CS110 计算机体系结构

Lec01 课程介绍

课程信息

- 给分标准：
  - 作业(**40%**) 8次
  - Labs(**5%**) 14次，每次由线下 check 助教进行评分
  - 期中考试1(**14%**) 4.23 8:00 - 10:00，1张双面A4手写Cheatsheet，会发RISC-V绿卡，考到4.21 课后
  - 期中考试2(**14%**) 2张双面A4手写Cheatsheet，包括期中1之前的内容
  - 期末考试(**25%**) 3张双面A4手写Cheatsheet，包括期中1和2之前的内容
  - 课堂参与(**2%**)
- 作业发*Gradescope*和*Piazza*，课件和安排发布在课程官网

Great ideas in CA

- Abstraction (Layers of Representation / Interpretation) —— 抽象（表示/解释的分层）
- Moore's Law (designing through trends) —— 摩尔定律（通过趋势进行设计）
- Make the common case fast (Amdahl's Law) —— 让常见情况变快（阿姆达尔定律）
- Principle of locality (memory hierarchy) —— 局部性原理（内存层次结构）
- Parallelism (pipeline as special case) —— 并行性（流水线作为特例）
- Performance measurement and improvement —— 性能测量与提升
- Dependability via redundancy —— 通过冗余实现可靠性

Lec02 万物皆数

- 计算机内，万物皆数

数字在计算机内存储在固定大小的空间内：

- 1 bit：1位 0/1
- 1 nibble: 4 bit
- 1 byte字节: 8 bit
- 1 word: 32 bit
- 在C/C++中，`unsigned int` 使用32 bit存储，即表示 $0 \sim 2^{32}-1$ 的数

有符号整数表示法

sign-magnitude

- 首位为符号位（0正1负），其余 $n-1$ 位为数字位
- 最多表示 $0 \sim 2^{n-1}-1$ 和 $-0 \sim -2^{n-1}+1$ 的数

- 计算起来不方便，故较少使用

One's-complement representation

- 首位为符号位（0正1负），其余 $n-1$ 位为数字位
- 负数所有数字位取反（称为**反码**）
- 最多表示 $0 \sim 2^{n-1}-1$ 和 $-0 \sim -2^{n-1}+1$ 的数
- 计算起来仍然不方便，故较少使用

Two's-complement representation

- 首位为符号位（0正1负），其余 $n-1$ 位为数字位
- 负数所有数字位取反，然后加1（称为**补码**）
- `0` 被认为是正数，表示为类似于 `0000 0000` 的形式
- 思考来源：从*One's-complement representation*思考，发现其存在按数字直接相加时 `A + (-A) = 0 - 1` 的性质，故调整负数的表示方法上加1，以符合 `A + (-A) = 0` 的性质
- 最多表示 $-2^{n-1} \sim 2^{n-1}-1$ 的数
- 计算方便，是计算机内使用的表示方法

符号拓展性质

- 从较小位数拓展为较大位数时，只需要将最高位（符号位）复制，向前填充满数位即可：
- 例：4-bit `3` 为 `0011`，拓展为 8-bit `3` 为 `0000 0011`
- 例：4-bit `-3` 为 `1101`，拓展为 8-bit `-3` 为 `1111 1101`

加法：直接数字相加，并检验溢出 Overflow：

- 正负数相加，不可能出现溢出 Overflow
- 两个正数/两个负数相加，校验符号位（最高位）是否变化；若变化则出现溢出 Overflow

小数表示法

定点数 Fixed-point numbers

- 表示为 `P.Q` 的形式，表示小数点前 `P` 位，小数点后 `Q` 位的小数
- 例：`0010.1010` 的十进制为 $1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3}$，即 `2.625`
- 加减法方便，乘法需要小数点移位
- 表示较大的数与更高精度的数不方便调整
- 在定制化系统中较多采用

浮点数 Floating point

- 启发于科学计数法的表示方式
- 基于 *IEEE 754 standard* 规定的标准

32位单精度浮点数(FP32)

- 1位符号位 `Sign` (0正1负) + 8位指数位 `Exponent` (无符号，带 `-127` 的偏置bias，$-126 \sim 127$，`0000 0000` (E = -127) 或 `1111 1111` (E = 128) 是特殊规则处理的) + 23位尾数 `Mantissa`（省略小数点前的1）

- 实际符号为 $S = (-1)^{Sign}$
- 实际指数为 $E = Exponent_2 - 127_{10}$
- 实际小数位为 $M = 1.Mantissa$
- 实际数值为 $Value = S \times 2^E \times M$

例：32-bit单精度浮点数 `0 0111 1011 110 0000 0000 0000 0000 0000` 转换为十进制

- 符号位为 `0`, 表示正数，$S = 1$
- 指数位为 `0111 1011`，表示 $E = -4$
- 尾数为 `110 0000 0000 0000 0000 0000`, 表示小数位
  $M = (1.11000000000000000000000)_2 = (1.75)_{10}$
- 故原数为 $Value = S \times 2^E \times M = 0.109375$

例：`0.09375` 转换为32-bit单精度浮点数

- `0.09375` 是正数，故符号位 `Sign` 为 `0`
- `0.09375` 转换为二进制为 $(0.00011)_2$ （每步取小数部分乘2，依次取首位为小数点后第1,2,3,4,5位）
- 转换为二进制科学计数法表示：$(0.00011)_2 = 1.1_2 \times 2^{-4}$
- 确定偏移后指数位 `Exponent`：$-4_{10} + 127_{10} = 123_{10} = 01111011_2$
- 确定去掉小数点前的1后的尾数：`Mantissa = 100 0000 0000 0000 0000 0000` （末尾补0）
- 故浮点数表示为 `0 0111 1011 100 0000 0000 0000 0000 0000`

特殊规则：∞

- 拓展绝对值极大的数域
- 规定 $\pm\infty$ 的表达为：`0/1 1111 1111 000 0000 0000 0000 0000 0000`
- 符号位 `Sign` 表示正(0)负(1)
- 指数位 `Exponent` 为 `1111 1111`
- 尾数 `Mantissa` 为 `000 0000 0000 0000 0000 0000`
- 可能由 ∞ 计算而来，也可能由空间压缩的类型转换、除以0、0取对数、数值上溢 Overflow 得到

特殊规则：0

- 规定 ± 0 的表达为：`0/1 0000 0000 000 0000 0000 0000 0000 0000`
- 符号位 `S` 表示正负
- 指数位 `E` 为 `0000 0000`
- 尾数 `Mantissa` 为 `000 0000 0000 0000 0000 0000`

特殊规则：非规格化数 Denorm / Subnormal

- 拓展绝对值极小的数域
- 规定指数位 `E` 为 `0000 0000`，而尾数 `Mantissa` 不全为 0
- 默认指数位 `E` 表示 `-126`，小数位 `M = Mantissa` 而不再添加首位 1 （`0.Mantissa`）
- 可拓展表示出 $2^{-149} \sim 2^{-126} - 2^{-149}$ 和 $-2^{-149} \sim -2^{-126} + 2^{-149}$

非数字：NaN (Not a Number)

- 指数位 `E` 为 `1111 1111`，而尾数 `Mantissa` 不全为 0
- 由 `NaN` 计算而来（这被称为**quiet invalid operations**），或由非法运算计算而来（$0 \times \infty$, $\sqrt{n}$ for $n < 0$, $\frac{0}{0}$, $\frac{\infty}{\infty}$）

浮点数计算

舍入

- 默认采取**四舍六入五成双**规则
- 也存在上取整、下取整、向0取整、四舍五入等其它取整方式

加法

- 对齐移位 Preshift / Alignment shift：调整较小的数，使加数的指数相等
- 小数位相加 Add the significands：小数位 `M` 直接相加，但可能存在未规格化的小数位 `M`（即不是 `1.***` 形式的小数位）
- 归一化移位 Normalization shift/postshift：把结果做规格化（即变为 `1.***` 的形式）
- 舍入 Round：结果超过精度（超过23位）的尾数
- 如果舍入造成了未归一化，继续归一化后再重新舍入（最多再执行一次，必定归一化）

乘法

- 还原真实指数，并计算新的指数，加偏移再转换为 `E`（考虑偏置bias `-127`）
- 小数位相乘，可能存在未规格化的小数位 `M`
- 归一化移位
- 舍入
- 如果舍入造成了未归一化，继续归一化后再重新舍入（最多再执行一次，必定归一化）
- 计算符号位

Lec03 C语言

C语言介绍

- C语言是几乎在所有平台都可以使用的语言
- 有更完整的变量类型定义
- 更好的内存管理
- 需要编译过程（而非解释执行），不同设备上编译生成的文件不同
- 编译过程有优化；只会重新编译修改的文件

C语言是如何工作的

执行全过程

- 高级语言：例如C语言
- 编译 $\rightarrow$ 汇编语言：例如RISC-V（指令文本）
- 汇编器 $\rightarrow$ 机器语言：RISC-V（二进制命令）

- CPU通过机器解释(Machine Interpretation)来执行二进制命令
- 架构实现：逻辑电路描述(Logic Circuit Description)

编译全过程

- `main.c` C语言源文件文本
- `Pre-processing` 预编译过程，展开预编译指令（如 `#include` `#define` 等）
- `Parsing & Semantic Analysis` 语法分析，生成抽象语法树(Abstract Syntax Tree, AST)
- `Code generation & Optimization` 代码生成与优化，将抽象语法树表达为中间表达(intermediate representation)，然后生成汇编语言代码（`.s`，如 RISC-V）
- `Assembler` 汇编器，将汇编代码转换为二进制机器码
- `main.o` 二进制机器码文件
- `Linker + lib.o` 链接预编译的库文件
- `main.out` 机器码可执行文件

C语言概览

- 在宿主环境（有操作系统）下，C语言必须从 `main` 函数开始执行（加载器决定）
- 在非宿主环境（无操作系统）下，C语言不一定从 `main` 函数开始执行（无加载器，直接加载特定内存地址的代码）
- `false` 判定：`0` 或 `NULL`；其余均为 `true`
- 没有显式的 `bool` 类型（在 `stdbool.h` 有）

Lec04 C语言内存管理

指针

- 指针类型的大小与机器有关，在本课程中统一为 `4 byte(32 bit)`，即默认使用32位计算机
- `char` 类型大小为 `1 byte`，`int` 类型大小为 `4 byte`

数组

- 定义时必须知道需要多大的空间来存储，因此需要知道数组的长度
- 实际存储时，会使用连续的内存单元格，靠前的元素会占据更低的地址
- 在C语言中，数组几乎等效于指针，如 `ar[2]` 等效于 `*(ar+2)`
- 但是也有例外：

```
char string1[] = "abc"; // 可以修改
char *string2 = "abc";  // 不能修改
```

- 主函数传参：

```
int main(int argc, char *argv[])
```

其中 `argc` 表示参数数量Argument Count（即命令行上空格隔开字符串的数量），`argv` Argument Vector为一个指向这些字符串指针的数组。例如：

```
% clang -ansi test001.c -o test001.out
```

此时 `argc = 5`, `argv = ["clang", "-ansi", "test001.c", "-o", "test001.out"]`

- 数组作为参数传入函数时，**务必**同时传入数组长度，否则如果发生越界访问，会极难发现

字符串

- 在字符串尾部会添加 `\0` 来标记字符串长度
- 特殊地，有：

```
char s[] = "abc", t[3] = "abc";
// 等价于
char s[] = {'a', 'b', 'c', '\0'},
     t[] = {'a', 'b', 'c'};
```

指针运算

- 硬件的存储是按照 `8 bit` 为基础单元的，因此数据需要对齐(Alignment)
  - 对于整型 `int` 数据(`4 byte`)，要求存储时需要最低位存储在 `4 byte` 的整倍数的内存地址上
  - 对于短整型 `short` 数据(`2 byte`)，要求存储时需要最低位存储在 `2 byte` 的整倍数的内存地址上
- `sizeof()` 可以传入变量或数据类型，输出该变量或数据类型占据的字节数(byte)（但是byte和bit的换算是非标准定义的）

```
int foo (int array[], unsigned int size)
{
    printf("%d\n", sizeof(array));  // sizeof(int*) = 4 byte
}

int main (void)
{
    int a[10];
    foo(a, 10);
    printf("%d\n", sizeof(a));  // sizeof(int[10]) = 40 byte
}
```

- 指针加减会自动使用指针对应类型的单位偏移量

```
int arr[] = {3,5,6,7,9};
int *p = arr;
int (*p1)[5] = &arr;
printf("%d\n", *(arr+1));   // 向后 sizeof(int) 字节 = 5
printf("%d\n", *(p+1));     // 向后 sizeof(int) 字节 = 5
printf("%d\n", *(p1+1));    // 向后 sizeof(int[5]) 字节 = undefined
```

- C语言定义数组越界1位处必为合法地址，但是对这个地址解引用是非法的
- 指针定义的运算：
  - 加减整数
  - 指针作差

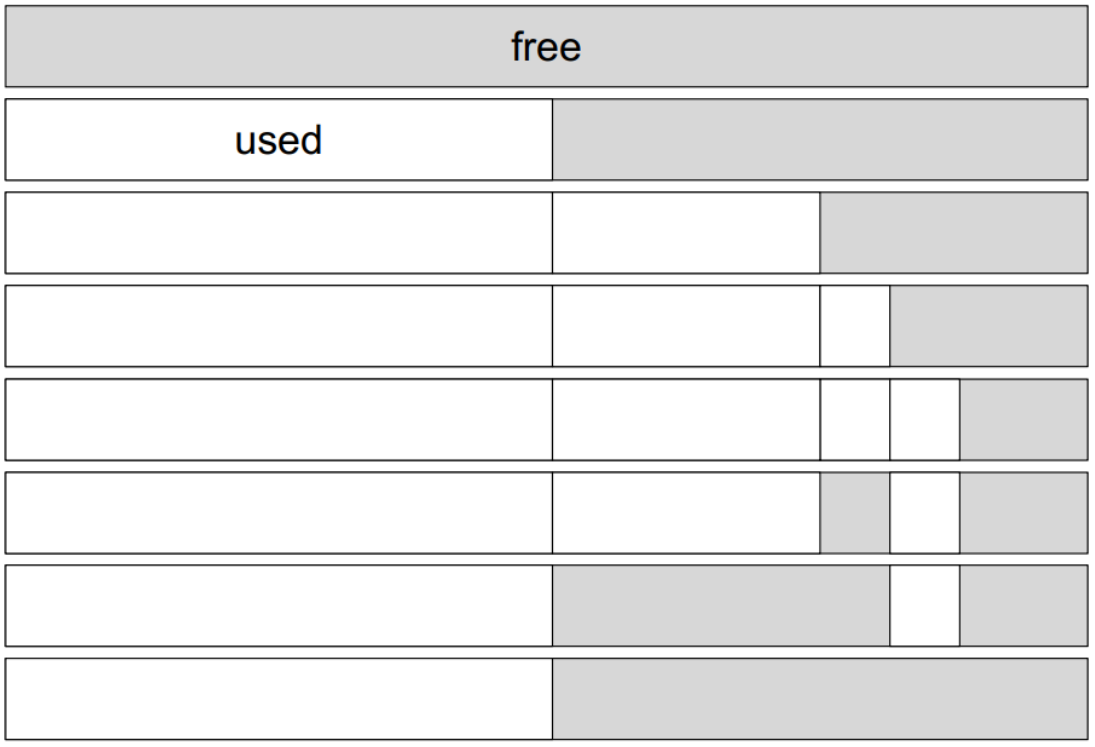
- 指针比较
- 和 `NULL` 比较

内存管理

- C语言的内存分为4个区域，分别为：栈区 stack，堆区 heap，静态数据区域 static data，代码区 code
- 栈区 stack：存储连续地址的栈帧（包含函数的返回地址（调用者）、函数参数、函数内部定义的变量）；函数被调用时入栈，函数终止时弹栈
- 堆区 heap：运行时分配不连续内存的区域，支持 `malloc()` `calloc()` `free()` `realloc()` 手动分配空间

```c
int *ip1;
ip1 = (int*)malloc(sizeof(int));    // malloc() 返回类型为 void*
if (!ip1) // 处理内存不足，分配失败，ip1 = NULL
{
    fprintf(stderr, "Error when allocating for ip1!\n");
    return 1;
}
free((void*) ip);   // 不能free如 (ip1+1) 等非malloc直接请求的空间
```

分配空间时，会默认以上取整2的幂次大小来分配，防止内存碎片化



- 静态数据区域 static data
  - .rodata区：read-only，存储字符串字面量、被初始化的const修饰的全局/静态变量
  - .data区：存储已初始化且非零的全局变量/静态变量
  - .bss区：Block started by symbol，存储未初始化的全局/静态变量，或显式初始化为0的全局/静态变量
- 代码区 code：存储代码，运行时一般只读

小端序 Little Endian & 大端序 Big Endian

- 表示字节 Byte 的存储顺序（每个字节内部的8位固定按照从左往右存）
- 小端序：最低位(Byte 0)在最右侧（RISC-V使用小端序）
- 大端序：最低位(Byte 0)在最左侧
- 具体顺序由CPU管理

Lec05 RISC-V (I)

ISA

- 指令集体系结构
- ISA 是计算机抽象模型的一部分，它明确了软件（如程序、操作系统）如何向 CPU 发送指令、控制 CPU 的行为。同时，ISA 充当了硬件（CPU、内存、外设等物理组件）和软件之间的“桥梁”：软件通过 ISA 规定的指令和规则与硬件交互，硬件则按照 ISA 定义的规范响应软件的操作。

常见的ISA

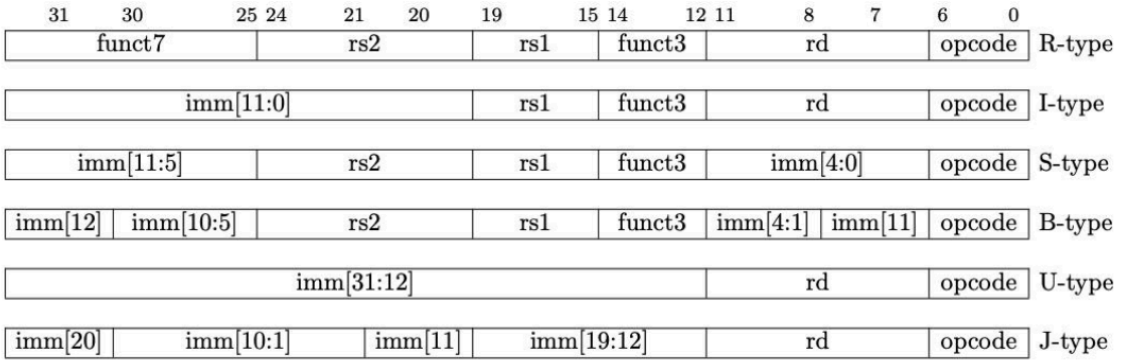
- CISC：complex instruction set computer，复杂指令集计算机，是早期计算机的架构。早期计算机编译功能有限，因此需要复杂的指令集来完成功能
- RISC: reduced instruction set computer，精简指令集计算机，硬件设计更简单，产热更少。每条指令的长度是固定的（如 `32 bit`）
- X86：Intel等
- ARM：Apple，Huawei等
- RISC-V: 是UC Berkeley在2010年启动的项目，开源，有可扩展性。本课程以 `RV32I` 为例

Name	Description
Base	
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 register
RV64I	Base Integer Instruction Set, 64-bit
RV64E	Base Integer Instruction Set (embedded), 64-bit
RV128I	Base Integer Instruction Set, 128-bit
Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point

- RISC-5的指令不能在X86的CPU上运行，而ARM可以在Linux OS上正常编译运行

RISC-V

- 每一行都是一条指令
- 使用 `#` 来注释，记得写注释!
- 不能使用变量，可以操作寄存器
- 有32个寄存器，分别命名为 `x0` `x1` ... `x31`，其中 `x0` 的值始终为0
- 有一个特殊的寄存器 PC (Program Counter)，记录程序执行到哪里
- 根据功能，把指令分成6个类别：



R-Type

- 对寄存器进行操作的指令，用编号表示寄存器 register
- 加法：`add rd, rs1, rs2` (符号，目标寄存器，寄存器1，寄存器2)
  - `add x5, x2, x1` 把 `x1` 和 `x2` 寄存器内的值直接相加，并存储在寄存器 `x5` 内
  - 有进位；会忽略溢出位
- 减法：`sub rd, rs1, rs2`
  - `sub x5, x2, x1` 把 `x2 - x1` 的值存入 `x5` 中
- 逻辑操作：`and/or/xor rd, rs1, rs2`
  - `xor x6, x1, x5` 把 `x1 ^ x5` 存入 `x6` 中
  - 通过与 `0x FFFF FFFF` 取异或来达到取反目的
- 比较：`slt/sltu rd, rs1, rs2` (set if less than for signed/unsigned)
  - `slt x5, x2, x1` 用有符号类型（2's complement）解读，如果 `x2 < x1`，则把 `x5` 设为 `1`，否则设为 `0`
  - `sltu x6, x5, x3` 用无符号类型（直接读数）解读，如果 `x5 < x3`，则把 `x6` 设为 `1`，否则设为 `0`
- 移位操作：`sll/srl/sra rd, rs1, rs2` (shift left/right logic/arithmetic)
  - `sll x5, x2, x4` 把 `x2` 左移 `x4` 位，并存入 `x5` 中
  - `srl x6, x1, x4` 把 `x1` 右移 `x4` 位，用0填充左侧高位，并存入 `x6` 中
  - `sra x7, x3, x4` 把 `x3` 右移 `x4` 位，用原符号位（最高位）填充左侧高位，并存入 `x7` 中
  - 移位次数范围为 `[0, 31]`

I-type

- 对寄存器和立即数操作的指令 immediate
- 其中立即数(imm)只有12位存储
- 加法：`addi rd, rs1, imm`
  - `addi x5, x4, -10` 把 `x4 + (-10)` 存入 `x5` 中
- 类似地， 定义 `andi` `ori` `xori` `slti` `sltui`
  - `sltui` 会把立即数也识别成无符号模式，因此立即数为负数会出现大数比较
- 移位：`slli/srli/srai rd, rs1, imm`
  - 只移位 `imm` 的最低的5位的值

- 无操作：`no-op` 等效于 `addi x0, x0, 0`
- 加载字：`lw rd, imm(rs1)` (load word)
  - `lw x1, 12(x4)` 从地址 `x4 + 12` (Byte)开始，读取连续的 `4 byte = 32 bit` 存入 `x1` 中
- 加载字节：`lb/lbu rd, imm(rs1)` (load byte for signed/unsigned)
  - `lb/lbu x1, 12(x4)` 从地址 `x4 + 12` (Byte)开始，读取连续的 `1 byte`，以有符号（原最高位填充高位）/无符号（0填充高位）模式拓展为 `4 byte` 存入 `x1` 中
- 加载半字：`lh/lhu rd, imm(rs1)` (load halfword for signed/unsigned)
  - `lh/lhu x1, 12(x4)` 从地址 `x4 + 12` (Byte)开始，读取连续的 `2 byte`，以有符号（原最高位填充高位）/无符号（0填充高位）模式拓展为 `4 byte` 存入 `x1` 中
- 加载地址：`la rd, variable` (load address，伪指令，可近可远)
  - `la x1, test_input` 把变量 `test_input` 的地址存入寄存器 `x1` 中
- 跳转与链接寄存器 `jalr rd, imm(rs1)` (jump and link register)
  - `jalr x1, imm(ra)` 跳转到 `(ra + imm)&~1`，并保存返回地址 `PC+4` （即下一条指令的地址）到 `x1`
  - `jalr x0, 0(ra)` / `ret` / `jr ra` 无条件跳转到命令计数 `ra` ，一般在函数返回时使用

S-type

- 将寄存器数据储存到内存的指令 save
- 保存字：`sw rs2, imm(rs1)` （save word）
  - `sw x1, 12(x4)` 将 `x1` 的值(`4 byte = 32 bit`)存入内存地址 `x4 + 12` 处
- 保存半字：`sh rs2, imm(rs1)` (save halfword)
  - `sh x1, 12(x4)` 将 `x1` 的值的低位 `2 byte = 16 bit` 存入 `x4 + 12` 处
- 保存字节：`sb rs2, imm(rs1)` (save byte)
  - `sb x1, 12(x4)` 将 `x1` 的值的低位 `1 byte = 8 bit` 存入 `x4 + 12` 处
- 强烈建议（但是语法不强制要求）存储对齐：字保存在 `4 byte` 的整数倍地址上，半字保存在 `2 byte` 的整数倍地址上

Lec06 RISC-V (II)

B-type

- 分支执行的指令 branch
- 取等跳转：`beq/bne rs1, rs2, label` (branch if equal/not equal)
  - `beq x1, x2, loop` 如果 `x1 == x2`，则跳转到标签 `loop` 处
  - `bne x1, x2, loop` 如果 `x1 != x2`，则跳转到标签 `loop` 处
- 比较跳转：`blt/bltu/bge/bgeu rs1, rs2, label` (branch if less than/greater or equal to for signal/unsignal)
  - `blt x1, x2, loop` 如果按有符号数值比较 `x1 < x2`，则跳转到标签 `loop` 处

函数调用 Function call

函数调用惯例 Calling Convention

- `x1` Return Address 调用者在函数执行后的下一行代码地址
- `x2` Stack Pointer 函数栈最低位（栈顶）地址
- `x3` / `x4` Global pointer/Thread pointer 操作系统相关指针（不用了解）
- `x10 ~ x11` Function arguments/Return values 函数参数/返回值
- `x12 ~ x17` Function arguments 函数参数

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller

调用函数的过程

- 把参数放在函数可以获取的位置
- 把控制权交给函数
- 获取函数必需的本地存储资源
- 函数运行
- 把函数返回值放在调用代码可以获取的位置
- 把控制权返还给调用者代码

J-type

- 跳转与链接 `jal rd, label` (jump and link)
  - `jal x1, func` / `call func` / `jal func` 跳转到标签 `func`，并把返回地址 `PC+4` （即下一条指令的地址）保存到 `x1`(即 `ra`)
  - `jal x0, loop` / `j loop` 无条件跳转到标签 `loop`

栈帧 Stack frame

- 在程序的内存的栈区里
- 操作
  - 入栈 Push
  - 弹栈 Pop
- 栈指针 Stack pointer 指向当前栈顶（栈区最低地址，即已使用/未使用栈区的分界线），存储在寄存器 `sp`(即 `x2`)

函数调用时寄存器的管理

- 调用函数 Caller 管理的寄存器：`ra`, `t0-t7`, `a0-a7`
- 被调用函数 Callee 管理的寄存器：`sp`, `s0/fp`, `s1-s11`
- 谁管理的寄存器，谁就要负责在函数调用结束后把寄存器的值恢复原状
    - Caller 先把参数放在 `a0-a7` 里（如果寄存器不够，开栈区空间存储参数）
    - 如有必要，Caller 把 caller-saved 寄存器的值压入栈区中
    - 使用 `jal`/`jalr` 来跳转到 Callee 函数，此时 `ra` 被设置为Caller的下一行指令的地址
    - 改变 `sp` 来把 callee-saved 寄存器的值压入栈区
    - 如有必要，改变 `sp` 来把本地需要存储的变量压入栈区
    - Callee 函数执行
    - 把返回值放在 `a0-a1` 中
    - 把 callee-saved 寄存器的值从栈区弹出，恢复寄存器的值
    - 返回Caller调用点 `ra`
    - Caller 恢复 caller-saved 寄存器的值

```c
int Leaf(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

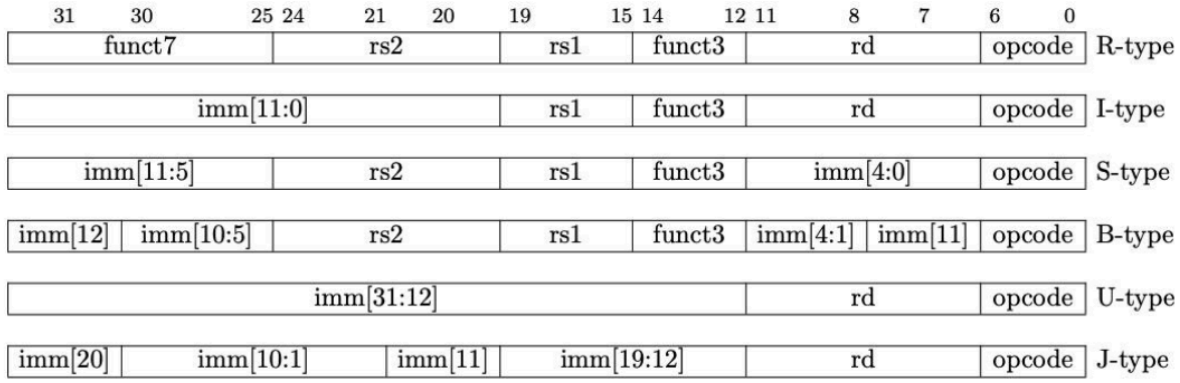
```asm
Leaf:
    addi sp, sp, -4      # adjust stack for 1 items, callee saved s1
    sw s1, 0(sp)         # save callee saved s1 to stack
    add s1, a0, a1       # s1 = g + h
    add a2, a2, a3       # j = i + j
    sub a0, s1, a2       # calculate result (g + h) - (i + j)
    # return value (g + h) - (i + j)
    lw s1, 0(sp)         # restore register s1 for caller
    addi sp, sp, 4       # adjust stack to delete 1 items
    jr ra                # jump back to caller (pseudo-assembly:ret)
```

也可以使用Caller管理的寄存器 `t1`：

```asm
Leaf:
    add t1, a0, a1       # s1 = g + h
    add a2, a2, a3       # j = i + j
    sub a0, t1, a2       # calculate result (g + h) - (i + j)
    # return value (g + h) - (i + j)
    jr ra                # jump back to caller (pseudo-assembly:ret)
```

Lec07 RISC-V (III)

RISC-V指令的编码



R-type 编码

- funct7: [31:25] 和funct3共同决定指令操作是什么
- rs2: [24:20] 决定是32个寄存器中的哪一个作为 `rs2`
- rs1: [19:15] 决定是32个寄存器中的哪一个作为 `rs1`
- funct3: [14:12] 和funct7共同决定指令操作是什么
- rd: [11:7] 决定是32个寄存器中的哪一个作为目标寄存器 `rd`
- opcode: [6:0] `0110011`，表示R-type指令

I-type 编码

- imm[11:0]: [31:20] 12位立即数，值范围[-2048, 2047]；`slli/srli/srai` 指令采用imm的低5位 [24:20] 作为立即数参数，前7位作为 funct7
- rs1: [19:15] 决定是32个寄存器中的哪一个作为 `rs1`
- funct3: [14:12] 首位决定是否无符号（无符号1，有符号0），后两位决定指令操作是什么
- rd: [11:7] 决定是32个寄存器中的哪一个作为目标寄存器 `rd`
- opcode: [6:0] `0010011`，表示I-type指令

S-type 编码

- imm[11:5]: [31:25] 立即数的高7位
- rs2: [24:20] 决定是32个寄存器中的哪一个作为 `rs2`
- rs1: [19:15] 决定是32个寄存器中的哪一个作为 `rs1`
- funct3: [14:12] 首位为0，后两位决定指令操作是什么
- imm[4:0]: [11:7] 立即数的低5位
- opcode: [6:0] `0100011`，表示S-type指令

B-type 编码

- imm[12]: [31] 立即数的第12位（最高位）
- imm[10:5]: [30:25] 立即数的5-10位
- rs2: [24:20] 决定是32个寄存器中的哪一个作为 `rs2`

- rs1: [19:15] 决定是32个寄存器中的哪一个作为 `rs1`
- funct3: [14:12] 决定指令操作是什么
- imm[4:1]: [11:8] 立即数的1-4位
- imm[11]: [7] 立即数的第11位
- opcode: [6:0] `1100011`，表示B-type(等价于SB-type)指令
- 由于需要兼容RVC，故立即数（用于存储执行命令的地址偏移）需要保留第1位（第二低的位）；第0位（最低位）默认是0，不必存储；在非RVC指令集中，第1位也是0
- 能够跳转的距离为 [-4096, 4095]
- 如果跳转距离不足，可以用以下指令拓展到 [-1048576, 1048575]：

```
bne x10, x0, next
j far
next: # next instruction
```

- 对于指令 `jalr`，其编码格式属于I-type，且存储的跳转地址是包含第0位的，因此跳转距离为 [-2048, 2047]

J-type 编码

- imm[20]: [31] 立即数的第20位（最高位）
- imm[10:1]: [30:21]
- imm[11]: [20]
- imm[19:12]: [19:12]
- rd: [11:7] 决定是32个寄存器中的哪一个作为 `rd`
- opcode: [6:0]
- 20位立即数，省去末尾第0位的0，可以表示 [-1048576, 1048575] 的地址跳转

U-type 编码

- 加载更高位立即数 Load upper immediate `lui rd, imm[31:12]`
  - `lui x5, 0xABCDE` 在寄存器 `x5` 的高20位中加载立即数，即寄存器的值变为 `0xABCDE000`
  - `li x5, 0xABCDE112` 伪代码，拓展成 `lui x5, 0xABCDE` `addi x5, x5, 0x112`
  - `li x5, 0xDEADBEEF` 伪代码，拓展成 `lui x5, 0xDEADC` `addi x5, x5, 0xEEF` （低12位最高位拓展为 `0xFFFFFEEF`，要高位补1）
- 更高位立即数加上目前PC Add upper immediate to program counter `auipc rd, imm[31:12]`
  - `auipc x6, 0xABCDE` 把 `PC + 0xABCDE000` 存入寄存器 `x6` 中
- imm[31:12]: [31:12] 高20位的立即数
- rd: [11:7] 决定是32个寄存器中的哪一个作为 `rd`
- opcode: [6:0]

Lec08 编译-汇编-链接-加载 C.A.L.L.

汇编器 Assembler

- 输入：由编译器 Compiler 生成的含有伪指令的汇编代码
- 操作步骤
  - 读取并使用伪操作：根据含有伪指令的汇编代码来组织代码和数据，如确定代码段、数据段的位置，声明全局符号等
  - 替换伪指令为标准指令
  - 生成机器语言：将处理后的汇编指令转换为机器语言（二进制）
  - 创建目标文件：将生成的机器语言、信息表等打包为目标文件 `.o`，供后续链接器使用
- 输出：目标代码 object code 和信息表 information tables
  - 目标文件头 Object file header 包含目标文件中其它部分的大小和位置信息
  - 文本段 Text segment 存储可执行的机器码
  - 静态数据段 Static data segment 存储静态数据
  - 符号表 Symbol table 列出文件中的标签和静态数据，方便被其他程序引用
  - 重定向表 Relocation table 包含之后需要被链接器 Linker 修正的代码
  - 调试信息 Debugging info 用于调试程序，记录变量名、函数名、行号与机器代码的对应关系

伪操作 Directives

- `.text` 代码段，存储汇编语言指令
- `.data` 用户数据段，存储程序中使用的变量、常量等数据
- `.globl sym` 把一个符号 `sym` 声明为全局的，让其它文件也可引用
- `.asciiz str` 存储字符串 `str` 并自动加空终止符 `\0`
- `.word` 在连续的内存字里存储32位的整数
- `.align[int]` 将数据/代码对齐到 2^{int} 整数倍的地址
- `.option` 指定汇编选项

伪指令 Pseudo-instruction

- `nop` 无操作
  - 拓展成 `addi x0, x0, 0`
- `j loop` 无条件跳转
  - 拓展成 `jal x0, loop`
- `jr rd` 跳转到 `rd`
  - 拓展成 `jalr x0, 0(rd)`
- `jal func` 跳转到标签 `func`，并把返回地址 `PC+4` （即下一条指令的地址）保存到 `x1`
  - 拓展成 `jal x1, func`
- `call func` 调用函数（同上）；可近可远
  - 拓展成 `auipc ra, offset[31:12]` `jalr ra, offset[11:0](ra)`
- `tail offset` 尾调用优化，不保存返回地址，保持返回地址仍为父级返回地址；可近可远

    - 拓展成 `auipc t1, offset[31:12]` `jalr x0, offset[11:0](t1)`
- `not rd, rs` 按位取反
    - 拓展成 `xori rd, rs, -1`
- `beqz rs, offset` 等于0则分支跳转
    - 拓展成 `beq rs, x0, offset`
- `bgt rs1, rs2, offset` 大于则分支跳转
    - 拓展成 `blt rs2, rs1, offset`
- `ret` 函数返回到返回地址 `ra` 处
    - 拓展成 `jalr x0, 0(ra)`
- `li rd, imm` 加载立即数（自动处理过大的立即数）
    - `li x5, 0xABCDE112` 拓展成 `lui x5, 0xABCDE` `addi x5, x5, 0x112`
    - `li x5, 0xDEADBEEF` 拓展成 `lui x5, 0xDEADC` `addi x5, x5, 0xEEF` （低12位最高位拓展为 `0xFFFFFEEF`，要高位补1）
- `la x1, test_input` 把变量 `test_input` 的地址（可近可远）存入寄存器 `x1` 中
    - 拓展成 `auipc x1, offset[31:12]` `addi x1, x1, offset[11:0]`
- `mv rs1, rs2` 将寄存器 `rs2` 的值赋值给 `rs1`
    - 拓展成 `addi rd, rs, 0`

尾调用 Tail Call

- 在函数尾部返回值处发生函数调用

```
int foo(int x)
{
    return bar(x * 2);
}
```

- 可以优化汇编指令为：

```
main:
    [prologue]
    call/jal foo
    [epilogue]
foo:
    slli x10, x10, 1
    tail bar
bar:
    ...
    ret
```

指令计数器寻址：两次检验

- 遍历代码两次，来解决所有的label问题
- 第一次遍历：在符号表里记录所有标签的位置
- 第二次遍历：用符号的位置来生成机器码

链接器 Linker

- 输入：目标代码 object code，信息表 information tables
- 操作步骤：整合几个目标文件 `.o` 成一个单独的可执行文件
  - 把几个目标文件的 `text` 段拼在一起
  - 把几个目标文件的 `data` 段拼在一起，附加在 `text` 段的末尾
  - 为所有的数据标签和指令标签分配确定的地址
  - 遍历重定位记录，解析所有引用到真实绝对地址
- 输出：可执行代码

三种跳转

- 程序内部跳转：由于代码相对位置不变，不需要链接器（通常是 `beq/bne/jal`）
- 外部函数引用：总是需要重定位，需要链接器（通常是 `jal`）
- 外部静态数据访问：总是需要重定位，需要链接器（通常是 `auipc/lui/addi`）

静态链接与动态链接

- 静态链接：整个库都会被链接器写入可执行文件，且不会随库更新；运行较快，可执行文件较大
- 动态链接：使用时才被链接；运行较慢，可执行文件较小

加载器 Loader

- 输入：可执行的代码
- 操作步骤：把存在磁盘上的可执行文件加载到内存中，并让代码开始运行
  - 读取可执行文件的文件头，确定 `text` 段和 `data` 段分别的大小
  - 申请一个足够大的内存空间，来存储 `text` 段、`data` 段和一个栈区
  - 把指令和数据从可执行文件里复制到新申请的内存空间
  - 把传递给程序的参数复制到栈区上
  - 初始化寄存器：大部分寄存器清除数据，栈指针 `sp/x2` 指向第一块空栈区地址
  - 开始运行程序，把程序的参数从栈区复制到寄存器上，设定程序计数器寄存器 PC
- 作用：运行代码
- 在实际情况中，加载器是操作系统的功能之一

Lec09 数字电路与系统 (I)

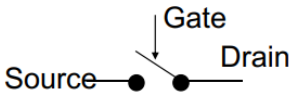
数字系统

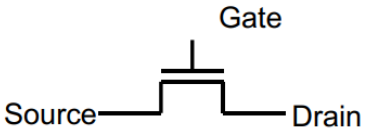
晶体管

- 闭合状态 On-switch/"1"/asserted
- 开放状态 Off-switch/"0"
- 电压控制开关 voltage-controlled switches
- 高电压（约为1V）表示逻辑1，低电压（约为0V）表示逻辑0

NMOS 与 PMOS

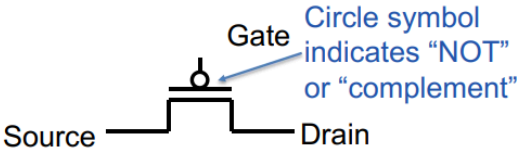
- Three terminals: source, gate, and drain
  - Basic model





n-channel transitor
off when voltage at Gate is low
on when:
voltage (Gate) > voltage (Threshold)

(**High** resistance when gate voltage **Low**,
Low resistance when gate voltage **High**)



p-channel transistor
on when voltage at Gate is low
off when:
voltage (Gate) > voltage (Threshold)

(**Low** resistance when gate voltage **Low**,
High resistance when gate voltage **High**)

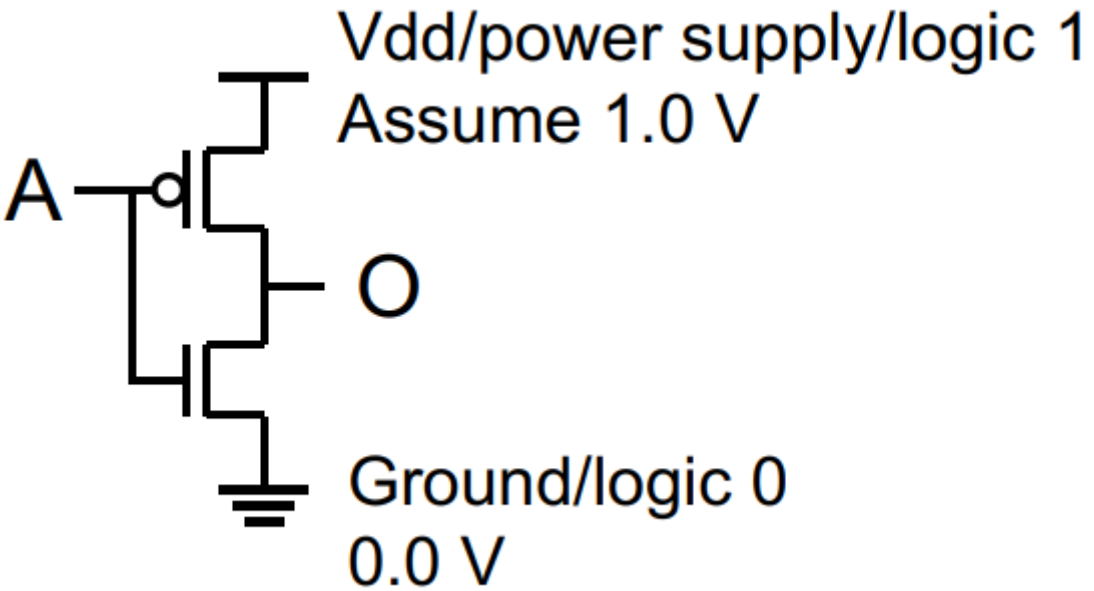
- Gate 栅极
- Source 源极
- Drain 漏极
- NMOS管擅长传递逻辑0，而逻辑1会有偏差；PMOS管擅长传递逻辑1，而逻辑0会有误差
- 成对的 N/P-type 可以实现精确传递，即CMOS

非理想效应

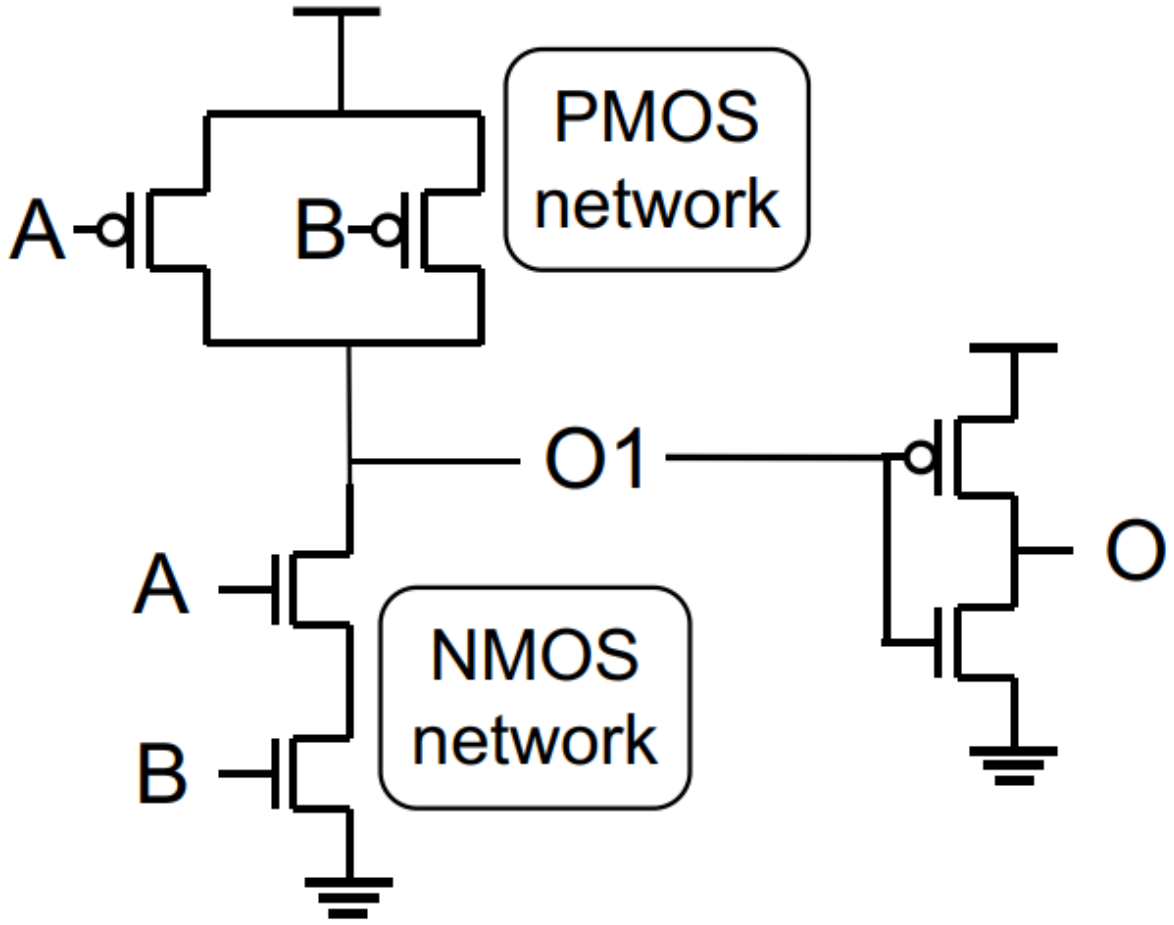
- 漏电流 leakage 可能造成发热
- 延迟 delay

从晶体管到逻辑门

非门 Inverter/Not gate

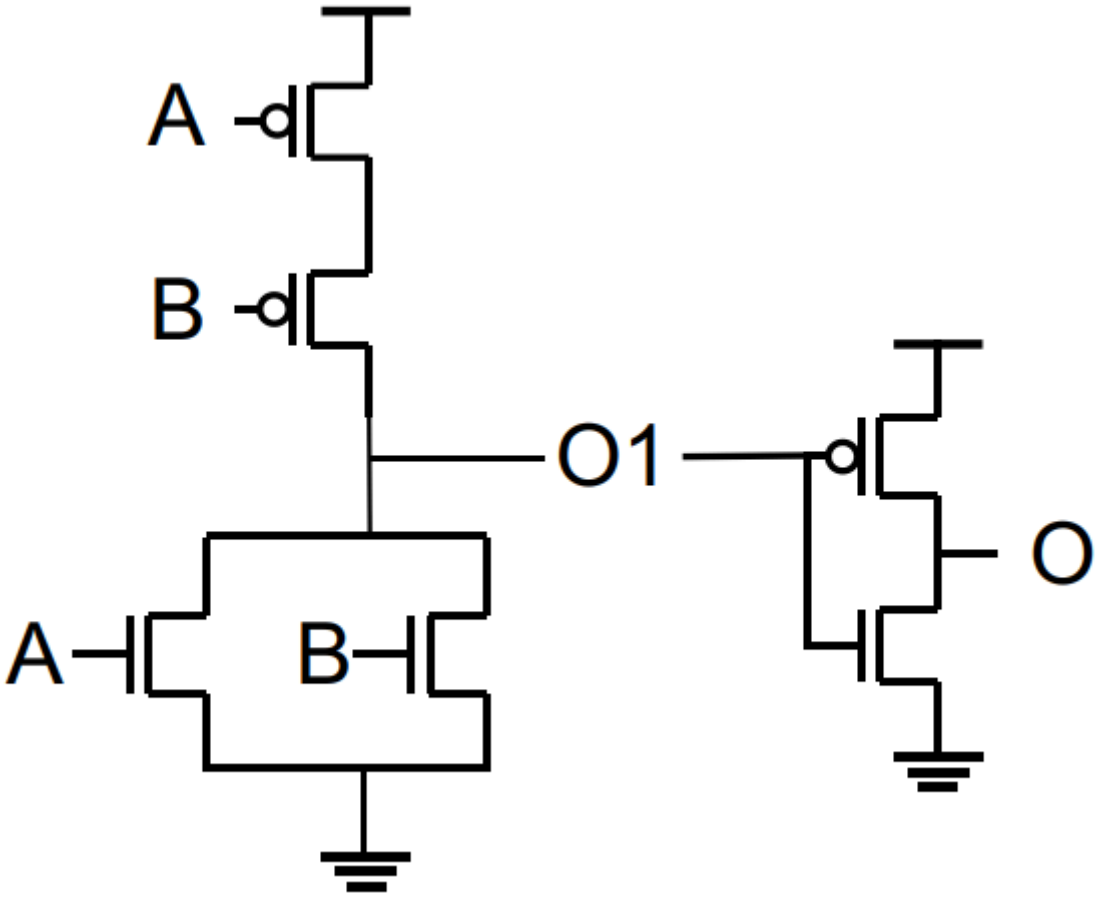


与门 And gate



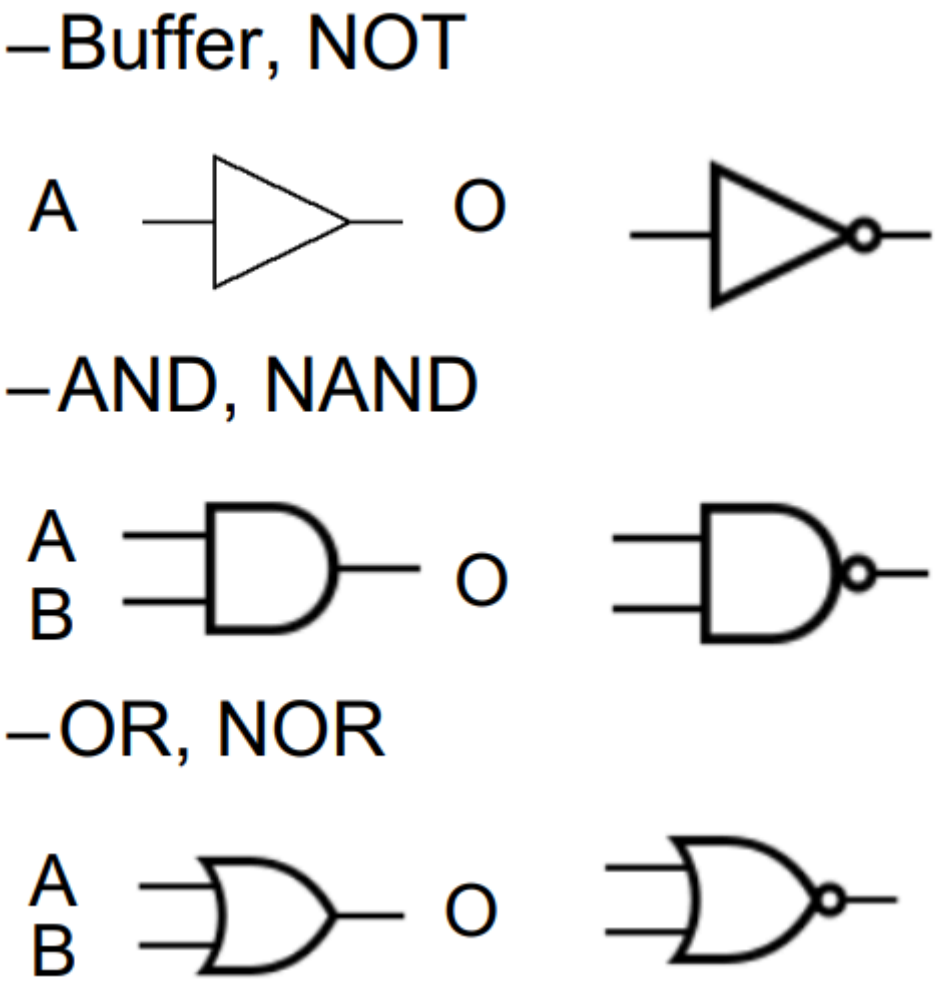
- 可以通过多并联一个PMOS管、多串联一个NMOS管来实现3输入AND

或门 Or gate



- 可以通过多串联一个PMOS管、多并联一个NMOS管来实现3输入OR

逻辑门的符号表示



- Buffer 可以用于增加电路延时

从逻辑门到组合电路

- 组合电路 Combinational circuits 电路输出仅取决于输入的电路，一般用无反馈逻辑门构建

布尔逻辑运算 Boolean Algebra

- 用 + 代替或
- 用 · （或省略）代替与
- 用 $\bar{a}$ 代替非

例如 $ab + a + \hat{c}$ 代表 (a AND b) OR a OR (NOT c)

最小项之和 sum-of-minterm

可以等效代替卡诺图 Karnaugh Map，卡诺图在本课程不作要求

1. 画出所需逻辑的真值表

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

2. 找出所有输出为1的行，把它们的计算式相加

   $\bar{A}B + A\bar{B}$

3. 使用布尔运算法则 Laws of Boolean algebra 来化简

Laws of Boolean Algebra

AND form	OR form	
$X\overline{X} = 0$	$X+\overline{X} = 1$	Complementarity
$X0 = 0$	$X+1 = 1$	Laws of 0's and 1's
$X1 = X$	$X+0 = X$	Identities
$XX = X$	$X+X = X$	Idempotent Laws
$XY = YX$	$X+Y = Y+X$	Commutativity
$(XY)Z = X(YZ)$	$(X+Y)+Z = X+(Y+Z)$	Associativity
$X(Y+Z) = XY+XZ$	$X+YZ = (X+Y)(X+Z)$	Distribution
$XY+X = X$	$(X+Y)X = X$	Absorption
$\overline{XY} = \overline{X}+\overline{Y}$	$\overline{X+Y} = \overline{X}\,\overline{Y}$	DeMorgan's Law

构建更大的模块

- 可以通过拼接较小的模块（如门电路、半加器等）构建成更大的模块

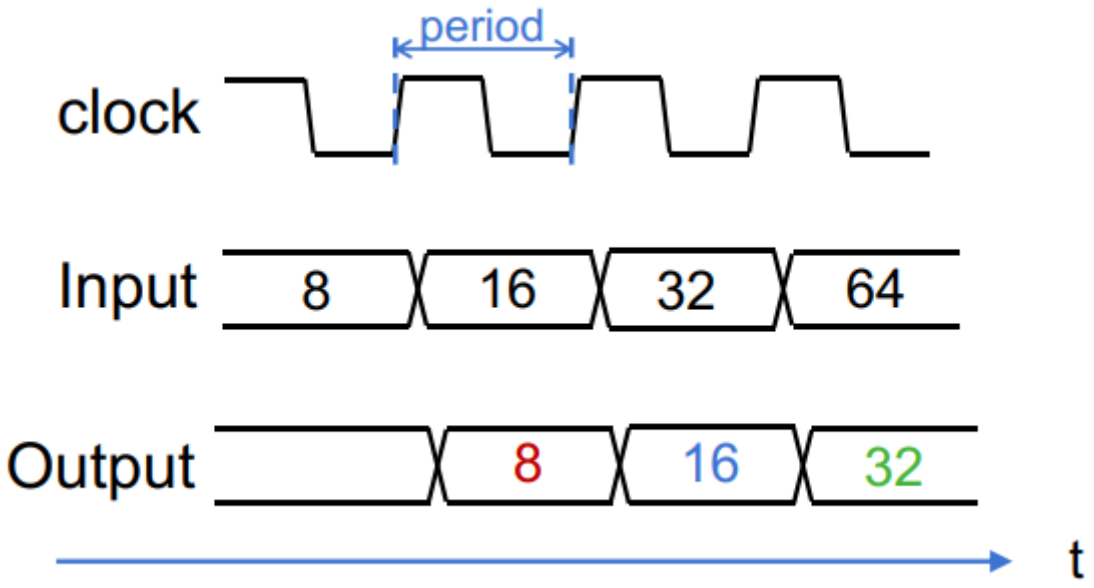
Lec10 数字电路与系统 (II)

状态元件

- 可以存储信息的电路
- 有限状态机
- 有时序

寄存器

- 需要配合时钟 clock 使用
- 在每次时钟方波处于上升沿时，读取输入端的信号，作为输出



- 寄存器由多个 D触发器 D flip-flops (DFFs)组成
- 每个D触发器都可以存储1 bit 的数据，可以看作1位寄存器
- 同一个寄存器内的所有D触发器共享同一个时钟，以达到同步的效果
- 寄存器有一个重置 reset 接口，当 `reset = 1` 时，寄存器值保持为0

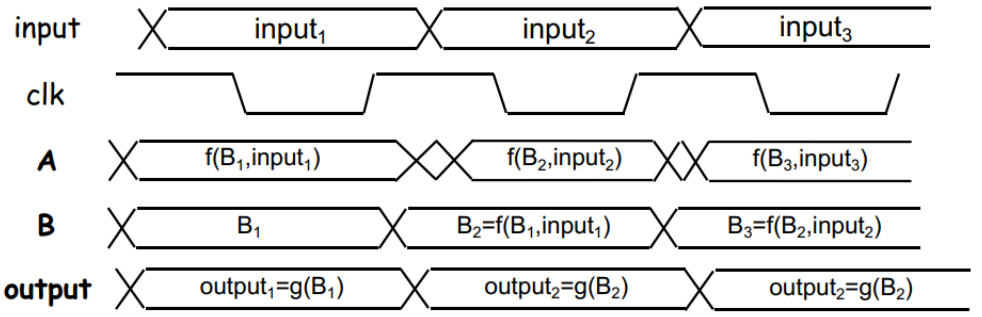
有限状态机 Finite state machine (FSM)

- 一种计算用数学模型
- 在任何时间只能在有限状态中的一种状态
- 会由输入而改变状态，称为转移 transition
- 初始态 initial state (entrance)，决定转移的输入，输出（可选）

Moore状态机 Moore machine

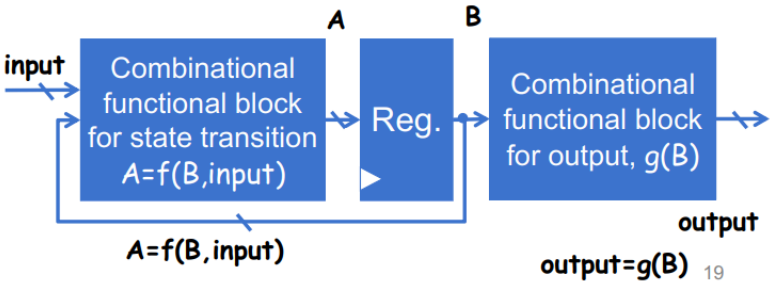
- 输出仅由当前状态决定，与当前输入无关
- 响应速度受时间信号约束，但可以有效消除毛刺信号
- 对于任意的转移真值表，通用的Moore状态机为：

- Timing diagram (general case)



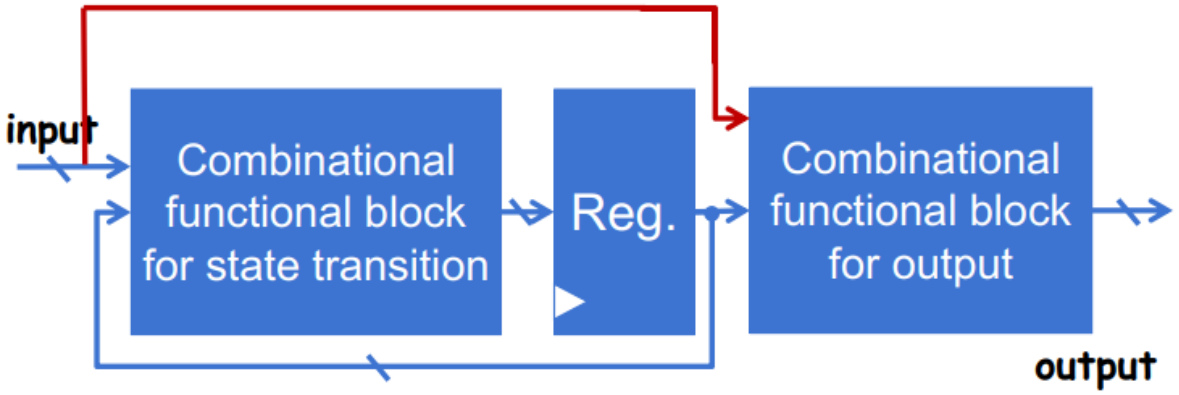
input	Q_{k-1}	Q_k	output
0	0	0	0
0	1	0	0
1	1	1	1
1	0	1	1

Template of a synchronous circuit that implements the FSM



Mealy状态机 Mealy machine

- 输出由当前状态和当前输入决定



- 响应快，但是容易受到毛刺信号影响

设计同步（时序）电路 Synchronous Circuits

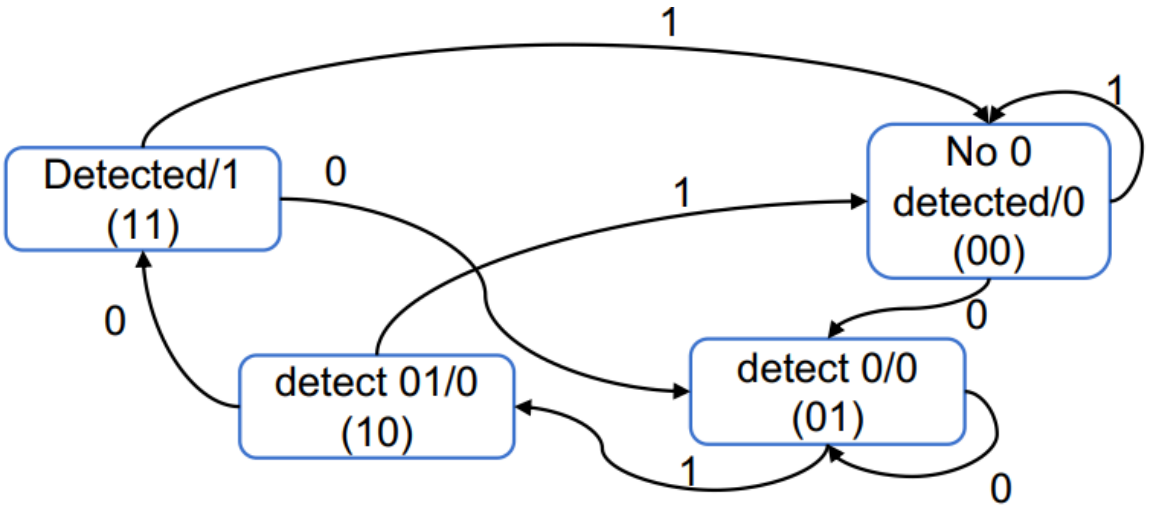
- 画出有限状态机
- 定义二进制数，来表示每个状态、输入和输出
- 写出输入-当前状态-下一时刻状态的真值表、当前状态-输出的真值表
- 用Moore/Mealy状态机模板来搭建电路

例子

- 在一串二进制输入中不重叠地检测 `010`，每次检测到后下一位输出 `1`

```
Input:  0 1 0 0 1 0 1 0 1 1 0
Output: 0 0 0 1 0 0 1 0 0 0 0
```

- 先画出状态机：



- 写出真值表：（`S[1]S[0]` 表示状态机的一个状态，`input` 为当前输入，表示一个状态经过输入得到另一个状态；`output` 表示当前状态对应的输出）

	Previous state		Current state		
input	$S[1]_{k-1}$	$S[0]_{k-1}$	$S[1]_{k}$	$S[0]_{k}$	output
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	0	0

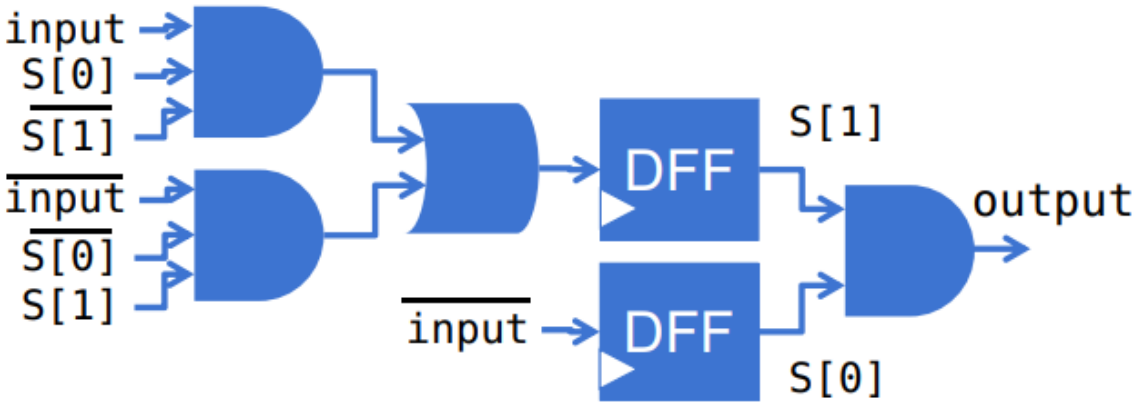
- 观察真值表，化简为布尔逻辑运算：

$$output=S[1]_kS[0]_k$$

$$S[1]_k=\overline{S[1]_{k-1}}S[0]_{k-1}input+S[1]_{k-1}\overline{S[0]_{k-1}}\,\overline{input}$$

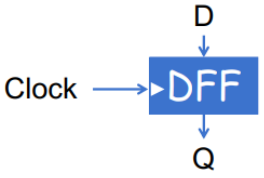
$$S[0]_k=\overline{input}$$

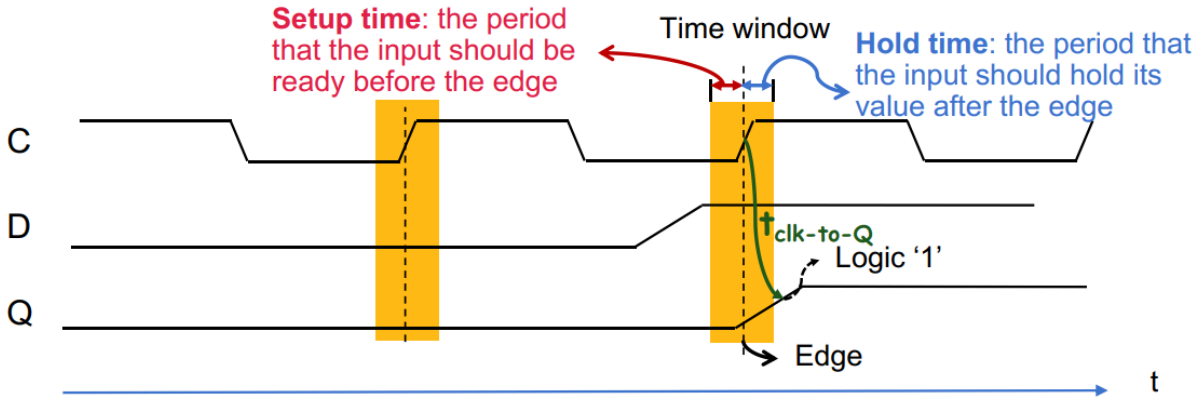
- 画出电路图：（DFF可以看作一位寄存器）



时间约束 Timing constraints

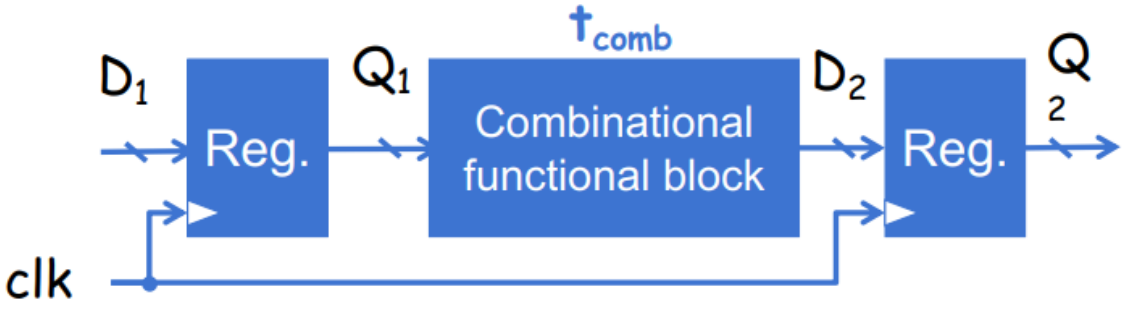
- DFF also has delay: $t_{clk\text{-}to\text{-}Q}$

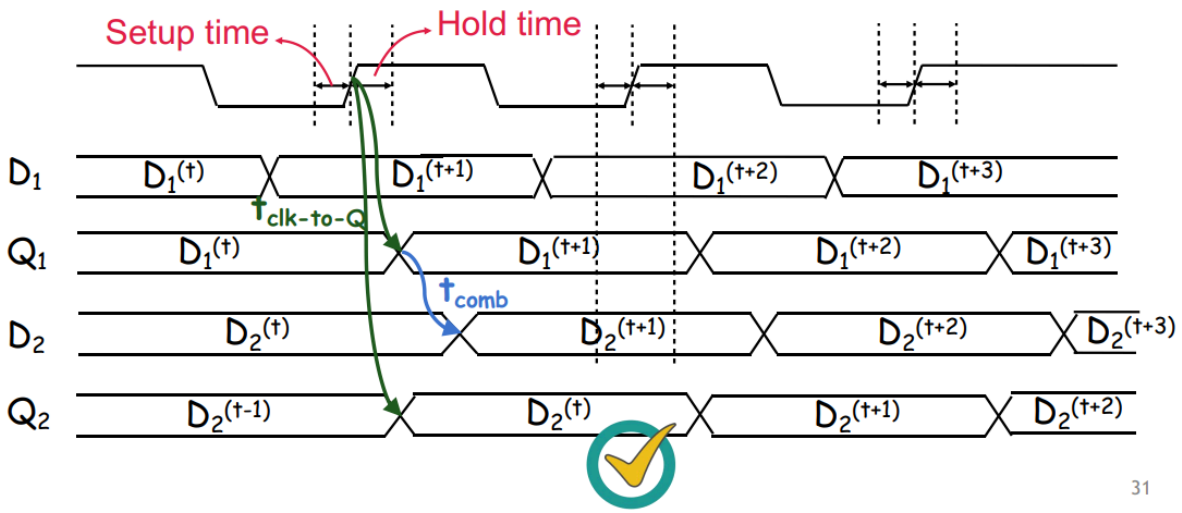


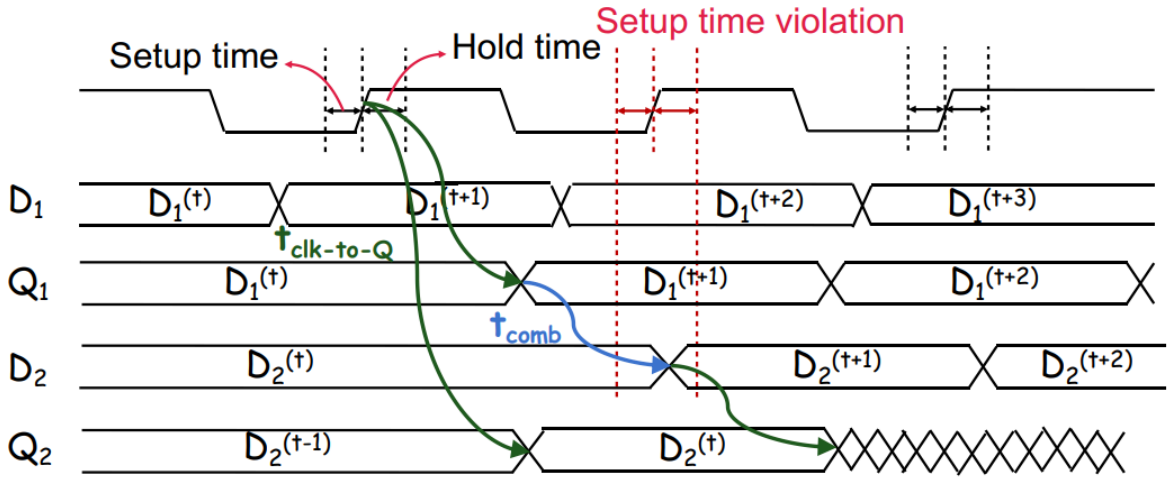


- `clock-to-Q` 时钟到输出延迟：从时钟上升沿（正中间）到DFF输出稳定信号所需要的时间
- `Time window` 时间窗口（包括建立时间 Setup time 和保持时间 Hold time）：在这段时间内，DFF的输入需要保持稳定（否则寄存器的值是不稳定的，电路工作出错）

估测最大工作频率

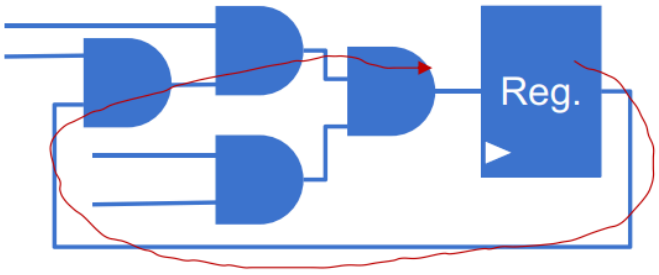






- $t_{clk-to-Q} + t_{comb} \leq min\_clock\_period - setup\_time$
- 最大时钟频率 `maximum clock frequency` 为最小时钟周期 `min clock period` 的倒数
- 对于多步这样的通路 Path，延迟 $t_{clk-to-Q} + t_{comb}$ 最大的通路决定最大工作频率，这个通路被称为关键通路 critical path
- 一般同型号寄存器的 $t_{clk-to-Q}$ 相同，而不同计算电路的 t_{comb} 取决于电路元件数量、元件延迟

例子



Given
Clock->Q 1ns
Setup 1ns
Hold 1ns
AND delay 1ns

What is maximum clock frequency?

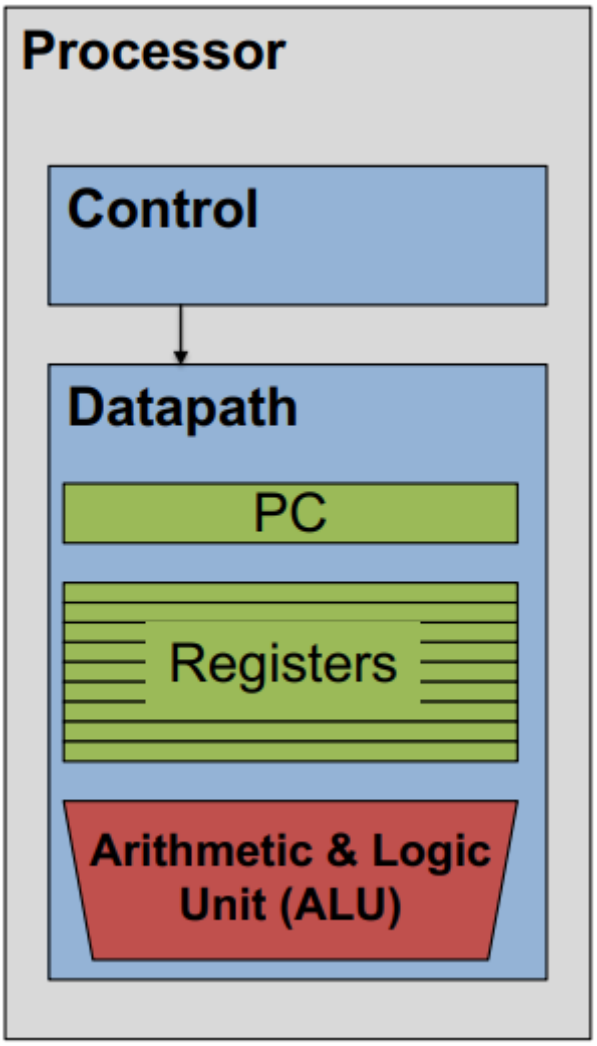
- A: 5 GHz
- B: 500 MHz
- C: 200 MHz
- D: 250 MHz
- E: 1/6 GHz

- 红色标出的为关键通路 critical path
- 总共经过1个寄存器（1 Clock to Q）、3个与门（3 AND delay）、1个建立时间（1 Setup time），共计5ns
- 故最大时钟频率为 $200MHz$，选 `C`

Lec11 数据通路 Datapath

控制器与数据通路 Controller & Datapath

- 控制器 Controller 在内存获取指令放入指令寄存器，解析指令，并发出控制信号，可以被建模为有限状态机(FSM)
- 数据通路 Datapath 包括了数据处理与存储



算术逻辑单元 Arithmatic Logic Unit (ALU)

- 可以执行所有算术与逻辑操作：`ADD` `SUB` `SLL` `SLT` `SLTU` `XOR` `SRL` `SRA` `OR` `AND` `ADDI` `SLTI` `SLTIU` `XORI` `ORI` `ANDI`
- 使用多路选择器 Multiplexer (MUX) 来连接完成各种操作的电路的输出，选择需要输出的信号
- 每一条 RV32I 指令都可以在1时钟周期内完成

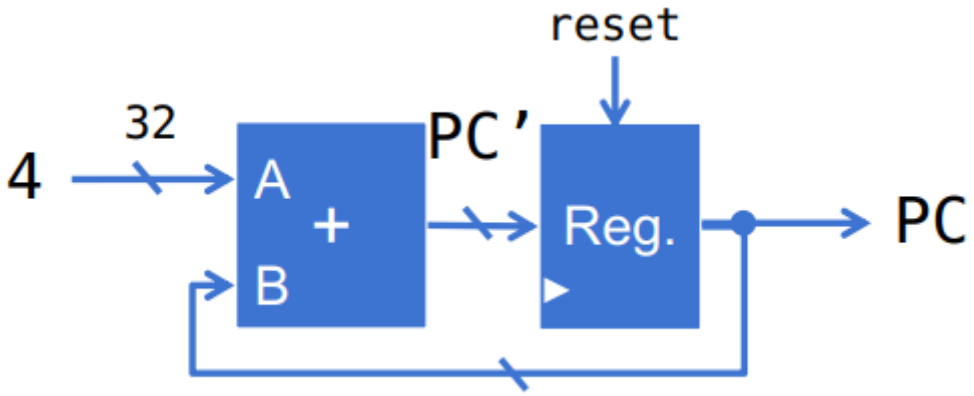
寄存器（组） Register (file)

- 寄存器组 Register file：32个32位寄存器
- 可以使用多路选择器来实现选择一个编号，让对应寄存器输出

改变特定寄存器的值

- 这里仅演示一种实现方式，且忽略 `x0` 寄存器
- 控制信号独热编码，只在需要控制的寄存器上输出1；这路控制信号与时钟信号分别逻辑与，然后分别输入每个寄存器；把所有寄存器的输出连接到2个多路选择器，选择2个信号进行输出

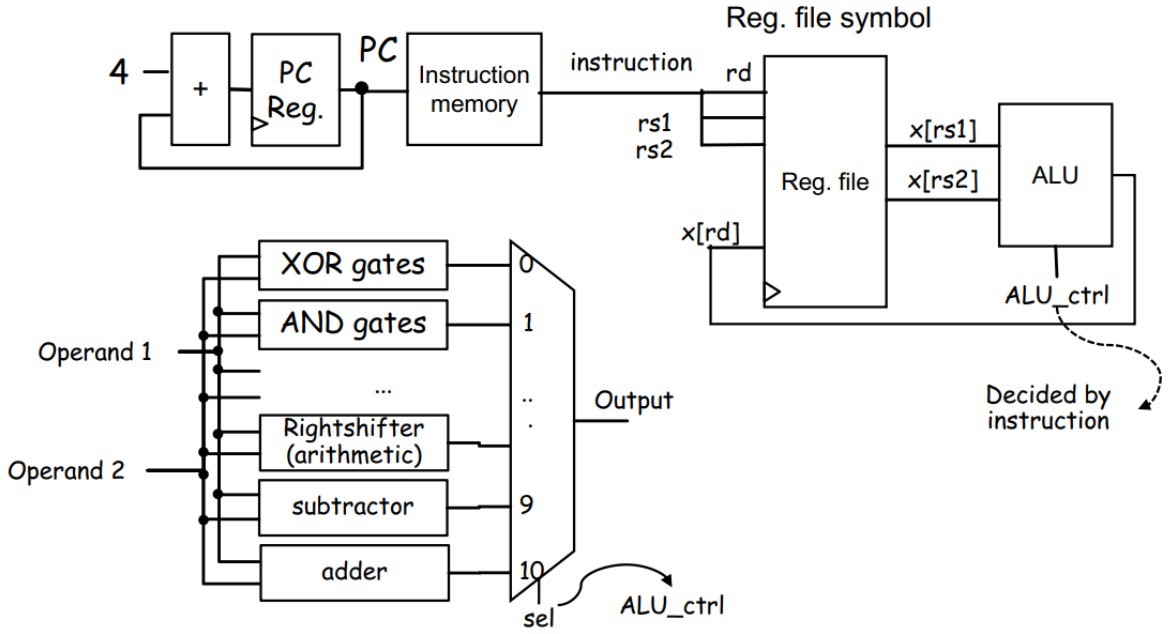
PC寄存器 Program Counter



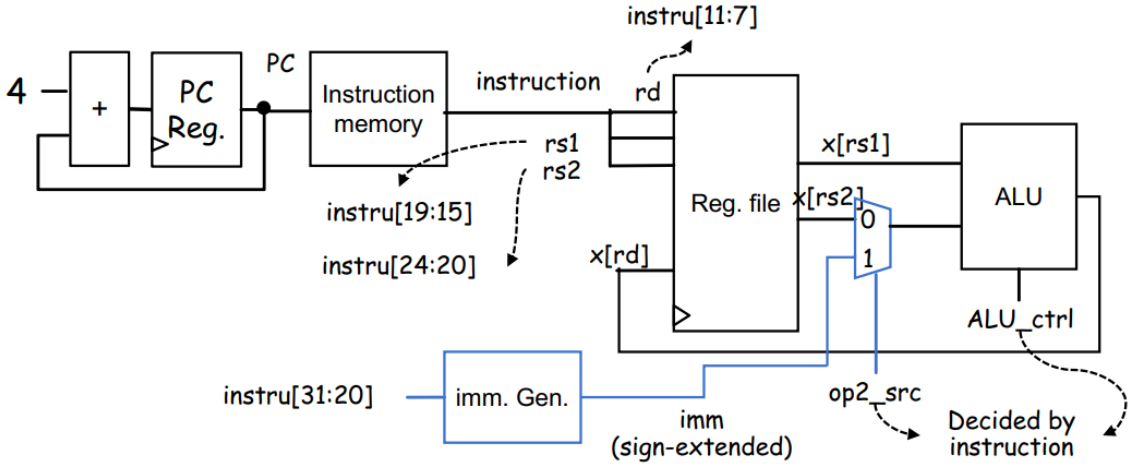
Lec12 数据通路 Datapath (II)

数据通路 Datapath

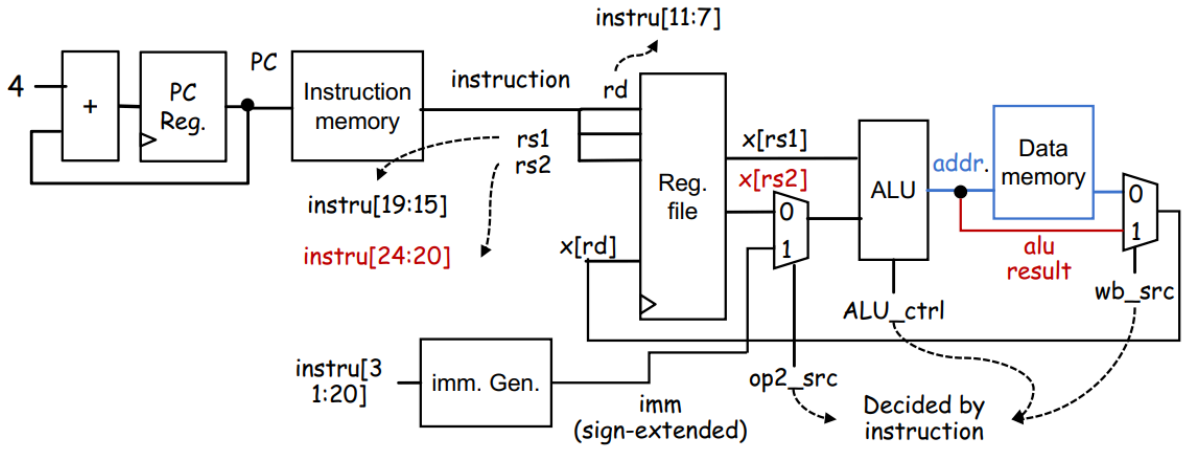
- `R-type` 指令的数据通路：



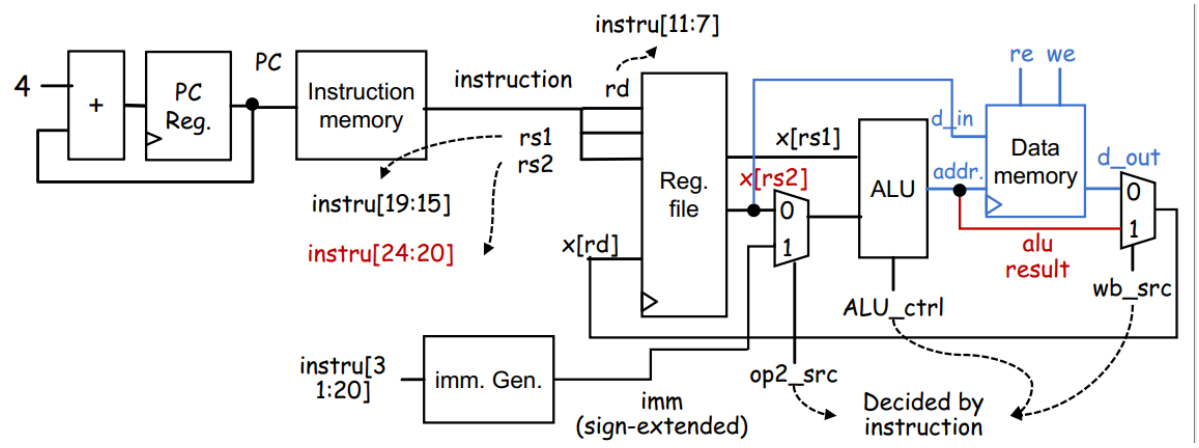
- `I-type` 指令的数据通路：



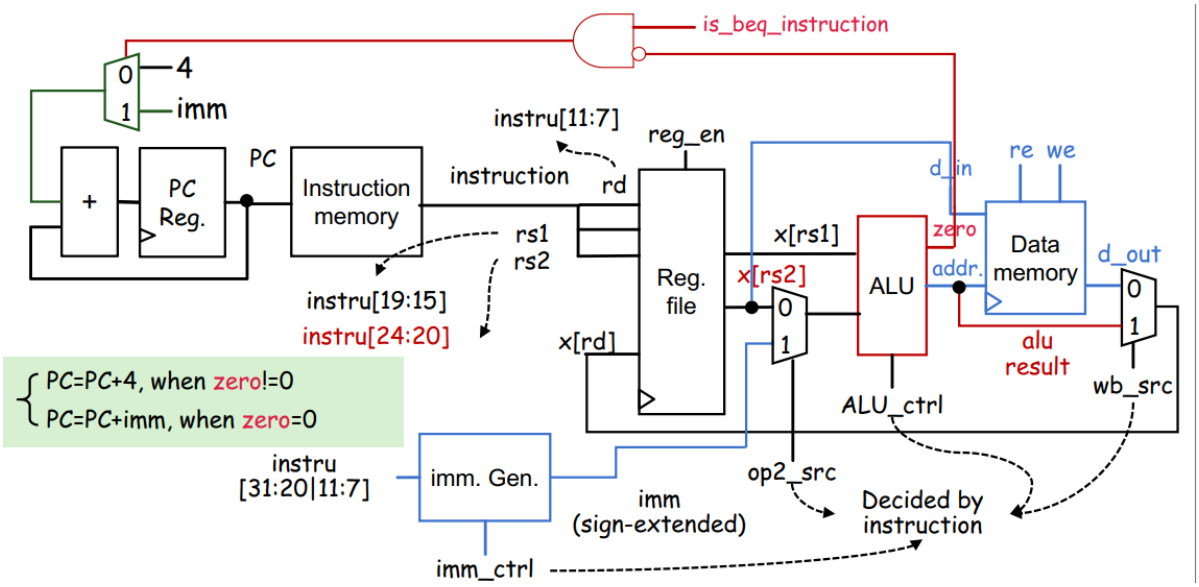
- `I-type` `Load` 指令数据通路



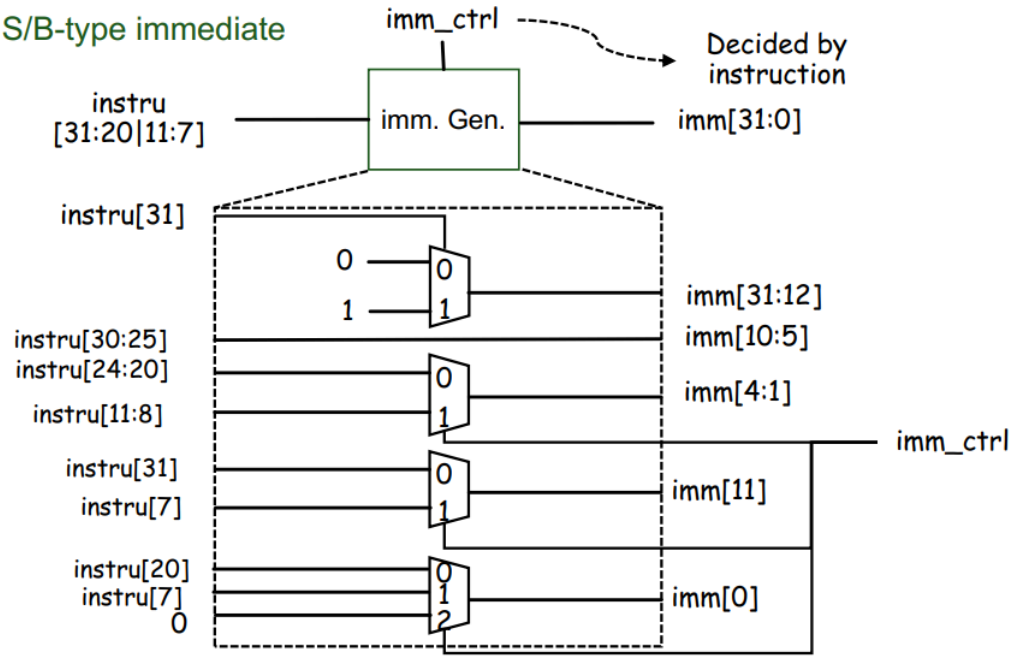
- `S-type` 指令的数据通路：



- 由于硬件结构不是可变的，故所有计算的中间结果（包括不必要的计算）都会得到，而由多路选择器来选择要输出/继续处理的数据
- `B-type` 指令的数据通路：



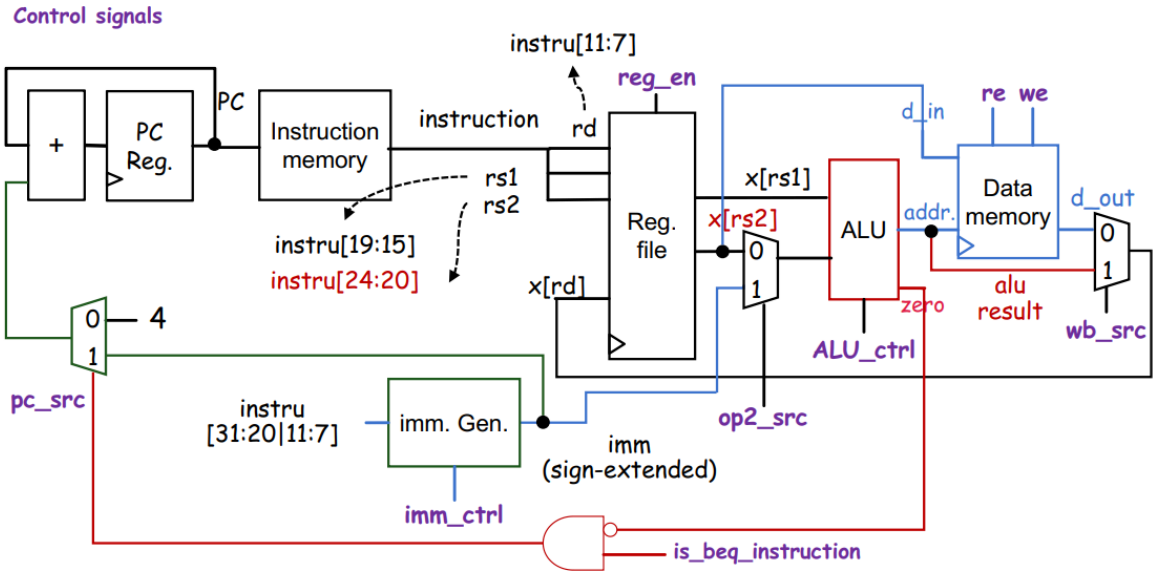
- I-type & S/B-type immediate



控制器 Controller

控制信号 Control Signals

- 即下图中紫色的字段，是由指令解码得到的各种控制器的正确状态



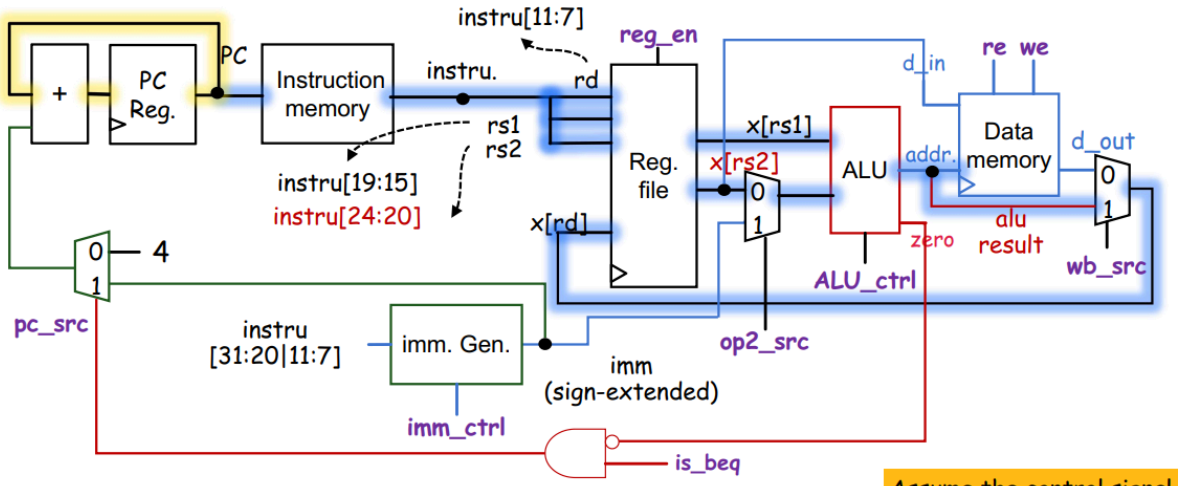
- `reg_en` Register Enable，是否允许寄存器组进行更新
- `re` Read Enable，是否允许从数据寄存器 Data memory 读取数据
- `we` Write Enable，是否允许写入数据寄存器 Data memory
- `alu_ctrl` ALU Control，决定ALU执行哪种算术或逻辑运算
- `imm_ctrl` Immediate Control，控制立即数生成器 imm.Gen （对于不同指令类型）如何拓展立即数
- `wb_src` Write-Back Source Select，决定最终写入寄存器的数据来自ALU计算还是数据寄存器
- `op2_src` Operand 2 Source Select，决定ALU的第二个操作数来自寄存器组还是立即数
- `is_beq_instruction` 决定是否为 `beq` 指令（相等则分支）

控制器

- 控制从指令到控制信号的转换
- 可以由有限状态机 FSM 建模，也可以用只读内存 ROM 实现

延时分析 Timing analysis

- 最小时钟周期 $= t_{clk-to-q} + t_{comb} + t_{setup}$
- `R-type` 数据通路的延迟：（蓝色黄色取max，一般认为蓝色更长）



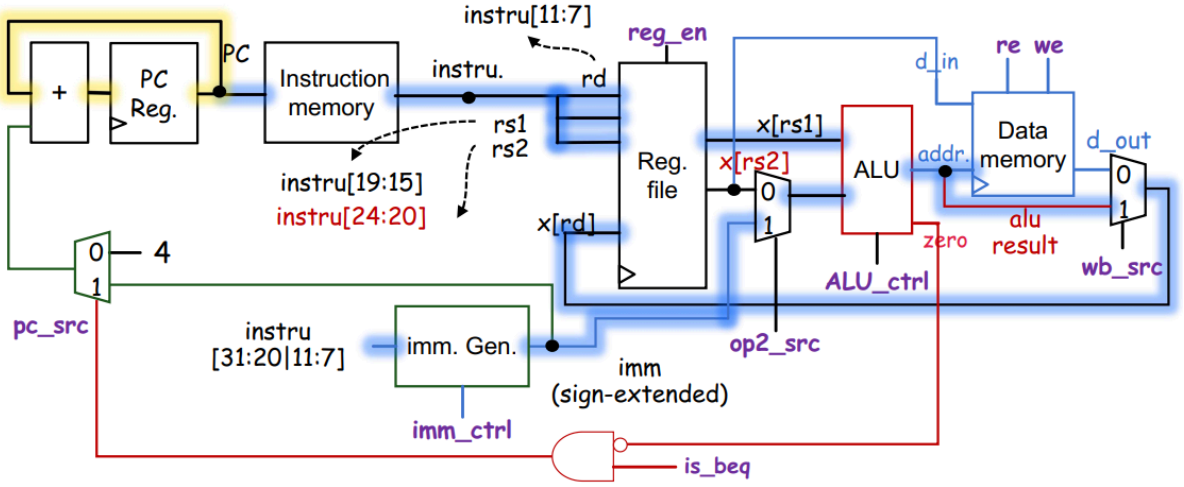
$t_{clk-to-q} + t_{add} + t_{setup}$

$\underbrace{t_{clk-to-q} + t_{Imem}}_{t_{IF}} + \underbrace{t_{reg} + t_{mux}}_{t_{DEC}} + \underbrace{t_{alu}}_{t_{EX}} + \underbrace{t_{mux} + t_{setup}}_{t_{WB}}$

R-type datapath

（蓝色：t_{IF} 读取指令 Instruction Fetch + t_{DEC} 解码 Decode + t_{EX} 执行 Execute + t_{WB} 写回 Write Back）

- `I-type` 数据通路的延迟：（三条路径延迟取max，一般认为蓝色两个更长）



$t_{clk-to-q} + t_{add} + t_{setup}$

$t_{clk-to-q} + t_{Imem} + t_{reg} + t_{alu} + t_{mux} + t_{setup}$

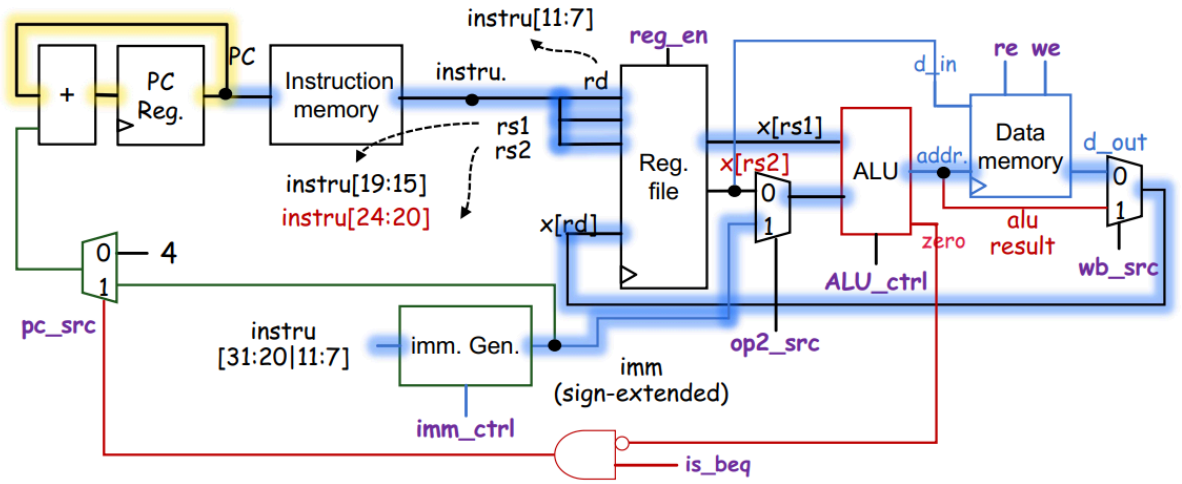
$t_{clk-to-q} + t_{Imem} + t_{imm} + t_{mux} + t_{alu} + t_{mux} + t_{setup}$

$\Big\}$ max

$t_{IF} + t_{DEC} + t_{EX} + t_{WB}$

I-type arithmetic & logic datapath

- `I-type` `Load` 指令数据通路的延迟（一般是所有指令中延迟最大的）：



$t_{clk-to-q} + t_{add} + t_{setup}$

$t_{IF} + t_{DEC} + t_{EX} + t_{MEM} + t_{WB}$

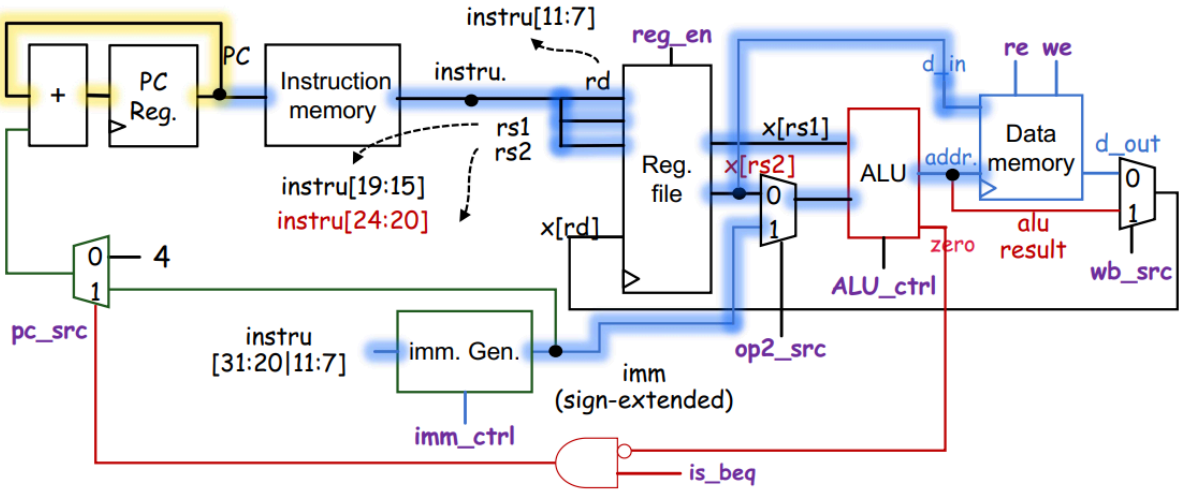
$t_{clk-to-q} + t_{Imem} + t_{reg} + t_{alu} + t_{Dmem} + t_{mux} + t_{setup}$

$t_{clk-to-q} + t_{Imem} + t_{imm} + t_{mux} + t_{alu} + t_{Dmem} + t_{mux} + t_{setup}$ } max

I-type load datapath

(蓝色：t_{MEM} 从Data Memory中取数据的延迟)

- `S-type` 数据通路的延迟：



$t_{clk-to-q} + t_{add} + t_{setup}$

$t_{clk-to-q} + t_{Imem} + t_{reg} + t_{Dmem}$

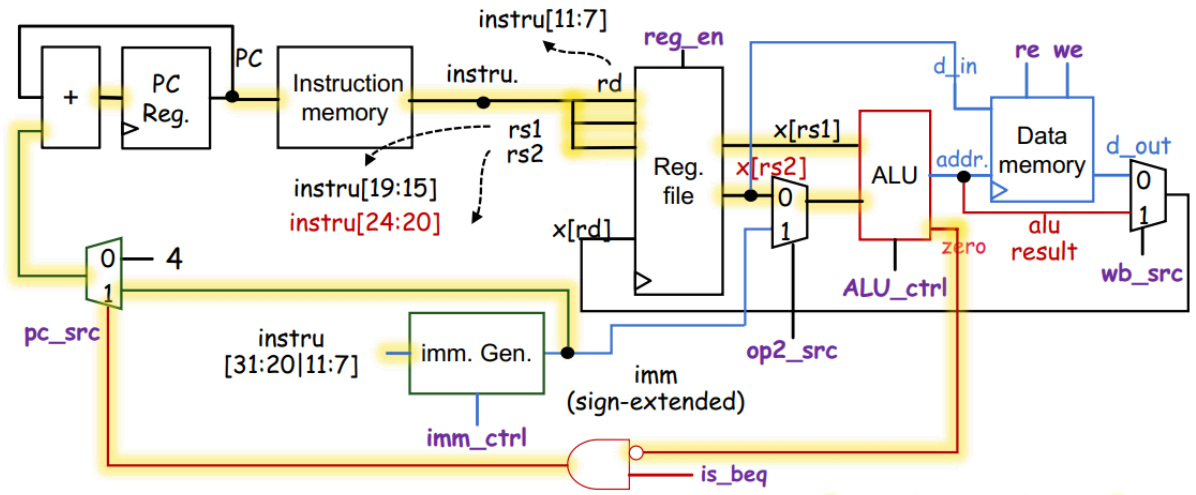
$t_{clk-to-q} + t_{Imem} + t_{reg} + t_{alu} + t_{Dmem}$

$t_{clk-to-q} + t_{Imem} + t_{imm} + t_{mux} + t_{alu} + t_{Dmem}$ } max

$t_{IF} + t_{DEC} + t_{EX} + t_{MEM}$

S-type datapath

- `B-type` 数据通路的延迟：



$$t_{IF} + t_{DEC} + t_{EX} + t_{WB}$$

$$\left.\begin{array}{l} t_{clk-to-q} + t_{Imem} + t_{reg} + t_{mux} + t_{alu} + t_{and} + t_{mux} + t_{add} + t_{setup} \\ t_{clk-to-q} + t_{Imem} + t_{imm} + t_{mux} + t_{add} + t_{setup} \end{array}\right\} \max$$

B-type datapath

Lec13 流水线 Pipeline

性能铁律 Iron law of performance

- **CPU (execution) time** (ignore I/O, operating system overhead etc.)

$$\frac{Time}{Program} = \frac{Instructions}{Program} \cdot \frac{Cycles}{Instruction} \cdot \frac{Time}{Cycle}$$

- Can be obtained by profiling or hardware counter
- ISA (e.g., RISC vs. CISC)
- The program itself
- Compiler
- Programming language
- etc.

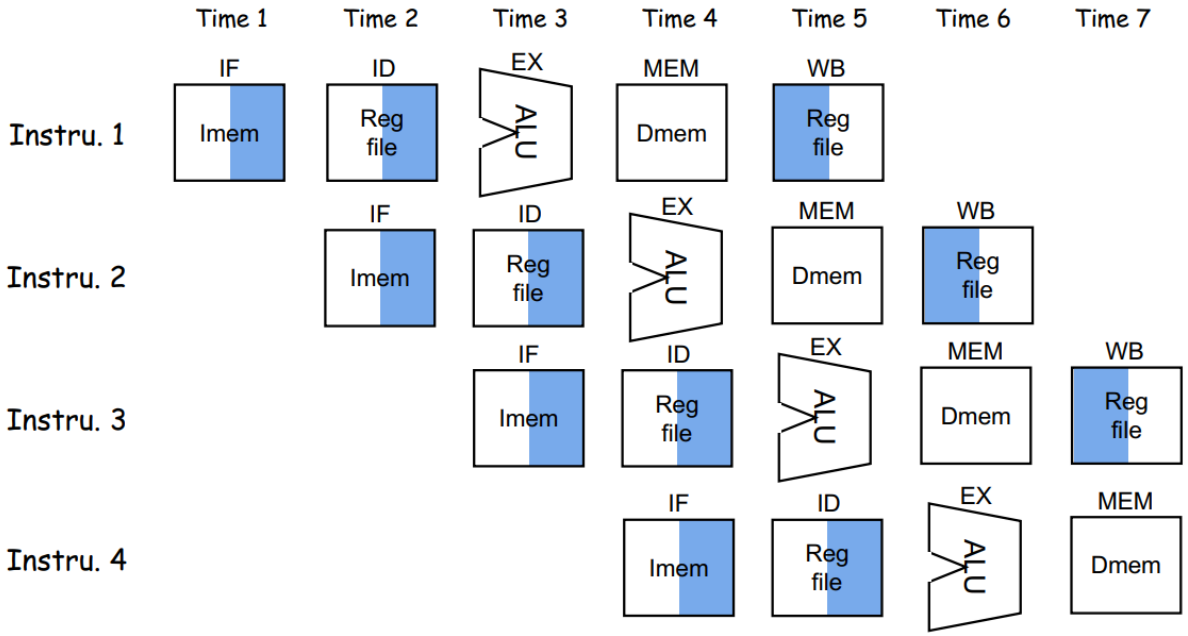
- Short as “CPI”
- Microarchitecture implementation or circuit design/ISA
- Compiler
- Program
- Programming language

- Microarchitecture implementation or circuit design/ISA

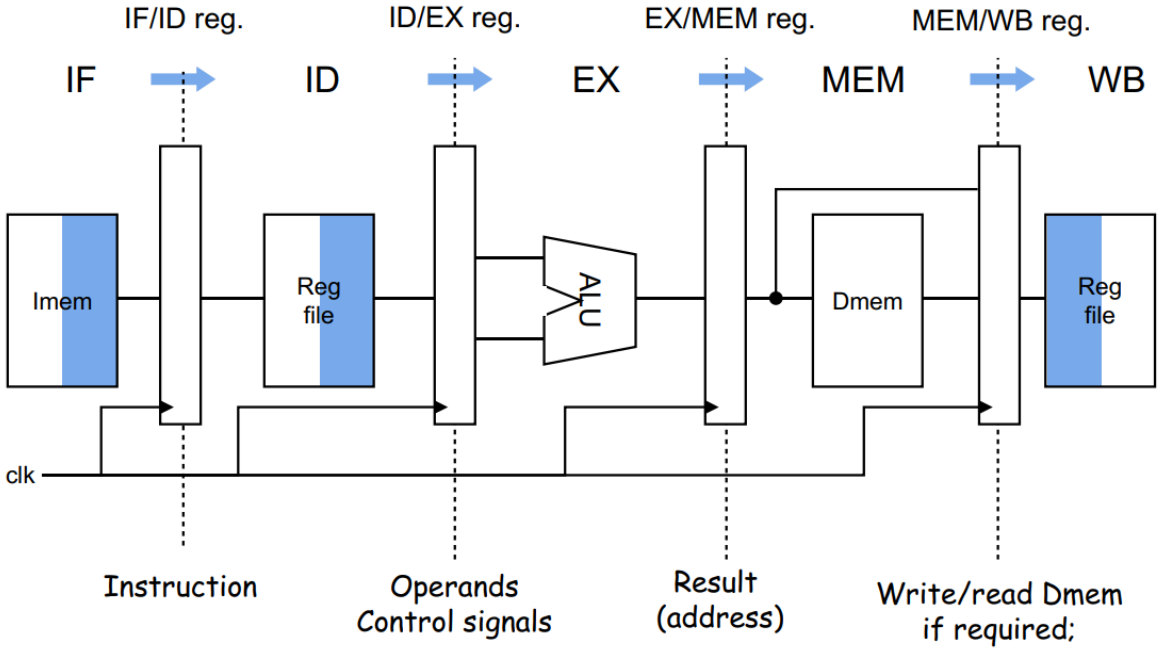
- `每个程序执行的时间` = `每个程序的平均指令数` * `每条指令执行的CPU周期数` * `每个CPU周期的时间`
- CISC 每条指令执行时间更长，但每个程序的平均指令数更少
- 我们之前简化认为每条指令执行1个CPU周期，但现代计算机一般 $CPI \neq 1$

流水线 Pipeline

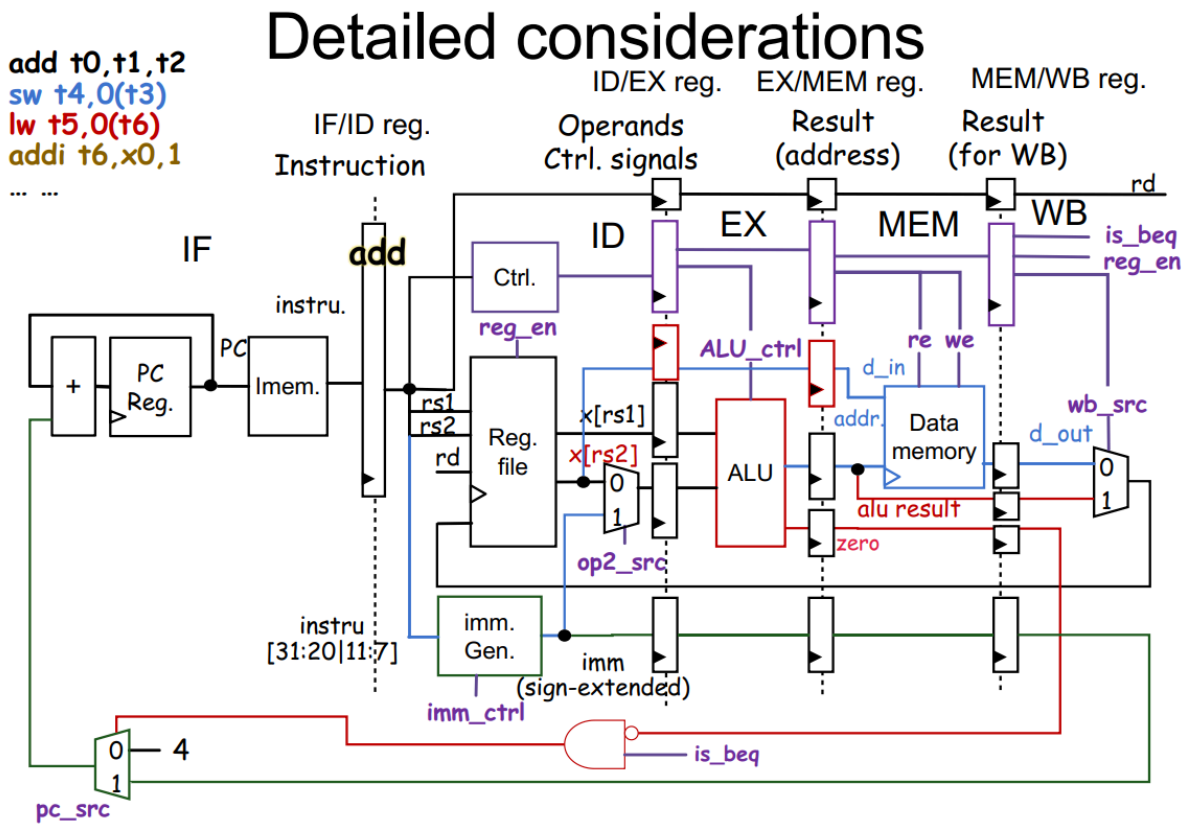
- 我们可以把CPU处理分为5个阶段：IF -> ID/DEC -> EX -> MEM -> WB
  - ID 为译码 Instruction Decode
- 我们可以进行一个流水线操作（五级流水），把不同元件的不同阶段并行执行以节省时间：



- 对于单周期处理器，最小时钟周期 $= t_{IF} + t_{ID} + t_{EX} + t_{MEM} + t_{WB}$
- 对于流水线CPU，最小时钟周期 $= max\{t_{IF}, t_{ID}, t_{EX}, t_{MEM}, t_{WB}\}$
- 在每两个阶段之间，放一个寄存器，被称为流水线寄存器 Pipeline Registers



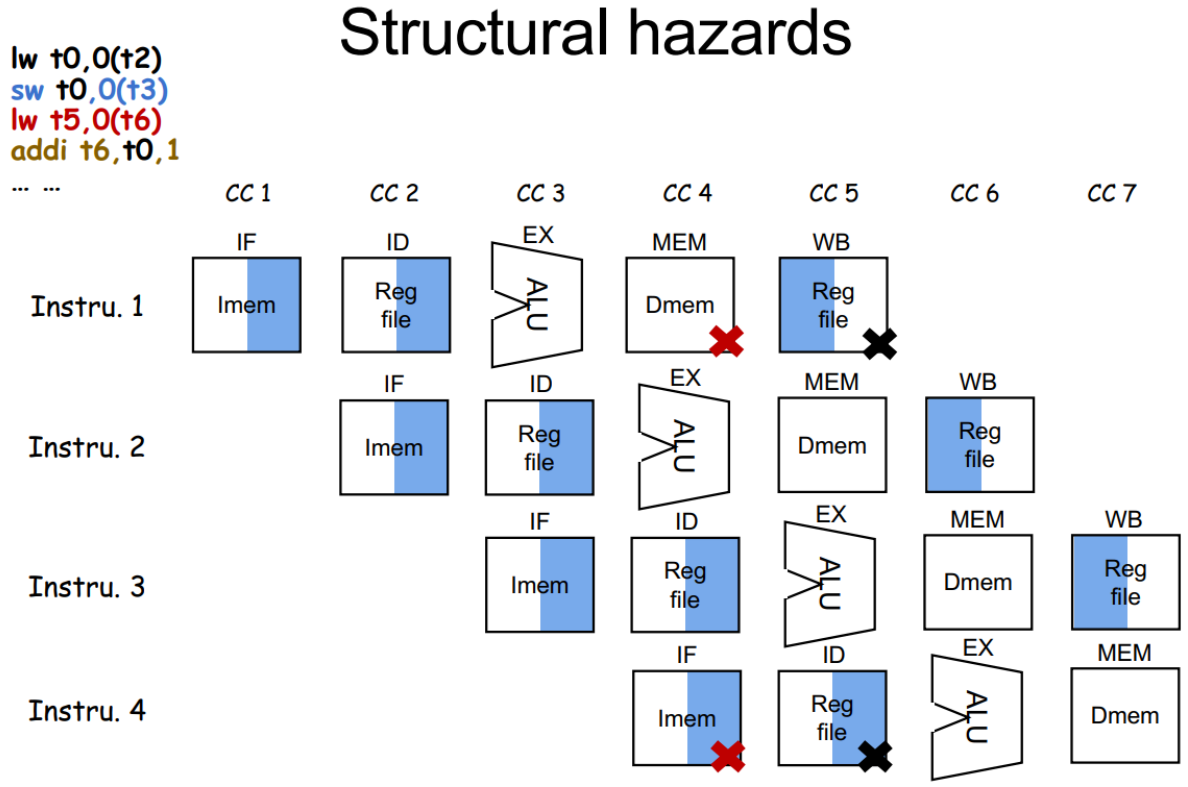
- 对于不需要进行 MEM 操作的指令，也会经过一次流水线寄存器，从而空过一个时钟周期



冲突 Hazard

结构冲突 Structural hazards

- 当多条指令同时使用同一个硬件元件（如同时读/写寄存器组等）时，会发生结构冲突 Structural Hazard

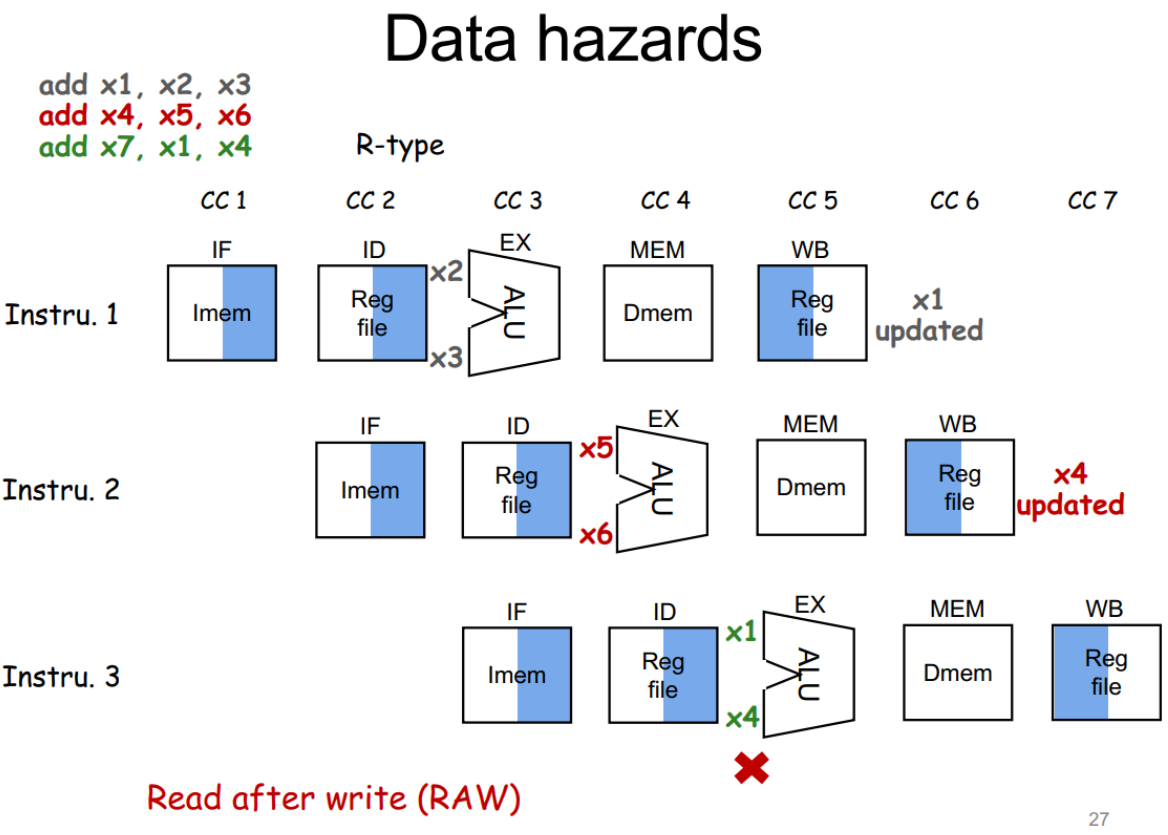


- 解决方案：修改硬件
  - 把 I-memory（指令内存）和 D-memory（数据内存）分离
  - 独立设计寄存器组，允许其在前半周期写、后半周期读（寄存器组的双半周期访问）

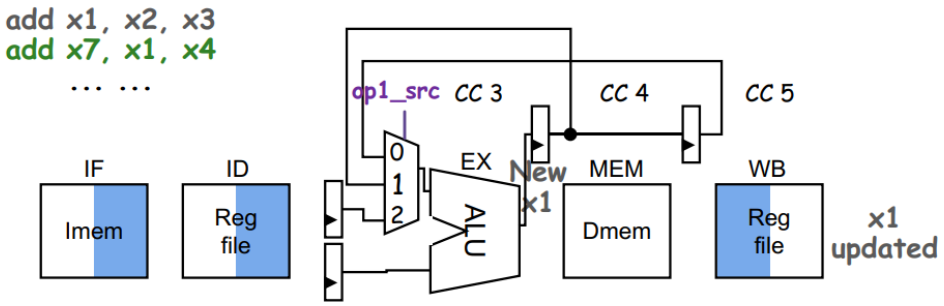
- 如果实在无法修改硬件，就在指令之间插入一个气泡 bubble（即 `nop` 指令，无效果），让冲突的指令错开运行

数据冲突 Data hazards

- 当多条指令存在读写顺序逻辑，而流水线操作使得时序逻辑发生了改变，则会发生数据冲突 Data hazard
- 其中包括写后读 (Read after Write, RAW)，读后写 (WAR) 和写后写 (WAW)
- 在简单五级流水中，后两者通常不会发生（课程中不讨论），而只有最常见的写后读冲突

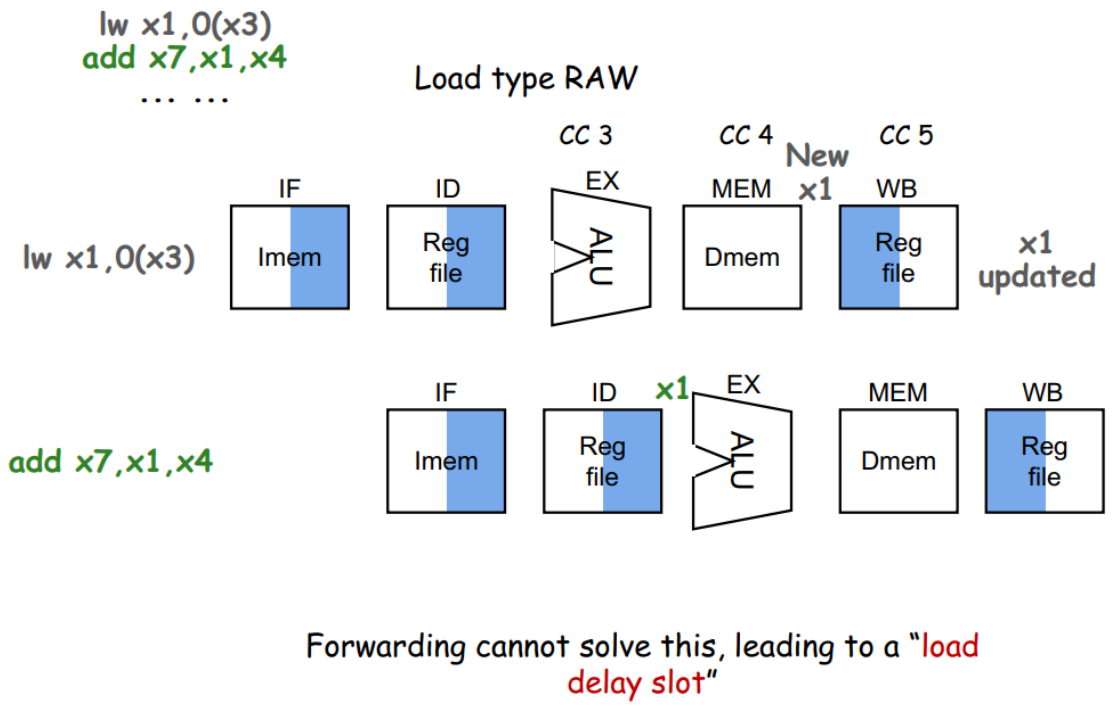


- 解决方案：
  - 插入气泡 bubble，让数据冲突的指令等到前置操作更新完成后再执行
  - 数据前推 Forwarding/Bypass：在ALU前设计2个控制信号 `op1_src`/`op2_src`，支持控制ALU从 `ID/EX流水线寄存器`（下图ppt中存在笔误）中正常读取，或从 `EX/MEM流水线寄存器` 或 `MEM/WB流水线寄存器` 中读取前1或2条指令的计算结果作为输入



- How to decide op1_src?
  - Select the forwarded value or the value from EX/MEM register
  - 1. rd of the add instruction (may be the other type instructions) equals rs1 in add (may be the other type instructions) instruction
  - 2. Ignore write to x0
  - 3. The first instruction must write the register and the second instruction must read the register

    - 特殊地，对于 Load type RAW（即 `lw rd, ...` 指令的下一条指令需要使用 `rd` 的值），数据前推无法解决这个问题，故必须存在一个 load delay slot（即插入一条无关指令，或插入 `nop`），防止时序错误



    - 可以通过编译器的代码调度 Code scheduling 来完成代码的重排，在保持代码功能的前提下尽可能解决 load delay slot

---

Lec14 流水线 Pipeline (II)

冲突 Hazard

控制冲突 Control hazards

- 控制跳转指令（如 `beq`）的计算完成之前，已经有接下来的指令进入流水线，如果发生跳转，则存在计算错误
- 解决方案：

- 插入气泡 bubble，让其它指令等待跳转指令执行完成后再执行
- 静态预测：假设（不）发生跳转，按照假设执行接下来的指令；如果预测正确，则正常执行；如果预测失败，则清空所有流水线寄存器的值，重新开始执行接下来的指令
- 动态预测：额外维护一个寄存器，储存这条指令的跳转历史，用于预测是否发生跳转（如使用两位状态机，存储 strong/weak (not) taken，每次执行语句时状态转移）（现代CPU多采用这种预测方式）

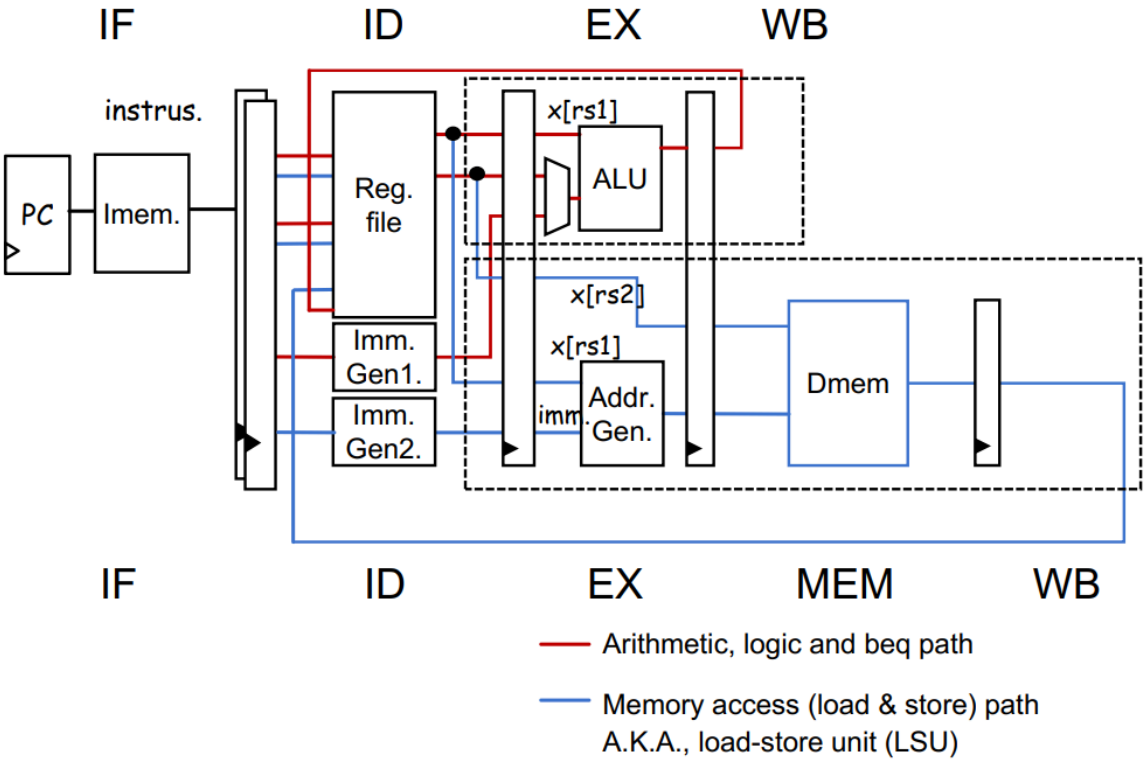
Lec15 多发射 Multi-issue

概念

- 单发射 Single-issue：每次取指令时只取1条指令执行
- 多发射 Multi-issue：每次取多条指令（同时）执行，需要硬件支持
- 目前的CPU一般是每次发射4-6条指令
- 多发射不是多核
- 多发射可以和流水线 pipelining，单指令流多数据流 SIMD，多线程等结合设计
- 多发射可以减少每条指令需要执行的周期数 CPI

多发射的硬件支持

- 可以把指令分成运算 Arithmetic 和访存 Memory access 两种，通过两条流水线同时处理
- 运算流水线不需要经过数据寄存器，故直接跳过该步骤



- 两条流水线之间也会存在数据前推 forwarding，故需要支持流水线间的数据传输

指令排布 Instruction Scheduling

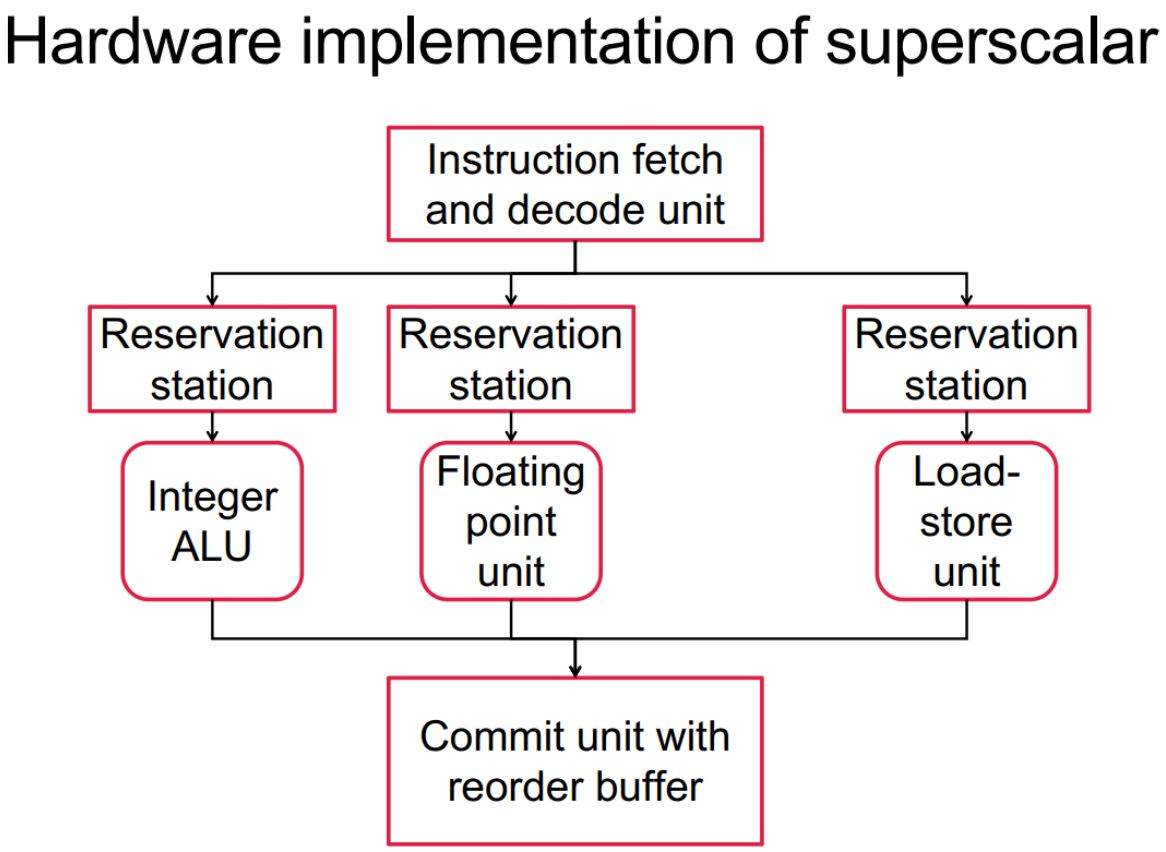
- 对于循环指令 `for (int i = 1000; i>0; i-=1)`，在双发射流水线上，可以改为 `for (int i=1000; i>0; i-=2)`，并改变循环内代码

静态多发射

- 使用软件进行指令排布，需要开发多种编译器来适配不同发射数的硬件
- 在编译阶段，编译器将可以并行执行的指令打包到不同的发射槽 issue slots 中，并负责检测这些指令之间是否存在冲突 Hazards
- 也叫超长指令字 very long instruction word, VLIW
- 在AI加速运算中常用（适合对特定代码进行加速）

动态多发射

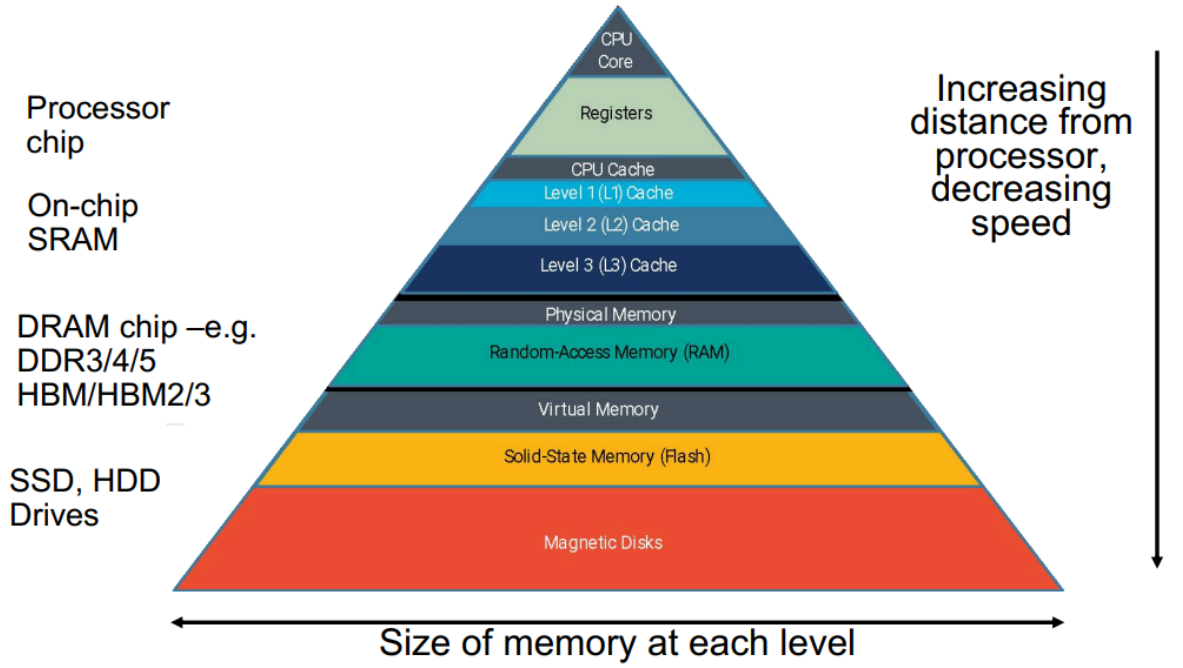
- 使用硬件动态检查指令流，找出互不干扰的指令，并将它们分配到可用的发射槽中
- 硬件需要实时检测并解决冲突
- 也叫超标量流水线 superscalar，是大部分主流CPU/GPU的设计（保证兼容性）

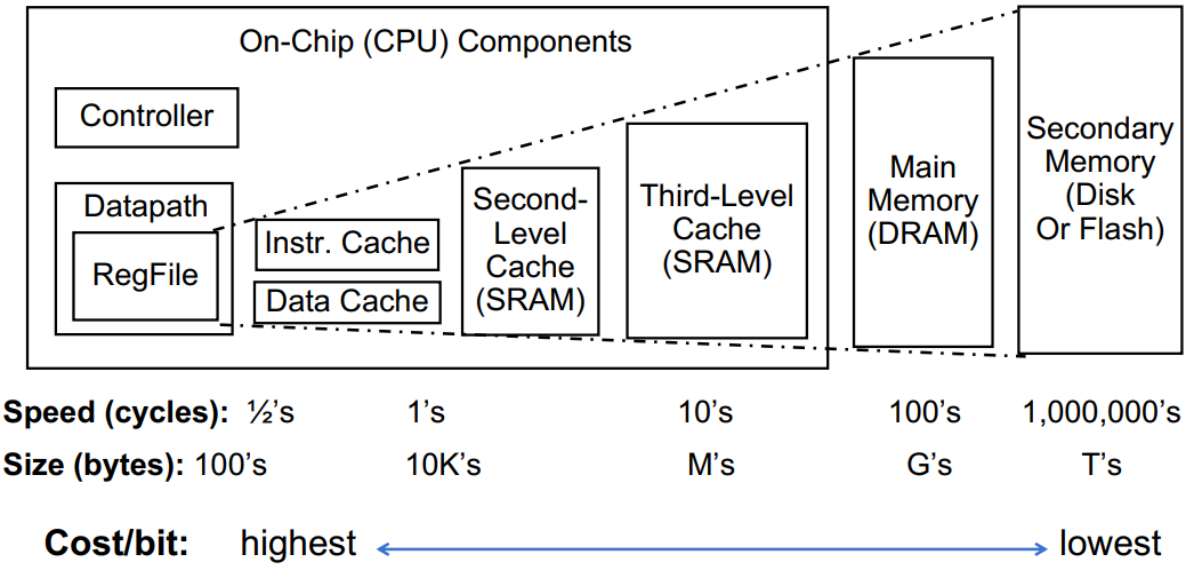


- 采用 Out of Order, OoO 不按序执行指令：指令只要数据准备好了，且对应执行单元有空，即可直接执行，可以跳级执行

Lec16 缓存 Cache (I)

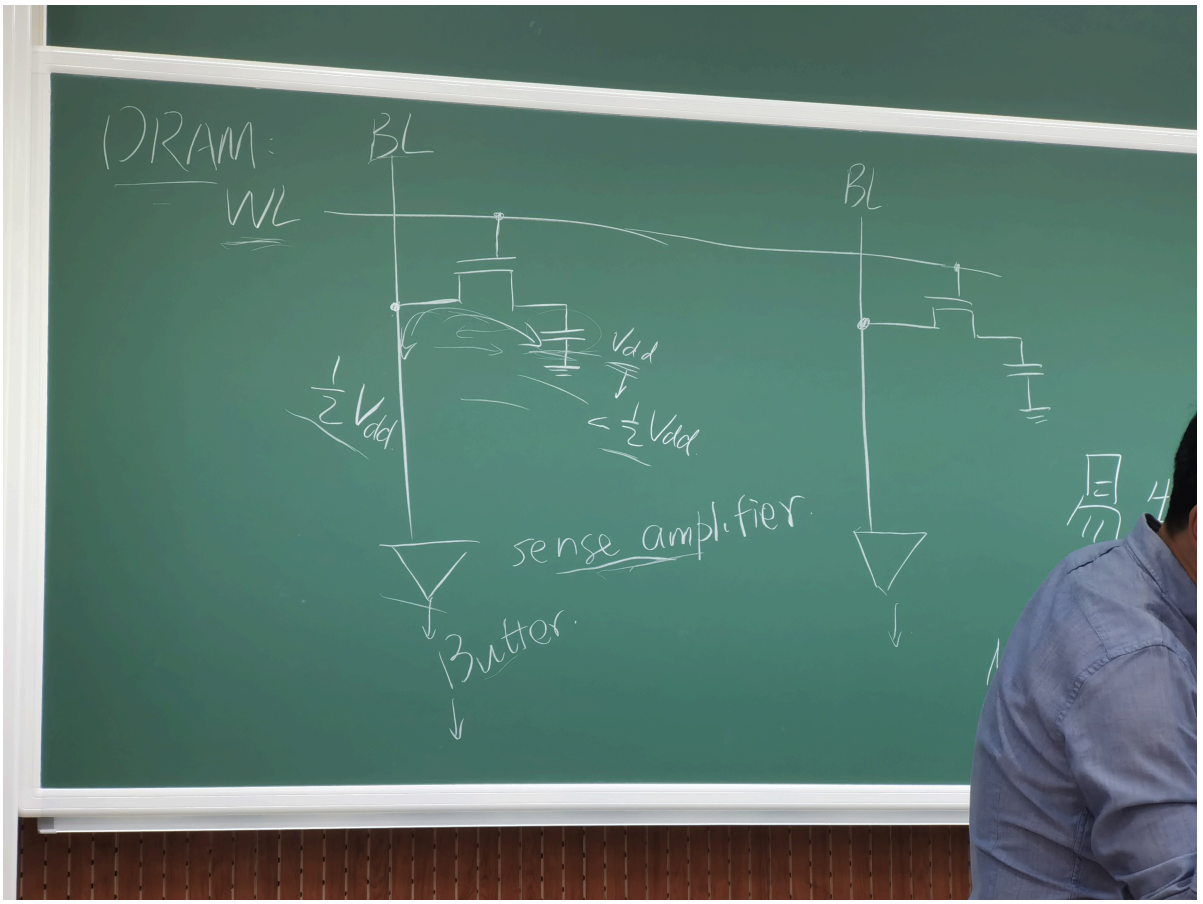
多级存储 Memory Hierarchy





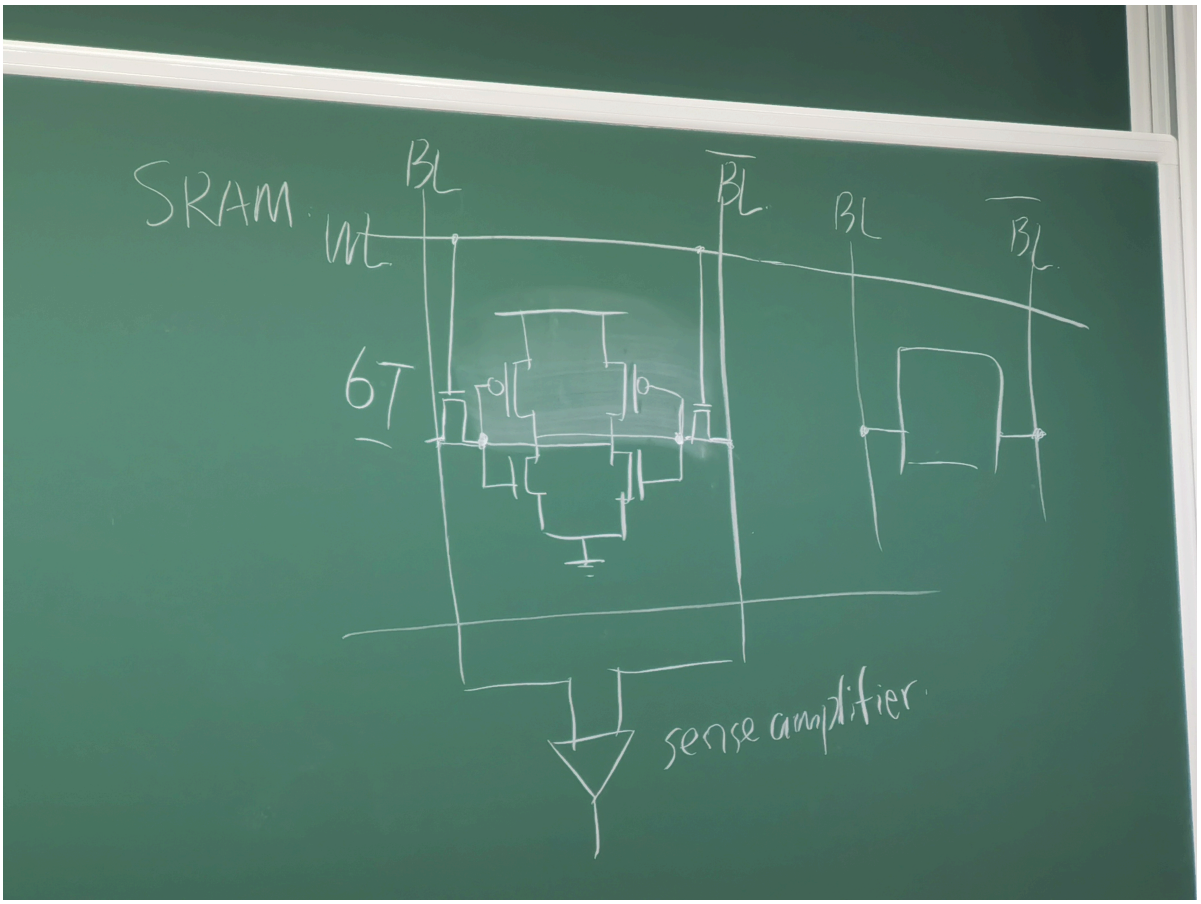
DRAM：Dynamic Random Access Memory

- 主存 Main memory 使用 DRAM
- 属于易失性 Volatile 存储器：掉电丢数据
- 电量会自然泄漏，需要定期刷新（充电）
- 读取操作是破坏性的，读取后需要重新写入（充电）
- 读取速度 0.5ns ~ 1ns



SRAM：Static Random Access Memory

- 属于易失性 Volatile 存储器：掉电丢数据
- 电量不会泄漏，不需要定期刷新
- 相比于DRAM，造价更贵，速度更快，密度更低
- 读取速度 0.5ns



硬盘（二级存储 Secodary Memory）

- 非易失性 non-volatile 存储器
- 硬盘 Hard Disk Drive, HDD：更便宜，更慢（5ms ~ 10ms）
- 固态硬盘 Solid-State Drive, SSD：更贵，更快（$40\mu s \sim 100\mu s$）

缓存 Cache

- 用于存储数据的拷贝
- 慢于 Register，快于 DRAM
- 有些设计中，利用HBM技术，缓存被集成在CPU/GPU的芯片周围，以达到更快访问的目的，这种拼图式的设计理念叫做芯粒 Chiplet

局部性原理 Principle of Locality

- 时间局部性 Temporal Locality：一段数据如果当前被访问，在一段时间内很有可能被再次访问
  - 如循环变量、每个循环更新的变量、栈操作等
- 空间局部性 Spatial Locality：一段数据如果被访问，在一段时间内，它周围地址的数据很有可能被访问
  - 如按序遍历访问数组、指令按序执行等
- 局部性原理和多级存储共同使得计算机可以拥有近似于最便宜的存储量级的存储和近似于最快存储的速度

单级缓存 Simple Cache

- 缓存的管理由硬件缓存管理器 hardware cache controller 负责
- 缓存不是CPU的必要结构，但是可以提升CPU性能

缓存结构

- 缓存以缓存块 cache block/line 为单位存储数据
- 缓存块大小 cache block/line size：每个缓存块能存储的字节 `byte` 数（按字节数计，$C_B = 2^b$，b 为偏移量 offset 位数）
- 缓存容量 Capacity：整个缓存能够存储的字节 `byte` 数 （按字节数计，C）
- 缓存标签 Cache Tag：内存地址高几位，优化后长度为 $t = w - b$ （w 为地址位数）
- 优化：由于块对齐 block alignment，对于16位地址，只需要存储高14位即可（低2位固定为0）
- 优化：对于更大的缓存块（如 `8 byte`），每个缓存块可以存储多个数据（如两个 word `4 byte * 2`），每个缓存块的标签 Tag 对应多个连续地址，地址低位可以进一步压缩（省略低3位）
- 合法位 Valid bit：标记缓存块内容对于当前程序是否可访问；如果地址匹配后发现合法位是 `0`，则直接判断 cache miss
- 冷启动 Cold Start：缓存为空，合法位全0

缓存访问步骤

- 处理器 Processor 试图访问内存地址 `0x12F0`
- 缓存 Cache 检查是否存储了 `@address 0x12F0`
  - 如果存储了，缓存命中 cache hit 并直接读入地址对应内容
  - 如果没存储，缓存未命中 cache miss，缓存将地址发送给内存，内存读取地址对应内容并返回给缓存，然后缓存用取到的数据替换 Cache Replacement 掉某些缓存内的旧数据(victim)
- 缓存将地址对应内容发送给处理器，处理器把数据加载进寄存器

Lec17 缓存 Cache (II)

替换策略 Replacement Policy

- 当缓存充满后再次发生未命中 cache miss，需要踢出 evict 一个牺牲者数据 victim

最近最少访问算法 Least recently used, LRU

- 总是先踢出上一次使用最早的数据
- 实现：在缓存中，为每一行增加一个 LRU 字段，记录访问的顺序（记录离上次访问的次数；当前被访问的数据 LRU=0）
- 对于时间局部性 Temporal locality 较高的情况，LRU命中效率优秀，但是由于更新时间复杂度过高，且空间复杂度过高，在实际工程中较少使用（常用随机替换/FIFO/伪LRU）

其它替换策略

- 最近最多访问算法 Most Recently Used, MRU：总是替换最新被访问的数据
- 先进先出 First in First out, FIFO：总是替换最早进入缓存的数据
- 后进先出 Last in First out, LIFO：总是替换最新进入缓存的数据
- 随机算法 Random：随机替换某一条数据（当时间局部性 Temporal locality 较低时，随机算法也有不错的表现）

写入策略 Write Policy

- 此处特指使用全相联映射缓存 Fully associative cache 时的写入策略
- 当发生写入操作时，如果仅写入内存，缓存内的数据未被更新，会影响后续的读入逻辑，因此需要写入策略

写穿 Write-through

- 当发生写入时，同时写入缓存和内存
- 可以使用写缓冲器 Write buffer 进行优化：在不影响程序运行逻辑的前提下，写入内存步骤可以交给写缓冲器，而不用让CPU停下来等待存储过程

写回 Write-back

- 为每一个缓存行设置一个 dirty bit，存储内容是否发生修改，初始化为 0
- 当发生写入时，仅写入缓存，并把对应的 dirty bit 设置为 1
- 当发生缓存替换时，检查被踢出的缓存行，如果 dirty bit 为 1，则将值写回内存

写分配 Write-Allocate

- 当发生写入而数据不在缓存中时，应当先将待写数据导入缓存，再进行缓存数据更新

非写分配/绕写 No-Write-Allocate / Write Around

- 当发生写入而数据不在缓存中时，直接在内存中写入数据，不将数据存入缓存

存储策略 Placement Policy

全相联映射缓存 Fully associative cache

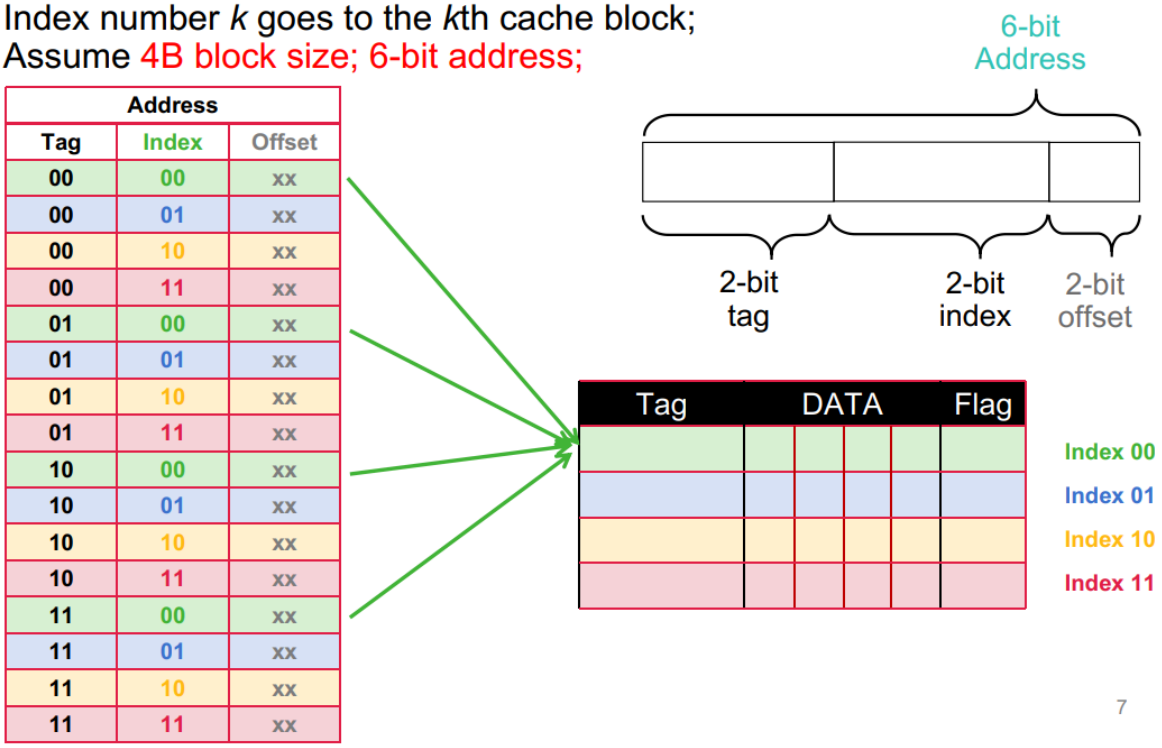
- 主存中的任何一个数据块 block 都可以存放在缓存中的任意一个缓存行中
- 以上所有的缓存策略都是对于全相联映射缓存 Fully associative cache 而言的

直接映射缓存 Direct Mapped Cache

- 对于每一条数据，都有唯一的缓存行与之对应
- 一条内存地址被分为3部分：
  - Tag：长度为剩余的长度
  - Index：和唯一的缓存行对应，长度为 log_2缓存行的数量
  - Offset：表示同一缓存行的第几个数据，长度为 log_2缓存行存储数据的字节数

Direct Mapped Cache-Examples I

Index number *k* goes to the *k*th cache block;
Assume 4B block size; 6-bit address;



- 在访问数据时，仅查找对应编号 Index 的缓存行的标签 Tag 即可
- 在替换数据时，仅替换同一编号 Index 的缓存行
- 硬件设计更简单，单次访问/替换数据更快
- 如果采用写回 Write-back 策略，则设置 dirty bit，踢出时更新内存数据
- 如果采用写穿 Write-through 策略，仍然可以使用 Write buffer 优化
- 同时，也可以使用写分配/非写分配策略

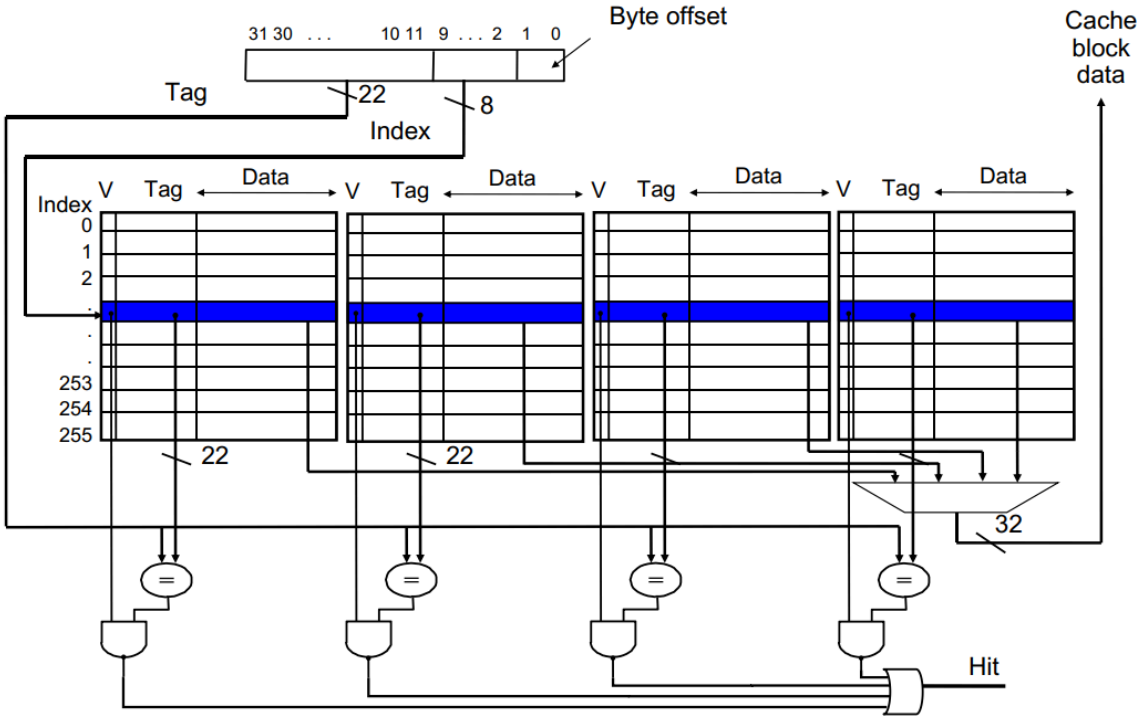
缓存失效 Cache Misses：3Cs

- 强制缺失/冷启动缺失 Compulsory Misses
  - 程序第一次访问某个内存块（冷启动/上下文切换）
  - 解决方案：增加缓存块 block 大小
- 容量缺失 Capacity Misses
  - 缓存无法包含程序运行需要的所有内存块
  - 解决方案：增加缓存容量
- 冲突缺失 Conflict/Collision Misses
  - 多个内存地址映射到相同的缓存位置，即使缓存没有占满，也会发生冲突
  - 乒乓效应 Ping-pong effect：两个相同 index 的内存数据发生冲突缺失，并交替进出缓存
  - 仅在直接映射缓存中有明显体现
  - 解决方案：增加缓存容量；增加相联度 Associativity

Lec19 组相联缓存 Set-Associative Cache

组相联缓存 Set-Associative (SA) Cache

- 数据只能放置在指定索引 Index 的缓存处，但是每个索引 Index 对应 `N` 个缓存行（1组），称为N路组相联缓存 N-way Set-Associative Cache
- 每个缓存行仍然将内存地址分为 tag, index, offset 三段
- 每个缓存行仍然需要一个 valid bit，标记数据是否可访问
- 仍然有写策略：Write-back, Write-through
- 仅在缓存组内有替换策略
- 写回 Write-back 时，只需要更改数据、Flag（Dirty bit、LRU、Valid bit）数据，不用修改整个组的索引 index
- Index 位数由组的数量决定
- 可以避免较多冲突缺失 Conflict/Collisioin Misses，避免乒乓效应 Ping-pong effect



- 在上图中，蓝色的缓存区域是一个组，整个缓存有 255 个缓存组（8位索引 index）

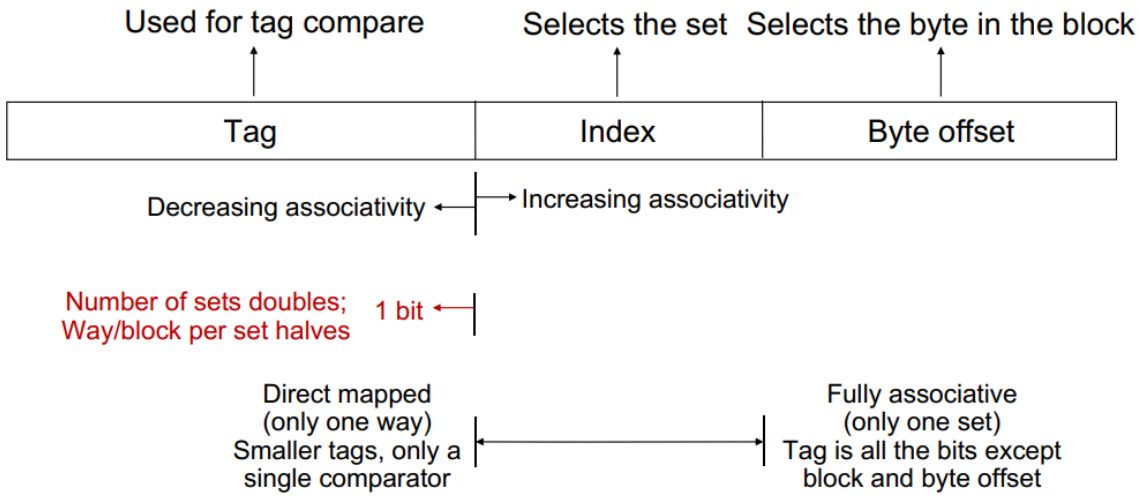
相关术语与参数 Set-Associative Cache Terminology

- 缓存大小 Cache capacity/size: `C`
- 偏移位宽 Bit width of offset bits：`o`
- 缓存块（行）大小 Cache block size：$C_B = 2^o$
- 缓存块数量 Number of cache blocks：$M = \frac{C}{C_B}$
- 内存地址位宽 Bit width of memory address：`w`
- N路组相联缓存 N-way SA cache：每个组有 `N` 个缓存块（行）
- 索引位宽 Bit width of index：`i`
- 组数量 #set / associativity：$\frac{M}{N} = 2^i$ （必为2的指数幂）

- 标签位宽 Bit width of Tag：`t = w - o - i`
- $C = M \times C_B = 2^o \times 2^i \times N$

缓存指标与性能 Cache Metrics & Performance

缓存指标 Cache Metrics



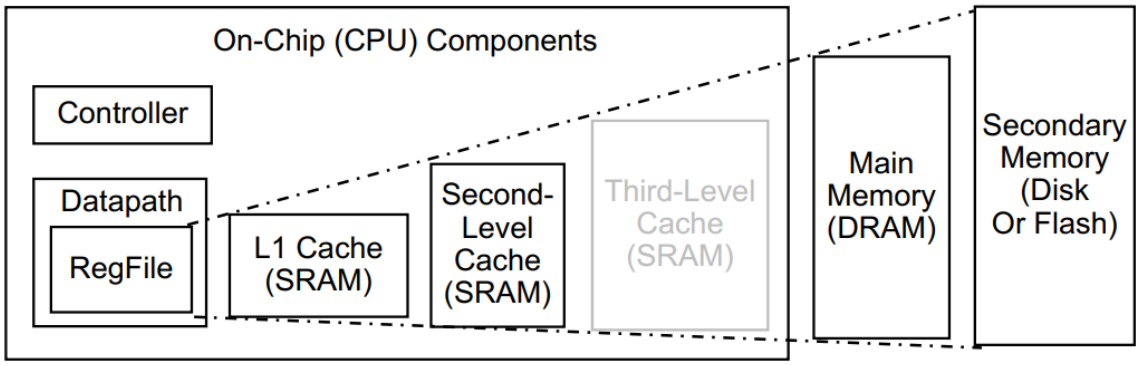
- 命中率 Hit rate：在一段程序执行中的命中频率
- 未命中率 (Local) Miss rate：1 - Hit rate
- 未命中惩罚 Miss penalty：把一个缓存块（行）数据从低一级存储转移进入当级缓存消耗的时间，即下一级缓存的 AMAT 或内存的访问时间
- 命中时间 Hit time：命中情况下，访问缓存的时间
- （单级缓存）平均访存时间 Average Memory Access Time (AMAT)：
  - `AMAT = Hit_time + Miss_rate * Miss_penalty`
- 多级缓存未命中率
  - 全局未命中率 Global Miss rate（假设两级缓存）

    $= \frac{L2\ Misses}{Total\ Accesses} = Local\ Miss\ rate\ of\ L_1 \times Local\ Miss\ rate\ of\ L_2$
  - 局部未命中率 Local Miss rate：$Local\ Miss\ rate\ of\ L_2 = \frac{L2\ Misses}{L1\ Misses}$

AMAT: Multi-Level Cache

- Assume L1 & L2 cache only:
- L1 hit time 1 cycle; miss rate 5%;
- L2 hit time 5 cycles, miss rate 10%, miss penalty 100 cycles;

L1 AMAT = Time for an L1 hit + L1 Miss rate × L2 AMAT



L2 AMAT = 5 clock cycles + 10% × 100 clock cycles = 15 clock cycles

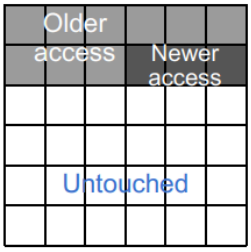
AMAT = 1 clock cycle + 5% × 15 clock cycles = 1.75 clock cycles

缓存性能 Cache Performance

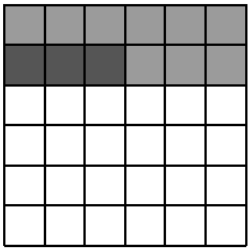
- 如果试图减少命中时间 Hit time，需要使用更小的缓存，但这会导致缓存命中率下降
- 为了减少未命中惩罚 Miss penalty，可以使用预取 Prefetch 技巧（可以硬件/软件实现）：在使用数据之前提前将数据载入缓存，以减少流水线停顿；会占据缓存空间，因此对于更大的缓存更有效；会占用CPU访存的总线带宽 Bandwidth，因此对于更高的总线带宽更有效
- 为了减少未命中惩罚 Miss penalty，还可以使用牺牲者缓存 Victim Cache，将被替换出缓存的地址置入 Victim Cache（一般是一个非常小的全相联缓存）
- 代码优化：矩阵乘法

Cache-aware Program Optimization

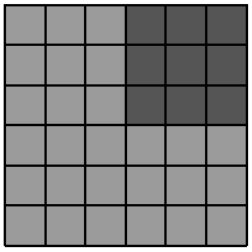
- Cache blocking: GEMM example C = A x B (row-major)



C



A



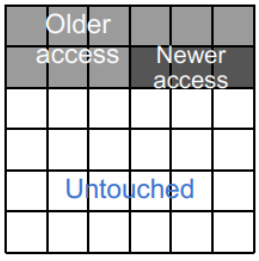
B

```
for (int i = 0; i<n; i++){
for (int j = 0; j<n; j++)
  {
     int cij = C[i*n+j];
     for (int k = 0; k < n; k++)
     {
        cij += A[i*n+k] * B[k*n+j];
        C[i*n+j] = cij;
     }
  }}
```

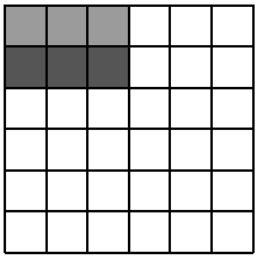
```
for (int i = 0; i < n; i++){
for (int j = 0; j < n; j++)
  {
     int cij = C[i*n+j];
     for (int blk_k = 0; blk_k < blk_num; blk_k++)
     {  for (int sk=0; sk < blk_size; sk++){
        cij += A[i*n+blk_k*blk_size+sk] *
               B[(blk_k*blk_size+sk)*n+j];
        C[i*n+j] = cij;}
     }
  }}   %blk_num = 2; sk: 0-2
```

Cache-aware Program Optimization

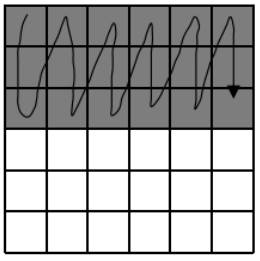
- Cache blocking: GEMM example C = A x B (row-major)



C



A



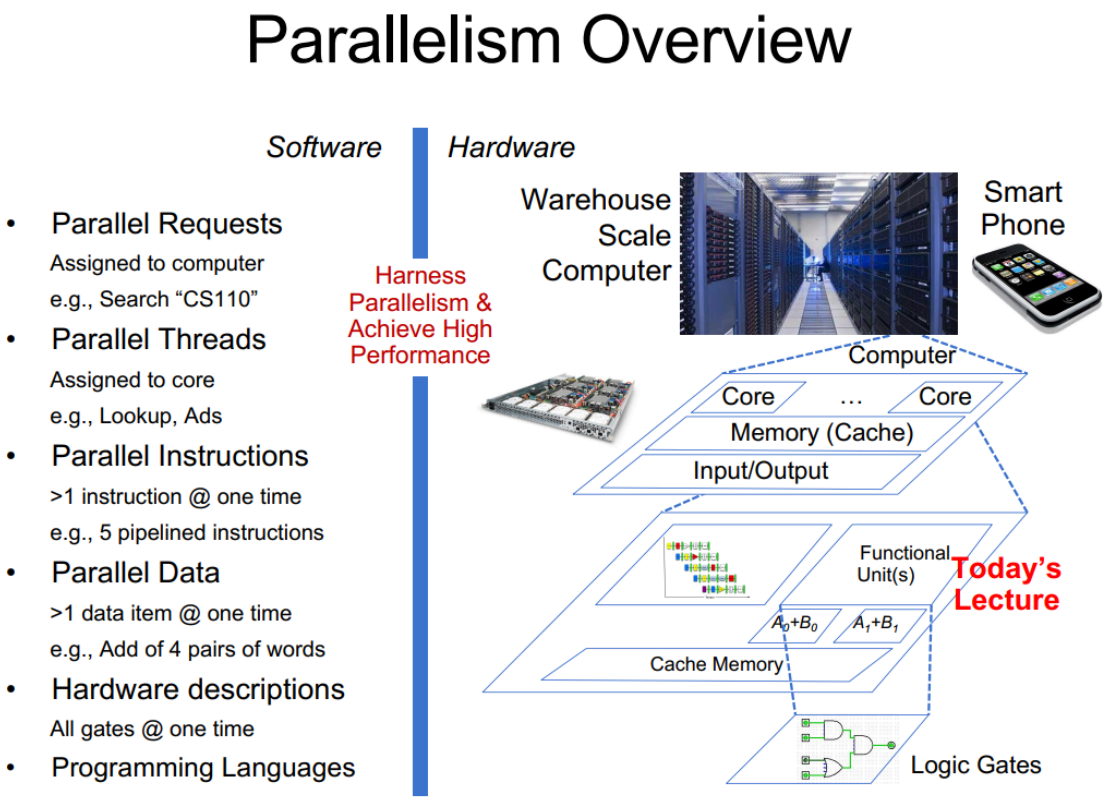
B

```
for (int blk_k = 0; blk_k < blk_num; blk_k++){
for (int i = 0; i < n; i++){
for (int j = 0; j < n; j++)
  {
     int cij = C[i*n+j];
     for (int blk_k = 0; blk_k < blk_num; blk_k++)
     {  for (int sk=0; sk < blk_size; sk++){
        cij += A[i*n+blk_k*blk_size+sk] *
               B[(blk_k*blk_size+sk)*n+j];
        C[i*n+j] = cij;}
     }
}}}  blk_num = 2; sk 0-2;   blk_k: 0-1
```

Better A reuse and less A cache miss

Lec20 并行与阿姆达尔定律 Paralellism, Amdahl's Law

并行处理概述



- 其中，线程级并行是在程序间的，指令级并行是在同一个程序中的
- 对于运行加速，可以采用两种基本方式：
  - 多程序并行 Multiprogramming：并行运行多个程序；较好实现
  - 并行计算 Parallel computing (parallel processing program)：加速一个程序的运行；较难实现
- 单任务并行是更难实现的

阿姆达尔定律 Amdahl's Law

- 加速比 $Speed\_up = \frac{Original\_execution\_time}{Execution\_time\_after\_improvement} = \frac{1}{(1-F)+\frac{F}{S}}$
- 其中 F 为受改进部分（可并行部分）占比，S 为受改进部分的加速比
- 强扩展性 Strong Scaling：问题总规模保持不变，但增加处理器的数量，以缩短同一任务时间
- 弱扩展性 Weak Scaling（多数情况）：随着处理器数量的增加，相应地增加问题的总规模，使得每个处理器负责的工作量保持不变，以增加相同时间处理的任务量

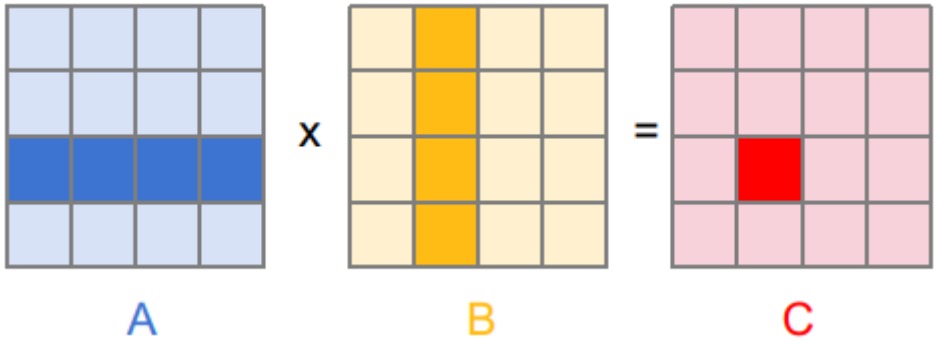
费林分类法 Flynn's Taxonomy

- 单指令单数据 Single Instruction, Single Data (SISD)：任意给定时刻，CPU只能执行一条指令，这条指令只能处理一个数据对象
- 单指令多数据 Single Instruction, Multiple Data (SIMD)：任意给定时刻，CPU只能执行一条指令，这条指令能够处理多个数据对象
- 多指令多数据 Multiple Instruction, Multiple Data (MIMD)：任意给定时刻，CPU能执行多条指令，这条指令能够处理多个数据对象

- 多指令单数据：不常见

单指令多数据 SIMD

数据级并行 Data-Level Parallelism, DLP



- 如上图，可以采用4个乘法器来进行并行运算
- DLP在图像处理中应用广泛，可以通过增加硬件数量来达到并行加速目的
- 可以使用RISC-V向量拓展的方式来达到并行执行目的

SIMD 指令

- 获取一条指令，做多个指令的工作
- 需要有对应的存储结构：向量寄存器堆 vector register file, VRF
- 需要有并行执行模块
- 需要考虑带宽 Bandwidth，保证数据能够并行传输到执行模块中
- 加速的原理是减少指令数量 Instruction per program

Intel SIMD 指令集

- 多媒体扩展指令集 Multi-Media eXtension, MMX
- 流式单指令多数据扩展 Streaming SIMD Extension, SSE
- 高级向量扩展 Advanced Vector eXtension, AVX

多媒体扩展指令集 Multi-Media eXtension, MMX

- 使用64位的宽寄存器
- 数据打包 Packing：64位的寄存器可以被拆分成不同的份数，如8个8位 Packed byte，4个16位 Packed word，2个32位 doubleword 或 1个64位 quadword

SSE/SSE2 Floating-Point Instructions

Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm, mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm, mem/xmm	
	DIV{SS/PS/SD/PD} xmm, mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

xmm: one operand is a 128-bit SSE2 register
mem/xmm: another operand is in memory or an SSE2 register
{SS} Scalar Single precision FP: one 32-bit operand in a 128-bit register
{PS} Packed Single precision FP: four 32-bit operands in a 128-bit register
{SD} Scalar Double precision FP: one 64-bit operand in a 128-bit register
{PD} Packed Double precision FP, or two 64-bit operands in a 128-bit register
{A} 128-bit operand is aligned in memory
{U} means the 128-bit operand is unaligned in memory
{H} means move the high half of the 128-bit operand
{L} means move the low half of the 128-bit operand

- 在C语言中，存在内建函数 Intrinsics 来实现并行运行，其编译出的汇编代码会使用 SIMD 指令执行对应功能
- 例如，对于数组 `array` 中全部元素做开方操作，可以做如下优化：

```
for each f in array
{
   load f to the floating-point register
   calculate the square root
   write the result from the register to memory
}
```

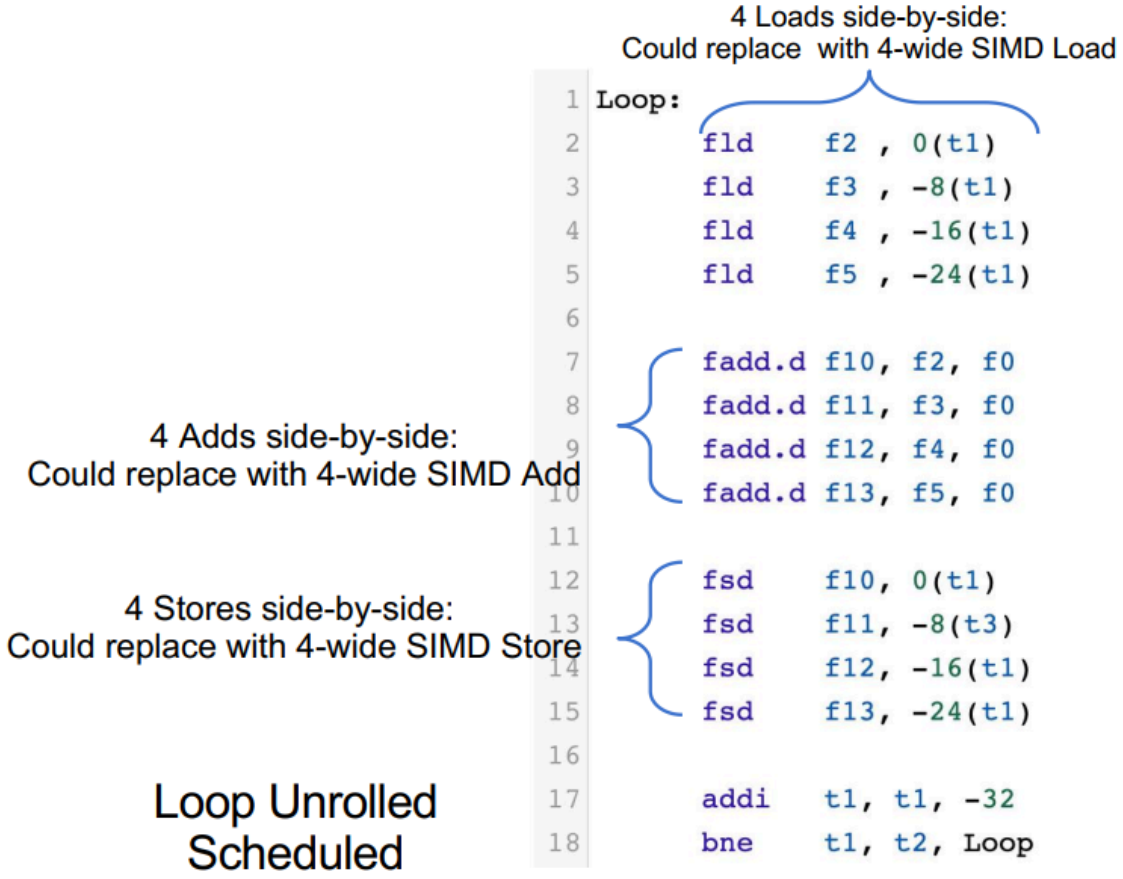
```
for each 4 members in array
{
   load 4 members to the SSE register
   calculate 4 square roots in one operation
   store the 4 results from the register to memory
}
```

SIMD style

循环展开 Loop Unrolling

- SIMD 希望能够把内存中相邻的值并行操作，而循环能够满足这一点
- 假设循环次数 loop limit 是4的倍数，则以下例子成立：

```
1 Loop: fld    f2 , 0(t1)      # $f2=array element
2       fadd.d f10, f2, f0     # add s to $f2
3       fsd    f10, 0(t1)      # store result
4       addi   t1, t1, -8      # t1 = t1 -8
5       bne    t1, t2, Loop    # repeat loop if t1 != t2
```



- 如果存在不满载的循环（即：零头），则需要再多消耗一个满载循环的时间来处理

RISC-V 向量拓展

RISC-V Vector Extension

- 32 vector registers
- Need to setup length of data and number of parallel registers to work on before usage (vconfig)! vflw.s: vector float load word . stride: load a single word, put in v1 'vector length' times
- vsetvl: ask for certain vector length – hardware knows what it can do (maxvl)!

```
    # assume x1 contains size of array
    # assume t1 contains address of array
    # assume x4 contains address of scalar s
    vconfig 0x63        # 4 vregs, 32b data (float)
    vflw.s v1.s, 0(x4)  # load scalar value into v1

loop:
    vsetvl x2, x1       # will set vl and x2 both to min(maxvl, x1)
    vflw v0, 0(t1)      # will load 'vl' elements out of 'vec'
    vfadd.s v2, v1, v0  # do the add
    vsw v2, 0(t1)       # store result back to 'vec'
    slli x5, x2, 2      # bytes consumed from 'vec' (x2 * sizeof(float))
    add t1, t1, x5      # increment 'vec' pointer
    sub x1, x1, x2      # subtract from total (x1) work done this iteration (x2)
    bne x1, x0, loop    # if x1 not yet zero, still work to do
```

SSE 内建代码 Intrinsics

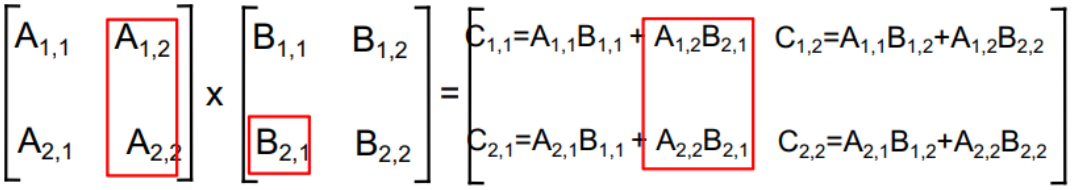
Intrinsics:	Corresponding SSE instructions:
• Vector data type:	
_m128d	
• Load and store operations:	
_mm_load_pd	MOVAPD/aligned, packed double
_mm_store_pd	MOVAPD/aligned, packed double
_mm_loadu_pd	MOVUPD/unaligned, packed double
_mm_storeu_pd	MOVUPD/unaligned, packed double
• Load and broadcast across vector	
_mm_load1_pd	MOVSD + shuffling/duplicating
• Arithmetic:	
_mm_add_pd	ADDPD/add, packed double
_mm_mul_pd	MULPD/multiple, packed double

- 并行操作对应的元素需要相邻（处理器也可以宽容一隔一的访问，但是简单起见，假设只能相邻）
- 并行操作的操作应该相同，且没有互相依赖关系
- 对于矩阵乘法，存在以下并行机会：对应元素相乘；结果不同位置值的计算，例如：

  目标：$A_{2\times2} \times B_{2\times2} = C_{2\times2}$

  并行：同时并行2个计算

  - 并行计算 $A_{1,1}B_{1,1}$ 和 $A_{2,1}B_{1,1}$
  - 并行计算 $A_{1,2}B_{2,1}$ 和 $A_{2,2}B_{2,1}$
  - 并行计算 $A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$ 和 $A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$



- Toy example, 2x2 matrix multiplication

```
//Toy example, 2x2 matrix multiplication for CS110 2025, ShanghaiTech University, all rights reserved
#include <stdio.h>
#include <time.h>
#include <emmintrin.h>
#include <immintrin.h>
void main(){
  double A[4]__attribute__((aligned(16)));
  double B[4]__attribute__((aligned(16)));
  double C[4]__attribute__((aligned(16)));
  int ida = 2;
  int i = 0;
  __m128d c1, c2, a, b1, b2;
  A[0] = 1.0;A[1] = 1.1;A[2] = 0.0;A[3] = 1.0;
  B[0] = 1.0;B[1] = 2.0;B[2] = 5.0;B[3] = 4.0;
  C[0] = 0.0;C[1] = 0.0;C[2] = 0.0;C[3] = 0.0;
  c1 = _mm_load_pd(C+0*ida);
  c2 = _mm_load_pd(C+1*ida);
  for (i=0;i<2;i++){
   a = _mm_load_pd(A+i*ida);
   b1 = _mm_load1_pd(B+i+0*ida);
   b2 = _mm_load1_pd(B+i+1*ida);
   c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
   c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
   }
   _mm_store_pd(C+0*ida,c1);
   _mm_store_pd(C+1*ida,c2);
   printf("[%g,%g,%g,%g]\n",C[0],C[1],C[2],C[3]);
  return;
}
```

Lec21 线程级并行 Thread-level Paralellism

线程 Thread

- 每个线程拥有自己的寄存器 / 栈指针 / 程序计数器 PC，与同进程线程共享内存（堆 / 全局变量）
- 线程间通过快速切换来达到类似于并行的效果
- 线程间存在调度机制

分叉-合并模型 Fork-Join Model

- 程序进程可以把自己分叉 fork 为不同的线程，达到（理论）并行的效果（多核/多线程）
- 多指令多数据 MIMD：可以在多核/多计算机系统上执行

OpenMP

- OpenMP 是用于多线程并行处理 Multi-processing 的C语言拓展
- 编译时加上Flag `-fopenmp`
- 使用 Shared / Private 来区别变量是否在线程间共享

- 代码示例：

```
#include <stdio.h>
#include <omp.h>

int main()
{
    #pragma omp parallel
    {   // '{' Must be on this line without empty lines!
        int tid = omp_get_thread_num();
        printf("Hello World from thread %d!\n", tid);
        if (id == 0)    // Main thread's id = 0
        {
            printf("Number of threads = %d\n", omp_get_num_threads());
        }
    }
}
```

常用函数

- `omp_set_num_threads(x)` 把线程数量设置为 `x` （如果不指定，会自动使用计算机支持并行的最大线程数量，即内核数量 * 每个内核支持并行的线程数量，即使用虚拟线程技术 hyper-threading）
- `num_th = omp_get_num_threads()` 获取线程数量
- `th_ID = omp_get_thread_num()` 返回正在执行的线程ID

共享变量/私有变量 Shared / Private Varible

- 共享变量 Shared variables：所有线程都可以读/写的同一个变量对象
- 私有变量 Private variables：每个线程专有一份的变量对象
- 示例代码：

```
int var1, var2;
char *var3 = malloc(10 * sizeof(char));
# pragma omp parallel private(var2)
{
    int var4;
}
// var1: shared (default)
// var2: private
// var3: shared (heap)
// var4: private (in thread's stack)
```

优势与劣势 Pros & Cons

- OpenMP 显式地编程并行化模型，使程序员对并行化有完全控制权
    - 优势：编译器指令简单易用
    - 优势：传统的串行代码 serial code 不需要重写
    - 劣势：编译器必须支持 OpenMP
    - 劣势：阿姆达尔定律 Amdahl's law 会一定程度上限制性能继续提升

- OpenMP 使用共享内存环境中的多线程
  - 优势：利用了共享内存的优势，程序员不用过度担心数据存放位置
  - 劣势：代码只能在共享内存中运行，无法跨机器或在集群上运行
  - 劣势：同步共享资源的使用非常困难

代码优化

- 可以使用 `pragma omp for` 来实现多线程同步运行的 `for` 循环
- 注意，循环中不能含有 `break` `return` `exit` `goto` 等退出循环的代码，也不能在循环中存在数据依赖关系

```
#pragma omp parallel for        // 可以省略外层的 #pragma omp parallel
for (int i=0; i < LENGTH; i++)
{
    arr[i] = ...;
}
```

- 可以使用 `double omp_get_wtime(void);` 来获取当前时间（秒）

例：矩阵乘法优化

- 使用 OpenMP 线程级并行 TLP 优化：

```
double dgemm_scalar(int N, double *A, double *B, double *C)
{
    double start_time = omp_get_time();
    double Cij;
    int i,j,k;
    #pragma omp parallel for private (Cij, i, j, k)
    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            Cij = 0;
            for (k=0; k<N; k++)
            {
                Cij += A[i+k*N] * B[k+j*N];
            }
            C[i+j*N] = Cij;
        }
    }
    double run_time = omp_get_wtime() - start_time;
    return run_time;
}
```

- 增加数据级并行优化 DLP：

```
double dgemm_simd(int N, double *A, double *B, double *C)
{
    memset(C, 0, N * N * sizeof(double));
    double start_time = omp_get_wtime();
    #pragma omp parallel for        // 默认线程级并行优化下一次循环
    for (int i = 0; i < N; i++)
```

```
    {
        for (int j = 0; j < N; j++)
        {
            __m256d sum_vec = _mm256_setzero_pd();
            for (int k=0; k <= N - 4; k += 4)
            {
                __m256d a_vec = _mm256_set_pd(
                A[i + (k+3)*N], A[i + (k+2)*N],A[i + (k+1)*N],A[i + k*N]);
                __m256d b_vec = _mm256_loadu_pd(&B[k + j*N]);
                #ifdef __FMA__ {sum_vec = _mm256_fmadd_pd(a_vec, b_vec,
sum_vec);}
                #else {sum_vec = _mm256_add_pd(_mm256_mul_pd(a_vec, b_vec),
sum_vec);}
                #endif
            }
            double sum = sum_vec[0] + sum_vec[1] + sum_vec[2] + sum_vec[3];
            for (; k < N; k++) {sum += A[i + k*N] * B[k + j*N];} // trailing
elements
            C[i + j*N] = sum;
        }
    }
    double run_time = omp_get_wtime() - start_time;
    return run_time;
}
```

- 使用 OpenMP 指令简写为：

```
double dgemm_reordered(int N, double *A, double *B, double *C)
{
    memset(C, 0, N * N * sizeof(double));
    double start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < N; i++)
    {
        for (int k = 0; k < N; k++)
        {
            double aik = A[i + k*N];
            #pragma omp simd    // SIMD 显式要求循环向量化执行
            for (int j = 0; j < N; j++)
            {
                C[i + j*N] += aik * B[k + j*N];
            }
        }
    }
    double run_time = omp_get_wtime() - start_time;
    return run_time;
}
```

- 也可以进分块优化：

```
#define BLOCK_SIZE 64
double dgemm_blocked_simd(int N,double*A,double*B,double*C){
    memset(C,0, N * N *sizeof(double));
    double start_time =omp_get_wtime();
```

```
    #pragma omp parallel for collapse(2)    // 线程级并行优化接下来的2层循环
    for(int i0 =0; i0 < N; i0 += BLOCK_SIZE)
    {
        for(int j0 =0; j0 < N; j0 += BLOCK_SIZE)
        {
            int i_max =(i0 + BLOCK_SIZE < N)? i0 + BLOCK_SIZE : N;
            int j_max =(j0 + BLOCK_SIZE < N)? j0 + BLOCK_SIZE : N;
            // Outer loop for blocks, inner loop for inside-block data processing
with SIMD
            for(int i = i0; i < i_max; i++)
            {
                for(int j = j0; j < j_max; j++)
                {
                    __m256d sum_vec =_mm256_setzero_pd();
                    for(int k =0; k <= N -4; k +=4)
                    {
                        __m256d a_vec =_mm256_set_pd(A[i +(k+3)*N], A[i +
(k+2)*N], A[i +(k+1)*N], A[i + k*N]);
                        __m256d b_vec =_mm256_loadu_pd(&B[k + j*N]);
                        #ifdef __FMA__ {sum_vec =_mm256_fmadd_pd(a_vec, b_vec,
sum_vec);}
                        #else {sum_vec =_mm256_add_pd(_mm256_mul_pd(a_vec,
b_vec), sum_vec);}
                        #endif
                    }
                    double sum = sum_vec[0]+ sum_vec[1]+ sum_vec[2]+ sum_vec[3];
                    for(int k =(N /4)*4; k < N; k++)
                    {
                        sum += A[i + k*N]* B[k + j*N];
                    }
                    C[i + j*N]= sum;
                }
            }
        }
    }
    double run_time =omp_get_wtime()- start_time;
    return run_time;
}
```

OpenML的其它语法

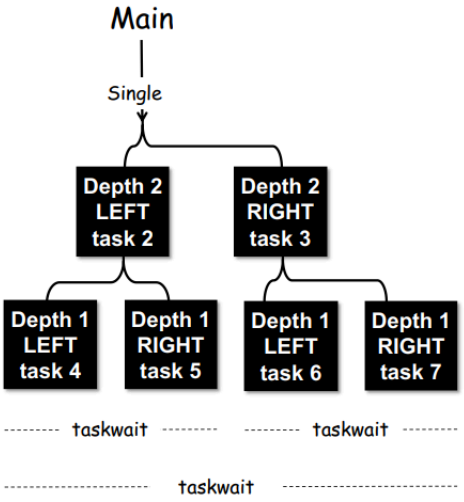
- `#pragma omp single` 接下来的语法块由随机一条线程执行
- `#pragma omp task` 产生一个任务，并自动分配给空闲的线程运行
- `#pragma omp taskwait` 线程等待，直到之前产生的所有任务全部完成
- 例:

```
#include<stdio.h>
#include<omp.h>
int main(){
    #pragma omp parallel num_threads(4)
    {
        #pragma omp single      // Single thread creating tasks
        {
            printf("Parent thread (creating tasks):%d\n", omp_get_thread_num());
```

```
            // Create multiple tasks
            for(int i=0; i<8; i++)
            {
                #pragma omp task
                {
                printf("Task %d executed by thread:%d\n", i,
omp_get_thread_num());
                }
            }
            #pragma omp taskwait
            printf("All tasks completed\n");
        }
    }
    return 0;
}
```

OpenMP Other Usage: Tree Example

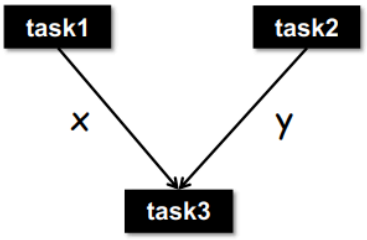
```
#include <stdio.h>
#include <omp.h>
void process_tree(int depth, int id) {
   if (depth <= 0) return;
   #pragma omp task
   {
      printf("Creating LEFT task at depth %d (by thread
%d)\n", depth, omp_get_thread_num());
      process_tree(depth - 1, id * 2);
   }
   #pragma omp task
   {
      printf("Creating RIGHT task at depth %d (by thread
%d)\n", depth, omp_get_thread_num());
      process_tree(depth - 1, id * 2 + 1);
   }
   #pragma omp taskwait
   printf("Completed node %d at depth %d (thread %d)\n",
id, depth, omp_get_thread_num());
}
int main() {
    #pragma omp parallel
    #pragma omp single
   process_tree(2, 1);
   return 0;}
```



- `#pragma omp task depend(dependency type: var list)` 声明依赖关系，如下
  - `in` 当前任务需要读取这些数据，等待这些数据之前的写操作全部完成
  - `out` 当前任务需要写入这些数据，等待这些数据之前的读/写操作全部完成；后续读取任务需要等待当前任务写入结束
  - `inout` 当前任务需要读取+写入这些数据，等待这些数据之前的读/写操作全部完成；后续读取/写入任务需要等待当前任务写入结束（按序处理）
  - `mutexinoutset` 不保证执行顺序，但是保证同时只能有最多一个任务正在处理该变量

OpenMP: Dependency Example

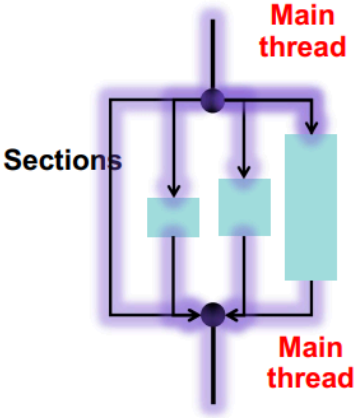
```c
#include <stdio.h>
#include <omp.h>

int main() {
   int x = 0, y = 0;

    #pragma omp parallel
   #pragma omp single
   {
      // task1: write x
        #pragma omp task depend(out: x)
      {
         printf("Task 1: writing x = 10\n");
         x = 10;
      }
      // task2: write y (no dependency with x, execute in parallel)
        #pragma omp task depend(out: y)
        {
         printf("Task 2: writing y = 20\n");
         y = 20;
      }
      // task3: read x & y, depending on tasks 1&2
        #pragma omp task depend(in: x, y)
      {
         printf("Task 3: x + y = %d\n", x + y);
      }
   }
   return 0;
}
```



- `#pragma omp sections` 按照线程编号，静态分配任务段执行

OpenMP Sections

```c
#pragma omp parallel
{
   #pragma omp sections
   {
      #pragma omp section
      {
         // Code block 1
      }
      #pragma omp section
      {
         // Code block 2
      }
      #pragma omp section
      {
         // Code block 3
      }
   } // Implicit barrier
```



数据竞争 Data Race

- 不同线程在同一位置并行执行，至少一个线程包含写操作，共享内存导致运行结果不确定
- 为了解决数据竞争 Data Race 问题，我们需要对数据进行同步

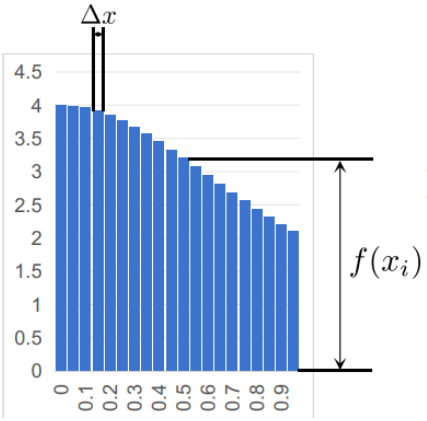
例：积分累加

- 原始版本：

Example: Calculating π

- Serial version

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <stdio.h>
void main(){
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum = 0.0;
  for (int i=0; i<num_steps; i++){
     double x = (i+0.5)*step;
     sum += 4.0*step/(1.0+x*x);
  }
  printf("pi=%6.12f\n",sum);
}
```

pi=3.141800986893.

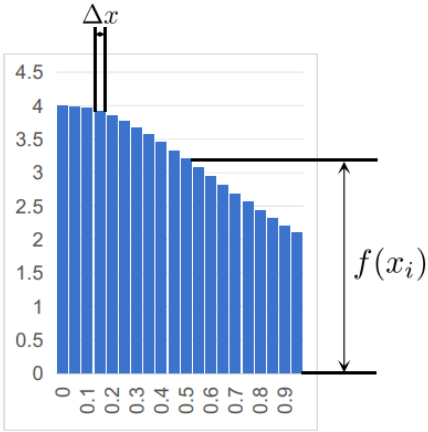
- Resembles π, but not very accurate
- Let's increase **num_steps** and parallelize

- 错误（Data Race）示例：

Example: Calculating π

- OpenMP version 1

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <stdio.h>
#include <omp.h>
void main(){
  const long num_steps = 20;
  double step = 1.0/((double)num_steps);
  double sum = 0.0;
  #pragma omp parallel for
  for (int i=0; i<num_steps; i++){
     double x = (i+0.5)*step;
     sum += 4.0*step/(1.0+x*x);
  }
  printf("pi=%6.12f\n",sum);
}
```

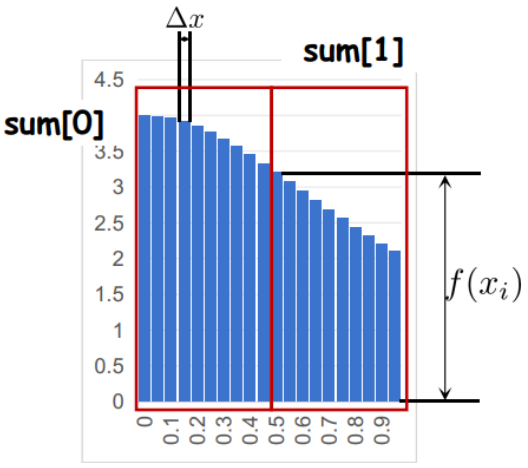
- Problem: each thread needs access to the shared variable **sum**
- Code runs sequentially …

- 正确示例：分治

Example: Calculating π

- OpenMP version 2

$$\sum f(x_i)\Delta x \approx \pi$$




```
#include <stdio.h>
#include <omp.h>
void main(){
 const int NUM_THREADS = 4;
 const long num_steps = 20;
 double step = 1.0/((double)num_steps);
 double sum[NUM_THREADS];
 for (int i=0;i<NUM_THREADS;i++)
  sum[i]=0;
 omp_set_num_threads(NUM_THREADS);
 #pragma omp parallel
 {
   int id = omp_get_thread_num();
  for (int i=id;i<num_steps;i+=NUM_THREADS){
   double x = (i+0.5)*step;
   sum[id] += 4.0*step/(1.0+x*x);
   printf("i=%3d,id=%3d\n",i,id);
  }
 }
  double pi=0;
 for (int i=0;i<NUM_THREADS;i++)
  pi += sum[i];
 printf("pi=%6.12f\n",pi);
}
```

Increase num_step to obtain higher accuracy.

归约操作 Reduction

- 将多个元素合并成同一个元素（累加/累乘等），称为归约 Reduction
- 在OpenMP中，可以使用 `reduction(operation:var)` 声明归约操作，使其操作结果正确
- 例如，在积分累加中：

```
#include <omp.h>
#include <stdio.h>
static long num_steps = 100000;
double step;
void main ()
{
    int i; double x, pi, sum = 0.0;
    step = 1.0 / (double)num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)    // Correct
    for (i=1; i<= num_steps; i++)
    {
        x = (i - 0.5) * step;
        sum = sum + 4.0 / (1.0+x*x);
    }
    pi = sum * step;
    printf("pi = %6.12f\n", pi);
}
```

同步 Synchronization

- `#pragma omp barrier` 强制进程等待，直到所有线程都到达这里
- `#pragma omp critical` 创建一个临界区 critical section，一次只能有最多一个线程进入临界区

- `#pragma omp critical(resourceA)` 创建一个临界区，在拿到 `resourceA` 的锁后允许进入

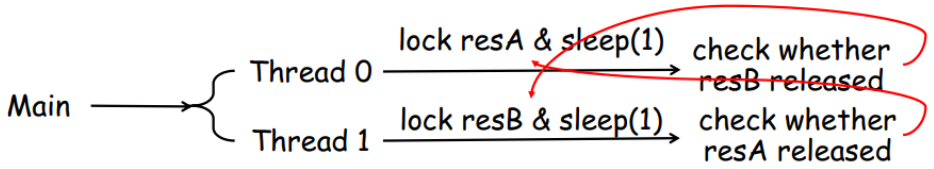
OpenMP Hello World, Synchronized

```
#include <stdio.h>
#include <omp.h>
int main() {
  int x = 0;                    /* shared variable */
  #pragma omp parallel
  {
    int tid = omp_get_thread_num(); /* private variable */
    #pragma omp critical
    {
      x++;
      printf("Hello World from thread = %d, x = %d\n", tid, x);
    }
    #pragma omp barrier
    if (tid == 0) {
      printf("Number of threads = %d\n", omp_get_num_threads());
    }
  }
}
```

死锁 Deadlock

Deadlock Example

```
int main() {
   int shared_resource_A = 0;
   int shared_resource_B = 0;
   #pragma omp parallel num_threads(2)
   {
      int tid = omp_get_thread_num();
      if (tid == 0) {
         // Thread 0: lock A then B
         #pragma omp critical(resA)
         {   printf("Thread 0 locked A\n");
            sleep(1);  // simulate work
            #pragma omp critical(resB)
            {printf("Thread 0 locked B\n");
               shared_resource_B++;}
         }
      }
      else if (tid == 1) {
         // Thread 1: lock B then A  <-- opposite order!
         #pragma omp critical(resB)
         {   printf("Thread 1 locked B\n");
            sleep(1);
            #pragma omp critical(resA)
            {printf("Thread 1 locked A\n");
               shared_resource_A++;}
         }
      }
   }
   printf("Done\n");
   return 0;
}
```



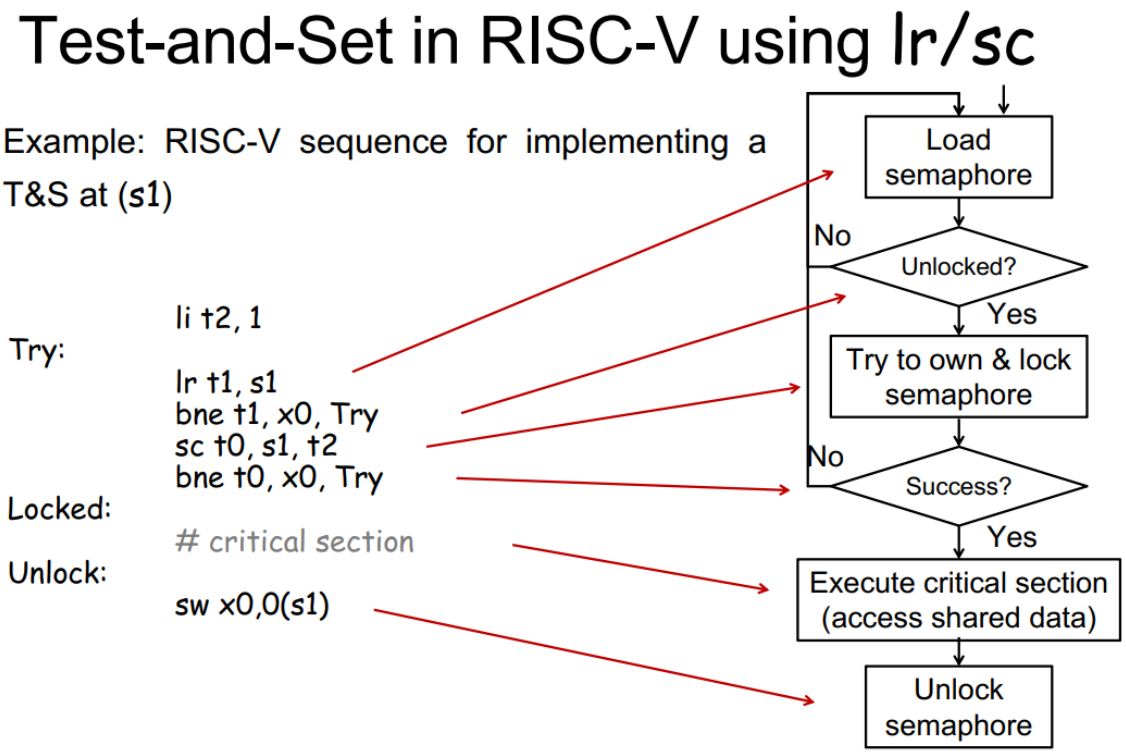
Lec23 线程级并行 Thread-Level Paralellism II

锁同步 Lock Synchronization

- 在进程的共享内存中定义锁：`0` 表示未上锁，`1` 表示上锁
- 基本流程：检查锁并上锁 Acquire -> 临界区操作 -> 释放锁 Release
- 原子操作 Atomic operation：最小的不可分割操作，不会在执行期间被中断
- 原始的 RISC-V 不支持原子操作指令，需要自行添加原子指令（如RV32A，即原子扩展）

解决方案1：读写对

- 一般利用监听总线实现保留标记的失效
- Test-and-Set（属于自旋锁 Spin-lock）
  - 检查内存地址是否被上锁，并建立保留标记，若上锁（`s1 = 1`），则重新检查；若未上锁（`s1 = 0`），则可以尝试获取锁
  - 尝试把 1 写入锁 `s1`（上锁），如果 `s1` 没有发生过变化，则成功写入；否则从上一步开始重新操作
  - 进行临界区操作
  - 把锁 `s1` 置为0，解锁
- 这种做法可以避免 ABA 情况（在多线程环境下，一个变量的值经历了“从 A 变成 B，再从 B 变回 A”的过程，而另一个观察者线程只看到了前后的 A，从而误以为这个变量在这期间没有被改过）



解决方案2：原子内存操作

- 进行原子寄存器交换操作 `AMOSWAP rd, rs2, (rs1)` 将寄存器 `rs1` 的旧值存下来，然后交换 `rs1` 和 `rs2` 的值，最后将旧值写入 `rd`
- 对于 Write-allocate 缓存，可以在缓存块上增加一个 Lock 位，来实现锁操作

```
    li t0, 1
Try:
    amoswap.w.aq t1, t0, (a0)    # 把当前线程的1（上锁）试图换入锁中
    bnez t1, Try                 # 如果交换得到0（解锁）则继续，否则重新交换
Locked:
    # Critical section
Unlock:
    amoswap.w.rl x0, x0, (a0)    # 重新交换，让锁回到0（解锁）
```

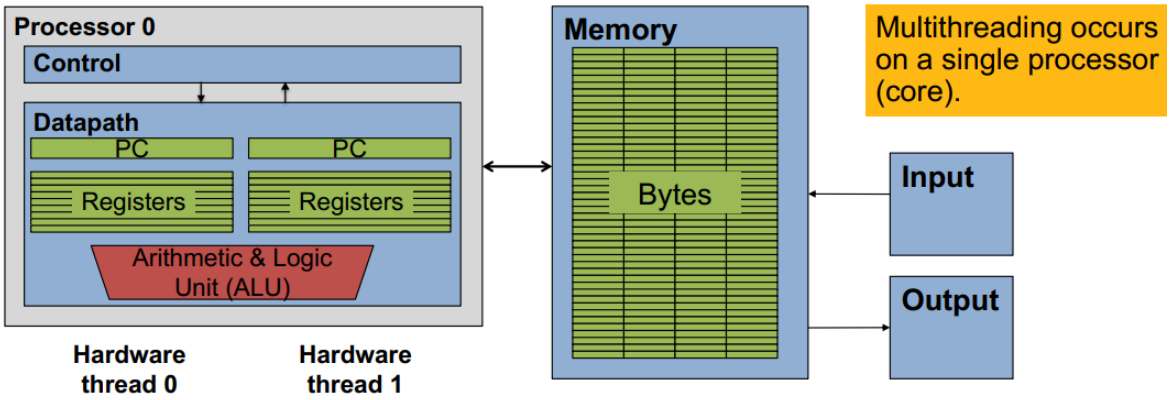
Lec24 硬件多线程 Hardware Multithreading

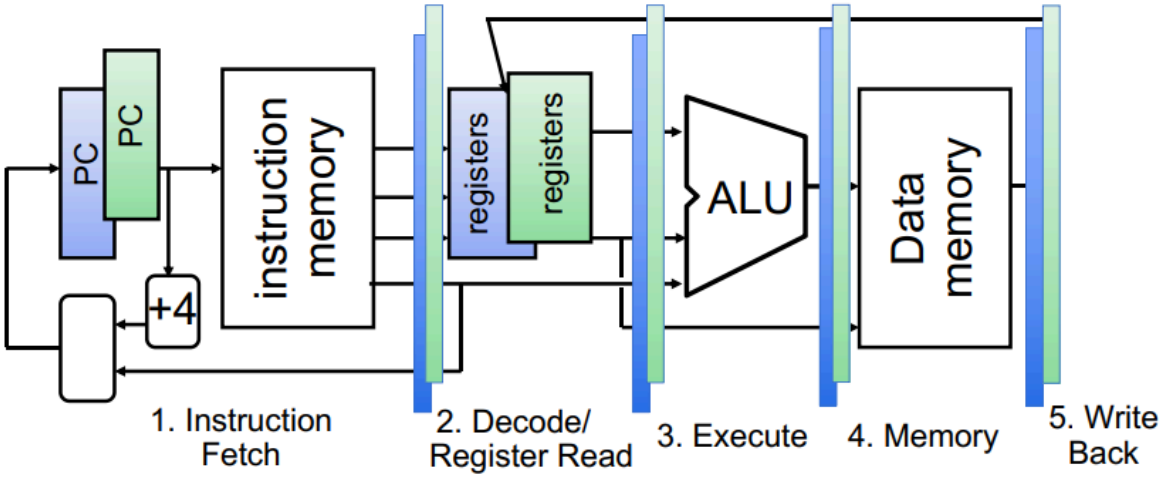
多核处理器 Multi-core Processor

- 一个多核处理器包括多个处理器 processor（核 core）
- 每个处理器执行独立的指令流，各自独有高级缓存（如 L1&L2 cache）
- 所有处理器共享主存 DRAM （和可能的 L3 cache）
- 各处理器通过共享内存来交互，需要设计同步原语
- 多核处理器用于处理单个可并行的任务或者多个任务
- 设计多核处理器内核的数量：取舍性能与能耗
- 每个核心对称访存 uniform memory access, UMA
- 对于较大的集群中，芯片核心较多，需要采用非对称访存 non-uniform memory access, NUMA：不同核心访问不同内存时有不同的延迟

硬件多线程 Hardware Multithreading

- 多个线程在同一个处理器中同时执行
- 逻辑上可以看作是“多核”，但物理硬件上还是单核

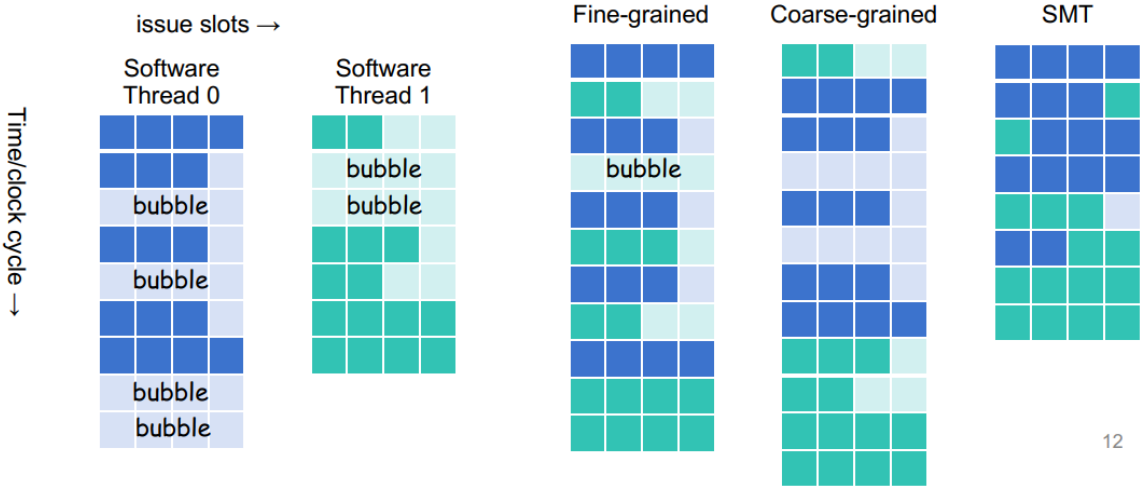




- 多个线程在同一个内核上交替运行，当一个线程处于等待状态（如正在访存）时，另一个线程继续执行；通过选择器进行数据的物理隔离

同步多线程 Simultaneous Multithreading, SMT

- 可以采用细粒度 Fine-grained，粗粒度 Coarse-grained，或者同步多线程 Simultianeour Multithreading, SMT（也叫超线程 hyperthreading）来实现硬件多线程



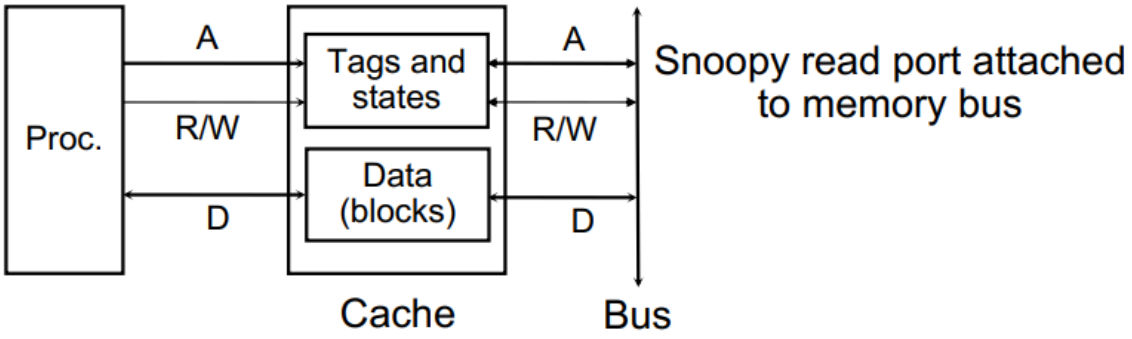
- 相比于硬件多核，硬件多线程对性能提升的性价比更高，但是设计更复杂

高级缓存 Advanced Cache

- 即使是单处理器，访存也是性能瓶颈
- 可以采用进程私有缓存 private caches 来减少访存带宽需求

缓存不一致 Cache incoherence

- 多个私有缓存发生写入时，会发生缓存不一致 Cache incoherence 的现象，即产生了新的缓存缺失类型：一致性缺失 coherence miss（也叫通信缺失 communication miss）
- 可以采用监听 Snooping 的方式来解决一致性缺失：每个线程在总线上监听写入信号，当有线程发生写入时，其它线程对应更新
- 假共享问题 False sharing：不同线程同时存储了一个 Cache block，其中一个修改了某一位，而另一个只用到其它 offset 的缓存也需要发生更新



写失效 Write invalidate

- 其它线程监听到总线上的地址时，将对应地址（如有）的缓存块设为 invalid

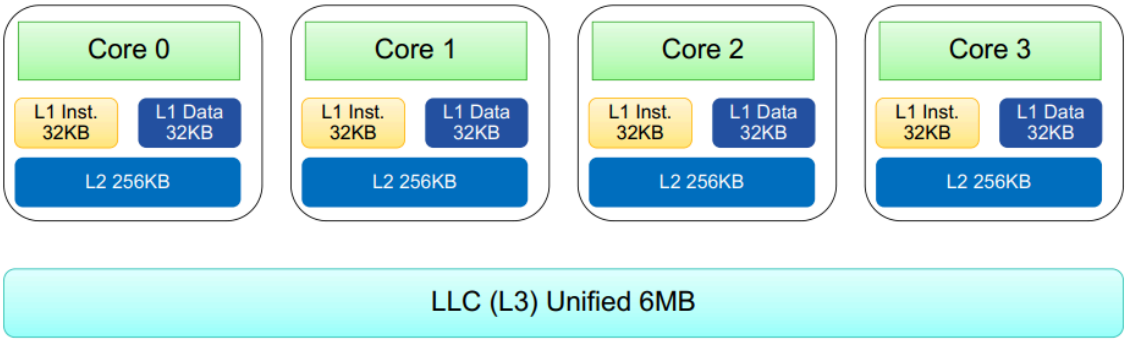
写更新 Write update

- 线程广播数据更改时，同时广播新的数据；其它线程监听到总线上的数据和地址时，更新（如有）私有缓存内部的数据

M(O)ESI 协议 M(O)ESI Protocols

- 对于每个缓存块，跟踪它的状态：
- 共享 Shared：最新的数据，其它缓存可能同时拥有
- 已修改 Modified：最新的数据，已经被修改过（dirty），其它缓存没有这个地址
- 非法 Invalid：不在缓存中，必须重新获取
- 排他 Exclusive：（可选状态）最新的数据，其它缓存没有这个地址
- 所有者 Owner：（可选状态）最新的数据，其它缓存可能同时拥有且必须处于 Shared 状态，是唯一需要更新内存的线程

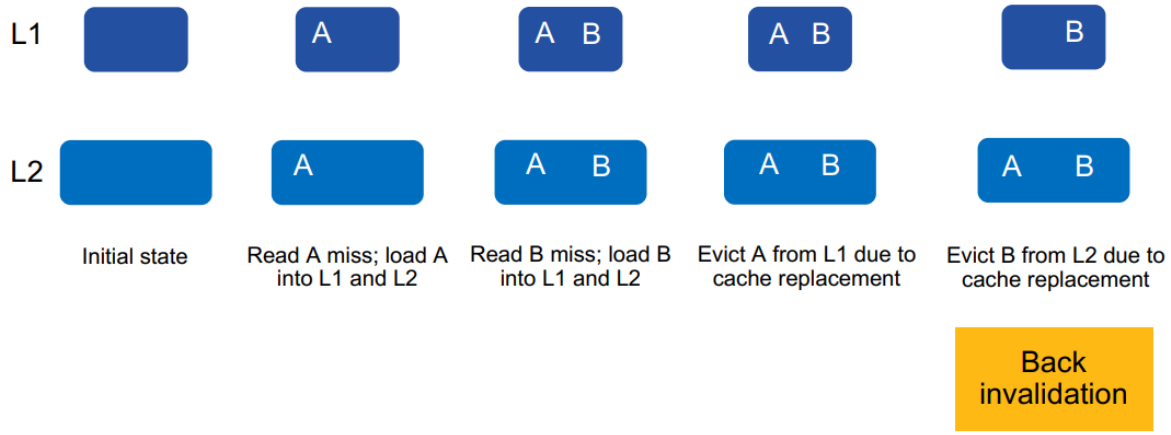
多级缓存



Intel Ivy Bridge Cache Architecture (Core i5-3470)

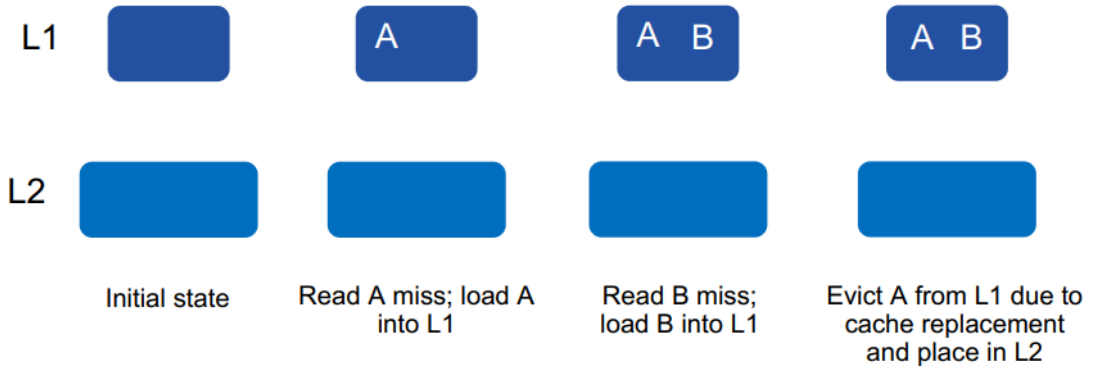
- 包含型 Inclusiveness：高级缓存的内容在低级缓存中存在备份

$$L_n \subsetneq L_{n+1} \ (n \geq 1)$$

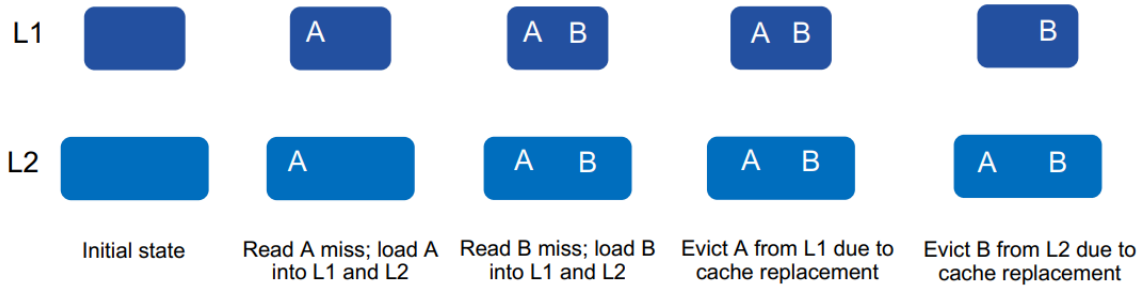


- 排除型 Exclusiveness：高级缓存内容和低级缓存内容不重叠

$$L_n \cap L_{n+1} = \emptyset \ (n \geq 1)$$



- 非包含型 Non-inclusive：不规定是否包含



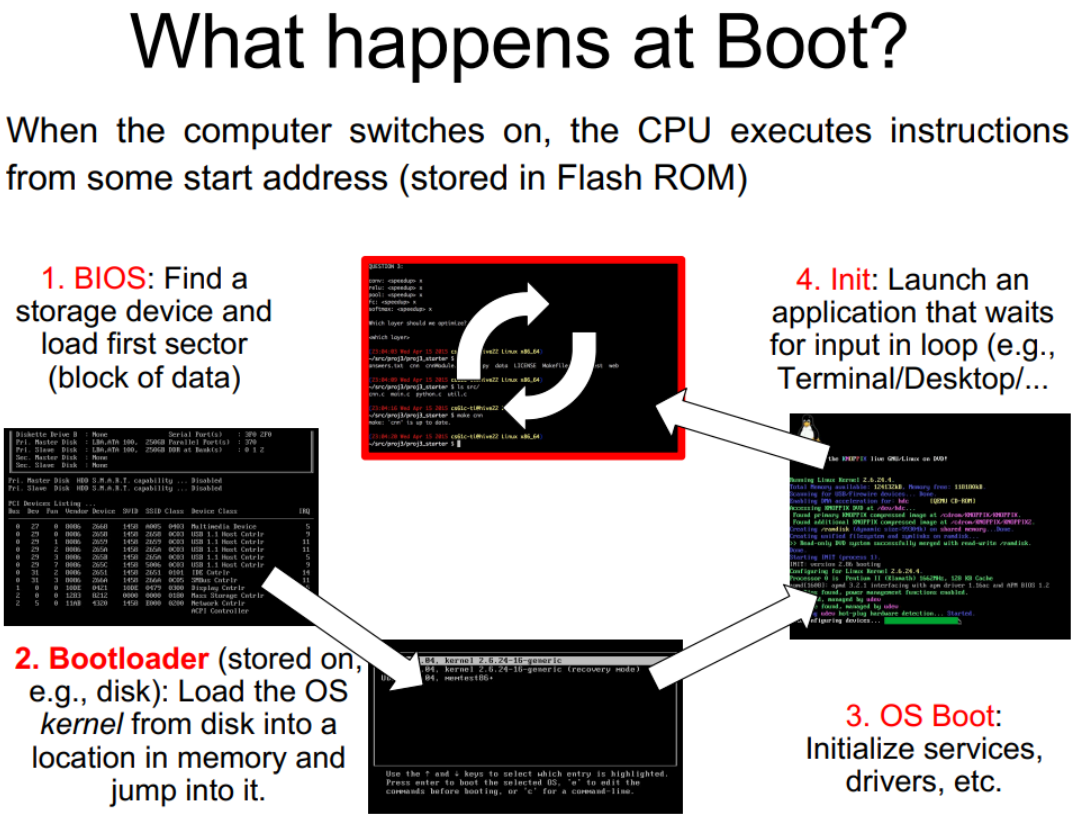
Lec25 操作系统与输入输出 Operating System & I/O

操作系统 Operating System, OS

- 操作系统大约地定义为内核 kernel 大小，不包含各种外设及交互界面
- 操作系统是计算机启动时最早运行的几个程序之一

启动流程 Boot

- 当电源开启时，CPU完成复位，并让 PC 置为某个默认值（如 `0x2000`），启动基本输入输出系统 Basic I/O System, BIOS
- 加载内核进入内存，并跳转至内核代码入口
- 操作系统启动



- 现代操作系统的 BIOS 已经进化到统一可扩展硬件接口 Unified Extensible Firmware Interface, UEFI，更加复杂且强大，支持图形化界面、网络等

输入输出，中断与异常 I/O, Interrupt & Exception

- 输入输出设备的控制，可以抽象为控制信号和数据信号
- 操作系统提供了标准化的接口，实现对外设的控制输入输出
- 这些信号的输入输出一般由内存映射输入输出 Memory Mapped I/O, MMIO 来实现（输入输出到外部控制器的寄存器上，然后寄存器上的数据实时映射到内存上）
- 但是，输入输出设备一般速度慢于CPU，因此采用轮询 Polling 或中断 Interrupt 来实现

轮询 Polling

- 每固定时间从控制寄存器中读取数据

Cost of Polling

- Assume for a processor with a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling
  - Mouse: polled 30 times/sec so as not to miss user movement
- Mouse Polling [clocks/sec]

  = 30 [polls/s] * 400 [clocks/poll] = 12K [clocks/s]
- % Processor for polling:

  $12*10^3$ [clocks/s] / $1*10^9$ [clocks/s] = 0.0012%

  => Polling mouse little impact on processor

中断 Interrupt

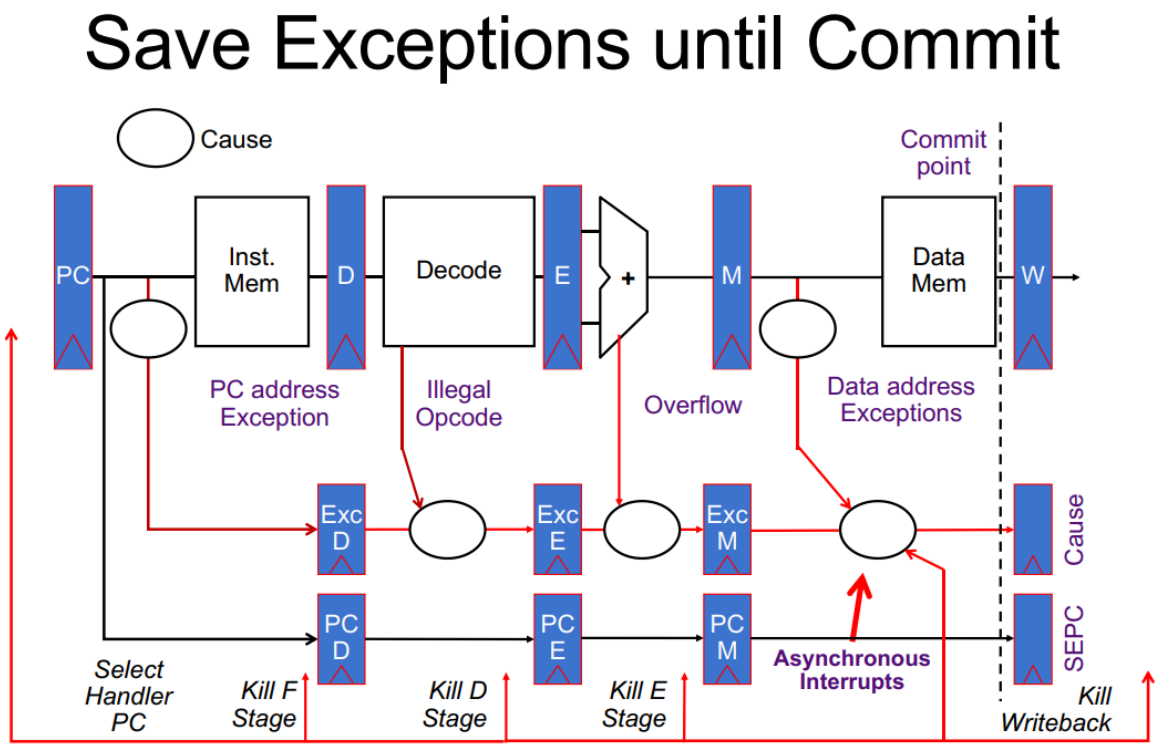
- 当发生输入输出事件时，信号传递到CPU内部的硬件中断控制器 Interrupt Controller 中断CPU来处理事件
- 中断发生时，中断信号传入CPU，CPU通过访问存在独立的寄存器上中断向量表的地址，查询中断向量表 Interrupt Vector Table 来找到对应的中断处理程序 interrupt handler 地址

In CA (you'll see other definitions in use elsewhere):

- Interrupt – caused by an event *external* to current running program (e.g. key press, mouse activity)
  - Asynchronous to current program, can handle interrupt on any convenient instruction
- Exception – caused by some event during execution of one instruction of current running program (e.g., page fault, bus error, illegal instruction)
  - Synchronous, must handle exception on instruction that causes exception
- Trap – action of servicing interrupt or exception by hardware jump to "trap handler" code

陷阱 Trap

- 异常可能在任意时间发生（同步 Synchronous），需要按照时间顺序，用陷阱 Trap 处理



- 指令异常发生时，异常被标记，在当前引起指令异常的指令到达提交点后报错（写入过程不会被执行），然后将后续指令从流水线中清除

Lec26 虚拟内存 Virtual Memory

虚拟内存简介

设计目的

- 需要将磁盘 disk 加入多级存储的架构
- 需要提供进程之间的保护
- 每个进程需要访问从 0 开始的连续空间

早期设计

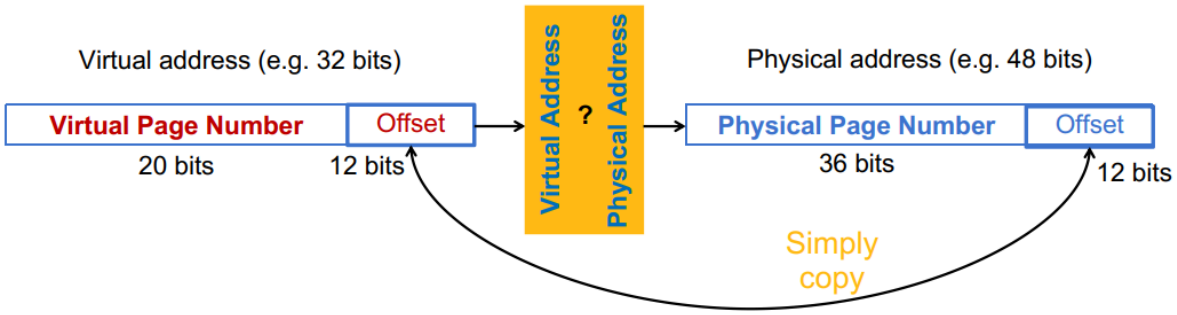
- 设计一个基地址寄存器 base register 和一个界限存储器 bound register
- 一个程序只能访问 `[base, base + bound]` 范围内的内存地址
- 但是这样直接分配物理内存的方法，会导致内存碎片化 Memory Fragmentation

内存分页 Paged Memory

- 将物理内存、虚拟内存分为若干页 Page
- 每页 Page 的大小一般为 `4 KiB`，在页内需要 `12 bit` 地址来表达数据的偏移量 offset

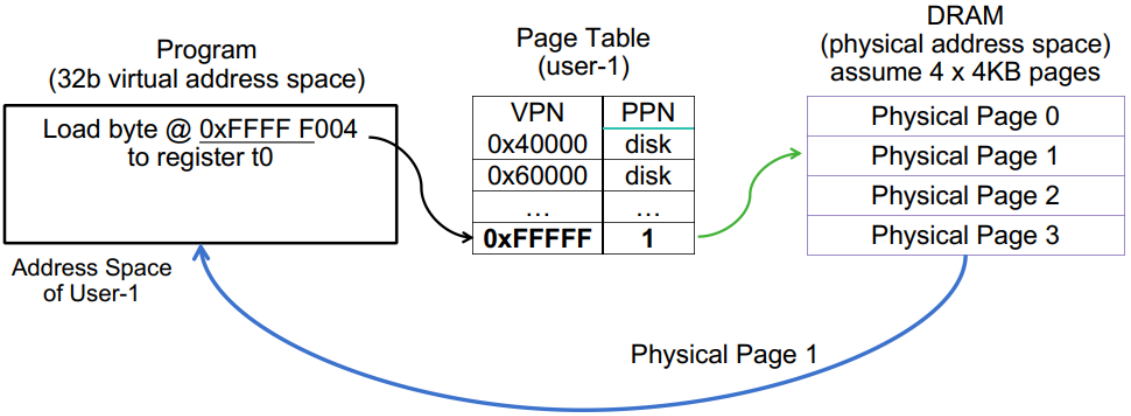
地址翻译 Address Translation

- 一个虚拟页 Virtual Page 对应一个物理页 Physical Page
- 对于一个32位计算机，内存地址的低12位为偏移量 Offset，高20位为虚拟页编号
- 对于一个有256TB内存空间的计算机，物理页编号为高36位



页表

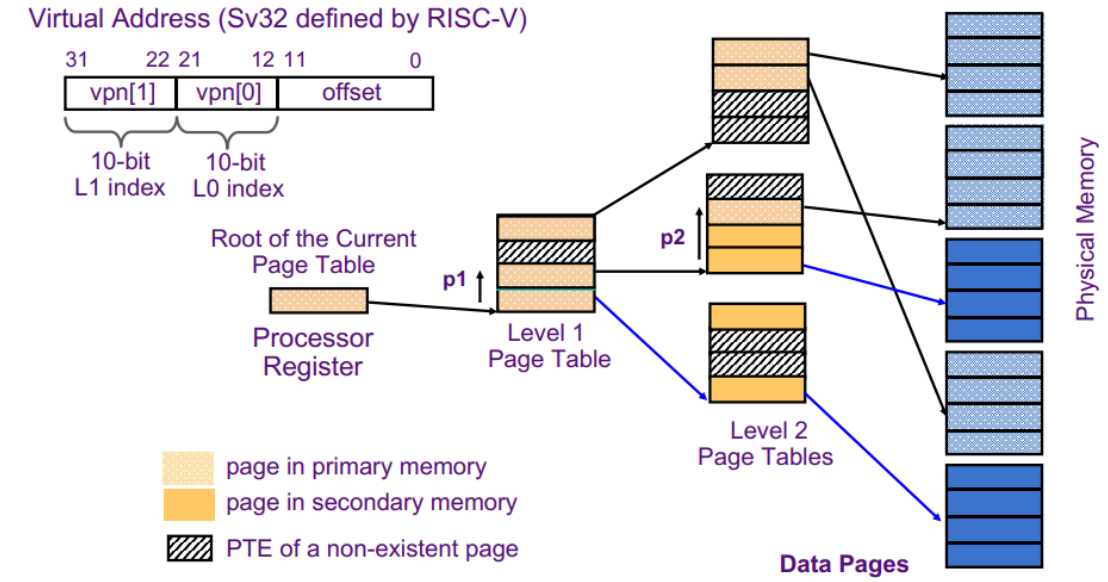
- 使用一个页表 Page table 来记录每页的物理基地址，页表存储在内存上
- 内存管理单元 Memory Management Unit, MMU 从虚拟地址中提取出虚拟页号，通过查页表来找到对应的物理页号，转换为物理地址，访问内存，获取数据
- 如果页表中记录了该页的物理地址，说明该页在内存上，直接访问



- 如果页表中记录该页在磁盘中，访问会触发一个缺页中断 page fault，从硬盘上重新获取数据存入内存，并更新页表后，再次访问内存获取页
- 页表存在保护位，记录数据的状态、权限等；存在状态位，表示是否在内存上
- 所有的虚拟页号 Virtual Page Number, VPN 在页表中都有一条记录
- 当页表满时，可以采用 LRU / Random 等策略进行交换
- 页表总是采取写回策略 Write-back，维护脏位 Dirty Bits，仅在页被换出时写回磁盘
- 每个处理器内核有独立的监管者地址转换和保护寄存器 satp(supervisor address translation protection) register，负责存储当前进程页表的起始物理地址
- 在多任务系统中，每个进程都有独立的页表

多级页表 Hierarchical (Multi-level) Page Table

- 为了防止页表占用空间过大，采用多级页表，但是访存次数会增加



More at: https://docs.riscv.org/reference/isa/priv/supervisor.html#sv32

- 将内存地址的虚拟地址部分分为若干段，每一段对应一级页表
- 每一级页表查表得到下一级页表的基地址

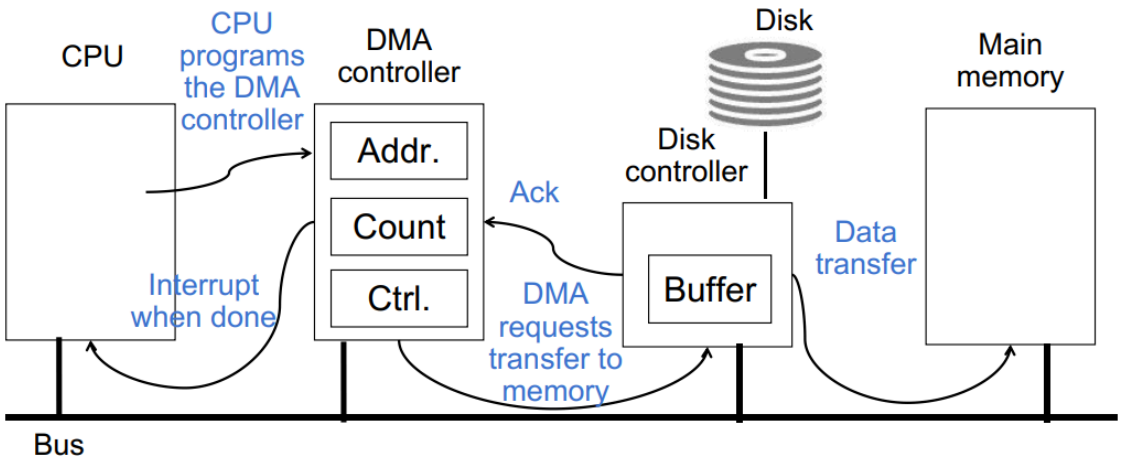
快表 Translation Lookaside Buffer, TLB

- 对于较为常用的页表项，快表中会存储一份，以做到更快的访问速度
- 快表是页表的缓存，不存储数据，只存储虚拟页号-物理页号映射关系
- 一般能缓存 32 - 128 个页表项
- 有独立的替换策略、全相联/组相联，甚至多级快表
- TLB 覆盖范围 TLB Reach = 快表项数量 * 页大小

Lec27 More I/O

直接内存访问 Direct Memory Access, DMA

- CPU 直接负责搬运数据时，会浪费 CPU 资源，因此引入硬件 DMA 控制器 DMA engine
- DMA engine 接收由 CPU 写入的寄存器，包含以下内容：
  - 需要访问的内存地址
  - 需要读取的比特数量
  - 输入输出设备（从哪里到哪里）
  - 每次 burst 需要传输的数据量



- 在 CPU 控制 DMA engine 从磁盘中读取数据到内存后，CPU可以在数据传输时间内完成其它工作；当传输完成时，DMA 会产生一个中断 interrupt，使 CPU 处理传输完成的数据

问题1：DMA设计在哪里

- 如果 DMA 接入在 CPU 和 L1 缓存之间，CPU永远可以读到最新的数据，但是会造成缓存抖动 thrashing（DMA 搬运的数据会把 CPU 正在使用的数据挤出缓存），导致 CPU 运行变慢
- 如果 DMA 接入在末级缓存和内存之间，DMA 和 CPU 将互不干涉，但是可能造成数据不一致，需要额外的机制来保证数据一致性（现代常用做法）

问题2：总线仲裁 Bus Arbitration

- 采用突发模式 Burst Mode：DMA 一旦占用总线，会一直占用直到传输完成：有最快的传输效率，但是可能造成 CPU 饥饿
- 采用周期窃取模式 Cycle Stealing Mode：DMA 每传输一个字节，就释放总线让 CPU 运行一段时间：CPU 可以正常运行，但是传输速率受影响
- 采用透明模式 Transparent Mode：DMA 仅在 CPU 未占用总线时运输数据：CPU 完全不受影响，但传输效率最低

常用 I/O 设备：固态硬盘 SSD (闪存 Flash Memory)

- 技术原理：基于闪存 Flash Memory 技术，使用带有额外导体的 NMOS 晶体管来"捕获"电子。电子的存在或缺失代表逻辑 `1` 或 `0`
- 非易失性：掉电后仍能保留数据
- 磨损均衡 Wear Leveling：闪存单元的擦写次数 program-erase cycles 有限
- 控制器通过该技术将写操作均匀分布在所有块上，以延长寿命
- 优点：低功耗、无机械运动部件（抗震）、启动速度快

HDD vs. SSD 对比

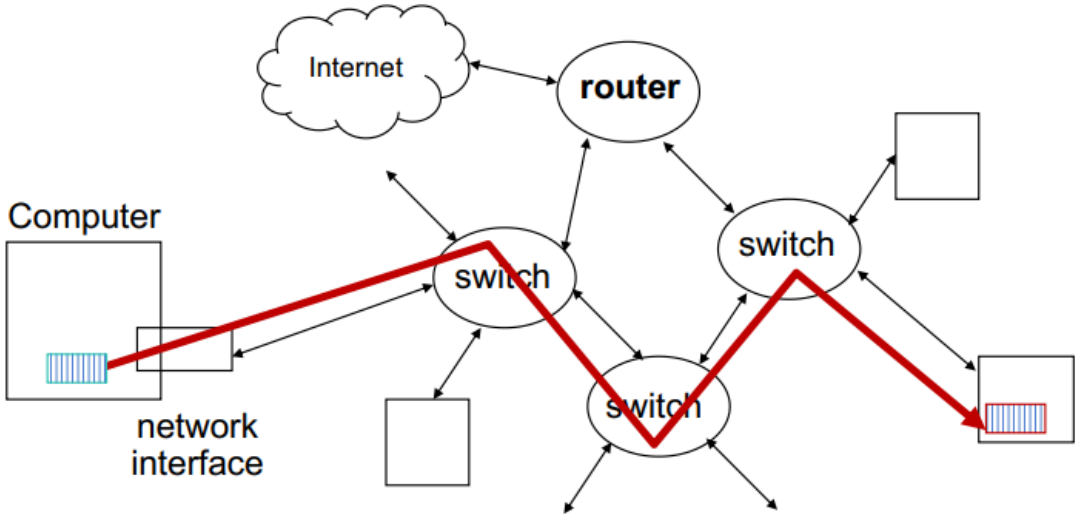
特性	机械硬盘 (HDD)	固态硬盘 (SSD)
每比特成本	更便宜	

特性	机械硬盘 (HDD)	固态硬盘 (SSD)
容量	更大	
耐用性		更耐用 (防震)
性能		更快
功耗		更低
体积		更紧凑 compact
寿命 Endurance	更好	

网络 Network

共享式/交换式网络 Shared / Switched-Based Networks

- 共享式网络 Shared Networks：所有节点 Node 都连接在一根总线上，每次只能同时进行一组连接
- 交换式网络 Switched Networks：引入一个交换机 Crossbar Switch，支持同时的任意点对点连接（现代的交换机）



- 核心功能：
  - 命名 Naming：能够识别组件。
  - 路由 Routing：将信息包 Packets 从源移动到目的地。
- 设计思想： 采用分层 Layering、冗余Redundancy、协议Protocols 和封装 Encapsulation 来实现复杂系统的抽象

软件协议发送与接收步骤

- 发送端 SW Send：
  1. 应用程序将数据拷贝到 OS 缓冲区
  2. OS 计算校验和 Checksum ， 并启动定时器
  3. OS 将数据发送到网络接口硬件 NIC 并启动传输
- 接收端 SW Receive：

1. OS 从网络接口硬件拷贝数据到 OS 缓冲区
2. OS 计算校验和：若 OK，发送 ACK 确认；若出错，删除消息（发送端定时器超时后会重发）
3. 若数据正确，OS 将数据拷贝到用户地址空间，并通知应用继续

数据包结构 (Packets of data)

- 头部 (Header)：包含目标 ID Dest、源 ID Src、长度 Len 和控制信息 ACK INFO
- 负载 (Payload)：实际传输的数据 CMD/Address/Data
- 尾部 (Trailer)：包含校验和 Checksum，用于错误检测

分层协议体系 (Hierarchy of Layers)

为了应对通信的复杂性，网络被划分为不同的层级，每一层为上一层提供服务：

1. 应用层 (Application)：如 HTTP (WWW), FTP (文件传输), SMTP (邮件)
2. 传输层 (Transport)：处理端到端通信 TCP（保证交付）或 UDP（视频/音频流，不保证可靠性）
3. 网络层 (Network)：负责路由。如 IP 协议
4. 数据链路层 (Data Link)：如以太网 Ethernet
5. 物理层 (Physical Layer)：铜线、无线电波、光纤等

封装与协议族概念 (Protocol Family & Encapsulation)

- 封装 Encapsulation：每一层将上一层的信息包作为本层的“负载”，并加上自己的头部 Header 和尾部 Trailer，就像给信件套上多层信封
- 逻辑通信 vs 物理通信：
  - 逻辑上：每一层与其对等层 Peers 直接对话
  - 物理上：数据必须逐层向下经过物理链路发送，再在接收端逐层拆封向上递交

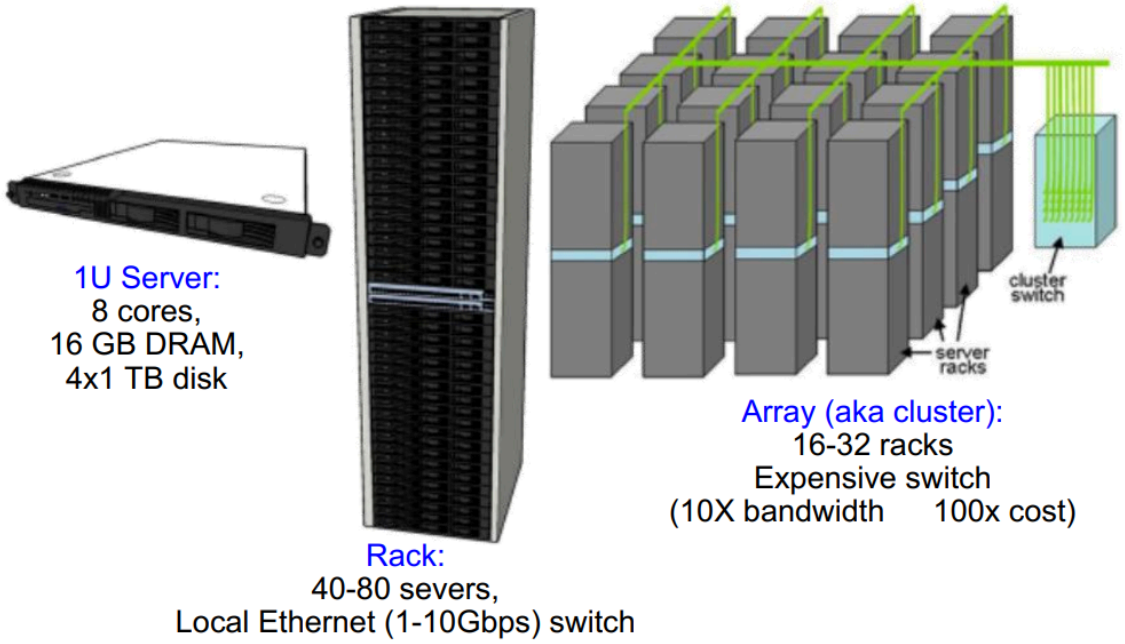
TCP/IP 协议栈

- Internet Protocol, IP：“尽力而为”的交付，数据包可能会丢失或损坏
- Transmission Control Protocol, TCP：在 IP 之上运行，保证数据可靠交付
- 应用：即使是局域网 LAN 通信，也广泛使用 TCP/IP

Lec28 仓储型计算 Warehouse Scale Computing, WSC

仓储型计算 Warehouse Scale Computing, WSC

- 是一种请求级并行 Request-level Parallelism
- 在 Google 的仓库里，一般建立在河边，主要解决散热问题
- 服务器 Server：刀片式服务器，一层，和正常计算机类似
- 机柜 Rack：一摞服务器，一个柜子的服务器
- 阵列 Array：把多个机柜的服务器通过网线/光纤连接起来，一个机房/集装箱



- 开放计算项目 Open Compute Project：多个互联网巨头参与的工程，保证不同服务器厂家的服务器兼容性
- 仓储型计算机 Warehouse-Scale Computers：数据中心 Datacenter，单个大型计算机 Single gigantic machine 等

特性

- 组织结构不同：不同服务器不共用总线，而是使用交换机连接
- 请求级并行 Request-level Parallelism, RLP：很多用户同时发起互不相干的请求
- 数据级并行 Data-level Parallelism, DLP：多用户同时对同一个服务的不同数据发起请求（一般 WSC 上的数据级并行不叫 SIMD）
- 硬件更新周期长（10年左右换一批计算机），电力开销大
- 较高的延迟、带宽
- 高出错率 Failure rate
- 不同时段工作压力不同

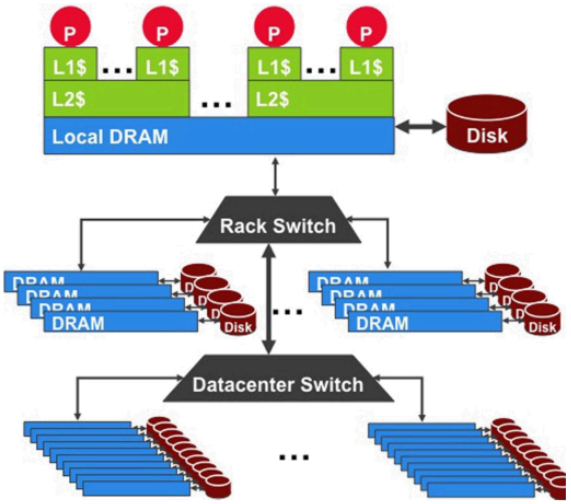
存储架构 Storage Hierarchy

- 单台服务器访问其它服务器的内存 DRAM 的延迟比访问本地磁盘的延迟更低
- 单台服务器访问其它服务器的内存 DRAM 的带宽低于访问本地磁盘

1U Server:
DRAM: 16GB, 0.1us, 20GB/s
Disk: 2TB, 10^4us, 200MB/s

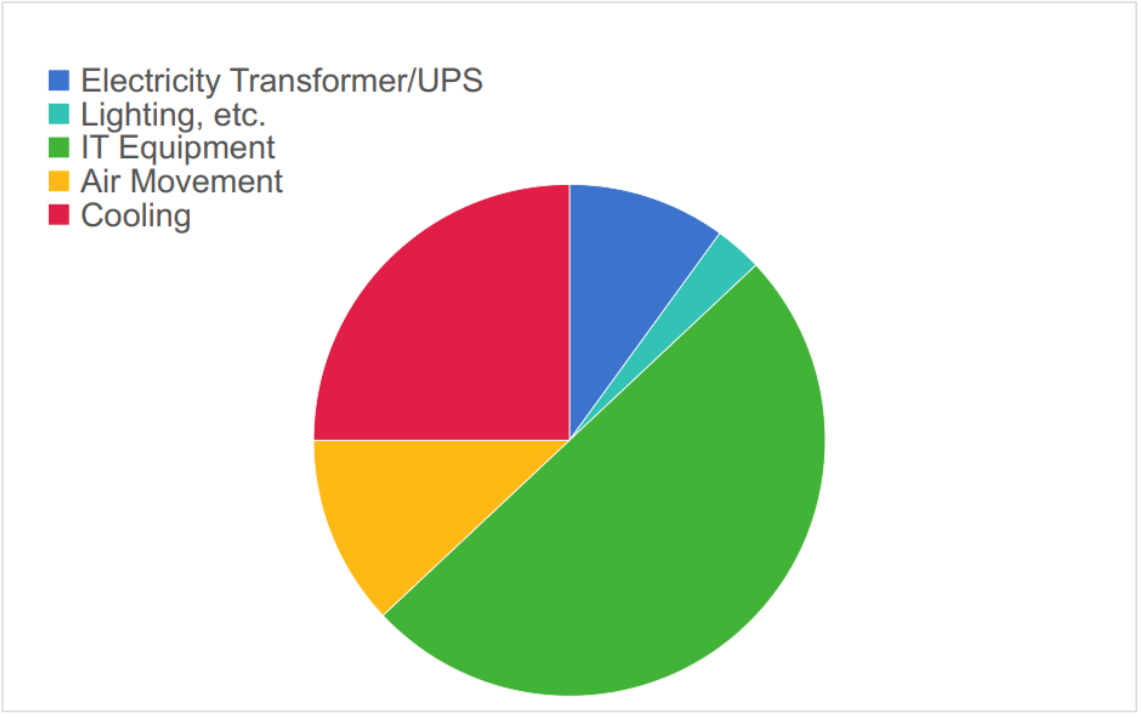
Rack(80 severs):
DRAM: 1TB, 300us, 100MB/s
Disk: 160TB, 11ms, 100MB/s

Array(30 racks):
DRAM: 30TB, 500us, 10MB/s
Disk: 4.80PB, 12ms, 10MB/s



能量使用效率 Power Usage Effectiveness, PUE

- $PUE = \frac{Total\_Building\_Power}{IT\_Equipment\_Power}$



请求级并行 Request-Level Parallelism, RLP

- 同一时刻有大量请求，其中请求之间数据依赖较小
- 请求大部分都是读请求
- 例如：网络搜索

- Search “Ne Zha 2”
  - Direct request to “closest” WSC;
  - Front-end load balancer directs request to one of many arrays (cluster of servers) within WSC;
  - Within array, select one of many Google Web Servers (e.g. GWS) to handle the request and compose the response pages;
  - GWS communicates with Index Servers to find documents that contains the search word, “Ne Zha 2”;
  - Return document list with associated relevance score;

- In parallel, Ad system: run ad auction for bidders on search terms
- Use docids (Document IDs) to access indexed documents
- Compose the page
  - Result document extracts (with keyword in context) ordered by relevance score
  - Sponsored links (along the top) and advertisements (along the sides)

MapReduce

- 提高并行系统的利用率的框架
- 映射过程 Map：可同步执行的部分
- 归约过程 Reduce：异步执行的部分

例：计算 $\sum_{i=1}^{4} n^2$

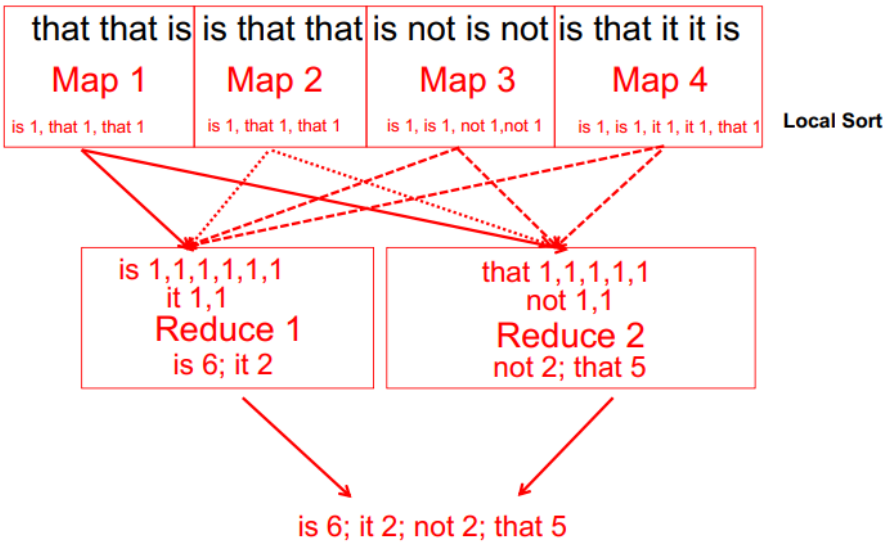
```
from functools import reduce
A = [1,2,3,4]
def square(x):
    return x * x
def sum(x, y):
    return x + y
reduce(sum, map(square, A))
```

例：计算 1~4 内奇数的平方和

对每个数，返回一个键值对 `num -> (key, value)`：`1 -> (1,1)`，`2 -> (0,4)`，... 对于不同的键 key，在计算后分配到不同的服务器进行汇总运算

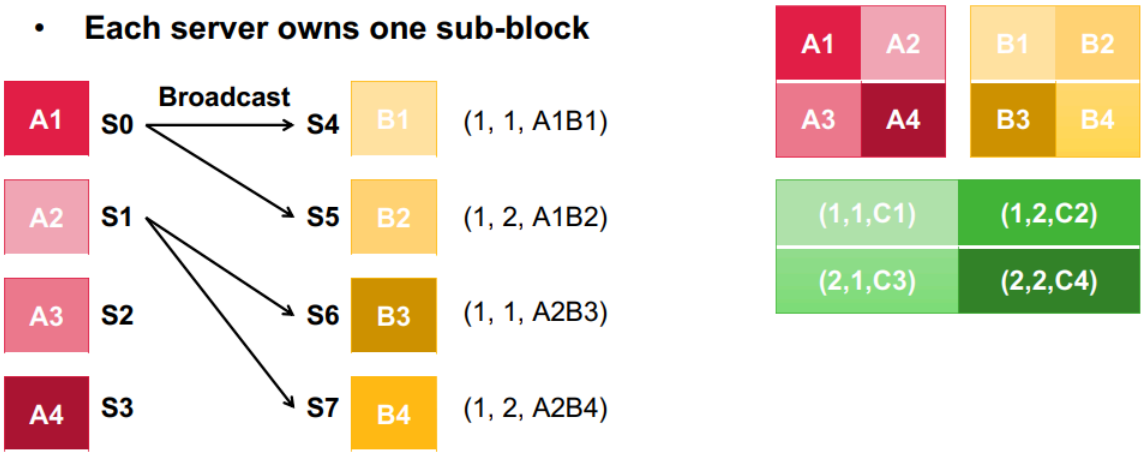
例：字数统计

- **Distribute**
- **Shuffle**
- **Collect**

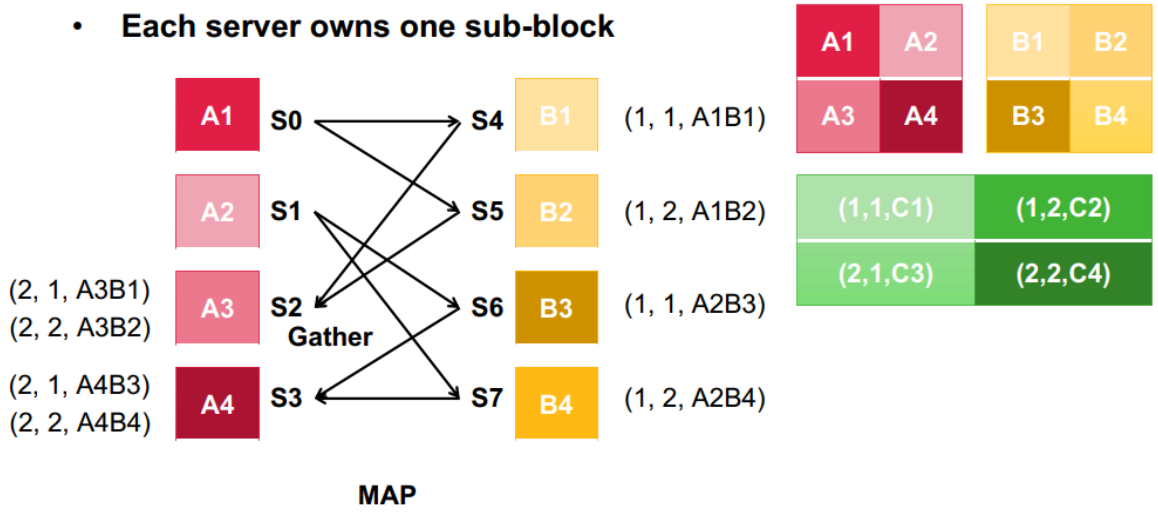


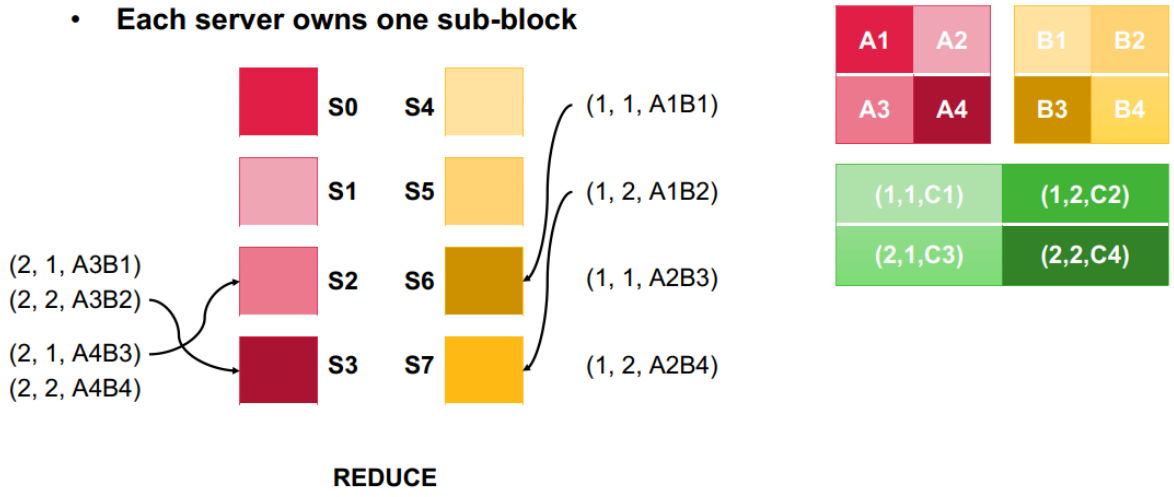
例：矩阵乘法

- **Each server owns one sub-block**



- **Each server owns one sub-block**





- All reduce：把最终的所有数据（如例子中的4个子矩阵）汇总到同一台服务器上

Lec29 容错 Fault-tolerance

可靠性 Dependability

- 空间冗余性 Spatial Redundancy：重复存储数据、信息或使用重复的硬件来预防软错误、硬错误
- 时间冗余性 Temporal Redundancy：多次运行代码，保证结果正确性
- 可靠性由系统的"短板"决定（木桶效应，阿喀琉斯之踵 Achilles' Heel）
- 存储设备的故障率相对较高

可靠性的衡量标准 Dependability Measures

- 修理后下次出错期望时间 Mean Time To Failure, MTTF
- 期望修复时间 Mean Time To Repair, MTTR
- 出错间隔期望时间 Mean Time Between Failures, MTBF；`MTBF = MTTF + MTTR`
- 可用性 `Availability = MTTF / MTBF`
- 我们期望提升系统的可用性 Availability
- 使用 9 的数量来描述可用性：
  - 1 nine：$\geq 90\%$ Availability
  - 2 nines：$\geq 99\%$ Availability
  - 3 nines: $\geq 99.9\%$ Availability
  - ...
- 年故障率 Annualized Failure Rate, AFR：每年硬件故障的概率；`AFR = 8760 hour / 寿命`

错误检测/纠正码 Error Dectection / Correction Codes, ECC

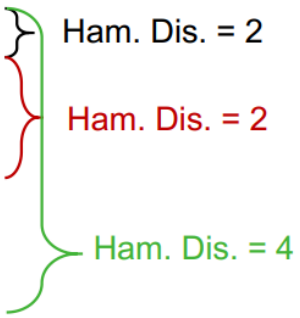
- 在数据传输过程中，由于系统电平波动，内存可能存在随机的翻转比特 flipped-bits
- 这种由随机原因产生的错误称为软错误 Soft error，一般系统会自动恢复正常，错误不可复现
- 硬件发生错误被称为硬错误 Hard error，一般只能通过替换硬件来修复
- 传输的数据被称为 dataword，编码后的数据被称为 codeword；一般为了传输的安全性与正确性， codeword 长于 dataword

- 汉明距离 Hamming distance：两个（长度相同的）数据之间不同的比特数量
- 对于 n-error dectection，$|Parity\_bit| \geq n+1$；对于 n-error correction，$|Parity\_bit| \geq 2n+1$

奇偶校验

- 使用一个校验位 Parity Bit 来保护数据，其中校验位的值会将数据中 `1` 的数量补成奇数（奇校验 Odd parity）/偶数（偶校验 Even parity），使用异或实现
- 奇偶校验保证了任意两个不同的 dataword 对应的 codeword 汉明距离至少为 `2`，可检测单比特的错误

Dataword	Even number of '1's?	Codeword
000	True	000 1
001	False	001 0
010	False	010 0
011	True	011 1
100	False	100 0
101	True	101 1
110	True	110 1
111	True	111 0



汉明纠错码 Hamming ECC

- 内存中常使用单错误纠正、双错误检验的机制 Single error correction, double error detection, SEC/DED，被称为汉明纠错码 Hamming ECC
- 在 codeword 第 $1, 2, 4, 8, \cdots$ 位上设置 parity bit，剩余位置填充 dataword
- 从左往右，parity bit 1 保证 codeword 的第 `模2余1` 位上 `1` 的总数为偶数
- parity bit 2 保证 codeword 的第 `模4余2,3` 位上 `1` 的总数为偶数
- parity bit 4 保证 codeword 的第 `模8余4,5,6,7` 位上 `1` 的总数为偶数
- 纠错时，将错误的 parity bit 编号相加，即为错误在 Codeword 上的位置
- 编码例子：对于 `dataword = 0110 0101`：
  - `codeword = p1 p2 0 p4 1 1 0 p8 0 1 0 1`
  - `p1 = 1` `p2 = 0` `p4 = 1` `p8 = 0`
- 纠错例子：对于 `codeword = 1001 1100 0111`：

	1	0	0	1	1	1	0	0	0	1	1	1	Error
Parities	1	0		1				0					
Bit 1 check	1		0		1		0		0		1		Error
Bit 2 check		0	0			1	0			1	1		Error
Bit 4 check				1	1	1	0					1	√
Bit 8 check								0	0	1	1	1	Error

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
Parity bit coverage	p1	X		X		X		X		X		X		X		X
	p2		X	X			X	X			X	X			X	X
	p4				X	X	X	X					X	X	X	X
	p8								X	X	X	X	X	X	X	X

Bit 1 + Bit 2 + Bit 8 = Bit 11

1 0 0 1 1 1 0 0 0 1 **1** 1

Error correction

1 0 0 1 1 1 0 0 0 1 **0** 1

- 对于有双错误检验需求的情况，需要使用拓展汉明码 Extended Hamming Code：在 codeword 前增加一个 parity bit 0，保证所有位数的 `1` 的总数为偶数

独立磁盘冗余阵列 Redundant Array of Inexpensive / Independant Disks，RAID

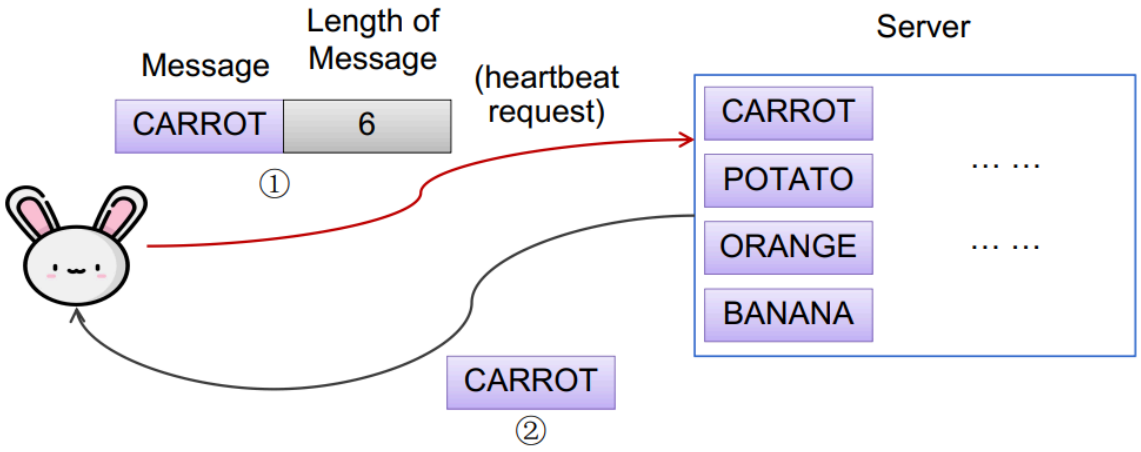
- `RAID 0`：分块 Striping：将数据分块存储在每个硬盘上，提升了读写性能
- `RAID 1`：镜像 Mirror(s)：将相同的数据同时存储在多个硬盘上，提升了读性能，降低了写性能，增加了数据安全性
- `RAID 10`：分块镜像 Striped Mirrors：先镜像，再分块存储，可靠性更高
- `RAID 01`：镜像分块 Mirrored Stripes：先分块，再镜像存储，可靠性较低，极少使用
- `RAID 2`：校验盘 Parity Disk：设置多块校验盘进行汉明码校验，使用比特位分割 bit-level striping
- `RAID 3`：校验盘 Parity Disk：设置单块校验盘进行奇偶校验，使用字节分割 byte-level striping
- `RAID 4`：使用块交错存储，采用独立的校验盘 Parity Disk
- `RAID 5`：在 RAID 3 的基础上，使用块交错存储，将校验盘中的数据分块存储在不同的硬盘中

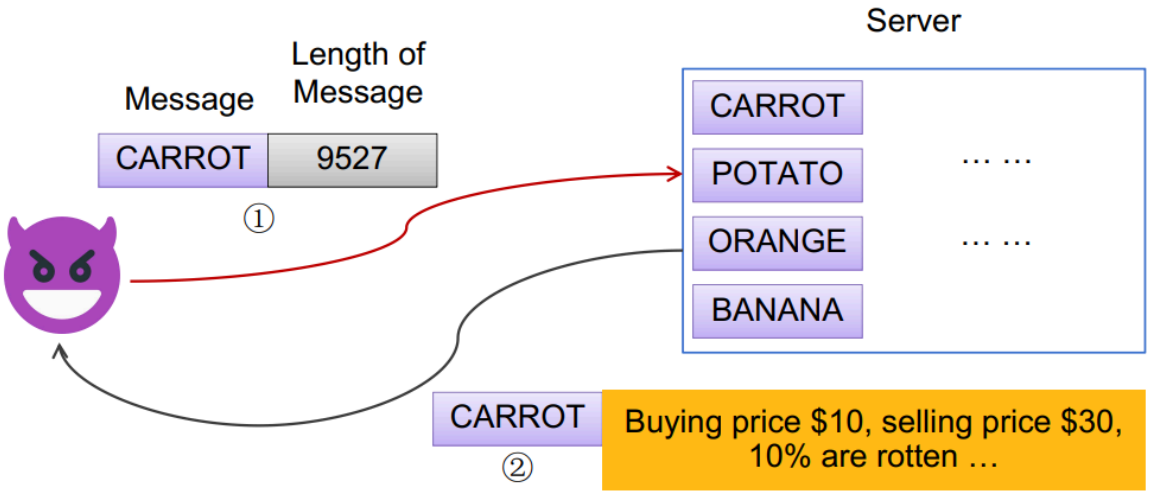
Lec30 安全性与异构计算 Security & Heterogeneous Computing

安全性 Security

Heartbleed

- Heartbeat 机制：发送端发送一个数据包，并在最后发送信息的长度；接收端发送一个相同长度的信息
- Heartbleed 攻击：每次发送数据包时，发送一个长于信息长度的字数，导致服务器返回更长的信息，导致数据泄露

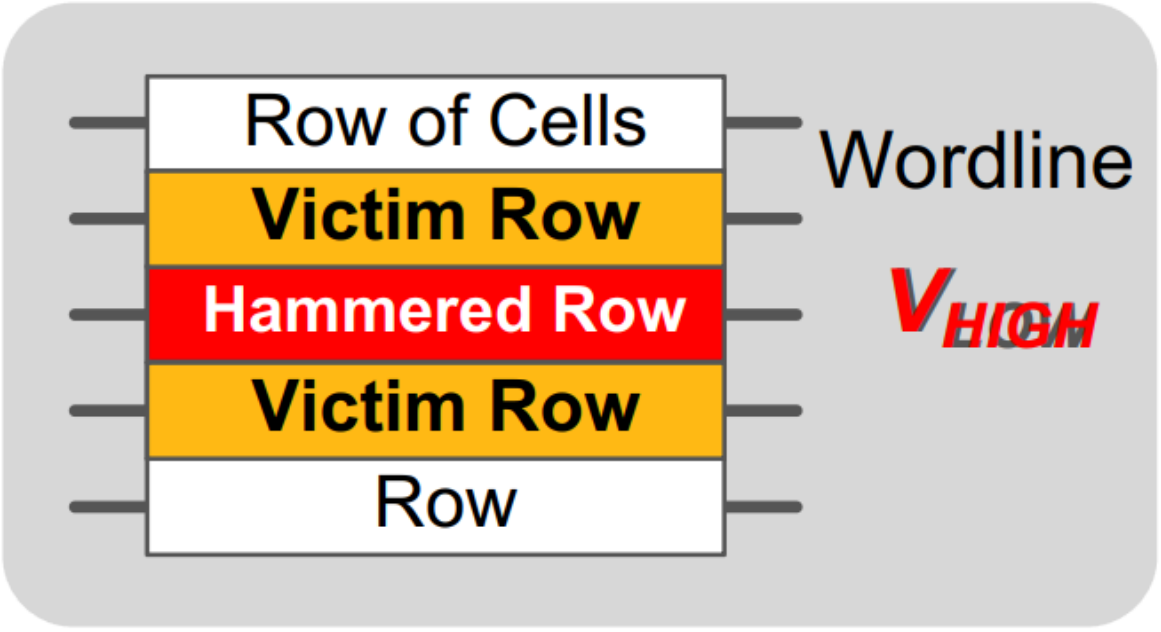




- 解决方法：进行长度检验 length check

行锤 Rowhammer

- 对于内存使用的 DRAM，对某一行（Hammered Row）进行高频反复充电（高频-低频），导致附近的内存行存储的值发生变化



- 解决方法：使用数据保护与恢复机制，如错误纠正码 ECC；防止高频访问

侧信道攻击 Side-channel Attack

- 通过测量程序执行时间、访问缓存行、功耗、声音等信息，推测秘钥的值，减少猜测空间
- 一般对 L3 Cache 进行攻击
- 冲刷与重载 Flush & Reload：攻击者先冲刷 flush 缓存，然后等待一段时间；后重新加载部分值，测量重新加载时访问的速度，以得知被攻击程序使用的内存地址；需要多次尝试，以恰好碰到加密程序段运行时间；对于包含型缓存 Inclusive Cache 更容易攻击成功

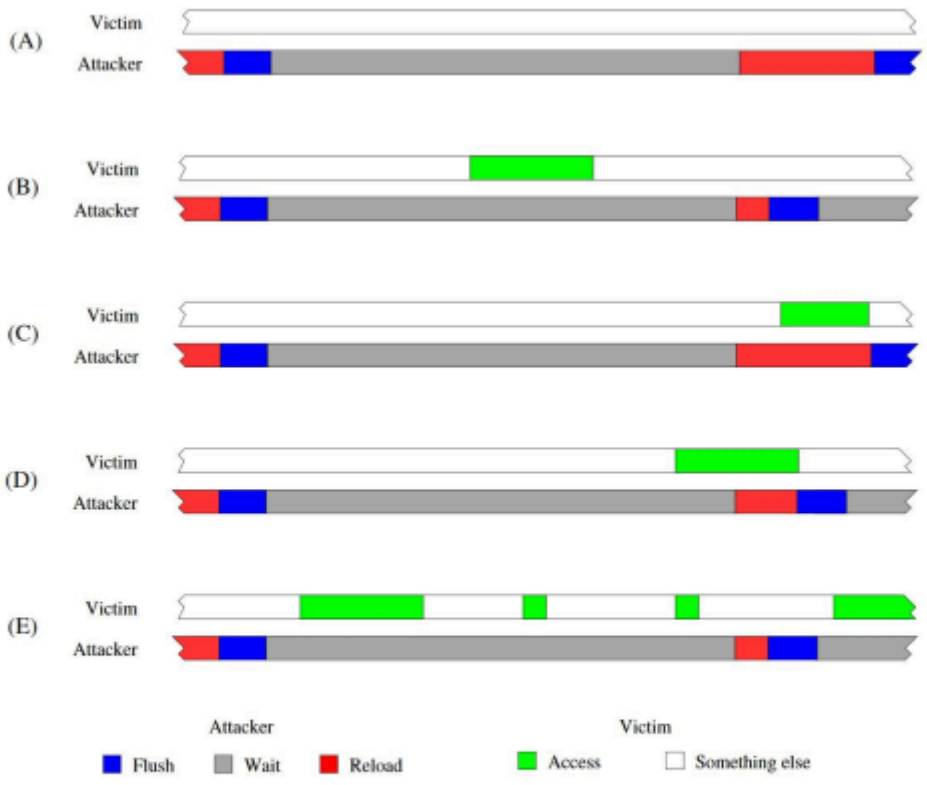


Figure 3: Timing of FLUSH+RELOAD. (A) No Victim Access (B) With Victim Access (C) Victim Access Overlap (D) Partial Overlap (E) Multiple Victim Accesses

- 解决方法：运行时间对齐（但是会损失性能）；程序访问地址隔离；对清除缓存行的指令 `clflush` 进行权限检验

熔断 Meltdown

- 对于支持乱序执行、页预取的 CPU，不应该被执行的指令会在缓存中留下痕迹
- 一般通过访问非法地址 `secret * 4096`（4KB = 1 Page），并侧信道攻击这个地址的访问速度，来倒推秘钥的值
- 例：
  - 准备：攻击者在自己的用户空间准备一个大数组 `probe_array`
  - 触发乱序执行：攻击者尝试读取内核中的 `secret` 值（假设为 84）
  - 留下痕迹：虽然 CPU 最终会报错，但在报错前，乱序执行已经完成了代码：`access(probe_array[secret * 4096])` 的执行，在缓存中预加载了 `probe_array` 的第 84 个内存页

    - 侧信道探测（Reload）： 攻击者遍历自己的 `probe_array` 中的每一页，并测量读取时间，推断出 `secret` 的值为 84
- 会受到噪声的影响，因此也需要清空缓存
- 解决方法：在敏感代码段禁用分支预测（损失性能）；后续 CPU 已经进行了硬件的加固

幽灵 Spectre

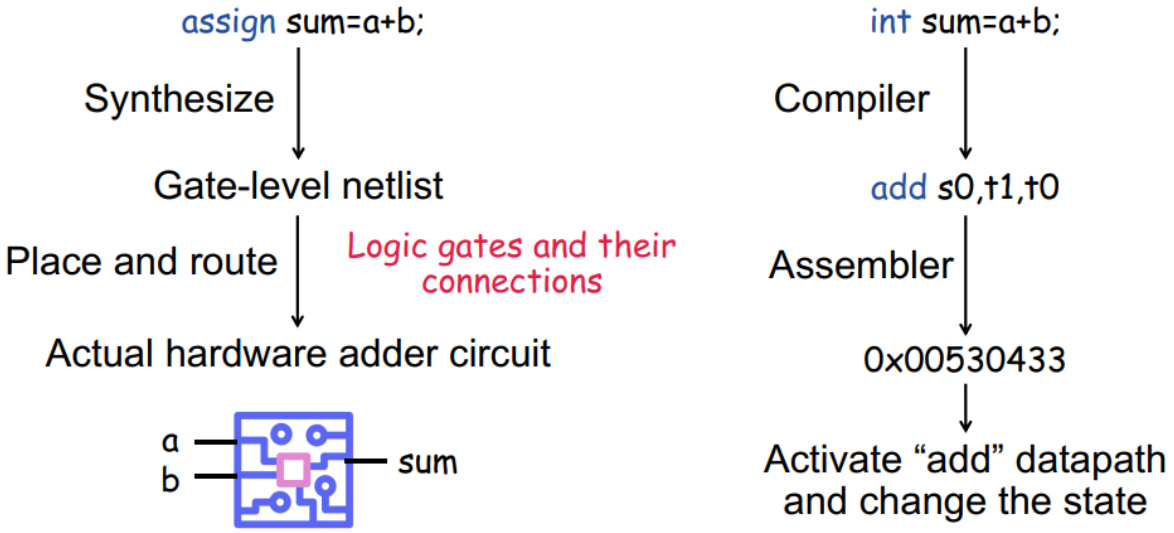
- 利用 CPU 的分支预测失败产生的数据会残留在缓存中，使用数组进行非法的越界访问，并侧信道攻击这个地址的访问速度，来倒推整个内存的值
- 解决方法：在敏感代码段禁用分支预测（损失性能）；后续 CPU 已经进行了硬件的加固

异构计算 Heterogeneous Computing

- 异构计算 Heterogeneous Computing：有多种内核或处理器单元的计算系统，使用协处理器 Coprocessor 进行特化的辅助计算（如独立显卡）
- 现代 CPU 内部使用 System-on-chip, SoC 的方法进行设计，同一片 CPU 上集成了多个异构的系统，支持并行调度执行，支持共享内存

现场可编程门阵列 Field programmable gate array, FPGA

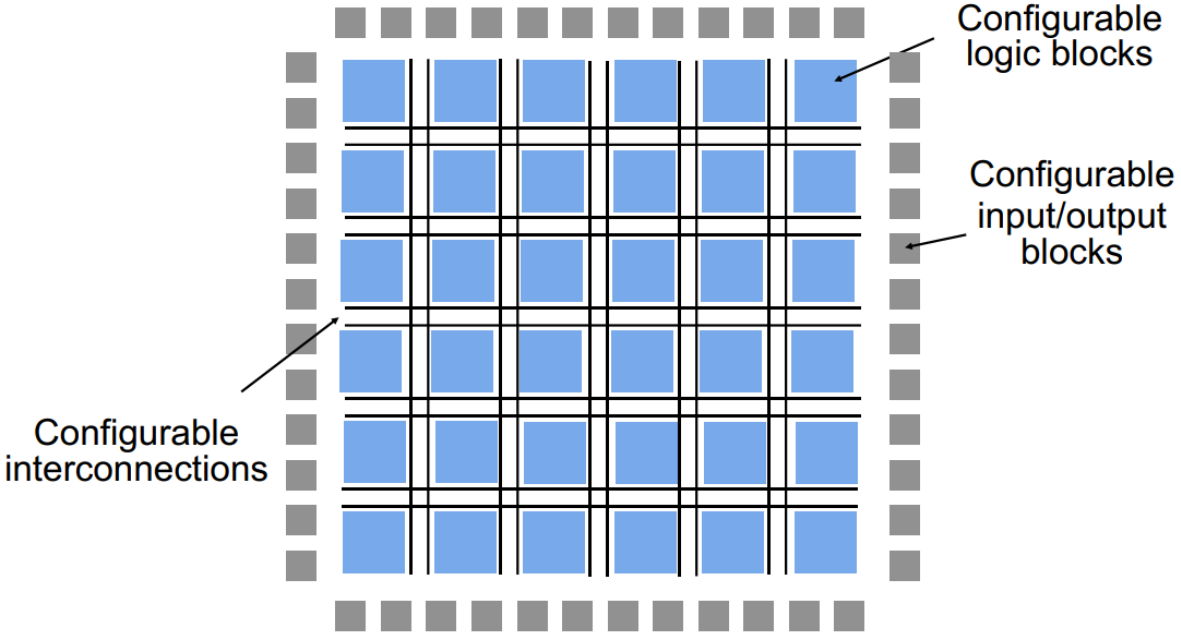
- 使用硬件进行编程，达到硬件级加速的效果
- 使用硬件设计语言进行编程，如 Verilog HDL；在编程完成后，先烧录进入 FPGA 硬件，然后可以直接使用
- 在芯片设计领域，设计师常在 FPGA 上进行芯片测试



- 在每个可配置逻辑块 Configurable logic block 中，使用 2-input 查找表 Lookup table, LUT 来实现任意逻辑门

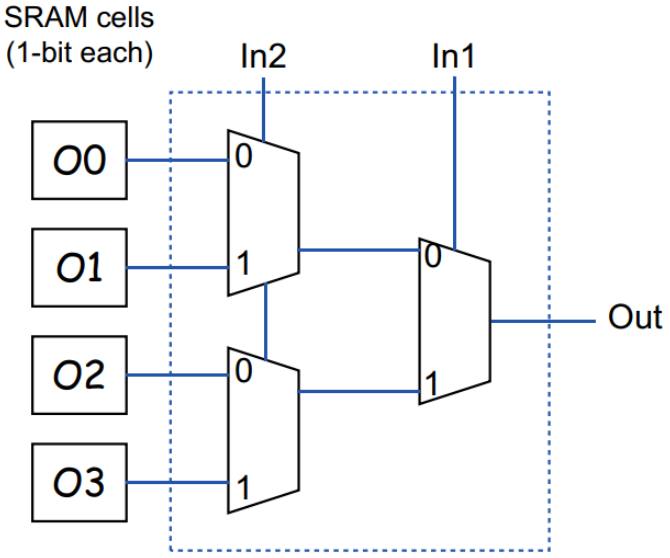
FPGA implements any logics

(given enough hardware resources)



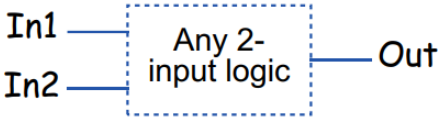
LUTs inside configurable logic blocks

- Look-up table (LUT)



In1	In2	Out
0	0	O0
0	1	O1
1	0	O2
1	1	O3

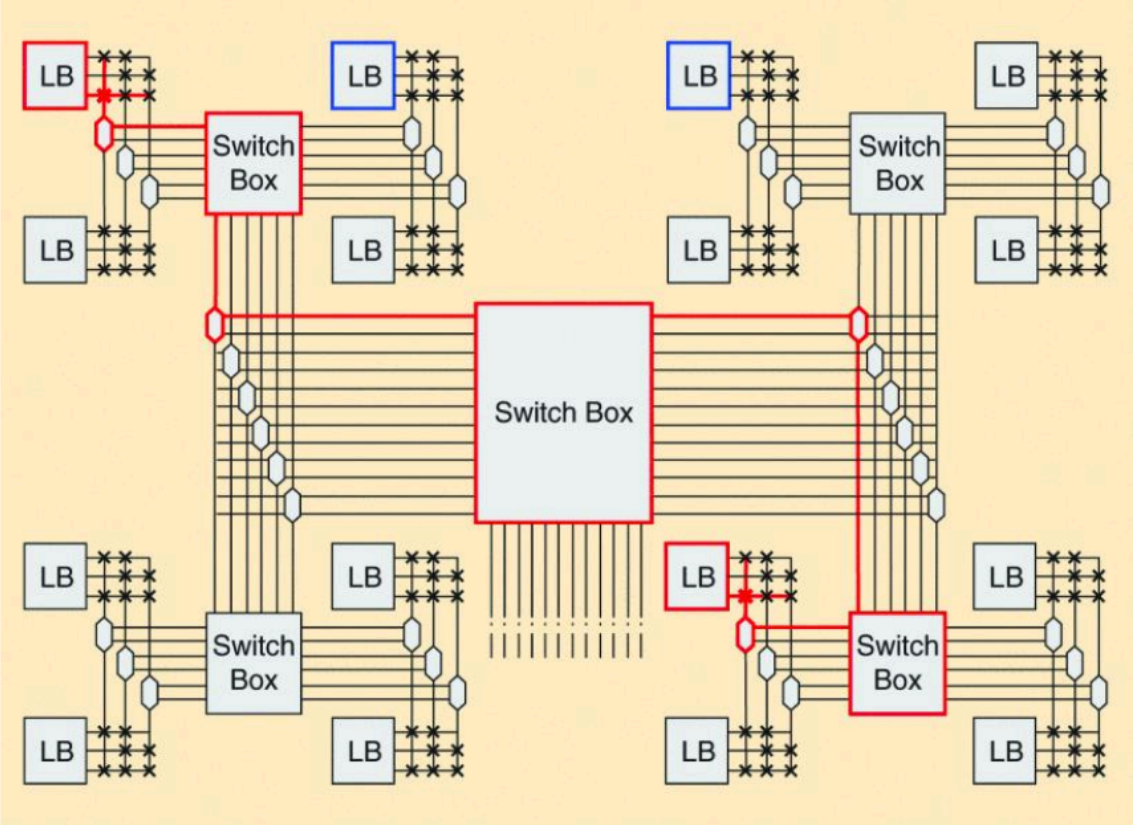
Equivalent to



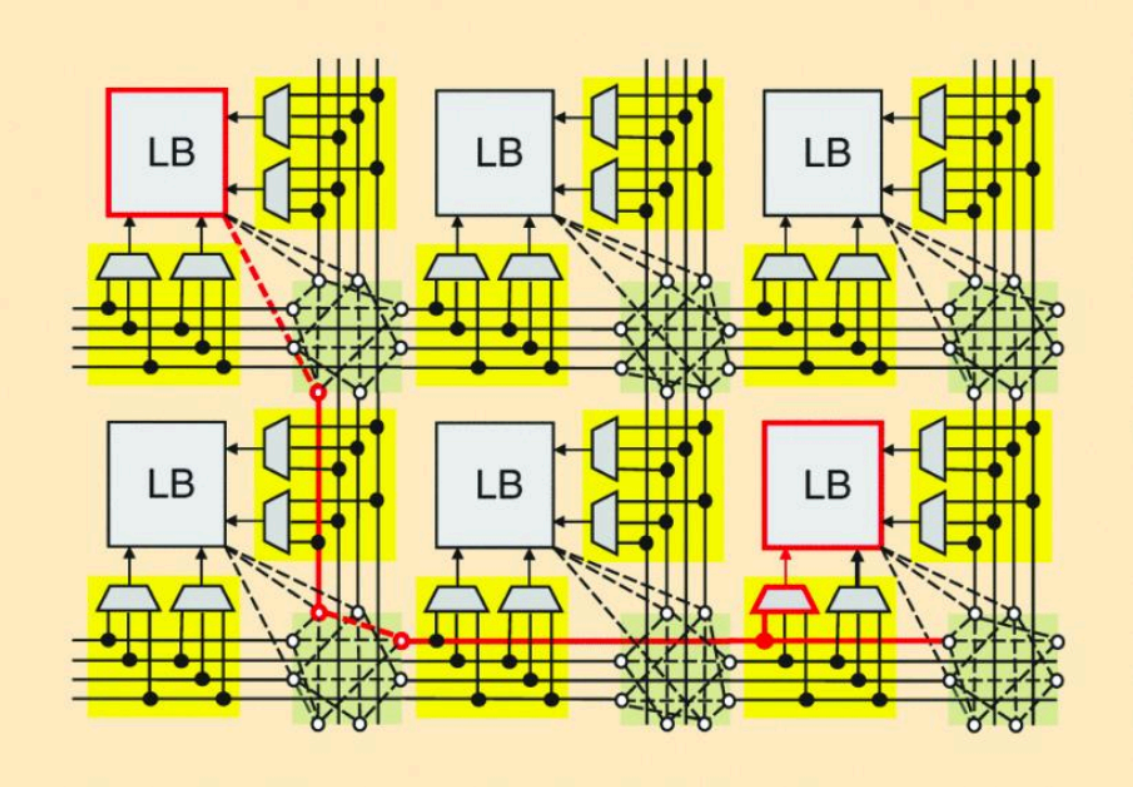
- Similarly, we can build n-input LUT implementing any n-input logic with 2^n single-bit SRAM cells

12

- 在现代的 FPGA 的可配置逻辑块中，一般采用6输入查找表；还会有加法进位链、可配置寄存器等
- 不同种类的 FPGA 设计：
  - 早期设计：分级 Hierarchical FPGA



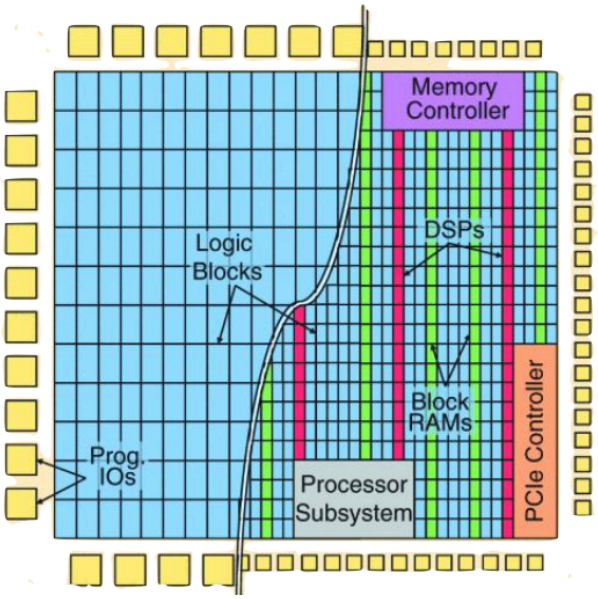
- 现代设计：岛型 Island-style FPGA



- 现代的 FPGA 有更多的功能模块可供使用

Heterogeneous

Traditional FPGAs



Modern FPGAs