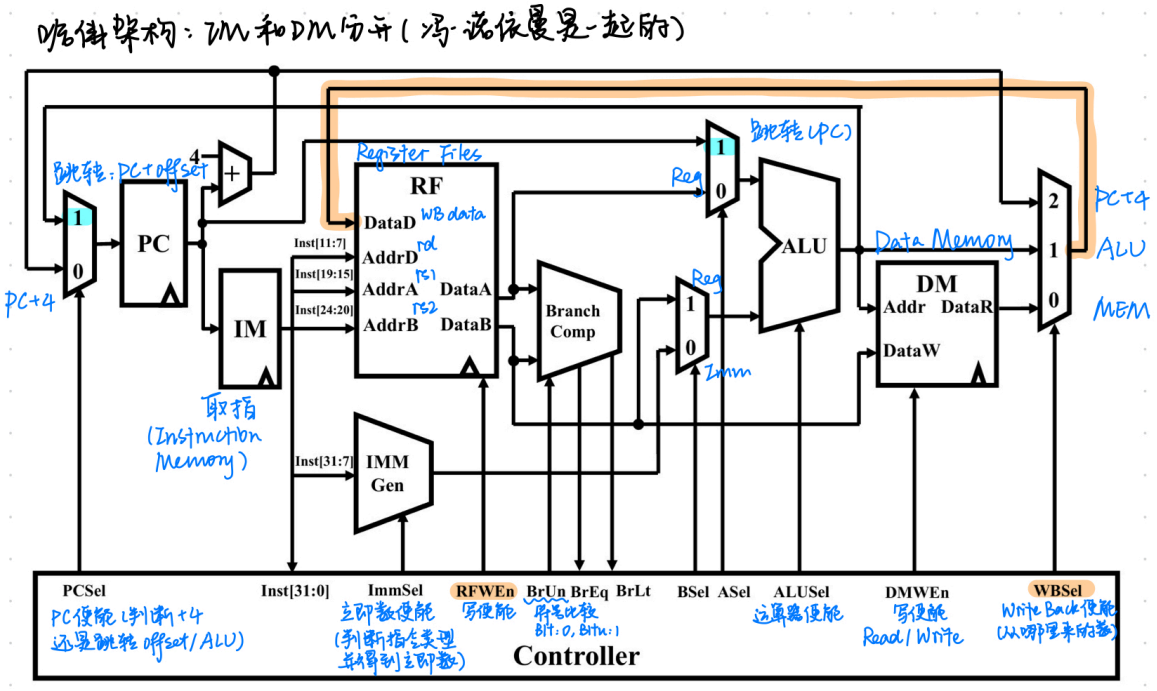
16位运算器的完整组成



3 CPU 结构设计

3.1 单周期 CPU

RISC-V RV32I 单周期CPU 数据通路



inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	-	-	+4	-	-	Reg	Reg	Add	Read	1	ALU
sub	-	-	+4	-	-	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	-	-	+4	-	-	Reg	Reg	(Op)	Read	1	ALU
addi	-	-	+4	I	-	Reg	Imm	Add	Read	1	ALU
lw	-	-	+4	I	-	Reg	Imm	Add	Read	1	Mem
sw	-	-	+4	S	-	Reg	Imm	Add	Write	0	-
beq	0	-	+4	B	-	PC	Imm	Add	Read	0	-
	1	-	ALU	B	-	PC	Imm	Add	Read	0	-
bne	0	-	ALU	B	-	PC	Imm	Add	Read	0	-
	1	-	+4	B	-	PC	Imm	Add	Read	0	-
blt	-	1	ALU	B	0	PC	Imm	Add	Read	0	-
bltu	-	1	ALU	B	1	PC	Imm	Add	Read	0	-
jalr	-	-	ALU	I	-	Reg	Imm	Add	Read	1	PC+4
jal	-	-	ALU	J	-	PC	Imm	Add	Read	1	PC+4
auipc	-	-	+4	U	-	PC	Imm	Add	Read	1	ALU

- **优点**
  每条指令占用一个时钟周期
  逻辑设计简单，时序设计也简单
- **缺点**
  各组成部件的利用率不高（维持有效信号）
  时钟周期应满足执行时间最长指令（Critical Path）的要求（`load` 指令）
- 每条指令平均使用的CPU周期个数 / **CPI = 1**

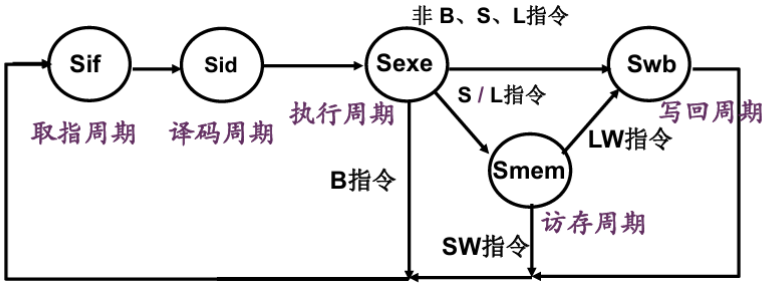
3.2 多周期 CPU

> 由存储器、寄存器堆、ALU部件、控制部件4部分组成

- **控制部器**
  - **硬连线控制器（组合逻辑控制器）**：采用组合逻辑线路、依据指令及其执行步骤直接产生控制信号

    由程序计数器PC、指令寄存器IR、节拍发生器Timer、控制信号产生部件组成。
  - **微程序控制器**：采用存储器电路把控制信号存储起来，依据指令执行的步骤读出要用到的信号组合
- **状态转移图**

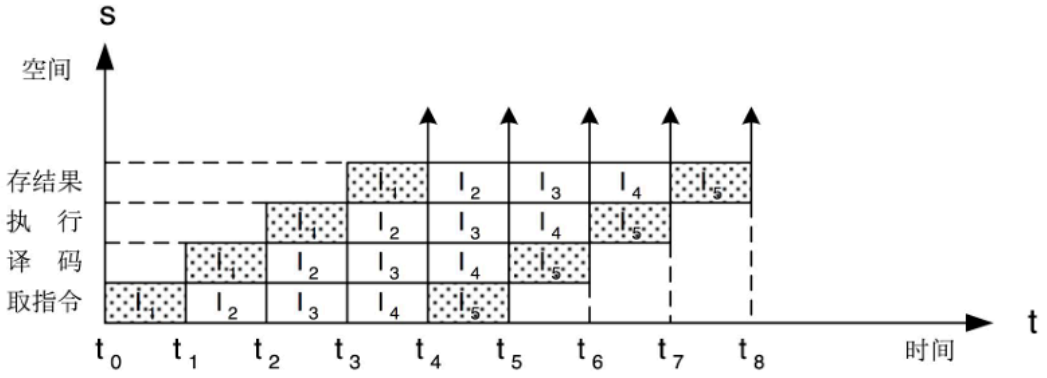
  IF / 取指 => ID / 译码 => EXE / 执行 => MEM / 访存 => WB / 写回



指令步骤	读取指令	指令译码	执行运算	内存读写	数据写回
B 指令	IR← MEM[PC] PC ← PC+4 PC_now← PC	C ← PC + offset	若条件成立 则 PC ← C		
R/I 运算		A ← Reg[rs1] B ← Reg[rs2]	C←A op B C←A + 符号扩展(Imm)		Reg[rd]←C
S 指令				Mem[C]←B	
L 指令				DR←Mem[C]	Reg[rd]←DR

3.3 流水线 CPU

- **流水线操作的前提**
  任务分解成多个步骤完成
  每个步骤使用不同的资源
  任务内部存在关联，完成每个任务的步骤顺序一致
  任务之间相互独立，没有依赖关系
- **流水型的特性**
  流水线并没有缩短单个任务的延迟，但提高了整个系统的吞吐率
  多个任务同时运行，占用不同的资源
  可能的加速比 = 流水段数
  但：效率受限于用时最长的阶段 & 装入和排空流水线 & 冲突处理 => 降低流水线效率
  表示法：连接图标（不推荐）、时空图



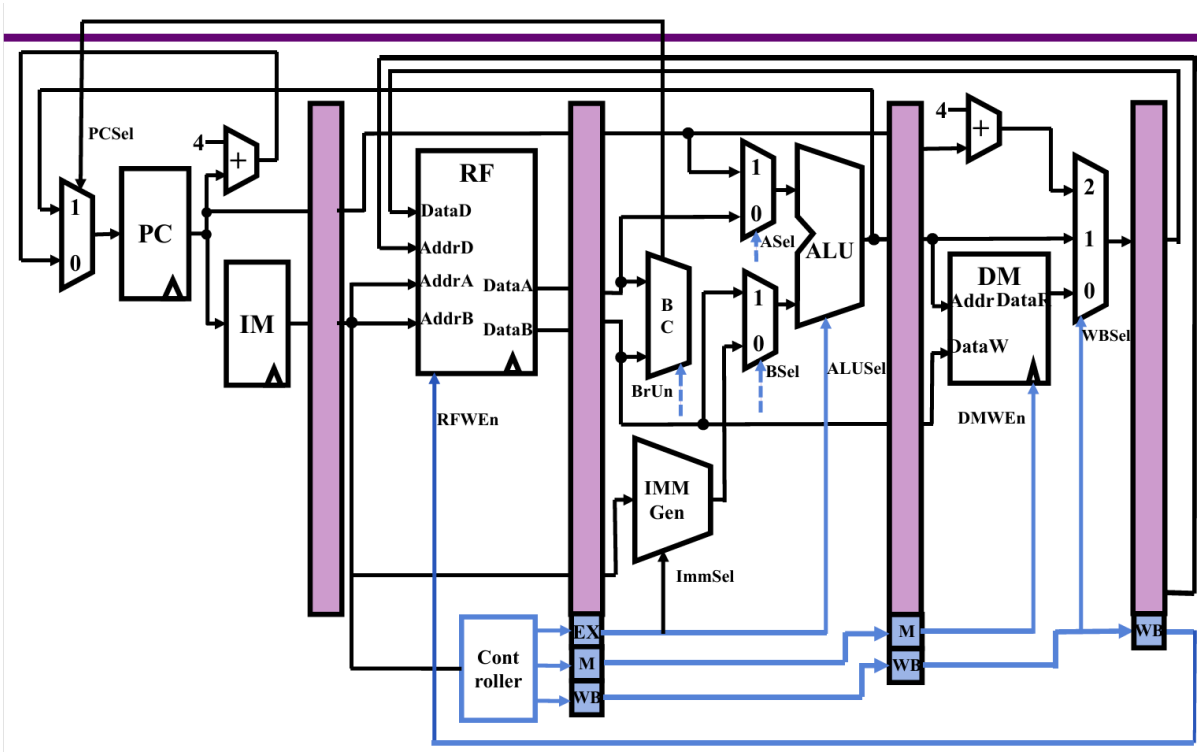
指令流水线时空图

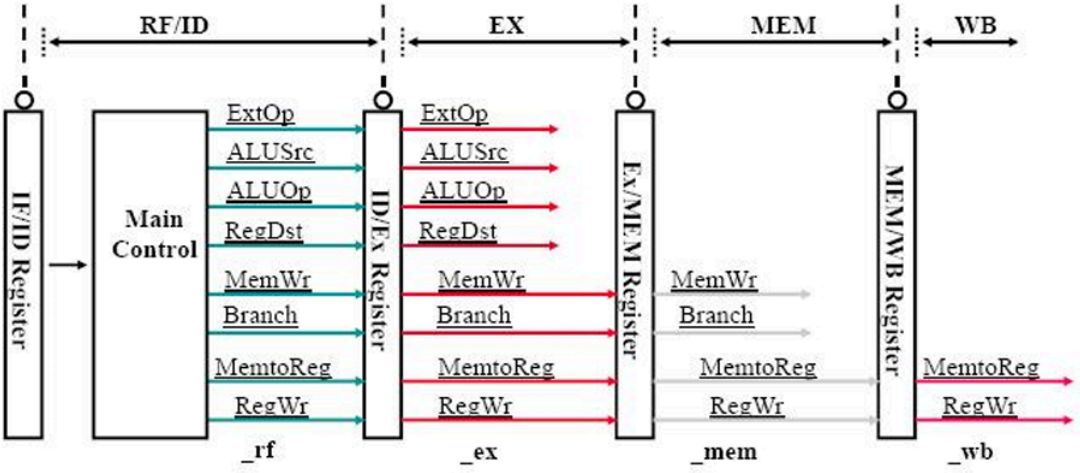
- **性能指标**
  吞吐率：单位时间执行指令的数量（GIPS，对应μs）
  加速比：与串行执行时速度提高的比率
  流水线"最佳段数"：stage越少，流水线在寄存器上的开销比例越低

1. 流水线寄存器

分段+锁存：流水线每一个功能段部件后面都要有一个缓冲寄存器，或称为锁存器，其作用是保存本流水段的结果。

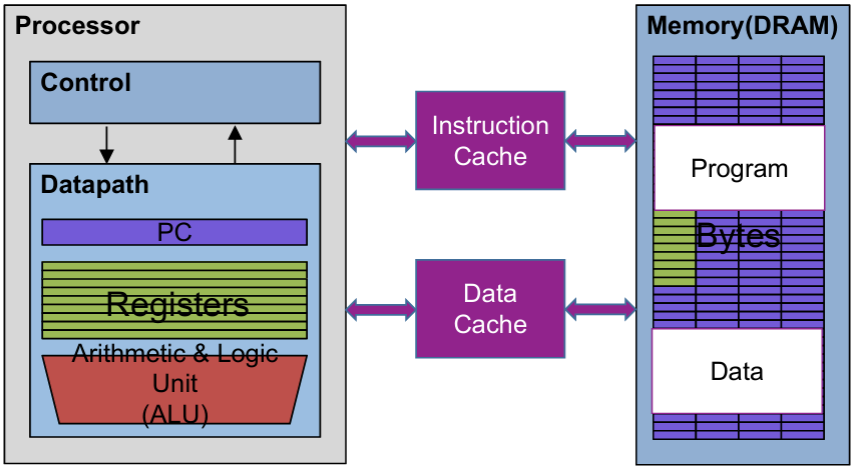
> **IF / 取指 => ID / 译码 => EXE / 执行 => MEM / 访存 => WB / 写回**
>
> 各步骤占用的资源：
> IF：IM，PC，总线
> ID：RF，控制信号发生部件
> EXE：ALU
> MEM：DM，总线
> WB：RF





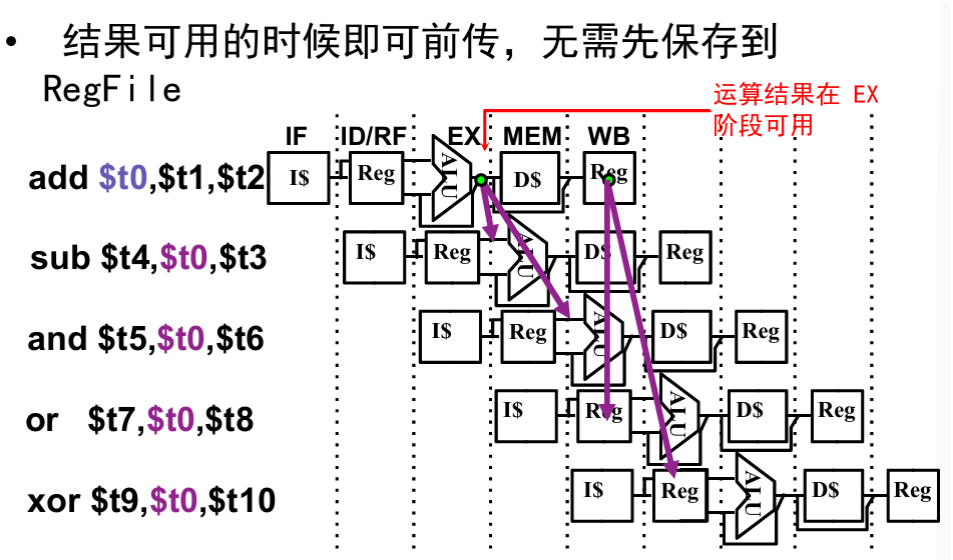
2. 流水线冲突（Hazard）

- **结构/资源冲突**：硬件资源满足不了指令重叠执行的要求
  - 寄存器结构冲突
    解决方案：设置两个独立的读端口和一个独立的写端口（`WB` / `ID`）
  - 内存结构冲突（`MEM` / `IF`）
    解决方案1：暂停流水线（插入气泡/空指令）
    解决方案2：将指令内存和数据内存分开（哈佛结构：L1区分IC和DC）



Caches: small and fast “buffer” memories
哈佛结构，通常使用在L1缓存中

- **数据冲突**：一条指令依赖于前面指令执行结果数据
  - **写后读冲突（RAW / Read After Write）**：数据前传（Data Forwarding）/ 数据旁路（Data Bypass）



写后写冲突（WAW / Write After Write）
RISC-V指令流水不会发生WAW冲突，因为流水中只有WB stage才会写寄存器

读后写冲突（WAR / Write After Read）
RISC-V指令流水不会发生，因为ID stage完成所有的读操作，而WB stage完成所有写操作

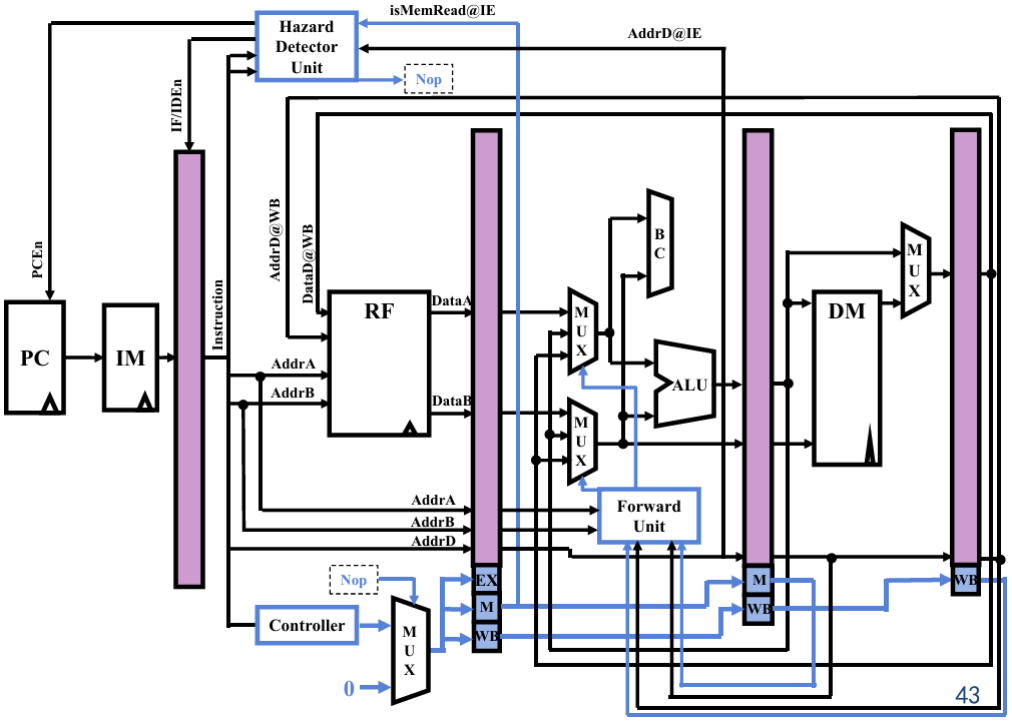
读后读不冲突（RAR / Read After Read）：不改变状态

——**检测数据冲突**（EXE冲突检测点）：EXE/MEM段指令的源寄存器之一与上一条指令的目的寄存器相同，且上一条指令需要改写目的寄存器（且不是0寄存器）

- **Load-Use 冲突**：需要硬件暂停流水线，无法通过数据前传完成

  解决方案1：
  数据装入之后的那个周期被称为是load delay slot（在周期中间插入bubble）
  检测点：上一条指令是Load指令，且它的写入寄存器和当前指令的某一源寄存器相同

  解决方案2：
  让汇编器assemble或者程序员放一条不相关的指令，避免冲突 => 无需暂停



- **控制/转移地址冲突**：分支指令或者其他需要改写PC的指令造成的冲突
  branch 和 jump 指令

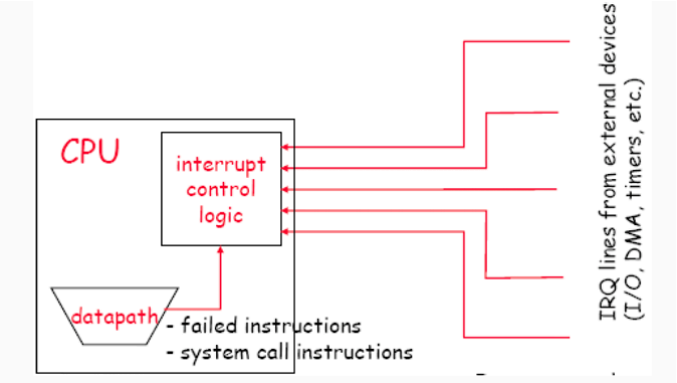
  解决方案1：暂停流水线（造成性能的降低）
  解决方案2：预测 - 分支不成功，`PC <= PC + 4`
  解决方案3：预测 - 分支成功，`PC <= PC + offset`
  解决方案4：动态预测，硬件根据上次分支的结果进行本次预测

  控制冲突的动态调度：**分支目标缓冲技术 / Branch Target Buffer (BTB)**
  缓存预测的分支目标的PC，不需要等到取指计算出跳转的offset了（类比`for`循环中每次判断跳转地址都是一致的）
  增加一位BHT (Branch History Table)，记录上一次是否跳转，根据上次分支的结果进行本次预测（可以修改成：连续两次预测错误再改变预测方向）

3. 中断异常

> 来自外部设备的是“中断”，来自CPU内部的是“异常”：
> 1 指令执行过程中发生错误：取指令、指令译码、计算、访存
> 2 外部设备提出服务请求
> 3 多进程运行时与其他进程发生资源冲突

- **异常处理的实现**



多周期的异常处理：增加一个检查异常/中断是否发生的步骤

硬件：`mepc`（保存发生异常指令的地址），`mcause`（异常代码）

- **多特权级模式**：用户态 vs 内核态，user mode / supervisor mode / machine mode
  处理器大部分时间运行在低权限，处理中断和异常时会将控制权移交到更高的权限模式

  RISC-V 特权指令（RV32/64 Privileged Instructions）：通过控制状态寄存器（CSR / Control State Register）来实现

  `mret`：机器模式返回

  `sret`：监管者模式返回

  `wfi`：等待中断（进入低功耗模式直到`mie&mip != 0`）

  `sfence.vma`：虚拟地址屏障指令

RV32/64 Privileged Instructions

machine-mode / supervisor-mode } trap return

supervisor-mode fence virtual memory address

wait for interrupt

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
0001000		00010	00000	000	00000	1110011	R sret
0011000		00010	00000	000	00000	1110011	R mret
0001000		00101	00000	000	00000	1110011	R wfi
0001001		rs2	rs1	000	00000	1110011	R sfence.vma

M Mode下的8个CSR异常处理

- mtvec(Machine Trap Vector)：发生异常时处理器需要跳转到的地址
- mepc(Machine Exception PC)：指向发生异常的指令
- mcause(Machine Exception Cause)：发生异常的种类
- mie(Machine Interrupt Enable)：处理器目前能处理和必须忽略的中断
- mip(Machine Interrupt Pending)：正准备处理的中断
- mtval(Machine Trap Value)：trap的附加信息：地址异常中出错的地址，非法指令异常的指令等
- mscratch(Machine Scratch)：暂时存放一个字大小的数据
- mstatus(Machine Status)：机器的状态

  - `mstatus`状态寄存器

XLEN-1	XLEN-2	23	22	21	20	19	18	17	
SD	0		TSR	TW	TVM	MXR	SUM	MPRV	
1	XLEN-24		1	1	1	1	1	1	

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	0	SPP	MPIE	0	SPIE	UPIE	MIE	0	SIE	UIE
2	2	2	2	1	1	1	1	1	1	1	1	1

`MIE`：可以作为是否允许**中断嵌套**的enable（M Mode）

`MPIE`：`MIE`的旧值

`MPP`：保留异常之前的权限模式

- 举例：时钟中断处理代码

```
# save registers
csrrw a0, mscratch, a0  # save a0; set a0 = &temp storage
sw a1, 0(a0)            # save a1
sw a2, 4(a0)            # save a2
sw a3, 8(a0)            # save a3
sw a4, 12(a0)           # save a4

# decode interrupt cause
csrr a1, mcause         # read exception cause
bgez a1, exception      # branch if not an interrupt
andi a1, a1, 0x3f       # isolate interrupt cause
li a2, 7                # a2 = timer interrupt cause
bne a1, a2, otherInt    # branch if not a timer interrupt
```

mie[7]对应于M模式中的时钟中断。控制状态寄存器mip具有相同的布局，并且它指示当前待处理的中断。

如果 mstatus.MIE = 1，mie[7] = 1，且 mip[7] = 1，则可以处理机器的时钟中断。

```
# handle timer interrupt by incrementing time comparator
la a1, mtimecmp         # a1 = &time comparator
lw a2, 0(a1)            # load lower 32 bits of comparator
lw a3, 4(a1)            # load upper 32 bits of comparator
addi a4, a2, 1000       # increment lower bits by 1000 cycles
sltu a2, a4, a2         # generate carry-out
add a3, a3, a2          # increment upper bits
sw a3, 4(a1)            # store upper 32 bits
sw a4, 0(a1)            # store lower 32 bits

# restore registers and return
lw a4, 12(a0)           # restore a4
lw a3, 8(a0)            # restore a3
lw a2, 4(a0)            # restore a2
lw a1, 0(a0)            # restore a1
csrrw a0, mscratch, a0  # restore a0; mscratch = &temp storage
mret                    # return from handler
```

- 可抢占式异常处理：在处理异常的过程中转到处理更高优先级的中断

- M Mode是RISC-V中**hart（硬件线程，hardware thread）**——

> 大多数处理器核心都只有一个hart，其概念和软件线程是相对的，是硬件的上下文执行环境（如果需要在一个处理器核心实现多个hart，需要多个硬件的上下文环境和多套寄存器）

可以执行的最高权限模式，其最重要的特性是**拦截和处理异常**：

- 同步异常 / 异常（Exceptions）：执行期间产生，访问无效的寄存器地址，或者执行了具有无效操作码的指令

  访问错误异常（物理内存不支持访问类型时发生）
  断点异常（执行`ebreak`指令，或者地址与数据与调试触发器匹配时发生）
  环境调用异常（在执行`ecall`的时候发生，相当于**系统调用**）
  非法指令异常（IF Stage中发现无效操作码）
  非对齐地址异常（有效地址不能被访问大小整除时）

- 异步异常 / 中断（Interrupts）：指令流异步的外部事件/中断，如鼠标点击

  软件中断（比如用户一个hart中断另外一个hart）
  时钟中断（实时计数器`mtime`大于hart的时间比较寄存器`mtimecmp`）
  外部中断（由中断控制器触发）

RISC-V要求实现**精确异常**：保证异常之前的所有指令都完整执行，后续指令都没有开始执行

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位
标识了中断或异常的具体原因。只有在实现了监管者模式时才能处理监管者模式中断和页面错误异常

- **内存限制**
- **S模式的异常处理**

4 存储器与缓存

存储介质：磁颗粒、半导体（电平 Transistor / 电容 Capacitor）、光——两个稳定状态表示二进制；便于转换状态；容易识别

SRAM：6T单元结构，速度更快且稳定，但占用更多空间
DRAM：1T1C结构，具有更高的密度，但需要定期刷新（refresh）防止电容中的电荷泄露

按访问方式分类：

随机访问存储器（RAM）：访问时间与存放位置无关；半导体存储器
顺序访问存储器（SAM）：按照存储位置依次访问；磁盘存储器
直接访问存储器（DAM）：随机 + 顺序（磁转片）；磁盘存储器
关联访问存储器（CAM）：根据内容访问（内容哈希）；Cache和TLB

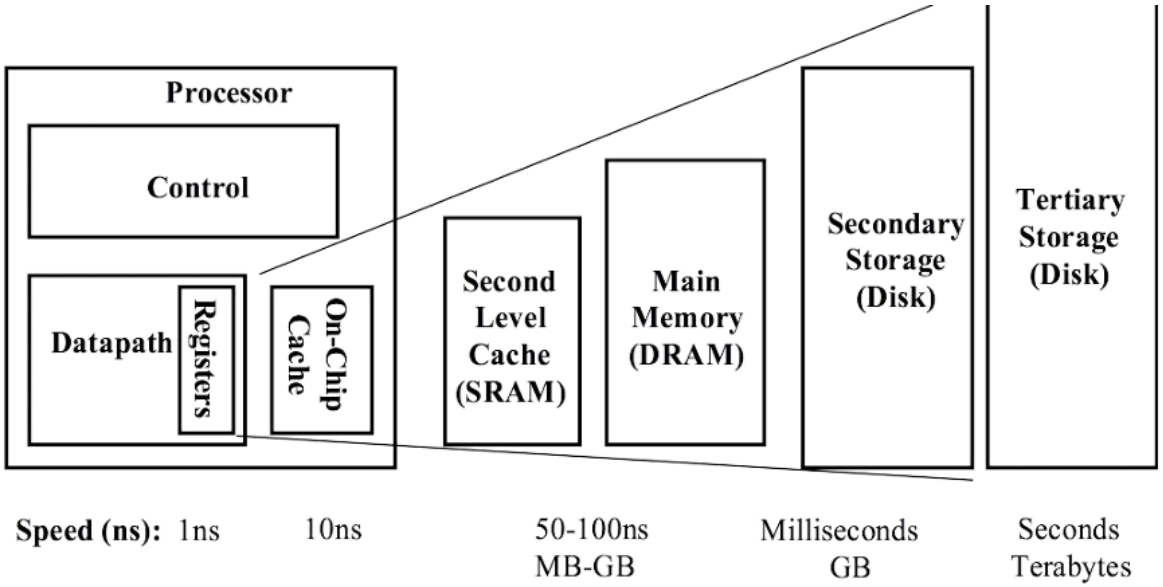
4.1 存储与映射关系

Register => Cache (SRAM) => Main Memory (DRAM) => Magnetic Disk (SSD/Disk) => Tape / Optical Disk

Locality：时间局部性 + 空间局部性

一致性原则：处在不同层次存储器中的同一个信息应保持相同的值
包含原则（现代设计不一定满足）：处在内层的信息一定被包含在其外层的存储器中，反之则不成立。即内层存储器中的全部信息，是其相邻外层存储器中一部分信息的复制品。

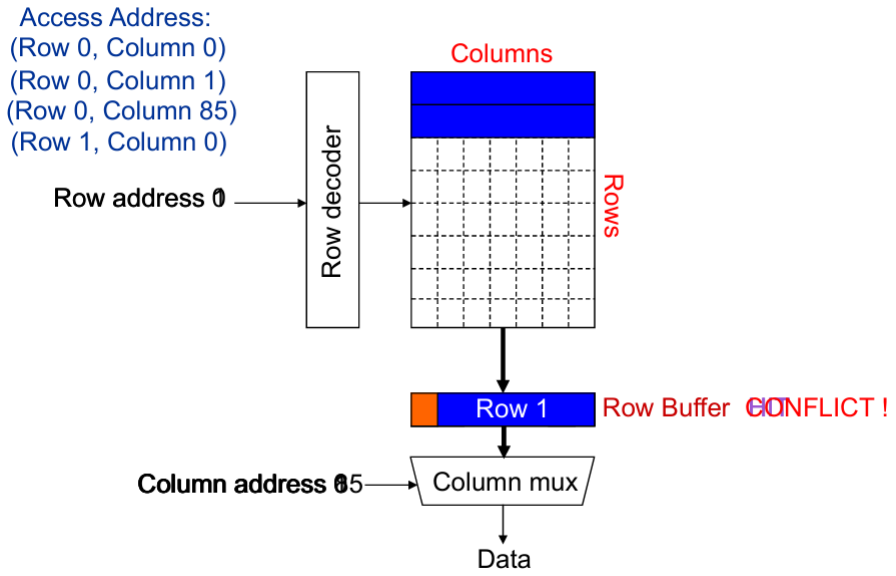
=> 合理地把程序和数据分配在不同存储介质中
Cache：最近被访问的信息项、最近被访问的信息项临近的信息



1. DRAM

动态存储器：是用金属氧化物半导体（MOS）的单个MOS管来存储一个二进制位（bit）信息的。信息被存储在MOS管T的源极的寄生电容CS中，例如，用CS中存储有电荷表示1，无电荷表示0。

DRAM formation: Row/Column => Bank => Chip => Rank => DIMM => Channel

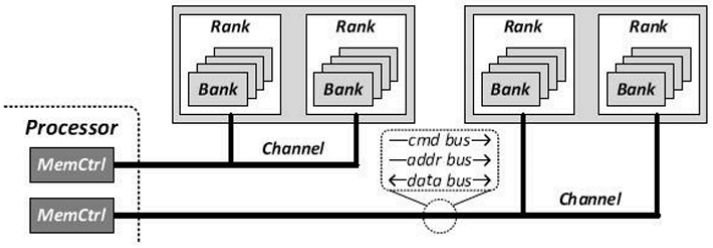


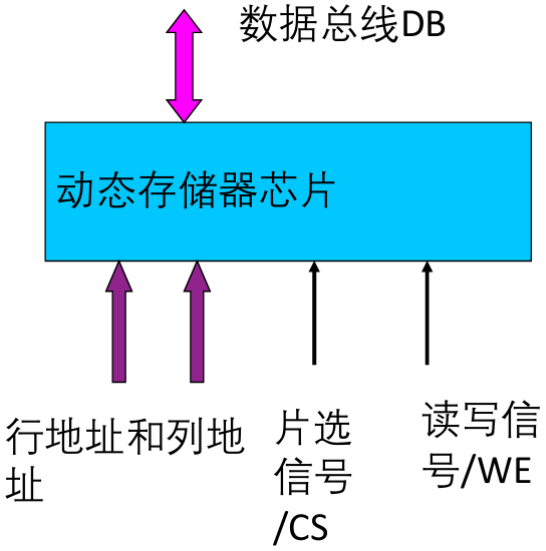
- **破坏性读出（Destructive Readout）**：读出时被强制清零；**预充电**延迟（precharge delay）

> - DRAM 中的数据存储在**存储单元**中，这些单元由电容器和晶体管组成，组织成行和列。
> - 访问数据时，首先通过**行选通**（Row Activation）将对应行的数据加载到**行缓冲区**（Row Buffer）中，称为**激活操作（Activate）**。快速分页组织的存储器（行地址锁存，只需要给出列地址）
> - 当一行数据被访问后，下一次访问前需要将该行断开连接并关闭（即**预充电**操作，Precharge）。这是因为：
>
> 1. DRAM 的存储单元需要恢复到一种标准的初始状态，以便接下来的访问。
> 2. 每一行的操作会改变存储阵列的电压分布，预充电将电压恢复到均衡态。
>
> 行激活（Activate）=> 数据访问（Read/Write）=> 预充电（Precharge）

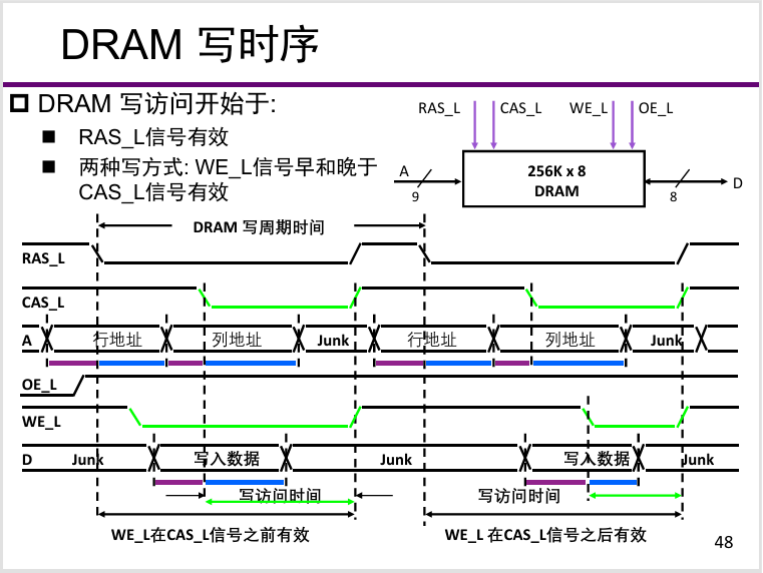
- **刷新操作（Refresh）**：集中刷新（停止读写并逐行刷新），分散刷新（每次读写后刷新该行），轮流把各行刷新一遍 => 恢复因电荷泄漏导致的数据衰减
- 保持力（retention）：~ms 级别
  耐久性（endurance）：

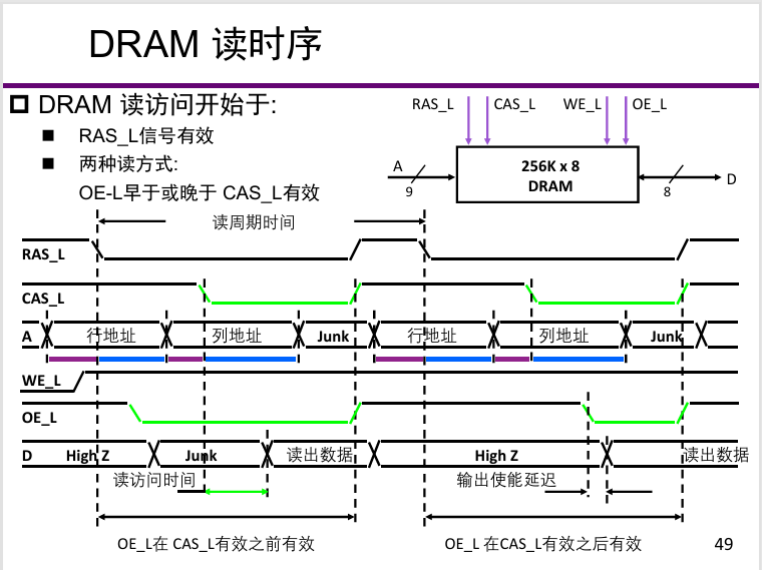
主存储器的链接





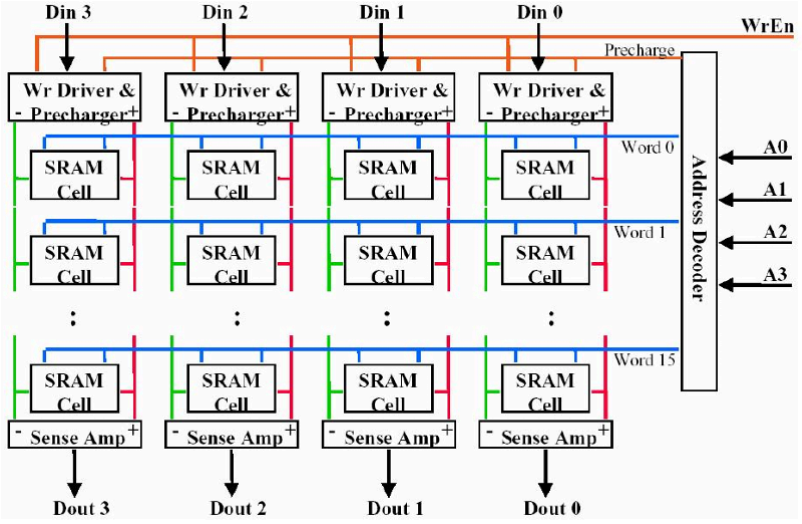
1. 地址总线：地址总线用于选择主存储器的一个存储单元；其位数决定了能够访问的存储单元的最大数目，即**最大可寻址空间。**
2. 数据总线：数据总线用于在计算机各功能部件之间传送数据；数据总线的位数（总线的**宽度**）与总线时钟频率的乘积，与该总线所支持的**最高数据吞吐**（输入/输出） 能力成正比。
3. 控制总线：控制总线用于指明总线的工作周期类型和本次入/出完成的时刻；即用不同的总线周期来区分要用哪个部件和操作的性质，以及直接存储器访问（DMA）总线周期。
   工作周期：主存储器读周期，主存储器写周期，I/O设备读周期，I/O设备写周期

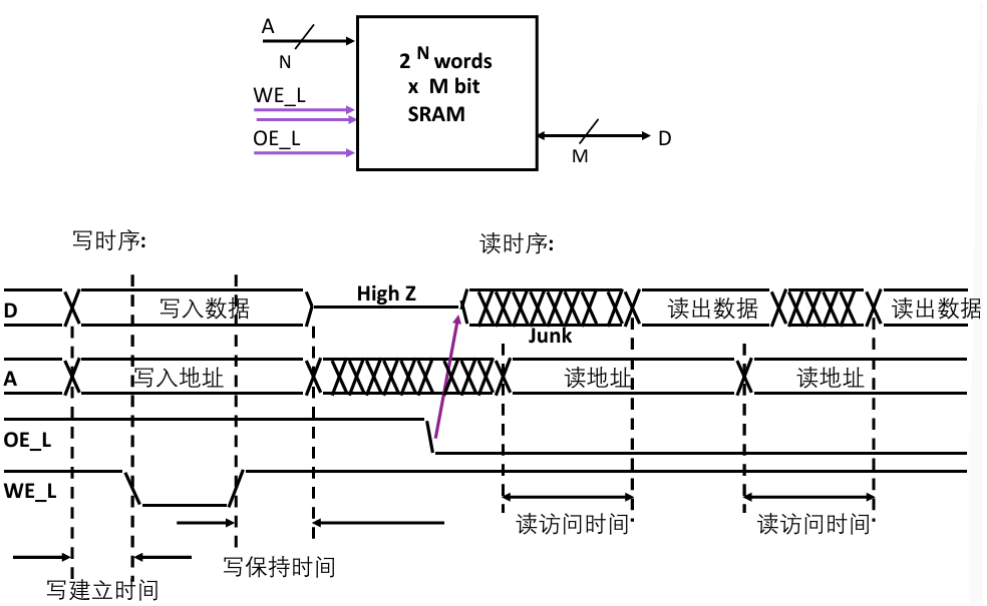




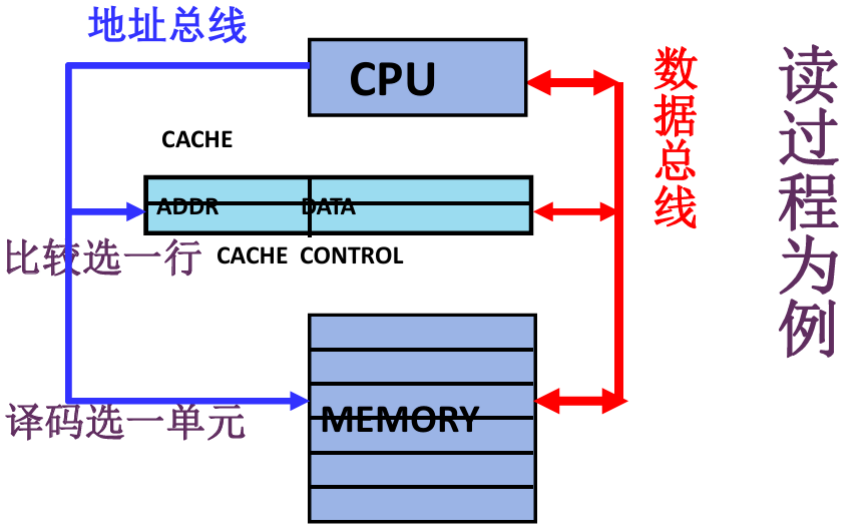
2. SRAM - 高速缓冲存储器 Cache

SRAM formation: 6-Transistor SRAM Cell => 典型组织方式





高速缓冲存储器 Cache：设置于主存和CPU之间的存储器，用高速的静态存储器实现，缓存了CPU频繁访问的信息。



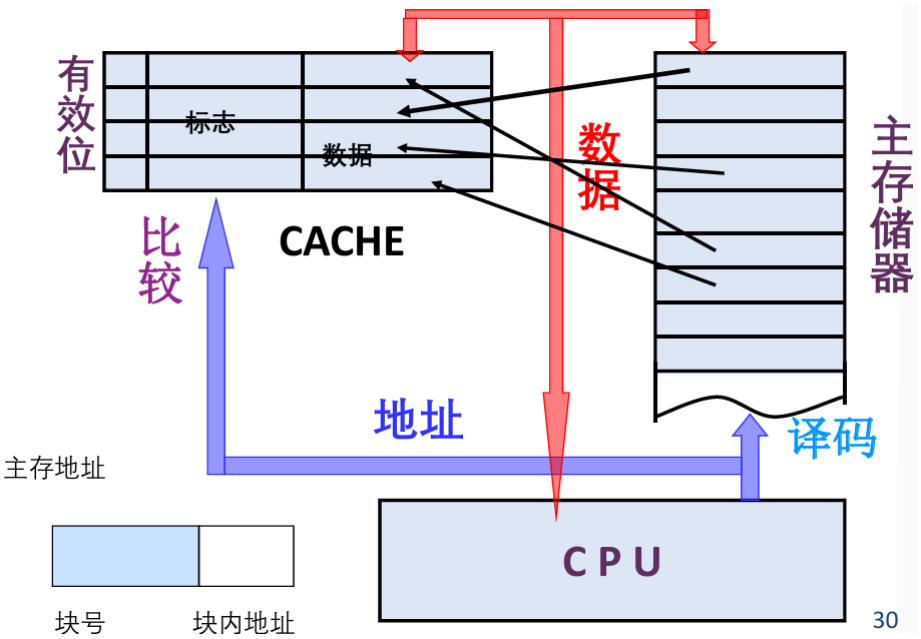
- **Cache 参数**（总容量在1KB~256KB）

  块（Line）：数据交换的最小单位，一般在4~128 Bytes
  命中（Hit）：在较高层次中发现要访问的内容
  　　命中率（Hit Rate）：命中次数 / 访问次数，一般在80%~99%
  　　命中时间：访问在较高层次中数据的时间，一般在1~4周期
  确实（Miss）：需要在叫低层次中访问块
  　　缺失率（Miss Rate）：1 - 命中率
  　　缺失损失（Miss Penalty）：替换较高层次数据块的时间 + 交付给处理器的时间
  　　失效损失：访问时间（一般在6~10个周期），传输时间（一般在2~22个周期）

  命中时间 << 缺失损失

  平均访问时间 = HR * 命中时间 + (1 - HR) * 缺失损失

- **全相联方式（Fully Associative）**



1. 无固定映射：主存中的一个数据块可以被存储到缓存中的任意一个位置，而不受特定映射规则（如直接映射或组相联映射）的限制。这意味着缓存需要检查所有行的标签来决定是否命中——利用率高且方式灵活；标志位较长且比较电路的成本太高。

   若主存空间有2^m块，则~~标志位（tag）要~~地址有m位；同时如果Cache有n块，则需要有n个比较电路。

   （主存地址：分为块号30bit 和块内地址2bit）

2. 缓存块（Line）的组成

   有效位（Valid Bit）：用于指示缓存中的数据是否有效

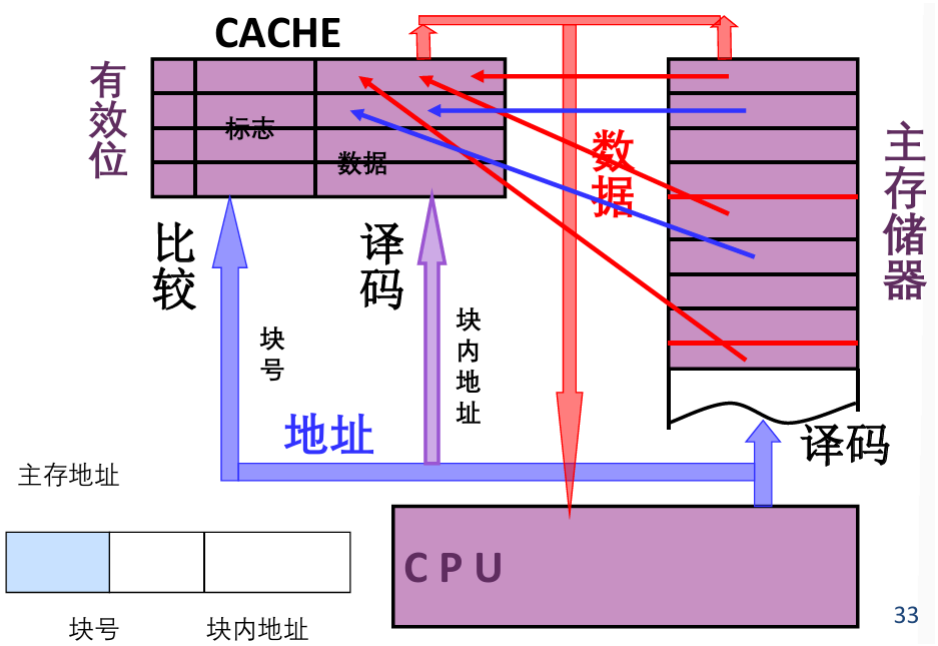
   标志（Tag）：用于标识缓存块中的数据与主存中的哪一部分数据相对应

   主存地址：块号 + 块内偏移

   标志 = 块号

   数据（Data）：通常存储4~128Bytes

- **直接映射方式**

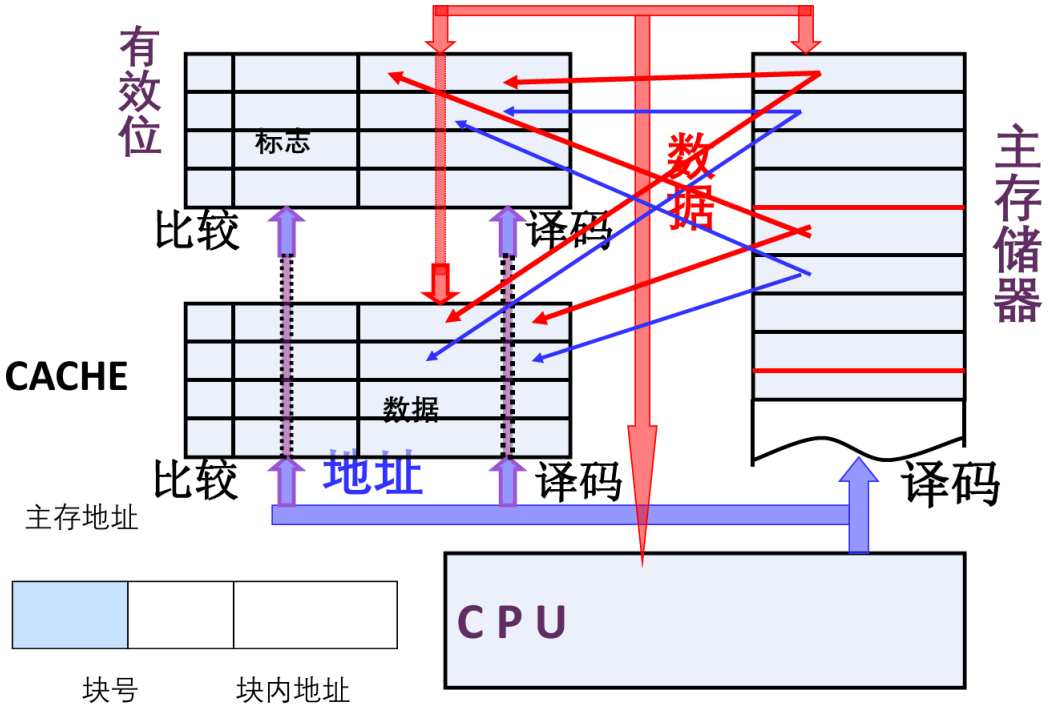


1. 固定映射：内存中的每个单元在Cache中只会有一个唯一的位置和它对应——方式直接但利用率低；标志位较短且比较电路的成本低（只需比较一次）。

若主存空间有2^m块，Cache中字块有2^c块，则标志位只有m-c位。

（主存地址：20bits tag + 10bits set index + 2bits 块内地址，假设块大小=4byte（指数据部分），1024个缓存行）

- **两路组相连映射**

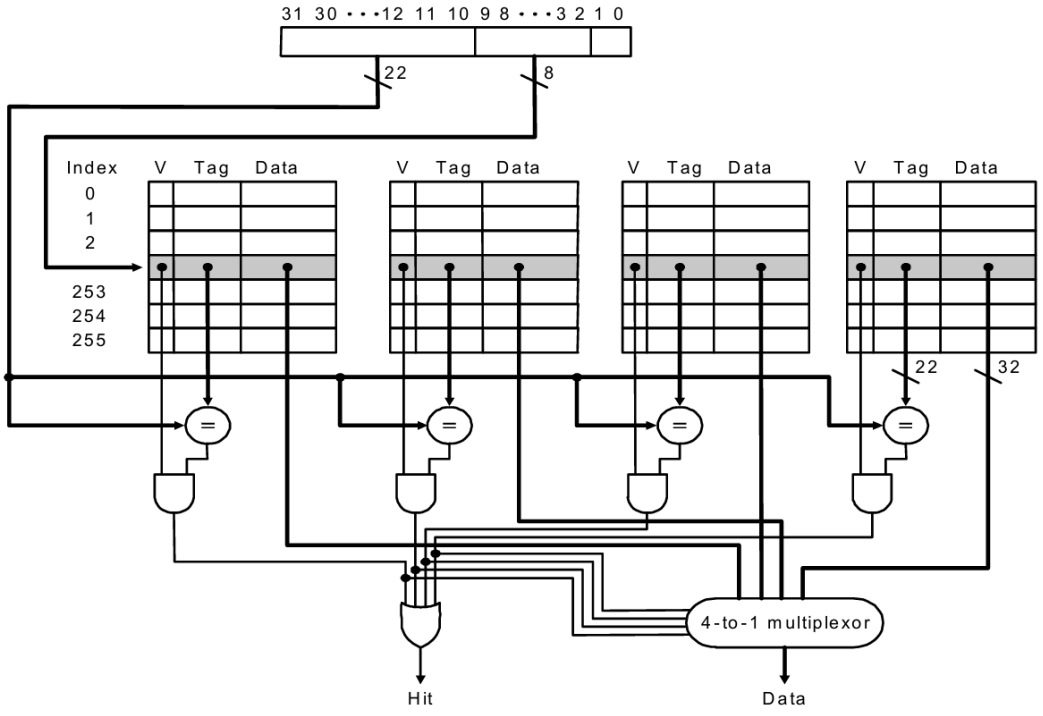


1. 综合了直接映射和全相联：组内全相联，组间直接映射

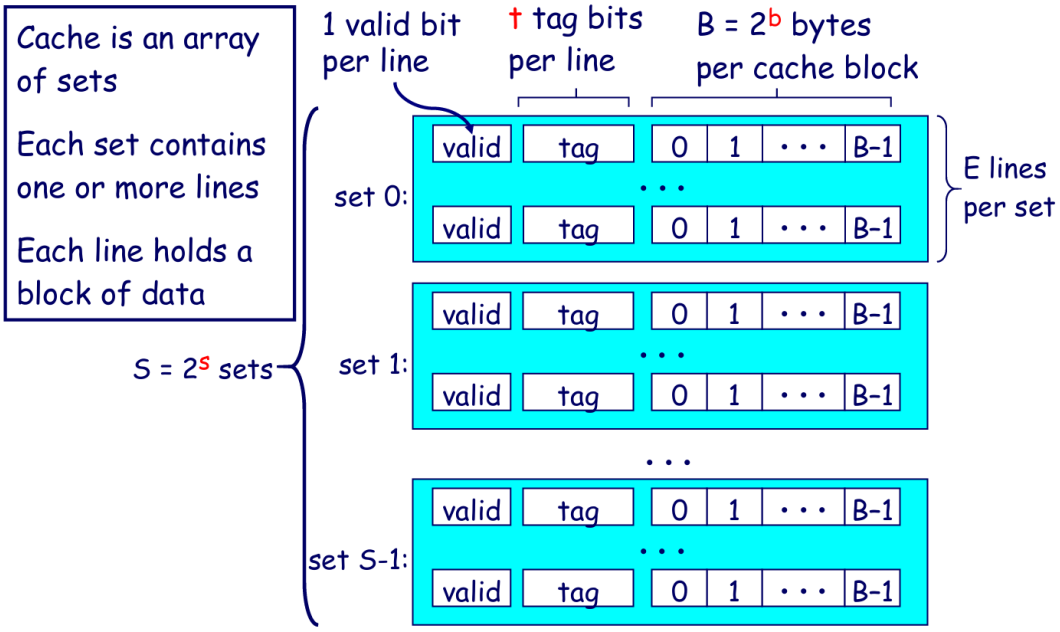
   （主存地址：21bits tag + 9bits set index + 2bits 块内地址，假设块大小=4byte，1024个缓存行，2路组相联）

   > set：组
   > way：路

2. 四路组相联



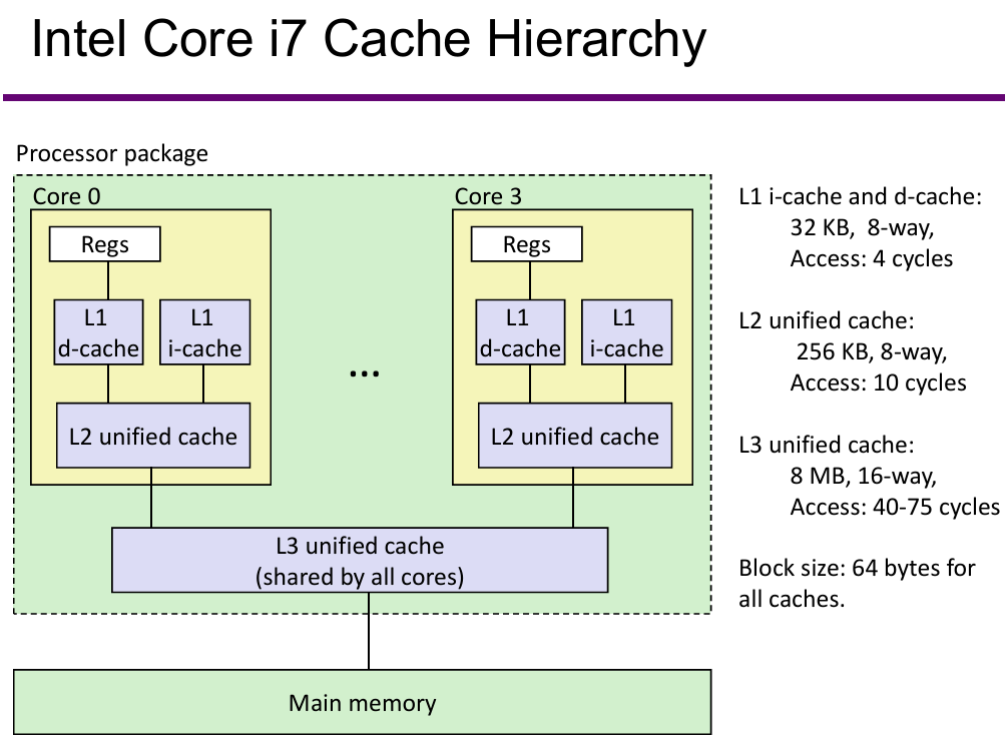
3. General Organization of a Cache：E路S组组相联



Slide from CSAPP

***主存地址：t tag bits + s bits set index + b bits inline addr*，其中的tag位是根据机器字长（地址字长）减去s和b得到的。**

存储层次结构：多级缓存（L1, L2, L3）



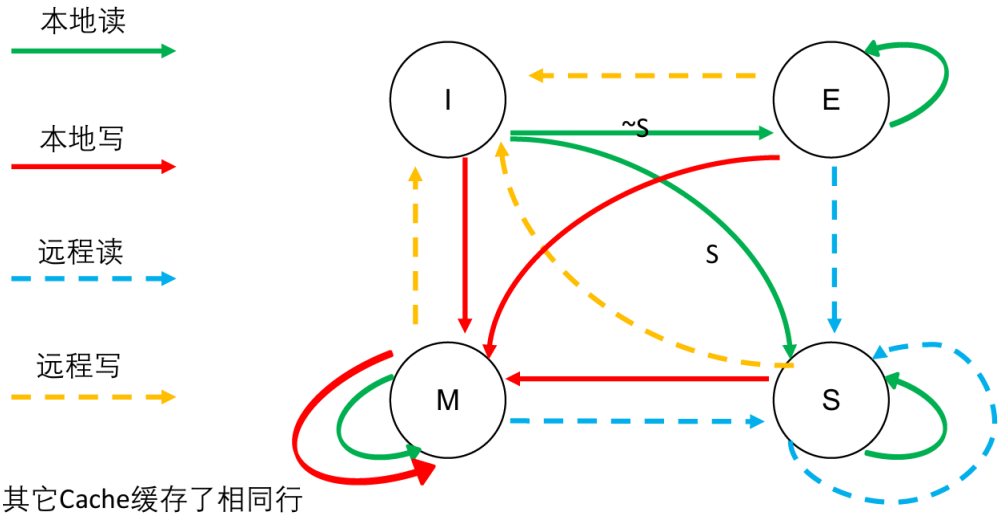
4.2 层次一致性与存储效率

- **写直达（Write Through）**：cache命中时在cache和memory都要修改
  强一致性保证，效率低
  没有cache命中：写分配（write allocate，在cache中分配），非写分配（not write allocate，不在cache中分配）
- **拖后写（Write Back）**：cache命中时只在cache中修改，替换时再写主存
  弱一致性保证；分为主动替换（指令）和被动替换（踢掉）
  实现复杂，效率比较高

** write back - no fetch on miss：直接在cache里面写，不从内存读（硬件支持write partial cache，可以标记部分有效位）*

	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

- **多核一致性保证策略（MESI）**：要保证本地cache的数据、其它核cache的数据、内存的数据有一个一致的视图
  修改态（M）：处于这个状态的cache块中的数据已经被修改过，和主存中对应的数据已不同，只能从cache中读到正确的数据
  独占态（E）：处于本状态的cache块的数据和主存中对应的数据块内容相同，而且在其它cache中没有副本
  共享态（S）：处于本状态的cache块的数据和主存中对应的数据块内容相同，而且可能在其它cache中有该块的副本
  无效态（I）：处于本状态的cache块中尚未装入数据



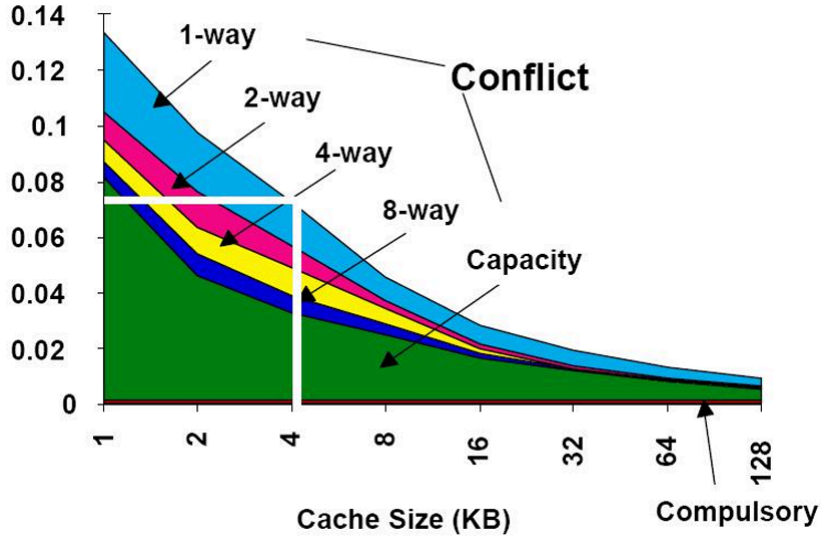
S:其它Cache缓存了相同行
~S：其它Cache没有缓存

存储访问效率：平均访问时间 = 命中时间×命中率 + 缺失损失×缺失率

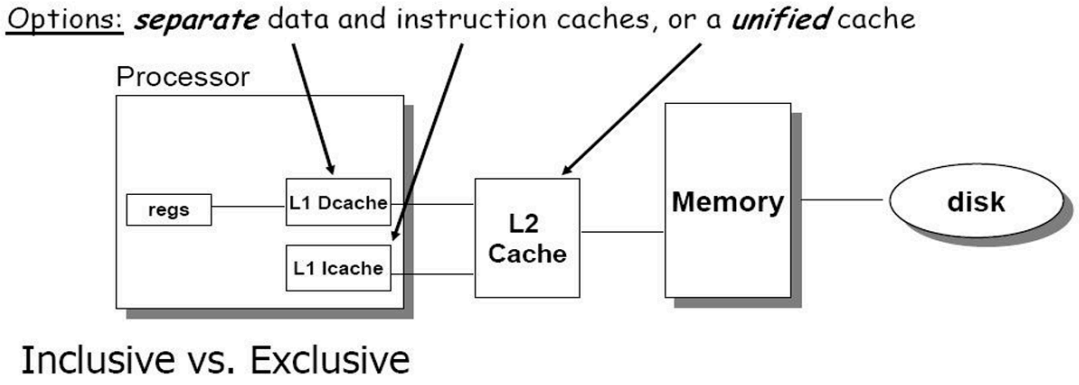
- **Cache 缺失的四类原因**
  必然缺失（Compulsory Miss）：开机或进程切换；首次访问数据块 => 预取/prefetch?
  容量缺失（Capacity Miss）：活动数据集超出了Cache的大小 => 扩容
  冲突缺失（Conflict Miss）：多个内存块映射到统一Cache块；某一Cache块已满，但空闲块在其他组
  无效缺失：其他进程修改了主存数据（此时 Cache 中是无效数据）

- **影响缺失率的因素**

  经验总结：容量为N、采用直接映射方式Cache的缺失率和容量为N/2、采用2路组相联映射方式Cache的缺失率相当



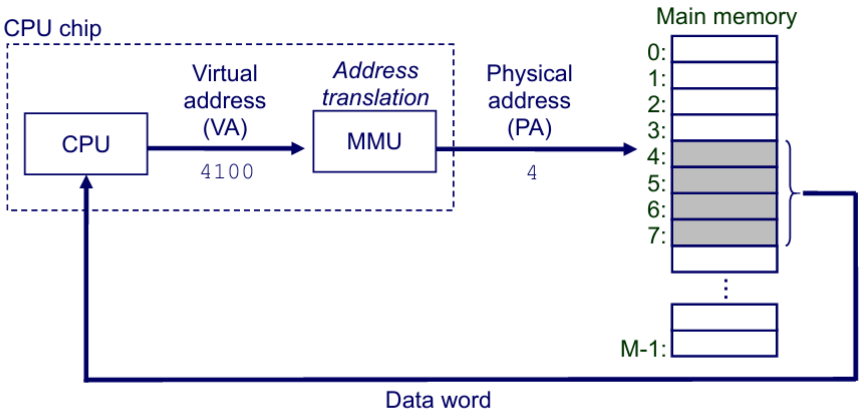
  cache容量：单调下降
  cache块大小：先降后（缓存行个数少 + 缺失损失大）
  地址映射方式：直接相连、全相联、N路组相联
  替换算法：最近最少使用LRU（Least Recently Used，硬件实现复杂但有较高命中率），先进先出FIFO，随机替换RAND（不差）
  多级cache：多级cache / 分为指令icache和数据dcache



- **Cache接入系统的体系结构：**目前基本都是片内缓存
  侧接法（CPU和cache都接入总线）、隔断法（CPU经过cache访问内存）

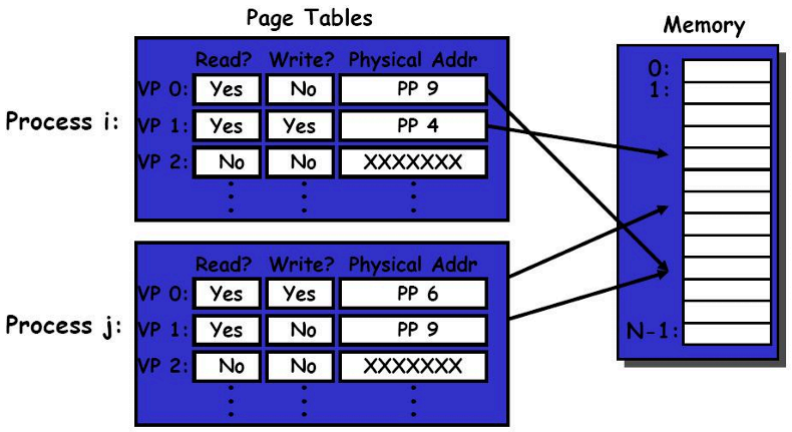
4.3 虚拟内存

页式内存管理（TLB）、段式内存管理



- 虚拟地址（逻辑地址）：程序员编程使用的地址
- 物理地址（实地址）：物理存储器的地址

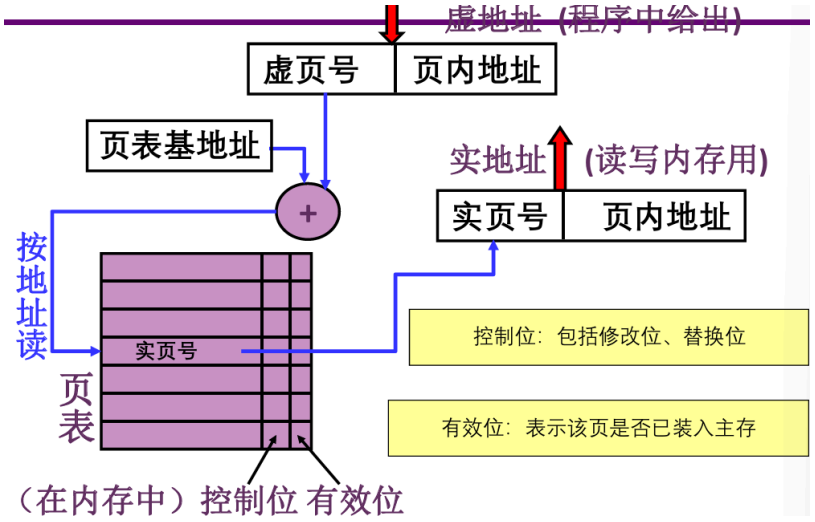
独立的逻辑地址空间
实现内存共享（进程间共享相同物理地址，e.g., read-only library code）
实现内存的保护（页表中存放访问权限）



虚拟内存是有一些“缓存”的功能的：因为可以决定hot的部分放在内存，其他的部分在外存

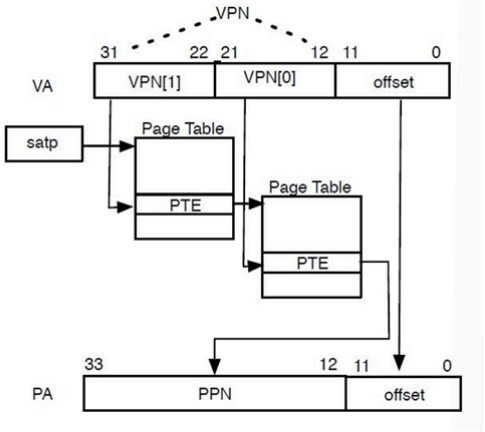
1. 页式内存管理 Page Table

> 将主存和虚存划分为固定大小的页（e.g., 4KB），以页为单位进行管理和数据交换
> 虚地址 VA = 虚页号 VPN + 页内地址 offset
> 实地址 PA = 实页号 PPN + 页内地址 offset
>
> 这个页表 / page table 会被放在内存 / Memory 里面（由页表基地址寄存器`satp`指示），有可能被cache



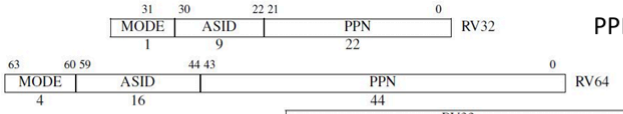
如果有效位为False（该页尚未装入主存），则调用页面失效处理程序 => 修改页面（主存未满）或进行页面替换（主存已满）

页表转换：



- **satp**

  Supervisor Address Translation and Protection / 监管者地址转换和保护
  ASID / Address Space Identifier / 地址空间标识符（可选），用以降低上下文切换开销



PPN指向根页表的物理地址

RV32		
Value	Name	Description
0	Bare	No translation or protection.
1	Sv32	Page-based 32-bit virtual addressing.

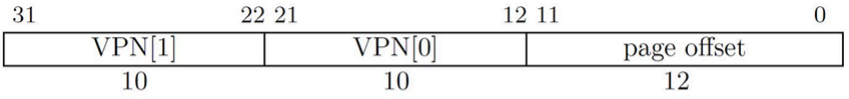
RV64		
Value	Name	Description
0	Bare	No translation or protection.
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.

MODE域编码

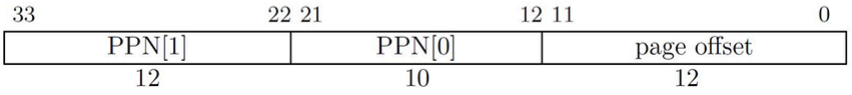
  进程切换：通过更换satp完成

- **Sv32的虚拟地址与物理地址**

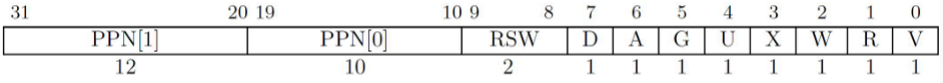
  > page offset 12位 => 2^{12} = 4KB，对应页面大小
  > 每个页表页包含2^{10} 个页表项（PTE / Page Table Entry）
  > 每个页表项的大小为 4Byte（页面大小 4KB = 4 Byte * 2^{10}）



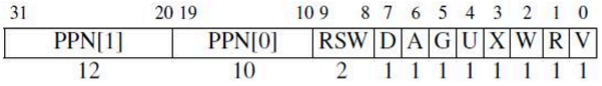
Sv32虚拟地址



Sv32物理地址

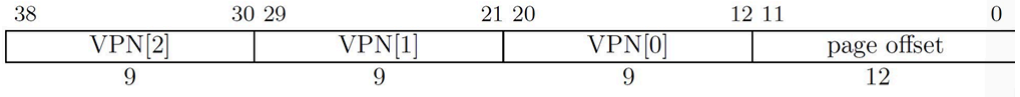


Sv32页表项

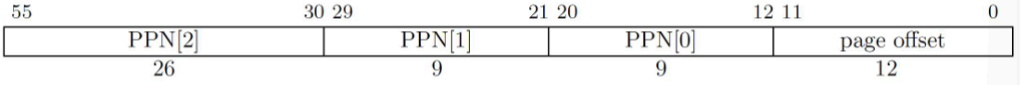


- V：有效位
- R,W,X：读，写，执行位
- U：0代表U模式不能访问，但是S模式可以；1代表U模式可以访问，S模式可以
- G：Global是否对所有地址空间有效
- A：Access，是否被访问过
- D：Dirty，是否被修改过
- RSW：操作系统使用，被硬件忽略
- PPN：物理页号，这是物理地址的一部分。如果这个页表项是一个叶子结点，那么PPN是转换后物理地址的一部分。否则PPN给出的是下一级页表的地址

- **Sv39的虚拟地址与物理地址**



Sv39虚拟地址

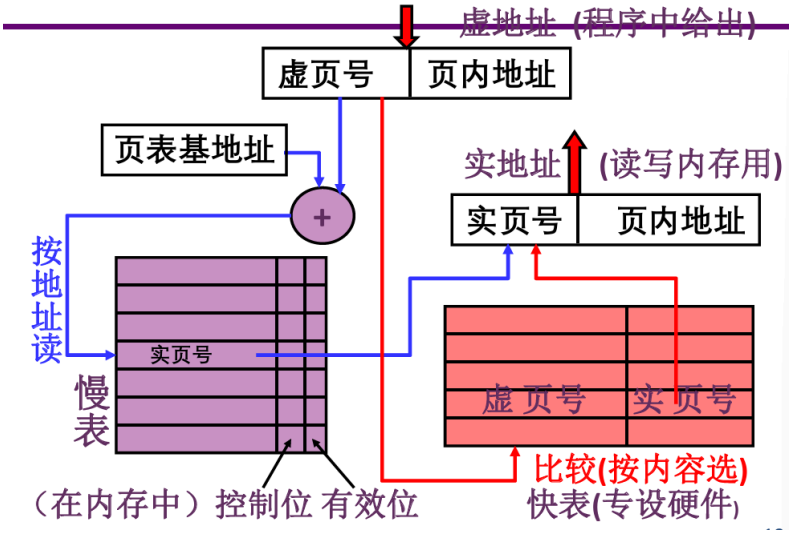


Sv39物理地址

Sv39页表项

2. 快表 TLB

加速页表查询：快表 / 转换旁路缓冲 TLB

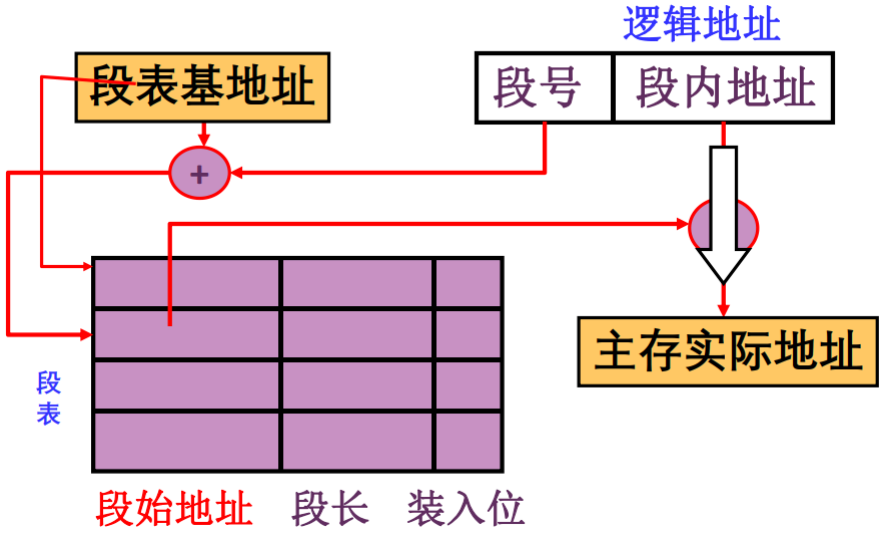


TLBs 通常为容量较小，甚至在高端计算机上也一般不超过128 - 256个表项。这样，可以使用全相连映射方式。在大多数中档计算机上，一般采用N路组相联映射方式。

进程替换：`sfence.vma`通知处理器，软件可能已经修改了页表，处理器刷新TLB

3. 段式存储管理

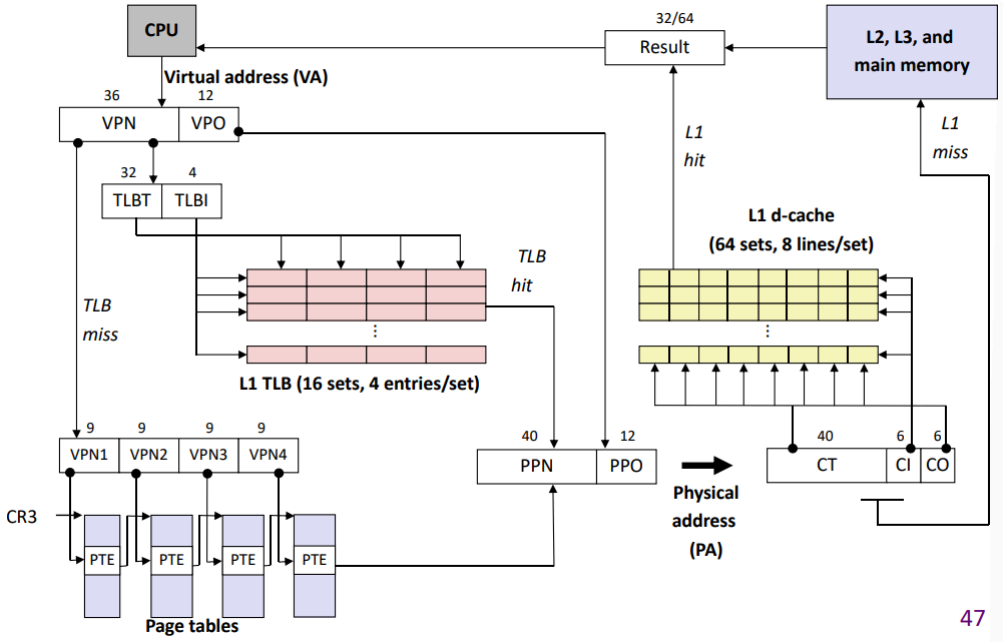
设置段表进行管理（段标记地址、段起始地址、段长、装入位、保护及共享等标志）



1. 逻辑地址空间划分为了**大小可变**的段（因此有自己的段号、段长和段基址）
   ——因此段式存储管理主要可以避免内部碎片，而页式存储管理则主要避免外部碎片
2. 支持用户程序的逻辑划分（**用户可见**）
3. 拓展：段页式虚拟内存管理
   它先把程序按逻辑单位分为段，再把每段分成固定大小的页。操作系统对主存的调入调出是按页面进行的，但它又可以按段实现共享和保护，可以兼取页式和段式系统的优点。其缺点是需要在地址映射过程中多次查表。其地址映射通过一个段表和一组页表来进行。

4. Application

1. X86-64 Linux Layout
2. End-to-end Core i7 Address Translation



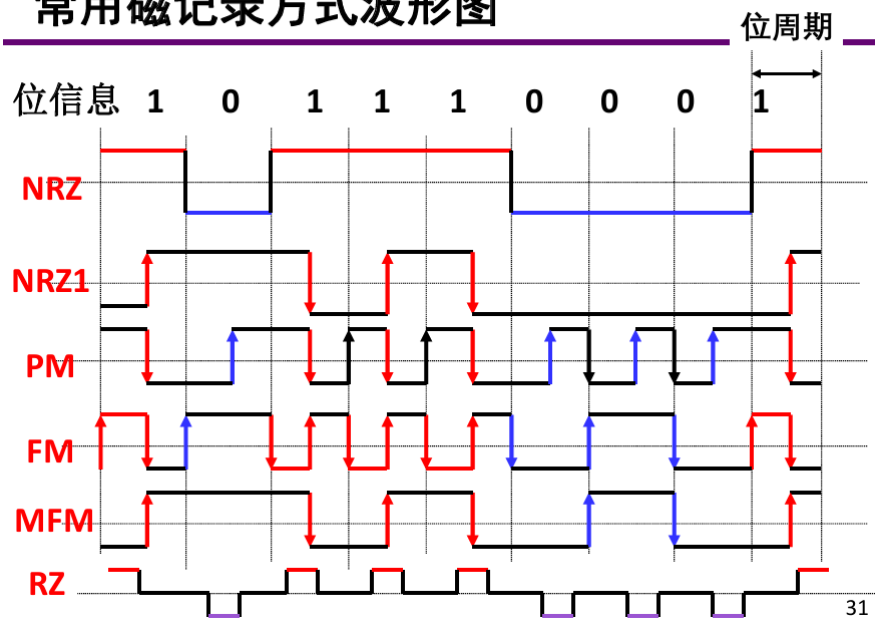
4.4 外存

非易失性存储器：磁盘/磁带、光盘、SSD/固态存储器（目前流行）——掉电后信息不丢失，访问力度大
易失性存储：静态存储器（SRAM / Cache），动态存储器（DRAM）——掉电后信息丢失，访问粒度小（字节 / 缓存）

> 随机访问： 随机访问任何单元，访问时间与信息存放位置无关；每一位都有各自的读写设备
> 串行访问：顺序地一位一位地进行，访问时间与存储位的物理位置有关；共用一个读写设备；顺序访问和直接访问

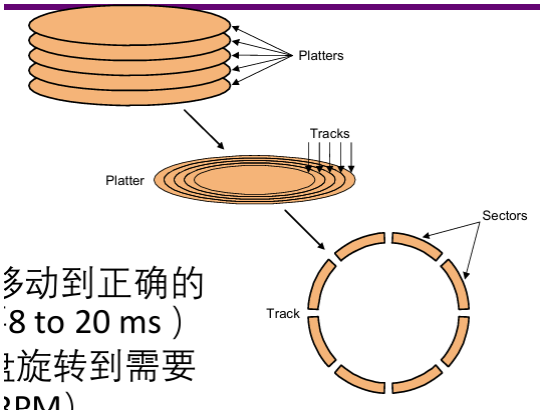
1. 磁盘 Disk

- **磁芯存储器**：大存储容量、断电后保存数据
  核心原理：磁颗粒的不同偏转方向来区分不同的状态
  主要技术指标：存储密度（单位长度或面积磁层表面所存储的二进制信息量），存储容量（二进制字节），寻址时间，数据传输率，误码率，价格
  磁记录方式：归零值RZ / 不归零制 NRZ / 见1翻转的不归零制 NRZ1；调向制 PM / 调频制 FM / 改进的调频制 MFM



硬磁盘内部结构（访问过程）：寻道（读写磁头移动到正确的磁道上~10ms => 寻找扇区~.5/RPM（扇区是磁盘访问的最小单位） => 数据传输 10MB/sec）

—— 访问时间 = 寻道时间 + 旋转延迟 + 传输时间 + 磁盘控制器延迟



柱面：位于同一半径的磁道的集合

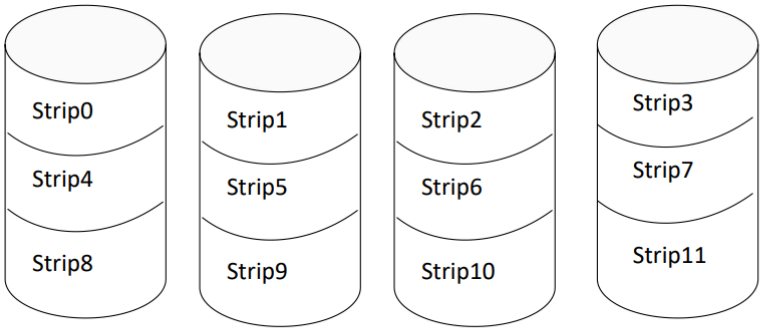
保持力（retention）：~5年

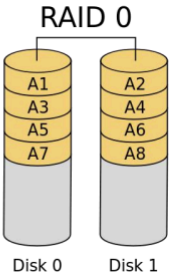
耐久性（endurance）：

- 页容量 > 单个扇区容量：可用性（扇区物理损坏时直接废弃 / 检错纠错码分布在每个扇区因此检错速度快）、灵活性（适配不同操作系统的不同页面大小）
- 可靠性：设备出现故障的几率
  可用性：系统能正常运行的几率（比如增加校验码可以提高可用性）

- **RAID盘**（Redundant Array of Inexpensive Disks / 廉价磁盘的冗余阵列）：用N个低价磁盘构成一个统一管理的阵列，以取代特贵单一磁盘（N个磁盘的容量以换取1/N的访问时间）

  RAID0：DataStriping（每带/strip对应k个扇区，数据交叉循环写入连续带中/分带）

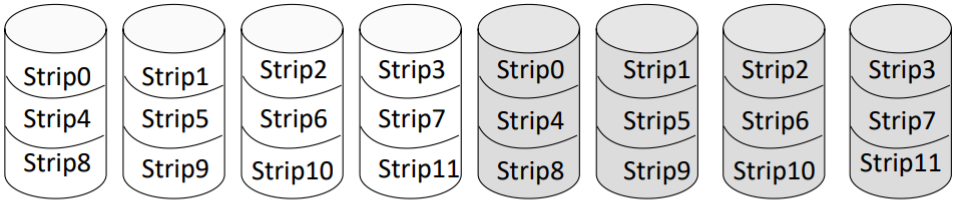


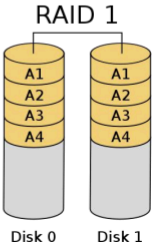


1. 适合数据请求量比较大的情况

2. 没有冗余，可靠性差，不算真正的RAID

RAID1：DriveMirroring（主辅磁盘冗余备份）



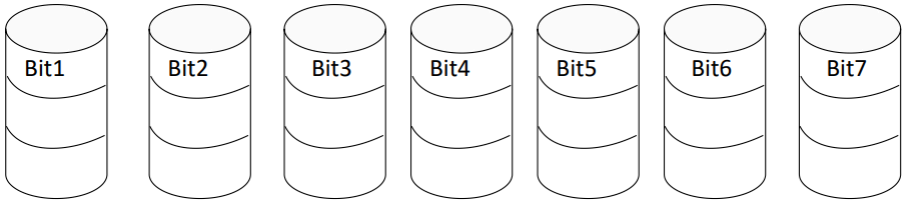


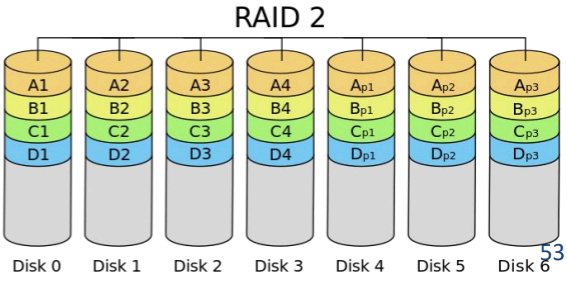
1. 冗余备份，可靠性高

2. 写性能不高，但读性能却提高了两倍

3. 成本较高

RAID2：Bit-level Striping，工作单位为字/字节，每个半字节加上3位海明码形成7位字，7个驱动器的磁头同步旋转（每个驱动器写有1位）；已被淘汰





1. 驱动器必须同步旋转

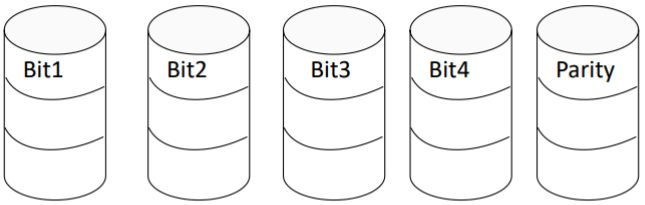
2. 驱动器个数要足够多

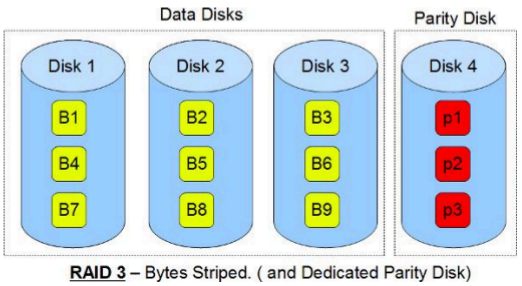
3. 需要多个控制器

RAID3：Byte Striping，RAID2的简化版本，每个字计算一个奇偶校验位（并写到一个校验驱动器上）

1. 驱动器之间要严格同步

2. 对整个磁盘崩溃的错误，能够进行恢复





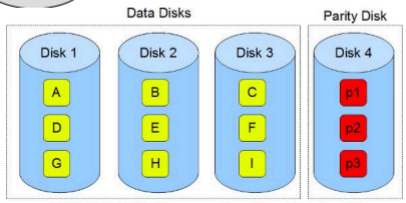
RAID 3 – Bytes Striped. (and Dedicated Parity Disk)

RAID4：(Blocks Striped) Data Guarding，奇偶校验位单独存放在校验驱动器上，按照strip计算校验

1. 不对字进行校验，也不需要驱动器同步

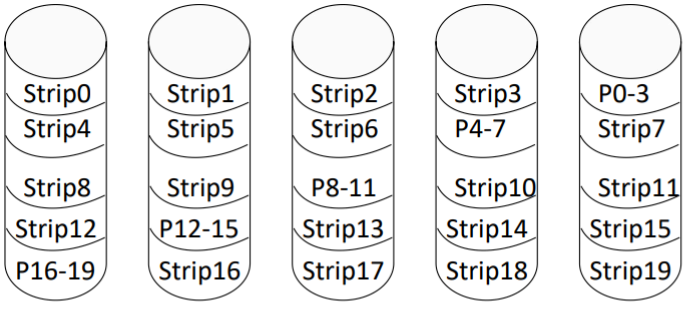
2. 可以防止整块盘崩溃，但对盘上部分字节数据出错的纠错性能相当差

3. 校验盘负载沉重

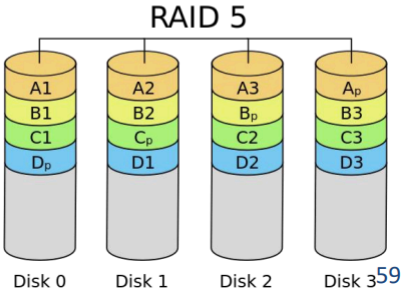


RAID 4 – Blocks Striped. (and Dedicated Parity Disk)

RAID5：Distributed Data Guarding，将校验位循环均匀分布到所有的驱动器上



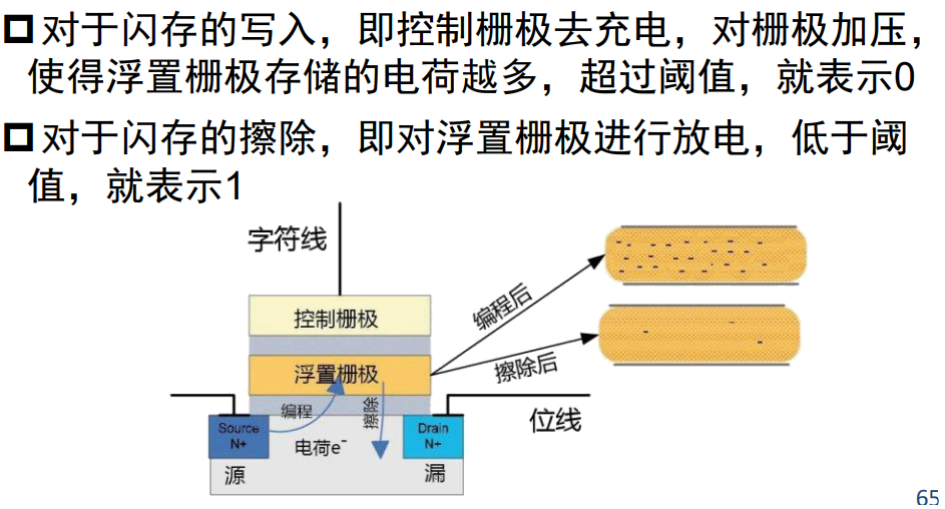
1.如果RAID 5的磁盘崩溃的话，修复磁盘内容的将是一个复杂的过程。



RAID6：二维校验（行列校验码分别循环均匀分部到所有驱动器上）

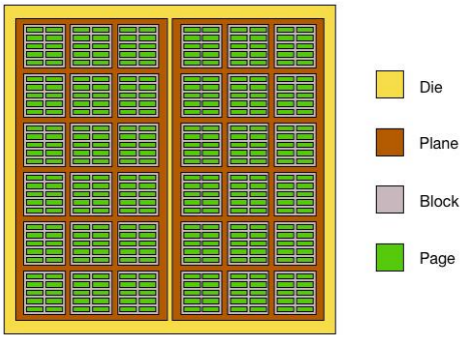
2. 固态盘 SSD

> Solid State Drive
> 没有机械结构，没有移动的部分，价格比硬盘高，有限的擦除次数
> SSD主要由**SSD控制器**，**FLASH存储阵列**，板上 DRAM（可选），以及跟HOST接口（诸如SATA，SAS, PCIe等）组成。
>
> SSD的存储单元是闪存（NAND Flash / 存储阵列），常见的类型分为SLC / MLC / TLC / QLC（Single / Multi / Triple / Quad - Level Cell，表示每个存储单元存储的比特数）



65

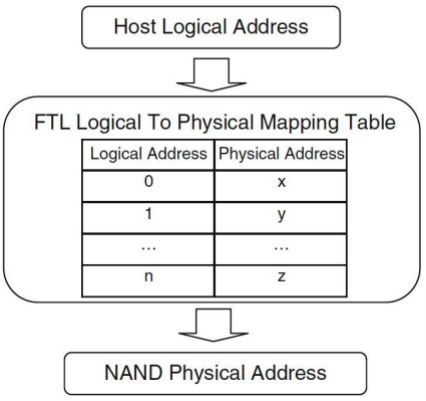
Package是NAND Flash的外部封装形式，内部包括**Die** => **Plane** => **Block** => **Page** 的内部组织结构



每个Die包含1或2个Plane，可以并行操作
每个Plane包含多个Block，Block是最小的擦除单位（~ 2/4/8MB，ms延迟）
每个Block里面有多个Page，Page是最小的读写单位（~ 4/8/16KB，us延迟）

- **SSD的写入**：写入之前需要进行擦除操作；不会写入到原来的page，FTL维护上层管理软件的逻辑地址和底层物理地址的映射关系

  FTL（Flash Translation Layer）/ 逻辑地址到物理地址的翻译：地址转换 + 磨损均衡
  动态磨损均衡：在回收和分配时选择用的次数较少的Block继续用
  静态磨损均衡：定期将静态数据占用的数据块搬移再利用



- **垃圾回收（Garbage Collection）：** 选择回收Block => 移动有效Page => 擦除Block
  垃圾回收开销 = 擦除开销 + 页面移动开销
- **SSD性能**：读数据~50us（Disk：~ms），擦除~2ms，写数据~200us（Disk：~ms）

5 外设

5.1 输入输出系统

- **程序直接控制**：CPU直接使用输入/输出指令来控制外部设备
  如“读串口”操作：CPU循环查询接口状态直到接受到字符，外设往接口数据缓冲中送字符并置状态寄存器（缺点：严重占用CPU资源）
- **程序中断**：外部设备请求，CPU响应，CPU与外设并行工作

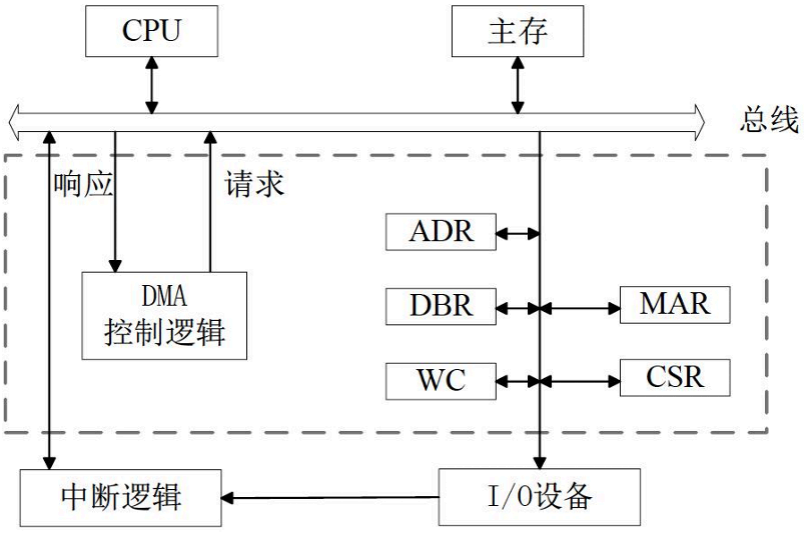
  中断源：外中断（I/O设备等），内中断/异常（处理器硬件故障或trap），中断触发器，中断状态寄存器

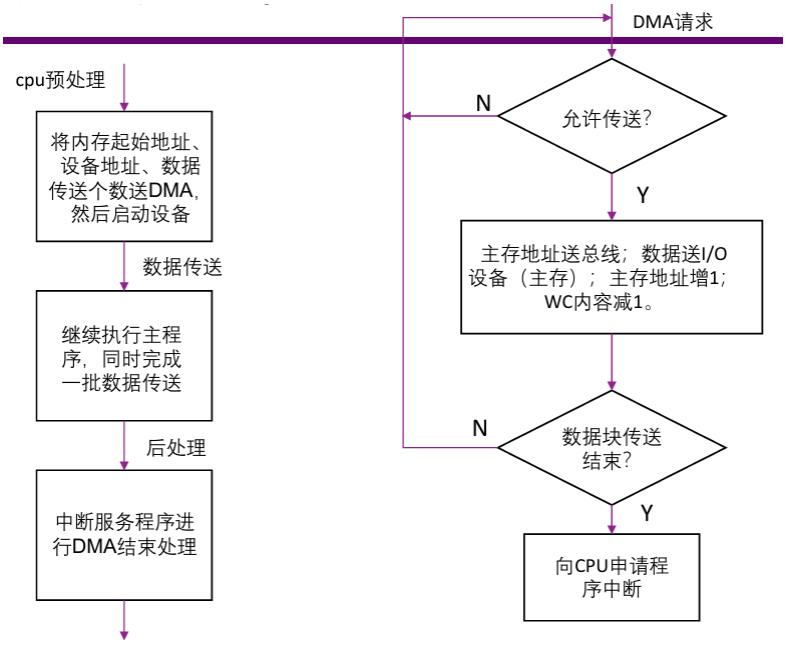
  中断设备接口组成：中断请求寄存器、中断屏蔽寄存器、优先排队线路、数据缓冲线路、中断控制和工作状态逻辑、设备选择器、中断向量表

- **直接存储访问（DMA）**：专用输入/输出控制器
  I/O设备和主存储器之间的直接数据通路，为专设的硬件，用于高速I/O设备和主存储器之间成组传送数据；数据传输过程由DMA自行控制；数据传送开始前和结束后通过程序或中断方式对DMA进行预处理和后处理。

工作方式（DMA使用内存总线的方式）：独占总线方式、周期窃取方式、DMA与CPU交替访存

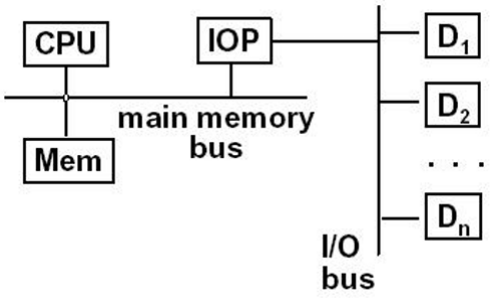
方式问题：虚拟地址 / 实地址；cache一致性





- **通道**

I/O通道是计算机系统中代替CPU管理控制外设的独立部件，是一种能执行有限I/O指令集合——通道命令的I/O处理机



通道功能：

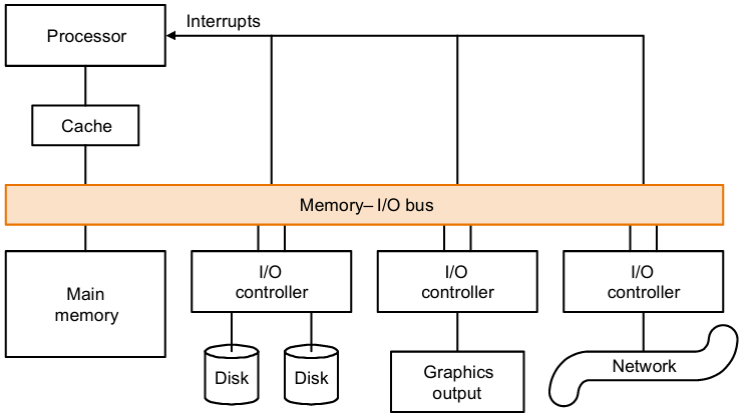
1. 根据CPU要求选择某一指定外设与系统相连，向该外设发出操作命令，进行初始化
2. 指出外设读/写信息的位置以及与外设交换信息的主存缓冲区地址
3. 控制外设与主存之间的数据交换
4. 指定数据传送结束时的操作内容，检查外设的状

通道类型：字节多路通道（分时处理的共享通道）、选择通道（外设独占+组传送）、数组多路通道（前两种方式结合）

- **外围处理机**

  通道型处理机（共享内存）
  外围处理机（通用计算机，独立完成输入/输出功能，通过通道方式与主机进行交互）

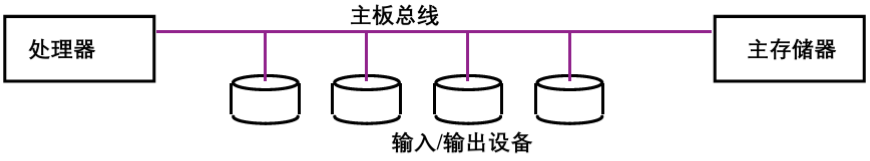
5.2 总线



计算机总线：共享的信息通道，用于连接计算机多个子系统（部件）
缺点：总线带宽限制了整条总线的吞吐量（总线长度、负载设备数、负载设备特性）

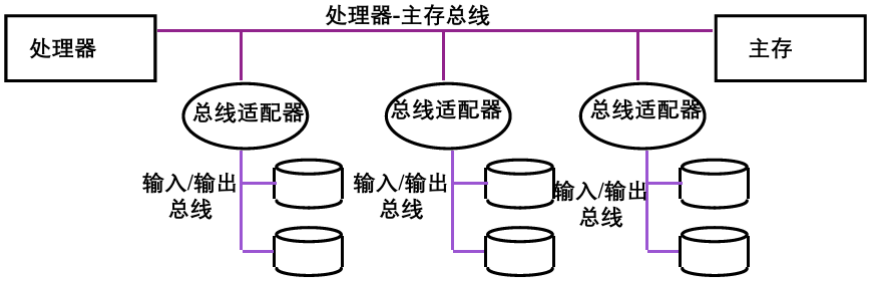
计算机总线可以分成：处理器内部总线、系统总线、I/O总线

- **单总线计算机：主板总线**



- 使用一条总线：
  - 处理器和主存储器之间通信
  - 主存储器和输入/输出设备之间通信
- 优点：简单、成本低
- 缺点：速度慢，总线将成为系统瓶颈
- 应用： IBM PC – ISA EISA、PDP-1

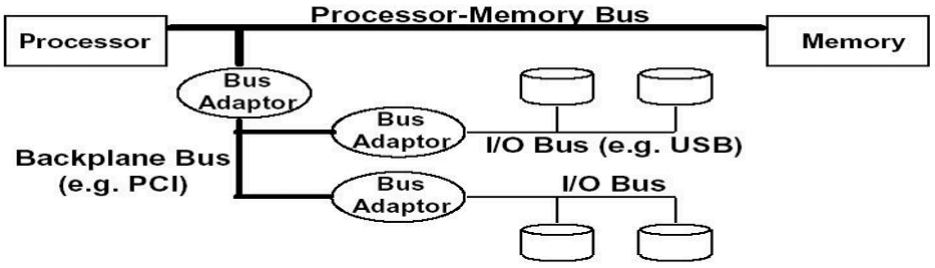
- **双总线系统**



- 输入/输出总线通过适配器和处理器-主存总线相连：
  - 处理器-主存总线：主要用于处理器和主存储器之间的通信
  - 输入/输出总线：为输入/输出设备提供信息
- 应用举例：
  - Apple Macintosh II
    - NuBus： 处理器、主存和选定的少量I/O设备
    - SCSI总线：其余I/O设备

- **三总线系统**

  处理器 - 主存总线（专用） => 主板总线（行业标准或专门设计）=> 输入/输出总线（行业标准）



- 主板总线连接到处理器-主存总线
  - 处理器-主存总线主要用于处理器和主存之间数据交换
  - I/O总线连接到主板总线
- 优点
  - 大大减少处理器-主存总线负载
- 例：现代PC基本采用的结构

1. 总线的结构

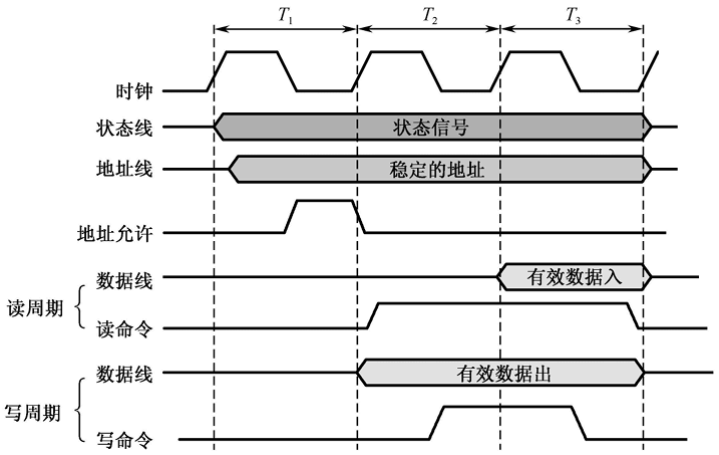
控制线：总线请求信号及数据接收信号
数据线：在源设备和目标设备间传送信息（数据和地址、复杂的命令）

- **仲裁：获得总线使用权**
  处理器作为唯一的总线主设备：避免冲突，但被卷入到每一个总线事务中
  总线仲裁：某总线主设备发出总线请求 => 总线授权 => 主设备使用完后通知仲裁器

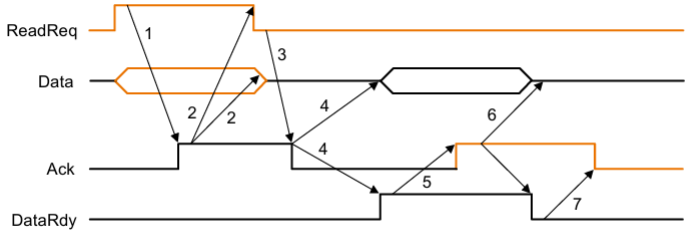
  1. 集中仲裁
     菊链仲裁（所有设备共用一个总线请求信号）：简单但无法保证公平（低优先级设备可能得不到总线使用权）、总线授权信号的逐级传递限制了总线的速度
     集中平行仲裁（通过集中的仲裁器进行）：用于几乎所有处理器-主存总线和一些高速输入/输出总线
  2. 分布仲裁：
     通过自我选择进行分布式仲裁（总线设备将自己的标识放在总线上）
     碰撞检测（以太网）

- **同步和异步总线**

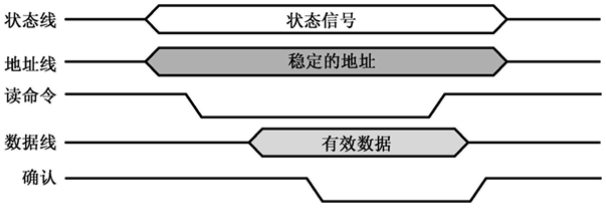
  同步总线：控制线中包含有一根时钟信号线，传输协议根据时钟信号制定；总线上所有设备必须按照时钟频率工作，高速工作时总线距离必须足够短



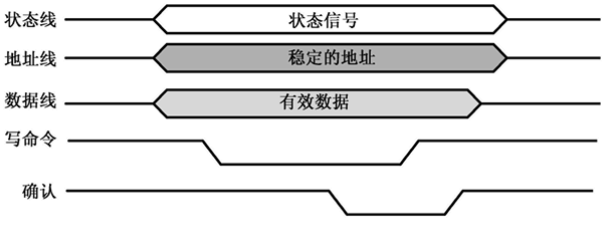
异步总线：不使用统一的时钟，使用握手协议



1. 主存储器收到外部设备发出的ReadReq信号，从数据总线读到地址，并发出Ack信号。
2. 外部设备发现Ack信号为高 =>释放ReadReq和数据
3. 主存发现ReadReq信号为低，将 Ack信号置低
4. 主存读出数据后，将数据送总线，并将DataRdy置高
5. 外部设备发现DataRdy为高，读数据，并发出Ack信号
6. 主存发现Ack为高，将DataRdy拉低，并释放数据线
7. 外部设备发现DataRdy为低，拉低 Ack信号，指示传送结束



(a) 系统总线读周期



(b) 系统总线写周期

2. 总线的性能

- **总线的带宽**（每个周期传送数据的量）

  增加总线宽度

  采用成组传送的方式：一个总线事务传送多个数据，每次只需要在开始的时候传送一个地址， 直到数据传送完毕才释放总线

  分别设置数据总线和地址总线：可同时传送数据和地址

- **多主设备总线提高事务数量**

  仲裁重叠：在当前事务时，为下一总线事务进行仲裁

  总线占用：在没有其他主设备请求总线的情况下，某主设备一直占用总线，完成多个总线事务

3. PCI总线

> Peripheral Component Interconnect / 外设部件互连：一种通用的计算机**硬件接口标准**，用于连接计算机主板上的各种外设（网卡/声卡/显卡/硬盘控制器等）和主机系统（CPU/内存）
>
> - 时钟频率：33MHz或66MHz（CLK）
> - 集中仲裁方式：集中平行仲裁 + 和上一事务重叠（REQ#，GNT#）
> - 32位地址和数据线互用：V2.1为64位（AD）
> - 总线协议：总线周期（C/BE#） + 地址握手和保持（FRAME#和IRDY#） + 数据宽度（C/BE#） + 通过握手信号传输变长的数据块
> - 最大带宽：133MB/s（33MHz）或528MB/s（66MHz）

5.3 接口电路和外部设备

> 接口：总线和外部设备的链接（总线由多个设备共享，设备之间存在差异）
> 设备：完成输入/输出任务

接口的功能：设备识别、数据缓冲、协议实现、屏蔽差异

通用可编程接口电路

串行接口芯片8251A

1. USB接口

> Universal Serial Bus：一种串行总线标准

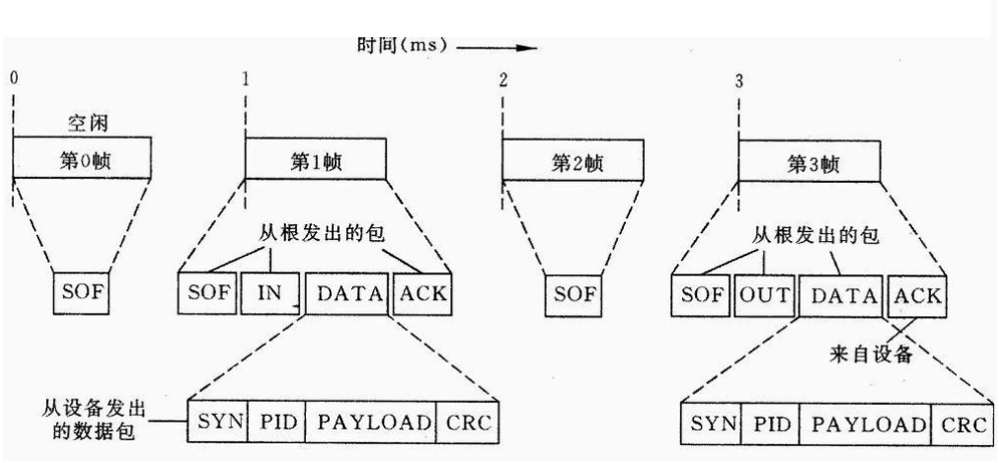
USB线缆：由4根线组成——电源、地和双数据线；采用同步传输方式
USB结构：主机（Root Hub） => Hubs => Devices 层次结构

- 设备检测：Root Hub 定时查询接口状态，若检测到有设备接入到接口上，则为该设备赋地址（7位）。设备初始地址为0，每个设备上应有ROM，保存设备参数。
- 加载设备驱动：主机操作系统根据设备类型选择对应的驱动程序
- 数据通信：采用主从模式（所有通信由主机发起），主机轮询设备状态，数据通过USB帧结构进行组织和传输

USB帧：控制帧（Control Frame - 向设备发送命令/查询状态/配置设备）、同步帧（Isochronous Frame - 实时传输数据）、块传送帧（Bulk Frame - 大量数据传输）、中断帧（Interrupt Frame - 处理设备中断请求）

USB协议：每1ms定时发出一个SOF包进行时间同步（所有设备）

- 协议包：令牌包（SOF/IN/OUT/SETUP），数据包（Data），握手包（ACK/NAK/STALL），特别包



6 复习

1. **程序是如何在硬件上运行的?**
   高级语言 => 汇编语言 => 机器语言
2. **指令在计算机内的表示?**
   操作码（指令）、操作数（立即数、寄存器编号、相对/绝对地址）
3. **数据通路和控制信号如何设计?**
4. **数据存储的需求是什么?**
   Register => SRAM => DRAM => SSD/Disk
   虚拟内存、缓存
5. **数据如何出入计算机?**
   输入输出系统、总线、接口