Great ideas in CA: Abstraction; Make common case faster; Parallelism; Pipeline; Redundancy for accuracy; Performance Measurement; Amdahl's law: 加速比 $S(N) = \frac{1}{1-P+P/N}$ N: 处理器数量 P: 可并行化比例

Moore's Law: chips 中 transistors 数量两年翻一倍 △: prediction, not law

Memory Hierarchy: CPU → CPU cache → physical memory → solid state memory → virtual mem

reg. on-chip cache (L1,2,3) main mem; RAM 固态内存; SSD; 闪存 从左至右, 速度↓, 但容量↑

Info_representation: LSB: 最右; MSB: 最左: ob(0⋯0)

表示负数: 2's complement: MSB → sign bit; 负, 则全位翻转后加1: $(a_n a_{n-1} \cdots a_1 a_0)_2 = -a_n 2^n + a_{n-1} 2^{n-1} + \cdots + a_1 2^1 + a_0 2^0$

小心 overflow!! (signed operation 中) ⇒ 最高位 (MSB) 接受的进位与输出的进位不同, 则 overflow! (此rule是signed中的)

*: 不论 signed or unsigned, 操作 +− 结果只可能造成 overflow; 无 underflow (它只在 subnormal 有)

FP32: $(-1)^S \times Mantissa \times 2^E$, M&E 在不同 type 下算法不同: normal: bias 为 −127

Expo	Fraction	Value
全1	0	±∞
全1	非0	NaN
全0	0	Zero
全0	非0	Subnormal

$M = (1.\text{Fraction})_2$, $E = (\text{Expo})_{10} - 127$ Expo不可全为1!!

Subnormal: $E = -126$, $M = (0.\text{Fraction})_2$

Range: normal: $\pm 1.00\cdots \times 2^{-126} \sim \pm 1.11\cdots \times 2^{127}$ 注意!

subnormal: $\pm 0.00\cdots1 \times 2^{-126} \sim \pm 0.11\cdots1 \times 2^{-126}$ → $E = 2 - 2^{t-1}$

Core: ① Expo 全1与全0, 有特殊意义, 切记! ② bias: 设 Expo 有 t bit, 则 $|bias| = 2^{t-1} - 1$; sub中: →

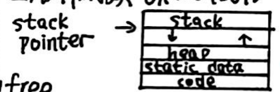
C: ① 宏的括号问题 ② True or False in C: False ⇔ 0(int) & Null; True ⇔ ¬False.

③ Size of type:

	char	short	short int	int	long int	unsigned int	void*	size_t	float	double
32 bit	1	2	2	4	4	4	4	4	4	8
64 bit	1	2	2	4	8	4	8	8	4	8

④ Little Endian: | 12 | 34 | 56 | 78 | (3 2 1 0) 地址从低到高, 从LSB至MSB放bit! 如左, 该存的数: 0x12345678

⑤ structure: alignment ⑥ func传参, 欲修改参于func内, 传其指针



⑦ Mem management: stack: func() 中局部变量; heap: malloc/calloc/realloc, 手动 free

⑧ 只有 Array 可隐式转化为 pointer, 反之不行! func 传参传 arr, arr 强制退化为 pointer

RISC-V: R-type: x[rs1] x[rs2] 间加\减\位运算\比较, 将 x[rd] 返回给 rd; 还可是 shift 操作

shift 中: sll, 左移, 总用0填; srl, 右移, 前面总用0填; 而 sra 右移, 前面用符号位填充!!

移动多少位? rs2 中低5位所代表的十进制数 (unsigned); 除此之外, 操作如比较, +−等, 都是进行 signed operation (除非指令声明是视作 unsigned 来操作)

I-type: instr rd, rs1, imm; instr rd, imm(rs1) 后者为 load 指令, 前者为非 load 指令

Warning ⚠ RISC-V 中所有 imm 均为 signed ⊙⊙⊙ (2byte) (1byte)

指令操作 imm 时, 可用 u 声明视之为 unsigned imm (I type 中只有 sltiu); lh, lb 时, 用 byte 与 halfword 的 MSB 填充至 32 bit, 4 byte, 存入 rd; 而 lhu, lbu 时, 用0填充

S-type: sw/sb/sh rs2, imm(rs1), 将 rs2 中 word/byte/halfword 存进 addr., 而 addr. = x[rs1] + imm; ⚠ sb. sh 不进行 sign-extension! 直接低4.8位放过去, 0填充

B-type: instr rs1, rs2, L (imm/label), go to PC+offset if ⋯条件, else PC+4

offset = imm / offset = (label − current PC)

△: 为了保证跳转是 2-byte aligned, 故 offset 最后一位会强制归0, 把 [12:1] 编码这样, 跳转指令范围为: $\pm 2^{10}$ 条 (offset 13位, MSB 表±, 12位 ⇒ $\pm 2^{12}$ byte ⇒ 2^{10} 条)

J-type: jalr rd, rs1, label|imm, PC+4 给 rd, 然 jump to rs1+offset; $\pm 2^{18}$ 条 range

offset = imm or (label − PC); 为保证 2-byte aligned, offset 最后一位归0, [20:1] 编码 ↗

jal rd, label PC+4 给 rd, jump to label, i.e., PC = PC + offset, offset = label − PC

△: 相当于: jalr 允许在 rs1 中地址上跳转, jal 只能在 PC 基础上 jump

Pseudo: j label ⇔ jal x0 label; jal label ⇔ jal ra label; jr rs1 ⇔ jalr x0 rs1 0 小心引索

U-type: lui/auipc rd, imm, lui: rd ← imm<<12; auipc: rd ← PC + imm<<12

△: li x1, imm ⇒ (l)ui x1, imm[31:12] + addi x1, x1, imm[11:0] OK!

△: 若 imm[11:0] 代表负数, 则 +1! 因为 addi 要 sign-extension! +11⋯1 & +1 ⇒ +1, 00⋯0

Misc: imm 范围: 可以看 greencard instr 组成中, imm 显示的最高位是多少, 便多少位

Calling convention: 约定: ① 传参用 ai reg 序列 ② 返回值放在 a0/a1 reg

③ caller & callee: A func 中调用 B func 中, 则调用前, A 要把自己 caller-saved 值存起来 (sp & sw), 如 A func 中的运算中间值, 然后按照①, 把给 B 的参数放在 ai 中之后调用并进入 B, 一进 B 要把 callee-saved reg 中存起来 (sp & sw), 它们便可供 B 自由使用, 然后把返回值放入 a0, a1; 恢复 callee-saved reg (sp & lw), 返回至 A; A 恢复 caller-saved reg (sp & lw), 可使用之前 a0, a1⋯ 中的 B 返回值 | 跳至 ra 中的 PC 地址

Prologue: 存 ra, 存 callee-saved reg, SP↓; Epilogue: 恢复 ra, 恢 callee-saved reg, SP↑,

imm 范围补: I-type: 除 slli、srli、srai 为 5位 unsigned, 其余均 12位 signed

*: 在转为机器码时, 它们有类似于 funct7 的编码, 体现于 imm[11:5]

S-type: 12位 signed; SB/B-type: 13位 signed; U: 20位 unsigned; UJ 21位 signe

ISA: X86-32 (IA-32) X86-64 (AMD64). ARM (advanced RISC Machine)

RISC; ISA 定义一个特定 CPU 支持怎样的 operation, 以及如何 implement, 即: 什么 CPU 支持什么汇编. 如 X86 汇编可在 X86 架构 CPU 跑, 但不能在 ARM CPU 跑。

Wrap up: 指令集 ←决定— CPU 架构 = ISA ←对其开发— OS, 如 Linux 支持 x86. arm. RISC-V ISA

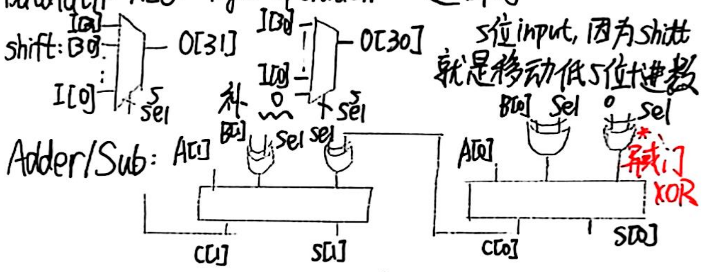
△. python 可在不同 OS 不同 ISA 上运行, 只要有合适的 python 解释器

△. 一个 OS 上的可执行文件一般不能在另一 OS 上运行; 21 signed: [−1048576, 1048575]

Imm: 12 signed: [−2048, 2047] 5 unsigned: [0, 31] 13 signed [−4096, 4095] 20 unsigned [0, 1048575]

DataPath: ALU: logic operation: 逻辑门

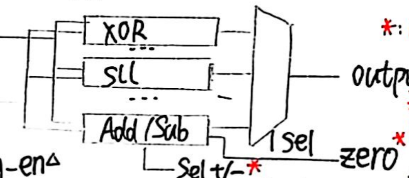
shift: 5位input, 因为shift就是移动低5位个数



Adder/Sub: 补码 XOR

operand1 output 因题而异！ zero: 可大或门 *(全0,则0!) *: 此两处设计可能

*复制一个, 以输出 X[rs1] & X[rs2]

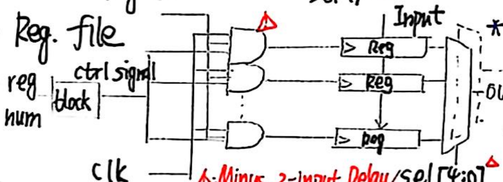


Reg. file

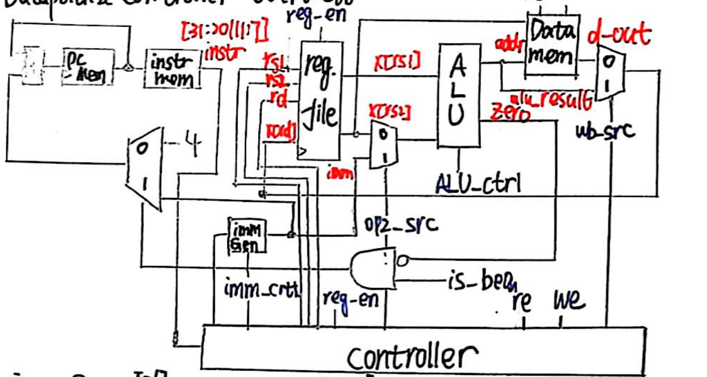
△: 由5个2-plexer 构成, 故sel[4:0]

△: 它为1时, 才允许 rd 写入!

△: Minus 3-input Delay/sel[4:0]

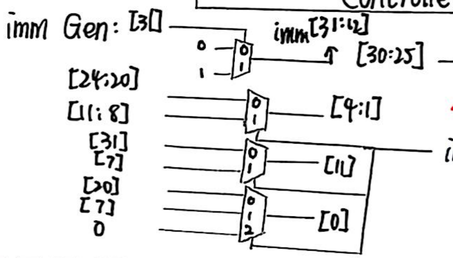


Datapath & Controller overview:



imm Gen:

△: 若要支持左移12位的imm, mux中加0; 且 imm[31:12]可为instr [31:12]



$2^?$	0	1	2	3	4	5	6	7	8	9	10	11	12
	1	2	4	8	16	32	64	128	256	512	1024	2048	4096

Misc 不会!!

static/global 数据会空初始化!(值→0, ptr→NULL); 而main中

$1ps = 10^{-12}s$, $1GHz = 10^9 Hz$.

Pipeline: $\frac{Time}{Program} = \frac{Instr}{Program} \cdot \frac{Cycles}{Instr} \cdot \frac{Time}{Cycle}$ CPI

5-stage: IF ID EX MEM WB, timing 取决于 谁延时最长

两stage间加一层reg; 但有Hazard: ① Structural: 如 reg. file 能否又读又写, I&Dmem 能否同时又读又写 ⇒ 硬件解决

② Data: reg-file: write after read (读不到updated), 一种是 nop (无操作)等, 一种是 类似右图

type1

type2: lw后 1条指令就读 (load来还不及update), 一种是等(nop)

并Dmem load入的数直接给ALU, 另一种是换instr执行order, load与EX使用instr中间至少隔一条 (rescheduling)

③ Control: 对beq: 满足条件则跳转至…, 若满足, 则在beq指令后四条岂不白执行了? 一种是等4条nop; 另一种是若跳转, 则那几条flush掉, 即restore state; 另一种是 将zero发挥作用的阶段设在EX后的MEM (原: zero一直到WB才起作用; 现: MEM即有)

Timing: R-type: IF+DEC+EX+WB (DEC=ID)

I-type: 其中DEC = max { treg, timm+tmux }

IF+DEC+EX+WB *tALU

S-type: IF+DEC+EX+MEM = max { treg, treg+talu, timm+tmux+talu } X[rs2]⇒d-in, X[rs1]→sum, imm

B-type: IF+DEC+EX+WB = max { treg+tmux+talu+tand, timm }

*: I的WB: → rd; B的WB: imm→PC reg WB = tmux+tadd

Misc: C中位运算: 与& 或| 异或^ 取反~ 右左移 >> <<

sltu 检测 add/sub 在 signed/unsigned 数式中是否overflow:

unsigned: { add x2, x1, x3; sltu x4, x2, x1 or x3.

△: a=b+c (unsigned). a<b or a<c ⇒ 溢

signed: { add t0, t1, t2; slti t3, t2, 0; slt t4, t0, t1; bne t3, t4, overflow

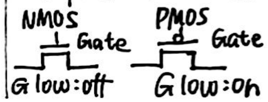
△signed: a<b or c iff c<0; a>b or c iff c>0 iff不满足⇒overflow

or (xor t5, t3, t4; t5=1: overflow).

int func(int *x) { *x += 2; }, 注意解引用!

宏定义中一定要括号充分打上!! 如

```
#define MIN(a,b) ((a)>(b) ? (b):(a))
```



NMOS Gate G low: off; PMOS Gate G low: on

NOP: addi x0, x0, 0: 0x00000013

Prologue中, 若叶函数, 不必存ra!

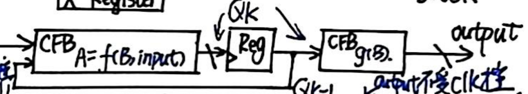
Digital Circuit:

与 或 非 与非 或非 异或 同或门 *相同为0不同为1

同步电路:

CFB: Combinational Function Block

非于FSM



Output 受clk控, output不受clk控

Moore: 输出仅于当前状态有关; Mealy: 与先前状态与input有关

它们可相互转换; Their outputs can co-exist in one 同步电路

Timing:

△: $1ns = 10^{-9}s$, $1000MHz = 1GHz = 1/ns$

$t_{clk-to-Q} + t_{CFB} + t_{setup} \le$ min period

CALL: C代码 →(compiler) 汇编语言 →(assembler) 机器码 →(linker) 可执行程序 →(loader) 放至内存

assembler: 输出object file, 含 ①header: size/position of other pieces of obj-file ② Text: machine code ③ Data: static ④ Symbol table: files label ⑤ Reallocation table△

step1: read & use directive: .text⇒②, .data⇒③, etc 不产生机器码

△: 进内存后, 分别存依赖于绝对路径的 instr & data words

step2: pseudo→normal instr & tail call优化 step3: 生成机器码

同时, track label addr 并与 data transfer instr 一并记于symbol table 中

linker: obj file & symbol table → 可执行代码: 将许多.o文件整合至一个执行文件: 将code与data module 拼接起来,

然后决定data与instr labels的addr, 最后依次将所有目标addr填为 absolute addr

reallocate

① External Ref (eg. jal) ← always relocate

② static data ref (auipc/lui&addi) reallocate

△: PC related, eg. beq. bne. jal, never need to

若何fill in? linker知 each text & data长度, 知order, 则重新算 label 与data 在.out文件中绝对addr 与symbol tables

Loader: 读.o文件header, 开辟内存, copy, 初始化reg, set PC

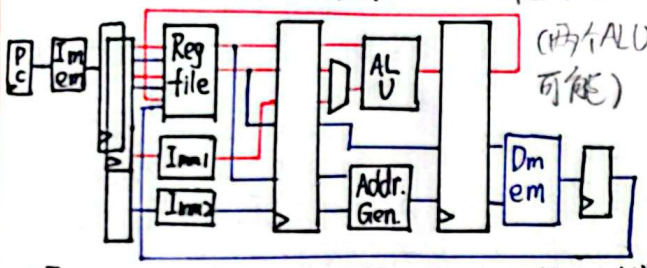
Boolean: $X\bar{X}=0$ $X0=0$ $X1=1$ $XX=X$ $(XY)Z=X(YZ)$

$X(Y+Z)=XY+XZ$ $XY+X=X$ $\overline{XY}=\bar{X}+\bar{Y}$

$X+\bar{X}=1$ $X+1=1$ $X+0=X$ $X+X=X$ $X+YZ=(X+Y)(X+Z)$

$(X+Y)X=X$ $\overline{X+Y}=\bar{X}\bar{Y}$ 保险: "∧∨" 来推一遍

Multi-issue：原来single-issue，datapath中最多1clk issue 1指令；现在一次送多条。 CPI可以<1！



IF ID EX MEM WB

红线：Arithmetic, logic and beq path

蓝线：Memory access (Ld/St) path

实战中如上：build different datapaths for different types of instruction；Ideally. issue two instructions with different type to ALU/mem datapaths. If so:

	CC1	CC2	CC3	CC4	CC5	CC6
ALU/Branch	IF	ID	EX	WB		
Load/Store	IF	ID	EX	MEM	WB	
A/B		IF	ID	EX	WB	
L/S		IF	ID	EX	MEM	WB

Very long Instr Word 即可

依然，可能有例如Data等的Hazard，forwarding etc

Multi-issue：Static：主要编译时决定instr放入issue slots的顺序，且检验hazards。硬件本身也可detect/solve hazards ← Superscalar

Dynamic，主要在硬件运行时安排issue slots order与detect hazards，当然Compiler也可help avoid hazard

△：Multi-issue不是multi-core也非SIMD。它可与SIMD or Pipeline技术结合（多线程也可与之结合）

RAW WAW WAR Hazard 命名以来源为依据！！

Parallel：Pipeline & multi-issue：instr-level

SIMD：Data level Multi-thread：(MIMD)：Thread-level

补：CALL中 Static vs. Dynamic Linking：

Static中Address resolve，若symbol table中没有，则会搜library files. 没错. 库被编译进了执行文件中，若库更新，要从头重来！且it includes entire 库 even if not all of it will be used.

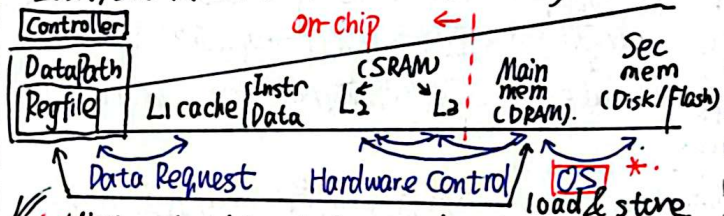
Dynamic：库文件在程序运行时会被动态加载，而非被纳入exe文件中。占用更少空间，内存更高效，库可独立更新。但运行时产生额外开销以让程序查找与加载所需库文件。

Cache：核心概念一览 Cache Blocking

Locality：Temporal：If a memory location is referenced, then it will tend to be referenced again soon.

Spatial：If…referenced, addr nearby tend to be…soon.

一些数据结构如linked list, tree, 在locality上表现差！



△：With cache, datapath does not directly access from main mem.

Term：line/block：a single entry in cache

size of block：#byte* per block. Capacity：total #byte

Fully Associative cache：有tag & offset，无index

Cache size：C，block size C_B；offset bits：b

number of block：N，bit width of mem addr (w) tag (t)

则 $2^b = C_B$，$N * C_B = C$，$w = t+b$

Nway：一组中有N个block！index：i，共k组，

则 $2^i = k$，$w = t+b+i$

Direct Mapping：1-way！！

tag index offset

Other util in cache：valid bit. an indicator to tell if each entry is valid for program

LRU：(least recently used) 满员情况下，要找victim换成新元素→最不常用的作victim

0：最常使用；N-1：LRU；选LRU数据最大的换。LRU上数据：hit 或作了一次victim，LRU→0；一次访问中未访问该cache，LRU++.

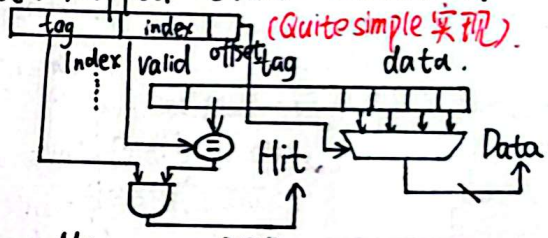
还有M(most)RU. FIFO(oldest line)LIFO(newest line)NRU

Write Policy：Write-through：write to cache and memory at the same time (longer time)

Write Back：Write data in cache and set a dirty bit to 1；when blocks get replaced and dirty bit is 1，write to mem，dirty bit→0.

Write around：Directly write to mem

Direct Mapped Cache：Hardware. (Quite simple 实现)



Cache Misses：3Cs

Compulsory：cold start or process migration

First access to block, impossible to avoid!

Sol：Block Size↑ (i.e., N↓, so fewer cold starts)

Capacity：Cache can't contain all blocks accessed by program.

Sol：Cache Size↑ (but access time↑)

Conflict (Collision)：Multiple memory locations mapped to the same cache location, conflict even when the cache has not reached full capacity

Sol：Cache size↑ Associativity (N)↑ (But access time↑)

Cache Performance & Metric:

Average Memory Accessing Time. AMAT:

AMAT = Time for a hit + Miss rate × Miss Penalty

在多级(如有 L1,2,&3)中, $AMAT_i$ 中的 $Miss Penalty_i$ 实则为 $AMAT_{i+1}$　　L1 Miss, 才在 L2 中 ref

Δ: Local miss rate: 在该level中ref中miss比例:

Eg. L2$ = $L2 Misses / L1$ Misses.

Global: all levels: $\prod_i L_i\$$ (eg. 2-levels: L2$ × L1$ (local))

Parallelism: Strong scaling: 不必 problem size ↑ 就能 Speed↑; Weak scaling: speed↑ 体现在 size↑, time 不变/↓

SIMD: Single Instruction, Multiple Data

SIMD provide performance↑ for data Level Parallel: Fetch One instr, do the work of multiple instructions

Instr Pool	
Data Pool	→ PU, → PU

Source 1 ··· x_1　x_0
Source 2 ··· y_1　y_0
　　　op　op
Dest　op(x1,y1) op(x0,y0)

常见 Intel SIMD Instr: MMX SSE AVX

有 Intrinsics, program 能直接访问汇编指令(间接). SIMD扩展指令集与Intrinsic一一对应

Generalizing Loop Unrolling: A loop of n iterations, k copies of the loop body, ⇒ 1 copy: n mod k times

k copy: ⌊n/k⌋ times; A: $A_{1,i}$ ¦ $A_{2,i}$　C: $C_{1,1}$ ¦ $C_{2,1}$

2×2 matrix multi: B_1, B_2: $B_{i,1}$ ¦ $B_{i,1}$, $B_{i,2}$ ¦ $B_{i,2}$　MISD (2)

则 $\vec{C_1} = \vec{A_1} \cdot \vec{B_1}_{i=1} + \vec{A_2} \cdot \vec{B_1}_{i=2}$

Hazard: Structral
① Seperate IMEM & DMEM
② Dual-Port memory
③ Reg write at ↑ read at ↓

Data: ① Forwarding
② NOP
③ Code scheduling

Control: ① NOP
② Flush
③ Dynamic Branch Prediction
④ Reduce Delay of branches

#pragma omp
parellel: create threads
for: Divide loop → 线(程)
critical: 串行, 1 线可执
barrier: All 线 waits
single: 进1线执, 其他等
master: 主线执, no wait
sections: 一个线一个section

TLP: Each Threads has: shared mem (heap, global var) with other threads, but has own register & PC

Each processor provides one (or more) hardware threads

OS来安排 software 线程至 hardware 线程

Fork-Join model: 程序可 split into 多个线程 (可同时进行)

多线程 program 可在 SISD与MIMD 上都可跑

OS决定 哪个线程在哪个CPU跑; 现代PC 线程数 >> 核心数。Context Switch: 将 software 线程移除, 要 save Reg & PC → mem; 将另一线程 Reg & PC 加载至 hardware 线程中, 可恢复执行 (Fork-Join)

OpenMP: Open multi-processing, a language extension

Easy to compile: #include <omp.h> (下方括号弹出执行)
Use: #pragma omp parallel
以及 compile 时: cc -fopenmp name.c

#Threads = #physical cores × #threads / core

OpenMP Shared / Private variables:

Shared: 在并行区外声明的 / Heap上的 / Static.　make a copy of it

Private: 并行区内声明的 / 用 pragma ··· private (var).

In 并行区: no premature exits! (break, return, exit ···).

Data Race: 当多线程访问同一个addr且至少其中一个 write, 便发生 Data Race. 注意它不是 data hazard!

为了解决上述问题, Synchronization (同步) 重要!

为此有一种设计: critical session with openmp

临界区是一段代码 that must be executed by a single thread at a time

在 openmp 中可通过特定指令来声明临界区。如: # pragma omp critical, 确保仅一个线程执行该段

OpenMP 会将临界区中代码编译成原子指令. 并行机制很严格, 只适用于循环的并行化

另一种设计是: 锁同步。Avoid data races by 同步读写 done by user-level routines that rely on hardware 同步 instr

---

Lock 用于授权一个线程进入 Critical Session. 只有得锁的线程才能访问临界区

Processors read lock and either wait (if locked) or set lock and go into critical session

check lock ↺ → set lock → critical session (change shared var) → unset lock

但有问题: t1 lw 得 lock, 在 bne 前, t2 也 lw 得 lock。t1, t2 都可能认为 lock 是 0 了!

RISC-V 上: two sols! ① lr rd, rs: 从 rs 地址中加载 word 至 rd, 并为硬件线程注册保留

sc rd, rs1, rs2, 只有在先前 lr 指令所保留的地址未被修改时, sc 才将 rs2 值存至 rs1 所指向的内存且返回一个 status 至 rd。(原子交换) if fail 重新 lr & sc.

try: lr t1, s1　　bne (if succ ss. t0==0) t0, x0, try
　　sc t0, s1, s4　　add s4, x0, t1

这其实是 test & valid: load 锁 x —unlocked?→ try own lock 锁 x —success→ execute. sw x0, 0(s1)

② Atomic Memory Operations (AMOs)

li t0, 1 // 用于 set lock (lock)

Try: amoswap.w.aq t1, t0, (a0) // 将 t0 (1) 写入 a0 指向内存, 并将内存旧值加载至 t1

bnez t1, try

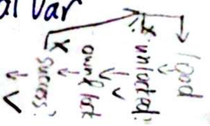
··· critical session ···

amoswap.w.rl x0, x0 (a0)

↳ 将 0 存回锁位置, 释放锁.

TLP中有 'barrier', 要 fork 都干完了才 Join

多线程访问同一 global var 不好, private 又线程间该 var 无关, 如 sum += 1, 2 ···。reduction (operation 如 + - : Var) 为每个分线程开私有 var, 操作完后 Var 过 ope, 返给 Global Var



#pragma omp barrier: Forces all threads to wait until all threads have hit the barrier

#pragma omp critical: Creates a critical section within a parallel code segment, only one thread can run at a time.

#pragma omp parallel for: for-loop automatic work-sharing

#pragma omp parallel reduction (operation: var)

#pragma omp single: code block executed by one thread only. other threads will wait. I/O操作中常见

#pragma omp master: 主线程执行, 但其他线程不等它完成

硬件并行: 有多个 processor. 它们执行独立的指令流, 且有独立的L1&L2 cache, 但共享DRAM与L3 cache(也许). 不同 processor 连动方法: ①Shared Var in memory & load/store instr ②Coordinated access to shared data through synchronization primitives (lock) that restrict access to one processor at a time.

△: 每个processor(core) 可支持多个threads!

☆ Context Switching: 从HW线程中移除一个SW线程, 需要阻挡执行, 并 save registers to memory (含PC)

SMT (Simultaneous): 利用硬件冗余, 在单核心中同时跑多个活跃线程, 以减少因内存延迟和上下文切换导致的资源闲置问题。Leveraging multi-issue

Advanced cache: 多处理器在同一memory上操作, 每个processor 又有自己的cache。但可能发生 cache incoherence! (也是 cache miss type!) P_i cache 未接到 P_j改该地址值的信息!

即: One's write invalidate other copies in other pro's cache.

Sol: Snoopy Protocol: 每个核心的缓存控制器监视总线上的读写请求; 另一核心写时, 检查自己缓存状态

False sharing: 当cache block过大时, 对不同offset 的写, 会频繁导致不必要的缓存一致性开销 (One sol: 调小!)

Optimized snoop: 若 invalid cache, 则miss: 可用 write back cache, 有dirty bit, 这样一个处理器的多次写只触发一次 broadcast. Write Back Scenario: ①eviction ②它人读该addr, 我的 cache 中该addr为dirty, 写回

而现实中的 cache coherence protocols 更复杂, 如 MOESI

△ 可以不止一个处理器 cache any memory location at a time.

若所有高级缓存的块也都在低级缓存中, 则称低级包含高级 (inclusive). 同理也有 Exclusive: $L_n \cap L_{n+1} = \emptyset \ (n \geq 1)$

不inclusive ⇒ non-inclusive.

OS & IO: OS 功能: ①开机后提供 service: 如File System ②Loads, runs and manages programs ③I/O with the rest of computer

开机后, CPU从开始地址执行指令 (in Flash ROM中存的)

Memory mapped I/O: certain addresses are not regular mem, but correspond to registers in I/O device. I/O通过读写字节流来进行操作. 部分处理器有特定I/O指令。

挑战: 处理器 I/O吞吐与设备速率差 9个数量级 (不太对!)

交互方式: ①Polling: CPU循环读取控制寄存器, 等待 ready bit (0→1), 然后读写 ②Interrupt: I/O准备好时触发, 暂定程序, 转至OS陷阱处理程序。

Term: Interrupt: 外部事件 (如按键), 异步, 可在任意指令处理

Exception: 执行中事件 (如page fault), 同步, 必在引发指令处处理。 Trap: 跳至trap handler以处理 Interrupt / Exception

处理流程: 中断/异常暂停指令流, 跳转至处理程序并保存状态。陷阱前指令完成, 后指令未执行, SPEC记录中断地址, 简化处理 (精确陷阱)。流水线处理: 异常标志至提交点 (M阶段), 早期异常覆盖晚期, 外部中断注入提交点。

进程(程序)管理: 时间共享CPU, 定时器中断触发 context switching; 用户进程请求OS服务 (如文件操作、进程创建), 其调用内容由内核执行; 同时防止程序互相覆盖内存, 虚拟内存提供每个进程独占地址空间

Virtual Memory: Process 用 virtual address, memory 用 physical address. Memory manager: Virtual → Physical

现象: Memory Fragmentation: 程序下去, storage会片段化

在层级内存管理中: DRAM ⟷(OS) Disk: V→P address 在硬件下协助实现 (TLB, a cache)

Paged Memory: DRAM与磁盘分割为固定大小的页面。通过分页机制从磁盘中加载到内存 (整个页面). 只存PPN+ Flags!

Page Table: 页表, 将 virtual 翻译为 physical 地址。表中每个元素对应一个 VPN (Virtual Page Number). 若对应一个page, 则那个VPN对应一个PPN (Physical Page Number)。若还未对应, 会让OS trigger page fault to load page from disk. VPN外, addr中还有offset, 直接copy, 与PPN结合为 Physical Addr. (PA)

△: Each User has a page table ☆

Setup*: 1个page 大小为 x, 则offset: $\log_2 x$ 位

若电脑N位, Disk RAM大小为y, 则 VPN: $N-\log_2 x$ 位

PPN: $(\log_2 y - \log_2 x)$ 位

More Detail in TB: TB中每一个元素对应一个VPN, 且含 status / protection bits. 同时: PT is not a cache! 且所有VPN都在PT中有一个位置

Demanding Page: Only load a page into memory if user requests it.

Status Bit: 访问时, 先检查PT该entry的 status bit. 若valid, 如以往; 若invalid, page fault exception。再若DRAM空间不够, evict, victim status bit → 0, 且写回 disk; 够后, 磁盘对应page写入DRAM, 且 VPN对应 PPN 更新, status bit 为 1 (valid)

☆: Page table 与 Disk copy 来的 Page 均在 main mem 中

Dirty Bit: 写入Disk, 采取 write-back 策略。Page被换时, 若Dirty Bit为1, 则把新内容写回 Disk 对应地方

每个进程都有独立的PT, 隔离其虚拟地址空间; PT限制访问范围。PT由OS管理。

Hierarchical (Multi-level) PT: P_1 (L_1 index), P_2 (L_2 index), offset

计算DRAM中地址: L_i层: L_i Base Addr + $p_i \times 4$ (决定)

(4是因为: 32位中, PT entry 大小常为 4 Byte)

TLB: Cache some 地址翻译 in TLB, 保存的是 VPN-PPN mapping. * TLB is much closer to CPU and caches. While TB只存PPN+Flags!

⚠⚠ 普通 Page Table中, 以4也是必要操作! (on DRAM)

(如算PT大小: $2^{\text{VPN-bit}} \times$ 4B/entry, 可见VPN是index非addr)

TLB是cache! 一个VPN先看 TLB是否Hit, 若miss 则 Page-Table Walk, 并以策略填写TLB (在PT更新后). TLB常 Fully Associativity, 策略有 FIFO, random 等

TLB Reach: How many VA可被立即翻译. #TLBentry × Page Size

Only one TLB per core, but page table are per process

- VA management 允许OS把程序load到任意 physical memory space.
- Single-core processor 无需维持cache-coherence.

Misc: More I/O : CPU与I/O设备同步方式 高开销

Polling: CPU不断查询; Interrupt, 设备通知CPU, 但频繁I/O↑

程序I/O: CPU用lw/sw处理所有数据移动. 两部分① 将数据从设备传输至DRAM ② 使用数据计算. 但CPU被占

直接内存访问(DMA): DMA允许I/O设备直接访问DRAM, 降CPU负担。DMA引擎含由CPU写入的register。传输过程: ①CPU接受设备中断, 初始化DMA引擎 ②它处理数据传输 CPU可执行其它任务 ③ 传输完成后, 再次中断CPU

优势: 释放CPU资源, 适合高数据率设备

常见I/O设备: SSD与HDD(传统硬盘)

Redundancy for Accuracy: 依Amdahl: 系统可靠性取决于最薄弱环节。冗余方式: 空间、时间、信息冗余

错误检验: 奇偶校验: 附加位使1个数为奇(奇校验)或偶(偶校验) ⇒ 单比特错误

Hamming ECC: Hamming距离是对位数不同的位个数。加额外校验区分单/双错误 ⇒ 1-bit纠错 2-bit验错

磁盘冗余: 优势: 提高数据可用性, 防止硬件问题

异构计算: 利用多种处理器或核心, 通过专用加速器提升能

FPGA: 现场可编程门阵列, 通过硬件描述语言实现特定逻辑 (从硬件电路下手!) CPU

★给定足够资源, FPGA可实现任意逻辑, 包括RISC-V

后三章核心: **DMA**: 除了上述之外: 访问冲突解决方式:

Burst: 传Data Block间, CPU无法访问内存

Cycle Stealing: DMA传1个Byte后释放控制权, 与CPU交错访问

Transparent Mode: DMA仅在CPU不使用总线时传输

Networking: 类型: shared: 一次一device; switched: 多对多同时

其software protocal: Send: Application data→OS buffer, OS算校验并启动定时器, Buffer Data→网络接口 ACK

Receive: 网络接口 Data→OS Buffer, 算校验和, if OK send; 否则丢弃等重传; Buffer Data→地址空间, 且通知应用

More Parallelism: Request-level: Web服务, 每个请求并行

Data level: SIMD, on WSC: MapReduce & Scalable filesys

Map Reduce: 用于large-scale data并行处理, 将Data处理分为Map与Reduce两个阶段

Map: 输入Data"分为Key-Value Pairs, 并行处理 中间

Reduce: 中间键值对聚合→拼KV 分KV 生成KV'

Map & Reduce中间有shuffle: 将Map结的中间xxx按键分组, 确保同key pair发送给同一个Reduce任务

Hamming ECC: 相信我的大脑已记住流程 (左→右).

RAID: 独立磁盘冗余阵列。RAID 0: 无冗余

RAID1: 镜像 RAID3: Parity Disk 现在! ← striped&

RAID4: Parity + Block level RAID5: 4上 + interleaved

AFR = $\frac{N\text{disks} \times 8760\ \text{hrs/year}}{MTTF \times N\text{disks}}$

MTBetween F = MTTF + MTTR

Availability = MTTF/(MTTF+MTTR).

Security: Heartbleed: OpenSSL漏洞, 可读内存缓冲区, 泄露密钥; Rowhammer: 针对DRAM漏洞, 通过物理干扰引发位翻转; Side-channel: 利用系统中物理行为(如时间、缓存访问、功耗、声音)泄露信息。Meltdown: 利用乱序执行漏洞, 允许程序读取不应访问的内存。Spectre: 利用推测执行和分支预测, 诱导处理器加载受害者数据至缓存, 再侧信道提取。

C中: 宏无类型; const与#define速度大致一样