Midterm 1 – CS 152

Lectures 1 + 2

ISA: types of instr supported by processor – ISA implemented by μarchitecture
↳ Designed for particular μarch. CISC ⇒ Microcoded RISC ⇒ hardwired, pipelined

Microcode: lowest/simplest level, technique to manage control unit

Datapath: where #s stored, arithm ops **Control**: sequences ops on data path

Loop misses (32 bit elems) Row major [i][j] A: i,j B: j,i
(ex) 8KB direct map, 4-word (4 32bit ints) $ lines (128 x 32)
A: 1024 compulsory (128 rows x 8 lines a row; 1 miss, then 3 hits)
B: 32 word stride, 128 lines go 0, 8, 16, 32 → 128x32 compulsory misses (64 sets fully percol)
(ex) 8KB 4 ways, 4-word – #lines = $2^{13}/2^4 \times 2^2 = 2^7 = 128$
B: 128/8 = 16 (cols) misses → way 1 in sets 0, 8, 16, 32 → fill → 128 x 32

ROM # bits (to reduce control size)
|μaddr| = |μPC| + |opcode| + 1 + 1
|data| = |μPC| + |control signals| (aka +18)
Total ROM size = (2 ** |μaddr|) x |data|
★ reduce ROM height (# addr bits); reduce # opcodes/states
★ reduce ROM width (# data bits); reduce μPC encod., encode cont. sigs

Horizontal μcode: wider (more bits) μinstr, mul parallel ops per instr; fewer microcode steps per macro (VS)
Vertical: (RISC) narrow (less) μinstr, more microsteps per macro, compact, single datapath op per μinstr

	μBr	Tricks
IRld = protect imm, rd, rs1, rs2	N	{ MA := rd; Mem[rd] := { mem := A
RegWr = never random, PC = A+1	J	To set return addr:
RegEn = 1 when (ex) ... := PC / RegSel: target	EZ	MA := A+B PC := Mem
Ald/Bld = 0 if using (ex) A := A-B; 1 else	NZ	**GPUs**: 2+ multithreaded SIMD
ALUEn = 1 if using ALU operation	D	cores; remap ops always
MAld = protect if MA := , 0 if doing := Mem	S	scatter-gather, coalesced into
MemWr = never random, Mem :=	↑ μem:= or :=Mem	cont. block transfers when possible
MemEn = 1 if := Mem, else 0		SIMT: indiv scalar instr streams for each thread (CUDA) grouped together 32 (CUDA threads) into warps
ImmSel = ImmI, U, S, J, B or *		

50 cycles, inner, L1 miss = 40, offset?
```
for (int j=0; j<M) {
  for (int i=0; i<N) {
    prefetch(&A[i][j] + OFFSET)
    sum += A[i][j]
}
```
① 50 – 40 = 10 time lost
② 40/10 = 4 iter (byte granularity)
③ 4 x 32 == 128 offset (int)

CPU →(VA) $ → TLB →(PA) mem
VA addressed $, TLB virtually addressed, Aliasing possible, faster $ hit, must flush $ on context switch

CPU →(VA) TLB →(phys translation) $ →(PA) mem
PA addressed $, TLB virtually addr., Aliasing not possible

Lecture 3

Iron law = instr/prog. × cycles/instr × time/cycle

Microarchitecture	CPI	cycle time
μcoded	>1	short
single-cycle unpipelined	1	long
pipelined (RISC)	1	short

(yellow) = src code, ISA, compiler, tech
(purple) = ISA & μarch
(blue) = μarch & base

Law	Increase	Decrease	No impact
Instr/program	RISC (more instr for complex), ISA visible, branch, prefetching instr, Adding VM, strip mining non-div, delay slot, ↳ TLB refills + OS exception handler	CISC, dynamic binary translation (cached)	Non-ISA visible: microcoding, br. predict, pipeling, L2 change, PT hierarchy/policy
CPI	Increase in PT hierarchy (walks), Adding VM, adding CISC / complex instr.	Good prefetching, pipeling, L2 added, delay slots, decrease usage of I$ space (compress nops)	
Time/cycle	wide registers (longer to read), wide μcode (slower ROM), Not-pipelined, L2, logic, SMT/precise excp. support	Pipelining, higher clock freq.	PT walk (TLB accessed in parallel), prefetching (not expensive), L2 (same times), pipeling if super deep
Total	PT hierarchy, VM, CISC heavy overhead	CPI decrease, DLP, L2, Deep PT hierarchy (slow TLB misses)	prefetch

pipelining F → D → X → M → W
μcoding: I1 I2 I3, 2 cycles; unpipelined: 3 cycles, 3 instr = 1 CPI (I1 I2 I3); pipel.: 3 cycles
Classic RISC 5 stage

→ **Structural**: sharing resources sol: add more hardware, stall newer instr, has no struct haz

RAW	WAR	WAW
r3 ← r1 op r4; r5 ← r3 op r4	r3 ← r1 op r2; r1 ← r4 op r5	r3 ← r1 op r2; r3 ← r6 op r7

Interlock, Bypass, Speculate

→ **Control**: branch/exp. for next instr addr ★ j(D), jr(X), beq etc (X)
↳ **Branch delay slot**: stuff another instr or nop → performance issues (complex), I$ misses
} Sol: branch prediction

Exceptions → **Exception:** unusual internal event caused by prog. during exec → **interrupt:** external event
→ **trap:** forced transfer of control caused by exc. or interrupt ☆ not all excep. cause traps
→ **Asynch interr:** I/O asserts on prioritized interrupt req lines, choose to process interr, stops at instr I_i, completing until I_{i-1} (**precise interrupt**), set EPC to whichever didn't w/commit
↳ **Trap handler:** save EPC before enabling interr, ERET (enable + rstrt) **Synch trap:** particular instr, usually restart

S-stage: hold exception flags until **commit point** (M stage, speculation best)
☆ only wr at commit, else throw away & flush & launch exp handler ☆ **Bypassing** allows use of uncommitted instr results by follow

F F D X M (W) — committed
F D X (M) — NOT comitted
F D X

☆ raise → wait for all to get to FD— F— ↙ then start | **Imprecise:** FDX M1 M2 W
X overpass M1 ... F D X M1 M2 W | Reduce latency: unroll, kill precisely

Latency: t taken for single op to start to finish **Bandwidth:** 1/latency rate at which ops can be performed **Occupancy:** t during which unit is blocked on op (struct haz)

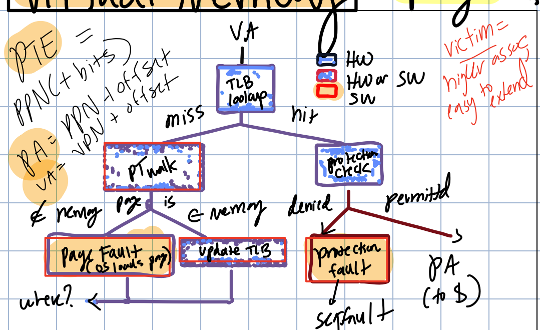
Memory **DRAM** (refresh; optimized for density) **SRAM** (optimized for speed), not destructive
CPU ⇔ SRAM ⇔ DRAM ☆ Temporal: for self, ref. again ☆ Spatial: near locations } predictable forms

$ Replacement ☆ **Random** (Fast) ☆ **LRU** (best for low assoc, complex) ☆ **FIFO** (high assoc) ☆ **NMRU** (FIFO except most recent)
$ Policies Cache hit ☆ **write-thru** ($ + mem) ☆ **write back** ($ only, eviction spawns wr) } wr thru + no wr alloc
Cache miss ☆ **No-write alloc** — wr to mem ☆ **write alloc** — fetch into $ (only to mem if evicted) } wr back + wr alloc

C's: **larger line size:** ↓ compulsory ↑ conflict **larger $ size:** ↓ capacity/conflict ↑ hit T **Higher assoc:** ↓ conflict ↑ HT
Write buffer: hold evicted wr before $, reduce read MP ☆ reduce wr penalties (not block) ☆ reduce read penalty cost of evictions **Misses per instr**
☆ **local miss r** = miss in $ / access to $ **global miss r** = miss in $ / CPU mem access **(MPI)** = misses / # instr

Prefetching **Accuracy** (useful / prefetches) **coverage** (useful / total mem. access) **Time** (on-time / total prf.)
Algos: **Next-line** (next N $ lines after comp. miss) **strided** (after seeing N w/ dist D → get next C+D) bt N, bt 2N

Virtual Memory **page:** fixed size, internal frag, efficient ☆ **segment:** variable, ext.



ⓔⓧ 4GiB VA, 4KiB pages, 4 byte PTEs ↳ $2^{PPN} \times$ page size = PA size addressable
↳ Page offset = $\log_2$(page size in bytes) = $\log_2(4096) = 12$
↳ VPN bits = Size of Mem addr − page offset bits = 32 − 12 = 20
↳ # virtual pages = $2^{\text{VPN bits}} = 2^{20}$ PTEs ☆ VPN = $2^{VA} / 2^{\text{page size}}$
↳ 1 page mapped to PA → 1 valid PTE
↳ linear page table → 2^{20} (PTEs) × 4B (size) = 4MiB sized PT
↳ 2 level: 1 page mapped to PA → 2 valid PTEs → 2^{10} PTEs × 4 } L1 vs. L2

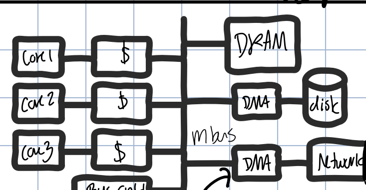
VA: 0x5|8 → 0x5 = 0101 → 01 = index 1 (64 bit words → 0x00 + 8) → go to 0x040 → apply 1/2/3 → PA
Final PA: Content + offset ⓔⓧ 0x17 content → 0x178 final ☆ Bring in pages for faults

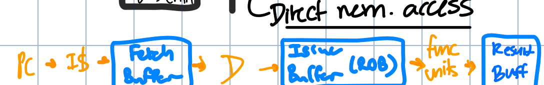
AMAT: **HT:** incr if crit path ↑ **MP:** increases w/ Block size / outer mem, **MR:** decr. w/ bigger tag
ⓔⓧ Matrix: Bits 0 to 11 offset + index → 2^{11} vs. +4096 bytes btw col → 2^{12} → same index

TLB: $ for PTEs, fully assoc ☆ TLB miss ≠ Page fault ☆ TLB lookup = tag check
TLB entry: valid / dirty bits, tag, PPN ⟶ PPN + offset = PA, use VPN for tag

VIVT: cache index & cache tag derived from VA **VIPT:** VA for index PA for tag → parallelism

Digits											
0000	0	0101	5	1010	A	2^2	4	2^8	256	Ki	10
0001	1	0110	6	1011	B	2^3	8	2^9	512	Mi	20
0010	2	0111	7	1100	C	2^4	16	2^{10}	1024	Gi	30
0011	3	1000	8	1101	D	2^5	32	2^{11}	2048	Ti	40
0100	4	1001	9	1110	E	2^6	64	2^{12}	4096	Pi	50
				1111	F	2^7	128	2^{13}	8192	Ei	60





TLB: Tag (VPN) ⟷ PPN

RISC instrs

lw rd, rs1	beq =	bgeu ≥
sw rs1 rs2	bne ≠	slt <
mv rd, rs1	blt <	slti < imm
	bge ≥	sltu < u
	bltu <	sltiu < imm u

Midterm 2 - CS152 - Katrina Sharonin - SID: 3036613337

OoO (ILP) Fetch: instr bits retrieved from I$ → Decode: instr dispatched to issue (window) buff → Dispatch: place into ROB, ready to issue (dealloc, issue, ld/st queue/speculate) → Issue: send to exec units → Completes: results & excp. available (ex) load → commit: "retirement" exit ROB (ex) store modifies state ☆ Always commit in order for precision!

RAW: r3 ← r1 op r2; r5 ← r3 op r4 WAR: r3 ← r1 op r2; r1 ← r4 op r5 WAW: r3 ← r1 op r2; r3 ← r6 op r2 ☆ WAW/WAR in OoO comp. unless in order issue

VS InO: issue stalls on RAW dependencies / struct / WAR/WAW hazards

OoO: dispatched in order to reservation station/instr buffer (issue window) → wait for ops → dispatch/issue out of order

OoO Design: reservation station (store instr that haven't issued) → in ROB vs. indep. (issue window) (Expensive to build)

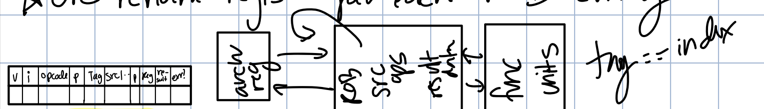
Renaming Design: Tags + data in reserve station VS Tags only in reserv. st., data in unified phys reg file

ROB: executing, not committed Rename Table: lookup ISA (architectural, x6) ↔ phys (p6)

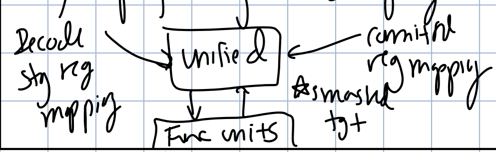
Register renaming ☆ Make WAR/WAW impossible ☆ sd, branches not renamed, still in ROB

Data-in-ROB • Circular (head = oldest) • p set on result
• Set v on dispatch • set i on issue
• Completion → search src tags → set p & copy data into src on tag match
• Commit → check Excp. → copy into arch. reg file if good • Trap/err → flush, set free = oldest, clean ROB

☆ One rename register per each ROB entry



Unified Phys Rename all arch registers into single phys reg file (temporary, commit status, wr on completion)



Rollback: go in reverse order ☆ expensive / inefficient since reverses a lot of work

superscalar renaming is not possible (2+ instr issued in parallel, RAW → "fear")

Rename

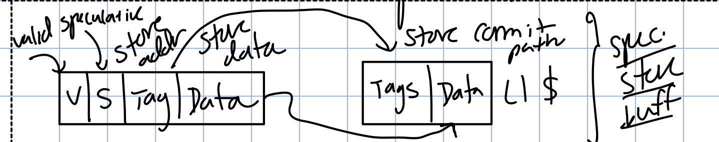
x0	p3
x1	~~p8~~ p0

ROB (args; put back old)

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
v	x	ld	p	p7			x1	p8	p0

Free

~~p0~~
p4



Speculative store buff / ld/st queue: allot in decode → don't know addr of store yet → hold speculative data. Commit when oldest instr & both addr/data available → clear S bit, move data to $, X if err

Load Bypass: | SSB ←pref.| Load addr → L1 $ | ☆ Use youngest sd that's older than load

VS In order Mem Queue: sd, ld: X in program order, ld/st can't leave ROB until older done X, still OoO to spec/others

VS Conservative OoO: sd, ld: don't execute load (can exec ld later) if any prev. st addr not known, cmp ld addr w/ all uncommitted strs

VS Addr spec: sd x1(x2) ld x3(x4): "aliasing" guess x4 != x2, X load before st addr known → hold uncommitted InO → flush if bad (☆ high penalty if bad guess)

(ex) If UK in SSB, override $. If not in SSB, use $, else UK (UK/UK could impact any)

VS (speculative) assume last known / in $ even if UK marked

Control Flow Penalty: needs to jump way back → sols: (SW) elim br / unroll, early cond. eval (HW) delay ex br, speculate, predict

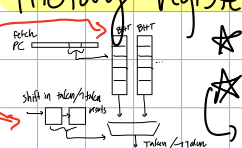
Branch Prediction Directional (dest addr = inspecting instr bits + PC) VS Indirectional (reads other reg. for dst ex jr x1)

Conditional (deterministic using input) VS Uncond. (return / go to) Forward (jumping) VS Backward (looping, 90%)

Temporal (the way the same br resolves indicates → BHT) VS Spatial (several br correlated → Global ctr)

Branch Hist Table (BHT): FSM to predict, update if correct/not, picked using lower bits of PC (so don't alias)

History register: records dir of last N branches (aka global history)



☆ Branch predict VS both dirs → not efficient → focus rsrcs on predicted dir

☆ Limits: BHT at D but must wait till X → penalty, if did other instr & now has to flush

Solution: BTB! BTB in F → inserts addr for prediction, BHT at D + use BTB value

Branch Target Buff (BTB): Entry PC (tag), Valid, Target predicted PC. If Tag = match → use, else PC+4

tag	V	Target PC
0x2008	1	0x2000

VS helps with cond., direct, indirect (☆ expensive entries) ☆ only taken branches & jumps held in BTB, next PC calc before br F/D

Jump register (JR): (VS BTB) reg designated where program should cont, calcs location, good for:

Switch statements (indir) VS BTB (if same case used N+) Dynamic func call (runtime addr) VS. BTB (same func called ex virtual func) Subroutine rets (ret addr) VS BTB if ret to same place (same func)

SubRoutine Return Stack / Return Addr stack (RAS): store retr. addr as called, accelerate jr subr. rets more acc than BTB

ex: fa() { fb(); } fb() { fc(); } → [&fc() / &fb()] → ① Lookup BTB for jmp ② push ret. addr ③ return, pop stack ④ jmp back to addr

InO: no instr issued after Br can WB before branch resolves VS OoO can complete OoO but cannot commit

VLIW (V. long instr Word)

force compiler to schedule so that HW doesn't worry abt interlocks

VLIW = N+ **Bundles**, each is **a type specific slot** (ex: 2 muls, 1 add) ★ **Guarantee intra-instruct. parallelism**

Loop unrolling: mul iter in one big one, faster stride, a lot of overhead

SW pipelining: same # iter, only 1 time overhead, modify instr to hide latency

★ write out two iterations: 1st enters, 2nd starts in loop, then last is covered w/o loop

→ Best to introduce if MEM in loop: is not fully utilized (spaces exist)

VLIW Issues: obj-code compatibility, wasteful size, unpredictable latency is dangerous to it (ex. br, $)

→ Compilation techniques: static schedule (fixed/no prediction), SW ppa, unroll, trace scheduling

Predicated execution: Mispredicted br limit ILP → eliminate hard-predict br via "if-else encoded into instr"

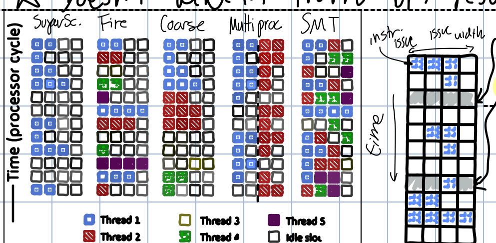
Instr conditionally execute; avoid var. len latency; IF cond → do this else → NOP

Trace Scheduling / Trace: path of blocks for most taken br path, schedule whole trace at once

Multithreading

(TLP) Incr. Thruput (more instr sent) Worsen latency (competing concurrently for rsrcs)

★ Doesn't benefit from br. resolution latency (can hide); arch. identical to multicore system to ISA



Vertical waste: no instr issued in single cycle (Reduce w/ OoO via ILP, InO stalls)

Horizontal waste: not all exec units in use in cycle ★Constrained by # phys reg

Fine-grain: every cycle, ✓hide thruput losses for stalls, Slows execution of indiv thr.

Coarse: when blocked, Doesn't need low overhead switch, reduce penalty of high cost stalls

MultiP/CMP: Split thr into its own core, reduce hor. limits upper thruput

SMT: smart interleave, full thruput. ROB size const, storage req. not changed, low cost to add (✓hardware, x needs multi-issue processor)

i-count policy: for SMT, fetch thread that has least instr in flight (if stalling/in flight, will get stuck again...)

Sharing Resources vs. Dup: (Duplicated) PC, Rename, Arch File Reg. (Shared) Fetch unit, Phys Regfile, issue window, Func units, ROB

loop: ld 3 $a, A($i)	+1	Finegrain	Data-Dep	loop: addi a1 a1 -0x1	3N+1	
1 ld 4 $b, B($i)	N+1	① Find max	① Ellipse	4N+1 lw a0 0(a0)	+1	50 cycles
2 ld 5 $y, Y($i)	2N+1	② (5N+1)-(N+1) ≥ 50	② steady = max(: group)	brez a1 loop	2N+1	
3 addi 6 $i, $i, 8	3N+1	③ Find N	③ $ceil(\frac{50-4}{7})+1$	Fine grain: +1 & 4N+1 same time		
4 addi 7 $n, $n, -1	4N+1			Data Dep: $ceil(\frac{50-3}{3})+1$		
... (+50)				since circles 'round!		
fmul $t, $a, $b (+5)	5N+1					
...						
fadd $s, $t, $y (+5)	6N+1					
...						
1 sd $s, S($i)	7N+1					
2 bnez $n, loop	8N+1					

Vector ISA

vsetvli t0, a0, e8, m2, ta, ma (config vlen & vtype)	a0 = N, e8: SEW=8, m2: LMUL=2
vlse32.v vsd (rs1) rs2, vm (32 bit strided load)	rs2 (stride), vm (mask)
vmslt v0, v2, x0 (set if less than (signed), v0.t = mask	v0: set mask
vluxei32.v v2 (a1) v1 (load a[idx[i]], use elems of v1 to index into a1 → stuff v2	
vmul.vx v2 v2 a4 v0.t (Scale * a[idx[i]] IF cond. is true)	
vsse32.v v2 (a2) a5 v0.t (Store into b[dilation * i]	

Vectors (DLP)

ISA separation from march

Compact, Scalable, expressive (vs) **Packed SIMD:** fixed vector len, limited instr set, no strided lds/st/scatter/gath, must align, req. sync st. dispatch to stay busy, loop unrolling ↑ reg pressure

```
for (unsigned int i=0, i< N, i++) {
    if ( a[idx[i]] < 0) {
        b[dilation * i] = Scale * a[idx[i]];
}}
```

Note: a0 = n, *a = a1, *b = a2, *idx = a3, scale = a4, dilat = a5

	slli a5, a5, 2	dilate to bytes
loop:	vsetvli t0, a0, e32, m1, ta, mu	
	vle32.v v1 (a3)	load idx[i]
	vsll.vi v1 v1 2	convert to byte offsets
	vluxei32.v v2 (a1) v1	load a[idx[i]]
cond	vmslt.vx v0 v2 x0	set mask (a[idx[i]]<0)
	vmul.vx v2 v2 a4 v0.t	Scale * a[idx[i]]
	vsse32.v v2 (a2) a5 v0.t	Store b[dilation * i]
	sub a0 a0 t0	decrement n
	mul t1 t0 a5	vl * dilation in bytes
	slli t0 t0 2	vl to bytes
	add a3 a3 t0	bump ptr to idx
	add a2 a2 t1	bump ptr to b
	brez a0, loop	
end:	ret	$VLMAX = LMUL * (\frac{VLEN}{SEW})$

Vectors Cont.

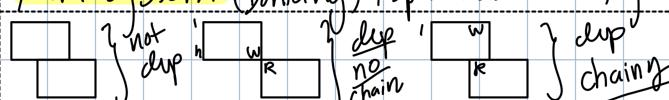
Vector len register = # valid opt, MAXVL / VLMAX

Standard elem width (SEW): default width of each vec elem width

Len Multiplier (LMUL): int pow of 2, grouping consecutive vectors ex: =2 vn & vn+1 operated tgt

VL = # lanes → 32 elem vector, 8 lanes → 4 vectors issued per instr

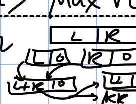
Mem system: (Banking) split address out, go to sep. bank w/ high probability



Chaining: Bypass vals early; two Rs but separate func units

Strip-mining: finite V size → split across mul Vs; Best when iter len > MaxVL

Vector reduction: BinTree reduction, recursive aggregation



sum += A[i] → sum = sum + A[i] depends A[i-1] + A[i-2]...

SIMD (Vector) **MIMD** (thrd) **SISD** (OoO) | GPU: SIMT (multi thr) = DLP x TLP

Thread blocks: range over iteration space **Warp:** sub blocks not visible to user, microarch usually together

VA | PA translated addr vs. using VA to address data in $ → Mul VA to single PA (**Aliasing**: sim w/reads)

VIPT: offset bits = virtual index → P tag & check ★ if issue → incr. assoc (less index) / incr. page size ★ Page offset == offset + index bits, else L2 to check

Final Cheat Sheet

$ coherence | **Coherence**: legal values a single mem addr can return ★transient if not atomic | **MESI** to reduce broadcast

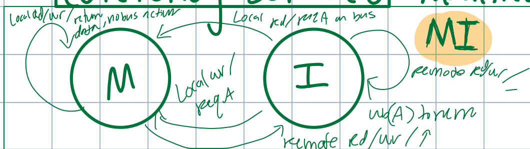
Write policies: wr-back: only $, mem on flush, **write thru**: main mem

Wr Miss policies: **No-write-alloc**: only to main mem wr, no $ wr.
Write-alloc: fetch into $, avoid wr to DRAM unless evicted

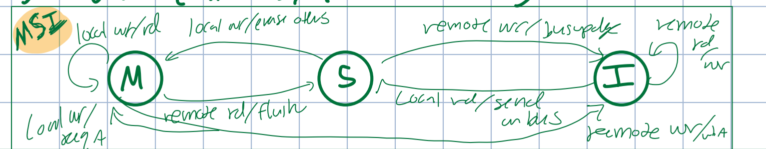
Inclusive / Exclusive: **Inclusive** (L1 == L2), **Exclusive** (L1 != L2)

if local wr in S state (we might be the only cores) ↕ vs.

Coherency schemes Modified (I own, not shared) → Owned (shared), exclusive (clean)



} causes threads to invalidate eachother



Sharing status: **Dir based**: block of mem (centralized - SymMP) (distributed: SGI, diff nodes)
Snooping: $ line, monitor bus; needs totally ordered broadcast network, can go w/o wire bus

Memory access: **producer-consumer**: wait, use bit to indicate → OoO may give stale data
Mutual exclusion: one process at a time, use lock

new ⓔⓧ coherence issue
old 3: lw f0, (a0) | 2: sw f0, (a0)
1: lw f1, (a1)

a0 = a1 = A
1, 2, 3
old new

Preserve program order: R after W should be up to date
Write serialization: 2 W to same loc. by any 2 Ps are globally seen in O by all processors

Eventuality: read by processor from location X that follows a wr by another processor to X returns Wr if:
① Read & wr sufficiently separated in time ② No other wr occur b/w R & W to X

Full bit vector scheme ★Bad scaling: each dir entry has state of cache line & bit vector w/ one share bit per p
4 dir states → dir bits per cache line? ⓔⓧ 128 cores w/ priv $ → 2 bit ($2^2=4$), 128 bits (bit_{vec}) + 2

Hierarchical bit vector scheme: aggr. cores into grp → rep as single bit, invalids sent to all in grp
ⓔⓧ 1024 core 64 byte cache lines, # cores / group to reduce dir stor by 10% of phys mem?
$2 + 1024/N \leq 64 \times 8/10 \rightarrow 2 + 1024/N \leq 512/10 \rightarrow N \geq 21 \rightarrow$ at least 21 cores

★ Must store dir bits for every line in mem even if not $d → hierarchical dir struct.
↳ or add **last level cache (LLC)**: stores only recently asked; inclusive of smaller $

False sharing: 2+ thrds modify data that happens to reside on same $ line; inefficiencies since each thr. may invalidate $ line for other thread
ⓔⓧ sharing $ line w/ single bit; even if val didn't change its force to be I

Mem consistency across all mem addr; even if no caches, important for Multi thr.

Sequential Consistency (SC): order-preserving P1: A, B P2: C, D → BADC not allowed
★ Most real time not SC → slower, needs complex HW, needs TSO to work

Store buffer optimization: CPU → St buff → Shared mem; later loads can go ahead of st if diff

Total store ordering (TSO): Relaxes W → R (R can be visible before W)
★ allow use of store buff / sd → sd could take ∞ time

Weak - multicopy atomic mem: all processors see writes by another processor in same order (relaxes everything outside but internally everything is in order) "pt of visibility"
... NON-multicopy ...: write can be partially visible to all cores (intermediate buffer) AKA hierachial shared buffering, relax all, maintain inner order

Weak	Simpler to achieve high performance, easy HW, expose HW to SW	**Strong**	easy on ISA, high complexity

Fences

producer → flag / data → consumer

sw xdata, (xdatap)
li xflag, 1
fence r/r
sw xflag, (xflagp)

spin: lw xflag, (xflagp)
beqz xflag, spin
fence r/r
lw xdata, (xdatap)

Relaxed mem models:

X → Y: X must be visible before Y is visible

R → R, R → W, W → R, W → W

Release consistency: relax all four

★ compilers can reorder → bad performance

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
clean Exclusive	no	none	none	CE	yes		yes
★ CPU wr/rd = self processor issues, write hit/miss		CPU read	none	CE	yes		yes
		CPU write	none	OE	yes		
		replace	none	I			yes
	bus signals	CR	none or CCI'	CS	yes	yes	yes
		CRI	none or CCI'	I		yes	
		CI	none		impossible (if shared		
		WR	none		impossible maybe...)		
		CWI	none	I			yes

ID	Event	Message Shared	Cache0 State	Cache1 State	Cache2 State	Main Memory Up-to-Date?
0	CPU0: read A	0:CR	CE	I	I	Yes
1	CPU2: write B	2:CRI	I	I	OE	No
2	CPU1: read B	1:CR; 2:CCI	I	CS	OS	No
3	CPU1: write B	1:CI	I	OE	I	No
4	CPU2: write A	2:CRI; 1:CCI	I	I	OE	No
5	CPU0: write B	0:CRI; 2:CCI	OE	I	I	No

Guest lectures

Apple

- Branch predictions
- Cost of mispredict grows w/ deeper pipes
- Static (simplest)
  - ↳ backward (loops) usually taken
  - ↳ fwd is more unknown
- Dynamic (history)
  - ↳ Bimodal predictor: FSM indexed w/ PC
- predictor aliasing: 2 diff branches w/ diff PC & diff bias share same predictor index
  - ↳ Agree predictor: BTB + tags
  - ↳ Gskewed: odd # tables; majority vote
  - ↳ Perceptron (ML)

(#1 is TAGE CBP)

- next line
- Instr stride (use PC to index into PF Table)
- Irregular
  - ↳ correlation
  - ↳ content-directed
  - ↳ execution-based
- Hybrid: use mul. PF to cover patterns, but complex/BW intensive
- Feedback: throttle PF using performance; change loc of $ insert

Prefetching

occupancy = latency / thruput

Thruput = window size / latency

WSC

- WSC: single org, homog, common layers/SW platforms, in-house services, renting VMs, request-level parallelism
- Parts: server, racks, Top of R networking switch, custom Accelerators (demand for ML): GPUs, TPUs, FPGAs, VCU (youtube)
- Networking: hard to fix biBW
  - ↳ N ports for N blocks vs. fat tree, Clos @ Jupiter
- DC fabric → rack switch → DRAM
- Power: transformers, UPS, PDUs
- Uninterruptible Power Sys (UPS): generator, renewable sys / distant
- WSC cooling: hierarchy of loops ★fresh air best
  - Open loop: new medium replace.
  - Close loop: re-circulate, transfer

Efficiency = compute / total energy = $\left(\frac{1}{PUE}\right) \times \left(\frac{1}{SPUE}\right) \times \left(\frac{\text{Comp}}{\text{total Energy to Comp}}\right)$

PUE = facility power / IT equip power

SPUE = server power conversion efficiency

(c) server arch eff. (a) facility efficiency

- Fault tolerance:
  - ↳ Power/cooling: hierarchy, scheduler
  - ↳ Net.: redundancy, high BW
  - ↳ Scheduling: replication, dist

AWS

- Nitro: more cloud server aka virtualize
  - ↳ server CPU not shared, no limit on inst. size, no tradeoff of network/storage/custom instances
- Scale I/O workload w/ high durability & low-latency
- Live migration
- Security
- Scalability (decouple VA functions from HW), easier mngment