

期末复习

考试题型

- 不定向选择 - 30分
 - 共6题，每小题5分
- 简答题 - 20分
 - 共4题，每小题5分
- 问答题 - 50分
 - 前2题每题10分
 - 后2题每题15分

Performance Metrics

Basic Performance Equation

- Latency = seconds / program =
 - (instructions / program) * (cycles / instruction) * (seconds / cycle)
- **Instructions / program**: dynamic instruction count
 - Function of program, compiler, instruction set architecture (ISA)
- **Cycles / instruction**: CPI
 - Function of program, compiler, ISA, micro-architecture
- **Seconds / cycle**: clock period
 - Function of micro-architecture, technology parameters
- Optimize each component
 - **this class focuses mostly on CPI (caches, parallelism)**
 - ...but some on dynamic instruction count (compiler, ISA)
 - ...and some on clock frequency (pipelining, technology)

Cycles per Instruction (CPI) and IPC

- **CPI**: Cycle/instruction **on average**
 - **IPC** = $1/\text{CPI}$
 - Used more frequently than CPI
 - Favored because “bigger is better”, but harder to compute with
 - Different instructions have different cycle costs
 - E.g., “add” typically takes 1 cycle, “divide” takes >10 cycles
 - Depends on **relative instruction frequencies**
- CPI example
 - A program executes equal: integer, floating point (FP), memory ops
 - Cycles per instruction type: integer = 1, memory = 2, FP = 3
 - What is the CPI? $(33\% * 1) + (33\% * 2) + (33\% * 3) = 2$
 - **Warning**: this sort of calculation ignores many effects
 - Back-of-the-envelope arguments only

CPI Example

- Assume a processor with instruction frequencies and costs
 - Integer ALU: 50%, 1 cycle
 - Load: 20%, 5 cycle
 - Store: 10%, 1 cycle
 - Branch: 20%, 2 cycle
- Which change would improve performance more?
 - A. "Branch prediction" to reduce branch cost to 1 cycle?
 - B. Faster data memory to reduce load cost to 3 cycles?
- Compute CPI
 - Base = $0.5 * 1 + 0.2 * 5 + 0.1 * 1 + 0.2 * 2 = 2$ CPI

Frequency as a performance metric

- 1 Hertz = 1 cycle per second
1 Ghz is 1 cycle per nanosecond, 1 Ghz = 1000 Mhz
- (Micro-)architects often ignore dynamic instruction count...
- ... but general public (mostly) also ignores CPI
 - and instead equate **clock frequency** with performance!
- Which processor would you buy?
 - Processor A: CPI = 2, clock = 5 GHz
 - Processor B: CPI = 1, clock = 3 GHz
 - Probably A, but B is faster (assuming same ISA/compiler)
- **partial performance metrics are dangerous!**

Amdahl's Law

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

How much will an optimization improve performance?

P = proportion of running time affected by optimization

S = speedup

What if I speedup 25% of a program's execution by 2x?

1.14x speedup

What if I speedup 25% of a program's execution by ∞ ?

1.33x speedup

ISA

What Is An ISA?

- **ISA (instruction set architecture)**
 - A well-defined hardware/software interface
 - The **“contract”** between software and hardware
 - **Functional definition** of storage locations & operations
 - Storage locations: registers, memory
 - Operations: add, multiply, branch, load, store, etc
 - **Precise description** of how to invoke & access them
- Not in the “contract”: non-functional aspects
 - How operations are implemented
 - Which operations are fast and which are slow
 - Which operations take more power and which take less

Some Key Attributes of ISAs

- Instruction encoding
 - Fixed length (32-bit for MIPS)
 - Variable length (x86 1 byte to 16 bytes, average of ~ 3 bytes)
 - Limited variability (ARM insns are 16b or 32b)
- Number and type of registers
 - MIPS has 32 “integer” registers and 32 “floating point” registers
 - ARM & x86 both have 16 “integer” regs and 16 “floating point” regs
- Address space
 - ARM: 32-bit addresses at 8-bit granularly (4GB total)
 - Modern x86 and ARM64: 64-bit addresses (16 exabytes!)
- Memory addressing modes
 - MIPS: address calculated by “reg+offset”
 - x86 and others have much more complicated addressing modes

CISC vs. RISC

- Complex Instruction Set Computer (CISC)
 - A single instruction can be used to do all of the loading, evaluating and storing operations
 - Minimise the number of instructions per programme
 - **Increases the number of cycles per instruction**
 - E.g., x86
- Reduced Instruction Set Computer (RISC)
 - Instruction set is composed of a few basic steps for loading, evaluating and storing operations
 - Reduces the number of cycles per instruction
 - **Increase the number of instructions per program**
 - E.g., MIPS, ARM, RISC-V etc

CISC vs. RISC

- RISC

- Emphasis on software
- Small number of fixed length instructions
- Simple, standardised instructions
- Single clock cycle instructions
- Low cycles per insn with large code sizes

- CISC

- Emphasis on hardware
- Large number of instructions
- Complex, variable-length instructions
- Instructions can take several clock cycles
- Small code sizes with high cycles per insn

RISC vs. CISC: Which is better? (x86 is CISC, but transfers each insn to several micro-insns...)

Pipelining

Example Pipeline Perf. Calculation

- Single-cycle
 - Clock period = 50ns, CPI = 1
 - Performance = 50ns/insn
- 5-stage pipelined
 - Clock period = **12ns** approx. (50ns / 5 stages) + overheads
 - + CPI = **1** (each insn takes 5 cycles, but 1 completes each cycle)
 - + Performance = **12ns/insn**
 - Well actually ... CPI = 1 + some penalty for pipelining (next)
 - CPI = **1.5** (on average insn completes every 1.5 cycles)
 - Performance = **18ns/insn**
 - Much higher performance than single-cycle

Q1: Why Is Pipeline Clock Period ...

- ... > (delay thru datapath) / (number of pipeline stages)?
 - Three reasons:
 - Registers add delay
 - Pipeline stages have different delays, clock period is **max** delay
 - Extra datapaths for pipelining (bypassing paths)
 - These factors have implications for ideal number pipeline stages
 - Diminishing clock frequency gains for longer (deeper) pipelines

Q2: Why Is Pipeline CPI...

- ... > 1?
 - CPI for scalar in-order pipeline is 1 + **stall penalties**
 - Stalls used to resolve hazards
 - **Hazard**: condition that jeopardizes sequential illusion
 - **Stall**: pipeline delay introduced to restore sequential illusion
- Calculating pipeline CPI
 - **Frequency of stall * stall cycles**
 - Penalties add (stalls typically don't overlap in in-order pipelines)
 - $1 + (\text{stall-freq}_1 * \text{stall-cyc}_1) + (\text{stall-freq}_2 * \text{stall-cyc}_2) + \dots$
- Correctness/performance/make common case fast
 - Long penalties OK if they are rare, e.g., $1 + (0.01 * 10) = 1.1$
 - Stalls also have implications for ideal number of pipeline stages

Data Dependences

- Types of data dependences
 - Flow dependence (true data dependence – read after write, **RAW**)
 - Output dependence (write after write, **WAW**)
 - Anti dependence (write after read, **WAR**)
- Which ones cause stalls in a pipelined machine?
 - For all of them, we need to ensure semantics of the program is correct
 - Flow dependences always need to be obeyed because they constitute true dependence on a value
 - Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value
 - We will later see what we can do about them

Dependence types

- **RAW** (Read After Write) = “true dependence” (true)

mul r0 * r1 → **r2**

...

add **r2** + r3 → r4



- **WAW** (Write After Write) = “output dependence” (false)

mul r0 * r1 → **r2**

...

add r1 + r3 → **r2**



- **WAR** (Write After Read) = “anti-dependence” (false)

mul r0 * **r1** → r2

...

add r3 + r4 → **r1**



- WAW & WAR are “false”, Can be **totally eliminated** by “renaming”

Register Renaming


- WAR and WAW dependencies are not true dependencies
 - WHY? The same register refers to values that have nothing to do with each other
 - **They exist because not enough register ID's (i.e. names) in the ISA**
- How register renaming eliminates WAR and WAW dependencies?
 - Require: enough registers

Fixing Register Data Hazards

- Can only read register value three cycles after writing it
- **Option #1: make sure programs don't do it**
 - Compiler puts two independent insns between write/read insn pair
 - If they aren't there already
 - Independent means: "do not interfere with register in question"
 - Do not write it: otherwise meaning of program changes
 - Do not read it: otherwise create new data hazard
 - **Code scheduling**: compiler moves existing insns to do this
 - If none can be found, must use **nops** (no-operation)
- This is called **software interlocks**
 - **MIPS**: **M**icroprocessor w/out **I**nterlocking **P**ipeline **S**tages

Software Interlock Example

```
add $3, $2, $1
nop
nop
lw $4, 8($3)
sw $7, 8($3)
sub $6, $2, $8
addi $3, $5, 4
```



- Can any of last three insns be scheduled between first two
 - `sw $7, 8($3)`? No, creates hazard with `add $3, $2, $1`
 - `sub $6, $2, $8`? Okay
 - `addi $3, $5, 4`? No, `lw` would read \$3 from it
 - Still need one more insn, use `nop`

```
add $3, $2, $1
sub $6, $2, $8
nop
lw $4, 8($3)
sw $7, 8($3)
addi $3, $5, 4
```

Software Interlock Performance

- Assume
 - Branch: 20%, load: 20%, store: 10%, other: 50%
- For software interlocks, let's assume:
 - 20% of insns require insertion of 1 `nop`
 - 5% of insns require insertion of 2 `nops`
- Result:
 - CPI is still 1 technically
 - But now there are more insns
 - $\#insns = 1 + 0.20*1 + 0.05*2 = \mathbf{1.3}$
 - **30% more insns (30% slowdown) due to data hazards**

Hardware Interlocks

- Problem with software interlocks? Not compatible
 - Where does **3** in “read register 3 cycles after writing” come from?
 - From structure (depth) of pipeline
 - What if next MIPS version uses a 7-stage pipeline?
 - Programs compiled assuming 5-stage pipeline won't work!
- **Option #2: hardware interlocks**
 - Processor detects data hazards and fixes them
 - Resolves the above compatibility concern
 - Two aspects to this
 - Detecting hazards
 - Fixing hazards

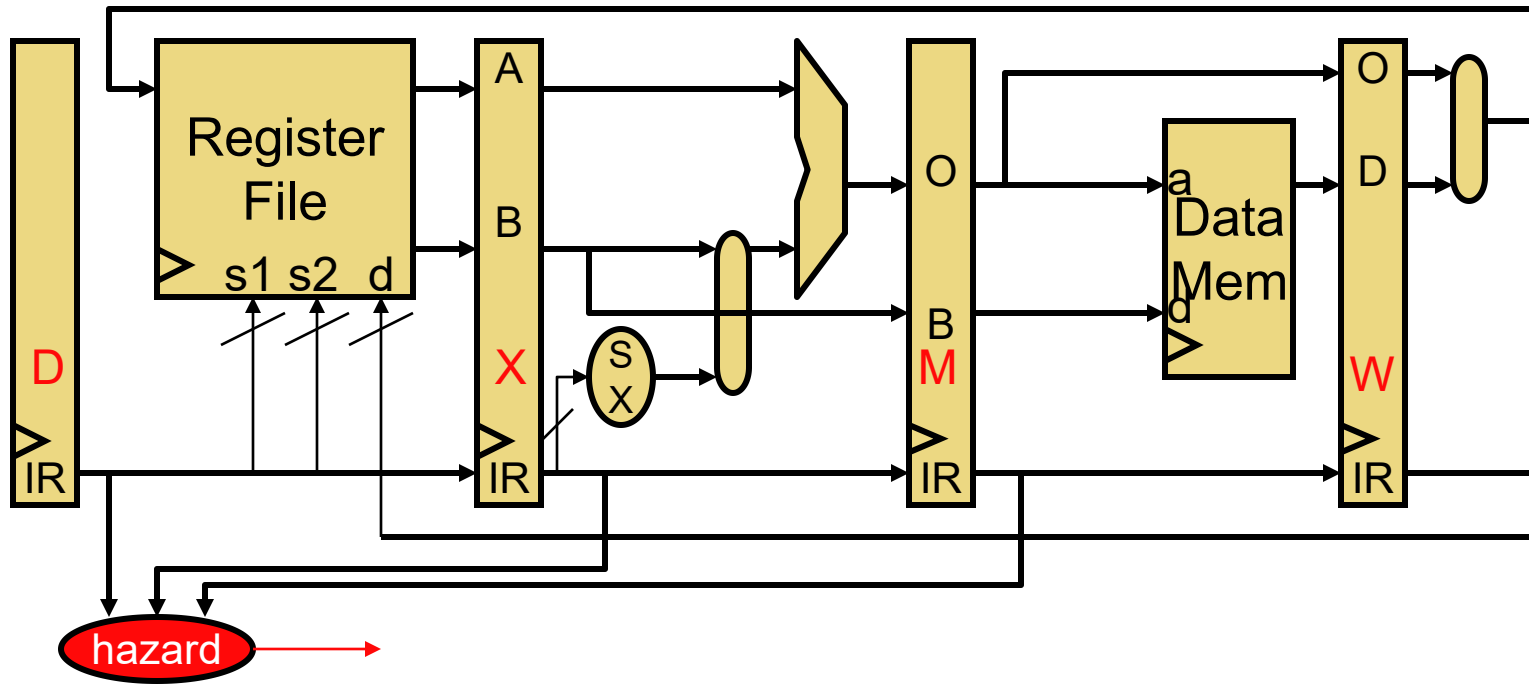
Approaches to Dependence Detection (I)

- Scoreboarding
 - Each register in register file has a Valid bit associated with it
 - An instruction that is writing to the register resets the Valid bit
 - An instruction in Decode stage checks if all its source and destination registers are Valid
 - Yes: No need to stall... No dependence
 - No: Stall the instruction
- Advantage:
 - Simple. 1 bit per register
- Disadvantage:
 - Need to stall for all types of dependences, not only flow dep.

Approaches to Dependence Detection (II)

- **Combinational dependence check logic**
 - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
 - Yes: stall the instruction/pipeline
 - No: no need to stall... no flow dependence
- **Advantage:**
 - No need to stall on anti and output dependences
- **Disadvantage:**
 - Logic is more complex than a scoreboard
 - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

Detecting Data Hazards

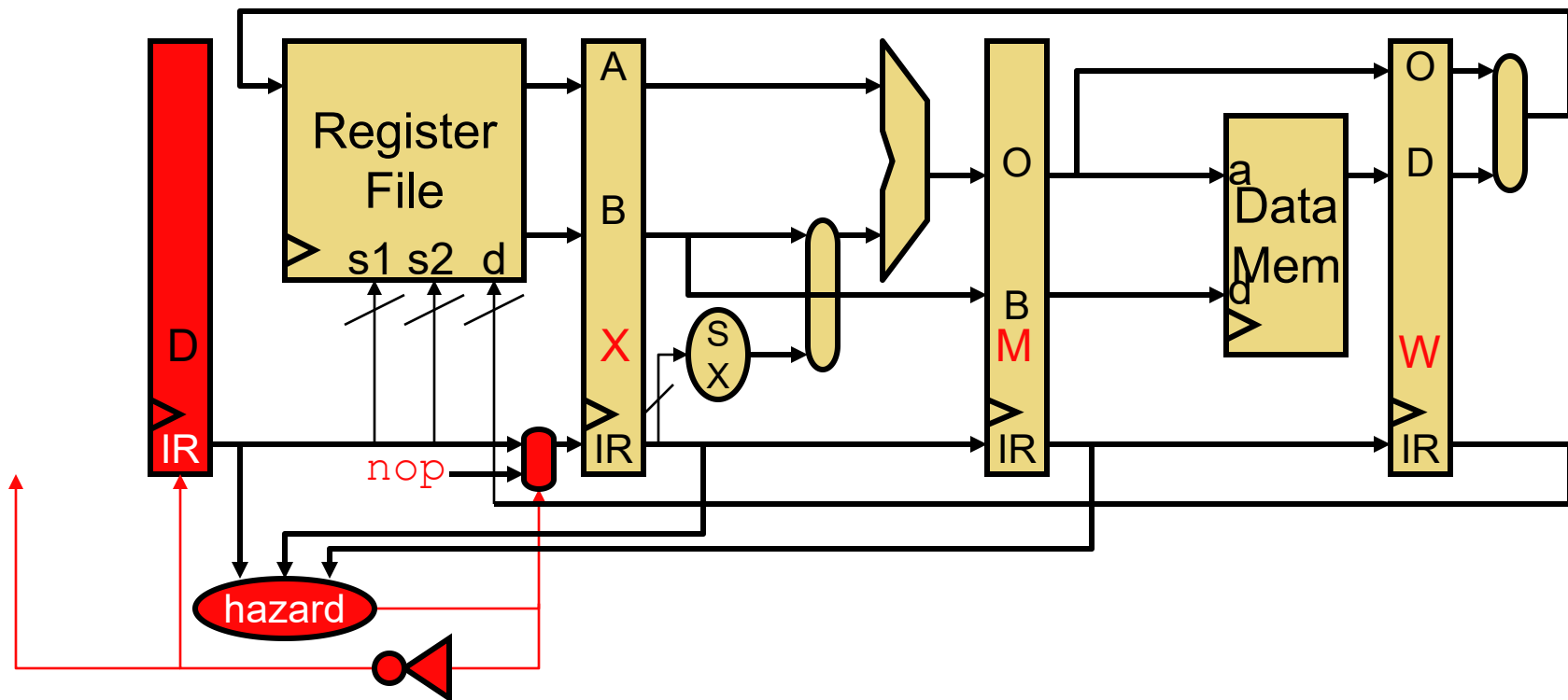


- Compare input register names of insn in D stage with output register names of older insns in pipeline

Stall =

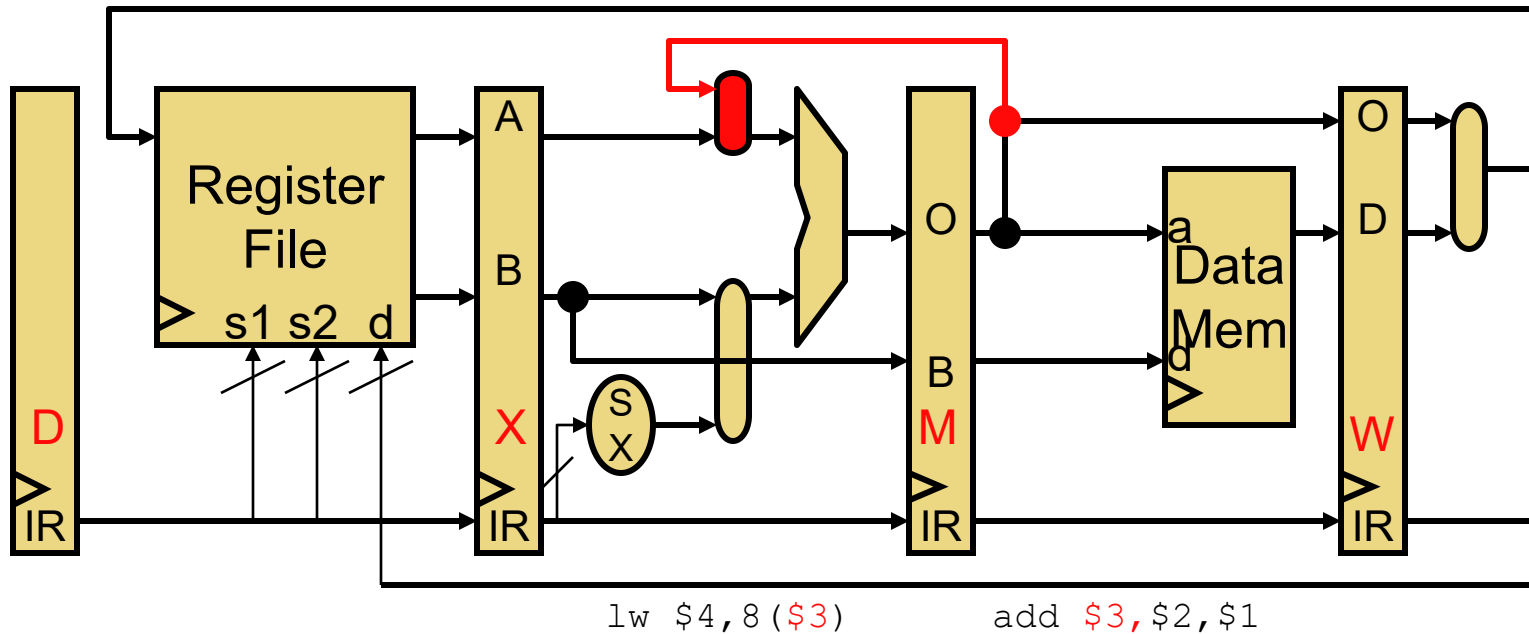
```
(D.IR.RegSrc1 == X.IR.RegDest) ||  
(D.IR.RegSrc2 == X.IR.RegDest) ||  
(D.IR.RegSrc1 == M.IR.RegDest) ||  
(D.IR.RegSrc2 == M.IR.RegDest)
```

Fixing Data Hazards



- Prevent D insn from reading (advancing) this cycle
 - Write **nop** into **X.IR** (effectively, insert **nop** in hardware)
 - Also reset (clear) the datapath control signals
 - Disable D register and PC write enables (why?)
- Re-evaluate situation next cycle

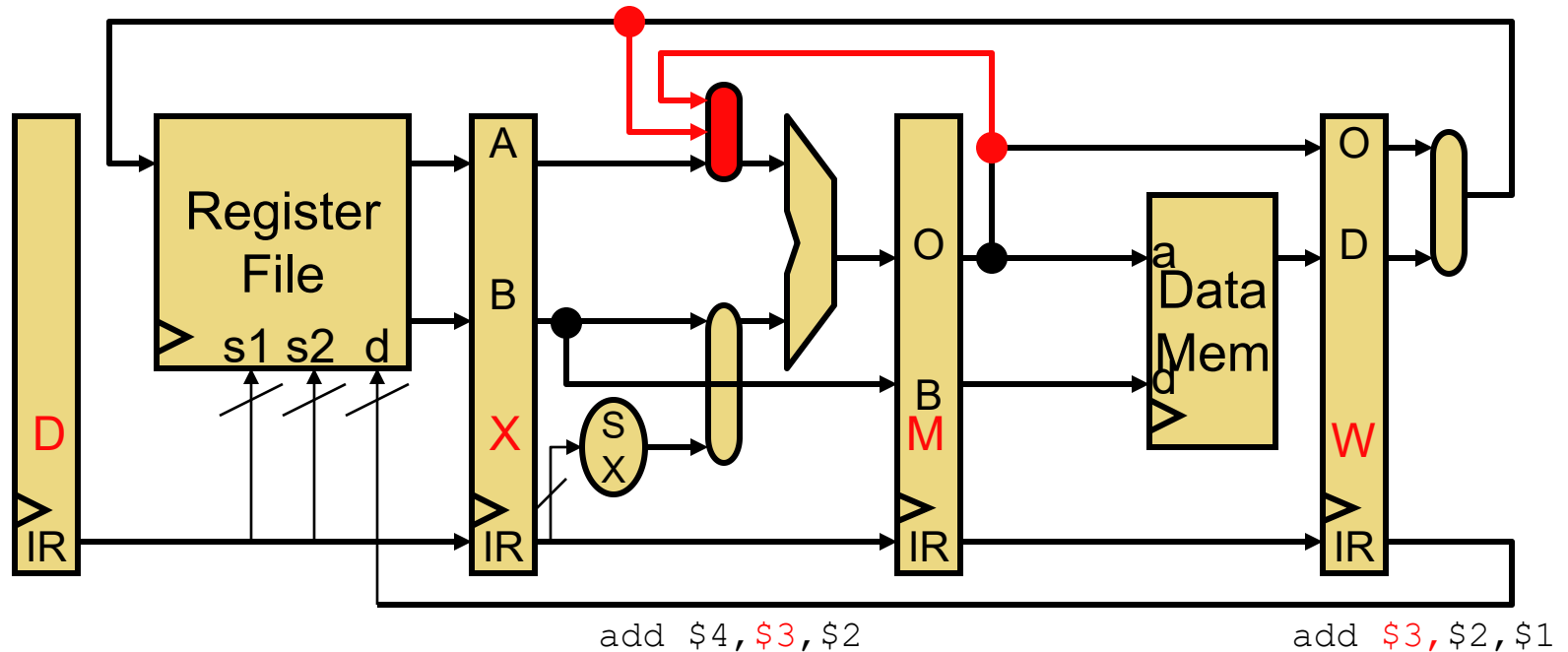
Bypassing



- **Bypassing**

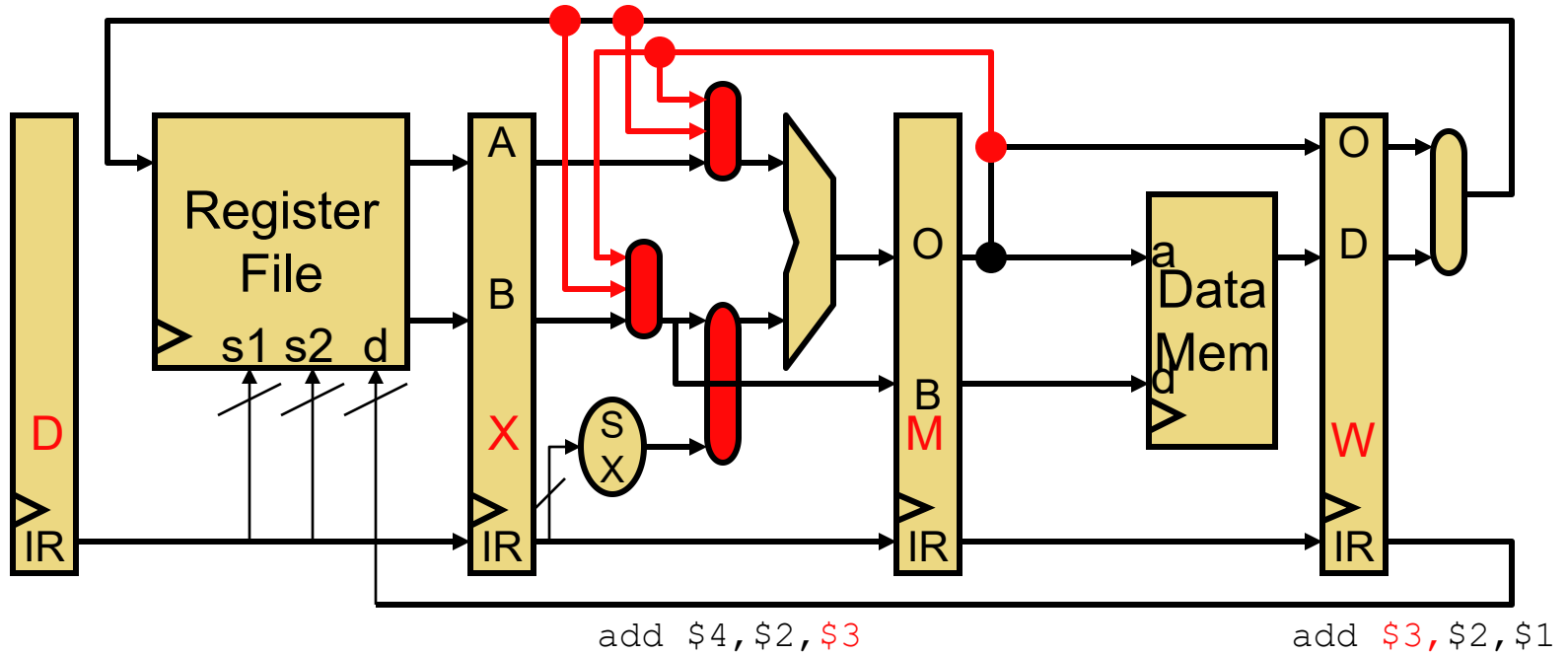
- Reading a value from an intermediate source
- Not waiting until it is available from primary source
- Here, we are bypassing the register file
- Also called **forwarding**

WX Bypassing



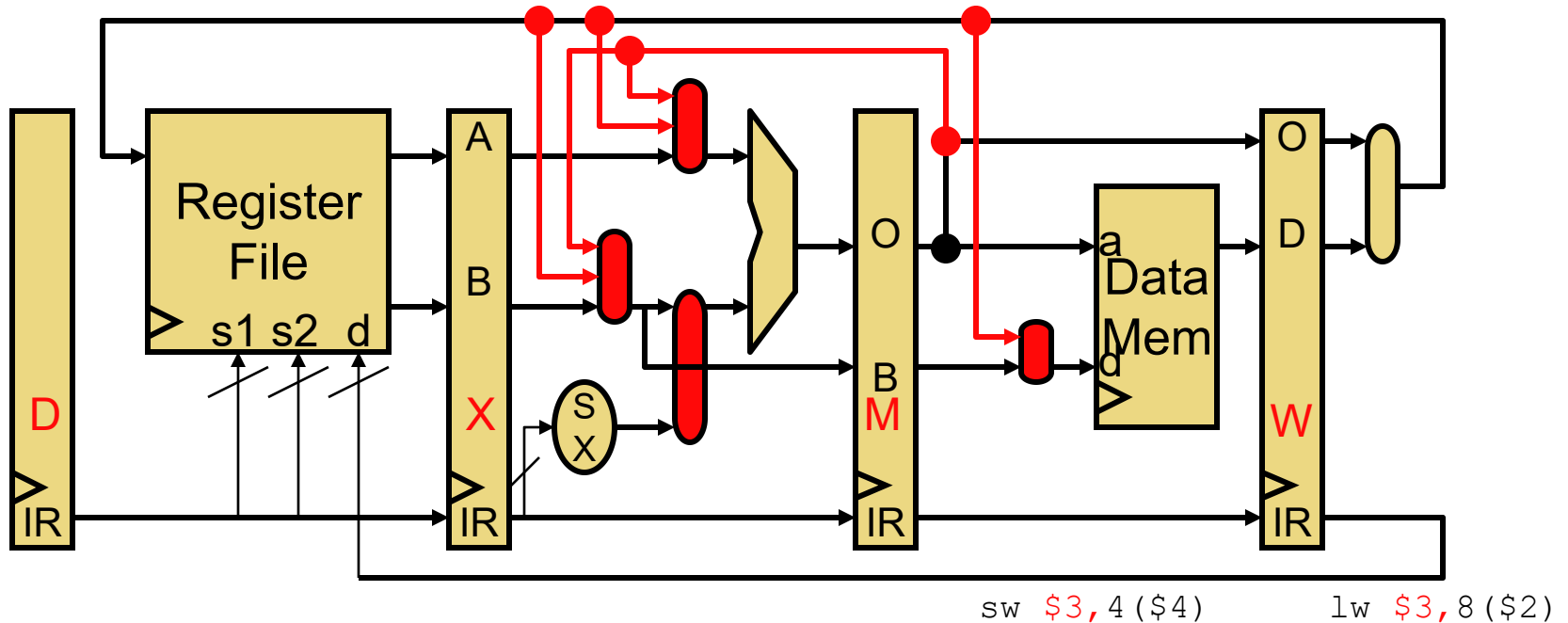
- What about this combination?
 - Add another bypass path and MUX (multiplexor) input
 - First one was an **MX** bypass
 - This one is a **WX** bypass

ALUinB Bypassing



- Can also bypass to ALU input B

WM Bypassing?



- Does WM bypassing make sense?
 - Not to the address input (why not?)

sw \$4, 4(\$3) lw \$3, 8(\$2)



- But to the store data input, yes

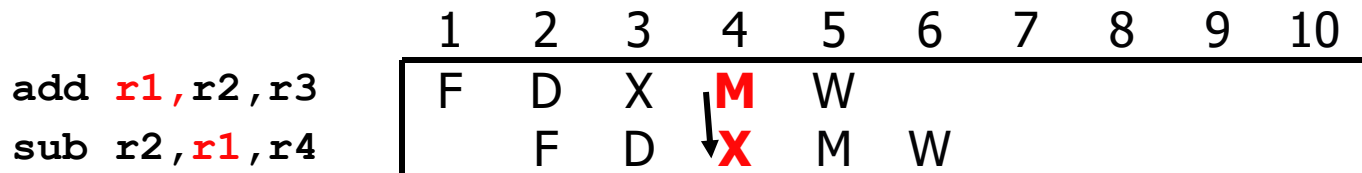
sw \$3, 4(\$4) lw \$3, 8(\$2)



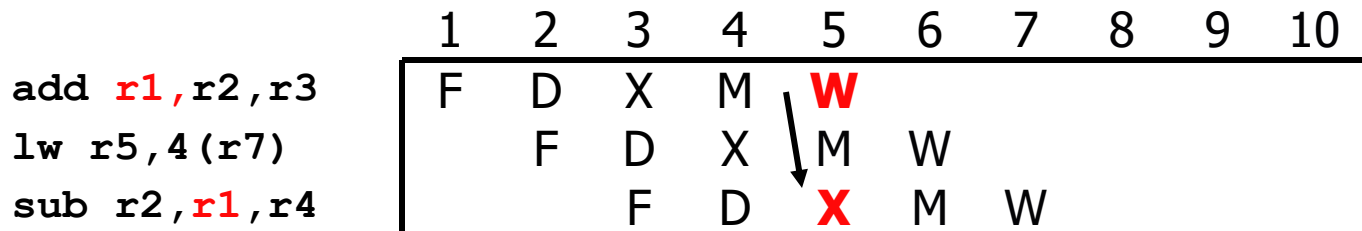
Pipeline Diagrams with Bypassing

- If bypass exists, "from"/"to" stages execute in same cycle

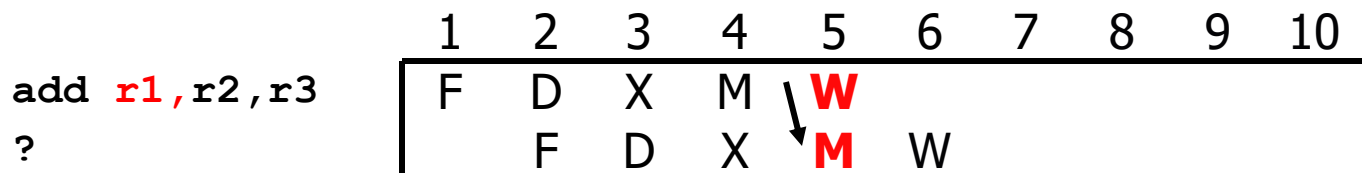
- Example: MX bypass



- Example: WX bypass

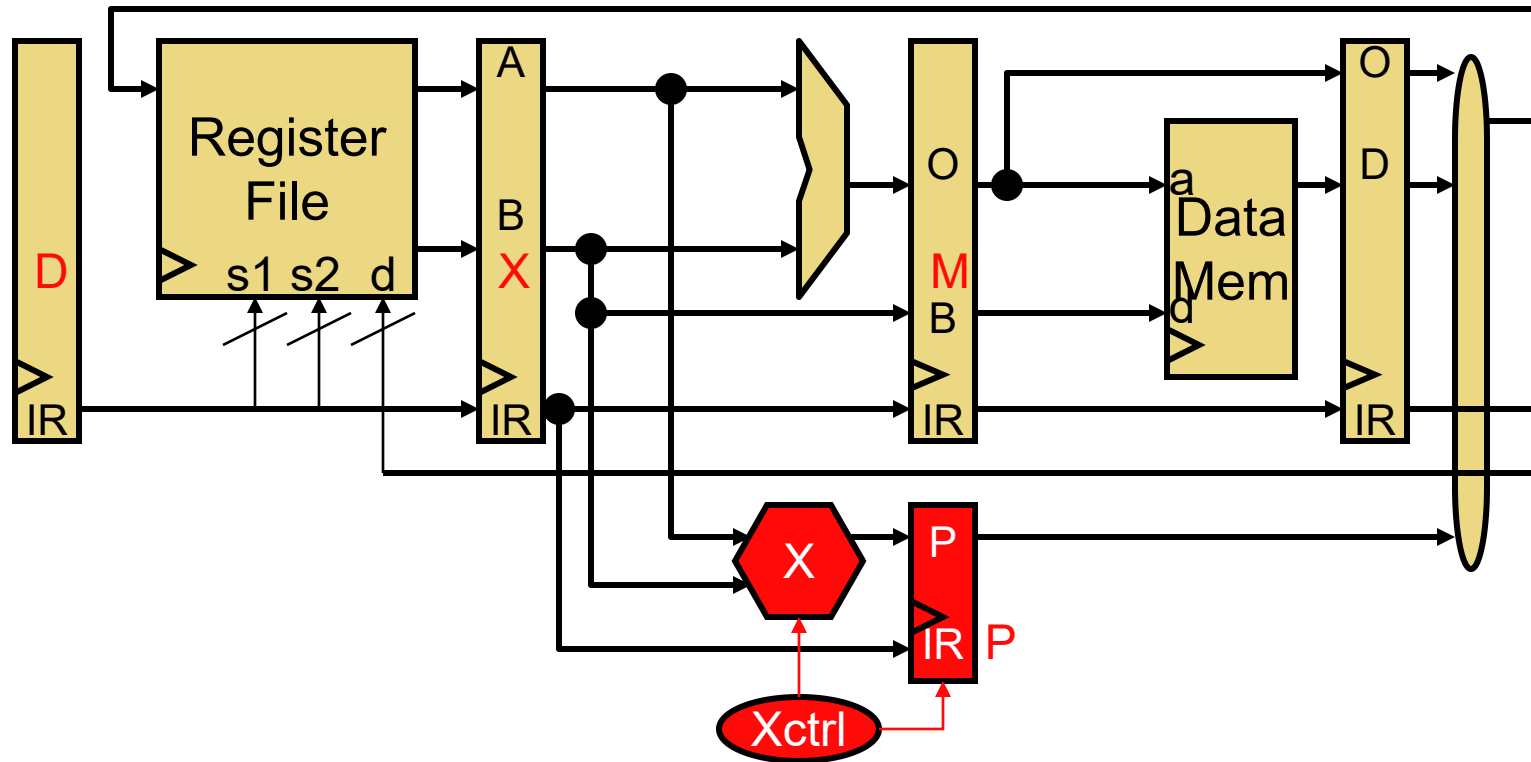


- Example: WM bypass



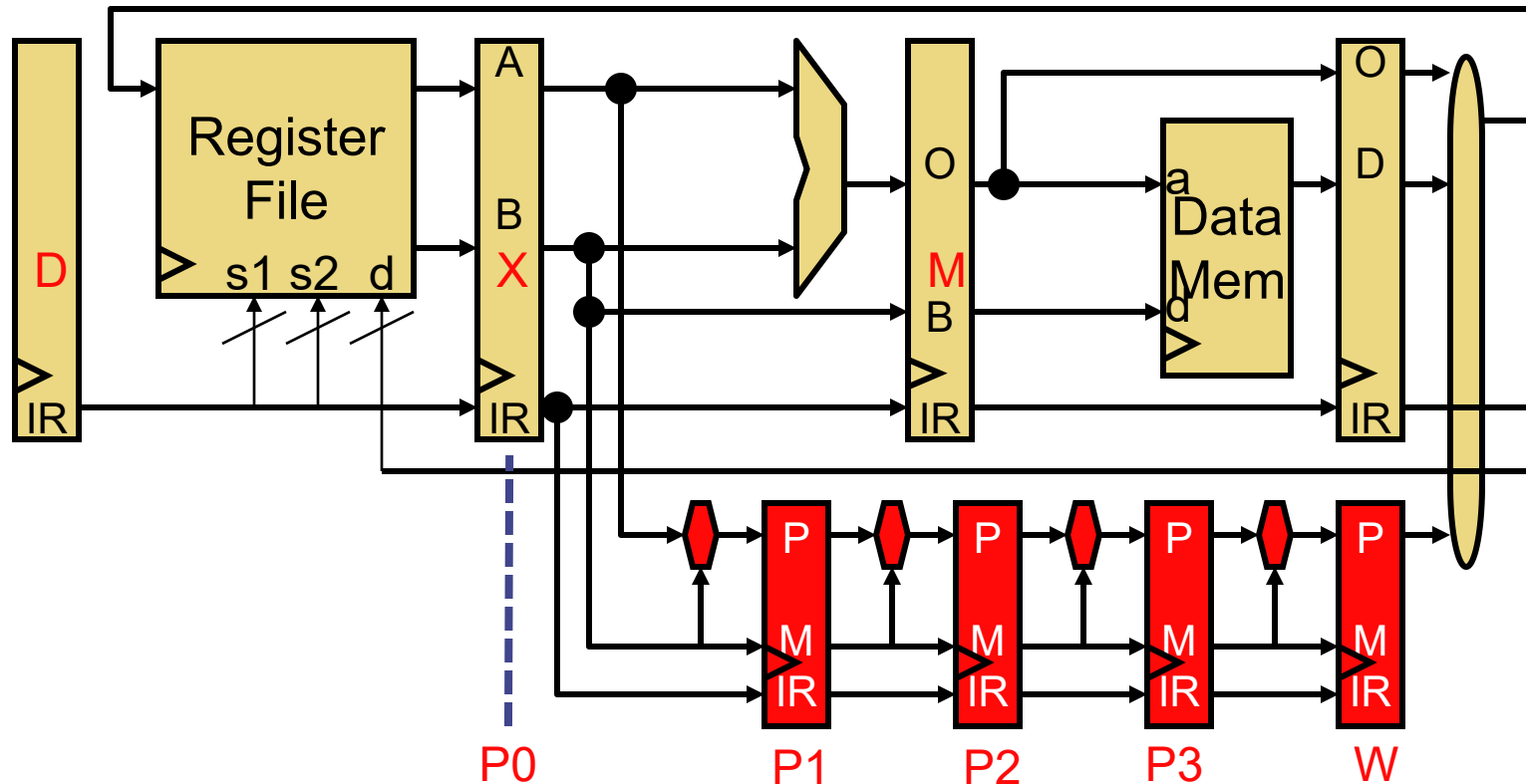
- Can you think of a code example that uses the WM bypass?

Pipelining and Multi-Cycle Operations



- What if you wanted to add a multi-cycle operation?
 - E.g., 4-cycle multiply
 - **P**: separate output register connects to W stage
 - Controlled by pipeline control finite state machine (FSM)

A Pipelined Multiplier



- Multiplier itself is often pipelined, what does this mean?
 - Product/multiplicand register/ALUs replicated
 - Can start different multiply operations in consecutive cycles
 - **But still takes 4 cycles to generate output value**

Pipeline Diagram with Multiplier


- Allow **independent** instructions

	1	2	3	4	5	6	7	8	9
mul \$4 ← \$3, \$5	F	D	P0	P1	P2	P3	W		
addi \$6 ← \$7, 1		F	D	X	M	W			

- Even allow **independent multiply** instructions

	1	2	3	4	5	6	7	8	9
mul \$4 ← \$3, \$5	F	D	P0	P1	P2	P3	W		
mul \$6 ← \$7, \$8		F	D	P0	P1	P2	P3	W	

- But must stall subsequent **dependent** instructions:

	1	2	3	4	5	6	7	8	9
mul \$4 ← \$3, \$5	F	D	P0	P1	P2	P3	W		
addi \$6 ← \$4 , 1		F	D	d*	d*	d*	 X	M	W

Multiplier Write Port Structural Hazard

- What about...
 - Two instructions trying to write register file in same cycle?
 - Structural hazard!
- Must prevent:

	1	2	3	4	5	6	7	8	9
<code>mul \$4,\$3,\$5</code>	F	D	P0	P1	P2	P3	W		
<code>addi \$6,\$1,1</code>		F	D	X	M	W			
<code>add \$5,\$6,\$10</code>			F	D	X	M	W		

- Solution? stall the subsequent instruction

	1	2	3	4	5	6	7	8	9
<code>mul \$4,\$3,\$5</code>	F	D	P0	P1	P2	P3	W		
<code>addi \$6,\$1,1</code>		F	D	X	M	W			
<code>add \$5,\$6,\$10</code>			F	D	d*	X	M	W	

Pipelining: Clock Frequency vs. IPC

- Increase number of pipeline stages (“pipeline depth”)
 - Keep cutting datapath into finer pieces
 - + Increases clock frequency (decreases clock period)
 - **Register overhead & unbalanced stages** cause sub-linear scaling
 - Double the number of stages won’t quite double the frequency
 - Increases CPI (decreases IPC)
 - More pipeline “hazards”, higher branch penalty
 - Memory latency relatively higher (same absolute lat., more cycles)
 - Result: after some point, deeper pipelining can decrease performance
 - “Optimal” pipeline depth is program and technology specific

Branch Prediction

More Sophisticated Direction Prediction

- Compile time (static)
 - Always not taken
 - Always taken
 - BTFN (Backward taken, forward not taken)
 - Profile based (likely direction)
 - Program analysis based (likely direction)
- Run time (dynamic)
 - Last time prediction (single-bit)
 - Two-bit counter based prediction
 - Two-level prediction (global vs. local)
 - Hybrid
 - Advanced algorithms (e.g., using perceptrons)

Dynamic Branch Prediction

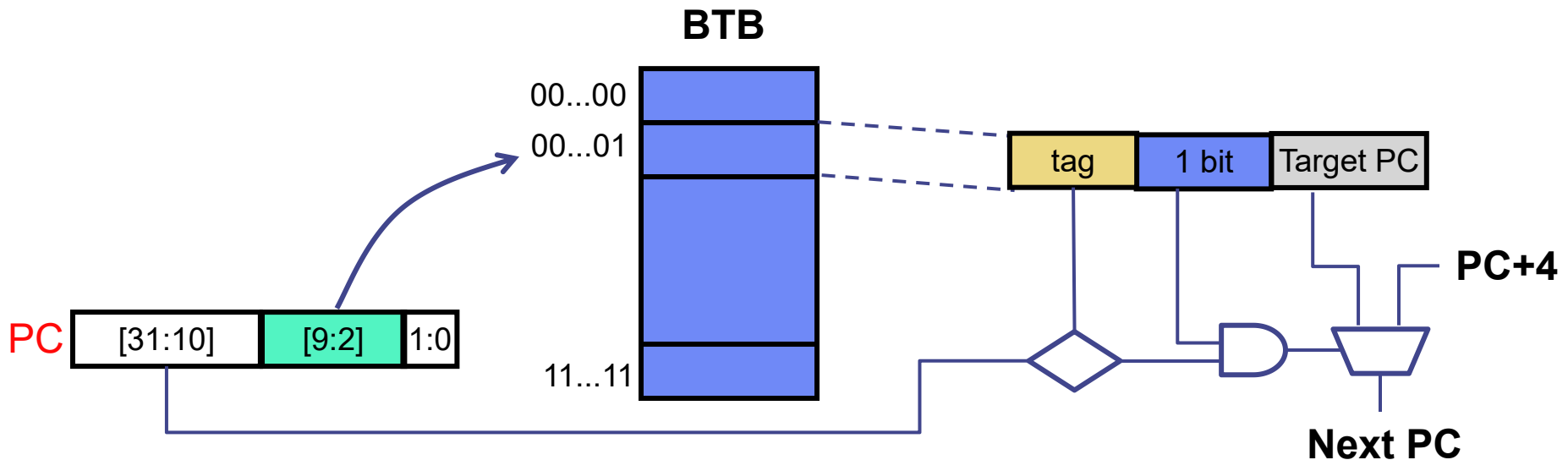
- Idea: Predict branches based on dynamic information (collected at run-time)
- Advantages
 - + Prediction based on history of the execution of branches
 - + It can adapt to dynamic changes in branch behavior
 - + No need for static profiling: input set representativeness problem goes away
- Disadvantages
 - More complex (requires additional hardware)

Last Time Predictor

- Last time predictor

- Single bit per branch (stored in BTB)
- 1 bit for prediction (0 = N, 1 = T)
- Indicates which direction branch went last time it executed

TTTTTTTTTTNNNNNNNNNN → 90% accuracy



Last Time Predictor

- Problem: **inner loop branch** below

```
for (i=0;i<100;i++)
  for (j=0;j<3;j++)
    // whatever
```

 - Two “built-in” mis-predictions per inner loop iteration
 - Branch predictor “changes its mind too quickly”

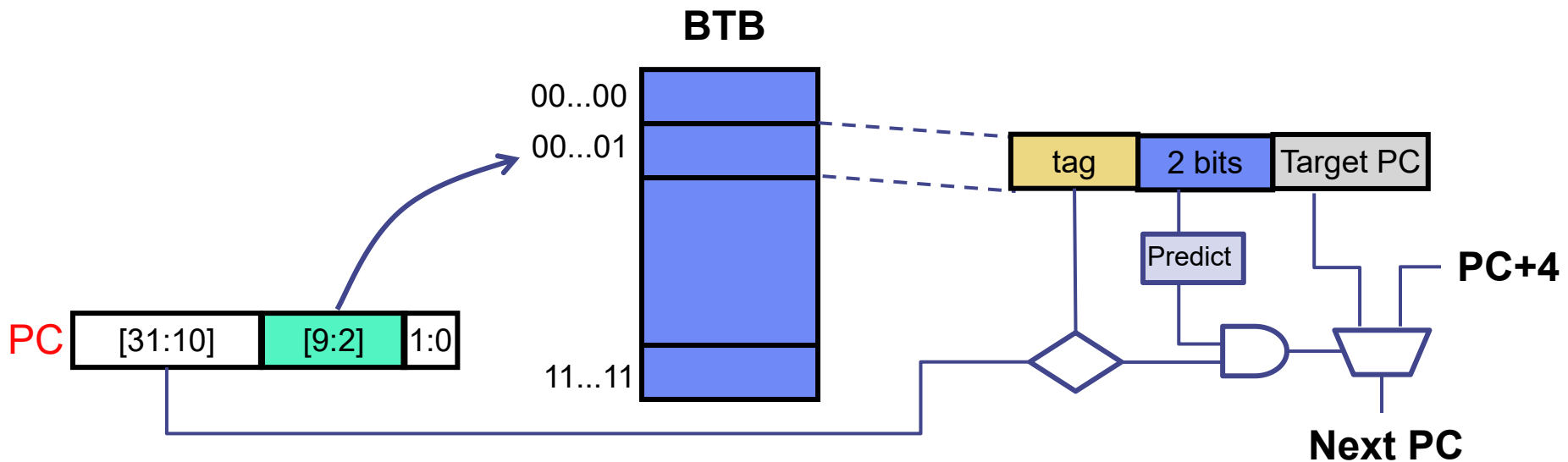
Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	T	T	T	Correct
3	T	T	T	Correct
4	T	T	N	Wrong
5	N	N	T	Wrong
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	N	N	T	Wrong
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Improving the Last Time Predictor

- Problem: A last-time predictor changes its prediction from $T \rightarrow NT$ or $NT \rightarrow T$ too quickly
 - even though the branch may be mostly taken or mostly not taken
- Solution Idea: Add hysteresis to the predictor so that prediction does not change on a single different outcome
 - Use two bits to track the history of predictions for a branch instead of a single bit
 - Can have 2 states for T or NT instead of 1 state for each
- Smith, "A Study of Branch Prediction Strategies," ISCA 1981.

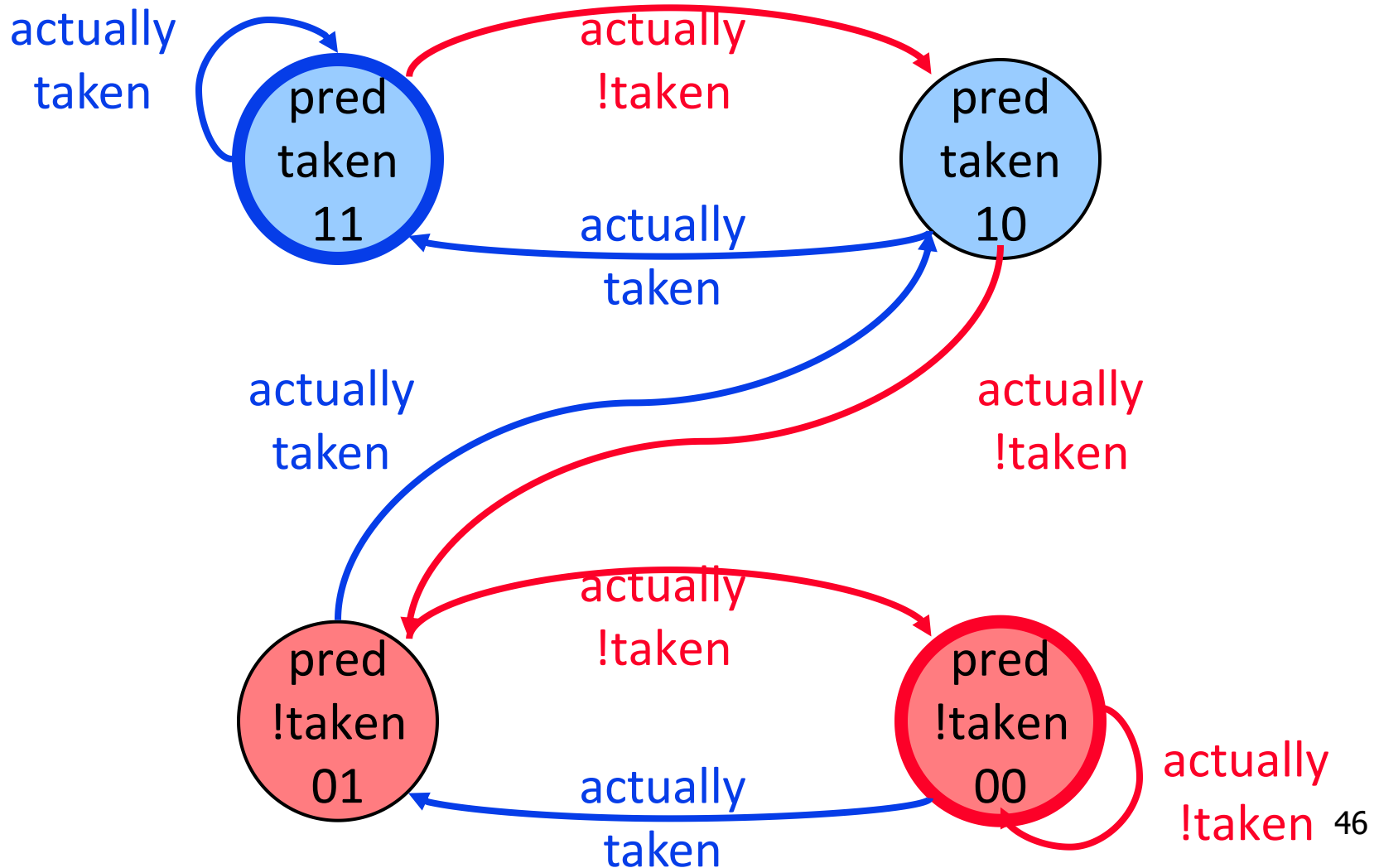
Two-Bit Counter Based Prediction

- Each branch associated with a two-bit counter
- One more bit provides hysteresis
- 2 bits: (00->N, 01->n, 10->t, 11->T)



State Machine for 2-bit Saturating Counter

- Counter using *saturating arithmetic*
 - Arithmetic with maximum and minimum values



Two-Bit Counter Based Prediction

- A strong prediction does not change with one single different outcome
- Accuracy for a loop with N iterations = $(N-1)/N$

TNTNTNTNTNTNTNTNTNTN →
50% accuracy

(assuming counter initialized to weakly taken)

- + Better prediction accuracy
- More hardware cost (but counter can be part of a BTB entry)

Time	State	Prediction	Outcome	Result?
1	N	N	T	Wrong
2	n	N	T	Wrong
3	t	T	T	Correct
4	T	T	N	Wrong
5	t	T	T	Correct
6	T	T	T	Correct
7	T	T	T	Correct
8	T	T	N	Wrong
9	t	T	T	Correct
10	T	T	T	Correct
11	T	T	T	Correct
12	T	T	N	Wrong

Can We Do Better: Two-Level Prediction

- Last-time and 2BC predictors exploit “last-time” predictability
- Realization 1: A branch’s outcome can be correlated with other branches’ outcomes
 - Global branch correlation
- Realization 2: A branch’s outcome can be correlated with past outcomes of the same branch (other than the outcome of the branch “last-time” it was executed)
 - Local branch correlation

Global Branch Correlation (I)

- Recently executed branch outcomes in the execution path are correlated with the outcome of the next branch

- If first branch not taken, second also not taken

```
if (cond1)
...
if (cond1 AND cond2)
```

- If first branch taken, second definitely not taken

```
branch Y: if (cond1) a = 2;
...
branch X: if (a == 0)
```

Global Branch Correlation (II)

- If Y and Z both taken, then X also taken
- If Y or Z not taken, then X also not taken

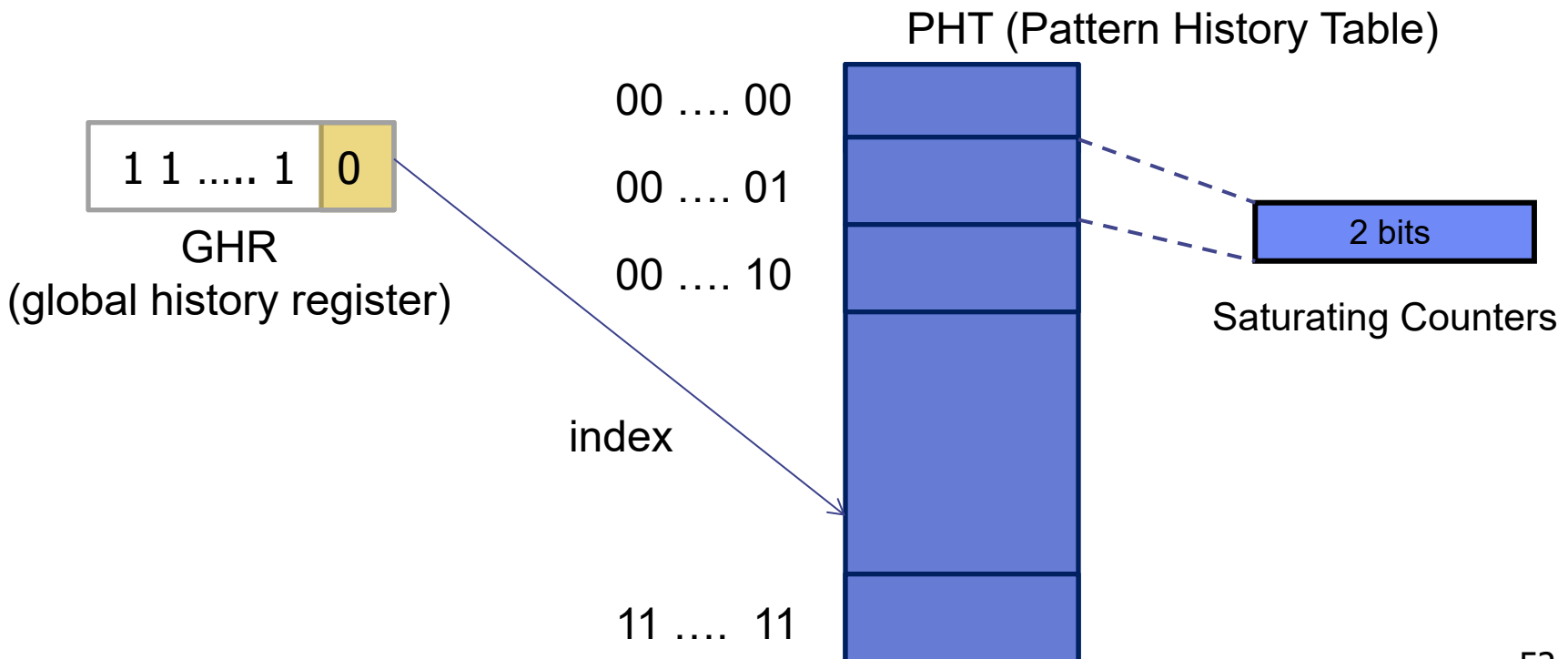
```
branch Y: if (cond1)
...
branch Z: if (cond2)
...
branch X: if (cond1 AND
cond2)
```

Capturing Global Branch Correlation

- Idea: Associate branch outcomes with “global T/NT history” of all branches
- Make a prediction based on the outcome of the branch the last time the same global branch history was encountered
- Implementation:
 - Keep track of the “global T/NT history” of all branches in a register → Global History Register (GHR)
 - Use GHR to index into a table that recorded the outcome that was seen for each GHR value in the recent past → Pattern History Table (table of 2-bit counters)

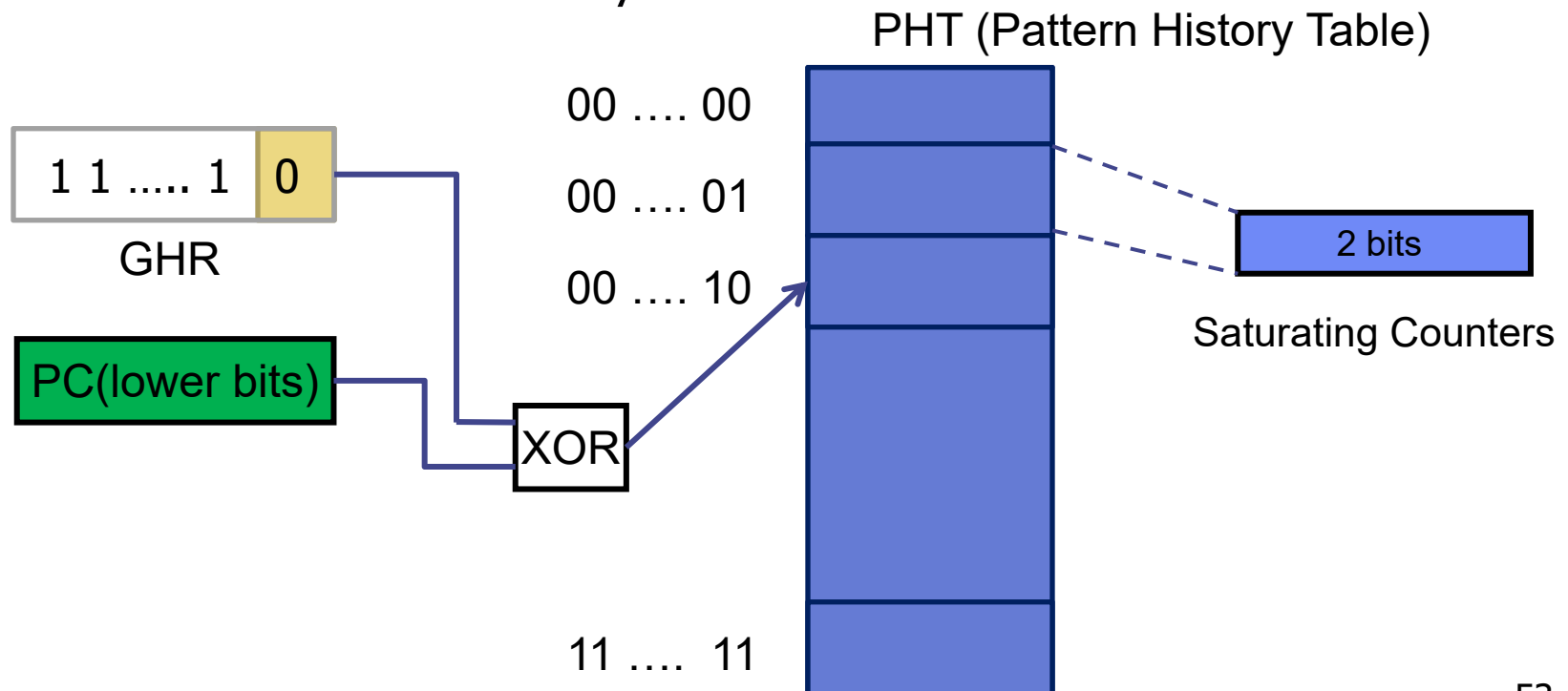
Two Level Global Branch Prediction

- First level: **Global branch history register** (N bits)
 - The direction of last N branches (1 bit per branch)
 - 1->tacken, 0->not taken
- Second level: **Table of saturating counters for each history entry**
 - The direction the branch took the last time the same history was seen



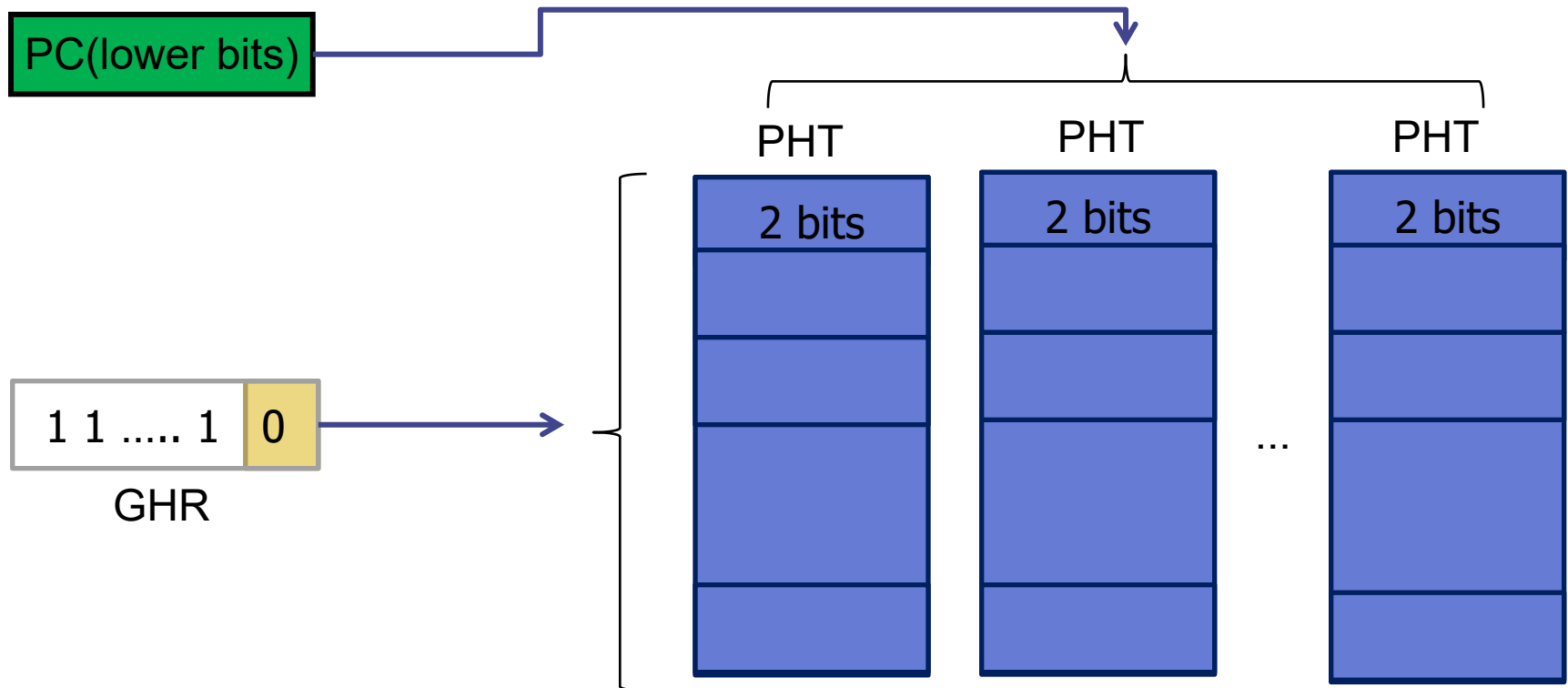
Improving Global Predictor Accuracy (I)

- Idea: Add more context information to the global predictor to take into account which branch is being predicted
 - **Gshare predictor**: GHR hashed with the Branch PC
 - + More context information used for prediction
 - + Better utilization of the two-bit counter array
 - Increases access latency



Improving Global Predictor Accuracy (II)

- Idea: Main separate PHT for different branches
 - use PC to determine the PHT
 - use GHR to determine the entry of the PHT
- + More context information used for prediction
- Increases hardware complexity



Cache

Memory Locality

- A “typical” program has a lot of locality in memory references
 - typical programs are composed of “loops”
- **Temporal**: A program tends to reference the same memory location many times and all within a small window of time
- **Spatial**: A program tends to reference a cluster of memory locations at a time
 - most notable examples:
 - 1. instruction memory references
 - 2. array/data structure references

Spatial and Temporal Locality Example

- Which memory accesses demonstrate spatial locality?
- Which memory accesses demonstrate temporal locality?

```
int sum = 0;  
int X[1000];  
  
for(int c = 0; c < 1000; c++){  
    sum += X[c];  
}
```

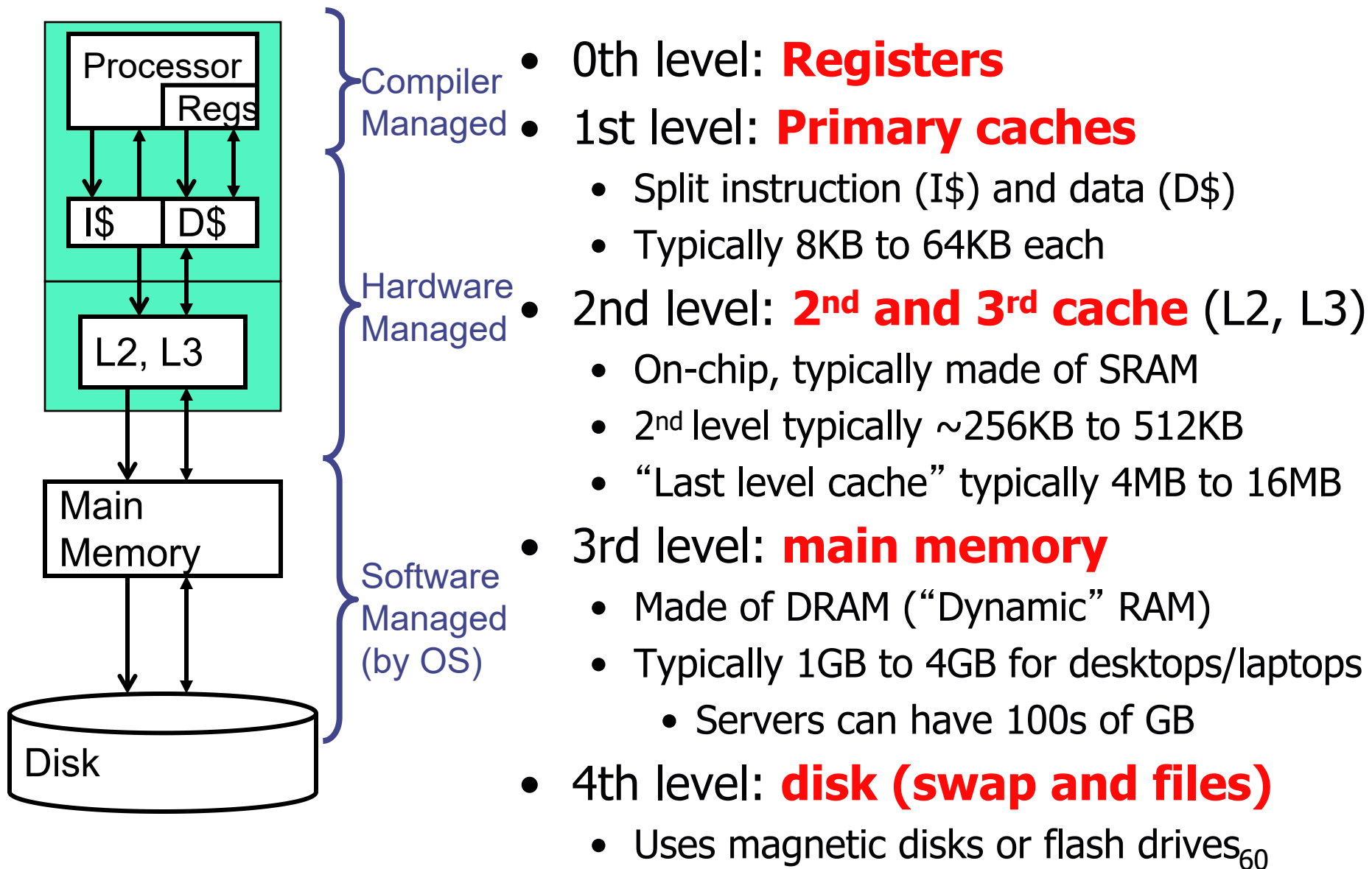
Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)
- Anticipation: the data will be accessed again soon
- Temporal locality principle
 - Recently accessed data will be again accessed in the near future

Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
 - Logically divide memory into equal size blocks
 - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon
- Spatial locality principle
 - Nearby data in memory will be accessed in the near future
 - E.g., sequential instruction access, array traversal

A Modern Memory Hierarchy



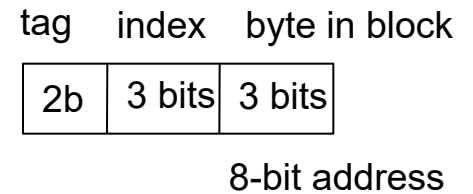
Caching Basics

- **Block (line):** Unit of storage in the cache
 - Memory is logically divided into cache blocks that map to locations in the cache
- On a reference:
 - **HIT:** If in cache, use cached data instead of accessing memory
 - **MISS:** If not in cache, bring block into cache
 - Maybe have to kick something else out to do it
- Some important cache design decisions
 - **Placement:** where and how to place/find a block in cache?
 - **Replacement:** what data to remove to make room in cache?
 - **Granularity of management:** large or small blocks?
 - **Write policy:** what do we do about writes?
 - **Instructions/data:** do we treat them separately?

Blocks and Addressing the Cache

- Memory is logically divided into fixed-size blocks
- Each block maps to a location in the cache, determined by the **index bits** in the address

- used to index into the tag and data stores



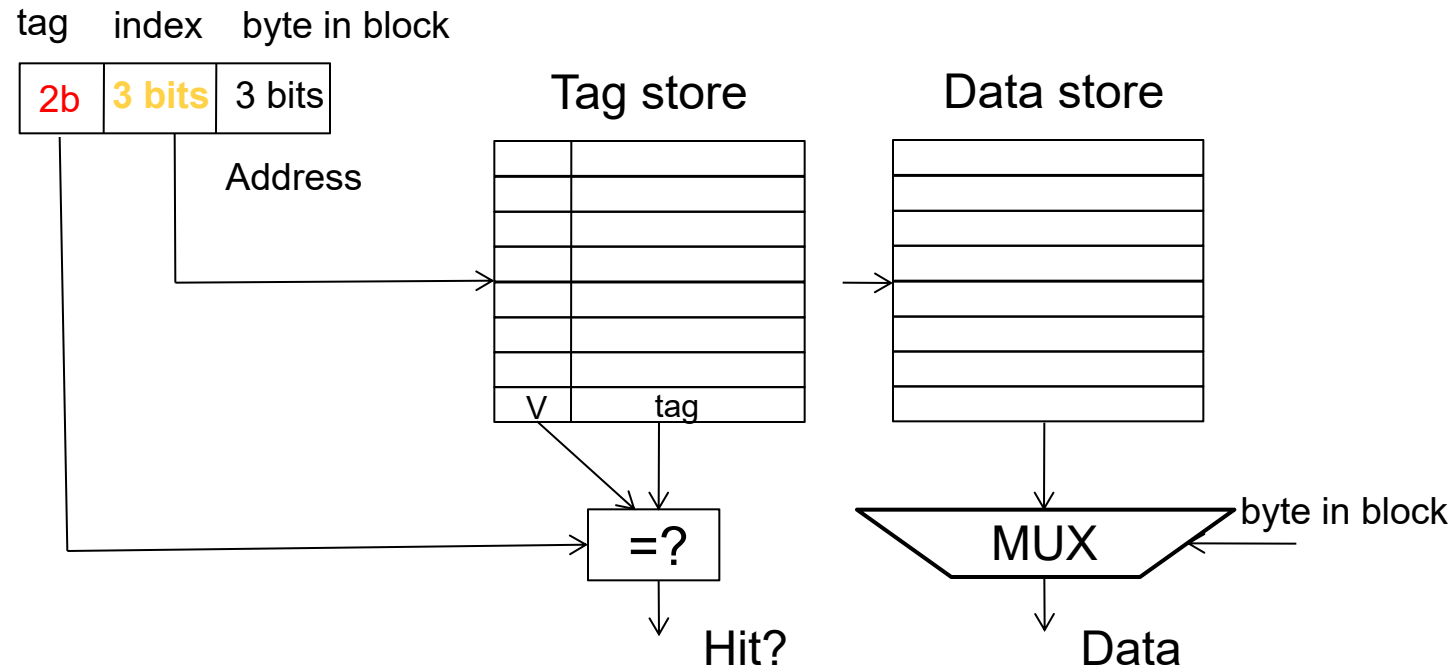
- Cache access:
 - 1) index into the tag and data stores with index bits in address
 - 2) check valid bit in tag store
 - 3) compare tag bits in address with the stored tag in tag store
- If a block is in the cache (cache hit), **the stored tag should be valid and match the tag of the block**

Direct-Mapped Cache: Placement and Access

Block: 00000
Block: 00001
Block: 00010
Block: 00011
Block: 00100
Block: 00101
Block: 00110
Block: 00111
Block: 01000
Block: 01001
Block: 01010
Block: 01011
Block: 01100
Block: 01101
Block: 01110
Block: 01111
Block: 10000
Block: 10001
Block: 10010
Block: 10011
Block: 10100
Block: 10101
Block: 10110
Block: 10111
Block: 11000
Block: 11001
Block: 11010
Block: 11011
Block: 11100
Block: 11101
Block: 11110
Block: 11111

Main memory

- Assume byte-addressable memory:
256 bytes, 8-byte blocks → 32 blocks
- Assume cache: 64 bytes, 8 blocks
 - **Direct-mapped: A block can go to only one location**



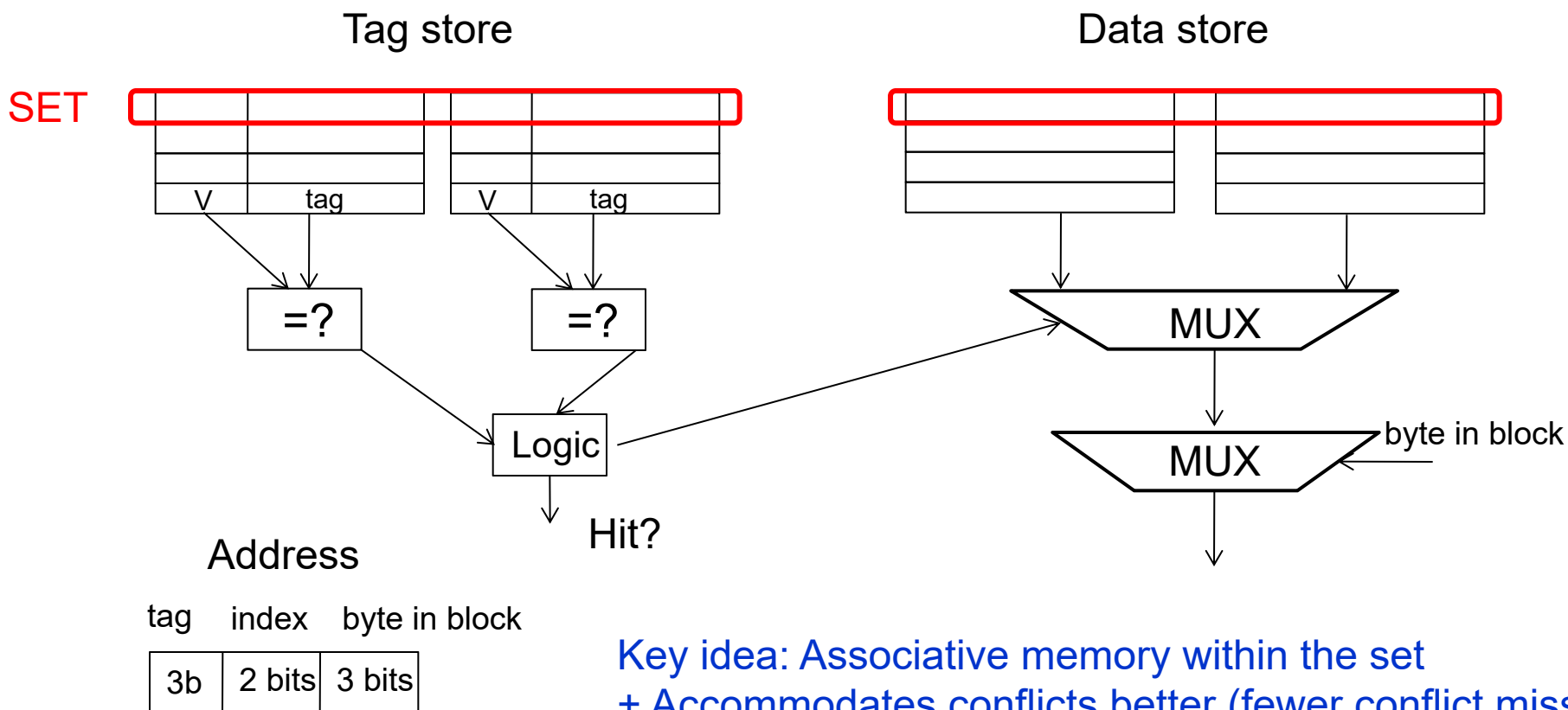
- **Addresses with same index contend for the same location**
 - **Cause conflict misses**

Direct-Mapped Caches

- **Direct-mapped cache:** Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
 - One index \rightarrow one entry
- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
 - Assume addresses A and B have the same index bits but different tag bits
 - A, B, A, B, A, B, A, B, ... \rightarrow conflict in the cache index
 - All accesses are **conflict misses**

Set Associativity

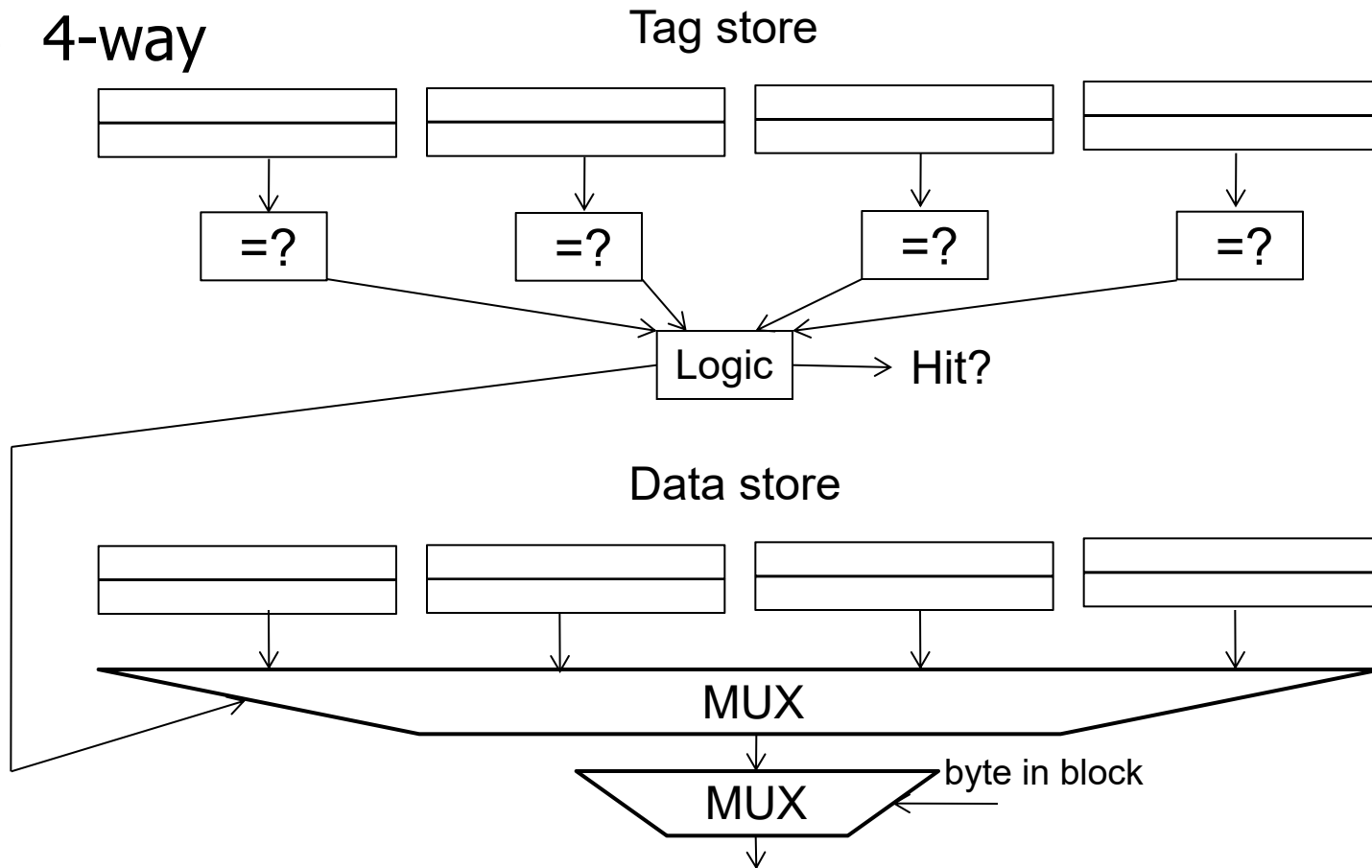
- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks



Key idea: Associative memory within the set
 + Accommodates conflicts better (fewer conflict misses)
 -- More complex, slower access, larger tag store

Higher Associativity

- 4-way

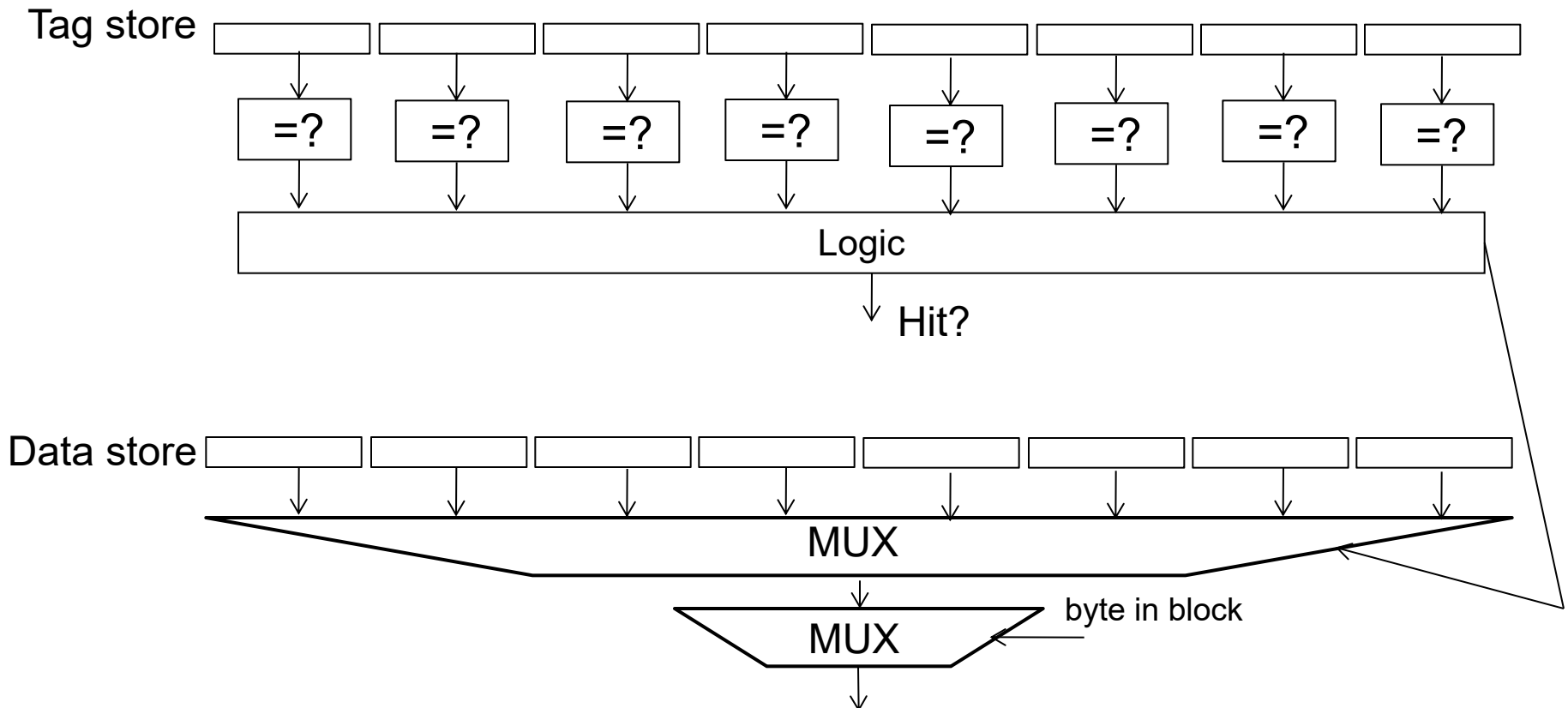


+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tags

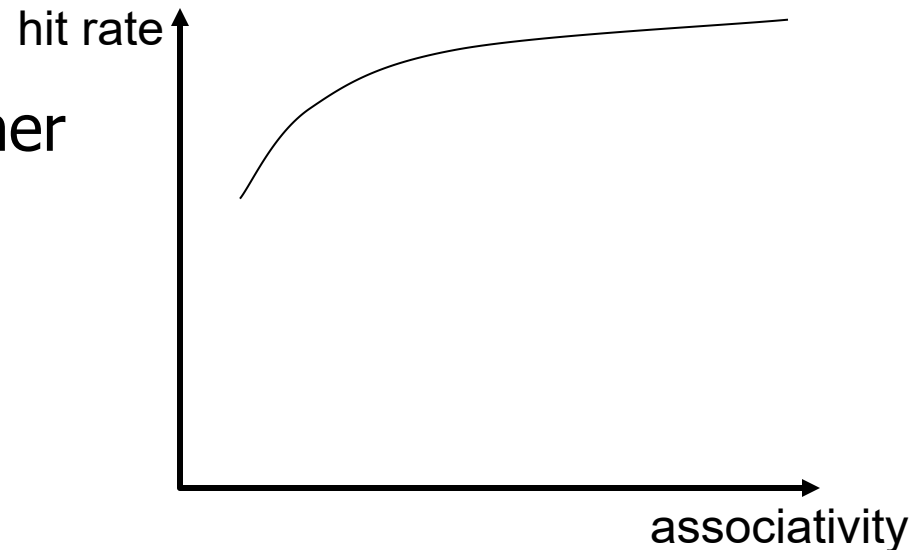
Full Associativity

- Fully associative cache
 - A block can be placed in **any** cache location



Associativity (and Tradeoffs)

- **Degree of associativity**: How many blocks can map to the same index (or set)?
- Higher associativity
 - ++ Higher hit rate
 - Slower cache access time (hit latency and data access latency)
 - More expensive hardware (more comparators)
- Diminishing returns from higher associativity



Eviction/Replacement Policy

- Which block in the set to replace on a cache miss?
 - Any invalid block first
 - If all are valid, consult the replacement policy
 - **Random**
 - **FIFO (first-in first-out)**
 - **LRU (least recently used)**
 - Fits with temporal locality, LRU = least likely to be used in future
 - **NMRU (not most recently used)**
 - An easier to implement approximation of LRU
 - Is LRU for 2-way set-associative caches
 - **Belady's**: replace block that will be used furthest in future
 - Unachievable optimum

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
 - What do you need to implement LRU perfectly?
- Question: 4-way set associative cache:
 - What do you need to implement LRU perfectly?
 - How many different orderings possible for the 4 blocks in the set?
 - How many bits needed to encode the LRU order of a block?
 - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement “true LRU” (also called “perfect LRU”) in highly-associative caches
- Why?
 - True LRU is complex
 - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)
- Examples:
 - **Not MRU** (not most recently used)
 - **Hierarchical LRU**: divide the N-way set into M “groups”, track the MRU group and the MRU way in each group

What Is the Optimal Replacement Policy?

- Belady's OPT
 - Replace the block that is going to be referenced furthest in the future by the program
 - Belady, “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, 1966.
 - How do we implement this? Simulate?
- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
 - No. Cache miss latency/cost varies from block to block!
 - Two reasons: Remote vs. local caches and miss overlapping
 - Qureshi et al. “A Case for MLP-Aware Cache Replacement,” ISCA 2006.

Write Issues

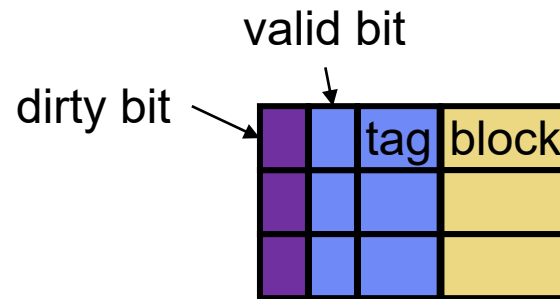
- So far we have looked at reading from cache
 - Instruction fetches, loads
- What about writing into cache?
- Several new issues
 - Write-through vs. write-back
 - Write-allocate vs. write-not-allocate

Write Propagation

- When to propagate new value to lower-level caches/memory?
- **Option #1: Write-through**: immediately
 - On hit, update cache
 - Immediately send the write to the next level
- **Option #2: Write-back**: when block is replaced
 - Now we have multiple versions of the same block in various caches and in memory!
 - Requires additional “**dirty**” bit per block
 - Evict **clean** block: **no extra traffic**
 - there was only 1 version of the block
 - Evict **dirty** block: **extra “writeback” of block**
 - the dirty block is the most up-to-date version

Write-back Cache Operation

- Each cache block has a **dirty bit** associated with it
 - state is either **clean** or **dirty**



initial state

-	1	-	-
---	---	---	---

after `ld r0 <= [A]`

C	V	A	0x01
---	---	---	------

after `st r1 => [A]`

D	V	A	0x02
---	---	---	------

Write Propagation Comparison

- **Write-through**

- Requires additional bus bandwidth
 - Consider repeated write hits
- Next level must handle small writes (1, 2, 4, 8-bytes)
- + No need for dirty bits in cache
- + No need to handle “writeback” operations
 - Simplifies miss handling
 - Used in GPUs, as they have low write temporal locality

- **Write-back**

- + Key advantage: uses less bandwidth
- Reverse of other pros/cons above
- Used in most CPU designs

Write Miss Handling

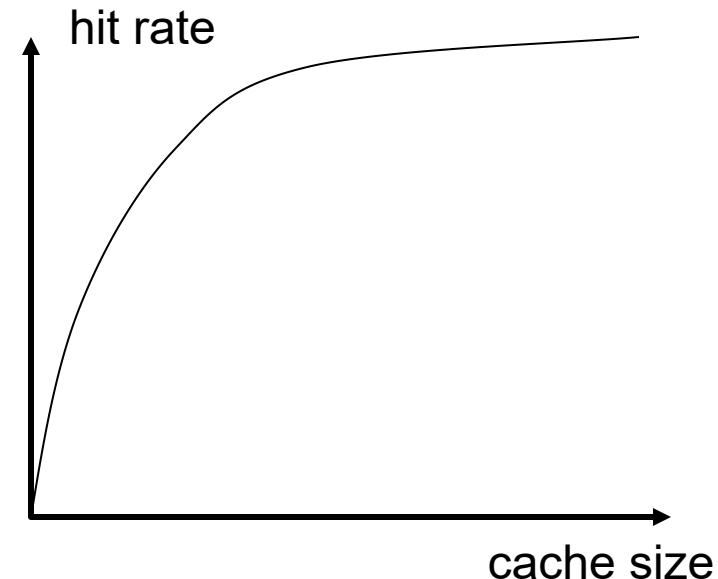
- How is a write miss actually handled?
- **Write-allocate**: fill block from next level, then write it
 - + Decreases read misses (next read to block will hit)
 - Requires additional bandwidth
 - Commonly used (especially with [write-back caches](#))
- **Write-non-allocate**: just write to next level, no allocate
 - Potentially more read misses
 - + Uses less bandwidth
 - Use with write-through

Cache Parameters vs. Miss/Hit Rate

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

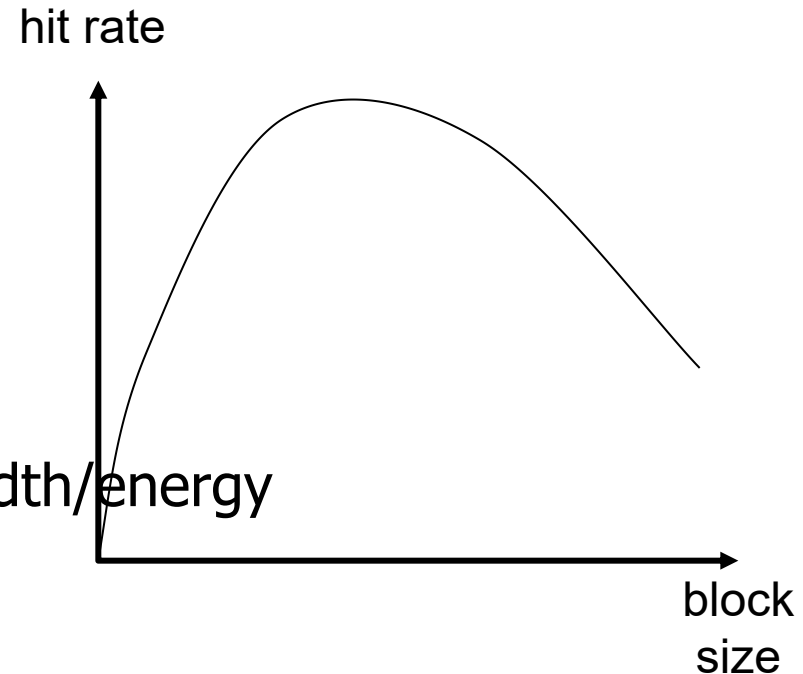
Cache Size

- Cache size: total data (not including tag) capacity
 - bigger can exploit temporal locality better
 - not ALWAYS better
- **Too large** a cache adversely affects hit and miss latency
 - smaller is faster => bigger is slower
 - access time may degrade critical path
- **Too small** a cache
 - doesn't exploit temporal locality well
 - useful data replaced often
- **Working set**: the whole set of data the executing application references
 - Within a time interval



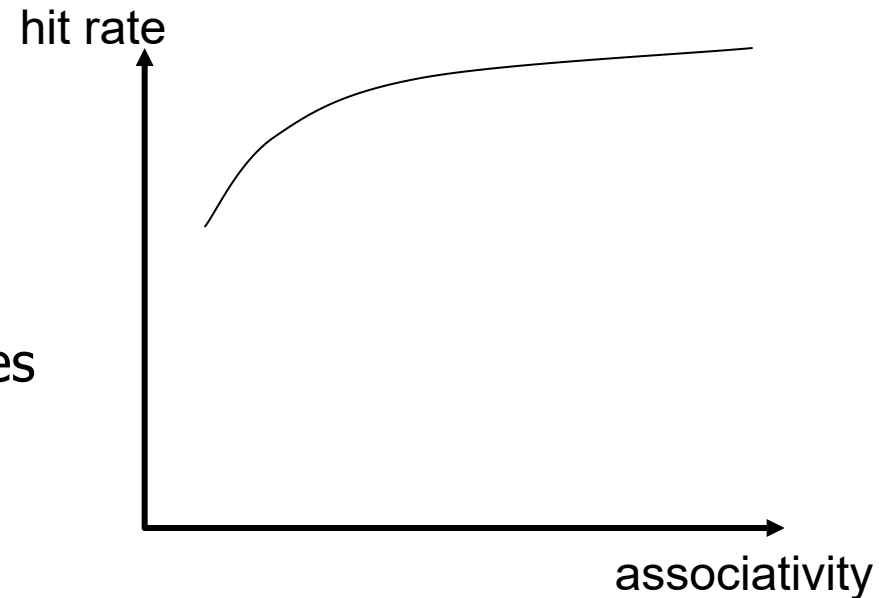
Block Size

- Block size is the data that is associated with an address tag
- **Too small** blocks
 - don't exploit spatial locality well
 - have larger tag overhead
- **Too large** blocks
 - too few total # of blocks → less temporal locality exploitation
 - waste of cache space and bandwidth/energy if spatial locality is not high



Associativity

- How many blocks can be present in the same index (i.e., set)?
- **Larger associativity**
 - lower miss rate (reduced conflicts)
 - higher hit latency and area cost (plus diminishing returns)
- **Smaller associativity**
 - lower cost
 - lower hit latency
 - Especially important for L1 caches



Classification of Cache Misses (3C)

- **Compulsory miss**
 - first reference to an address (block) always results in a miss
 - subsequent references should hit unless the cache block is displaced for the reasons below
- **Capacity miss**
 - cache is too small to hold everything needed (identify?)
 - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- **Conflict miss**
 - defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Miss Type

- **Compulsory**
 - Caching cannot help
 - Prefetching can: Anticipate which blocks will be needed soon
- **Conflict**
 - More associativity
 - Other ways to get more associativity without making the cache associative
 - Victim cache
 - Better, randomized indexing
- **Capacity**
 - Utilize cache space better: keep blocks that will be referenced
 - Software management: divide working set and computation such that each “computation phase” fits in cache

Prefetch

Challenges in Prefetching: How

- **Software** prefetching
 - ISA provides prefetch instructions
 - Programmer or compiler inserts prefetch instructions into code
 - Usually works well only for “regular access patterns”
- **Hardware** prefetching
 - Specialized hardware monitors memory accesses
 - Memorizes, finds, learns address strides/patterns/correlations
 - Generates prefetch addresses automatically
- **Execution-based** prefetching
 - A “thread” is executed to prefetch data for the main program
 - Can be generated by either software/programmer or hardware

Prefetcher Performance Metrics

- **Accuracy** (used prefetches / sent prefetches)
- **Coverage** (prefetched hits / all memory accesses)
- **Timeliness** (on-time prefetches / used prefetches)

- Bandwidth consumption
 - Memory bandwidth consumed with prefetcher / without prefetcher
 - Good news: **Can utilize idle bus bandwidth (if available)**

- Cache pollution
 - Extra demand misses due to prefetch placement in cache
 - More difficult to exactly quantify, but affects performance

Software Prefetching (I)

```
for (i=0; i<N; i++) {      while (p) {                  while (p) {
    __prefetch(a[i+8]);      __prefetch(p→next);      __prefetch(p→next→next→next);
    __prefetch(b[i+8]);      work(p→data);                  work(p→data);
    sum += a[i]*b[i];        p = p→next;                  p = p→next;
}                             }                               }
```

Which one is better?

- Can work for very regular array-based access patterns. Issues:
 - Prefetch instructions take up processing/execution bandwidth
 - **How early to prefetch?** Determining this is difficult
 - Prefetch distance depends on hardware implementation (memory latency, cache size, time between loop iterations) → portability?
 - Going too far back in code reduces accuracy (branches in between)
 - Need “special” prefetch instructions in ISA?
 - Alpha load into register 31 treated as prefetch (r31==0)
 - PowerPC *dcbt* (data cache block touch) instruction
 - Not easy to do for pointer-based data structures

Software Prefetching (II)

- Where should a compiler insert prefetches?
 - Prefetch for every load access?
 - Too bandwidth intensive (both memory and execution bandwidth)
 - Profile the code and determine loads that are likely to miss
 - What if profile input set is not representative?
 - How far ahead before the miss should the prefetch be inserted?
 - Profile and determine probability of use for various prefetch distances from the miss
 - What if profile input set is not representative?
 - Usually need to insert a prefetch far in advance to cover 100s of cycles of main memory latency → reduced accuracy

Hardware Prefetching

- Idea: Specialized hardware observes load/store access patterns and prefetches data based on past access behavior
- Tradeoffs:
 - + Can be tuned to system implementation
 - + Does not waste instruction execution bandwidth
 - More hardware complexity to detect patterns
 - Software can be more efficient in some cases

Next-Line Prefetchers

- Simplest form of hardware prefetching: always prefetch next N cache lines after a demand access (or a demand miss)
 - Next-line prefetcher (or next sequential prefetcher)
- Tradeoffs:
 - + Simple to implement. No need for sophisticated pattern detection
 - + Works well for sequential/streaming access patterns (instructions?)
 - Can waste bandwidth with irregular patterns
 - And, even regular patterns:
 - What is the prefetch accuracy if access stride = 2 and $N = 1$?
 - What if the program is traversing memory from higher to lower addresses?
 - Also prefetch “previous” N cache lines?

Stride Prefetchers

- Consider the following strided memory access pattern:
 - $A, A+N, A+2N, A+3N, A+4N\dots$
 - Stride = N
- Idea: Record the stride between consecutive memory accesses; if stable, use it to predict next M memory accesses
- Two types
 - Stride determined on a per-instruction basis
 - Stride determined on a per-memory-region basis

Runahead Execution

- A technique to obtain the memory-level parallelism benefits of a large instruction window
- When the oldest instruction is a long-latency cache miss:
 - **Checkpoint** architectural state and enter runahead mode
- In runahead mode:
 - **Speculatively pre-execute instructions**
 - **The purpose of pre-execution is to generate prefetches**
 - L2-miss dependent instructions are marked INV and dropped
- When the original miss returns:
 - **Restore checkpoint**, flush pipeline, resume normal execution
- Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

Runahead Execution Pros and Cons

- Advantages:
 - + Very accurate prefetches for data/instructions (all cache levels)
 - + Follows the program path
 - + Simple to implement: most of the hardware is already built in
 - + No waste of hardware context: uses the main thread context for prefetching
 - + No need to construct a special-purpose pre-execution thread for prefetching
- Disadvantages/Limitations
 - Extra executed instructions
 - Limited by branch prediction accuracy
 - Cannot prefetch dependent cache misses
 - Prefetch distance (how far ahead to prefetch) limited by memory latency
- Implemented in Sun ROCK, IBM POWER6, NVIDIA Denver

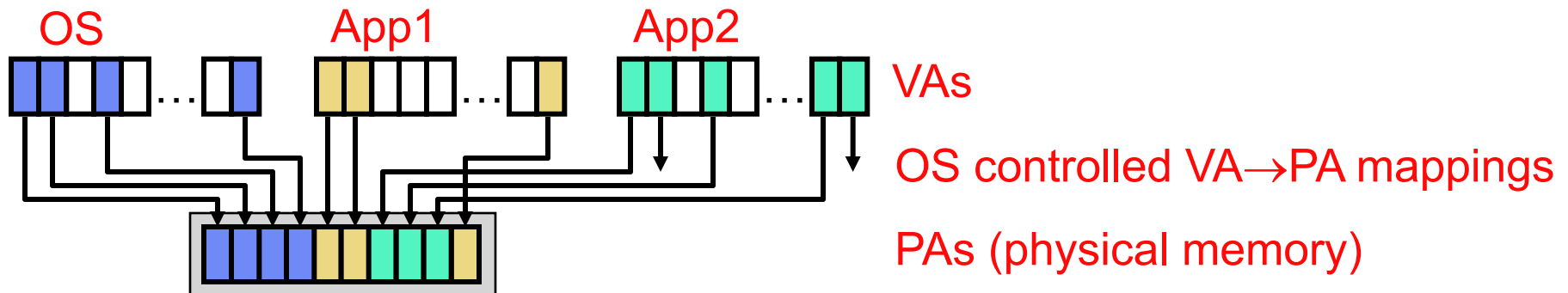
Virtualizing Memory

Virtualizing Main Memory

- How do multiple procs (and the OS) share main memory?
 - **Goal: each application thinks it has all of the memory**
- One process may want more memory than is in the system
 - Process insn/data footprint may be larger than main memory
 - **Requires main memory to act like a cache**
 - With disk as next level in memory hierarchy (slow)
 - Write-back, write-allocate, large blocks or “pages”
- Solution:
 - Part #1: treat memory as a “cache”
 - Store the overflowed blocks in “swap” space on disk
 - Part #2: add a level of indirection (address translation)

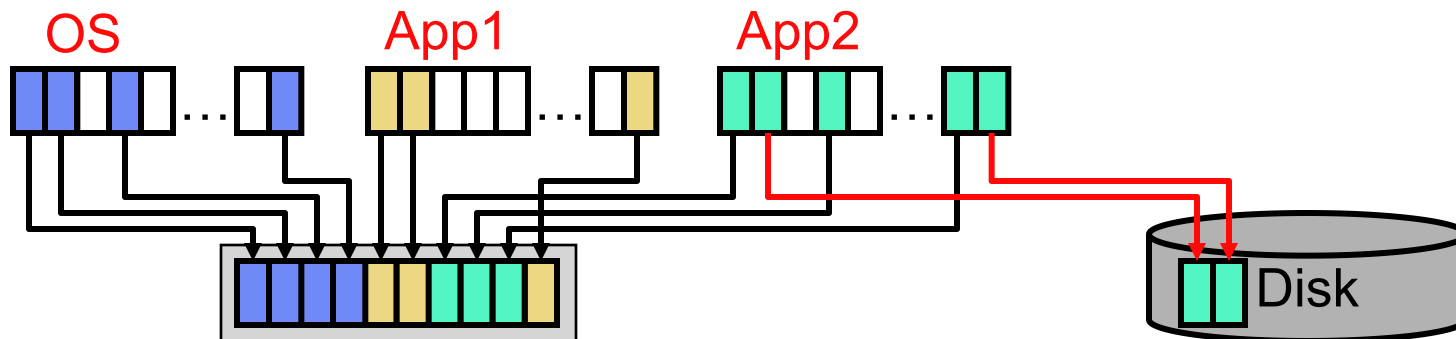
Virtual Memory (VM)

- **Virtual Memory (VM):**
 - Level of indirection
 - Application generated addresses are **virtual addresses (VAs)**
 - Each process *thinks* it has its own 2^N bytes of address space
 - Memory accessed using **physical addresses (PAs)**
 - VAs translated to PAs at coarse (page) granularity
 - OS controls VA to PA mapping for itself and all other processes
 - Logically: translation performed **before** every insn fetch, load, store
 - Physically: hardware acceleration removes translation overhead



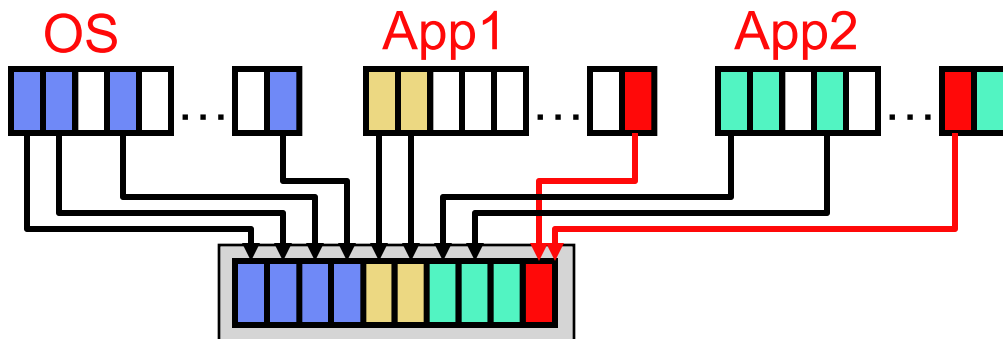
Virtual Memory (VM)

- Programs use **virtual addresses (VA)**
 - VA size (N) aka pointer size (these days, 64 bits)
- Memory uses **physical addresses (PA)**
 - PA size (M) typically $M < N$, especially if $N = 64$
 - 2^M is most physical memory machine supports
 - 48 bits in modern machines, Intel/AMD discussing 56 bits
- VA→PA at **page** granularity (VP→PP)
 - Mapping need not preserve contiguity
 - VP need not be mapped to any PP
 - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)

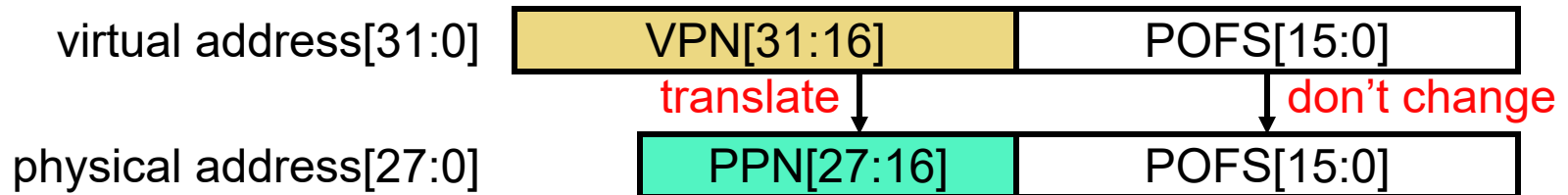


Uses of Virtual Memory

- **Isolation** and **multi-programming**
 - Each app thinks it has 2^N B of memory, its stack starts 0xFFFFFFFF,...
 - Apps prevented from reading/writing each other's memory
 - Can't even address the other program's memory!
- **Protection**
 - Each page with a read/write/execute permission set by OS
 - Enforced by hardware
- **Inter-process communication.**
 - Map same physical pages into multiple virtual address spaces
 - Or share files via the UNIX `mmap()` call



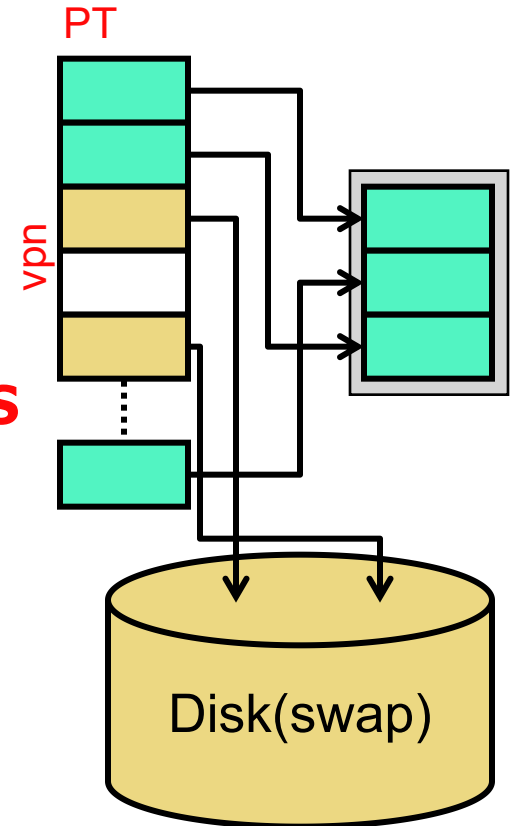
Address Translation



- VA→PA mapping called **address translation**
 - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
 - Translate VPN into **physical page number (PPN)**
 - POFS is not translated
 - $VA \rightarrow PA = [VPN, POFS] \rightarrow [PPN, POFS]$
- Example above
 - 64KB pages → 16-bit POFS
 - 32-bit machine → 32-bit VA → 16-bit VPN
 - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Address Translation Mechanics I

- How are addresses translated?
 - In software (for now) but with hardware acceleration (a little later)
- Each process has a **page table (PT)**
 - **Software data structure constructed by OS**
 - Maps VPs to PPs or to disk (swap) addresses
 - VP entries empty if page never referenced
 - Translation is table lookup



Page Table Example

Example: Memory access at address 0xFFA8AFBA

Address of Page Table Root

0xFFFF87F8

Virtual Page Number

Page Offset

1111 1111 1010 1000

1010 1111 1011 1010

0

11111111 10101000

...

...

...

...

1111 1010 1111

...

1111111111111111

...

Physical Address:

1111 1010 1111

1010 1111 1011 1010

Physical Page Number

Page Offset

Page Table Size

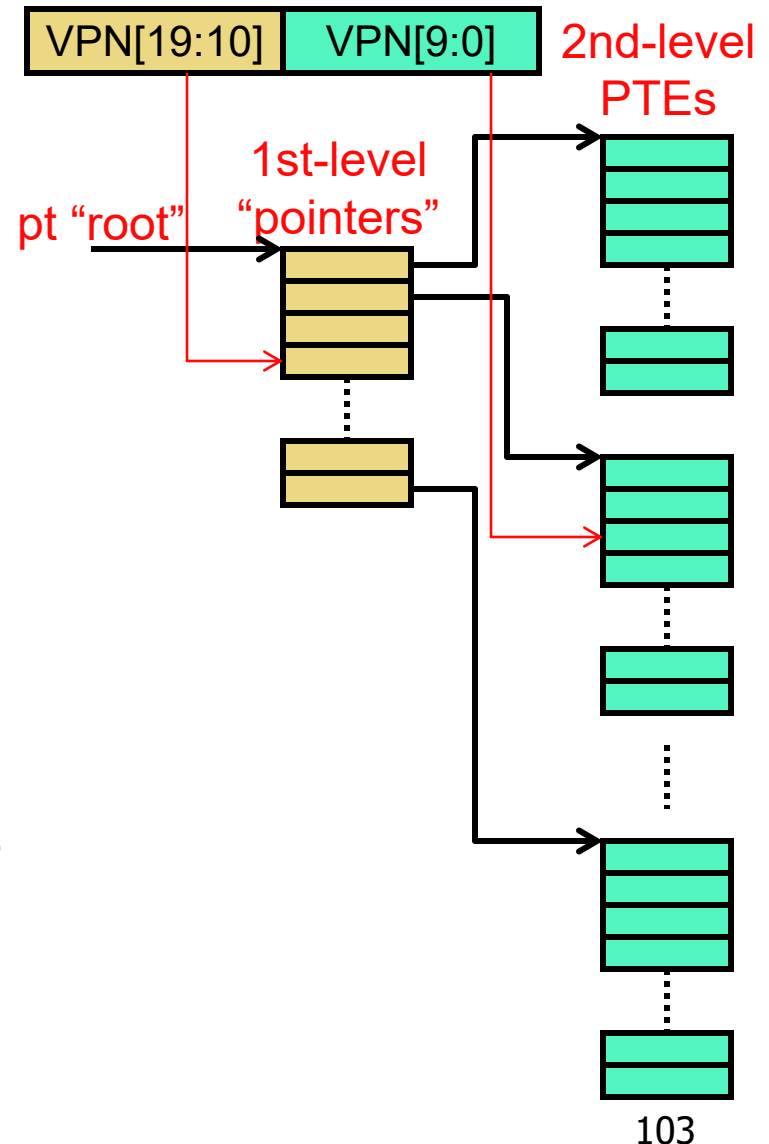
- How big is a page table on the following machine?
 - 32-bit machine
 - 4B page table entries (PTEs)
 - 4KB pages



- 32-bit machine \rightarrow 32-bit VA $\rightarrow 2^{32} = 4\text{GB}$ virtual memory
 - 4GB virtual memory / 4KB page size $\rightarrow 1\text{M}$ VPs
 - $1\text{M VPs} * 4 \text{ Bytes per PTE} \rightarrow 4\text{MB}$
- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get big
 - There are ways of making them smaller

Multi-Level Page Table (PT)

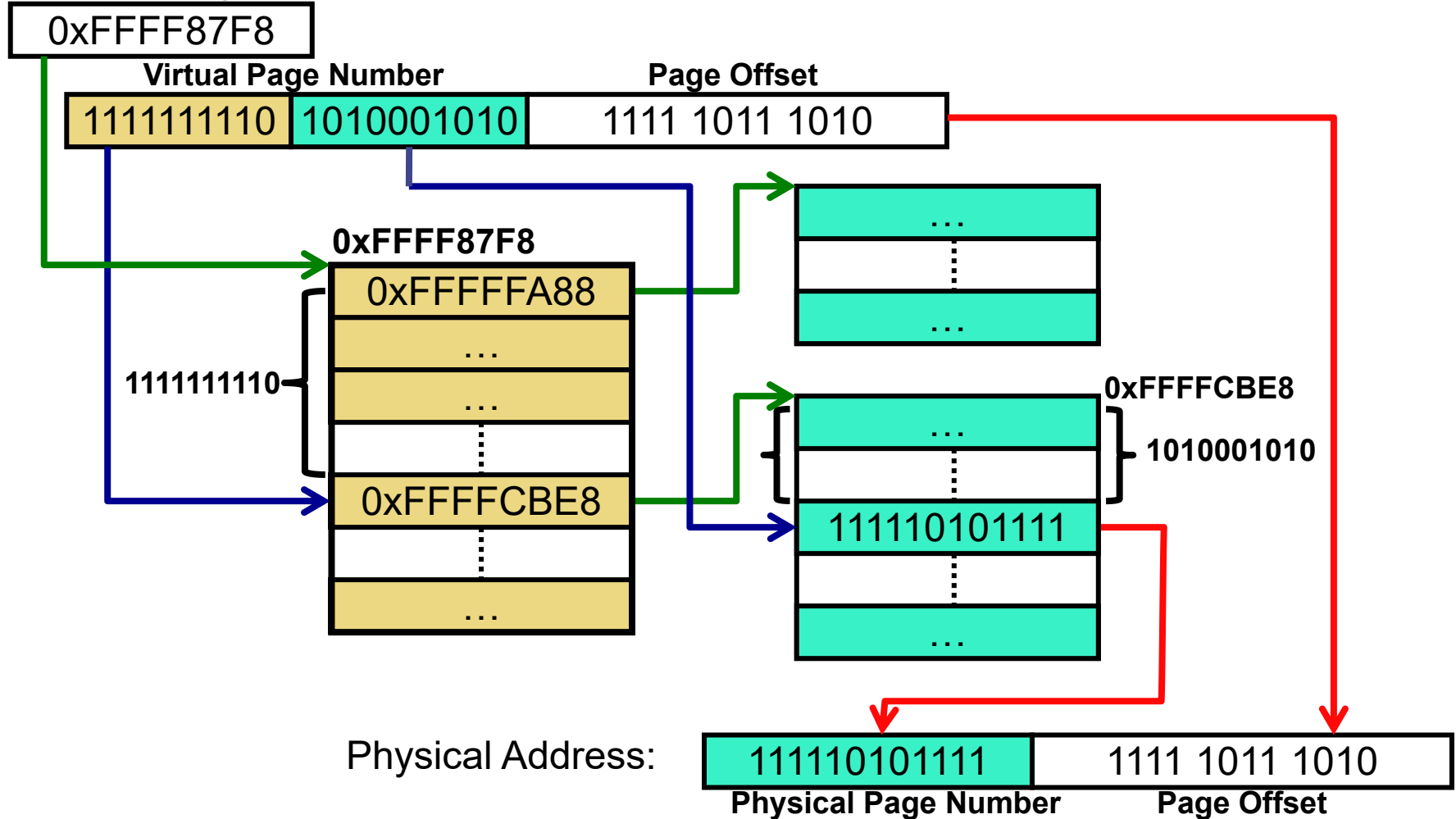
- One way:
multi-level page tables
 - Tree of page tables ("trie")
 - Lowest-level tables hold PTEs
 - Upper-level tables hold pointers to lower-level tables
 - Different parts of VPN used to index different levels
- 20-bit VPN
 - Upper 10 bits index 1st-level table
 - Lower 10 bits index 2nd-level table
 - In reality, often more than 2 levels



Multi-Level Address Translation

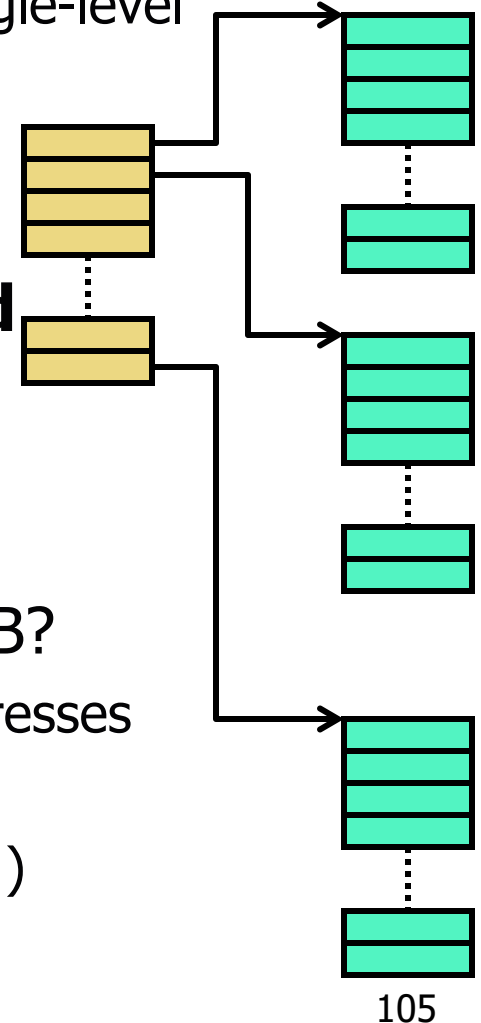
Example: Memory access at address 0xFFA8AFBA

Address of Page Table Root



Multi-Level Page Table (PT)

- Have we saved any space?
 - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
 - Yes, but...
- Large virtual address regions are **unused**
 - Corresponding 2nd-level tables unallocated
 - Corresponding 1st-level pointers are *null*
- How large for contiguous layout of 256MB?
 - Each 2nd-level table maps 4MB of virtual addresses
 - One 1st-level + 64 2nd-level pages
 - 65 total pages = 260KB (much less than 4MB!)



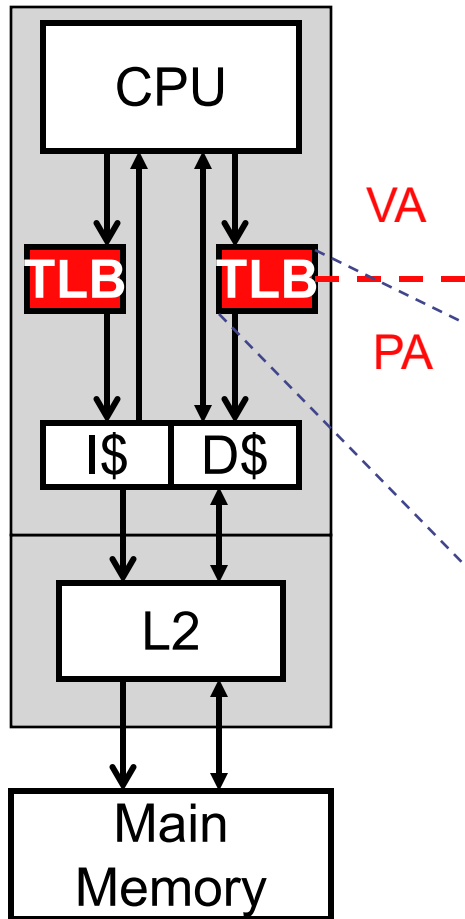
Page Sizes

- More ISAs support multiple page sizes
 - x86: 4KB, 2MB, 1GB
- larger pages have pros and cons
 - + reduce page table size
 - fewer entries needed to map a given amount of address space
 - + page table can be shallower
 - makes page table lookups faster
 - “internal fragmentation” that wastes physical memory
 - allocate 2MB page but use only 5KB of it
 - complex implementation
 - OS looks for opportunities to use large pages

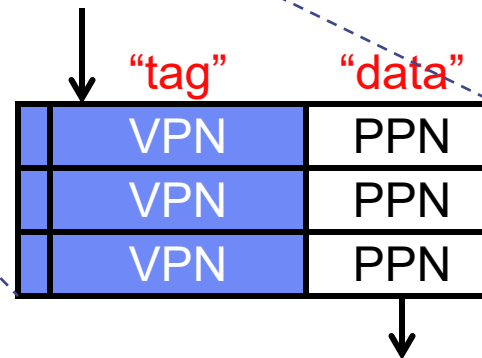
Address Translation Mechanics II

- Conceptually
 - Translate VA to PA **before every cache access**
 - Walk the page table before every load/store/insn-fetch
 - Would be terribly inefficient (even in hardware)
- In reality
 - **Translation Lookaside Buffer (TLB)**: cache translations
 - Only walk page table on TLB miss
- Computer system design truisms
 - Functionality problem? Add indirection (e.g., VM)
 - Performance problem? Add cache (e.g., TLB)

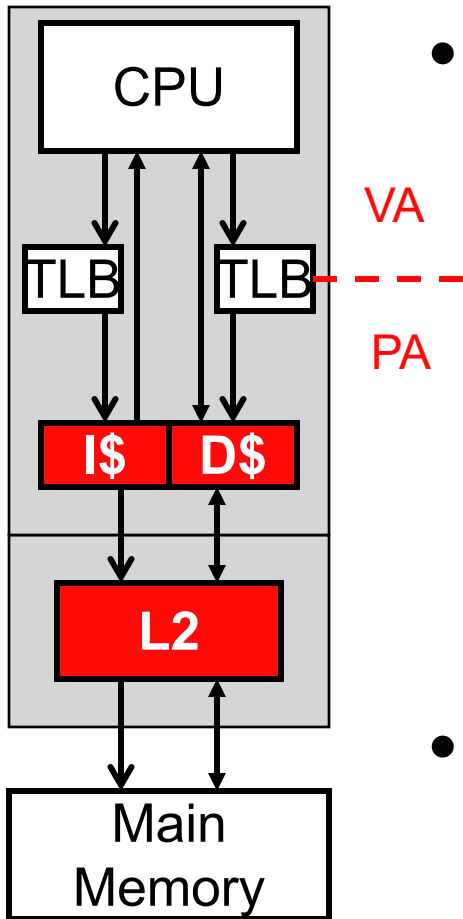
Translation Lookaside Buffer



- **Translation lookaside buffer (TLB)**
 - Small cache: 16–64 entries
 - Associative (4+ way or fully associative)
 - + Exploits temporal locality in page table
 - What if an entry isn't found in the TLB?
 - Invoke TLB miss handler, walk page table

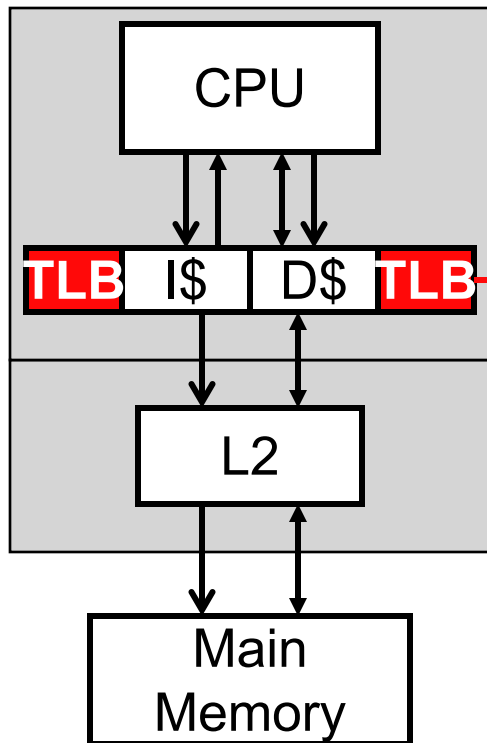


Serial TLB & Cache Access

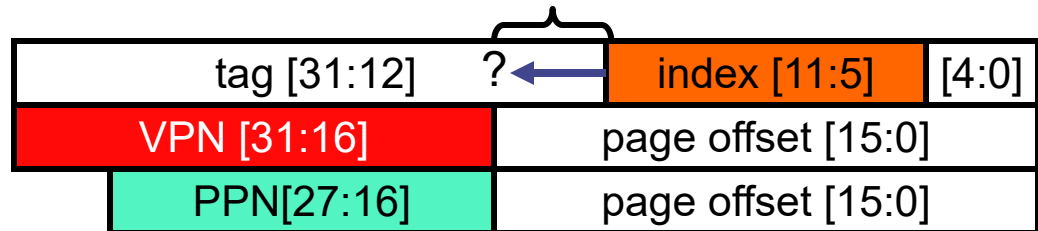


- **“Physical” caches**
 - Indexed and tagged by **physical addresses**
 - + Natural, “lazy” sharing of caches between apps/OS
 - VM ensures isolation (via **physical addresses**)
 - No need to do anything on context switches
 - Multi-threading works too
 - + Cached inter-process communication works
 - Single copy indexed by physical address
 - Slow: adds at least one cycle to t_{hit}
- Note: **TLBs are by definition “virtual”**
 - Indexed and tagged by **virtual addresses**
 - Flush across context switches
 - Or extend with process identifier tags (x86)

Parallel TLB & Cache Access



VA
PA



- What about parallel access?
 - Only if...
 - $(\text{cache size}) / (\text{associativity}) \leq \text{page size}$**
 - Index bits same in VA and PA!
- Access TLB in parallel with cache
 - Cache access needs tag only at very end
 - + Fast: no additional t_{hit} cycles
 - + No context-switching/aliasing problems
 - Dominant organization used today
- Index bits constrain capacity
 - but associativity allows bigger caches

TLB Organization

- **Like caches:** TLBs also have ABCs
 - Capacity
 - Associativity (At least 4-way associative, fully-associative common)
 - What does it mean for a TLB to have a block size of two?
 - Two VPs can a single tag
 - VPs must be aligned and consecutive
 - **Like caches:** there are second-level TLBs
- Example: AMD Opteron
 - 32-entry fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
 - 4KB pages, 48-bit virtual addresses, four-level page table
- **Rule of thumb:** TLB should “cover” size of on-chip caches
 - In other words: $(\text{\#PTEs in TLB}) * \text{page size} \geq \text{cache size}$
 - Why? Consider relative miss latency in each...

TLB Misses

- **TLB miss:** translation not in TLB, but in page table
 - Two ways to “fill” it, both relatively fast
- **Hardware-managed TLB:** e.g., x86, recent SPARC, ARM
 - Page table root in hardware register, hardware “walks” table
 - + Latency: saves cost of OS call (avoids pipeline flush)
 - Page table format is hard-coded
- **Software-managed TLB:** e.g., Alpha, MIPS
 - Short (~10 insn) OS routine walks page table, updates TLB
 - + Keeps page table format flexible
 - Latency: one or two memory accesses + OS call (pipeline flush)
- hardware TLB miss handler is popular today

Out of Order

Dynamic Scheduling

- Also called “**Out of Order Execution**”
 - Done by the hardware on-the-fly during execution
- Idea: Move the dependent instructions out of the way of independent ones (s.t. independent ones can execute)
 - Rest areas for dependent instructions: **Reservation stations**
- Monitor the source “values” of each instruction in the resting area
- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
 - Instructions dispatched in dataflow (not control-flow) order
- Benefit:
 - **Latency tolerance**: Allows independent instructions to execute and complete in the presence of a long-latency operation

Enabling OoO Execution

1. Need to link the consumer of a value to the producer
 - **Register renaming:** Associate a “tag” with each data value
2. Need to buffer instructions until they are ready to execute
 - Insert instruction into **reservation stations** after renaming
3. Instructions need to keep track of readiness of source values
 - **Broadcast the “tag”** when the value is produced
 - Instructions **compare their “source tags”** to the broadcast tag → if match, source value becomes ready
4. When all source values of an instruction are ready, need to dispatch the instruction to its functional unit (FU)
 - Instruction **wakes up** if all sources are ready
 - If multiple instructions are awake, need to **select** one per FU

Tomasulo's Algorithm for OoO Execution

Key Technologies

- Each compute unit maintains a **reservation station** (physical registers)
 - buffer the insts who are not ready
- **Register renaming** to remove WAW and WAR dependencies
 - Rename Register ID to RS entry ID
- Out of Order dispatching

Tomasulo's Algorithm

Reservation Station

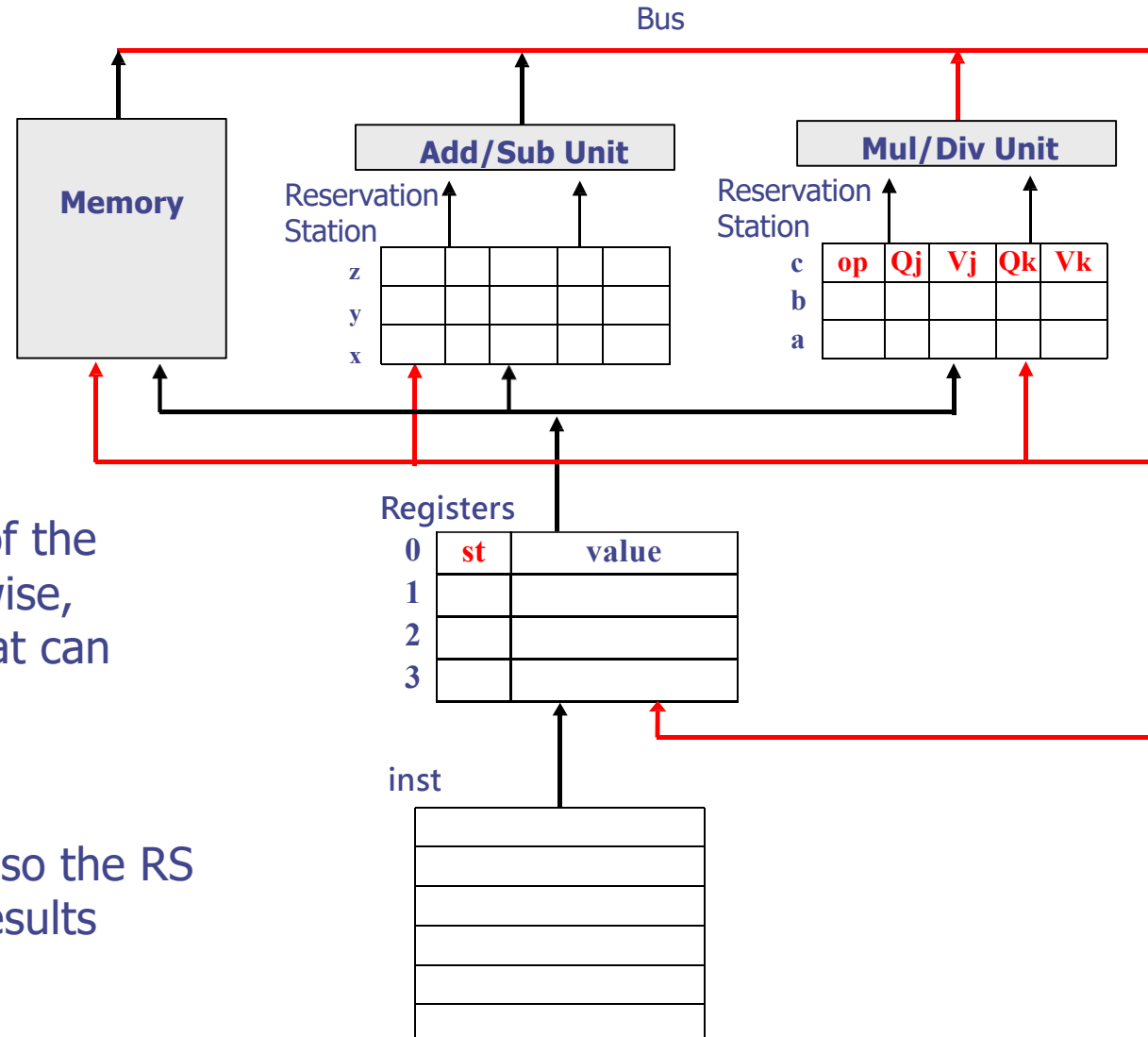
- **Op**: op code of the inst
- **Vj, Vk**: values of operands
- **Qj, Qk**: RS entry ID that will provide the value (0 means the value is ready)

Registers (add a new field):

- **st**: empty means the value of the register can be used, otherwise, indicates the RS entry ID that can provide the value

Bus

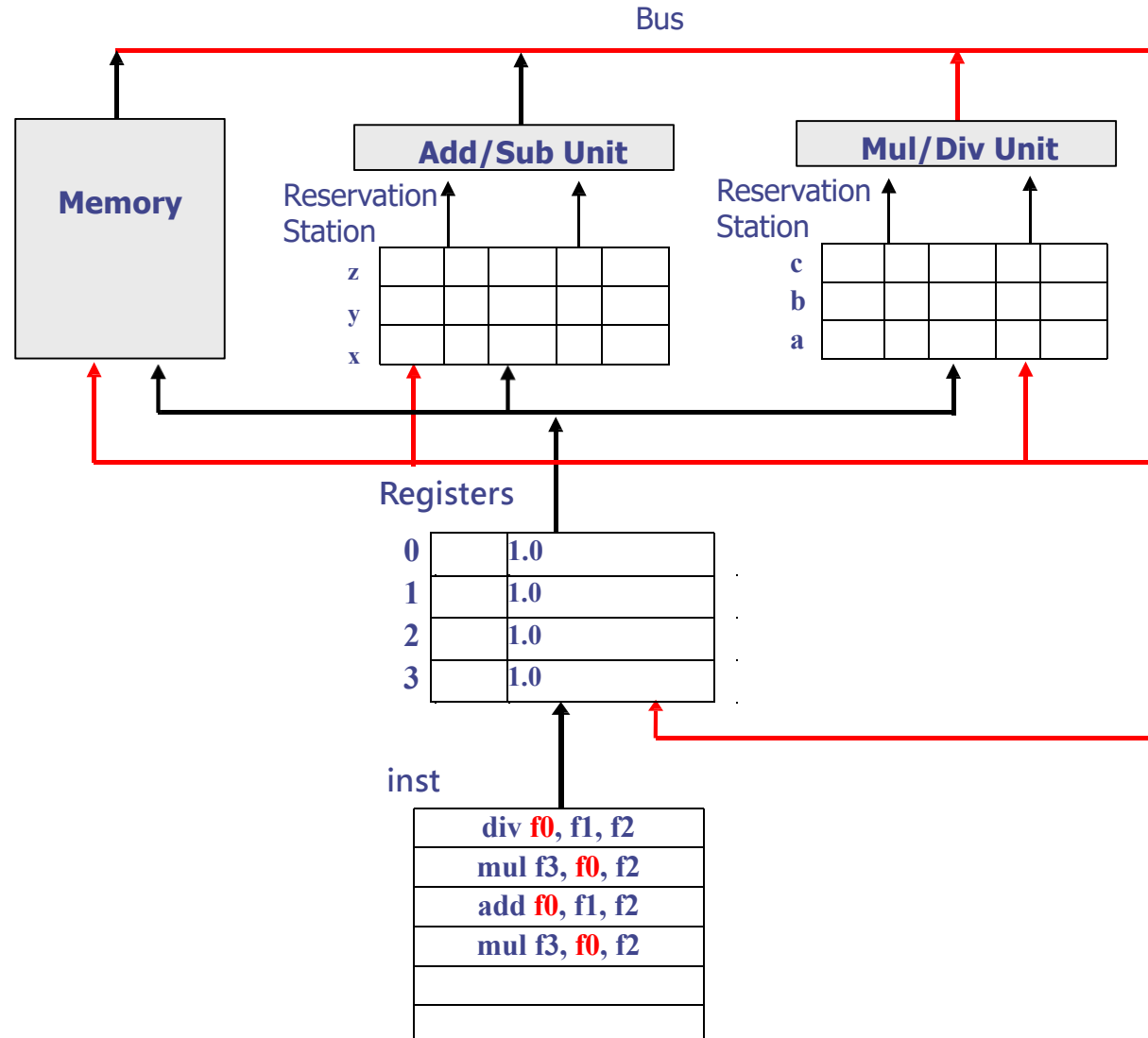
- broadcast the results, and also the RS entry ID that provides the results



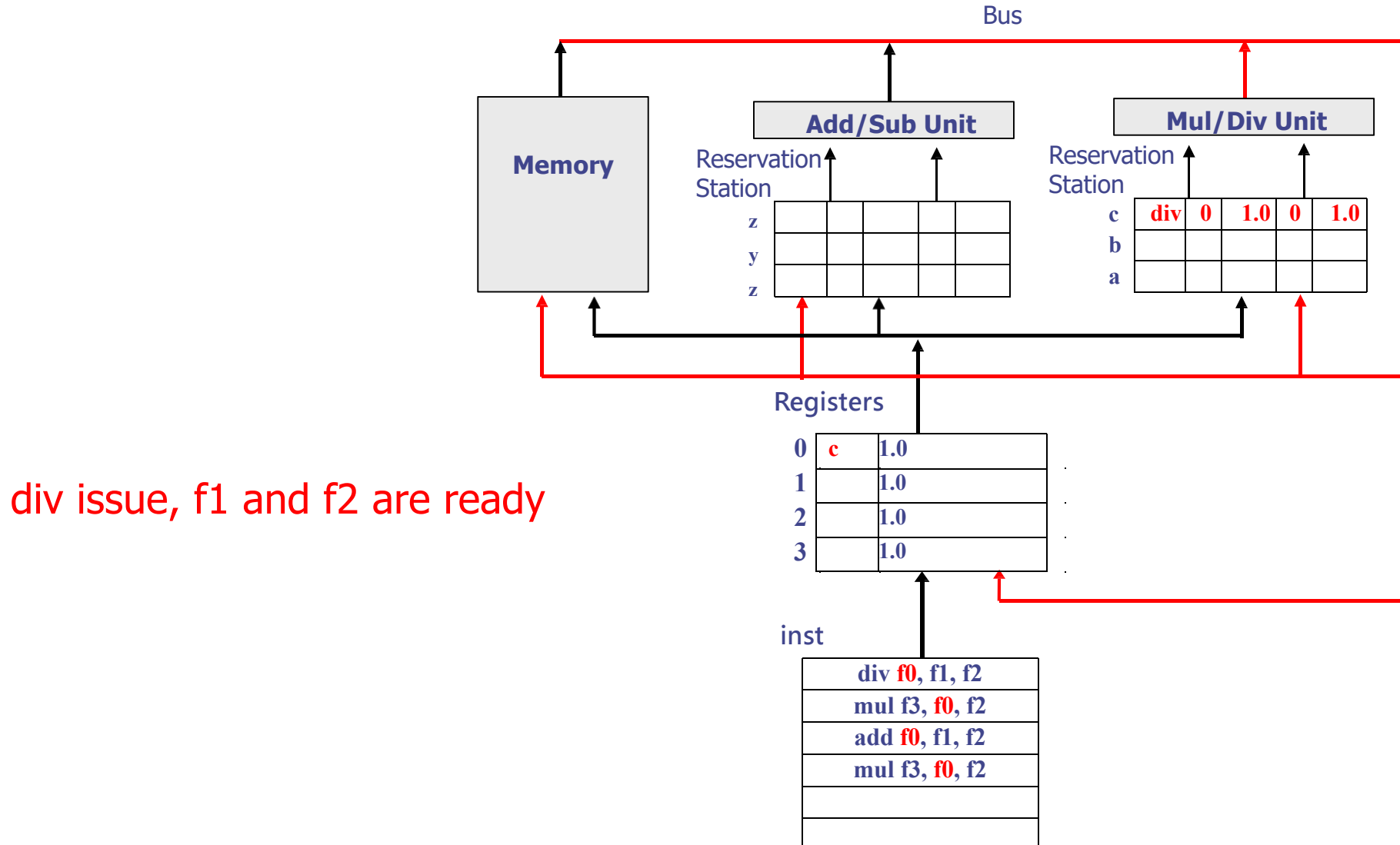
Tomasulo's Algorithm

- If reservation station available before renaming
 - Instruction + renamed operands (source value/tag) inserted into the reservation station
 - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
 - Watches common data bus (CDB) for tag of its sources
 - When tag seen, grab value for the source and keep it in the reservation station
 - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
 - Arbitrate for CDB
 - Put tagged value onto CDB (tag broadcast)
 - Register file is connected to the CDB
 - Register contains a tag indicating the latest writer to the register
 - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
 - Reclaim rename tag
 - no valid copy of tag in system!

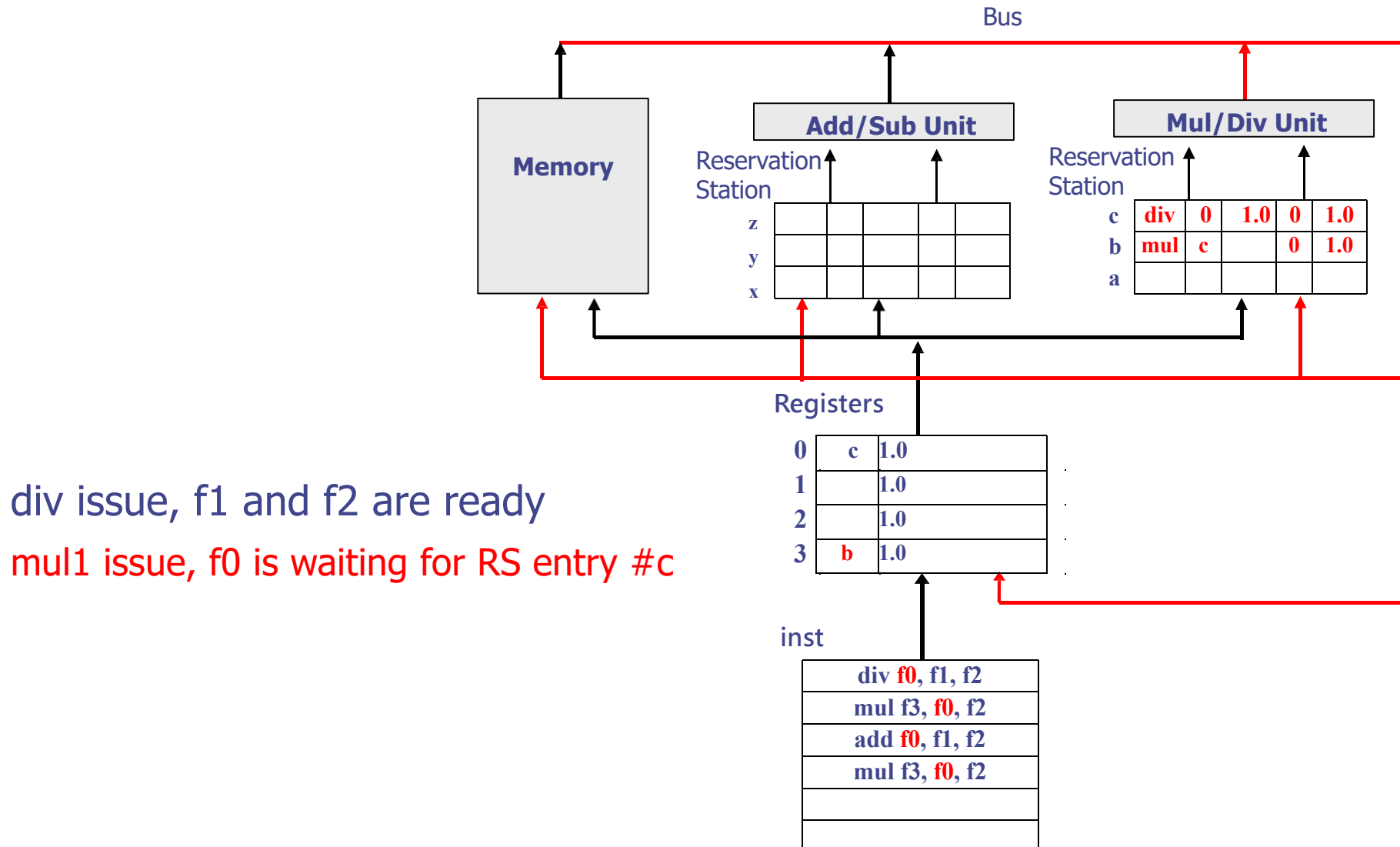
Tomasulo's Algorithm Example



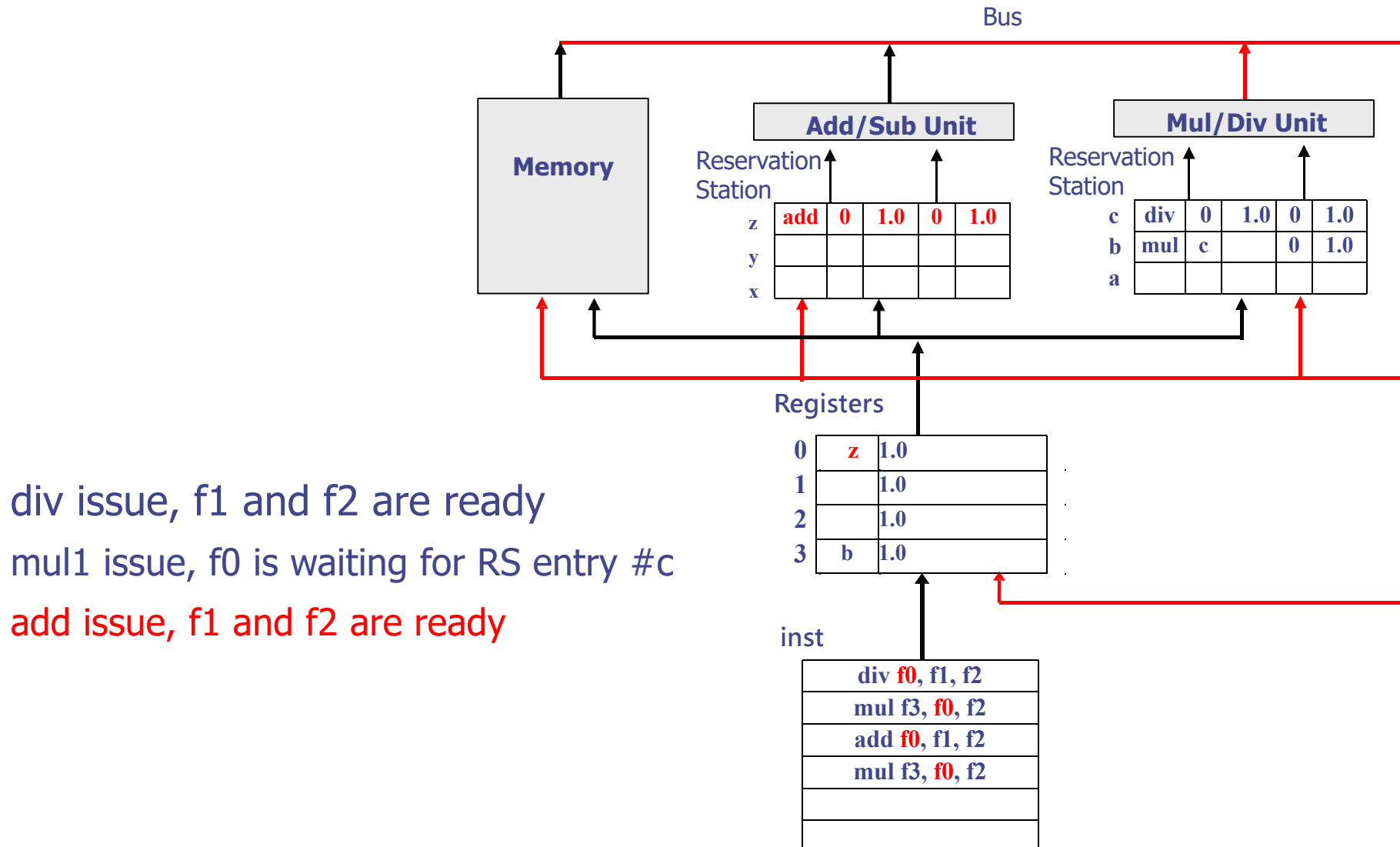
Tomasulo's Algorithm Example



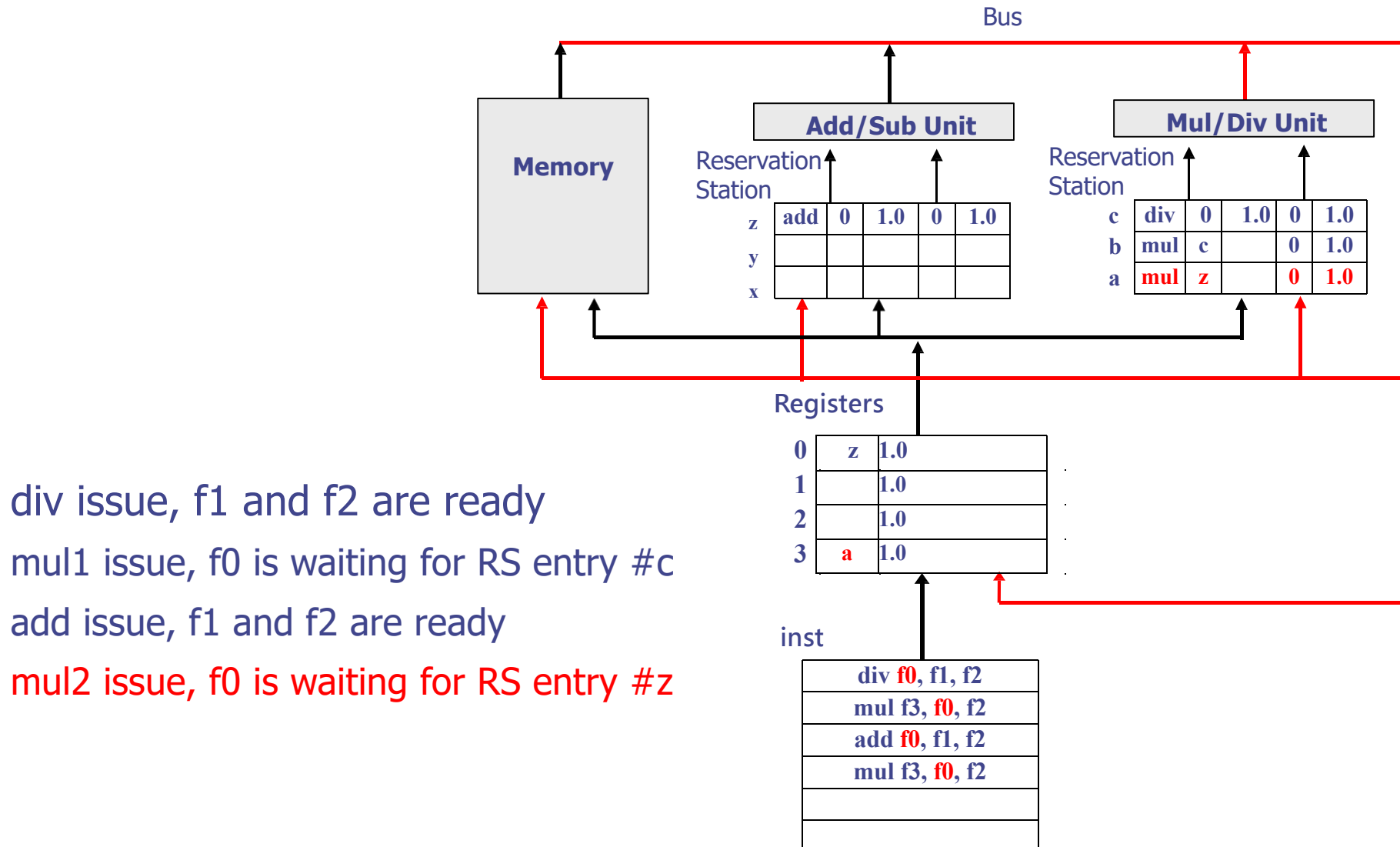
Tomasulo's Algorithm Example



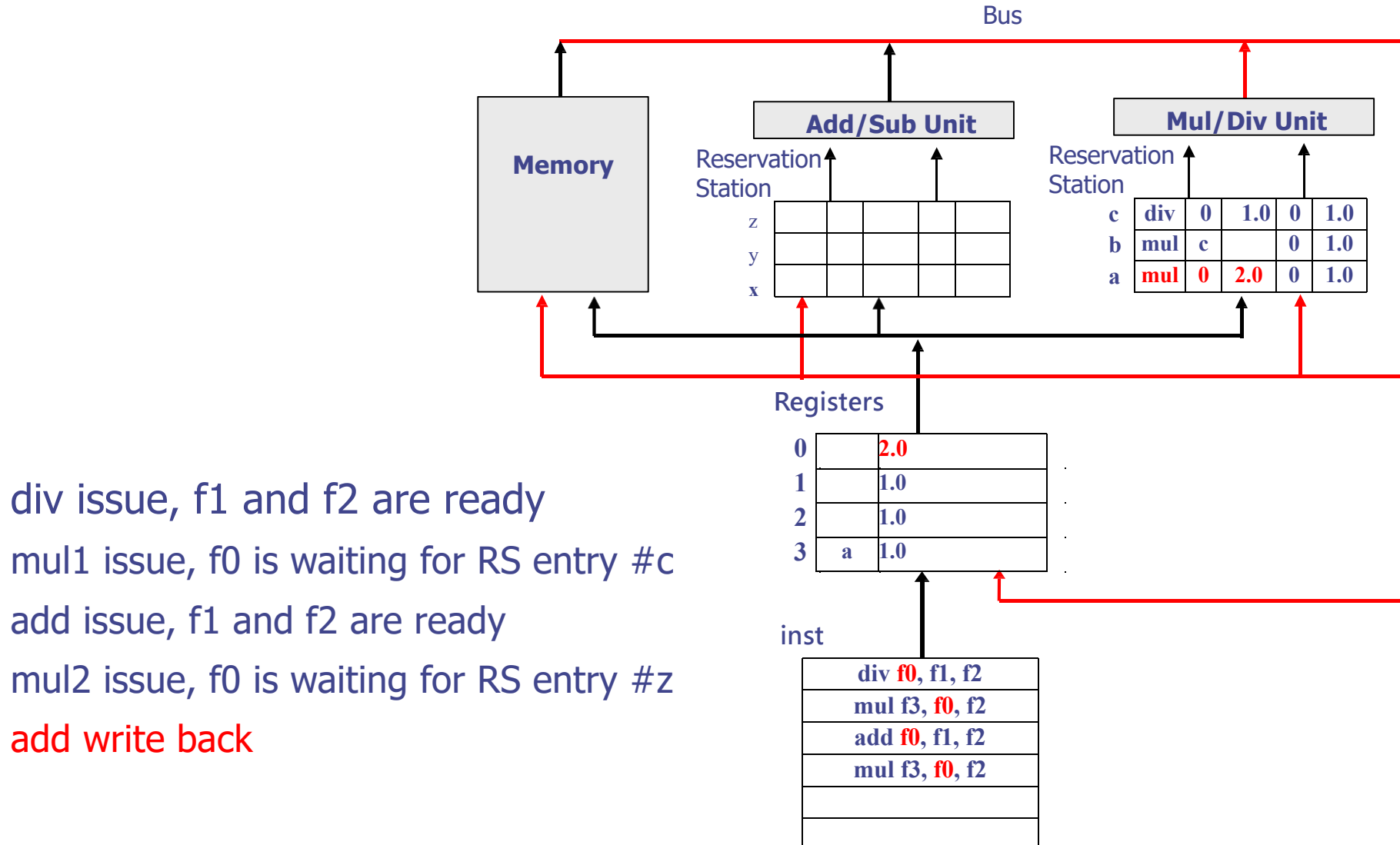
Tomasulo's Algorithm Example



Tomasulo's Algorithm Example

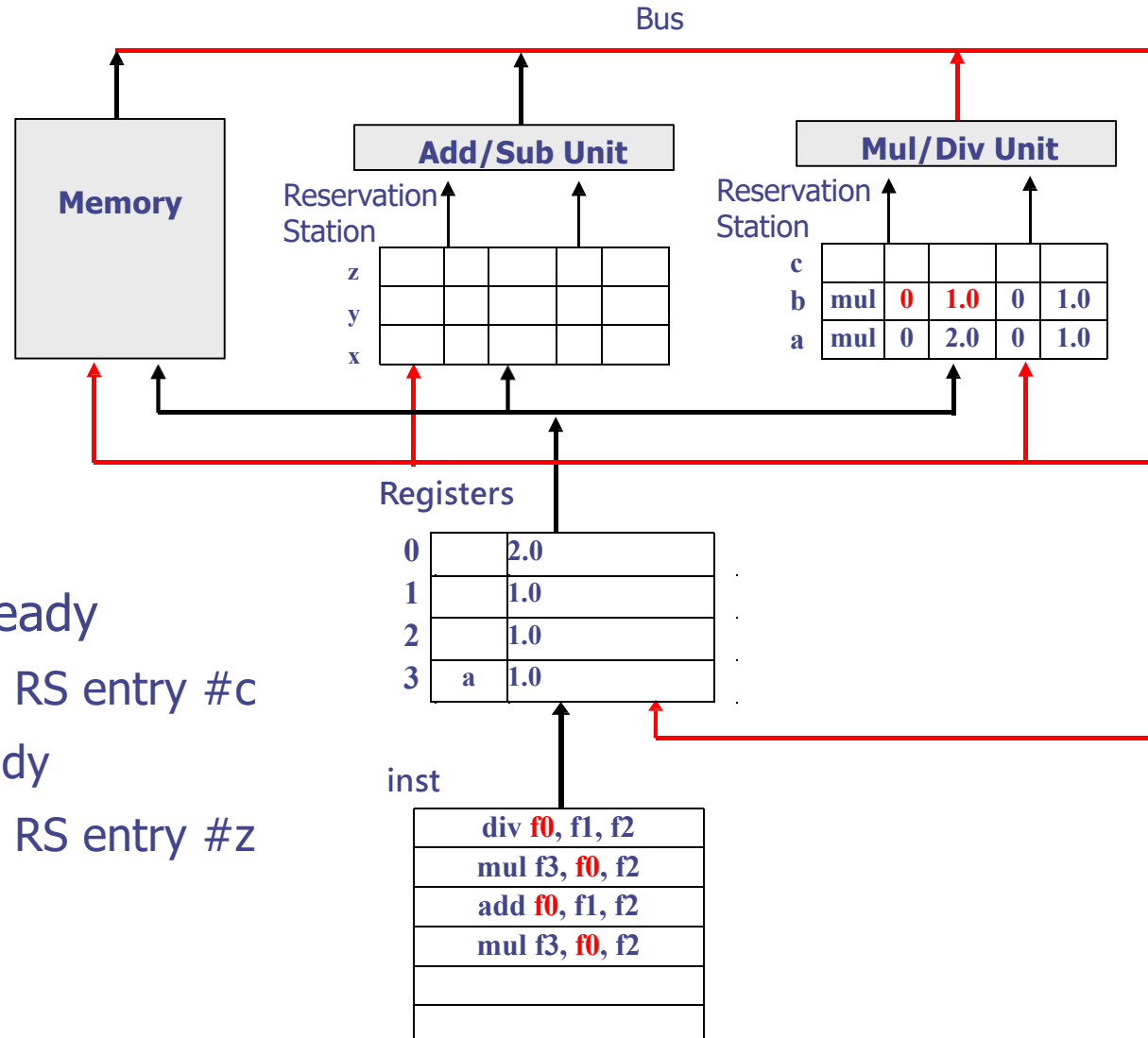


Tomasulo's Algorithm Example



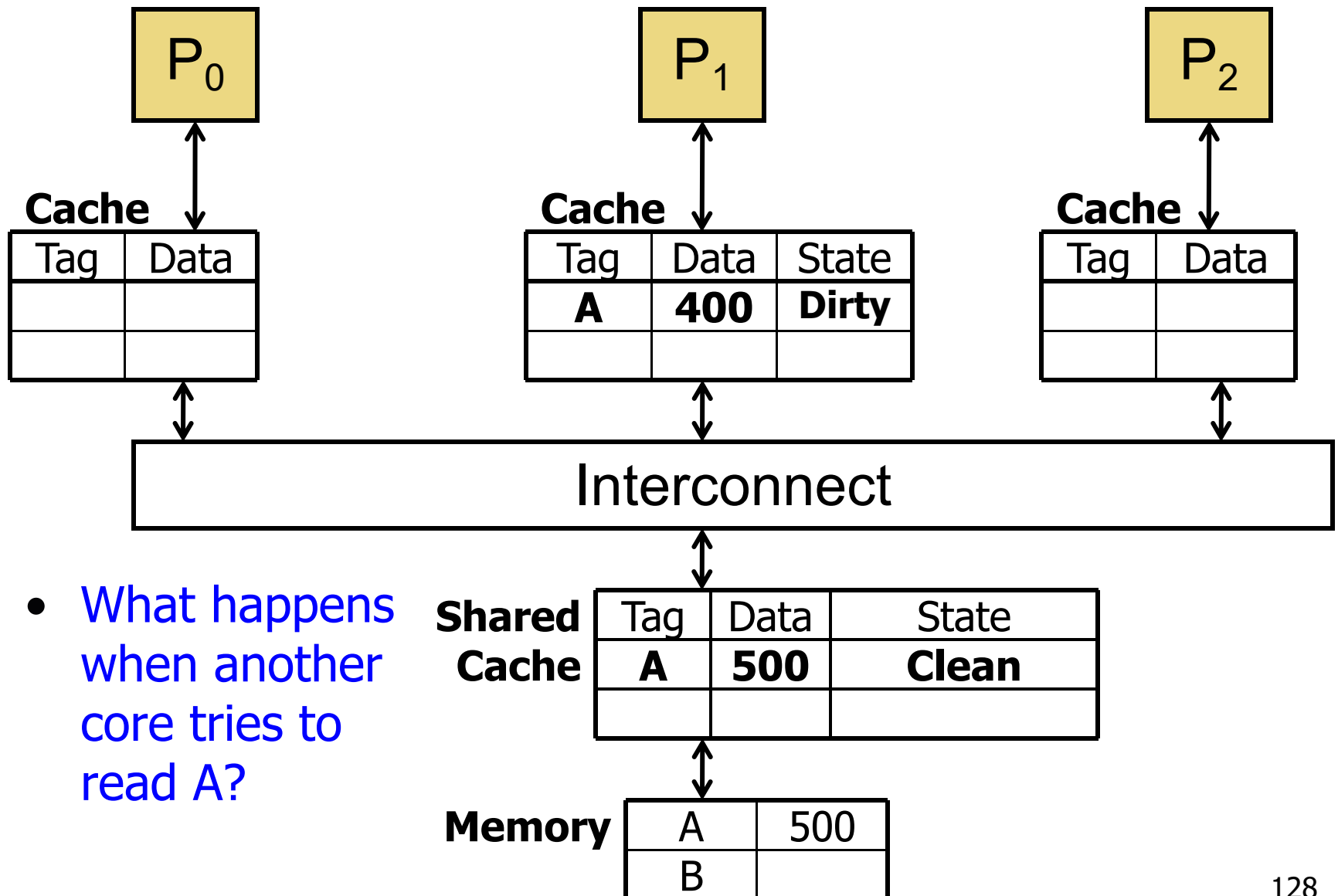
Tomasulo's Algorithm Example

div issue, f1 and f2 are ready
 mul1 issue, f0 is waiting for RS entry #c
 add issue, f1 and f2 are ready
 mul2 issue, f0 is waiting for RS entry #z
 add write back
 div write back



Cache Coherence

Private Cache Problem: Incoherence



“Valid/Invalid” Cache Coherence

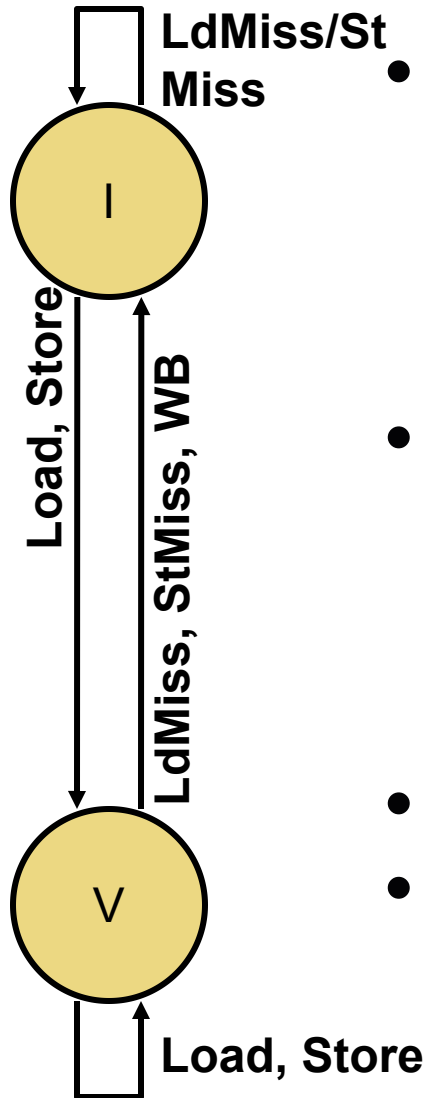
- To enforce the shared memory invariant...
 - “Loads read the value written by the most recent store”
- Enforce the invariant...
 - **“At most one valid copy of the block”**
 - Simplest form is a **two-state “valid/invalid” protocol**
 - If a core wants a copy, must find and “invalidate” it
- On a cache miss, how is the valid copy found?
 - Option #1 **“Snooping”**: broadcast to all, whoever has it responds
 - Option #2: **“Directory”**: track sharers with separate structure
- **Problem**: multiple copies can’t exist, even if read-only
 - Consider mostly-read data structures, instructions, etc.

VI Protocol State Transition Table

State	<i>This Processor</i>		<i>Other Processor</i>	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Load Miss → V	Store Miss → V	---	---
Valid (V)	Hit	Hit	Send Data → I	Send Data → I

- Rows are "states"
 - I vs V
- Columns are "events"
 - Writeback events not shown
- Memory controller not shown
 - **Memory sends data when no processor responds**

VI (Modified I) Coherence Protocol



- **VI (valid-invalid) protocol:** aka “MI”
 - Two states (per block in cache)
 - **V (valid):** have block
 - **I (invalid):** don’t have block
 - + Can implement with valid bit
- Protocol diagram (left & next slide)
 - Summary
 - If anyone wants to read/write block
 - Give it up: transition to **I** state
 - Write-back if your own copy is dirty
- This is an **invalidate protocol**
- **Update protocol:** copy data, don’t invalidate
 - Sounds good, but uses too much bandwidth

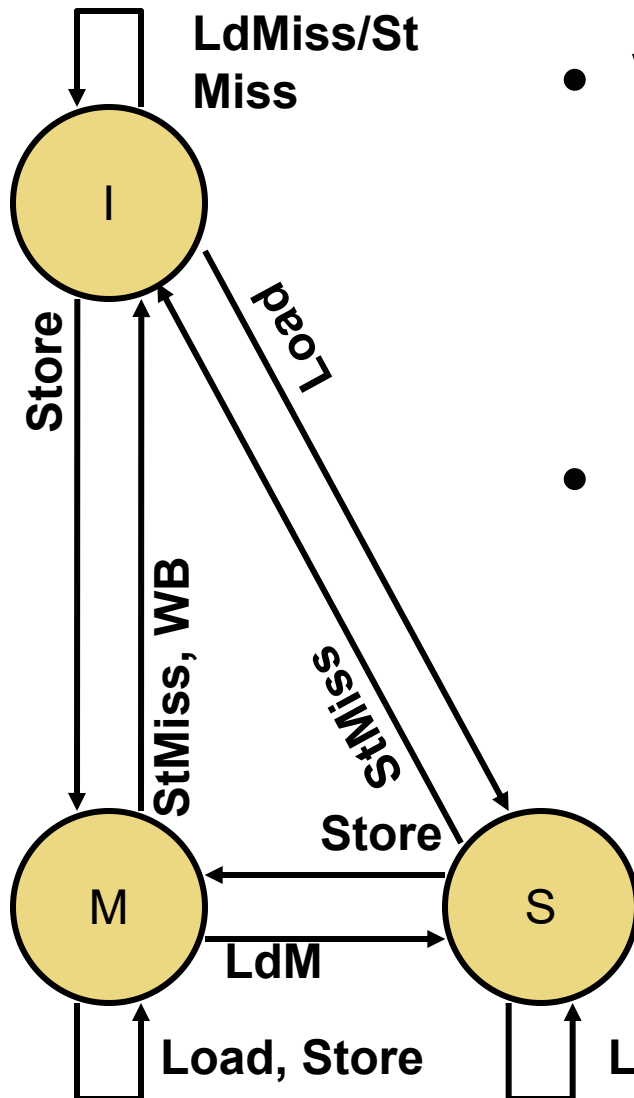
MSI Cache Coherence Protocol

- Solution: enforce the invariant...
 - **Multiple read-only copies** —OR—
 - **Single read/write copy**
- Track these MSI permissions (states) in per-core caches
 - **Modified (M): read/write permission**
 - **Shared (S): read-only permission**
 - **Invalid (I): no permission**
- Also track a **“Sharer” bit vector** in shared cache
 - One bit per core; tracks all shared copies of a block
 - Then, *invalidate all readers* when a write occurs
- Allows for many readers...
 - ...while still enforcing shared memory invariant (“Loads read the value written by the most recent store”)

MSI Protocol State Transition Table

State	<i>This Processor</i>		<i>Other Processor</i>	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Load Miss → S	Store Miss → M	---	---
Shared (S)	Hit	Upgrade Miss → M	Send Data → S	→ I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

VI → MSI



- VI protocol is inefficient
 - Only one cached copy allowed in entire system
 - Multiple copies can't exist even if read-only
 - Not a problem in example
 - Big problem in reality
- **MSI (modified-shared-invalid)**
 - Fixes problem: splits "V" state into two states
 - **M (modified)**: local dirty copy
 - **S (shared)**: local clean copy
 - Allows **either**
 - Multiple read-only copies (S-state) **--OR--**
 - Single read/write copy (M-state)

Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
 - **Upgrade miss:** stores to read-only blocks
 - Delay to acquire write permission to read-only block
 - **Coherence miss**
 - Miss to a block evicted by another processor's requests
- Making the cache larger...
 - Doesn't reduce these types of misses
 - So, as cache grows large, these sorts of misses dominate
- **False sharing**
 - Two or more processors sharing parts of the same block
 - But *not* the same bytes within that block (no actual sharing)
 - Creates pathological "ping-pong" behavior
 - Careful data placement may help, but is difficult

MESI Cache Coherence

- Ok, we have read-only and read/write with MSI
- But consider load & then store of a block by same core
 - Under coherence as described, **this would be two misses: “Load miss” plus an “upgrade miss”...**
 - ... even if the block isn’t shared!
 - Consider programs with 99% (or 100%) private data
 - Potentially doubling number of misses (bad)
- Solution:
 - Most modern protocols also include **E (exclusive)** state
 - Interpretation: “I have the only cached copy, and it’s a **clean** copy”
 - Has read/write permissions
 - Just like “Modified” but “clean” instead of “dirty”.

MESI Operation

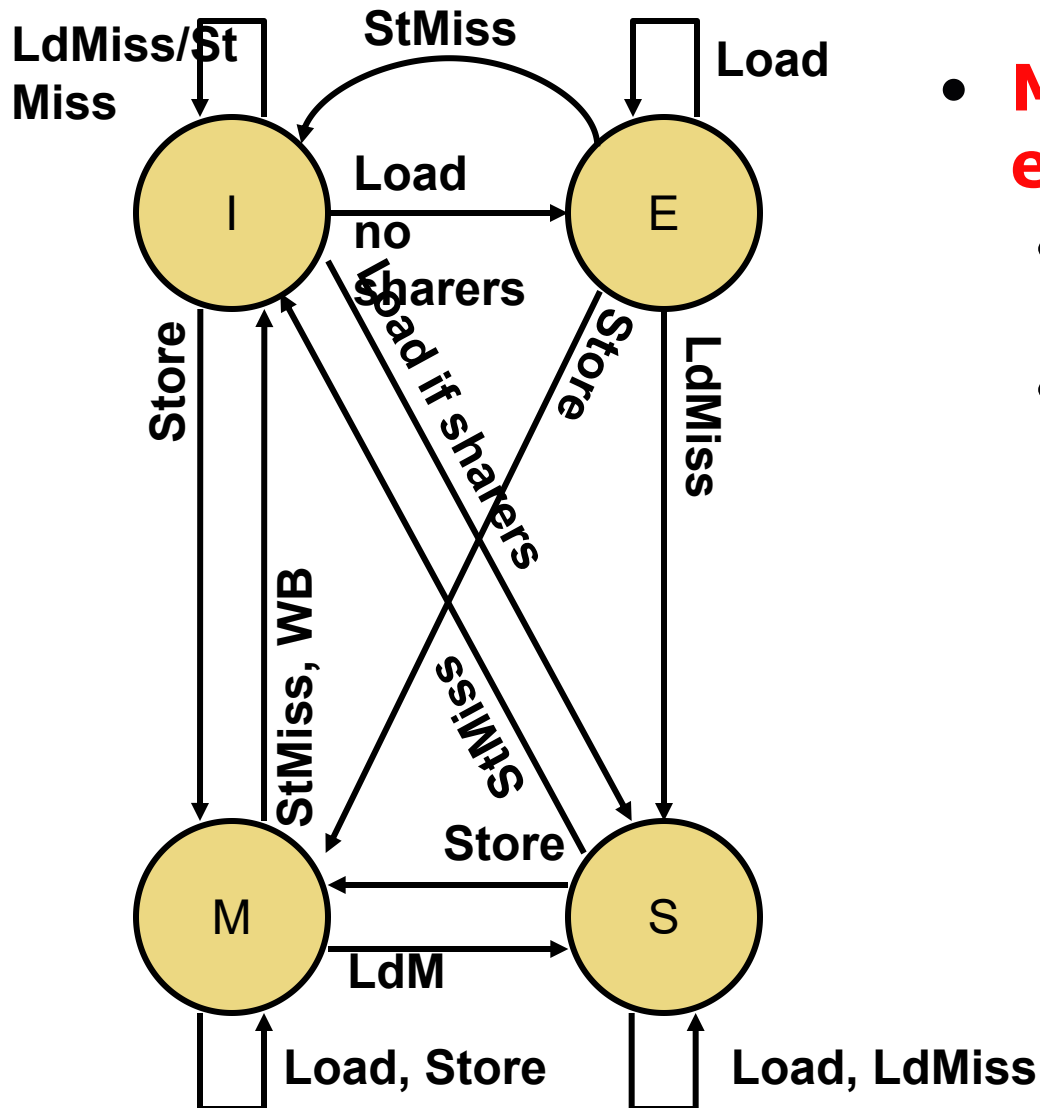
- Goals:
 - Avoid “upgrade” misses for non-shared blocks
 - While not increasing eviction (aka writeback or replacement) traffic
- Two cases on a load miss to a block...
 - **Case #1:** ... with no current sharers
(that is, no sharers in the set of sharers)
 - Grant requester “Exclusive” copy with read/write permission
 - **Case #2:** ... with other sharers
 - As before, grant just a “Shared” copy with read-only permission
- A store to a block in “Exclusive” changes it to “Modified”
 - **Instantaneously & silently** (no latency or traffic)
- On block eviction (aka writeback or replacement)...
 - If “Modified”, block is dirty, must be written back to next level
 - If “Exclusive”, writing back the data is not necessary
(but notification may or may not be, depending on the system)

MESI Protocol State Transition Table

State	<i>This Processor</i>		<i>Other Processor</i>	
	Load	Store	Load Miss	Store Miss
Invalid (I)	Miss → S or E	Miss → M	---	---
Shared (S)	Hit	Upg Miss → M	Send Data → S	→ I
Exclusive (E)	Hit	Hit → M	Send Data → S	Send Data → I
Modified (M)	Hit	Hit	Send Data → S	Send Data → I

- Load misses lead to "E" if no other processors is caching the block

MSI → MESI



- **MESI (modified-exclusive-shared-invalid)**
 - Eliminates the cost of coherence when there's no sharing
 - Keeps single-threaded programs fast on multicores

Coherence and writeback caches

- there's redundancy between dirty bit and coherence state
 - an Exclusive block is always clean
 - a Modified block is always dirty
 - a Shared block could be clean *or* dirty
 - block was Modified, then later downgraded to Shared
- When should we write back Shared blocks?
 - Track a separate dirty bit
 - Always write back Shared blocks?
 - if k writebacks occur, $k-1$ writebacks are redundant
 - Never write back Shared blocks?
 - previous stores to the Shared block are lost
- Solution: integrate dirty bit into coherence state

Memory Consistency

Memory Consistency

- **Cache coherence**
 - Creates globally uniform (consistent) view of **a single cache block**
 - Not enough on its own:
 - What about accesses to different cache blocks?
 - Some optimizations skip coherence(!)
- **Memory consistency model**
 - Specifies the semantics of shared memory operations
 - i.e., what value(s) a load may return
- Who cares? Programmers
 - Globally inconsistent memory creates mystifying behavior

3 Classes of Memory Consistency Models

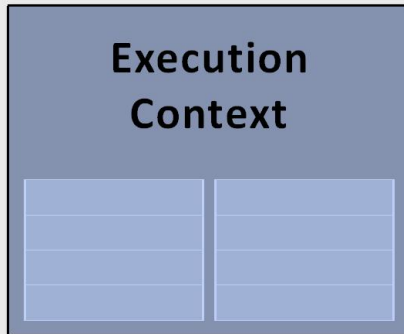
- **Sequential consistency (SC)** (MIPS, PA-RISC)
 - **Typically what programmers expect**
 - 1. Processors see their own loads and stores in program order
 - 2. Processors see others' loads and stores in program order
 - 3. All processors see same global load/store ordering
 - Corresponds to some sequential interleaving of uniprocessor orders
- **Total Store Order (TSO)** (**x86**, SPARC)
 - Allows an in-order (FIFO) store buffer
 - Stores can be deferred, but must be put into the cache in order
- **Release consistency (RC)** (**ARM**, Itanium, **PowerPC**)
 - Allows an un-ordered coalescing store buffer
 - Stores can be put into cache in any order
 - Loads re-ordered, too.

GPU

Slimming down

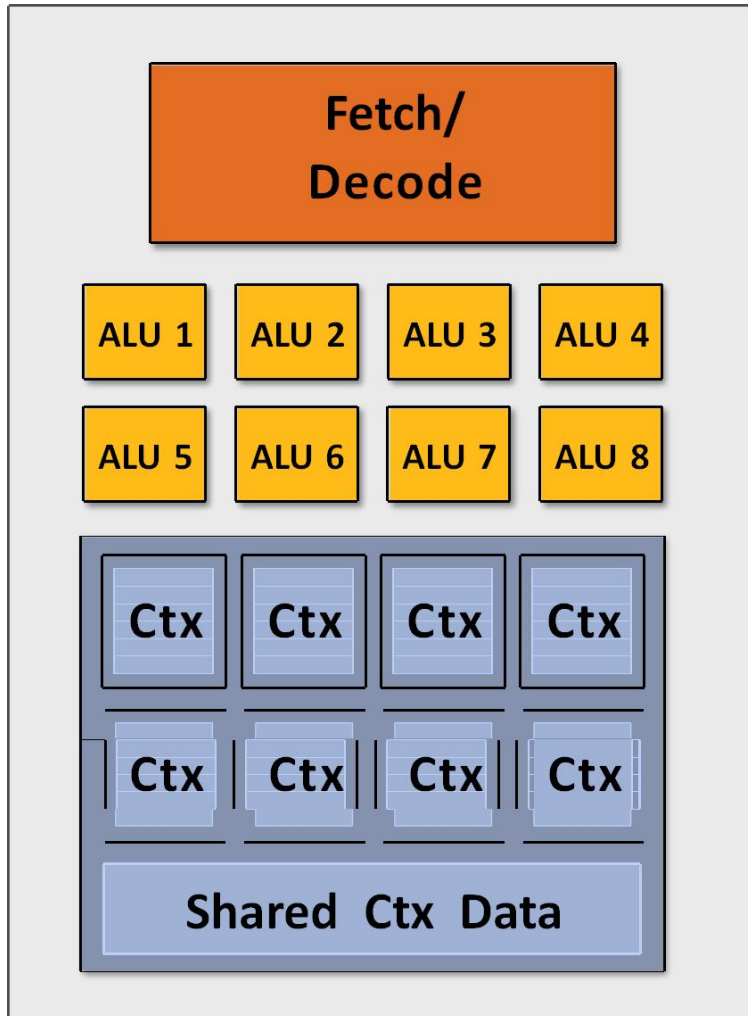
Fetch/
Decode

ALU
(Execute)



Idea #1:
**Remove components that help a
single instruction stream run fast**

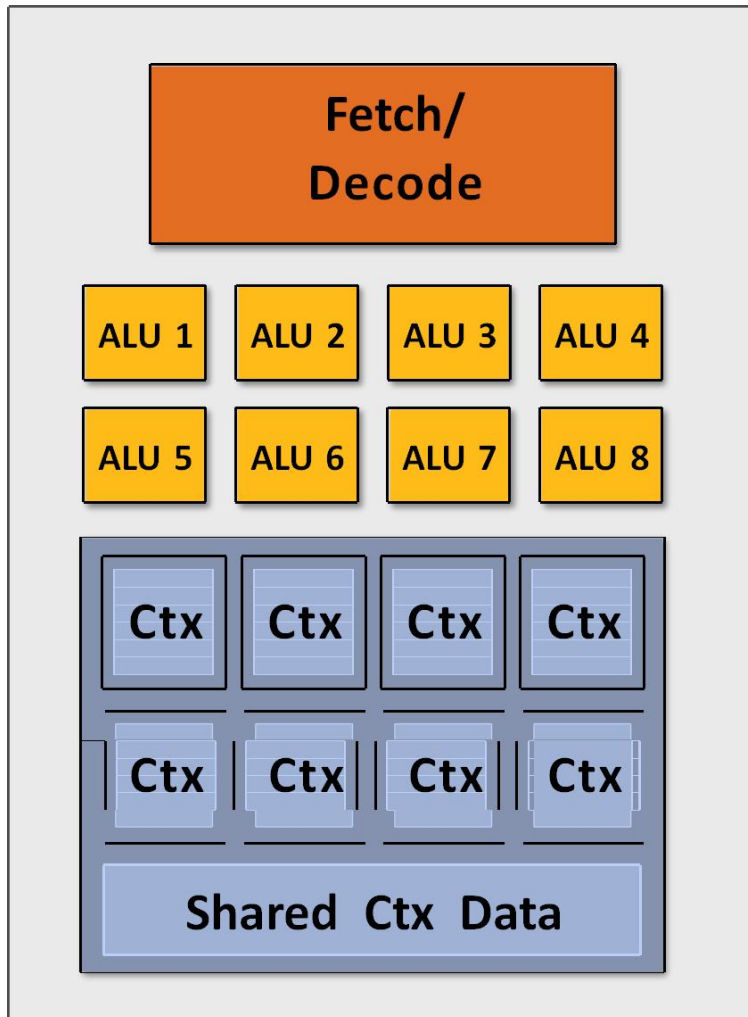
Add ALUs



Idea #2:
Amortize cost/complexity of
managing an instruction stream
across many ALUs

SIMD processing

Hiding shader stalls



Idea #3:
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

GPU Summary

- Three major ideas that all modern processors employ to varying degrees
 - Employ multiple processing cores
 - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
 - Amortize instruction stream processing over many ALUs (SIMD)
 - Increase compute capability with little extra cost
 - Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)
- Due to high arithmetic capability on modern chips, many parallel applications (on both CPUs and GPUs) are bandwidth bound
- GPU architectures use the same throughput computing ideas as CPUs: but GPUs push these concepts to extreme scales