上海科技大学
ShanghaiTech University

CS110 Midterm2 Review

Computer Architecture

Datapath · Pipeline · Cache · Parallelism

by Luntian Zhang

立志成才报国裕民

Contents

- Overview
- Datapath & Controller
- Pipeline & Multi-Issue
- Cache & Memory Hierarchy
- Thread-Level Parallelism

Overview

CS110 主线：

> “ 如何让 CPU 正确执行指令，并不断提高性能?

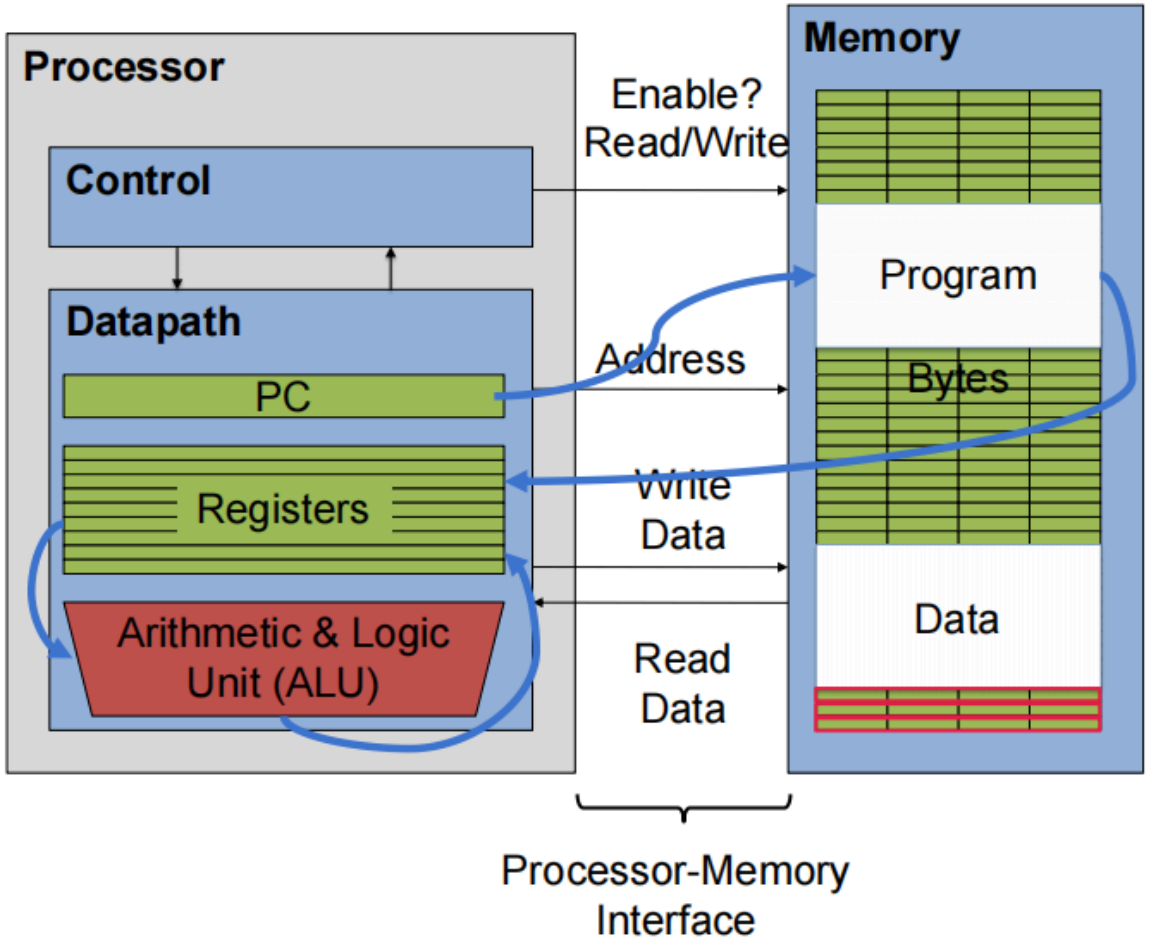
1. **Datapath / Controller**：一条指令如何被执行?
2. **Pipeline / Multi-Issue**：多条指令如何重叠执行?
3. **Cache**：如何缓解 CPU 与内存速度差距?
4. **Parallelism**：如何利用多线程、多核、SIMD 提升性能?

1. Datapath & Controller

核心问题：

> “ 给定一条 RISC-V 指令，CPU 内部的数据应该怎么流动?

Datapath Overview



Single-Cycle CPU

单周期 CPU：

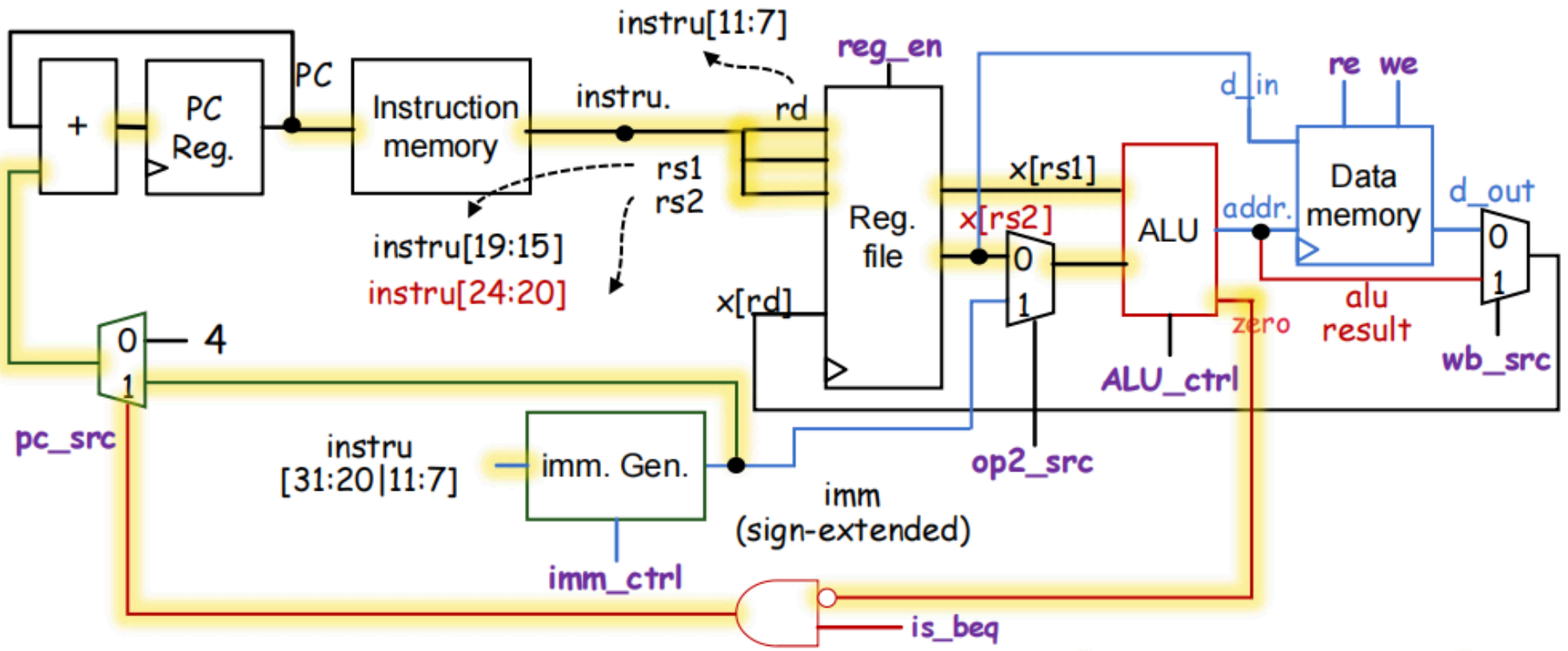
- 每条指令 一个 cycle 完成
- cycle time 由最慢指令决定
- 通常 `lw` 最慢：
  - 取指，读寄存器，ALU 算地址，访存，写回

关键缺点：

- 硬件利用率低
- 时钟周期长
- 所有指令都被迫等最慢路径

> “ 引出 pipeline

Datapath timing analysis



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

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

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

} max

2. Pipeline & Multi-Issue

核心问题：

> “ 如何让多条指令像流水线一样重叠执行？

Iron Law of Performance

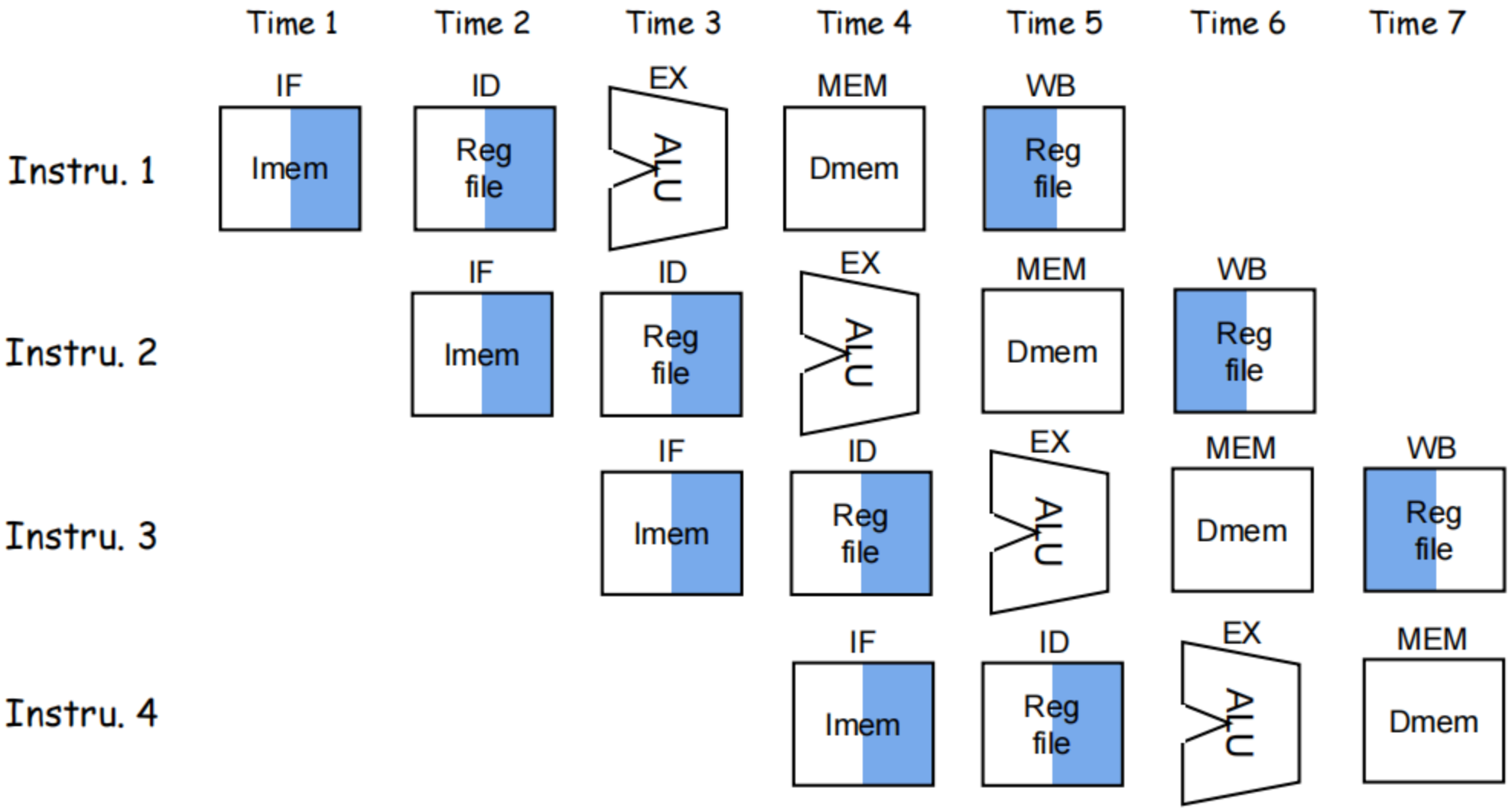
$$CPU\ Time = Instruction\ Count \times CPI \times Clock\ Cycle\ Time$$

优化方向：

1. 减少指令数
2. 降低 CPI
3. 缩短 clock cycle time

Pipeline对比单周期主要降低哪个factor?

5-Stage Pipeline



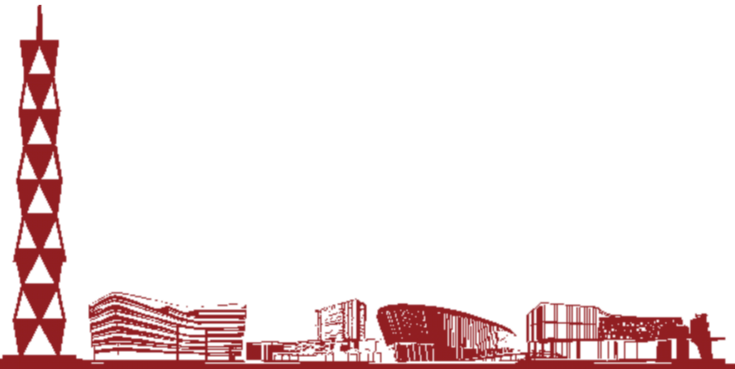
Three Hazards

Structural

硬件资源冲突

例如：

- 指令和数据共用单端口内存
- bubble, nop, stall?



Data

1. 插入气泡（Stall/NOP）：硬件停顿，性能差，但简单。

2. 数据前递/转发（Forwarding/Bypassing）：不等结果写回寄存器，直接从EX/MEM或MEM/WB流水线寄存器“偷”出来用。

- 关键判断条件：
  1. `EX/MEM.rd == ID/EX.rs1` (或 `rs2`)
  2. `EX/MEM.rd != x0`
  3. `EX/MEM.reg_en == 1`

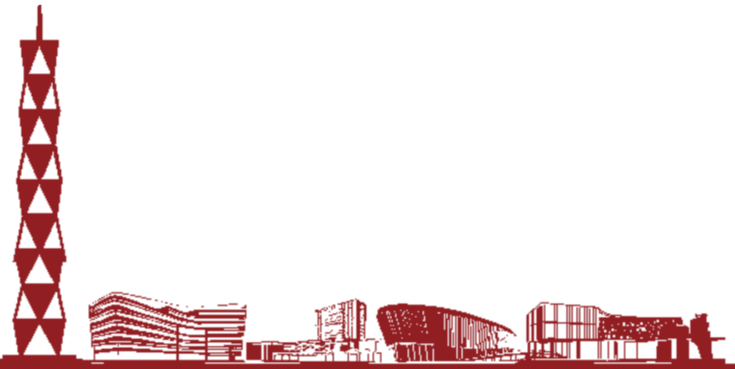
3. 编译器代码调度: 编译器重排指令顺序，拉开依赖距离，避免硬件停顿。

** Load-Use 冒险

- 场景

```
lw t0, 0(t1)
add t2, t0, t3    # 紧跟的指令就用到了 t0
```

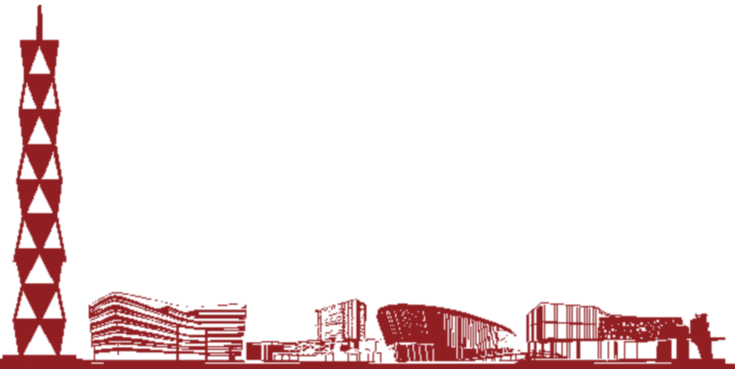
- 问题：`lw` 在 MEM阶段结束 才能拿到数据，但下一条 `add` 在 EX阶段开始 就需要。
- 后果：即使有转发，也必须强制停顿1个周期（插入一个气泡）。
- 硬件操作：检测到 Load-Use 冒险，冻结PC和IF/ID寄存器，在EX段插入一条空指令。



Example: Data hazard solution

```
add x5, x1, x2
sub x6, x5, x3
```

Solution: ?



Example：Data hazard solution

```
add x5, x1, x2
sub x6, x5, x3
```

Solution:

1. 插入 bubble
2. Forwarding / bypassing
3. 编译器调度

Example: Forwarding Rule

```
if EX/MEM.rd == ID/EX.rs1
if EX/MEM.rd == ID/EX.rs2
?
?
```

Also requires: ?

Example: Forwarding Rule

```
if EX/MEM.rd == ID/EX.rs1
if EX/MEM.rd == ID/EX.rs2
if MEM/WB.rd == ID/EX.rs1
if MEM/WB.rd == ID/EX.rs2
```

Also requires: ?

```
rd != x0
reg_en == 1
```

Control

- 静态预测
  - 总是预测“不跳转”（Not Taken）
  - 简单，但遇循环必有惩罚
- 动态预测
  - BHT (分支历史表)：记录分支历史
  - 2位饱和计数器：状态机决定预测方向，必须连续两次预测错才反转
- 预测失败惩罚
  - 需要冲刷（Flush） 流水线中已取但错误的指令

Example：预测失败冲刷

“ 某CPU采用“预测不跳”策略，遇beq指令，分支在EX阶段计算出结果。
若需跳转，预测失败，需冲刷几条指令?

Example：预测失败冲刷

“ 某CPU采用“预测不跳”策略，遇`beq`指令，分支在EX阶段计算出结果。
若需跳转，预测失败，需冲刷几条指令?

“ 答案：2条。
当前`beq`在EX，其后的`IF`和`ID`阶段各有一条错误指令，需将它们变为`nop`。

Multi-Issue

目标：

> “ 一个 cycle 发射多条指令

Improve which factor?

两类：

- Static multi-issue / VLIW：编译器负责打包
- Dynamic multi-issue / Superscalar：硬件动态判断

Multi-Issue Example

可以一起发射：

```
add x5, x1, x2
lw  x6, 0(x3)
```

原因：

- 一条走 ALU path
- 一条走 load/store path
- 无数据依赖

3. Cache & Memory Hierarchy

核心问题：

> “ CPU 很快，DRAM 很慢，如何让常用数据靠近 CPU?

Locality

Temporal Locality

刚访问过的数据，之后可能还会访问。

```
sum += A[i];sum += A[i];
```

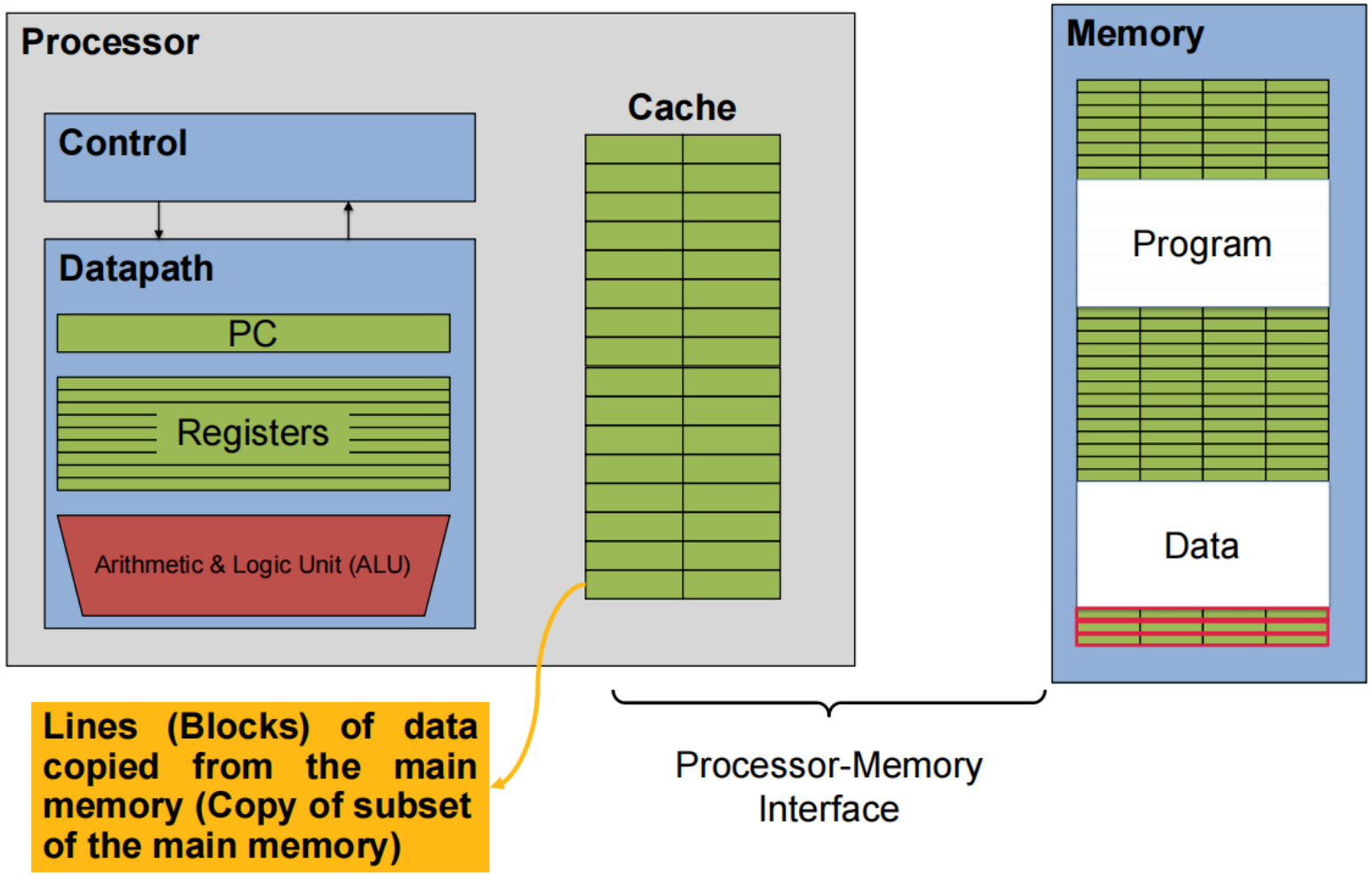
Spatial Locality

访问某个地址后，附近地址也可能被访问。

```
for (i = 0; i < N; i++)    sum += A[i];
```

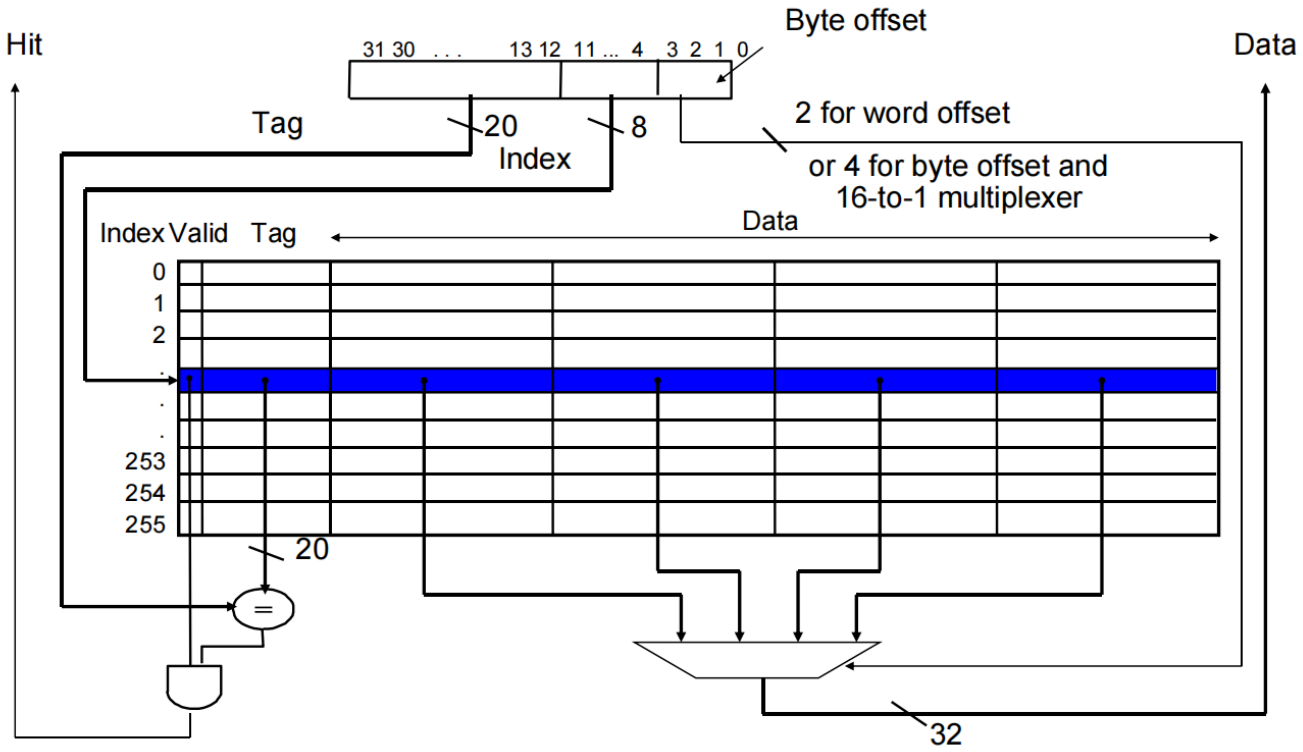
How about the linked list?

Add a cache


Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit (ALU)
Cache
Memory
Program
Data
Lines (Blocks) of data copied from the main memory (Copy of subset of the main memory)
Processor-Memory Interface

Cache Address Fields

```
address-> | Tag | Index | Offset |
```



Bit Calculation

- Address width = (w)
- Cache size = (C)
- Block size = (B)
- Associativity = (N)

$$offset = \log_2 B$$

$$sets = \frac{C}{B \times N}$$

$$index = \log_2 sets$$

$$tag = w - index - offset$$

Example: Cache Bits

32-bit address

Cache size = 4 KB

Block size = 16 B

4-way set associative

$$offset = \log_2 16 = 4$$

$$sets = \frac{4096}{16 \times 4} = 64$$

$$index = \log_2 64 = 6$$

$$tag = 32 - 6 - 4 = 22$$

Three Mapping Policies

Mapping	Placement	Comparator Cost	Main Problem
Direct-Mapped	One exact line	1	conflict miss
Fully Associative	Anywhere	all blocks	expensive
N-Way SA	One set, N ways	N	compromise

Direct-Mapped Cache

特点：

- 一个 memory block 只能放到唯一 cache line
- 用 index 直接定位
- 硬件简单

缺点：

> “ 多个常用地址如果 index 相同，会反复互相替换。

这就是 conflict miss。

Fully Associative Cache

特点：

- memory block 可以放到任意 cache line
- 没有 index
- 只分 Tag 和 Offset

优点：

- 几乎消除 conflict miss

缺点：

- 每个 block 都要比较 tag
- comparator 数量多，硬件贵

Set-Associative Cache

N-way set associative：

- 先用 index 找到一个 set
- 再在 set 内比较 N 个 tag
- replacement 只发生在同一个 set 内

常考点：

> LRU 是在 set 内做，不是在整个 cache 内做。

Valid / Dirty / LRU

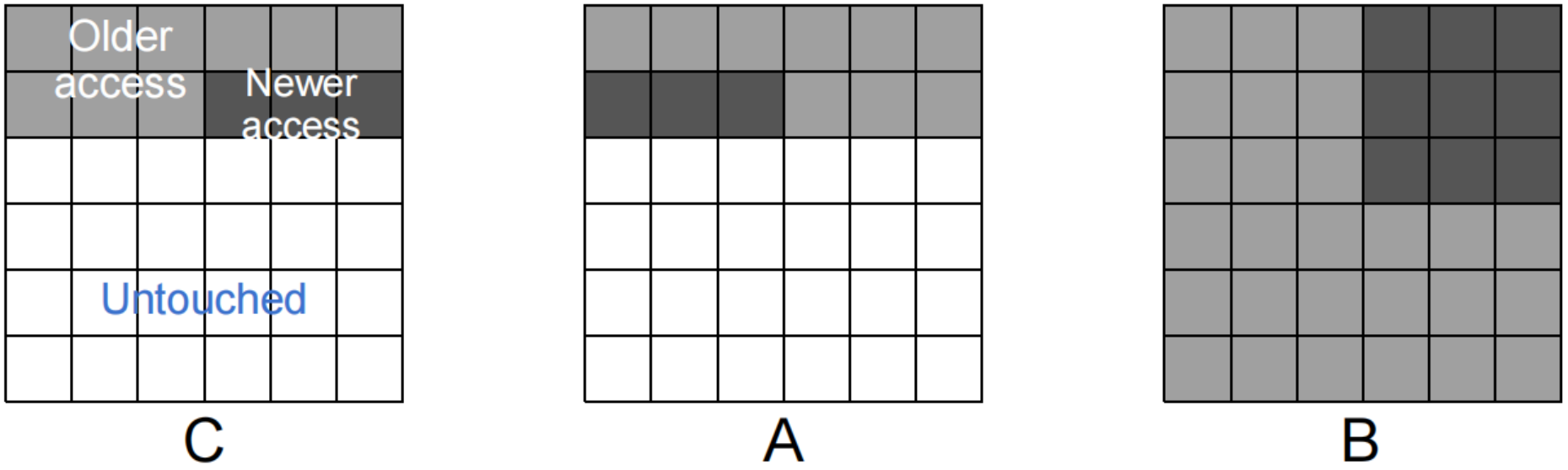
Bit	Meaning
Valid	当前 cache line 是否有效
Dirty	cache 中数据是否比 memory 新
LRU	替换时判断谁最久没用

冷启动：

> “ 所有 valid bit = 0，所以第一次访问通常 miss。

Not ideal, actually.

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



Write Policies

Write Hit

- Write-through：cache 和 memory 都写
- Write-back：只写 cache，dirty=1，替换时再写回

Write Miss

- Write-allocate：先把 block 拉进 cache，再写
- No-write-allocate：直接写下一级 memory

常见组合：

> Write-back + Write-allocate

AMAT

$$AMAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$$

多级 cache：

$$AMAT_{L1} = HT_{L1} + MR_{L1} \times AMAT_{L2}$$

$$AMAT_{L2} = HT_{L2} + MR_{L2} \times MP_{L2}$$

Local vs Global Miss Rate

L2 local miss rate：

> 在 L1 miss 的访问中，有多少比例又 L2 miss。

L2 global miss rate：

$$MR_{global,L2} = MR_{L1} \times MR_{local,L2}$$

易错点：

> AMAT 公式里通常使用 local miss rate 逐级计算。

3C Miss Model

Miss Type	Cause	Can be reduced by
Compulsory	第一次访问	prefetch / larger block
Capacity	cache 太小	larger cache
Conflict	映射冲突	higher associativity

> “ Fully associative cache 没有 conflict miss。

Example:

场景	缺失类型
(a) 程序首次访问数组 A[0]	?
(b) 工作集 200KB，但缓存只有 64KB	?
(c) 直接映射缓存中，A[0] 和 A[1024] 映射到同一行，交替访问	?
(d) 全相联缓存中，(c) 的情况还会发生吗?	?

4. Thread-Level Parallelism

核心问题：

> “ 多个线程如何共享资源、并行执行，又不破坏正确性?

Threads

每个线程有自己的：

- PC
- registers
- stack

线程之间共享：

- heap
- global variables
- address space

所以：

> “ 并行带来性能，也带来 data race。

Fork-Join Model

OpenMP 常用模型：

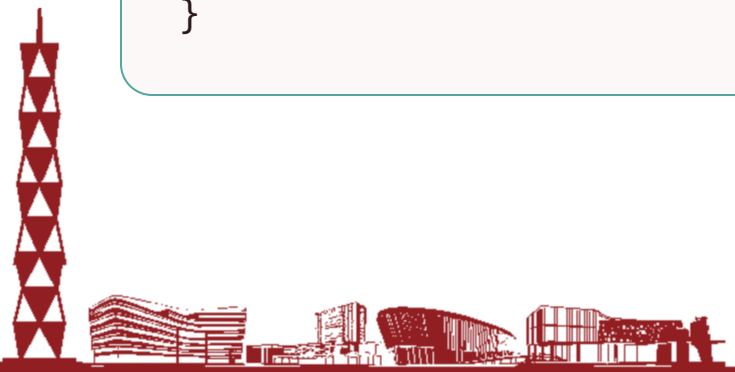
1. Main thread 顺序执行
2. 遇到 parallel region： fork 出多个线程
3. 多线程并行执行任务
4. 结束后 join 回 main thread

OpenMP Basics

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
}
```

并行 for：

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    A[i] = B[i] + C[i];
}
```



Shared vs Private

```
#pragma omp parallel for private(x)
for (int i = 0; i < N; i++) {
    x = A[i] + B[i];
    C[i] = x;
}
```

规则：

- loop index 通常 private
- 外部变量默认可能 shared
- 临时变量最好显式 private

Data Race

Data race 条件：

1. 不同线程
2. 访问同一个 memory location
3. 至少一个是 write
4. 没有同步保护

例子：

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    sum += A[i];
}
```

`sum` 会发生 data race。

Reduction

正确写法：

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
```

含义：

- 每个线程有自己的 private sum
- 最后自动合并
- 避免 data race

Critical Section

```
#pragma omp critical
{
    sum += local;
}
```

作用：

> 同一时间只允许一个线程进入临界区。

缺点：

- 太大的 critical section 会串行化
- 性能受 Amdahl's Law 限制

Lock and Atomicity

错误锁实现：

```
lw   t0, 0(s0)
bnez t0, loop
addi t1, x0, 1
sw   t1, 0(s0)
```

问题：两个线程可能同时读到 lock = 0，然后都进入临界区。

根本原因：

> “ `lw` 和 `sw` 不是一个原子操作。

RISC-V Atomic Lock

```
try:
    li t0, 1
    amoswap.w.aq t1, t0, (a0)
    bnez t1, try
    # critical section
    amoswap.w.rl x0, x0, (a0)
```

- `amoswap` 原子地读旧值、写新值
- `t1 = 0`：成功拿锁
- `t1 = 1`：锁已被占用，继续自旋

Multicore vs SMT

Concept	Meaning
Multicore	多个物理核心，真正并行
Hardware Thread	CPU 硬件支持的线程上下文
SMT / Hyper-Threading	一个核心同时维护多个硬件线程
Software Thread	OS 管理的线程

易错点：

> “ SMT 不是多个完整核心，而是在一个核心内提高资源利用率。

Amdahl's Law

$$Speedup = \frac{1}{(1 - F) + \frac{F}{S}}$$

其中：

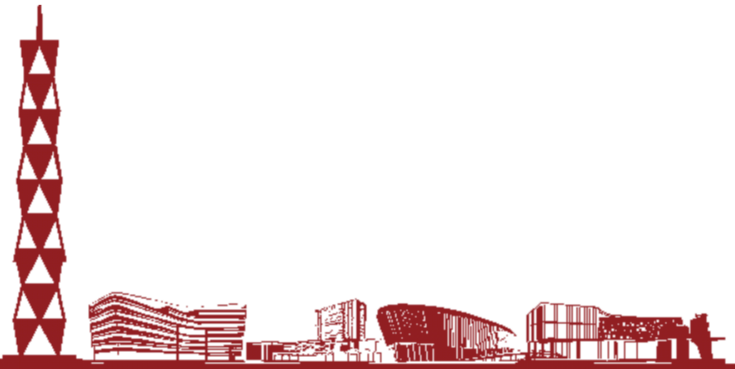
- (F)：可加速部分比例
- (S)：该部分加速倍数

结论：

> “ 串行部分再小，也会限制最终加速比。

上海科技大学
ShanghaiTech University

Good luck with your mid2term!



立志 成才 报国 裕民