THE UNIVERSITY of WISCONSIN MADISON

ECE 752 Review: Modern Processors

Lecture notes based in part on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Erika Gunadi, Mitch Hayenga, Vignyan Reddy, Dibakar Gope

High-IPC Processor Evolution

Desktop/Workstation Market

Scalar RISC Pipeline	2-4 Issue In-order	Limited Out-of-Order	Large ROB Out-of-Order
1980s: MIPS, SPARC, Intel 486	Early 1990s: IBM RIOS-I, Intel Pentium	Mid 1990s: PowerPC 604, Intel P6	2000s: DEC Alpha 21264, IBM Power4/5, AMD K8

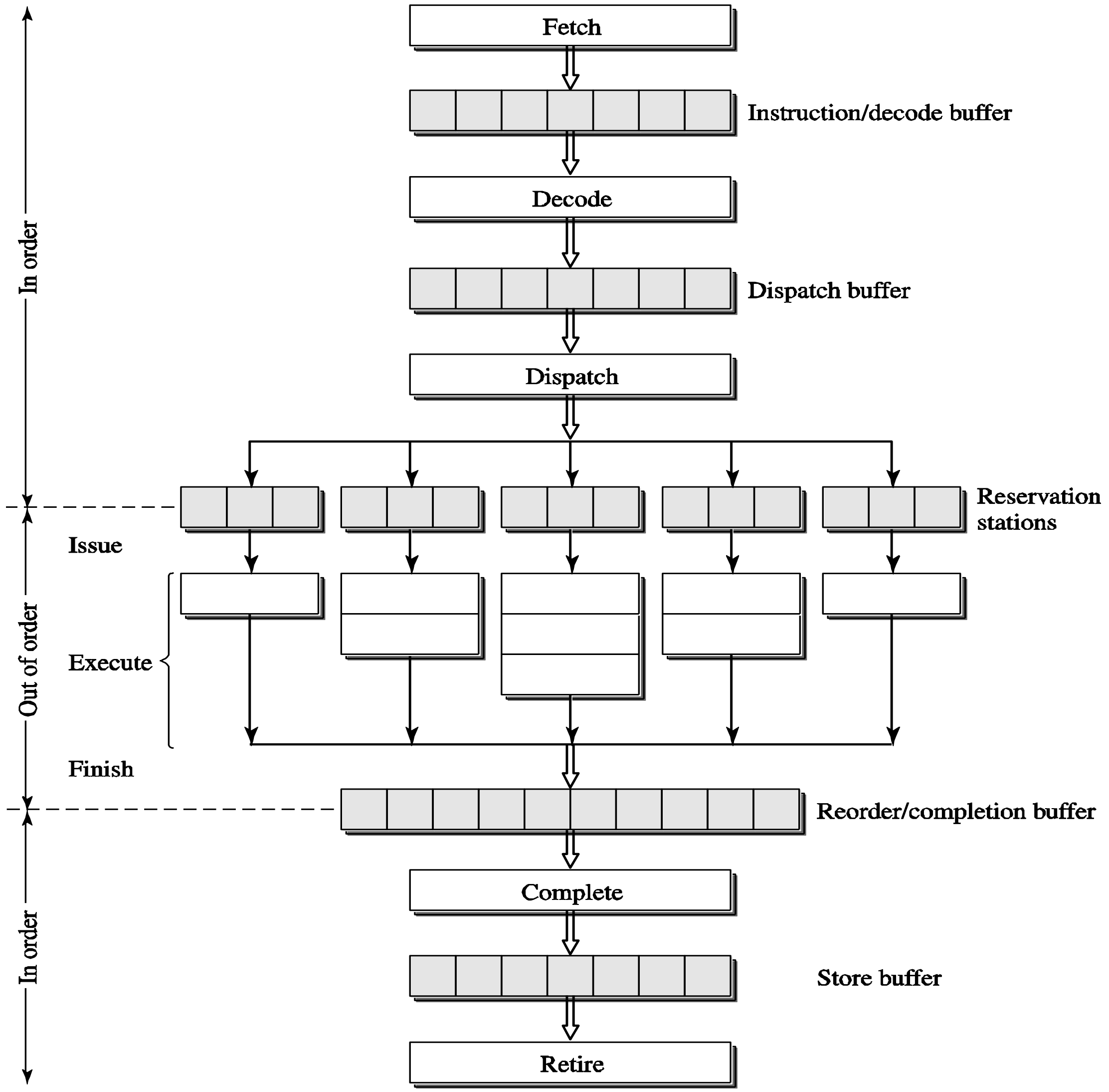
1985 – 2005: 20 years, 100x frequency

Mobile Market

Scalar RISC Pipeline	2-4 Issue In-order	Limited Out-of-Order	Large ROB Out-of-Order
2002: ARM11	2005: Cortex A8	2009: Cortex A9	2011: Cortex A15

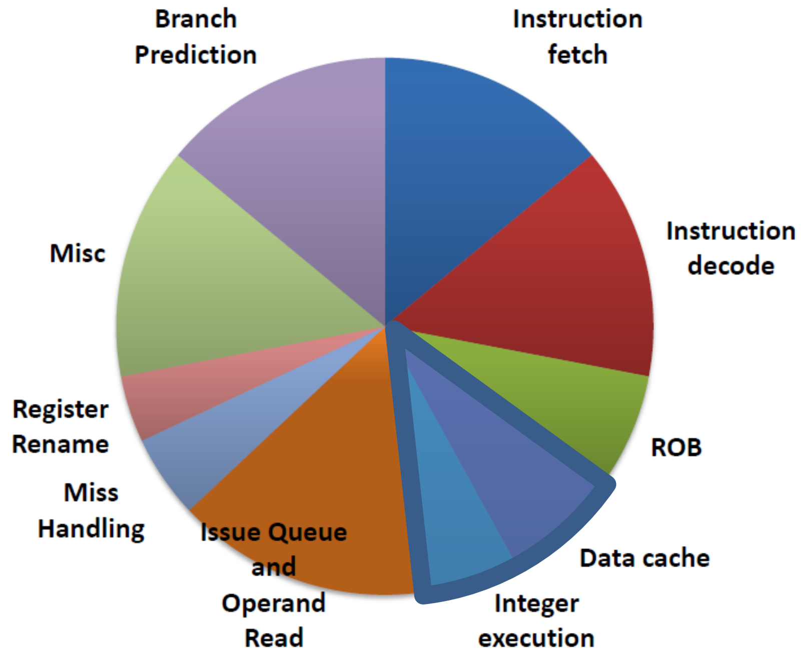
2002 – 2011: 10 years, 10x frequency

A Typical High-IPC Processor

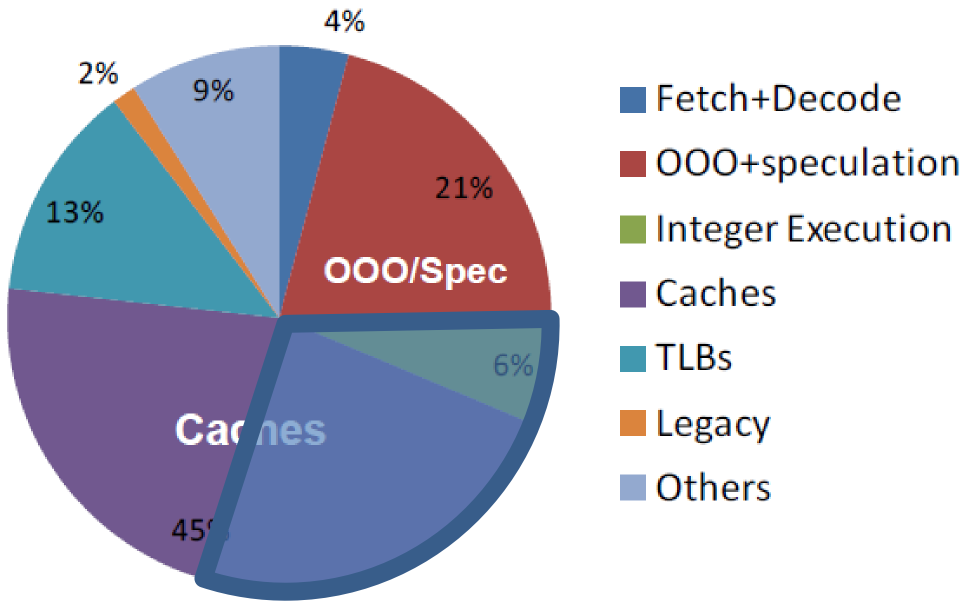


Power Consumption

ARM Cortex A15 [Source: NVIDIA]



Core i7 [Source: Intel]



- Actual computation overwhelmed by overhead of aggressive execution pipeline

Mobile CPUs: What Next?

NVIDIA Project Denver?

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \underbrace{\cancel{\frac{\text{Instructions}}{\text{Program}}}}_{\cancel{\text{(code size)}}} \times \underbrace{\frac{\text{Cycles}}{\text{Instruction}}}_{\text{(CPI)}} \times \underbrace{\cancel{\frac{\text{Time}}{\text{Cycle}}}}_{\cancel{\text{(cycle time)}}}$$

Architecture --> Implementation --> Realization

Compiler Designer   Processor Designer   Chip Designer

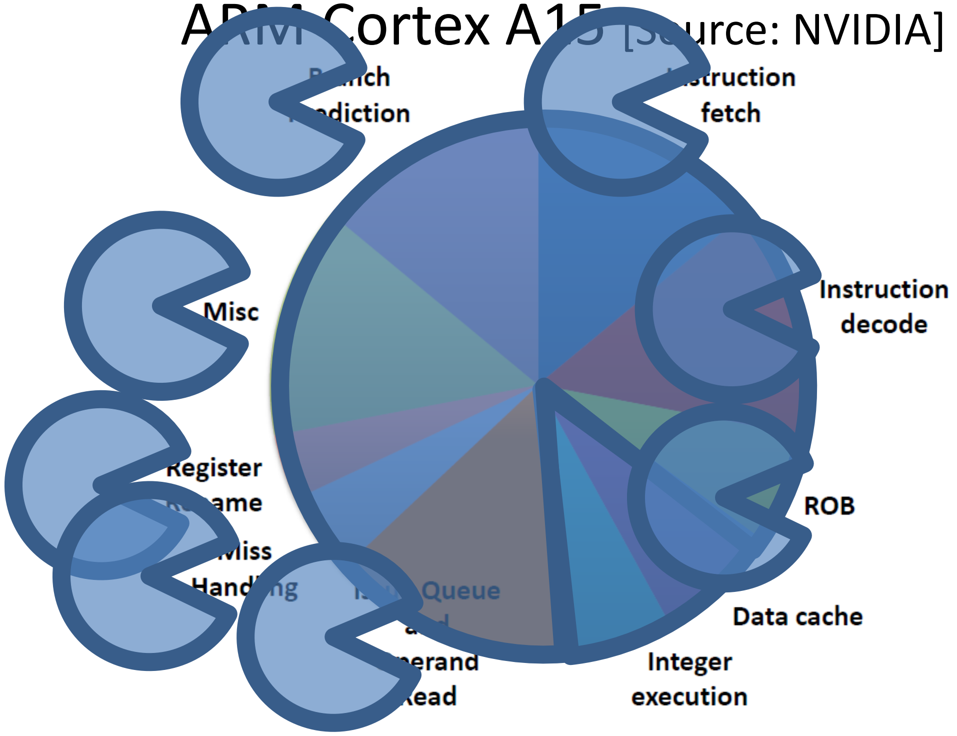
ARM ISA compatibility …

Frequency: maxed out due to power

ILP bag of tricks from desktop CPUs: empty

ILP is our only option

ARM Cortex A15 [Source: NVIDIA]

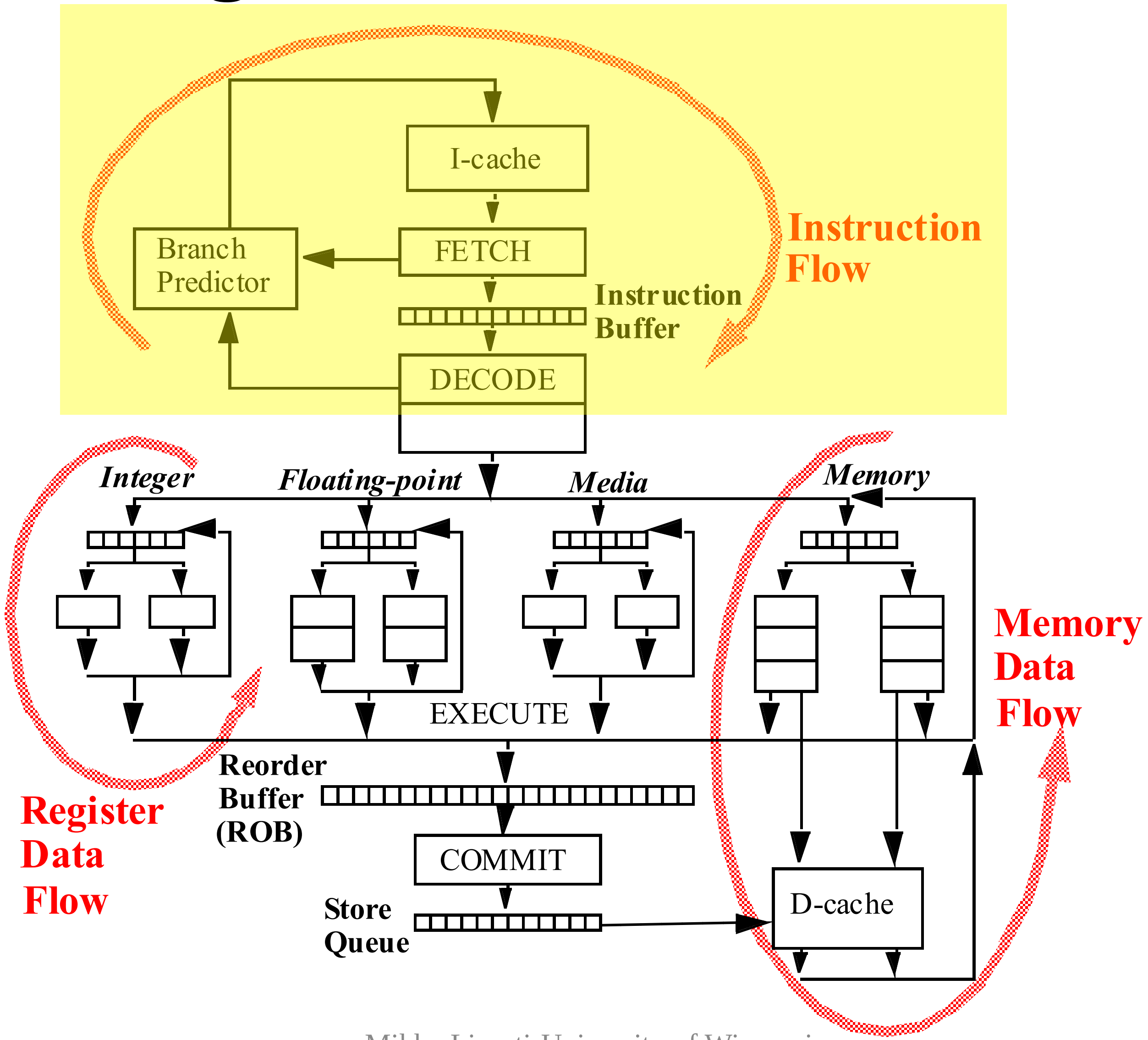


- Attack and reduce overheads one by one
- Free up power budget for actual computation

Lecture Summary

- Motivation
- Brief review: High-IPC, out-of-order processors
  - Instruction flow
  - Register Dataflow
  - Memory Dataflow
- Caches and Memory Hierarchy

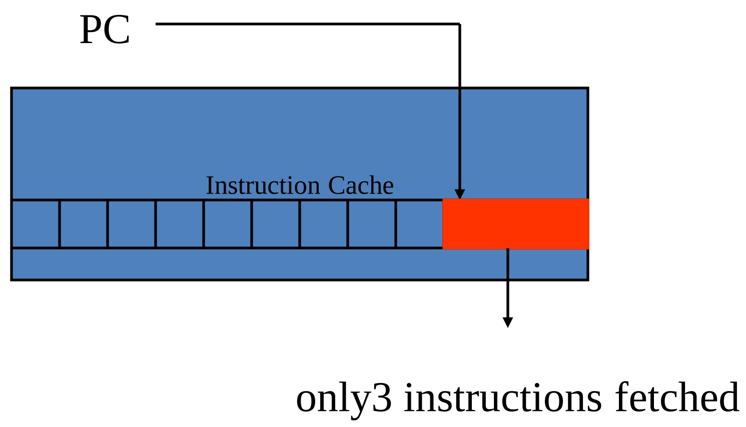
High-IPC Processor



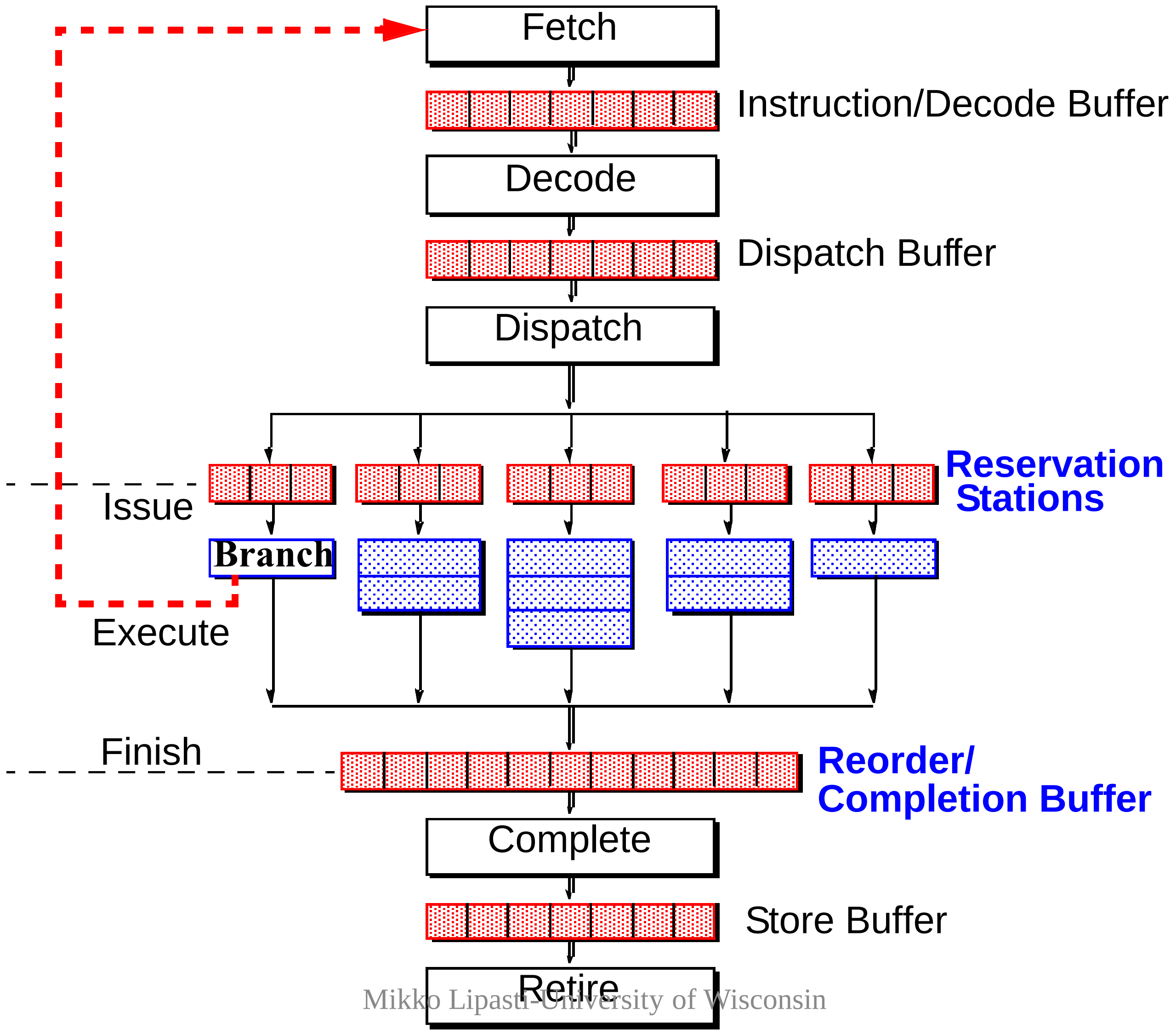
Instruction Flow

Objective: Fetch multiple instructions per cycle

- Challenges:
  - Branches: unpredictable
  - Branch targets misaligned
  - Instruction cache misses
- Solutions
  - Prediction and speculation
  - High-bandwidth fetch logic
  - Nonblocking cache and prefetching



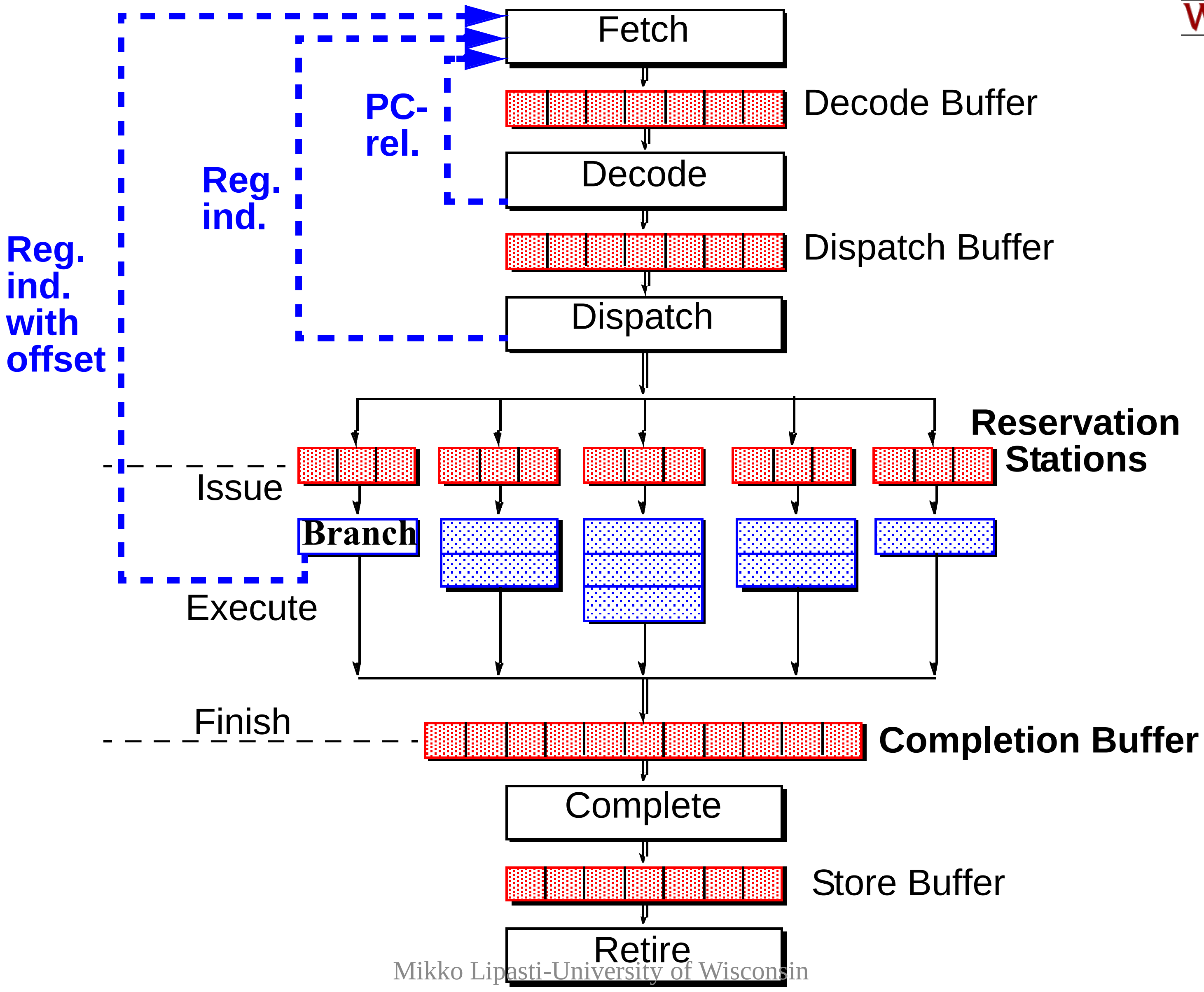
Disruption of Instruction Flow



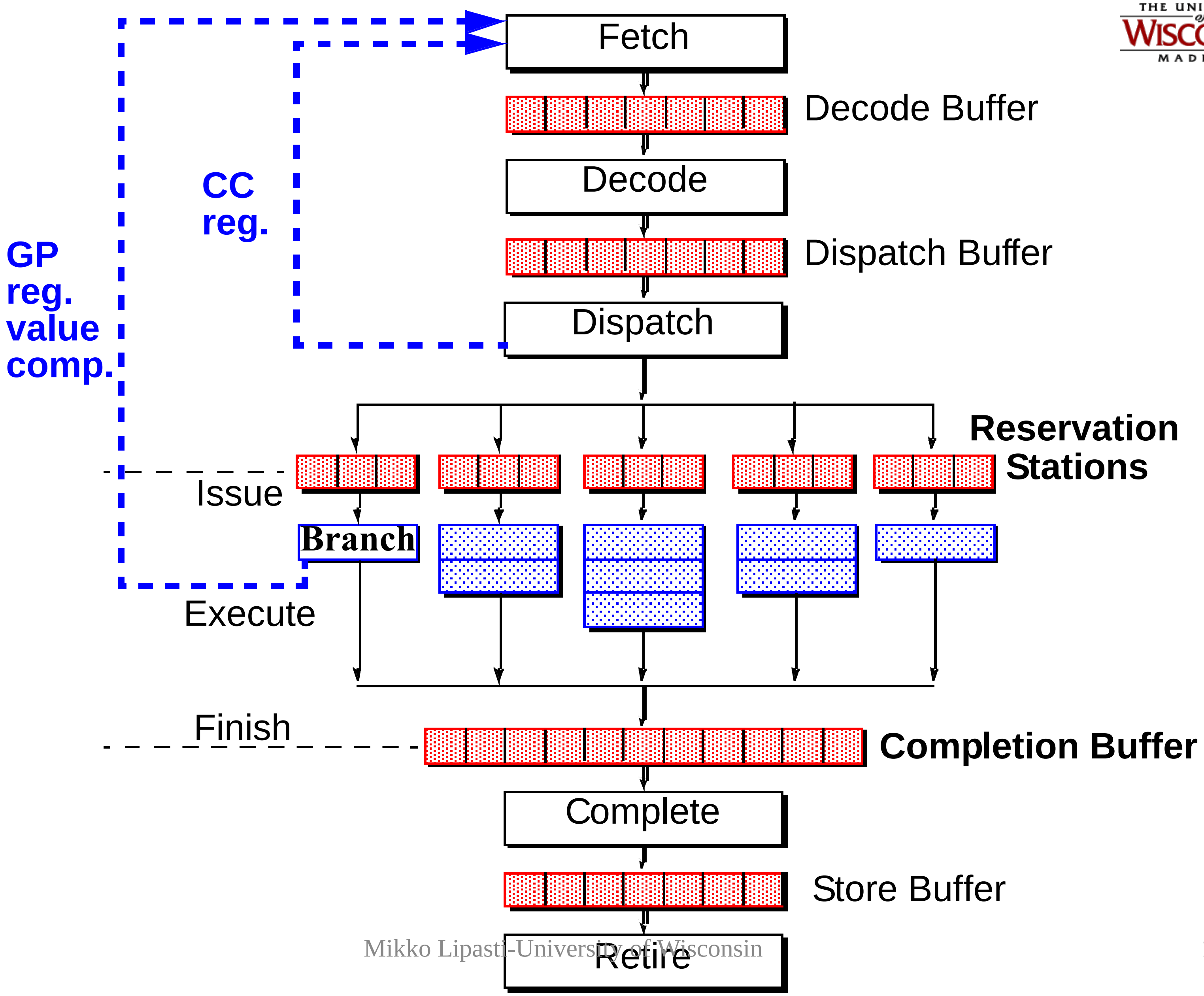
Branch Prediction

- Target address generation → <u>Target Speculation</u>
  - Access register:
    - PC, General purpose register, Link register
  - Perform calculation:
    - +/- offset, autoincrement
- Condition resolution → <u>Condition speculation</u>
  - Access register:
    - Condition code register, General purpose register
  - Perform calculation:
    - Comparison of data register(s)

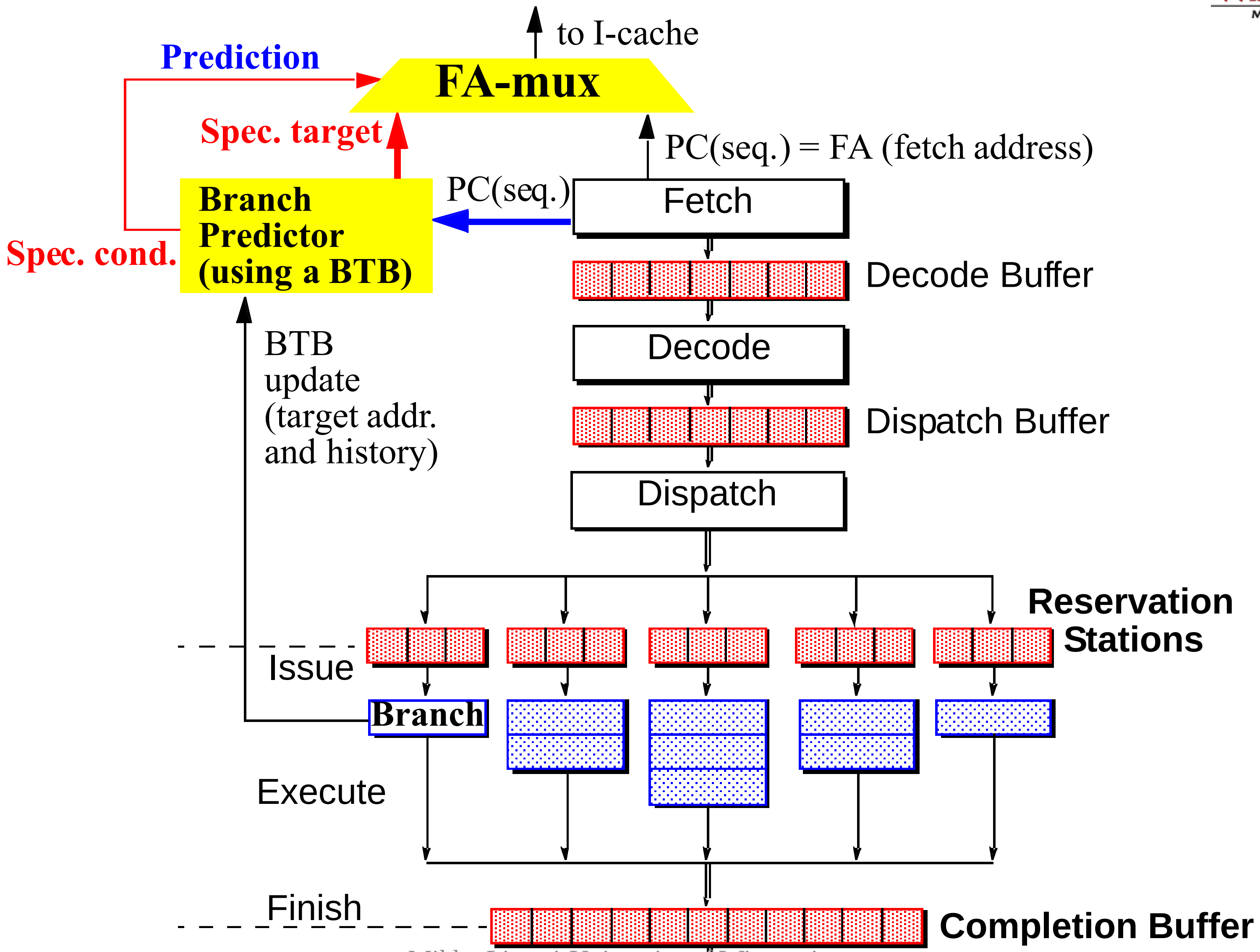
Target Address Generation



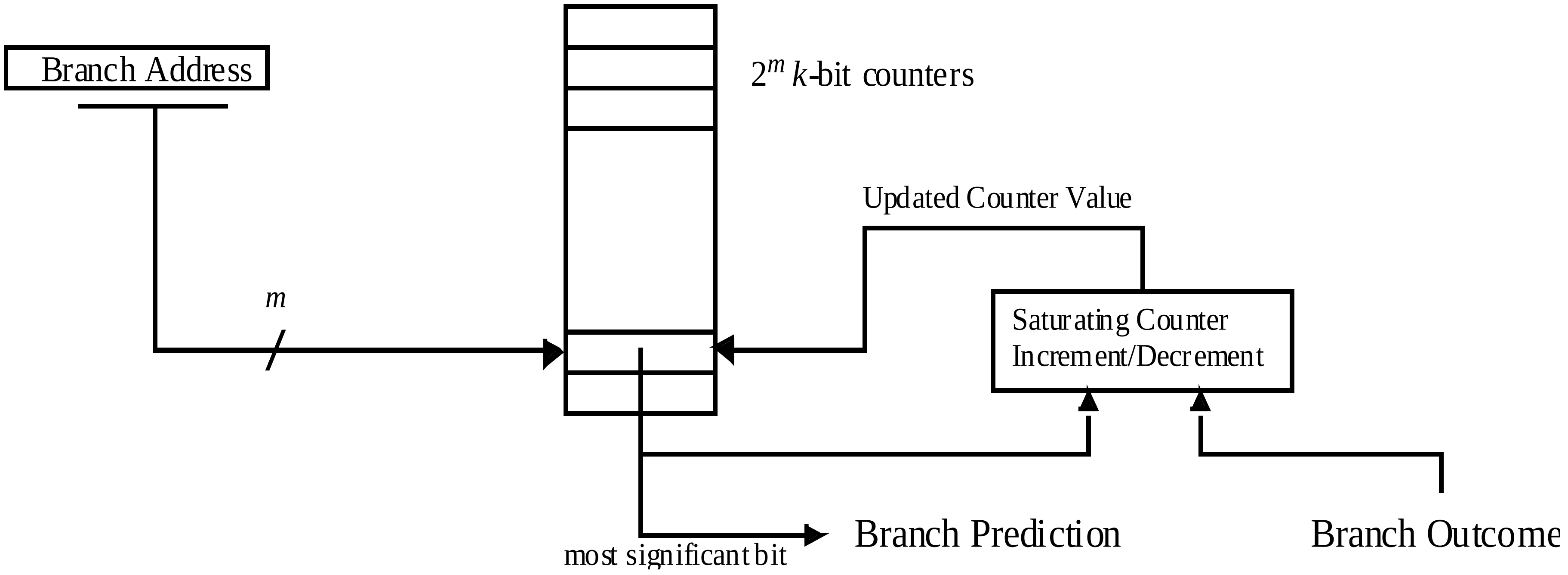
Branch Condition Resolution



Branch Instruction Speculation

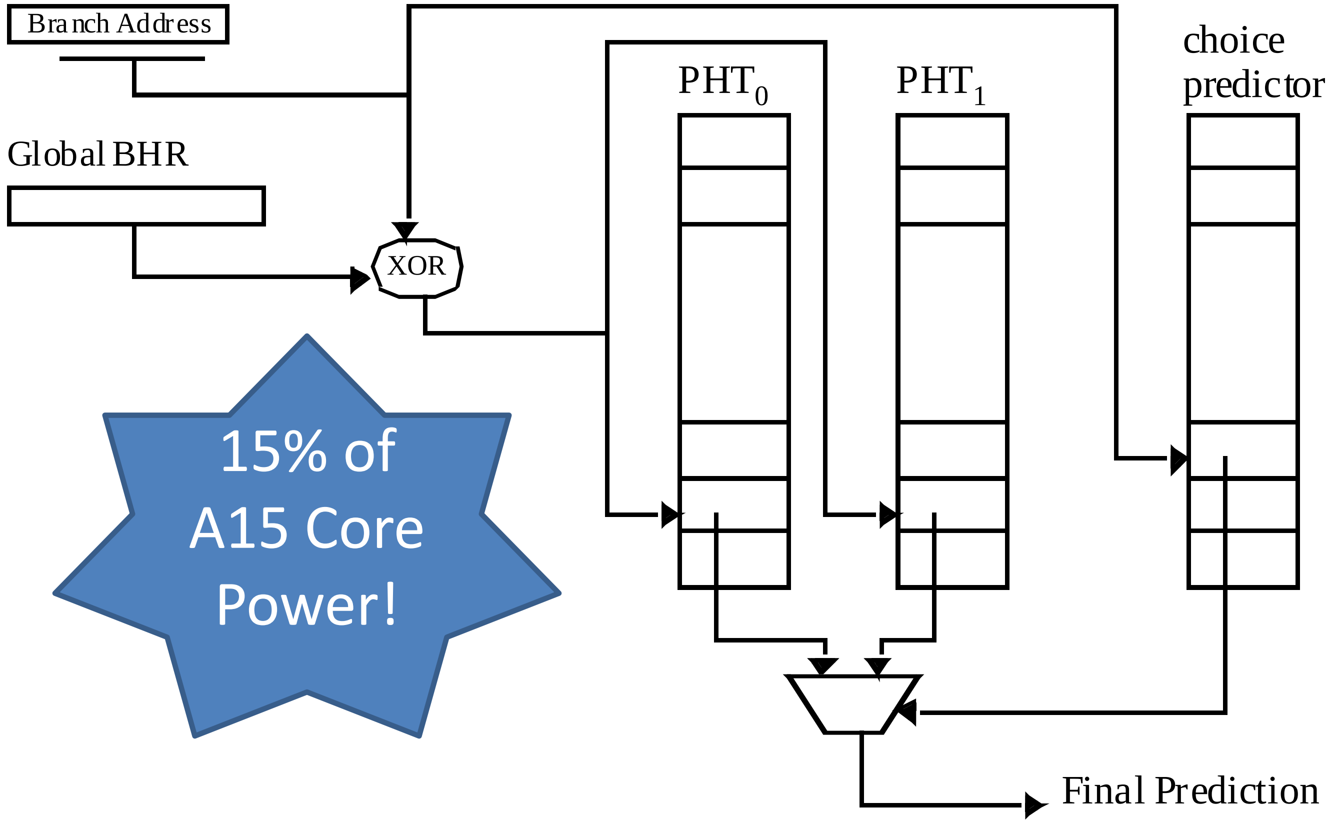


Hardware Smith Predictor



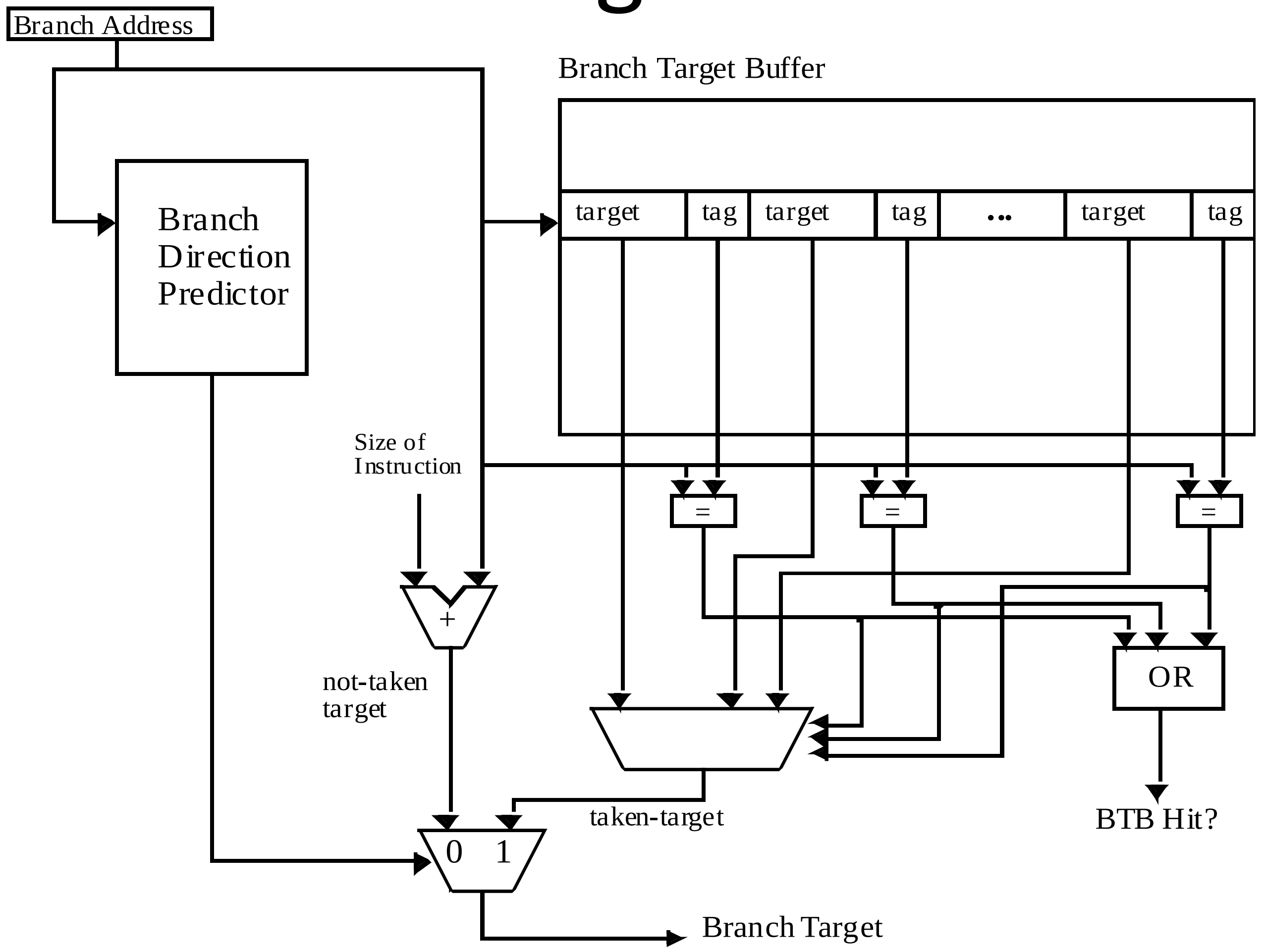
- Jim E. Smith. A Study of Branch Prediction Strategies. International Symposium on Computer Architecture, pages 135-148, May 1981
- Widely employed: Intel Pentium, PowerPC 604, MIPS R10000, etc.

Cortex A15: Bi-Mode Predictor



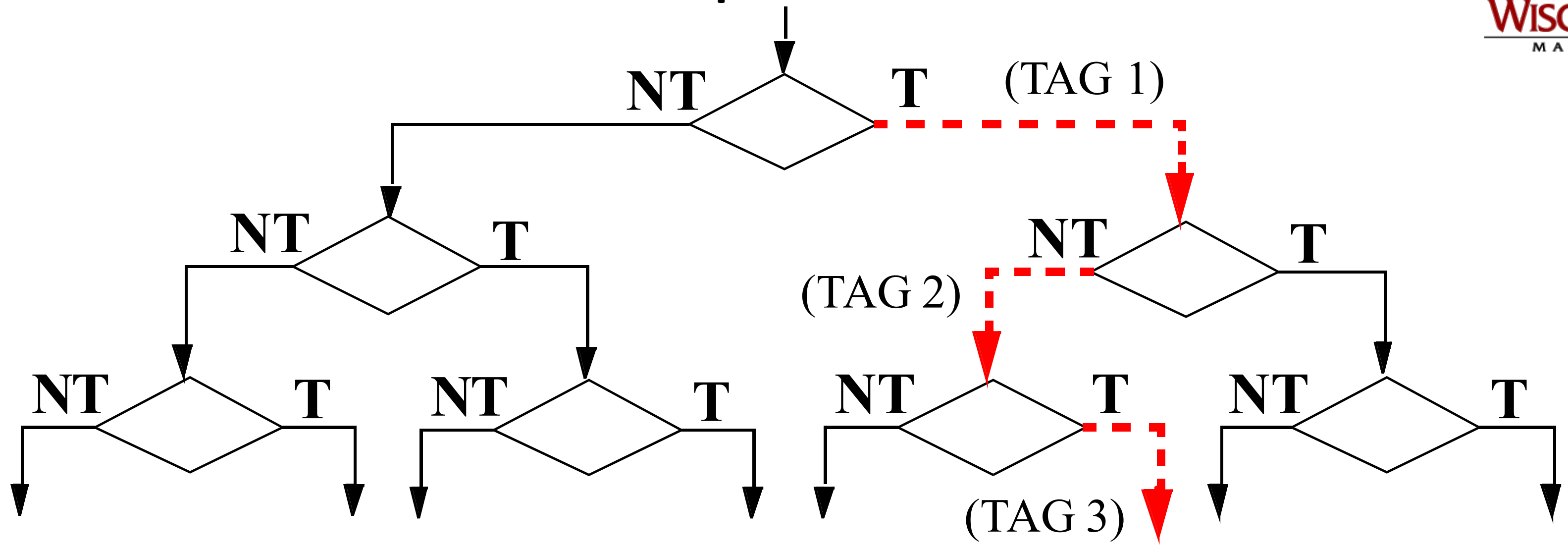
- PHT partitioned into T/NT halves
  - Selector chooses source
- Reduces negative interference, since most entries in PHT_0 tend towards NT, and most entries in PHT_1 tend towards T

Branch Target Prediction



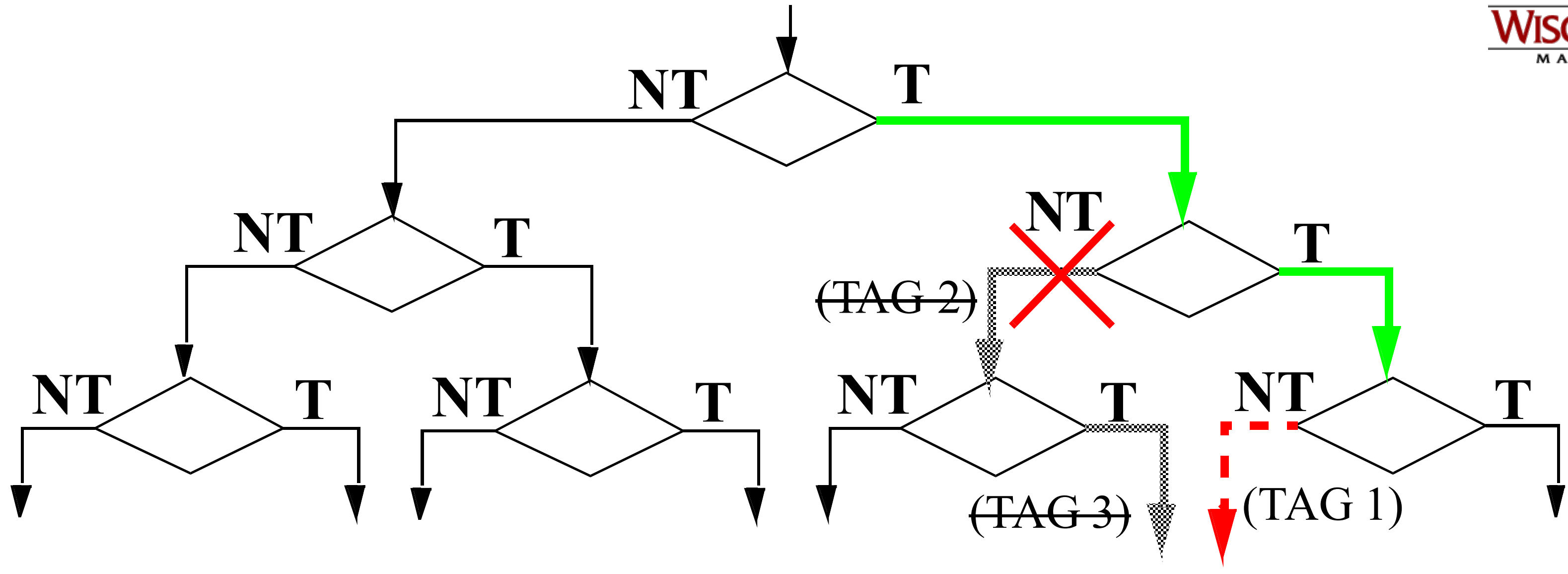
- Does not work well for function/procedure returns
- Does not work well for virtual functions, switch statements

Branch Speculation



- Leading Speculation
  - Done during the Fetch stage
  - Based on potential branch instruction(s) in the current fetch group
- Trailing Confirmation
  - Done during the Branch Execute stage
  - Based on the next Branch instruction to finish execution

Branch Speculation



- Start new correct path
  - Must remember the alternate (non-predicted) path
- Eliminate incorrect path
  - Must ensure that the mis-speculated instructions produce no side effects

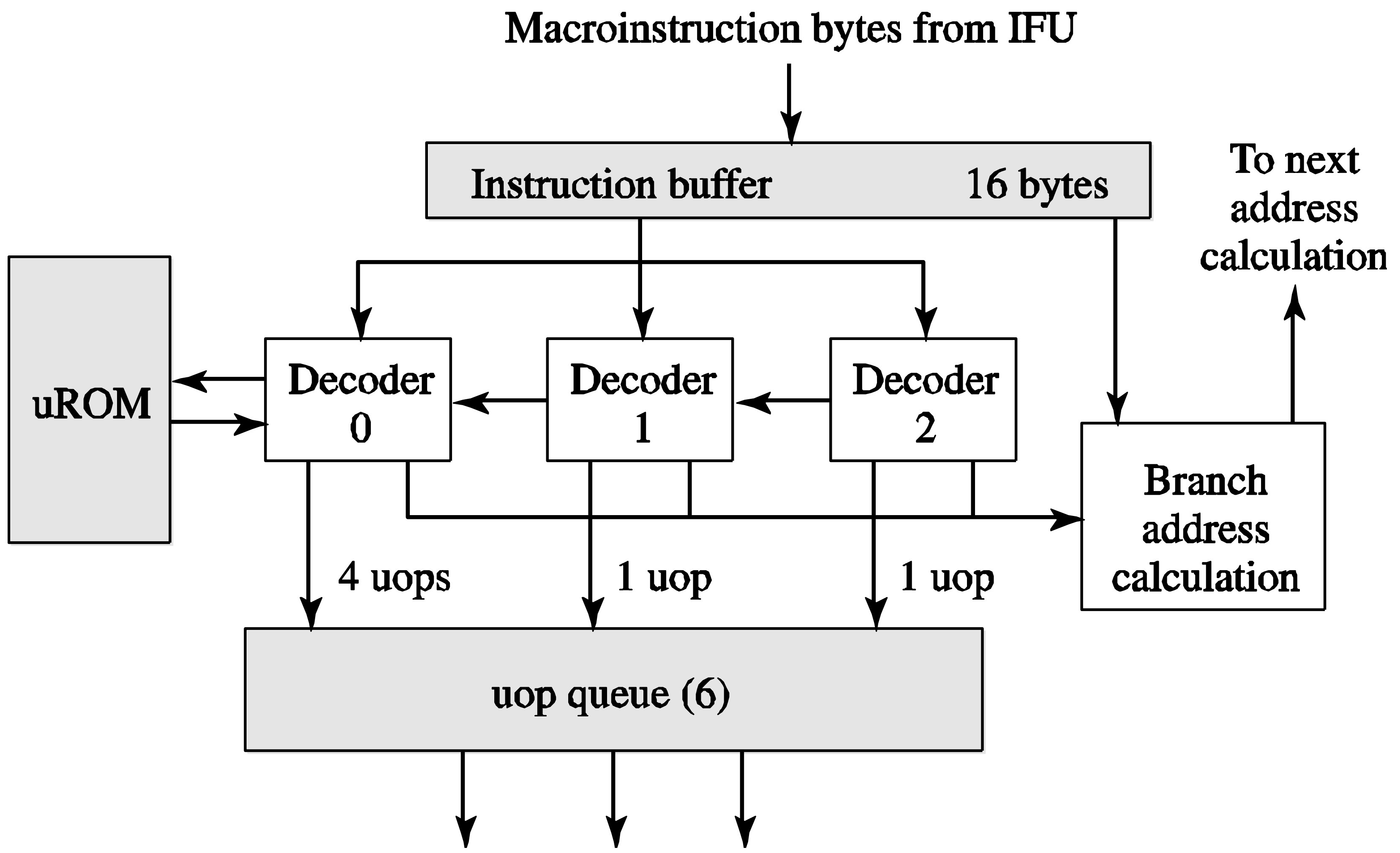
Mis-speculation Recovery

- Start new correct path
  1. Update PC with computed branch target (if predicted NT)
  2. Update PC with sequential instruction address (if predicted T)
  3. Can begin speculation again at next branch
- Eliminate incorrect path
  1. Use tag(s) to deallocate resources occupied by speculative instructions
  2. Invalidate all instructions in the decode and dispatch buffers, as well as those in reservation stations

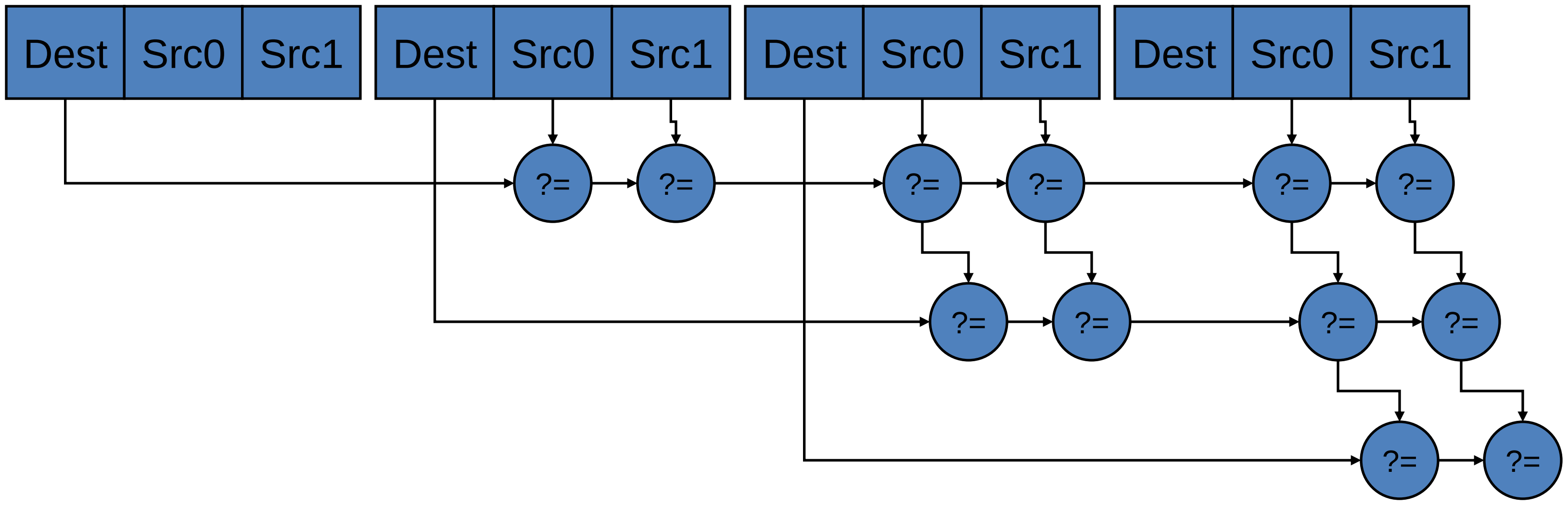
Parallel Decode

- Primary Tasks
  - Identify individual instructions (!)
  - Determine instruction types
  - Determine dependences between instructions
- Two important factors
  - Instruction set architecture
  - Pipeline width

Pentium Pro Fetch/Decode



Dependence Checking

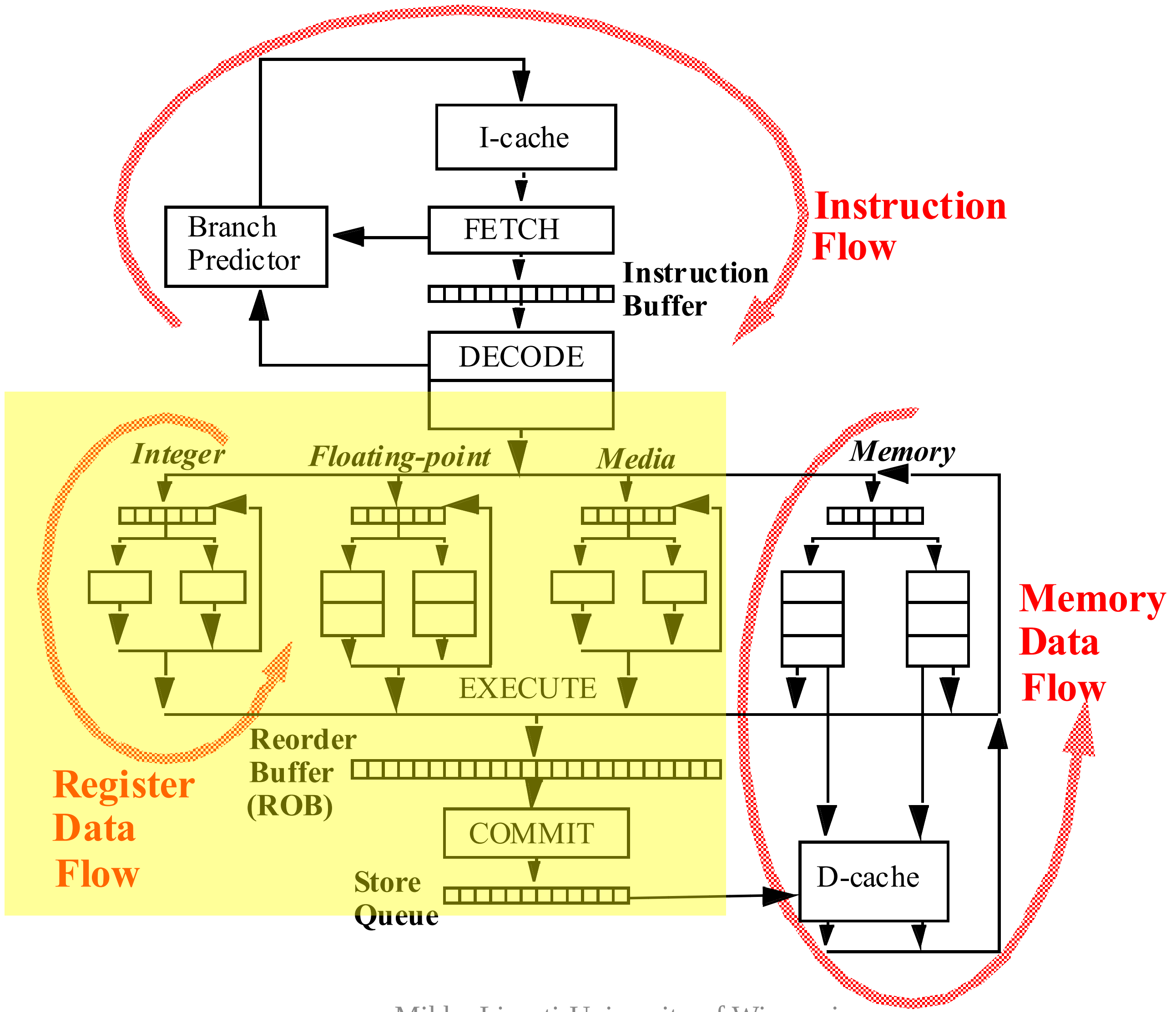


- Trailing instructions in fetch group
  - Check for dependence on leading instructions

Summary: Instruction Flow

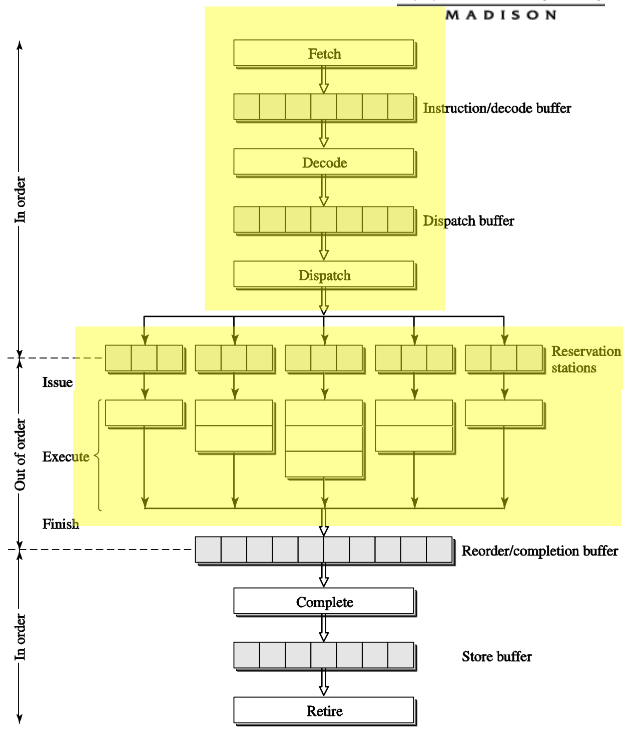
- Fetch group alignment
- Target address generation
  - Branch target buffer
- Branch condition prediction
- Speculative execution
  - Tagging/tracking instructions
  - Recovering from mispredicted branches
- Decoding in parallel

High-IPC Processor



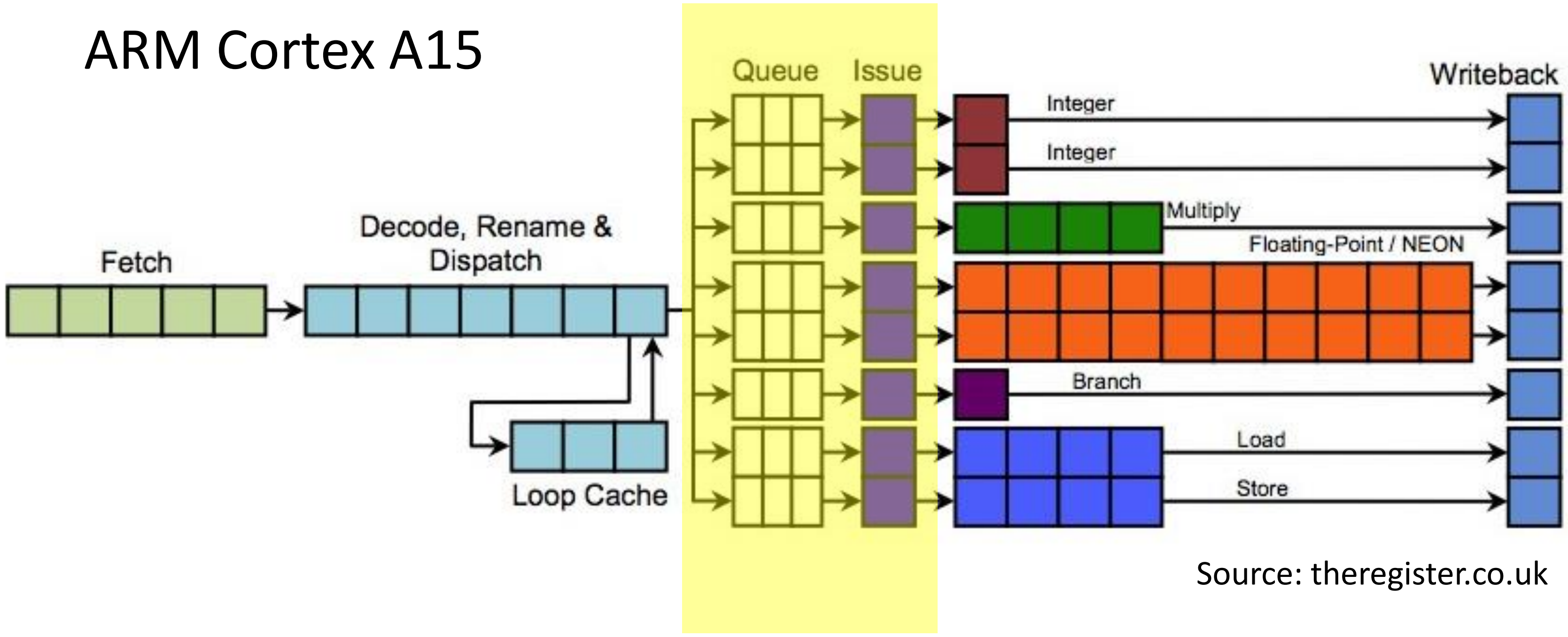
Register Data Flow

- Parallel pipelines
  - Centralized instruction fetch
  - Centralized instruction decode
- Diversified execution pipelines
  - Distributed instruction execution
- Data dependence linking
  - Register renaming to resolve true/false dependences
  - Issue logic to support out-of-order issue
  - Reorder buffer to maintain precise state



Issue Queues and Execution Lanes

ARM Cortex A15



Source: theregister.co.uk

Program Data Dependences

- True dependence (RAW)
  - j cannot execute until i produces its result

$$D(i) \cap R(j) \neq \phi$$

- Anti-dependence (WAR)
  - j cannot write its result until i has read its sources

$$R(i) \cap D(j) \neq \phi$$

- Output dependence (WAW)
  - j cannot write its result until i has written its result

$$D(i) \cap D(j) \neq \phi$$

Register Data Dependences

- Program data dependences cause hazards
  - True dependences (RAW)
  - Antidependences (WAR)
  - Output dependences (WAW)
- When are registers read and written?
  - Out of program order!
  - Hence, any and all of these can occur
- Solution to all three: register renaming

Register Renaming: WAR/WAW

- Widely employed (Core i7, Cortex A15, ...)
- Resolving WAR/WAW:
  - Each register write gets unique "rename register"
  - Writes are committed in program order at Writeback
  - WAR and WAW are not an issue
    - All updates to "architected state" delayed till writeback
    - Writeback stage always later than read stage
  - Reorder Buffer (ROB) enforces in-order writeback

```
Add R3 <= …          P32 <= …
Sub R4 <= …          P33 <= …
And R3 <= …          P35 <= …
```

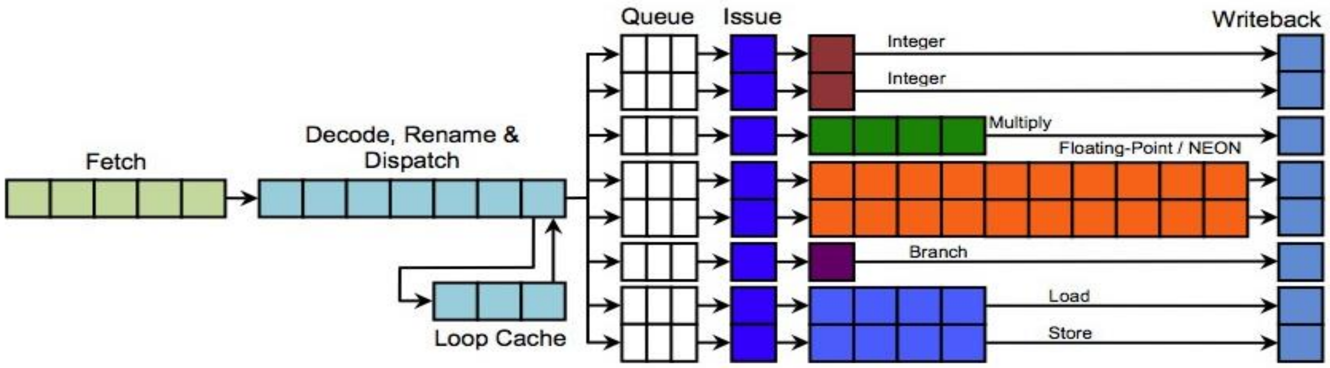
Register Renaming: RAW

- In order, at dispatch:
  - Source registers checked to see if "in flight"
    - Register map table keeps track of this
    - If not in flight, can be read from the register file
    - If in flight, look up "rename register" tag (IOU)
  - Then, allocate new register for register write

Add R3 <= R2 + R1	P32 <= P2 + P1
Sub R4 <= R3 + R1	P33 <= P32 + P1
And R3 <= R4 & R2	P35 <= P33 + P2

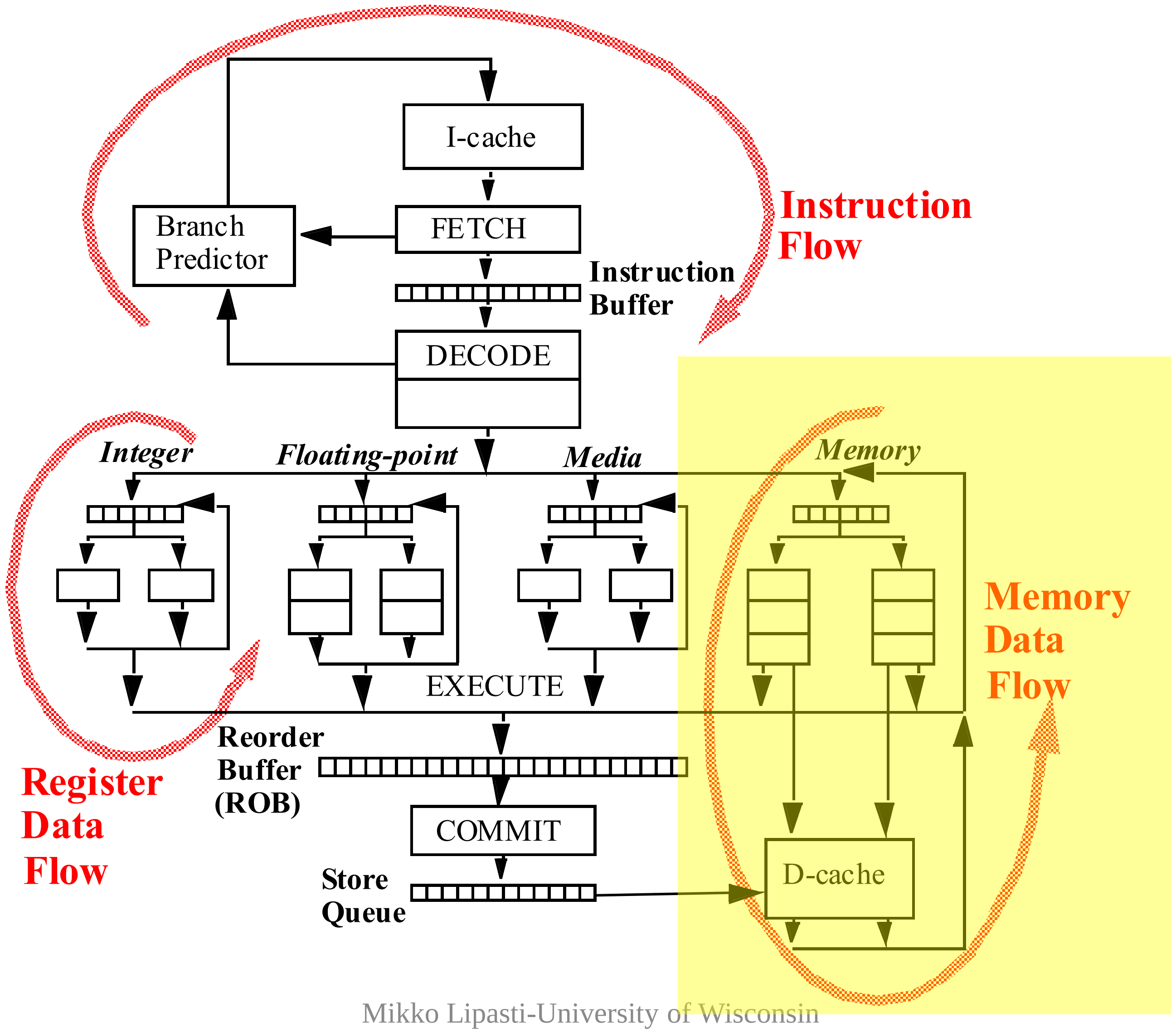
Register Renaming: RAW

- Advance instruction to instruction queue
  – Wait for rename register tag to trigger issue
- Issue queue/reservation station enables out-of-order issue
  – Newer instructions can bypass stalled instructions



Source: theregister.co.uk

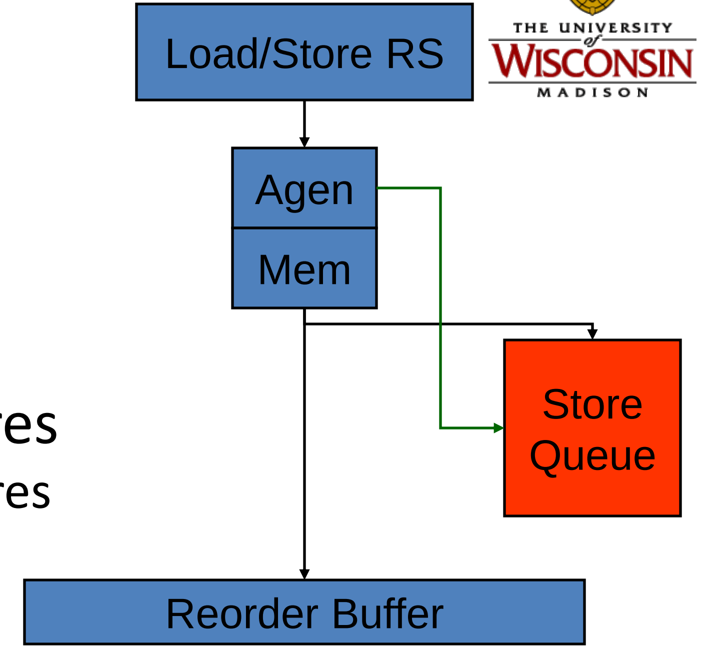
High-IPC Processor



Memory Data Flow

- Resolve WAR/WAW/RAW memory dependences
  - MEM stage can occur out of order
- Provide high bandwidth to memory hierarchy
  - Non-blocking caches

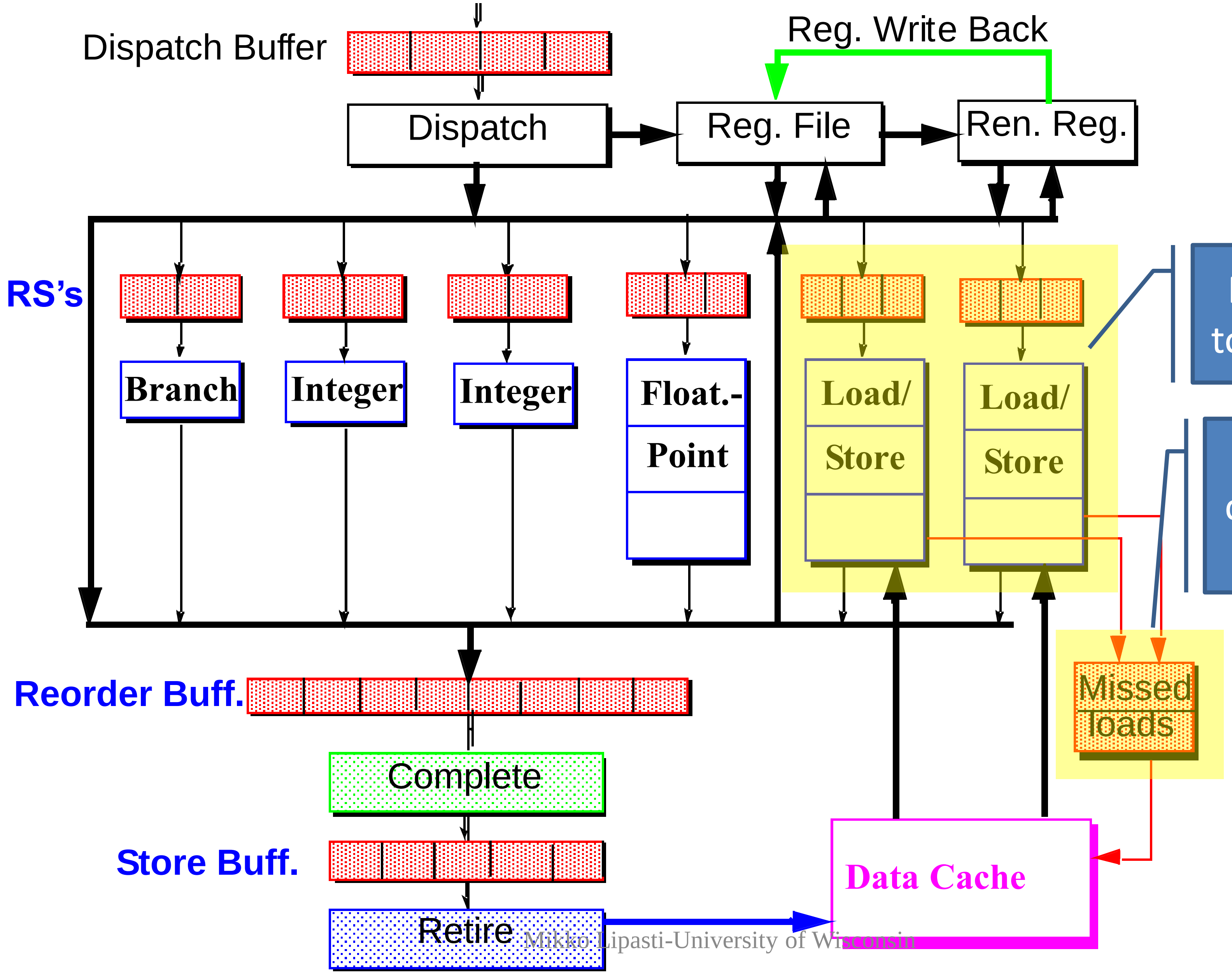
Memory Data Dependences



- WAR/WAW: stores commit in order
  - Hazards not possible.
- RAW: loads must check pending stores
  - Store queue keeps track of pending stores
  - Loads check against these addresses
  - Similar to register bypass logic
  - Comparators are 64 bits wide
  - Must consider position (age) of loads and stores

- Major source of complexity in modern designs
  - Store queue lookup is position-based
  - What if store address is not yet known?

Increasing Memory Bandwidth

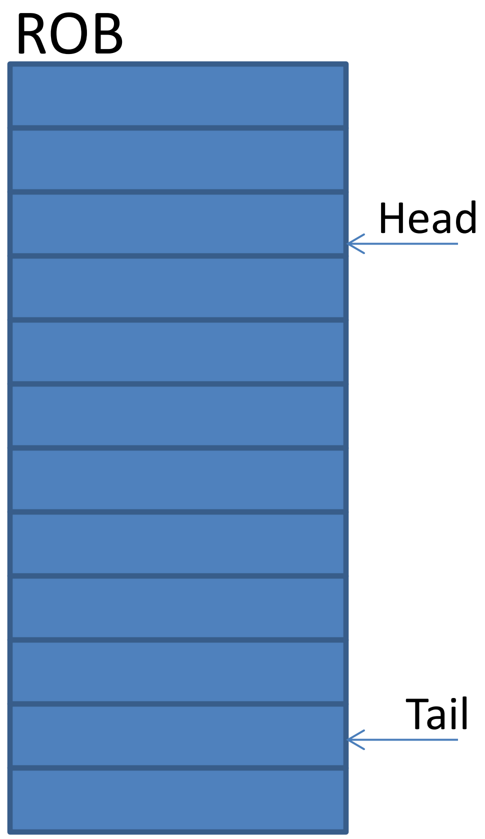


Expensive to duplicate

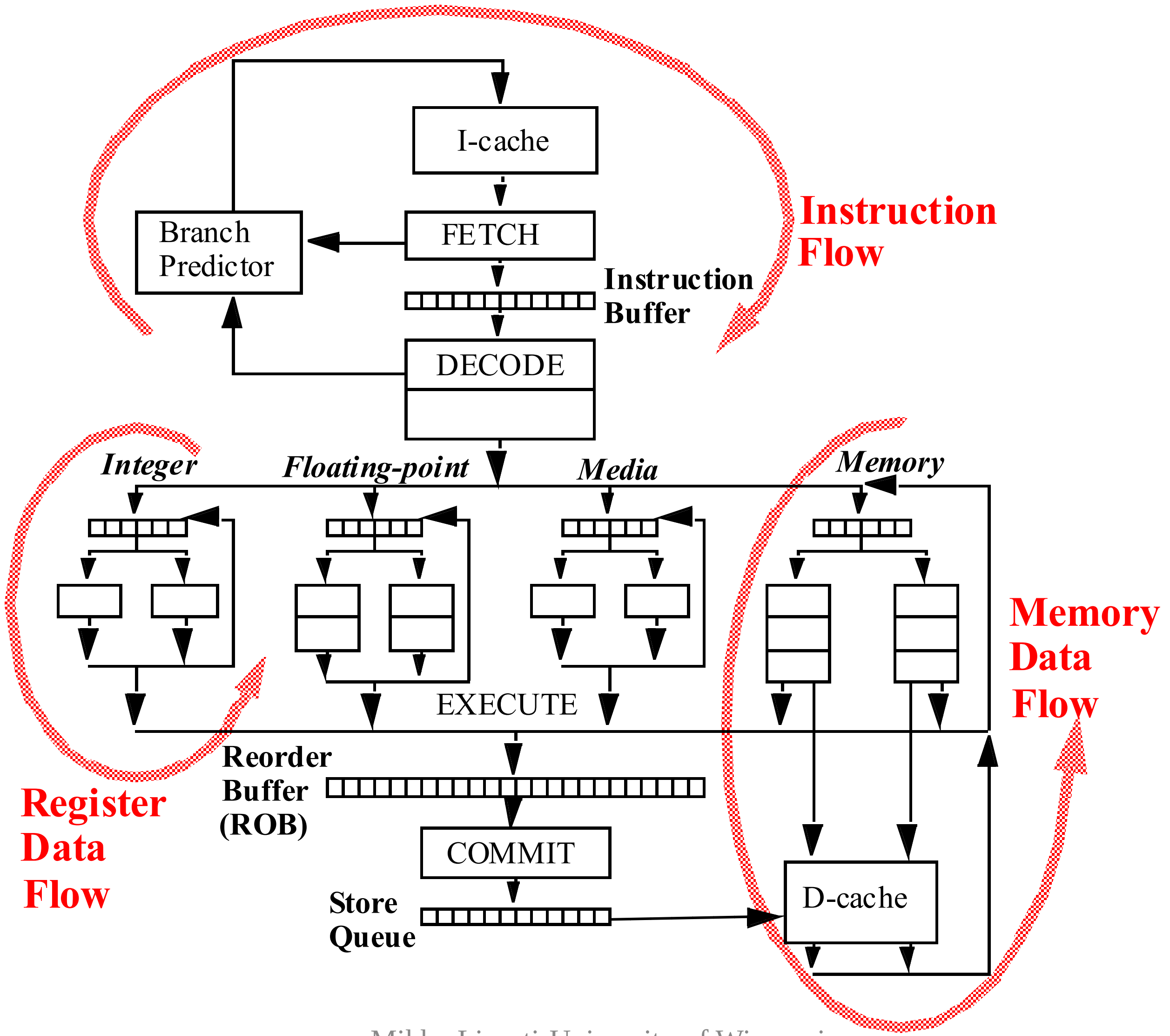
Complex, concurrent FSMs

Maintaining Precise State

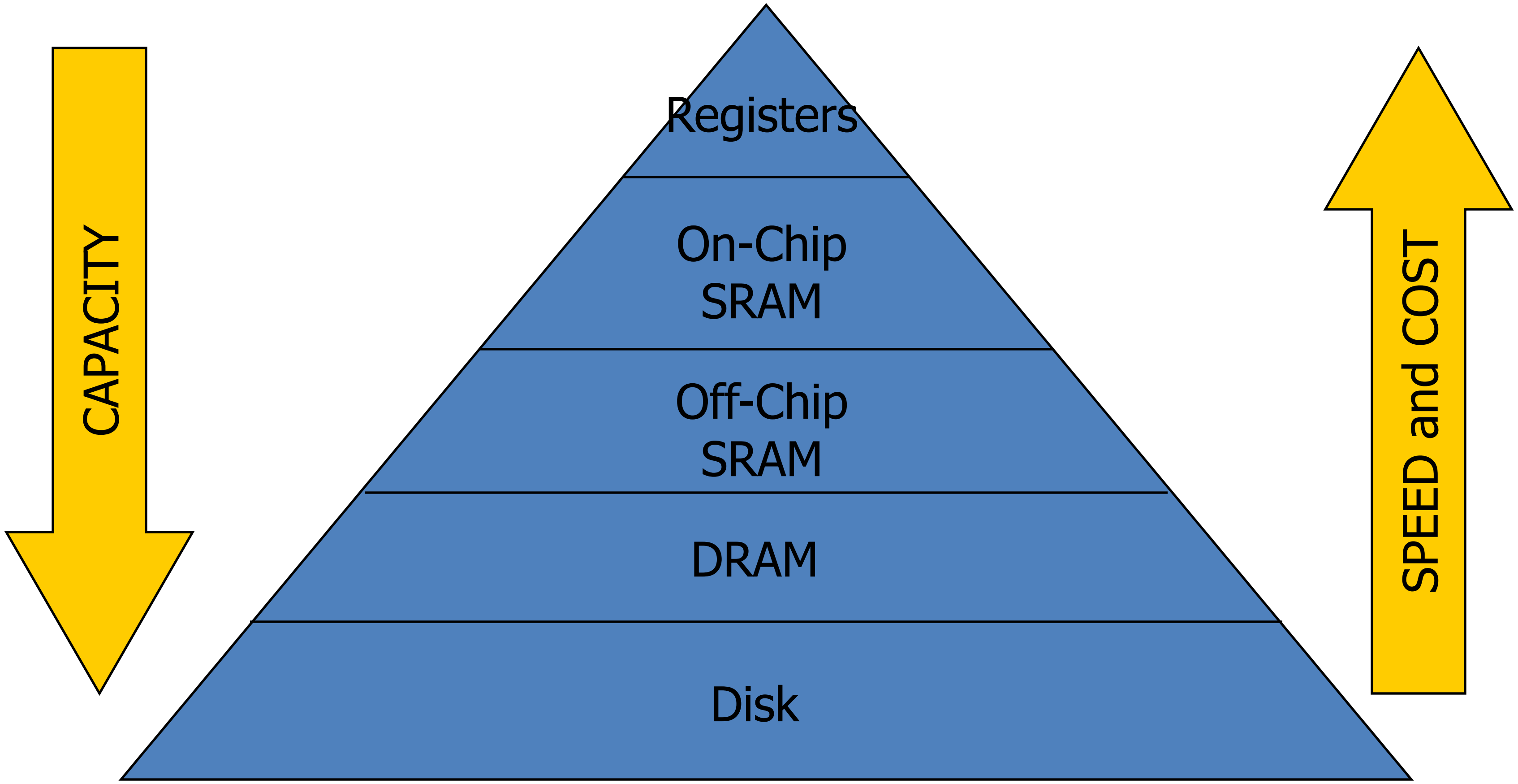
- Out-of-order execution
  - ALU instructions
  - Load/store instructions
- In-order completion/retirement
  - Precise exceptions
- Solutions
  - Reorder buffer retires instructions in order
  - Store queue retires stores in order
  - Exceptions can be handled at any instruction boundary by reconstructing state out of ROB/SQ



Summary: A High-IPC Processor



Memory Hierarchy



Why Memory Hierarchy?

- Need lots of bandwidth

$$BW = \frac{1.0inst}{cycle} \times \left[\frac{1Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4Dref}{inst} \times \frac{4B}{Dref} \right] \times \frac{1Gcycles}{\sec}$$
$$= \frac{5.6GB}{\sec}$$

- Need lots of storage
  - 64MB (minimum) to multiple TB
- Must be cheap per bit
  - (TB x anything) is a lot of money!
- These requirements seem incompatible

Why Memory Hierarchy?

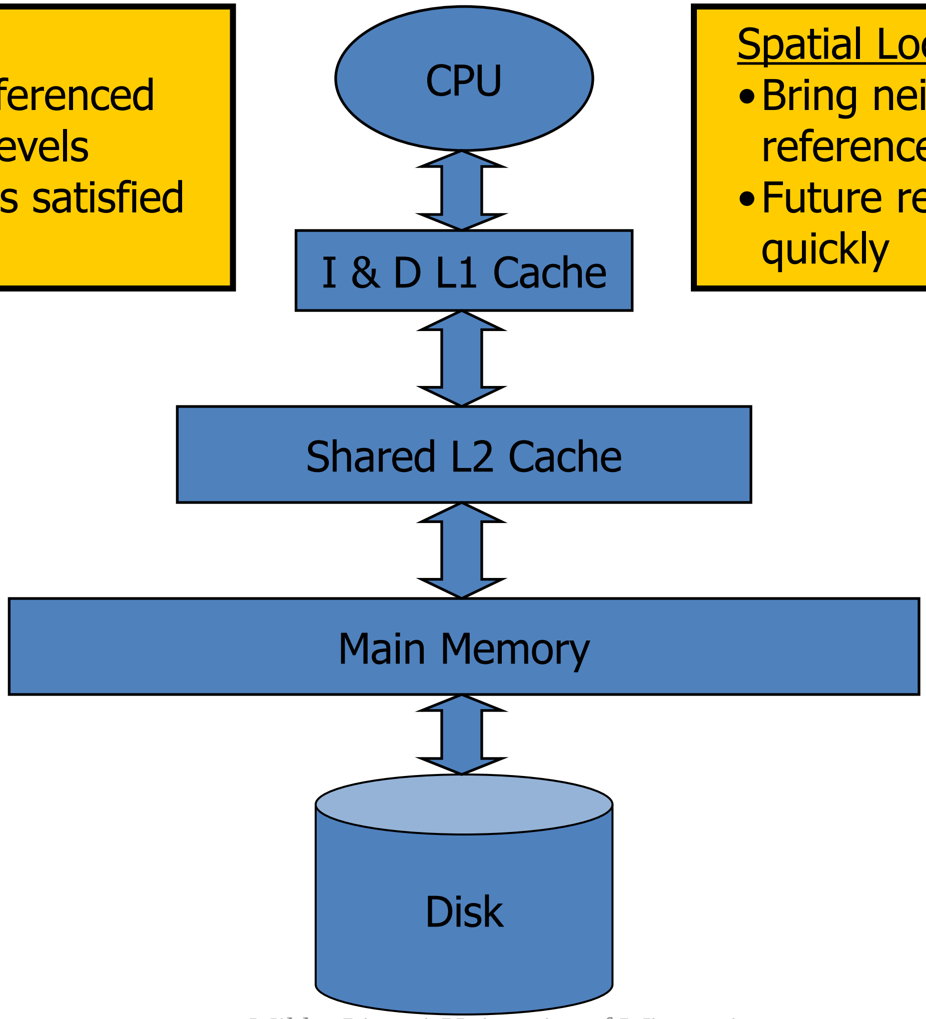
- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state
  - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

Why Does a Hierarchy Work?

- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

Memory Hierarchy

Temporal Locality
- Keep recently referenced items at higher levels
- Future references satisfied quickly



Spatial Locality
- Bring neighbors of recently referenced to higher levels
- Future references satisfied quickly

Four Burning Questions

- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?
- Consider these for caches
  - Built from SRAM, EDRAM, stacked DRAM

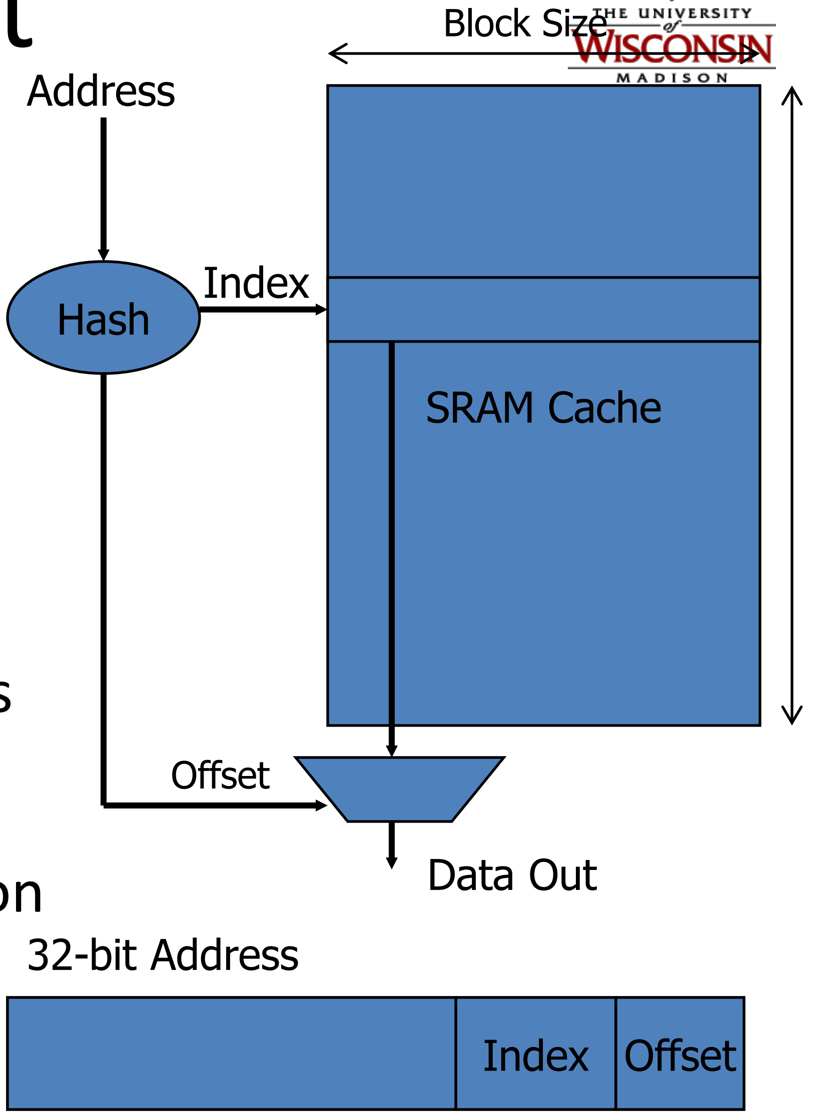
Placement

Memory Type	Placement	Comments
Registers	Anywhere; Int, FP, SPR	Compiler/programmer manages
Cache (SRAM)	Fixed in H/W	*Direct-mapped, set-associative, fully-associative*
DRAM	Anywhere	O/S manages
Disk	Anywhere	O/S manages

HUH?

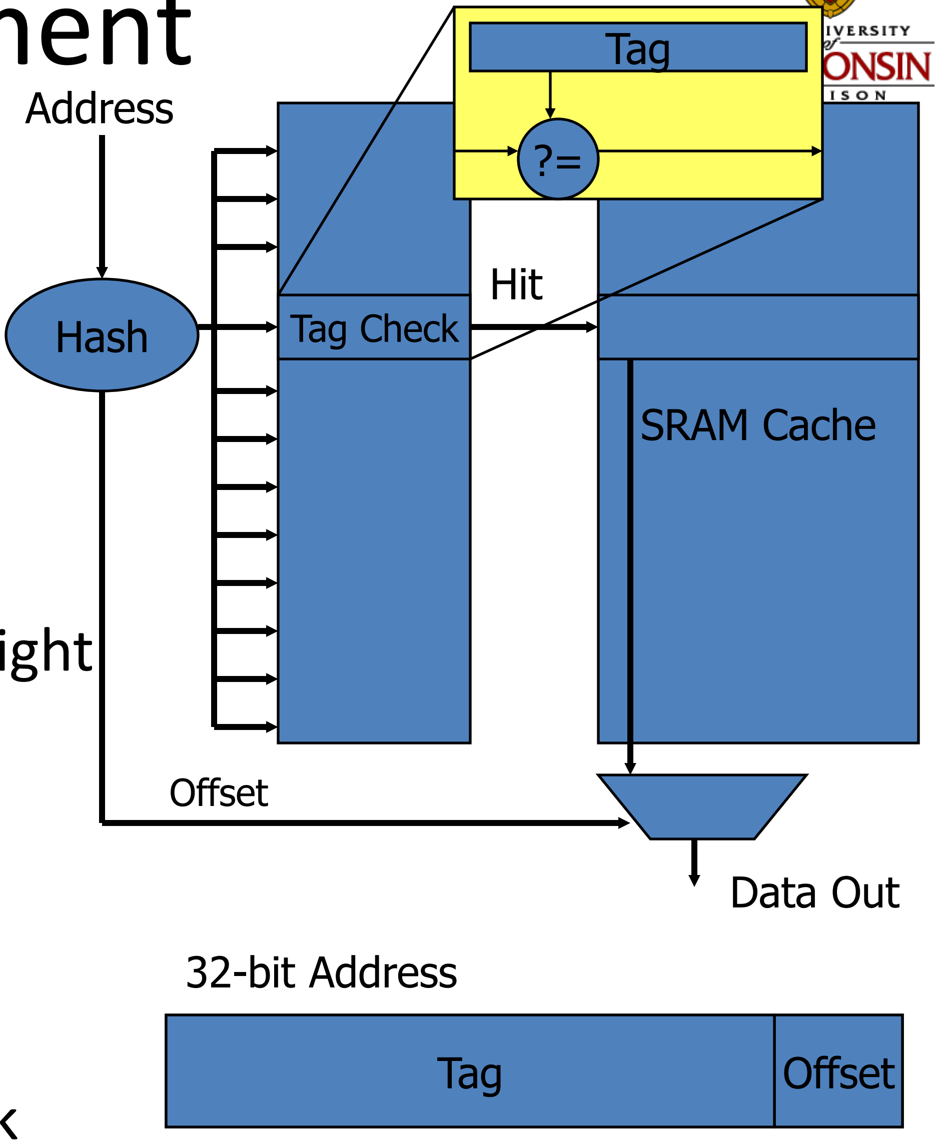
Placement

- Address Range
  - Exceeds cache capacity
- Map address to finite capacity
  - Called a *hash*
  - Usually just masks high-order bits
- *Direct-mapped*
  - Block can only exist in one location
  - Hash collisions cause problems



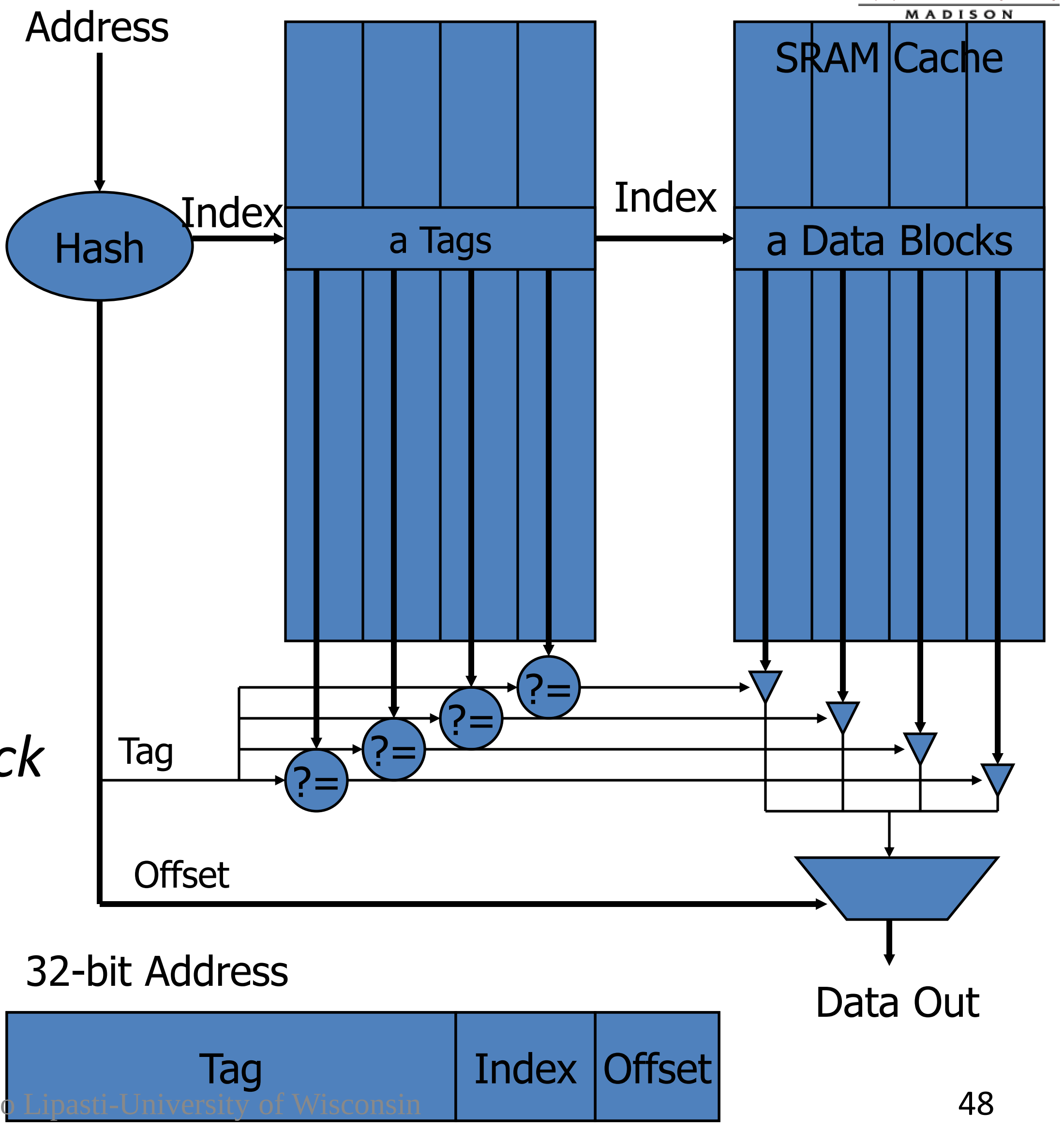
Placement

- *Fully-associative*
  - Block can exist anywhere
  - No more hash collisions
- *Identification*
  - How do I know I have the right block?
  - Called a *tag check*
    - Must store address tags
    - Compare against address
- Expensive!
  - Tag & comparator per block

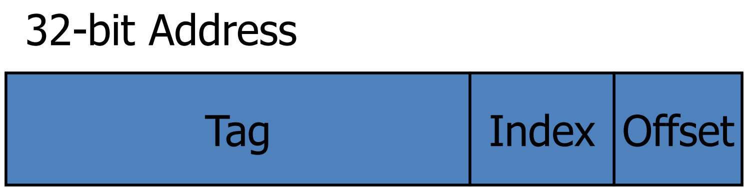


Placement

- *Set-associative*
  - Block can be in *a* locations
  - Hash collisions:
    - *a* still OK
- *Identification*
  - Still perform *tag check*
  - However, only *a* in parallel



Placement and Identification



Portion	Length	Purpose
Offset	$o=\log_2$(block size)	Select word within block
Index	$i=\log_2$(number of sets)	Select set of blocks
Tag	$t=32 - o - i$	ID block within set

- Consider: <BS=block size, S=sets, B=blocks>
  - <64,64,64>: o=6, i=6, t=20: direct-mapped (S=B)
  - <64,16,64>: o=6, i=4, t=22: 4-way S-A (S = B / 4)
  - <64,1,64>: o=6, i=0, t=26: fully associative (S=1)
- Total size = BS x B = BS x S x (B/S)

Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
- Same idea:
  - Move blocks to next level of cache

Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Several policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used)
  - NMRU (not most recently used)
  - Pseudo-random (yes, really!)
- Pick victim within *set* where a = *associativity*
  - If a <= 2, LRU is cheap and easy (1 bit)
  - If a > 2, it gets harder
  - Pseudo-random works pretty well for caches

Write Policy

- Memory hierarchy
  - 2 or more copies of same block
    - Main memory and/or disk
    - Caches
- What to do on a write?
  - Eventually, all copies must be changed
  - Write must *propagate* to all levels
    - And other processor's caches (later)

Write Policy

- Easiest policy: *write-through*
- Every write propagates directly through hierarchy
  - Write in L1, L2, memory, disk (?!?)
- Why is this a bad idea?
  - Very high bandwidth requirement
  - Remember, large memories are slow
- Popular in real systems only to the L2
  - Every write updates L1 and L2
  - Beyond L2, use *write-back* policy

Write Policy

- Most widely used: *write-back*
- Maintain *state* of each line in a cache
  - Invalid – not present in the cache
  - Clean – present, but not written (unmodified)
  - Dirty – present and written (modified)
- Store state in tag array, next to address tag
  - Mark dirty bit on a write
- On eviction, check dirty bit
  - If set, write back dirty line to next level
  - Called a *writeback* or *castout*

Write Policy

- Complications of write-back policy
  - Stale copies lower in the hierarchy
  - Must always check higher level for dirty copies before accessing copy in a lower level
- Not a big problem in uniprocessors
  - In multiprocessors: *the cache coherence problem*
- I/O devices that use DMA (direct memory access) can cause problems even in uniprocessors
  - Called coherent I/O
  - Must check caches for dirty copies before reading main memory

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
		0
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
		0

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss

Tag Array

Tag0	Tag1	LRU
		0
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss

Tag Array

Tag0	Tag1	LRU
10		1
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss

Tag Array

Tag0	Tag1	LRU
10	11	0
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10		1
11		1

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict
Store 0x29	101001	2/0	Hit/Dirty

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10 d		1
11		1

Cache Misses and Performance

- Miss penalty
  - Detect miss: 1 or more cycles
  - Find victim (replace block): 1 or more cycles
    - Write back if dirty
  - Request block from next level: several cycles
    - May need to **find** line from one of many caches (coherence)
  - Transfer block from next level: several cycles
    - (block size) / (bus width)
  - Fill block into data array, update tag array: 1+ cycles
  - Resume execution
- In practice: 6 cycles to 100s of cycles

Cache Miss Rate

- Determined by:
  - Program characteristics
    - Temporal locality
    - Spatial locality
  - Cache organization
    - Block size, associativity, number of sets

Cache Miss Rates: 3 C's [Hill]

- Compulsory miss
  - First-ever reference to a given block of memory
  - *Cold misses = m_c : number of misses for FA infinite cache*
- Capacity
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced
  - *Capacity misses = m_f - m_c : add'l misses for finite FA cache*
- Conflict
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as *capacity within set*
  - *Conflict misses = m_a - m_f : add'l misses in actual cache*

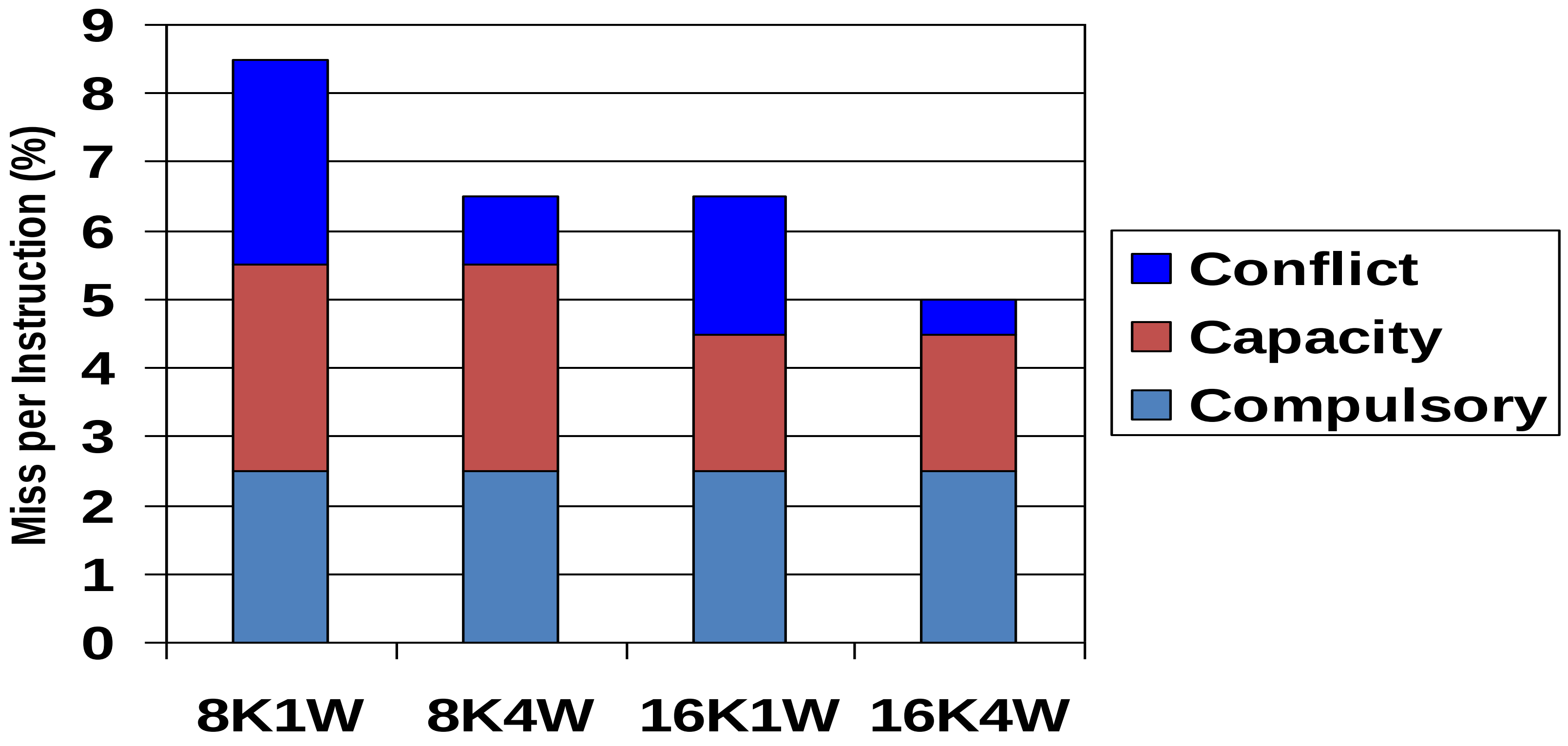
Cache Miss Rate Effects

- Number of blocks (sets x associativity)
  - Bigger is better: fewer conflicts, greater capacity
- Associativity
  - Higher associativity reduces conflicts
  - Very little benefit beyond 8-way set-associative
- Block size
  - Larger blocks exploit spatial locality
  - Usually: miss rates improve until 64B-256B
  - 512B or more miss rates get worse
    - Larger blocks less efficient: more capacity misses
    - Fewer placement choices: more conflict misses

Cache Miss Rate

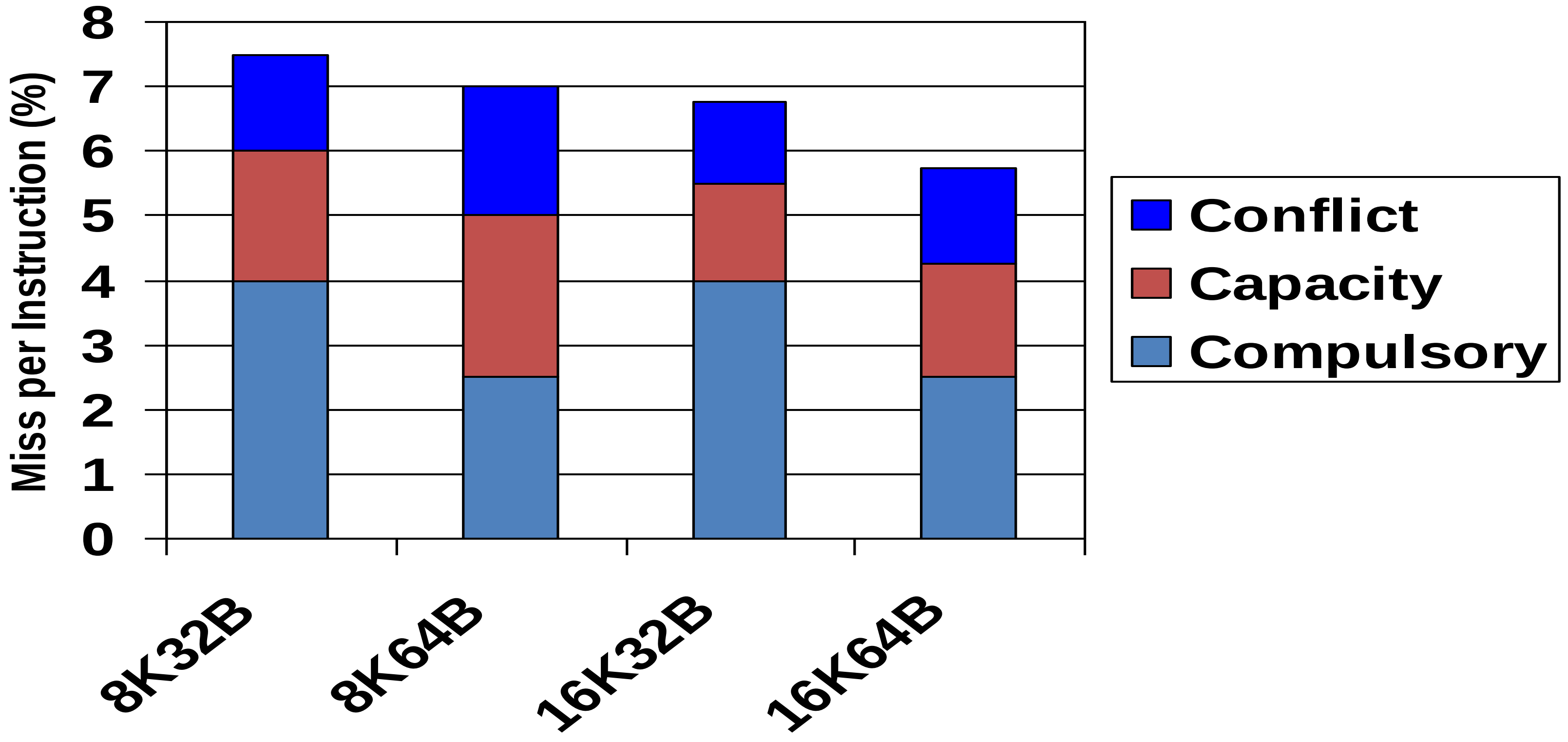
- Subtle tradeoffs between cache organization parameters
  - Large blocks reduce compulsory misses but increase miss penalty
    - #compulsory ~= (working set) / (block size)
    - #transfers = (block size)/(bus width)
  - Large blocks increase conflict misses
    - #blocks = (cache size) / (block size)
  - Associativity reduces conflict misses
  - Associativity increases access time
- Can associative cache ever have higher miss rate than direct-mapped cache of same size?

Cache Miss Rates: 3 C's



- Vary size and associativity
  - Compulsory misses are constant
  - Capacity and conflict misses are reduced

Cache Miss Rates: 3 C's

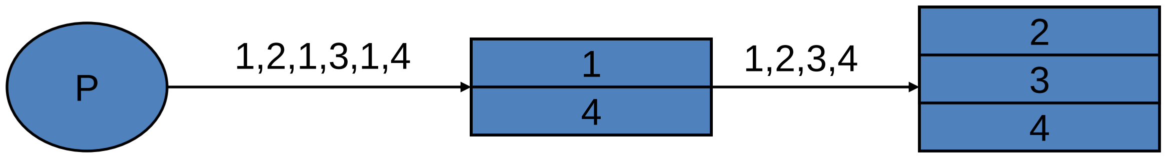


- Vary size and block size
  - Compulsory misses drop with increased block size
  - Capacity and conflict can increase with larger blocks

Multilevel Caches

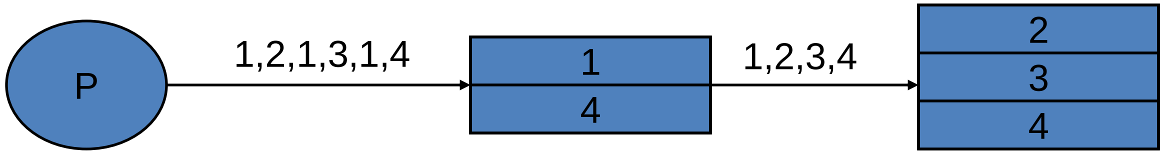
- Ubiquitous in high-performance processors
  - Gap between L1 (core frequency) and main memory too high
  - Level 2 usually on chip, level 3 on or off-chip, level 4 off chip
- Inclusion in multilevel caches
  - Multi-level inclusion holds if L2 cache is superset of L1
  - Can handle virtual address synonyms
  - Filter coherence traffic: if L2 misses, L1 needn't see snoop
  - Makes L1 writes simpler
    - For both write-through and write-back

Multilevel Inclusion



- Example: local LRU not sufficient to guarantee inclusion
  - Assume L1 holds two and L2 holds three blocks
  - Both use local LRU
- Final state: L1 contains 1, L2 does not
  - Inclusion not maintained
- Different block sizes also complicate inclusion

Multilevel Inclusion



- Inclusion takes effort to maintain
  - Make L2 cache have bits or pointers giving L1 contents
  - Invalidate from L1 before replacing from L2
  - In example, removing 1 from L2 also removes it from L1
- Number of pointers per L2 block
  - L2 blocksize/L1 blocksize
- Supplemental reading: [Wang, Baer, Levy ISCA 1989]

Multilevel Miss Rates

- Miss rates of lower level caches
  - Affected by upper level filtering effect
  - LRU becomes LRM, since "use" is "miss"
  - Can affect miss rates, though usually not important
- Miss rates reported as:
  - Miss per instruction
  - Global miss rate
  - Local miss rate
  - "Solo" miss rate
    - L2 cache sees all references (unfiltered by L1)

Cache Design: Four Key Issues

- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - **Replacement**
    - **How do I make space for new blocks?**
  - Write Policy
    - How do I propagate changes?
- Consider these for caches
  - Usually SRAM
- Also apply to main memory, disks

Replacement

- Cache has finite size
  - What do we do when it is full?
- Analogy: desktop full?
  - Move books to bookshelf to make room
  - Bookshelf full? Move least-used to library
  - Etc.
- Same idea:
  - Move blocks to next level of cache

Replacement

- How do we choose *victim*?
  - Verbs: *Victimize, evict, replace, cast out*
- Many policies are possible
  - FIFO (first-in-first-out)
  - LRU (least recently used), pseudo-LRU
  - LFU (least frequently used)
  - NMRU (not most recently used)
  - NRU
  - Pseudo-random (yes, really!)
  - Optimal
  - Etc

Optimal Replacement Policy?

[Belady, IBM Systems Journal, 1966]

- Evict block with longest reuse distance
  - i.e. next reference to block is farthest in future
  - Requires knowledge of the future!
- Can’t build it, but can model it with trace
  - Process trace in reverse
  - [Sugumar&Abraham] describe how to do this in one pass over the trace with some lookahead (Cheetah simulator)
- Useful, since it reveals ***opportunity***
  - (X,A,B,C,D,X): LRU 4-way SA $, 2nd X will miss

Least-Recently Used

- For a=2, LRU is equivalent to NMRU
  - Single bit per set indicates LRU/MRU
  - Set/clear on each access
- For a>2, LRU is difficult/expensive
  - Timestamps? How many bits?
    - Must find min timestamp on each eviction
  - Sorted list? Re-sort on every access?
- List overhead: $\log_2(a)$ bits /block
  - Shift register implementation

True LRU Shortcomings

- Streaming data/scans: $x_0, x_1, \ldots, x_n$
  - Effectively no temporal reuse
- Thrashing: *reuse distance > a*
  - Temporal reuse exists but LRU fails
- All blocks march from MRU to LRU
  - Other conflicting blocks are pushed out
- For *n>a* no blocks remain after scan/thrash
  - Incur many conflict misses after scan ends
- Pseudo-LRU sometimes helps a little bit

Segmented or Protected LRU

[I/O: Karedla, Love, Wherry, IEEE Computer 27(3), 1994]

[Cache: Wilkerson, Wade, US Patent 6393525, 1999]

- Partition LRU list into *filter* and *reuse* lists
- On insert, block goes into *filter* list
- On reuse (hit), block promoted into *reuse* list
- Provides scan & some thrash resistance
  - Blocks without reuse get evicted quickly
  - Blocks with reuse are protected from scan/thrash blocks
- No storage overhead, but LRU update slightly more complicated

Protected LRU: LIP

- Simplified variant of this idea: LIP
  – Qureshi et al. ISCA 2007
- Insert new blocks into LRU position, not MRU position
  – *Filter list* of size 1, *reuse list* of size (a-1)
- Do this adaptively: DIP
- Use *set dueling* to decide LIP vs. LRU
  – 1 (or a few) set uses LIP vs. 1 that uses LRU
  – Compare hit rate for sets
  – Set policy for all other sets to match best set

Not Recently Used (NRU)

- Keep NRU state in 1 bit/block
  - Bit is set to 0 when installed (assume reuse)
  - Bit is set to 0 when referenced (reuse observed)
  - Evictions favor NRU=1 blocks
  - If all blocks are NRU=0
    - Eviction forces all blocks in set to NRU=1
    - Picks one as victim (can be pseudo-random, or rotating, or fixed left-to-right)
- Simple, similar to virtual memory clock algorithm
- Provides some scan and thrash resistance
  - Relies on "randomizing" evictions rather than strict LRU order
- Used by Intel Itanium, Sparc T2

Least Frequently Used

- Counter per block, incremented on reference
- Evictions choose lowest count
  - Logic not trivial (a^2 comparison/sort)
- Storage overhead
  - 1 bit per block: same as NRU
  - How many bits are helpful?

Pitfall: Cache Filtering Effect

- Upper level caches (L1, L2) hide reference stream from lower level caches
- Blocks with “no reuse” @ LLC could be very hot (never evicted from L1/L2)
- Evicting from LLC often causes L1/L2 eviction (due to inclusion)
- Could hurt performance even if LLC miss rate improves

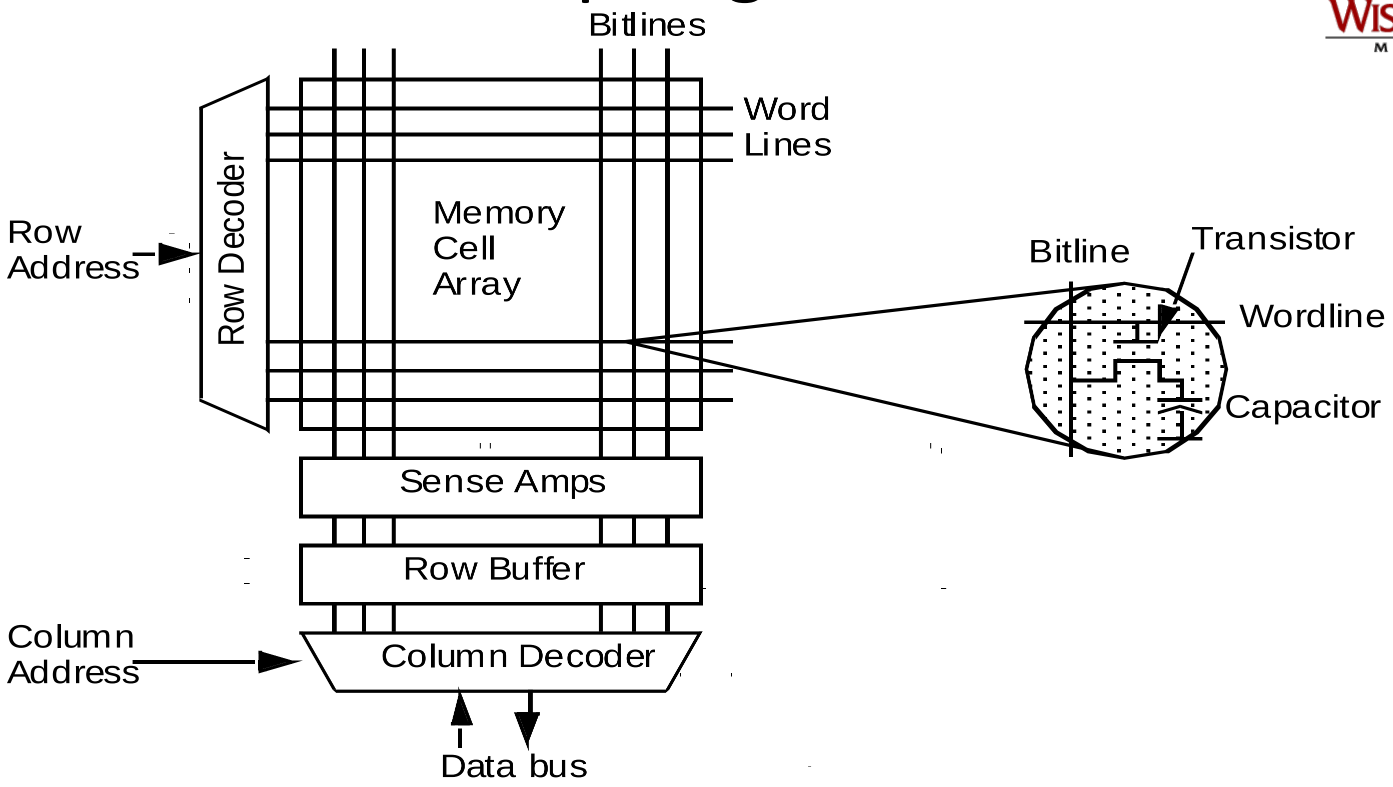
Replacement Policy Summary

- Replacement policies affect *capacity* and *conflict* misses
- Policies covered:
  - Belady's optimal replacement
  - Least-recently used (LRU)
  - Practical pseudo-LRU (tree LRU)
  - Protected LRU
    - LIP/DIP variant
    - *Set dueling* to dynamically select policy
  - Not-recently-used (NRU) or *clock* algorithm
  - Least frequently used (LFU)

Main Memory

- DRAM chips
- Memory organization
  - Interleaving
  - Banking
- Memory controller design

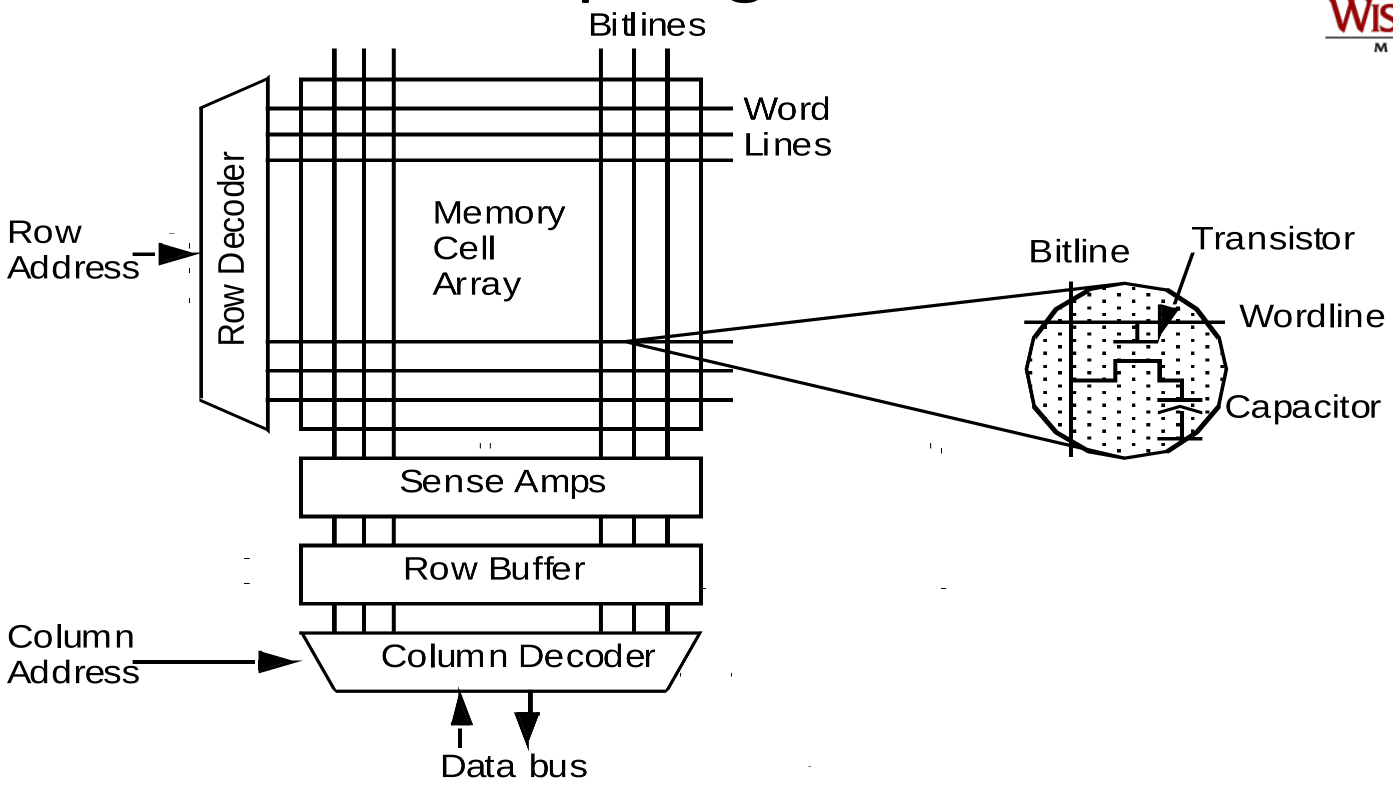
DRAM Chip Organization



- Optimized for density, not speed
- Data stored as charge in capacitor
- Discharge on reads => destructive reads
- Charge leaks over time
  - refresh every 64ms

- Cycle time roughly twice access time
- Need to precharge bitlines before access

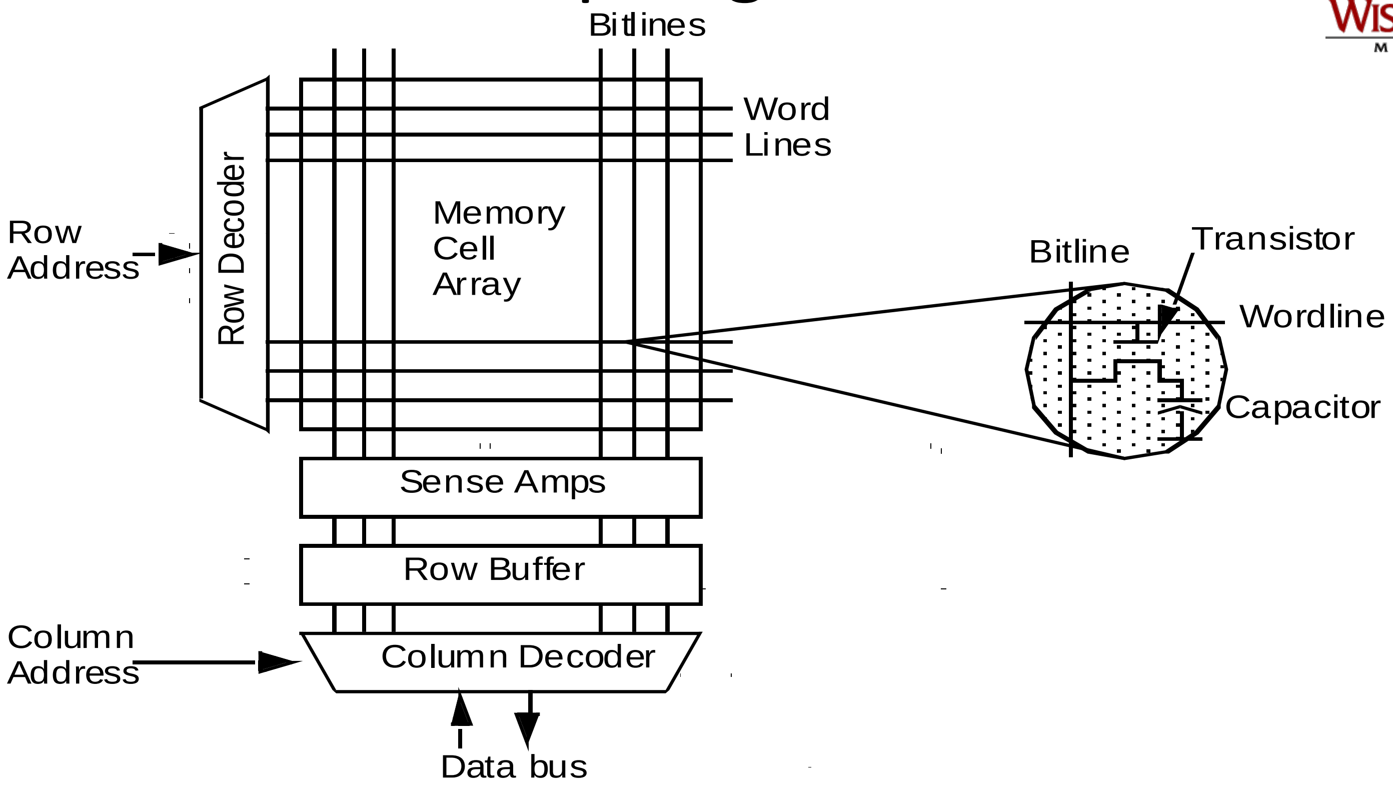
DRAM Chip Organization



- Current generation DRAM
  - 8Gbit @25nm
  - 266 MHz synchronous interface
  - Data clock 4x (1066MHz), double-data rate so 2133 MT/s

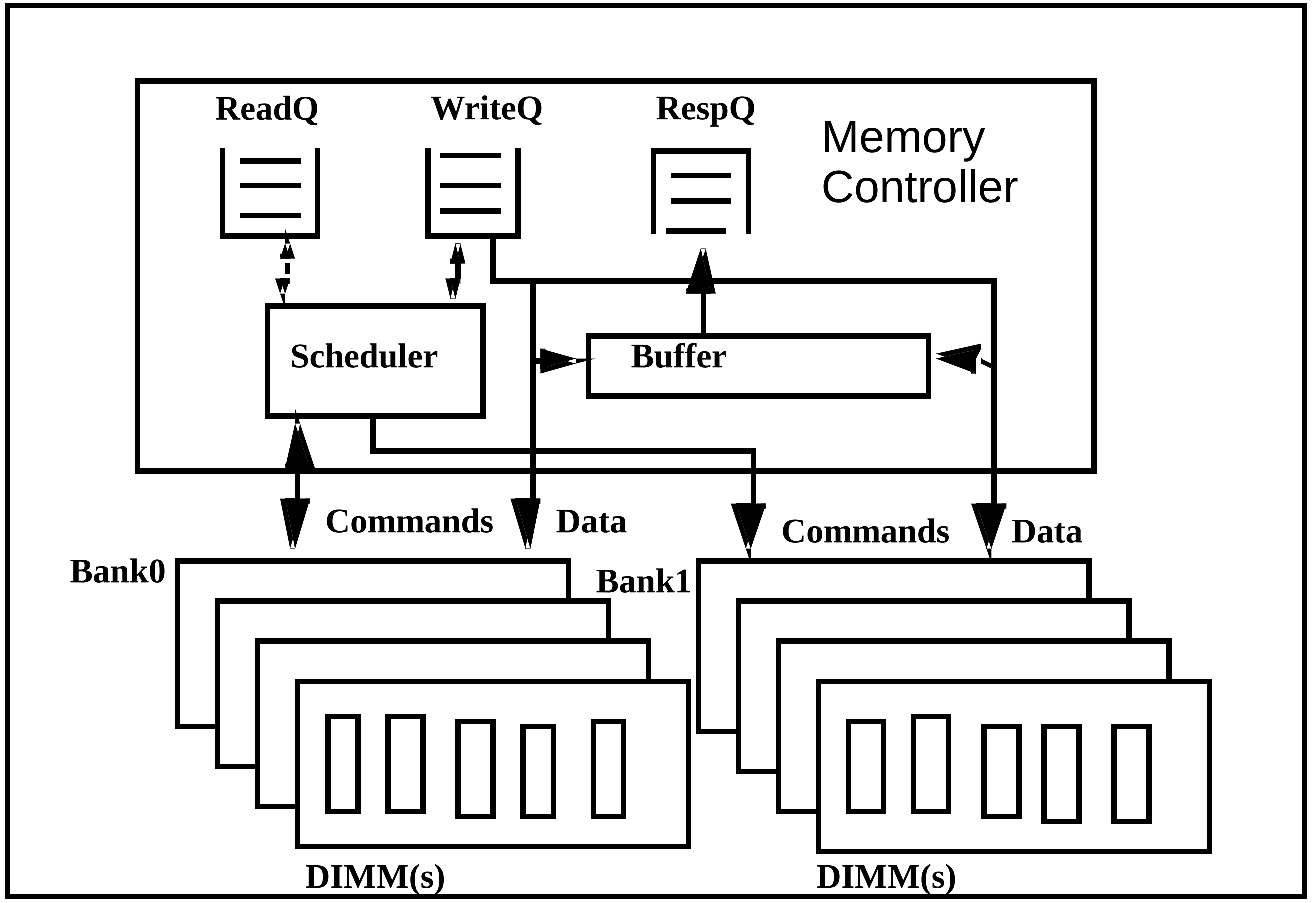
- Address pins are time-multiplexed
  - Row address strobe (RAS)
  - Column address strobe (CAS)

DRAM Chip Organization



- New RAS results in:
  - Bitline precharge
  - Row decode, sense
  - Row buffer write (up to 8K)
- New CAS
  - Read from row buffer
  - Much faster (3x)
- Streaming row accesses desirable

Memory Controller Organization



Memory Controller Organization

- **ReadQ**
  - Buffers multiple reads, enables scheduling optimizations
- **WriteQ**
  - Buffers writes, allows reads to bypass writes, enables scheduling opt.
- **RespQ**
  - Buffers responses until bus available
- **Scheduler**
  - FIFO? Or schedule to maximize for row hits (CAS accesses)
  - Scan queues for references to same page
  - Looks similar to issue queue with page number broadcasts for tag match
- **Buffer**
  - Builds transfer packet from multiple memory words to send over processor bus

Lecture Summary

- Brief review: High-IPC, out-of-order processors
  - Instruction flow
  - Register Dataflow
  - Memory Dataflow
- Caches and Memory Hierarchy
- Main memory (DRAM)