ECE552 Course Notes

Stephen Yang

September 2023

0.1

1 Preliminaries

Architecture: timing independent function of computer **MicroArchitecture**: Implementation techniques to improve perf. **Implicit Parallelism**: increase ILP, pipelining, caching **Explicit Parallelism**: Data parallelism, TLP **ISA**: SW/HW Interface

2 Performance Metrics

2.1 Latency

Latency (execution time) is the time required to finish some fixed task. Processor A is X times faster than B:

$$\mathrm{Lat}(P, A) = \mathrm{Lat}(P, B)/X$$

Processor A is X% faster than B:

$$\mathrm{Lat}(P, A) = \mathrm{Lat}(P, B)/\left(1 + \frac{X}{100}\right)$$

2.1.1 Adding Latencies

$$\mathrm{Lat}(P_1 + P_2, A) = \mathrm{Lat}(P_1, A) + \mathrm{Lat}(P_2, A)$$

2.2 Throughput (Bandwidth)

Proc. A has X times TP relative to B:

$$\mathrm{TP}(P, A) = \mathrm{TP}(P, B)/X$$

Proc. A has X% the TP relative to B:

$$\mathrm{TP}(P, A) = \mathrm{TP}(P, B)/\left(1 + \frac{X}{100}\right)$$

2.2.1 Adding Throughputs

$$\text{TP}(P_1 + P_2, A) = \frac{2}{\frac{1}{\text{TP}(P_1,A)} + \frac{1}{\text{TP}(P_2,A)}}$$

2.3 Speedup

$$\text{Speedup} = \frac{T_{R,i}}{T_i} = \frac{T_{old}}{T_{new}}$$

- $T_{R,i}$ execution time on reference machine
- T_i execution time on evaluated machine

2.4 Slowdown

$$\text{Slowdown} = F \cdot (R_{exe}) + (1 - F) \cdot 1$$

- F fraction of instructions that experience slowdown
- R_{exe} factor of how much slower the F instructions are

2.5 Execution Time

$$T_{exe} = \text{Lat}(P) = IC \times CPI \times T_c$$

- IC dynamic instruction count
- CPI is # of cycles per instr
- T_c is the seconds per cycle (clock period)

2.5.1 CPI (Cycles per Instr) & IPC

$$CPI = \frac{T_{exe}}{IC \times T_c} \qquad IPC = \frac{1}{CPI}$$

For 5-stage processor,

$$CPI = 1 + \sum f_{stall} \cdot c_{stall}$$

- f_{stall} is the frequency of instructions that stall
- c_{stall} is the # of stall cycles corresponding to instr with f_{stall} frequency

3 Analyzing Performance

3.0.1 Ratio of Means

$$\text{RoM} = \frac{\sum_{i=1,\ldots,N} T_{R,i}}{\sum_{i=1,\ldots,N} T_i}$$

- s_i is the speedup
- $T_{R,i}$ exec time on reference
- T_i exec time on evaluated

3.1 Arithmetic Average

For units proportional to time (e.g. latency)

$$\bar{L} = \frac{1}{N} \times \sum_{i=1}^{N} T_i = \frac{1}{N} \times \sum_{P=1,\ldots,N} \text{Lat}(P)$$

Where $\bar{L}$ is the average latency of all programs $T_i, ..., T_N$

3.2 Harmonic Average

For units inversely proportional to time

$$\bar{T} = \frac{N}{\sum_{P=1,\ldots,N} \frac{1}{\text{TP}(P)}}$$

Where $\bar{T}$ is the average throughput

3.3 Geometric Average

For unitless quantities (e.g. speedup)

$$\sqrt[N]{\Pi_{P=1,\ldots,N} \text{SpdUp}(P)} = \sqrt[N]{\text{SpdUp}(P_1) * \text{SpdUp}(P_2) * ... * \text{SpdUp}(P_N)}$$

4 Amdahl's Law

Assume an enhancement E, which speeds up fraction F of computation by factor S:

$$T_{exe}(\text{w/ E}) = T_{exe}(\text{w/o E}) \times \left[(1-F) + \frac{F}{S}\right]$$
$$\text{SpdUp}(E) = \frac{T_{exe}(\text{w/o E})}{T_{exe}(\text{with E})} = \frac{1}{(1-F) + \frac{F}{S}}$$

4.1 Parallel Case

Let P be number of cores, and F fraction of code that can be parallelized on P:

$$S_p = \frac{T_1}{T_p} = \frac{1}{1 - F + \frac{F}{P}} = \frac{P}{F + P(1-F)} < \frac{1}{1-F}$$

5 ISA

5.1 ISA Condition Codes

conditional execution (e.g. for branches) is set by condition codes, which differ for various ISA's. A typical condition code register: Z: Zero, C: Carry, V: Overflow, X: Extend, N: Negative

6 Pipelining

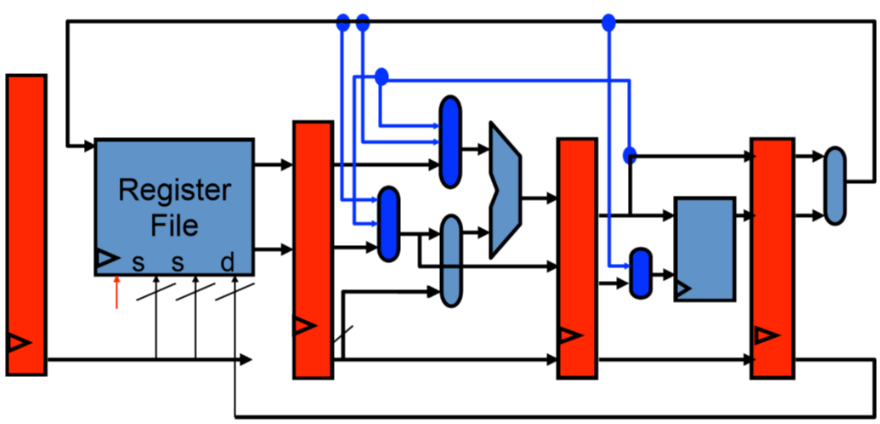
The classic 5 stage pipeline has

- Fetch: Fetch instruction from PC
- Decode: Read reg, find instr type
- eXecute: Execute instr (ALU)
- Memory: Handle memory instr
- Writeback: write completed instr result to register file

7 Dependencies and Hazards

type	T/F	Solution
RAW	T	stall, bypassing, reorder
LTU	T	stall + bypass
WAW	F	register rename
WAR	F	register rename
struct	T	stall, better design
ctrl	T	stall, flush F/D

7.1 Bypassing



bypass elements in 5 stage proc:

7.1.1 MX

beginning of M to input of X

$$XM.rd == DX.rs1$$

7.1.2 WX

beginning of W to input of X

$$MW.rd == DX.rs1$$

7.1.3 WM

beginning of W to input of M

7.2 LTU

even with bypassing, LTU hazard exists for instr with dist = 1

$$\text{stall} = (DX.op == LW)\&\&[(FD.rs1 == DX.rd)||(FD.rs2 == DX.rd)\&\&FD.op! = SW)]$$

8 Dynamic Branch Prediction

Compiler (Static): $\sim 85\%$. HW: $\sim 95\%$

8.1 Direction Prediction

Predict T/NT using BPB (conditional B).

8.1.1 1bit Predictor

1 bit (T/NT) - pred same way as last time.

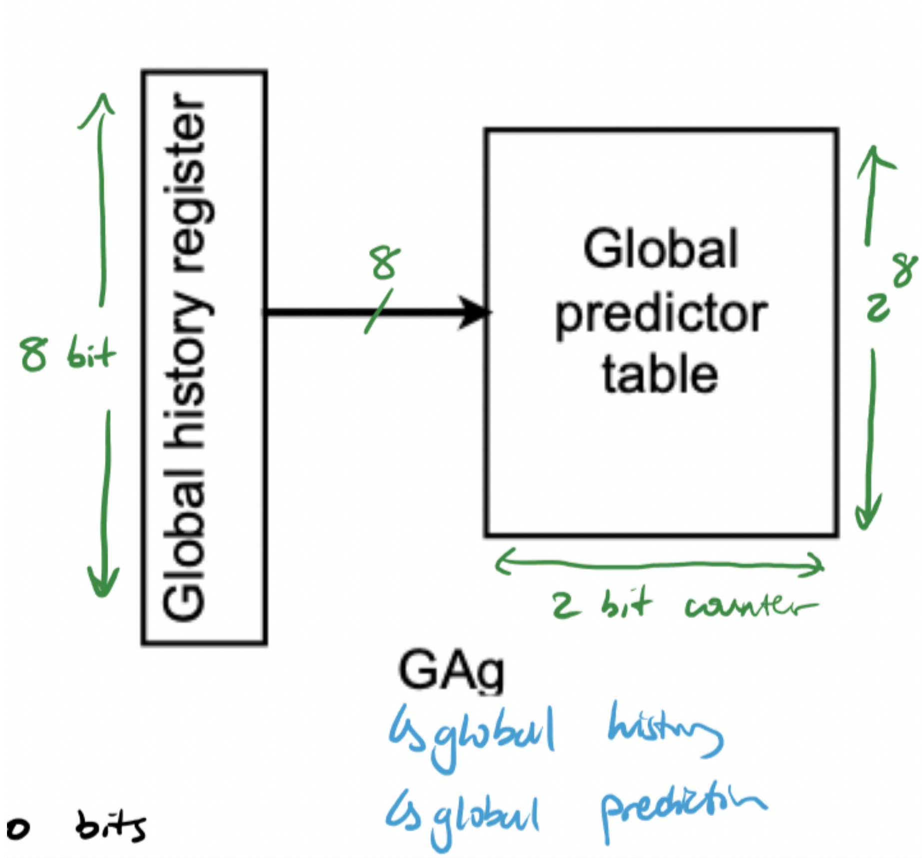
8.1.2 2bit Saturating Counter

2 bits (sT,wT,wNT,sNT) - branch pattern has some correlation.

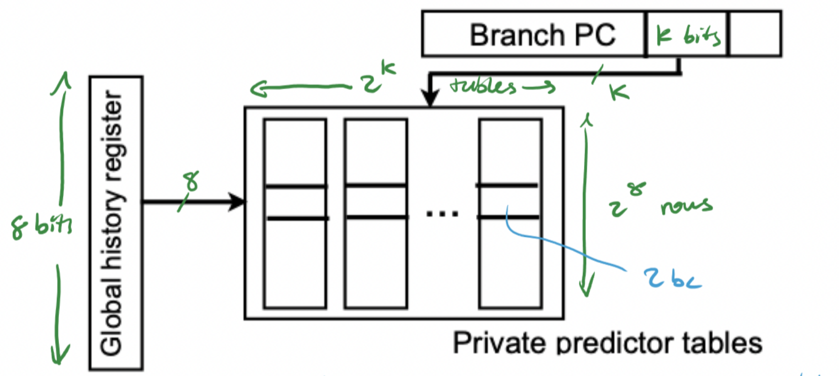
8.2 History-Based Methods (BHR)

GA = Global BHR, PA = Private BHR

8.2.1 GAg


Global history register
8 bit
8
Global predictor table
2^8
2 bit counter
GAg
↳ global history
↳ global prediction
0 bits

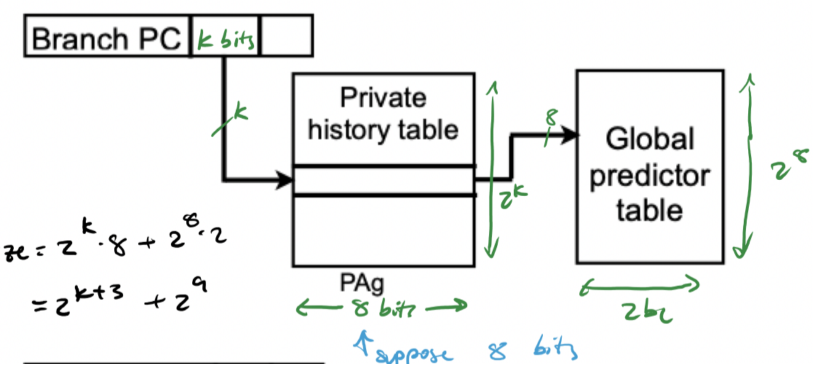
8.2.2 GAp



GAp — Global history, per address predictor table

Size $= 2^8 \cdot 2^k \cdot 2 + 8 = 2^{9+k} + 8$ bits $\quad k = 2$, size $=$

8.2.3 PAg

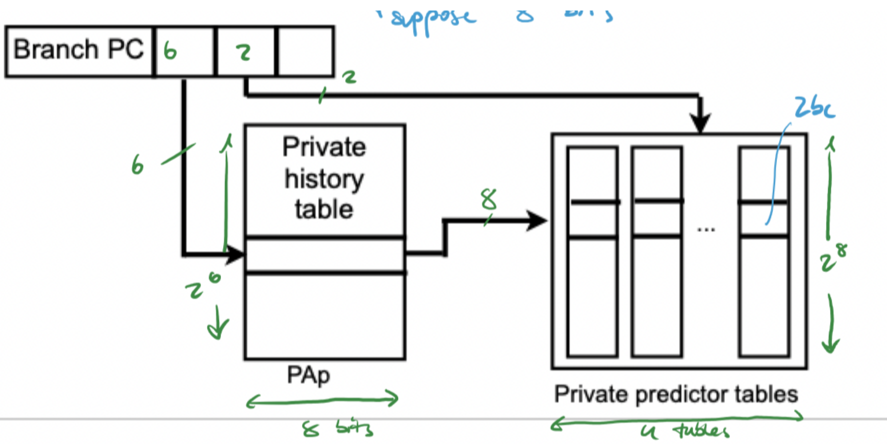


PAg

Size $= 2^k \cdot 8 + 2^8 \cdot 2$

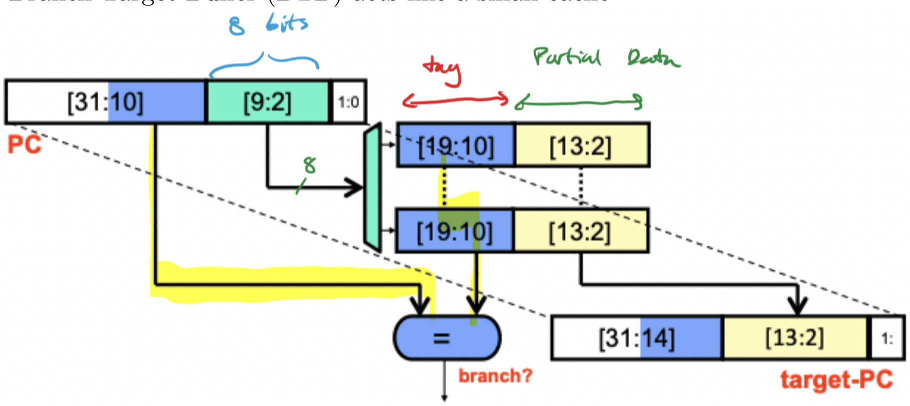
$= 2^{k+3} + 2^9$

8.2.4 PAp



8.3 Target Prediction

The Branch Target Buffer (BTB) acts like a small cache



BTB handles

- direct control branches, jumps, calls

BTB does **not** handle

- indirect control branches, jumps, calls
- indirect control jump (switch)
- returns

8.3.1 Returns

store return address on **Return Address Stack** (RAS)

9 Exceptions

Interrupts, exceptions, page faults, illegal op

9.1 Handling Exceptions

Save processor state, restart execution. Instr in flight become NOP

10 Dynamic Scheduling

aka out-of-order execution. benefits:

1. reduce RAW stalls
2. increase pipeline, FU utilization
3. increase ILP

11 Tomasulo's Algorithm

Features: **register renaming** using **tag**'s (avoid WAW, WAR). **Reservation Station** to buffer instructions (instr q). **Common Data Bus** (CDB) to broadcast completed instr to RS's.

11.1 Processor Structure in Tomasulo

- Fetch: Fetch instruction from PC
- Dispatch: Check for structural hazard (RS full), rename output reg to allocated RS, check input registers ready
- Issue: Waits for RAW and Struct. hazards. If reg ready, send to X
- eXecute: Execute instr (ALU)
- Memory: Handle memory instr
- Writeback: Broadcast on CDB (wait for structural hazard), clear RS entry and tag on tag match

11.2 Register Renaming

Storage locations referred to by RS# tags

$$Tag == 0- > \text{val in reg table}$$
$$Tag! = 0- > \text{val not rdy (being computed)}$$

12 Precise State

Speculation requires ability to abort and restart. Tomasulo has ooo completion, hard to restore precise instr state.

12.1 Re-Order Buffer (ROB)

Register writes executed in dispatch order. ROB stores completion flag of instr, new and old register mapping. Enables in-order dispatch, ooo execution, in-order completion.

12.2 Load/Store Queue (LSQ)

Completed stores write to LSQ. LSQ writes to memory when store retires. Loads access LSQ and data cache in parallel if $\exists$ older store with matching addr. Forward LSQ value if exists.

13 MIPS R10K

13.1 Physical Register File

MIPS R10K has **Physical Registers** (PR) instead of named architectural registers. Conceptually, big bank of physical registers which can be associated via the ROB.

13.2 R10K Structures

- ROB: T_{old} PR prev. mapped to this instr, T PR corresponding to this instr's logical output
- RS: T PR, $S1, S2$ PR tags corresponding to instr inputs, rdy ready bit
- Map Table: T PR, rdy ready bit
- Free List: PR#

13.3 Processor Structure

- Fetch: Fetch instruction from PC
- Dispatch: In-order, Check for structural hazard (RS, ROB, PR#), Allocate RS + ROB entries, new PR#, Read PR tags for input regs (store in RS S1, S2)
- Issue: Waits for RAW and struct. hazards. If reg ready, send to X
- eXecute: Execute instr (ALU). Can free RS entry at end of X since RS# is not a tag
- Complete: Write destination PR. Set inst output reg ready in map table and RS
- Retire: If instr at ROB head not complete, stall. Handle exceptions. If store, right value from LSQ into data. Free T_{old}, ROB and LSQ entries.

14 Recovering from Misspeculation

Two ways to restore precise state

14.1 Serial Rollback using T, T_{old}

Slow (serial), but simple and cheap (hardware)

14.2 Single Cycle Restore Checkpoint

Fast (single cycle), but very expensive

14.3 Hybrid

Checkpoint low confidence branches. Serial recovery for page faults.

15 Caches

15.1 Cache Organization

$$b_{addr} = b_{tag} + b_{index} + b_{offset}$$
$$Blocks = C/BS \quad \#Sets = \#Blocks/\#Ways$$

where C = capacity, BS = block size

$$b_{tag} = b_{addr} - \log_2(\#Sets) - \log_2(BS)$$

15.1.1 Tag Overhead

$$C = Data + OH = Data +$$

Where N = number of entries

15.2 Split $I\$/D\$$ Reasoning

Avoid structural hazard (read ports), store additional metadata for pred, exploit data locality for prefetch, $I\$$ can be RO.

15.3 Cache Performance Metrics

$$\%_{miss} = \frac{\text{\# Misses}}{\text{\# Accesses}}$$
$$\%_{hit} = \frac{\text{\# Hits}}{\text{\# Accesses}} = 1 - \%_{miss}$$

t_{hit}: time to access cache.
t_{miss}: time to bring data into cache.

15.4 Cache Performance Equation

$$t_{avg} = t_{hit} + \%_{miss} \cdot t_{miss}$$

15.5 Cache Misses: 3(4)C Hill Model

15.5.1 Cold Misses

Independent of the cache, equal to **number of blocks in the trace**

$$M_{cold} = \#\text{blocks used}$$

assuming cache initialized to 0

15.5.2 Capacity Misses

Independent of cache organization or replacement policy.

$$M_{cap} = M_{FA,LRU} - M_{cold}$$

where $M_{FA,LRU}$ is the number of misses in a FA LRU cache

15.5.3 Conflict Misses

Dependent on cache organization and replacement policy.

$$M_{conflict} = M_{total} - (M_{cap} + M_{cold})$$

15.5.4 Coherence Misses

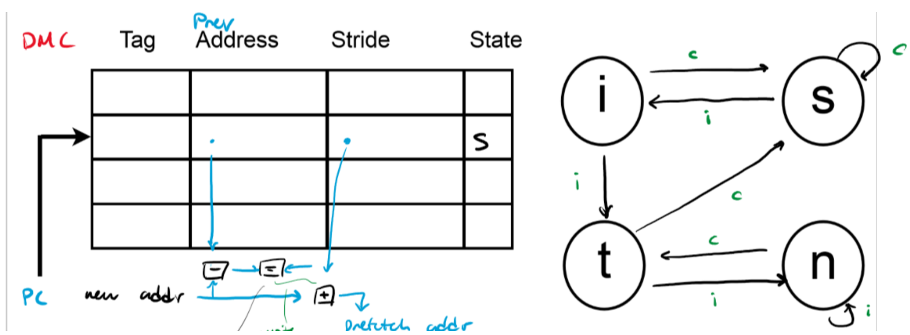
Miss due to external invalidation (only in shared memory multiprocessors)

15.6 Replacement Policies

1. **Random Replacement**
2. **FIFO/FILO**
3. **LRU (Least Recently Used)**: 2way=1 bit per set. $N >$ 2way=counter per way, OR $\log_2 N$ bits per set
4. **NMRU (Not Most Recent Use)**: 1 bit MRU set per line, random select NOT MRU to replace
5. **Belady's**: Furthest used in future replaced first

15.7 Prefetching

15.7.1 Stride Prefetcher



i: initial, s: stable, t: trans., n: incorrect

15.8 Write Propagation

- **write-through (WT)**: propagate value immediately to $
- **write-back (WB)**: write when block replaced (req. dirty bit)

15.9 Allocate

- **Write-allocate**: read from lower level, write value. Used with WB
- **Write-non-allocate**: write blk to next level. Used with WT

16 Multiprocessors

16.1 Coherence

16.1.1 VI (MI) Protocol

S	Current Proc		Other Proc	
	Load	Store	Load	Store
I	miss /V	miss /V	-	-
M	hit	hit	SD /I	SD /I

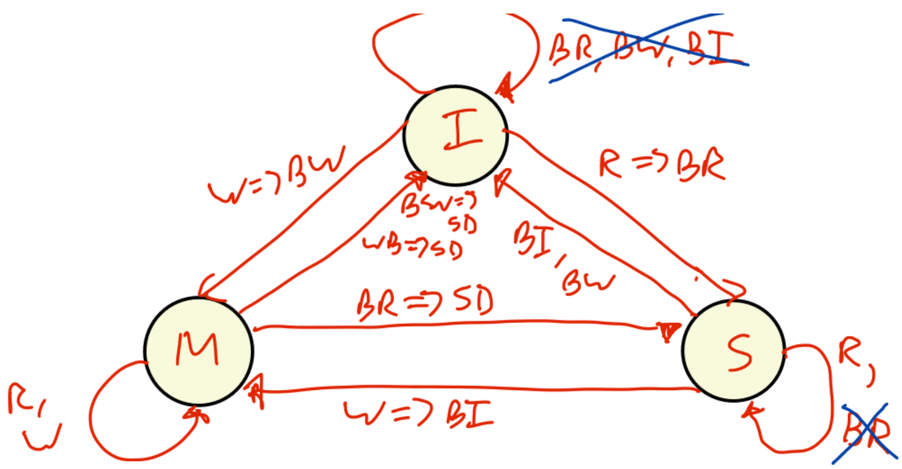
16.1.2 MSI Protocol

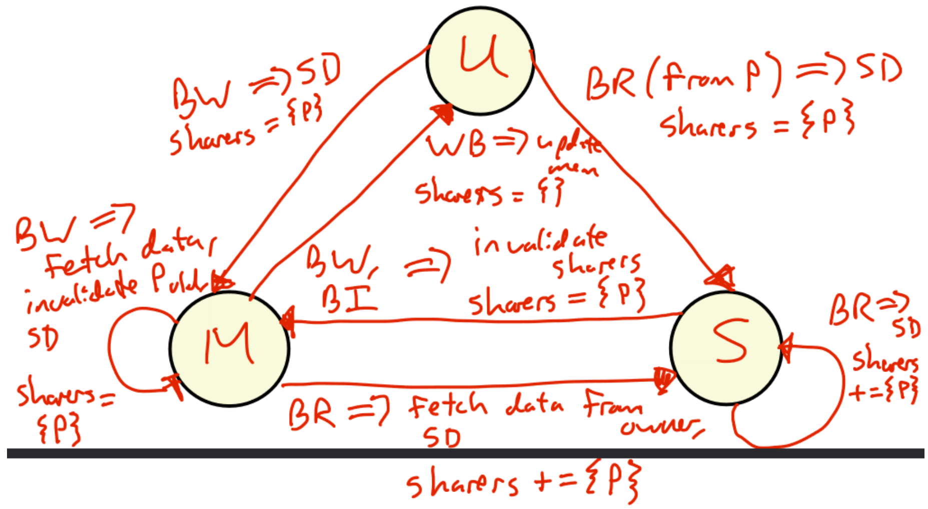
S	Current Proc		Other Proc	
	Load	Store	Load	Store
I	miss /S	miss /M	-	-
S	hit	upgrade miss /M	-	Invalid. /I
M	hit	hit	SD /S	SD /I

16.1.3 MSI - Directory

Tracks the following per cache block:

- Owner
- Sharers (bit vector)
- Home directory
- State





	Load	Store	Replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	Send GetS to Dir/IS^D	Send GetM to Dir/IM^{AD}										
IS^D	Stall	Stall	Stall			Stall		-/S		-/S		
IM^{AD}	Stall	Stall	Stall	Stall	Stall			-/M	-/IM^A	-/M	ack--	
IM^A	Stall	Stall	Stall	Stall	Stall						ack--	-/M
S	Hit	Send GetM to Dir/SM^{AD}	Send PutS to Dir/SI^A			Send Inv-Ack to Req/I						
SM^{AD}	Hit	Stall	Stall	Stall	Stall	Send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A		ack--	
SM^A	Hit	Stall	Stall	Stall	Stall						ack--	-/M
M	Hit	Hit	Send PutM +data to Dir/MI^A	Send data to Req and Dir/S	Send data to Req/I							
MI^A	Stall	Stall	Stall	Send data to Req and Dir/SI^A	Send data to Req/II^A		-/I					
SI^A	Stall	Stall	Stall			Send Inv-Ack to Req/II^A	-/I					
II^A	Stall	Stall	Stall				-/I					

	GetS	GetM	PutS-NotLast	PutS-Last	Put M+ data from Owner	PutM+data from Non-Owner	Data
I	Send data to Req, add Req to Sharers/S	Send data to Req, set Owner to Req/M	Send Put-Ack to Req	Send Put-Ack to Req		Send Put-Ack to Req	
S	Send data to Req, add Req to Sharers	Send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	Remove Req from Sharers, sent Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req/I		Remove Req from sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/S^D	Send Fwd-GetM to Owner, set Owner to Req	Send Put-Ack to Req	Send Put-Ack to Req	Copy data to memory, clear Owner, send Put-Ack to Req/I	Send Put-Ack to Req	
S^D	Stall	Stall	Remove Req from Sharers, send Put-Ack to Req	Remove Req from Sharers, send Put-Ack to Req		Remove Req from Sharers, send Put-Ack to Req	Copy data to memory/S

16.2 Synchronization

16.2.1 Exchange

EXCH r1, 0(&lock)	MOV r1 -¿ r2	
	LW [&lock] -¿ r1	Wasted BW (other P spins on lock)
	SW r2 -¿ [&lock]	

16.2.2 Load Locked/Store Conditional

LL [r1] -¿ r2
SC r3 -¿ [r1]

SC returns 0 in r3 if value in r1 modified

16.2.3 Test and Set

A0 EXCH r1, [&lock]
A1 BNEZ r1, A0

16.2.4 Test and Test and Set

EXCH	LL/SC
LW [&lock] -¿ r1	LL [r1] -¿ r2
BNEZ r1, A0	BNEZ r2, A0
ADDI r1, 1-¿r1	ADDI r2, 1-¿r2
EXCH r1, [&lock]	SC r2 -¿ [r1]
BNEZ r1, A0	BEQZ r2, A0

16.3 Consistency

Coherence: globally uniform view of single mem loc. **Consistency**: globally uniform view of *all* mem locs.

16.3.1 Sequential Consistency

Proc's see own LD/ST in prog order, see other Proc's LD/ST in prog order. All proc's see same global LD/St order.

16.3.2 Ordering Rules

$$R \to W, R \to R, W \to R, W \to W$$

- **Total Store Ordering (TSO)**: aka Processor Consistency, relaxes $W \to R$
- **Weak Ordering (WO)**. All relaxed, *acquire-release* define critical sections

17 Superscalar

17.1 CPI

$$CPI_{ideal} = \frac{1}{N} \quad IPC_{ideal} = \frac{N * c}{c} = \frac{instrs}{c}$$

N = number of issues/retires per c

17.2 Superscalar Challenges

17.2.1 Wide Instruction Fetch

Multiple instr/cycle, but could need to predict multiple branches/cycle. **Banked I$**: DRAM banked, simultaneous read. **Combining Network**: Combine banked instr blocks.

17.2.2 Wide Instruction Decode

Register R/W Ports: Nominal $2N$ read, $1N$ write. In reality, lower (not all instr have 2 src, values bypassed), stores/branch (25-35%) don't write regs

17.2.3 N^2 Bypassing

Full bypassing requires N^2 dependence checks. $N+1$ input muxes at each ALU input. Routing can be expensive.

17.2.4 Clustering

Mitigates N^2 bypassing. Group FU's into K clusters. Limited bypass (1 cycle delay).

$$\left(\frac{N}{K}+1\right) inputs/mux \quad \left(\frac{N}{K}\right)^2 bypass/cluster$$

17.2.5 Wide Execute/Memory Access

N ALU's ok, N FP expensive. Wide mem acc depends on instr mix, probably only necessary $N > 4$

18 Multithreading