CS 152 Cheat Sheet — Jeffrey Shen

1 2 Early/Simple Machines

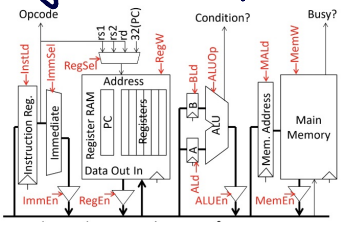
Computer Architecture: design of the abstraction layers to implement info processing apps

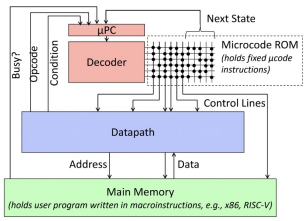
Instruction Set Architecture: contract between software & hardware, giving programmer visible state

3 Microcoding

Microcoding: technique to manage control unit

Single-Bus Datapath for Microcoded RISC-V





Instruction Fetch:
MA,A:=PC
PC:=A+4
wait for memory
IR:=Mem
dispatch on opcode

ALU:
A:=Reg[rs1]
B:=Reg[rs2]
Reg[rd]:=ALUOp(A,B)
goto instruction fetch

ALUI:
A:=Reg[rs1]
B:=Imml //Sign-extend 12b immediate
Reg[rd]:=ALUOp(A,B)
goto instruction fetch

LW:
A:=Reg[rs1]
B:=Imml //Sign-extend 12b immediate
MA:=A+B
wait for memory
Reg[rd]:=Mem
goto instruction fetch

JAL:
Reg[rd]:=A // Store return address (A:=PC from fetch)
A:=A-4 // Recover PC (PC incremented in fetch)
B:=Immj // Jump-style immediate
PC:=A+B // (Alternative: PC:=A+B-4)
goto instruction fetch

Branch:
A:=Reg[rs1]
B:=Reg[rs2]
if (!ALUOp(A,B)) *goto instruction fetch* //Not taken
A:=PC //Microcode fall through if branch taken
A:=A-4
B:=Immb// Branch-style immediate
PC:=A+B
goto instruction fetch

μPC jump = next | spin | fetch | dispatch | fcnx | fcnx

Horizontal	Vertical μCode
-fewer microcode steps	-single datapath op per μinstr
-sparser encoding → more bits	- more compact → less bits

4 5 Pipelining

Iron Law: $\frac{\text{time}}{\text{program}} = \frac{\text{instr}}{\text{program}} * \frac{\text{cycles}}{\text{instr}} * \frac{\text{time}}{\text{cycle}}$

Fetch F → Decode D → Execute X → Memory M → Writeback W

Structural Hazards: needs a resource being used by another instruction in pipeline
↳ Fix: adding more hardware

Data Hazard: depends on data from prev instruction
- RAW: Read after Write, data-dependance
- WAR: Write after Read, anti-dependence
- WAW: Write after Write, output dependence

↳ Fix: **Interlock**: wait by holding next instr in issue stage
↳ Fix: **Bypass**: Resolve hazard earlier by bypassing when available
↳ Fix: **Speculate**: guess on value, correct if wrong

Control Hazard: depends on branch/exception for next instr address
- **Branch Delay Slots**: next instr after branch/jump is always executed before control flow change
  ↓ complicates microarchitectures, removed in 1990s
  ↳ Fix: **Branch Prediction**: start executing next instr based on predictions

CPI: measure cycles from first instr start to last instr finish
Full bypassing may be too expensive to implement

Exception: unusual internal event caused by program during execution

Interrupt: external event outside of running program

Traps: Forced transfer of control to supervisor caused by exception or interrupt

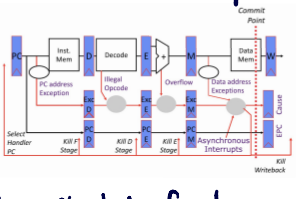
Asynchronous Interrupts: I/O device requests attention by asserting one of prioritized interrupt request lines

Trap Handler: saves EPC before enabling interrupts to allow nested interrupts

Synchronous Trap: caused by exception on particular instr, must be restarted

Hold exceptions in pipeline until commit pt, earlier overrides later exception

- Pipeline hazards avoided through software techniques: scheduling, loop unrolling
- Delay writebacks so all operations have same latency to write



6 7 Memory

CPU ↔ Small, Fast Memory RF, SRAM ↔ Big, Slow Memory (DRAM)

Latency: time taken for a single operation start to finish, memory access ≫ processor cycle time

Bandwidth: rate of which operations can be performed

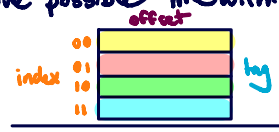
Occupancy: time during which unit blocked on a operation

In-order Superscalar Pipeline: fetch multiple instr per cycle

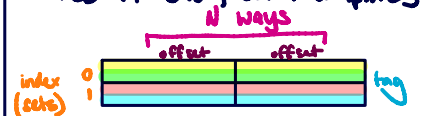
1) **Temporal Locality**: if referenced recently, likely to be referenced again soon
2) **Spatial Locality**: if referenced nearby, other close locations likely

Caches

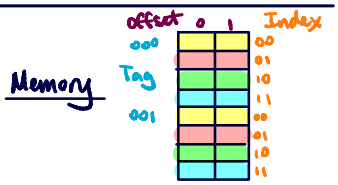
Direct Mapped
Each memory addr associated w/ one possible line within cache



N-Way Associative
Each set is fully associative w/ N lines in each set
N compares



Fully Associative
Line can go in any row, 1 set



Tag	Index	Offset
Distinguish diff lines w/ same index	specifies set	location of byte in the line
Bits: Address bits − Index bits − offset bits	Bits: $\log_2$(# sets)	Bits: $\log_2$(line size (bytes))
	# sets = # lines / N	
	# lines = Cache size / line size	

Replacement Policy

1) **Random**: fast
2) **Least Recently Used (LRU)**: complex, must be updated every access
3) **First-In, First-Out (FIFO)**: highly associative caches
4) **Not Most Recently Used (NMRU)**: FIFO w/ exception for most recently used line

Average Memory Access Time (AMAT):
hit time + miss rate * miss penalty
to improve performance, reduce any of the three

Cache Misses:

1) **Compulsory**: first reference to a line, misses even w/ infinite cache
2) **Capacity**: cache too small, miss would occur w/ perfect replacement policy
3) **Conflict**: miss occurs be of collisions w/ line placement, would not occur w/ full associativity

Larger cache size: + ↓ capacity, conflict misses − hit time↑
Higher associativity: + ↓ conflict misses, − increases hit time

Larger Line size: + ↓ compulsory misses − ↑ conflict, miss penalty less tag overhead

Write Policy

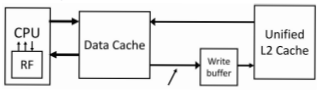
① Write Through: write both cache & memory

① Write Back: write cache only, memory written when entry evicted

② no-write-allocate: only write main memory on miss

② Write allocate: fetch into cache on miss

↳ Write Buffer: hold updated value of location needed by read miss



↳ Sub-blocks (sector cache): valid bit added to units smaller than full line, read sub block on a miss

↳ Multilevel Cache: increasing sizes of cache at each level

local miss rate: miss in cache / access to cache

Global miss rate: miss in cache / CPU memory accesses

Misses per instruction (MPI): misses in cache / # of instr

Inclusive multilevel cache: inner cache only lines also in outer

⑧ Prefetching

Prefetching: speculate on future instruction and data accesses and fetch into cache

Usefulness: should produce hits

Timeliness: not late and not early

1) Prefetch on miss: prefetch b+1 on b miss

2) One Block Lookahead (OBL) Scheme: prefetch b+1 for b access

3) Strided Prefetch: follow sequence b+N, b+2N, ...

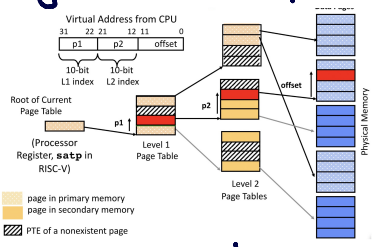
⑨ Address Translation

Base-and-Bound: base register and bound as boundaries

- External Fragmentation: non contiguous memory usage

Paged Memory System: program generated address split into page num and offset, can store large contiguous virtual memory space using non-contiguous physical memory pages

+ process memory grows and shrinks dynamically



- Internal Fragmentation: not all bytes on page used

Hierarchical Page Tables: multiple levels, allocate when needed

Translation Lookaside Buffers (TLB): cache translations typically fully associative, random or FIFO

⑩ Virtual Memory

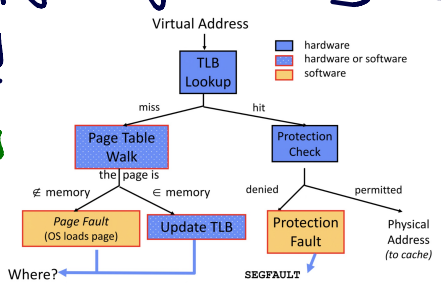
Protection & Privacy: several users w/ private address space & one or more shared address spaces

Demand Paging: can run programs larger than primary memory

Page faults handled by OS since it takes a while

Virtual Index / Virtual Tag (VIVT): cache before TLB, translate on miss

-Prevent aliasing in cache by direct mapped



Exam

Microcode

When M[x]:

Loop

Square

PseudoCode	IdIR	Reg Sel	Reg Wr	en Reg	Id A	Id B	ALUOp	en ALU	Id MA	Mem Wr	en Mem	Im m Sel	en Imm	Br	Next State
MA <- R[rs1]	0	rs1	0	1	*	*	*	0	1	0	0	*	0	N	*
A<-Mem	0	*	0	0	1	*	*	0	0	0	1	*	0	S	
if (A < B) goto DONE MA <- R[rd]	0	rd	0	1	0	0	SLTU	0	1	0	0	*	0	N Z	DONE
Mem<-A	*	*	0	0	0	*	COPY_A	1	0	1	0	*	0	S	
A, B <- R[rs1]	0	rs1	0	1	1	1	*	0	*	0	0	*	0	N	
R[rd] <- A	0	rd	1	0	*	0	COPY_A	1	*	0	0	*	0	N	
A <- A-1	0	*	0	0	1	0	DEC_A_1	1	*	0	0	*	0	N	
if (A==0) goto FETCH0 R[rs2] <- A	0	rs2	1	0	*	0	COPY_A	1	*	0	0	*	0	EZ	FETCH0

- always end on FETCH0

Memory

$$\#\text{virtual pages} = \frac{2^{\text{virtual address bits}}}{\text{page bytes}}$$

- max physical memory = PPN bits * page

- Page table entry is indexed by VPN,

- Stores PPN and has memory address

- Physical memory found by PPN + offset

In fully Bypassed CPU:

ADD → Store: store waits for updated accumulator

Digits

Decimal	Hex	Binary
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

8	7	6	5	4	3	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

	compulsory	conflict	capacity	hit time	miss rate	miss penalty
halve line size (assoc + #sets constant) → 1/2 cap	↑: shorter lines = fewer adj elems brought in w first access	↑: program accesses more lines in total → more misses	↑: capacity halved	↓: smaller cache better than small increased tag check	↑: smaller cap, + more comp misses	↓: smaller lines brought in faster
double #sets (capacity + line size constant) → halves assoc	—: assoc no impact on comp miss	↑: lower assoc = conflict misses ↑ and fewer places to put same element	—: capacity does not change	↓: #sets ↑ ⇒ tags smaller; fewer tags checked, fewer ways muxed out	↑: conflict misses	—: dominated by outer mem hierarchy
add good prefetching	↓: good prefetcher brings data before we need it	↓: prefetcher brings evicted lines back into $	↓: prefetch lines "just in time" ⇒ ↓ cap miss	no effect: prefetched not on hit path	↓: reduces miss rate	—: BUT may increase due to pollution
combine I$, D$ → L1$ w combined cap (assoc + line size constant)	—	may ↑: conflicts bt data + inst lines are introduced	↓: greater cap	↑: dual-port $ slower than single port OR can freq stall	may ↓: $ more flexible OR may ↑: edge cases↑ bt inst + data access	—: dominated by outer mem hierarchy

	IPC	CPI	seconds/cycle	execution time
add a branch delay slot	↑: must insert NOPs when the branch delay slot cannot be usefully filled	↓: some control hazards eliminated, also NOPs execute quickly	—: will not meaningfully change pipeline; or decrease: no branch kill	??: depends on how often delay slot can be filled w useful work
add a complex inst	↓: inst can condense a sequence of insts	↑: can mean more stages, or more control logic	↑: more control logic + interlocks → incr crit path (also, no effect)	??: only if new inst will be taken advantage of
reduce #regs in ISA	↑: values frequently spilled to stack; need to swap	↑: loads followed by dep. insts will cause more stalls	↓: fewer registers = shorter reg file access time	??: if program uses few regs
improve mem access speed	—	↓: less stall time for memory	↓: if mem is on crit path or mem is 1 cycle	↓
add 16-bit vers of common RISC-V insts	—	↓: code size shrunk → fewer $ misses + less waiting fetch time	↑: decode complexity incr	??: main adv=smaller code size; main disadv=complex decode
for CISC, μarch impl: μcoded engine → RISC pipeline + a decoder	—	↓: μcode engine needs mult clock cycles/inst; risc ~ 1	—: work done in 1 pipeline stage = 1 μcode cycle	↓: decrease in CPI from pipeline is huge
using wider μcode in a μcoded machine	—: μcode not visible @ ISA level	↓: fewer microinst needed to implement ISA inst	↑: sparse encoding = larger ROM = slower access	↓: wide μcode parallelism better than cycle time impact
pipelining μcode engine in a μcode machine	—	↓: pipeline is better than bus-based	—: single bus is fast, but so is deep pipeline	↓: inst throughput increases
add an L2 cache between L1 cache and DRAM	—: L2 $ not visible at ISA	↓: L2 $ improves AMAT for L1 miss	↑: larger SRAM = long c2q	↓: reduces L1 miss latency
add virtual memory	↑: page faults need extra inst for OS handler; SW TLB need refills on TLB miss	↑: inst fetch + mem ops use extra cycles for PT walks, but ok w TLB	—: TLB accessed in || w VIPT $; should not impact crit path	↑: impact when working w apps (size > TLB reach) and not all pgs in physical mem

Midterm 2 Cheatsheet

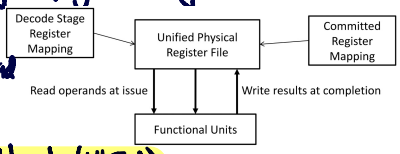
11 Complex Pipelines

Dispatch: instr decoded and enters ROB, always in-order
Issue Stage: holds instr waiting to issue for execution
#Registers limit instructions in pipeline
Register renaming: in decode, add to Reorder Buffer (ROB)
Completion: kept in ROB until commit for precise exceptions
- Limited by WAW, WAR

12 13 Out-of-Order Execution

In-order Issue: stalls on RAW, WAR/WAW, cannot issue unless all preceding instr issued to execution units
Out-of-order Issue: dispatched in order to reservation stations, following instr can be issued out of order
- data & tag in reservation stations vs. tags only in reservations, data in Unified Physical Register file
  - if "p" set, use value in register file

Reorder Buffer: active instr, decoded but not committed
Unified Physical Register: single physical register file during decode, tags in ROB
Address Speculation: execute ahead of time



14 Very Long Instruction Word (VLIW)

VLIW: multiple operations packed into one instr
parallelism within an instr, avoid data hazards
Trace Scheduling: trace through most frequent branch using whole trace, and use profiling feedback
↓ Object code compatability & obj code size
↓ Scheduling variable latency mem op
Static Scheduling vs Dynamic Scheduling instructions

15 Branch Prediction

Dynamic Branch Prediction: based on past behavior, temporal or spatial correlation
Branch History Table (BHT): change prediction after consecutive mistakes
Branch Target Buffer (BTB): keep branch PC, target PC in the BTB, only taken branches jumps in BTB
  - can redirect fetches earlier

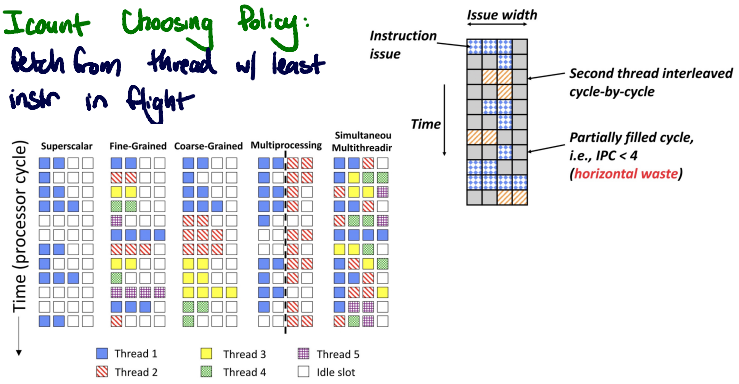
Jump Register: Switch statements, jump to addrers, func, ra
In order Br Pred: branch resolves before later instr complete
Out of order Br Pred: branch resolved after later instr complete
Speculative Store Buffer: store speculative memory accesses

16 Multithreading

Thread Level Parallelism (TLP): many workloads can use TLP, hard to get more optimization from ILP
- Multiprogramming: run independent sequential jobs
- Multithreaded: run one job faster using parallel threads
  - each thread requires PC, GPR, system state

Thread Scheduling Polices
1) Fixed Interleave N threads 1 instr every N cycles
2) Software Controlled interleave: S pipeline slots for N threads
3) Hardware Controlled thread scheduling: track ready threads

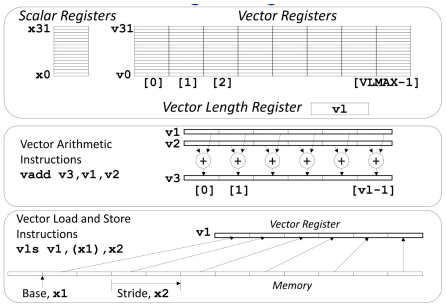
Coarse-gain: can run more cycles
Simultaneous Multithreading (SMT): instr from multiple threads to enter execution on same clock cycle
Icount Choosing Policy: fetch from thread w/ least instr in flight



17 Vectors

Vectors: One instr encodes N operations - independent, same functional unit, access contiguous block of memory, scalable Data level Parallelism (DLP)
Vector Startup:
- functional unit latency: time through pipeline
- dead time/recovery time: time before another vector instr can start down pipeline

Vector Chaining: allows bypassing, dependent instr as soon as first result appears
Vector Stripmining: break loops into pieces that fit in registers
Vector Mask: for conditional execution in vectors
Packed SIMD: limited instr set, limited vector register length

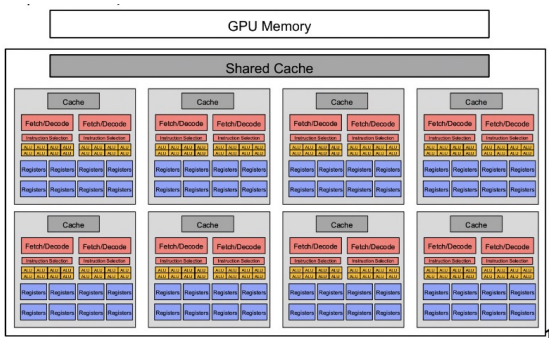


18 GPUs

GPU computational performance for general purpose computing

- Consists of multiple multi-thread SIMD cores inOrder
- CPU sends grid to GPU which distributes thread blocks among cores

Single Instruction, Multiple Thread (SIMT): individual scalar instruction streams for each CUDA thread grouped together 32 CUDA threads into warp



Use a mask vector for taken-not taken branches

Multithreading

Fixed Round Robin Scheduling

1. Find longest latency between instr
2. Label 1, N+1, 2N+1
3. last - first ≥ latency

Data dependent scheduling

steady state: max # instr thread can execute switching off

1. Find steady state
2. Find latency & subtract
3. (steady state) * N ≥ latency - (instr between longest latency)
4. N+1

Exam

ROB

Enter ROB, Issue, WB, Commit: sequential, one at a time
Issue + WB at same time
Commit must be in order
ROB sequential
Pay attention to ROB entries

Branching

A	j=0	j=1	j=2	j=3
Counter	00	01	10	11
Predict	0	0	1	1
Actual	1	1	1	1

Software Pipelining

label	ALU	FPU	FPU	MEM
	addi x1			fld f2
	addi x1	fadd f0	fmul f3	fld f2
loop:	addi x1	fadd f0	fmul f3	fld f2
	bne loop	fadd f1		
		fadd f0	fmul f3	
		fadd f1		
		fadd f1		

Final Cheatsheet

19 RISC-V Vectors

Vector Type Register (vtype): 32 vector data registers of size VLEN

`vsetvli rd, rs1, e8` ← vtype parameters (SEW, LMUL, VTA, UMA)

- rd: resulting machine vector length setting
- rs1: requested application vector length (AVL)

VLMAX = LMUL * VLEN / SEW

```
  # void *memcpy(void* dest, const void* src, size_t n)
  # a0=dest, a1=src, a2=n
  #
memcpy:
    mv a3, a0 # Copy destination
loop:
  vsetvli t0, a2, e8,m8,ta,ma   # Vectors of 8b
  vle8.v v0, (a1)               # Load bytes
    add a1, a1, t0              # Bump pointer
    sub a2, a2, t0              # Decrement count
  vse8.v v0, (a3)               # Store bytes
    add a3, a3, t0              # Bump pointer
    bnez a2, loop               # Any more?
    ret                         # Return
```

Set configuration, calculate vector strip length

Unit-stride vector load elements (bytes)

Unit-stride vector store elements (bytes)

Vector Unit-Stride Loads/Stores

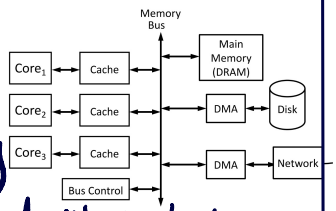
```
# vd destination, rs1 base address, vm is mask encoding
vle8.v    vd, (rs1), vm  #    8-bit unit-stride load
vle16.v   vd, (rs1), vm  #   16-bit unit-stride load
vle32.v   vd, (rs1), vm  #   32-bit unit-stride load
vle64.v   vd, (rs1), vm  #   64-bit unit-stride load
```

Vector Length Multiplier (LMUL): longer vector registers, vector register groups

Masking: all operations can be under a mask

21 Cache Coherence

Shared-Memory Multiprocessor:
- Multiple private caches for performance
- illusion of single shared memory



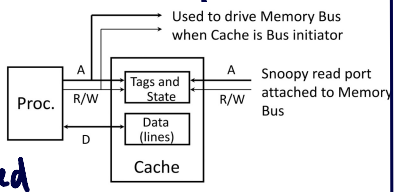
Coherence: what values can read return, r/w to single memory location

Consistency: when write visible to reads, r/w to multiple memory

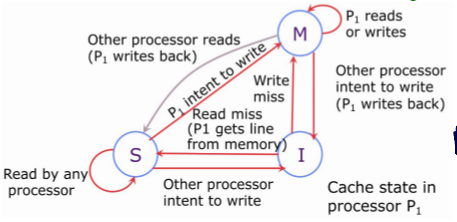
Snoopy Cache: cache watch other memory txn

Write Miss: address invalidated in other caches before write

Read Miss: if dirty copy is found, write-back is performed before memory is read



Cache State-Transition Diagram



Modified
Shared
Invalid

MESI: E state (Exclusive, clean)

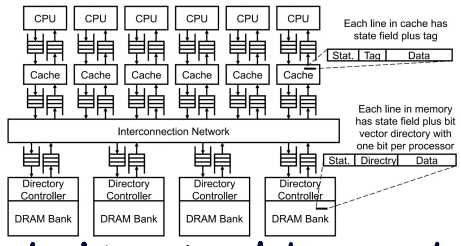
Inclusion: entries in L1 must be in L2

False Sharing: cache-coherence done at the line level not word level

Coherence Miss:
- ↳ True sharing miss: communication of data through cache coherence mechanism
- ↳ False sharing miss: line invalidated because some word in the line is written into

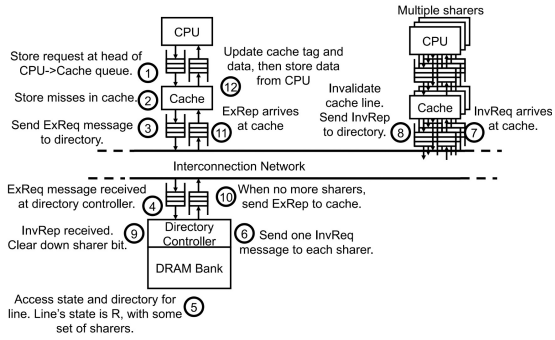
Snooping: must probe every other cache
- ↳ use multiple interleaved buses w/ interleaved tag banks, point-to-point network

Directories: every memory line has associated directory info
- track which caches have copies of data, probe those



Cache States:
1) C-invalid: accessed data not resident in cache
2) C-shared: data in cache, possibly other sites, data in memory valid
3) C-modified: exclusive to cache, has been modified, memory not up to date
4) C-transient: transient sent, not received

Write miss, to read shared line



22 Memory Consistency Models

Synchronization:

Producer-Consumer: consumer must wait until producer produced data

Mutual Exclusion: only one process uses resource at given time

Memory Consistency Model: sequential ISA says it in order, what values can be returned by ld
- coherence: ensure memory system in parallel comp as if caches were not there

Sequentially Consistency: result of any execution is same as if operations were in sequential order, order-preserving interleaving of memory ref
- requires SC made machines slower

Store Buffer: allows stores to buffered while waiting for access to shared memory

TSO (Total Store Order): strongest memory model in common use
- allows local buffering of stores by processor

Strong Memory Consistency Models: more guarantees on ordering of loads & stores across hardware threads, easier ISA-level programming model, more hardware

Weak, multi-copy-atomic memory models: processor sees writes by another processor in same order

Weak, non-multi-copy-atomic memory models: processor can see another writes in diff orders

Relaxed Memory Models: each program can have diff memory models

23 Synchronization

Mutual Exclusion: to avoid a deadlock, let process give up reservation while waiting

Dekker's Algorithm

Process 1

```
    ...
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

Regular loads and stores in SC model sufficient to implement mutual exclusion

Atomic Memory Operations (AMO): two ordering bits acqure and release

Non-blocking synchronization allows critical sections to execute w/o taking lock

Compare-and-Swap: complex instr, double Compare-and-Swap to access two words

Load-Reserved / Store-Conditional: line in cache in E/M state

RISC-V Atomic Instr: guaranteed forward progress for simple operations

Exam

VIPT: min associativity for no virtual address aliasing in VIPT

(num ways) * (num sets) * (Line size) ≤ (Page Size)

$2^{15} \leq 2^{12}$, 8 associativity needed

Coherence Protocol

initial state	other cached	ops	actions by this cache	final state	this cache	other caches	mem
Invalid	no	none	none	**I**			yes
		CPU read	**CR**	**CE**	yes		yes
		CPU write	**CRI**	**OE**	yes		
		replace	none	*impossible*			
		CR	none	**I**		yes	yes
		CRI	none	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes
Invalid	yes	none	same as above	**I**		yes	yes
		CPU read		**CS**	yes	yes	yes
		CPU write		**OE**	yes		
		replace		*impossible*			
		CR		**I**		yes	yes
		CRI		**I**		yes	
		CI		**I**		yes	
		WR		**I**		yes	yes
		CWI		**I**			yes

initial state	other cached	ops	Actions by this cache	final state	this cache	other caches	mem
cleanExclusive	no	none	none	**CE**	yes		yes
		CPU read	none	**CE**	yes		yes
		CPU write	none	**OE**	yes		
		replace	none	**I**			yes
		CR	none or CCI[1]	**CS**	yes	yes	yes
		CRI	none or CCI[1]	**I**		yes	
		CI	none	*impossible*			
		WR	none	*impossible*			
		CWI	none	**I**			yes

Table P5.7-1

Cache coherency

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