
ECE4074

Advanced Computer Architecture

Clayton 2014

Chapter 1: Abstractions and Technology

Damien Browne

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Course Administration

- ❑ Lecturer: Damien Browne
damien.browne@monash.edu
- ❑ Labs: Check Allocate+
- ❑ URL: Moodle 2, via my.monash
- ❑ Text: Required: *Computer Org and Design*, 4th Ed., Revised Printing, Patterson & Hennessy, ©2009
- ❑ Slides: Updated pdf on Moodle after lecture

Marking Information

Marking weights

- First Assignment 15%
 - Report due Friday of Week 7
- Second Assignment 7.5%
 - Due Friday Week 10
- Third Assignment 7.5%
 - Due Friday Week 12
- Final Exam 70%

Marks and feedback will be posted on Moodle

Course Structure & Schedule

- Lectures: 12:00pm to 1 pm Tuesdays and 9am to 10 am Fridays
 - Lectures are available online
- Lab classes (from week 2)
 - 3 hours per week. Time for implementing and testing your assignments
- Lectures:
 - 1 week introduction and performance metrics
 - 2.5 week review of processor arithmetic
 - 1.5 week memory hierarchies and memory design issues
 - 1 week storage and I/O design issues
 - 1.5 weeks superscalar/VLIW data path design issues
 - 1.5 weeks multiprocessor design issues
 - 1 Week FPGA architecture
 - 1 Week revision
 - 1 week exams

Course Content

- ❑ Memory hierarchy and design, CPU design, pipelining, multiprocessor architecture.
 - “This course will introduce students to the architecture-level design issues of a computer system. They will apply their knowledge of digital logic design to explore the high-level interaction of the individual computer system hardware components. Concepts of sequential and parallel architecture including the interaction of different memory components, their layout and placement, communication among multiple processors, effects of pipelining, and performance issues, will be covered. Students will apply these concepts by studying and evaluating the merits and demerits of selected computer system architectures.”
 - To learn what determines the capabilities and performance of computer systems and to understand the interactions between the computer’s architecture and its software so that **future software designers** (compiler writers, operating system designers, database programmers, application programmers, ...) can achieve the best cost-performance trade-offs and so that **future architects** understand the effects of their design choices on software.

The effect of understanding Architecture



- ❑ These two games were released on the same hardware (Xbox 360)
- ❑ 7 years apart
- ❑ Effect is exaggerated due to other influences

What You Should Know

- ❑ Basic logic design & machine organization
 - logical minimization, FSMs, component design
 - processor, memory, I/O
- ❑ Create, assemble, run, debug programs in an assembly language
 - MIPS preferred
- ❑ Create, simulate, and debug hardware structures in a hardware description language
 - Verilog
- ❑ Create, compile, and run C (C++, Java) programs

Assignment 1

- ❑ Available on Moodle 2 now.
- ❑ Labs are time for you work on and to ask your demonstrator about the assignment.
- ❑ Task is to create a version of MIPS processor that can complete a 15x15 matrix multiply.
- ❑ You will need to start this week.
 - You need a good understanding of MIPS instruction language.
 - Start by “compiling” a matrix multiplication.
- ❑ Marks are partially based on performance of the task.
- ❑ Optimizing your algorithm will help a lot.

Assignment 1

- ❑ Verilog crash course will be integrated into lectures in the coming weeks.
 - There are also links on the moodle website to an online Verilog tutorial
- ❑ Altera Tools have been updated in the labs recently
 - Familiarize yourself with the tools ASAP
- ❑ Recommend (as an absolute minimum) that you have at least 80% of the progressive lab milestones done before you arrive at the labs.

Classes of Computers

❑ Desktop computers

- Designed to deliver good performance to a single user at low cost usually executing 3rd party software, usually incorporating a graphics display, a keyboard, and a mouse

❑ Servers

- Used to run larger programs for multiple, simultaneous users typically accessed only via a network and that places a greater emphasis on dependability and (often) security

❑ Supercomputers

- A high performance, high cost class of servers with hundreds to thousands of processors, **terabytes** of memory and **petabytes** of storage that are used for high-end scientific and engineering applications

❑ Embedded computers (processors)

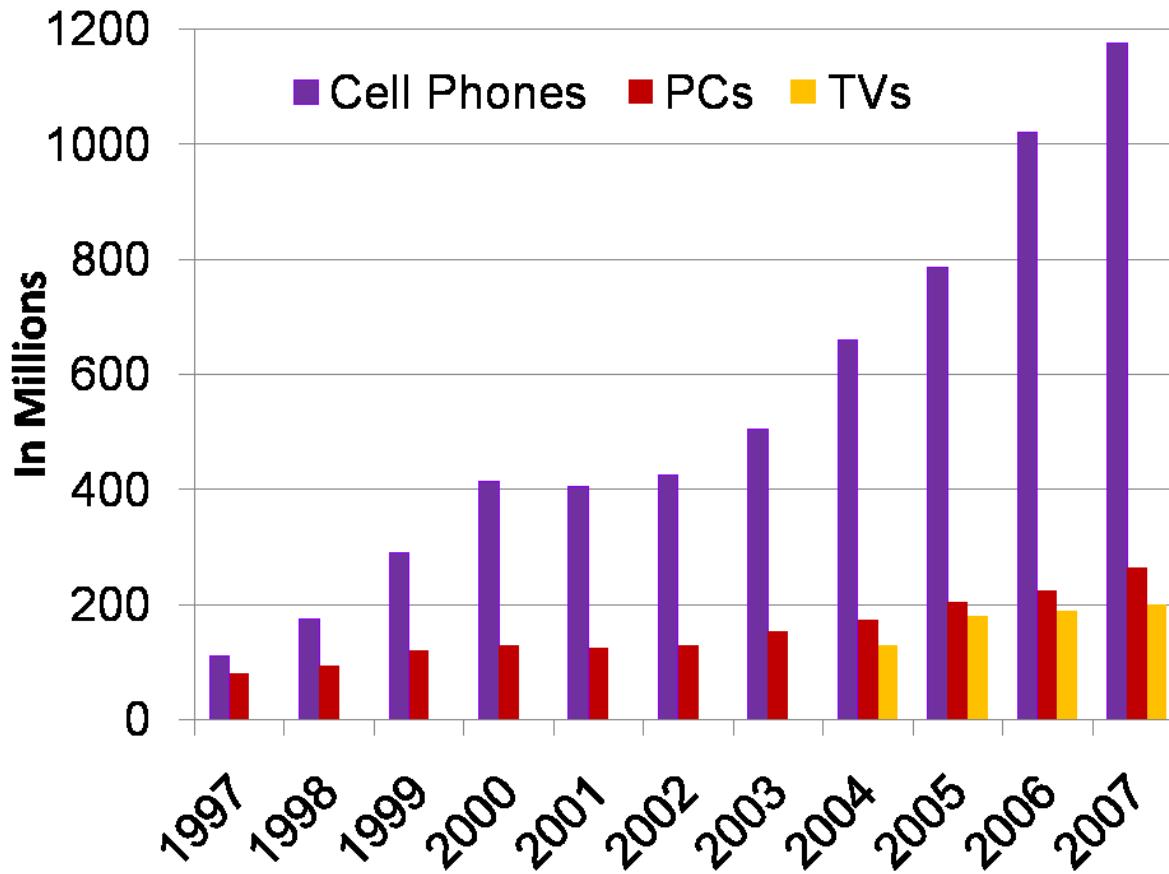
- A computer inside another device used for running one predetermined application

Review: Some Basic Definitions

- ❑ Kilobyte – 2^{10} or 1,024 bytes
- ❑ Megabyte – 2^{20} or 1,048,576 bytes
 - sometimes “rounded” to 10^6 or 1,000,000 bytes
- ❑ Gigabyte – 2^{30} or 1,073,741,824 bytes
 - sometimes rounded to 10^9 or 1,000,000,000 bytes
- ❑ Terabyte – 2^{40} or 1,099,511,627,776 bytes
 - sometimes rounded to 10^{12} or 1,000,000,000,000 bytes
- ❑ Petabyte – 2^{50} or 1024 terabytes
 - sometimes rounded to 10^{15} or 1,000,000,000,000,000 bytes
- ❑ Exabyte – 2^{60} or 1024 petabytes
 - Sometimes rounded to 10^{18} or 1,000,000,000,000,000,000 bytes

Growth in Cell Phone Sales (Embedded)

embedded growth >> desktop growth



- ❑ Where else are embedded processors found?

Embedded Processor Characteristics

The largest class of computers spanning the widest range of applications and performance

- ❑ Often have minimum performance requirements.
Example?
- ❑ Often have stringent limitations on cost. Example?
- ❑ Often have stringent limitations on power consumption.
Example?
- ❑ Often have low tolerance for failure. Example?

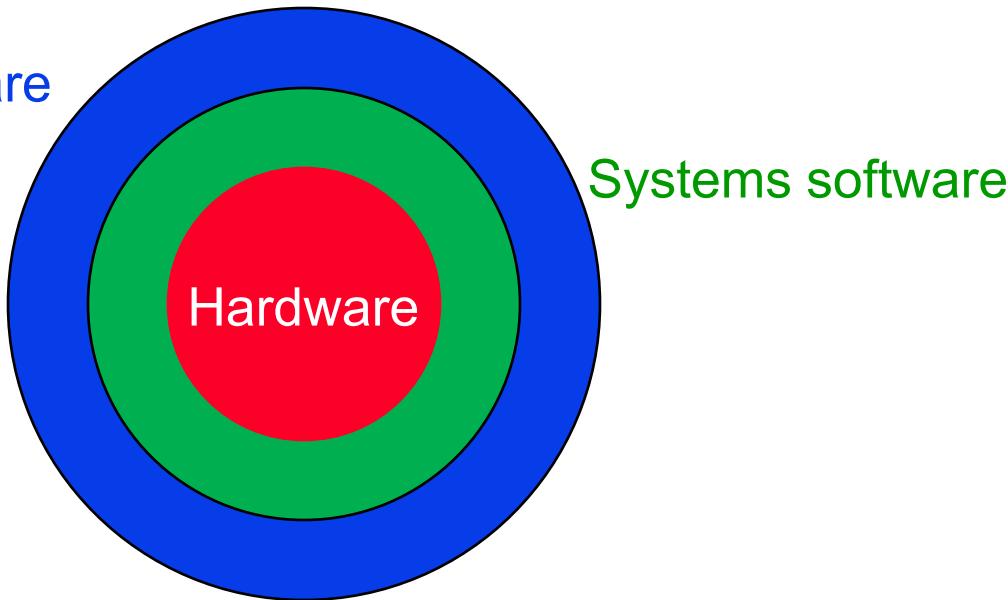
Embedded Processor Characteristics

The largest class of computers spanning the widest range of applications and performance

- ❑ Often have minimum performance requirements.
 - Phones: faster than previous generation.
- ❑ Often have stringent limitations on cost. Example?
 - Price of a PC vs price of a mobile phone.
 - iPhones cost ~US\$200 to manufacture
- ❑ Often have stringent limitations on power consumption.
 - Battery technology doesn't improve quickly.
 - Consumers are aware of battery life when making purchases
- ❑ Often have low tolerance for failure.
 - Mean time to failure measured in years.

Below the Program

Applications software



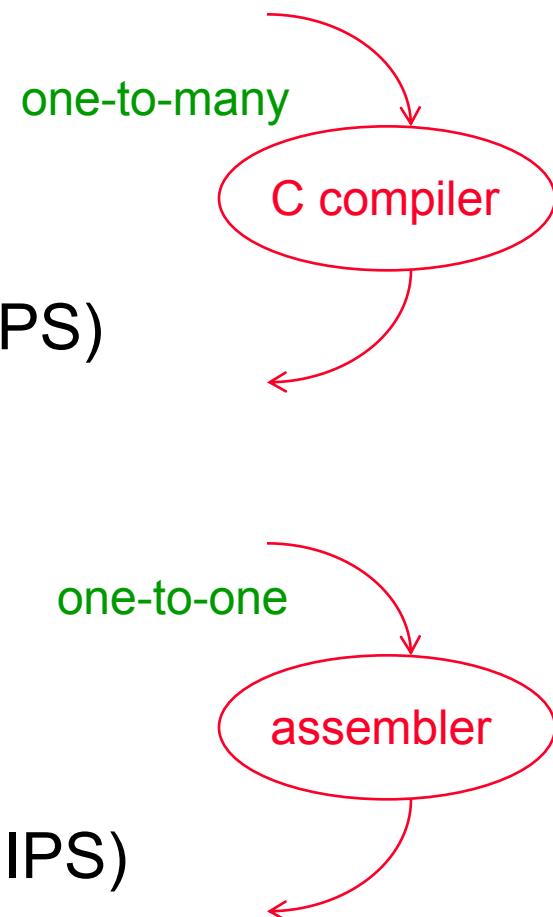
System software

- Operating system – supervising program that interfaces the user's program with the hardware (e.g., Linux, MacOS, Windows)
 - Handles basic input and output operations
 - Allocates storage and memory
 - Provides for protected sharing among multiple applications
- Compiler – translate programs written in a high-level language (e.g., C, Java) into instructions that the hardware can execute

Below the Program, Con't

❑ High-level language program (in C)

```
swap (int v[], int k)
(int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
)
```



❑ Assembly language program (for MIPS)

```
swap: sll    $2, $5, 2
      add    $2, $4, $2
      lw     $15, 0($2)
      lw     $16, 4($2)
      sw     $16, 0($2)
      sw     $15, 4($2)
      jr     $31
```

❑ Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
. . .
```

Advantages of Higher-Level Languages ?

❑ Higher-level languages

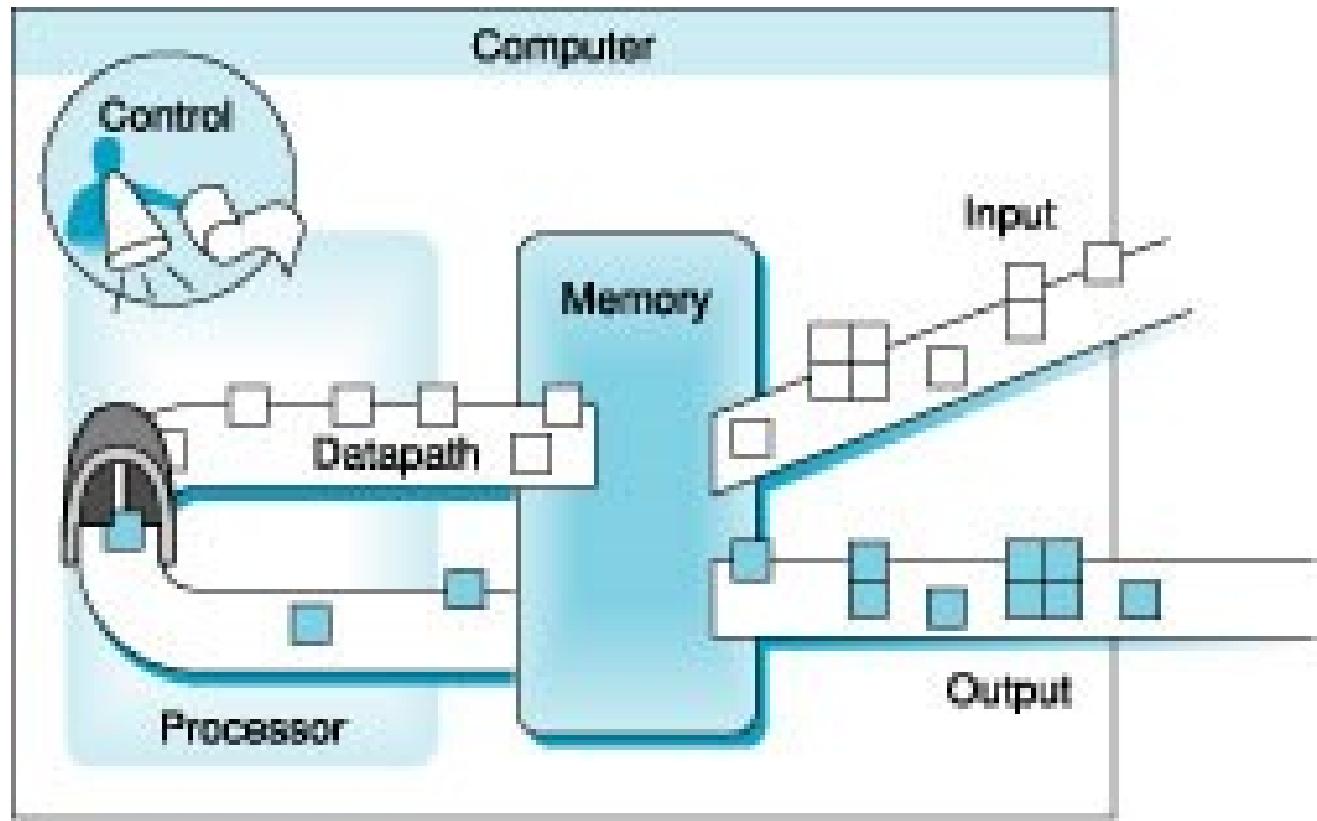
- Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, Java for web programming, ...)
- Improve programmer productivity – more understandable code that is easier to debug and validate
- Improve program maintainability
- Allow programs to be independent of the computer on which they are developed (compilers and assemblers can translate high-level language programs to the binary instructions of any machine)
- Emergence of optimizing compilers that produce **very** efficient assembly code optimized for the target machine

❑ As a result, very little programming is done today at the assembler level

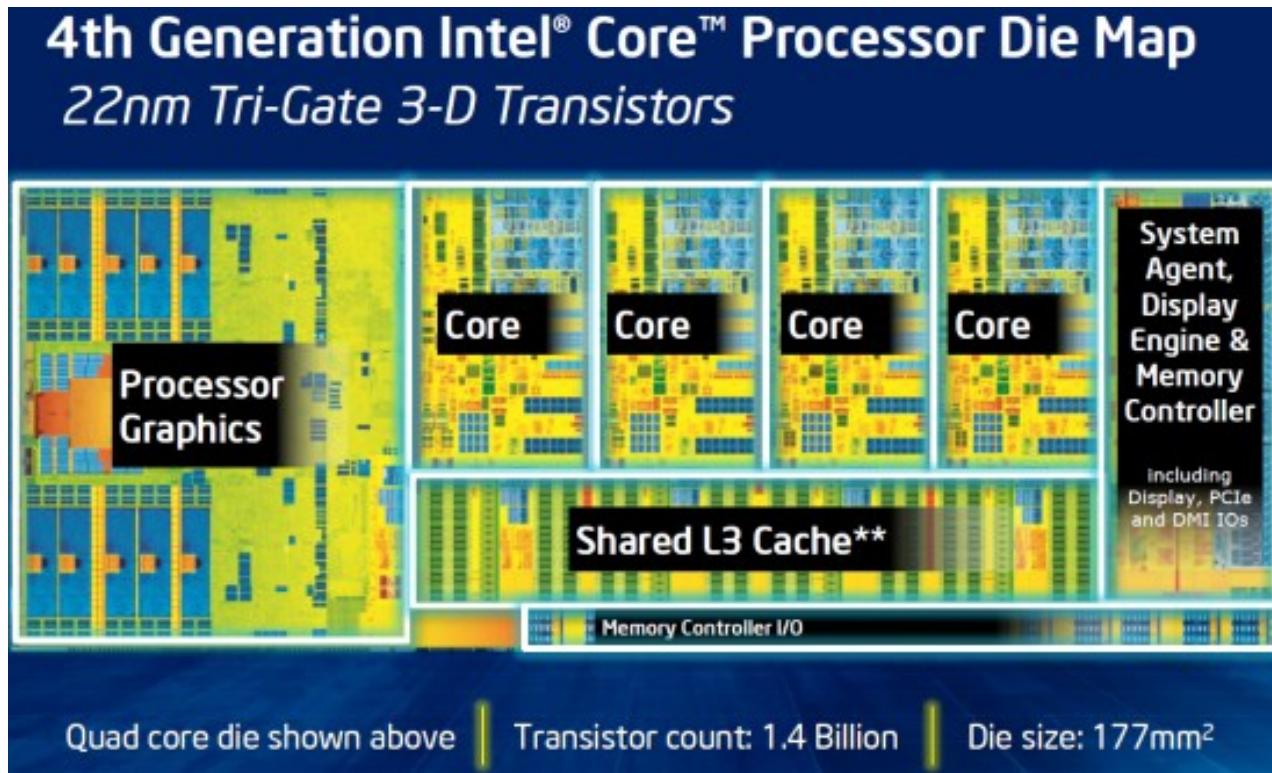
Under the Covers

- ❑ Five classic components of a computer – input, output, memory, datapath, and control

- ❑ datapath + control = processor (CPU)



Intel's i7 Core processor (Haswell)



- ❑ Four out-of-order cores on one chip
- ❑ 3.4 GHz clock rate
- ❑ 22nm technology
- ❑ Three levels of caches (L1, L2, L3) on chip
- ❑ Integrated graphics

Instruction Set Architecture (ISA)

- ❑ ISA, or simply architecture – the abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, I/O, ...
 - Enables **implementations** of varying cost and performance to run identical software
- ❑ The combination of the basic instruction set (the ISA) and the operating system interface is called the application binary interface (ABI)
 - ABI – The user portion of the instruction set plus the operating system interfaces used by application programmers. Defines a standard for binary portability across computers.

Examples of ISAs

❑ Intel x86

- PCs and Laptops
- CISC (Complicated Instruction Set Computing) Architecture
- Emphasis on computing performance at the expense of efficiency.

❑ ARM

- Mostly in embedded systems:
 - Smart Phones, Tablets
- RISC (Reduced Instruction Set Computing) Architecture
- Emphasis on efficiency, at the expense of high end performance.

❑ MIPs

- Older, simpler example.
- Shares many similarities with modern ARM processors.

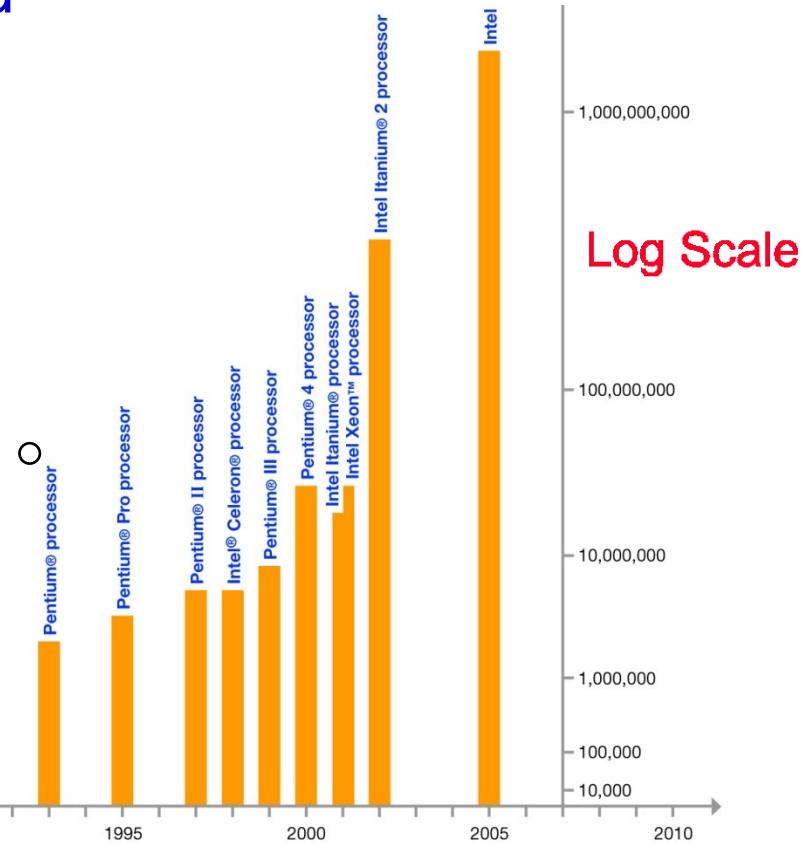
Moore's Law

- In 1965, Intel's Gordon Moore predicted that the number of transistors that can be integrated on single chip would double about every two years



*Note: Vertical scale of chart not proportional to actual Transistor count.

Dual Core
Itanium with
1.7B transistors



Technology Scaling Road Map (ITRS)

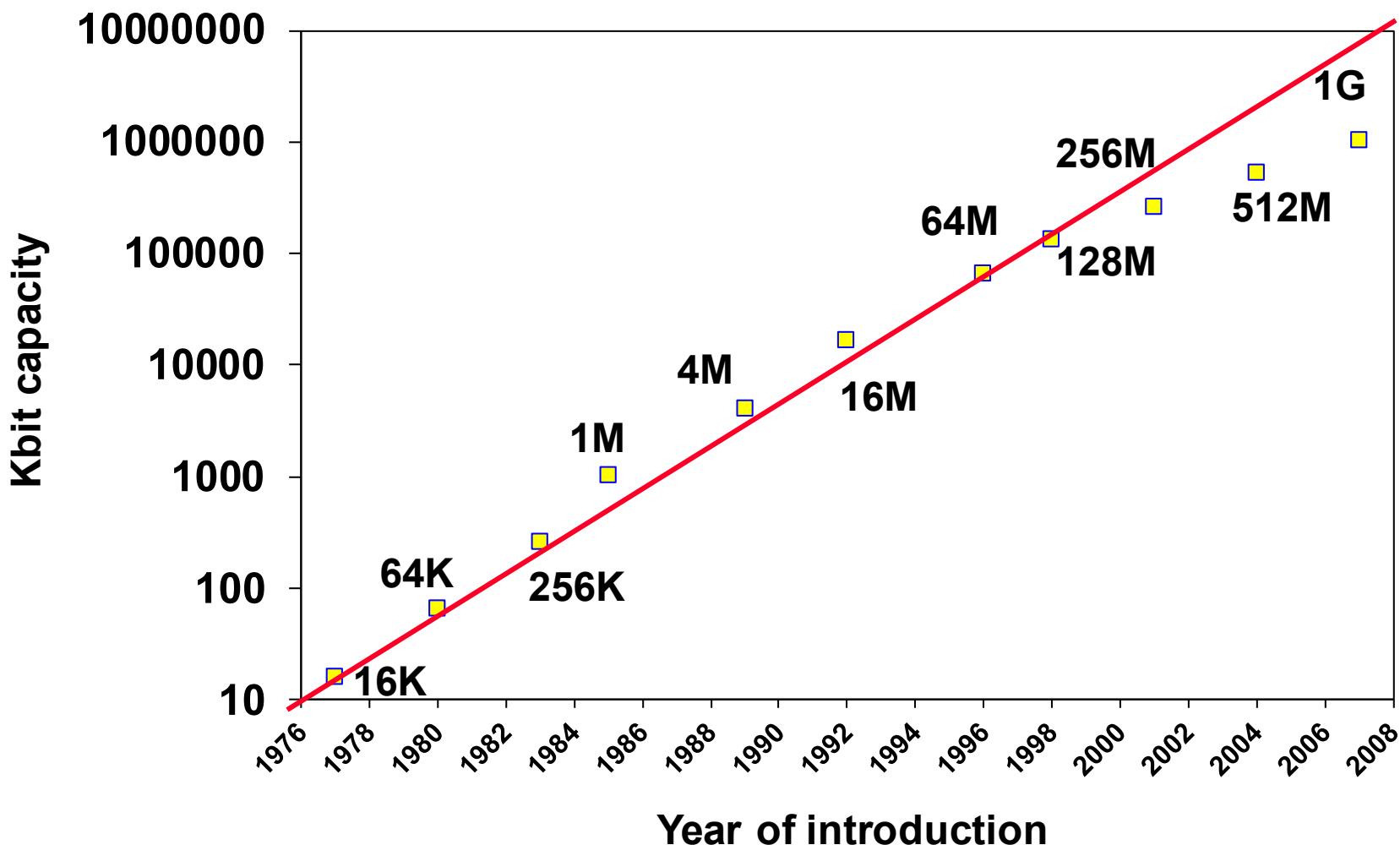
| Year | 2004 | 2006 | 2008 | 2010 | 2012 | 2014 |
|---------------------|------|------|------|------|------|------|
| Feature size (nm) | 90 | 65 | 45 | 32 | 22 | 14 |
| Intg. Capacity (BT) | 2 | 4 | 6 | 16 | 32 | 83 |

❑ Fun facts about 45nm transistors

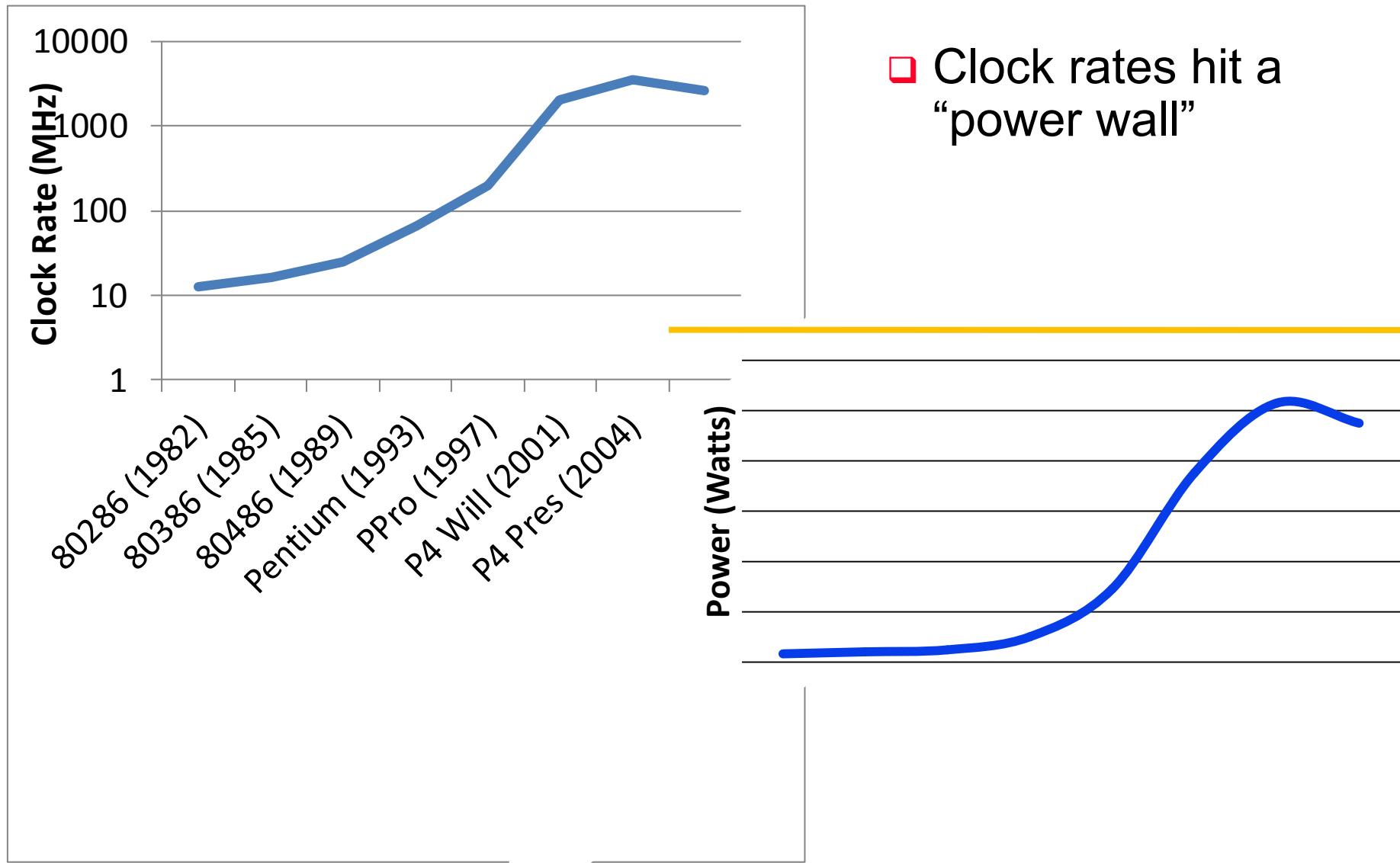
- 30 million can fit on the head of a pin
- You could fit more than 2,000 across the width of a human hair
- If car prices had fallen at the same rate as the price of a single transistor has since 1968, a new car today would cost about 1 cent

Another Example of Moore's Law Impact

DRAM capacity growth over 3 decades



But What Happened to Clock Rates and Why?



Reasons behind the change

❑ Marketing

- Historically, clock speed was an easier performance marker to sell.
- Increasing clock speed was driven by marketing, not optimal design. This is less of a problem now.

❑ Power

- There is only so much power that can be delivered by a main's socket.
- Heat dissipation is limited by cooling technology and size.
- People are green aware/Power cost money.

❑ Multi-Core

- In multi-threaded systems, context switching is a significant performance cost.
- As feature size halves, transistor area quarters. What do you do with all this extra space?

“For the P6, success criteria included performance above a certain level and failure criteria included power dissipation above some threshold.”

Bob Colwell, Pentium Chronicles

A Sea Change is at Hand

- ❑ The power challenge has forced a change in the design of microprocessors
 - Since 2002 the rate of improvement in the response time of programs on desktop computers has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- ❑ As of 2006 all desktop and server companies are shipping microprocessors with multiple processors – cores – per chip

| Product | AMD Barcelona | Intel Nehalem | IBM Power 6 | Sun Niagara 2 | Intel Core i7 |
|----------------|------------------|------------------|----------------|------------------|------------------|
| Cores per chip | 4 | 4 | 2 | 8 | 4 |
| Clock rate | 2.5 GHz | ~2.5 GHz? | 4.7 GHz | 1.4 GHz | 4.0 GHz |
| Power | 120 W | ~100 W? | ~100 W? | 94 W | 85W |

Reminders

❑ Labs start next week.

- Make sure you are enrolled correctly in allocate +.
- Assignment is available now.
- Start working on it now.

Lecture 2: Performance Metrics

Super Computer Example

- ❑ www.top500.org frequently updates (every 6 months) the fastest supercomputers on the planet.
- ❑ Current positions 2 and 3 show something interesting:
 - Spec 1 (Titan):
 - 560,640 cores
 - 2.2 GHz cores
 - 8,209 kW
 - Spec 2 (Sequoia):
 - 1,572,864 cores
 - 1.6 GHz cores
 - 7,890 kW
 - Spec 2 has 3 times the cores of Spec 1
 - Spec 1 cores has ~1.4 times the clock speed of Spec 2
- ❑ Spec 1 outperforms Spec 2

Performance Metrics

❑ Purchasing perspective

- given a collection of machines, which has the
 - best performance ?
 - least cost ?
 - best cost/performance?

❑ Design perspective

- faced with design options, which has the
 - best performance improvement ?
 - least cost ?
 - best cost/performance?

❑ Both require

- basis for comparison
- metric for evaluation

❑ Our goal is to understand what factors in the architecture contribute to overall system performance and the relative importance (and cost) of these factors

Throughput versus Response Time

- ❑ Response time (execution time) – the time between the start and the completion of a task
 - Important to individual users
- ❑ Throughput (bandwidth) – the total amount of work done in a given time
 - Important to data center managers
- ❑ Will need different performance metrics as well as a different set of applications to benchmark **embedded** and **desktop** computers, which are more focused on response time, versus **servers**, which are more focused on throughput

Contributors to Slow Response time

- ❑ Algorithmic changes can make large changes to response times.
- ❑ Software profilers can reveal when this is the case.
 - Software profilers take run time measurements of functions within your code to determine where a program spends its time.
- ❑ Example:
 - Thinking about how you call a function when compiling code to machine language.
 - When calling a function, there is significant overhead just to jump and set up the functions memory space.
 - If all your function does is add two numbers then the overhead is much larger than the actual algorithm.

Contributors to Slow Response time

- ❑ Example continued:

```
#define SQUARE_MACRO(X) ((X)*(X))
```

```
inline int square(int A){
```

```
    return A*A;
```

```
}
```

```
int main(){
```

```
    ...
```

```
    for(int i=0;i<100000000;i=i+1){
```

```
        ... square(i) ....;
```

```
}
```

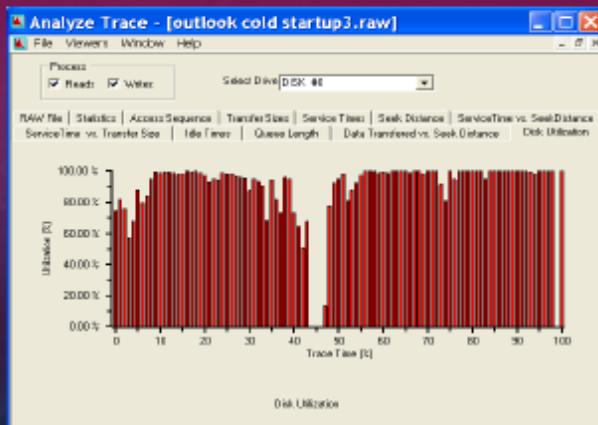
```
    return 0;
```

```
}
```

Response Time Matters

It's the Hard Disk, Stupid!

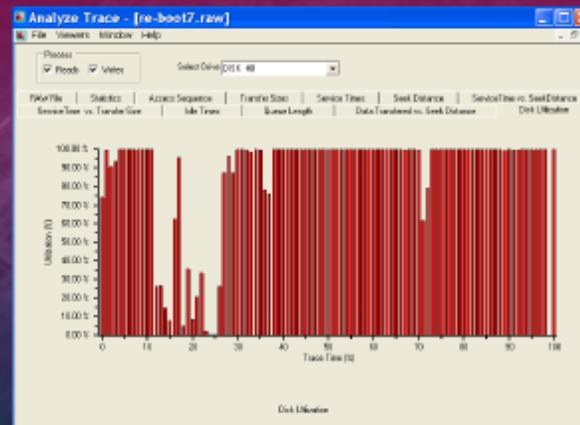
*Re-Boot/Startup
on Home PC*



Elapsed Time 105.213536, s
Disk Busy Time 91.368480, s
Average Data Rate 6.60669, MB/s

86% BUSY

Starting Outlook



Elapsed Time 45.700667, s
Disk Busy Time 41.056997, s
Average Data Rate 1.37389, MB/s

89% BUSY



Intel Requirements for Ultra book

- ❑ Specific Microprocessor Architecture:
 - 17W or lower power Intel processors
- ❑ Maximum height requirements (up to 23mm)
 - Translates to limit on cooling, power and battery size.
- ❑ Minimum battery life:
 - 5 hours (undefined operation) for early ultra books
 - 6 hours HD video playback (post June 2013)
 - 9 hours Windows 8 idle (post June 2013)
- ❑ Minimum response time:
 - Maximum 7 seconds resume from idle (pre June 2013)
 - Maximum 3 seconds resume from idle (post June 2013)

Defining (Speed) Performance

- ❑ To maximize performance, need to **minimize** execution time

$$\text{performance}_X = 1 / \text{execution_time}_X$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution_time}_Y}{\text{execution_time}_X} = n$$

- ❑ Decreasing response time almost always improves throughput

Relative Performance Example

- ❑ If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is n times faster than B if

$$\frac{\text{performance}_A}{\text{performance}_B} = \frac{\text{execution_time}_B}{\text{execution_time}_A} = n$$

The performance ratio is

$$\frac{15}{10} = 1.5$$

So A is 1.5 times faster than B

Performance Factors

- ❑ CPU execution time (CPU time) – time the CPU spends working on a task
 - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time} = \frac{\# \text{ CPU clock cycles for a program}}{\text{clock cycle time for a program}}$$

or

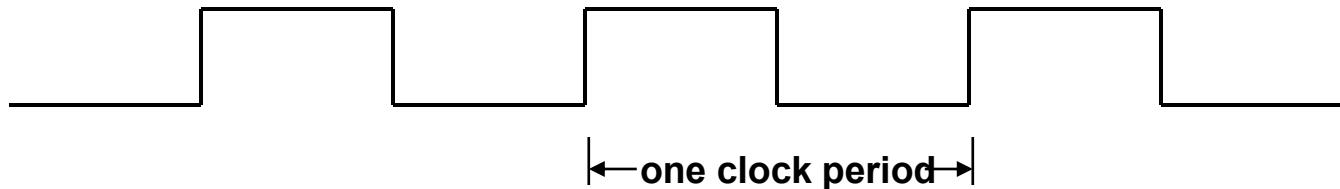
$$\text{CPU execution time} = \frac{\# \text{ CPU clock cycles for a program}}{\text{clock rate}}$$

- ❑ Can improve performance by reducing either the **length of the clock cycle** or the **number of clock cycles required for a program**

Review: Machine Clock Rate

- ❑ Clock rate (clock cycles per second in MHz or GHz) is inverse of clock cycle time (clock period)

$$CC = 1 / CR$$



Clock rates are dictated by logic delays.

Setup-Up Time: Minimum time a memory element needs the data before the clock edge. ~Maximum logic delay.

Hold Time: Minimum time data must remain after the clock edge.
~Minimum logic delay.

More logic between memory elements \approx More work per clock cycle.

More logic between memory elements = Longer clock cycles.

Improving Performance Example

- ❑ A program runs on computer A with a 2 GHz clock in 10 seconds. What clock rate must computer B run at to run this program in 6 seconds? Unfortunately, to accomplish this, computer B will require 1.2 times as many clock cycles as computer A to run the program.

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{clock rate}_A}$$

$$\begin{aligned}\text{CPU clock cycles}_A &= 10 \text{ sec} \times 2 \times 10^9 \text{ cycles/sec} \\ &= 20 \times 10^9 \text{ cycles}\end{aligned}$$

$$\text{CPU time}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{clock rate}_B}$$

$$\text{clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = 4 \text{ GHz}$$

Clock Cycles per Instruction

- ❑ Not all instructions take the same amount of time to execute
 - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction
- # CPU clock cycles = # Instructions Average clock cycles
for a program = for a program × per instruction
- ❑ Clock cycles per instruction (CPI) – the average number of clock cycles each instruction takes to execute
 - A way to compare two different implementations of the same ISA

| | CPI for this instruction class | | |
|-----|--------------------------------|---|---|
| | A | B | C |
| CPI | 1 | 2 | 3 |

Using the Performance Equation

- ❑ Computers A and B implement the same ISA. Computer A has a clock cycle time of 250 ps and an effective CPI of 2.0 for some program and computer B has a clock cycle time of 500 ps and an effective CPI of 1.2 for the same program. Which computer is faster and by how much?

Each computer executes the same number of instructions, I , so

$$\text{CPU time}_A = I \times 2.0 \times 250 \text{ ps} = 500 \times I \text{ ps}$$

$$\text{CPU time}_B = I \times 1.2 \times 500 \text{ ps} = 600 \times I \text{ ps}$$

Clearly, A is faster ... by the ratio of execution times

$$\frac{\text{performance}_A}{\text{performance}_B} = \frac{\text{execution_time}_B}{\text{execution_time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

Effective (Average) CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

$$\text{Overall effective CPI} = \sum_{i=1}^n (\text{CPI}_i \times IC_i)$$

- Where IC_i is the count (percentage) of the number of instructions of class i executed
- CPI_i is the (average) number of clock cycles per instruction for that instruction class
- n is the number of instruction classes

- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

THE Performance Equation

- ❑ Our basic performance equation is then

$$\text{CPU time} = \text{Instruction_count} \times \text{CPI} \times \text{clock_cycle}$$

or

$$\text{CPU time} = \frac{\text{Instruction_count} \times \text{CPI}}{\text{clock_rate}}$$

- ❑ These equations separate the **three key** factors that affect performance
 - Can measure the CPU execution time by running the program
 - The clock rate is usually given
 - Can measure overall instruction count by using profilers/ simulators without knowing all of the implementation details
 - CPI varies by instruction type and ISA implementation for which we must know the implementation details

Determinates of CPU Performance

$$\text{CPU time} = \text{Instruction_count} \times \text{CPI} \times \text{clock_cycle}$$

| | Instruction_count | CPI | clock_cycle |
|----------------------|-------------------|-----|-------------|
| Algorithm | X | X | |
| Programming language | X | X | |
| Compiler | X | X | |
| ISA | X | X | X |
| Core organization | | X | X |
| Technology | | | X |

A Simple Example

| Op | Freq | CPI _i | Freq x CPI _i | | | |
|----------------|------|------------------|-------------------------|-----|-----|------|
| ALU | 50% | 1 | .5 | .5 | .5 | .25 |
| Load | 20% | 5 | 1.0 | .4 | 1.0 | 1.0 |
| Store | 10% | 3 | .3 | .3 | .3 | .3 |
| Branch | 20% | 2 | .4 | .4 | .2 | .4 |
| $\Sigma = 2.2$ | | | | 1.6 | 2.0 | 1.95 |

- ❑ How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
CPU time new = $1.6 \times IC \times CC$ so $2.2/1.6$ means 37.5% faster
- ❑ How does this compare with using branch prediction to shave a cycle off the branch time?
CPU time new = $2.0 \times IC \times CC$ so $2.2/2.0$ means 10% faster
- ❑ What if two ALU instructions could be executed at once?
CPU time new = $1.95 \times IC \times CC$ so $2.2/1.95$ means 12.8% faster

Workloads and Benchmarks

- ❑ Benchmarks – a set of programs that form a “workload” specifically chosen to measure performance
- ❑ SPEC (System Performance Evaluation Cooperative) creates standard sets of benchmarks starting with SPEC89. The latest is SPEC CPU2006 which consists of 12 integer benchmarks (CINT2006) and 17 floating-point benchmarks (CFP2006).

www.spec.org

- ❑ There are also benchmark collections for power workloads (SPECpower_ssj2008), for mail workloads (SPECmail2008), for multimedia workloads (mediabench), ...

SPEC CINT2006 on i7-4790K (Freq = 4.4 GHz)

| Name | ICx10 ⁹ | CPI | ExTime | RefTime | SPEC ratio |
|----------------|--------------------|------|--------|---------|-------------|
| perl | 2,118 | 0.44 | 214 | 9,770 | 45.6 |
| bzip2 | 2,389 | 0.55 | 297 | 9,650 | 32.5 |
| gcc | 1,050 | 0.87 | 208 | 8,050 | 38.8 |
| mcf | 336 | 1.36 | 104 | 9,120 | 87.9 |
| go | 1,658 | 0.79 | 297 | 10,490 | 35.3 |
| hmmer | 2,783 | 0.17 | 107 | 9,330 | 87 |
| sjeng | 2,176 | 0.61 | 303 | 12,100 | 40 |
| libquantum | 1,623 | 0.04 | 13.5 | 20,720 | 1530 |
| h264avc | 3,102 | 0.41 | 291 | 22,130 | 76.2 |
| omnetpp | 587 | 1.62 | 216 | 6,250 | 28.9 |
| astar | 1,082 | 0.73 | 180 | 7,020 | 39.1 |
| xalancbmk | 1,058 | 0.40 | 95.5 | 6,900 | 72.3 |
| Geometric Mean | | | | | 65.2 |

Selection of Benchmarks

- ❑ Programs are often chosen as benchmarks due to the popularity of the program or similar algorithms.
- ❑ For benchmarks to be useful, they should be chosen to represent tasks that people are interested in gauging a system's performance.
- ❑ For very large systems (supercomputers), benchmarks may need to scale in size to provide some meaningful result.
- ❑ Example:
 - LINPACK is a standard benchmark for running on supercomputers. Essentially matrix inversion problem.
 - A computer with 1,000,000 cores isn't going to be utilized unless the problem is big enough.

Comparing and Summarizing Performance

- ❑ How do we summarize the performance for benchmark set with a **single** number?
 - First the execution times are normalized giving the “SPEC ratio” (bigger is faster, i.e., SPEC ratio is the inverse of execution time)
 - The SPEC ratios are then “averaged” using the **geometric mean** (GM)

$$GM = \sqrt[n]{\prod_{i=1}^n \text{SPEC ratio}_i}$$

- ❑ Guiding principle in reporting performance measurements is **reproducibility** – list everything another experimenter would need to duplicate the experiment (version of the operating system, compiler settings, input set used, specific computer configuration (clock rate, cache sizes and speed, memory size and speed, etc.))

Why Geometric Mean?

- ❑ It can be shown that the geometric mean is the only correct mean for results presented as ratios to a reference.

| | Computer A | Computer B | Computer C |
|-----------------|------------|------------|------------|
| Program 1 | 1 | 10 | 20 |
| Program 2 | 1000 | 100 | 20 |
| Arithmetic mean | 500.5 | 55 | 20 |
| Geometric mean | 31.622... | 31.622... | 20 |

Why Geometric Mean?

- ❑ All computers run “Program 2” 1000 times faster:

| | Computer A | Computer B | Computer C |
|-----------------|------------|------------|------------|
| Program 1 | 1 | 10 | 20 |
| Program 2 | 1 | 0.1 | 0.02 |
| Arithmetic mean | 1 | 5.05 | 10.01 |
| Geometric mean | 1 | 1 | 0.632... |

- ❑ All computers run “Program 2” 100 times faster and “Program 1” ten times faster:

| | Computer A | Computer B | Computer C |
|-----------------|------------|------------|--------------|
| Program 1 | 0.1 | 1 | 2 |
| Program 2 | 10 | 1 | 0.2 |
| Arithmetic mean | 5.05 | 1 | 1.1 |
| Geometric mean | 1 | 1 | 0.632 |

Important Considerations when benchmarking

- ❑ When comparing processors with different ISAs the compiler can be very important.
 - Does the compiler optimize “fairly”?
 - Does the compiler use advanced instructions?
 - Eg MMX instructions
- ❑ Are other programs affecting the results?
- ❑ What is the CPU temperature?
 - Many modern CPUs “throttle” the processing power as temperature increases.
- ❑ How does the operating system power management affect the results?
 - Features like “Power on demand” are operating system dependent.

Important Considerations when benchmarking

- ❑ Example 1:

- ❑ Anecdote:

- Running a Benchmark program (Dhrystone MIPS) from source code.
- The intention at the time was to port the benchmark to a processor I was designing.
- Ran the benchmark on my PC after compiling with gcc.
 - The results were surprisingly low.
- Ran the benchmark again using Intel compiler on the same PC
 - Result were around 2-3 times faster.

Important Considerations when benchmarking

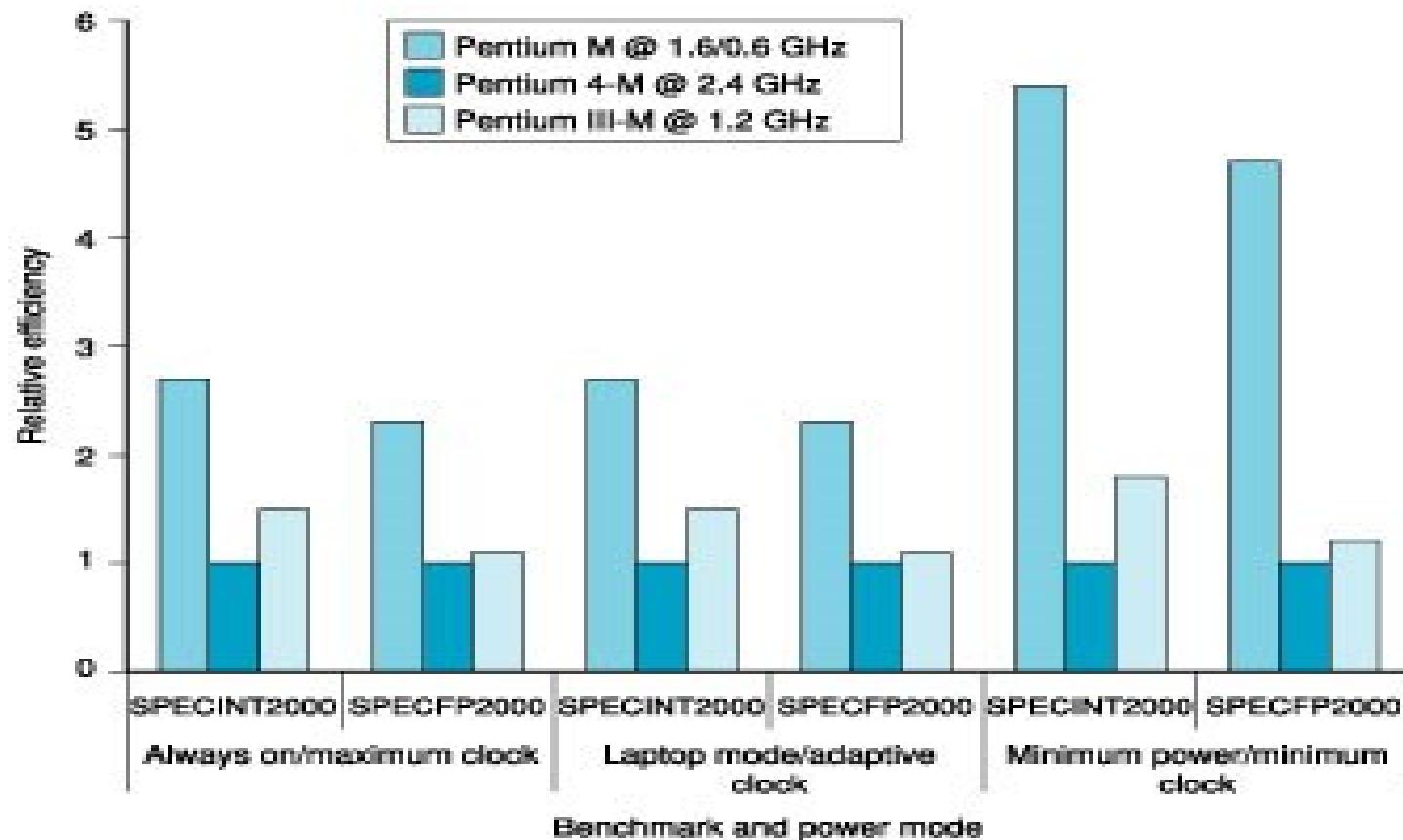
- ❑ Example 2:
- ❑ Another Anecdote (not mine):
 - A previous lecturer set an assignment for students to examine the performance differences for different matrix multiplication algorithms.
 - The student had remote access to a Cray supercomputer.
 - As the students worked they discovered that all algorithms gave the same performance.
 - They had been using a the Cray compiler for their code, which by default, would recognize their code as a matrix multiplication.
 - The compiler would then substitute the optimal machine code to perform the multiplication, regardless of the algorithm they wanted.
 - To get a fair comparison a particular compiler optimization flag had to be removed.

Important Considerations when benchmarking

- ❑ Example 3:
- ❑ Compiler Flags:
 - The difference between compiling in debug mode Vs “-O3” can be very large. >2 performance in some cases.

Other Performance Metrics

- ❑ Power consumption – especially in the embedded market where battery life is important
 - For power-limited applications, the most important metric is energy efficiency



Other Performance Metrics

- ❑ Top500 allows you to view top 500 super computers by Power efficiency (Mflops/Watt)
 - Currently range from 128 Mflops/Watt to 3,459 Mflops/Watt
 - For reference i7-4790K (22nm) is ~1,000 Mflops/Watt
 - ARM cortex-A15 (32nm) reports ~1.5-2 Mflops/watt
- ❑ Digital power on microchips have two contributors
 - Static Power: The power required by the logic component when not switching.
 - Dynamic Power: The power required by the logic to switch at a particular rate.
- ❑ Static Power is caused by transistor leakage and relatively increases as transistors get smaller.
- ❑ Dynamic Power is caused by transistors driving RC loads and decreases as transistors get smaller.

Other Performance Metrics

- ❑ Notes on Power usage and feature size:
 - As feature (transistor) sizes get smaller, digital switching power decreases.
 - As feature sizes gets smaller, leakage current increases.
 - Leakage (in transistors) is caused by the reverse bias diode that are necessary for form a transistor.
 - For very small transistor sizes, quantum tunnelling can be a significant contributor of leakage current.
- ❑ Some embedded system turn off (or lower the supply voltage) of areas of the microchip that are not being utilized.
 - This greatly saves on leakage current.

Summary: Evaluating ISAs

❑ Design-time metrics:

- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

❑ Static Metrics:

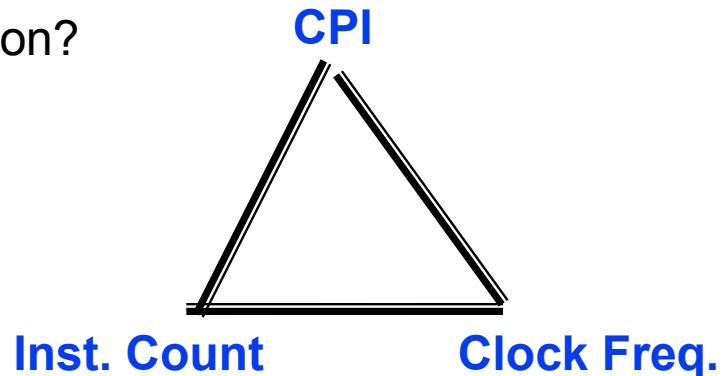
- How many bytes does the program occupy in memory?

❑ Dynamic Metrics:

- How many instructions are executed? How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

Best Metric: Time to execute the program!

depends on the instructions set, the processor organization, and compilation techniques.



Reminders

❑ Labs start next week

- Labs are for working on your assignment, no other lab material
- Read the assignment and get started on it now.

ECE4074

Computer Architecture

Semester 2 2014

Chapter 2: Instructions: Language of the Computer

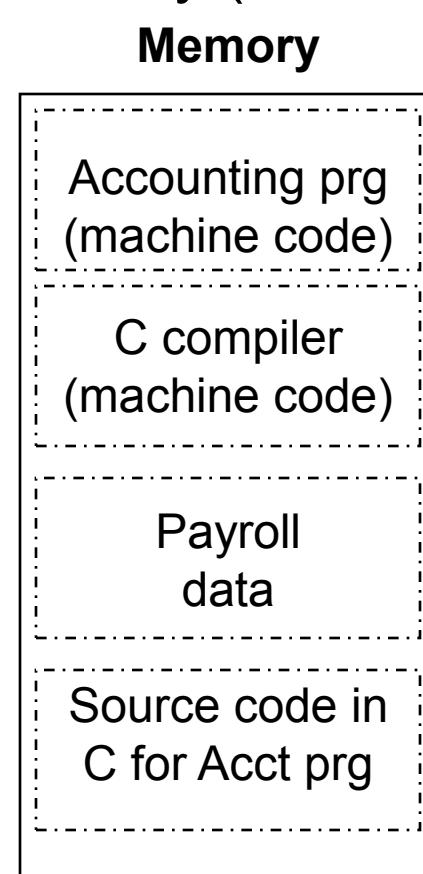
[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data

Stored-program concept

- Programs can be shipped as files of binary numbers – **binary compatibility**
- Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs



MIPS-32 ISA

❑ Instruction Categories

- Computational
- Load/Store
- Jump and Branch
- Floating Point
 - coprocessor
- Memory Management
- Special

Registers

R0 - R31

PC

HI

LO

3 Instruction Formats: **all 32 bits wide**

| 31 | op | rs | rt | rd | sa | 0 | R format |
|----|----|----|----|-------------|-----------|---|----------|
| | op | rs | rt | | immediate | | I format |
| | op | | | jump target | | | J format |

MIPS (RISC) Design Principles

❑ Simplicity favors regularity

- fixed size instructions
- small number of instruction formats
- opcode always the first 6 bits

❑ Smaller is faster

- limited instruction set
- limited number of registers in register file
- limited number of addressing modes

❑ Make the common case fast

- arithmetic operands from the register file (load-store machine)
- allow instructions to contain immediate operands

❑ Good design demands good compromises

- three instruction formats

MIPS Arithmetic Instructions

- ❑ MIPS assembly language arithmetic statement

add \$t0, \$s1, \$s2



- ❑ Each arithmetic instruction performs **one** operation
- ❑ Each specifies exactly **three** operands that are all contained in the datapath's register file (\$t0, \$s1, \$s2)
- ❑ Instruction Format (**R** format)

| | | | | | |
|-------|-------|-------|------|------|-------|
| 6'h00 | 5'd17 | 5'd18 | 5'd8 | 5'd0 | 6'h22 |
|-------|-------|-------|------|------|-------|

MIPS Instruction Fields

- ❑ MIPS fields are given names to make them easier to refer to

| | | | | | | | | | | | |
|----|----|----|----|----|----|----|-------|----|-------|---|---|
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
| op | rs | | rt | | rd | | shamt | | funct | | |

- op 6-bits **opcode** that specifies the operation
- rs 5-bits **register file address** of the first **source operand**
- rt 5-bits **register file address** of the second source operand
- rd 5-bits **register file address** of the result's **destination**
- shamt 5-bits **shift amount** (for shift instructions)
- funct 6-bits **function code** augmenting the opcode

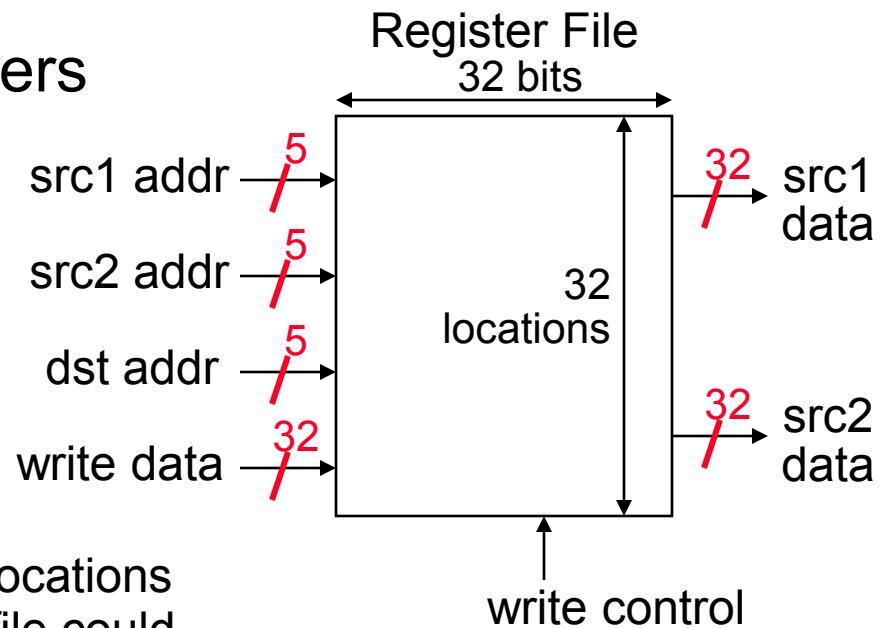
MIPS Register File

- ❑ Holds thirty-two 32-bit registers

- Two read ports and
 - One write port

- ❑ Registers are

- Faster than main memory
 - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
 - Read/write port increase impacts speed quadratically
 - Easier for a compiler to use
 - e.g., $(A*B) - (C*D) - (E*F)$ can do multiplies in any order vs. stack
 - Can hold variables so that
 - code density improves (since register are named with fewer bits than a memory location)



Review: MIPS Register Convention

| Name | Register Number | Usage | Preserve on call? |
|-------------|-----------------|---------------------------------|-------------------|
| \$zero | 0 | constant 0 (hardware) | n.a. |
| \$at | 1 | reserved for assembler | n.a. |
| \$v0 - \$v1 | 2-3 | returned values | no |
| \$a0 - \$a3 | 4-7 | arguments | yes |
| \$t0 - \$t7 | 8-15 | temporaries | no |
| \$s0 - \$s7 | 16-23 | saved values | yes |
| \$t8 - \$t9 | 24-25 | temporaries | no |
| \$gp | 28 | global pointer | yes |
| \$sp | 29 | stack pointer | yes |
| \$fp | 30 | frame pointer | yes |
| \$ra | 31 | return addr (hardware) | yes |

MIPS Memory Access Instructions

- ❑ MIPS has two basic **data transfer** instructions for accessing memory

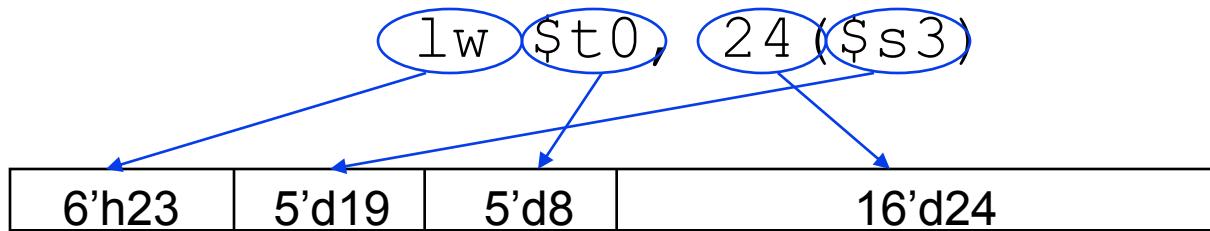
```
lw    $t0, 4($s3)    #load word from memory
```

```
sw    $t0, 8($s3)    #store word to memory
```

- ❑ The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address
- ❑ The memory address – a 32 bit address – is formed by adding the contents of the **base address register** to the **offset value**
 - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register

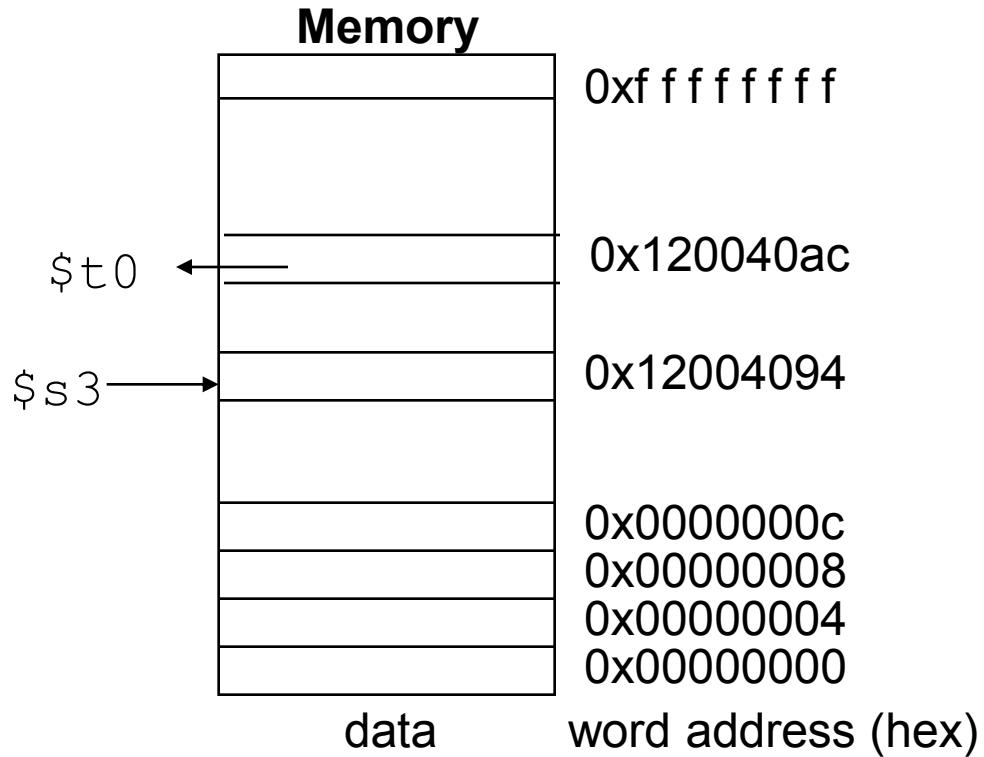
Machine Language - Load Instruction

- Load/Store Instruction Format (I format):



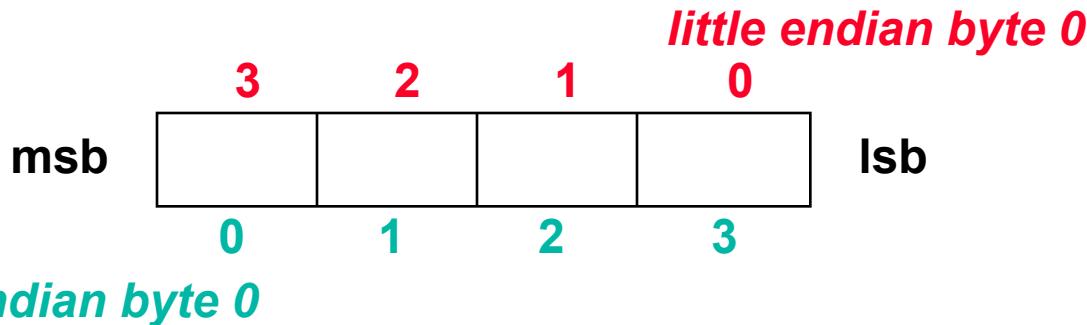
$$16' d24 + \$s3 =$$

$$\begin{array}{r} \dots 0001\ 1000 \\ + \underline{\dots 1001\ 0100} \\ \hline \dots 1010\ 1100 = \\ 0x120040ac \end{array}$$



Byte Addresses

- ❑ Since 8-bit bytes are so useful, most architectures address individual **bytes** in memory
 - **Alignment restriction** - the memory address of a **word** must be on natural word boundaries (a multiple of 4 in MIPS-32)
- ❑ **BigEndian:** leftmost byte is word address
IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- ❑ **LittleEndian:** rightmost byte is word address
Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Aside: Loading and Storing Bytes

- ❑ MIPS provides special instructions to move bytes

lb \$t0, 1(\$s3) #load byte from memory

sb \$t0, 6(\$s3) #store byte to memory

| | | | |
|-------|-------|------|---------------|
| 6'h28 | 5'd19 | 5'd8 | 16 bit offset |
|-------|-------|------|---------------|

- ❑ What 8 bits get loaded and stored?

- load byte places the byte from memory in the rightmost 8 bits of the destination register
 - what happens to the other bits in the register?
- store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
 - what happens to the other bits in the memory word?

MIPS Immediate Instructions

- ❑ Small constants are used often in typical code
- ❑ Possible approaches?
 - put “typical constants” in memory and load them
 - create hard-wired registers (like \$zero) for constants like 1
 - have special instructions that contain constants !

addi \$sp, \$sp, 4 # \$sp = \$sp + 4

slti \$t0, \$s2, 15 # \$t0 = 1 if \$s2 < 15

- ❑ Machine format (**I** format):

| | | | |
|-------|-------|------|--------|
| 6'h0A | 5'd18 | 5'd8 | 16'h0F |
|-------|-------|------|--------|

- ❑ The constant is kept **inside** the instruction itself!
 - Immediate format **limits** values to the range $+2^{15}-1$ to -2^{15}

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

| | | | |
|-------|------|------|----------------------|
| 6'h10 | 5'dX | 5'd8 | 16'b1010101010101010 |
|-------|------|------|----------------------|

- Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

| | |
|----------------------|----------------------|
| 16'b1010101010101010 | 16'b0000000000000000 |
|----------------------|----------------------|

| | |
|----------------------|----------------------|
| 16'b0000000000000000 | 16'b1010101010101010 |
|----------------------|----------------------|

| | |
|----------------------|----------------------|
| 16'b1010101010101010 | 16'b1010101010101010 |
|----------------------|----------------------|

MIPS Shift Operations

- ❑ Need operations to **pack** and **unpack** 8-bit characters into 32-bit words
- ❑ Shifts move all the bits in a word left or right

sll \$t2, \$s0, 8 #\$t2 = \$s0 << 8 bits

srl \$t2, \$s0, 8 #\$t2 = \$s0 >> 8 bits

- ❑ Instruction Format (**R** format)

| | | | | | |
|------|------|-------|-------|------|-------|
| 6'h0 | 5'dX | 5'd16 | 5'd10 | 5'd8 | 6'h00 |
|------|------|-------|-------|------|-------|

- ❑ Such shifts are called **logical** because they fill with **zeros**
 - Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or **31 bit positions**

MIPS Logical Operations

- There are a number of **bit-wise** logical operations in the MIPS ISA

and \$t0, \$t1, \$t2 # \$t0 = \$t1 & \$t2

or \$t0, \$t1, \$t2 # \$t0 = \$t1 | \$t2

nor \$t0, \$t1, \$t2 # \$t0 = not(\$t1 | \$t2)

- Instruction Format (**R** format)

| | | | | | |
|------|------|-------|------|------|-------|
| 6'h0 | 5'd9 | 5'd10 | 5'd8 | 5'd0 | 6'h24 |
|------|------|-------|------|------|-------|

andi \$t0, \$t1, 0xFF00 # \$t0 = \$t1 & ff00

ori \$t0, \$t1, 0xFF00 # \$t0 = \$t1 | ff00

- Instruction Format (**I** format)

| | | | |
|-------|------|------|----------|
| 6'h0D | 5'd9 | 5'd8 | 16'hFF00 |
|-------|------|------|----------|

MIPS Control Flow Instructions

❑ MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1  
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

- Ex: if (i==j) h = i + j;

```
bne $s0, $s1, Lbl1  
add $s3, $s0, $s1  
Lbl1:        ...
```

❑ Instruction Format (I format):

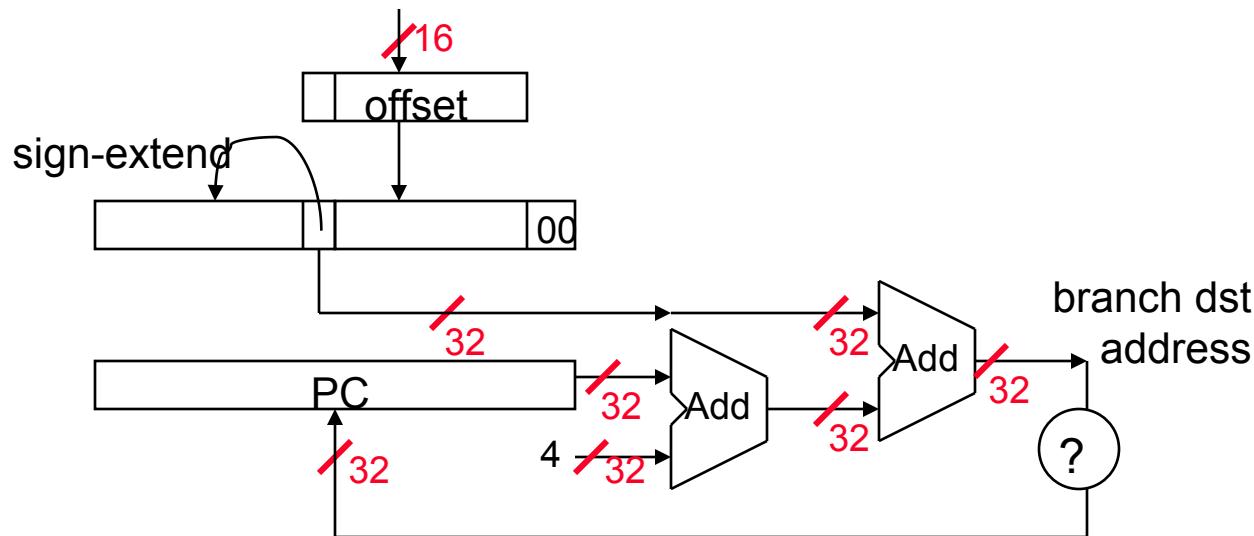
| | | | |
|-------|-------|-------|---------------|
| 6'h05 | 5'd16 | 5'd17 | 16 bit offset |
|-------|-------|-------|---------------|

❑ How is the branch destination address specified?

Specifying Branch Destinations

- ❑ Use a register (like in lw and sw) added to the 16-bit offset
 - which register? Instruction Address Register (the PC)
 - its use is automatically **implied** by instruction
 - PC gets updated (PC+4) during the **fetch** cycle so that it holds the address of the next instruction
 - limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway

from the low order 16 bits of the branch instruction



In Support of Branch Instructions

- ❑ We have beq, bne, but what about other kinds of branches (e.g., branch-if-less-than)? For this, we need yet another instruction, slt
- ❑ Set on less than instruction:

```
slt $t0, $s0, $s1      # if $s0 < $s1      then  
                      # $t0 = 1            else  
                      # $t0 = 0
```

- ❑ Instruction format (**R** format):

| | | | | | |
|------|-------|-------|------|------|-------|
| 6'h0 | 5'd16 | 5'd17 | 5'd8 | 5'd0 | 6'h24 |
|------|-------|-------|------|------|-------|

- ❑ Alternate versions of slt

```
slti $t0, $s0, 25      # if $s0 < 25 then $t0=1 ...  
sltu $t0, $s0, $s1     # if $s0 < $s1 then $t0=1 ...  
sltiu $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

- Can use slt, beq, bne, and the fixed value of 0 in register \$zero to **create** other conditions

- less than

blt \$s1, \$s2, Label

slt \$at, \$s1, \$s2 #\$at set to 1 if

bne \$at, \$zero, Label #\$s1 < \$s2

- less than or equal to

ble \$s1, \$s2, Label

- greater than

bgt \$s1, \$s2, Label

- great than or equal to

bge \$s1, \$s2, Label

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
 - Its why the assembler needs a reserved register (\$at)

Bounds Check Shortcut

- ☐ Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
sltu $t0, $s1, $t2      # $t0 = 0 if  
                          # $s1 > $t2 (max)  
                          # or $s1 < 0 (min)  
beq $t0, $zero, IOOB    # go to IOOB if  
                          # $t0 = 0
```

- ☐ The key is that negative integers in two's complement look like large numbers in unsigned notation. Thus, an unsigned comparison of $x < y$ also checks if x is negative as well as if x is less than y .

Other Control Flow Instructions

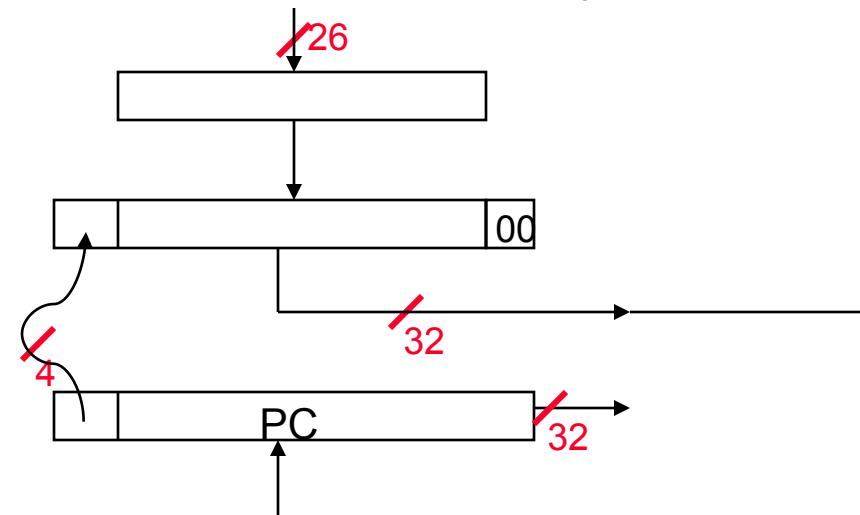
- MIPS also has an unconditional branch instruction or **jump** instruction:

j label #go to label

- Instruction Format (**J** Format):

| | |
|-------|----------------|
| 6'h02 | 26-bit address |
|-------|----------------|

from the low order 26 bits of the jump instruction



Aside: Branching Far Away

- ❑ What if the branch destination is further away than can be captured in 16 bits?
- ❑ The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

beq \$s0, \$s1, L1

becomes

bne \$s0, \$s1, L2
j L1

L2:

Instructions for Accessing Procedures

- ❑ MIPS **procedure call** instruction:

jal ProcedureAddress #jump and link

- ❑ Saves PC+4 in register \$ra to have a link to the next instruction for the procedure return
- ❑ Machine format (**J** format):

| | |
|-------|----------------|
| 6'h03 | 26 bit address |
|-------|----------------|

- ❑ Then can do procedure **return** with a

jr \$ra #return

- ❑ Instruction format (**R** format):

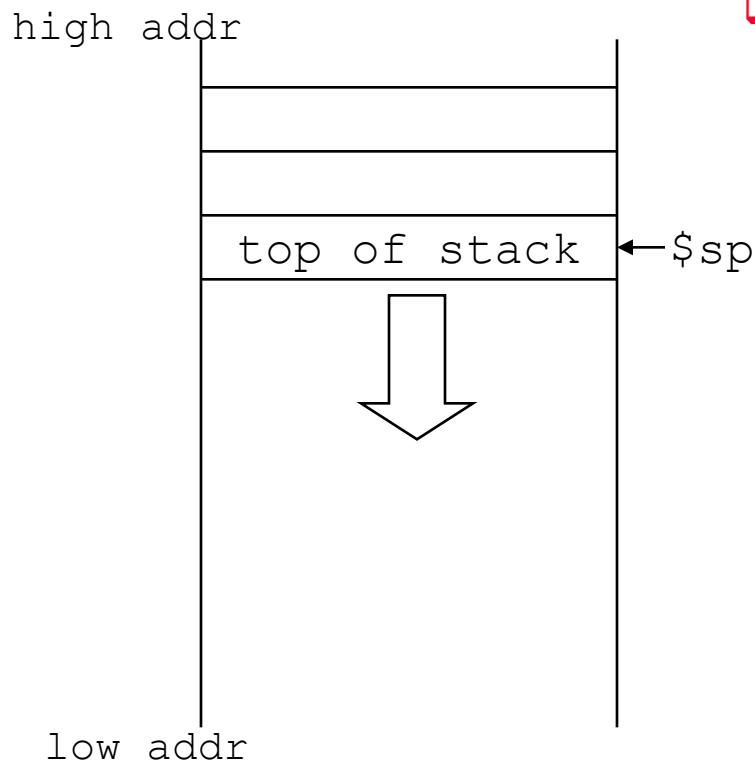
| | | | | | |
|------|-------|------|------|------|-------|
| 6'h0 | 5'd31 | 5'd0 | 5'd0 | 5'd0 | 6'h08 |
|------|-------|------|------|------|-------|

Six Steps in Execution of a Procedure

1. Main routine (**caller**) places parameters in a place where the procedure (**callee**) can access them
 - \$a0 - \$a3: four **argument** registers
2. **Caller** transfers control to the **callee**
3. **Callee** acquires the storage resources needed
4. **Callee** performs the desired task
5. **Callee** places the result value in a place where the **caller** can access it
 - \$v0 - \$v1: two **value** registers for result values
6. **Callee** returns control to the **caller**
 - \$ra: one **return address** register to return to the point of origin

Aside: Spilling Registers

- ❑ What if the **callee** needs to use more registers than allocated to argument and return values?
 - **callee** uses a **stack** – a last-in-first-out queue



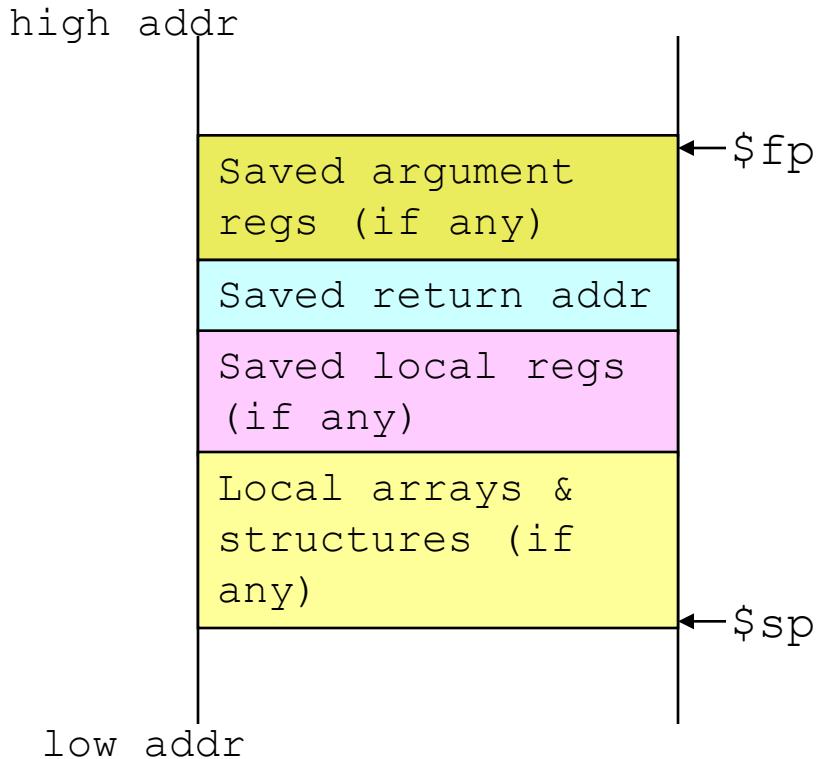
- ❑ One of the general registers, **\$sp** (\$29), is used to address the stack (which “grows” from high address to low address)

- add data onto the stack – **push**
$$\$sp = \$sp - 4$$

data **on** stack at new \$sp
- remove data from the stack – **pop**

data **from** stack at \$sp
$$\$sp = \$sp + 4$$

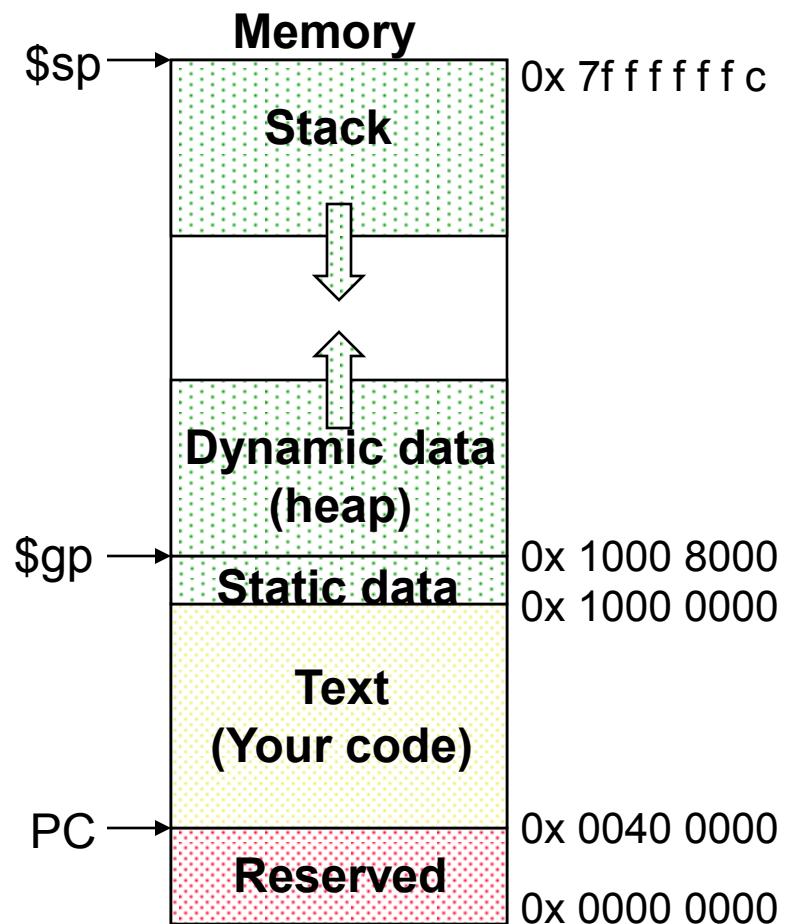
Aside: Allocating Space on the Stack



- The segment of the stack containing a procedure's saved registers and local variables is its **procedure frame (aka activation record)**
 - The frame pointer (\$fp) points to the first word of the frame of a procedure – providing a stable “base” register for the procedure
 - \$fp is initialized using \$sp on a call and \$sp is restored using \$fp on a return

Aside: Allocating Space on the Heap

- ❑ Static data segment for constants and other static variables (e.g., arrays)
- ❑ Dynamic data segment (aka **heap**) for structures that grow and shrink (e.g., linked lists)
 - Allocate space on the heap with `malloc()` and free it with `free()` in C



Atomic Exchange Support

- ❑ Need hardware support for synchronization mechanisms to avoid **data races** where the results of the program can change depending on how events happen to occur
 - Two memory accesses from different threads to the same location, and at least one is a write
- ❑ **Atomic exchange** (atomic swap) – interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)
 - Implementing an atomic exchange would require both a memory read and a memory write in a single, uninterruptable instruction. An alternative is to have a pair of specially configured instructions

```
ll    $t1, 0($s1)      #load linked
sc    $t0, 0($s1)      #store conditional
```

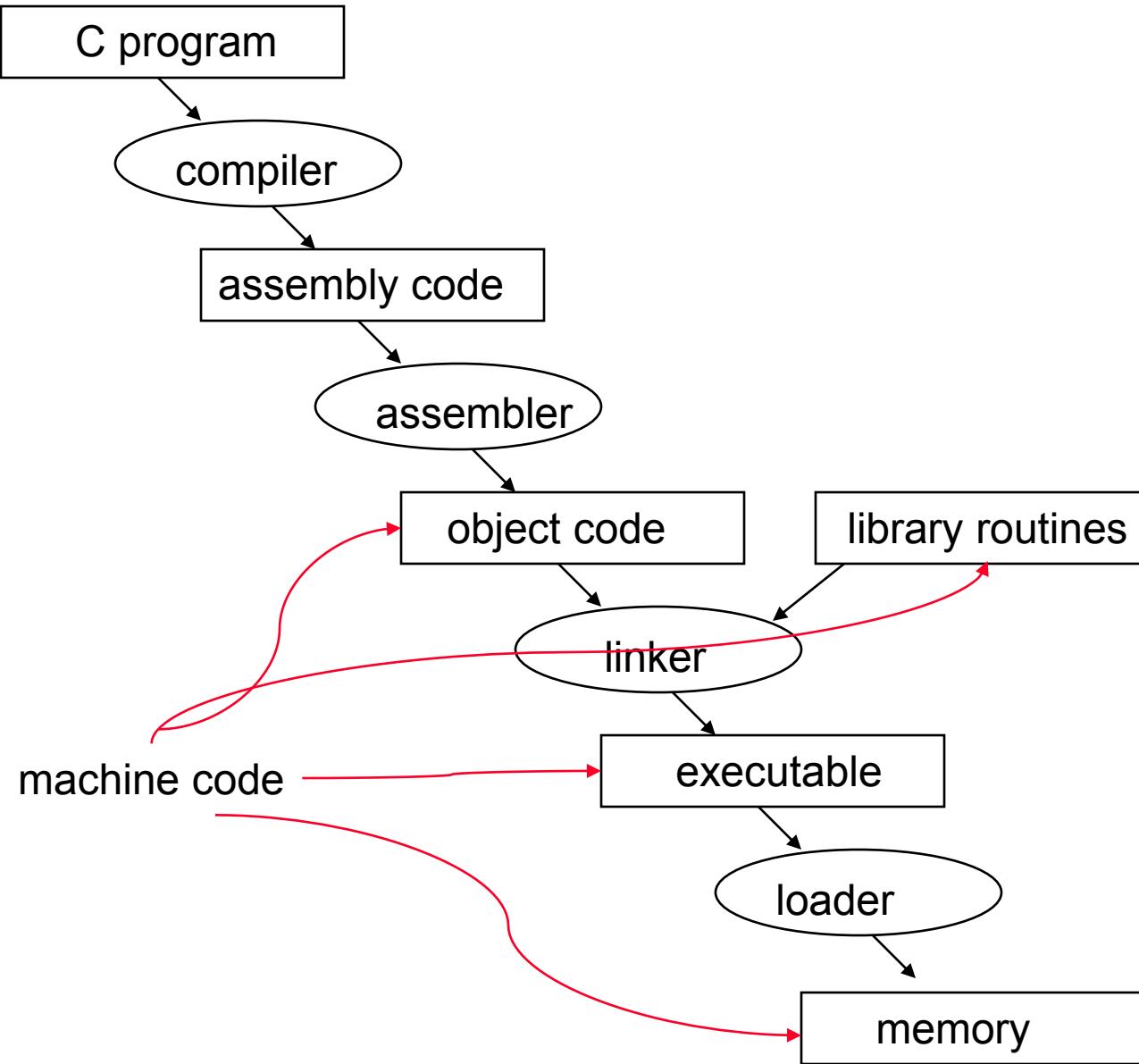
Automic Exchange with ll and sc

- ❑ If the contents of the memory location specified by the `ll` are changed before the `sc` to the same address occurs, the `sc` fails (returns a zero)

```
try:    add $t0, $zero, $s4      #$t0=$s4 (exchange value)
        ll   $t1, 0($s1)          #load memory value to $t1
        sc   $t0, 0($s1)          #try to store exchange
                                #value to memory, if fail
                                #$t0 will be 0
        beq $t0, $zero, try       #try again on failure
        add $s4, $zero, $t1        #load value in $s4
```

- ❑ If the value in memory between the `ll` and the `sc` instructions changes, then `sc` returns a 0 in `$t0` causing the code sequence to try again.

The C Code Translation Hierarchy



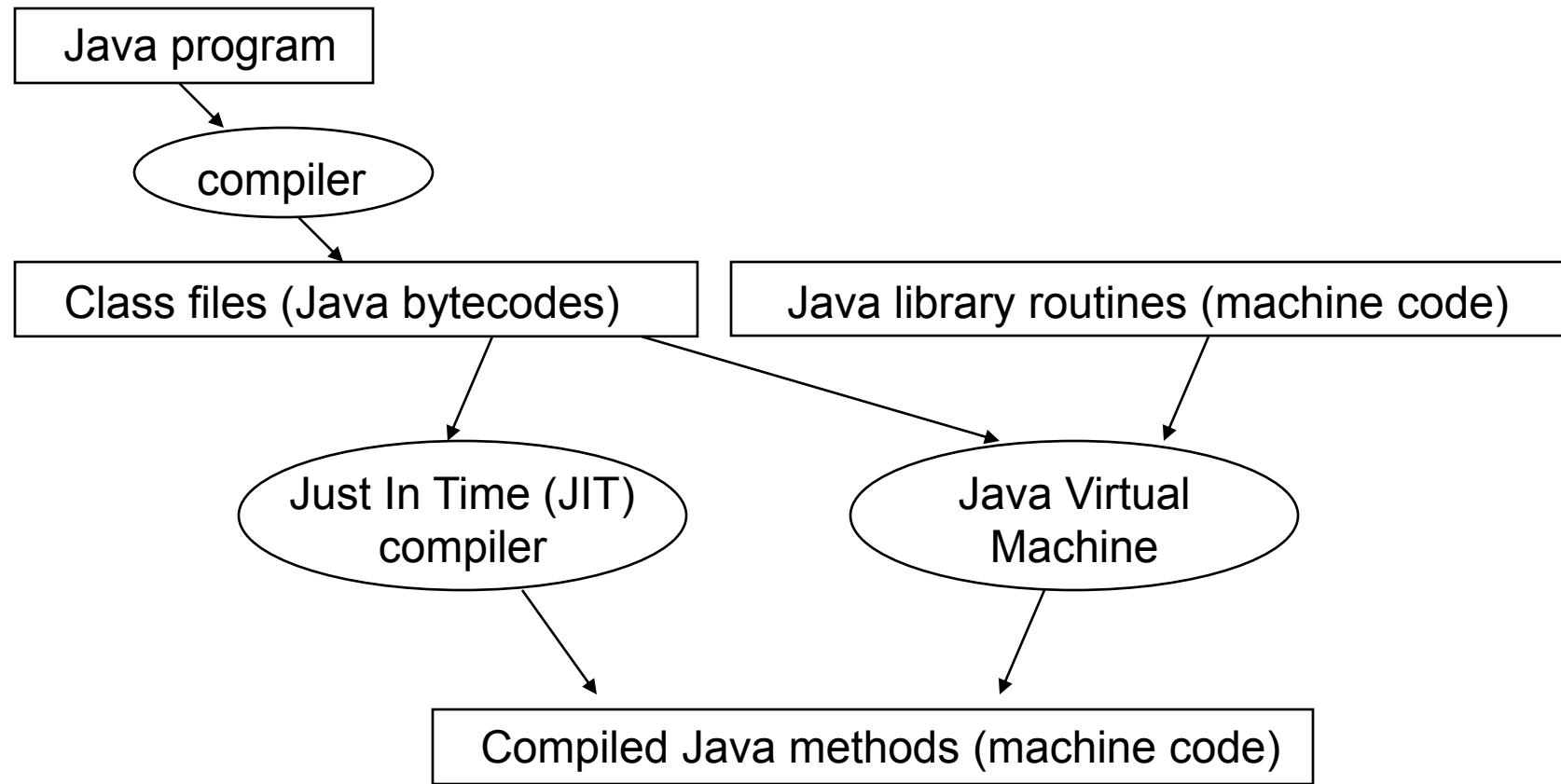
Compiler Benefits

- ❑ Comparing performance for bubble (exchange) sort
 - To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

| gcc opt | Relative performance | Clock cycles (M) | Instr count (M) | CPI |
|---------------|----------------------|------------------|-----------------|------|
| None | 1.00 | 158,615 | 114,938 | 1.38 |
| O1 (medium) | 2.37 | 66,990 | 37,470 | 1.79 |
| O2 (full) | 2.38 | 66,521 | 39,993 | 1.66 |
| O3 (proc mig) | 2.41 | 65,747 | 44,993 | 1.46 |

- ❑ The unoptimized code has the best CPI, the O1 version has the lowest instruction count, but the O3 version is the fastest. Why?

The Java Code Translation Hierarchy



Sorting in C versus Java

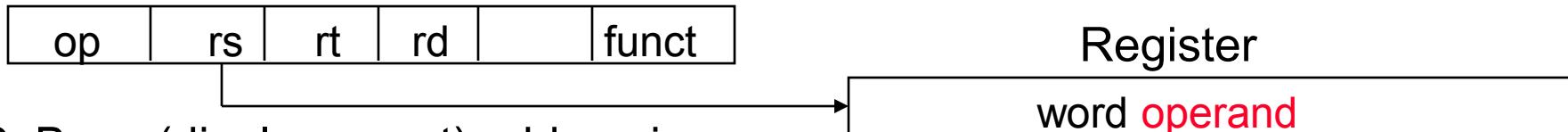
- ❑ Comparing performance for two sort algorithms in C and Java
 - The JVM/JIT is Sun/Hotspot version 1.3.1/1.3.1

| | Method | Opt | Bubble | Quick | Speedup quick vs bubble |
|------|--------------|------|-------------------------|-------|-------------------------------|
| | | | Relative performance | | |
| C | Compiler | None | 1.00 | 1.00 | 2468 |
| C | Compiler | O1 | 2.37 | 1.50 | 1562 |
| C | Compiler | O2 | 2.38 | 1.50 | 1555 |
| C | Compiler | O3 | 2.41 | 1.91 | 1955 |
| Java | Interpreted | | 0.12 | 0.05 | 1050 |
| Java | JIT compiler | | 2.13 | 0.29 | 338 |

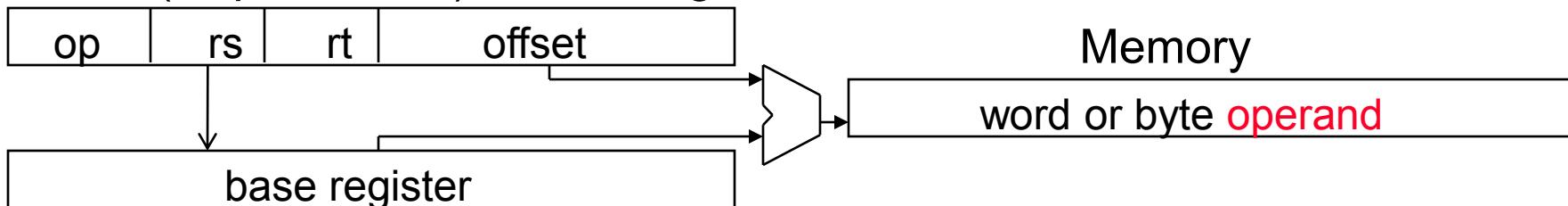
- ❑ Observations?

Addressing Modes Illustrated

1. Register addressing



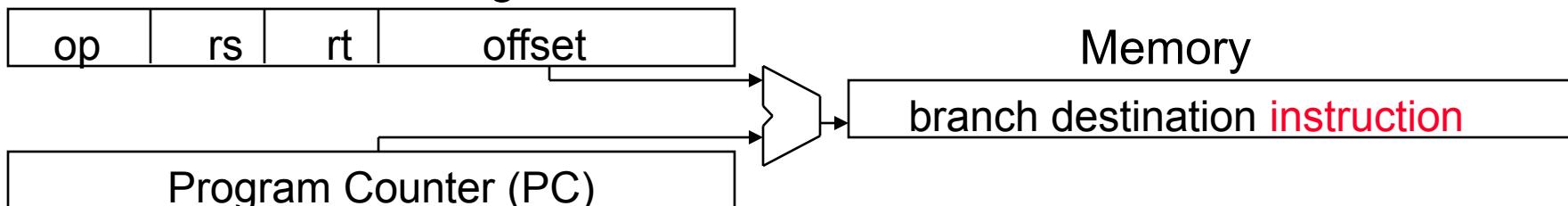
2. Base (displacement) addressing



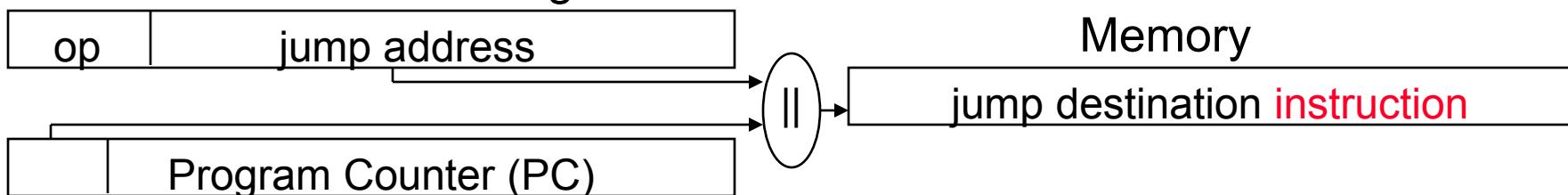
3. Immediate addressing



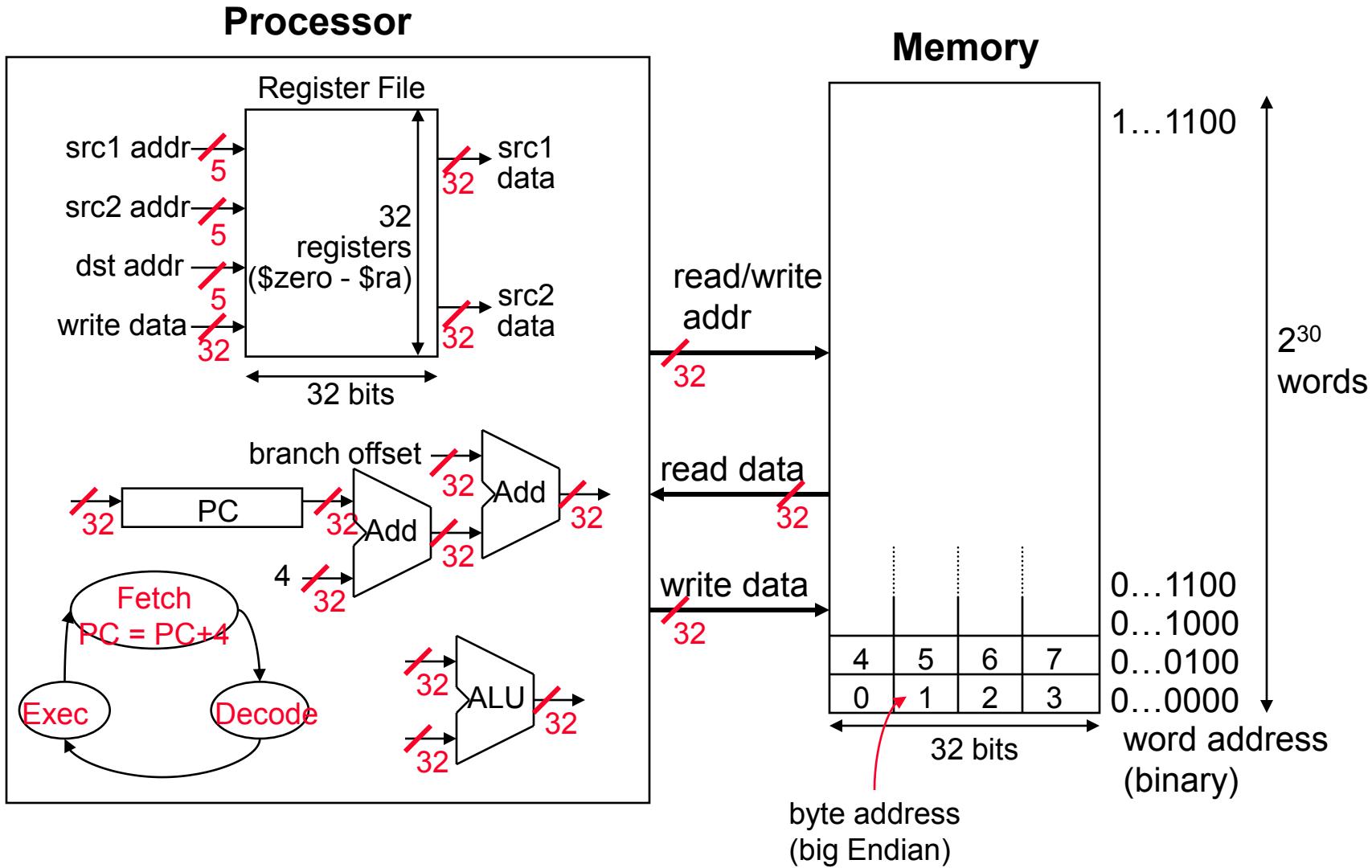
4. PC-relative addressing



5. Pseudo-direct addressing



MIPS Organization So Far



Next Lecture and Reminders

❑ Next lecture

- MIPS ALU Review
 - PH, Chapter 3

❑ Reminders

- Labs start this week. You must attend for safety induction.
- You should now have all information required for this weeks lab task.

ECE4074 Computer Architecture Semester 2 2014

Chapter 3: Arithmetic for Computers

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: MIPS (RISC) Design Principles

❑ Simplicity favors regularity

- fixed size instructions
- small number of instruction formats
- opcode always the first 6 bits

❑ Smaller is faster

- limited instruction set
- limited number of registers in register file
- limited number of addressing modes

❑ Make the common case fast

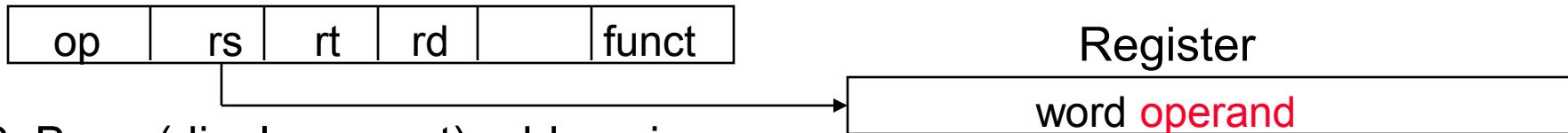
- arithmetic operands from the register file (load-store machine)
- allow instructions to contain immediate operands

❑ Good design demands good compromises

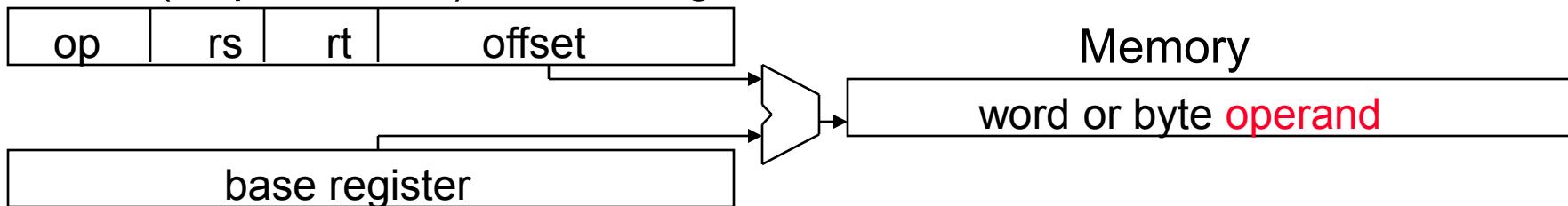
- three instruction formats

Review: MIPS Addressing Modes Illustrated

1. Register addressing



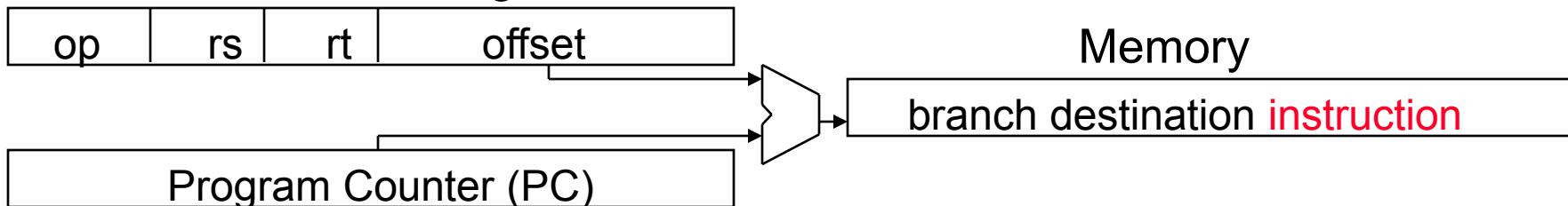
2. Base (displacement) addressing



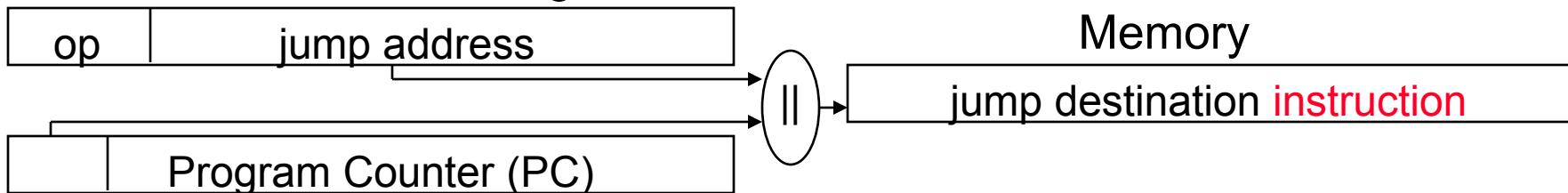
3. Immediate addressing



4. PC-relative addressing



5. Pseudo-direct addressing



Number Representations

❑ 32-bit signed numbers (2's complement):

| | | |
|--|----------------------------------|--------|
| 0000 0000 0000 0000 0000 0000 0000 0000 | = 0 _{ten} | maxint |
| 0000 0000 0000 0000 0000 0000 0000 0001 _{two} | = + 1 _{ten} | |
| ... | | |
| 0111 1111 1111 1111 1111 1111 1111 1110 _{two} | = + 2,147,483,646 _{ten} | |
| 0111 1111 1111 1111 1111 1111 1111 1111 _{two} | = + 2,147,483,647 _{ten} | |
| 1000 0000 0000 0000 0000 0000 0000 0000 | = - 2,147,483,648 _{ten} | |
| 1000 0000 0000 0000 0000 0000 0000 0001 _{two} | = - 2,147,483,647 _{ten} | |
| ... | | |
| 1111 1111 1111 1111 1111 1111 1111 1110 _{two} | = - 2 _{ten} | |
| 1111 1111 1111 1111 1111 1111 1111 1111 _{two} | = - 1 _{ten} | minint |
| MSB | | |
| LSB | | |

❑ Converting <32-bit values into 32-bit values

- copy the most significant bit (the sign bit) into the “empty” bits

0010 → 0000 0010

1010 → 1111 1010

- sign extend versus zero extend (lb vs. lbu)

MIPS Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

add, addi, addiu, addu

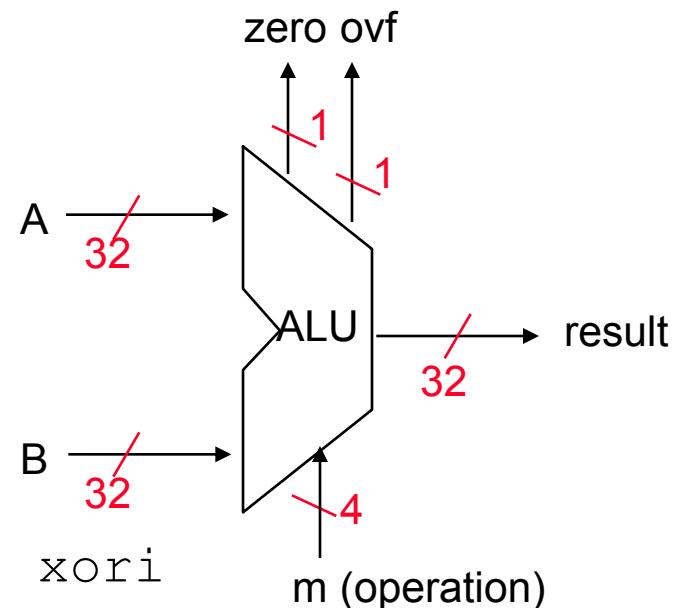
sub, subu

mult, multu, div, divu

sqrt

and, andi, nor, or, ori, xor, xori

beq, bne, slt, slti, sltiu, sltu



- With special handling for

- sign extend – addi, addiu, slti, sltiu
- zero extend – andi, ori, xori
- overflow detection – add, addi, sub

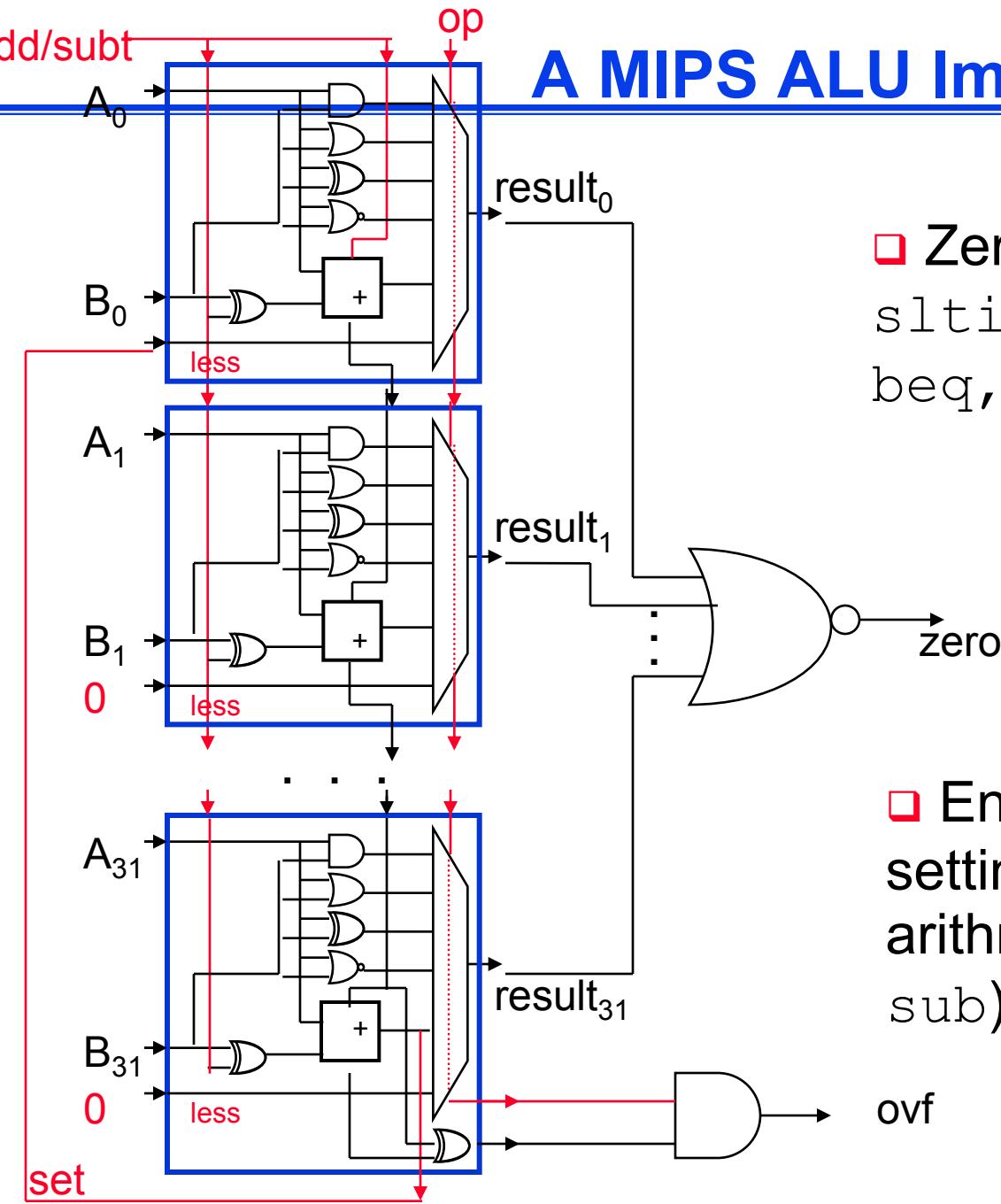
Dealing with Overflow

- ❑ Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a **value** bit of the result and not the proper **sign** bit
 - ❑ When adding operands with different signs or when subtracting operands with the same sign, overflow can *never* occur

| Operation | Operand A | Operand B | Result indicating overflow |
|-----------|-----------|-----------|----------------------------|
| $A + B$ | ≥ 0 | ≥ 0 | < 0 |
| $A + B$ | < 0 | < 0 | ≥ 0 |
| $A - B$ | ≥ 0 | < 0 | < 0 |
| $A - B$ | < 0 | ≥ 0 | ≥ 0 |

- ❑ MIPS signals overflow with an **exception** (aka interrupt) – an unscheduled procedure call where the EPC contains the address of the instruction that caused the exception

A MIPS ALU Implementation

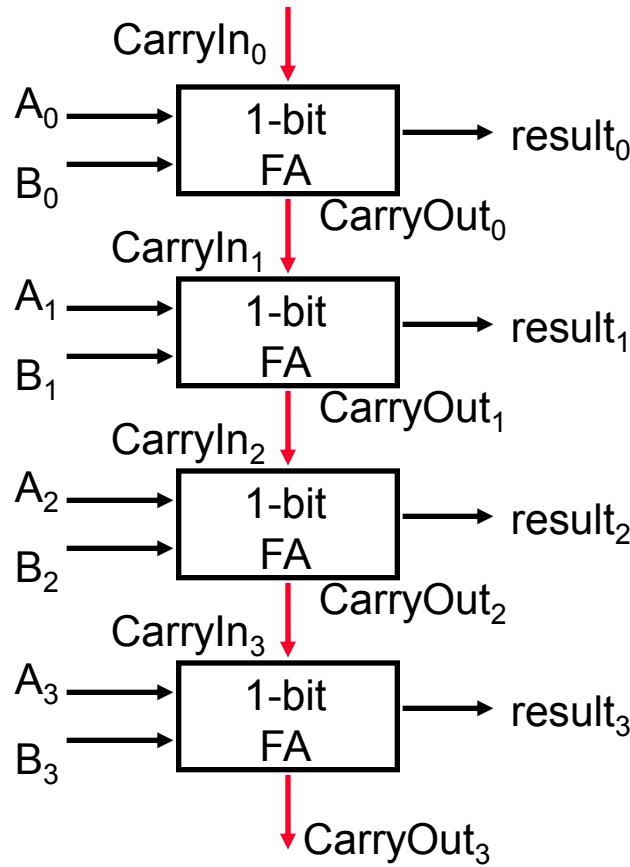


- Zero detect (`slt`, `slti`, `sltiu`, `sltu`, `beq`, `bne`)

- Enable overflow bit setting for signed arithmetic (`add`, `addi`, `sub`)

But What about Performance?

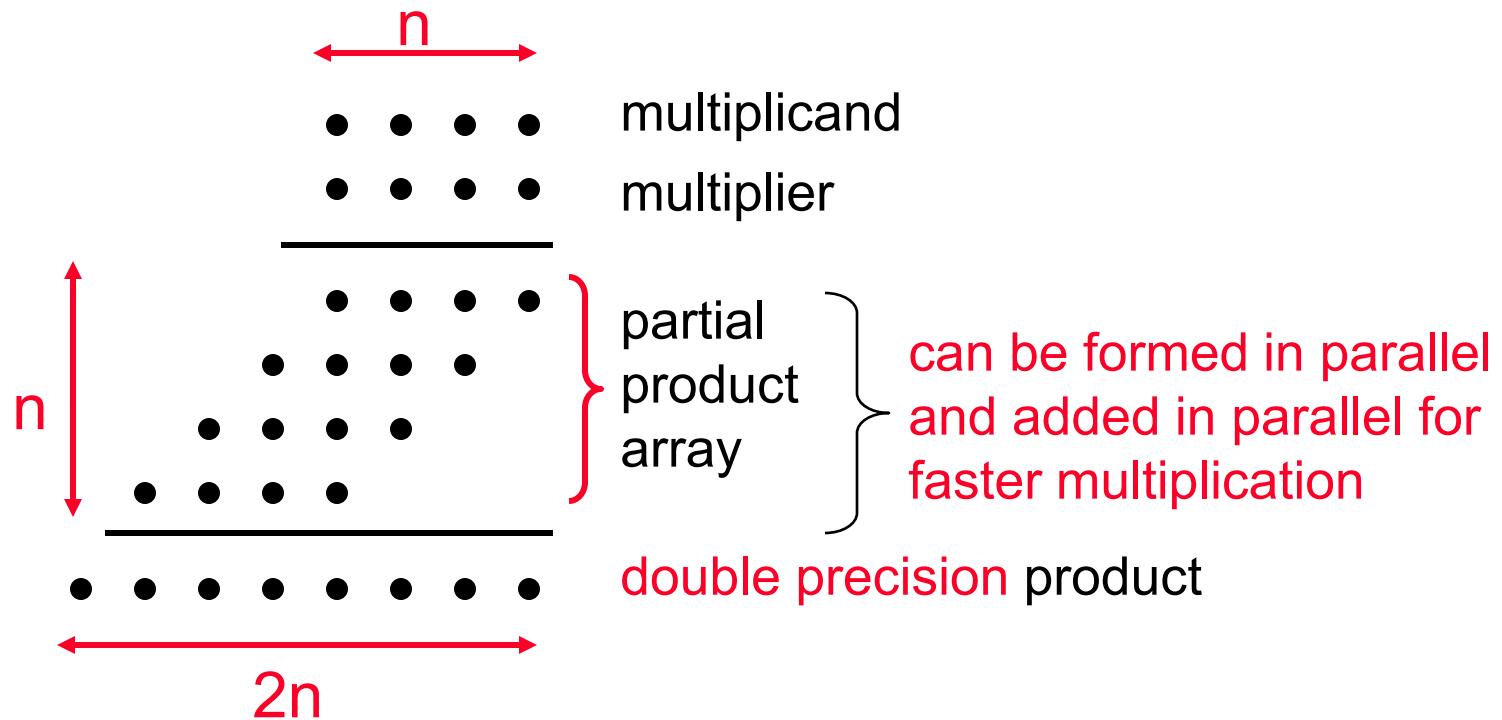
- Critical path of n-bit ripple-carry adder is $n \times CP$



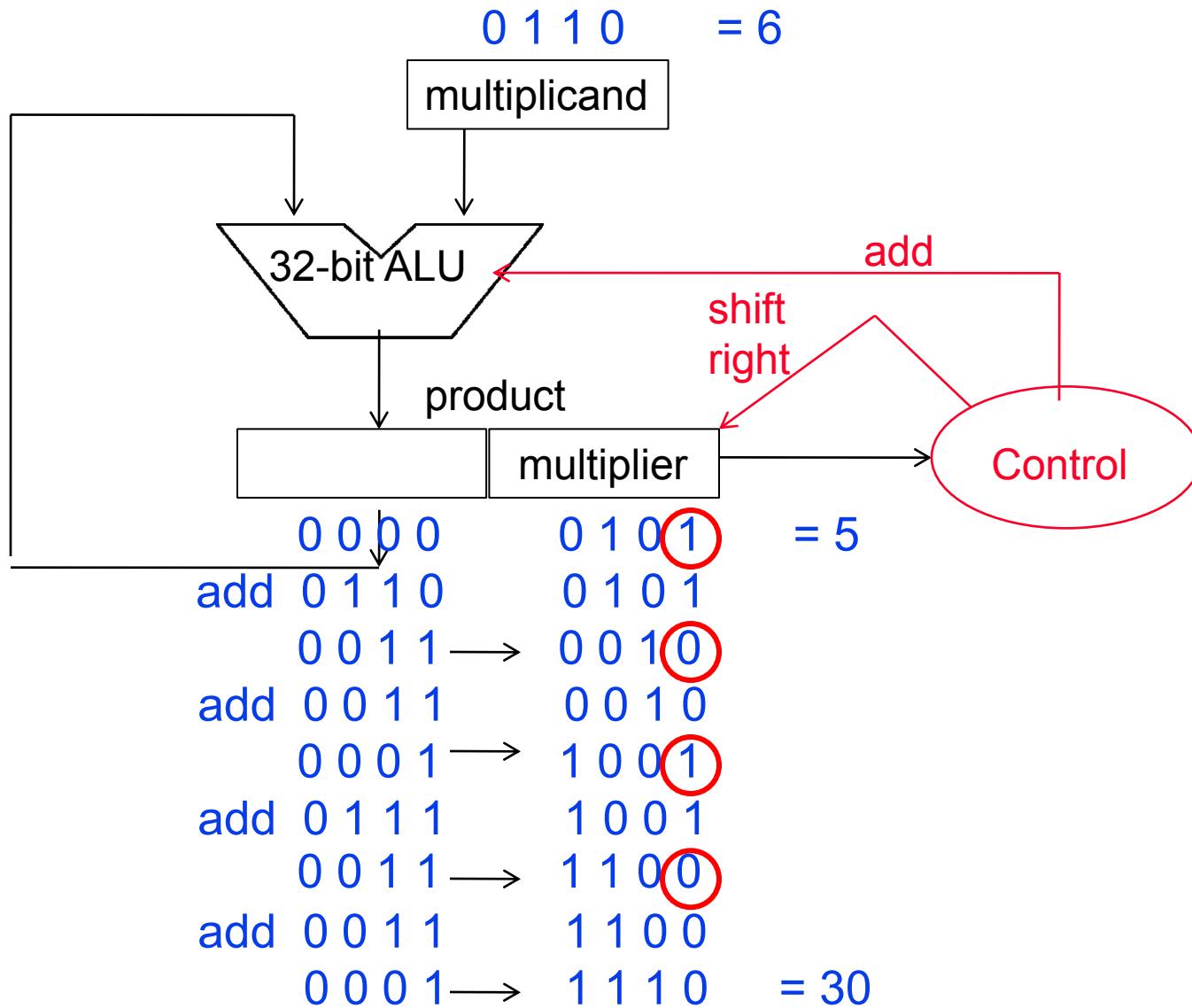
- Design trick – throw hardware at it (Carry Lookahead)

Multiply

- Binary multiplication is just a *bunch* of right shifts and adds



Add and Right Shift Multiplier Hardware



MIPS Multiply Instruction

- ❑ Multiply (mult and multu) produces a double precision product

```
mult      $s0, $s1          # hi || lo = $s0 * $s1
```

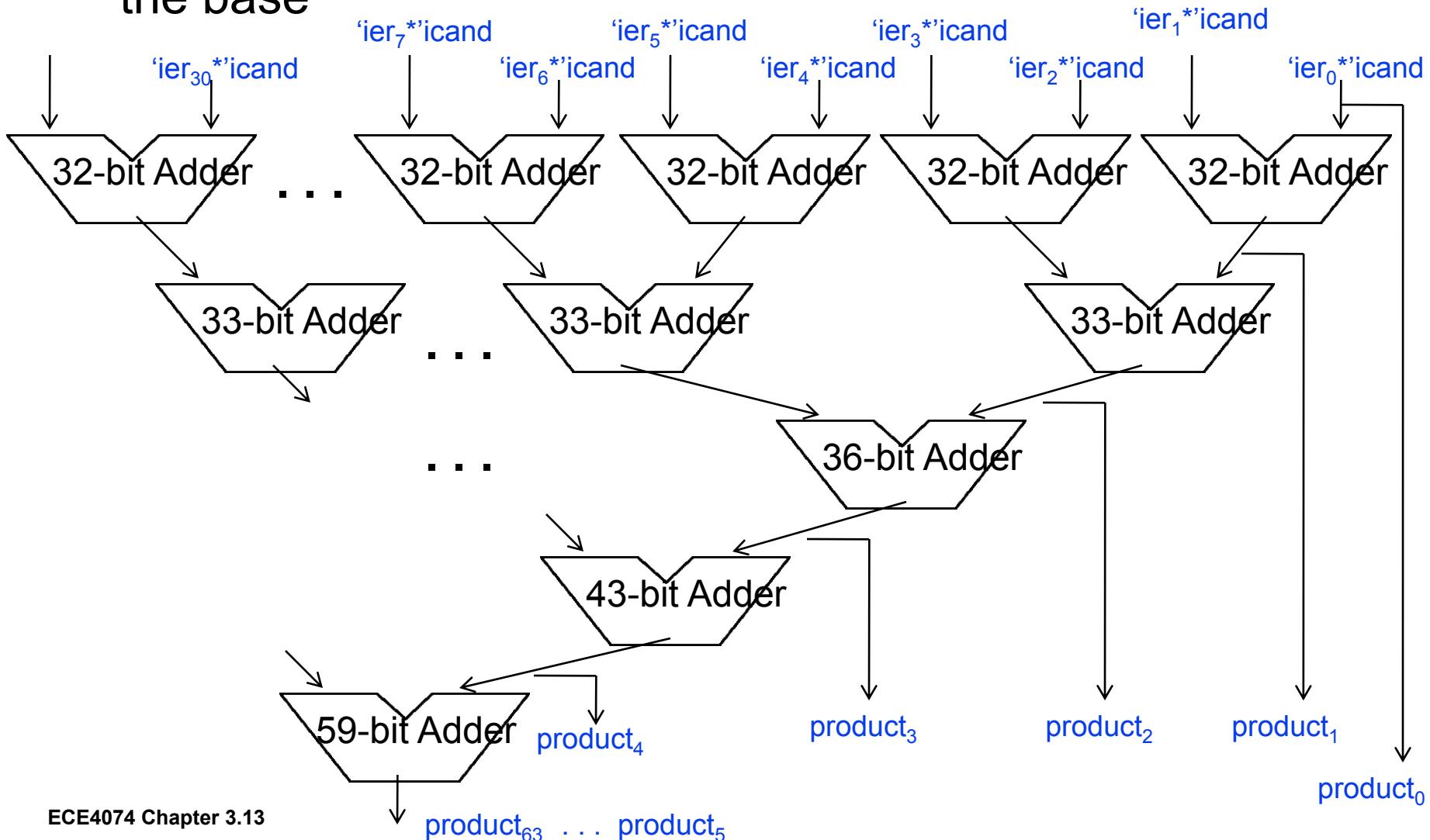
| | | | | | |
|------|-------|-------|------|------|-------|
| 6'd0 | 5'd16 | 5'd17 | 5'd0 | 5'd0 | 6'h18 |
|------|-------|-------|------|------|-------|

- Low-order word of the product is left in processor register lo and the high-order word is left in register hi
- Instructions mfhi rd and mflo rd are provided to move the product to (user accessible) registers in the register file

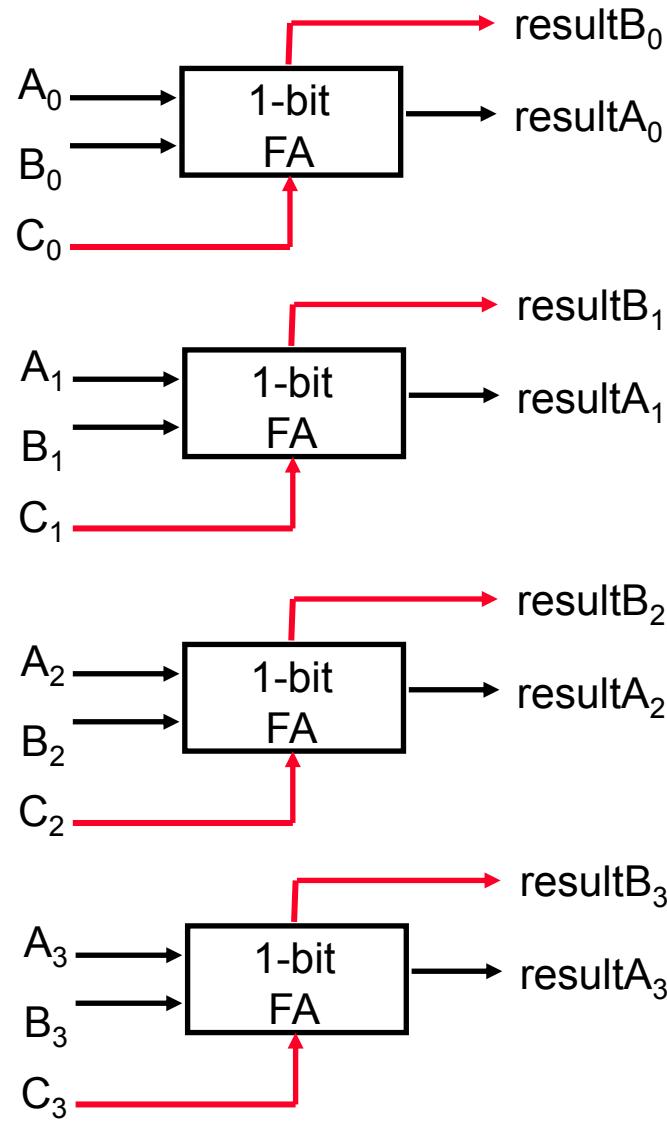
- ❑ Multiplies are usually done by fast, dedicated hardware and are much more complex (and slower) than adders

Fast Multiplication Hardware

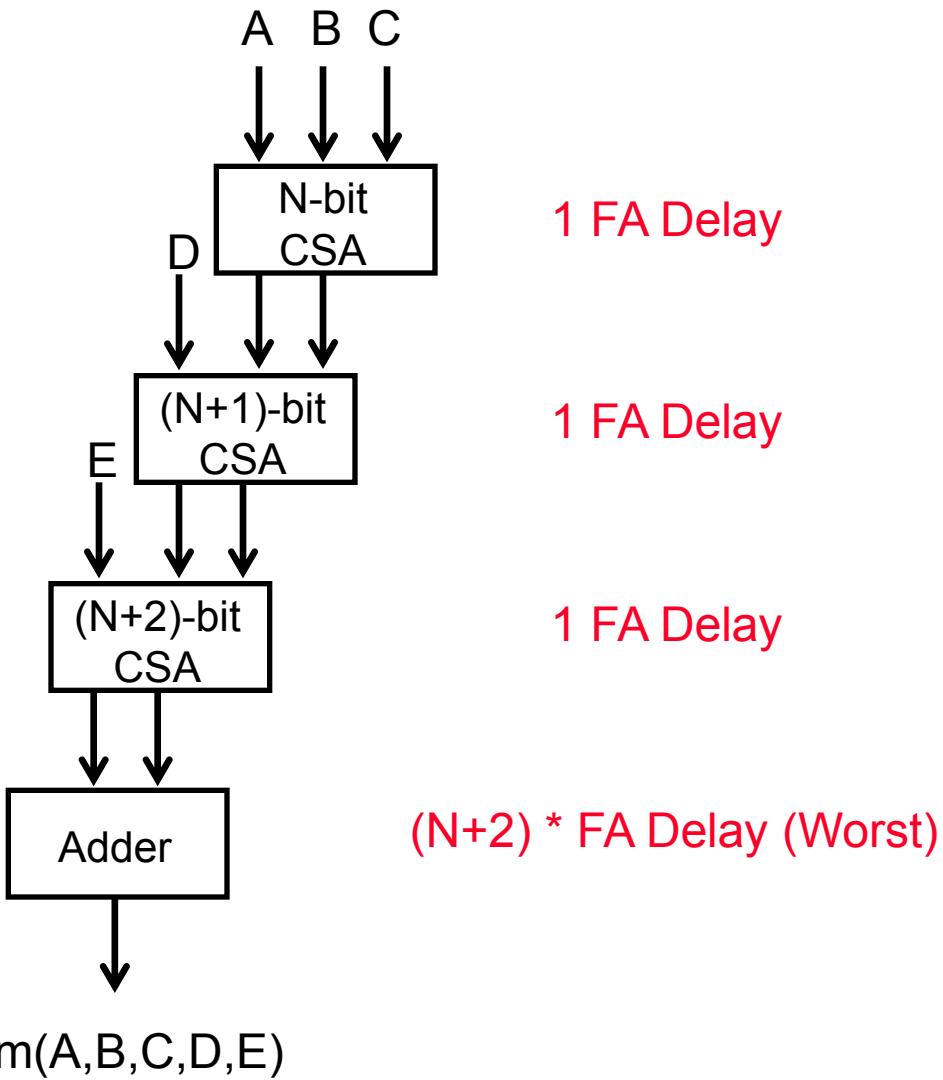
- Can build a faster multiplier by using a parallel tree of adders with one 32-bit adder for each bit of the multiplier at the base



Carry Save Adder



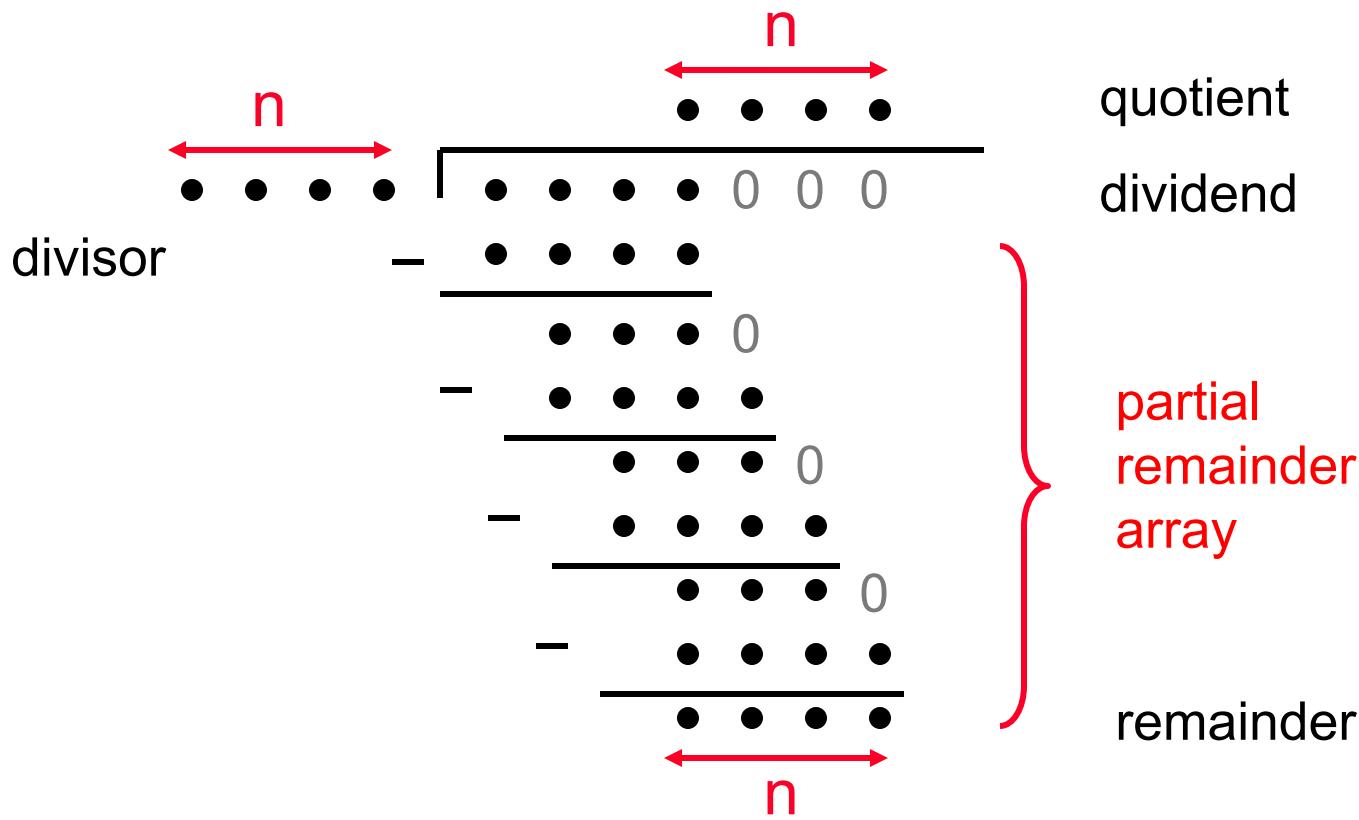
Carry Save Adder



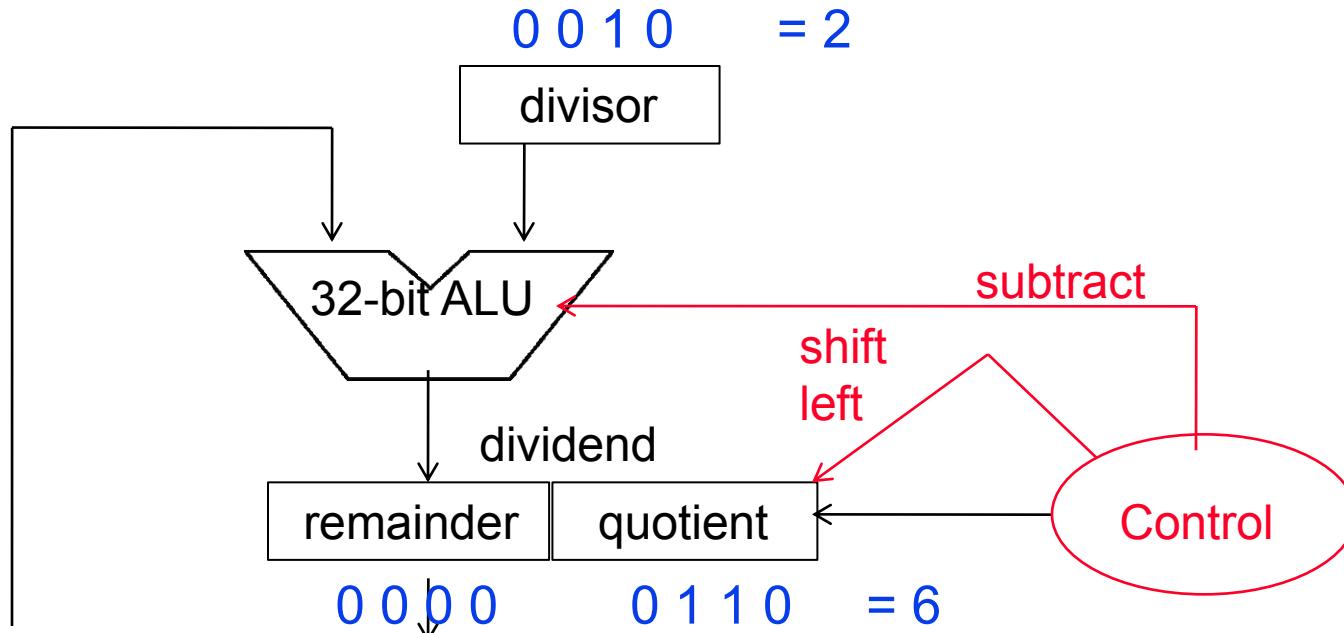
Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

$$\text{dividend} = \text{quotient} \times \text{divisor} + \text{remainder}$$



Left Shift and Subtract Division Hardware



= 6

rem neg, so 'ient bit = 0
restore remainder

rem neg, so 'ient bit = 0
restore remainder

rem pos, so 'ient bit = 1

rem pos, so 'ient bit = 1

= 3 with 0 remainder

MIPS Divide Instruction

- ❑ Divide (div and divu) generates the remainder in hi and the quotient in lo

```
div      $s0, $s1          # lo = $s0 / $s1  
                      # hi = $s0 mod $s1
```

| | | | | | |
|------|-------|-------|------|------|-------|
| 6'h0 | 5'd16 | 5'd17 | 5'd0 | 5'd0 | 6'h1A |
|------|-------|-------|------|------|-------|

- Instructions mfhi rd and mflo rd are provided to move the quotient and remainder to (user accessible) registers in the register file
- ❑ As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.

Note about Multiplication and Division

- ❑ These shift left/right add/subtract methods are a useful example of what a digital circuit needs to do to achieve a multiplication/division operation.
- ❑ The design shown in the previous slides attempt to minimize the logic required to perform the arithmetic.
- ❑ These are not high performance designs and I do not recommend using them in your assignment.

Representing Big (and Small) Numbers

- ❑ What if we want to encode the approx. age of the earth?

4,600,000,000 or 4.6×10^9

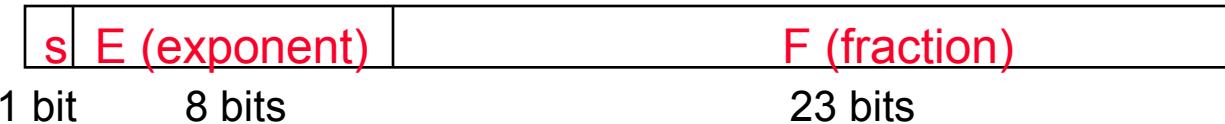
or the weight in kg of one a.m.u. (atomic mass unit)

0.000000000000000000000000000166 or 1.6×10^{-27}

There is no way we can encode either of the above in a 32-bit integer.

- ❑ Floating point representation $(-1)^{\text{sign}} \times F \times 2^E$

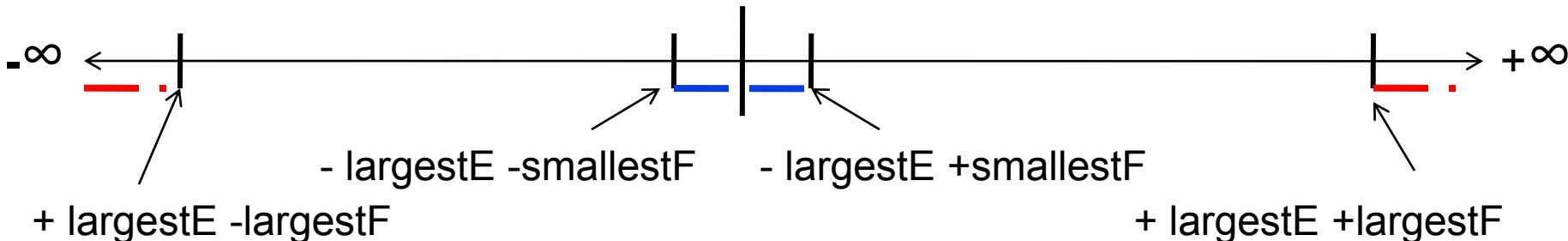
- Still have to fit everything in 32 bits (single precision)



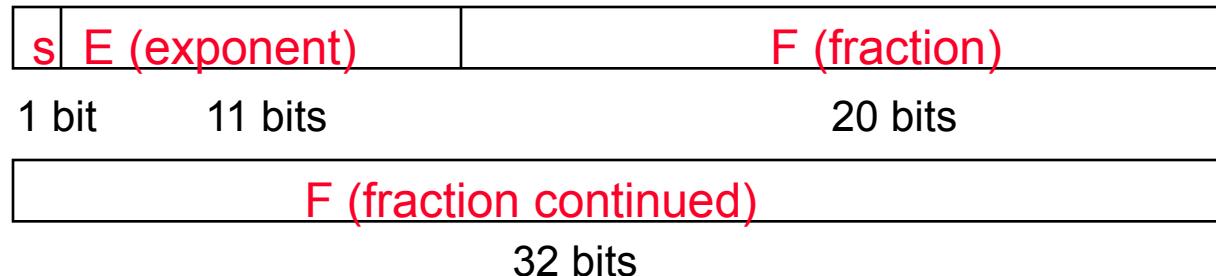
- The base (2, *not* 10) is hardwired in the design of the FPALU
 - More bits in the fraction (F) or the exponent (E) is a trade-off between **precision** (accuracy of the number) and **range** (size of the number)

Exception Events in Floating Point

- ❑ **Overflow** (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- ❑ **Underflow** (floating point) happens when a negative exponent becomes too large to fit in the exponent field



- ❑ One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
 - Double precision – takes two MIPS words



IEEE 754 FP Standard

- ❑ Most (all?) computers these days conform to the IEEE 754 floating point standard $(-1)^{\text{sign}} \times (1+F) \times 2^{\text{E-bias}}$
 - Formats for both single and double precision
 - F is stored in **normalized** format where the msb in F is 1 (so there is no need to store it!) – called the **hidden** bit
 - To simplify sorting FP numbers, E comes before F in the word and E is represented in **excess** (biased) notation where the bias is -127 (-1023 for double precision) so the most negative is 00000001 = $2^{1-127} = 2^{-126}$ and the most positive is 11111110 = $2^{254-127} = 2^{+127}$
- ❑ Examples (in normalized format)
 - Smallest+: 0 00000001 1.0000000000000000000000000000000 = $1 \times 2^{1-127}$
 - Zero: 0 00000000 0000000000000000000000000000000 = true 0
 - Largest+: 0 11111110 1.111111111111111111111111111111 = $2-2^{-23} \times 2^{254-127}$
 - $1.0_2 \times 2^{-1} = 0 0111110 1.0000000000000000000000000000000$
 - $0.75_{10} \times 2^4 = 0 1000010 1.1000000000000000000000000000000$

IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
 - \pm infinity for division by zero
 - NAN (not a number) for the results of invalid operations such as 0/0
 - True zero is the bit string all zero

| Single Precision | | Double Precision | | Object Represented |
|------------------------|----------|------------------------------|----------|-----------------------------|
| E (8) | F (23) | E (11) | F (52) | |
| 0000 0000 | 0 | 0000 ... 0000 | 0 | true zero (0) |
| 0000 0000 | nonzero | 0000 ... 0000 | nonzero | \pm denormalized number |
| 0111 1111 to +127,-126 | anything | 0111 ... 1111 to +1023,-1022 | anything | \pm floating point number |
| 1111 1111 | + 0 | 1111 ... 1111 | - 0 | \pm infinity |
| 1111 1111 | nonzero | 1111 ... 1111 | nonzero | not a number (NaN) |

Support for Accurate Arithmetic

- ❑ IEEE 754 FP rounding modes
 - Always round up (toward $+\infty$)
 - Always round down (toward $-\infty$)
 - Truncate
 - **Round to nearest even** (when the Guard || Round || Sticky are 100) – always creates a 0 in the least significant (kept) bit of F
- ❑ Rounding (except for truncation) requires the hardware to include extra F bits during calculations
 - Guard bit – used to provide one F bit when shifting left to normalize a result (e.g., when normalizing F after division or subtraction)
 - Round bit – used to improve rounding accuracy
 - Sticky bit – used to support **Round to nearest even**; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F during addition/subtraction)

$$F = 1 . \underset{\text{G}}{xxxxxx} \underset{\text{R}}{xxxxxxxx} \underset{\text{S}}{xxxxxxxx}$$

Floating Point Addition

□ Addition (and subtraction)

$$(\pm F_1 \times 2^{E_1}) + (\pm F_2 \times 2^{E_2}) = \pm F_3 \times 2^{E_3}$$

- Step 0: Restore the hidden bit in F1 and in F2
- Step 1: Align fractions by right shifting F2 by $E_1 - E_2$ positions (assuming $E_1 \geq E_2$) keeping track of (three of) the bits shifted out in G R and S
- Step 2: Add the resulting F2 to F1 to form F3
 - If F1 and F2 have the same sign $\rightarrow F_3 \in [1,4]$ \rightarrow 1 bit right shift F3 and increment E3 (check for overflow)
 - If F1 and F2 have different signs $\rightarrow F_3$ may require *many* left shifts each time decrementing E3 (check for underflow)
- Step 4: Round F3 and possibly normalize F3 again
- Step 5: Rehide the most significant bit of F3 before storing the result

Floating Point Addition Example

□ Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

- Step 0: Hidden bits restored in the representation above
- Step 1: Shift significand with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)
- Step 2: Add significands

$$1.0000 + (-0.111) = 1.0000 - 0.111 = 0.001$$

- Step 3: Normalize the sum, checking for exponent over/underflow
 $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = \dots = 1.000 \times 2^{-4}$
- Step 4: The sum is already rounded, so we're done
- Step 5: Rehide the hidden bit before storing

Floating Point Multiplication

□ Multiplication

$$(\pm F_1 \times 2^{E_1}) \times (\pm F_2 \times 2^{E_2}) = \pm F_3 \times 2^{E_3}$$

- Step 0: Restore the hidden bit in F1 and in F2
- Step 1: **Add** the two (biased) exponents and subtract the bias from the sum, so $E_1 + E_2 - 127 = E_3$
also determine the sign of the product (which depends on the sign of the operands (most significant bits))
- Step 2: **Multiply** F1 by F2 to form a double precision F3
- Step 3: **Normalize** F3 (so it is in the form 1.XXXXX ...)
 - Since F1 and F2 come in normalized $\rightarrow F_3 \in [1,4)$ \rightarrow 1 bit right shift F3 and increment E3
 - Check for overflow/underflow
- Step 4: **Round** F3 and possibly **normalize** F3 again
- Step 5: Rehide the most significant bit of F3 before storing the result

Floating Point Multiplication Example

□ Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

- Step 0: Hidden bits restored in the representation above
- Step 1: Add the exponents (not in bias would be $-1 + (-2) = -3$ and in bias would be $(-1+127) + (-2+127) - 127 = (-1-2) + (127+127-127) = -3 + 127 = 124$)
- Step 2: Multiply the significands
 $1.0000 \times 1.110 = 1.110000$
- Step 3: Normalized the product, checking for exp over/underflow
 1.110000×2^{-3} is already normalized
- Step 4: The product is already rounded, so we're done
- Step 5: Rehide the hidden bit before storing

MIPS Floating Point Instructions

- ❑ MIPS has a separate Floating Point Register File (\$f0, \$f1, ..., \$f31) (whose registers are used in *pairs* for double precision values) with special instructions to load to and store from them

```
lwcl    $f1, 54($s2)    #$f1 = Memory[$s2+54]  
swcl    $f1, 58($s4)    #Memory[$s4+58] = $f1
```

- ❑ And supports IEEE 754 single

```
add.s   $f2, $f4, $f6    #$f2 = $f4 + $f6
```

and double precision operations

```
add.d   $f2, $f4, $f6    #$f2 || $f3 =  
                           $f4 || $f5 + $f6 || $f7
```

similarly for sub.s, sub.d, mul.s, mul.d, div.s, div.d

MIPS Floating Point Instructions, Con't

- ❑ And floating point single precision comparison operations

c.x.s \$f2, \$f4 #if (\$f2 < \$f4) cond=1;
 else cond=0

where x may be eq, neq, lt, le, gt, ge

and double precision comparison operations

c.x.d \$f2, \$f4 #\$\$f2 || \$f3 < \$f4 || \$f5
 cond=1; else cond=0

- ❑ And floating point branch operations

bclt 25 #if (cond==1)
 go to PC+4+25

bclf 25 #if (cond==0)
 go to PC+4+25

Frequency of Common MIPS Instructions

- Only included those with >3% and >1%

| | SPECint | SPECfp |
|-------|----------------|---------------|
| addu | 5.2% | 3.5% |
| addiu | 9.0% | 7.2% |
| or | 4.0% | 1.2% |
| sll | 4.4% | 1.9% |
| lui | 3.3% | 0.5% |
| lw | 18.6% | 5.8% |
| sw | 7.6% | 2.0% |
| lbu | 3.7% | 0.1% |
| beq | 8.6% | 2.2% |
| bne | 8.4% | 1.4% |
| slt | 9.9% | 2.3% |
| slti | 3.1% | 0.3% |
| sltu | 3.4% | 0.8% |

| | SPECint | SPECfp |
|-------|----------------|---------------|
| add.d | 0.0% | 10.6% |
| sub.d | 0.0% | 4.9% |
| mul.d | 0.0% | 15.0% |
| add.s | 0.0% | 1.5% |
| sub.s | 0.0% | 1.8% |
| mul.s | 0.0% | 2.4% |
| l.d | 0.0% | 17.5% |
| s.d | 0.0% | 4.9% |
| l.s | 0.0% | 4.2% |
| s.s | 0.0% | 1.1% |
| lhu | 1.3% | 0.0% |

Next Lecture and Reminders

- ❑ Next lecture
 - MIPS single cycle datapath design

- ❑ Reminders
 - Your matrix multiplication should be handed into labs this week if you want to pass this weeks optional lab component.
 - Start Verilog implementation according to your assignment for your lab next week.

ECE4074

Advanced Computer Architecture

Semester 2 2014

Chapter 4A: The Processor, Part A I

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: MIPS (RISC) Design Principles

❑ Simplicity favors regularity

- fixed size instructions
- small number of instruction formats
- opcode always the first 6 bits

❑ Smaller is faster

- limited instruction set
- limited number of registers in register file
- limited number of addressing modes

❑ Make the common case fast

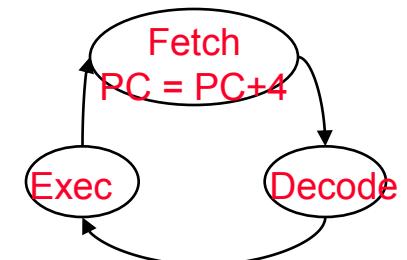
- arithmetic operands from the register file (load-store machine)
- allow instructions to contain immediate operands

❑ Good design demands good compromises

- three instruction formats

The Processor: Datapath & Control

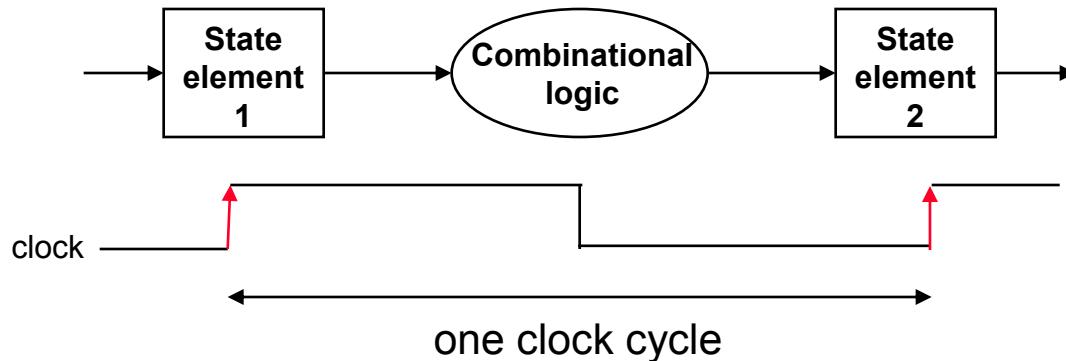
- ❑ Our implementation of the MIPS is simplified
 - memory-reference instructions: **lw**, **sw**
 - arithmetic-logical instructions: **add**, **sub**, **slt**, **mult**, **mflo**
 - control flow instructions: **beq**, **j**
- ❑ Generic implementation
 - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
 - decode the instruction (and read registers)
 - execute the instruction
- ❑ All instructions (except **j**) use the ALU after reading the registers



How? memory-reference? arithmetic? control flow?

Aside: Clocking Methodologies

- ❑ The **clocking methodology** defines when data in a state element is valid and stable relative to the clock
 - State elements - a memory element such as a register
 - Edge-triggered – all state changes occur on a clock edge
- ❑ Typical execution
 - read contents of state elements -> send values through combinational logic -> write results to one or more state elements

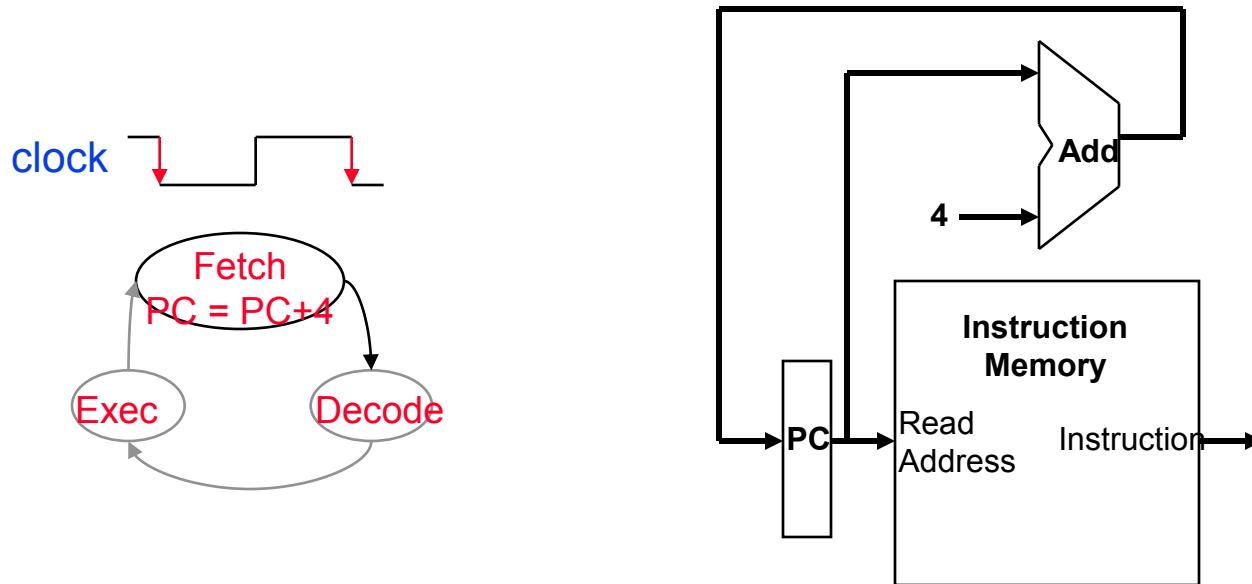


- ❑ Assumes state elements are written on every clock cycle; if not, need explicit write control signal
 - write occurs only when **both** the write control is asserted and the clock edge occurs

Fetching Instructions

❑ Fetching instructions involves

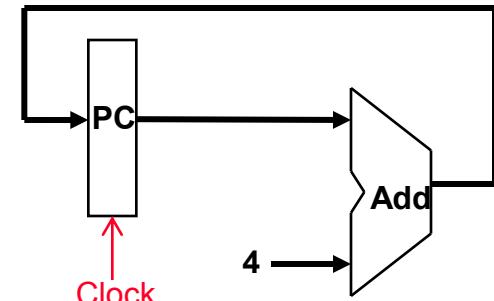
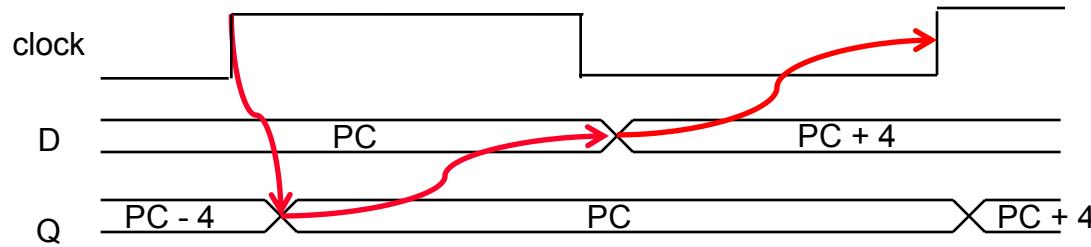
- reading the instruction from the Instruction Memory
- updating the PC value to be the address of the next (sequential) instruction



- PC is updated every clock cycle, so it does not need an explicit write control signal just a clock signal
- Reading from the Instruction Memory is a combinational activity, so it doesn't need an explicit read control signal

Aside: Clocks and Propagation Delays

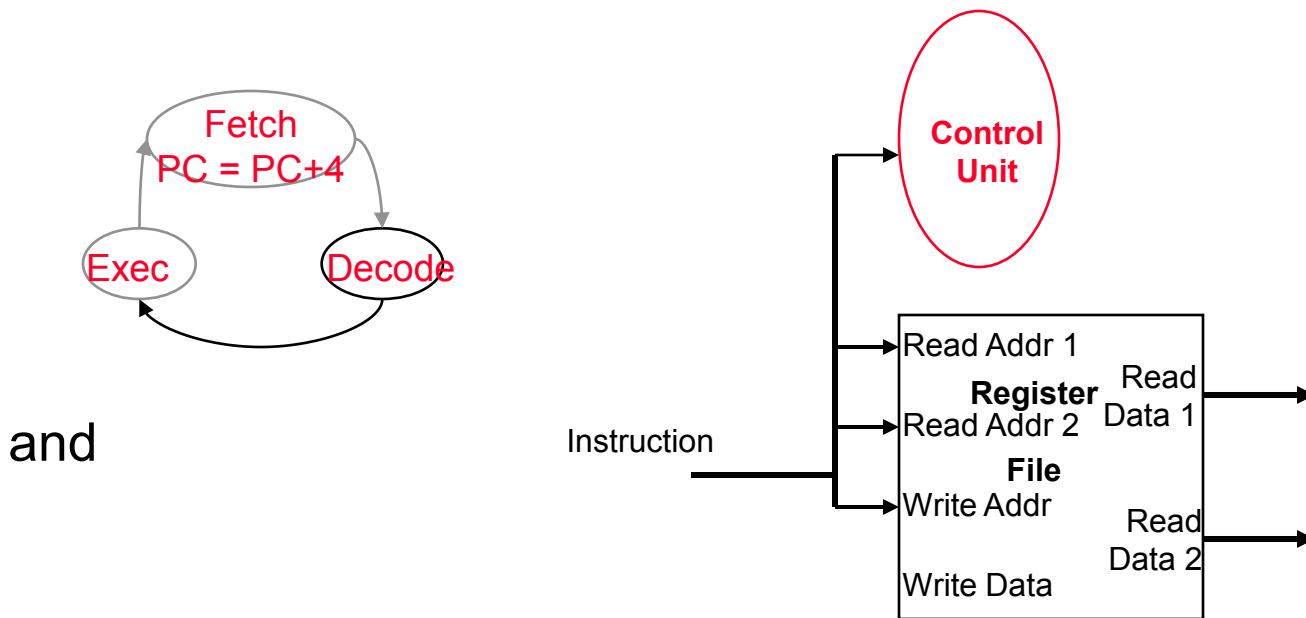
- ❑ PC is a 32 bit wide D flip-flop
- ❑ Propagation delays aren't instant



- ❑ Time is required for D to propagate to Q
- ❑ Minimum time data must be ready (correct value) before clock edge:
 - Setup Delay
- ❑ Minimum time data must be held (kept the same value) after clock edge:
 - Hold Delay

Decoding Instructions

- ❑ Decoding instructions involves
 - sending the fetched instruction's opcode and function field bits to the control unit

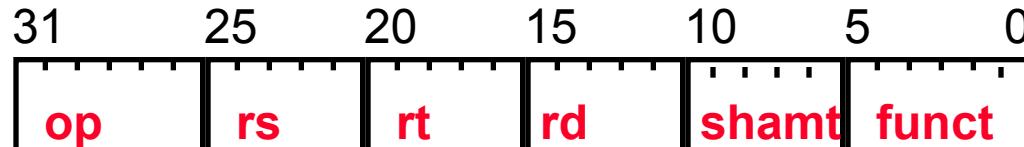


and

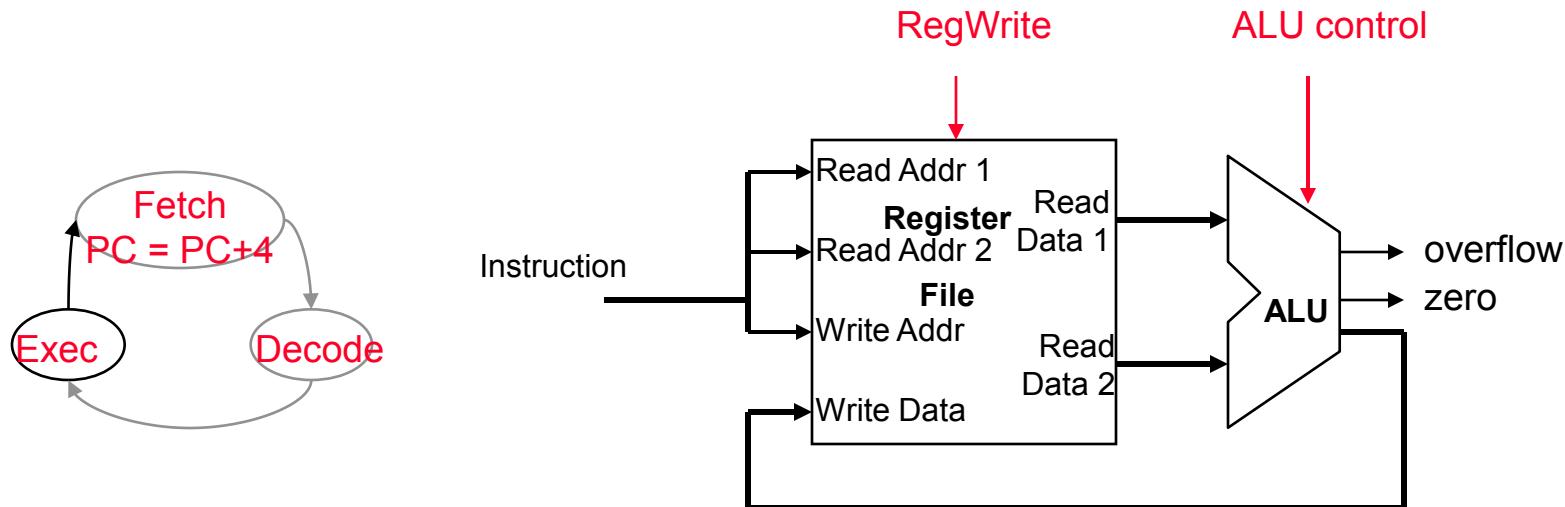
- reading two values from the Register File
 - Register File addresses are contained in the instruction

Executing R Format Operations

- R format operations (**add, sub, slt, and, or**)



- perform operation (**op** and **funct**) on values in **rs** and **rt**
- store the result back into the Register File (into location **rd**)

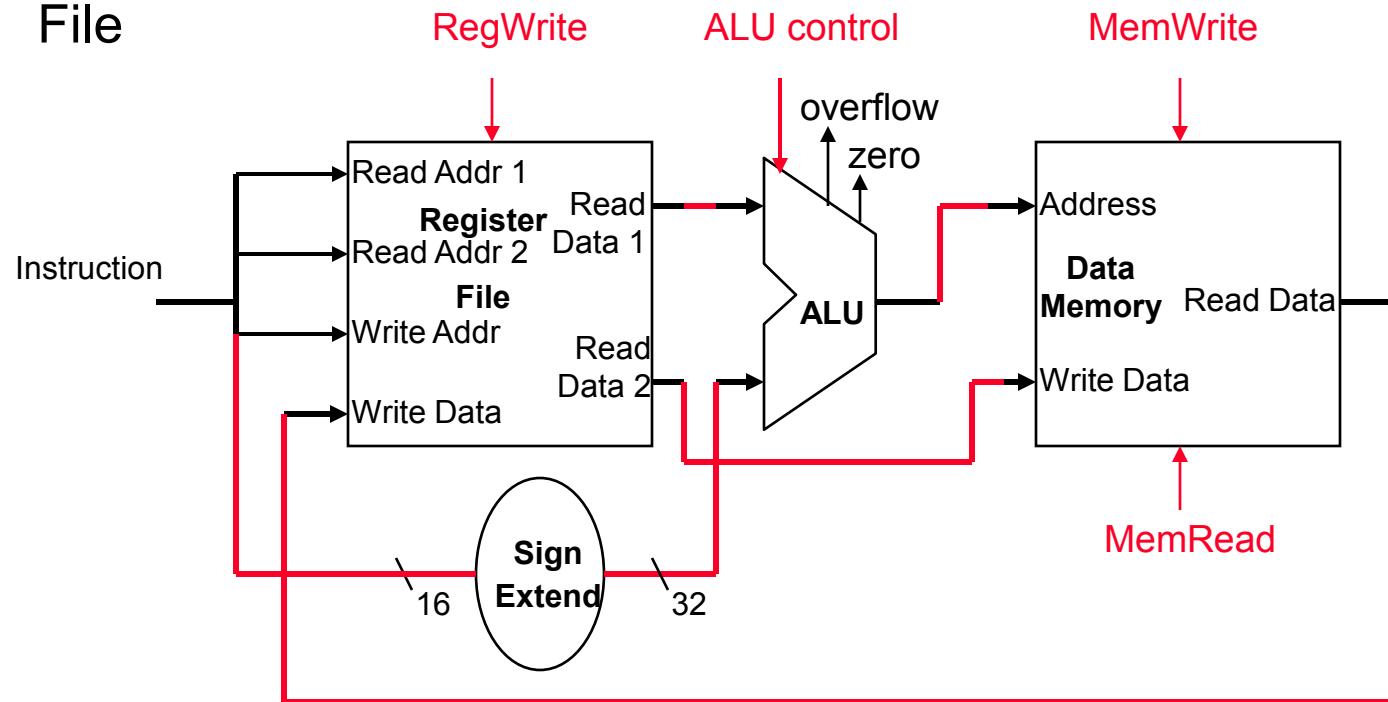


- Note that Register File is not written every cycle (e.g. **sw**), so we need an explicit write control signal for the Register File

Executing Load and Store Operations

❑ Load and store operations involves

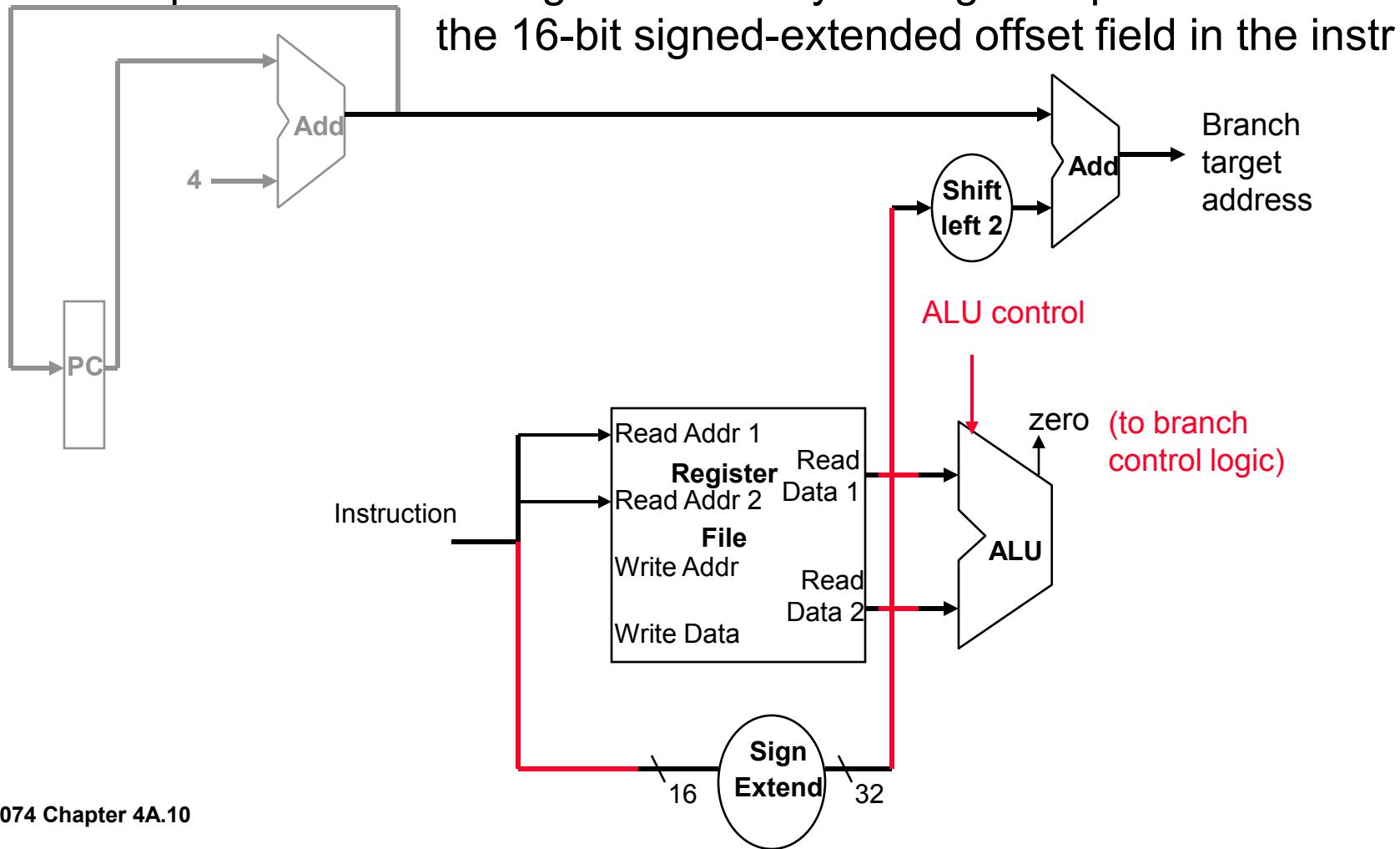
- compute memory address by adding the base register (read from the Register File during decode) to the 16-bit signed-extended offset field in the instruction
- **store** value (read from the Register File during decode) written to the Data Memory
- **load** value, read from the Data Memory, written to the Register File



Executing Branch Operations

❑ Branch operations involves

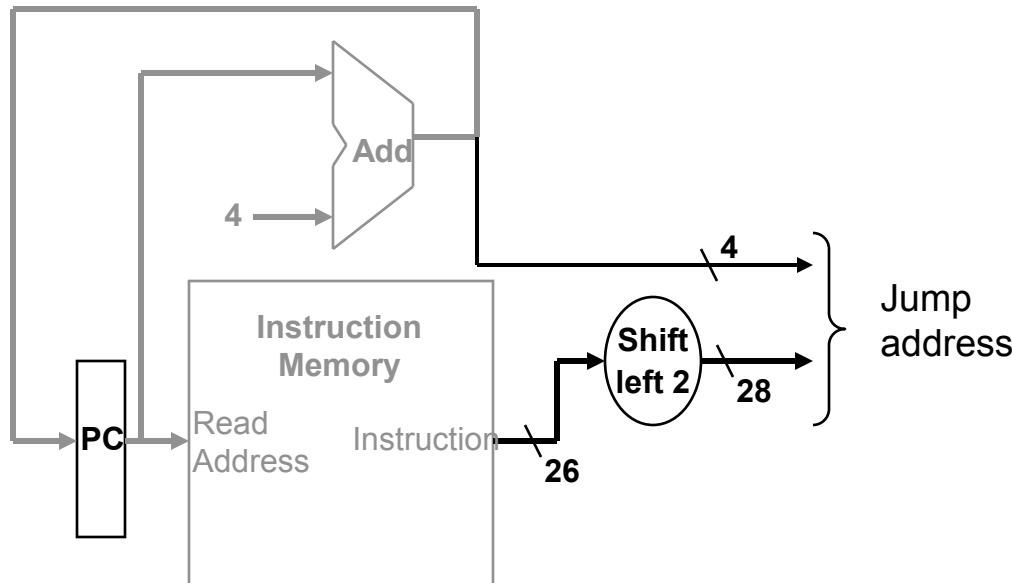
- compare the operands read from the Register File during decode for equality (**zero** ALU output)
- compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr



Executing Jump Operations

❑ Jump operation involves

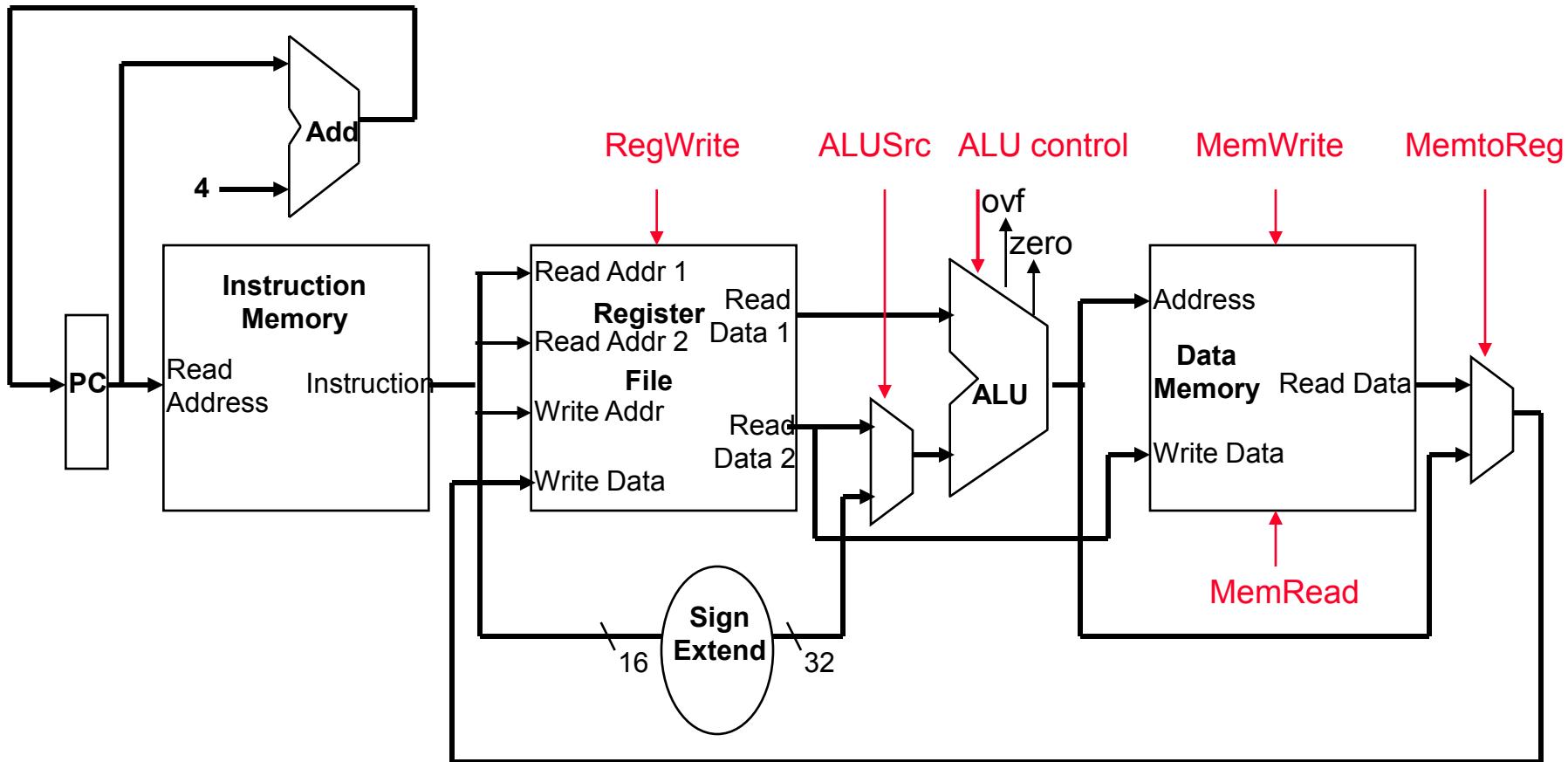
- replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits



Creating a Single Datapath from the Parts

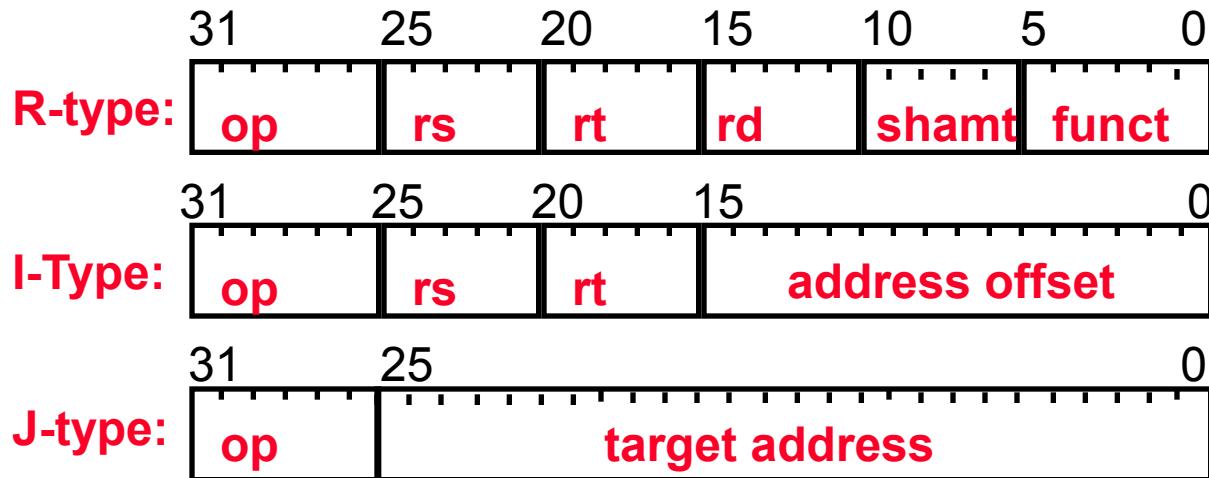
- ❑ Assemble the datapath segments and add control lines and multiplexors as needed
- ❑ **Single cycle** design – fetch, decode and execute each instructions in **one** clock cycle
 - no datapath resource can be used more than once per instruction, so some must be duplicated or ports added (e.g., separate Instruction Memory and Data Memory, several adders)
 - **multiplexors** needed at the input of shared elements with control lines to do the selection
 - write signals to control writing to the Register File and Data Memory
- ❑ Cycle time is determined by length of the longest path

Fetch, R, and Memory Access Portions



Adding the Control

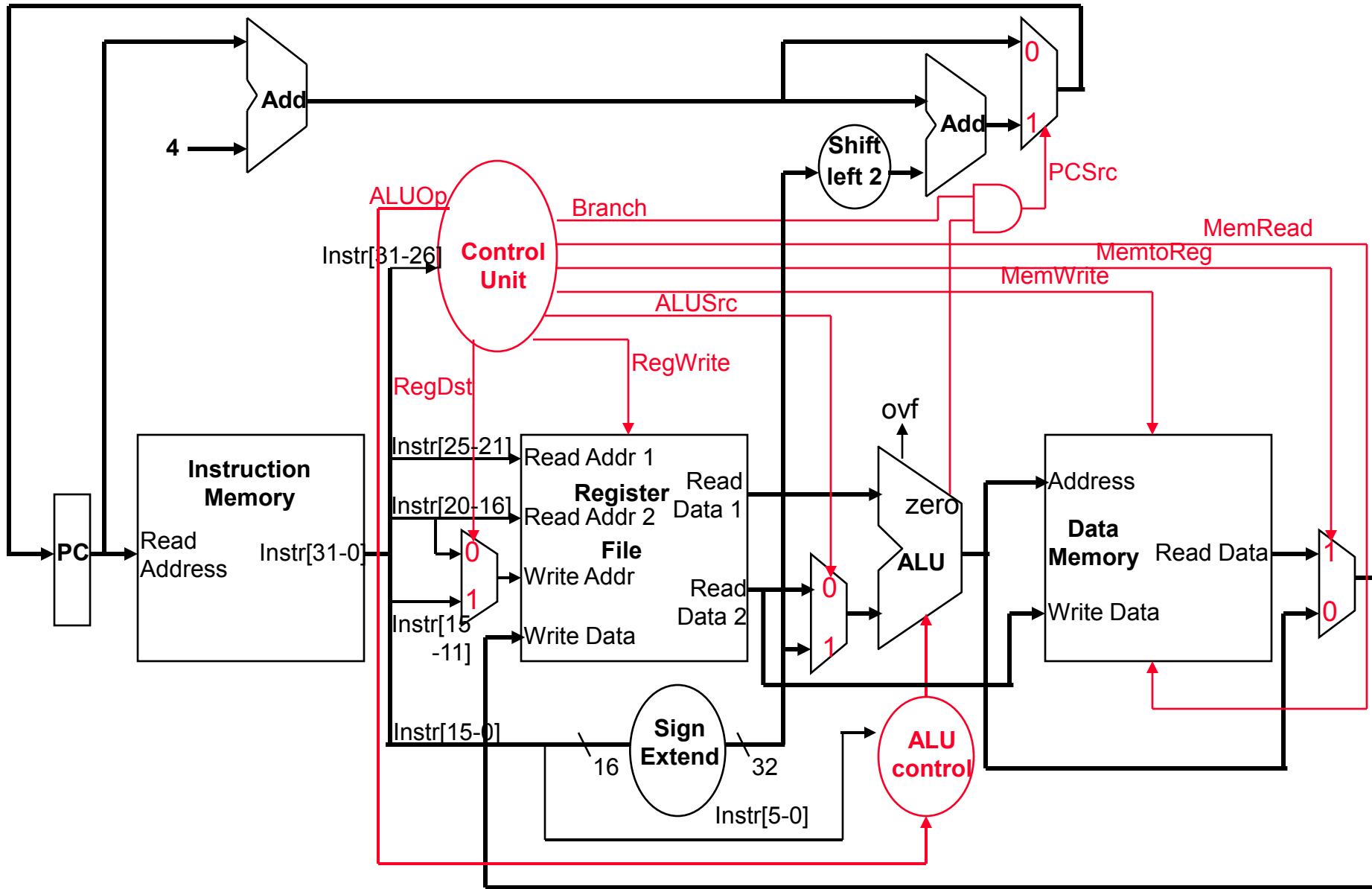
- ❑ Selecting the operations to perform (ALU, Register File and Memory read/write)
- ❑ Controlling the flow of data (multiplexor inputs)



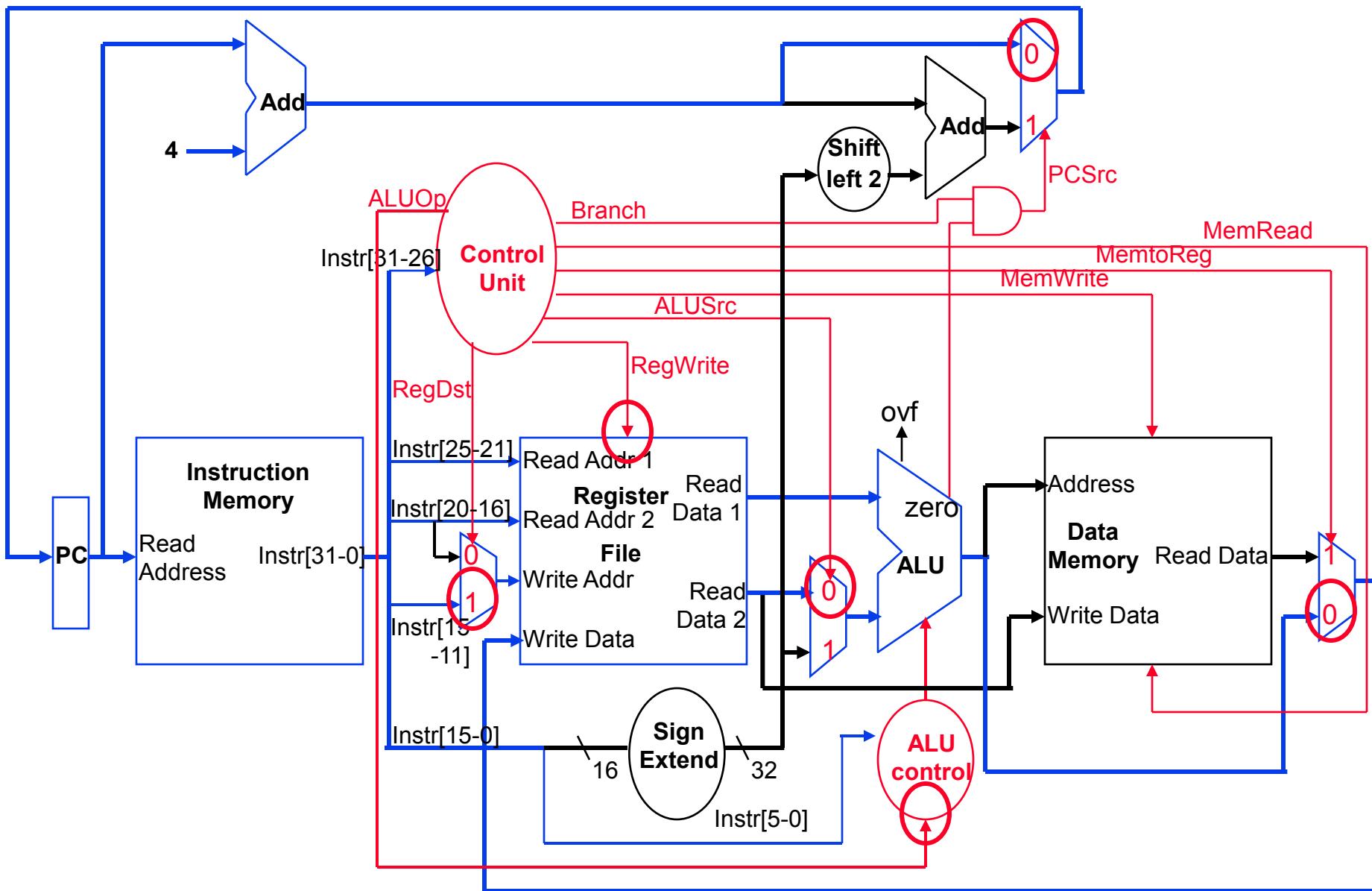
❑ Observations

- op field **always** in bits 31-26
- addr of registers to be read are **always** specified by the rs field (bits 25-21) and rt field (bits 20-16); for lw and sw rs is the base register
- addr. of register to be written is in one of **two** places – in rt (bits 20-16) for lw; in rd (bits 15-11) for R-type instructions
- offset for beq, lw, and sw **always** in bits 15-0

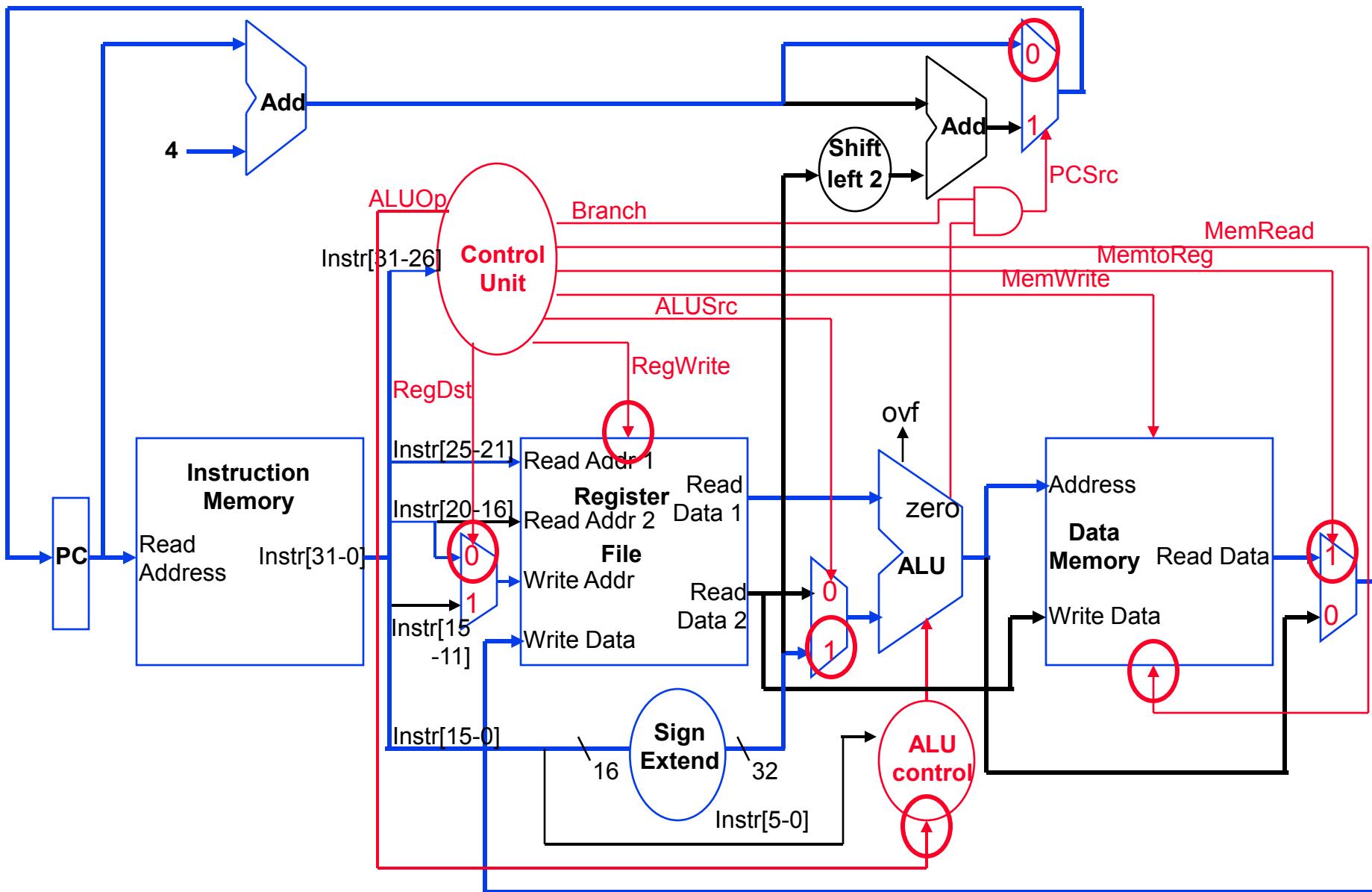
Single Cycle Datapath with Control Unit



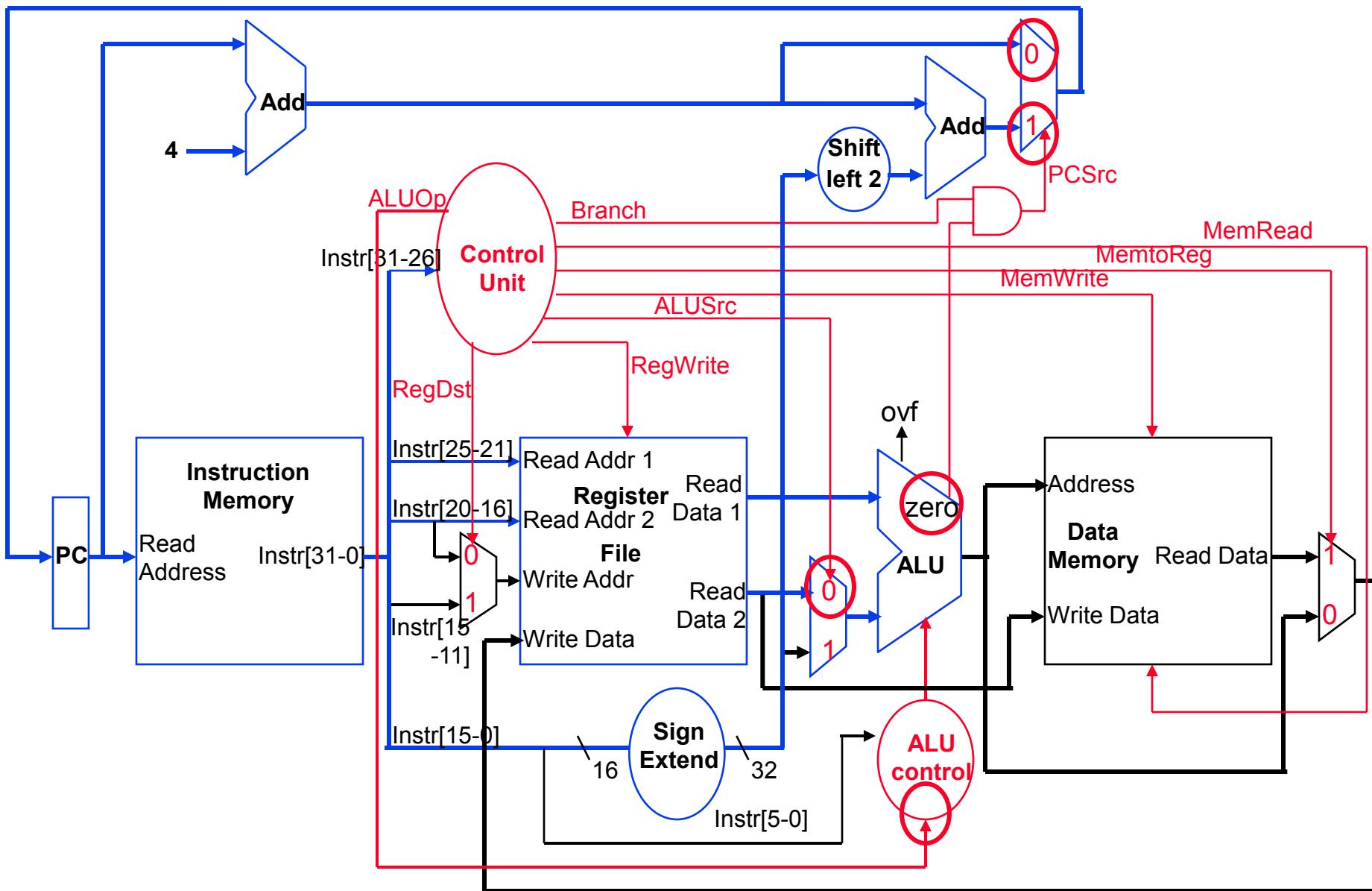
R-type Instruction Data/Control Flow



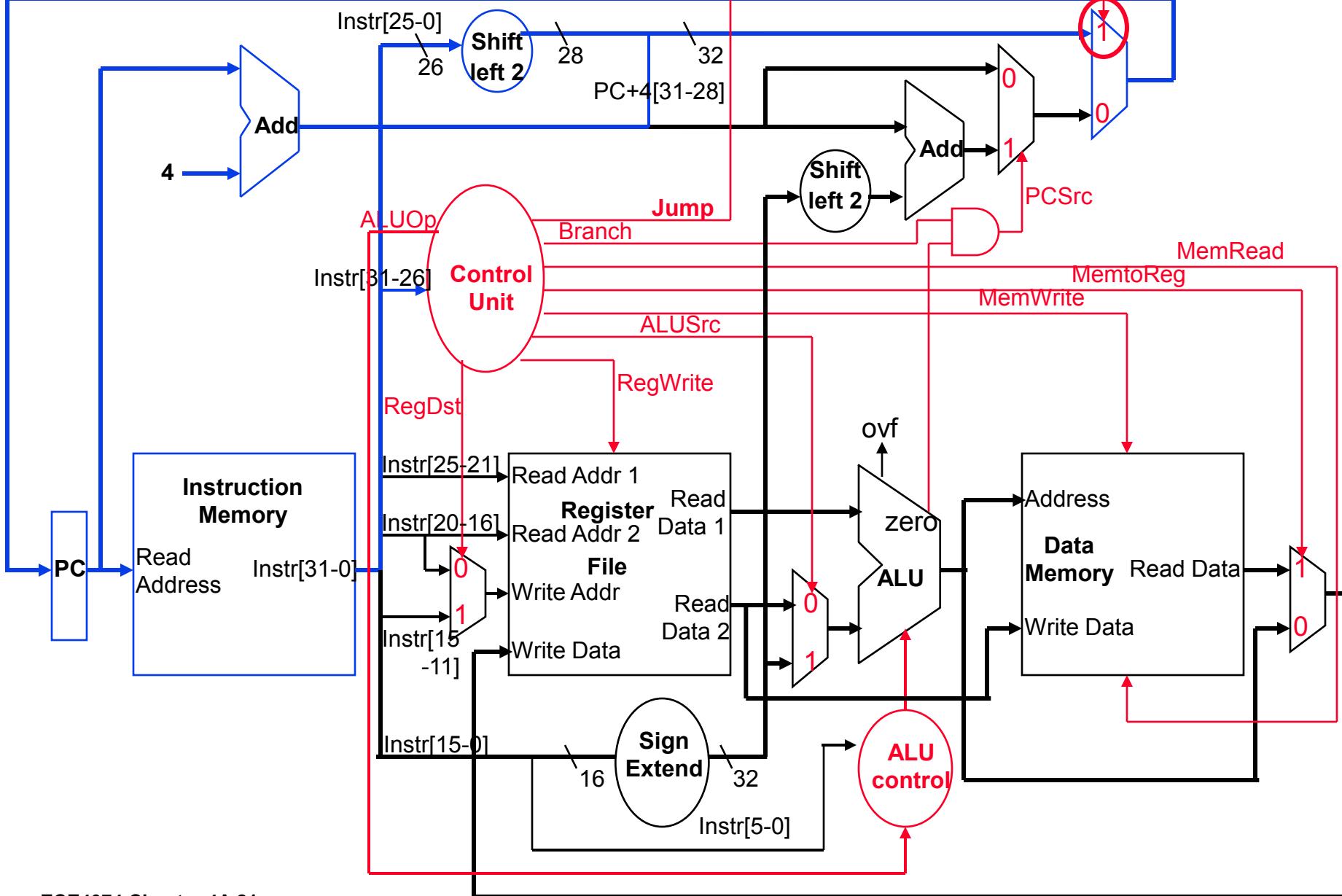
Load Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Adding the Jump Operation



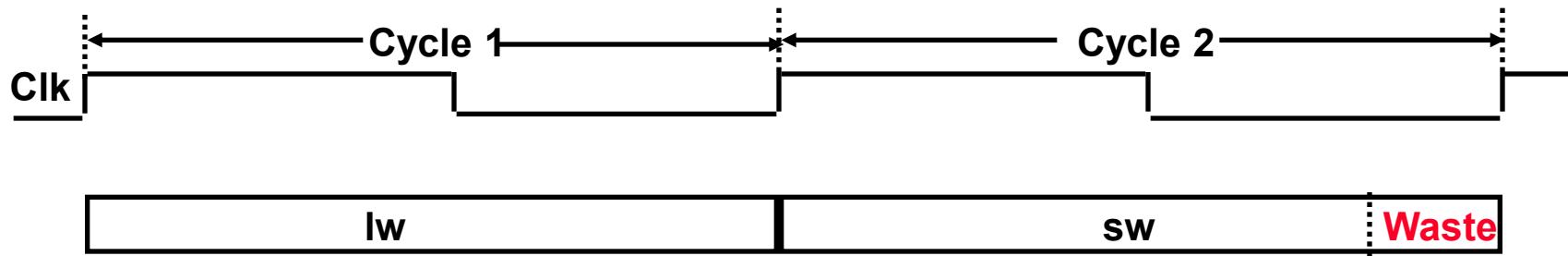
Instruction Critical Paths

- ❑ What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
 - Instruction and Data Memory (200 ps)
 - ALU and adders (200 ps)
 - Register File access (reads or writes) (100 ps)

| Instr. | I Mem | Reg Rd | ALU Op | D Mem | Reg Wr | Total |
|--------|-------|--------|--------|-------|--------|-------|
| R-type | 200 | 100 | 200 | | 100 | 600 |
| load | 200 | 100 | 200 | 200 | 100 | 800 |
| store | 200 | 100 | 200 | 200 | | 700 |
| beq | 200 | 100 | 200 | | | 500 |
| jump | 200 | | | | | 200 |

Single Cycle Disadvantages & Advantages

- ❑ Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instruction
 - especially problematic for more complex instructions like floating point divide



- ❑ May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- ❑ Is simple and easy to understand

How Can We Make It Faster?

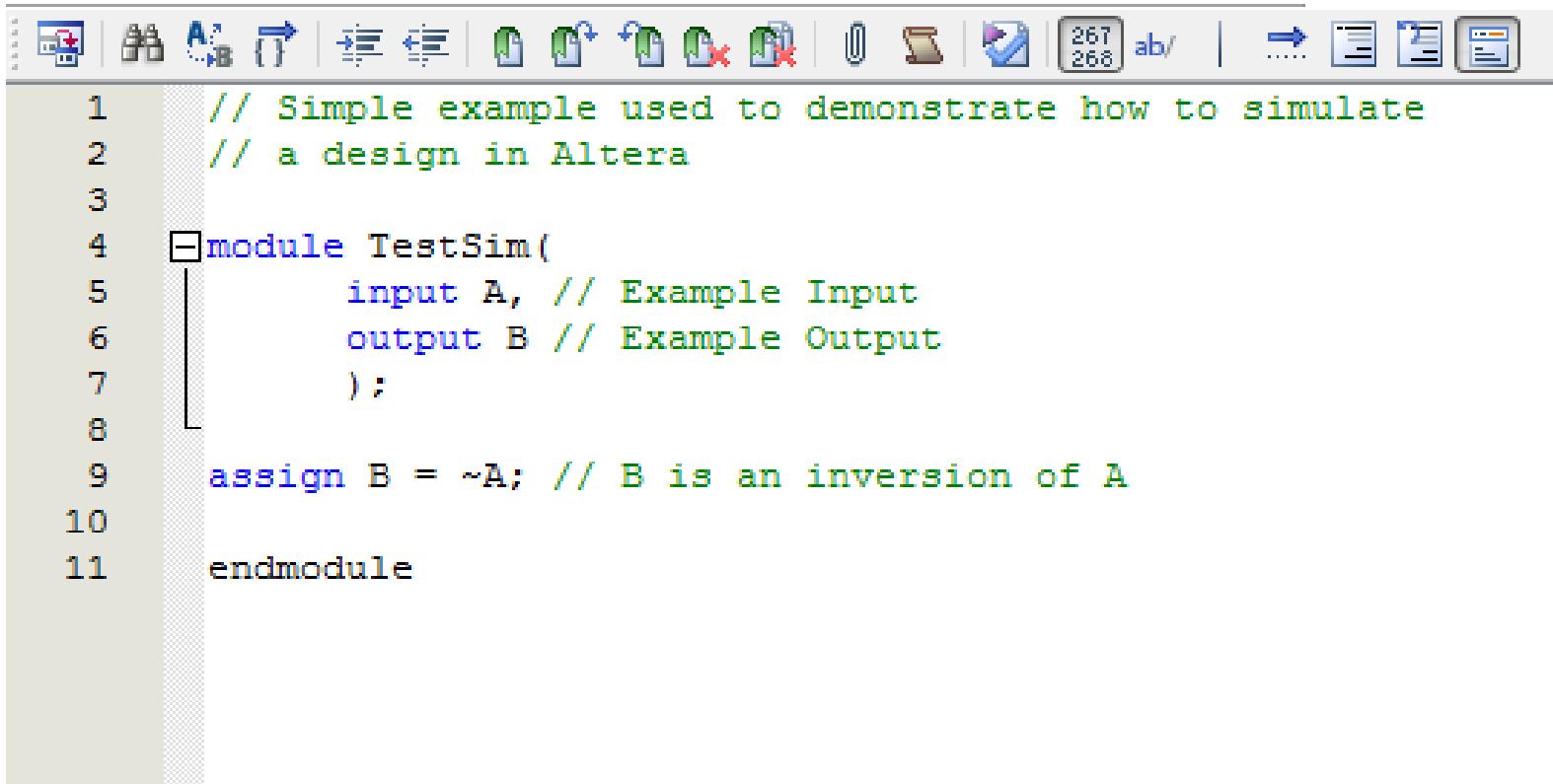
- ❑ Start fetching and executing the next instruction before the current one has completed
 - Pipelining – (all?) modern processors are pipelined for performance
 - Remember *the* performance equation:
$$\text{CPU time} = \text{CPI} * \text{CC} * \text{IC}$$
- ❑ Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
 - A five stage pipeline is nearly five times faster because the CC is nearly five times faster
- ❑ Fetch (and execute) more than one instruction at a time
 - Superscalar processing – stay tuned

End Lecture

- ❑ This week's lab:
 - Writing Verilog description of IF stage
 - Demonstrate that you understand how to simulate your design
 - Show that you can load an instruction into memory
- ❑ Following Slides on how to simulate with ModelSim

Simulating In Altera (via Model Sim)

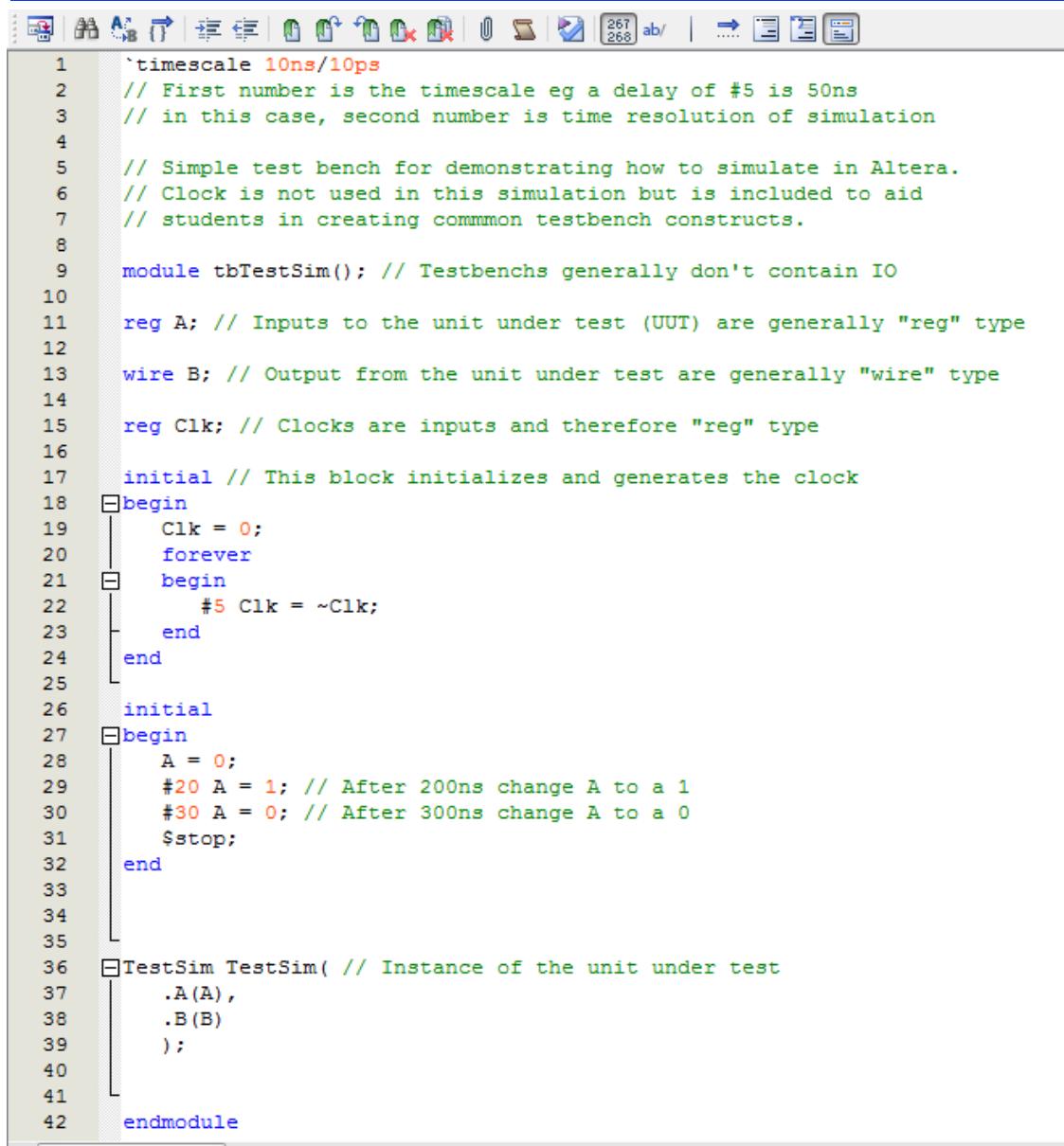
- ❑ Create your project and module for simulation



The screenshot shows a software interface with a toolbar at the top and a code editor window below. The code editor displays the following Verilog code:

```
1 // Simple example used to demonstrate how to simulate
2 // a design in Altera
3
4 module TestSim(
5     input A, // Example Input
6     output B // Example Output
7 );
8
9 assign B = ~A; // B is an inversion of A
10
11 endmodule
```

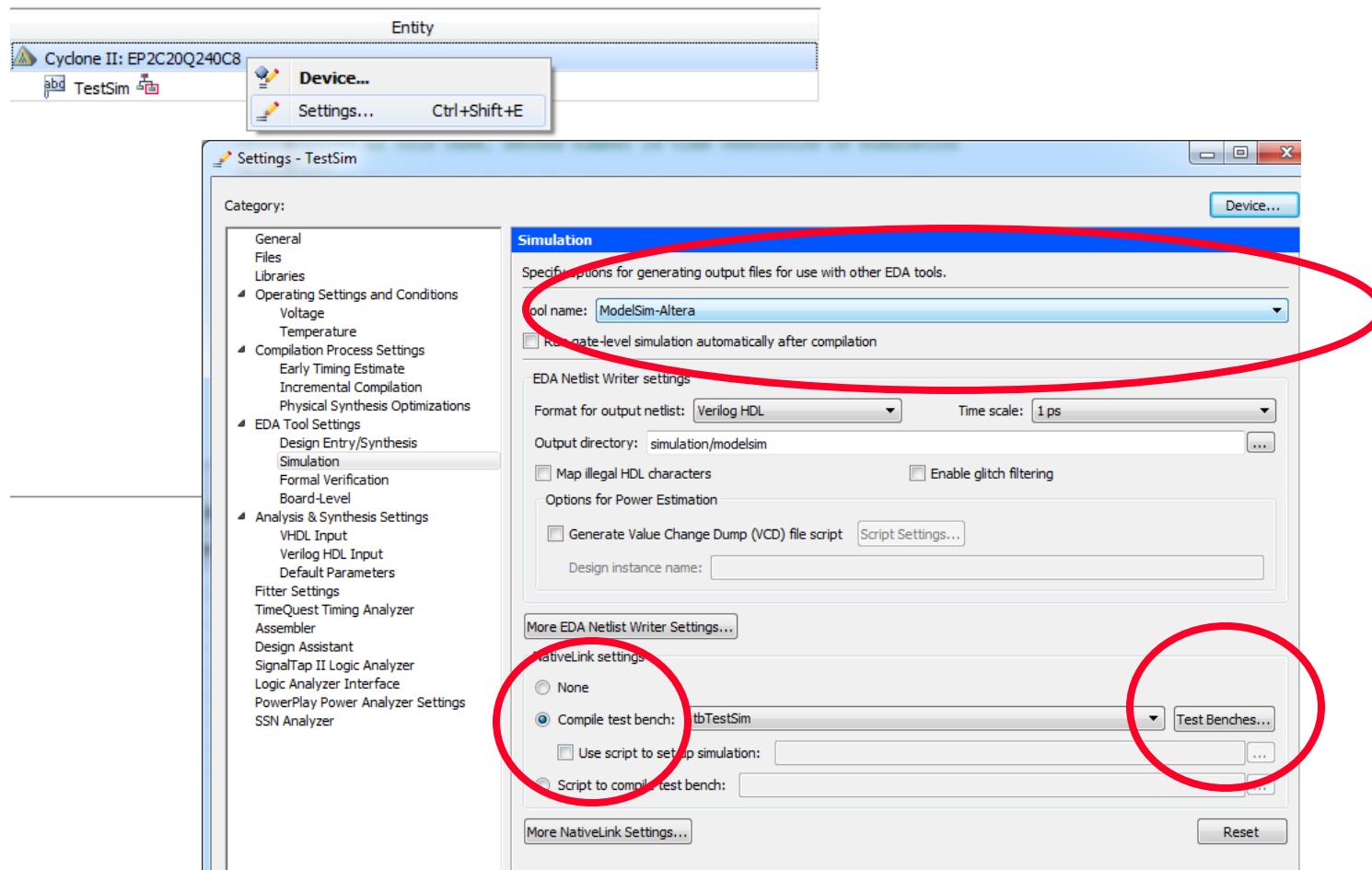
Create your test bench



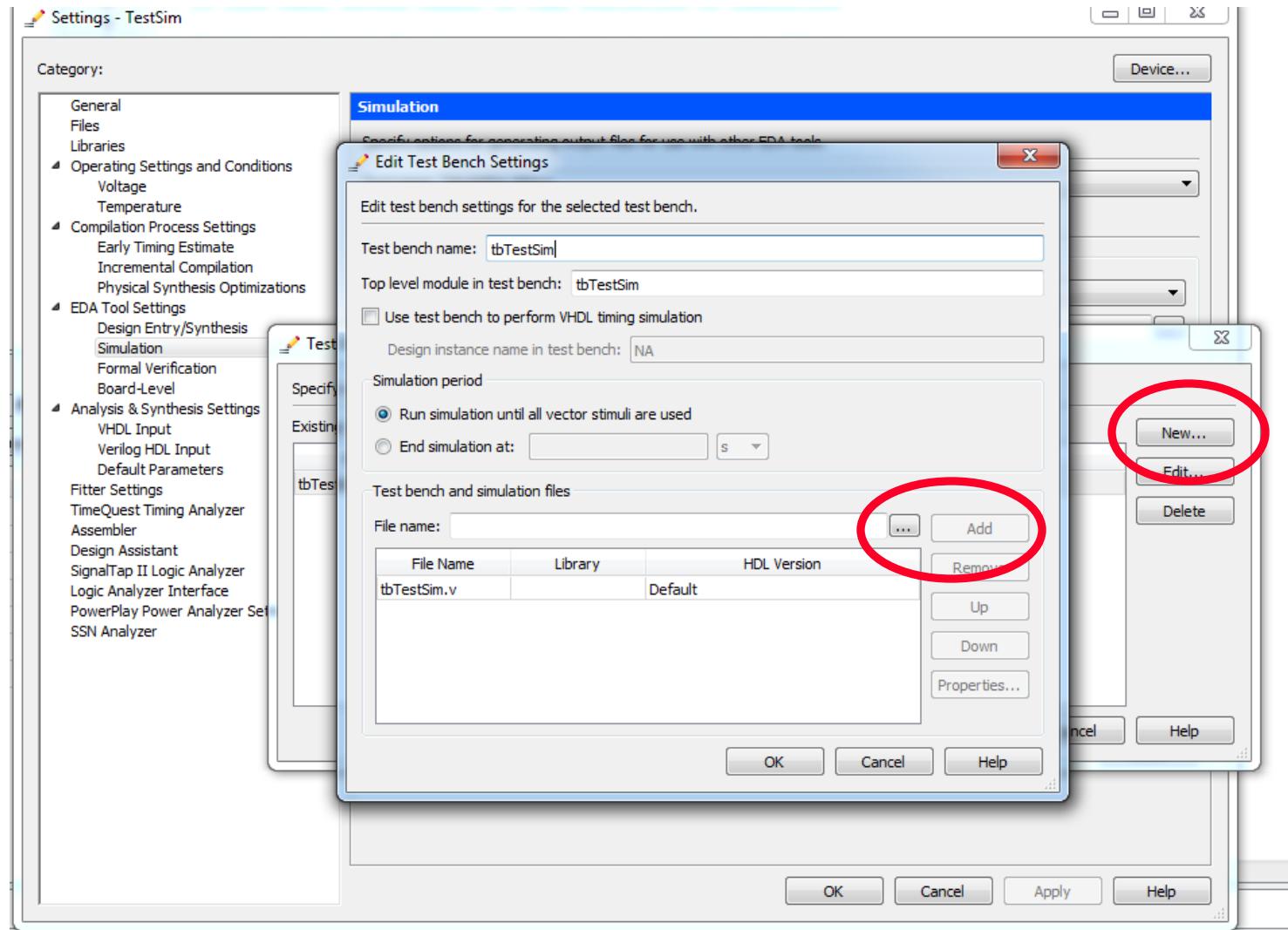
The screenshot shows a software interface with a toolbar at the top and a code editor window below. The code editor contains Verilog testbench code. The code defines a module tbTestSim() containing logic for generating a clock signal and driving an instance of a unit under test (TestSim). The code uses various Verilog constructs like `timescale, reg, wire, initial, begin-end blocks, and sequence points like #5 and \$stop.

```
1 `timescale 10ns/10ps
2 // First number is the timescale eg a delay of #5 is 50ns
3 // in this case, second number is time resolution of simulation
4
5 // Simple test bench for demonstrating how to simulate in Altera.
6 // Clock is not used in this simulation but is included to aid
7 // students in creating common testbench constructs.
8
9 module tbTestSim(); // Testbenches generally don't contain IO
10
11 reg A; // Inputs to the unit under test (UUT) are generally "reg" type
12
13 wire B; // Output from the unit under test are generally "wire" type
14
15 reg Clk; // Clocks are inputs and therefore "reg" type
16
17 initial // This block initializes and generates the clock
18 begin
19     Clk = 0;
20     forever
21     begin
22         #5 Clk = ~Clk;
23     end
24 end
25
26 initial
27 begin
28     A = 0;
29     #20 A = 1; // After 200ns change A to a 1
30     #30 A = 0; // After 300ns change A to a 0
31     $stop;
32 end
33
34
35
36 TestSim TestSim( // Instance of the unit under test
37     .A(A),
38     .B(B)
39 );
40
41
42 endmodule
```

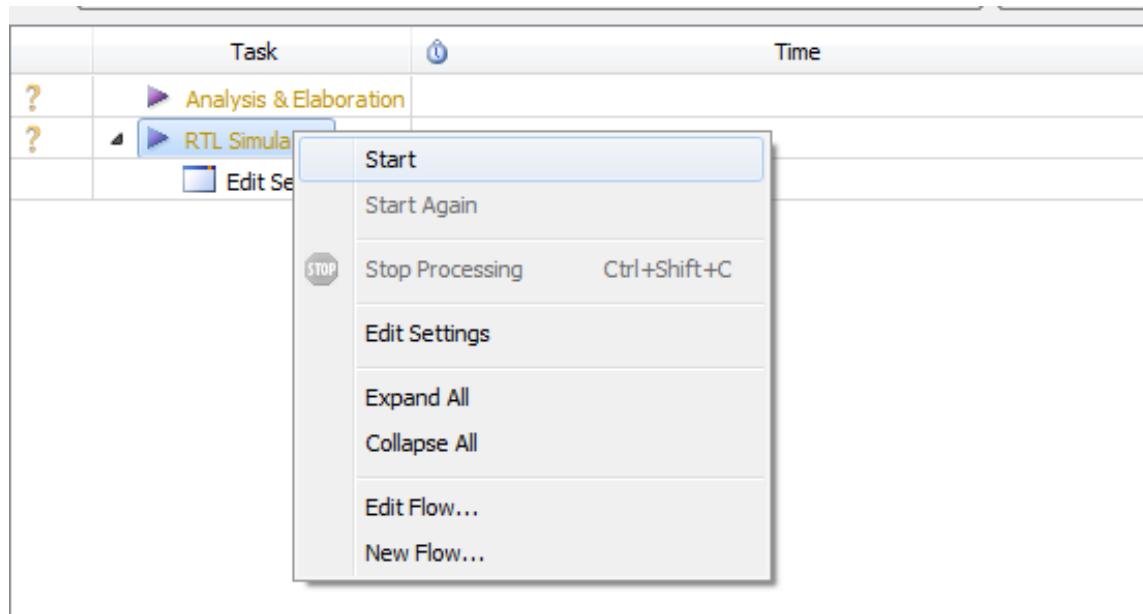
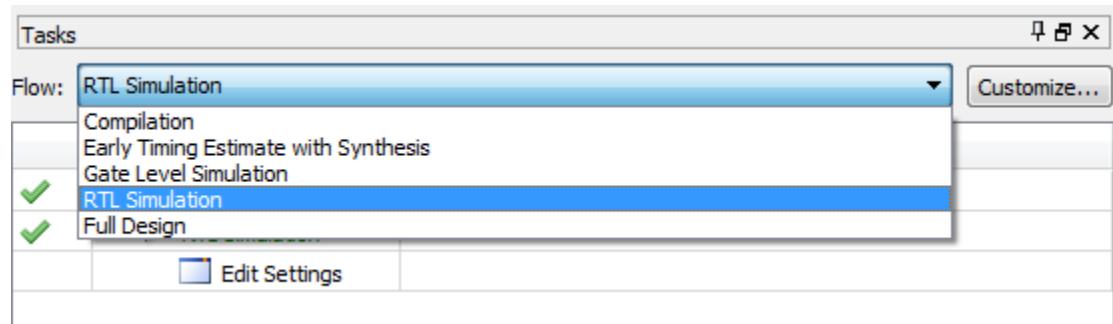
Set up your simulator



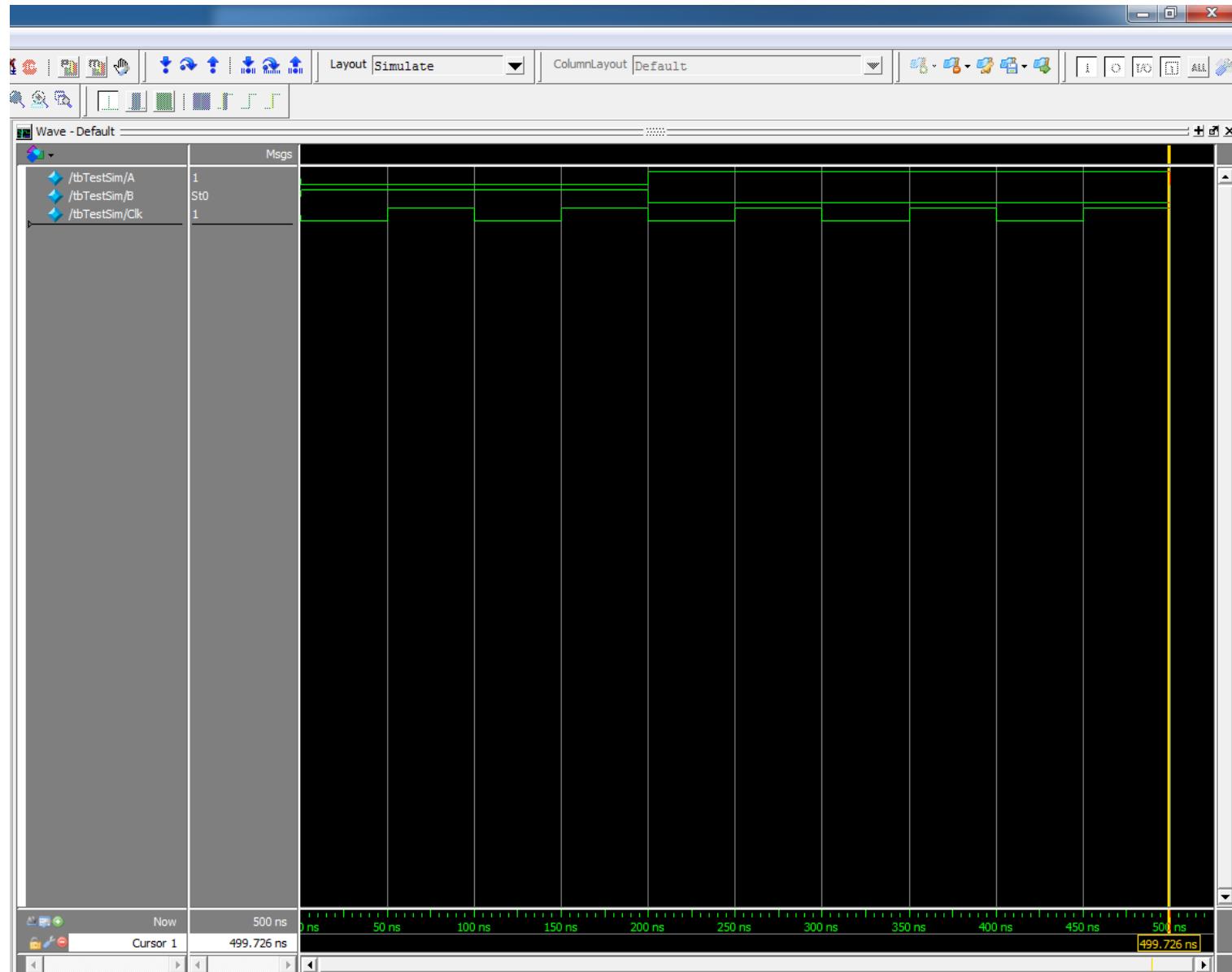
Point the simulator to the test bench



Switch to Simulation Flow and Run



View the output



Simulating your assignment

- ❑ Simulate each module often
 - Test all possible “branches” of your code
- ❑ Your final design wont need a complicated test bench as it is essentially just the clock.
- ❑ “Gate level simulation” includes timing delays inherent to your design.

ECE4074
Advanced Computer Architecture
Semester 2 2014

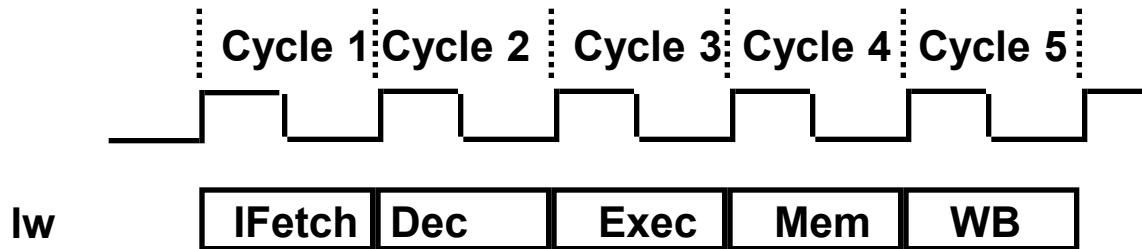
**Chapter 4A: The Processor,
Part A II**

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

How Can We Make It Faster?

- ❑ Start fetching and executing the next instruction before the current one has completed
 - Pipelining – (all?) modern processors are pipelined for performance
 - Remember *the* performance equation:
$$\text{CPU time} = \text{CPI} * \text{CC} * \text{IC}$$
- ❑ Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
 - A five stage pipeline is nearly five times faster because the CC is nearly five times faster (minus setup and hold delays)
- ❑ Fetch (and execute) more than one instruction at a time
 - Superscalar processing – stay tuned

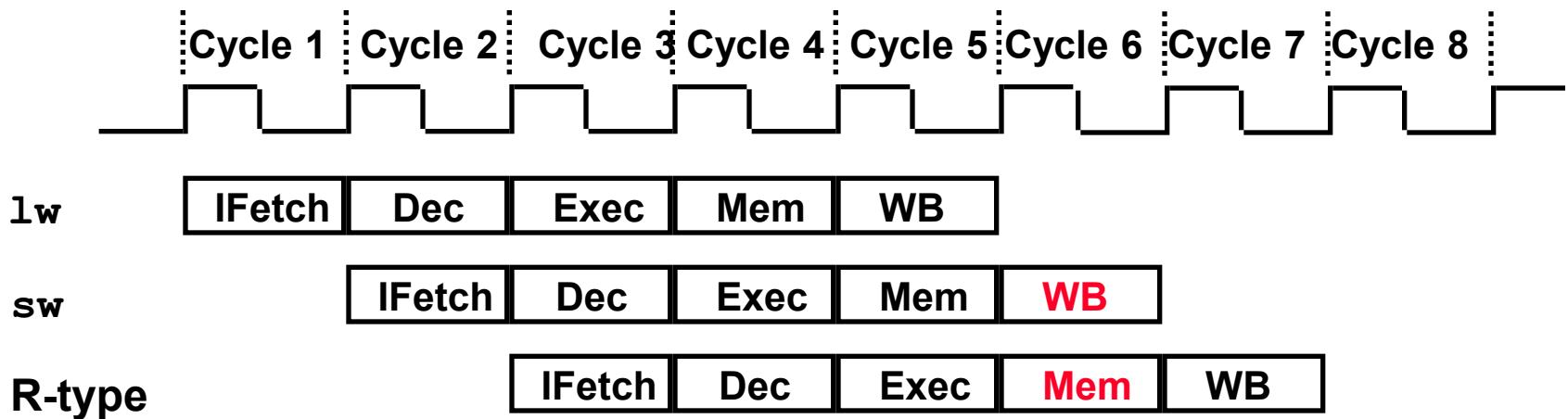
The Five Stages of Load Instruction



- ❑ IFetch: Instruction Fetch and Update PC
- ❑ Dec: Registers Fetch and Instruction Decode
- ❑ Exec: Execute R-type; calculate memory address
- ❑ Mem: Read/write the data from/to the Data Memory
- ❑ WB: Write the result data into the register file

A Pipelined MIPS Processor

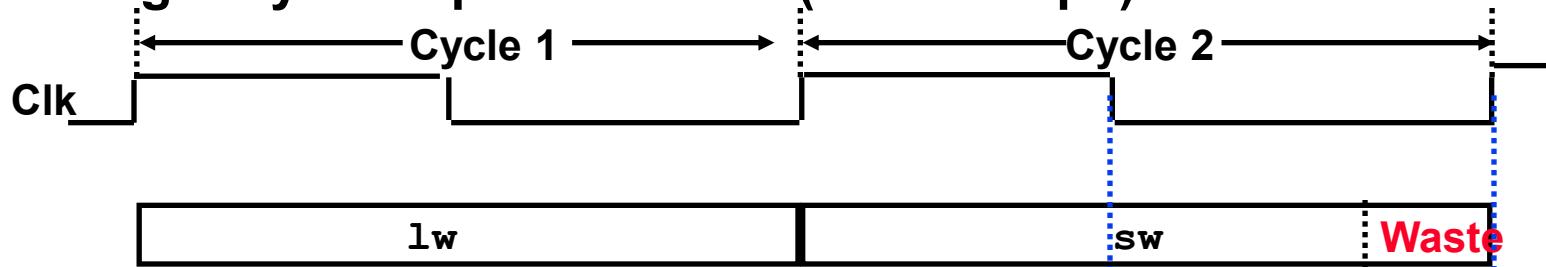
- Start the **next** instruction before the current one has completed
 - improves **throughput** - total amount of work done in a given time
 - instruction **latency** (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced. In practice, **latency** is increased.



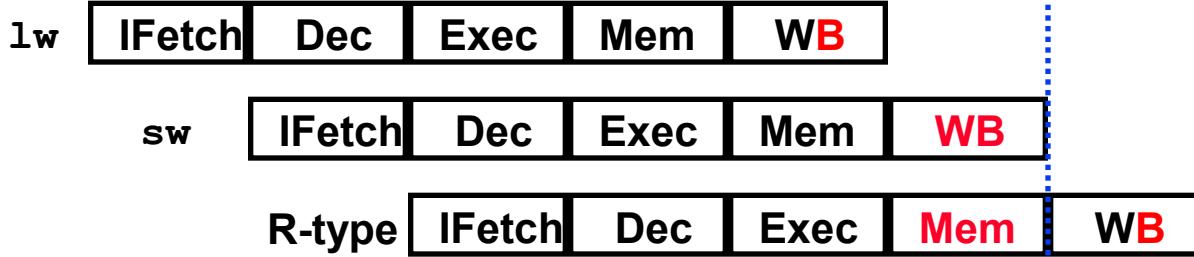
- clock cycle (pipeline stage time) is limited by the **slowest** stage
 - for some stages don't need the whole clock cycle (e.g., WB)
 - for some instructions, some stages are **wasted** cycles (i.e., nothing is done during that cycle for that instruction)

Single Cycle versus Pipeline

Single Cycle Implementation (CC = 800 ps):



Pipeline Implementation (CC = 200 ps):



- ❑ To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- ❑ How long does each take to complete 1,000,000 adds ?

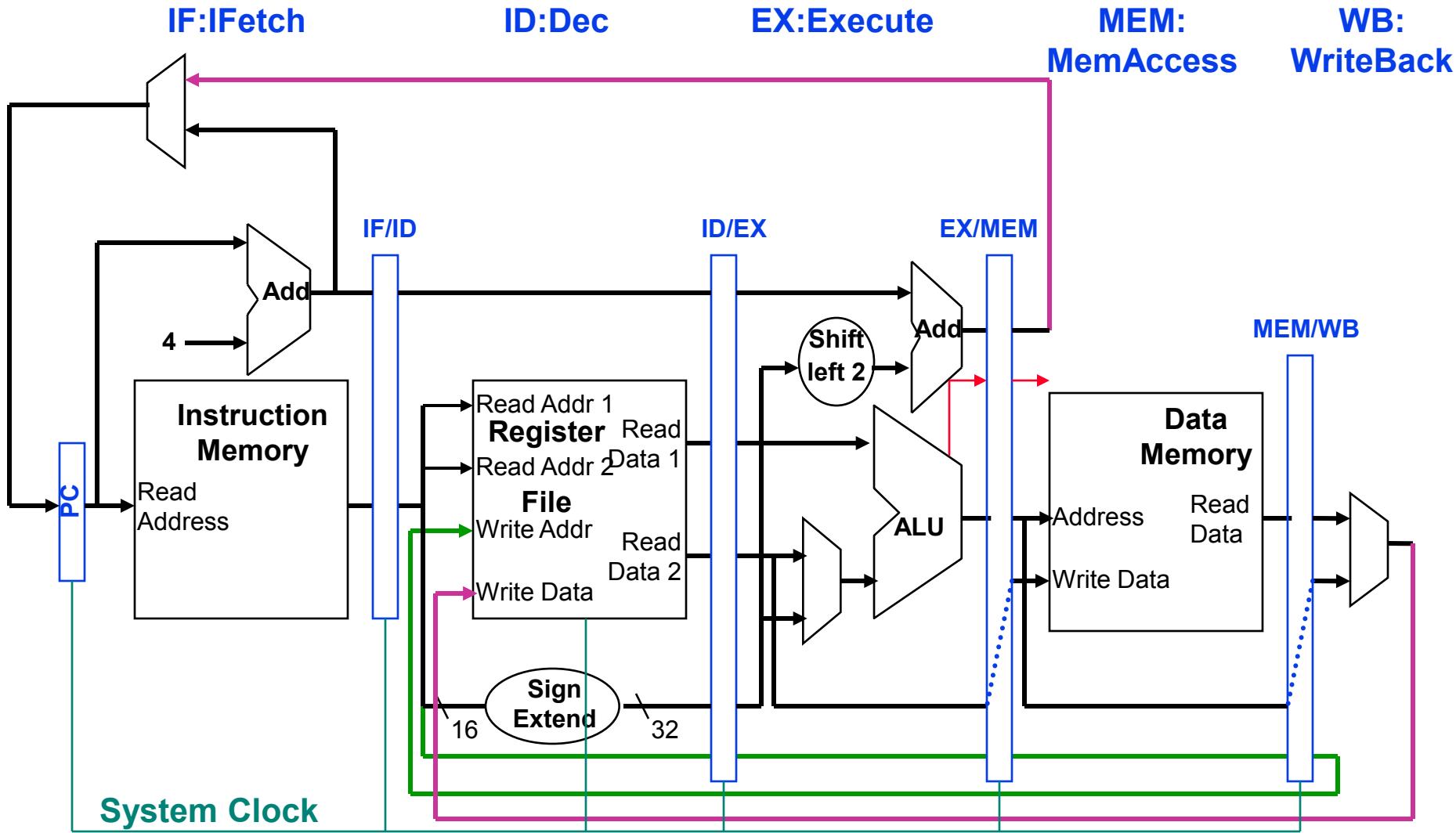
Pipelining the MIPS ISA

❑ What makes it easy

- all instructions are the same length (32 bits)
 - can fetch in the 1st stage and decode in the 2nd stage
- few instruction formats (three) with **symmetry** across formats
 - can begin reading register file in 2nd stage
- memory operations occur only in loads and stores
 - can use the execute stage to calculate memory addresses
- each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
- operands must be aligned in memory so a single data transfer takes only one data memory access

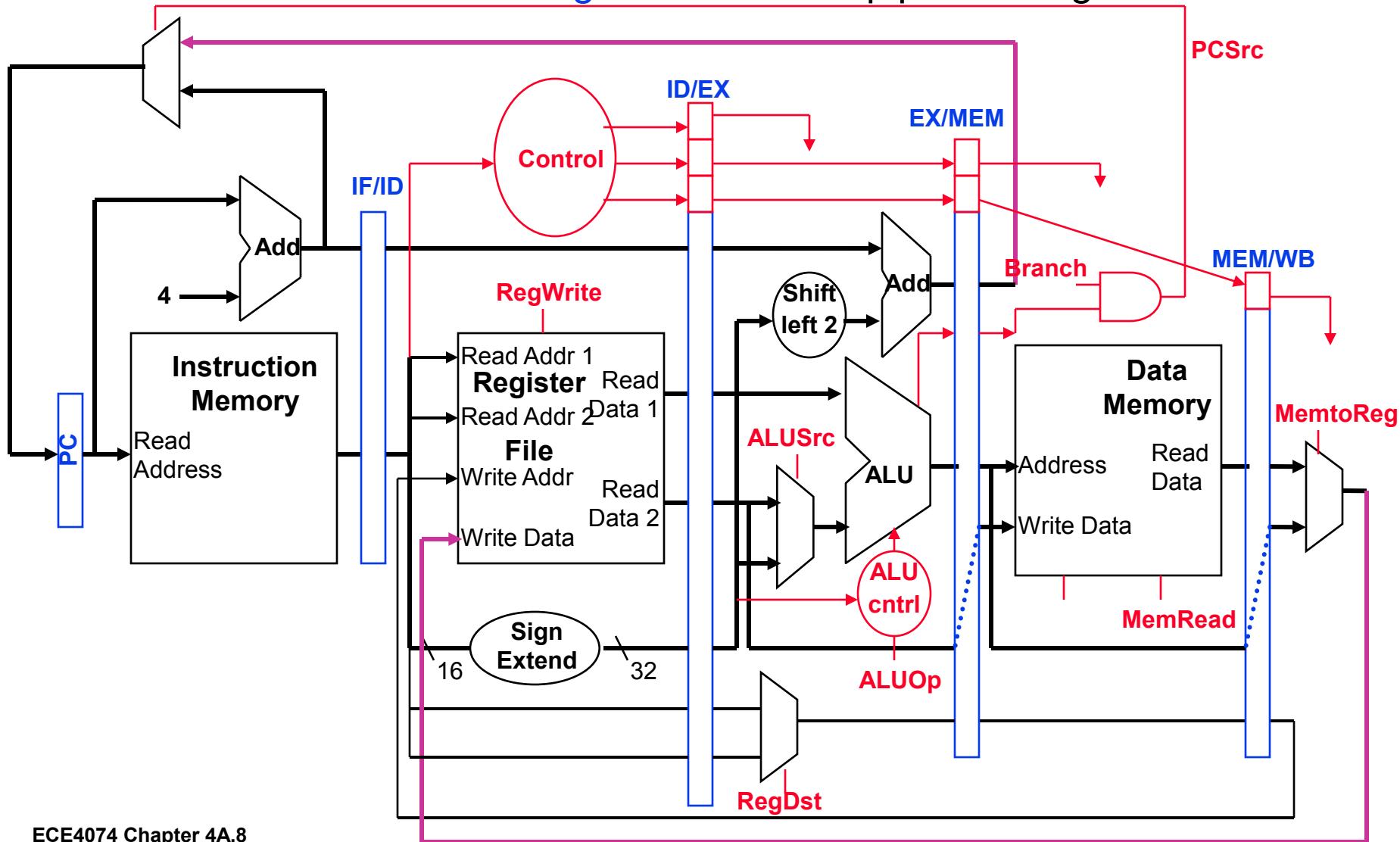
MIPS Pipeline Datapath Additions/Mods

- State registers between each pipeline stage to isolate them



MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
 - and held in the state registers between pipeline stages

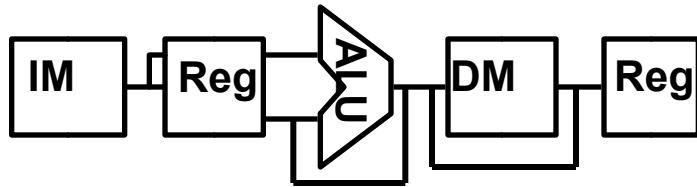


Pipeline Control

- ❑ IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)
- ❑ ID Stage: no control signals to set

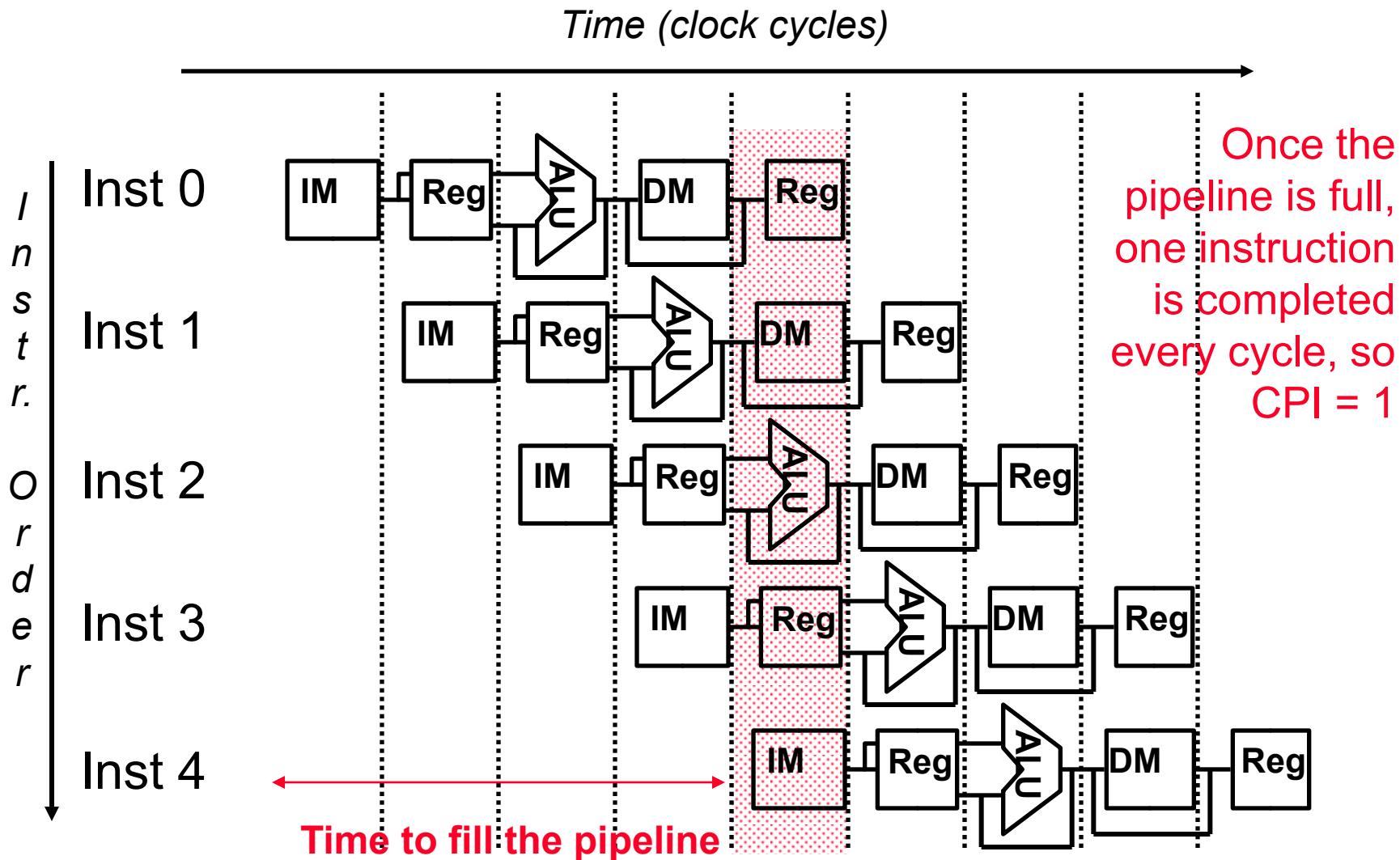
| | EX Stage | | | | MEM Stage | | | WB Stage | |
|-----|----------|---------|---------|---------|-----------|----------|-----------|-----------|-----------|
| | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Brch | Mem Read | Mem Write | Reg Write | Mem toReg |
| R | 1 | 1 | 0 | 0 | 0 | 0 | 0 | ? | 0 |
| lw | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| sw | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| beq | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

Graphically Representing MIPS Pipeline



- ❑ Can help with answering questions like:
 - How many cycles does it take to execute this code?
 - What is the ALU doing during cycle 4?
 - Is there a hazard, why does it occur, and how can it be fixed?

Why Pipeline? For Performance!



Can Pipelining Get Us Into Trouble?

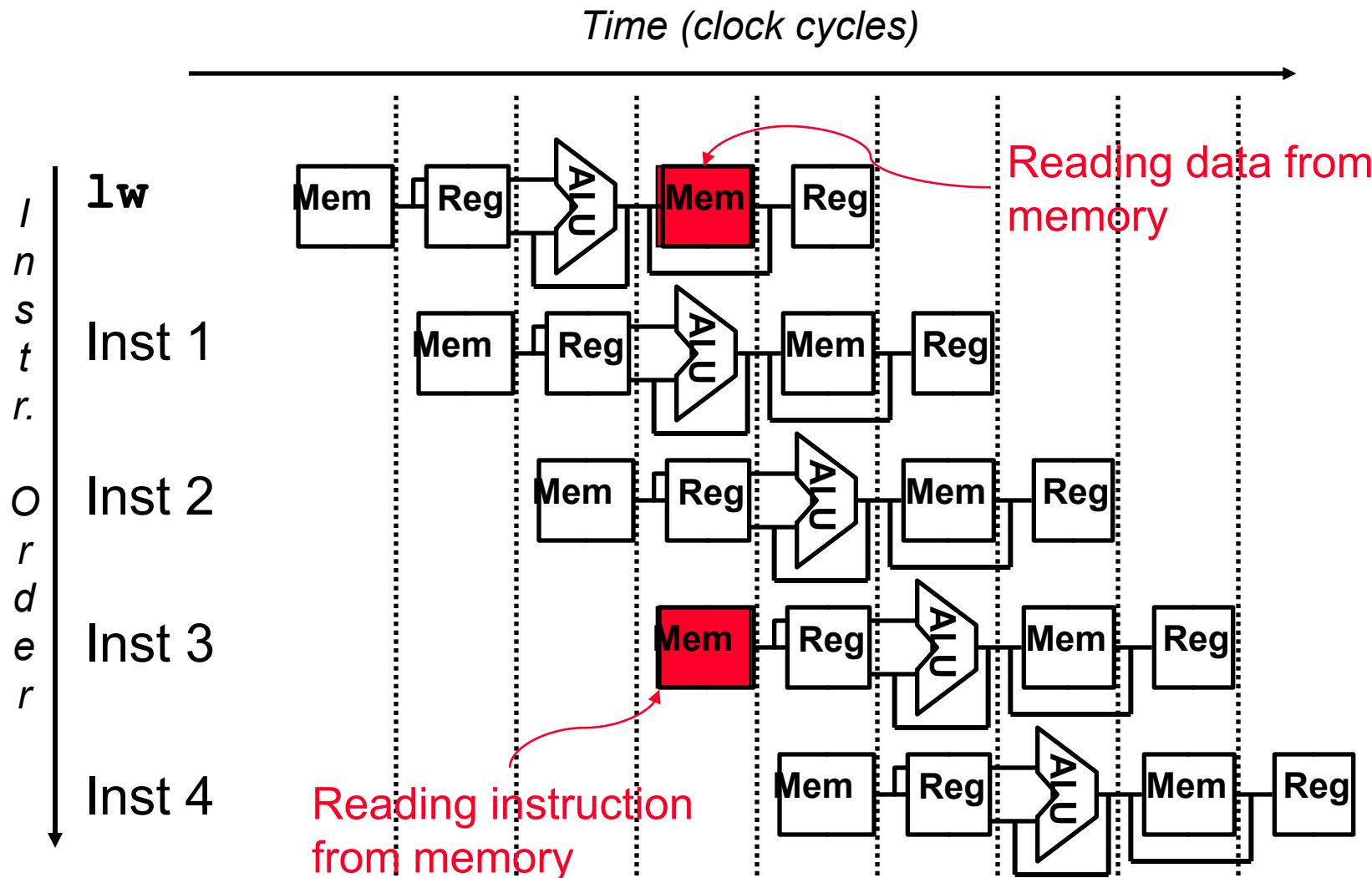
❑ Yes: Pipeline Hazards

- **structural hazards**: attempt to use the same resource by two different instructions at the same time
- **data hazards**: attempt to use data before it is ready
 - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - branch and jump instructions, exceptions

❑ Can usually resolve hazards by [waiting](#)

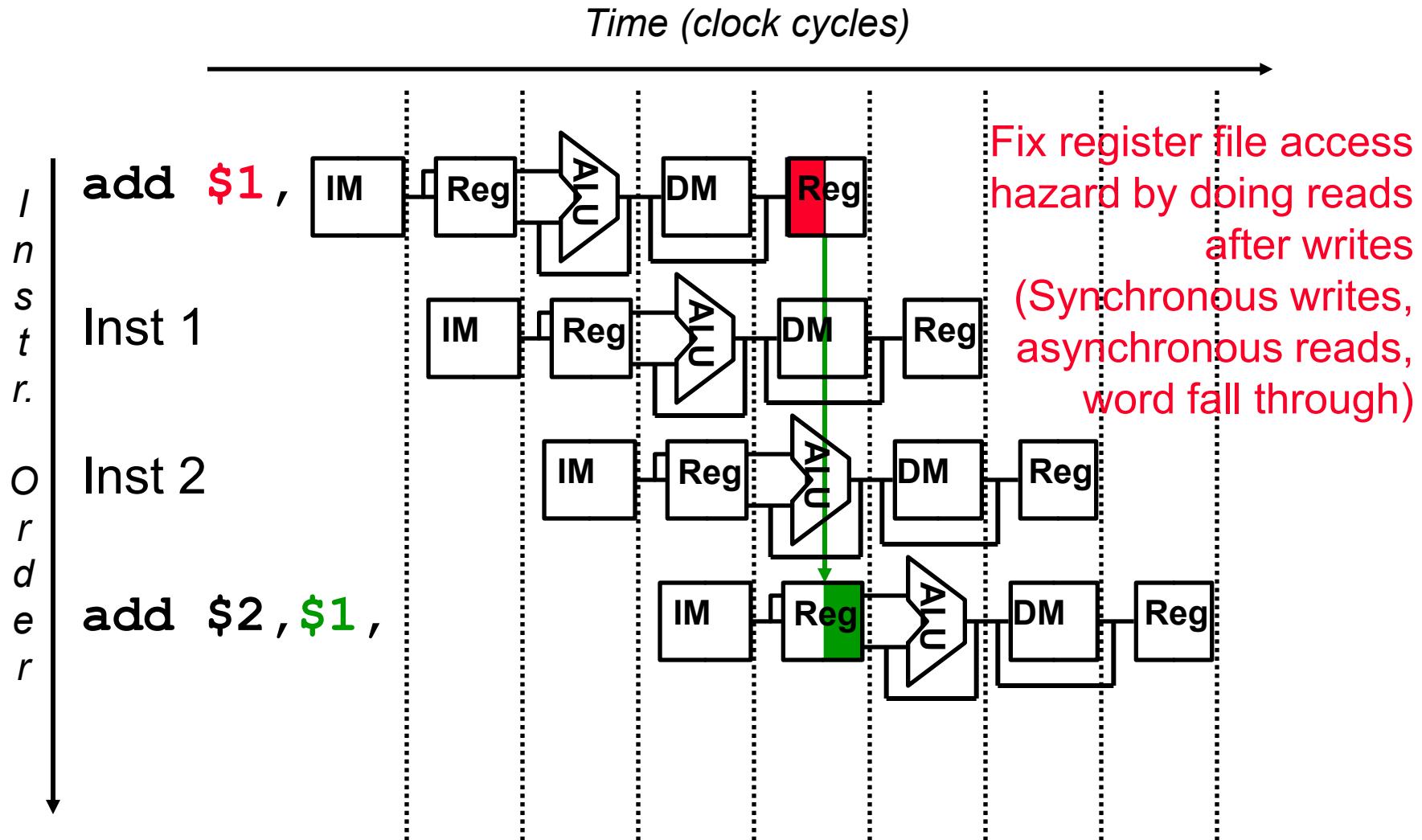
- pipeline control must **detect** the hazard
- and take action to **resolve** hazards

A Single Memory Would Be a Structural Hazard



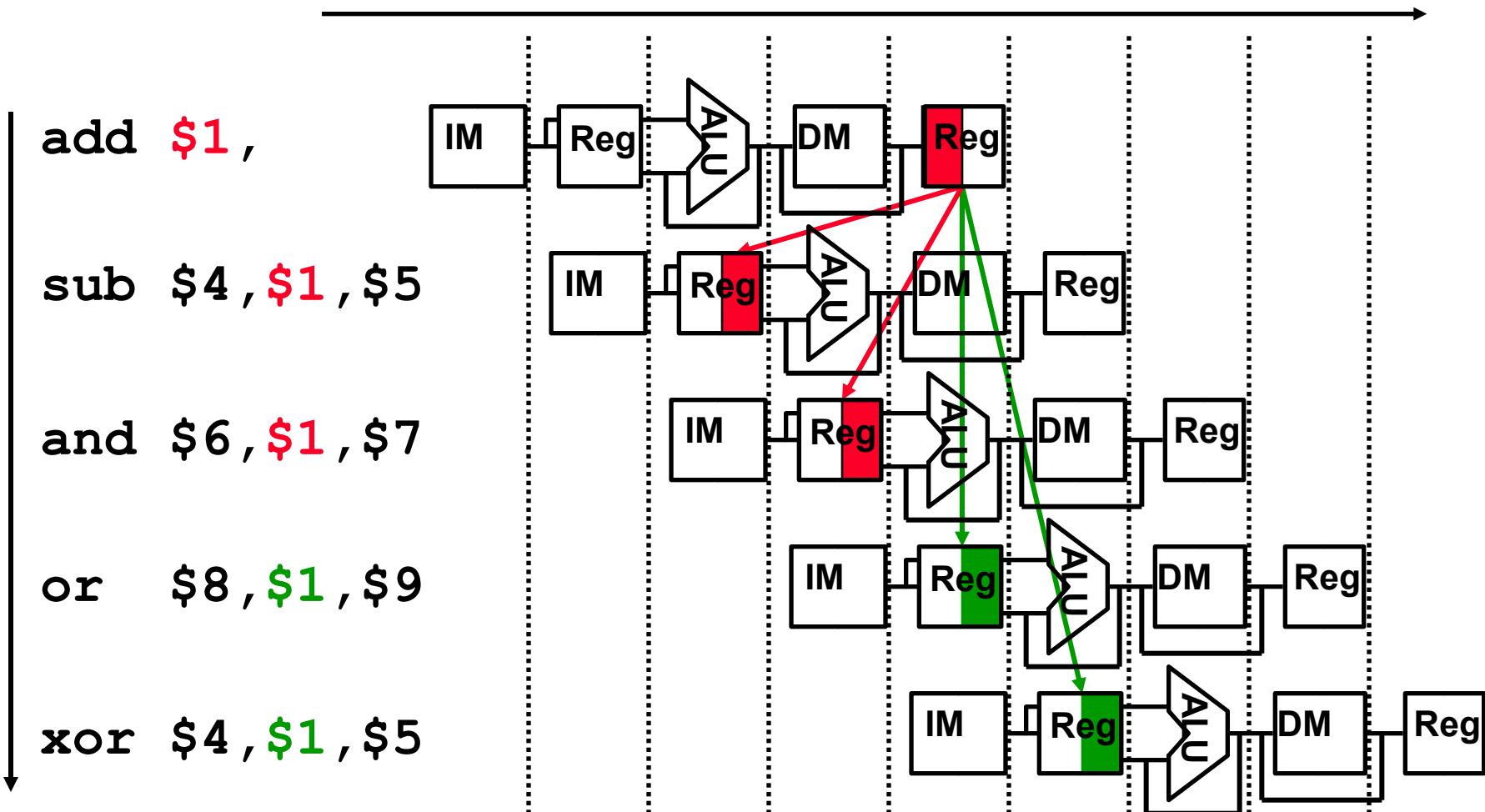
- ❑ Fix with separate (local) instr and data memories (I and D)

How About Register File Access?



Register Usage Can Cause Data Hazards

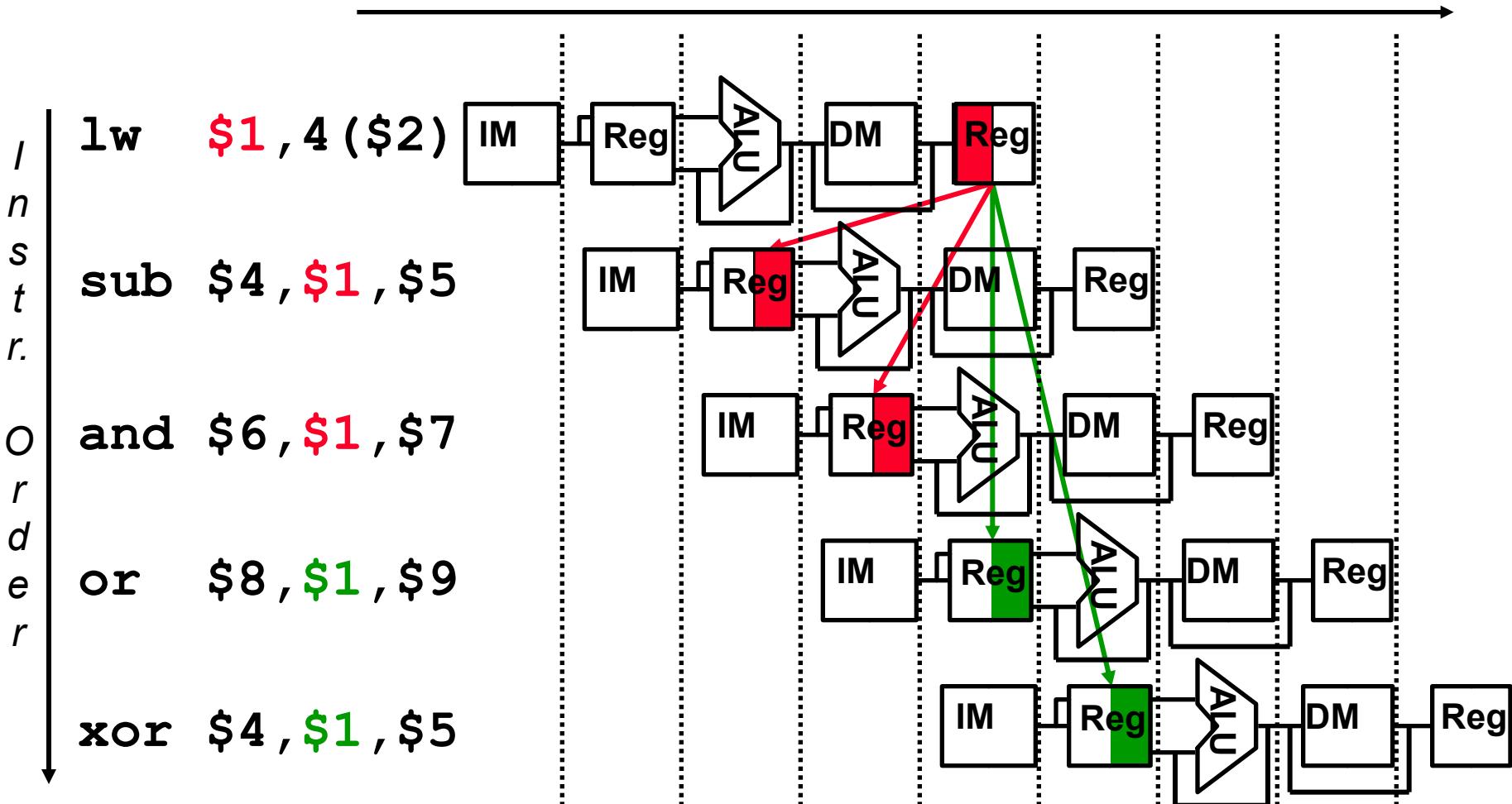
- Dependencies backward in time cause hazards



- Read before write data hazard

Loads Can Cause Data Hazards

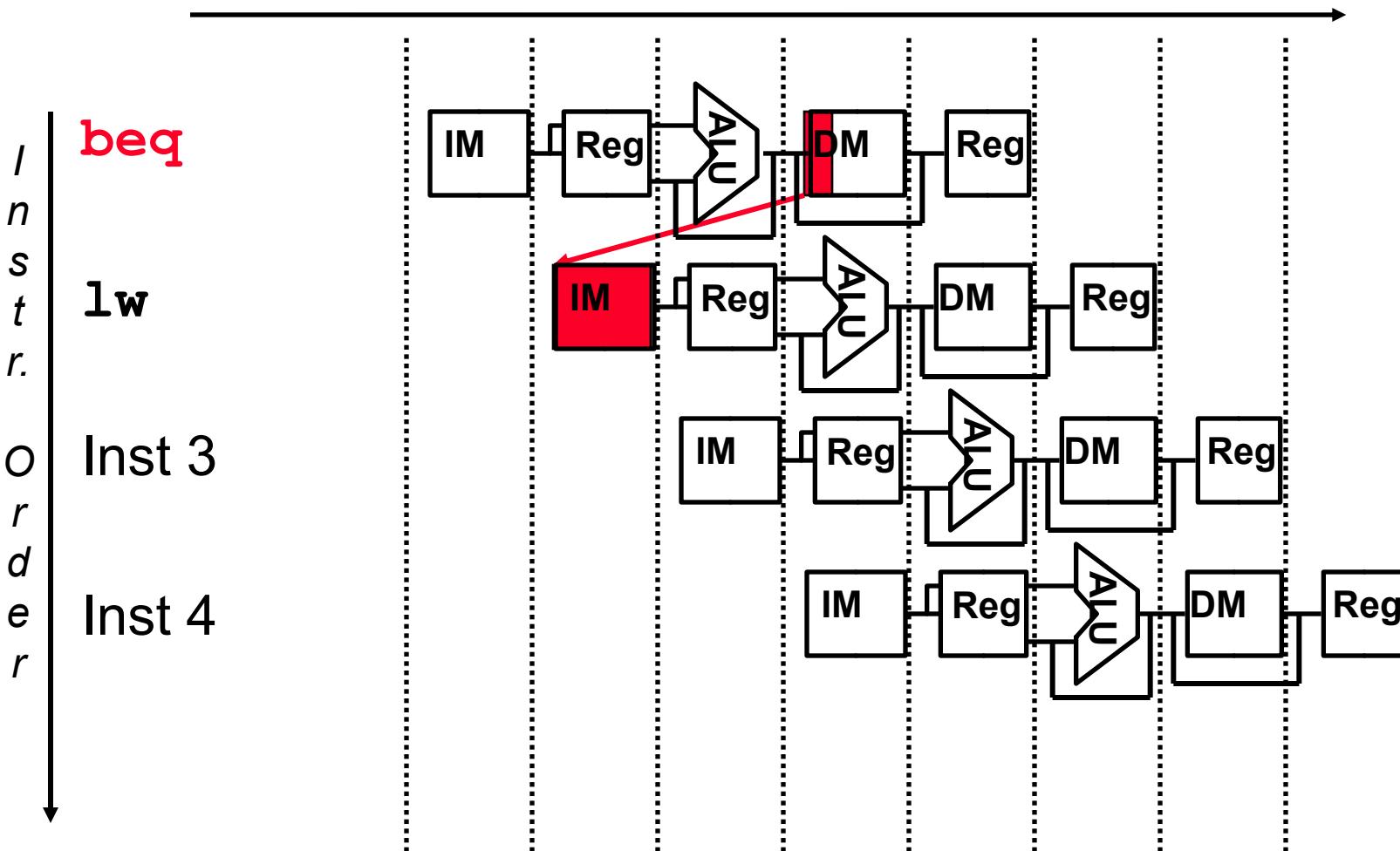
- Dependencies backward in time cause hazards



- Load-use data hazard

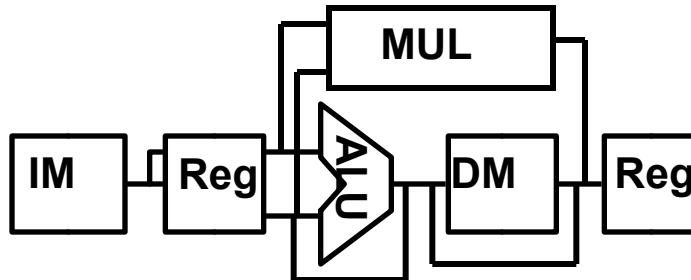
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards

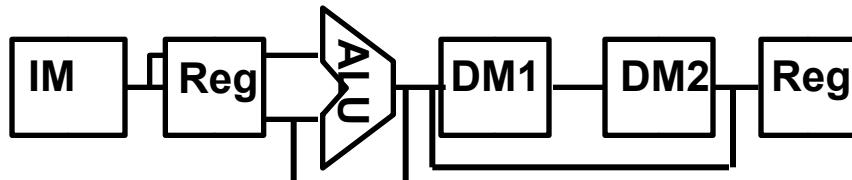


Other Pipeline Structures Are Possible

- ❑ What about the (slow) multiply operation?
 - Make the clock twice as slow or ...
 - let it take two cycles (since it doesn't use the DM stage)

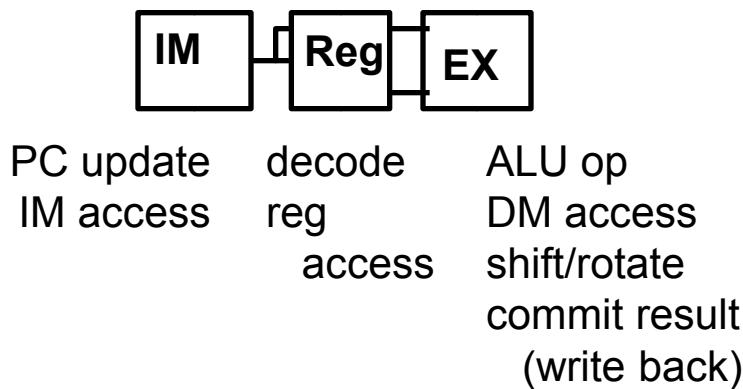


- ❑ What if the data memory access is twice as slow as the instruction memory?
 - make the clock twice as slow or ...
 - let data memory access take two cycles (and keep the same clock rate)

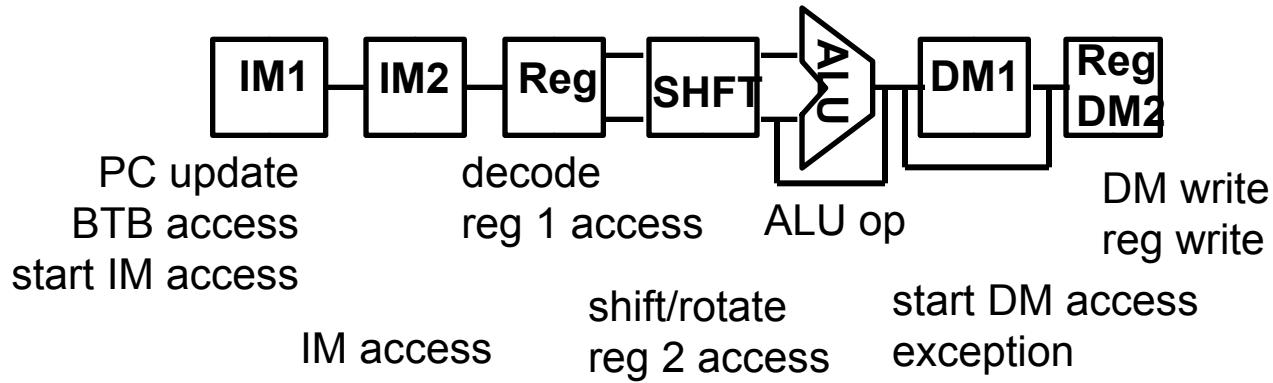


Other Sample Pipeline Alternatives

❑ ARM7



❑ XScale



Limitations to Pipelining

❑ Clock period

- All flip-flops have a setup constraint.
- In a long pipeline, the clock period asymptotes toward setup time plus one gate delay.

❑ Die/Resource Size

- Registers in silicon are big compared to logic
- FPGAs have limited resources

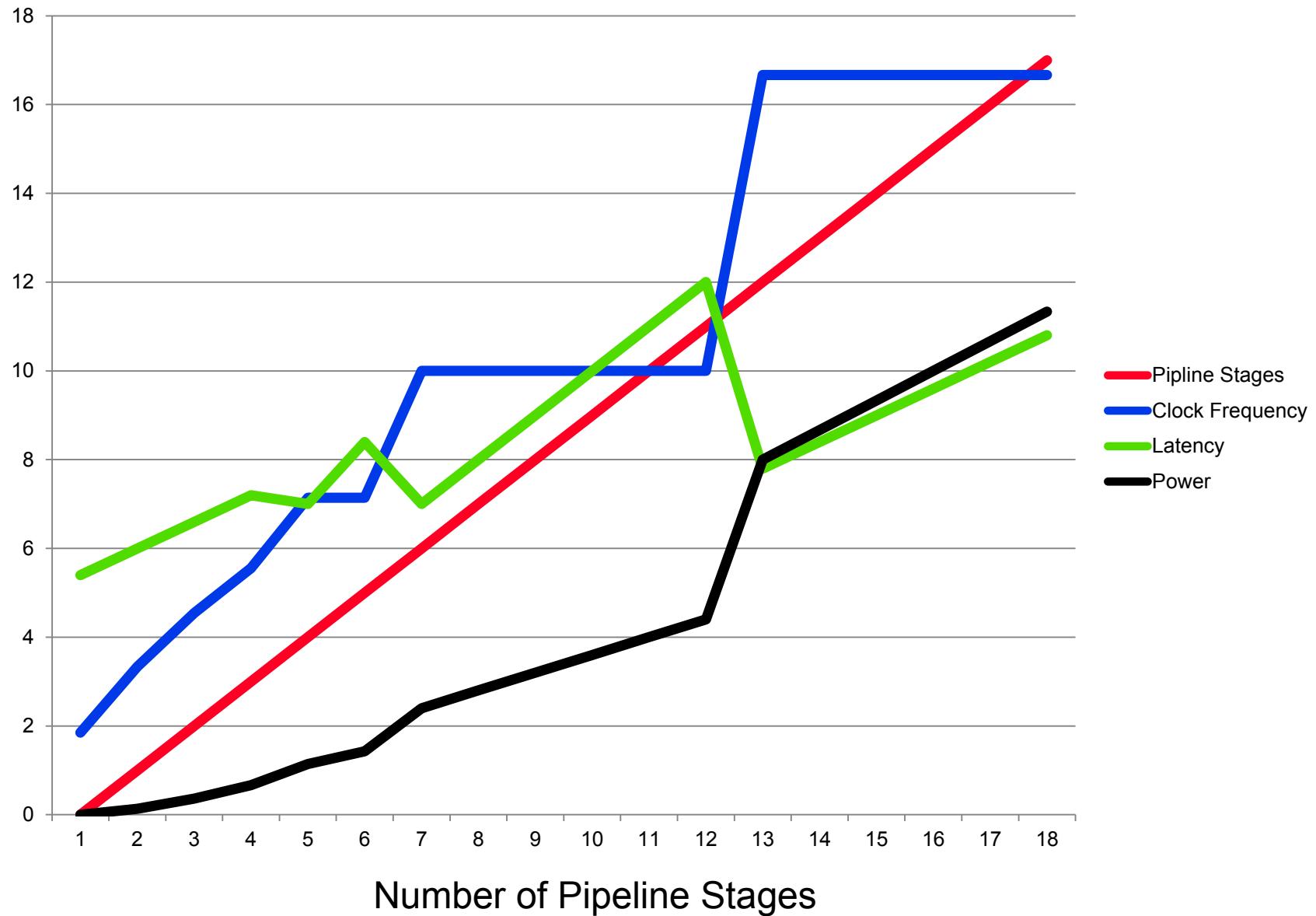
❑ Power

- More registers = more power
- Faster clock cycles = more power

❑ Heat

- More power = more heat
- More heat = slower logic!

Limitations to Pipelining



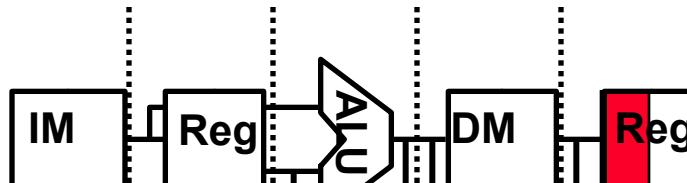
Review: Register Usage Can Cause Data Hazards

- Read before write **data hazard**

Value of \$1

10 10 10 10 10 / -20 -20 -20 -20 -20

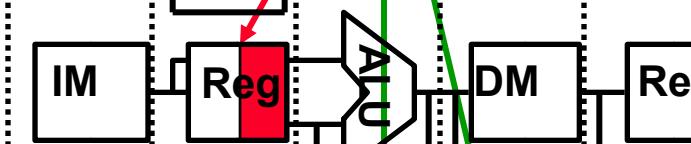
add \$1 ,



sub \$4 , \$1 , \$5



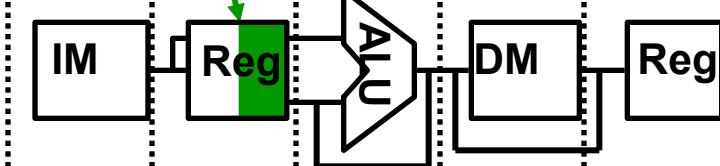
and \$6 , \$1 , \$7



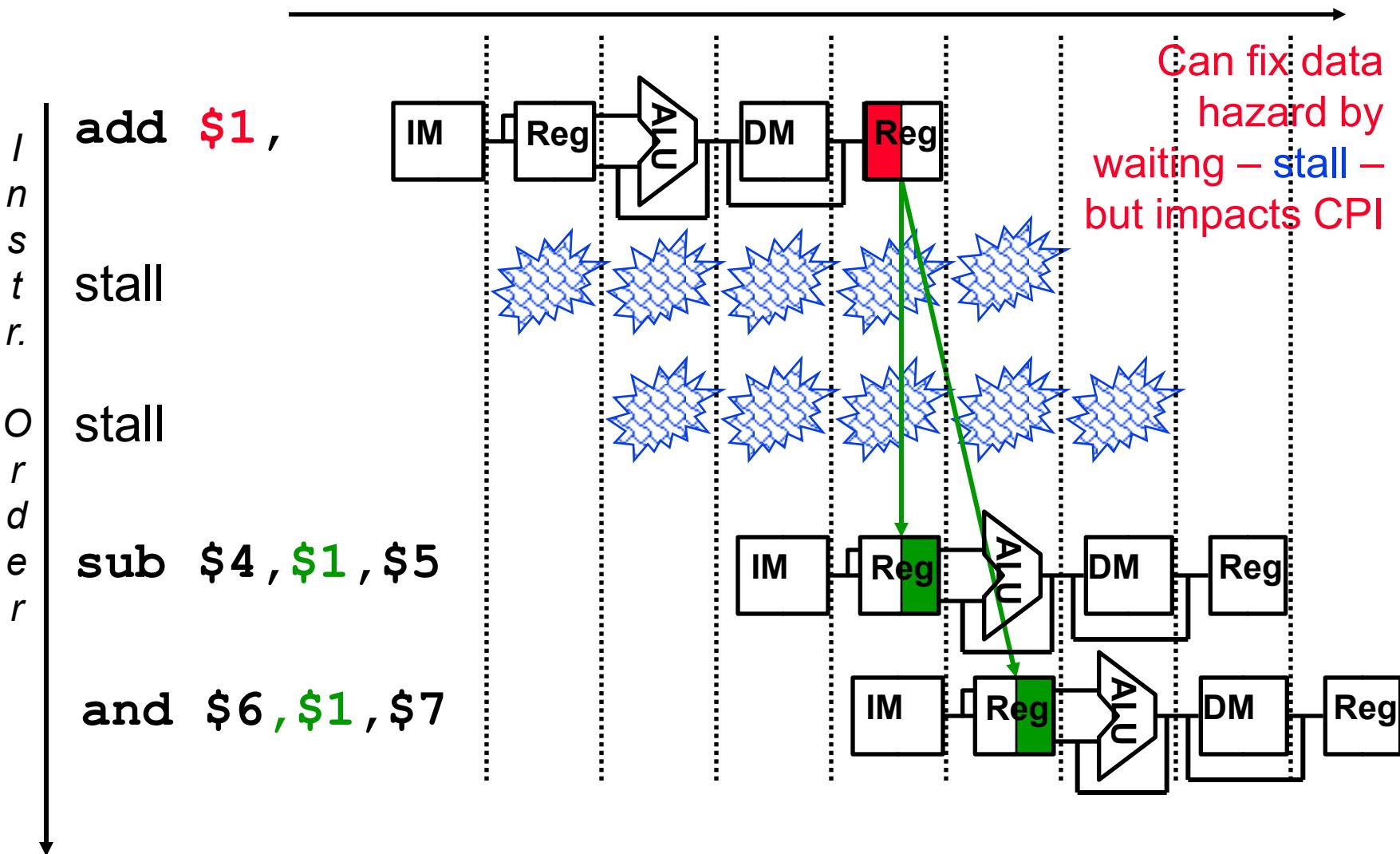
or \$8 , \$1 , \$9



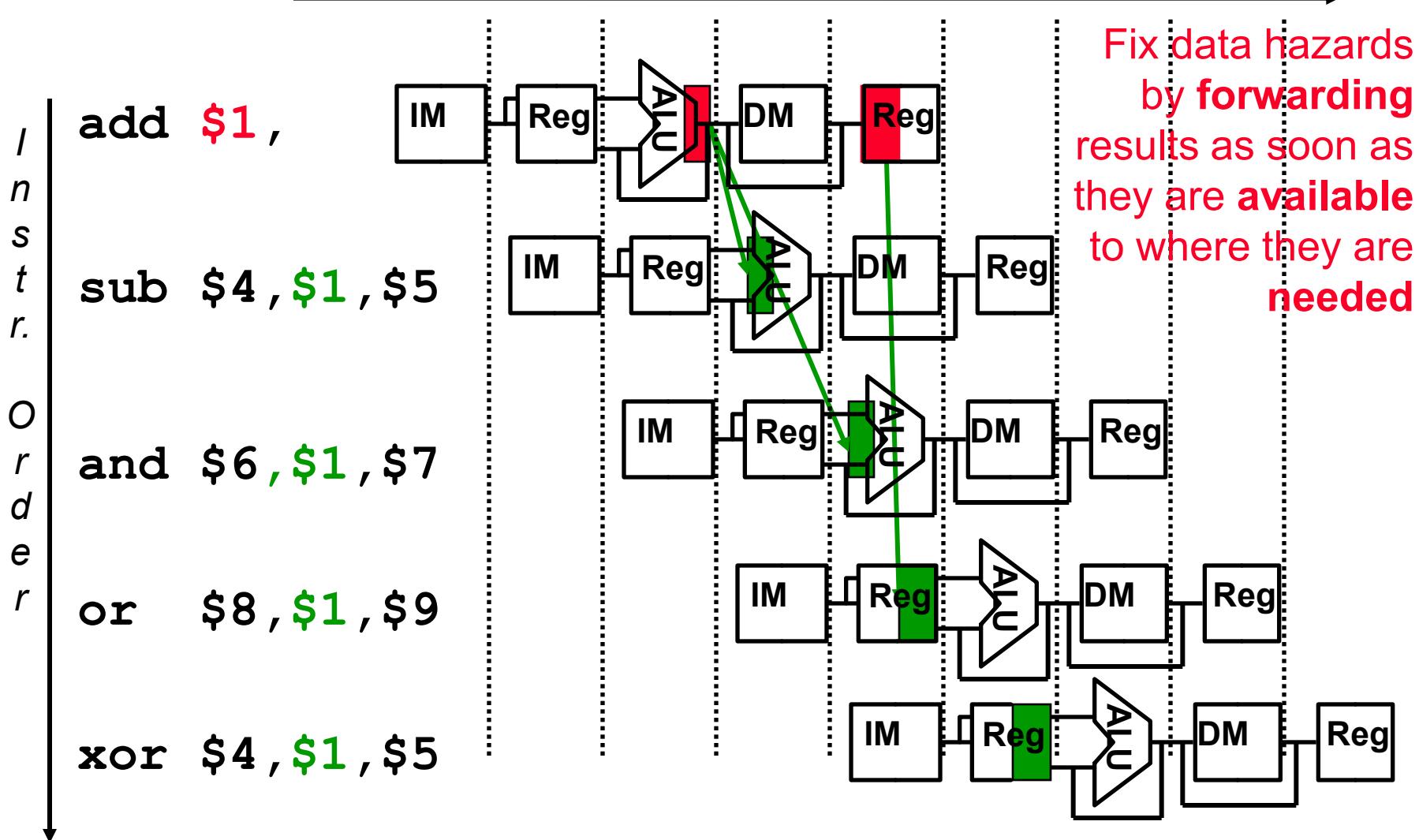
xor \$4 , \$1 , \$5



One Way to “Fix” a Data Hazard



Another Way to “Fix” a Data Hazard



Data Forwarding (aka Bypassing)

- ❑ Take the result from the earliest point that it exists in **any** of the pipeline state registers and forward it to the functional units (e.g., the ALU) that need it that cycle
- ❑ For ALU functional unit: the inputs can come from **any** pipeline register rather than just from ID/EX by
 - adding multiplexors to the inputs of the ALU
 - connecting the Rd write data in EX/MEM or MEM/WB to either (or both) of the EX's stage Rs and Rt ALU mux inputs
 - adding the proper control hardware to control the new muxes
- ❑ Other functional units may need similar forwarding logic (e.g., the DM)
- ❑ With forwarding can achieve a CPI of 1 even in the presence of data dependencies

Data Forwarding Control Conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 10  
if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 10
```

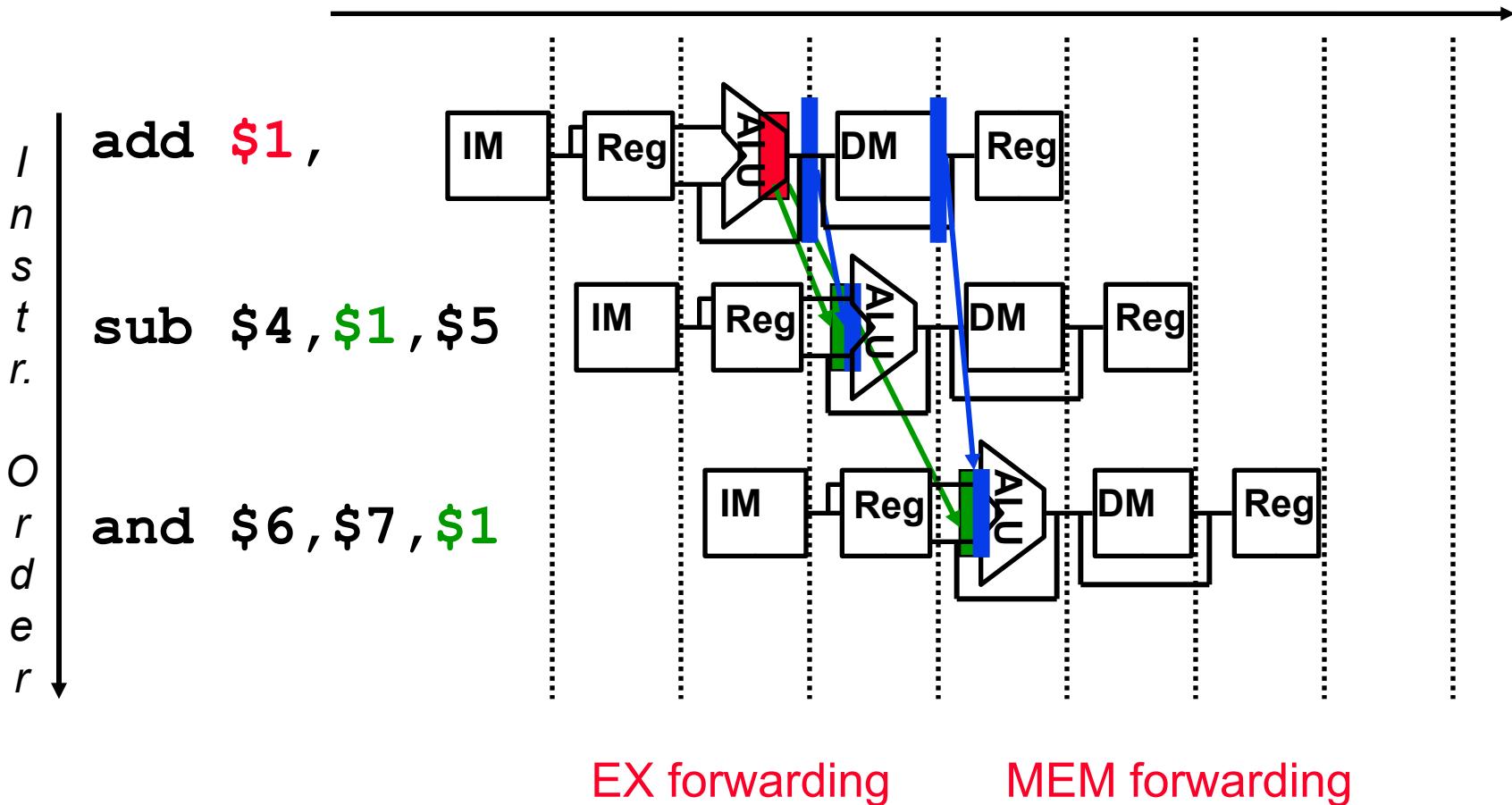
Forwards the result from the previous instr. to either input of the ALU

2. MEM Forward Unit:

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd != 0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 01  
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd != 0)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 01
```

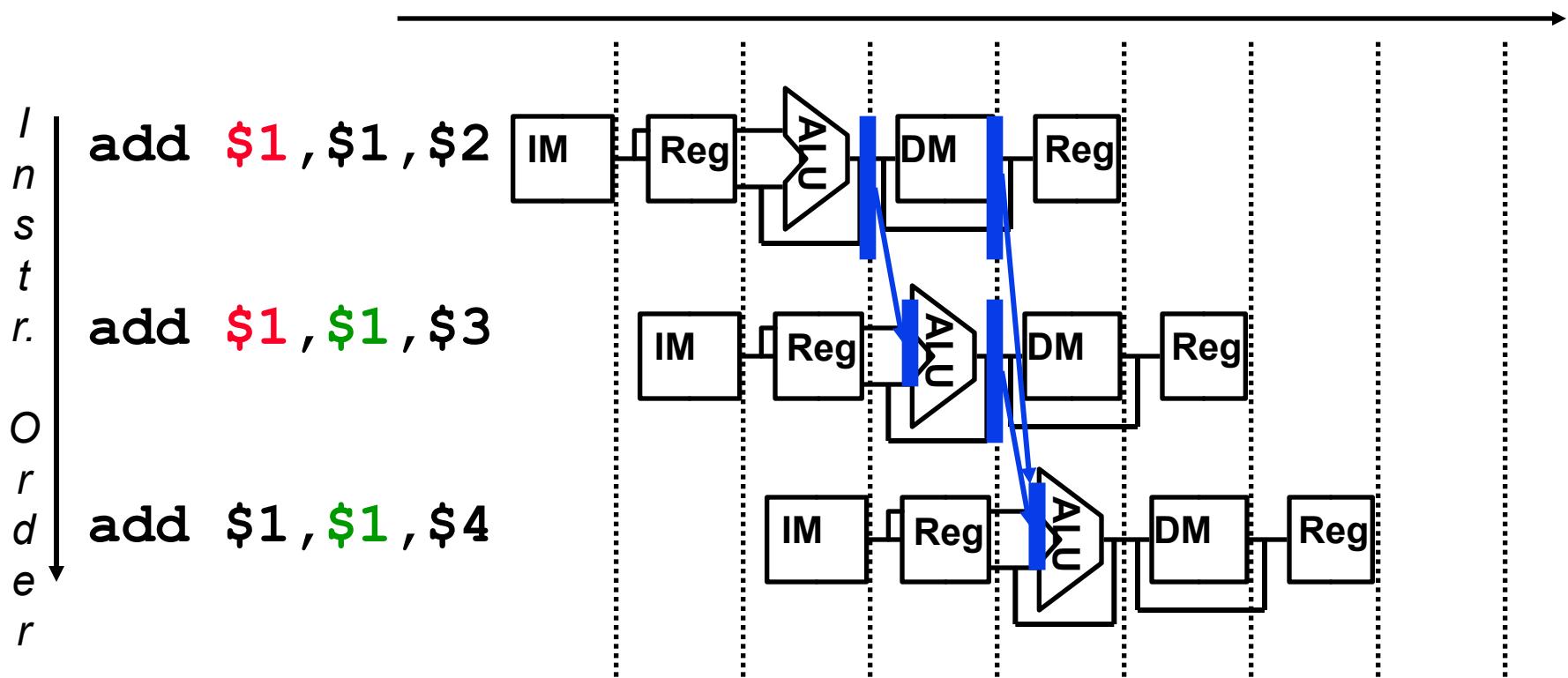
Forwards the result from the second previous instr. to either input of the ALU

Forwarding Illustration



Yet Another Complication!

- ❑ Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?



Corrected Data Forwarding Control Conditions

1. EX Forward Unit:

```
if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 10  
if (EX/MEM.RegWrite  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 10
```

Forwards the result from the previous instr. to either input of the ALU

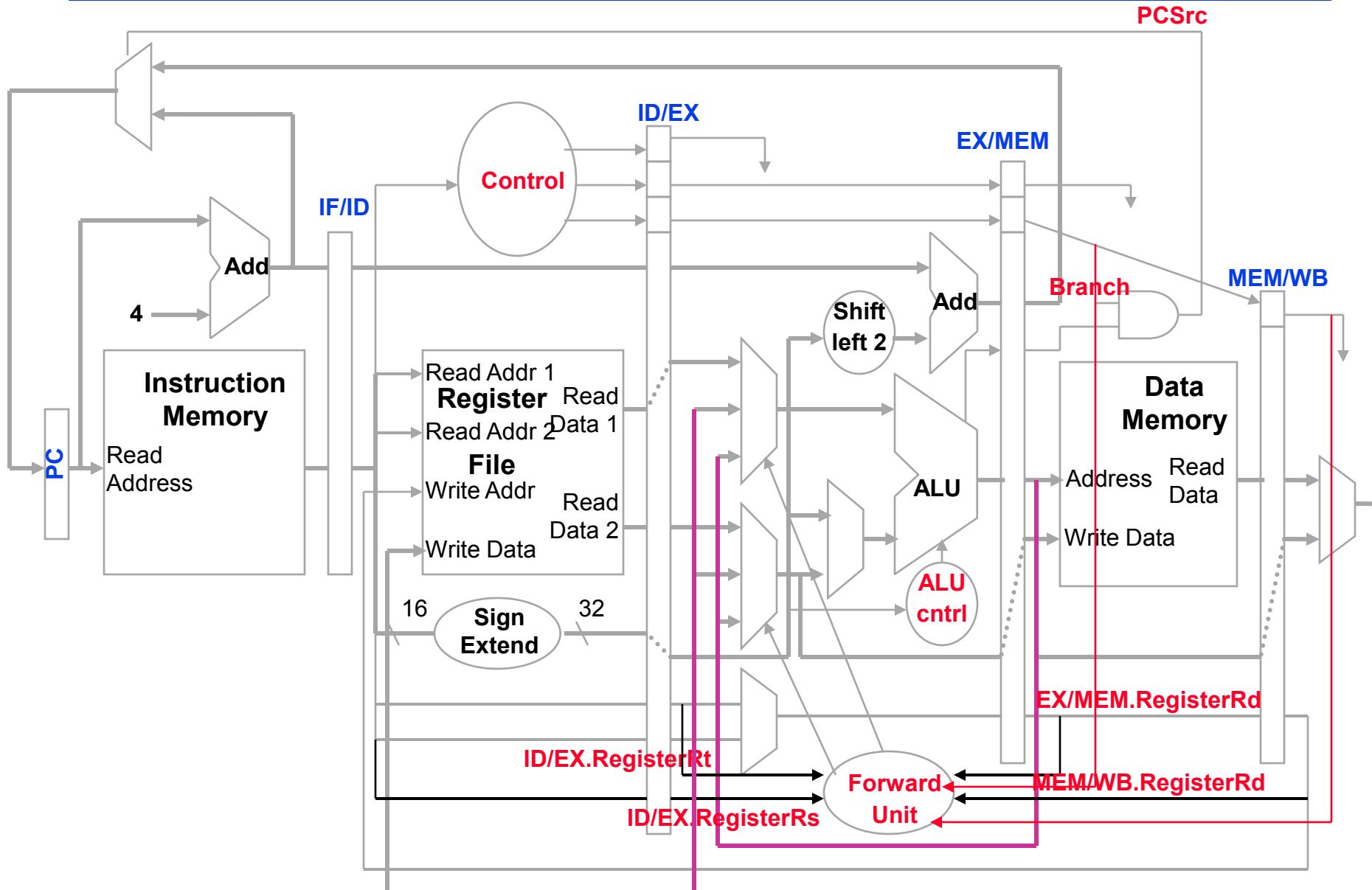
2. MEM Forward Unit:

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd != 0)  
and (EX/MEM.RegisterRd != ID/EX.RegisterRs)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))  
    ForwardA = 01
```

```
if (MEM/WB.RegWrite  
and (MEM/WB.RegisterRd != 0)  
and (EX/MEM.RegisterRd != ID/EX.RegisterRt)  
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))  
    ForwardB = 01
```

Forwards the result from the previous or second previous instr. to either input of the ALU

Datapath with Forwarding Hardware



Summary

- ❑ All modern day processors use pipelining
- ❑ Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- ❑ Potential speedup: a CPI of 1 and fast a CC
- ❑ Pipeline rate limited by **slowest** pipeline stage
 - Unbalanced pipe stages makes for inefficiencies
 - The time to “**fill**” pipeline and time to “**drain**” it can impact speedup for deep pipelines and short code runs
- ❑ Must detect and resolve hazards
 - Stalling negatively affects CPI (makes CPI less than the ideal of 1)

Next Lecture and Reminders

- ❑ Next lecture
 - Reducing pipeline data and branch hazards

ECE4074

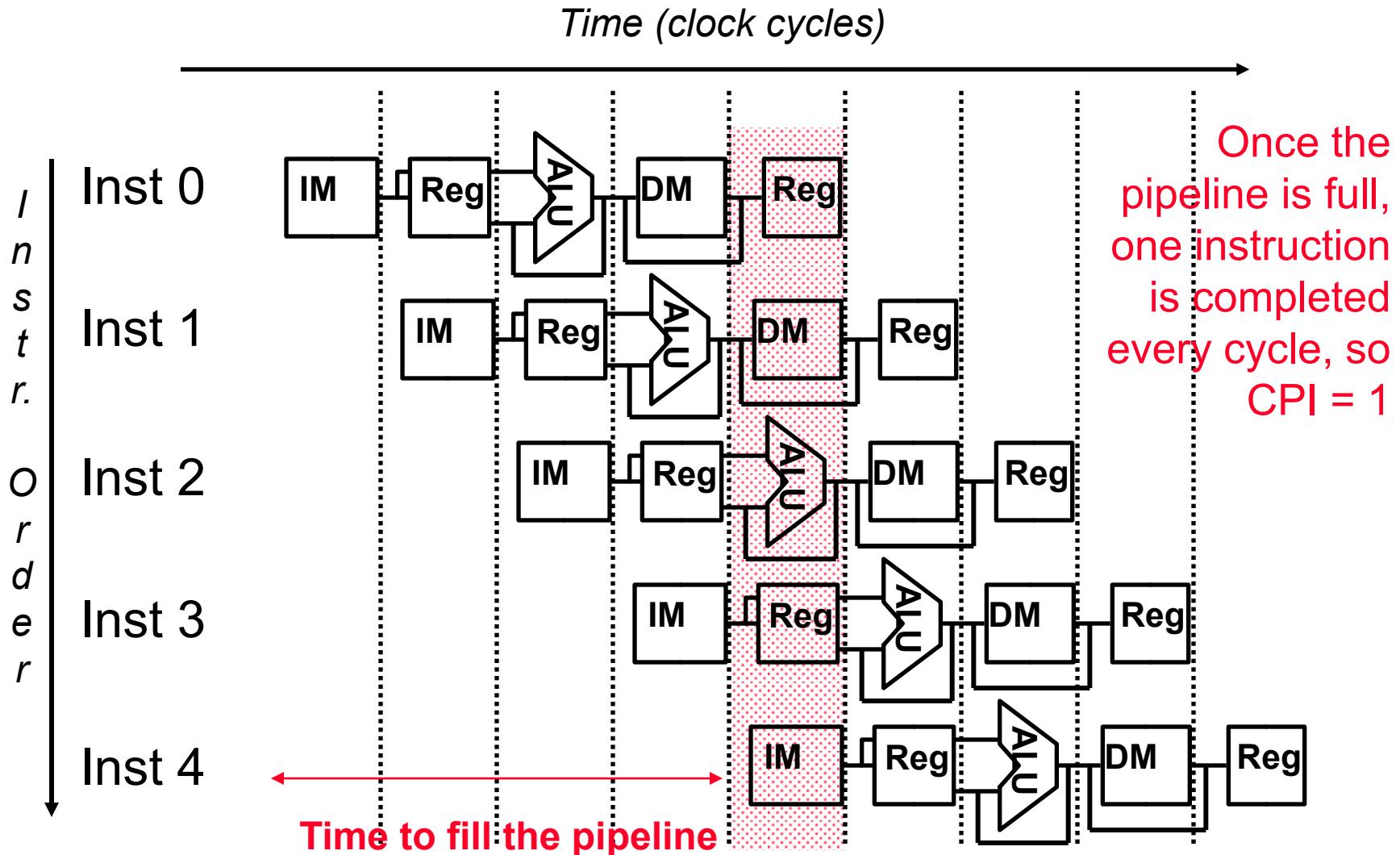
Computer Architecture

Semester 2 2014

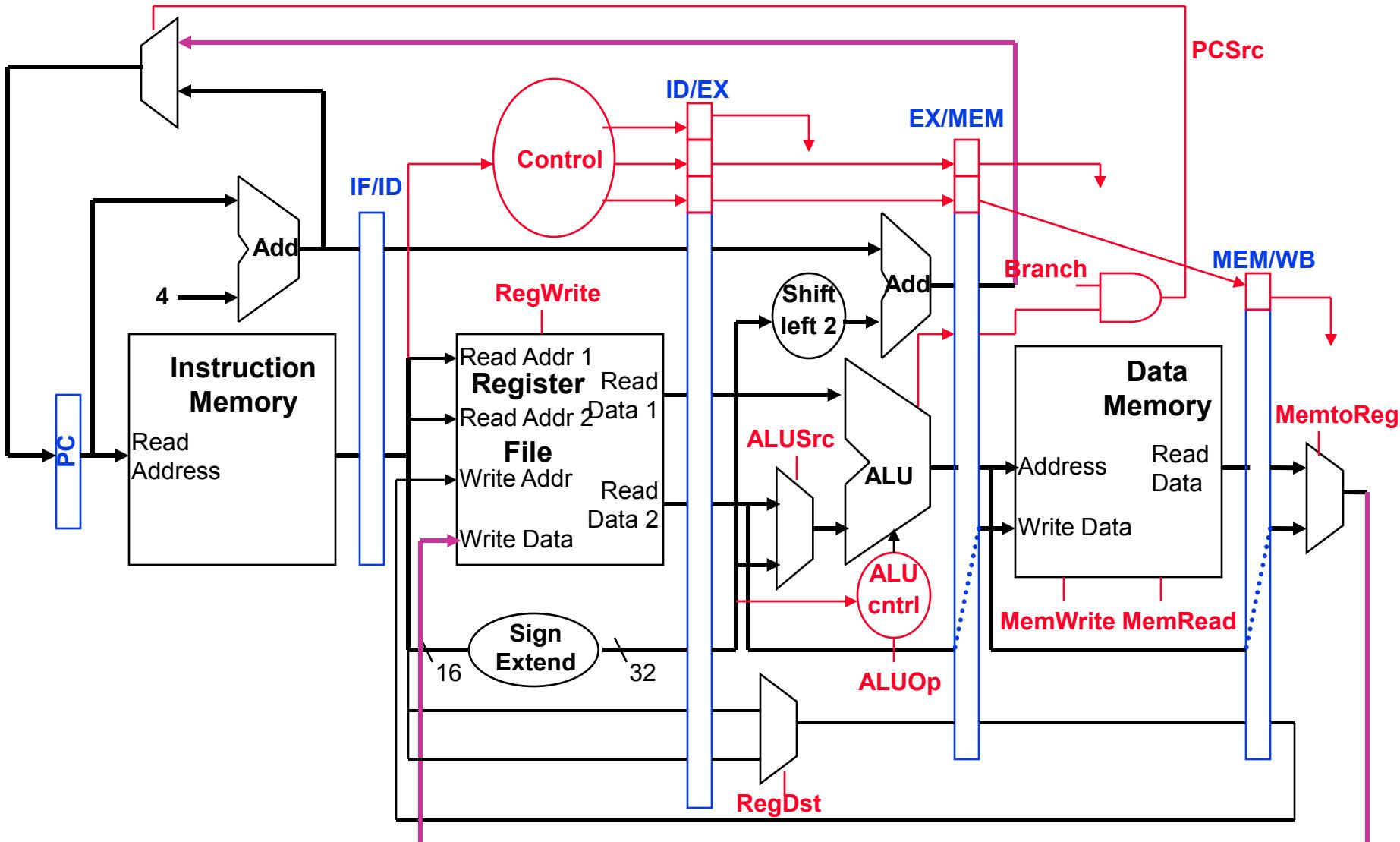
Chapter 4B: The Processor, Part B

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: Why Pipeline? For Performance!



Review: MIPS Pipeline Data and Control Paths



Review: Can Pipelining Get Us Into Trouble?

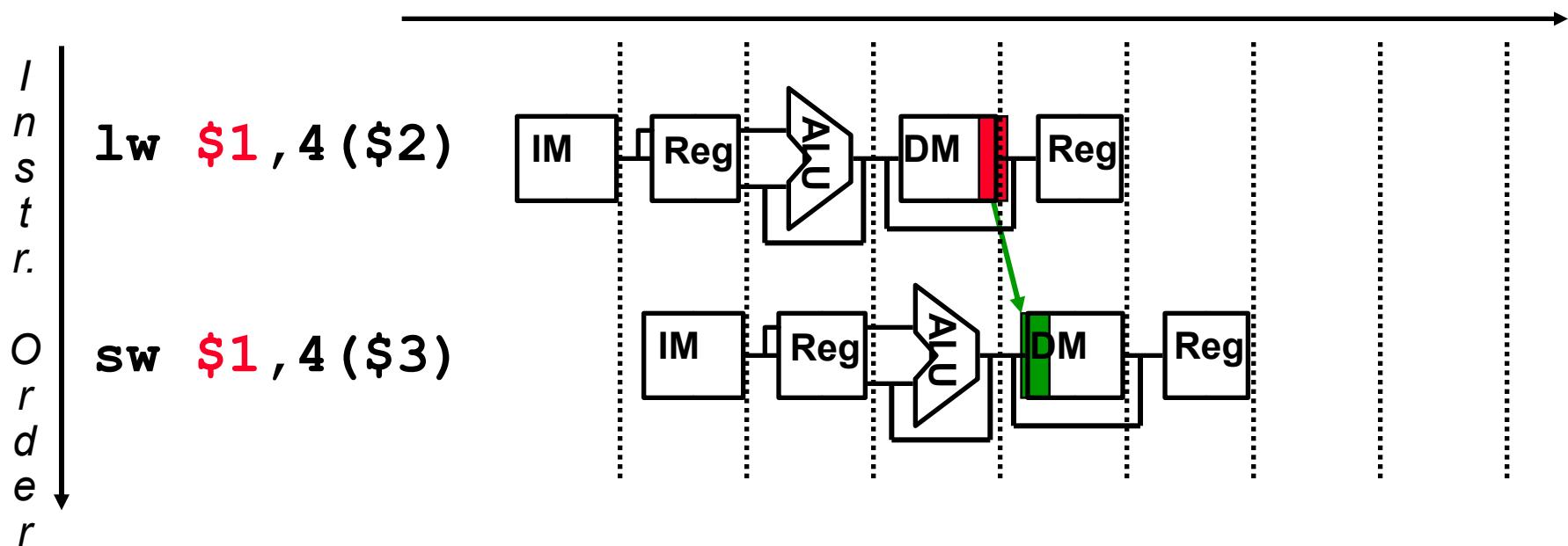
❑ Yes: Pipeline Hazards

- **structural hazards**: attempt to use the same resource by two different instructions at the same time
- **data hazards**: attempt to use data before it is ready
 - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- **control hazards**: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - branch and jump instructions, exceptions

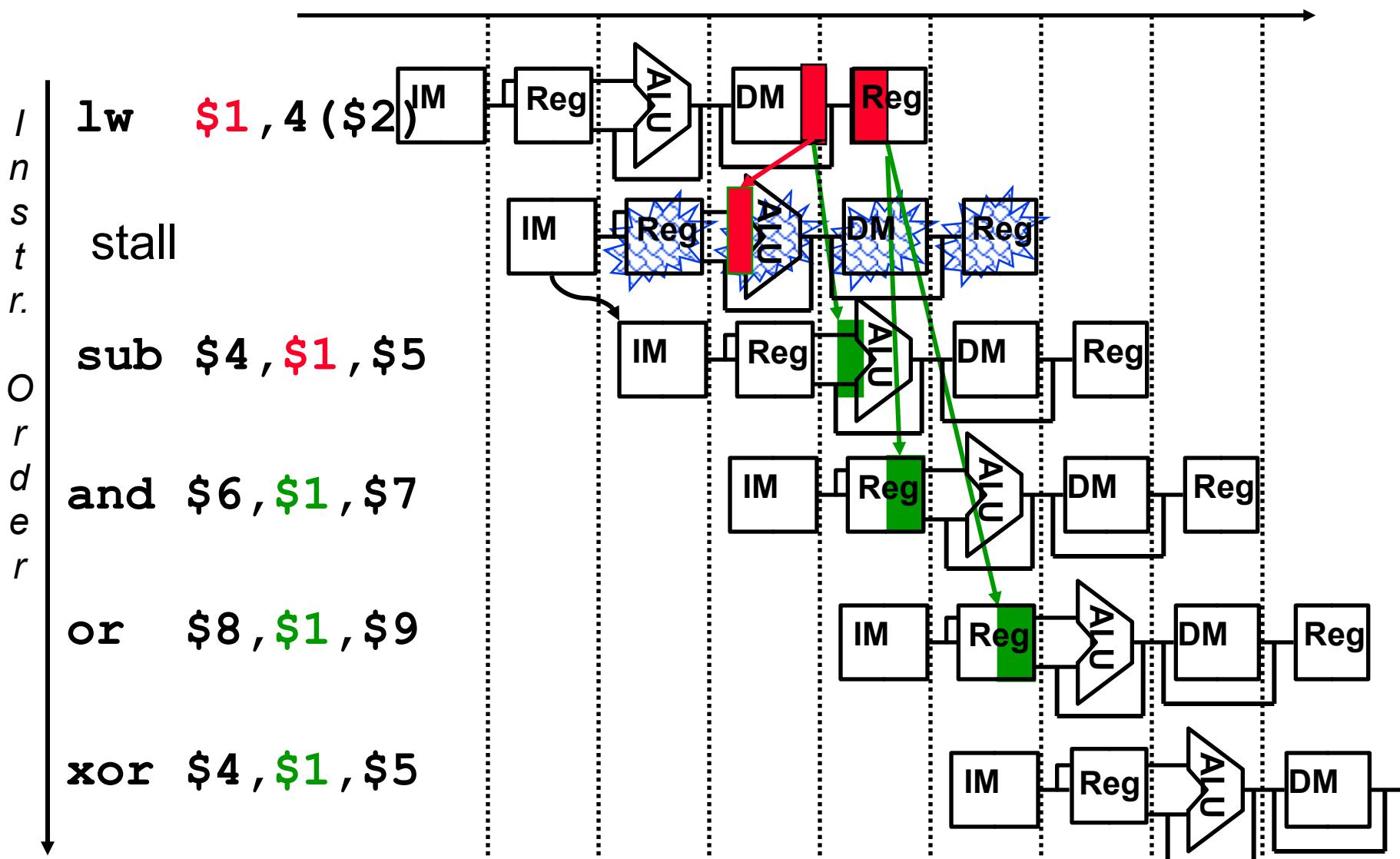
❑ Pipeline control must **detect** the hazard and then take action to **resolve** hazards

Memory-to-Memory Copies

- For loads immediately followed by stores (memory-to-memory copies) can avoid a stall by adding forwarding hardware from the MEM/WB register to the data memory input.
- Would need to add a Forward Unit and a mux to the MEM stage



Forwarding with Load-use Data Hazards



- Will still need **one stall cycle** even with forwarding

Load-use Hazard Detection Unit

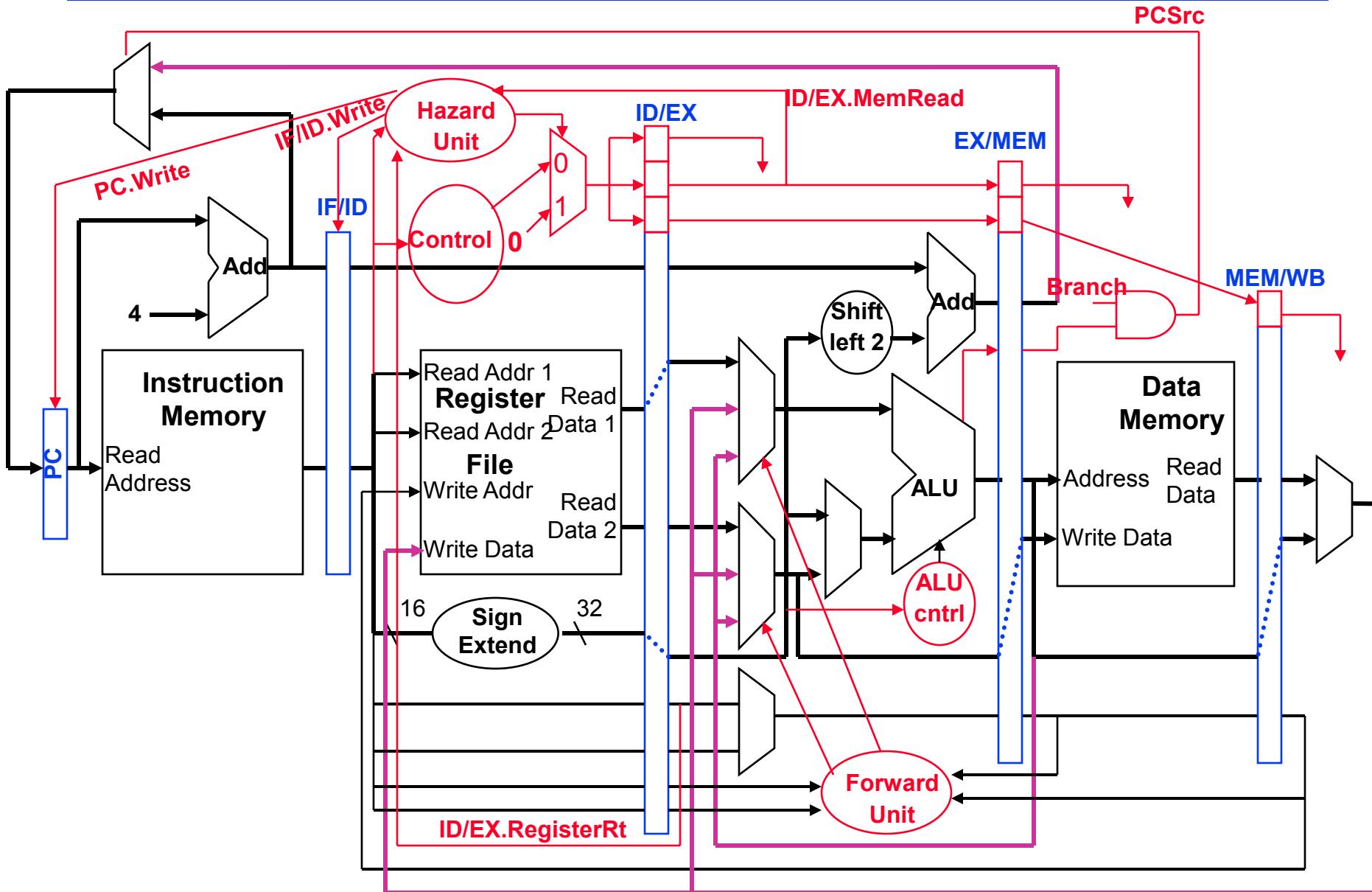
- ❑ Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use
 1. ID Hazard detection Unit:

```
if (ID/EX.MemRead  
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)  
or (ID/EX.RegisterRt = IF/ID.RegisterRs)))  
stall the pipeline
```
- ❑ The first line tests to see if the instruction now in the EX stage is a l_w ; the next two lines check to see if the destination register of the l_w matches either source register of the instruction in the ID stage (the load-use instruction)
- ❑ After this one cycle stall, the forwarding logic can handle the remaining data hazards

Hazard/Stall Hardware

- ❑ Along with the Hazard Unit, we have to implement the stall
- ❑ Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
 - Hazard detection Unit controls the writing of the PC (`PC.write`) and IF/ID (`IF/ID.write`) registers
- ❑ Insert a “bubble” **between** the `lw` instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a `noop` in the execution stream)
 - Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (`noop`). The Hazard Unit controls the mux that chooses between the real control values and the 0’s.
- ❑ Let the `lw` instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline

Adding the Hazard/Stall Hardware



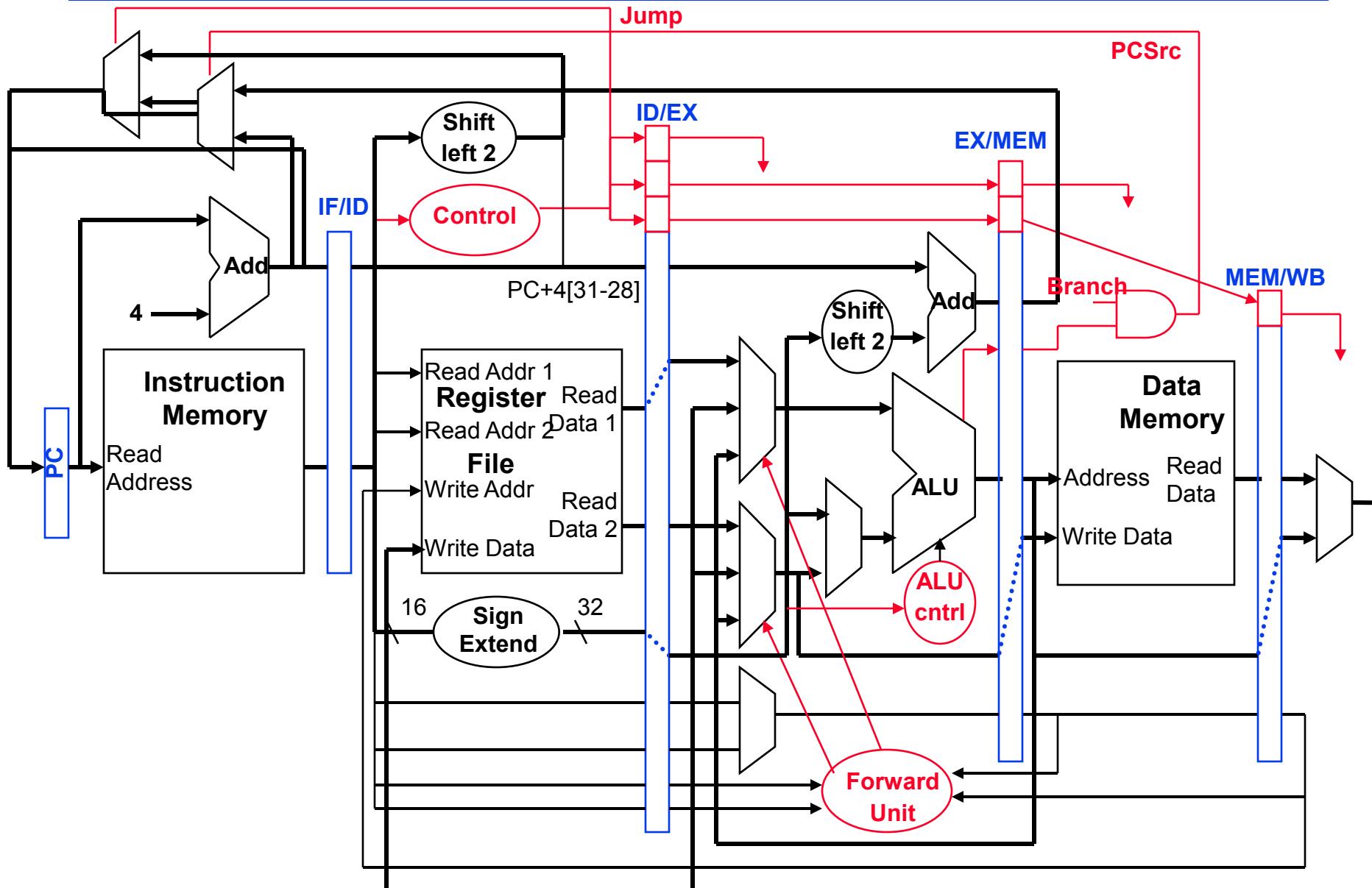
Load-use Data Hazards

- ❑ Why not disallow this hazard in the CPU specification?
 - Binary compatibility does not mean the logic internal logic is the same
 - Some versions of this ISA may support this instruction order
 - Easier for a programmer to understand/write a compiler
 - Memory use is lower with the removal of noop instructions

Control Hazards

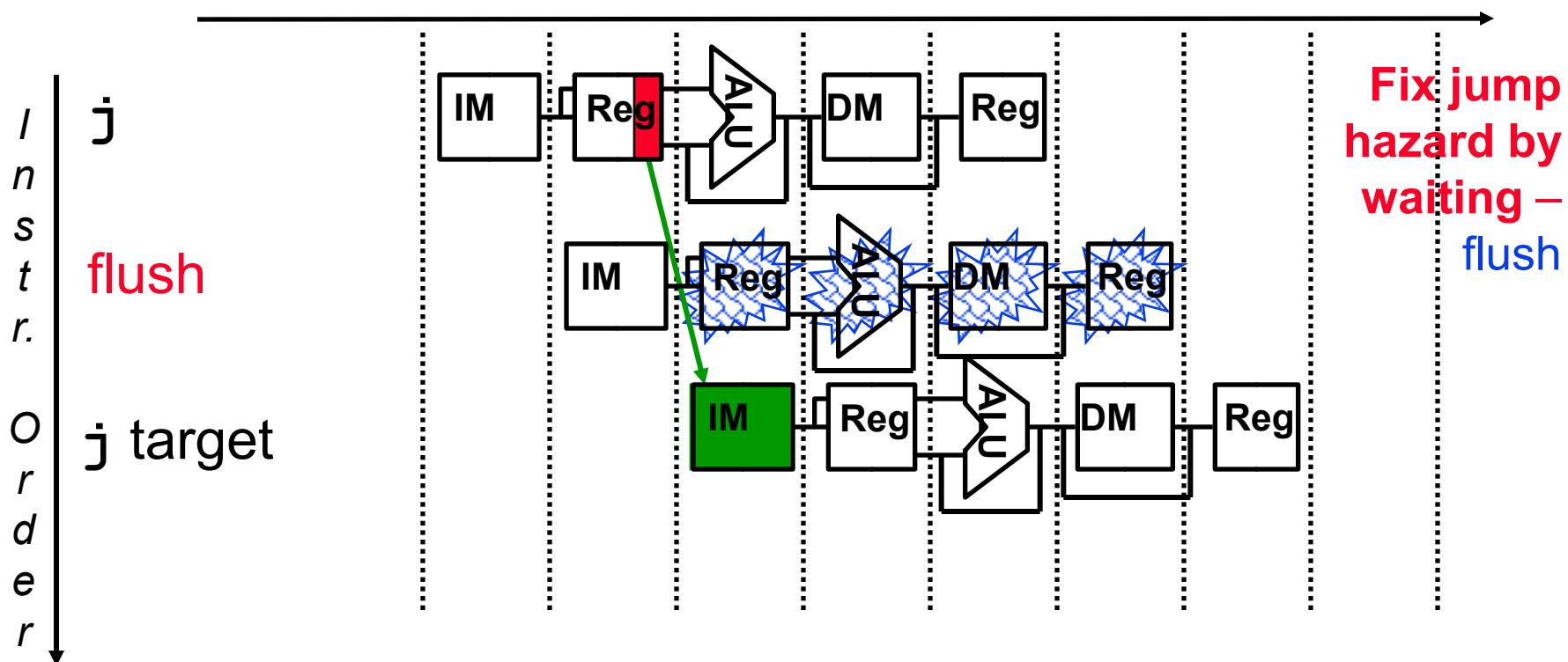
- ❑ When the flow of instruction addresses is not sequential (i.e., $PC = PC + 4$); incurred by change of flow instructions
 - Unconditional branches (`j`, `jal`, `jr`)
 - Conditional branches (`beq`, `bne`)
 - Exceptions
- ❑ Possible approaches
 - Stall (impacts CPI)
 - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
 - Delay decision (requires compiler support)
 - Predict and hope for the best !
- ❑ Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards

Datapath Branch and Jump Hardware



Jumps Incur One Stall

- ❑ Jumps not decoded until ID, so one **flush** is needed
 - To flush, set `IF.Flush` to zero the instruction field of the IF/ID pipeline register (turning it into a `noop`)

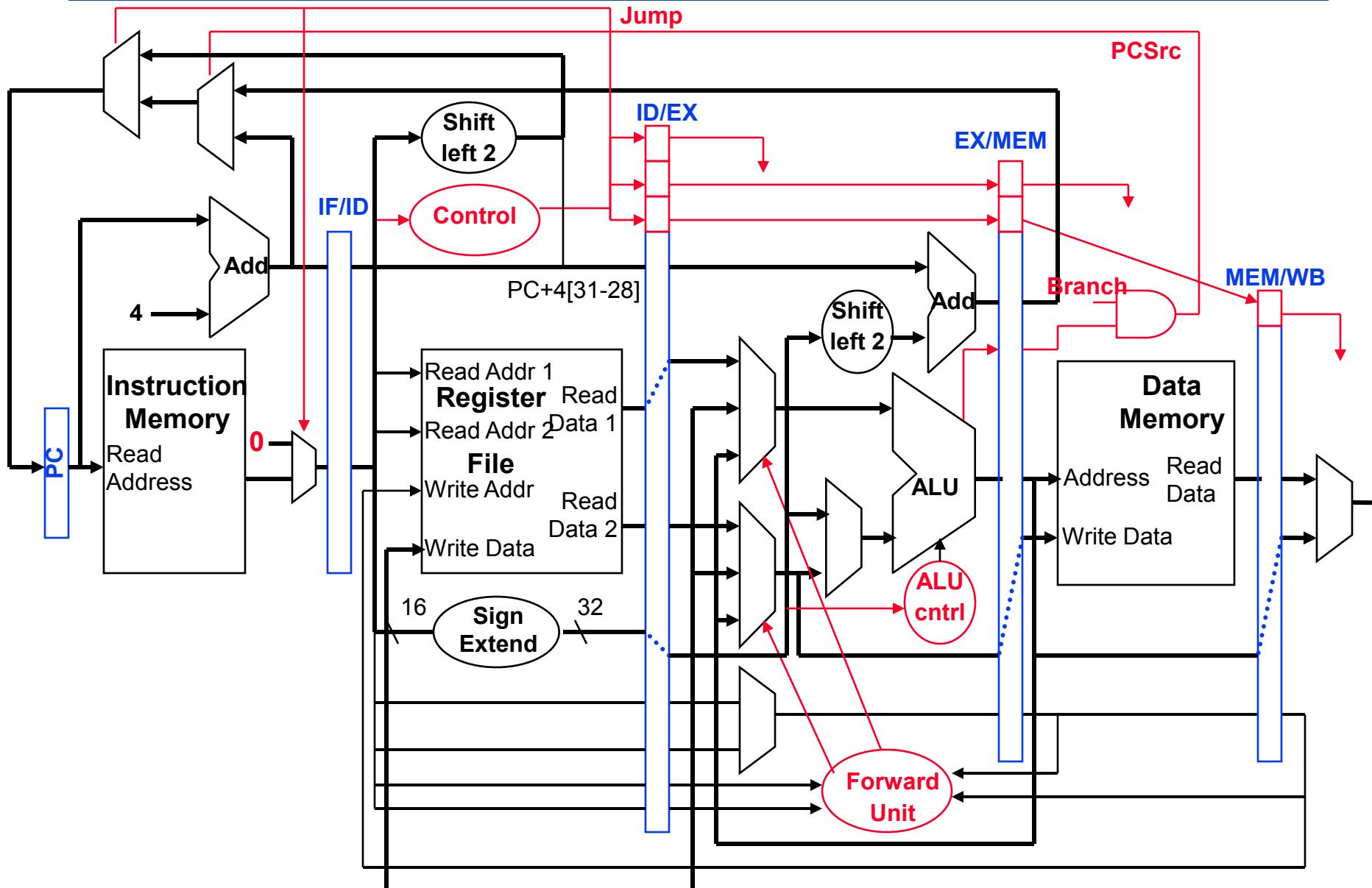


- ❑ Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Two “Types” of Stalls

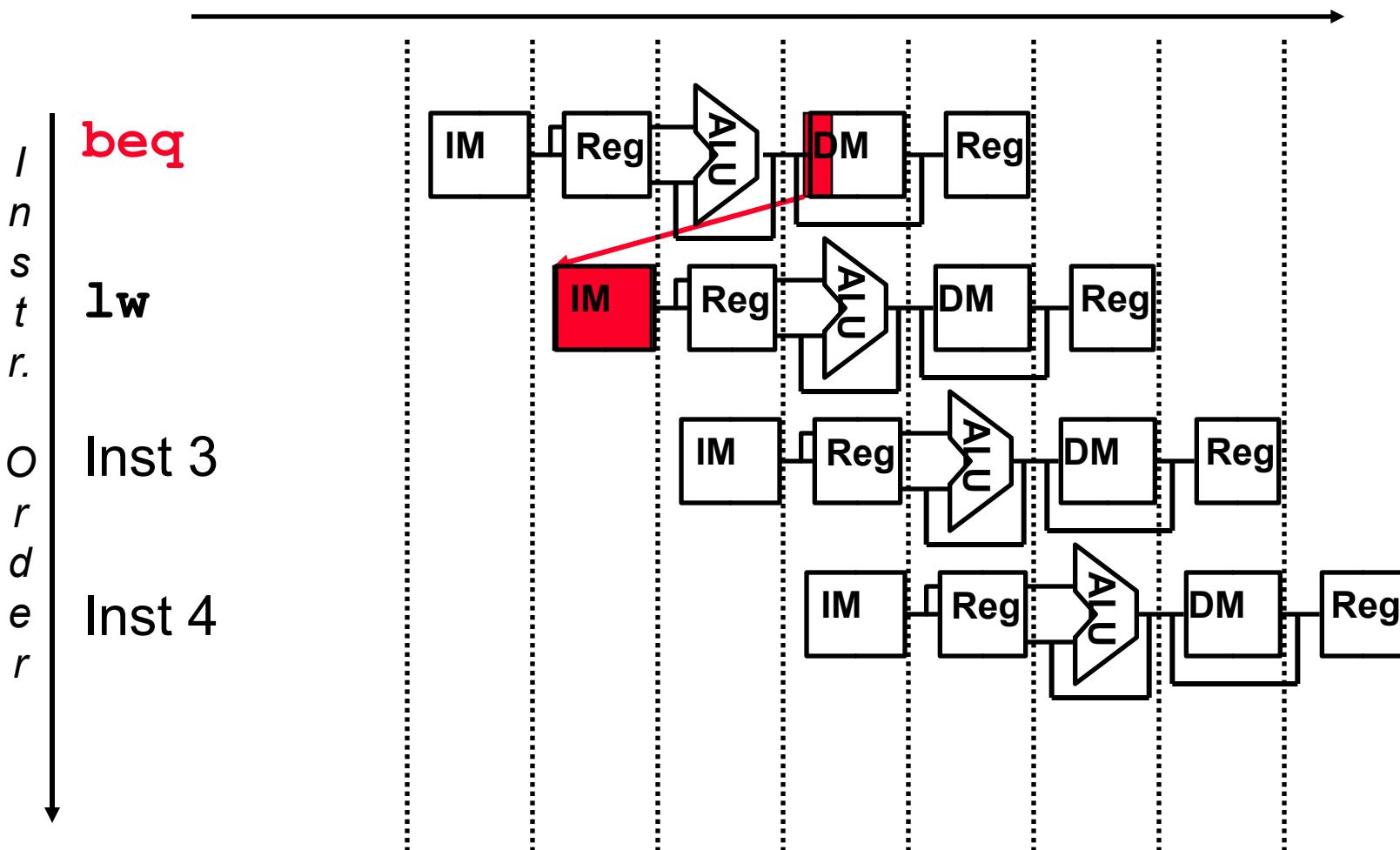
- ❑ Noop instruction (or bubble) **inserted** between two instructions in the pipeline (as done for load-use situations)
 - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“hold” them in place with write control signals)
 - Insert `noop` by zeroing control bits in the pipeline register at the appropriate stage
 - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- ❑ Flushes (or instruction squashing) were an instruction in the pipeline is **replaced** with a `noop` instruction (as done for instructions located sequentially after j instructions)
 - Zero the control bits for the instruction to be flushed

Supporting ID Stage Jumps

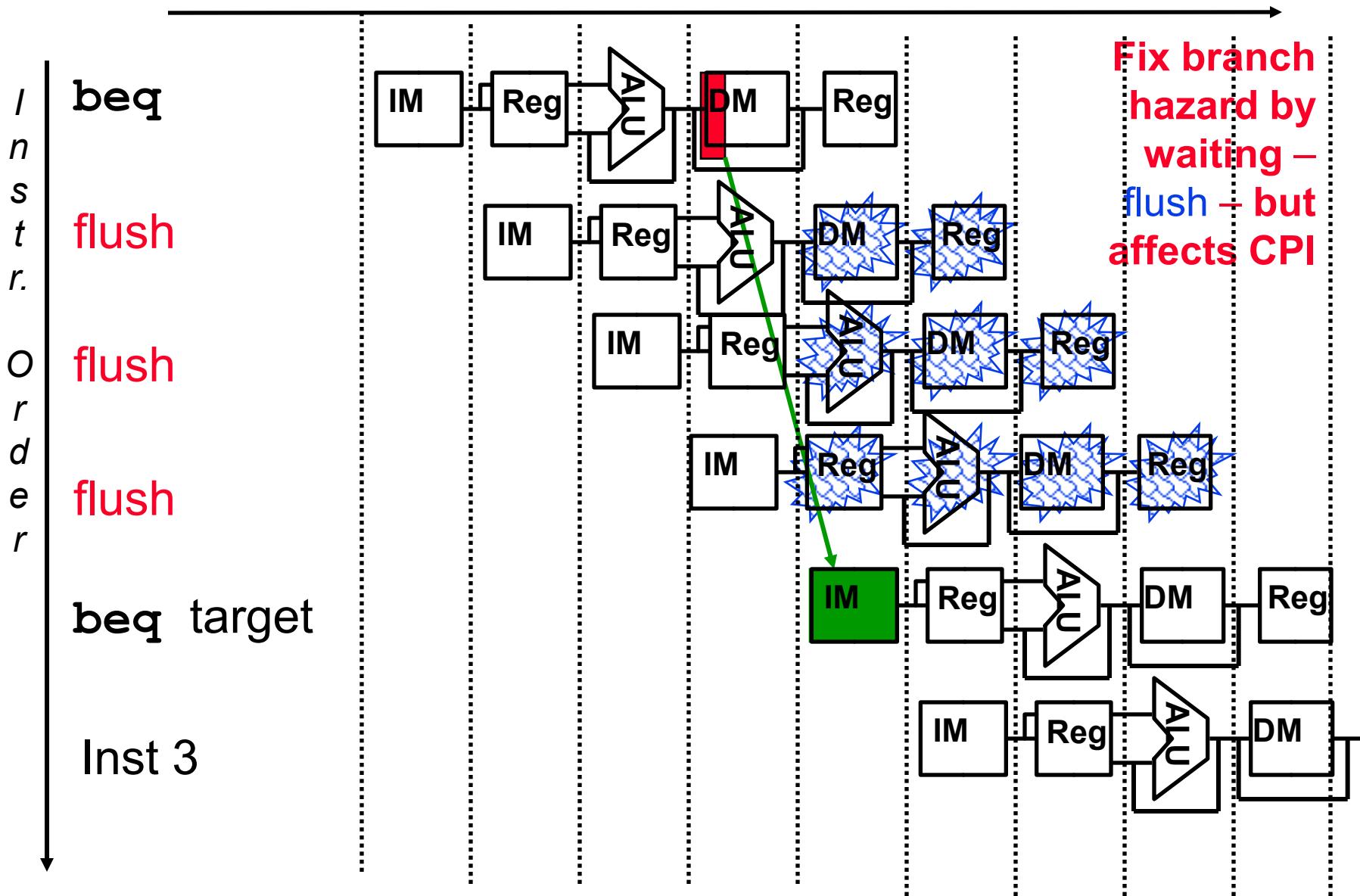


Review: Branch Instr's Cause Control Hazards

- Dependencies backward in time cause **hazards**

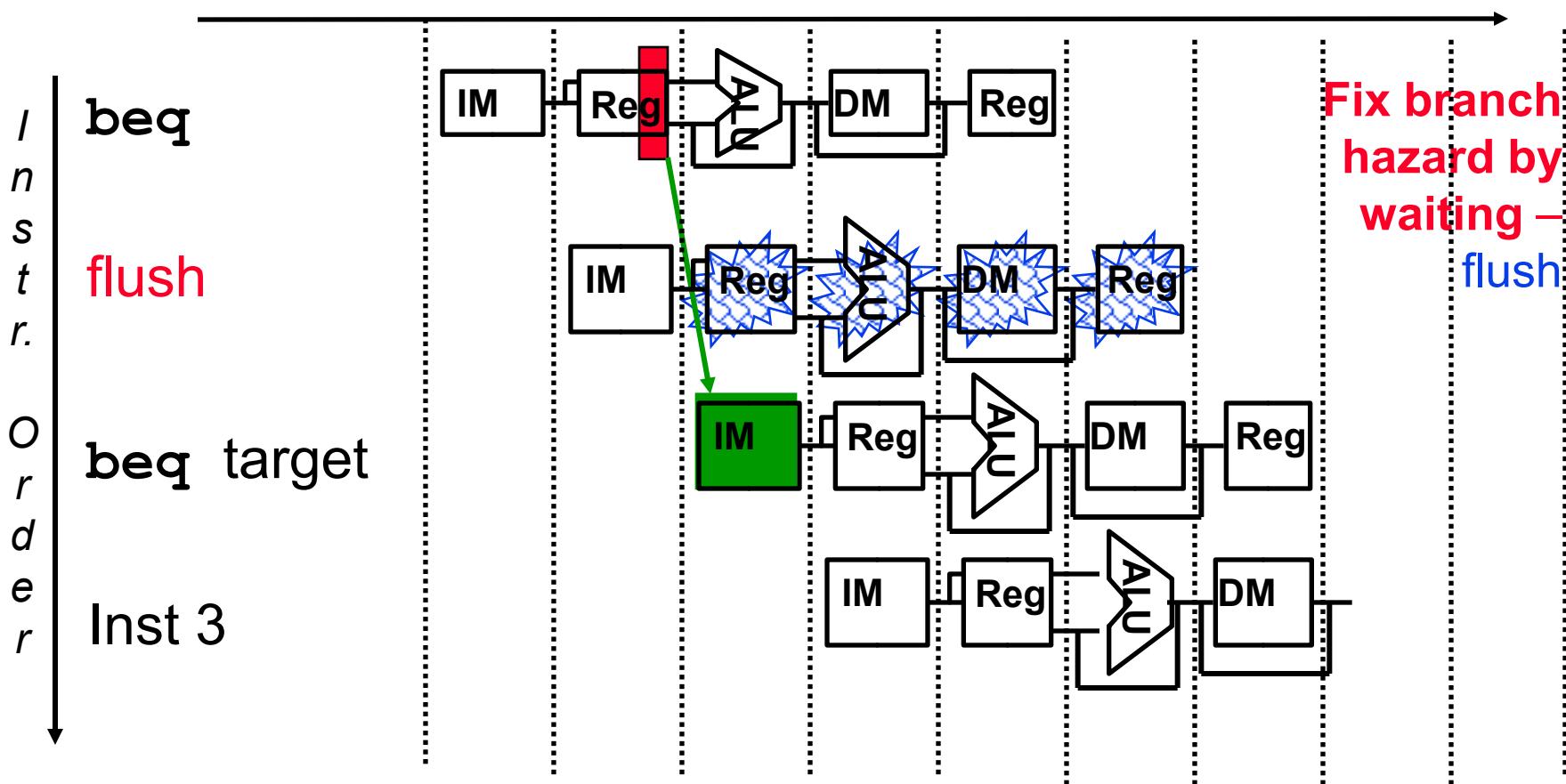


One Way to “Fix” a Branch Control Hazard



Another Way to “Fix” a Branch Control Hazard

- ❑ Move branch decision hardware back to as **early** in the pipeline as possible – i.e., during the decode cycle



Reducing the Delay of Branches

- ❑ Move the branch decision hardware back to the EX stage
 - Reduces the number of stall (flush) cycles to two
 - Adds an and gate and a 2×1 mux to the EX timing path
- ❑ Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
 - Reduces the number of stall (flush) cycles to one (like with jumps)
 - But now need to add **forwarding hardware** in ID stage
 - Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
 - Comparing the registers can't be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an and gate to the ID timing path
- ❑ For deeper pipelines, branch decision points can be even *later* in the pipeline, incurring more stalls

ID Branch Forwarding Issues

- ❑ MEM/WB “forwarding” is taken care of by the normal RegFile write before read operation

| | | |
|-----|----------------|----------------|
| WB | add3 | \$1, |
| MEM | add2 | \$3, |
| EX | add1 | \$4, |
| ID | beq | \$1, \$2, Loop |
| IF | next_seq_instr | |

- ❑ Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like

```
if (IDcontrol.Branch  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = IF/ID.RegisterRs))  
    ForwardC = 1
```

```
if (IDcontrol.Branch  
and (EX/MEM.RegisterRd != 0)  
and (EX/MEM.RegisterRd = IF/ID.RegisterRt))  
    ForwardD = 1
```

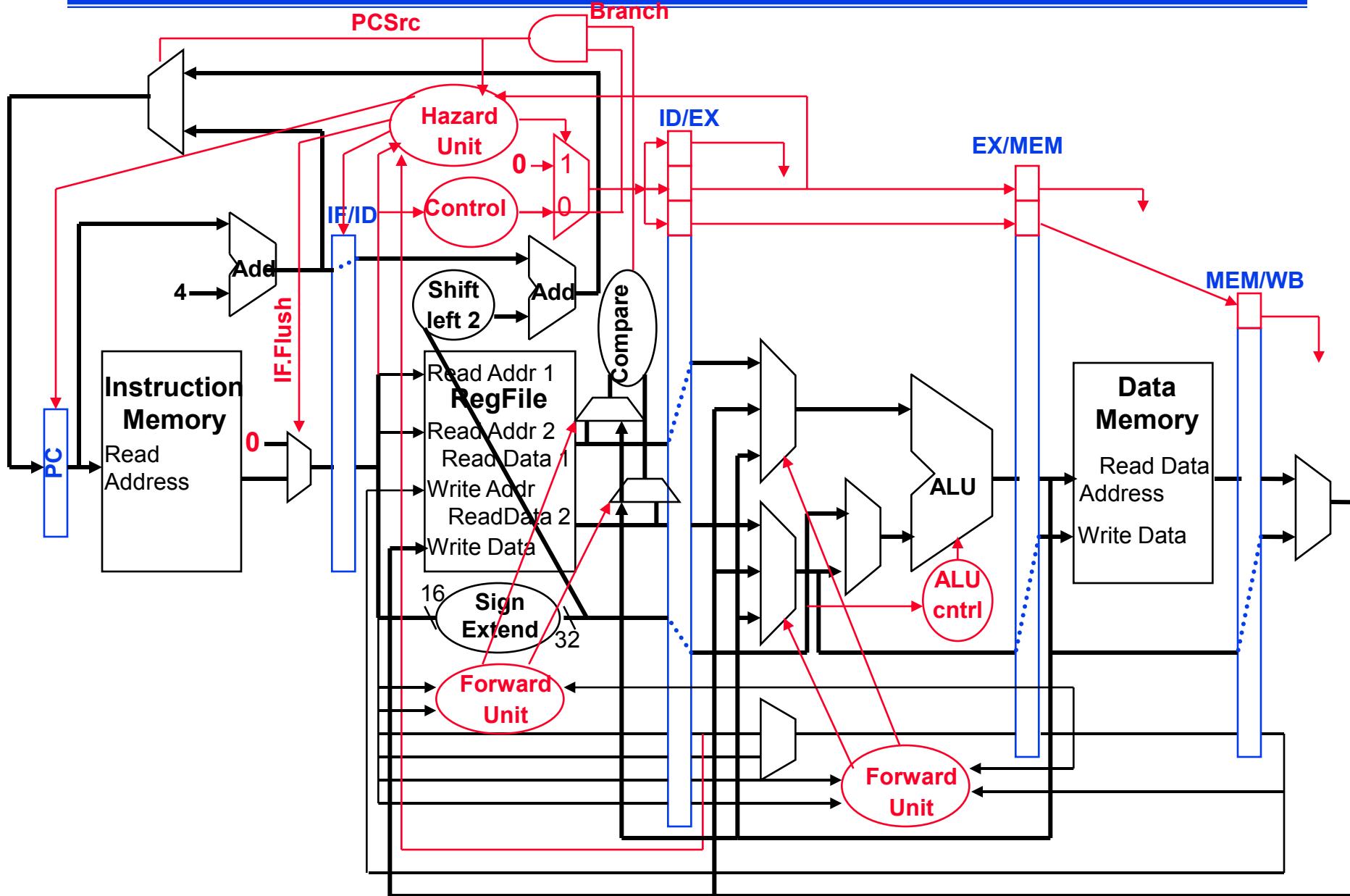
| | | |
|-----|----------------|----------------|
| WB | add3 | \$3, |
| MEM | add2 | \$1, |
| EX | add1 | \$4, |
| ID | beq | \$1, \$2, Loop |
| IF | next_seq_instr | |

Forwards the result from the second previous instr. to either input of the compare

ID Branch Forwarding Issues, con't

- ❑ If the instruction immediately before the branch produces one of the branch source operands, then a **stall** needs to be inserted (between the `beq` and `add1`) since the EX stage ALU operation is occurring at the *same time* as the ID stage branch compare operation
 - “Hold” the `beq` (in ID) and `next_seq_instr` (in IF) in place (ID Hazard Unit deasserts `PC.Write` and `IF.ID.Write`)
 - Insert a stall between the `add` in the EX stage and the `beq` in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)
 - ❑ If the branch is found to be taken, then flush the instruction currently in IF (`IF.Flush`)
- | | | |
|-----|----------------|------------------------|
| WB | add3 | \$3, |
| MEM | add2 | \$4, |
| EX | add1 | \$1, \$1, \$2, Loop |
| ID | beq | |
| IF | next_seq_instr | |

Supporting ID Stage Branches

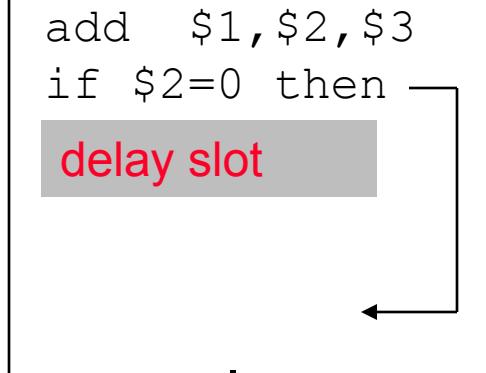


Delayed Branches

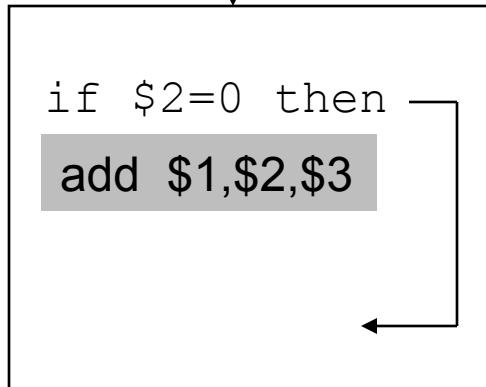
- ❑ If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with **delayed branches** which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
 - MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a **safe** instruction) thereby **hiding** the branch delay
- ❑ With deeper pipelines, the branch delay grows requiring more than one delay slot
 - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
 - Growth in available transistors has made hardware branch prediction relatively cheaper

Scheduling Branch Delay Slots

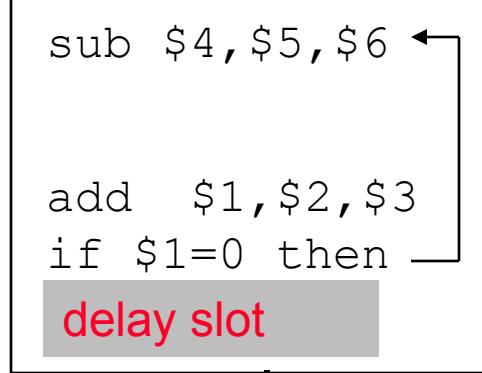
A. From before branch



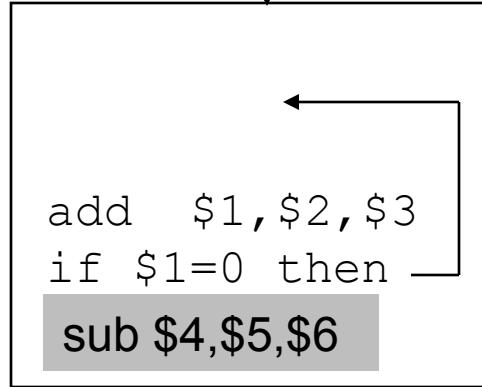
becomes



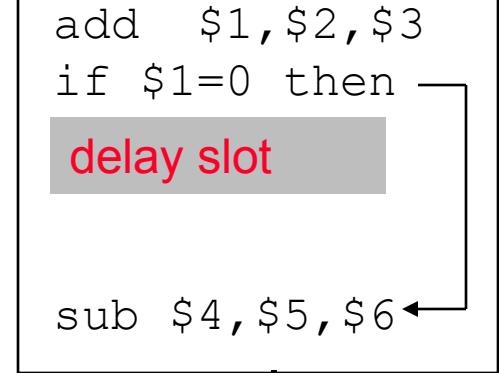
B. From branch target



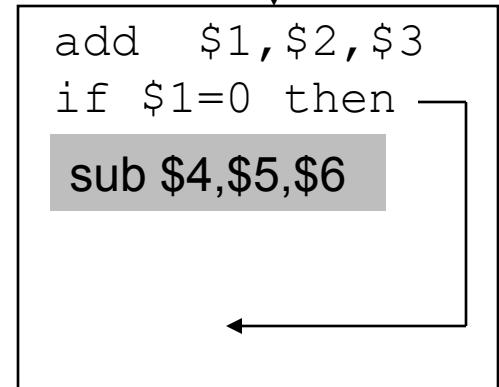
becomes



C. From fall through



becomes

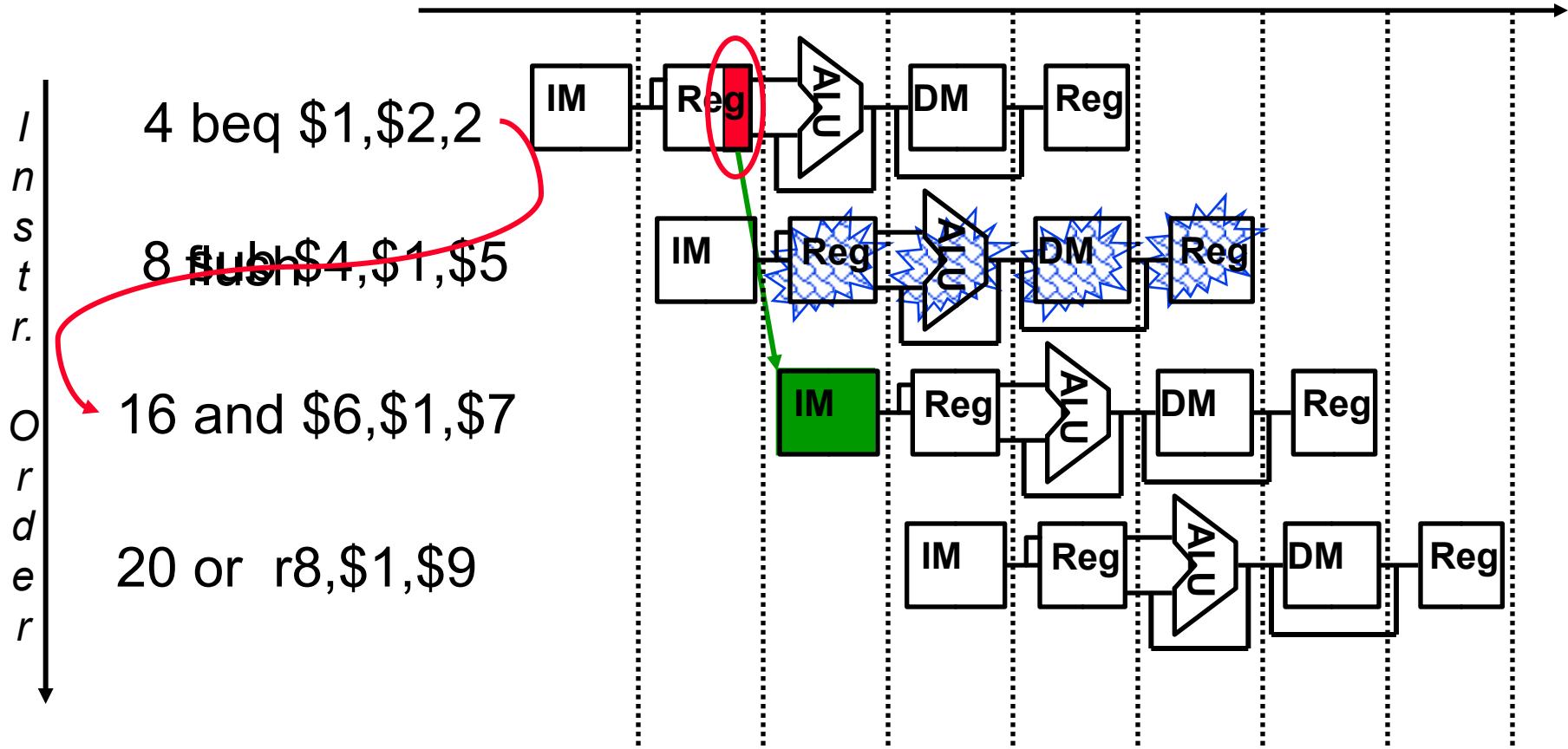


- ❑ A is the best choice, fills delay slot and reduces IC
- ❑ In B and C, the `sub` instruction may need to be copied, increasing IC
- ❑ In B and C, must be okay to execute `sub` when branch fails

Static Branch Prediction

- ❑ Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome
1. Predict not taken – always predict branches will **not** be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
 - If taken, **flush** instructions **after** the branch (earlier in the pipeline)
 - in IF, ID, and EX stages if branch logic in MEM – **three** stalls
 - In IF and ID stages if branch logic in EX – **two** stalls
 - in IF stage if branch logic in ID – **one** stall
 - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
 - restart the pipeline at the branch destination

Flushing with Misprediction (Not Taken)



- To flush the IF stage instruction, assert `IF.Flush` to zero the instruction field of the IF/ID pipeline register (transforming it into a noop)

Branching Structures

- ❑ Predict not taken works well for “top of the loop” branching structures

- But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

```
Loop: beq $1,$2,Out
      1nd loop instr
      .
      .
      last loop instr
      j Loop
Out: fall out instr
```

- ❑ Predict not taken doesn’t work well for “bottom of the loop” branching structures

```
Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

Static Branch Prediction, con't

- ❑ Resolve branch hazards by assuming a given outcome and proceeding
2. **Predict taken** – predict branches will always be taken
- Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
 - Is there a way to “cache” the address of the branch target instruction ??
- ❑ As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior **dynamically** during program execution
3. **Dynamic branch prediction** – predict branches at run-time using *run-time* information

Dynamic Branch Prediction

- ❑ A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
 - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn't affect correctness, just performance
 - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
 - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
 - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)

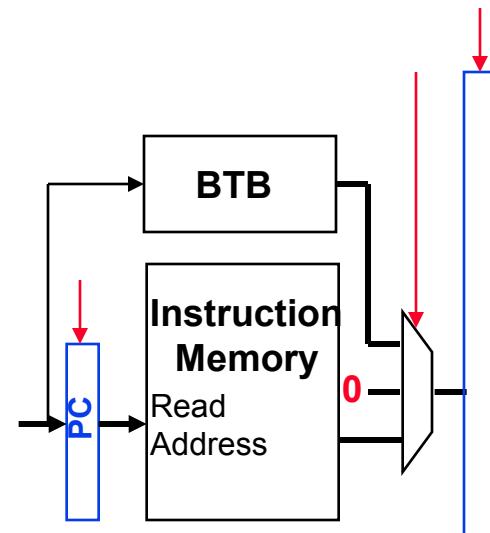
Branch Target Buffer

- ❑ The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!

- A **branch target buffer (BTB)** in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which “next” instruction will be loaded into IF/ID at the next clock edge

- Would need a two read port instruction memory

- Or the BTB can cache the branch taken **instruction** while the instruction memory is fetching the next sequential instruction



- ❑ If the prediction is correct, stalls can be avoided no matter which direction they go

1-bit Prediction Accuracy

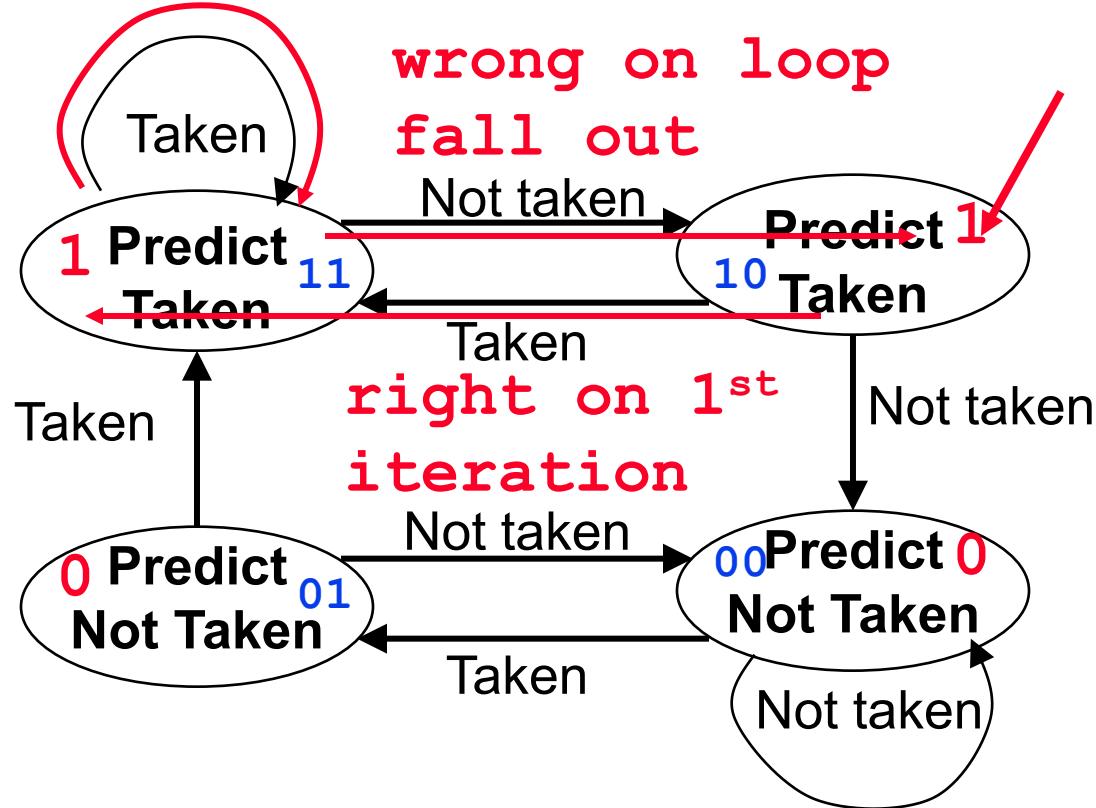
- ❑ A 1-bit predictor will be incorrect twice when not taken
 - Assume `predict_bit = 0` to start (indicating branch not taken) and loop control is at the bottom of the loop code
 1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (`predict_bit = 1`)
 2. As long as branch is taken (looping), prediction is correct
 3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (`predict_bit = 0`)
 - ❑ For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

Loop: 1st loop instr
2nd loop instr
.
.
.
last loop instr
`bne $1,$2,Loop`
fall out instr

2-bit Predictors

- ❑ A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed

right 9 times



Loop: 1st loop instr
2nd loop instr
.
. .
last loop instr
bne \$1,\$2,Loop
fall out instr

- ❑ BHT also stores the initial FSM state

Dealing with Exceptions

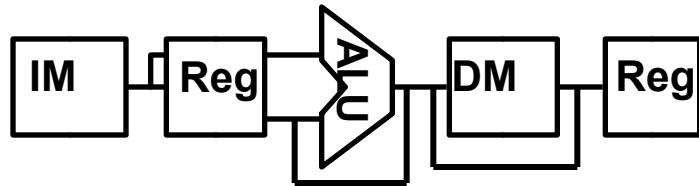
- ❑ Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
 - R-type arithmetic overflow
 - Trying to execute an undefined instruction
 - An I/O device request
 - An OS service request (e.g., a page fault, TLB exception)
 - A hardware malfunction
- ❑ The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- ❑ The software (OS) looks at the cause of the exception and “deals” with it

Two Types of Exceptions

- ❑ Interrupts – asynchronous to program execution
 - caused by **external events**
 - may be handled **between** instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
 - simply suspend and resume user program

- ❑ Traps (Exception) – synchronous to program execution
 - caused by **internal events**
 - condition must be remedied by the trap handler for **that** instruction, so must stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
 - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

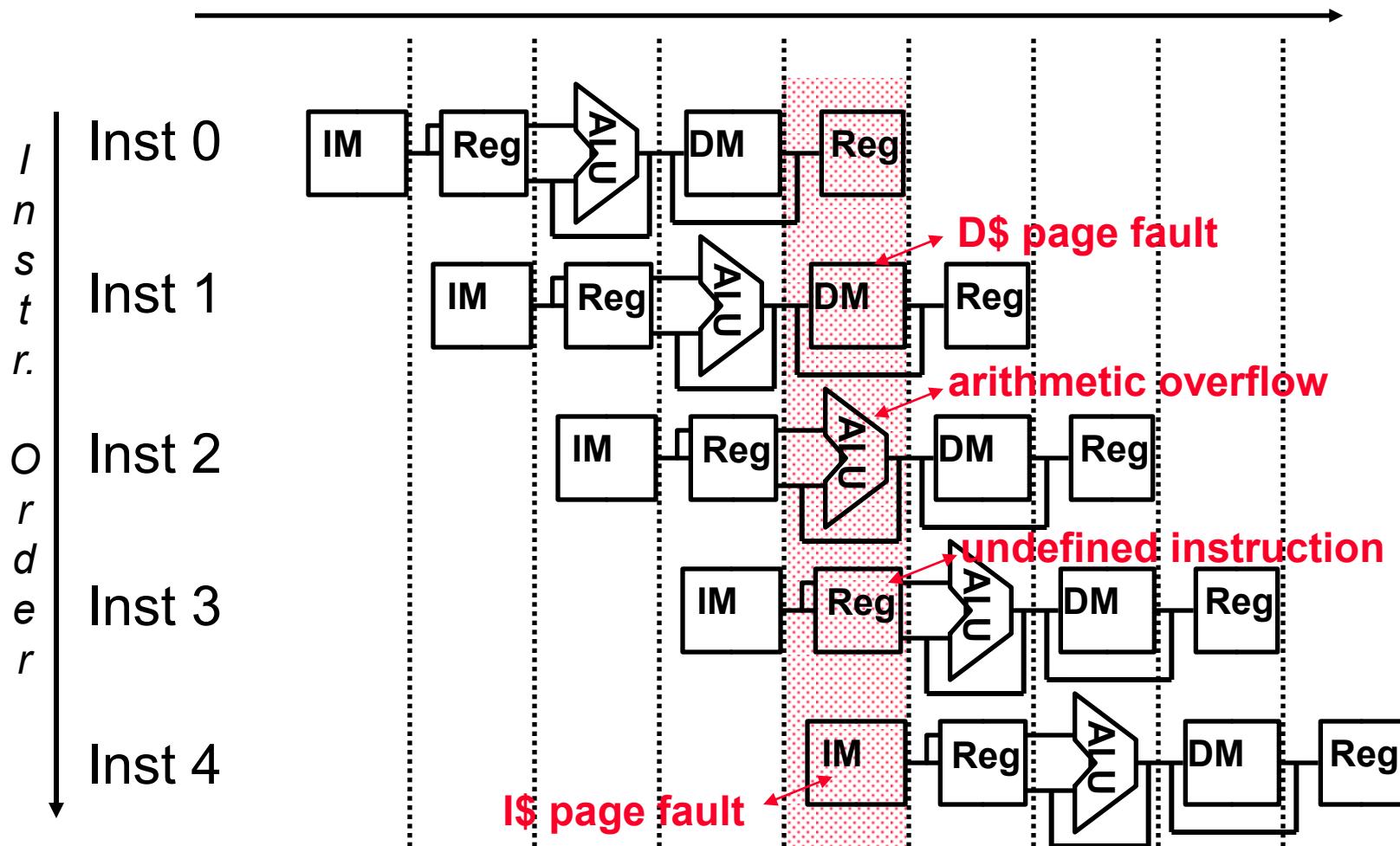
Where in the Pipeline Exceptions Occur



| | Stage(s)? | Synchronous? |
|-------------------------|-----------|--------------|
| ❑ Arithmetic overflow | EX | yes |
| ❑ Undefined instruction | ID | yes |
| ❑ TLB or page fault | IF, MEM | yes |
| ❑ I/O service request | any | no |
| ❑ Hardware malfunction | any | no |

- ❑ Beware that multiple exceptions can occur simultaneously in a *single* clock cycle

Multiple Simultaneous Exceptions

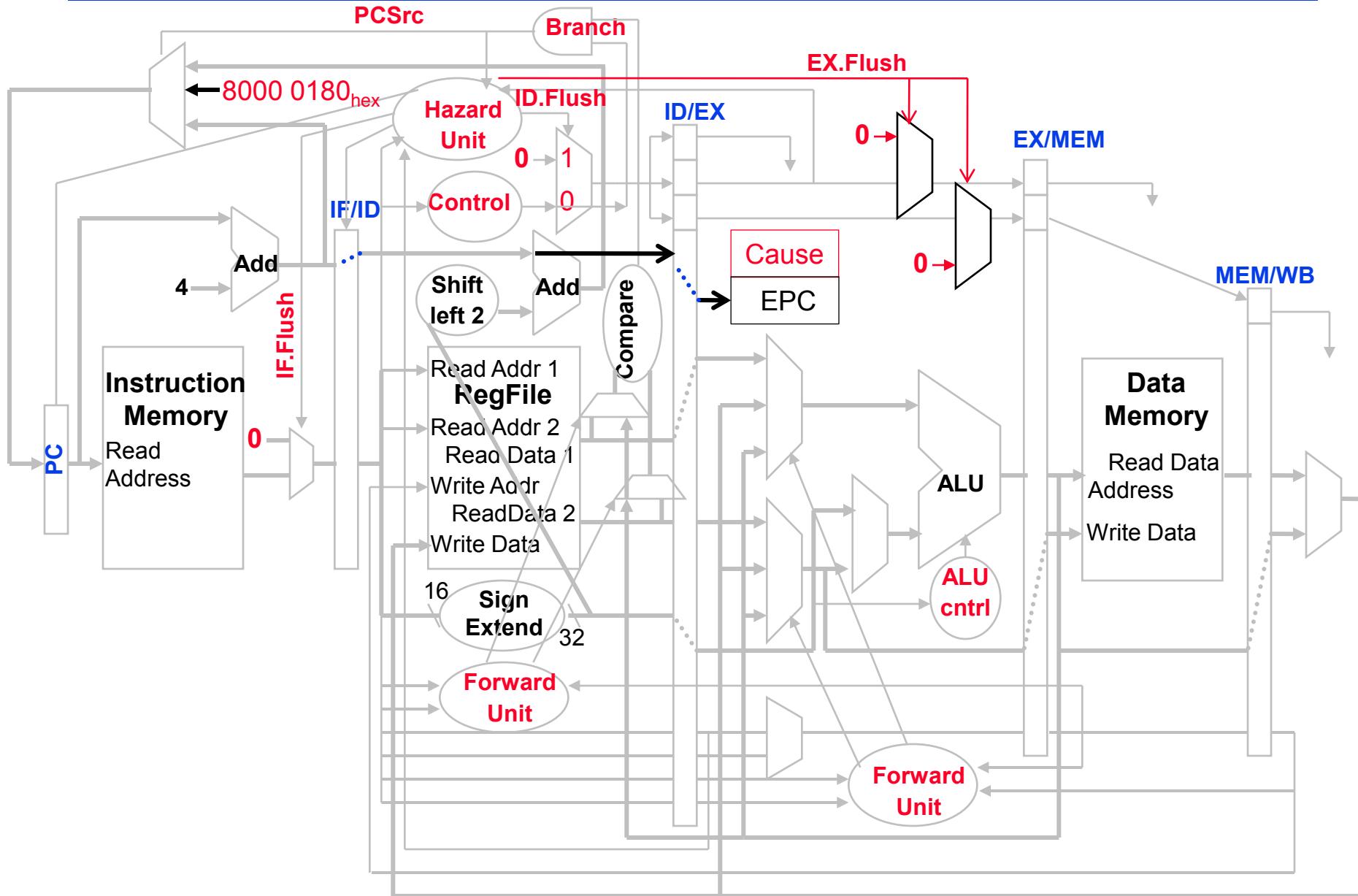


- ❑ Hardware sorts the exceptions so that the earliest instruction is the one interrupted first

Additions to MIPS to Handle Exceptions (Fig 6.42)

- ❑ Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (`CauseWrite`)
- ❑ EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (`EPCWrite`)
 - Exception software must match exception to instruction
- ❑ A way to load the PC with the address of the exception handler
 - Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., $8000\ 0180_{hex}$ for arithmetic overflow)
- ❑ A way to flush offending instruction and the ones that follow it

Datapath with Controls for Exceptions



Summary

- ❑ All modern day processors use pipelining for performance (a CPI of 1 and fast a CC)
- ❑ Pipeline clock rate limited by **slowest** pipeline stage – so designing a balanced pipeline is important
- ❑ Must detect and resolve hazards
 - Structural hazards – resolved by designing the pipeline correctly
 - Data hazards
 - Stall (impacts CPI)
 - Forward (requires hardware support)
 - Control hazards – put the branch decision hardware in as early a stage in the pipeline as possible
 - Stall (impacts CPI)
 - Delay decision (requires compiler support)
 - Static and **dynamic prediction** (requires hardware support)
- ❑ Pipelining complicates exception handling

Next Lecture and Reminders

- Next lecture
 - Multiple issue processor

- Labs:
 - This weeks task:
 - Demonstrate a functional simulation of Execute stage, pipe line registers and the ability to write to the register file (not nessisarily via the rest of the pipeline).
 - Control logic is likely to be the part you spend the most time.
 - If you want more than a pass mark, you should be ready to get ahead of the lab material.

ECE4074

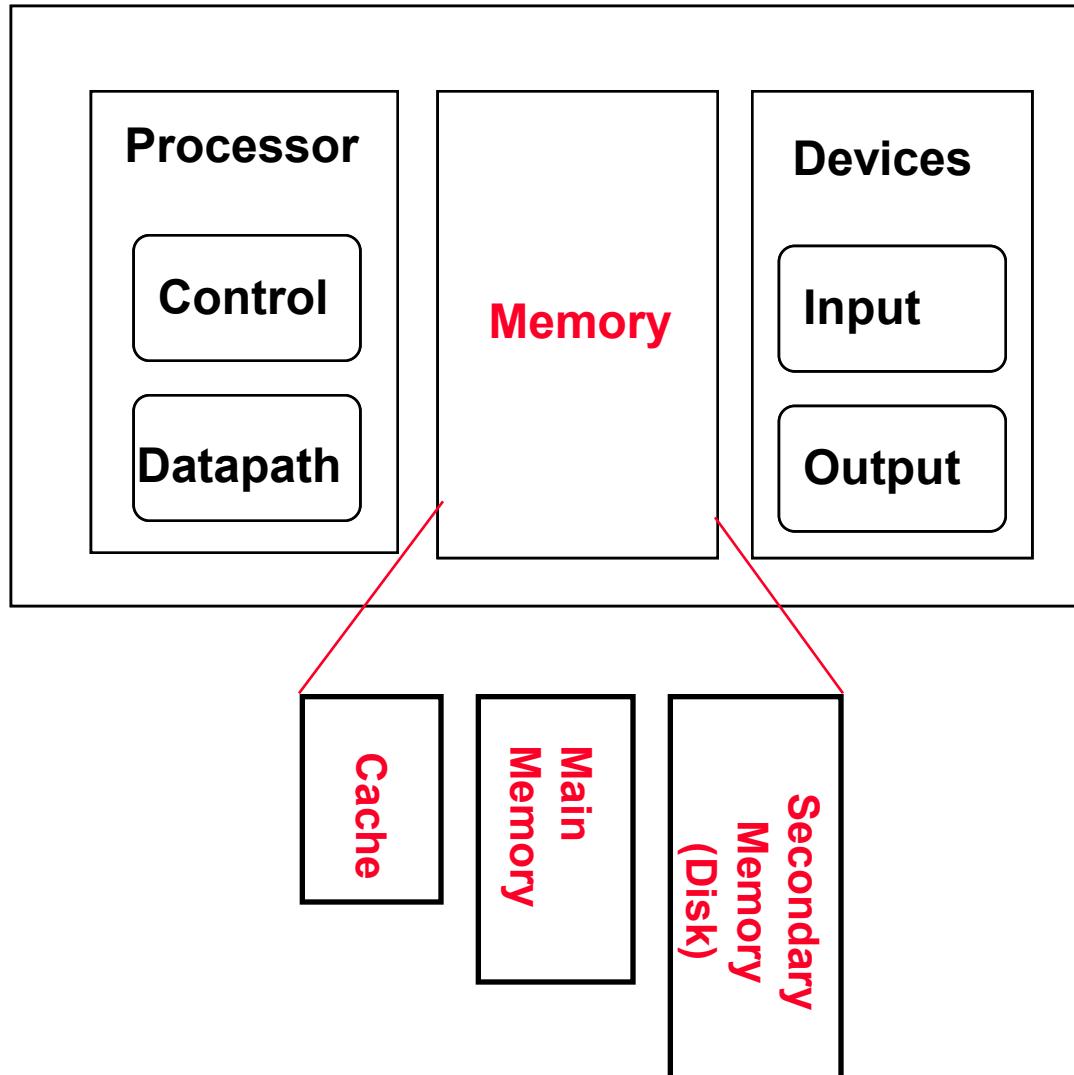
Computer Architecture

Semester 2 2014

Chapter 5A: Exploiting the Memory Hierarchy, Part 1

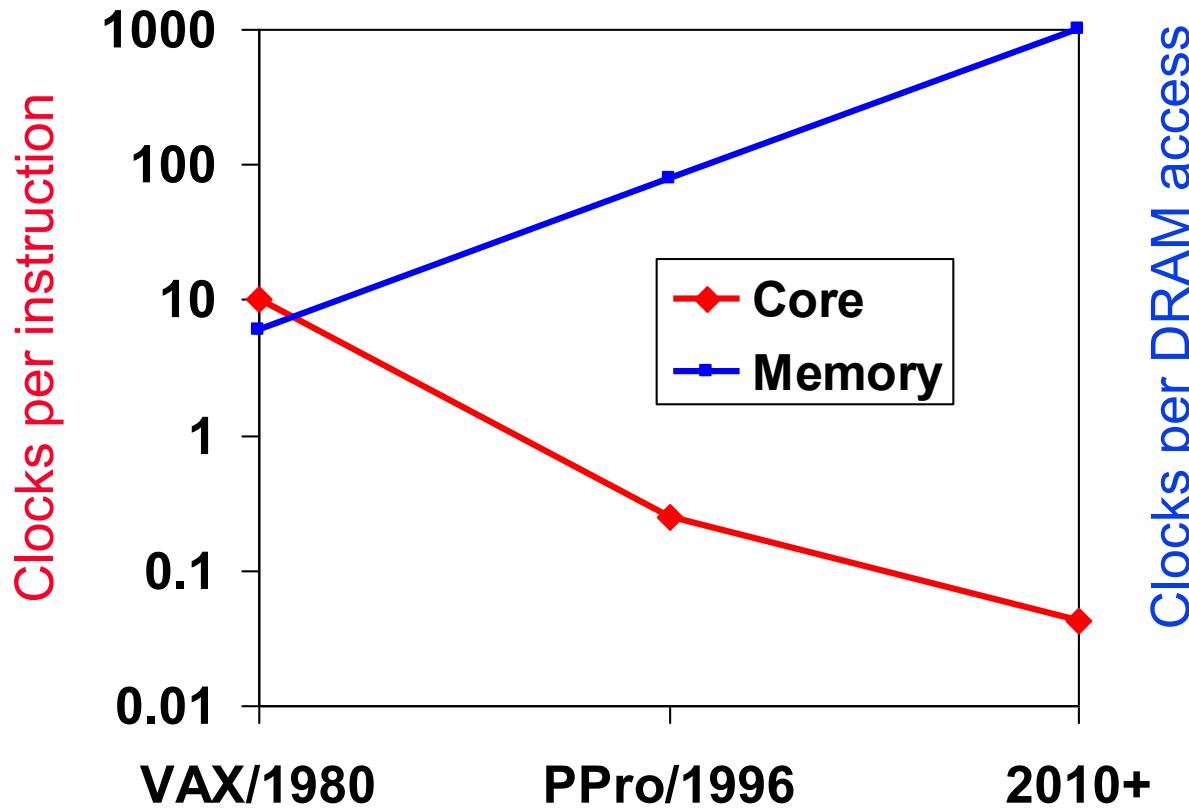
[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: Major Components of a Computer



The “Memory Wall”

- Processor vs DRAM speed disparity continues to grow



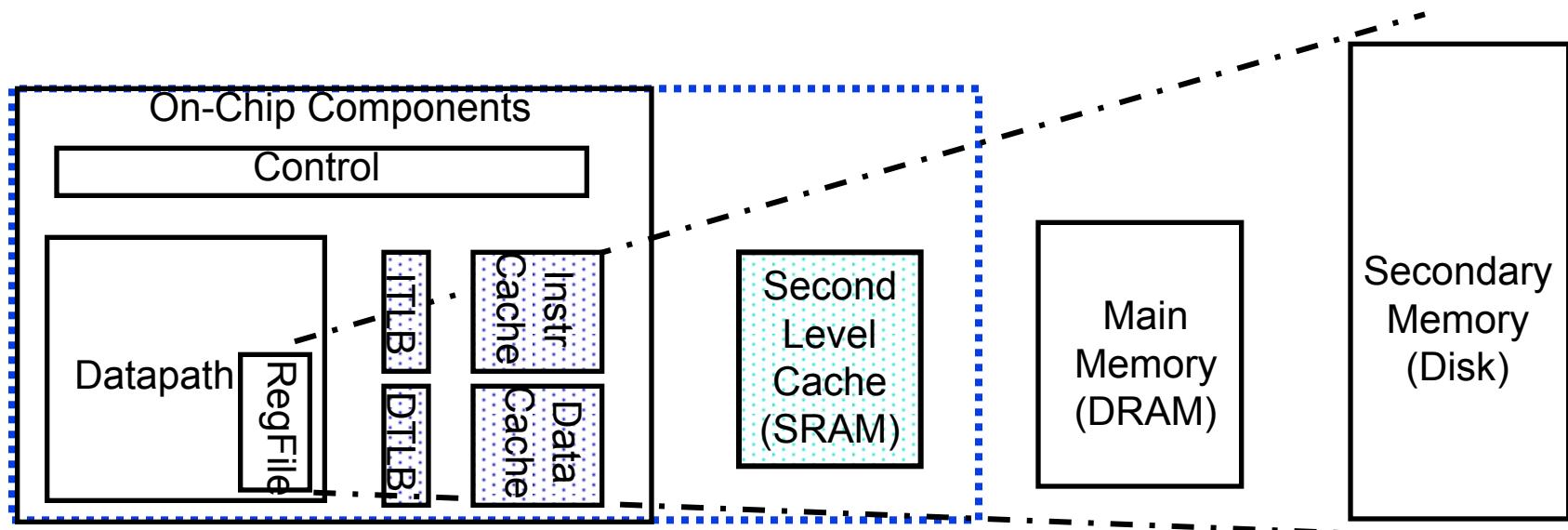
- Good memory hierarchy (cache) design is increasingly important to overall performance

The Memory Hierarchy Goal

- ❑ Fact: Large memories are slow and fast memories are small
 - Why?
 - Addressing requires MUXs, large addresses = big Muxs = more levels of logic (Fan in)
 - Select lines are distributed to many transistors leading to high capacitance lines and slow rise/fall times of digital signals (Fan out)
- ❑ How do we create a memory that gives the illusion of being large, cheap and fast (most of the time)?
 - With hierarchy
 - With parallelism

A Typical Memory Hierarchy

- ❑ Take advantage of the **principle of locality** to present the user with as much memory as is available in the *cheapest* technology at the speed offered by the *fastest* technology



| | | | | | |
|-------------------------|---------|-------|------|-------|----------|
| Speed (%cycles): | 1/2's | 1's | 10's | 100's | 10,000's |
| Size (bytes): | 100's | 10K's | M's | G's | T's |
| Cost: | highest | | | | lowest |

Memory Hierarchy Technologies

- ❑ Caches use **SRAM** for speed and technology compatibility
 - Fast (typical access times of 0.25 to 2.5 nsec)
 - Low density (6 transistor cells), higher power, expensive (\$2000 to \$5000 per GB in 2008)
 - Static: content will last “forever” (as long as power is left on)
- ❑ Main memory uses **DRAM** for size (density)
 - Slower (typical access times of 50 to 70 nsec)
 - High density (1 transistor cells), lower power, cheaper (\$20 to \$75 per GB in 2008)
 - Dynamic: needs to be “refreshed” regularly (~ every 8 ms)
 - consumes 1% to 2% of the active cycles of the DRAM
 - Addresses divided into 2 halves (row and column)
 - **RAS** or *Row Access Strobe* triggering the row decoder
 - **CAS** or *Column Access Strobe* triggering the column selector

The Memory Hierarchy: Why Does it Work?

□ Temporal Locality (locality in time)

- If a memory location is referenced then it will tend to be referenced again soon
⇒ Keep **most recently accessed** data items closer to the processor

□ Spatial Locality (locality in space)

- If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon
⇒ Move blocks consisting of **contiguous words** closer to the processor

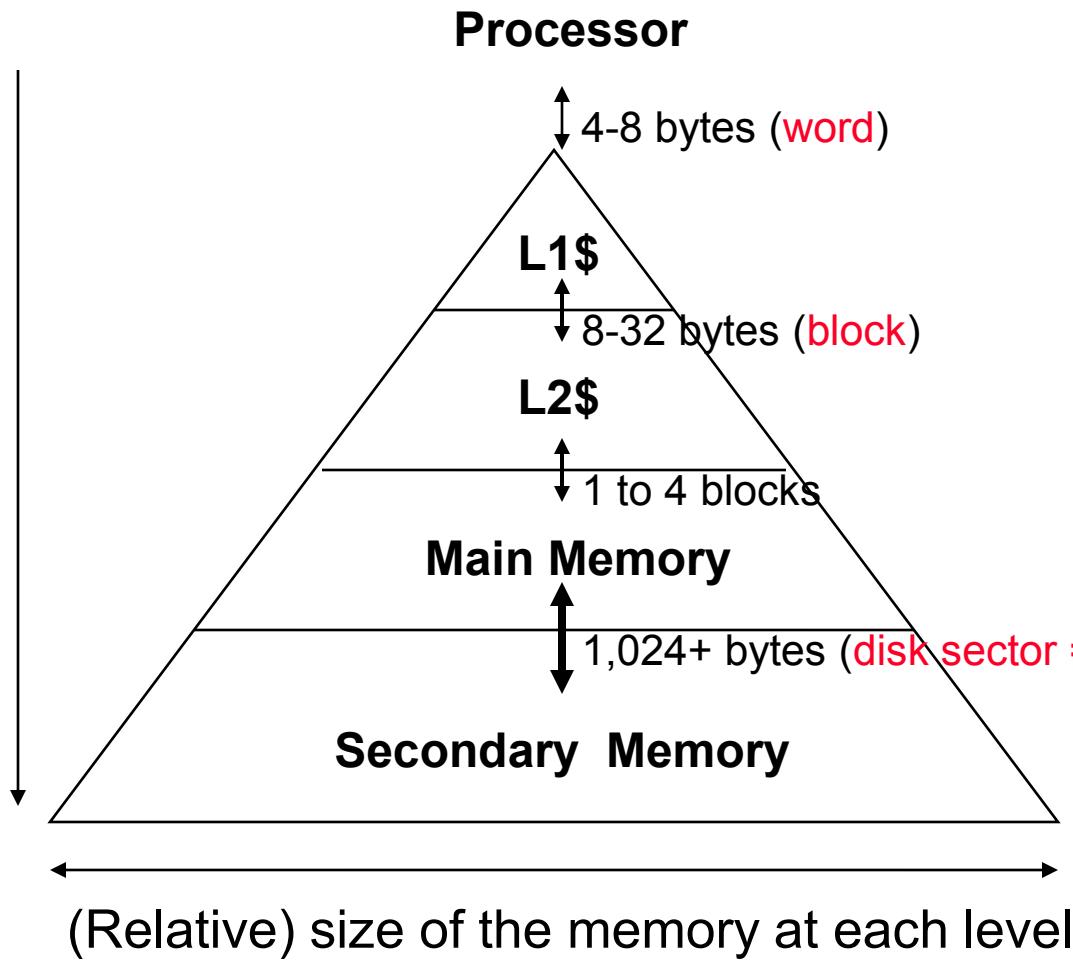
The Memory Hierarchy: Terminology

- ❑ **Block** (or line): the minimum unit of information that is present (or not) in a cache
- ❑ **Hit Rate**: the fraction of memory accesses found in a level of the memory hierarchy
 - **Hit Time**: Time to access that level which consists of
Time to access the block + Time to determine hit/miss
- ❑ **Miss Rate**: the fraction of memory accesses *not* found in a level of the memory hierarchy $\Rightarrow 1 - (\text{Hit Rate})$
 - **Miss Penalty**: Time to replace a block in that level with the corresponding block from a lower level which consists of
Time to access the block in the lower level + Time to transmit that block to the level that experienced the miss + Time to insert the block in that level + Time to pass the block to the requestor

Hit Time << Miss Penalty

Characteristics of the Memory Hierarchy

Increasing distance from the processor in **access time**



Inclusive—
what is in L1\$
is a subset of
what is in L2\$
is a subset of
what is in MM
that is a
subset of is in
SM

How is the Hierarchy Managed?

- registers \leftrightarrow memory

- by compiler (programmer?)

- cache \leftrightarrow main memory

- by the cache controller hardware

- main memory \leftrightarrow disks

- by the operating system (virtual memory)
 - virtual to physical address mapping assisted by the hardware (TLB)
 - by the programmer (files)

Cache Basics

- ❑ Two questions to answer (in hardware):
 - Q1: How do we know if a data item is in the cache?
 - Q2: If it is, how do we find it?

- ❑ Direct mapped
 - Each memory block is mapped to exactly one block in the cache
 - lots of lower level memory blocks must **share** blocks in the cache
 - Address mapping (to answer Q2):
(block address) modulo (# of blocks in the cache)

 - Have a **tag** associated with each cache block that contains the address information (the upper portion of the address) required to identify the block (to answer Q1)

Caching: A Simple First Example

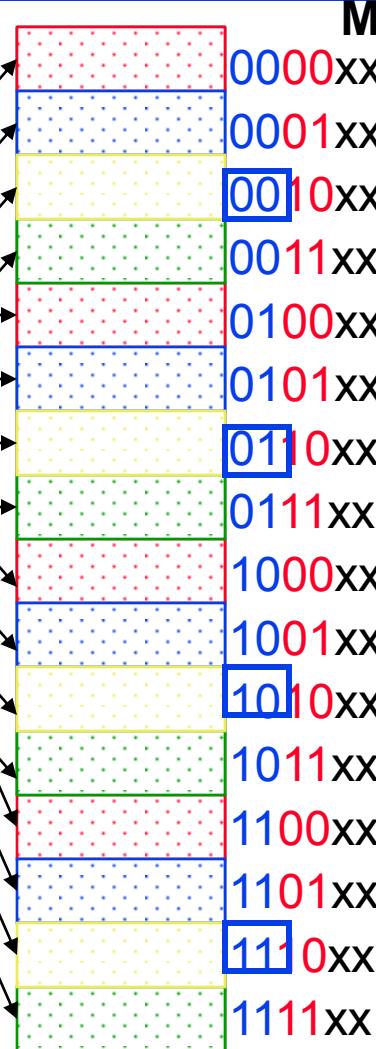
Cache

Index Valid Tag Data

| | | | |
|----|--|----------|---------------|
| 00 | | | red dotted |
| 01 | | | blue dotted |
| 10 | | blue box | yellow dotted |
| 11 | | | green dotted |

Q1: Is it there?

Compare the cache tag to the **high order 2 memory address bits** to tell if the memory block is in the cache



One word blocks
Two low order bits
define the byte in the
word (32b words)

Q2: How do we find it?

Use **next 2 low order memory address bits**
– the **index** – to determine which cache block (i.e., modulo the number of blocks in the cache)

(block address) modulo (# of blocks in the cache)

Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

| 0 miss | |
|--------|--------|
| 00 | Mem(0) |
| 01 | |
| 10 | |
| 11 | |

| 1 miss | |
|--------|--------|
| 00 | Mem(0) |
| 00 | Mem(1) |
| | |
| | |

| 2 miss | |
|--------|--------|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| | |

| 3 miss | |
|--------|--------|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

| 4 miss | |
|------------------|---------------------|
| 00 ⁰¹ | Mem(0) ⁴ |
| 01 | |
| 10 | |
| 11 | |

| 3 hit | |
|-------|--------|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

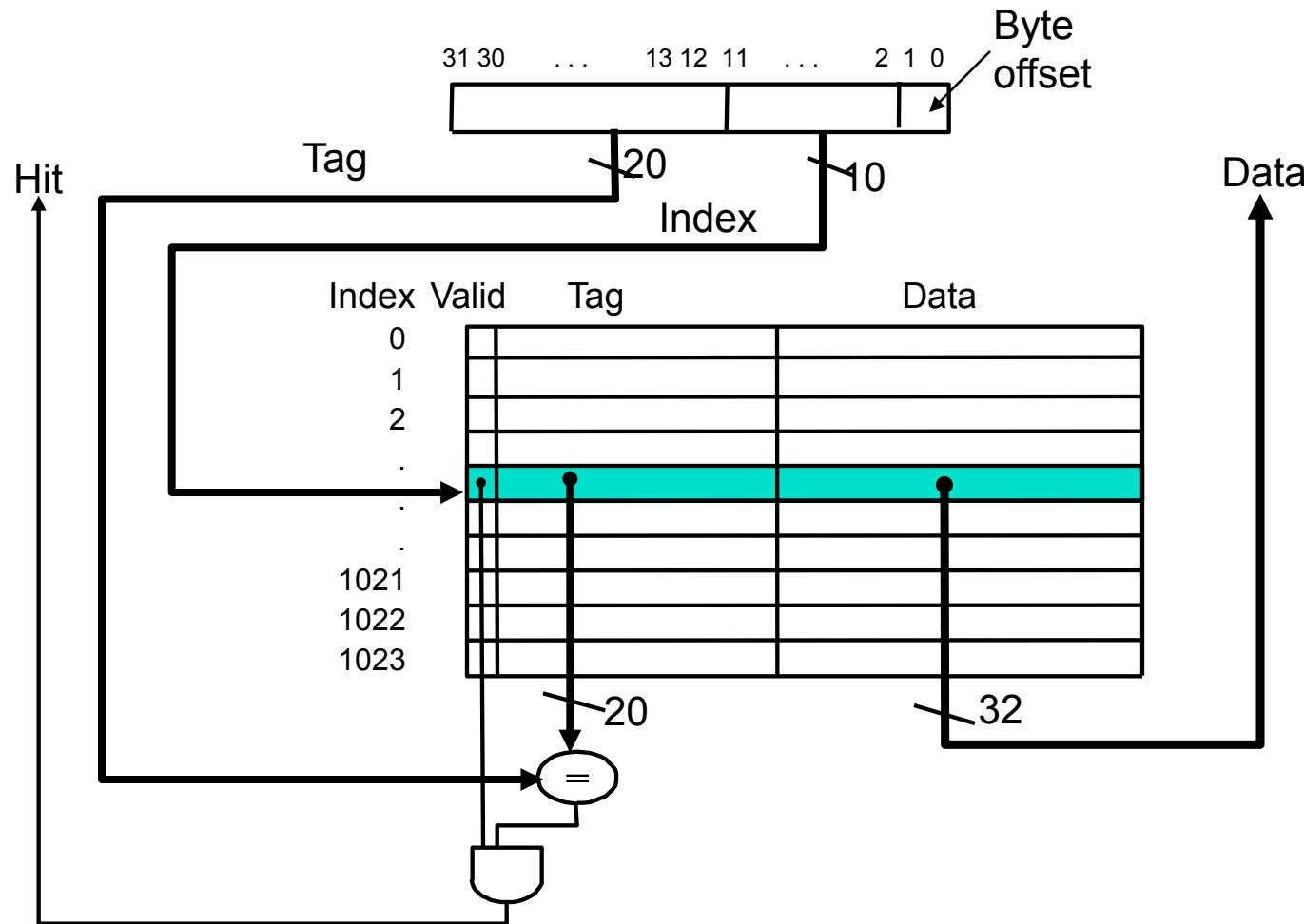
| 4 hit | |
|-------|--------|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

| 15 miss | |
|------------------|----------------------|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 ¹¹ | Mem(3) ¹⁵ |

- 8 requests, 6 misses

MIPS Direct Mapped Cache Example

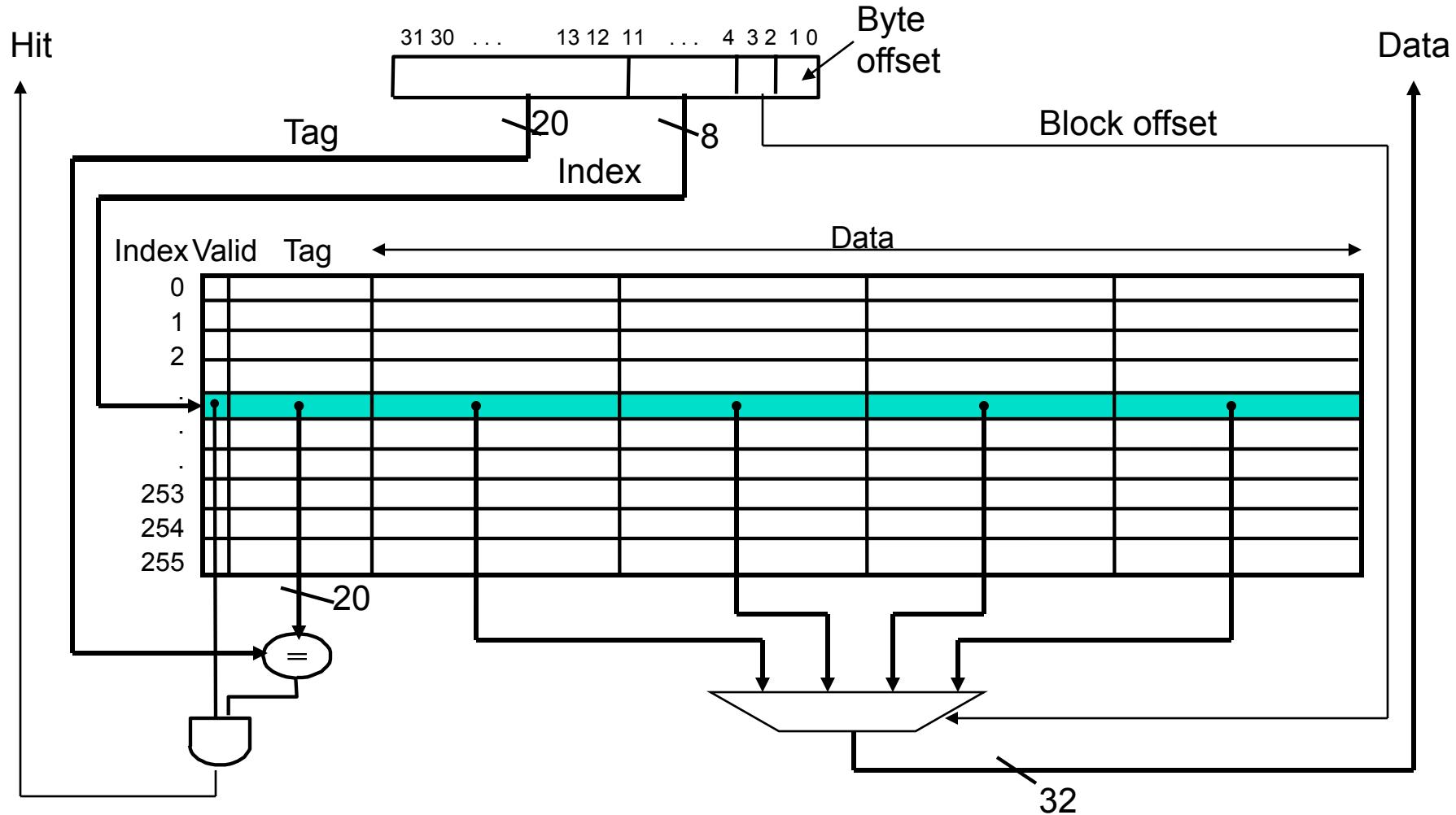
- ❑ One word blocks, cache size = 1K words (or 4KB)



What kind of locality are we taking advantage of?

Multiword Block Direct Mapped Cache

- Four words/block, cache size = 1K words



What kind of locality are we taking advantage of?

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

| 0 miss | | |
|--------|--------|--------|
| 00 | Mem(1) | Mem(0) |
| | | |

| 1 hit | | |
|-------|--------|--------|
| 00 | Mem(1) | Mem(0) |
| | | |

| 2 miss | | |
|--------|--------|--------|
| 00 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

| 3 hit | | |
|-------|--------|--------|
| 00 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

| 4 miss | | |
|--------|--------|--------|
| 01 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

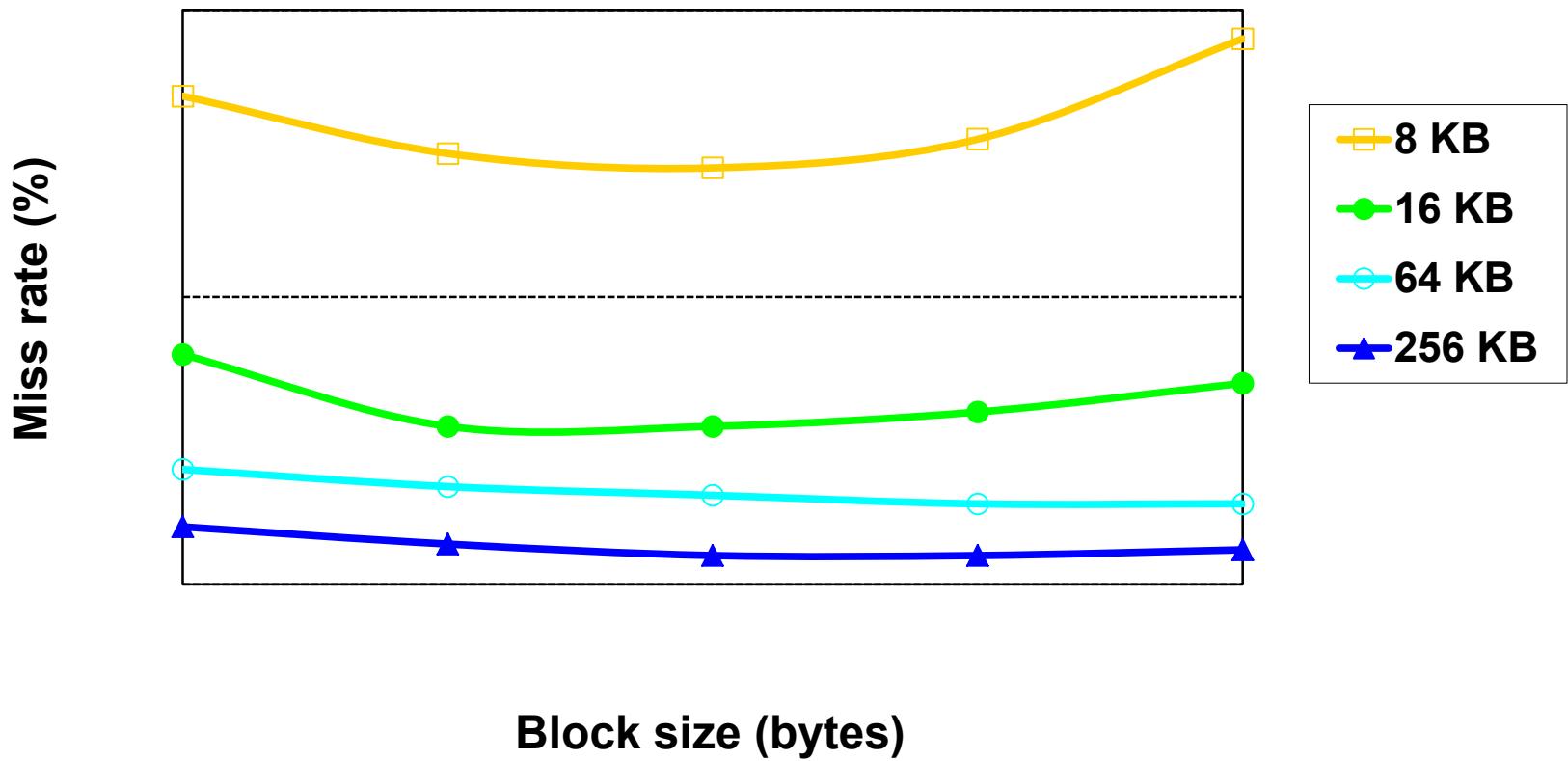
| 3 hit | | |
|-------|--------|--------|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

| 4 hit | | |
|-------|--------|--------|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

| 15 miss | | |
|---------|--------|--------|
| 11 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

- 8 requests, 4 misses

Miss Rate vs Block Size vs Cache Size



- ❑ Miss rate goes up if the block size becomes a significant fraction of the cache size because the number of blocks that can be held in the same size cache is smaller (increasing **capacity** misses)

Cache Field Sizes

- ❑ The number of bits in a cache includes both the storage for data and for the tags
 - 32-bit byte address
 - For a direct mapped cache with 2^n blocks, n bits are used for the index
 - For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block and 2 bits are used to address the byte within the word
- ❑ What is the size of the tag field? $32 - (n+m+2)$ bits
- ❑ The total number of bits in a direct-mapped cache is then
$$2^n \times (\text{block size} + \text{tag field size} + \text{valid field size})$$
- ❑ How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address? $2^{10} \times (4 \times 32 + 18 + 1) = 2^{10} \times 147 = 147\text{Kbits} = 18.4\text{KB}$

Handling Cache Hits

- ❑ Read hits (I\$ and D\$)
 - this is what we want!
- ❑ Write hits (D\$ only)
 - require the cache and memory to be **consistent**
 - always write the data into both the cache block and the next level in the memory hierarchy (**write-through**)
 - writes run at the speed of the next level in the memory hierarchy – so slow! – or can use a **write buffer** and stall only if the write buffer is full
 - allow cache and memory to be **inconsistent**
 - write the data only into the cache block (**write-back** the cache block to the next level in the memory hierarchy when that cache block is “evicted”)
 - need a **dirty** bit for each data cache block to tell if it needs to be written back to memory when it is evicted – can use a **write buffer** to help “buffer” write-backs of dirty blocks

Sources of Cache Misses

- ❑ **Compulsory** (cold start or process migration, first reference):
 - First access to a block, “cold” fact of life, not a whole lot you can do about it. If you are going to run “millions” of instruction, compulsory misses are insignificant
 - Solution: increase block size, more data is written to cache at first miss (increases miss penalty; very large blocks could increase miss rate)
- ❑ **Capacity**:
 - Cache cannot contain all blocks accessed by the program
 - Solution: increase cache size (may increase access time)
- ❑ **Conflict** (collision):
 - Multiple memory locations mapped to the same cache location
 - Solution 1: increase cache size
 - Solution 2: increase associativity (stay tuned) (may increase access time)

Handling Cache Misses (Single Word Blocks)

- ❑ Read misses (I\$ and D\$)
 - stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- ❑ Write misses (D\$ only)
 1. stall the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache (which may involve having to evict a dirty block if using a write-back cache), write the word from the processor to the cache, then let the pipeline resume
 - or
 2. **Write allocate** – just write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall
 - or
 3. **No-write allocate** – skip the cache write (but must invalidate that cache block since it will now hold stale data) and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn't full

Multiword Block Considerations

❑ Read misses (I\$ and D\$)

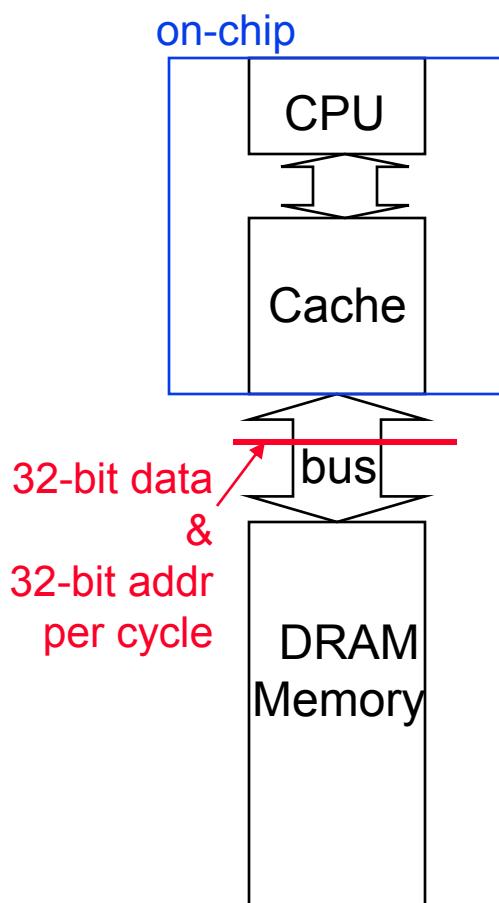
- Processed the same as for single word blocks – a miss returns the entire block from memory
- Miss penalty grows as block size grows
 - **Early restart** – processor resumes execution as soon as the requested word of the block is returned
 - **Requested word first** – requested word is transferred from the memory to the cache (and processor) first
- **Nonblocking cache** – allows the processor to continue to access the cache while the cache is handling an earlier miss

❑ Write misses (D\$)

- If using write allocate must *first* fetch the block from memory and then write the word to the block (or could end up with a “garbled” block in the cache (e.g., for 4 word blocks, a new tag, one word of data from the new block, and three words of data from the old block))

Memory Systems that Support Caches

- ❑ The off-chip interconnect and memory architecture can affect overall system performance in dramatic ways



One word wide organization (one word wide bus and one word wide memory)

- ❑ Assume

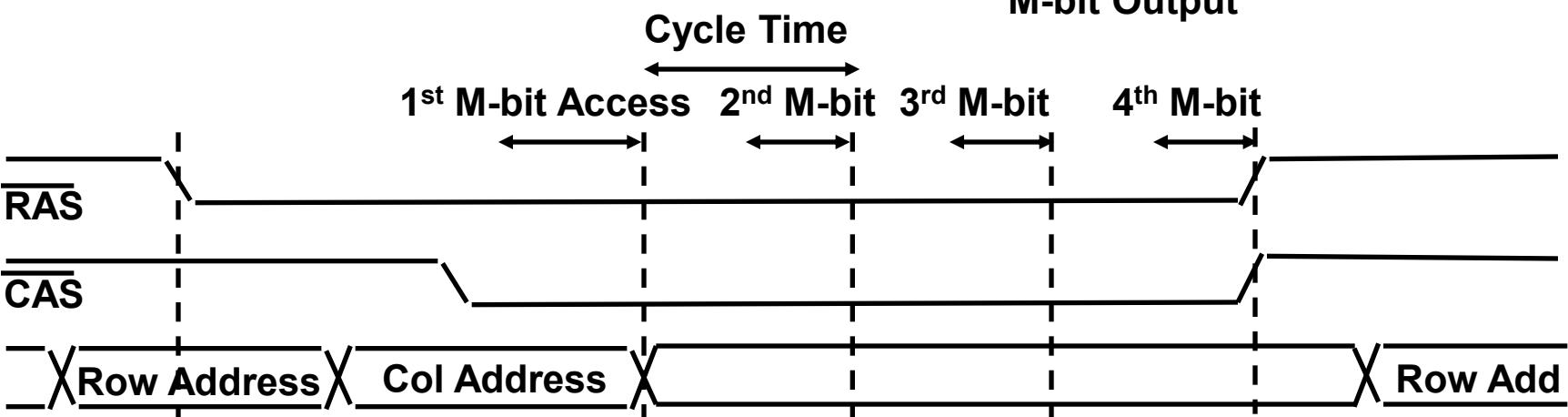
1. 1 memory bus clock cycle to send the addr
2. 15 memory bus clock cycles to get the 1st word in the block from DRAM (row **cycle** time), 5 memory bus clock cycles for 2nd, 3rd, 4th words (column **access** time)
3. 1 memory bus clock cycle to return a word of data

- ❑ Memory-Bus to Cache bandwidth

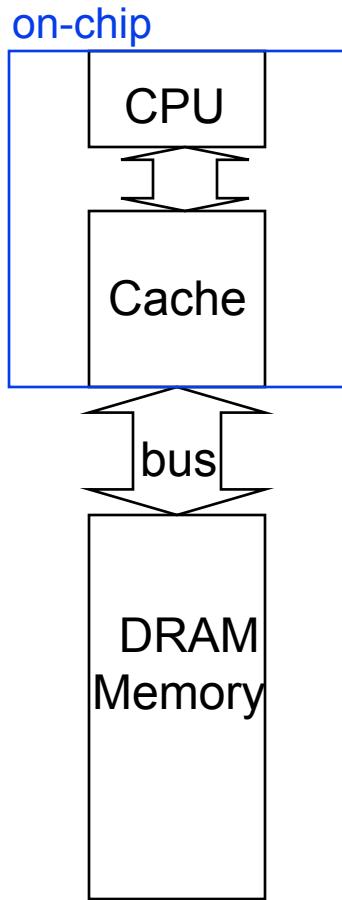
- number of bytes accessed from memory and transferred to cache/CPU per memory bus clock cycle

Review: (DDR) SDRAM Operation

- ❑ After a row is read into the SRAM register
 - Input CAS as the starting “burst” address along with a burst length
 - Transfers a burst of data (**ideally a cache block**) from a series of sequential addr’s within that row
 - The memory bus clock controls transfer of successive words in the burst



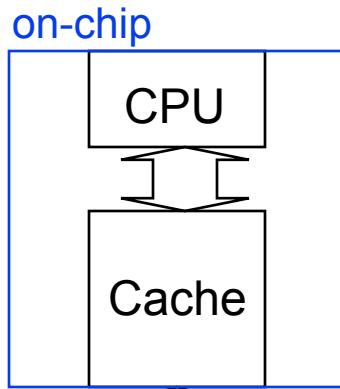
One Word Wide Bus, One Word Blocks



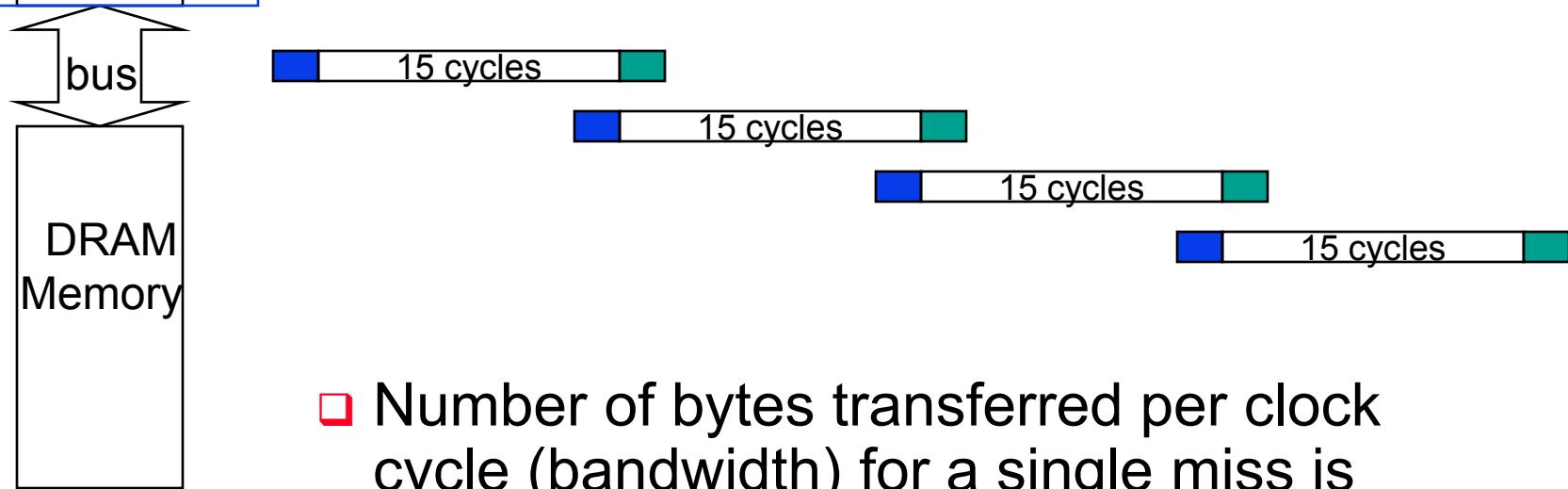
- ❑ If the block size is one word, then for a memory access due to a cache miss, the pipeline will have to stall for the number of cycles required to return one data word from memory
 - 1 memory bus clock cycle to send address
 - 15 memory bus clock cycles to read DRAM
 - 1 memory bus clock cycle to return data
 - 17 total clock cycles miss penalty
- ❑ Number of bytes transferred per clock cycle (bandwidth) for a single miss is
$$4/17 = 0.235 \text{ bytes per memory bus clock cycle}$$

One Word Wide Bus, Four Word Blocks

- ❑ What if the block size is four words and each word is in a different DRAM row?



1 cycle to send 1st address
 $4 \times 15 = 60$ cycles to read DRAM
1 cycles to return last data word
62 total clock cycles miss penalty

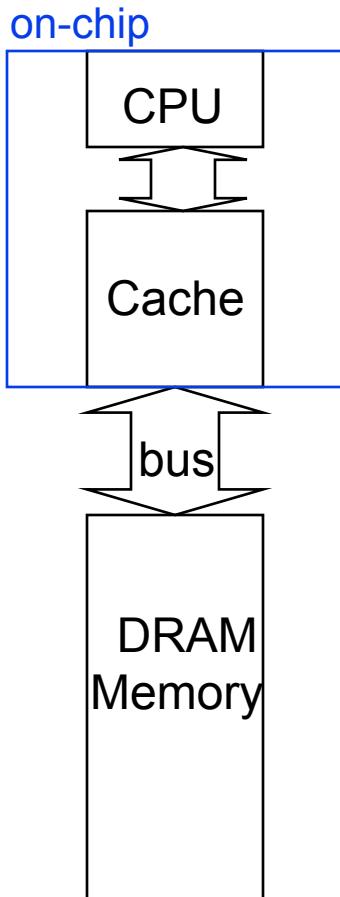


- ❑ Number of bytes transferred per clock cycle (bandwidth) for a single miss is

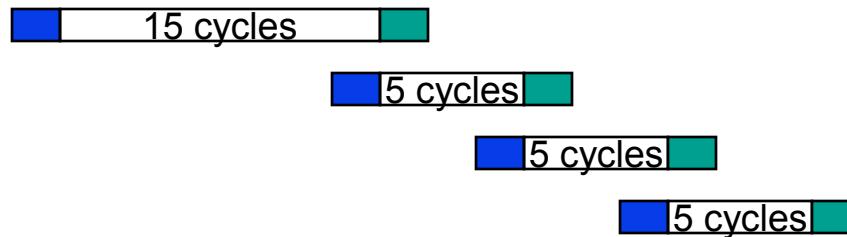
$$(4 \times 4)/62 = 0.258 \text{ bytes per clock}$$

One Word Wide Bus, Four Word Blocks

- ❑ What if the block size is four words and all words are in the same DRAM row?



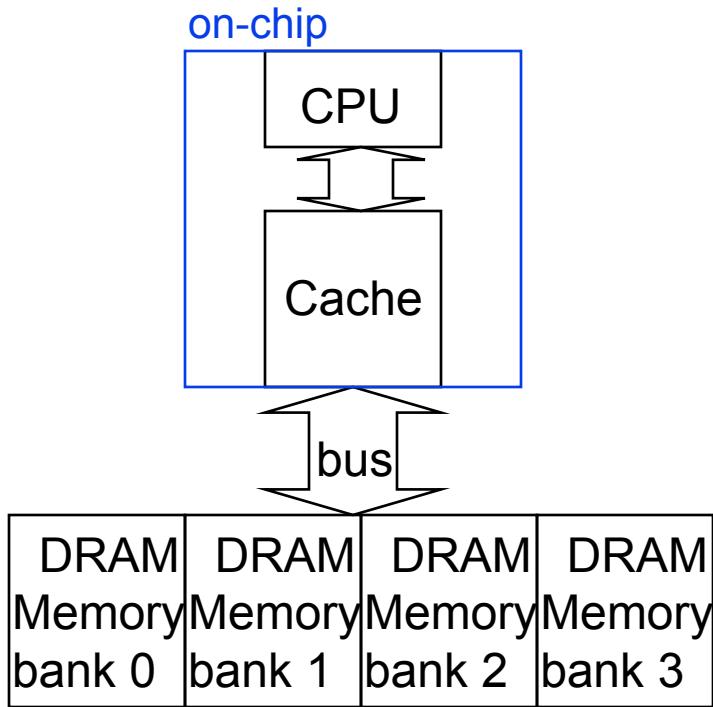
$$\begin{array}{rcl} & 1 & \text{cycle to send 1}^{\text{st}} \text{ address} \\ 15 + 3*5 = 30 & & \text{cycles to read DRAM} \\ & 1 & \text{cycles to return last data word} \\ \hline & 32 & \text{total clock cycles miss penalty} \end{array}$$



- ❑ Number of bytes transferred per clock cycle (bandwidth) for a single miss is $(4 \times 4)/32 = 0.5$ bytes per clock

Interleaved Memory, One Word Wide Bus

- For a block size of four words

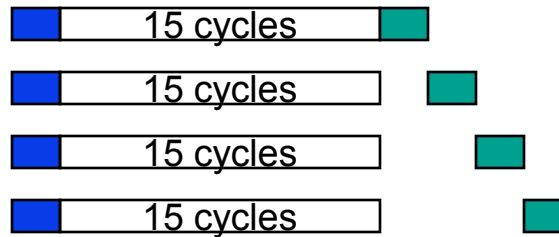


1 cycle to send 1st address

15 cycles to read DRAM banks

$4 * 1 = \underline{4}$ cycles to return last data word

20 total clock cycles miss penalty



- Number of bytes transferred per clock cycle (bandwidth) for a single miss is

$$(4 \times 4)/20 = 0.8 \text{ bytes per clock}$$

DRAM Memory System Summary

- ❑ Its important to match the cache characteristics
 - caches access one block at a time (usually more than one word)

- ❑ with the DRAM characteristics
 - use DRAMs that support fast multiple word accesses, preferably ones that match the block size of the cache

- ❑ with the memory-bus characteristics
 - make sure the memory-bus can support the DRAM access rates and patterns
 - with the goal of increasing the Memory-Bus to Cache bandwidth

End of Lecture

□ Reminders

- Assignment 1 report is due in 3 weeks time
- Assignment 1 is to be demonstrated in the labs
 - Learn to demonstrate your design using signal tap
 - Note the order and starting address of your result matrix and show the write signals to RAM. A random row will be checked.
 - Timing of your minimum clock period should be shown to your demonstrator. (Signal tap may slightly increase this, you may recompile without signal tap for the purposes your report)
 - At the start of the lab you will be given a random matrix with which to initialize your design for testing. You will need to recompile your design with this matrix.

ECE4074

Computer Architecture

Semester 2 2014

Chapter 5A: Exploiting the Memory Hierarchy, Part 2

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review

- ❑ Memory hierarchy exists to provide the appearance of large amount of memory to the user at the speed of small memory.
- ❑ Cache uses the principles of Temporal and Spatial locality to keep the most used data close to the processor in high speed RAM.
 - Temporal Locality:
 - If a program has used a particular piece of memory, it likely will again.
 - Spatial Locality:
 - If a program has used a particular piece of memory, it will likely use memory with an address close to that piece.

Measuring Cache Performance

- ❑ Assuming cache hit costs are included as part of the normal CPU execution cycle, then

$$\begin{aligned} \text{CPU time} &= \text{IC} \times \text{CPI} \times \text{CC} \\ &= \text{IC} \times (\text{CPI}_{\text{ideal}} + \text{Memory-stall cycles}) \times \text{CC} \end{aligned}$$



- ❑ Memory-stall cycles come from cache misses (a sum of read-stalls and write-stalls)

$$\text{Read-stall cycles} = \frac{\text{reads}/\text{program}}{} \times \text{read miss rate} \times \text{read miss penalty}$$

$$\begin{aligned} \text{Write-stall cycles} &= (\frac{\text{writes}/\text{program}}{} \times \text{write miss rate} \times \text{write miss penalty}) \\ &\quad + \text{write buffer stalls} \end{aligned}$$

- ❑ For write-through caches, we can simplify this to
- $$\text{Memory-stall cycles} = \frac{\text{accesses}/\text{program}}{} \times \text{miss rate} \times \text{miss penalty}$$

Impacts of Cache Performance

- ❑ Relative cache penalty increases as processor performance improves (faster clock rate and/or lower CPI)
 - The memory speed is unlikely to improve as fast as processor cycle time. When calculating CPI_{stall} , the cache miss penalty is measured in processor clock cycles needed to handle a miss
 - The lower the CPI_{ideal} , the more pronounced the impact of stalls
- ❑ A processor with a CPI_{ideal} of 2, a 100 cycle miss penalty, 36% load/store instr's, and 2% I\$ and 4% D\$ miss rates
 - Memory-stall cycles = $2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$
 - So $CPI_{stalls} = 2 + 3.44 = 5.44$
 - more than twice the CPI_{ideal} !
- ❑ What if the CPI_{ideal} is reduced to 1? 0.5? 0.25?
- ❑ What if the D\$ miss rate went up 1%? 2%?
- ❑ What if the processor clock rate is doubled (doubling the miss penalty)?

Average Memory Access Time (AMAT)

- ❑ A larger cache will have a longer access time. An increase in hit time will likely add another stage to the pipeline. At some point the increase in hit time for a larger cache will overcome the improvement in hit rate leading to a decrease in performance.
- ❑ Average Memory Access Time (AMAT) is the average to access memory considering both hits and misses

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

- ❑ What is the AMAT for a processor with a 20 psec clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

Reducing Cache Miss Rates #1

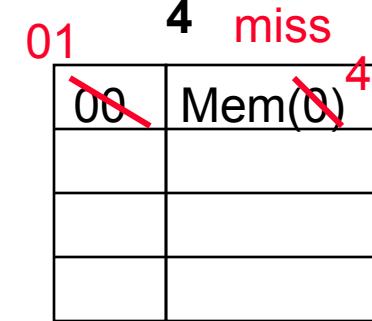
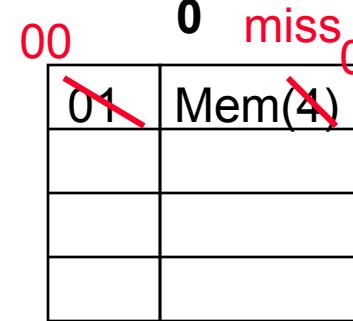
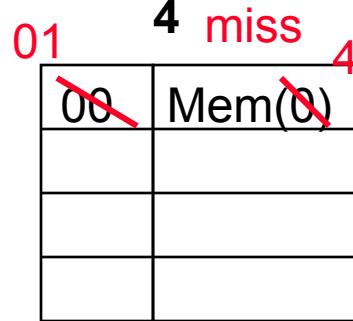
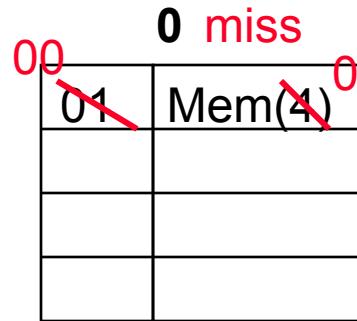
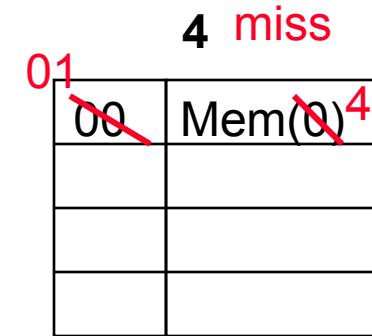
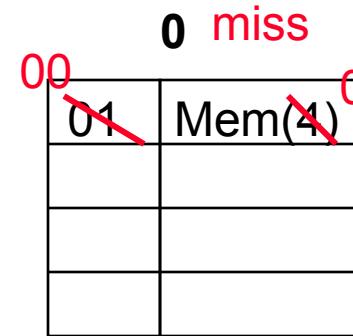
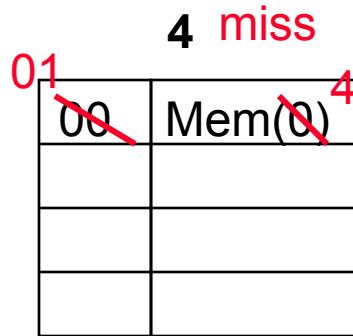
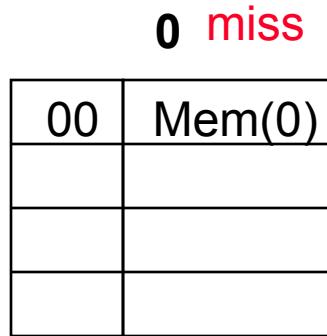
1. Allow more flexible block placement
 - ❑ In a **direct mapped cache** a memory block maps to exactly one cache block
 - ❑ At the other extreme, could allow a memory block to be mapped to **any** cache block – **fully associative cache**
 - ❑ A compromise is to divide the cache into **sets** each of which consists of n “ways” (**n -way set associative**). A memory block maps to a unique set (specified by the index field) and can be placed in any way of that set (so there are n choices)
(block address) modulo (# sets in the cache)

Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4



- 8 requests, 8 misses

- Ping pong effect due to **conflict** misses - two memory locations that map into the same cache block

Set Associative Cache Example

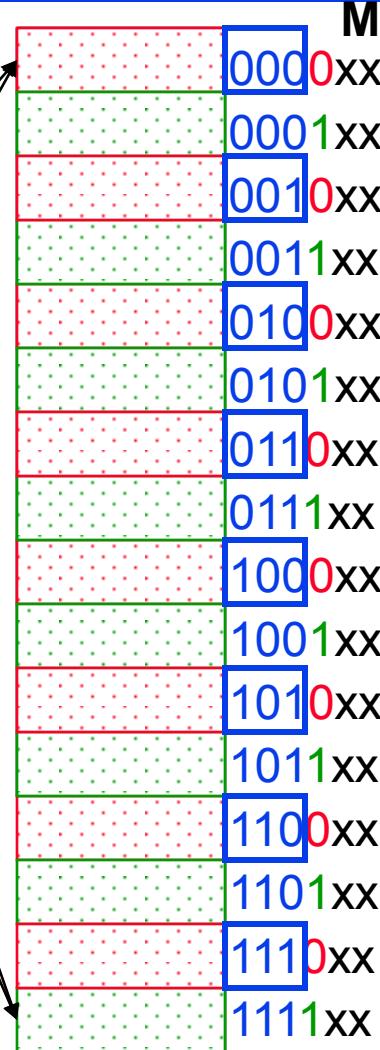
Cache

Way Set V Tag Data

| | 0 | 1 | V | Tag | Data |
|---|---|---|---|-----|------|
| 0 | 0 | 1 | | | |
| 1 | 0 | 1 | | | |

Q1: Is it there?

Compare *all* the cache tags in the set to the high order 3 memory address bits to tell if the memory block is in the cache



Main Memory
One word blocks
Two low order bits
define the byte in the
word (32b words)

Q2: How do we find it?

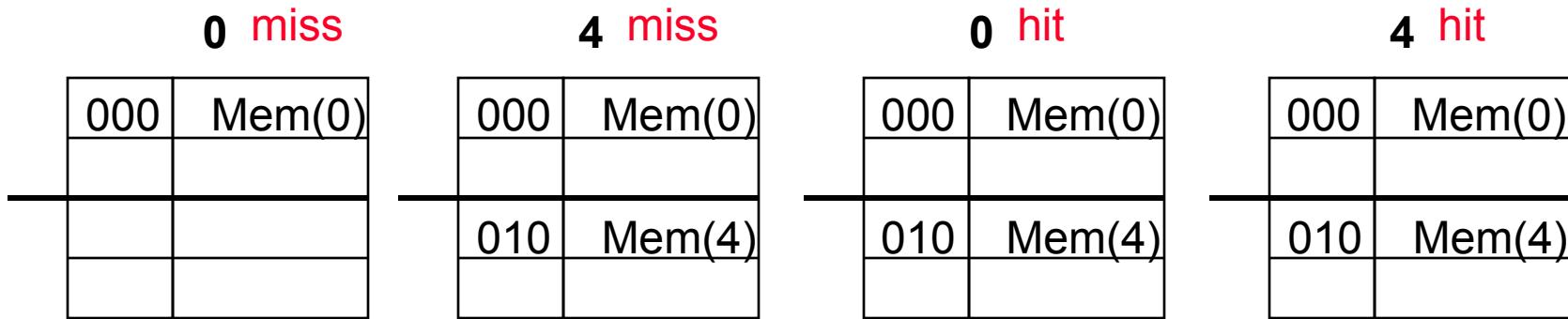
Use **next 1 low order memory address bit** to determine which cache set (i.e., modulo the number of sets in the cache)

Another Reference String Mapping

- ❑ Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

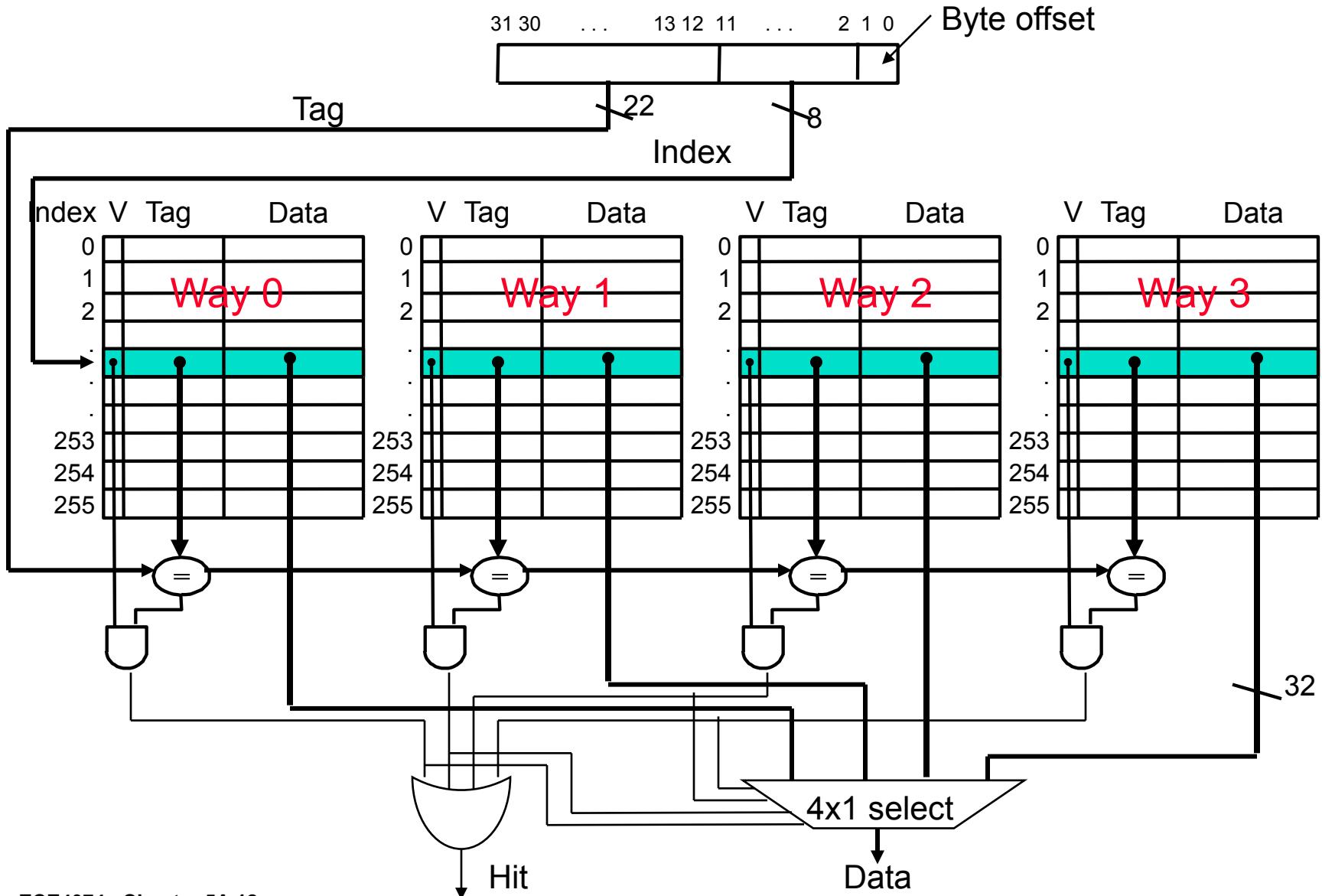


- 8 requests, 2 misses

- ❑ Solves the ping pong effect in a direct mapped cache due to **conflict** misses since now two memory locations that map into the same cache set can co-exist!

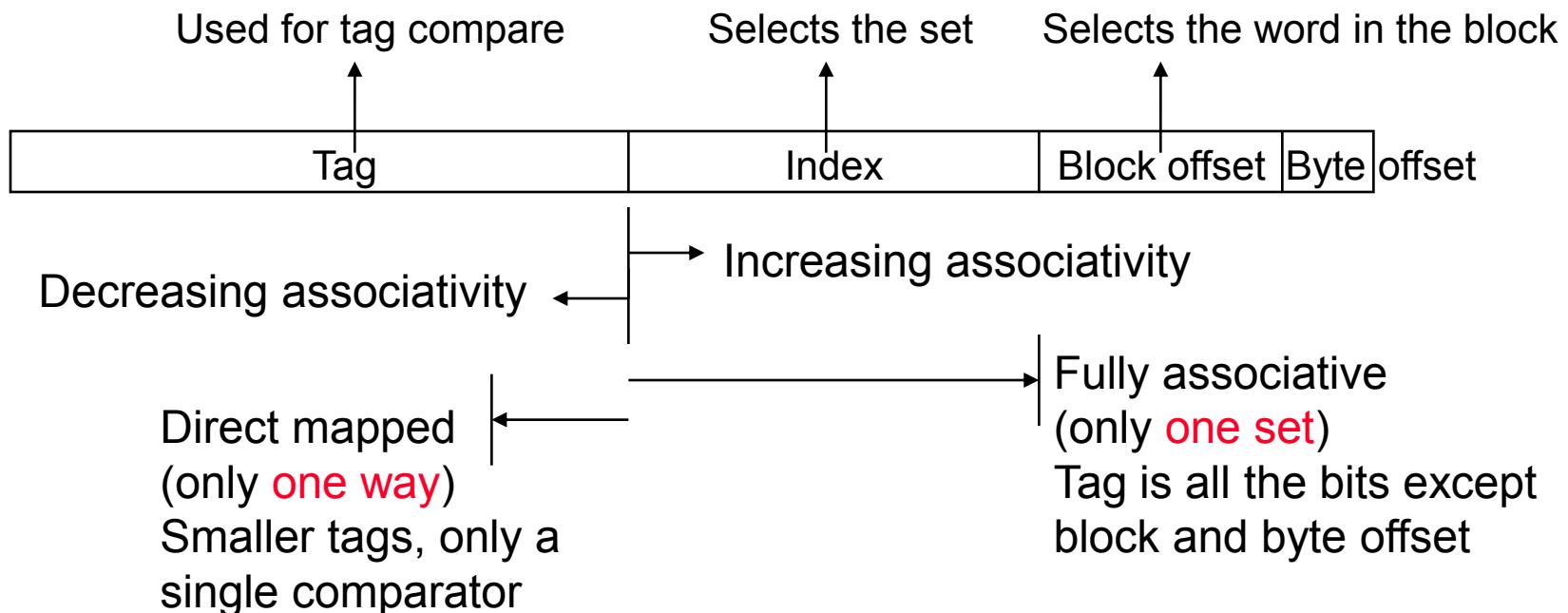
Four-Way Set Associative Cache

- $2^8 = 256$ sets each with four ways (each with one block)



Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number of ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

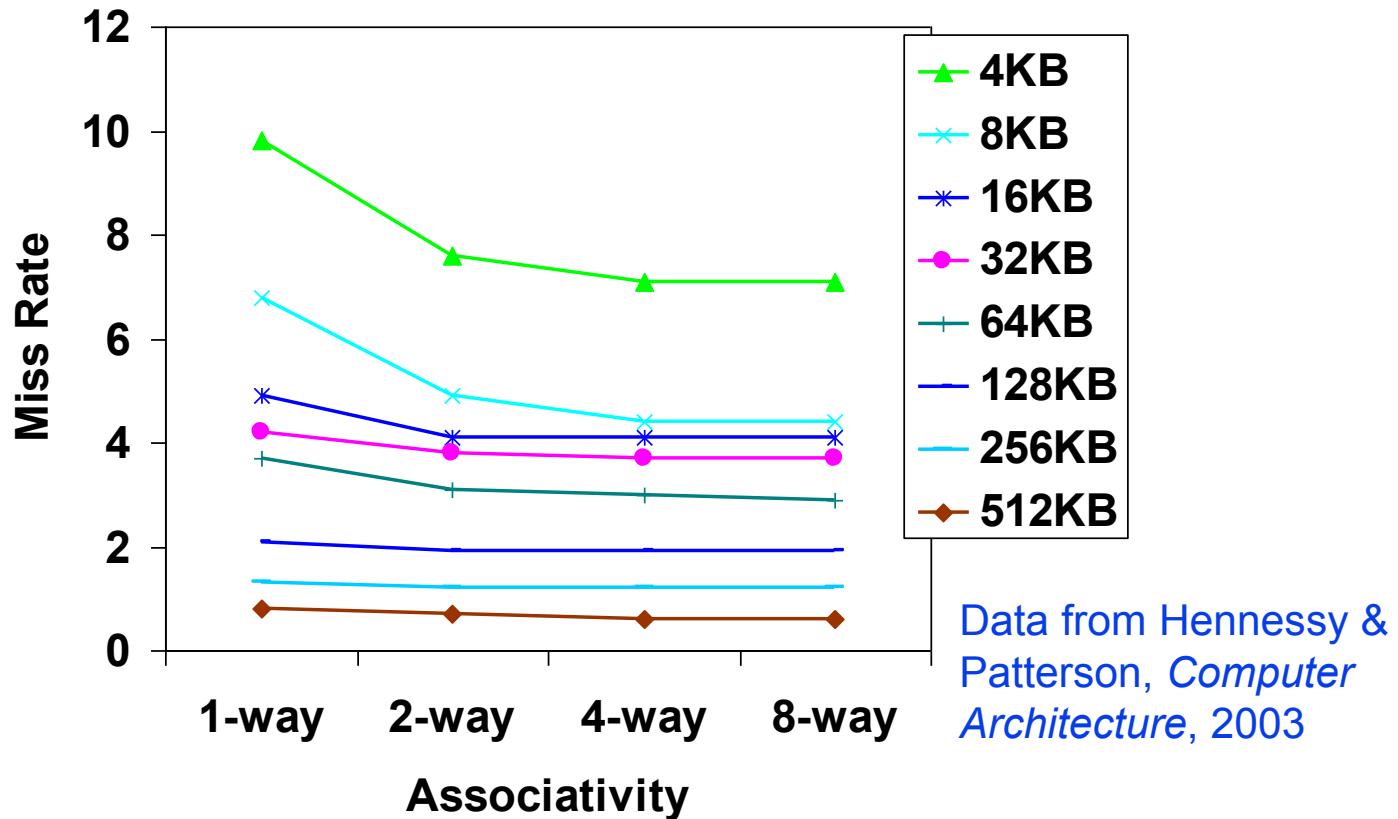


Costs of Set Associative Caches

- ❑ When a miss occurs, which way's block do we pick for replacement?
 - Least Recently Used (LRU): the block replaced is the one that has been unused for the longest time
 - Must have hardware to keep track of when each way's block was used relative to the other blocks in the set
 - For 2-way set associative, takes **one bit per set** → set the bit when a block is referenced (and reset the other way's bit)
- ❑ N-way set associative cache costs
 - N comparators (delay and area)
 - MUX delay (set selection) before data is available
 - Data available **after** set selection (and Hit/Miss decision). In a direct mapped cache, the cache block is available **before** the Hit/Miss decision
 - So its not possible to just assume a hit and continue and recover later if it was a miss

Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation



- Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)

Reducing Cache Miss Rates #2

2. Use multiple levels of caches
 - ❑ With advancing technology have more than enough room on the die for bigger L1 caches or for a second level of caches – normally a **unified** L2 cache (i.e., it holds both instructions and data) and in some cases even a unified L3 cache
 - ❑ For our example, CPI_{ideal} of 2, 100 cycle miss penalty (to main memory) and a 25 cycle miss penalty (to UL2\$), 36% load/stores, a 2% (4%) L1 I\$ (D\$) miss rate, add a 0.5% UL2\$ miss rate

$$\begin{aligned} CPI_{stalls} = & 2 + .02 \times 25 + .36 \times .04 \times 25 + .005 \times 100 + \\ & .36 \times .005 \times 100 = 3.54 \\ & \text{(as compared to 5.44 with no L2$)} \end{aligned}$$

Multilevel Cache Design Considerations

- ❑ Design considerations for L1 and L2 caches are very different
 - Primary cache should focus on **minimizing hit time** in support of a shorter clock cycle
 - Smaller with smaller block sizes
 - Secondary cache(s) should focus on **reducing miss rate** to reduce the penalty of long main memory access times
 - Larger with larger block sizes
 - Higher levels of associativity
- ❑ The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- ❑ For the L2 cache, hit time is less important than miss rate
 - The L2\$ hit time determines L1\$'s miss penalty
 - L2\$ local miss rate $>>$ than the global miss rate

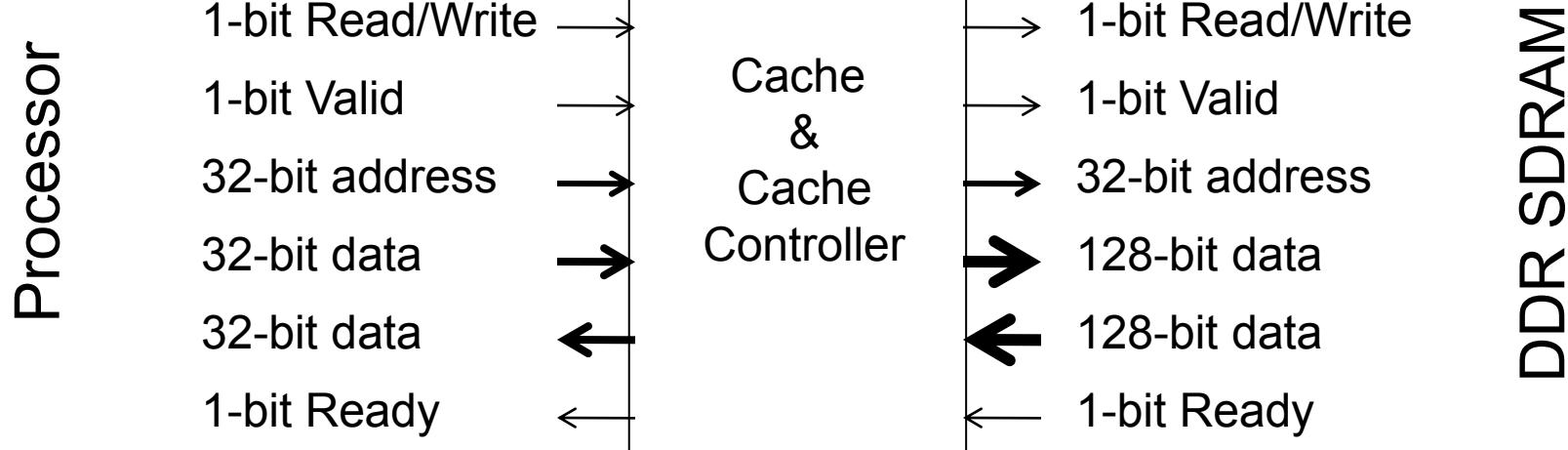
Two Machines' Cache Parameters

| | Intel Nehalem | AMD Barcelona |
|------------------------------|---|---|
| L1 cache organization & size | Split I\$ and D\$; 32KB for each per core; 64B blocks | Split I\$ and D\$; 64KB for each per core; 64B blocks |
| L1 associativity | 4-way (I), 8-way (D) set assoc.; ~LRU replacement | 2-way set assoc.; LRU replacement |
| L1 write policy | write-back, write-allocate | write-back, write-allocate |
| L2 cache organization & size | Unified; 256MB (0.25MB) per core; 64B blocks | Unified; 512KB (0.5MB) per core; 64B blocks |
| L2 associativity | 8-way set assoc.; ~LRU | 16-way set assoc.; ~LRU |
| L2 write policy | write-back | write-back |
| L2 write policy | write-back, write-allocate | write-back, write-allocate |
| L3 cache organization & size | Unified; 8192KB (8MB) shared by cores; 64B blocks | Unified; 2048KB (2MB) shared by cores; 64B blocks |
| L3 associativity | 16-way set assoc. | 32-way set assoc.; evict block shared by fewest cores |
| L3 write policy | write-back, write-allocate | write-back; write-allocate |

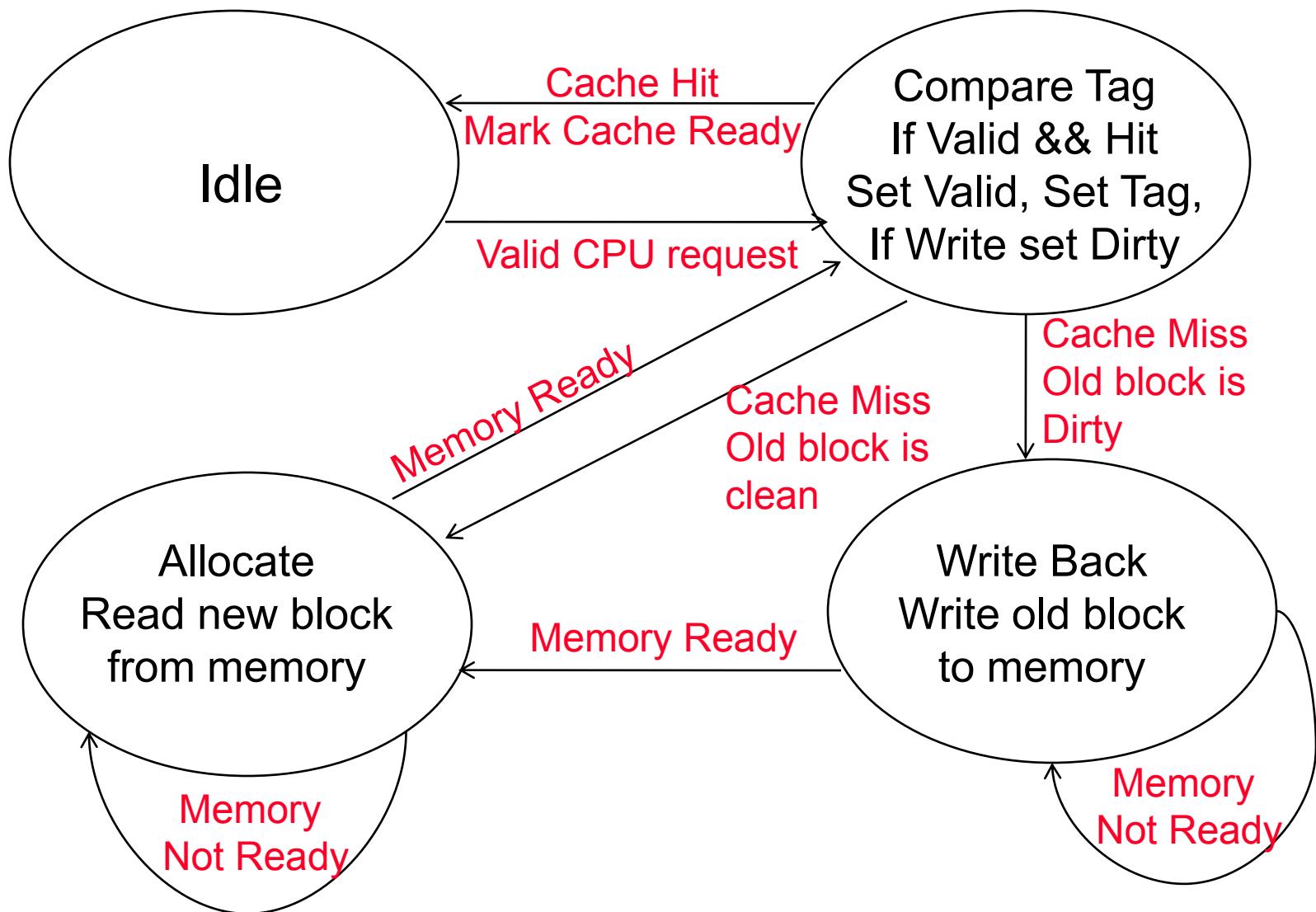
FSM Cache Controller

❑ Key characteristics for a simple L1 cache

- Direct mapped
- Write-back using write-allocate
- Block size of 4 32-bit words (so 16B); Cache size of 16KB (so 1024 blocks)
- 18-bit tags, 10-bit index, 2-bit block offset, 2-bit byte offset, dirty bit, valid bit, LRU bits (if set associative)

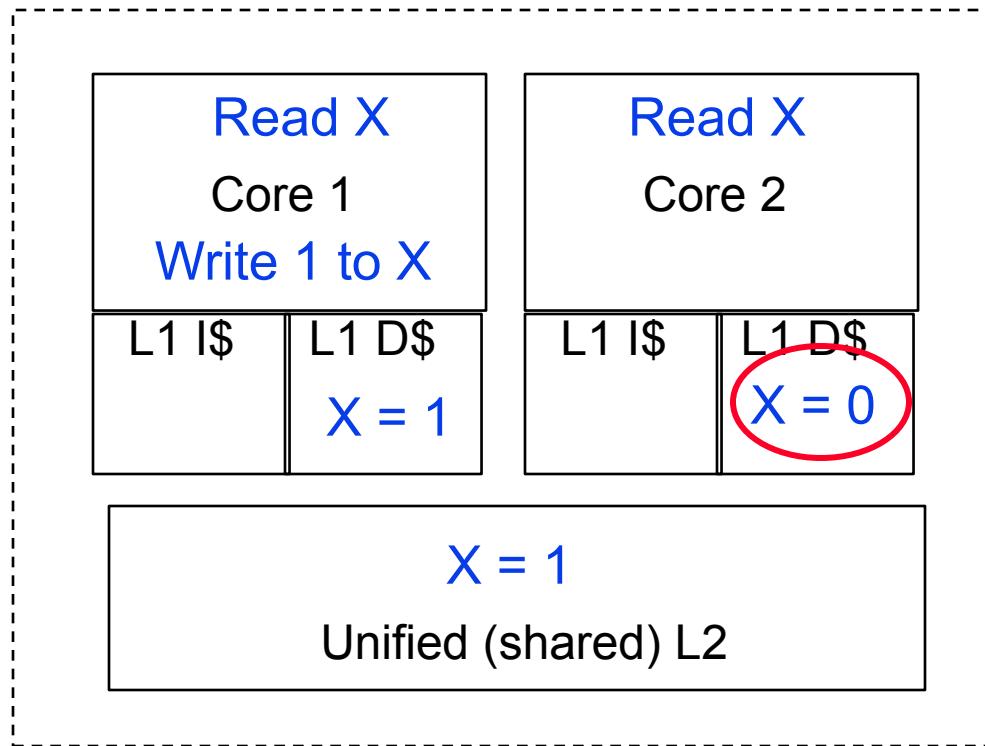


Four State Cache Controller



Cache Coherence in Multicores

- In future multicore processors it's likely that the cores will share a common physical address space, causing a cache coherence problem



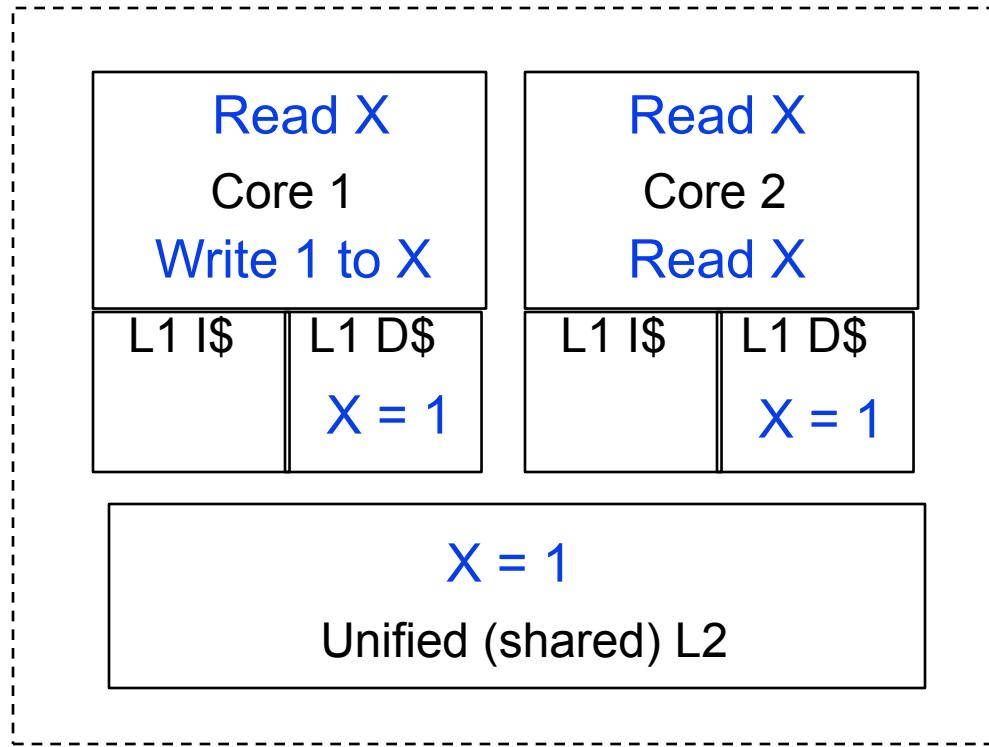
A Coherent Memory System

- ❑ Any read of a data item should return the most recently written value of the data item
 - Coherence – defines **what values** can be returned by a read
 - Writes to the same location are **serialized** (two writes to the same location must be seen in the same order by all cores)
 - Consistency – determines **when** a written value will be returned by a read
- ❑ To enforce coherence, caches must provide
 - **Replication** of shared data items in multiple cores' caches
 - Replication reduces both latency and contention for a read shared data item
 - **Migration** of shared data items to a core's local cache
 - Migration reduced the latency of the access the data and the bandwidth demand on the shared memory (L2 in our example)

Cache Coherence Protocols

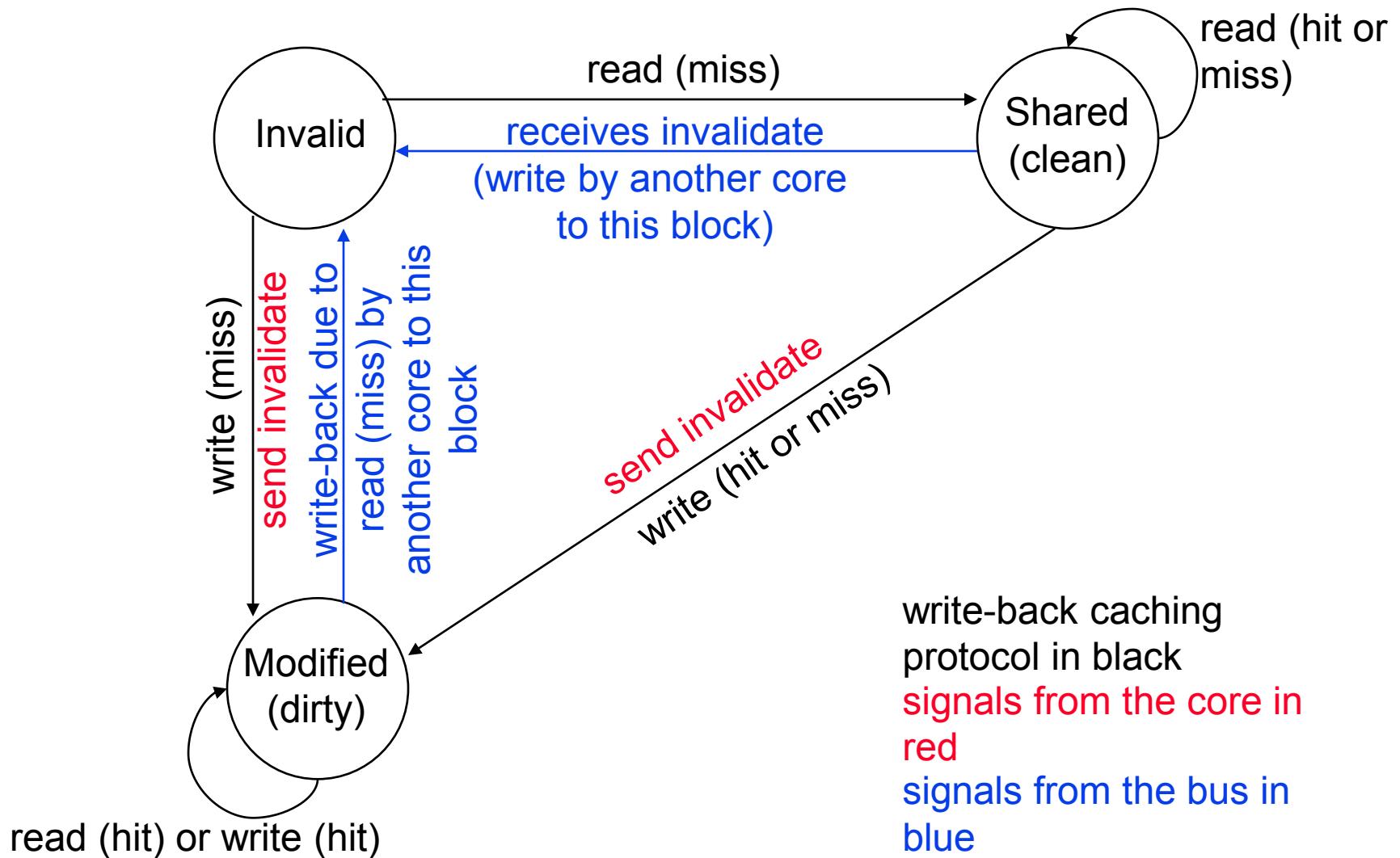
- ❑ Need a hardware protocol to ensure cache coherence the most popular of which is **snooping**
 - The cache controllers monitor (snoop) on the broadcast medium (e.g., bus) with duplicate address tag hardware (so they don't interfere with core's access to the cache) to determine if their cache has a copy of a block that is requested
- ❑ Write invalidate protocol – **writes** require exclusive access and **invalidate** *all* other copies
 - Exclusive access ensure that no other readable or writable copies of an item exists
- ❑ If two processors attempt to write the same data at the same time, one of them wins the race causing the other core's copy to be invalidated. For the other core to complete, it must obtain a new copy of the data which must now contain the updated value – thus enforcing **write serialization**

Example of Snooping Invalidation



- ❑ When the second miss by Core 2 occurs, Core 1 responds with the value canceling the response from the L2 cache (and also updating the L2 copy)

A Write-Invalidate CC Protocol

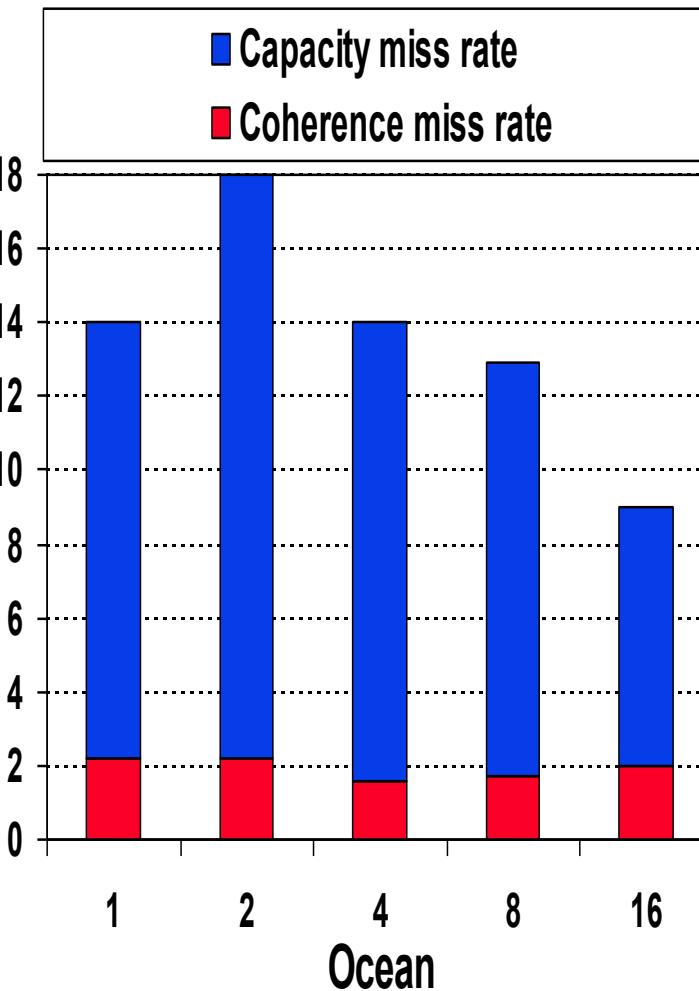
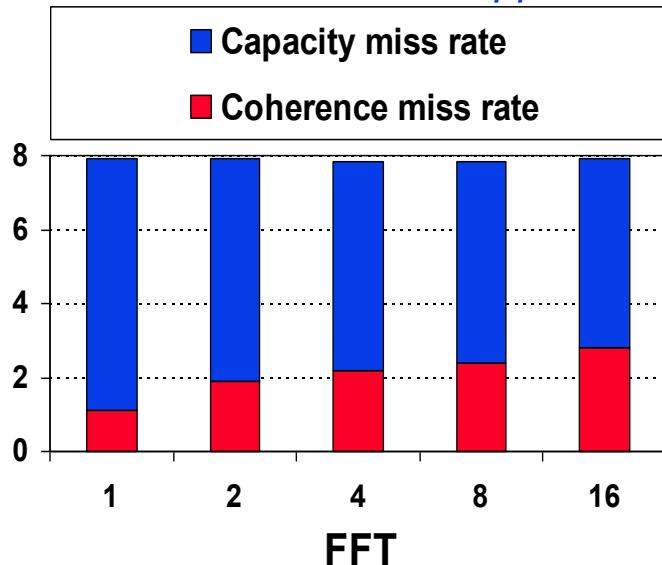


Data Miss Rates

- ❑ Shared data has lower spatial and temporal locality
 - Share data misses often dominate cache behavior even though they may only be 10% to 40% of the data accesses

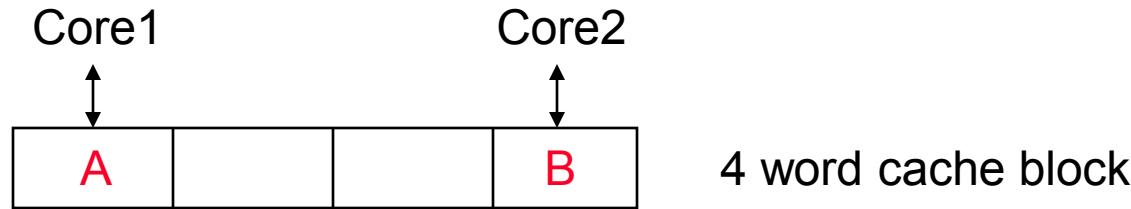
64KB 2-way set associative data cache with 32B blocks

Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*



Block Size Effects

- ❑ Writes to one word in a multi-word block mean that the full block is invalidated
- ❑ Multi-word blocks can also result in **false sharing**: when two cores are writing to two different variables that happen to fall in the same cache block
 - With write-invalidate false sharing increases cache miss rates



- ❑ Compilers can help reduce false sharing by allocating highly correlated data to the same cache block

Other Coherence Protocols

- ❑ There are many variations on cache coherence protocols
- ❑ Another write-invalidate protocol used in the Pentium 4 (and many other processors) is **MESI** with four states for each block:
 - **Modified** – data in block is dirty
 - **Exclusive** – only one copy of the shared data is cached; memory has an up-to-date copy
 - Since there is only one copy of the block, write hits don't need to send invalidate signal
 - **Shared** – multiple copies of the shared data may be cached (i.e., data permitted to be cached with more than one processor); memory has an up-to-date copy
 - **Invalid** – data in block is invalid

Summary: Improving Cache Performance

0. Reduce the time to hit in the cache

- smaller cache
- direct mapped cache
- smaller blocks
- for writes
 - no write allocate – no “hit” on cache, just write to write buffer
 - write allocate – to avoid two cycles (first check for hit, then write)
pipeline writes via a delayed write buffer to cache

1. Reduce the miss rate

- bigger cache
- more flexible placement (increase associativity)
- larger blocks (16 to 64 bytes typical)
- victim cache – small buffer holding most recently discarded blocks

Summary: Improving Cache Performance

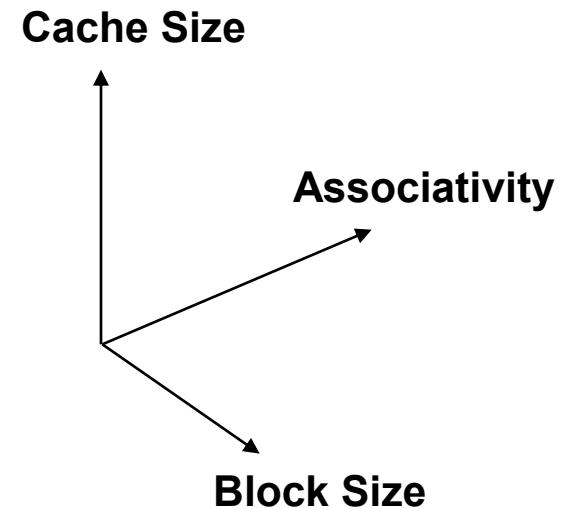
2. Reduce the miss penalty

- smaller blocks
- use a write buffer to hold dirty blocks being replaced so don't have to wait for the write to complete before reading
- check write buffer (and/or victim cache) on read miss – may get lucky
- for large blocks fetch critical word first
- use multiple cache levels – L2 cache not tied to CPU clock rate
- faster backing store/improved memory bandwidth
 - wider buses
 - memory interleaving, DDR SDRAMs

Summary: The Cache Design Space

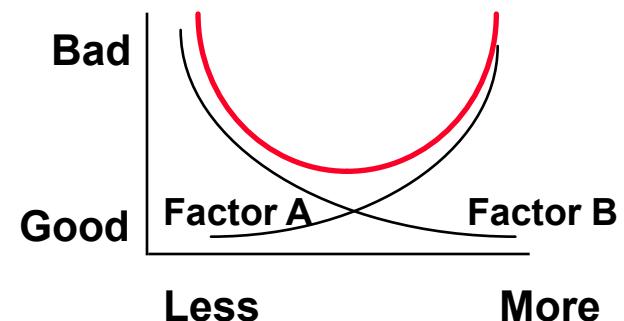
❑ Several interacting dimensions

- cache size
- block size
- associativity
- replacement policy
- write-through vs write-back
- write allocation



❑ The optimal choice is a compromise

- depends on access characteristics
 - workload
 - use (I-cache, D-cache, TLB)
- depends on technology / cost



❑ Simplicity often wins

Next Lecture and Reminders

- Next lecture

- Virtual memory hardware support

- Reminders

- Execute stage is to be written this week for progressive marks.
 - You should start to consider how you are going to demonstrate your system.

ECE4074

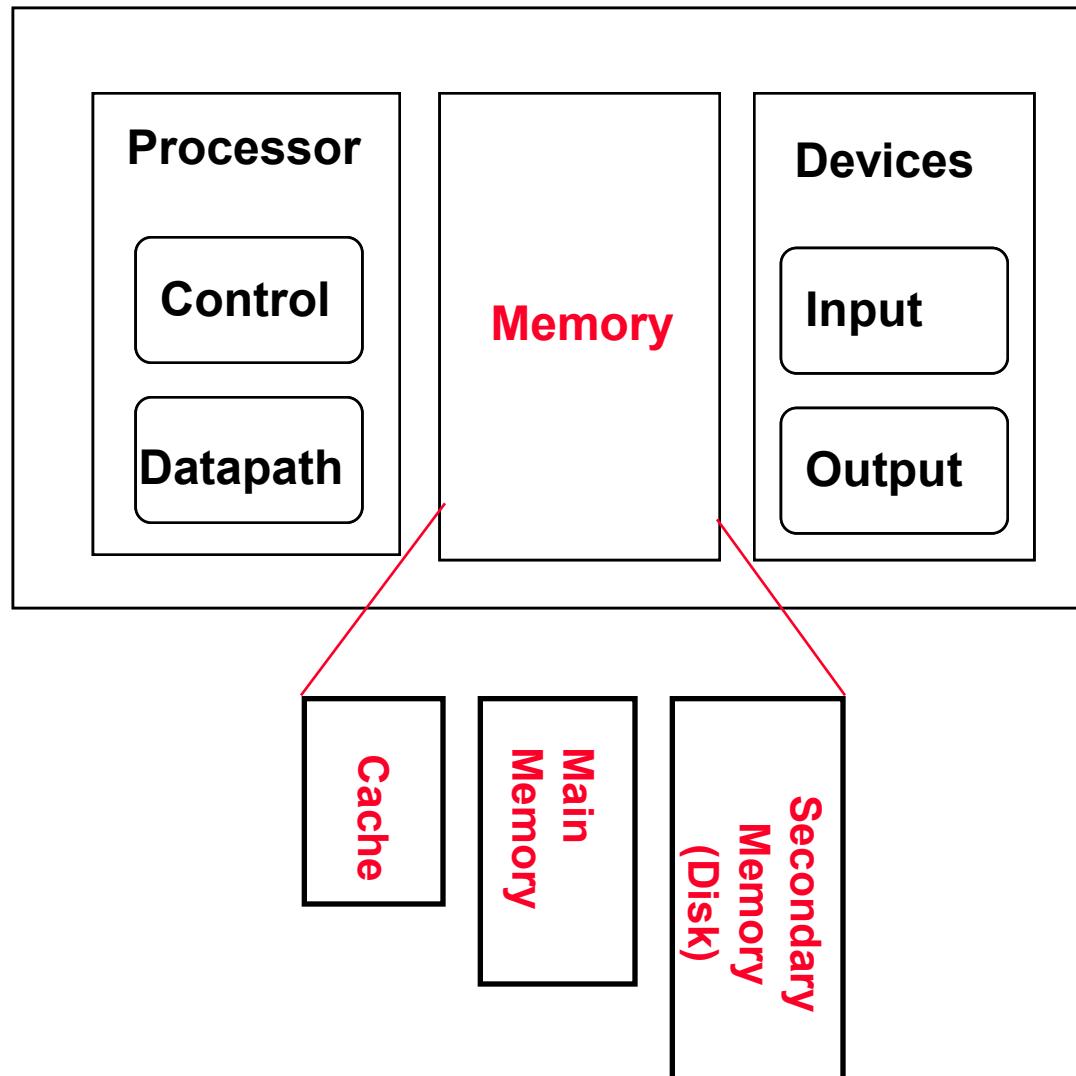
Computer Architecture

Semester 2 2014

Chapter 5B: Exploiting the Memory Hierarchy

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: Major Components of a Computer

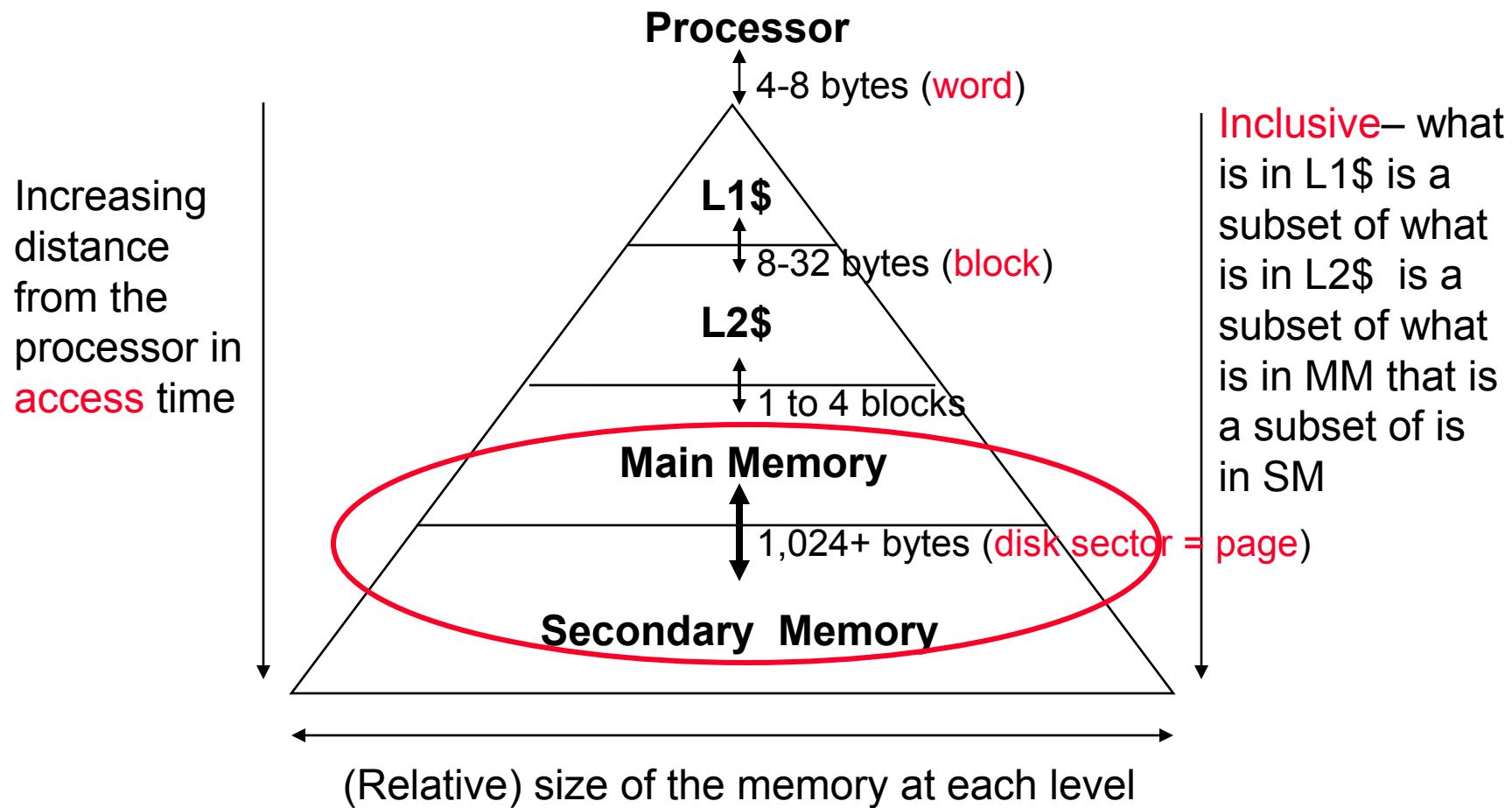


How is the Hierarchy Managed?

- registers ↔ memory
 - by compiler (programmer?)
- cache ↔ main memory
 - by the cache controller hardware
- main memory ↔ disks
 - by the operating system (virtual memory)
 - virtual to physical address mapping assisted by the hardware (TLB)
 - by the programmer (files)

Review: The Memory Hierarchy

- ❑ Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology

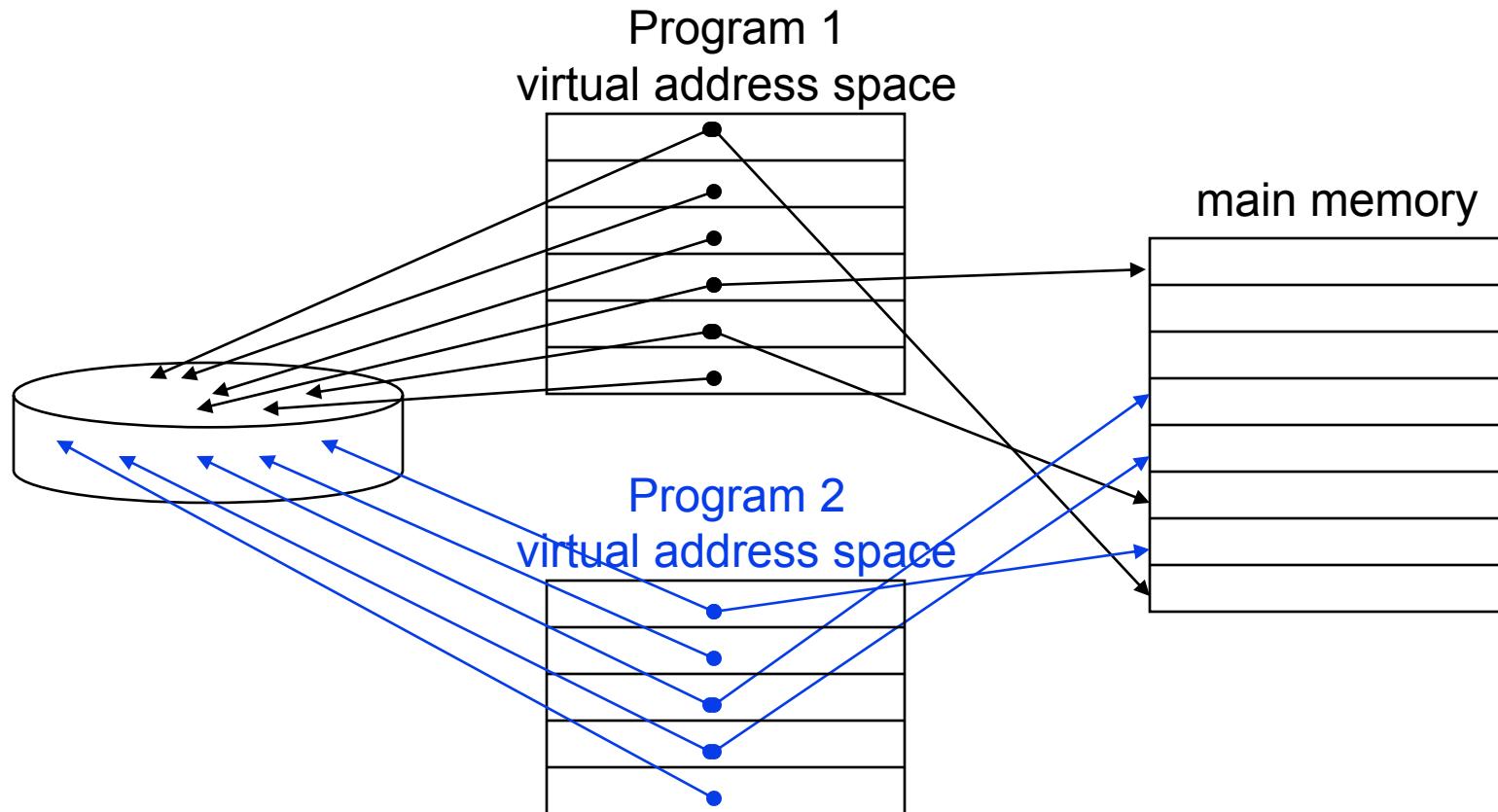


Virtual Memory

- ❑ Use main memory as a “cache” for secondary memory
 - Allows efficient and **safe** sharing of memory among multiple programs
 - Provides the ability to easily run programs larger than the size of physical memory
 - Simplifies loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)
- ❑ What makes it work? – again the Principle of Locality
 - A program is likely to access a relatively small portion of its address space during any period of time
- ❑ Each program is compiled into its own address space – a “virtual” address space
 - During run-time each **virtual** address must be translated to a **physical** address (an address in main memory)

Two Programs Sharing Physical Memory

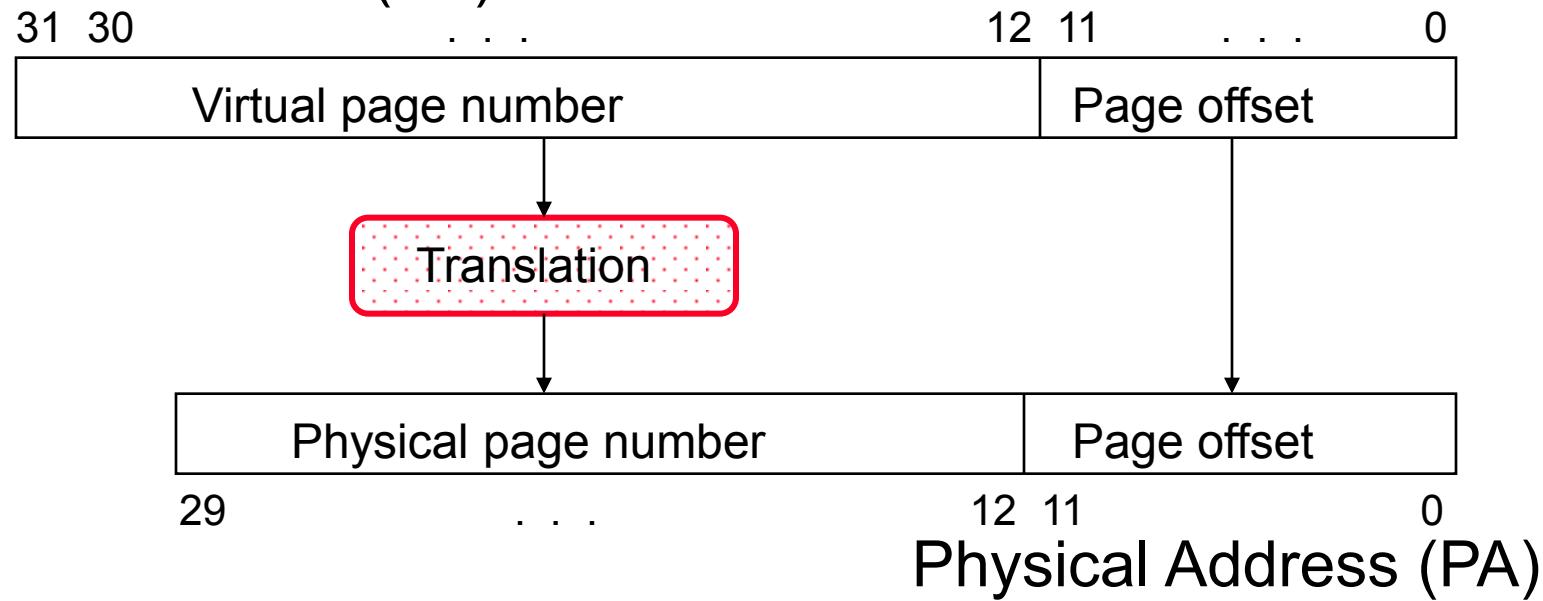
- ❑ A program's address space is divided into **pages** (all one fixed size) or segments (variable sizes)
 - The starting location of each page (either in main memory or in secondary memory) is contained in the program's **page table**



Address Translation

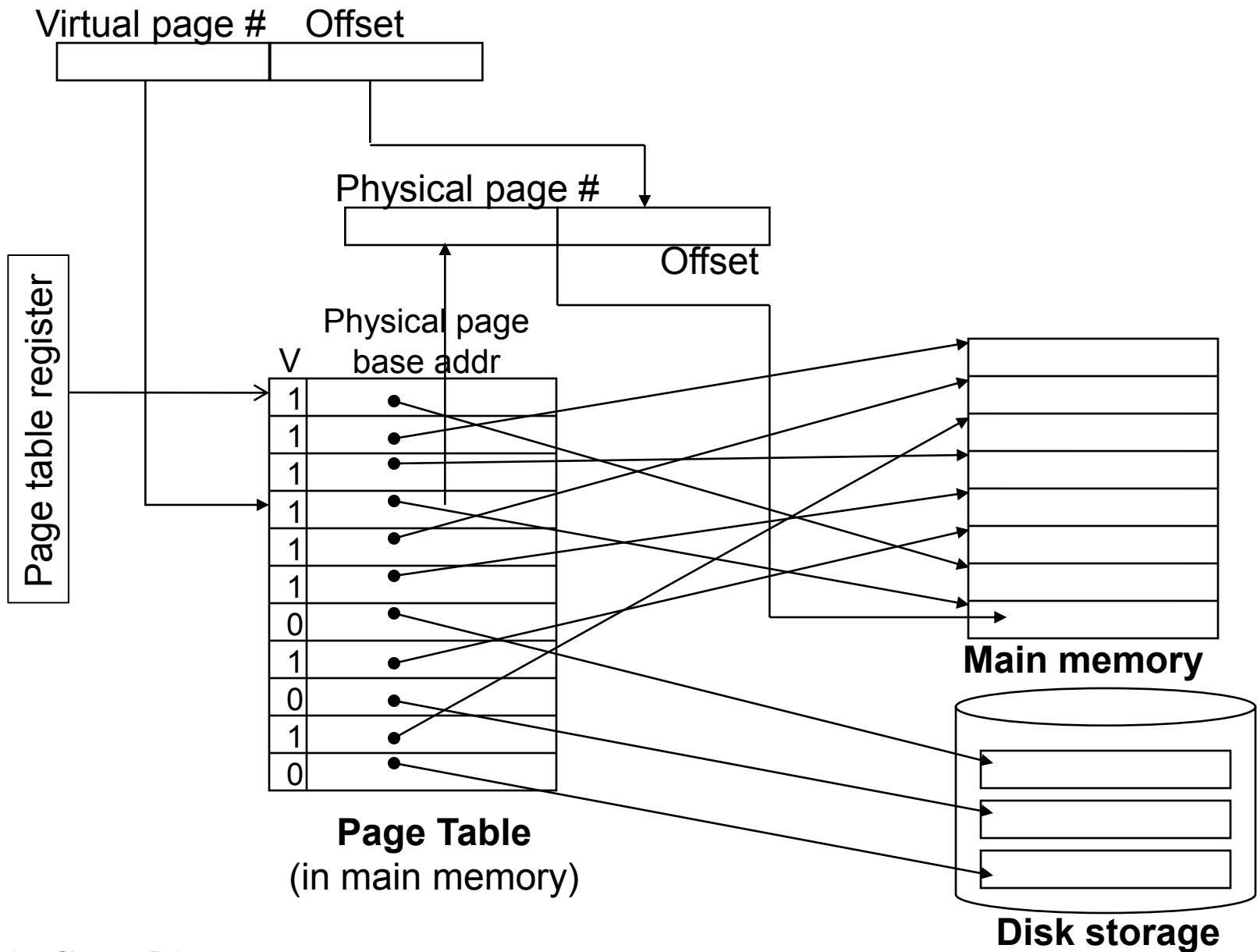
- A virtual address is translated to a physical address by a combination of hardware and software

Virtual Address (VA)



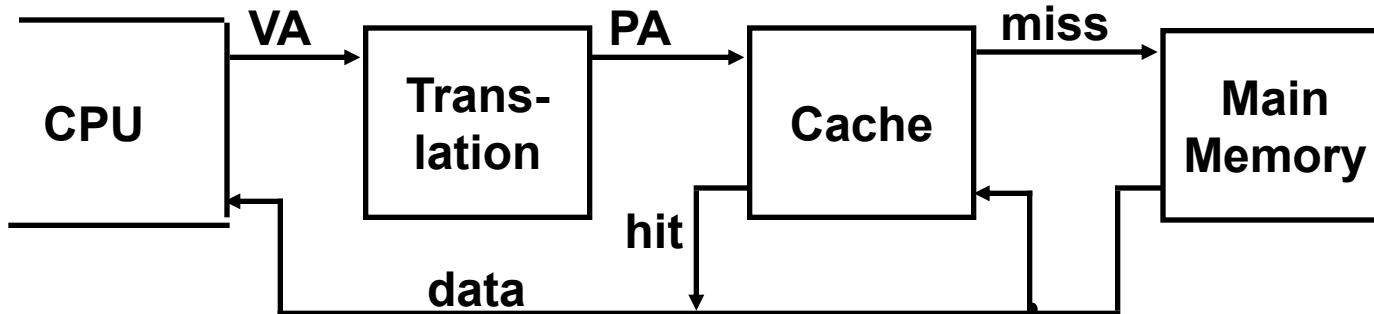
- So each memory request *first* requires an address **translation** from the virtual space to the physical space
 - A virtual memory miss (i.e., when the page is not in physical memory) is called a **page fault**

Address Translation Mechanisms



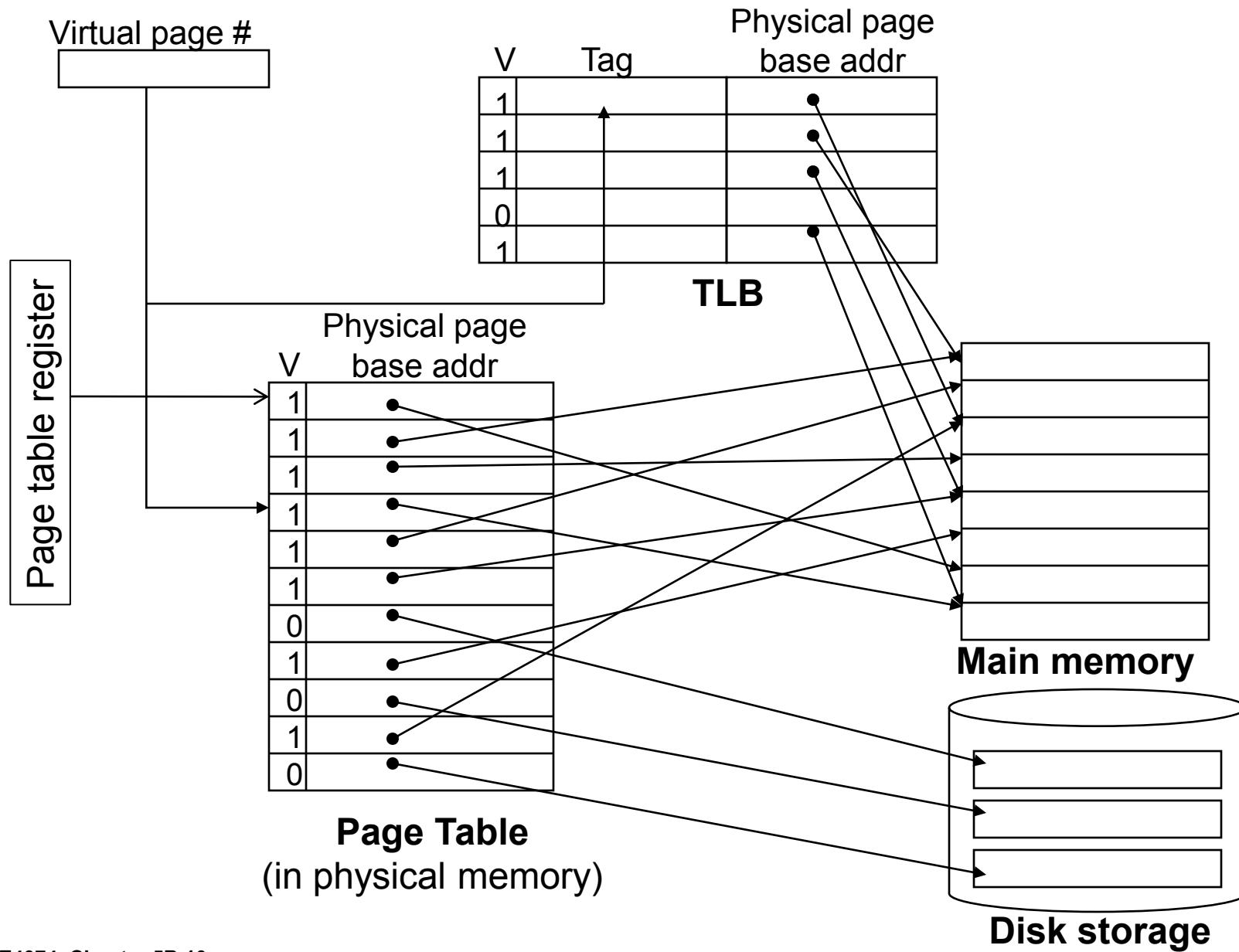
Virtual Addressing with a Cache

- ❑ Thus it takes an extra memory access to translate a VA to a PA



- ❑ This makes memory (cache) accesses **very expensive** (if every access was really *two* accesses)
- ❑ The hardware fix is to use a Translation Lookaside Buffer (TLB) – a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup

Making Address Translation Fast



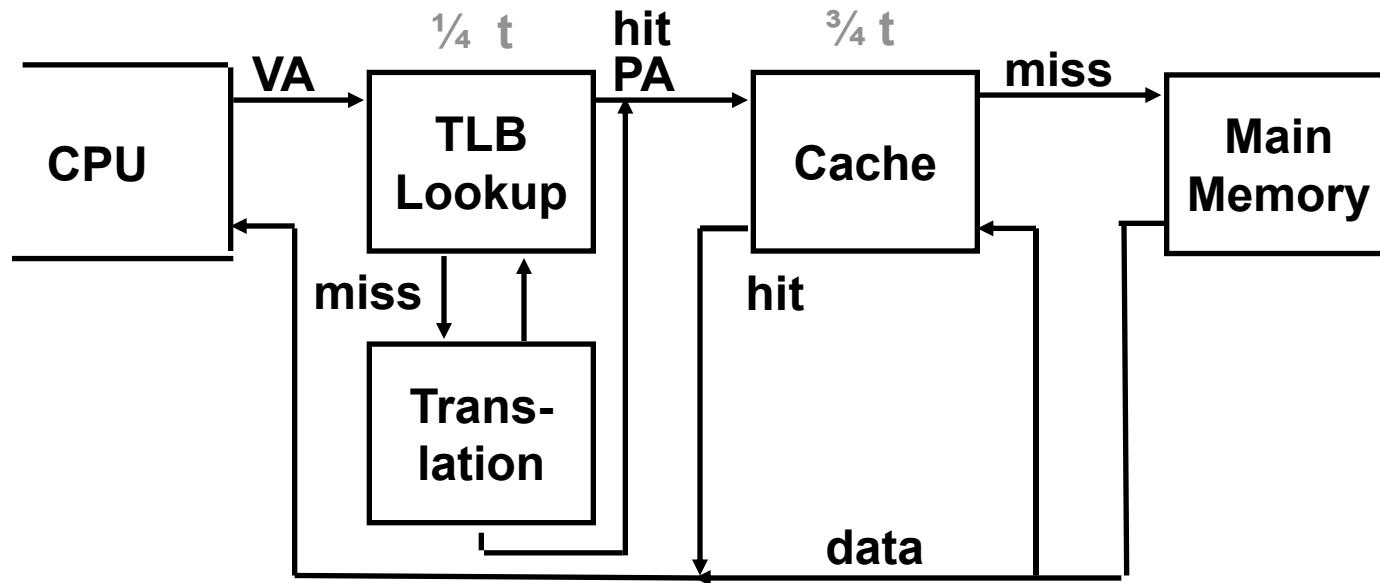
Translation Lookaside Buffers (TLBs)

- ❑ Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

| V | Virtual Page # | Physical Page # | Dirty | Ref | Access |
|---|----------------|-----------------|-------|-----|--------|
| | | | | | |

- ❑ TLB access time is typically smaller than cache access time (because TLBs are much smaller than caches)
 - TLBs are typically not more than 512 entries even on high end machines

A TLB in the Memory Hierarchy



- ❑ A TLB miss – is it a page fault or merely a TLB miss?
 - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
 - Takes 10's of cycles to find and load the translation info into the TLB
 - If the page is not in main memory, then it's a true page fault
 - Takes 1,000,000's of cycles to service a page fault
- ❑ TLB misses are much more frequent than true page faults

TLB Event Combinations

| TLB | Page Table | Cache | Possible? Under what circumstances? |
|------|------------|--------------|--|
| Hit | Hit | Hit | Yes – what we want! |
| Hit | Hit | Miss | Yes – although the page table is not checked if the TLB hits |
| Miss | Hit | Hit | Yes – TLB miss, PA in page table |
| Miss | Hit | Miss | Yes – TLB miss, PA in page table, but data not in cache |
| Miss | Miss | Miss | Yes – page fault |
| Hit | Miss | Miss/ Hit | Impossible – TLB translation not possible if page is not present in memory |
| Miss | Miss | Hit | Impossible – data not allowed in cache if page is not in memory |

Handling a TLB Miss

- ❑ Consider a TLB miss for a page that is present in memory (i.e., the Valid bit in the page table is set)
 - A TLB miss (or a page fault exception) must be asserted by the end of the same clock cycle that the memory access occurs so that the next clock cycle will begin exception processing

| Register | CP0 Reg # | Description |
|----------|-----------|------------------------------------|
| EPC | 14 | Where to restart after exception |
| Cause | 13 | Cause of exception |
| BadVAddr | 8 | Address that caused exception |
| Index | 0 | Location in TLB to be read/written |
| Random | 1 | Pseudorandom location in TLB |
| EntryLo | 2 | Physical page address and flags |
| EntryHi | 10 | Virtual page address |
| Context | 4 | Page table address & page number |

A MIPS Software TLB Miss Handler

- When a TLB miss occurs, the hardware saves the address that caused the miss in `BadVAddr` and transfers control to `8000 0000hex`, the location of the TLB miss handler

TLBmiss:

```
mfc0  $k1, Context      #copy addr of PTE into $k1
lw    $k1, 0($k1)        #put PTE into $k1
mtc0  $k1, EntryLo      #put PTE into EntryLo
tlbwr                         #put EntryLo into TLB
                               #      at Random
eret                           #return from exception
```

- `tlbwr` copies from `EntryLo` into the TLB entry selected by the control register `Random`
- A TLB miss takes about a dozen clock cycles to handle

Some Virtual Memory Design Parameters

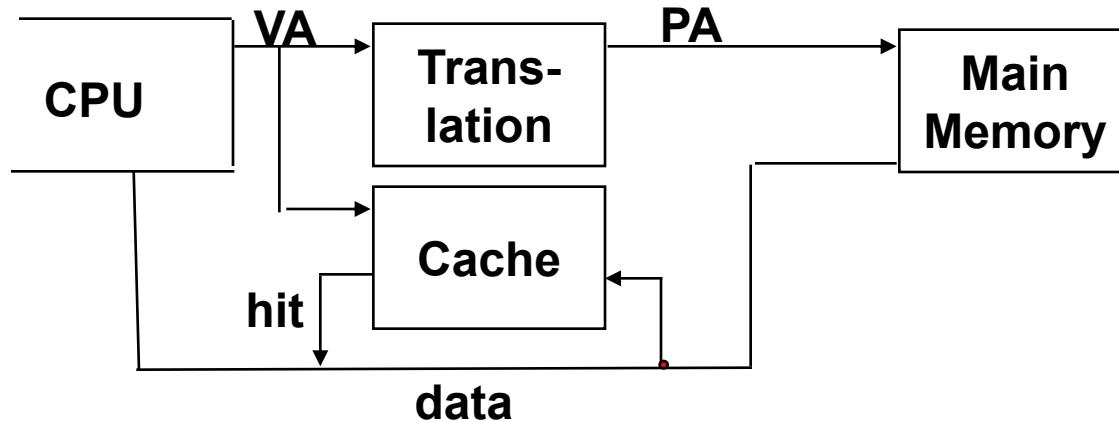
| | Paged VM | TLBs |
|-----------------------|------------------------------|-------------------------|
| Total size | 16,000 to 250,000 words | 16 to 512 entries |
| Total size (KB) | 250,000 to 1,000,000,000 | 0.25 to 16 |
| Block size (B) | 4000 to 64,000 | 4 to 8 |
| Hit time | | 0.5 to 1 clock cycle |
| Miss penalty (clocks) | 10,000,000 to 100,000,000 | 10 to 100 |
| Miss rates | 0.00001% to 0.0001% | 0.01% to 1% |

Two Machines' TLB Parameters

| | Intel Nehalem | AMD Barcelona |
|------------------|---|---|
| Address sizes | 48 bits (vir); 44 bits (phy) | 48 bits (vir); 48 bits (phy) |
| Page size | 4KB | 4KB |
| TLB organization | <p>L1 TLB for instructions and L1 TLB for data per core; both are 4-way set assoc.; LRU</p> <p>L1 ITLB has 128 entries, L2 DTLB has 64 entries</p> <p>L2 TLB (unified) is 4-way set assoc.; LRU</p> <p>L2 TLB has 512 entries</p> <p>TLB misses handled in hardware</p> | <p>L1 TLB for instructions and L1 TLB for data per core; both are fully assoc.; LRU</p> <p>L1 ITLB and DTLB each have 48 entries</p> <p>L2 TLB for instructions and L2 TLB for data per core; each are 4-way set assoc.; round robin LRU</p> <p>Both L2 TLBs have 512 entries</p> <p>TLB misses handled in hardware</p> |

Why Not a Virtually Addressed Cache?

- ❑ A virtually addressed cache would only require address translation on cache misses

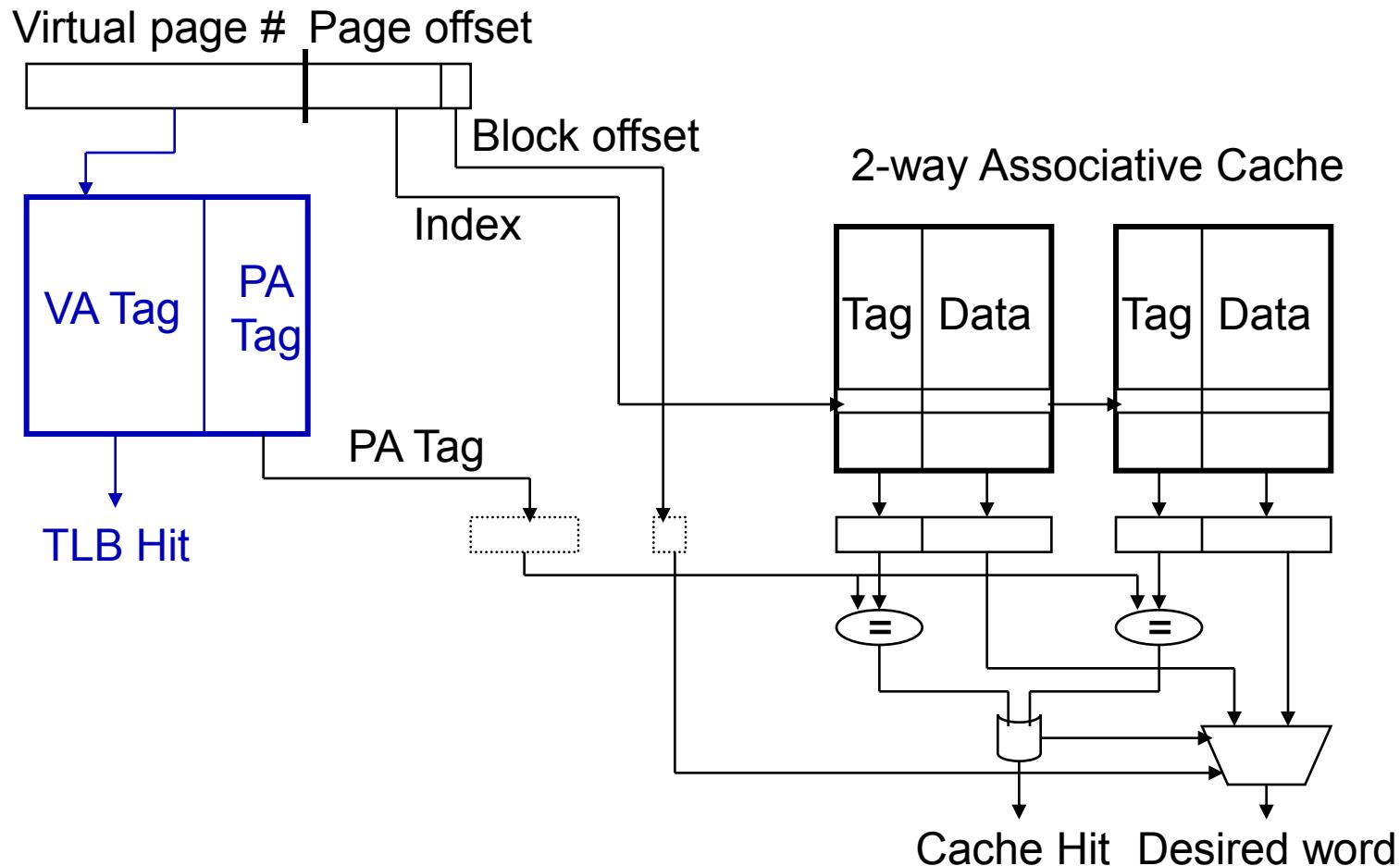


but

- Two programs which are sharing data will have two different virtual addresses for the same physical address – **aliasing** – so have two copies of the shared data in the cache and two entries in the TBL which would lead to coherence issues
 - Must update all cache entries with the same physical address or the memory becomes inconsistent

Reducing Translation Time

- ❑ Can **overlap** the cache access with the TLB access
 - Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache



The Hardware/Software Boundary

- ❑ What parts of the virtual to physical address translation is done by or assisted by the hardware?
 - Translation Lookaside Buffer (TLB) that caches the recent translations
 - TLB access time is part of the cache hit time
 - May allot an extra stage in the pipeline for TLB access
 - Page table storage, fault detection and updating
 - Page faults result in interrupts (precise) that are then handled by the OS
 - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables
 - Disk placement
 - Bootstrap (e.g., out of disk sector 0) so the system can service a limited number of page faults before the OS is even loaded

4 Questions for the Memory Hierarchy

- ❑ Q1: Where can a entry be placed in the upper level?
(Entry placement)

- ❑ Q2: How is a entry found if it is in the upper level?
(Entry identification)

- ❑ Q3: Which entry should be replaced on a miss?
(Entry replacement)

- ❑ Q4: What happens on a write?
(Write strategy)

Q1&Q2: Where can a entry be placed/found?

| | Entries per set | # of sets |
|-------------------|-------------------------------|-----------------------------------|
| Direct mapped | # of entries | 1 |
| Set associative | (# of entries)/ associativity | Associativity (typically 2 to 16) |
| Fully associative | 1 | # of entries |

| | Location method | # of comparisons |
|-------------------|---|-------------------------|
| Direct mapped | Index | 1 |
| Set associative | Index the set; compare set's tags | Degree of associativity |
| Fully associative | Compare all entries' tags Separate lookup (page) table | # of entries 0 |

Q3: Which entry should be replaced on a miss?

- ❑ Easy for direct mapped – only one choice
- ❑ Set associative or fully associative
 - Random
 - LRU (Least Recently Used)
- ❑ For a 2-way set associative, random replacement has a miss rate about 1.1 times higher than LRU
- ❑ LRU is too costly to implement for high levels of associativity (> 4-way) since tracking the usage information is costly

Q4: What happens on a write?

- ❑ **Write-through** – The information is written to the entry in the current memory level *and* to the entry in the next level of the memory hierarchy
 - Always combined with a write buffer so write waits to next level memory can be eliminated (as long as the write buffer doesn't fill)
- ❑ **Write-back** – The information is written only to the entry in the current memory level. The modified entry is written to next level of memory only when it is replaced.
 - Need a dirty bit to keep track of whether the entry is clean or dirty
 - Virtual memory systems always use write-back of dirty pages to disk
- ❑ Pros and cons of each?
 - Write-through: read misses don't result in writes (so are simpler and cheaper), easier to implement
 - Write-back: writes run at the speed of the cache; repeated writes require only one write to lower level

Summary

- ❑ The Principle of Locality:
 - Program likely to access a relatively small portion of the address space at any instant of time.
 - Temporal Locality: Locality in Time
 - Spatial Locality: Locality in Space
- ❑ Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
 1. Where can entry be placed?
 2. How is entry found?
 3. What entry is replaced on miss?
 4. How are writes handled?
- ❑ Page tables map virtual address to physical address
 - TLBs are important for fast translation

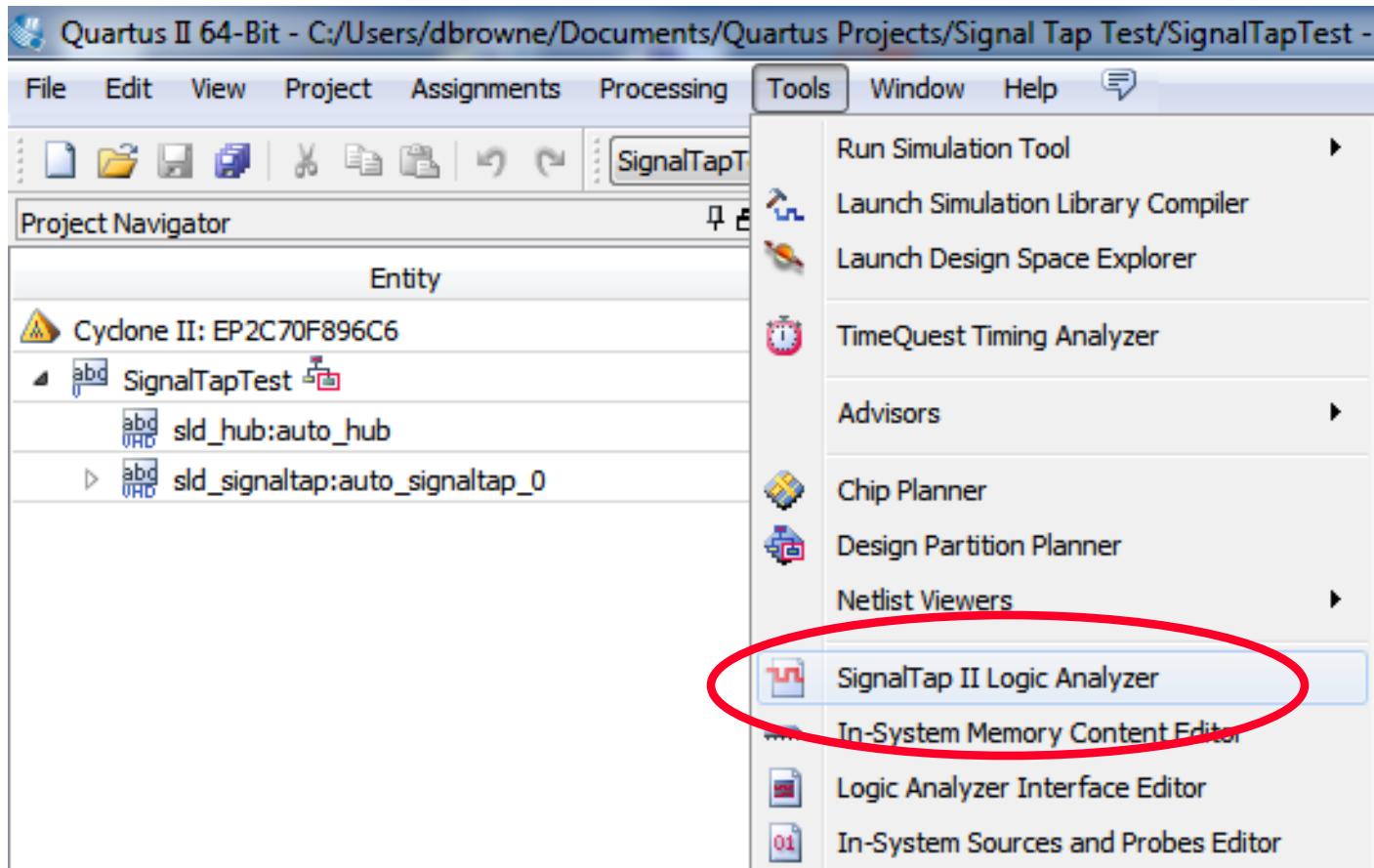
Next Lecture and Reminders

- ❑ Next lecture
 - I/O devices and systems
- ❑ Assignment 1 demonstration due Week 7 lab time
- ❑ Assignment 1 report due Friday Week 7

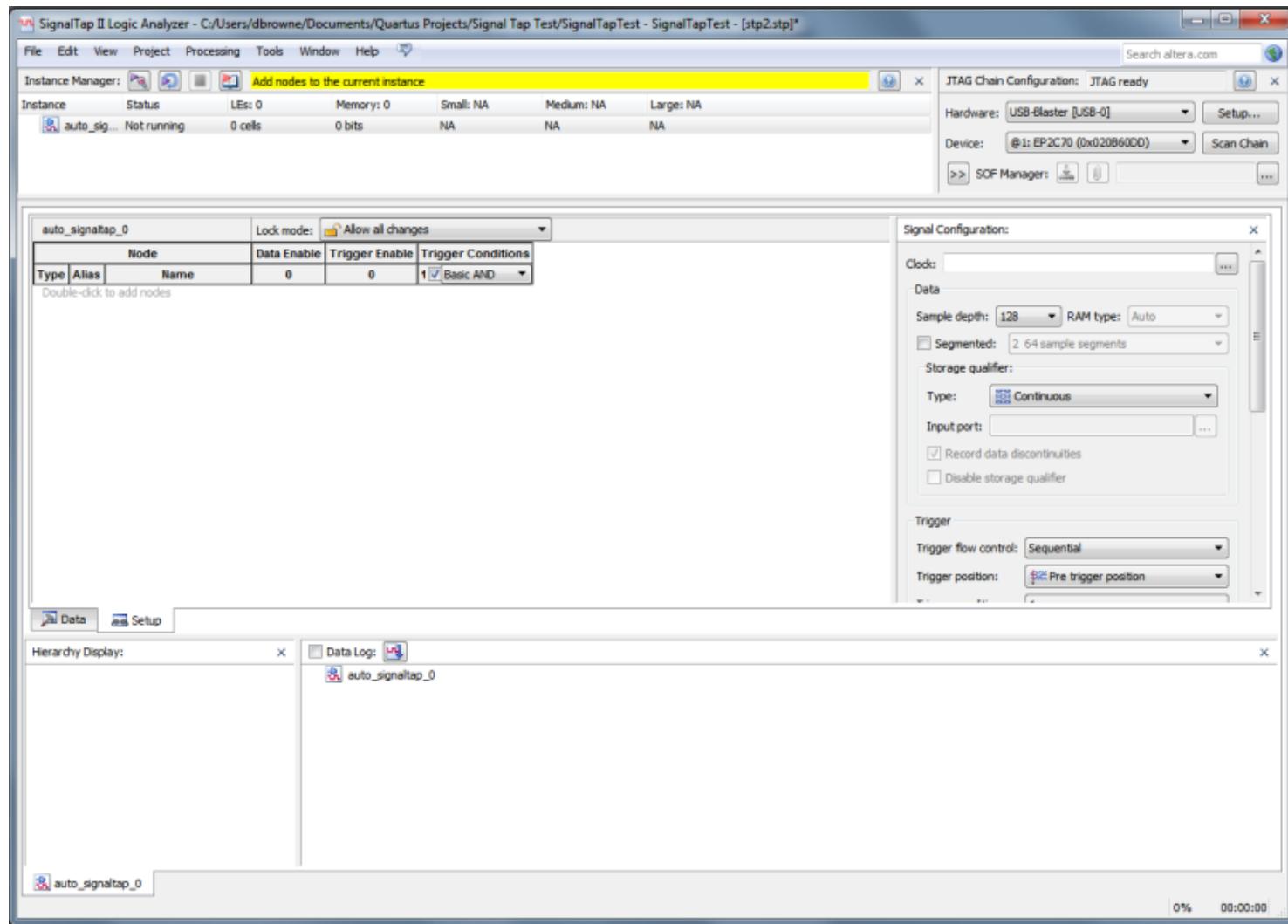
Getting Started with SignalTap

- ❑ SignalTap adds additional circuitry to your design.
 - This circuitry synchronously stores some data into block RAM.
 - On a “Trigger” event, this data is sent back to the PC so you can see what your circuitry is doing (while running on the DE2-board)
 - A trigger event can be something like:
 - A rising edge on a signal.
 - A particular level of a signal.
 - A specific value on a variable.
 - A manual trigger, sent by the PC
- ❑ To add this additional hardware, your design must be compiled with SignalTap configuration data added to the project.

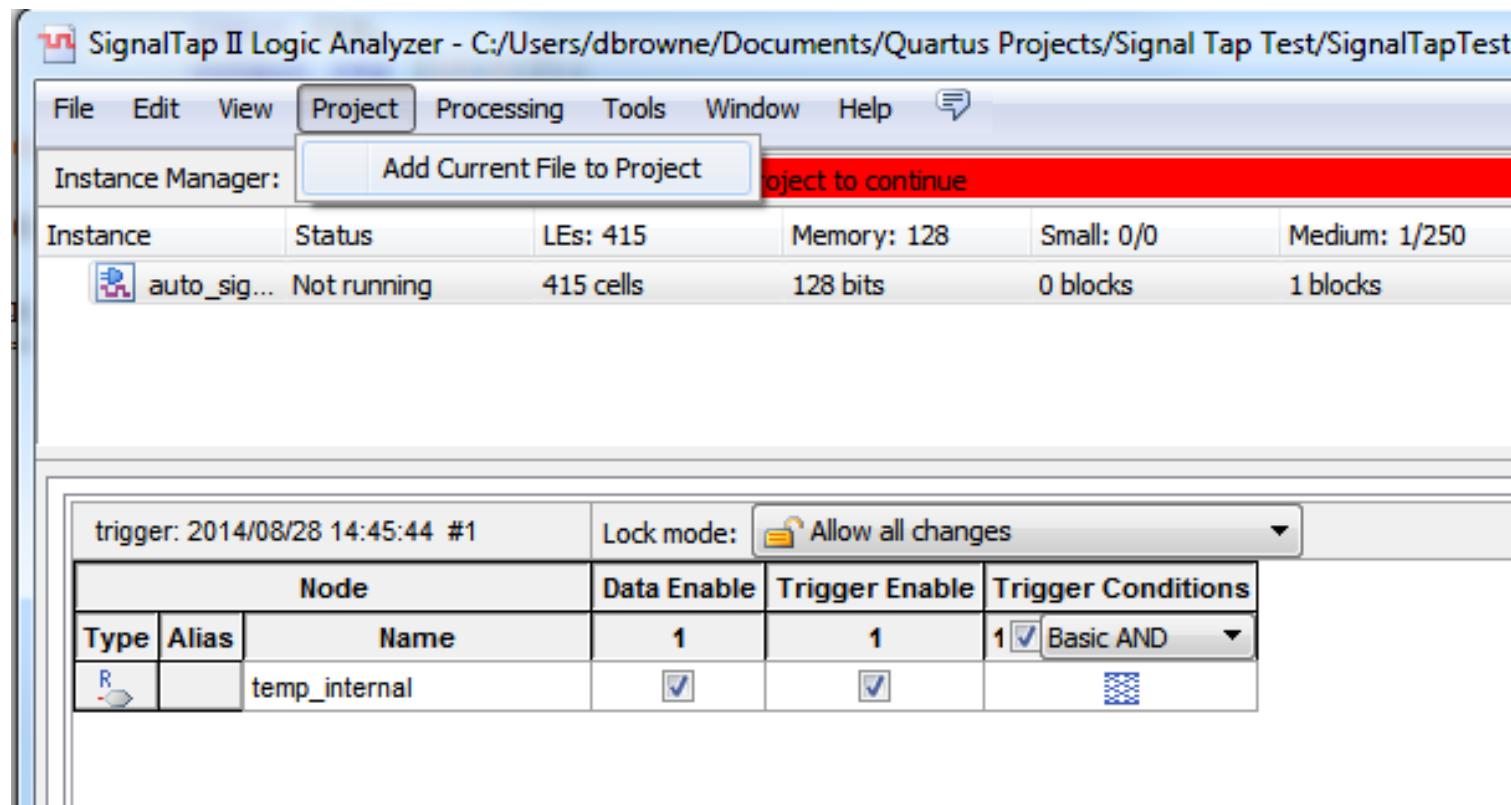
Getting Started with SignalTap



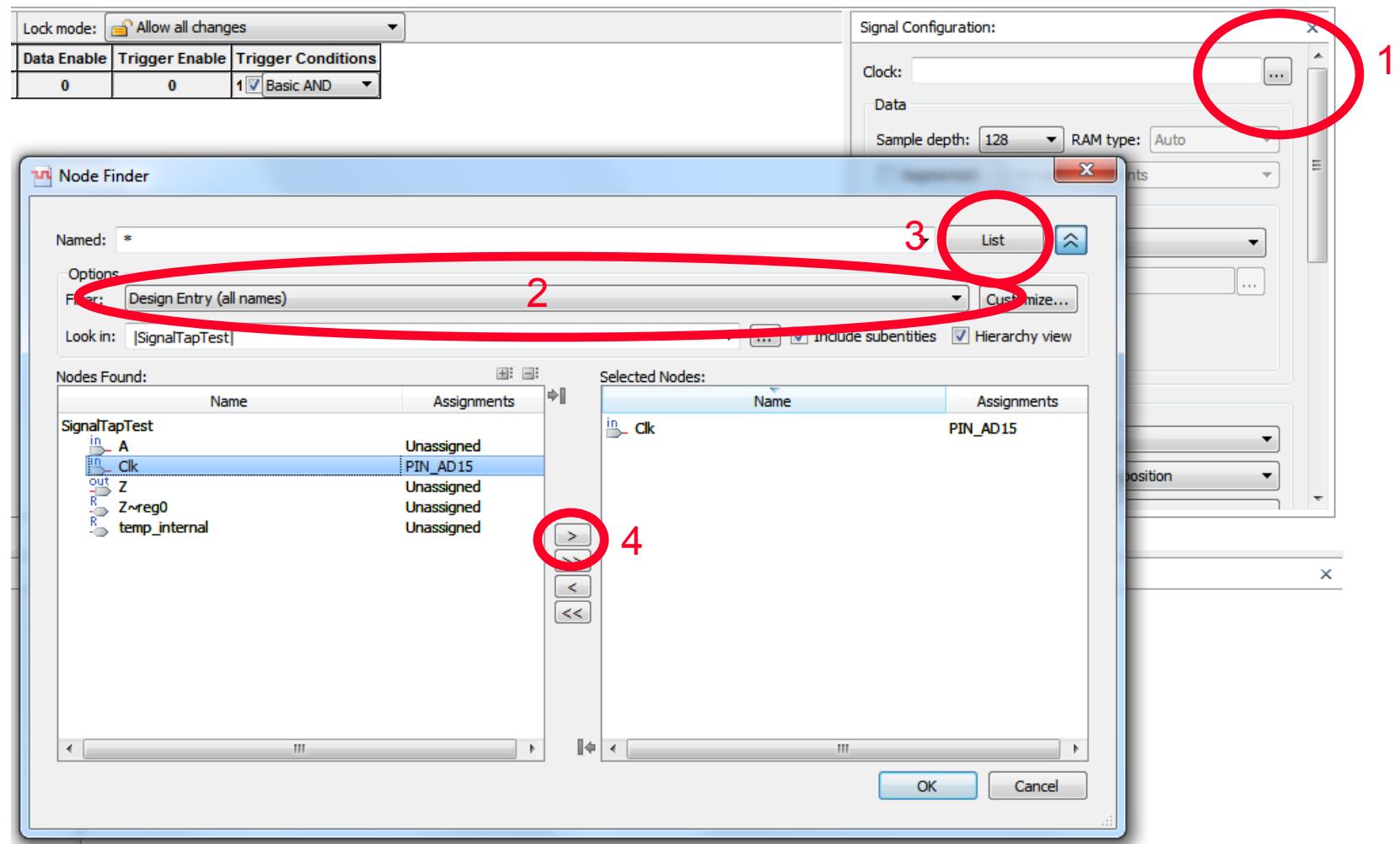
Getting Started with SignalTap



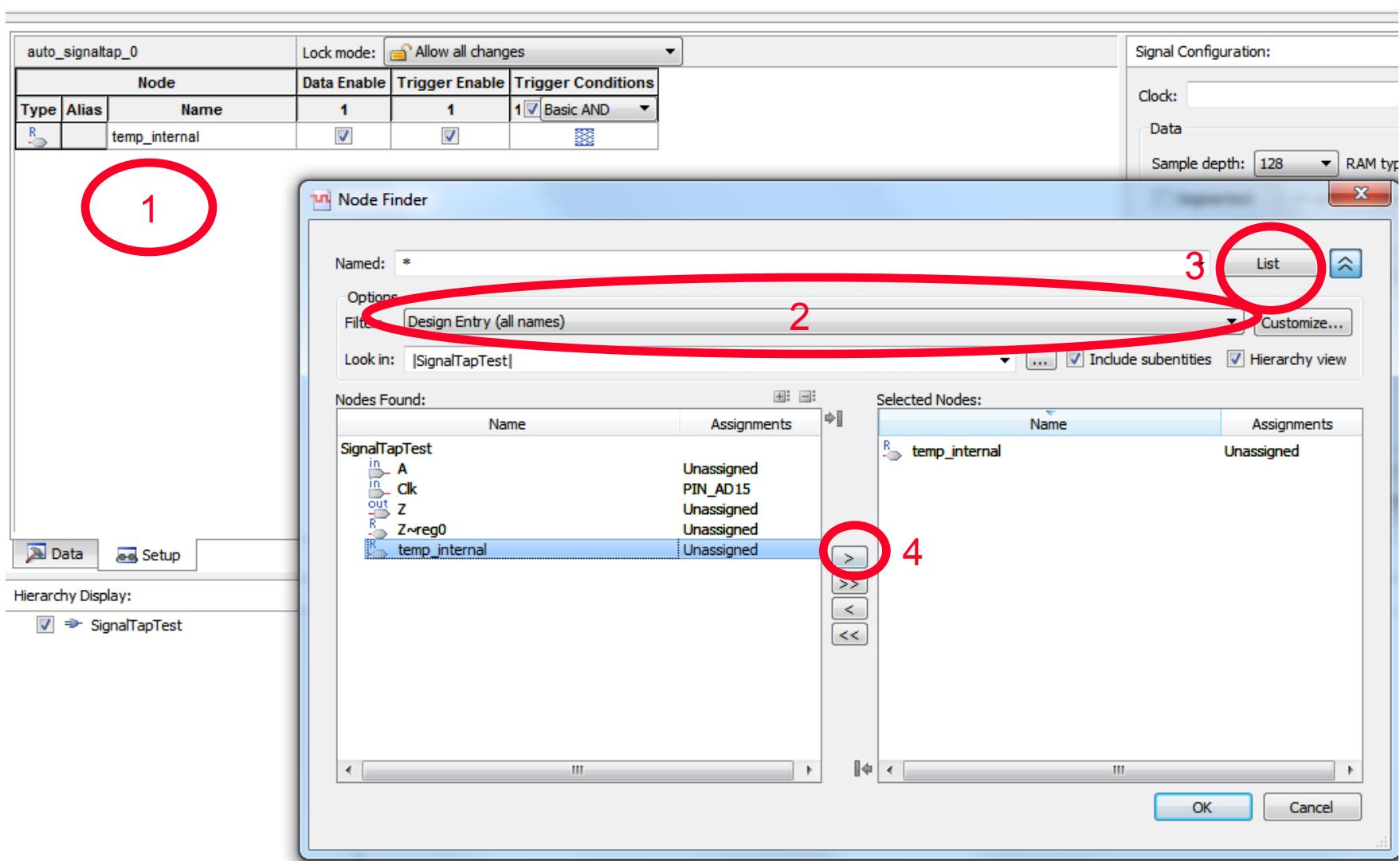
Adding the SignalTap File to the Project



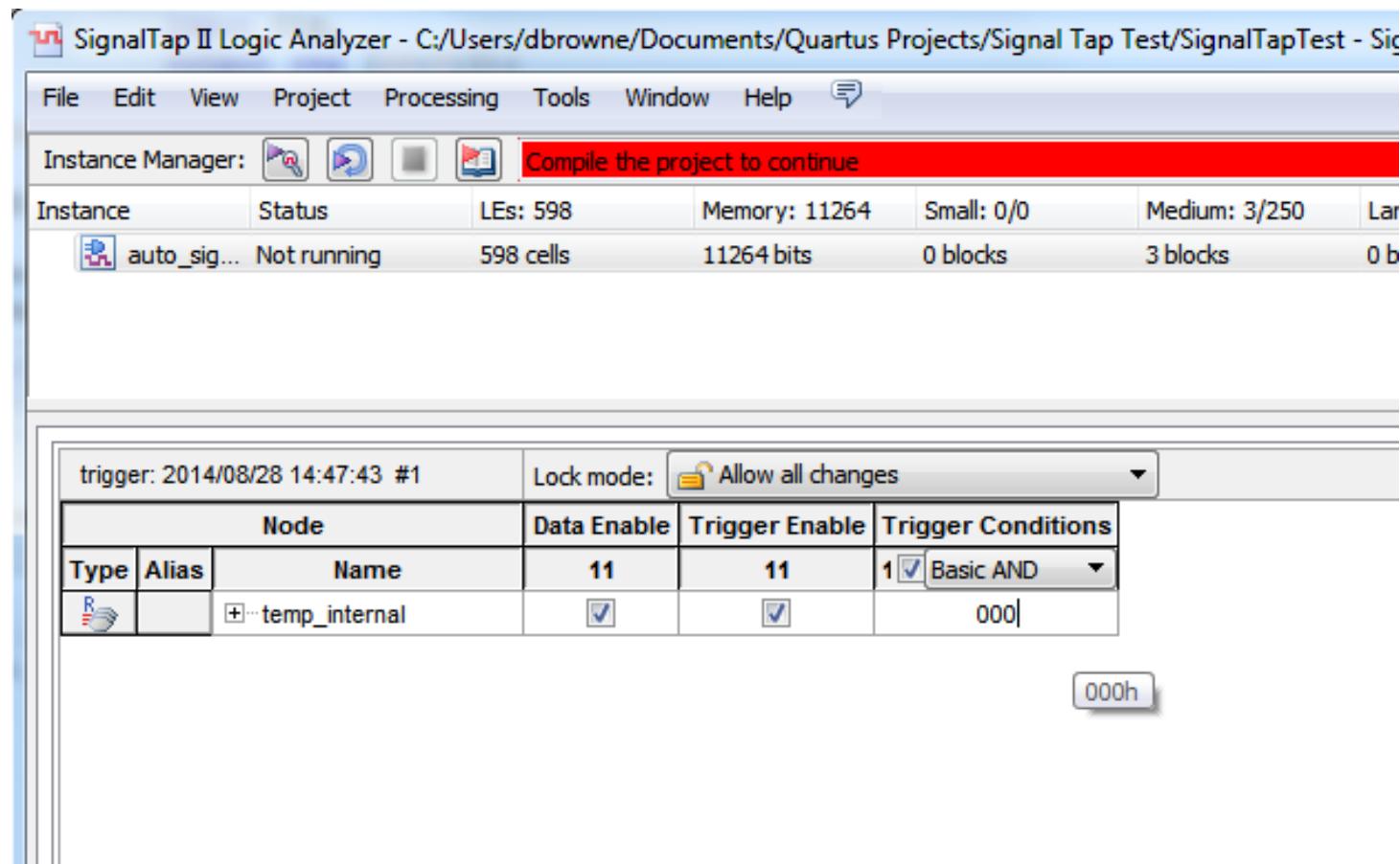
Declare the Clock you want to use



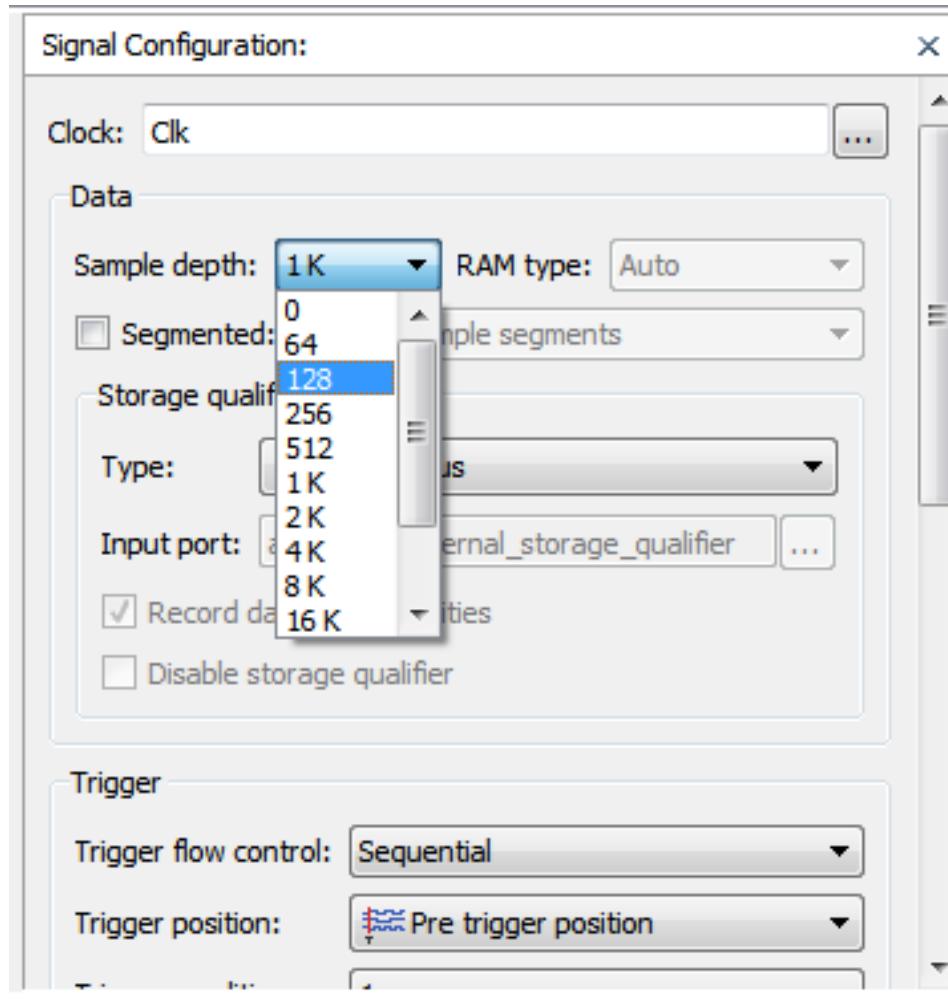
Select the signals you want to record



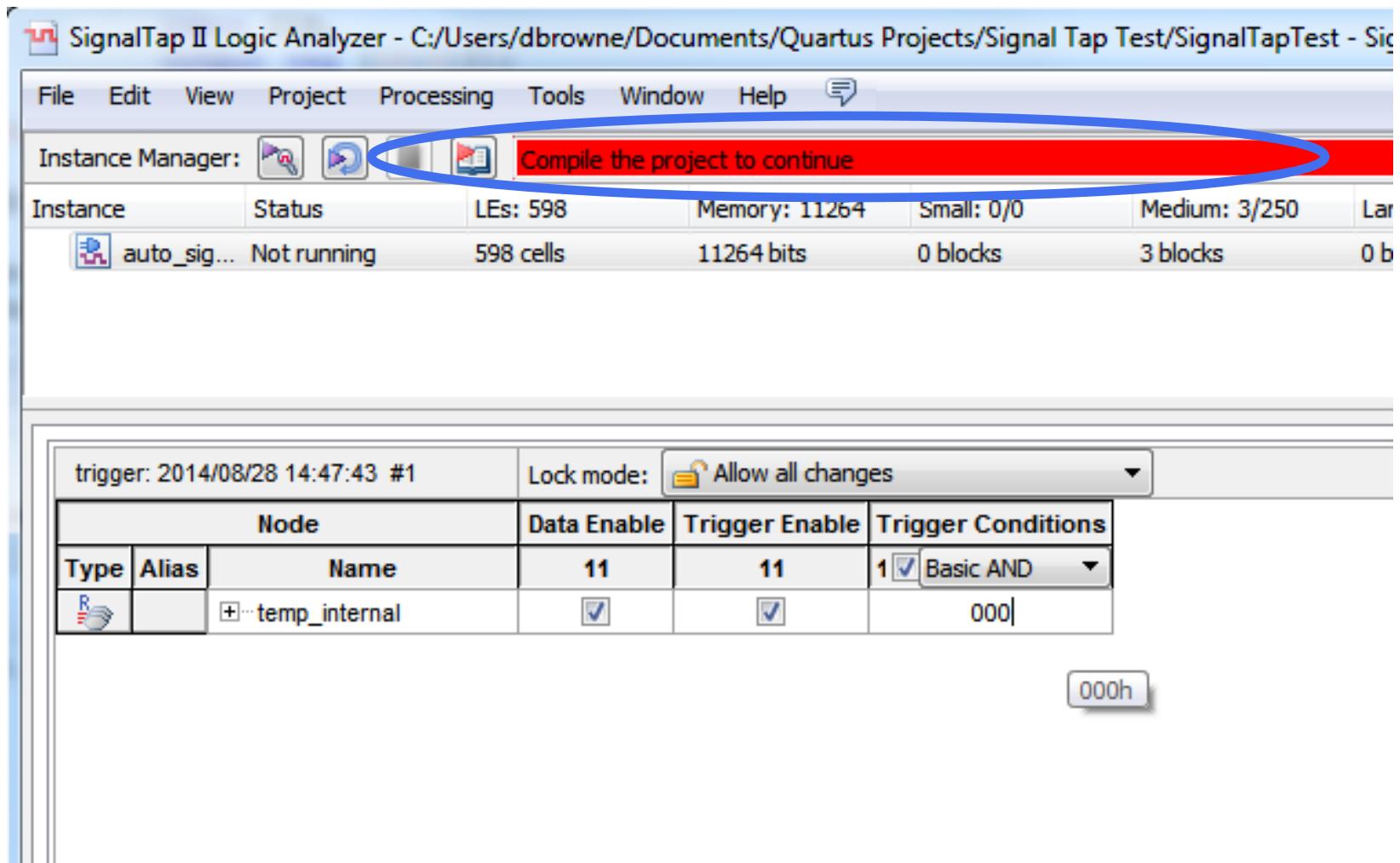
Add the trigger event



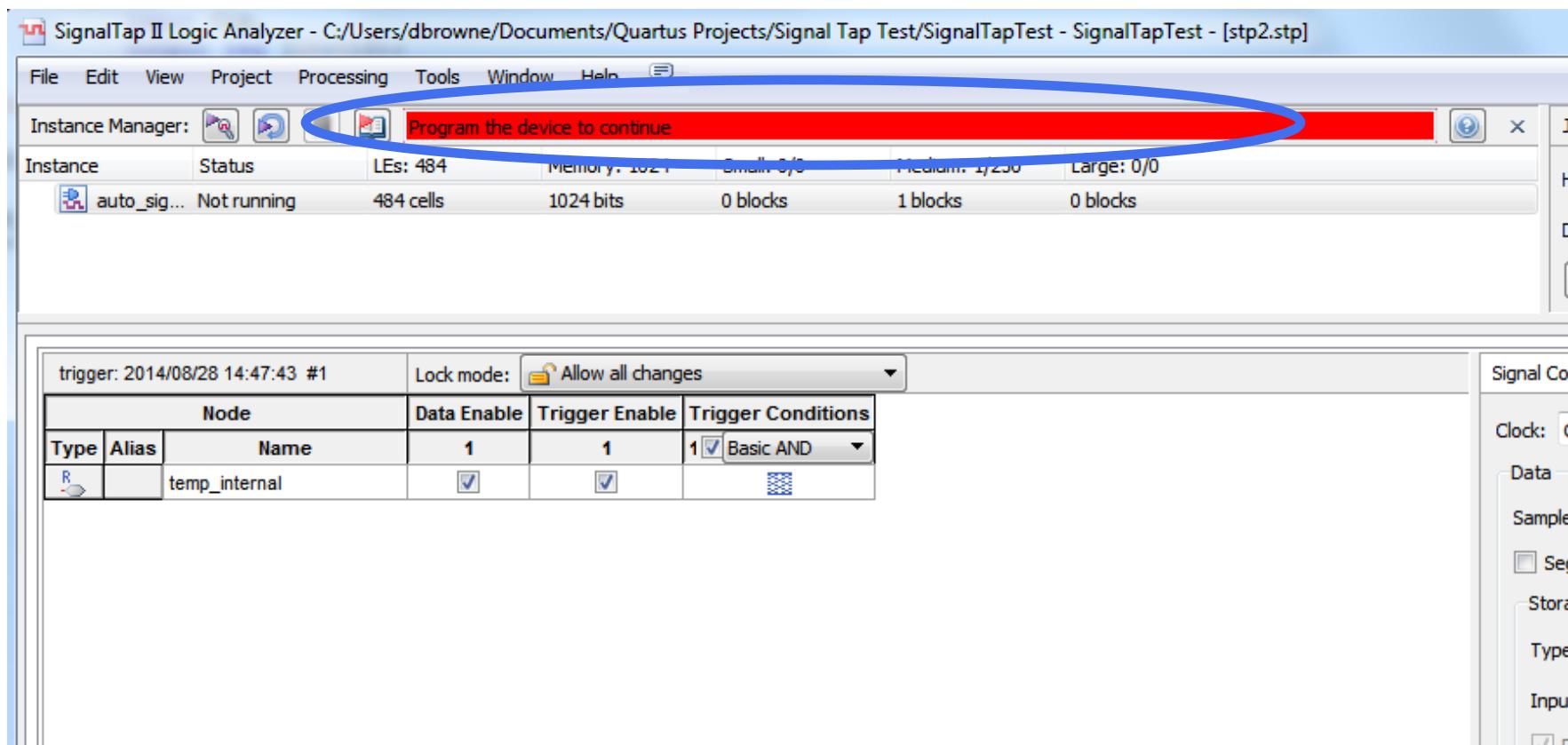
Select the depth of your recording



Compile your project



Program your device



Run analysis and observe your data



Demonstrating your design

- ❑ Trigger events can be:
 - Write signals to your RAM.
 - A specific address to your RAM.

- ❑ Beware:
 - If your trigger never occurs you won't get data back.
 - If your clock isn't a clock, you won't get data back.
 - If your algorithm doesn't terminate gracefully (ie end in a NOOP loop) then you may keep triggering and overwriting the data you want.
 - You can only save so much data before you use all the board's RAM.

ECE4074

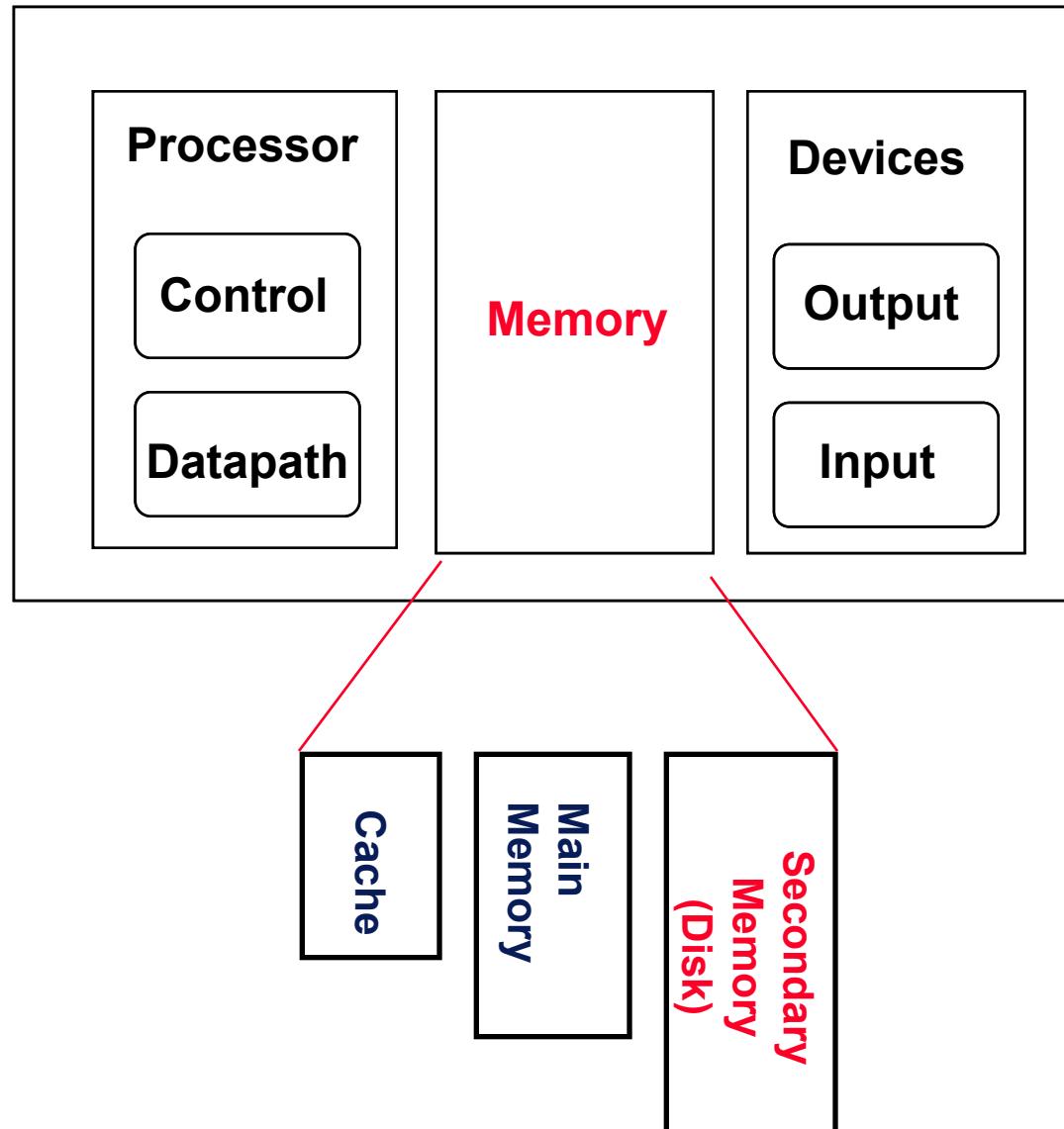
Computer Architecture

Semester 2 2014

Chapter 6A: Disk Systems

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: Major Components of a Computer



Magnetic Disk

❑ Purpose

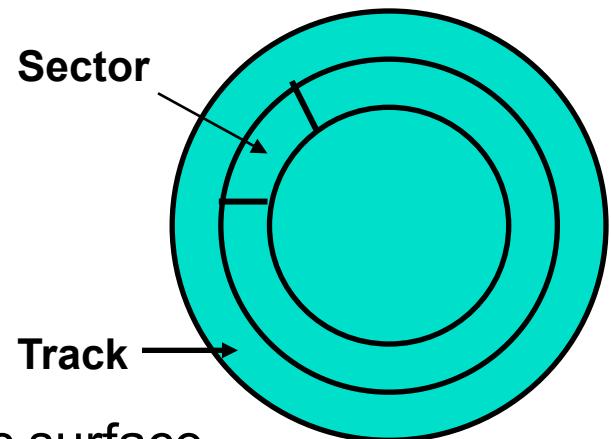
- Long term, **nonvolatile** storage
- Lowest level in the memory hierarchy
 - slow, large, inexpensive

❑ General structure

- A rotating platter coated with a magnetic surface
- A moveable read/write head to access the information on the disk

❑ Typical numbers

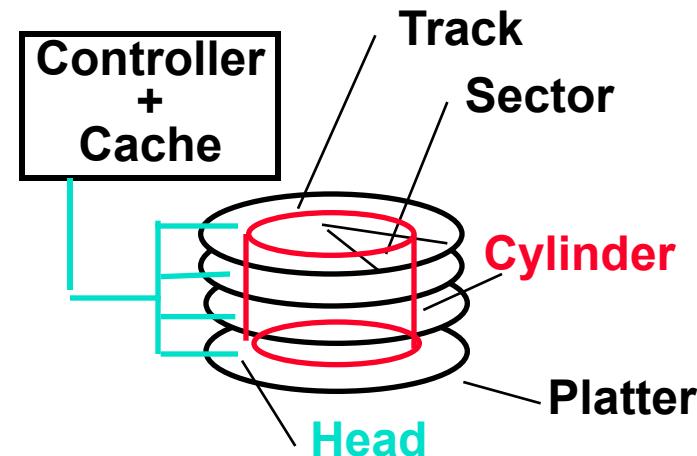
- 1 to 4 platters (each with 2 recordable surfaces) per disk of 1" to 3.5" in diameter
- Rotational speeds of 5,400 to 15,000 RPM
- 10,000 to 50,000 **tracks** per surface
 - **cylinder** - all the tracks under the head at a given point on all surfaces
- 100 to 500 **sectors** per track
 - the smallest unit that can be read/written (typically 512B)



Magnetic Disk Characteristic

Disk read/write components

1. **Seek time**: position the head over the proper track (3 to 13 ms avg)
 - due to locality of disk references the actual average seek time may be only 25% to 33% of the advertised number
2. **Rotational latency**: wait for the desired sector to rotate under the head ($\frac{1}{2}$ of 1/RPM converted to ms)
 - $0.5/5400\text{RPM} = 5.6\text{ms}$ to $0.5/15000\text{RPM} = 2.0\text{ms}$
3. **Transfer time**: transfer a block of bits (one or more sectors) under the head to the disk controller's cache (70 to 125 MB/s are typical disk transfer rates in 2008)
 - the disk controller's "cache" takes advantage of spatial locality in disk accesses
 - cache transfer rates are much faster (e.g., 375 MB/s)
4. **Controller time**: the overhead the disk controller imposes in performing a disk I/O access (typically < .2 ms)



Typical Disk Access Time

- ❑ The average time to read or write a 512B sector for a disk rotating at 15,000 RPM with average seek time of 4 ms, a 100MB/sec transfer rate, and a 0.2 ms controller overhead

$$\text{Avg disk read/write} = 4.0 \text{ ms} + 0.5/(15,000\text{RPM}/(60\text{sec/min}))+ \\ 0.5\text{KB}/(100\text{MB/sec}) + 0.2 \text{ ms} = 4.0 + 2.0 + 0.01 + 0.2 = 6.2 \text{ ms}$$

If the measured average seek time is 25% of the advertised average seek time, then

$$\text{Avg disk read/write} = 1.0 + 2.0 + 0.01 + 0.2 = 3.2 \text{ ms}$$

- ❑ The rotational latency is usually the largest component of the access time. Reading consecutive sectors is quicker.

Disk Interface Standards

- ❑ Higher-level disk interfaces have a microprocessor disk controller that can lead to performance optimizations
 - ATA (Advanced Technology Attachment) – An interface standard for the connection of storage devices such as hard disks, solid-state drives, and CD-ROM drives. Parallel ATA has been largely replaced by serial ATA.
 - SCSI (Small Computer Systems Interface) – A set of standards (commands, protocols, and electrical and optical interfaces) for physically connecting and transferring data between computers and peripheral devices. Most commonly used for hard disks and tape drives.
- ❑ In particular, disk controllers have SRAM disk **caches** which support fast access to data that was recently read and often also include prefetch algorithms to try to anticipate demand

Magnetic Disk Examples

| Feature | Seagate ST1000DM003 | Seagate ST5000NM0024 | West' Digital WD20NPVX |
|--------------------------|------------------------|-------------------------|---------------------------|
| Disk diameter (inches) | 3.5 | 3.5 | 2.5 |
| Capacity (GB) | 1,000 | 5,000 | 2,000 |
| # of surfaces (heads) | 2 | 8 | ?? |
| Rotation speed (RPM) | 7,200 | 7200 | ?? |
| Transfer rate (MB/sec) | 210 | 216 | 135 |
| Average seek (ms) | 8.5r-9.5w | 4.16 | ?? |
| MTTF (hours@25°C) | >870,000 | 1,400,000 | ?? |
| Dim (MM), Weight (kg) | 20x102x147, 0.4 | 26x4x102, 0.780 | 15x100x70, 0.18 |
| GB/watt | 170 | 443 | 1176 |
| Power: op/idle (watts) | 5.9/3.36 | 11.3/6.9 | 1.9/0.6 |
| Price in 2014, ¢/GB | ~6.9¢/GB | ~12.6¢/GB | ~10.5¢/GB |

Disk Latency & Bandwidth Milestones

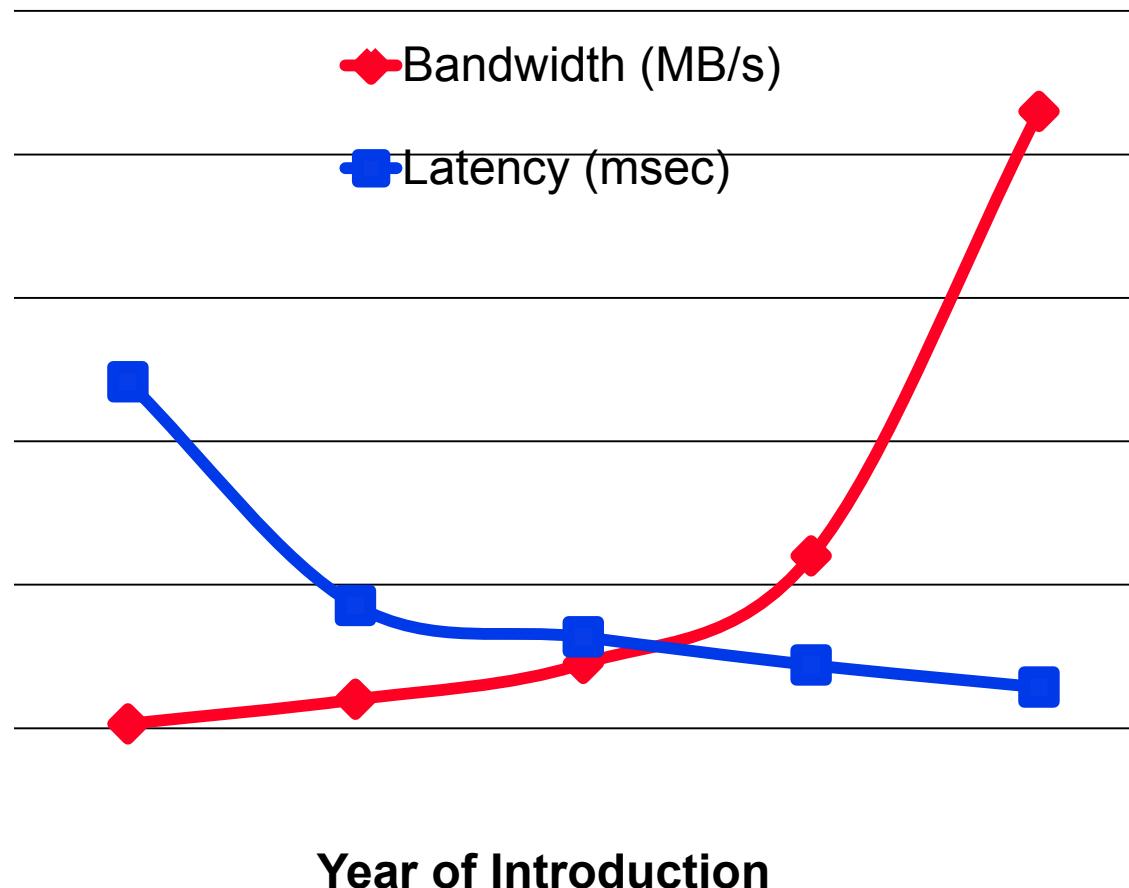
| | CDC Wren | SG ST41 | SG ST15 | SG ST39 | SG ST37 |
|-------------------|----------|---------|---------|---------|---------|
| RSpeed (RPM) | 3600 | 5400 | 7200 | 10000 | 15000 |
| Year | 1983 | 1990 | 1994 | 1998 | 2003 |
| Capacity (Gbytes) | 0.03 | 1.4 | 4.3 | 9.1 | 73.4 |
| Diameter (inches) | 5.25 | 5.25 | 3.5 | 3.0 | 2.5 |
| Interface | ST-412 | SCSI | SCSI | SCSI | SCSI |
| Bandwidth (MB/s) | 0.6 | 4 | 9 | 24 | 86 |
| Latency (msec) | 48.3 | 17.1 | 12.7 | 8.8 | 5.7 |

Patterson, CACM Vol 47, #10, 2004

- ❑ Disk **latency** is one average seek time plus the rotational latency.
- ❑ Disk **bandwidth** is the peak transfer time of formatted data from the media (not from the cache).

Latency & Bandwidth Improvements

- In the time that the disk **bandwidth doubles** the **latency** improves by a factor of only 1.2 to 1.4



Hard Disk Drives and Forward Error Correction

- ❑ Manufacturing of hard drives is not 100% reliable operation.
 - Generally hard disks contain more space than quoted.
 - Only the “good” space is made available to the end user.
- ❑ Writing and reading from magnetic media are not 100% reliable operations.
 - Forward error correction is used on the disk to correct errors that are apparent during reading.
 - Reed-Soloman coding is almost exclusively used on hard disks.
 - It corrects errors associated with burst defects (such as scratches or large imperfections on the disk surface).
 - Typically uses an additional 10% extra space to implement the FEC

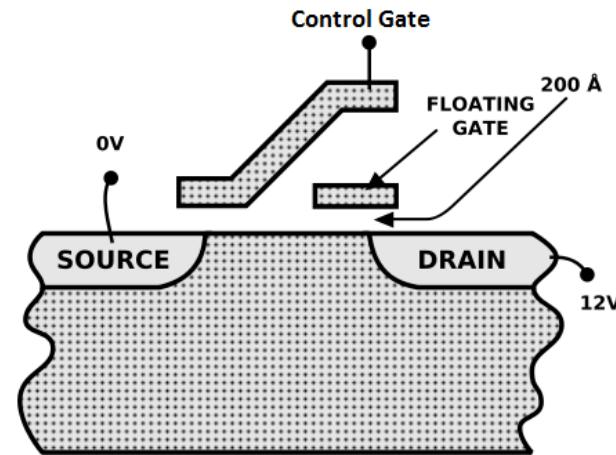
Flash Storage

- ❑ Flash memory is the first credible challenger to disks. It is **semiconductor** memory that is nonvolatile like disks, but has latency 100 to 1000 times faster than disk and is smaller, more power efficient, and more shock resistant.
 - In 2008, the price of flash is \$4 to \$10 per GB or about 2 to 10 times higher than disk and 5 to 10 times lower than DRAM.
 - Flash memory bits wear out (unlike disks and DRAMs), but **wear leveling** can make it unlikely that the write limits of the flash will be exceeded

| Feature | Kingston | Intel | Samsung | ASUS |
|-------------------------|------------|------------|------------|-----------|
| Capacity (GB) | 120 | 480 | 1000 | 240 |
| Transfer rates (MB/sec) | 550r, 510w | 540r, 490w | 540r, 520w | 830r,810w |
| MTTF | 1,000,000 | 1,200,000 | 1,500,000 | 620,000 |
| Price (2008) | ~ \$100 | ~\$390 | ~ \$480 | ~\$480 |
| Price/GB | \$1.2 | \$0.81 | \$0.48 | \$2.00 |

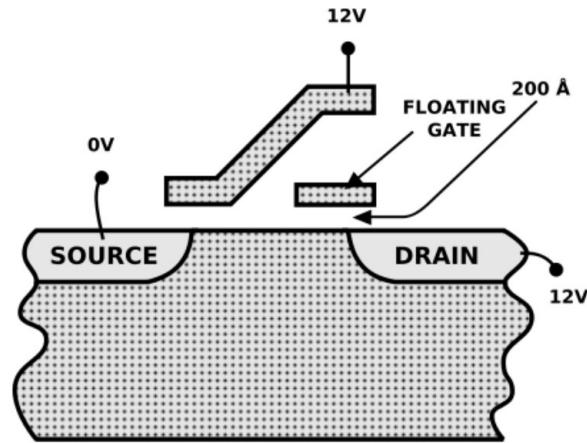
Flash Operation

- ❑ Double gate transistor.
- ❑ One gate is the control gate (CG) and the other is the floating gate (FG).
- ❑ The charge on the floating gate effects the V_t of the control gate.
- ❑ If FG is high V the transistor (nmos) will conduct at lower voltages on the CG
- ❑ If FG is low V the transistor (nmos) will conduct only if CG is higher.



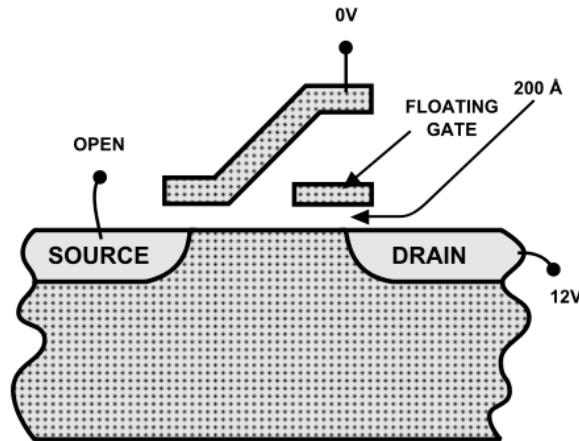
Flash Operation

Programming Via Hot Electron Injection



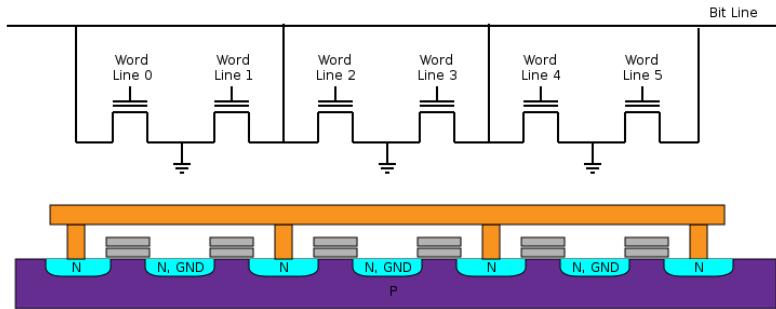
Programming a floating gate transistor.

Erasure Via Tunneling

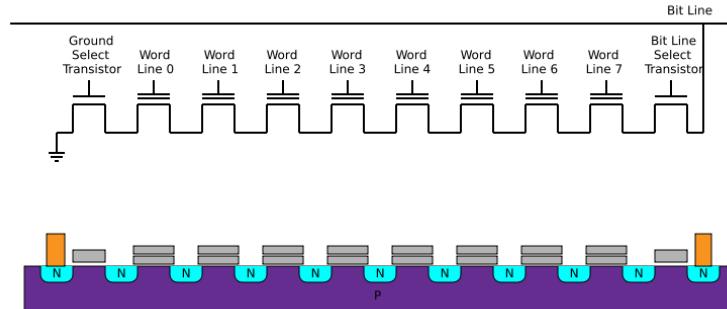


Erasing a floating gate transistor.

NOR gate memory



NAND Gate Memory



Images from Wikipedia

Flash Operation

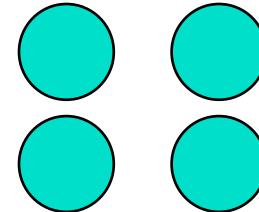
- ❑ Flash drives have proprietary hardware to control wear levelling, erasure and writing.
- ❑ Erasures occur per line of memory, while writes occur per bit.
 - Small changes to many lines is the worst case operation of SSDs.
 - If only some bit need to be changed from erase state (1s) to written state (0s) then this can be done without erasing an entire line. Eg 1111 can be changed to 0101 without erasing.
- ❑ Writing and reading are error prone so some forward error correction generally exists at the hardware level.
- ❑ Control has historically been non-standardized though there are groups attempting to standardize flash memory control.

Dependability, Reliability, Availability

- ❑ Reliability – measured by the **mean time to failure** (MTTF). Service interruption is measured by **mean time to repair** (MTTR)
- ❑ Availability – a measure of service accomplishment
$$\text{Availability} = \text{MTTF}/(\text{MTTF} + \text{MTTR})$$
- ❑ To increase MTTF, either improve the quality of the components or design the system to continue operating in the presence of faulty components
 1. Fault avoidance: preventing fault occurrence by construction
 2. Fault tolerance: using redundancy to correct or bypass faulty components (hardware)
 - Fault detection versus fault correction
 - Permanent faults versus transient faults

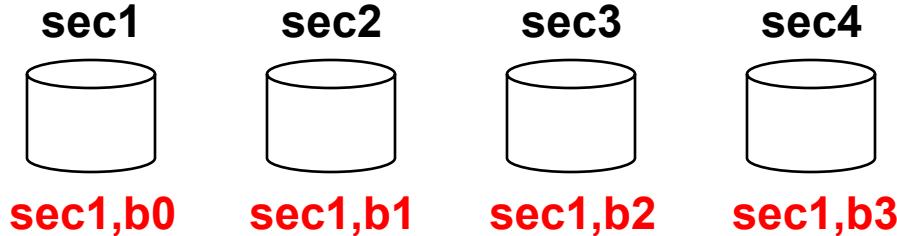
RAIDs: Disk Arrays

Redundant Array of
Inexpensive Disks



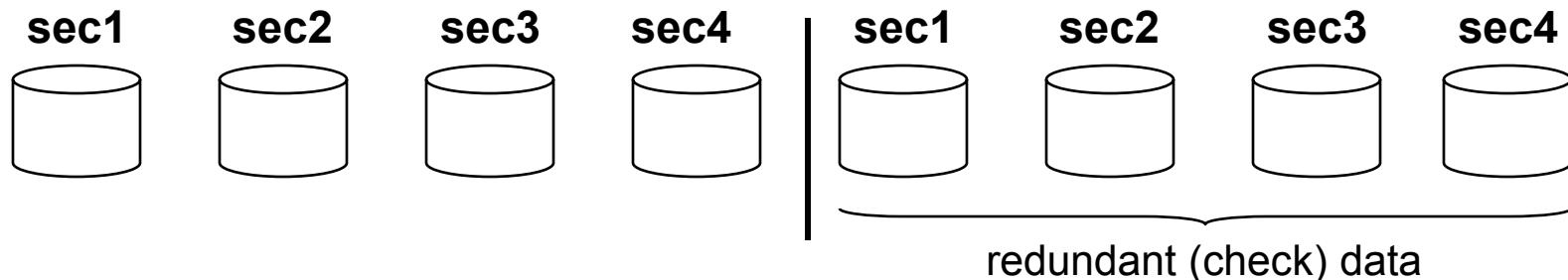
- ❑ Arrays of small and inexpensive disks
 - Increase potential **throughput** by having many disk drives
 - Data is spread over multiple disk
 - Multiple accesses are made to several disks at a time
- ❑ Reliability is lower than a single disk
- ❑ But **availability** can be improved by adding redundant disks (RAID)
 - Lost information can be reconstructed from redundant information
 - MTTR: mean time to repair is in the order of hours
 - MTTF: mean time to failure of disks is tens of years

RAID: Level 0 (No Redundancy; Striping)



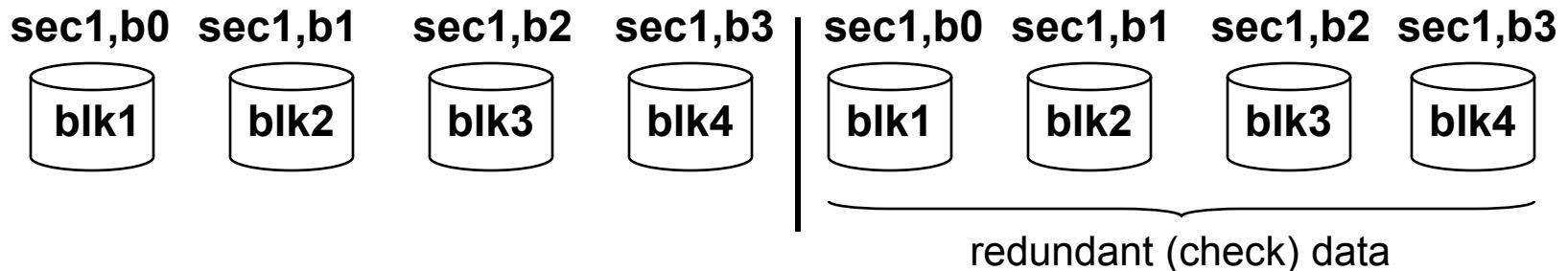
- ❑ Multiple smaller disks as opposed to one big disk
 - Spreading the sector over multiple disks – **striping** – means that multiple blocks can be accessed in parallel increasing the performance
 - A 4 disk system gives four times the throughput of a 1 disk system
 - Same cost as one *big* disk – assuming 4 small disks cost the same as one big disk
 - Latency is effectively the latency of the slowest disk.
- ❑ No redundancy, so what if one disk fails?
 - Failure of one or more disks is more likely as the number of disks in the system increases

RAID: Level 1 (Redundancy via Mirroring)



- ❑ Uses twice as many disks as RAID 0 (e.g., 8 smaller disks with the second set of 4 duplicating the first set) so there are always two copies of the data
 - # redundant disks = # of data disks so twice the cost of one big disk
 - writes have to be made to both sets of disks, so writes would be only 1/2 the performance of a RAID 0
- ❑ What if one disk fails?
 - If a disk fails, the system just goes to the “**mirror**” for the data

RAID: Level 0+1 (Striping with Mirroring)



- ❑ Combines the best of RAID 0 and RAID 1, data is striped across four disks and mirrored to four disks
 - Four times the throughput (due to striping)
 - # redundant disks = # of data disks so twice the cost of one big disk
 - writes have to be made to both sets of disks, so writes would be only 1/2 the performance of RAID 0
- ❑ What if one disk fails?
 - If a disk fails, the system just goes to the “**mirror**” for the data

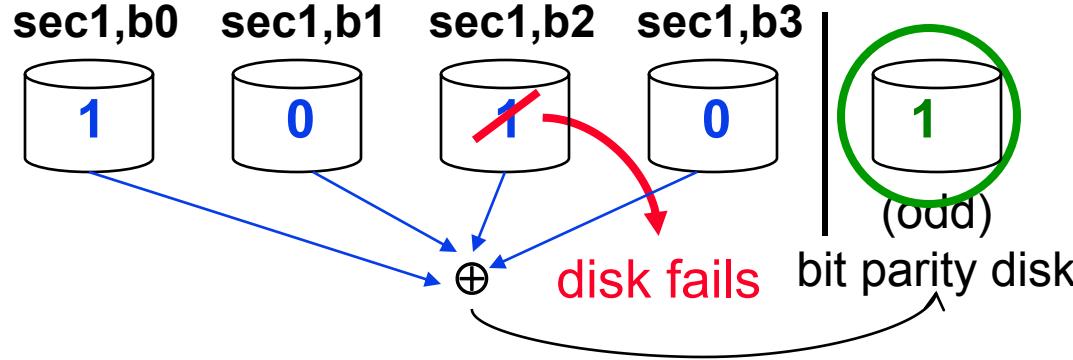
Disk Parity and Error Correcting Codes

- ❑ Imagine 3 disk, the first two store the data and the third stores the parity of the first two.

| Disk 1 (Data) | Disk 2 (Data) | Disk 3 (Parity) |
|---------------|---------------|--------------------|
| A | B | $A \text{ xor } B$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | ✗ | 0 |
| ✗ | 0 | 1 |
| 1 | 0 | ✗ |

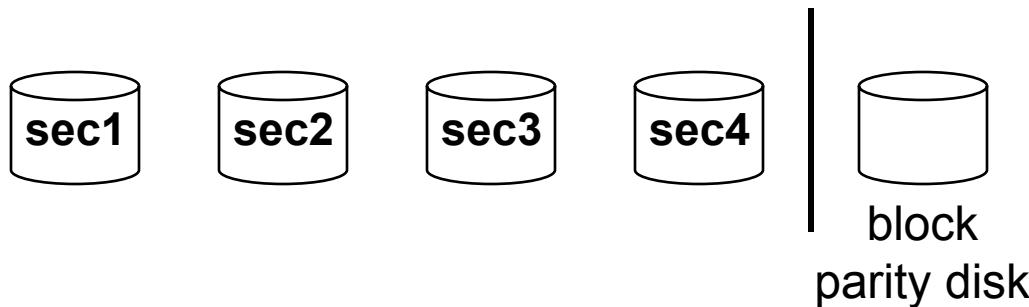
- ❑ Data can be recovered no matter which disk fails.

RAID: Level 3 (Byte-Interleaved Parity)



- ❑ Cost of higher availability is reduced to $1/N$ where N is the number of disks in a **protection group**
 - # redundant disks = $1 \times$ # of protection groups
 - writes require writing the new data to the data disk as well as computing the parity, meaning reading the other disks, so that the parity disk can be updated
- ❑ Can tolerate *limited* (single) disk failure, since the data can be **reconstructed**
 - reads require reading all the operational data disks as well as the parity disk to calculate the missing data that was stored on the failed disk

RAID: Level 4 (Block-Interleaved Parity)

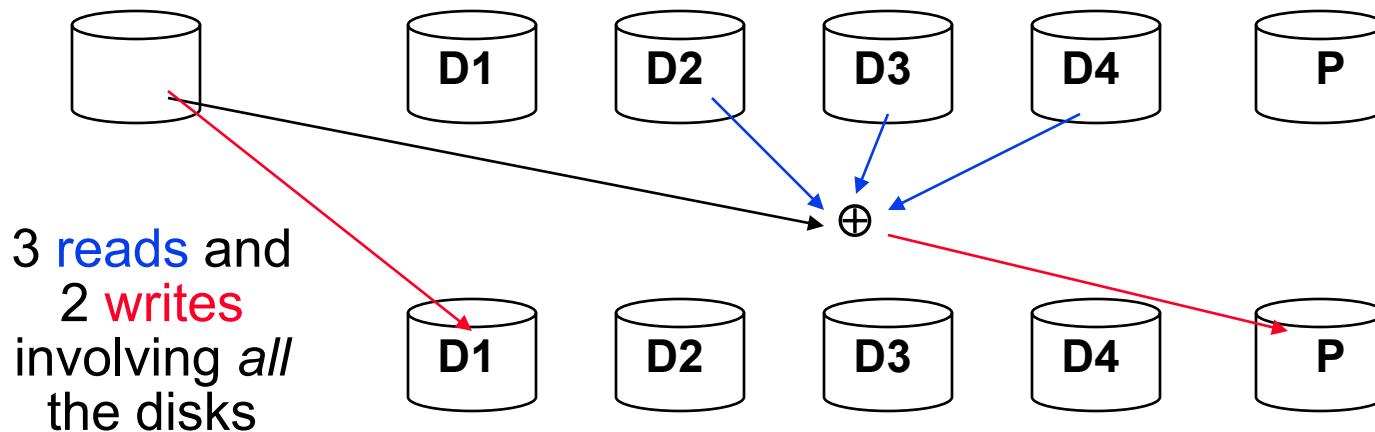


- ❑ Cost of higher availability still only $1/N$ but the parity is stored as **blocks** associated with sets of data blocks
 - Four times the throughput (striping)
 - # redundant disks = $1 \times$ # of protection groups
 - Supports “**small reads**” and “**small writes**” (reads and writes that go to just one (or a few) data disk in a protection group)
 - by watching which bits change when writing new information, need only to change the corresponding bits on the parity disk
 - the parity disk must be updated on every write, so it is a bottleneck for back-to-back writes
- ❑ Can tolerate *limited (1)* disk failure, since the data can be reconstructed

Small Writes

RAID 3 writes

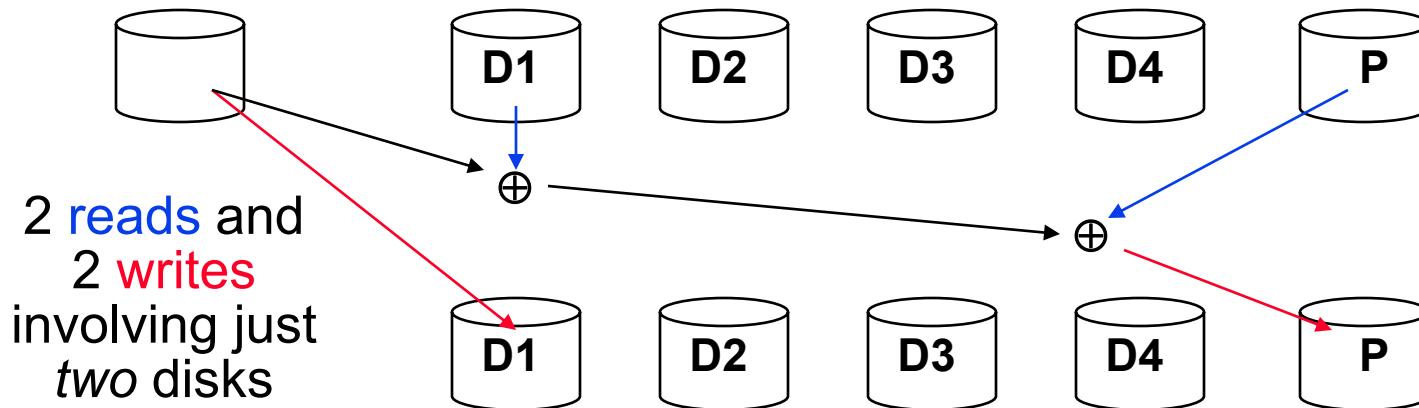
New D1 data



3 reads and
2 writes
involving *all*
the disks

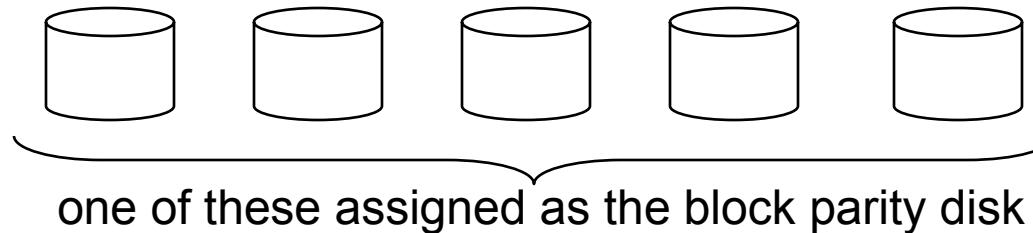
RAID 4 *small* writes

New D1 data



2 reads and
2 writes
involving just
two disks

RAID: Level 5 (Distributed Block-Interleaved Parity)

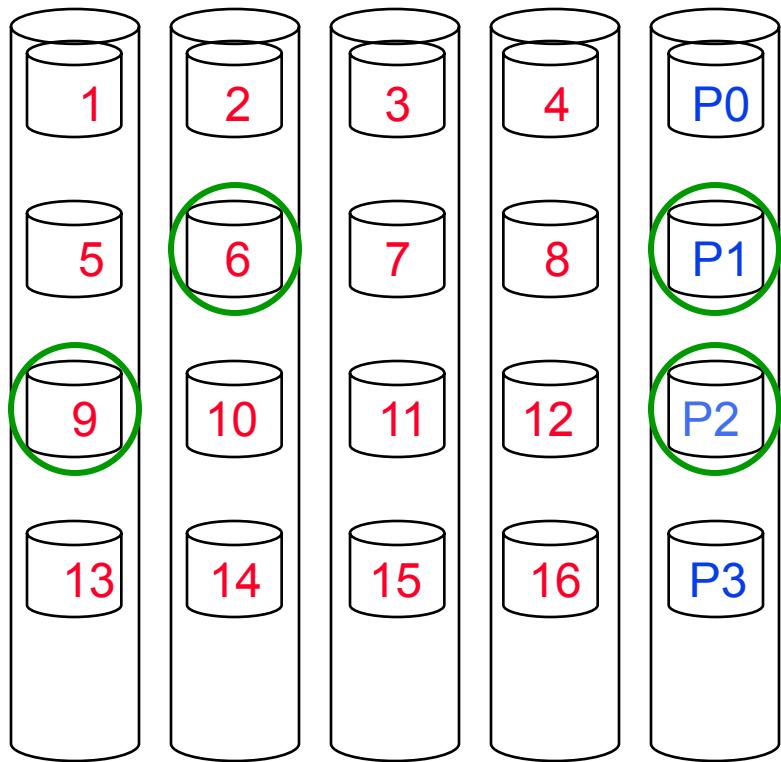


- ❑ Cost of higher availability still only $1/N$ but the parity block can be located on any of the disks so there is no single bottleneck for writes
 - Still four times the throughput (striping)
 - # redundant disks = $1 \times$ # of protection groups
 - Supports “**small reads**” and “**small writes**” (reads and writes that go to just one (or a few) data disk in a protection group)
 - Allows multiple simultaneous writes as long as the accompanying parity blocks are not located on the same disk
- ❑ Can tolerate *limited* (1) disk failure, since the data can be reconstructed

Distributing Parity Blocks

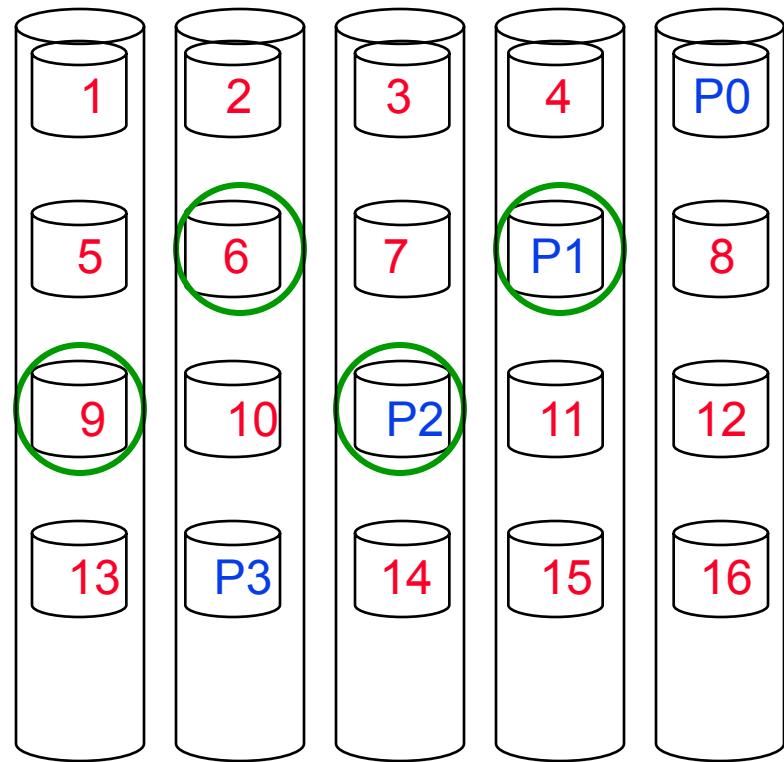
RAID 4

Time
↓



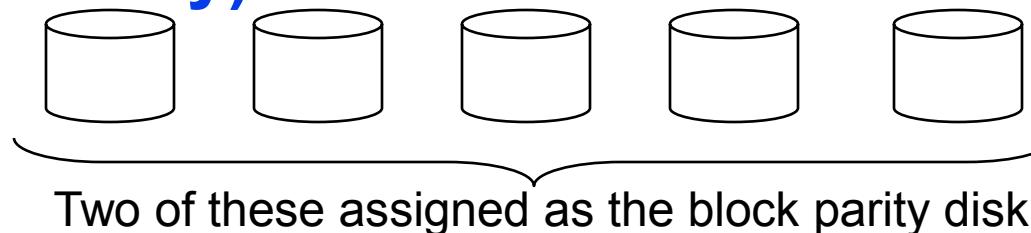
RAID 5

Can be done in parallel



- By distributing parity blocks to all disks, some small writes can be performed in parallel

RAID: Level 6 (Distributed Block-Interleaved double Parity)



- ❑ Cost of higher availability is $2/N$ but the parity blocks are located on any of the disks so there is no single bottleneck for writes
 - Three times the throughput (striping, assuming 5 disks)
 - # redundant disks = $2 \times$ # of protection groups
 - Supports “**small reads**” and “**small writes**” (reads and writes that go to just one (or a few) data disk in a protection group)
 - Allows multiple simultaneous writes as long as the accompanying parity blocks are not located on the same disk
- ❑ Can tolerate *limited* (2) disk failures, since the data can be reconstructed

Summary

- ❑ Four components of disk access time:
 - Seek Time: advertised to be 3 to 14 ms but lower in real systems
 - Rotational Latency: 5.6 ms at 5400 RPM and 2.0 ms at 15000 RPM
 - Transfer Time: 30 to 170 MB/s
 - Controller Time: typically less than .2 ms
- ❑ RAIDS can be used to improve availability
 - RAID 1 and RAID 5 – widely used in servers, one estimate is that 80% of disks in servers are RAIDs
 - RAID 0+1 (mirroring) – EMC, Tandem, IBM
 - RAID 3 – Storage Concepts
 - RAID 4 – Network Appliance
- ❑ RAIDS have enough redundancy to allow continuous operation, but not **hot swapping**

Next Lecture and Reminders

- ❑ Next lecture
 - I/O devices and systems

- ❑ Reminders
 - Assignment 1 due end next week
 - Try out Signaltap and the example test data this week.

ECE4074

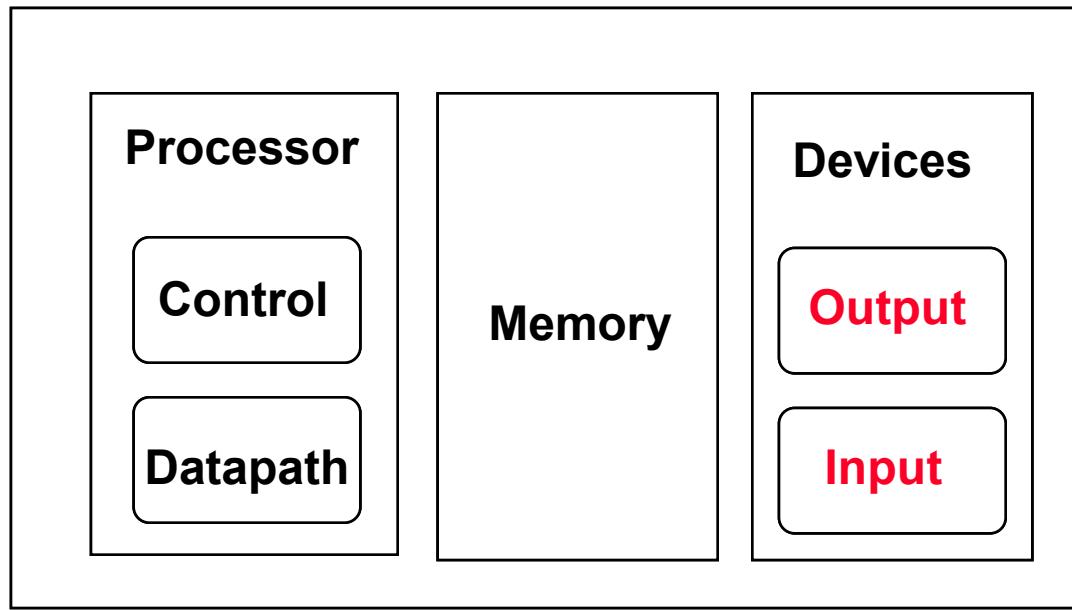
Computer Architecture

Semester 2 2014

Chapter 6B: I/O Systems

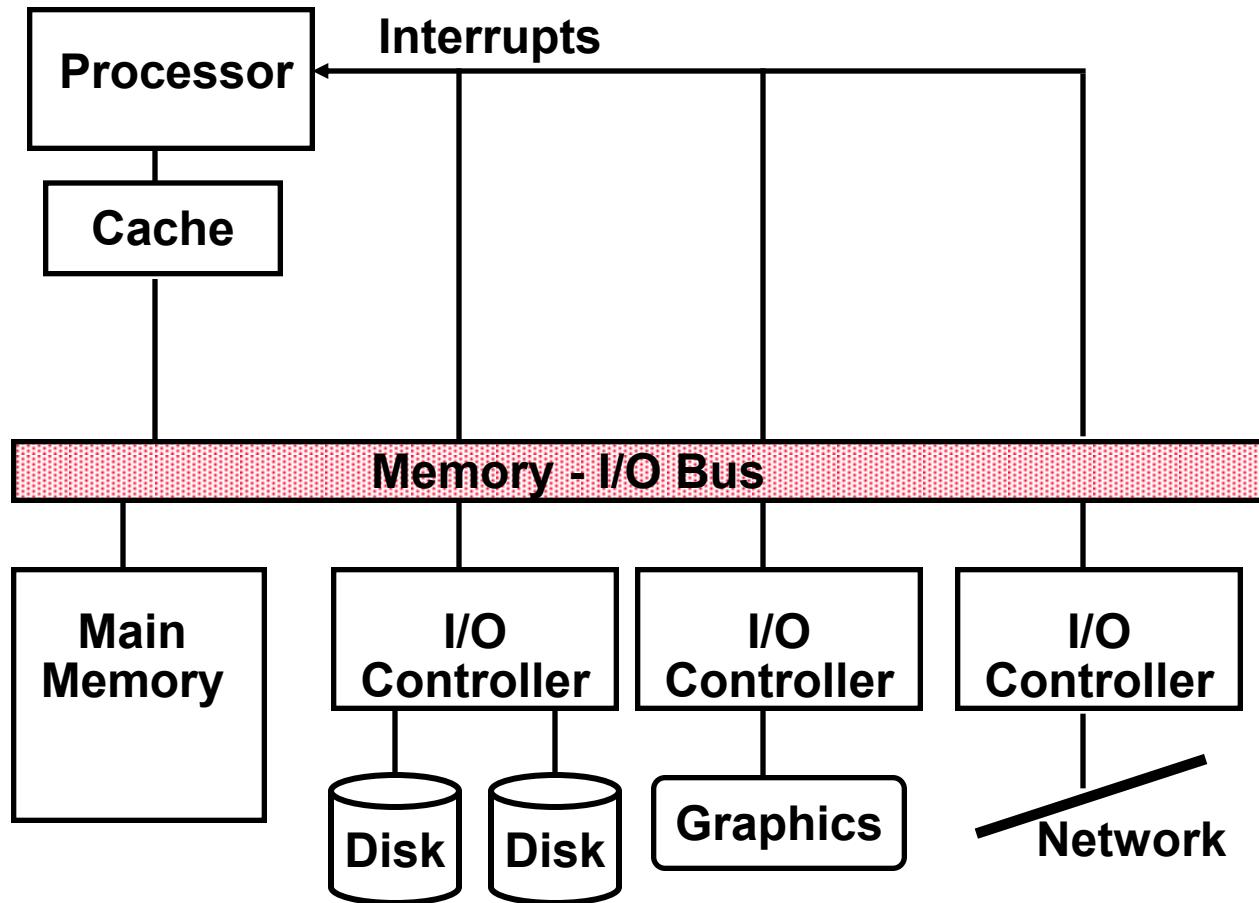
[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: Major Components of a Computer



- ❑ Important metrics for an I/O system
 - Performance
 - Expandability
 - Dependability
 - Cost, size, weight
 - Security

A Typical I/O System



Input and Output Devices

- ❑ I/O devices are incredibly diverse with respect to
 - Behavior – input, output or storage
 - Partner – human or machine
 - Data rate – the peak rate at which data can be transferred between the I/O device and the main memory or processor

| Device | Behavior | Partner | Data rate (Mb/s) |
|------------------|-----------------|----------------|-------------------------|
| Keyboard | input | human | 0.0001 |
| Mouse | input | human | 0.0038 |
| Laser printer | output | human | 3.2000 |
| Magnetic disk | storage | machine | 800.0000-3000.0000 |
| Graphics display | output | human | 800.0000-18000.0000 |
| Network/LAN | input or output | machine | 100.0000-10000.0000 |

8 orders of magnitude
range

I/O Performance Measures

- ❑ I/O bandwidth (throughput) – amount of information that can be input (output) and communicated across an interconnect (e.g., a bus) to the processor/memory (I/O device) per unit time
 1. How much data can we move through the system in a certain time?
 2. How many I/O operations can we do per unit time?
- ❑ I/O response time (latency) – the total elapsed time to accomplish an input or output operation
 - An especially important performance metric in real-time systems
- ❑ Many applications require *both* high throughput and short response times

I/O System Interconnect Issues

- A **bus** is a shared communication link (a single set of wires used to connect multiple subsystems) that needs to support a range of devices with widely varying latencies and data transfer rates
 - Advantages
 - Versatile – new devices can be added easily and can be moved between computer systems that use the same bus standard
 - Low cost – a single set of wires is shared in multiple ways
 - Disadvantages
 - Creates a communication bottleneck – bus **bandwidth** limits the maximum I/O **throughput**
- The maximum bus speed is largely limited by
 - The **length** of the bus
 - The **number** of devices on the bus

Types of Buses

- ❑ Processor-memory bus (“Front Side Bus”, proprietary)
 - Short and high speed
 - Matched to the memory system to maximize the memory-processor bandwidth
 - Optimized for cache block transfers
- ❑ I/O bus (industry standard, e.g., SCSI, USB, Firewire)
 - Usually is lengthy and slower
 - Needs to accommodate a wide range of I/O devices
 - Use either the processor-memory bus or a backplane bus to connect to memory
- ❑ Backplane bus (industry standard, e.g., ATA, PClexpress)
 - Allow processor, memory and I/O devices to coexist on a single bus
 - Used as an intermediary bus connecting I/O busses to the processor-memory bus

I/O Transactions

- ❑ An I/O transaction is a sequence of operations over the interconnect that includes a request and may include a response either of which may carry data. A transaction is initiated by a single request and may take *many* individual bus operations. An I/O transaction typically includes two parts
 1. Sending the address
 2. Receiving or sending the data
- ❑ Bus transactions are defined by what they do to memory
 - output** • A **read** transaction reads data from memory (to either the processor or an I/O device)
 - input** • A **write** transaction writes data to the memory (from either the processor or an I/O device)

Synchronous and Asynchronous Buses

❑ Synchronous bus (e.g., processor-memory buses)

- Includes a clock in the control lines and has a fixed protocol for communication that is **relative** to the clock
- Advantage: involves very little logic and can run very fast
- Disadvantages:
 - Every device communicating on the bus must use same clock rate
 - To avoid clock skew and RC delays, they cannot be long if they are fast

❑ Asynchronous bus (e.g., I/O buses)

- It is not clocked, so requires a handshaking protocol and additional control lines (ReadReq, Ack, DataRdy)
- Advantages:
 - Can accommodate a wide range of devices and device speeds
 - Can be lengthened without worrying about clock skew or synchronization problems
- Disadvantage: slow(er), prone to design errors.

ATA Cable Sizes

- ❑ Companies have transitioned from synchronous, parallel wide buses to asynchronous narrow buses
 - Reflection on wires and clock skew makes it difficult to use 16 to 64 parallel wires running at a high clock rate (e.g., ~400MHz) so companies have moved to buses with a few one-way wires running at a very high “clock” rate (~2GHz)



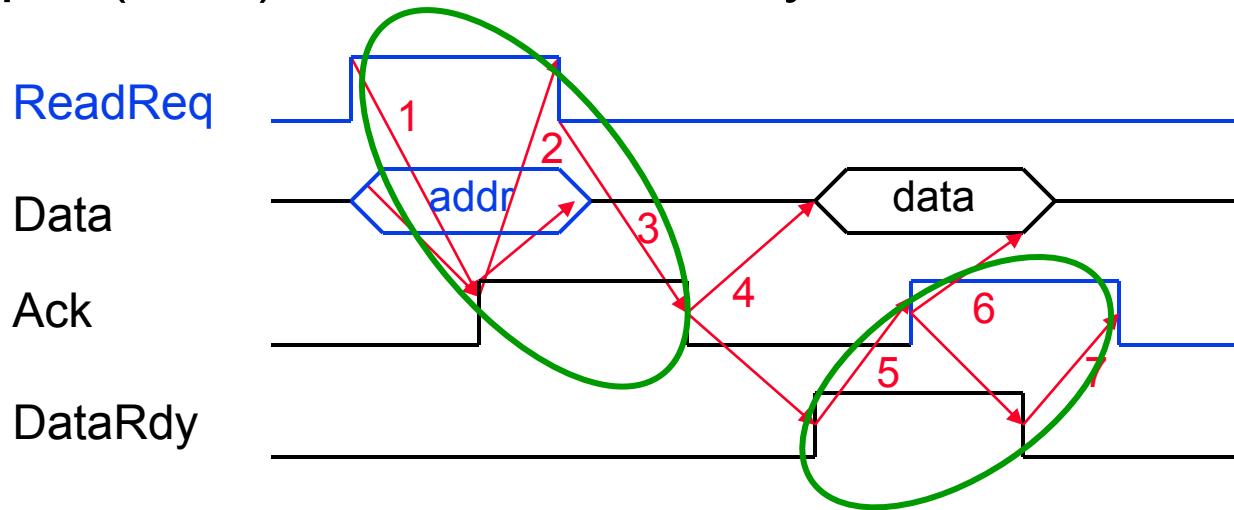
- Serial ATA cables (**red**) are much thinner than parallel ATA cables (**green**)

USB 3.0 Cables

- ❑ The latest revision of USB 3.0 uses more wires per cable.
- ❑ USB 1.0 & 2.0 use 4 pins
- ❑ USB 3.0 uses 9 pins
 - 2 pin differential shared Rx/Tx signal for USB 2.0 backward compatibility
 - 2 pin differential high speed Rx
 - 2 pin differential high speed Tx
 - 2 pins ground
 - 1 pin power
- ❑ Differential pairs are commonly used to reduce common mode noise, like differential amplifiers. Delays on these two lines must be precisely matched.

Asynchronous Bus Handshaking Protocol

- Output (read) data from memory to an I/O device



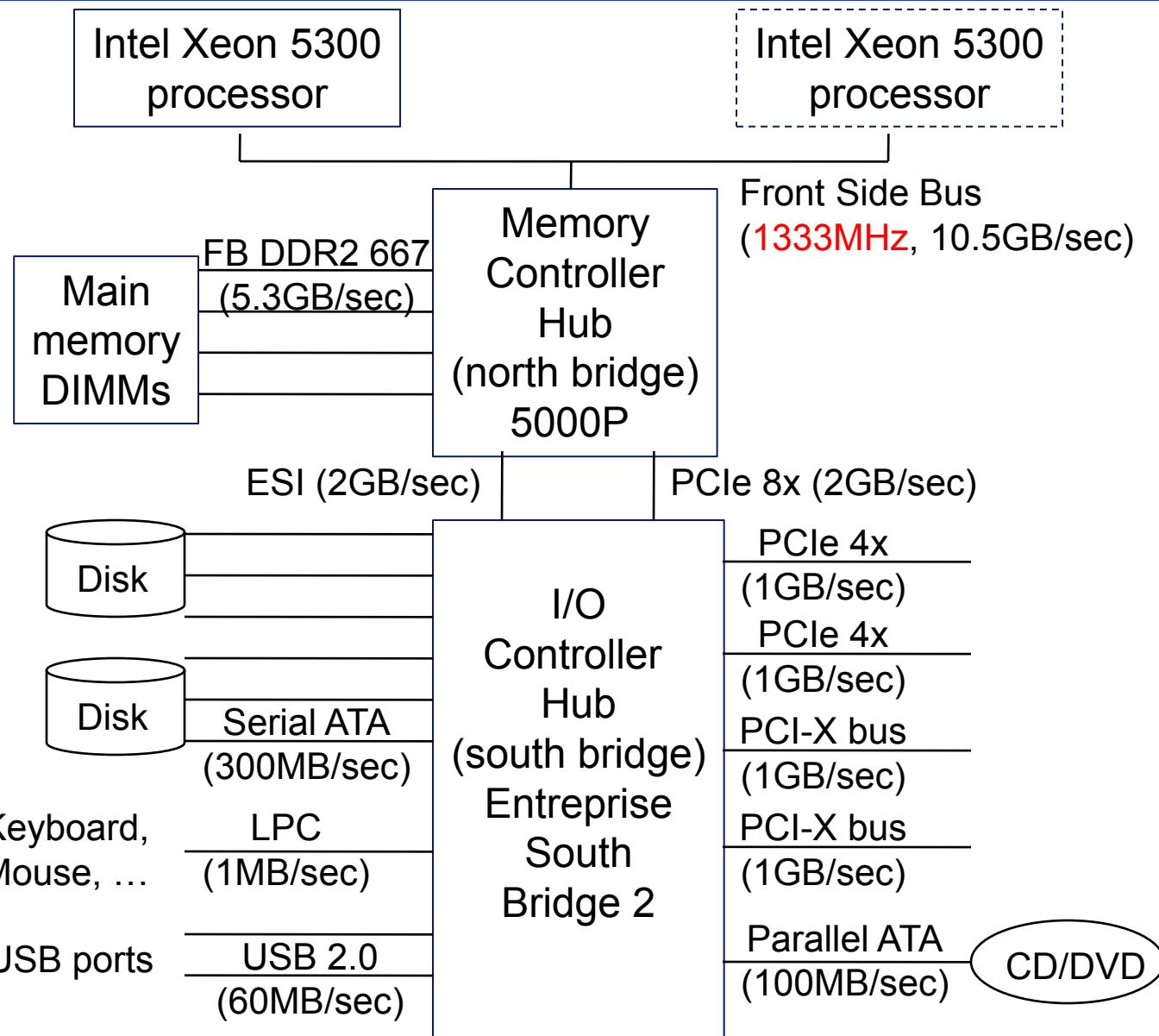
I/O device signals a request by raising **ReadReq** and putting the **addr** on the data lines

1. Memory sees **ReadReq**, reads **addr** from data lines, and raises **Ack**
2. I/O device sees **Ack** and releases the **ReadReq** and data lines
3. Memory sees **ReadReq** go low and drops **Ack**
4. When memory has data ready, it places it on data lines and raises **DataRdy**
5. I/O device sees **DataRdy**, reads the data from data lines, and raises **Ack**
6. Memory sees **Ack**, releases the data lines, and drops **DataRdy**
7. I/O device sees **DataRdy** go low and drops **Ack**

Key Characteristics of I/O Standards

| | Thunderbolt | USB 3.0 | PCIe v3.0 | SATA 3.0 | SA SCSI |
|----------------------------|-------------------------------|----------------|---|-----------------|----------------|
| Use | External | External | Internal | Internal | External |
| Devices per channel | 6 | 127 | 1 | 1 | 4 |
| Max length | 3 meters | 5 meters | 0.5 meters | 1 meter | 8 meters |
| Data Width | 2 per channel (rx,tx) | 2 (rx,tx) | 2 per lane (rx,tx) | 2 (rx,tx) | 4 |
| Raw Bitrate | 10 Gb/s (per channel up to 2) | 5 Gb/s | 8Gb/s per lane (up 16 per device, 40 per CPU) | 6 Gb/s | 12 Gb/s |
| Data Rate (after overhead) | 1000 MB/s (per channel) | 400 MB/s | 985MB/s (per lane) | 600 MB/s | 1228.8 MB/s |
| Hot pluggable? | Yes | Yes | Depends | Yes | Yes |

A Typical I/O System



Interfacing I/O Devices to the Processor, Memory, and OS

- ❑ The operating system acts as the interface between the I/O hardware and the program requesting I/O since
 - Multiple programs using the processor **share** the I/O system
 - I/O systems usually use interrupts which are handled by the OS
 - Low-level control of an I/O device is complex and detailed
- ❑ Thus OS must handle interrupts generated by I/O devices and supply routines for low-level I/O device operations, provide equitable access to the shared I/O resources, protect those I/O devices/activities to which a user program doesn't have access, and schedule I/O requests to enhance system throughput
 - OS must be able to give commands to the I/O devices
 - I/O device must be able to notify the OS about its status
 - Must be able to transfer data between the memory and the I/O device

Communication of I/O Devices and Processor

❑ How the processor directs the I/O devices

- Special I/O instructions
 - Must specify both the device and the command
- Memory-mapped I/O
 - Portions of the high-order memory address space are assigned to each I/O device
 - Read and writes to those memory addresses are interpreted as commands to the I/O devices
 - Load/stores to the I/O address space can *only* be done by the OS

❑ How I/O devices communicate with the processor

- Polling – the processor periodically checks the status of an I/O device (through the OS) to determine its need for service
 - Processor is totally in control – but does **all** the work
 - Can waste a lot of processor time due to speed differences
- Interrupt-driven I/O – the I/O device issues an interrupt to indicate that it needs attention

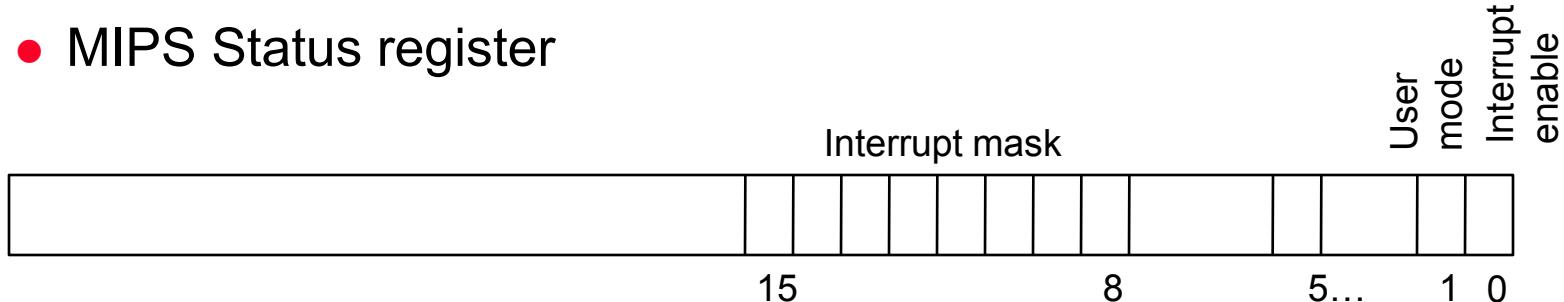
Interrupt Driven I/O

- ❑ An I/O interrupt is **asynchronous** wrt instruction execution
 - Is not associated with any instruction so doesn't prevent any instruction from completing
 - You can pick your own convenient point to handle the interrupt
- ❑ With I/O interrupts
 - Need a way to identify the device generating the interrupt
 - Can have different urgencies (so need a way to **prioritize** them)
- ❑ Advantages of using interrupts
 - Relieves the processor from having to continuously poll for an I/O event; user program progress is only suspended during the actual transfer of I/O data to/from user memory space
- ❑ Disadvantage – special hardware is needed to
 - Indicate the I/O device causing the interrupt and to save the necessary information prior to servicing the interrupt and to resume normal processing after servicing the interrupt

Interrupt Priority Levels

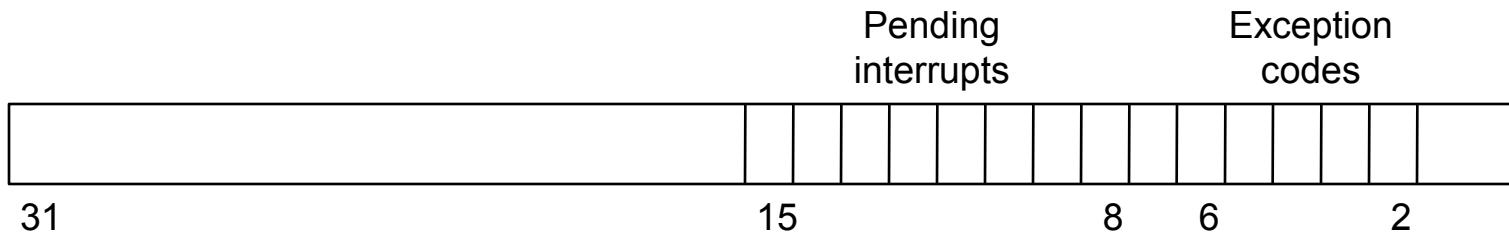
- Priority levels can be used to direct the OS the order in which the interrupts should be serviced

- MIPS Status register



- Determines who can interrupt the processor (if Interrupt enable is 0, none can interrupt)

- MIPS Cause register



- To enable a Pending interrupt, the correspond bit in the Interrupt mask must be 1
- Once an interrupt occurs, the OS can find the reason in the Exception codes field

Interrupt Handling Steps

1. Logically AND the Pending interrupt field and the Interrupt mask file to see which enabled interrupts could be the culprit. Make copies of both Status and Cause registers.
 2. Select the higher priority of these interrupts (leftmost is highest)
 3. Save the Interrupt mask field
 4. Change the Interrupt mask field to disable all interrupts of equal or lower priority
 5. Save the processor state prior to “handling” the interrupt
 6. Set the Interrupt enable bit (to allow higher-priority interrupts)
 7. Call the appropriate interrupt handler routine
 8. Before returning from interrupt and restoring the status and cause registers, disable interrupts by setting Int' Enbl' to 0. *rfe* instruction restores previous level's interrupt enable and level bits.
- ❑ Interrupt priority levels (IPLs) assigned by the OS to each process can be raised and lowered via changes to the Status's Interrupt mask field

Direct Memory Access (DMA)

- ❑ For high-bandwidth devices (like disks) interrupt-driven I/O would consume a *lot* of processor cycles
- ❑ With DMA, the DMA controller has the ability to transfer large blocks of data **directly** to/from the memory without involving the processor
 1. The processor initiates the DMA transfer by supplying the I/O device address, the operation to be performed, the memory address destination/source, the number of bytes to transfer
 2. The DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
 3. When the DMA transfer is complete, the DMA controller interrupts the processor to let it know that the transfer is complete
- ❑ There may be multiple DMA devices in one system
 - Processor and DMA controllers contend for bus cycles and for memory

The DMA Stale Data Problem

- ❑ In systems with caches, there can be two copies of a data item, one in the cache and one in the main memory
 - For a DMA input (from disk to memory) – the processor will be using **stale** data if that location is also in the cache
 - For a DMA output (from memory to disk) and a write-back cache – the I/O device will receive **stale** data if the data is in the cache and has not yet been written back to the memory
- ❑ The coherency problem can be solved by
 1. Routing all I/O activity through the cache – expensive and a large negative performance impact
 2. Having the OS invalidate all the entries in the cache for an I/O input or force write-backs for an I/O output (called a **cache flush**)
 3. Providing hardware to *selectively* invalidate cache entries – i.e., need a **snooping** cache controller

DMA and Virtual Memory Considerations

- ❑ Should the DMA work with virtual addresses or physical addresses?
- ❑ If working with physical addresses
 - Must constrain all of the DMA transfers to stay within one page because if it crosses a page boundary, then it won't necessarily be contiguous in memory
 - If the transfer won't fit in a single page, it can be broken into a series of transfers (each of which fit in a page) which are handled individually and *chained* together
- ❑ If working with virtual addresses
 - The DMA controller will have to translate the virtual address to a physical address (i.e., will need a TLB structure)
- ❑ Whichever is used, the OS must cooperate by not remapping pages while a DMA transfer involving that page is in progress

I/O System Performance

- ❑ Designing an I/O system to meet a set of bandwidth and/or latency constraints means
 1. Finding the weakest link in the I/O system – the component that constrains the design
 - The processor and memory system ?
 - The underlying interconnection (i.e., bus) ?
 - The I/O controllers ?
 - The I/O devices themselves ?
 2. (Re)configuring the weakest link to meet the bandwidth and/or latency requirements
 3. Determining requirements for the rest of the components and (re)configuring them to support this latency and/or bandwidth

I/O System Design Example

❑ Given a server system with

- Workload: 64KB disk reads
 - Each I/O op requires 200,000 user-code instructions and 100,000 OS instructions
- Each CPU: 10^9 instructions/sec
- FSB: 10.6 GB/sec peak
- DRAM DDR2 667MHz: 5.336 GB/sec
- PCI-E 8× bus: $8 \times 250\text{MB/sec} = 2\text{GB/sec}$
- Disks: 15,000 rpm, 2.9ms avg. seek time, 112MB/sec transfer rate, 8 disks total

❑ What I/O rate can be sustained?

- For random reads, and for sequential reads

Design Example (cont)

❑ I/O rate for CPUs

- Per core: $10^9 / (100,000 + 200,000) = 3,333$
- 8 cores: 26,667 ops/sec

❑ Random reads, I/O rate for disks

- Assume actual seek time is average/4
- Time/op = seek + latency + transfer
 $= 2.9\text{ms}/4 + 4\text{ms}/2 + 64\text{KB}/(112\text{MB/s}) = 3.3\text{ms}$
- 303 ops/sec per disk, 2424 ops/sec for 8 disks

❑ Sequential disk reads

- $112\text{MB/s} / 64\text{KB} = 1750 \text{ ops/sec per disk}$
- 14,000 ops/sec for 8 disks

Design Example (cont)

- ❑ PCI-E I/O rate

- $2\text{GB/sec} / 64\text{KB} = 31,250 \text{ ops/sec}$

- ❑ DRAM I/O rate

- $5.336 \text{ GB/sec} / 64\text{KB} = 83,375 \text{ ops/sec}$

Design Example (cont)

❑ FSB I/O rate

- Assume we can sustain half the peak rate
- $5.3 \text{ GB/sec} / 64\text{KB} = 81,540 \text{ ops/sec}$ per FSB
- 163,080 ops/sec for 2 FSBs

❑ Weakest link: disks

- 2424 ops/sec random, 14,000 ops/sec sequential
- Other components have ample headroom to accommodate these rates

Next Lecture and Reminders

- ❑ Next lecture
 - Multiple issue processors
- ❑ Assignment 1 Due Next Week
- ❑ In your report include:
 - Max processor frequency/Min clock period
 - Cause of critical path.
 - Number of clock cycles required to execute your program
 - MIPS code/Verilog code.
 - Explanations and justifications of the features of your processor and MIPS code.
 - What are the disadvantages of your design decisions?
 - What you would change given more time?

ECE4074 Computer Architecture Semester 2 2014

Chapter 4C: The Processor, Part C

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Extracting Yet More Performance

- ❑ Increase the depth of the pipeline to increase the clock rate – **superpipelining**
 - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline flip flop overhead (i.e., the pipeline flip flop accounts for a larger and larger percentage of the clock cycle time)
- ❑ Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – **multiple-issue**
 - The instruction execution rate, CPI, will be less than 1, so instead we use **IPC**: instructions per clock cycle
 - E.g., a 3 GHz, four-way multiple-issue processor can execute at a peak rate of 12 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
 - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?

Types of Parallelism

- ❑ Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor *might* be able to execute at the same time
 - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
- ❑ Data-level parallelism (DLP)

```
for(i = 0;i<100;i++)
    A[i] = A[i] + 1;
```
- ❑ Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
 - Determined by the number of instructions that can be fetched and executed at the same time
- ❑ To achieve high performance, need *both* ILP and machine parallelism

Multiple-Issue Processor Styles

❑ Static multiple-issue processors (aka **VLIW**)

- Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
- E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
 - 128-bit “bundles” containing three instructions, each 41-bits plus a 5-bit template field (which specifies which FU each instruction needs)
 - Five types of functional units (IntALU, Mmedia, Dmem, FPALU, Branch)
 - Extensive support for speculation and predication

❑ Dynamic multiple-issue processors (aka **superscalar**)

- Decisions on which instructions to execute simultaneously (in the range of 2 to 8) are being made dynamically (at run time by the hardware)
- E.g., IBM Power series, Pentium (all), MIPS R10K, AMD K5 and above

Multiple-Issue Datapath Responsibilities

- ❑ Must handle, with a combination of hardware and software fixes, the fundamental limitations of
 - How many instructions to issue in one clock cycle – **issue slots**
 - Storage (data) dependencies – aka data hazards
 - Limitation more severe in a SS/VLIW processor due to (usually) low ILP
 - Procedural dependencies – aka control hazards
 - Ditto, but even more severe
 - Use dynamic branch prediction to help resolve the ILP issue
 - Resource conflicts – aka structural hazards
 - A SS/VLIW processor has a much larger number of potential resource conflicts
 - Functional units may have to arbitrate for result buses and register-file write ports
 - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Speculation

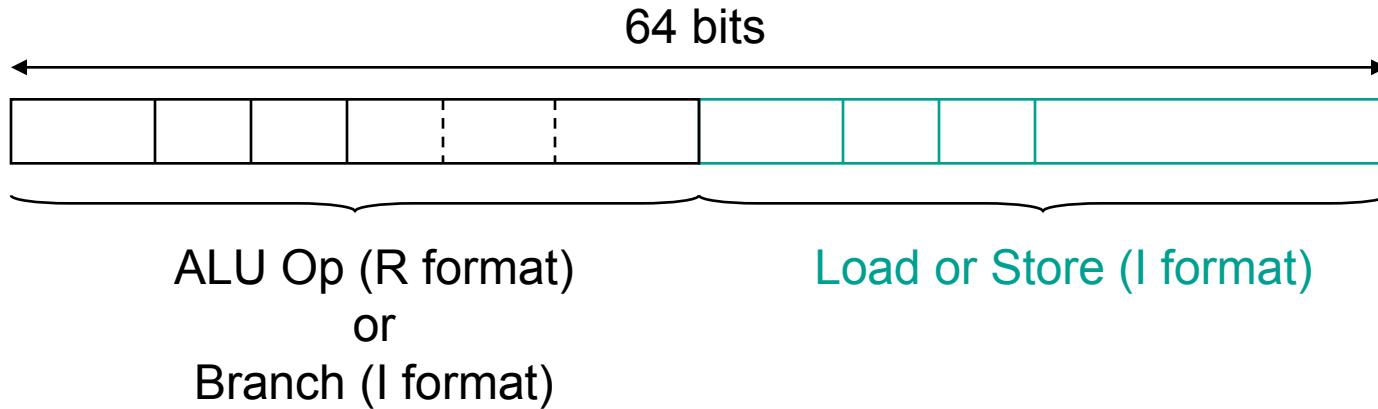
- ❑ Speculation is used to allow execution of future instr's that (may) depend on the speculated instruction
 - Speculate on the outcome of a conditional branch (**branch prediction**)
 - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (**load speculation**)
- ❑ Must have (hardware and/or software) mechanisms for
 - Checking to see if the guess was correct
 - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
- ❑ Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur

Static Multiple Issue Machines (VLIW)

- ❑ Static multiple-issue processors (aka VLIW) use the compiler (at compile-time) to statically decide which instructions to issue and execute simultaneously
 - Issue packet – the set of instructions that are bundled together and issued in one clock cycle – think of it as one large instruction with multiple operations
 - The mix of instructions in the packet (bundle) is usually restricted – a single “instruction” with several predefined fields
 - The compiler does static branch prediction and code scheduling to reduce (control) or eliminate (data) hazards
- ❑ VLIW's have
 - Multiple functional units
 - Multi-ported register files
 - Wide instruction bus

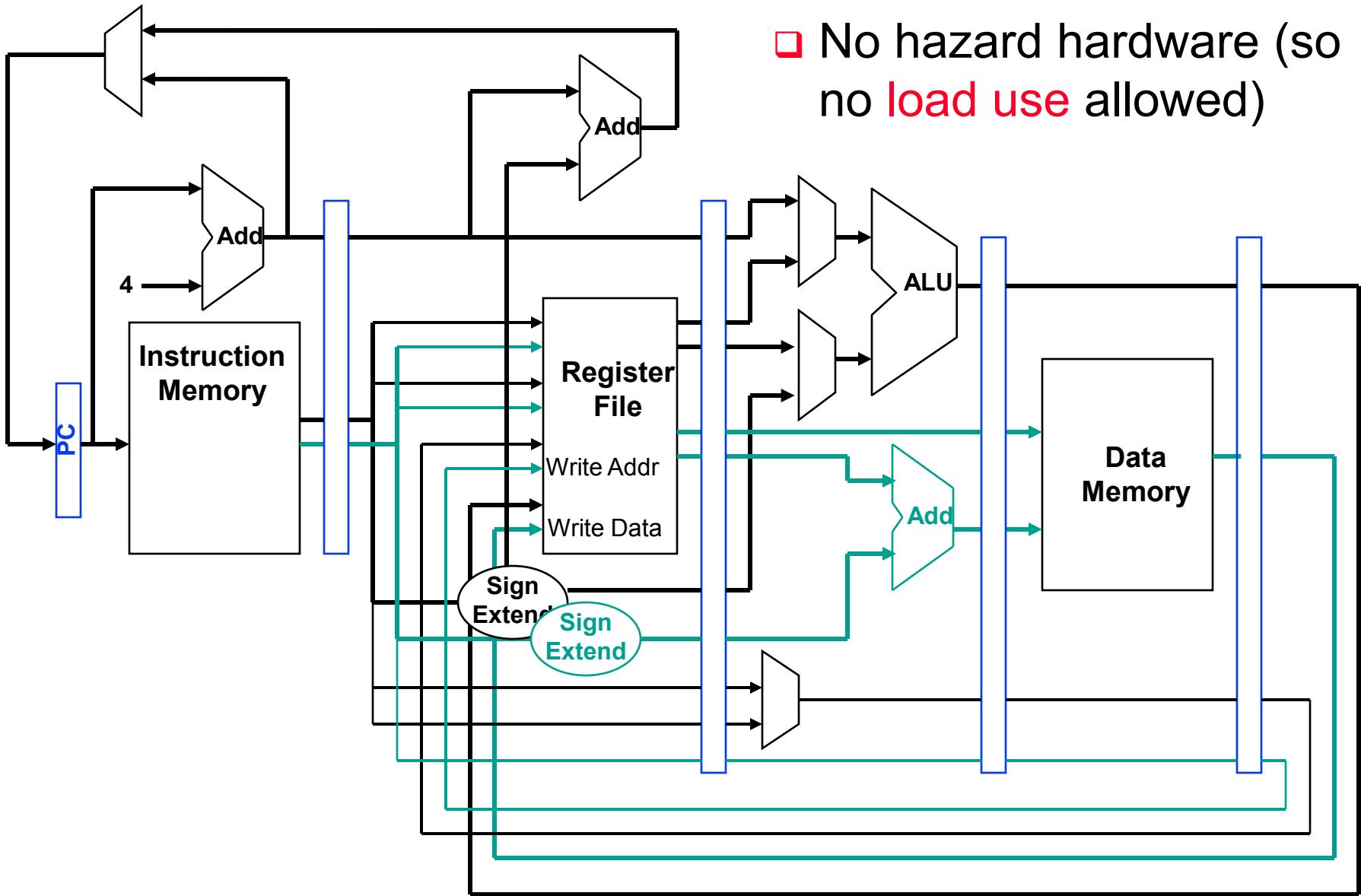
An Example: A VLIW MIPS

- Consider a 2-issue MIPS with a 2 instr bundle



- Instructions are always fetched, decoded, and issued in pairs
 - If one instr of the pair can not be used, it is replaced with a noop
- Need 4 read ports and 2 write ports and a separate memory address adder

A MIPS VLIW (2-issue) Datapath



Code Scheduling Example

- ❑ Consider the following loop code

```
lp:    lw      $t0,0($s1)      # $t0=array element
        addu   $t0,$t0,$s2      # add scalar in $s2
        sw     $t0,0($s1)      # store result
        addi   $s1,$s1,-4      # decrement pointer
        bne   $s1,$0,lp       # branch if $s1 != 0
```

- ❑ Must “schedule” the instructions to avoid pipeline stalls
 - Instructions in one bundle *must* be independent
 - Must separate load use instructions from their loads by one cycle
 - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
 - Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

| | ALU or branch | Data transfer | CC |
|-----|-----------------------|------------------|----|
| lp: | | lw \$t0, 0(\$s1) | 1 |
| | addi \$s1, \$s1, -4 ← | | 2 |
| | addu \$t0, \$t0, \$s2 | | 3 |
| | bne \$s1, \$0, lp | sw \$t0, 4(\$s1) | 4 |
| | | | 5 |

- ❑ Four clock cycles to execute 5 instructions for a
 - CPI of 0.8 (versus the best case of 0.5)
 - IPC of 1.25 (versus the best case of 2.0)
 - noops don't count towards performance !!

Loop Unrolling

- ❑ Loop unrolling – multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP
- ❑ Apply loop unrolling (4 times for our example) and then **schedule** the resulting code
 - Eliminate unnecessary loop overhead instructions
 - Schedule so as to avoid load use hazards
- ❑ During unrolling the compiler applies **register renaming** to eliminate all data dependencies that are not true data dependencies

Unrolled Code Example

```
lp:    lw     $t0,0($s1)      # $t0=array element
        lw     $t1,-4($s1)      # $t1=array element
        lw     $t2,-8($s1)      # $t2=array element
        lw     $t3,-12($s1)     # $t3=array element
        addu $t0,$t0,$s2        # add scalar in $s2
        addu $t1,$t1,$s2        # add scalar in $s2
        addu $t2,$t2,$s2        # add scalar in $s2
        addu $t3,$t3,$s2        # add scalar in $s2
        sw    $t0,0($s1)        # store result
        sw    $t1,-4($s1)       # store result
        sw    $t2,-8($s1)       # store result
        sw    $t3,-12($s1)      # store result
        addi $s1,$s1,-16        # decrement pointer
        bne  $s1,$0,lp          # branch if $s1 != 0
```

The Scheduled Code (Unrolled)

| | ALU or branch | Data transfer | CC |
|-----|---------------------|------------------|----|
| lp: | addi \$s1,\$s1,-16 | lw \$t0,0(\$s1) | 1 |
| | | lw \$t1,12(\$s1) | 2 |
| | addu \$t0,\$t0,\$s2 | lw \$t2,8(\$s1) | 3 |
| | addu \$t1,\$t1,\$s2 | lw \$t3,4(\$s1) | 4 |
| | addu \$t2,\$t2,\$s2 | sw \$t0,16(\$s1) | 5 |
| | addu \$t3,\$t3,\$s2 | sw \$t1,12(\$s1) | 6 |
| | | sw \$t2,8(\$s1) | 7 |
| | bne \$s1,\$0,lp | sw \$t3,4(\$s1) | 8 |

- ❑ Eight clock cycles to execute 14 instructions for a
 - CPI of 0.57 (versus the best case of 0.5)
 - IPC of 1.8 (versus the best case of 2.0)

Compiler Support for VLIW Processors

- ❑ The compiler packs groups of **independent** instructions into the bundle
 - Done by code re-ordering (trace scheduling)
- ❑ The compiler uses loop unrolling to expose more ILP
- ❑ The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur
- ❑ While superscalars use dynamic prediction, VLIW's primarily depend on the compiler for branch prediction
 - Loop unrolling reduces the number of conditional branches
 - Prediction eliminates if-the-else branch structures by replacing them with predicted instructions
- ❑ The compiler predicts memory bank references to help minimize memory bank conflicts

VLIW Advantages & Disadvantages

❑ Advantages

- Simpler hardware (potentially less power hungry)
- Potentially more scalable
 - Allow more instr's per VLIW bundle and add more Functional Units (FUs)

❑ Disadvantages

- Programmer/compiler complexity and longer compilation times
 - Deep pipelines and long latencies can be confusing (making peak performance elusive)
- Lock step operation, i.e., on hazard all future issues stall until hazard is resolved
- **Object (binary) code incompatibility**
- Needs lots of program memory bandwidth
- **Code bloat**
 - Noops are a waste of program memory space
 - Loop unrolling to expose more ILP uses more program memory space

Dynamic Multiple Issue Machines (SS)

- ❑ Dynamic multiple-issue processors (aka **SuperScalar**) use hardware at run-time to dynamically decide which instructions to issue and execute simultaneously
- ❑ **Instruction-fetch and issue** – fetch instructions, decode them, and *issue* them to a FU to await execution
 - Defines the **Instruction lookahead** capability – fetch, decode and issue instructions beyond the current instruction
- ❑ **Instruction-execution** – as soon as the source operands and the FU are ready, the result can be calculated
 - Defines the **processor lookahead** capability – complete execution of issued instructions beyond the current instruction
- ❑ **Instruction-commit** – when it is safe to, write back results to the RegFile or D\$ (i.e., change the machine state)

In-Order vs Out-of-Order

- ❑ Instruction fetch and decode units are **required** to issue instructions in-order so that dependencies can be tracked
- ❑ The commit unit is **required** to write results to registers and memory in program fetch order so that
 - if exceptions occur the only registers updated will be those written by instructions before the one causing the exception
 - if branches are mispredicted, those instructions executed after the mispredicted branch don't change the machine state (i.e., we use the commit unit to correct incorrect speculation)
- ❑ Although the front end (fetch, decode, and issue) and back end (commit) of the pipeline run in-order, the FUs are free to initiate execution whenever the data they need is available – out-of-(program) order execution
 - Allowing out-of-order execution increases the amount of ILP

Out-of-Order Execution

- With out-of-order execution, a later instruction may execute **before** a previous instruction so the hardware needs to resolve both **read before write** and **write before write** data hazards

```
lw      $t0, 0($s1)  
addu   $t0, $t1, $s2  
.  
.  
.  
sub    $t2, $t0, $s2
```

- If the `lw` write to `$t0` occurs **after** the `addu` write, then the `sub` gets an incorrect value for `$t0`
- The `addu` has an **output dependency** on the `lw` – **write before write**
 - The issuing of the `addu` might have to be stalled if its result could later be overwritten by a previous instruction that takes longer to complete

Antidependencies

- ❑ Also have to deal with **antidependencies** – when a later instruction (that executes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that executes later)

```
R3 := R3 * R5  
R4 := R3 + 1  
R3 := R5 + 1
```

Antidependency
True data dependency
Output dependency

- ❑ The constraint is similar to that of true data dependencies, except *reversed*
 - Instead of the later instruction using a value (not yet) produced by an earlier instruction (**read before write**), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (**write before read**)

Dependencies Review

- ❑ Each of the three data dependencies
 - True data dependencies (**read before write**)
 - Antidependencies (**write before read**)
 - Output dependencies (**write before write**)
- ❑ manifests itself through the use of registers (or other storage locations)
- ❑ True dependencies represent the flow of data and information through a program
- ❑ Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations leading to **storage conflicts**

Storage Conflicts and Register Renaming

- ❑ Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
 - Provide additional registers that are used to reestablish the correspondence between registers and values
 - Allocated dynamically by the hardware in SS processors
- ❑ **Register renaming** – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

$$\begin{array}{l} \textcolor{blue}{\circlearrowleft} \quad \textcolor{red}{R3} := \textcolor{red}{R3} * R5 \\ R4 := \textcolor{blue}{R3} + 1 \\ \textcolor{red}{\circlearrowleft} \quad \textcolor{red}{R3} := R5 + 1 \end{array} \quad \Rightarrow \quad \begin{array}{l} R3b := R3a * R5a \\ R4a := \textcolor{blue}{R3b} + 1 \\ R3c := R5a + 1 \end{array}$$

- The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

Resource Usage and Utilization

- ❑ If multiple instructions are to be executed simultaneously then existing resources (ALUs, Data Memory, Instruction Memory, Register Files) need to be able to accommodate multiple executions.
- ❑ For memory units:
 - More ports -> Slower speed, more transistors
- ❑ For ALUs:
 - More ALUs -> More transistors
- ❑ In modern systems:
 - More transistors -> more leakage current = more power when not being used.
- ❑ It is not power efficient to power resources that are not being used.

Resource Usage and Utilization

- ❑ Higher peak performance requires more possibilities of parallel resources (high MLP), increasing the problem of efficiency during “idle” time.
- ❑ Idle time is caused by lack of need for tasks (i.e. operating system is idle) or lack of ILP.
- ❑ Solutions to the power efficiency problem:
 - Switch off power (or reduce voltage) to parts of the processor that aren't being used.
 - Used in many modern systems.
 - Reverting from an idle state takes time as large capacitance power lines need to get to required voltage.
 - Turn off clock (or reduce clock frequency) to parts of the processor that aren't being used.
 - Great where Flip-Flops are used.
 - Doesn't help static (leakage) current.

Resource Usage and Utilization

- ❑ Solutions to the power efficiency problem:
 - Increase ILP
- ❑ Different threads of execution are guaranteed to have 100% ILP (with the exception of inter-thread communication)
- ❑ Intel HyperThreading (Simultaneous Multi-Threading) allows the operating system to send multiple threads to the same physical core.
- ❑ Therefore Intel HyperThreading increases ILP.

HyperThreading (SMT)

- ❑ Switching threads is costly:
 - Processor state (registers) has to be saved to memory between each thread switch.
- ❑ To execute two threads on the same physical core, two different processor states are needed.
 - Two sets register files, program counters, exception registers etc. are needed to make the one physical core appear as two logical cores.
 - Functional units (ie ALU, Memory access etc) are shared between the two logical cores.
 - By sharing execution units, they are used more of the time and hence the ratio of dynamic power to static power goes up.
- ❑ HyperThreading requires more transistors and hence creates its own power issues, so it is not used in very low power designs ie Mobile Phone processors.

Summary: Extracting More Performance

- ❑ To achieve high performance, need both **machine parallelism** and **instruction level parallelism (ILP)** by
 - Superpipelining
 - Static multiple-issue (VLIW)
 - Dynamic multiple-issue (superscalar)
- ❑ A processor's instruction issue and execution policies impact the available ILP
 - In-order fetch, issue, and commit and out-of-order execution
 - Pipelining creates **true** dependencies (**read before write**)
 - Out-of-order execution creates **antidependencies** (**write before read**)
 - Out-of-order execution creates **output dependencies** (**write before write**)
 - In-order commit allows speculation (to increase ILP) and is required to implement precise interrupts
- ❑ Register renaming can solve these storage dependencies

CISC vs RISC vs SS vs VLIW

| | CISC | RISC | Superscalar | VLIW |
|-------------------------|---|-------------------------------|-----------------------------------|-----------------------------|
| Instr size | More variable size | fixed size | either | fixed size (but large) |
| Instr format | More variable format | fixed format | either | fixed format |
| Registers | few, some special Limited # of ports | Many GP Limited # of ports | GP and rename (RUU) Many ports | many, many GP Many ports |
| Memory reference | embedded in many instr's | load/store | load/store | load/store |
| Key Issues | decode complexity | data forwarding, hazards | hardware dependency resolution | (compiler) code scheduling |

Evolution of Pipelined, SS Processors

| | Year | Clock Rate | # Pipe Stages | Issue Width | OOO? | Cores /Chip | Power |
|----------------------------|------|------------|---------------|-------------|------|-------------|-------|
| Intel 486 | 1989 | 25 MHz | 5 | 1 | No | 1 | 5 W |
| Intel Pentium | 1993 | 66 MHz | 5 | 2 | No | 1 | 10 W |
| Intel Pentium Pro | 1997 | 200 MHz | 10 | 3 | Yes | 1 | 29 W |
| Intel Pentium 4 Willamette | 2001 | 2000 MHz | 22 | 3 | Yes | 1 | 75 W |
| Intel Pentium 4 Prescott | 2004 | 3600 MHz | 31 | 3 | Yes | 1 | 103 W |
| Intel Core | 2006 | 2930 MHz | 14 | 4 | Yes | 2 | 75 W |
| Sun USPARC III | 2003 | 1950 MHz | 14 | 4 | No | 1 | 90 W |
| Sun T1 (Niagara) | 2005 | 1200 MHz | 6 | 1 | No | 8 | 70 W |
| Intel i7 Haswell | 2013 | 4400 MHz | 14-19 | 4 | Yes | 4 | 88 W |

Next Lecture and Reminders

- ❑ Next lecture two lectures
 - Overview of a sample superscalar processor data path

- ❑ Reminders
 - Assignment demonstration and report due this week.
 - Assignment files (Report and code) can be uploaded to moodle now.
 - Assignment 2 and sample pseudo code will be available on Friday.

ECE4074

Computer Architecture

Semester 2 2014

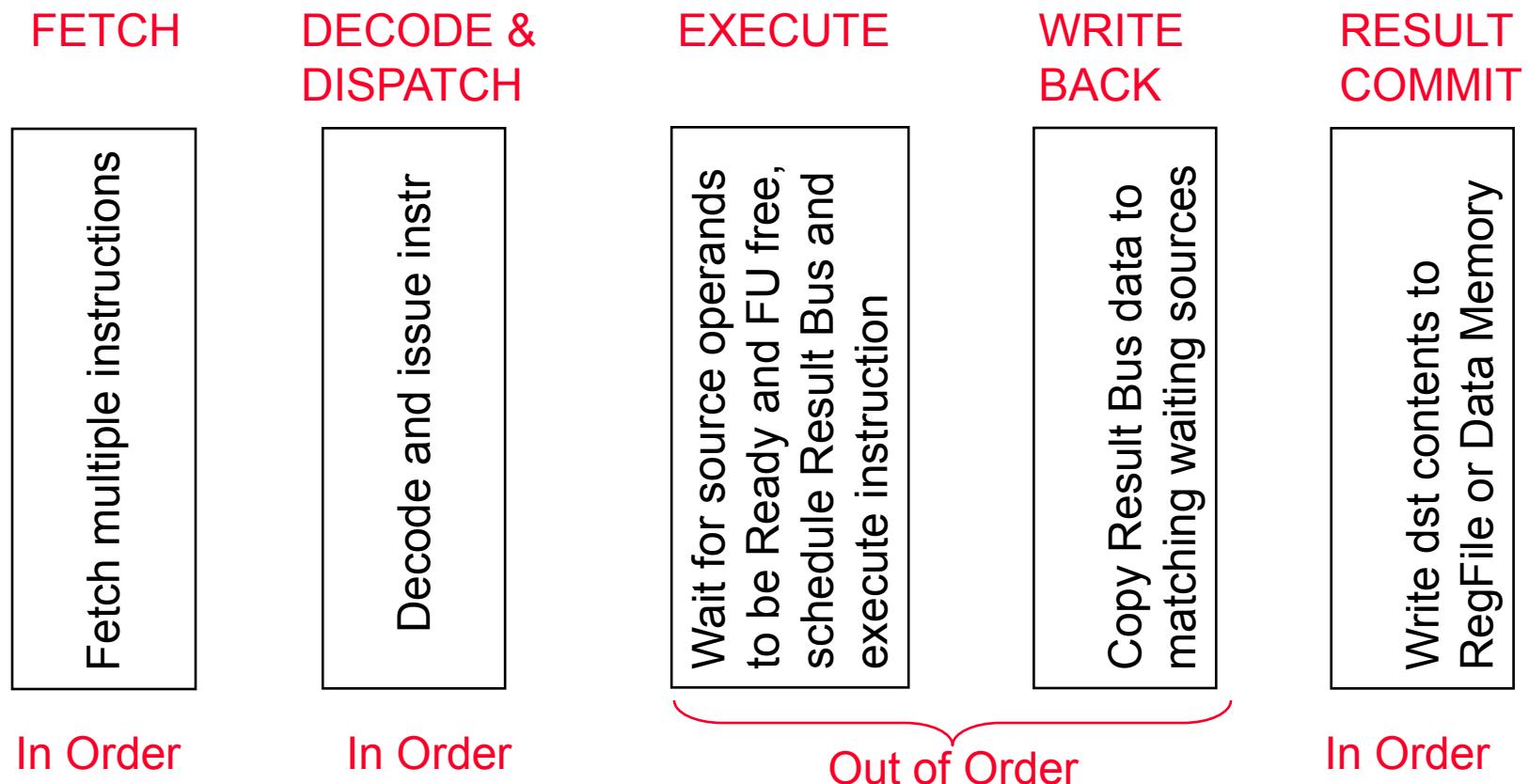
A “Simple” SS Processor

[Adapted from Sohi, Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers, *IEEE Transactions on Computers*, Vol 39, No 3, 1990.]

Review: Extracting More Performance

- ❑ To achieve high performance, need both **machine parallelism** and **instruction level parallelism (ILP)** by
 - Superpipelining
 - Static multiple-issue (VLIW)
 - Dynamic multiple-issue (superscalar (SS))
- ❑ A SS processor's instruction issue and execution policies impact the available ILP
 - In-order fetch, issue, and commit and out-of-order execution
 - Pipelining creates **true dependencies** (**read before write**)
 - Out-of-order execution creates **antidependencies** (**write before read**) and **output dependencies** (**write before write**)
 - In-order commit allows speculation (to increase ILP) and is required to implement precise interrupts
- ❑ Register renaming can solve these storage dependencies

Review: SS Pipeline Stage Functions



Speedup Measurements

- ❑ The speedup of the SS processor is

- Assumes the processors have the same IC & CC

$$\text{speedup } = s_n = \frac{\# \text{ scalar cycles}}{\# \text{ superscalar cycles}}$$

- ❑ To compute average speedup performance can use

- Geometric mean

$$GM = \sqrt[n]{\prod_{i=1}^n s_i}$$

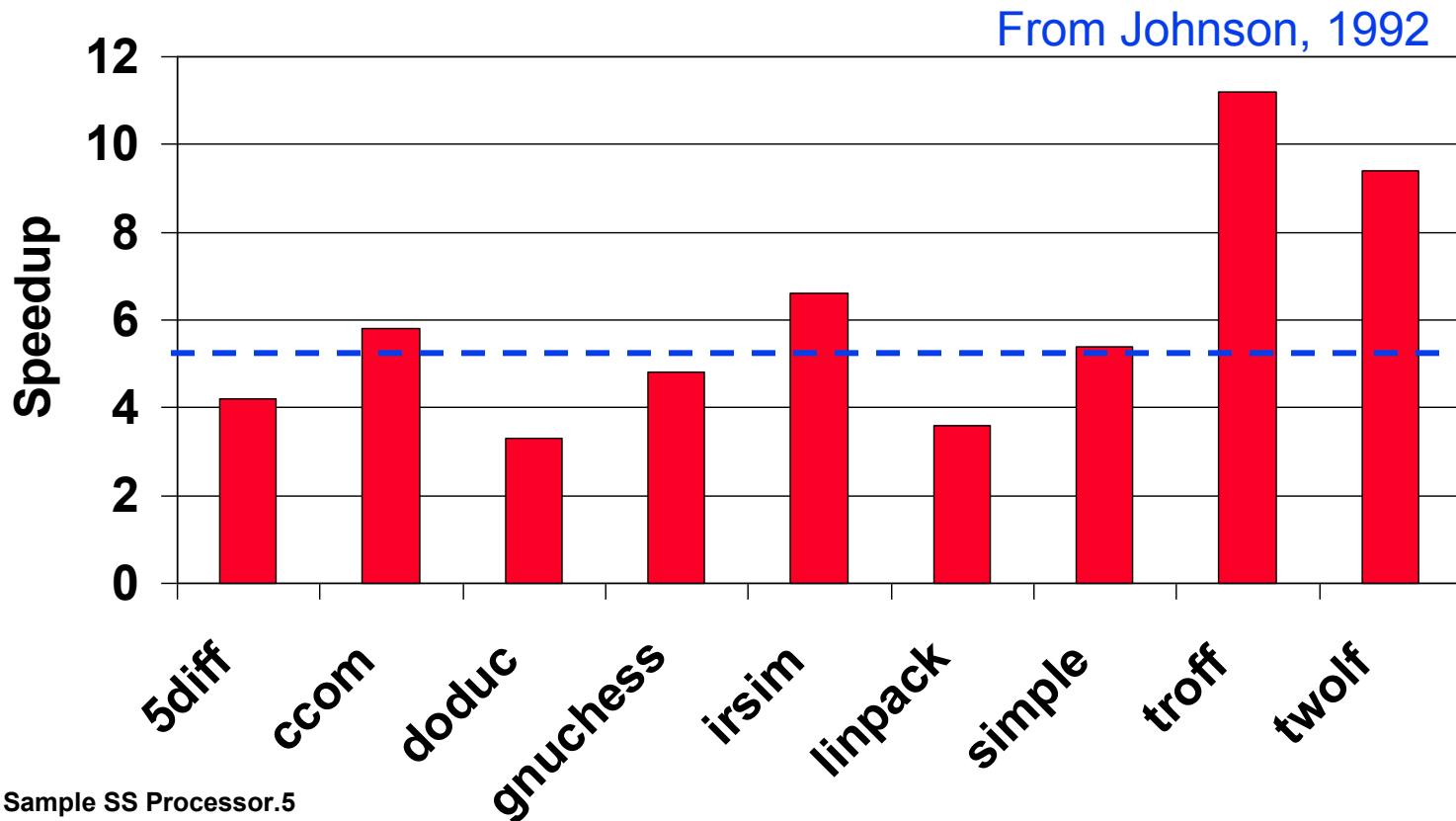
- Harmonic mean

$$HM = n / \left(\sum_{i=1}^n 1/s_i \right)$$

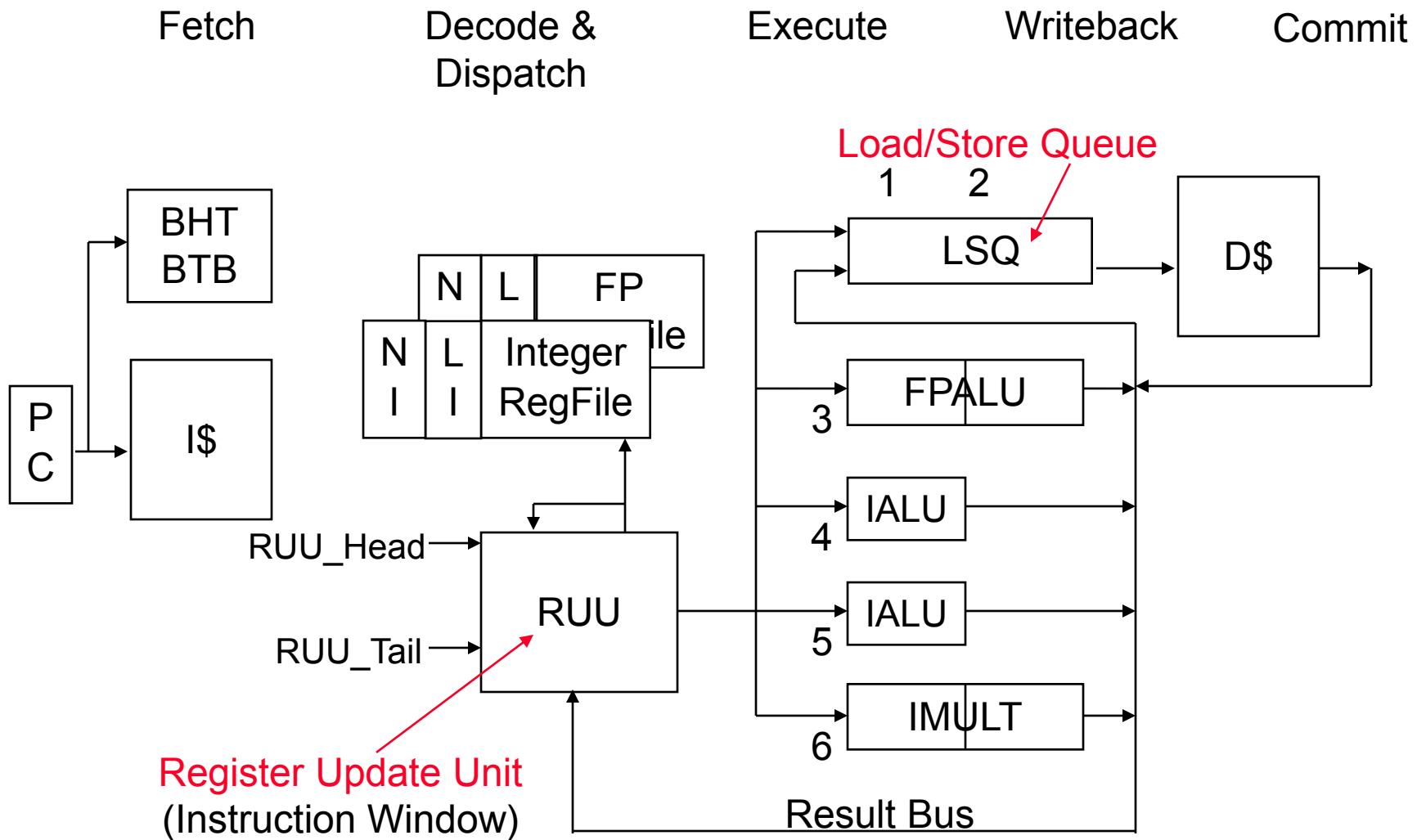
- assigns a larger weighting to the programs with the smallest speedup
 - EX: two programs with same scalar cycles, with a SS speedup of 2 for program1 and 25 for program2
 - $GM = \sqrt{(2 * 25)} = 7.1$
 - $HM = 2 / (.5 + .04) = 2 / .54 = 3.7$

Maximum (Theoretical) SS Speedups

- ❑ The highest speedup that can be achieved with “ideal” machine parallelism (ignoring structural hazards, data dependencies, and control dependencies)
 - HM of 5.4 is the highest average speedup for these benchmarks that can be achieved even with **ideal** machine parallelism!



Baseline Superscalar MIPS Processor Model



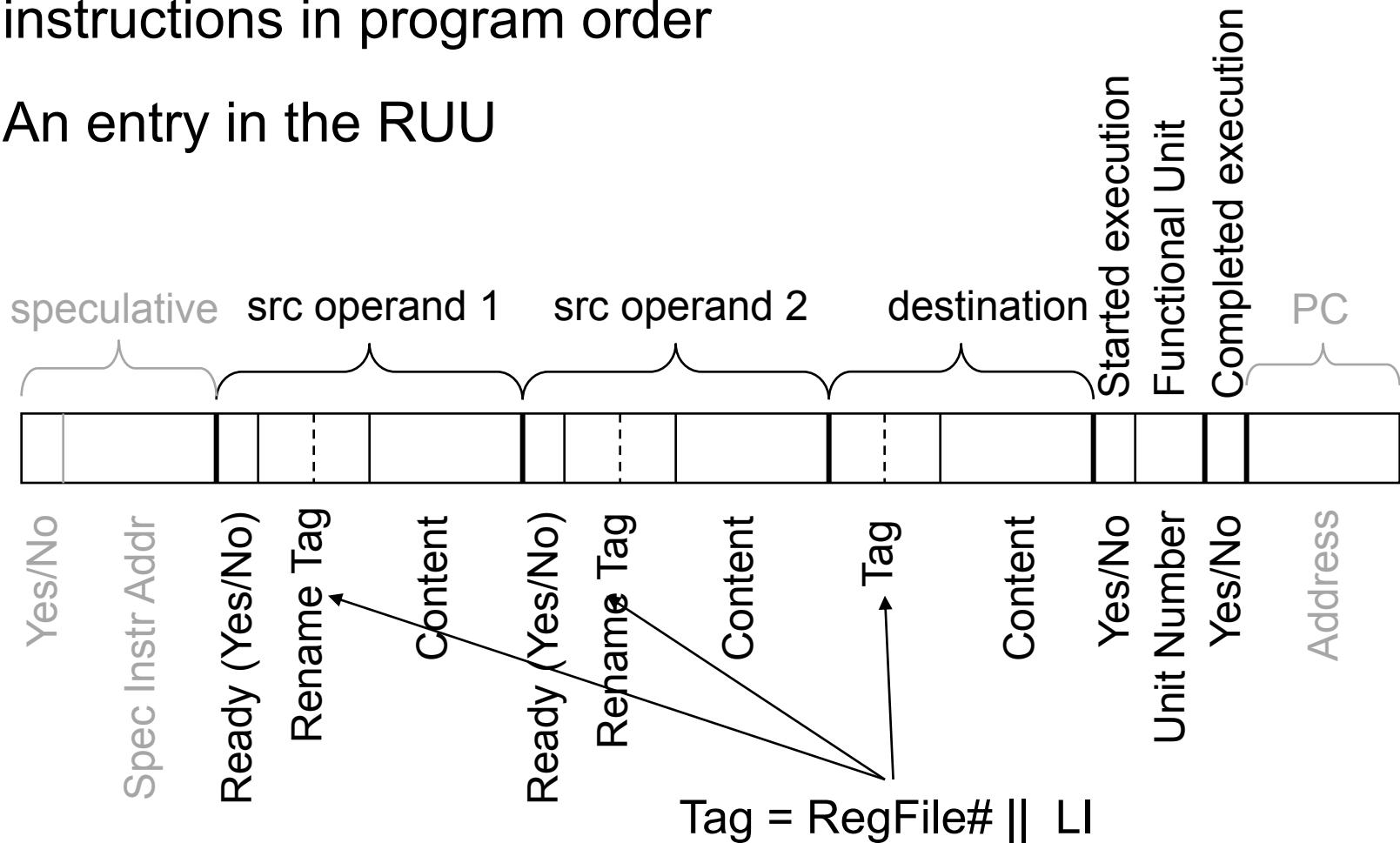
(Or done as **Reorder Buffer** together with FU Reservation Stations)

Additional RegFile Fields

- ❑ Each register in the general purpose RegFile has two associated n-bit counters (n of 3 is typical)
 - NI (**number of instances**) – the number of instances of a register as a destination register in the RUU
 - LI (**latest instance**) – the number of the latest instance
- ❑ When an instruction with destination register address R_i is dispatched to the RUU, both its NI and LI are incremented
 - Dispatch is blocked if a destination register's NI is $2^n - 1$, so only up to $2^n - 1$ instances of a register can be present in the RUU at any one time
- ❑ When an instruction is committed (updates the R_i value) the associated NI is decremented
 - When NI = 0 the register is “free” (there are no instruction in the RUU that are going to write to that register) and LI is cleared

Register Update Unit (RUU)

- ❑ A **hardware** data structure that is used to resolve data dependencies by keeping track of an instruction's data and execution needs and that commits completed instructions in program order
- ❑ An entry in the RUU



Basic Instruction Flow Overview

- ❑ Fetch (in program order): **Fetch** multiple instructions in parallel from the I\$
- ❑ Decode & Dispatch (in program order):
 - In parallel, **decode** the instr's just fetched and schedule them for execution by **dispatching** them to the RUU
 - Loads and stores are dispatched as two (micro)instr's – one to compute the effective addr and one to do the memory operation
- ❑ Execute (out of program order): As soon as the RUU has the instr's source data and the FU is free, the instr's are sent to the FU for **execution**
- ❑ Writeback (out of program order): When done, the FU puts its results on the Result Bus which allows the RUU and the LSQ to be updated – the instr completes
- ❑ Commit (in program order): When appropriate, **commit** the instr's result data to the state locations (i.e., update D\$ and RegFile)

Managing the RUU as a Queue

- ❑ By managing the RUU as a queue, and committing instruction from RUU_Head, instruction are committed (aka **retired**) in the order they were received from the Decode & Dispatch logic (in program order)
 - Stores to state locations (RegFile and D\$) are buffered (in the RUU and LSQ) until commit time
 - Supports precise interrupts (the *only* state locations updated are those written by instructions *before* the interrupting instr)
- ❑ The counter (LI) allows multiple instances of a specific destination register to exist in the RUU at the *same* time via **register renaming**
 - Solves **write before write** hazards if results from the RUU are returned to the RegFile in program order

Major Functions of the RUU

- ❑ Each of the tasks are done in parallel every cycle
1. Accepts new instructions from the Decode & Dispatch logic
 2. Monitors the Result Bus to resolve (i.e., forward) true (**read before write**) dependencies and to do write back of result data to the RUU
 3. Determines which instructions are ready for execution, reserves the Result Bus, and sends the instruction to the appropriate FU for execution
 4. Determines if an instruction can commit (i.e., change the machine state) and commits the instruction if appropriate

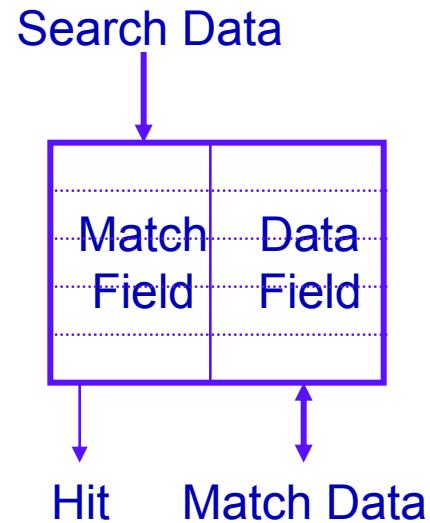
The First Function of the RUU

1. Accepts new instructions from the Decode & Dispatch logic – for instructions in the fetch packet
 - The dispatch logic acquires entries in the RUU (a circular queue)
 - The RUU_Tail entries (if there are enough open slots for all the instructions in the fetch packet) are allocated to the instructions in the fetch packet and RUU_Tail is updated
 - If there aren't enough open slots (e.g., RUU_Head = RUU_Tail) then further instruction fetch stalls until the RUU_Head advances (as a result of a commit)
 - For each source operand, if the contents of the source register is available, then it is copied to the source Content field of the RUU entry and its Ready bit is set. If not, the source register's LI is incremented, RegFile#||LI is copied to the source's Tag field, and the Ready bit is cleared.
 - For the destination operand, the RegFile#||LI is copied to the allocated RUU destination Tag field
 - The Started execution bit and Completed execution bit are set to No, the number of the FU needed for the execution is entered, and the PC address of the instruction is copied to the PC Address field

Aside: Content Addressable Memories (CAMs)

- ❑ Memories that are addressed by their **content**. Typical applications include RUU source tag field comparison logic, cache tags, and translation lookaside buffers

- Memory hardware that compares the Search Data to the Match Field entries for *each word* in the CAM in *parallel* !
- On a match the Data Field for that entry is output to Match Data on read or Match Data is written into the Data Field on write and the Hit bit is asserted.
- If no match occurs, the Hit bit is asserted.
- CAMs can be designed to accommodate multiple hits.



The Second Function of the RUU

2. Monitors the Result Bus to resolve true dependencies and to do write back of result data to the RUU
 - The Result Bus's RegFile#||LI is compared **associatively** to the source Tag fields (for those source operands that are not Ready). If a match occurs, the data on the Result Bus is loaded into the Content field for matching source operands.
 - The Result Bus contains the result data, the destination's RegFile#||LI, and the RUU entry address for that instruction
 - For the RUU entry that matches the RUU entry address of the instruction on the Result bus, loads the result data into the destination Content field and set the Completed execution bit
- ❑ Resolves true dependencies (**read before write**) through the Ready bits (i.e., must wait for the source operands to be Ready before sending the instruction for execution)
- ❑ Solves anti-dependencies (**write before read**) through LI (making sure that the source fields get updated only for the **correct** version of the data)

The Third Function of the RUU

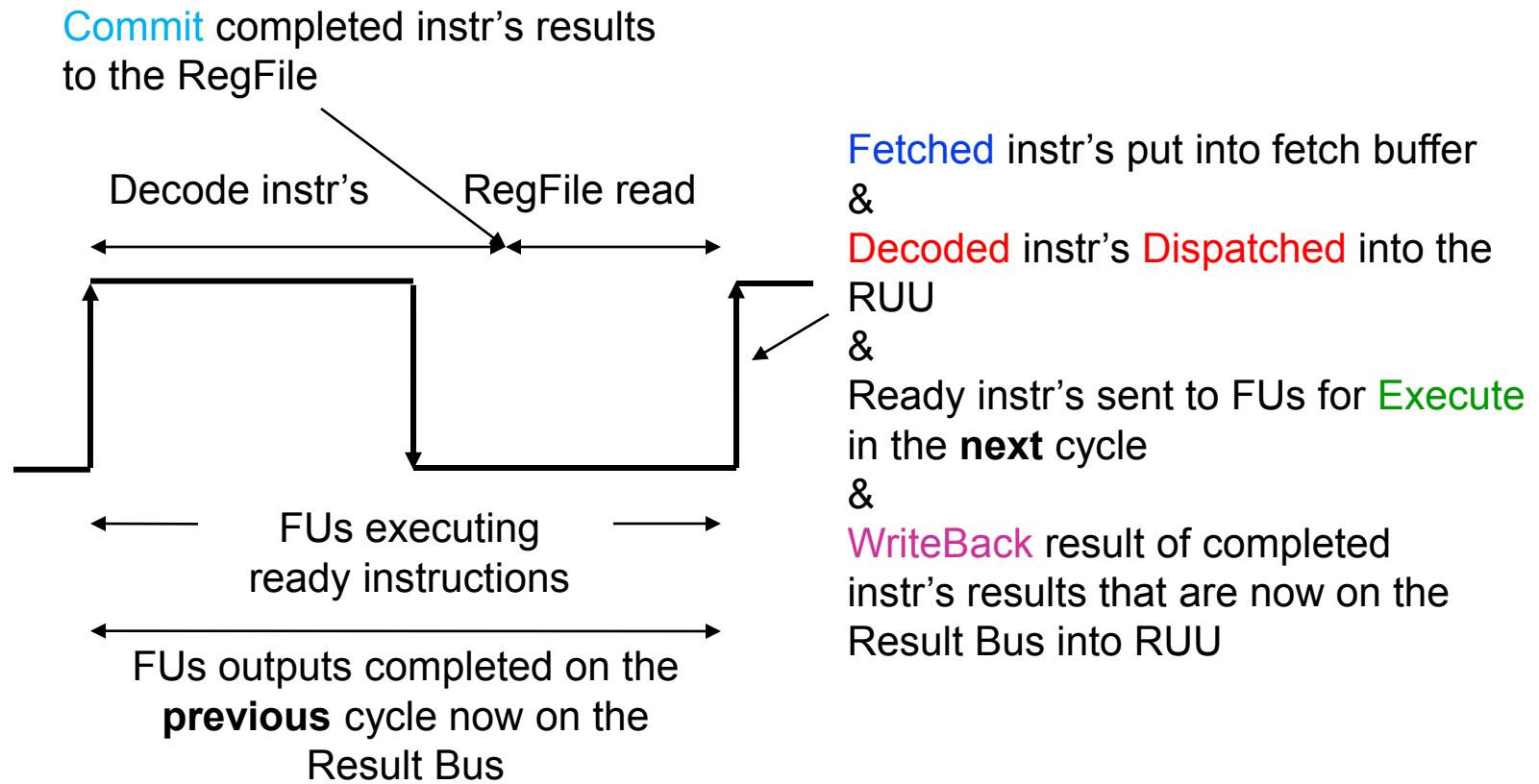
3. Determines which instructions are ready for execution, reserves the Dispatch Bus, and sends the ready instructions to the appropriate FU's for execution
 - When both source operands of an RUU entry are Ready and the FU is free, the RUU sends the **highest priority** instruction to the FU for execution (sends to the FU the source operands, the RUU entry address, and the destination's RegFile#||LI)
 - priority is given to load and store instr's and then to the instructions that entered the RUU the earliest (i.e., the ones closest to RUU_Head).
 - Reserves the Dispatch Bus
 - Sets the Started execution bit to Yes
- Multiple instructions can be issued in parallel, if they are ready, if they can reserve the Dispatch Bus, and if they are destined for different FU's

The Fourth Function of the RUU

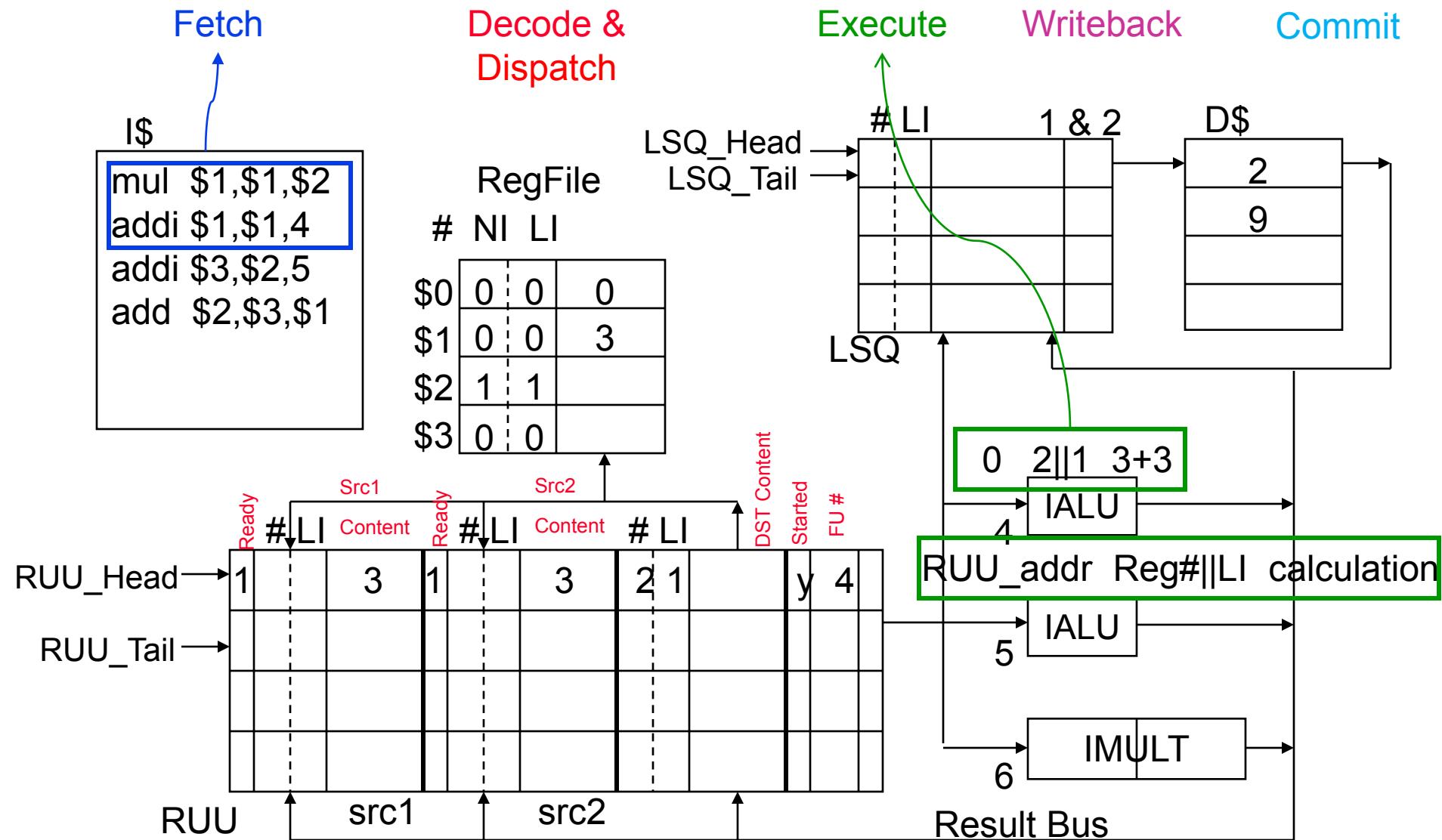
4. Determines if instructions at the head of the RUU can commit (i.e., change the machine state) and commits them in order
 - Monitors the Completed execution bit of the RUU_Head entry. If the bit is set and the instruction is not speculative, then the destination content data is written to the RegFile and the associated RegFile entry's NI counter is decremented (and LI is cleared if NI becomes zero).
 - Associatively matches the destination's RegFile#||LI against the RUU's source Tag fields and on match copies (i.e., forwards) the destination content data into source Content fields
 - Releases the RUU entry by incrementing the RUU_Head pointer
- ❑ Solves output dependencies (**write before write**) by writing to RegFile in program order
- ❑ Multiple instructions can commit in parallel if they are ready to commit, if they are writing to different RegFile registers, and if there are multiple RegFile write ports

Timing of RUU Activities

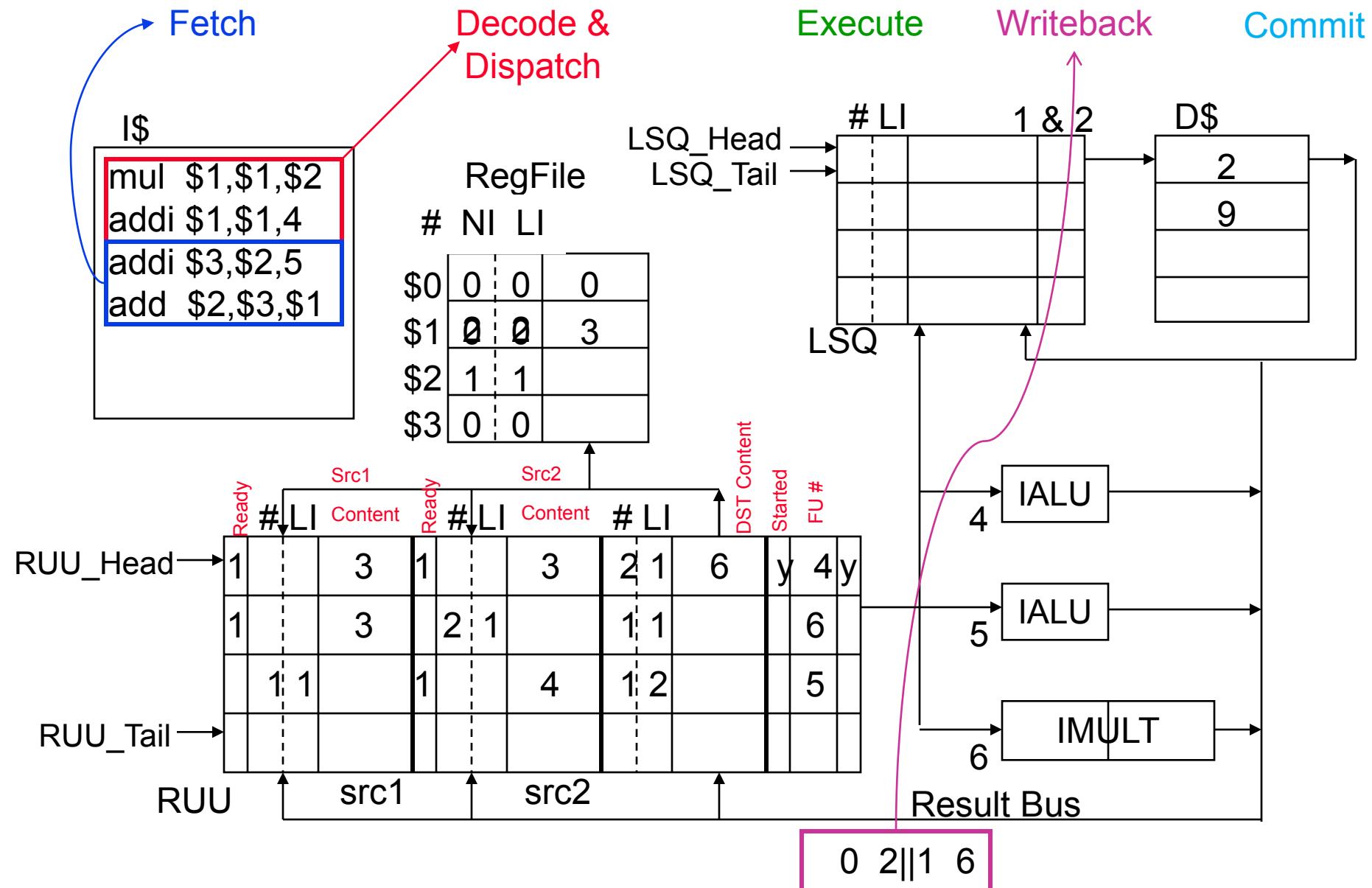
- The SS machine also does register write (i.e., Commit from the RUU) **first** then **read** (as part of Decode & Dispatch), same as word fall through in MIPS.



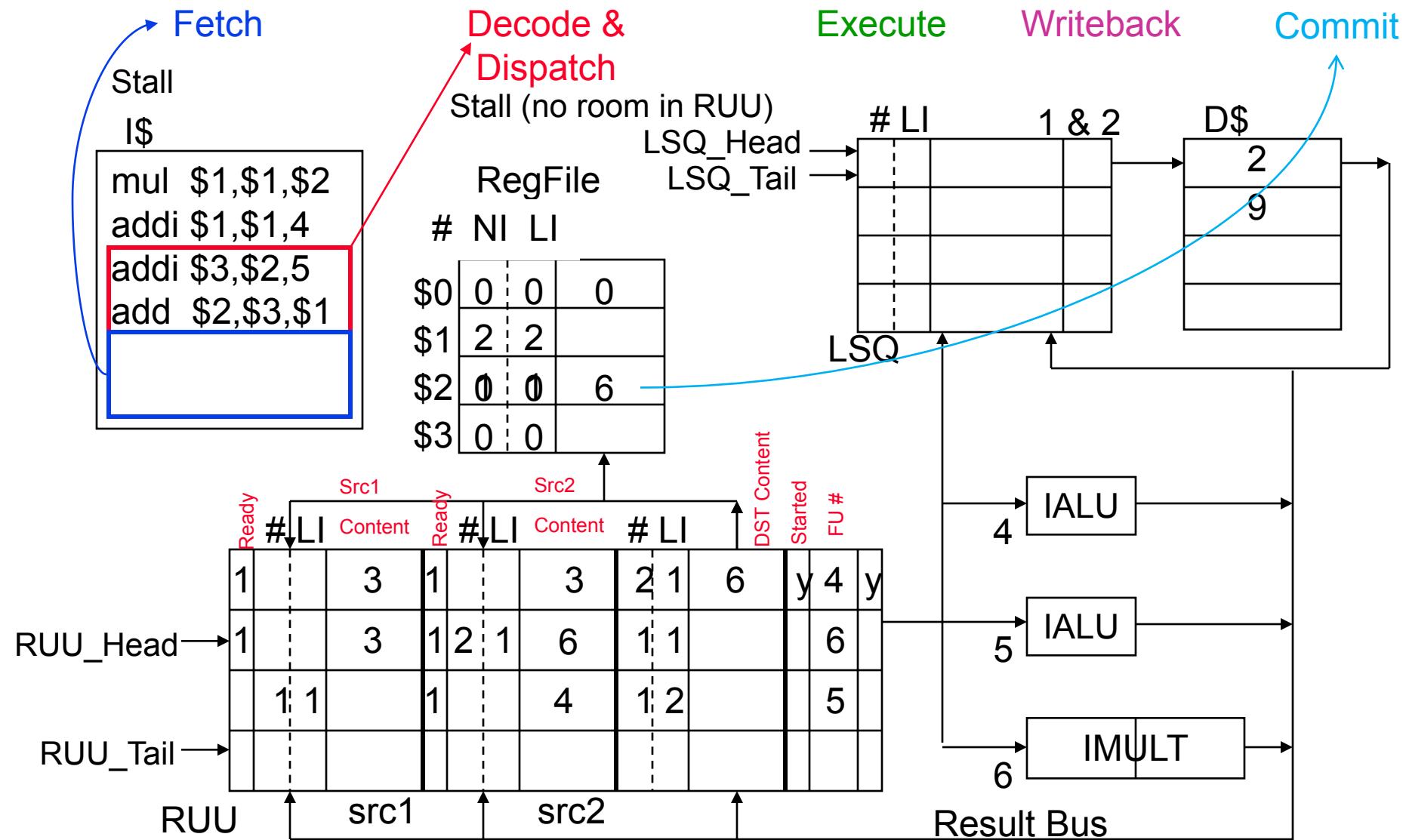
SS MIPS – Cycle 0



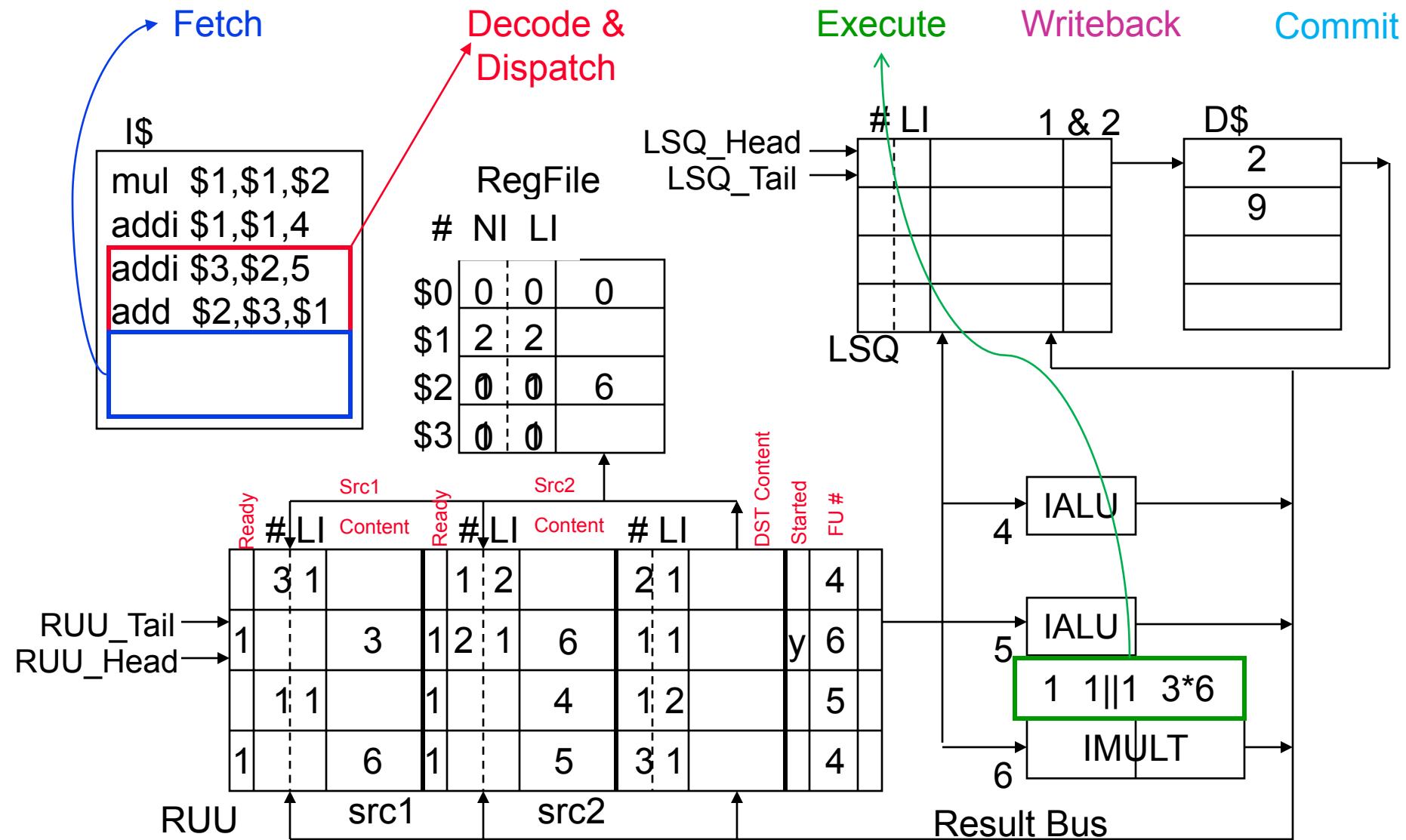
SS MIPS – Cycle 1



SS MIPS – Cycle 2



SS MIPS – Cycle 3



Summary

- ❑ The RUU and control logic associated with it controls determines the dispatch of instructions to functional units.
- ❑ Register numbers are “renamed” through the use of a latest instance number.
- ❑ Logic like CAM can be used to write values to the RUU according to the register and latest instance number.
- ❑ Once the RUU entry reaches the head it can be committed and written back to the register file. This decrements the NI and once NI reaches zero LI is reset as there are no instances left.

End of Lecture

- Assignment 2 will be available this afternoon.
 - There is a relatively short amount of time before it is due, start it ASAP.

ECE4074

Computer Architecture

Semester 2 2012

A “Simple” SS Processor cont’

[Adapted from Sohi, Instruction Issue Logic for High-Performance, Interruptable, Multiple Functional Unit, Pipelined Computers, *IEEE Transactions on Computers*, Vol 39, No 3, 1990.]

Review

- ❑ To achieve high performance, need both **machine parallelism** and **instruction level parallelism (ILP)** by
 - Superpipelining
 - Static multiple-issue (VLIW)
 - Dynamic multiple-issue (superscalar (SS))
- ❑ A SS processor's instruction issue and execution policies impact the available ILP
 - In-order fetch, issue, and commit and out-of-order execution
 - Pipelining creates **true** dependencies ([read before write](#))
 - Out-of-order execution creates **antidependencies** ([write before read](#)) and **output dependencies** ([write before write](#))
 - In-order commit allows speculation (to increase ILP) and is required to implement precise interrupts
- ❑ Register renaming can solve these storage dependencies

MicroOperations of Load and Store

- ❑ Recall that loads and stores are dispatched to the RUU as two (micro)instr's – one to compute the effective addr and one to do the memory operation

| | | | | |
|---------|----------------|---------|------|-------------|
| ● Load | lw \$1, 2(\$2) | becomes | addi | \$0, \$2, 2 |
| | | | lw | \$1, \$0 |
| ● Store | sw \$1, 6(\$2) | becomes | addi | \$0, \$2, 6 |
| | | | sw | \$1, \$0 |

- ❑ At the same time a Load/Store Queue (LSQ) entry is allocated

- Each LSQ entry consists of a Tag field (RegFile#||LI) and a Content field. The LI counter allows for multiple instances of stores (writes) to the same memory address
- When a load completes (the D\$ returns the data on the Result Bus) or a store commits (in program order) the LSQ entry is released
- Instruction dispatch is blocked if there is not a free LSQ entry and two free RUU entries

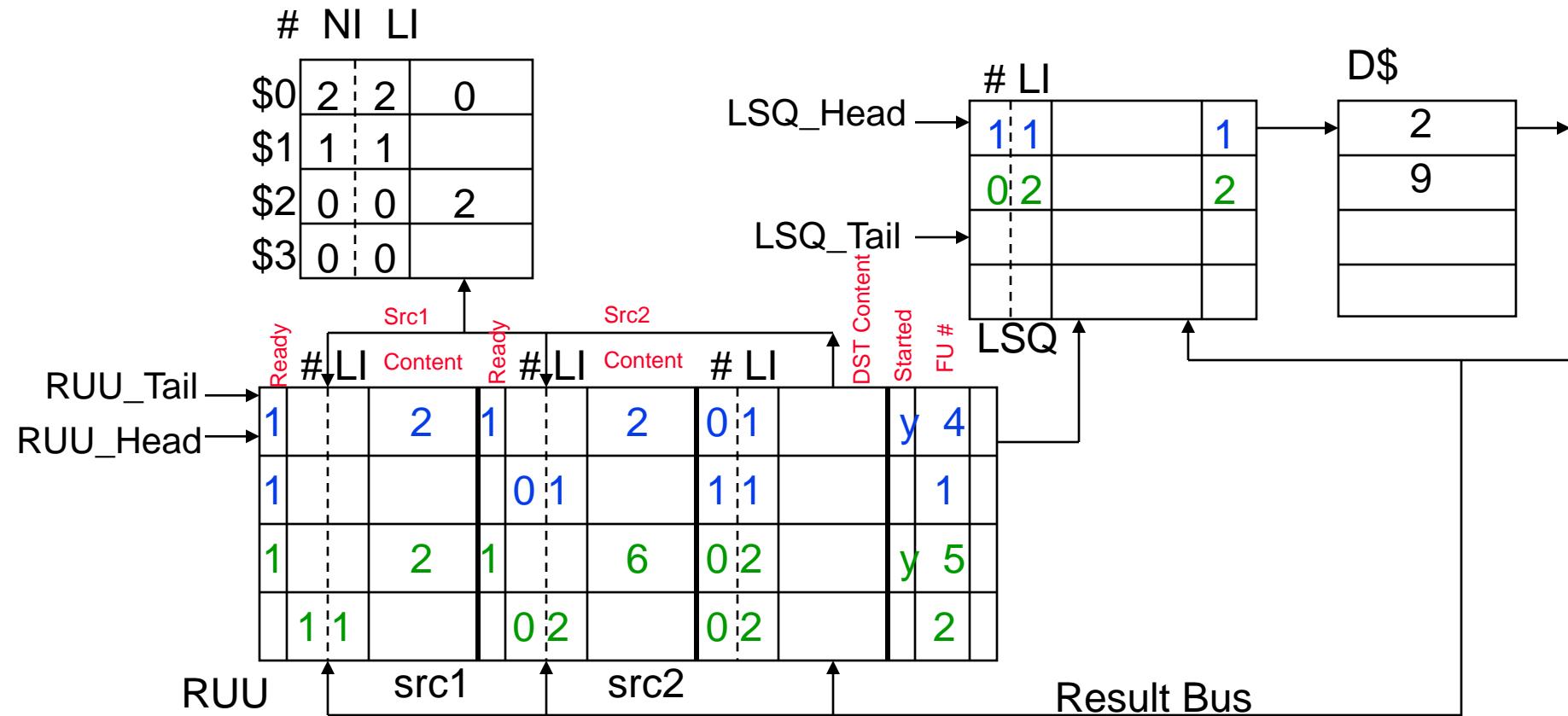
Load & Store RUU & LSQ Operation

Load

```
addi $0,$2,2
lw    $1,$0
```

Store

```
addi $0,$2,6
sw    $1,$0
```



Memory Location Data Dependencies

- ❑ We can have true, anti- and output dependencies on memory locations as well
 - Memory storage conflicts are infrequent since memory locations are not reused in the same way that registers are and since the number of true dependencies is small (loads and stores are less than 30% of the instruction mix)
- ❑ Can improve performance if loads are allowed to bypass previous stores (**load bypassing**) – but can do so only if the memory addresses of *all* previous stores dispatched to the RUU are known since must first check for load data dependency on previous uncommitted stores
- ❑ Stores are not allowed to bypass previous loads, so there are no antidependencies between loads and stores
- ❑ Stores are committed to the D\$ in program order, so there can be no output dependencies between stores

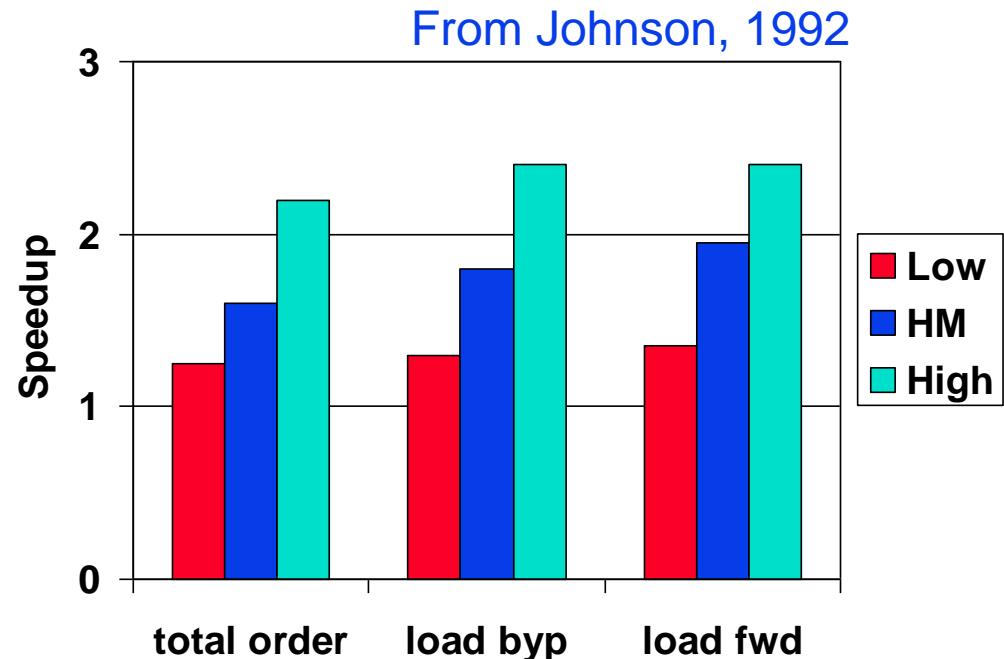
Loads from Memory

- ❑ When a load's address becomes known, the address is compared (associatively) to see if it matches an entry already in the LSQ (i.e., if there is a pending operation to the same memory address)
 - If the match in the LSQ is for a **load**, the current load is not executed since the matching pending load will load in the data
 - If the match in the LSQ is for a **store**, the current load does not need to be executed since the matching pending store can directly supply the destination Content for the current load
- ❑ If there is no match, the load is issued to the LSQ and executed when the D\$ is next available
- ❑ When the RUU# of the load instruction appears on the Result Bus (along with the memory data) the load results are written back to the RUU and the LSQ entry is released

Load Bypassing with Forwarding

- ❑ Load forwarding – when a load is satisfied directly from the LSQ
 - The most recent matching LSQ data entry is supplied, since the LSQ may have more than one matching entry

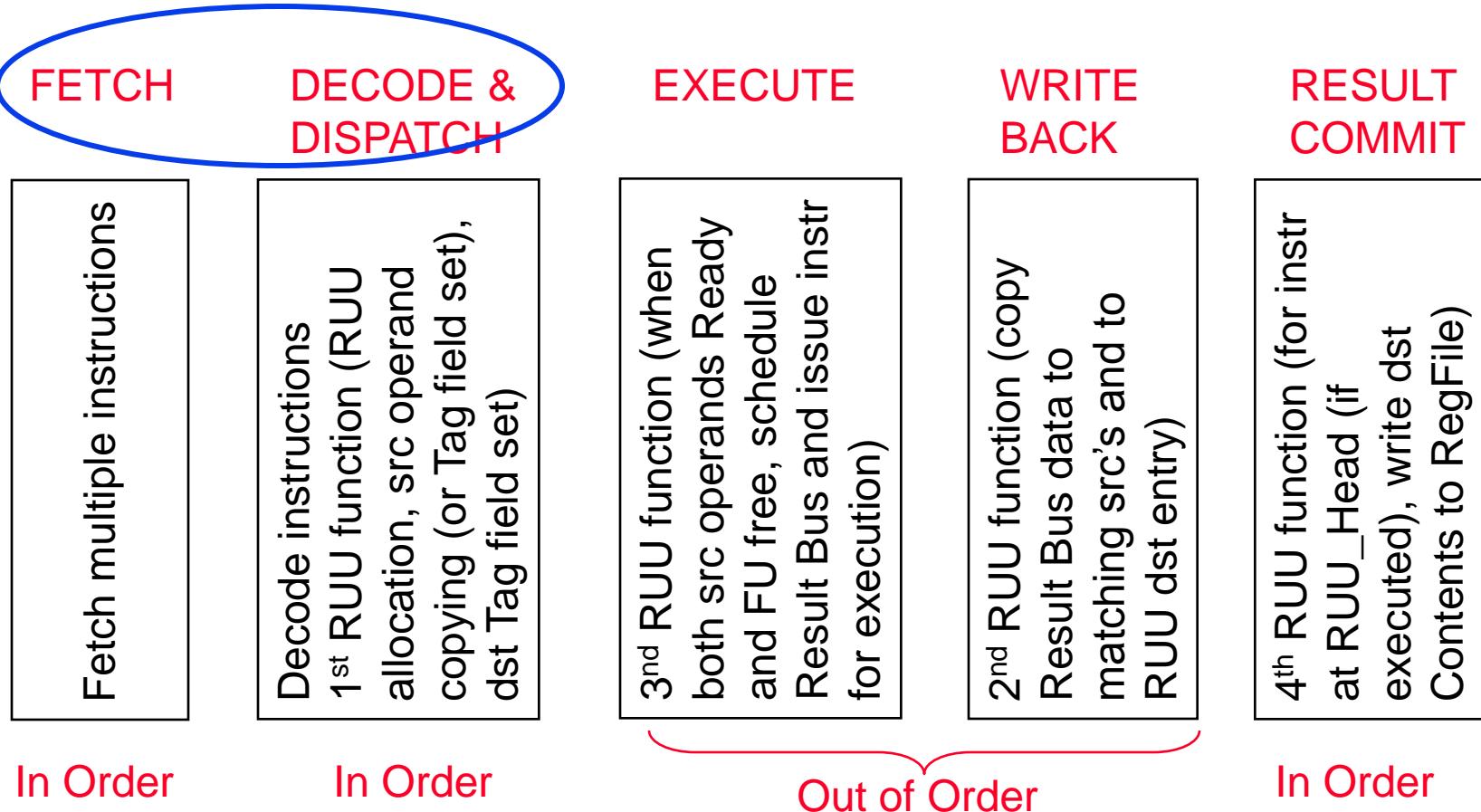
- ❑ Load bypassing gives 19% speedup improvement with a 4-way instr decoder
- ❑ Load forwarding gives an additional 4% speedup improvement



Stores to Memory

- ❑ When a store's address (and the store data) becomes known, the address is compared (associatively) to see if it matches an entry already in the LSQ (i.e., if there is a pending operation to the same memory address)
 - If the match in the LSQ is for a **load**, the current store is issued to the LSQ
 - If the match in the LSQ is for a **store**, the current store is issued to the LSQ with an incremented LI
 - If there is no match, the store is dispatched to the LSQ
- ❑ Stores are held in the LSQ until the store is ready to commit (i.e., until its partner instr reaches the RUU_Head) at which time the store is executed (i.e., the data and address are sent to the D\$) and the RUU and LSQ entries are released

SS Pipeline – Front End Considerations

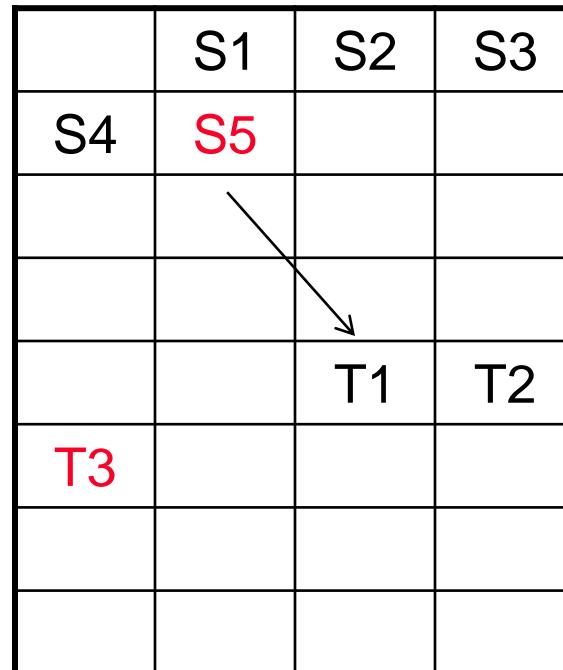


Instruction Fetch Sequences

- ❑ Instruction run – number of instructions (run length) fetched between **taken** branches
 - Instruction fetcher operates most efficiently when processing long runs – unfortunately runs are usually quite short (about six instructions)

- ❑ For a 4-way fetcher, instruction bandwidth of only 2 instructions per cycle (with branch prediction support)
 - 8 instructions in 4 cycles

| | | | |
|----|----|----|----|
| | S1 | S2 | S3 |
| S4 | S5 | | |
| | | | |
| | | | |
| | | | |
| | | T1 | T2 |
| | T3 | | |
| | | | |
| | | | |

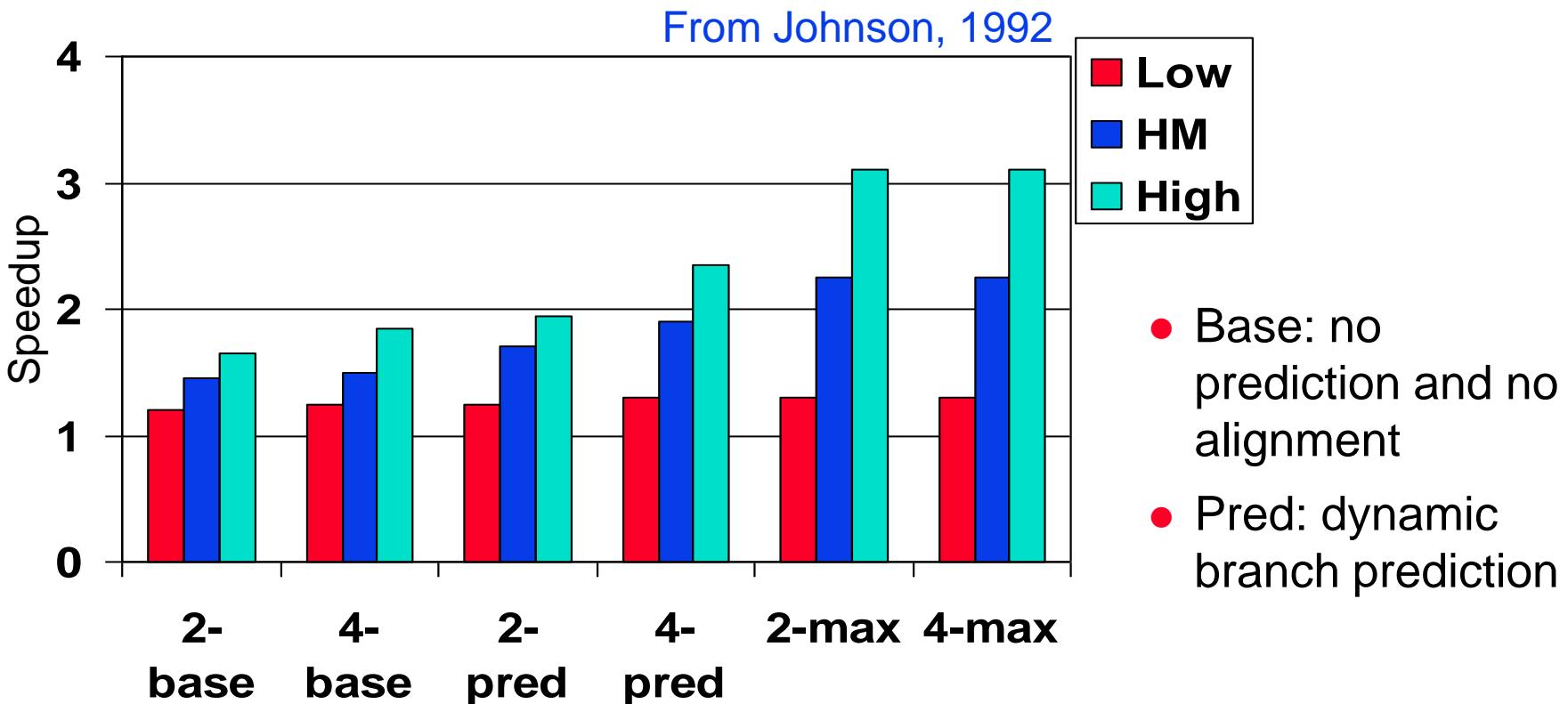


The diagram illustrates a 4-way instruction fetcher's internal state over 8 cycles. The columns represent S1, S2, and S3 stages. The rows represent instruction slots. The sequence starts with S4 in slot 1, followed by S5 in slot 1. A branch occurs at the end of S5, leading to either T1 or T2. The final instruction T3 is fetched in slot 1. The other slots remain empty.

Instruction Fetch Inefficiencies

- ❑ Fetcher can't provide adequate bandwidth to the decoder to exploit the available ILP because
 - Instruction fetch misalignment prevents the decoder from operating at full capacity
 - The fetcher can **align** fetched instructions to avoid wasted decoder slots
 - If supported by dynamic branch prediction, the fetcher can also **merge** instructions from different runs
- ❑ Aligning and merging can only be done if the fetcher has the sufficient bandwidth (i.e., the fetch rate is faster than the decode rate)

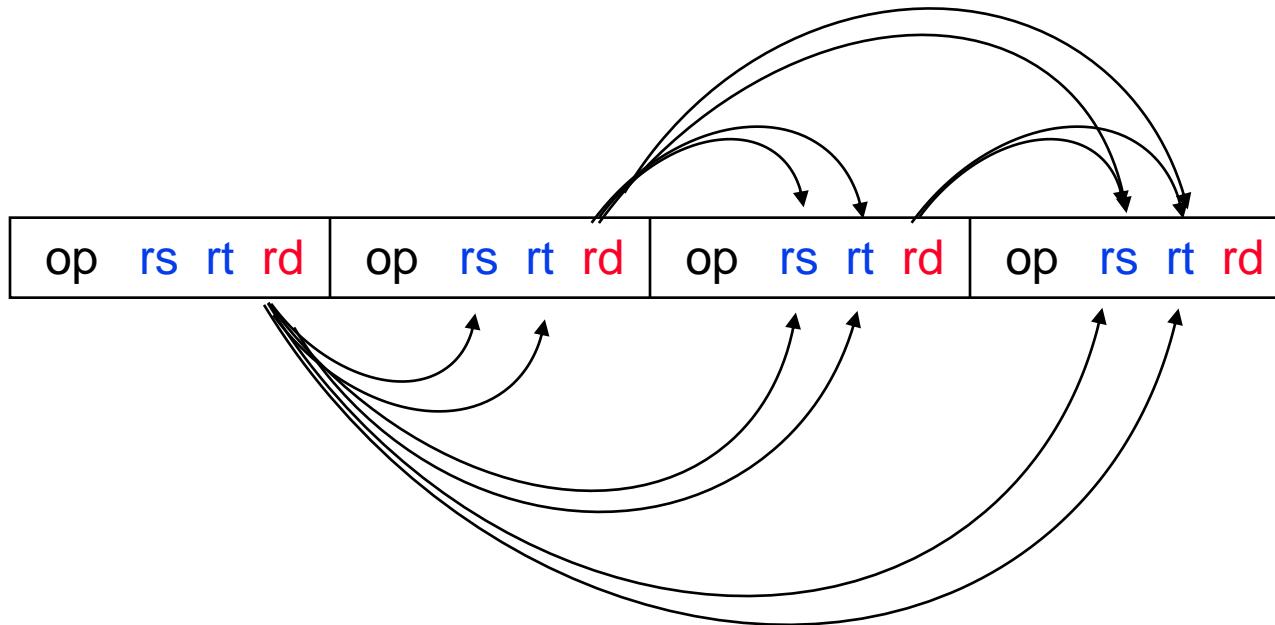
Speedups of Fetch Alternatives



- ❑ A 4-way instr fetcher out performs a 2-way instr fetcher
 - It has twice the potential instruction bandwidth
 - But it requires twice as much decoder hardware to keep up (e.g., in decoders and in ports and buses)

4-Way Decoder Implementation

- ❑ A 4-way instr fetcher has higher fetch bandwidth but at what cost ?
 - 12 dependency checks between the 4 decode instruction

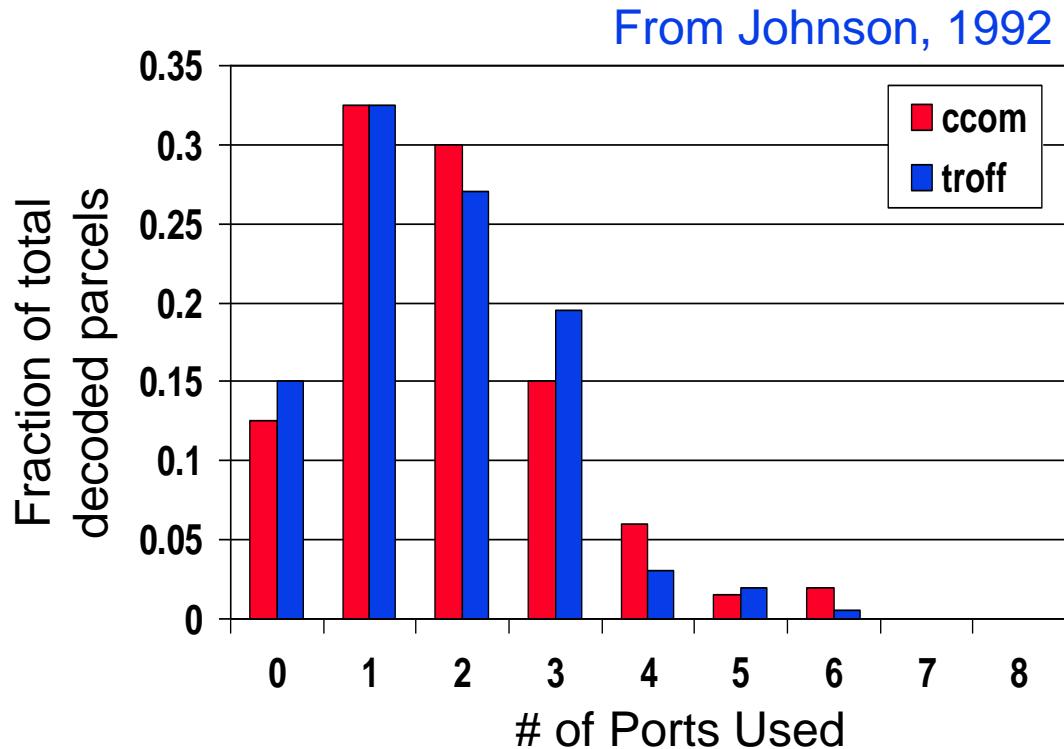


- 8 read ports on the RegFile and 8 write ports to the RUU and 8 buses to distribute those **source operands** (or their RegFile#||LI)

Reducing 4-Way Decoder Hardware

- ❑ Limiting the number of RegFile read ports, buses and RUU write ports is acceptable since
 - Not all decoded instructions access two registers
 - Not all decoded instruction are valid (because of misalignment)
 - Some decoded instructions have dependences on one or more simultaneously decoded instructions

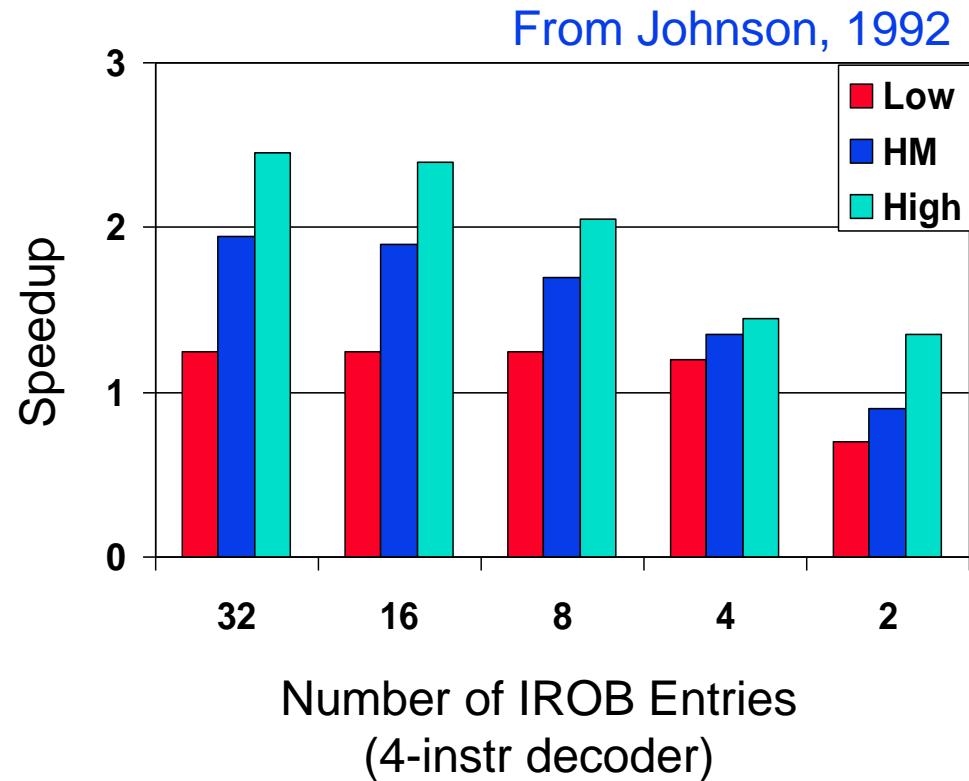
- ❑ The register demand means a 8 port capacity would be wasted
- ❑ 4 ports reduces average performance < 2%



Effects of RUU Size on Performance

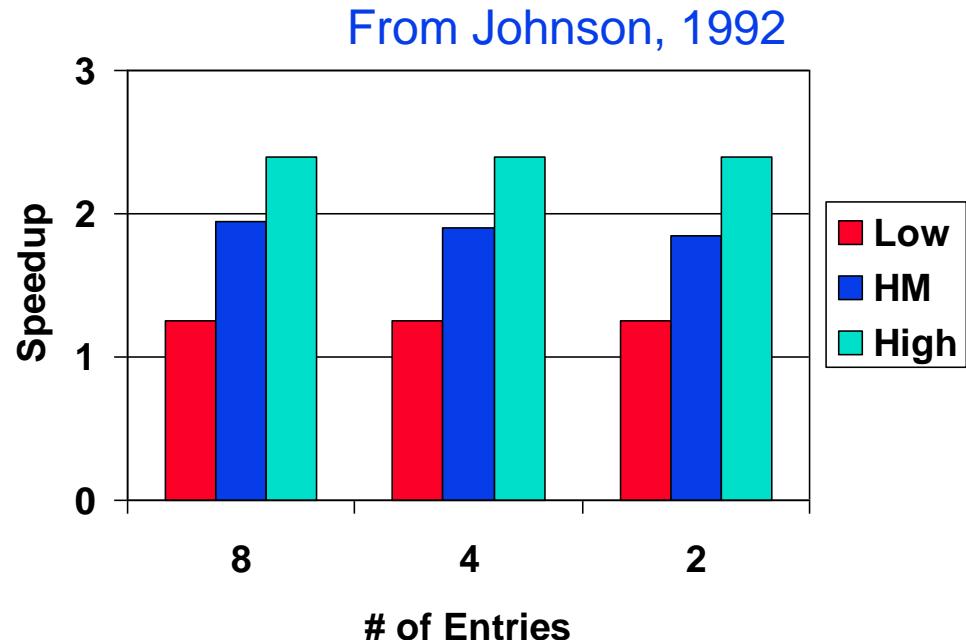
- Since instruction decoding must stall when there is no free RUU entry, the RUU should be large enough to accept *all* instructions during the expected dispatch-to-commit time period

- Performance decreases markedly with 8 and 4 entries
- For 2 entries the performance is worse than the scalar processor
 - Why?



Effects of LSQ Size on Performance

- Since instruction decoding must stall when there is no free LSQ entry, the LSQ should be large enough but the LSQ size has relatively little impact on performance
 - With a 4-way decoder, a 4-entry LSQ only incurs a 1% speedup loss over an 8-entry LSQ
 - Smaller LSQ facilitate dependency checking

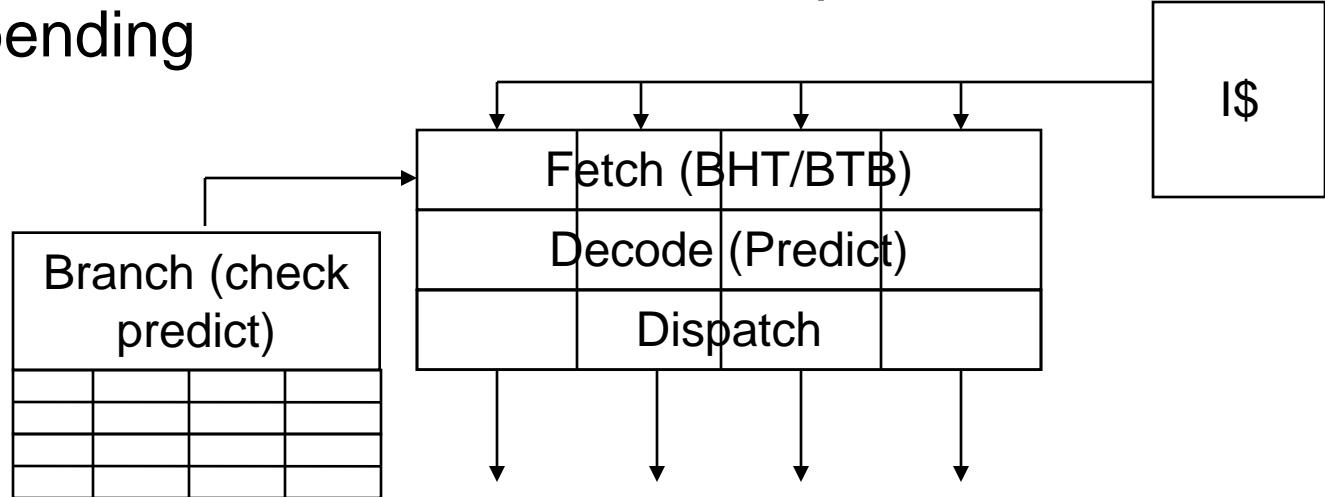


SS Branch Prediction

- ❑ Recall from earlier lectures that for branch prediction in a scalar pipeline we needed
 - A mechanism to predict the branch outcome: a BHT (branch history table) in the fetch stage
 - A way to fetch two instructions – the sequential instruction (I\$) *and* the branch target instruction (BTB (branch target buffer))
 - A way to ensure that instructions active in the pipeline following the branch didn't change the machine state until the branch outcome was known
 - Allowed to complete (in order commit) on correct prediction
 - Flushed on mispredict and restart
- ❑ With a SS machine, it is possible to have many such instructions after predicted branches *active* in the pipeline
 - Flag instructions following branches as **speculative** in their RUU entry until the branch outcome is known

Implementing Branches

- ❑ A SS processor could have more than one branch per fetch set and could have several uncompleted branches pending at any time

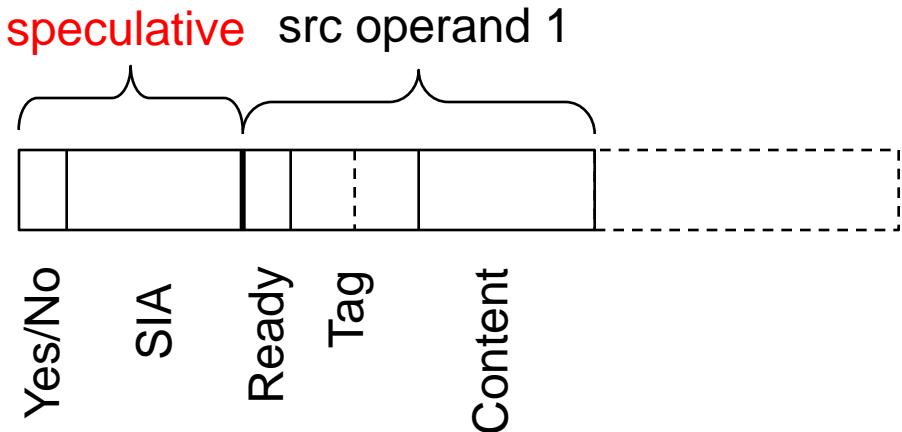


- ❑ Must access BHT/BTB for all branch instr's in the fetch set during fetch to reduced branch delay (i.e., need a 4 read-port BHT/BTB for 4-instr fetcher)
 - Pass BHT information to decode stage
 - After decode, choose between I\$ set and BTB sets to determine the next fetch set

RUU Speculation Field Support

- For dependent speculative instr's, i.e., **branches**, the speculative flag is set to Yes until the outcome of the driving instruction is determined. Then an associate comparison of that branch's PC address and the RUU's SIA (Speculative Instr Addr) fields can be done.

- If the branch was **not** mispredicted, then the branch and its trailing instructions can commit when at RUU_Head



- If the branch **was** mispredicted, then **all** subsequent instr's must be discarded (even though subsequent *branches* may have been correctly predicted). All of the RUU entries are discarded in a *single cycle* and instruction stream fetching restarts on the next cycle. (This also comes in handy when dealing with exceptions.)

SS Pipeline – Back End Considerations

FETCH

Fetch instructions, Align and merge, Access BHT/BTB for PC, PC+4, PC+8, ...

DECODE & DISPATCH

Decode instructions
1st RUU function
If branch, use BHT info to determine next PC value and whether the I\$ or BTB instr's are the next fetch set

EXECUTE

3rd RUU function (when both src operands Ready and FU free, schedule Result Bus and issue instr for execution)

WRITE BACK

2nd RUU function (copy Result Bus data to matching src's and to RUU dst entry)

RESULT COMMIT

4th RUU function (for instr at RUU_Head (if executed), write dst Contents to RegFile)

In Order

In Order

Out of Order

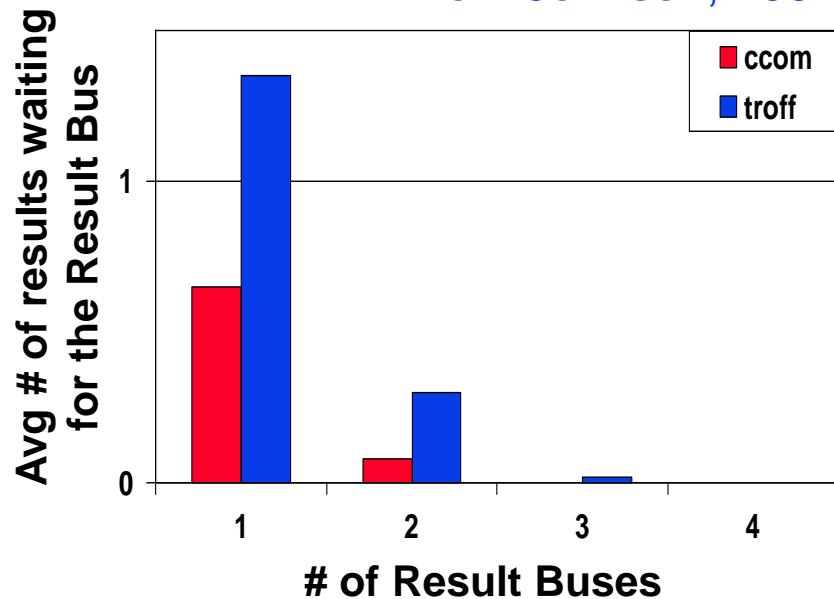
In Order

Result Buses Utilization

- ❑ Our SS model has only one Result Bus to carry results generated by the FU's to the RUU and LSQ
 - Utilization of the Result Bus is less than ~70% (i.e., the fraction of capacity actually used)
- ❑ If a FU requests for the Result Bus is not granted, instruction issue to that FU is stalled until the bus request can be granted (i.e., the FU remains “busy”)

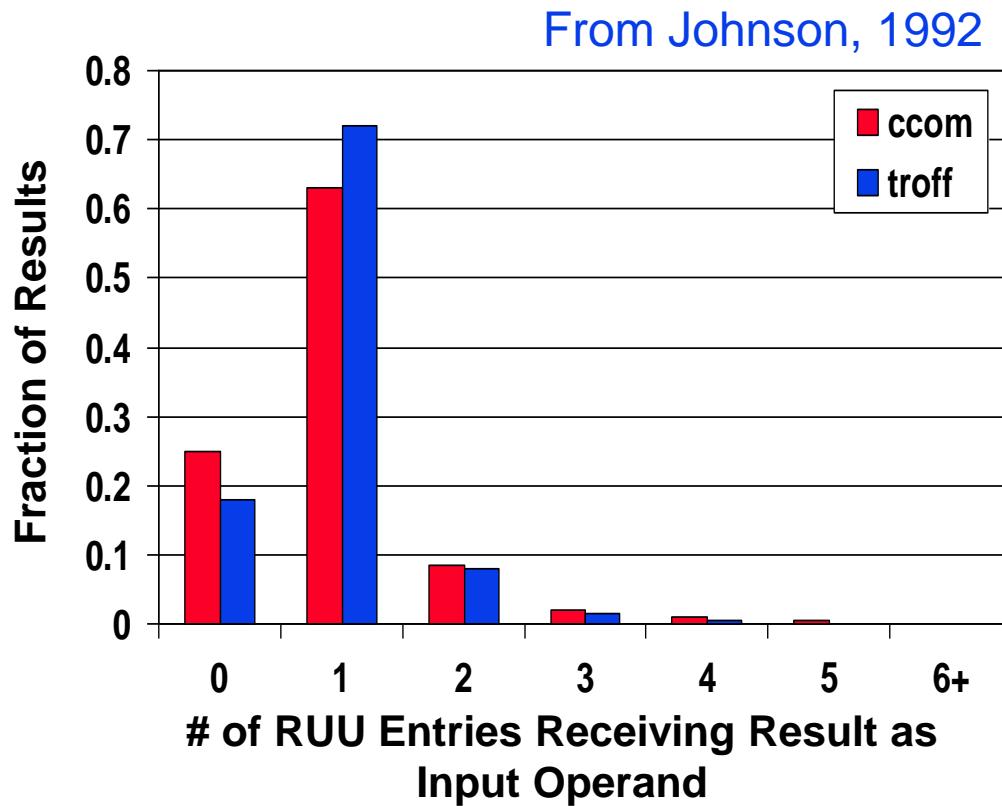
From Johnson, 1992

- ❑ Reducing the impact of bus contention by adding a second Result Bus improves performance (by almost 19%), adding a third adds little (~3%)



Result Forwarding

- ❑ Result forwarding supplies operands directly to the waiting instr's in the RUU to resolve true dependencies that could not be resolved during decode
 - The cost of forwarding is the comparison logic in the RUU to compare the Result Bus Tag to the source operand Tags
 - Need a set of comparators for each Result Bus
- ❑ About 2/3rd of all results are forwarded to one waiting operand, and about 1/6th are forwarded to more than one



Performance Advantages

- ❑ Hardware complexity arises from four major hardware features
 - Out-of-order execution
 - Register renaming
 - Branch prediction
 - 4-way instruction fetch and decode

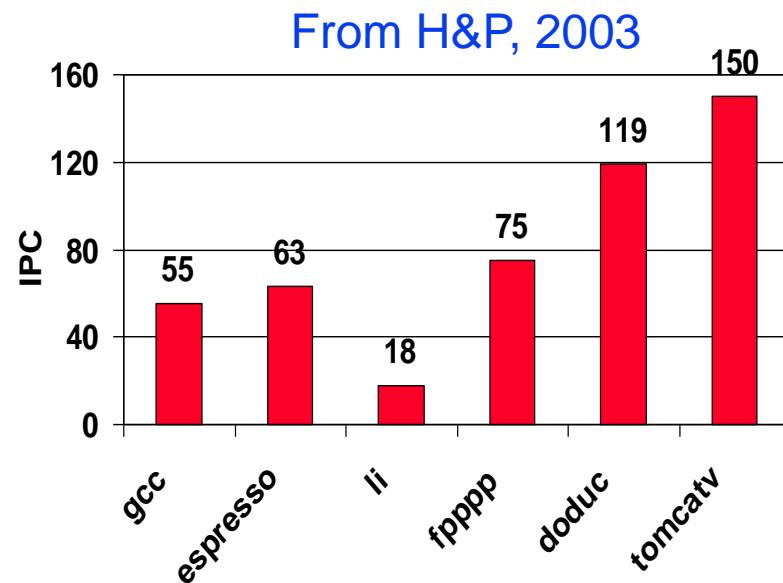
From Johnson, 1992

| OOE | Register Renaming | Branch Prediction | 4-way Fetch & Decode |
|------------|--------------------------|--------------------------|---------------------------------|
| 52% | 36% | 30% | 18% |

ILP in a Perfect SS Processor

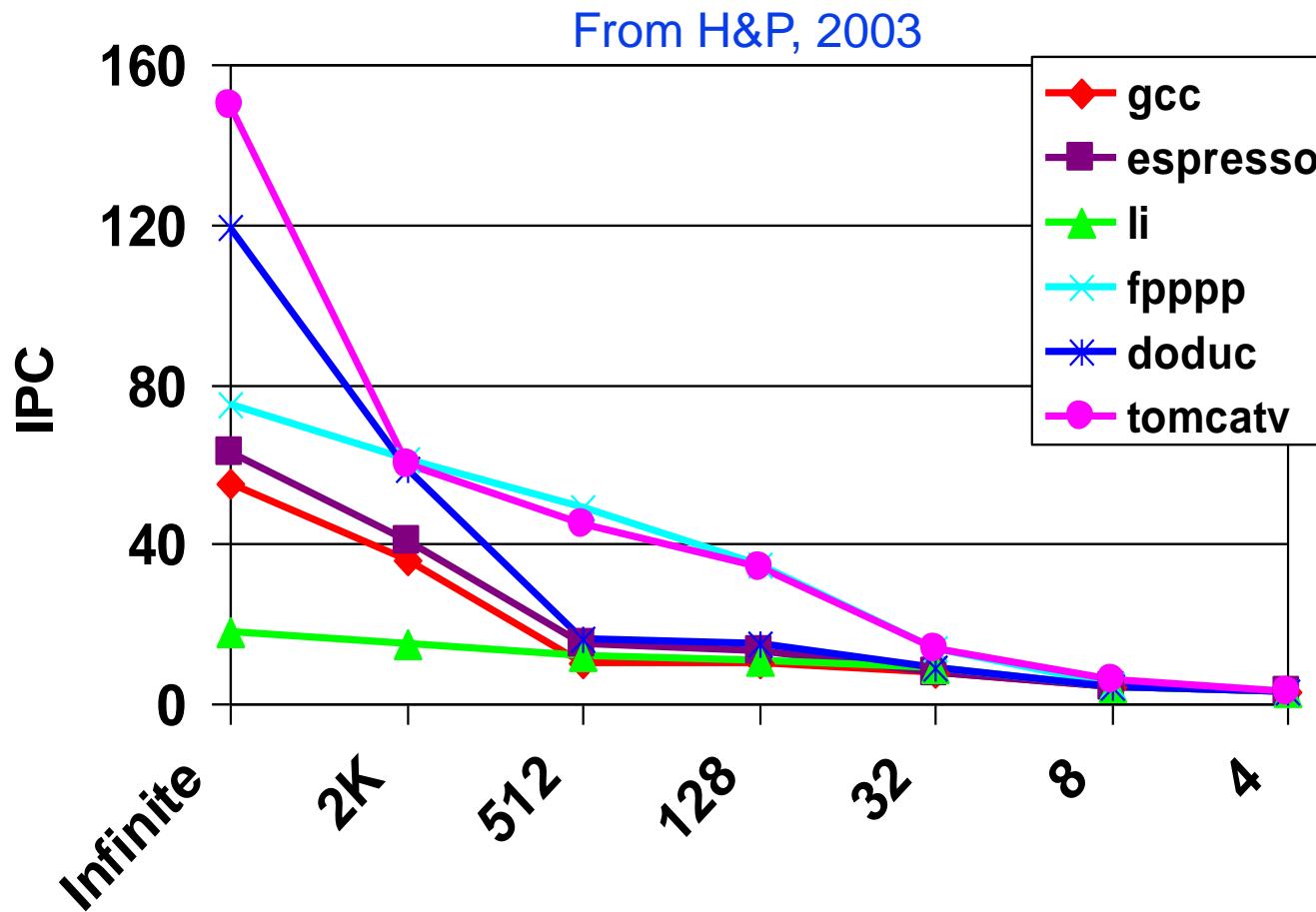
❑ The perfect processor has

- An infinite number of rename registers that eliminates all storage hazards (i.e., write before write and write before read)
- No (fetch, decode, dispatch, issue, FU, buses, ports) limit on the number of instr's that can begin execution simultaneously as long as read-before-write true data hazards are not present
- Perfect branch and jump (including jump register) prediction
- Loads can be moved before stores (as long as the addresses are not identical) with memory address analysis
- All FU's have a 1 cycle latency
- Perfect caches with 1 cycle latency



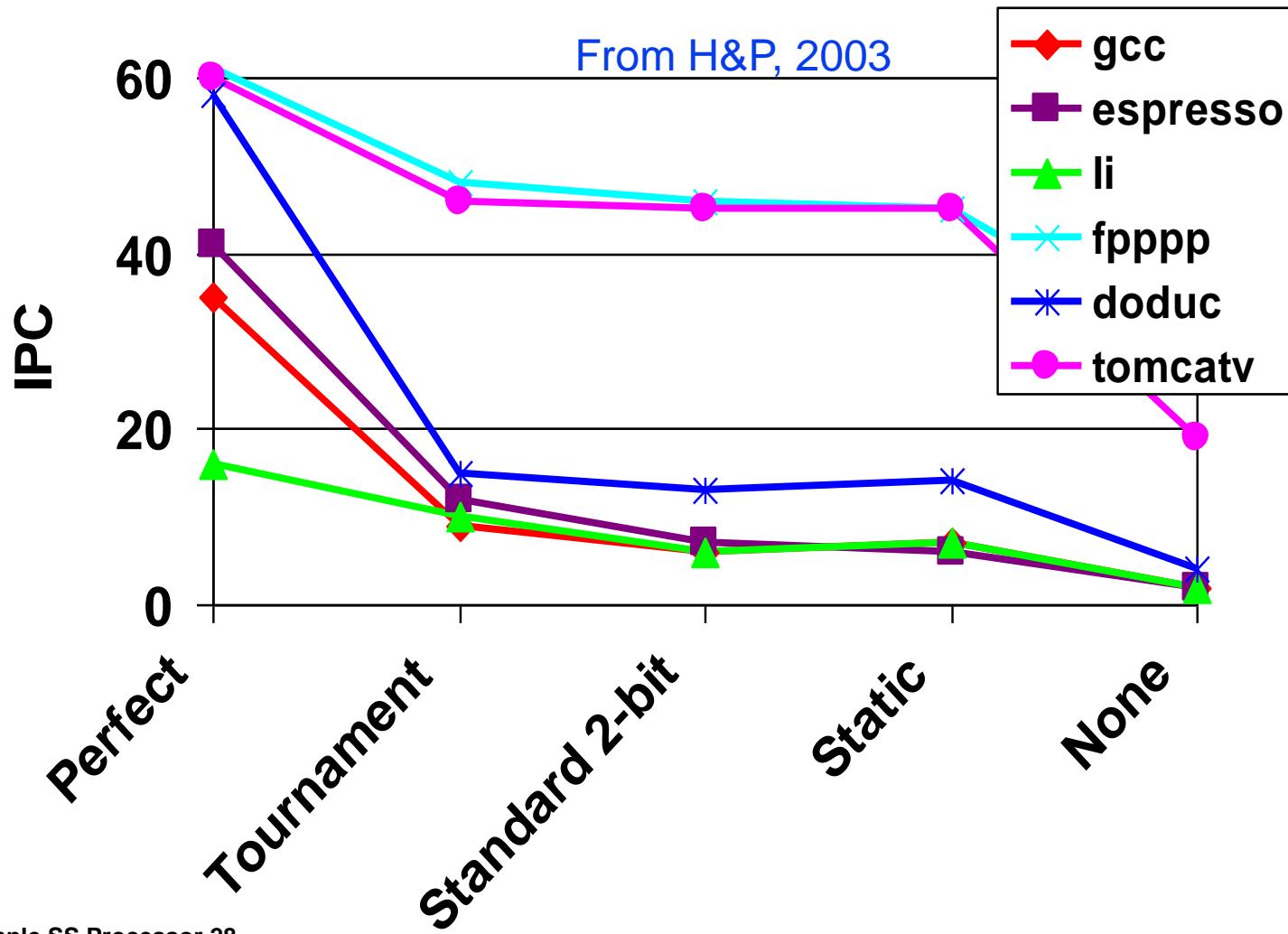
Effect of Instruction Window Size on ILP

- Instruction window – the set of instructions that are examined simultaneously for execution



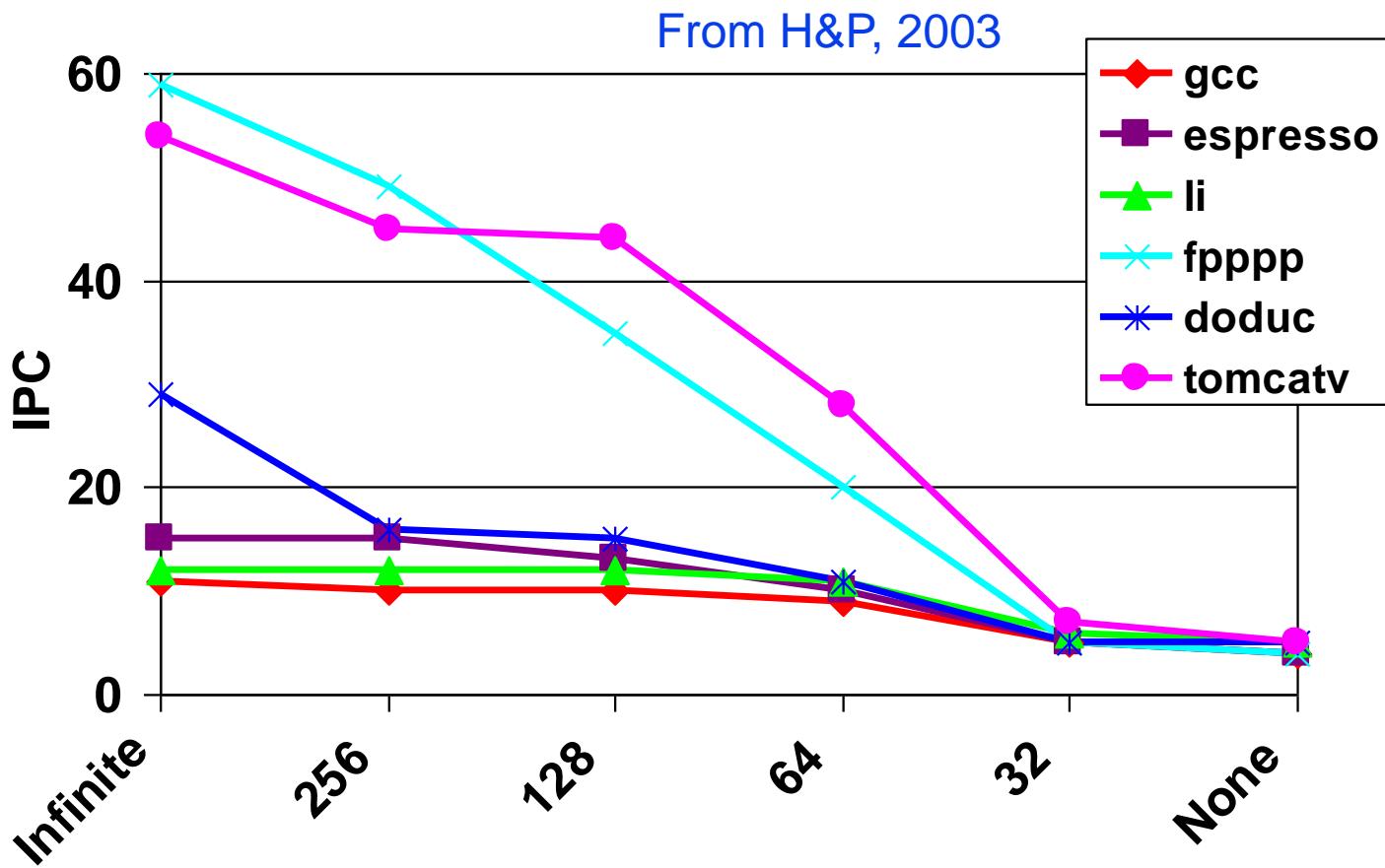
Effect of Realistic Branch Prediction on ILP

- On a processor with an instruction window size of 2K and maximum 64-way issue capability



Effect of Finite Rename Registers

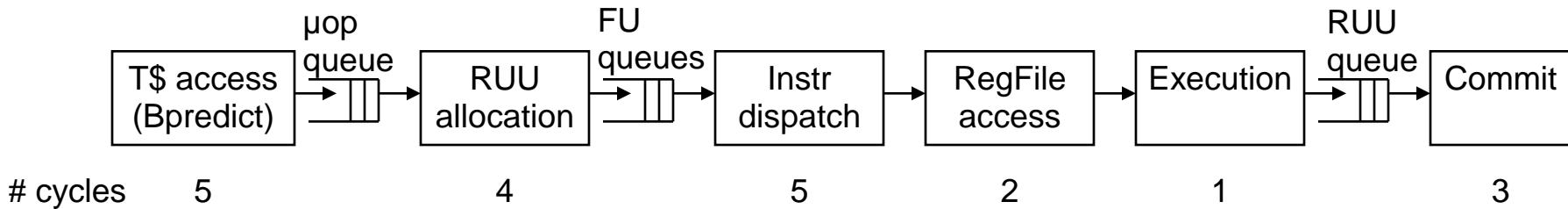
- On a processor with an instruction window size of 2K, maximum 64-way issue capability, and a tournament branch predictor with 8K entries



A SS Example

❑ Intel Pentium 4 (IA-32 ISA)

- Decodes the IA-32 instructions into microoperations
- Does register renaming with a RUU-like structure
- Has a 20 stage pipeline

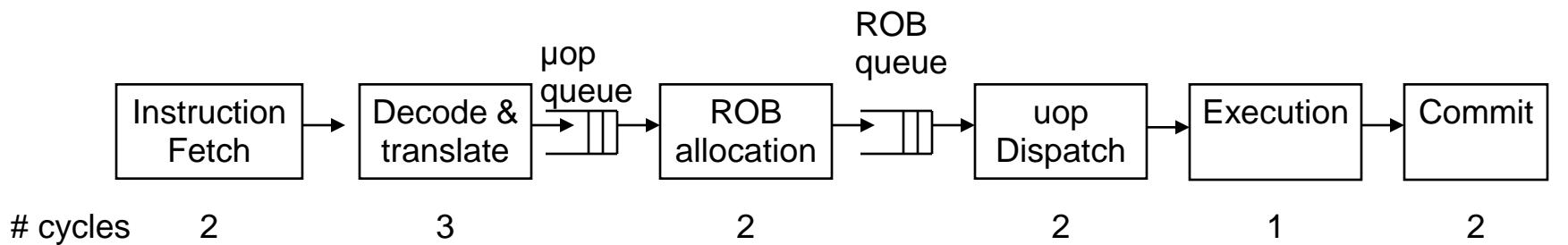


- 7 FUs: 2 integer ALUs, 1 FP ALU, 1FP move, load, store, complex
- Up to 126 instructions in flight, including 48 loads and 24 stores
- 4K entry branch predictor

Another SS Example

AMD Opteron

- Decodes the IA-32 instructions into microoperations
- Does register renaming with a Reorder Buffer (RUU-like)
- Has a 12 stage (integer) pipeline



- More details at

http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html

Next Lecture and Reminders

- ❑ Next lecture

- Multi-processor systems.

- ❑ Reminders

- Assignment 2 is now available
 - Lab attendance will be recorded for survey purposes.

ECE4074

Computer Architecture

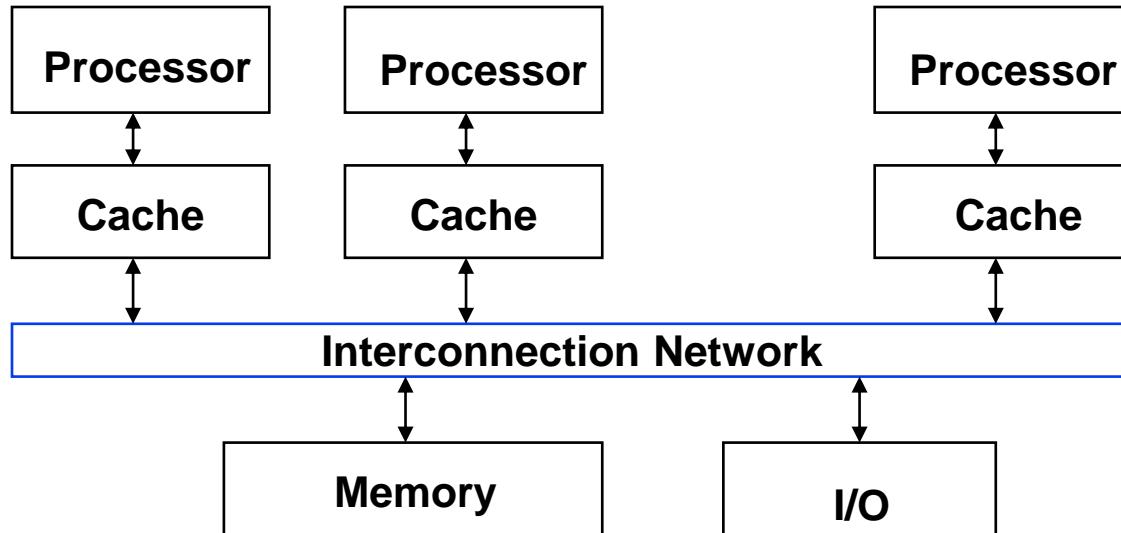
Semester 2 2012

Chapter 7A: Intro to Multiprocessor Systems

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

The Big Picture: Where are We Now?

- ❑ Multiprocessor – a computer system with at least two processors



- Can deliver high throughput for independent jobs via **job-level parallelism** or **process-level parallelism**
- And improve the run time of a *single* program that has been specially crafted to run on a multiprocessor - a **parallel processing program**

Multicores Now Common

- ❑ The power challenge has forced a change in the design of microprocessors
 - Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- ❑ Today's microprocessors typically contain more than one core – **Chip Multicore microProcessors (CMPs)** – in a single IC
 - The number of cores is expected to double every two years

| Product | AMD Barcelona | Intel Nehalem | IBM Power 6 | Sun Niagara |
|----------------|------------------|------------------|----------------|-------------|
| Cores per chip | 4 | 4 | 2 | 8 |
| Clock rate | 2.5 GHz | ~2.5 GHz? | 4.7 GHz | 1.4 GHz |
| Power | 120 W | ~100 W? | ~100 W? | 94 W |

Other Multiprocessor Basics

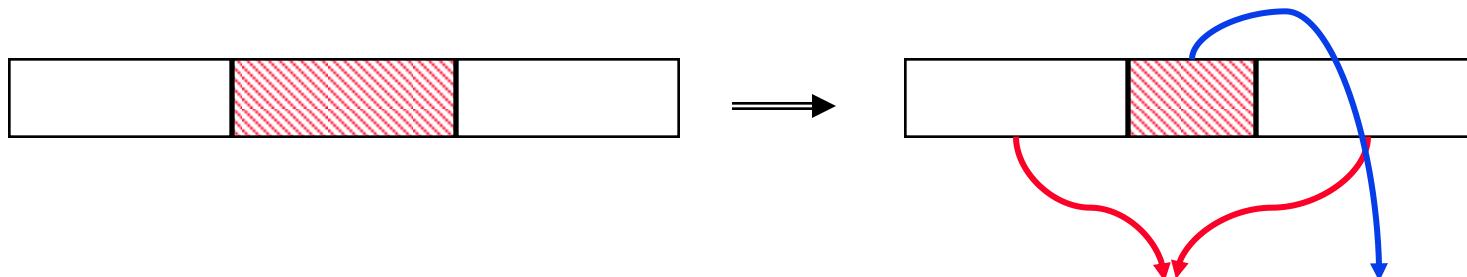
- ❑ Some of the problems that need higher performance can be handled simply by using a **cluster** – a set of independent servers (or PCs) connected over a local area network (LAN) functioning as a single large multiprocessor
 - Search engines, Web servers, email servers, databases, ...
- ❑ A key challenge is to craft parallel (concurrent) programs that have high performance on multiprocessors as the number of processors increase – i.e., that **scale**
 - Scheduling, load balancing, time for synchronization, overhead for communication

Encountering Amdahl's Law

- ❑ Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- ❑ Suppose that enhancement E accelerates a fraction F ($F < 1$) of the task by a factor S ($S > 1$) and the remainder of the task is unaffected



$$\text{ExTime w/ E} = \text{ExTime w/o E} \times ((1-F) + F/S)$$

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$

Example 1: Amdahl's Law

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$

- ❑ Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

$$\text{Speedup w/ E} = 1/(.75 + .25/20) = 1.31$$

- ❑ What if its usable only 15% of the time?

$$\text{Speedup w/ E} = 1/(.85 + .15/20) = 1.17$$

- ❑ Amdahl's Law tells us that to achieve linear speedup with 100 processors, **none** of the original computation can be scalar(no ILP)!
- ❑ To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less

$$\text{Speedup w/ E} = 1/(.001 + .999/100) = 90.99$$

Example 2: Amdahl's Law

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$

- ❑ Consider computing 10 scalar operations and two 10 by 10 matrices (matrix sum) on 10 processors

$$\text{Speedup w/ E} = 1/(.091 + .909/10) = 1/0.1819 = 5.5$$

- ❑ What if there are 100 processors ?

$$\text{Speedup w/ E} = 1/(.091 + .909/100) = 1/0.10009 = 10.0$$

- ❑ What if the matrices are 100 by 100 (or 10,010 operations in total) on 10 processors?

$$\text{Speedup w/ E} = 1/(.001 + .999/10) = 1/0.1009 = 9.9$$

- ❑ What if there are 100 processors ?

$$\text{Speedup w/ E} = 1/(.001 + .999/100) = 1/0.01099 = 91$$

Scaling

- ❑ To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.
 - **Strong scaling** – when speedup can be achieved on a multiprocessor without increasing the size of the problem
 - **Weak scaling** – when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors
- ❑ Load balancing is another important factor. Just a single processor with twice the load of the others cuts the speedup almost in half

Multiprocessor/Clusters Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- ❑ Q1 – Single address space shared by all processors
- ❑ Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
 - Use of shared data must be coordinated via **synchronization** primitives (locks) that allow access to data to only one processor at a time
- ❑ They come in two styles
 - Uniform memory access (**UMA**) multiprocessors
 - Nonuniform memory access (**NUMA**) multiprocessors
- ❑ Programming NUMAs are harder
- ❑ But NUMAs can scale to larger sizes and have lower latency to local memory

Summing 100,000 Numbers on 100 Proc. SMP

- ❑ Processors start by running a loop that sums their subset of vector A numbers (vectors A and sum are **shared** variables, Pn is the processor's number, i is a **private** variable)

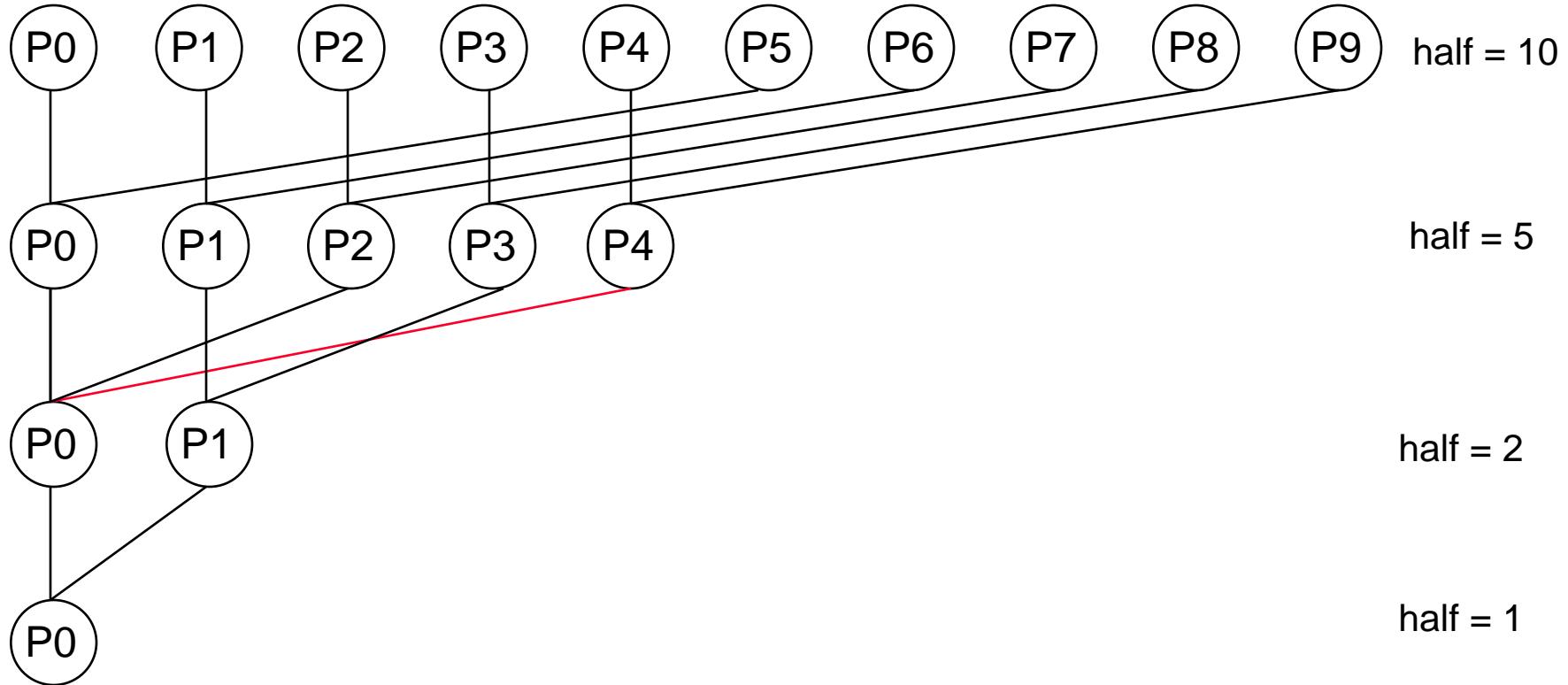
```
sum[Pn] = 0;  
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)  
    sum[Pn] = sum[Pn] + A[i];
```

- ❑ The processors then coordinate in adding together the partial sums (half is a **private** variable initialized to 100 (the number of processors)) – reduction

```
do {  
    synch();           // synchronize first  
    if (half%2 != 0 && Pn == 0)  
        sum[0] = sum[0] + sum[half-1];  
    half = half/2;  
    if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half];  
} while(half != 1);      // final sum in sum[0]
```

An Example with 10 Processors

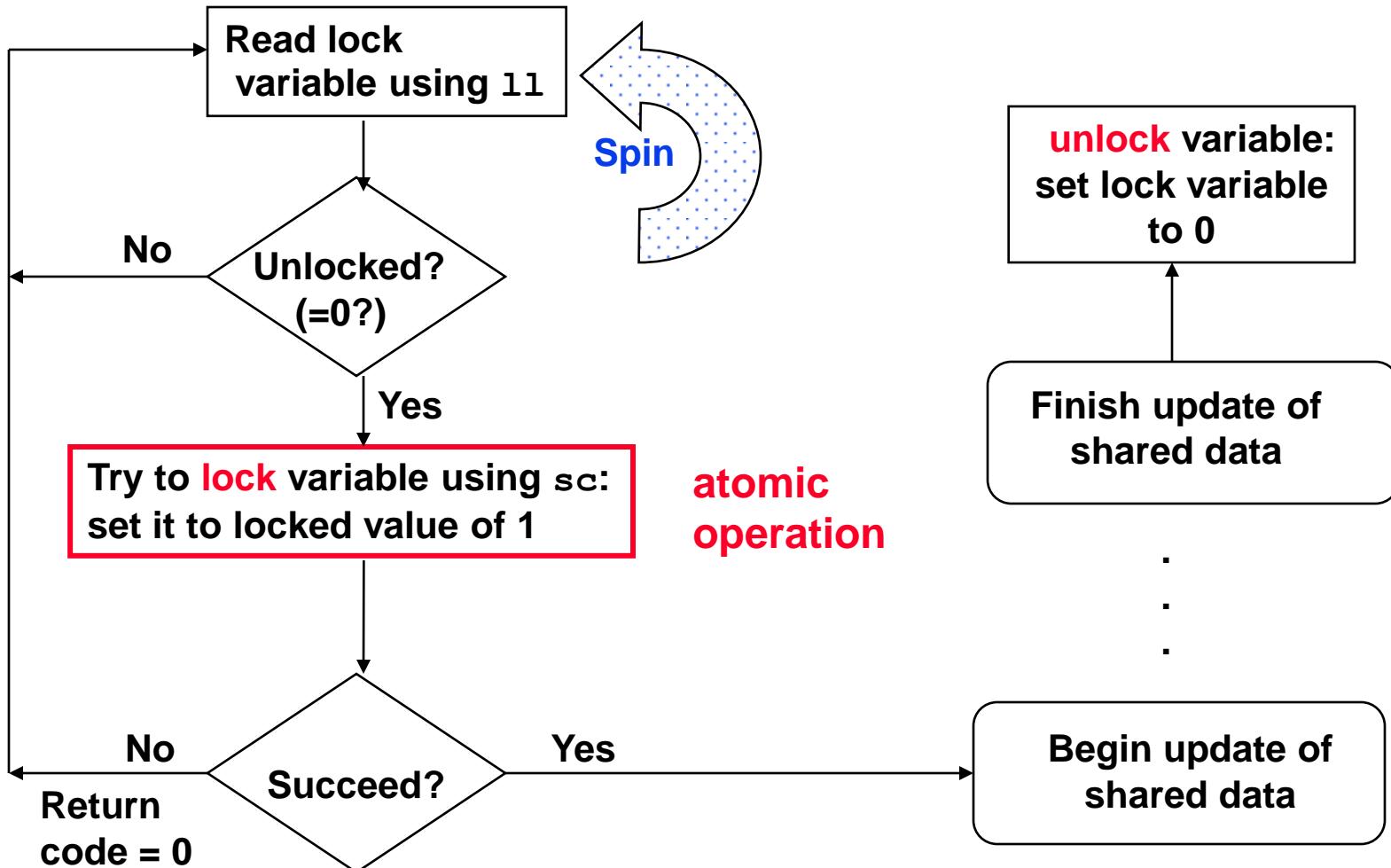
sum[P0]sum[P1]sum[P2] sum[P3]sum[P4]sum[P5]sum[P6] sum[P7]sum[P8] sum[P9]



Process Synchronization

- ❑ Need to be able to coordinate processes working on a common task
- ❑ Lock variables (**semaphores**) are used to coordinate or synchronize processes
- ❑ Need an architecture-supported arbitration mechanism to decide which processor gets access to the lock variable
 - Single bus provides arbitration mechanism, since the bus is the only path to memory – the processor that gets the bus wins
- ❑ Need an architecture-supported operation that locks the variable
 - Locking can be done via an **atomic swap operation** (on the MIPS we have `ll` and `sc` one example of where a processor can both read a location *and* set it to the locked state – **test-and-set** – in the same bus operation)

Spin Lock Synchronization



The *single winning* processor will succeed in writing a 1 to the lock variable - all others processors will get a return code of 0

Review: Summing Numbers on a SMP

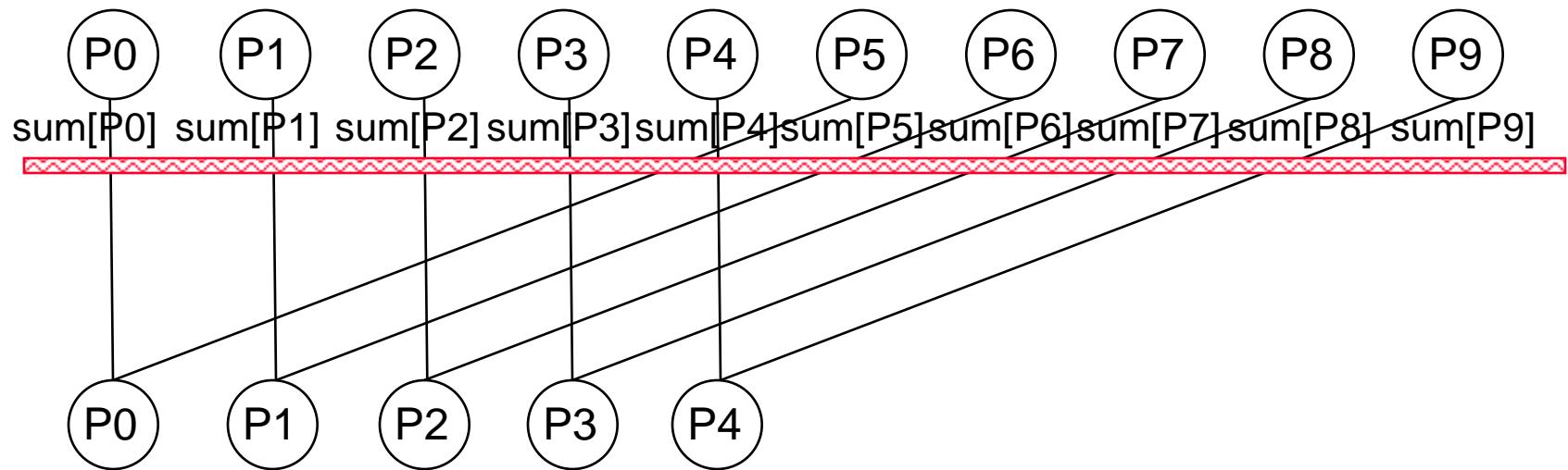
- ❑ P_n is the processor's number, vectors A and sum are **shared** variables, i is a **private** variable, half is a **private** variable initialized to the number of processors

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
                                // each processor sums its
                                // subset of vector A

do{                               // adding together the
                                // partial sums
    synch();                  // synchronize first
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
    half = half/2;
    if (Pn<half) sum[Pn] = sum[Pn] + sum[Pn+half];
}while(half != 1);           //final sum in sum[0]
```

An Example with 10 Processors

- ❑ `synch()`: Processors must synchronize before the “consumer” processor tries to read the results from the memory location written by the “producer” processor
 - **Barrier synchronization** – a synchronization scheme where processors wait at the barrier, not proceeding until every processor has reached it



Barrier Implemented with Spin-Locks

- ❑ n is a **shared** variable initialized to the number of processors, count is a **shared** variable initialized to 0, arrive and depart are **shared** spin-lock variables where arrive is initially unlocked and depart is initially locked

```
void synch() {  
    lock(arrive);  
    count = count + 1;      // count the processors as  
    if( count < n )        // they arrive at barrier  
        then unlock(arrive)  
    else unlock(depart);  
  
    lock(depart);  
    count = count - 1;      // count the processors as  
    if( count > 0 )        // they leave barrier  
        then unlock(depart)  
    else unlock(arrive); }
```

Spin-Locks on Bus Connected ccUMAs

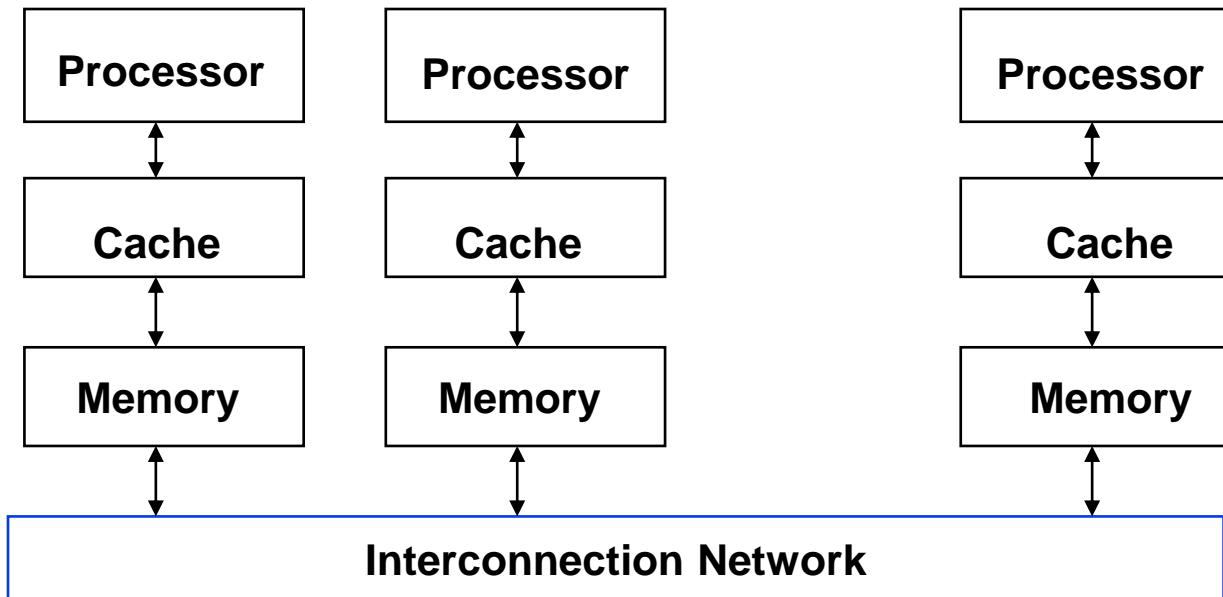
- ❑ With a bus based cache coherency protocol (write invalidate), spin-locks allow processors to wait on a local copy of the lock in their caches
 - Reduces bus traffic – once the processor with the lock releases the lock (writes a 0) all other caches see that write and invalidate their old copy of the lock variable. Unlocking restarts the race to get the lock. The winner gets the bus and writes the lock back to 1. The other caches then invalidate their copy of the lock and on the next lock read fetch the new lock value (1) from memory.
- ❑ This scheme has problems scaling up to many processors because of the communication traffic when the lock is released and contested

Aside: Cache Coherence Bus Traffic

| | Proc P0 | Proc P1 | Proc P2 | Bus activity | Memory |
|---|-------------------|--------------------------|--------------------------|---------------------------------|-------------------------------|
| 1 | Has lock | Spins | Spins | None | |
| 2 | Releases lock (0) | Spins | Spins | Bus services P0's invalidate | |
| 3 | | Cache miss | Cache miss | Bus services P2's cache miss | |
| 4 | | Waits | Reads lock (0) | Response to P2's cache miss | Update lock in memory from P0 |
| 5 | | Reads lock (0) | Swaps lock (ll, sc of 1) | Bus services P1's cache miss | |
| 6 | | Swaps lock (ll, sc of 1) | Swap succeeds | Response to P1's cache miss | Sends lock variable to P1 |
| 7 | | Swap fails | Has lock | Bus services P2's invalidate | |
| 8 | | Cache miss /Spins | Has lock | Bus services P1's cache miss | |

Message Passing Multiprocessors (MPP)

- ❑ Each processor has its own private address space
- ❑ Q1 – Processors share data by *explicitly* sending and receiving information (**message passing**)
- ❑ Q2 – Coordination is built into message passing primitives (**message send** and **message receive**)



Summing 100,000 Numbers on 100 Proc. MPP

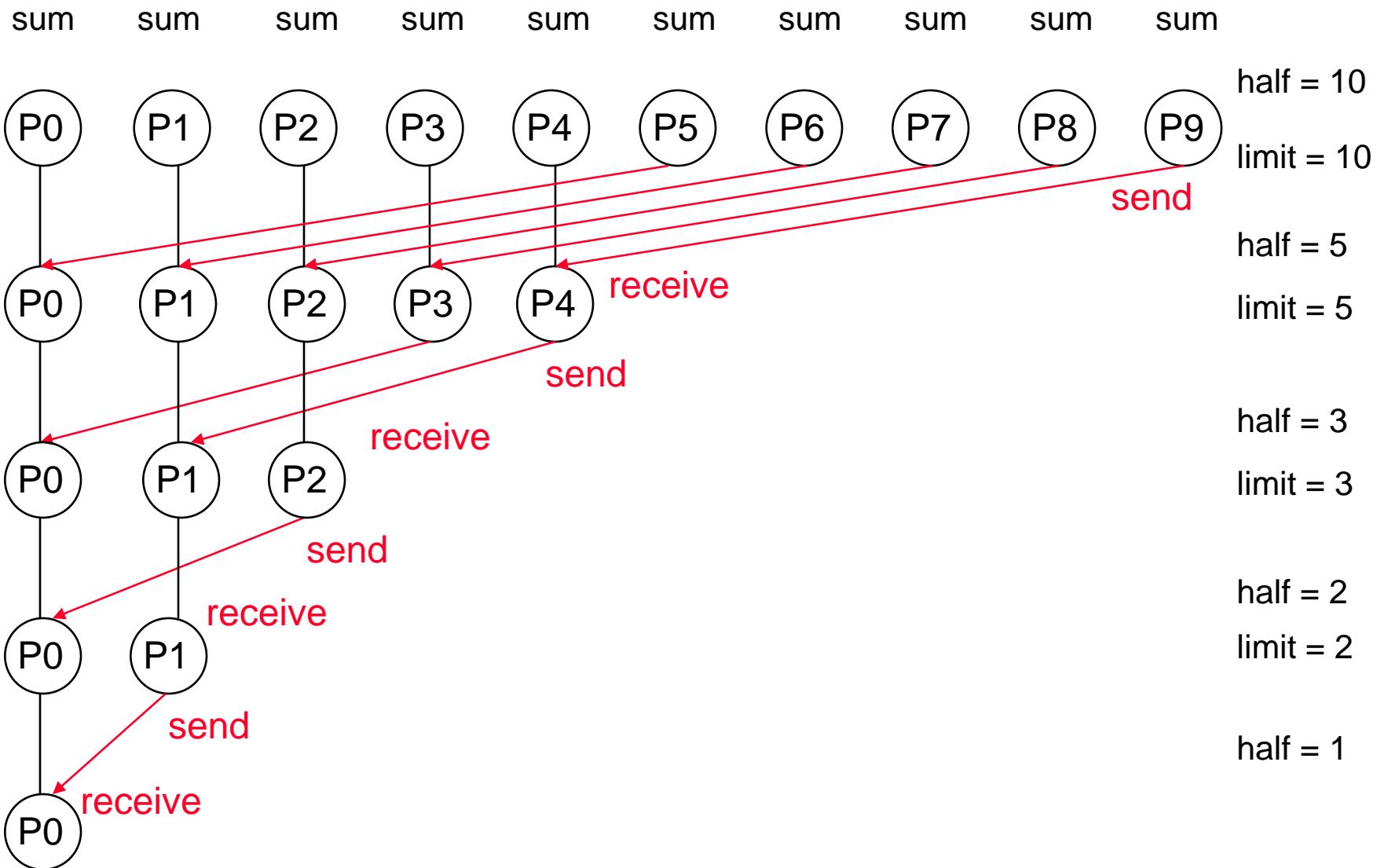
- ❑ Start by distributing 1000 elements of vector A to each of the local memories and summing each subset in parallel

```
sum = 0;  
for (i = 0; i<1000; i = i + 1)  
    sum = sum + Al[i];      // sum local array subset
```

- ❑ The processors then coordinate in adding together the sub sums (Pn is the number of processors, send(x, y) sends value y to processor x, and receive() receives a value)

```
half = 100;  
limit = 100;  
do{  
    half = (half+1)/2;      //dividing line  
    if (Pn>= half && Pn<limit) send(Pn-half,sum);  
    if (Pn<(limit/2)) sum = sum + receive();  
    limit = half;  
}while(half != 1);        //final sum in P0's sum
```

An Example with 10 Processors



Pros and Cons of Message Passing

- ❑ Message sending and receiving is *much* slower than addition, for example
- ❑ But message passing multiprocessors are much easier for hardware designers to design
 - Don't have to worry about cache coherency for example
- ❑ The advantage for programmers is that communication is explicit, so there are fewer “performance surprises” than with the implicit communication in cache-coherent SMPs.
 - Message passing standard MPI-2 (www.mpi-forum.org)
- ❑ However, it's harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance.
 - With cache-coherent shared memory the hardware figures out what data needs to be communicated

Networks of Workstations (NOWs) Clusters

- ❑ Clusters of off-the-shelf, whole computers with multiple private address spaces connected using the I/O bus of the computers
 - lower bandwidth than multiprocessor that use the processor-memory (front side) bus
 - lower speed network links
 - more conflicts with I/O traffic
- ❑ Clusters of N processors have N copies of the OS limiting the memory available for applications
- ❑ Improved system availability and expandability
 - easier to replace a machine without bringing down the whole system
 - allows rapid, incremental expandability
- ❑ Economy-of-scale advantages with respect to costs

Multithreading on A Chip

- ❑ Find a way to “hide” true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are **independent** of those stalling instructions
- ❑ **Hardware multithreading** – increase the utilization of resources on a chip by allowing multiple processes (**threads**) to share the functional units of a single processor
 - Processor must duplicate the state hardware for each thread – a separate register file, PC, instruction buffer, and store buffer for each thread
 - The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
 - The memory can be shared through virtual memory mechanisms
 - Hardware must support *efficient* thread context switching

Types of Multithreading

❑ Fine-grain – switch threads on every instruction issue

- Round-robin thread interleaving (skipping stalled threads)
- Processor must be able to switch threads on every clock cycle
- Advantage – can hide throughput losses that come from both short and long stalls
- Disadvantage – increases the latency of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads

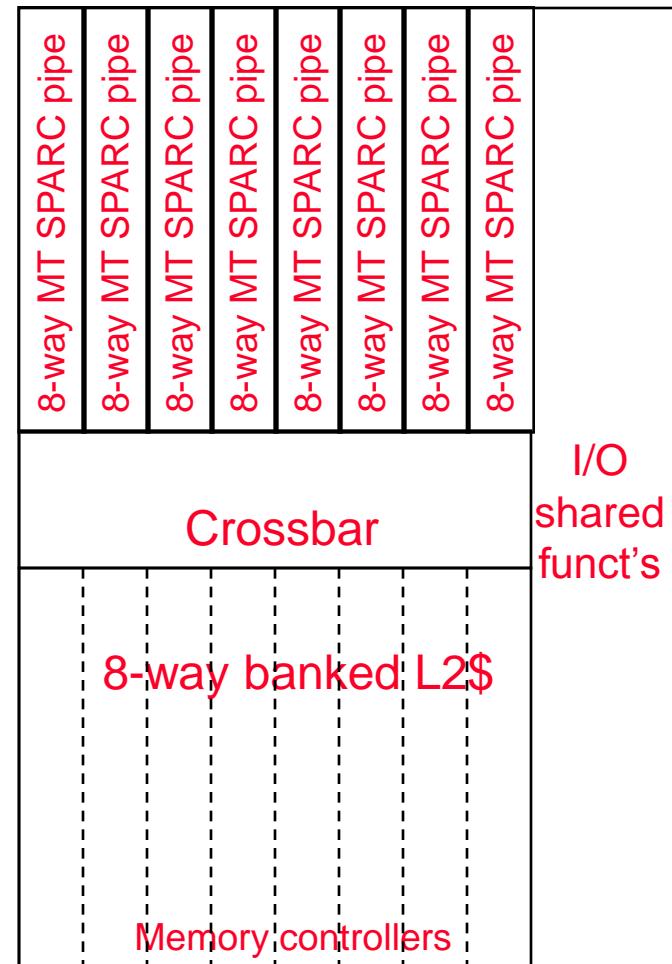
❑ Coarse-grain – switches threads only on costly stalls (e.g., L2 cache misses)

- Advantages – thread switching doesn't have to be essentially free and much less likely to slow down the execution of an individual thread
- Disadvantage – limited, due to pipeline start-up costs, in its ability to overcome throughput loss
 - Pipeline must be flushed and refilled on thread switches

Multithreaded Example: Sun's Niagara (UltraSparc T2)

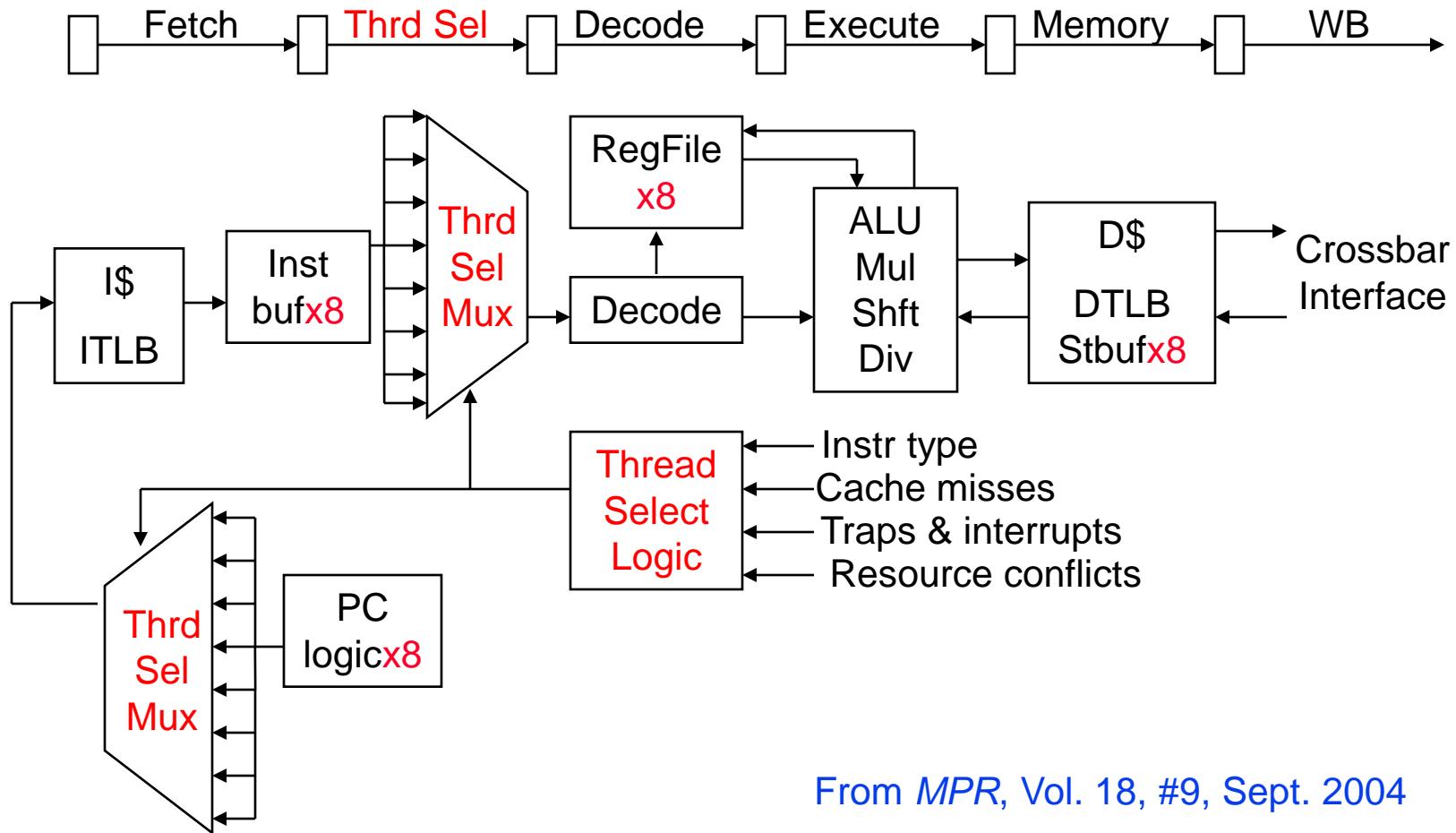
- Eight fine grain multithreaded single-issue, in-order cores (no speculation, no dynamic branch prediction)

| | |
|----------------|-------------|
| | Niagara 2 |
| Data width | 64-b |
| Clock rate | 1.4 GHz |
| Cache (I/D/L2) | 16K/8K/4M |
| Issue rate | 1 issue |
| Pipe stages | 6 stages |
| BHT entries | None |
| TLB entries | 64I/64D |
| Memory BW | 60+ GB/s |
| Transistors | ??? million |
| Power (max) | <95 W |



Niagara Integer Pipeline

- Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient

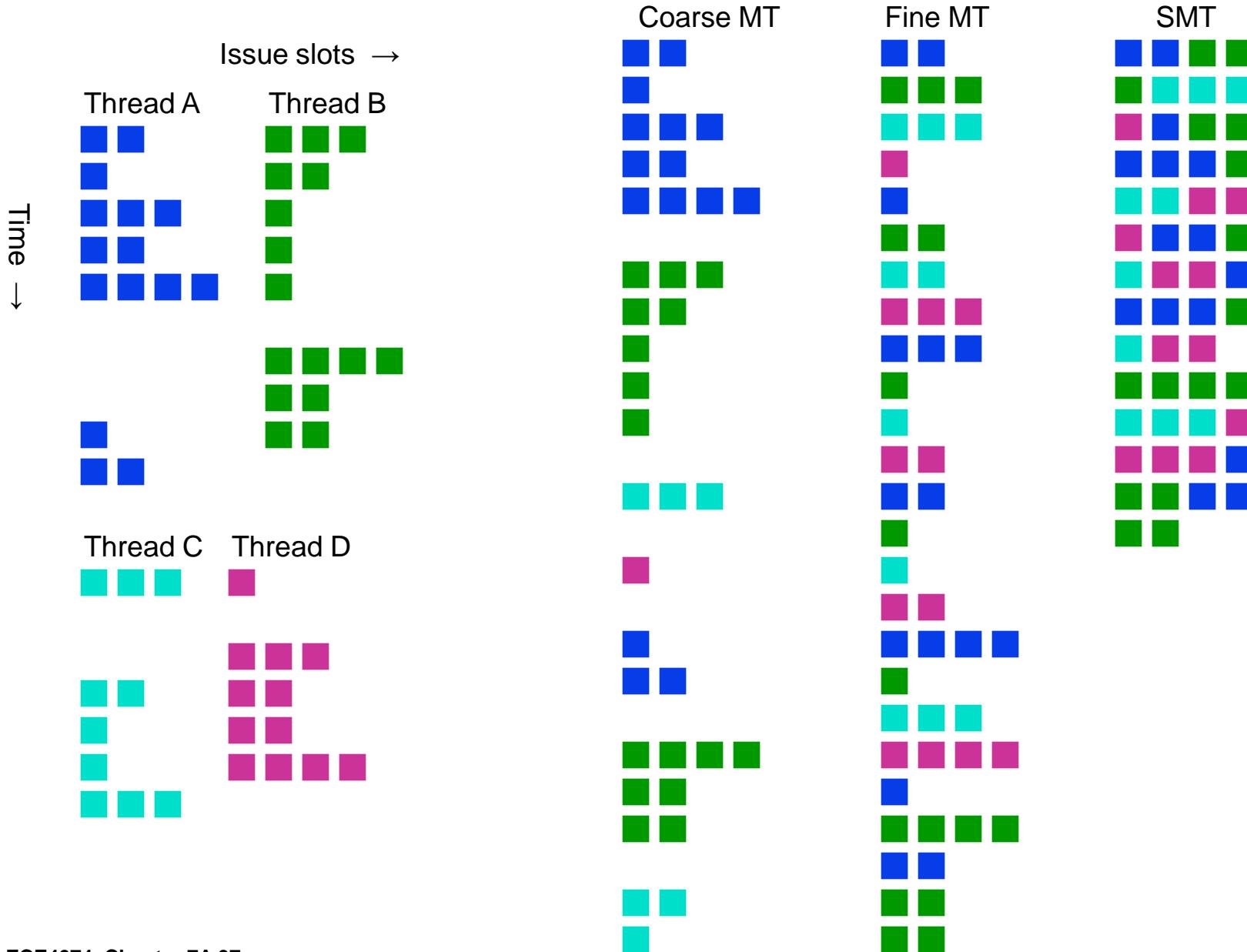


From *MPR*, Vol. 18, #9, Sept. 2004

Simultaneous Multithreading (SMT)

- ❑ A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and **thread-level parallelism (TLP)**
 - Most SS processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)
 - With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them
 - Need separate rename tables (RUUs) for each thread or need to be able to indicate which thread the entry belongs to
 - Need the capability to commit from multiple threads in one cycle
- ❑ Intel's Pentium 4 and above SMT is called **hyperthreading**
 - Supports just two threads (doubles the architecture state)

Threading on a 4-way SS Processor Example



Review: Multiprocessor Basics

- ❑ Q1 – How do they share data?
- ❑ Q2 – How do they coordinate?
- ❑ Q3 – How scalable is the architecture? How many processors?

| | | | # of Proc |
|---------------------|-----------------|------|-----------|
| Communication model | Message passing | | 8 to 2048 |
| | Shared address | NUMA | 8 to 256 |
| | | UMA | 2 to 64 |
| Physical connection | Network | | 8 to 256 |
| | Bus | | 2 to 36 |

Next Lecture and Reminders

- ❑ Next lecture

- Multiprocessor architectures

- ❑ Reminders

- Lab attendance is being taken for survey purposes
 - Assignment 2 is due at the end of week 10

ECE4074

Computer Architecture

Semester 2 2012

Chapter 7B: SIMDs, Vectors, and GPUs

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

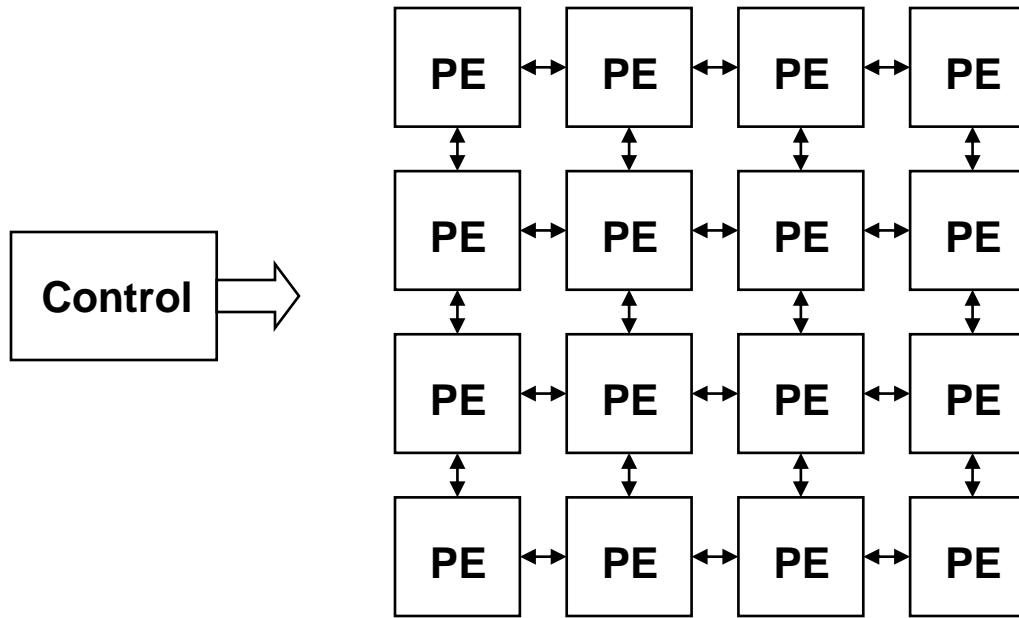
Multiprocessor/Clusters Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors can be supported?

Flynn's Classification Scheme

- ❑ SISD – single instruction, single data stream
 - aka uniprocessor - what we have been talking about all semester
- ❑ SIMD – single instruction, multiple data streams
 - single control unit broadcasting operations to multiple datapaths
- ❑ MISD – multiple instruction, single data
 - no such machine (although some people put vector machines in this category)
- ❑ MIMD – multiple instructions, multiple data streams
 - aka multiprocessors (SMPs, MPPs, clusters, NOWs)
- ❑ Now obsolete terminology except for . . .

SIMD Processors



- ❑ Single control unit (one copy of the code)
- ❑ Multiple datapaths (Processing Elements – PEs) running in parallel
 - Q1 – PEs are interconnected (usually via a mesh or torus) and exchange/share data as directed by the control unit
 - Q2 – Each PE performs the same operation on its own local data

Example SIMD Machines

| | Maker | Year | # PEs | # b/ PE | Max memory (MB) | PE clock (MHz) | System BW (MB/s) |
|-----------|----------------------|------|--------|------------|-----------------------|----------------------|------------------------|
| Illiac IV | UIUC | 1972 | 64 | 64 | 1 | 13 | 2,560 |
| DAP | ICL | 1980 | 4,096 | 1 | 2 | 5 | 2,560 |
| MPP | Goodyear | 1982 | 16,384 | 1 | 2 | 10 | 20,480 |
| CM-2 | Thinking Machines | 1987 | 65,536 | 1 | 512 | 7 | 16,384 |
| MP-1216 | MasPar | 1989 | 16,384 | 4 | 1024 | 25 | 23,000 |

- ❑ Did SIMDs die out in the early 1990s ??

Multimedia SIMD Extensions

- ❑ The most widely used variation of SIMD is found in almost every microprocessor today – as the basis of MMX and SSE instructions added to improve the performance of multimedia programs
 - A single, wide ALU is partitioned into many smaller ALUs that operate in parallel



- Loads and stores are simply as wide as the widest ALU, so the same data transfer can transfer one 32 bit value, two 16 bit values or four 8 bit values
- ❑ There are now hundreds of SSE instructions in the x86 to support multimedia operations

Vector Processors

- ❑ A vector processor (e.g., Cray) **pipelines the ALUs** to get good performance at lower cost. A key feature is a set of **vector registers** to hold the operands and results.
 - Collect the data elements from memory, put them in order into a large set of registers, operate on them sequentially in registers, and then write the results back to memory
 - They formed the basis of supercomputers in the 1980's and 90's
- ❑ Consider extending the MIPS instruction set (VMIPS) to include vector instructions, e.g.,
 - `addv.d` to add two double precision vector register values
 - `addvs.d` and `mulvs.d` to add (or multiply) a scalar register to (by) each element in a vector register
 - `lv` and `sv` do vector load and vector store and load or store an entire vector of double precision data

MIPS vs VMIPS DAXPY Codes: $Y = a \times X + Y$

| | | | |
|-------|-------|----------------|--------------------------|
| | l.d | \$f0,a(\$sp) | ; load scalar a |
| | addiu | \$s4,\$s0,#512 | ; upper bound to load to |
| loop: | l.d | \$f2,0(\$s0) | ; load X(i) |
| | mul.d | \$f2,\$f2,\$f0 | ; a × X(i) |
| | l.d | \$f4,0(\$s1) | ; load Y(i) |
| | add.d | \$f4,\$f4,\$f2 | ; a × X(i) + Y(i) |
| | s.d | \$f4,0(\$s1) | ; store into Y(i) |
| | addiu | \$s0,\$s0,#8 | ; increment X index |
| | addiu | \$s1,\$s1,#8 | ; increment Y index |
| | subu | \$t0,\$s4,\$s0 | ; compute bound |
| | bne | \$t0,\$0,loop | ; check if done |

| | | | |
|--|---------|----------------|--------------------------|
| | l.d | \$f0,a(\$sp) | ; load scalar a |
| | lv | \$v1,0(\$s0) | ; load vector X |
| | mulvs.d | \$v2,\$v1,\$f0 | ; vector-scalar multiply |
| | lv | \$v3,0(\$s1) | ; load vector Y |
| | addv.d | \$v4,\$v2,\$v3 | ; add Y to a × X |
| | sv | \$v4,0(\$s1) | ; store vector result |

Vector verus Scalar

- ❑ Instruction fetch and decode bandwidth is dramatically reduced (also saves power)
 - Only six instructions in VMIPS versus almost 600 in MIPS for 64 element DAXPY
- ❑ Hardware doesn't have to check for data hazards within a vector instruction. A vector instruction will only stall for the first element, then subsequent elements will flow smoothly down the pipeline. And control hazards are nonexistent.
 - MIPS stall frequency is about 64 times higher than VMIPS for DAXPY
- ❑ Easier to write code for data-level parallel app's
- ❑ Have a known access pattern to memory, so heavily interleaved memory banks work well. The cost of latency to memory is seen only once for the entire vector

Example Vector Machines

| | Maker | Year | Peak perf. | # vector Processors | PE clock (MHz) |
|-----------------|-------|------|------------------|---------------------|----------------|
| STAR-100 | CDC | 1970 | ?? | 113 | 2 |
| ASC | TI | 1970 | 20 MFLOPS | 1, 2, or 4 | 16 |
| Cray 1 | Cray | 1976 | 80 to 240 MFLOPS | | 80 |
| Cray Y-MP | Cray | 1988 | 333 MFLOPS | 2, 4, or 8 | 167 |
| Earth Simulator | NEC | 2002 | 35.86 TFLOPS | 8 | 1000 |

- ❑ Did Vector machines die out in the late 1990s ??

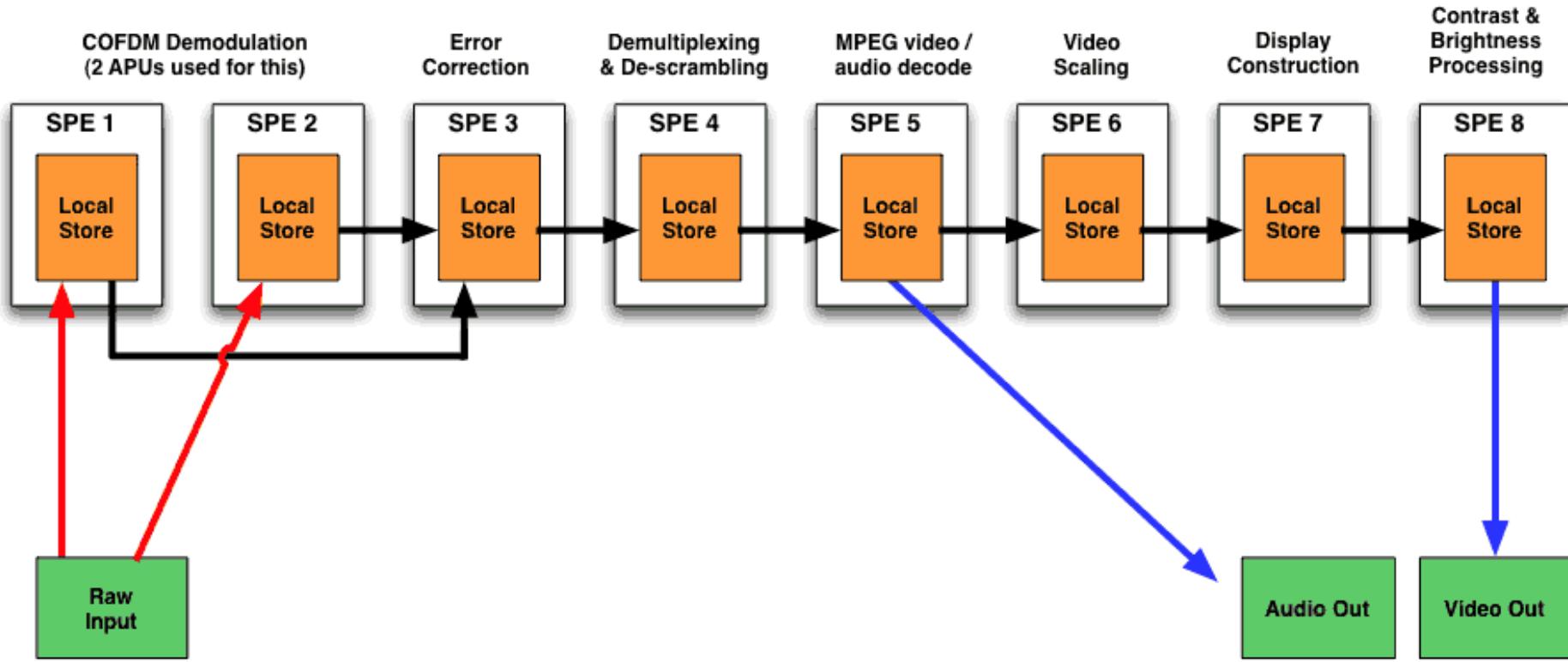
The PS3 “Cell” Processor Architecture

- ❑ Composed of a non-SMP architecture
 - 234M transistors @ 4Ghz
 - 1 Power Processing Element (PPE) “control” processor. The PPE is similar to a Xenon core
 - Slight ISA differences, and fine-grained MT instead of real SMT
 - And 8 “Synergistic” (**SIMD**) Processing Elements (SPEs). The real compute power and differences lie in the SPEs (21M transistors each)
 - An attempt to ‘fix’ the memory latency problem by giving each SPE complete control over it’s own 256KB “scratchpad” memory – 14M transistors
 - Direct mapped for low latency
 - 4 **vector** units per SPE, 1 of everything else – 7M transistors
 - 512KB L2\$ and a massively high bandwidth (200GB/s) processor-memory bus
- ❑ X-Box has 3 similar PPE cores, each SMT and SIMD

How to make use of the SPEs

Stream Processing

Decoding digital TV is a complex process
but it can be broken into a stream



Graphics Processing Units (GPUs)

- ❑ GPUs are accelerators that supplement a CPU so they do not need to be able to perform all of the tasks of a CPU. They dedicate *all* of their resources to graphics
 - CPU-GPU combination – **heterogeneous** multiprocessing
- ❑ Programming interfaces that are free from backward binary compatibility constraints resulting in more rapid innovation in GPUs than in CPUs
 - Application programming interfaces (APIs) such as OpenGL and DirectX coupled with high-level graphics shading languages such as NVIDIA's Cg and CUDA and Microsoft's HLSL
- ❑ GPU data types are vertices (x, y, z, w) coordinates and pixels (red, green, blue, alpha) color components
- ❑ GPUs execute many threads (e.g., vertex and pixel shading) in parallel – *lots* of data-level parallelism

Typical GPU Architecture Features

- ❑ Rely on having enough threads to hide the latency to memory (not caches as in CPUs)
 - Each GPU is highly multithreaded
- ❑ Use extensive parallelism to get high performance
 - Have extensive set of SIMD instructions; moving towards multicore
- ❑ Main memory is bandwidth, not latency driven
 - GPU DRAMs are wider and have higher bandwidth, but are typically smaller, than CPU memories
- ❑ Leaders in the marketplace (in 2008)
 - NVIDIA GeForce 8800 GTX (16 multiprocessors each with 8 multithreaded processing units)
 - AMD's ATI Radeon and ATI FireGL

Mobile Devices

- ❑ Video and audio decoding and processing often require large amount of data performing the same operation. Streaming processing takes advantage of this.
- ❑ Many mobile devices have DSP/SIMD coprocessors for this type of rolls.
- ❑ Simple hardware allows for high performance and low energy use.
- ❑ Examples:
 - ARM Processors:
 - NEON in ARM 8, optional in ARM 9
 - Apple A5 (really an ARM 9 with NEON)
 - Tegra 3 (multiple ARM cores with NEON)

Next Lecture and Reminders

❑ Next lecture

- Multiprocessor network topologies

❑ Reminders

- Assignment 2 Due end week 10
 - No demonstration, only report
 - Explain your results
 - Performance is not marked
 - Choose your write strategy.

ECE4074

Computer Architecture

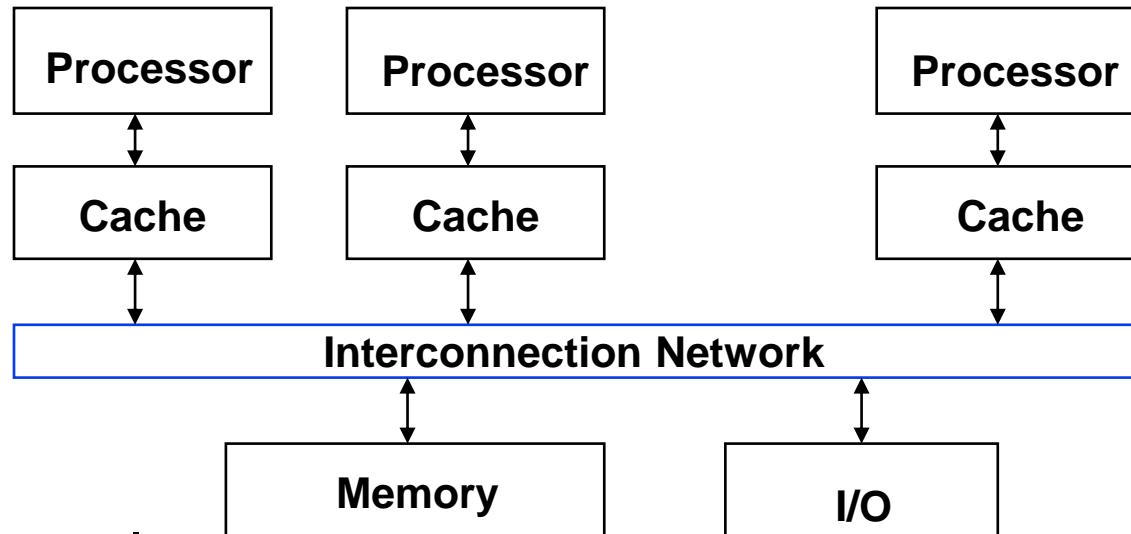
Semester 2 2012

Chapter 7C: Multiprocessor Network Topologies

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Review: Shared Memory Multiprocessors (SMP)

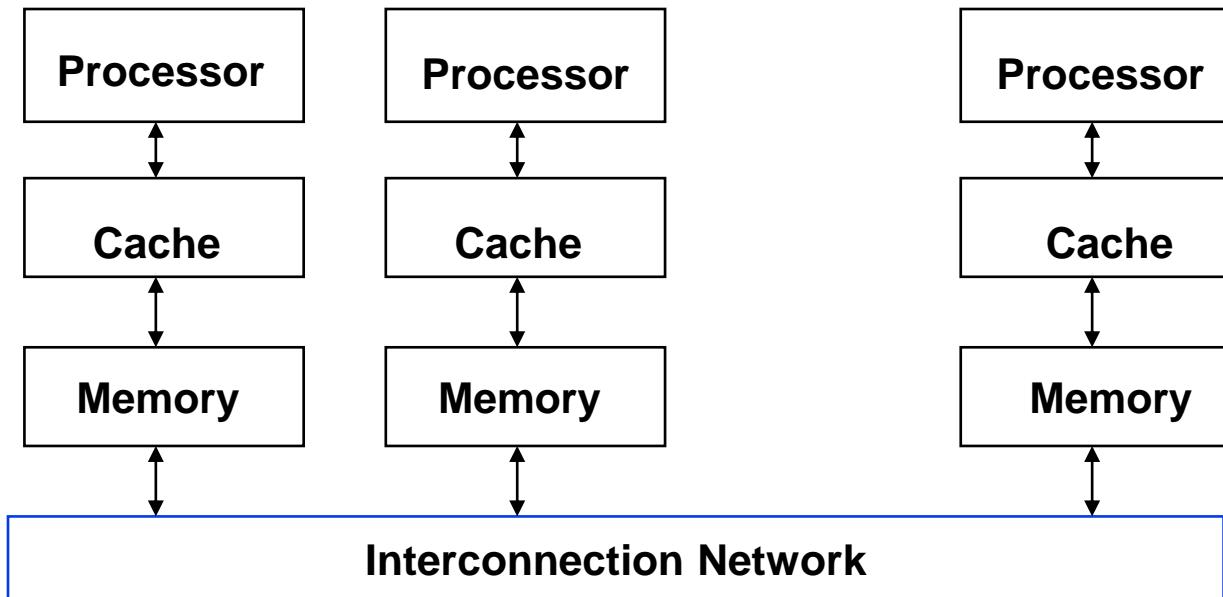
- ❑ Q1 – Single address space shared by all processors
- ❑ Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
 - Use of shared data must be coordinated via **synchronization** primitives (locks) that allow access to data to only one processor at a time



- ❑ They come in two styles
 - Uniform memory access (**UMA**) multiprocessors
 - Nonuniform memory access (**NUMA**) multiprocessors

Message Passing Multiprocessors (MPP)

- ❑ Each processor has its own private address space
- ❑ Q1 – Processors share data by *explicitly* sending and receiving information (**message passing**)
- ❑ Q2 – Coordination is built into message passing primitives (**message send** and **message receive**)



Communication in Network Connected Multi's

❑ Implicit communication via loads and stores

- hardware designers have to provide coherent caches and process (thread) synchronization primitive (like `ll` and `sc`)
- lower communication overhead
- harder to overlap computation with communication
- more efficient to use an address to remote data when *needed* rather than to send for it in case it *might* be used

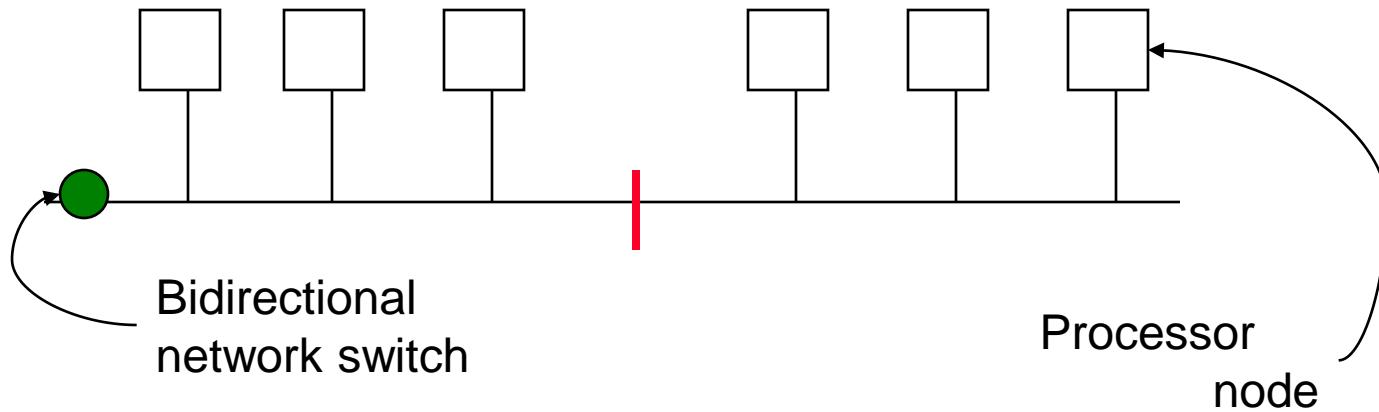
❑ Explicit communication via sends and receives

- simplest solution for hardware designers
- higher communication overhead
- easier to overlap computation with communication
- easier for the programmer to optimize communication

IN Performance Metrics

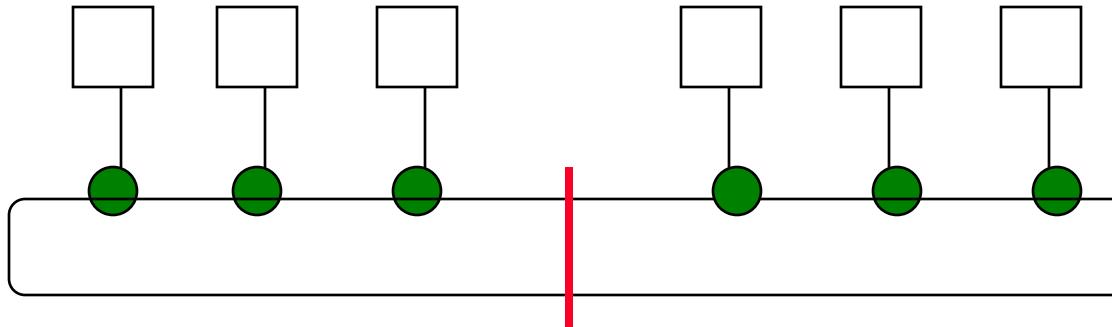
- ❑ Network cost
 - number of switches
 - number of (bidirectional) links on a switch to connect to the network (plus one link to connect to the processor)
 - width in bits per link, length of link wires (on chip)
- ❑ Network bandwidth (NB) – represents the **best** case
 - Usually: bandwidth of each link \times number of links
- ❑ Bisection bandwidth (BB) – closer to the **worst** case
 - divide the machine in two parts, each with half the nodes and sum the bandwidth of the links that cross the dividing line
- ❑ Other IN performance issues
 - latency on an unloaded network to send and receive messages
 - throughput – maximum # of messages transmitted per unit time
 - # routing hops worst case, congestion control and delay, fault tolerance, power efficiency

Bus IN



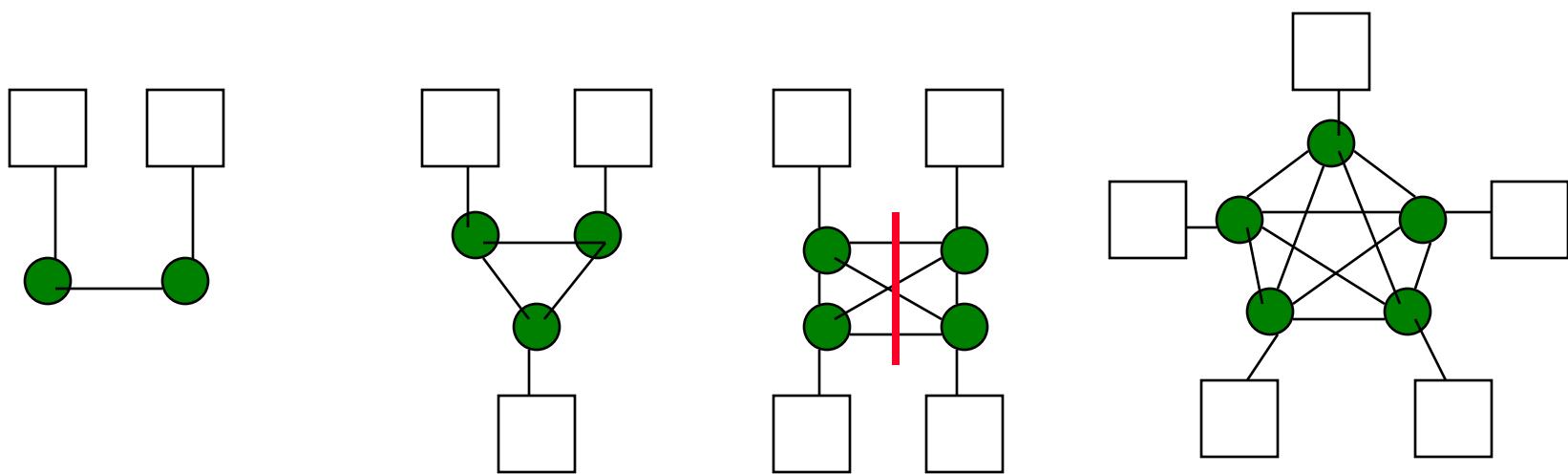
- ❑ N processors, 1 switch (●), 1 link (the bus)
- ❑ Only 1 simultaneous transfer at a time
 - NB = link (bus) bandwidth * 1
 - BB = link (bus) bandwidth * 1

Ring IN



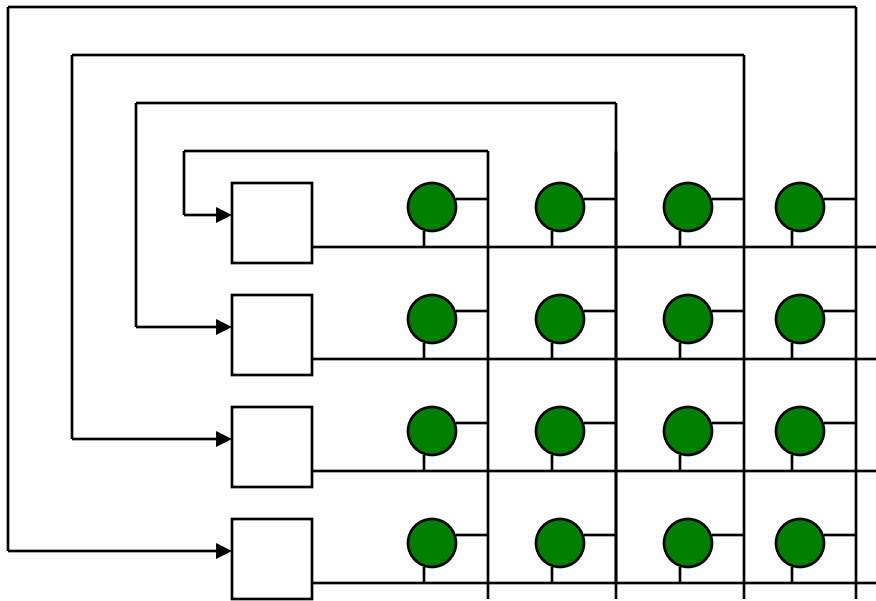
- ❑ N processors, N switches, 2 links/switch, N links
- ❑ N simultaneous transfers
 - NB = link bandwidth * N
 - BB = link bandwidth * 2
- ❑ If a link is as fast as a bus, the ring is only twice as fast as a bus in the worst case, but is N times faster in the best case

Fully Connected IN



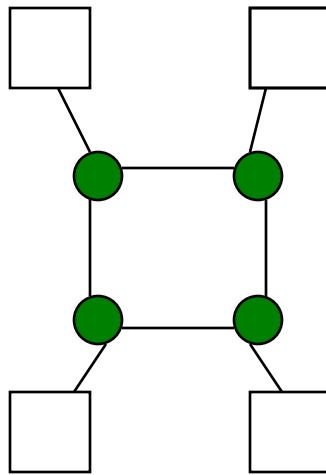
- ❑ N processors, N switches, N-1 links/switch,
 $(N*(N-1))/2$ links
- ❑ N simultaneous transfers
 - NB = link bandwidth * $(N * (N-1))/2$
 - BB = link bandwidth * $(N/2)^2$

Crossbar (Xbar) Connected IN

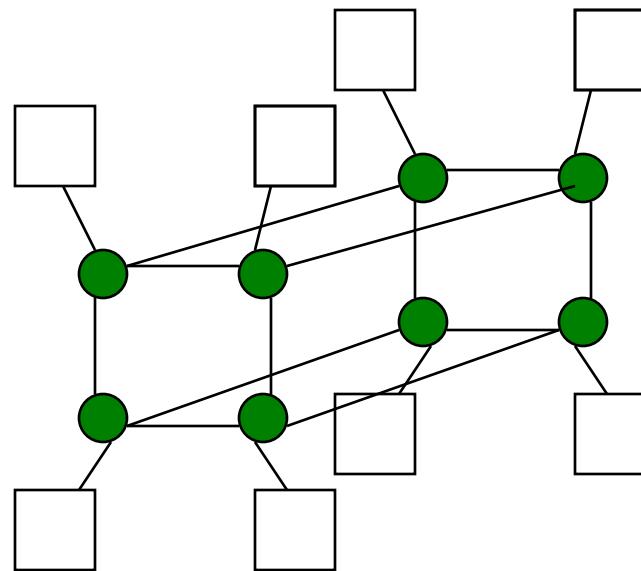


- ❑ N processors, N^2 switches (unidirectional), 2 links/switch, N^2 links
- ❑ N simultaneous transfers
 - NB = link bandwidth * N
 - BB = link bandwidth * N

Hypercube (Binary N-cube) Connected IN



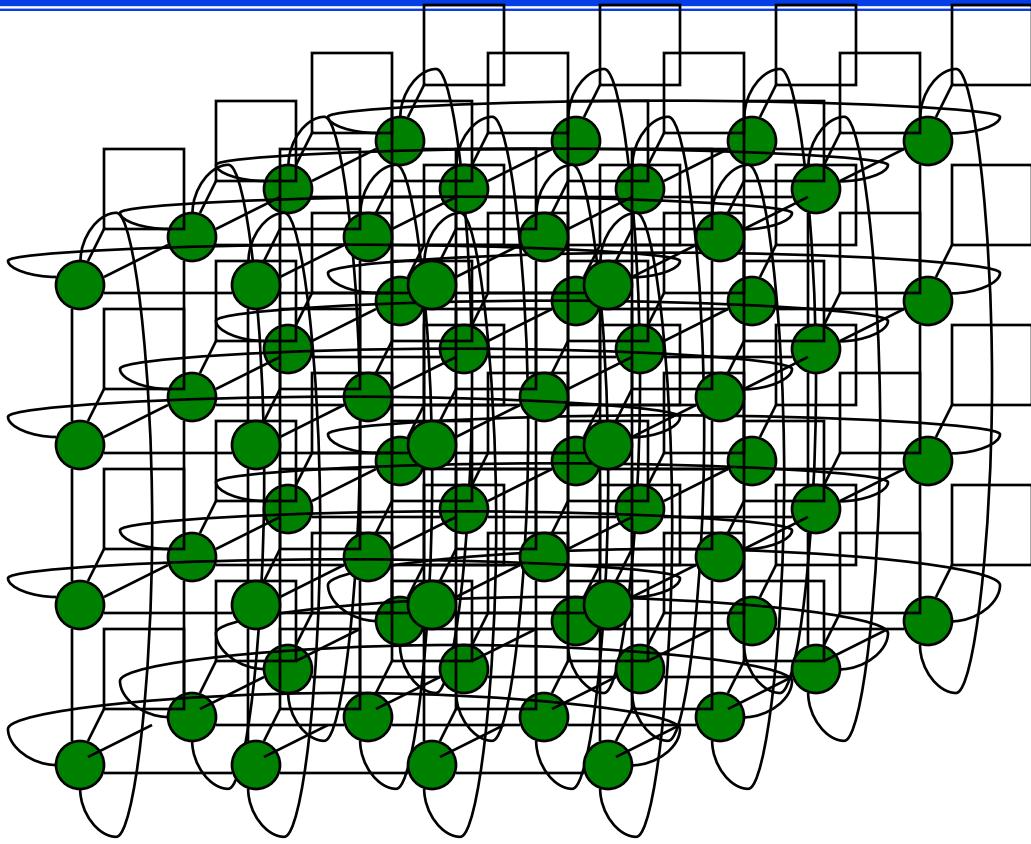
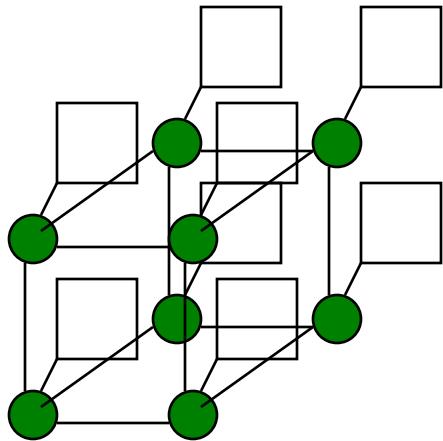
2-cube



3-cube

- ❑ N processors, N switches, $\log N$ links/switch, $(N \log N)/2$ links
- ❑ N simultaneous transfers
 - NB = link bandwidth * $(N \log N)/2$
 - BB = link bandwidth * $N/2$

2D and 3D Mesh/Torus Connected IN



- ❑ N processors, N switches, 2, 3, 4 (2D torus) or 6 (3D torus) links/switch, $4 N/2$ links or $6 N/2$ links
- ❑ N simultaneous transfers
 - $NB = \text{link bandwidth} * 4N$ or $\text{link bandwidth} * 6N$
 - $BB = \text{link bandwidth} * 2 N^{1/2}$ or $\text{link bandwidth} * 2 N^{2/3}$

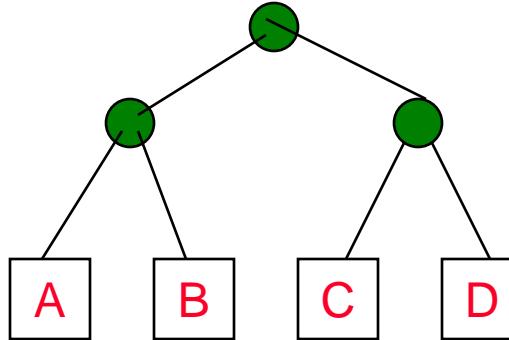
IN Comparison

- ❑ For a 64 processor system

| | Bus | Ring | 2D Torus | 6-cube | Fully connected |
|-------------------------|-----|-------|----------|--------|-----------------|
| Network bandwidth | 1 | 64 | 256 | 192 | 2016 |
| Bisection bandwidth | 1 | 2 | 16 | 32 | 1024 |
| Total # of switches | 1 | 64 | 64 | 64 | 64 |
| Links per switch | | 2+1 | 4+1 | 6+7 | 63+1 |
| Total # of links (bidi) | 1 | 64+64 | 128+64 | 192+64 | 2016+64 |

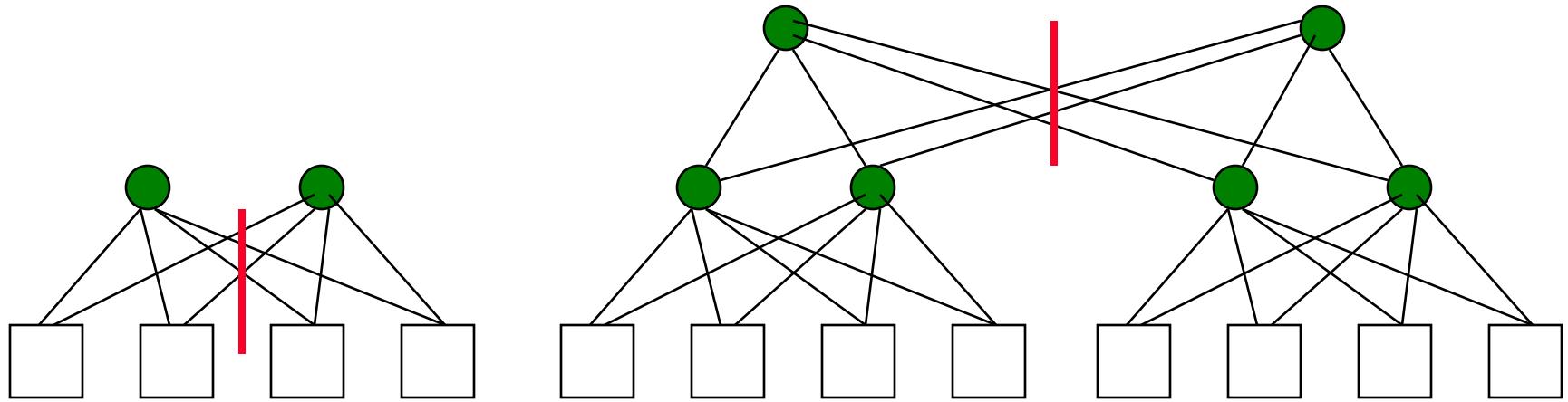
“Fat” Trees

- ❑ Trees are good structures. People in CS use them all the time. Suppose we wanted to make a tree network.



- ❑ Any time A wants to send to C, it ties up the upper links, so that B can't send to D.
 - The bisection bandwidth on a tree is horrible - 1 link, at all times
- ❑ The solution is to 'thicken' the upper links.
 - Have more links as you work towards the root of the tree increases the bisection bandwidth
- ❑ Rather than design a bunch of N-port switches, use pairs of switches

Fat Tree IN



- ❑ N processors, $\log(N-1) * \log N$ switches, 2 up + 4 down = 6 links/switch, $N * \log N$ links
- ❑ N simultaneous transfers
 - NB = link bandwidth * $N \log N$
 - BB = link bandwidth * 4

SGI NUMalink Fat Tree

01-22-03

512 Processor SN2-IFF 400MB/s/p Dual Plane Bisection Bandwidth

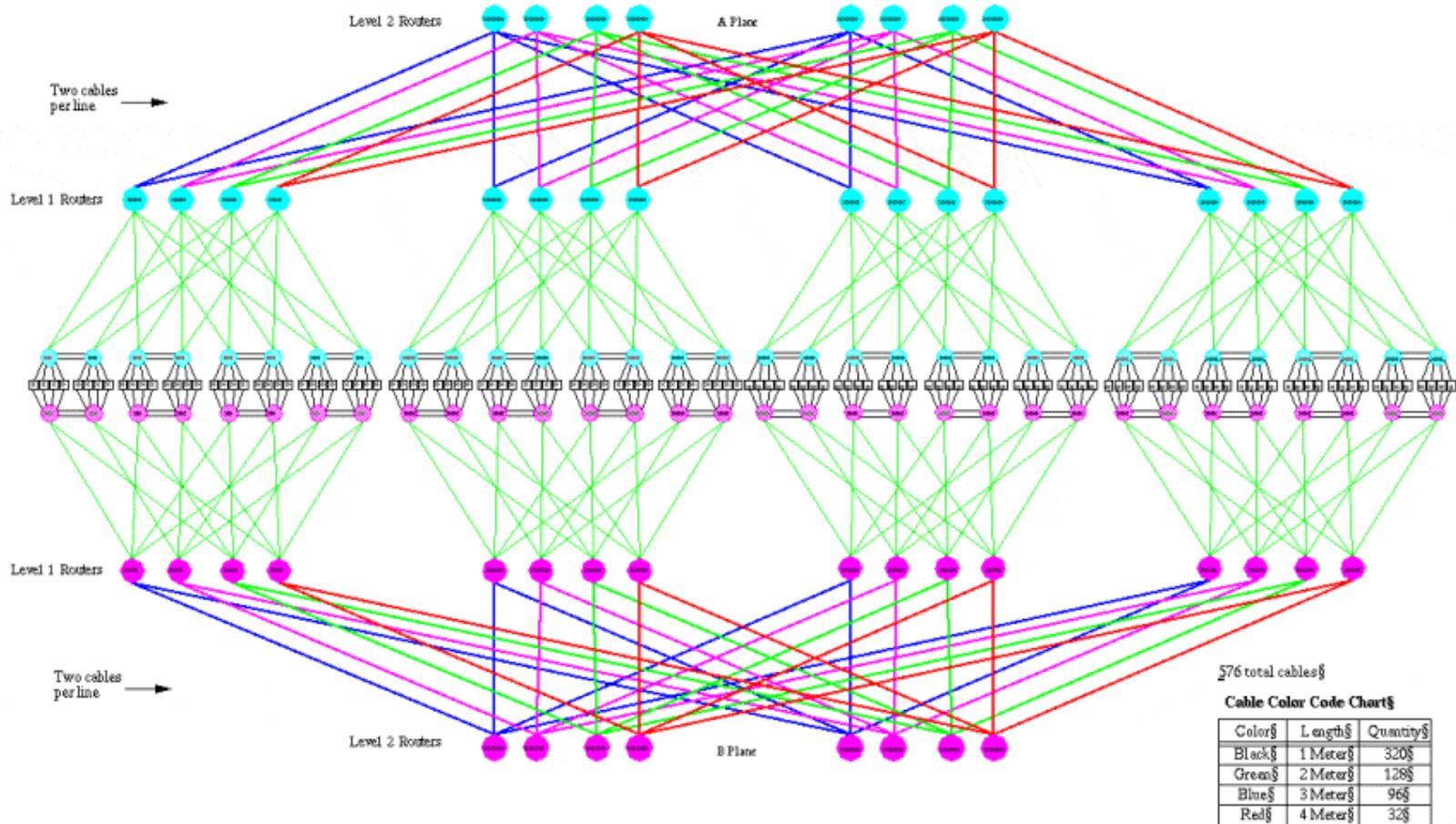


Figure 2. 512-Processor Dual “Fat-Tree” Interconnect Topology

www.embedded-computing.com/articles/woodacre

Cache Coherency in IN Connected SMPs

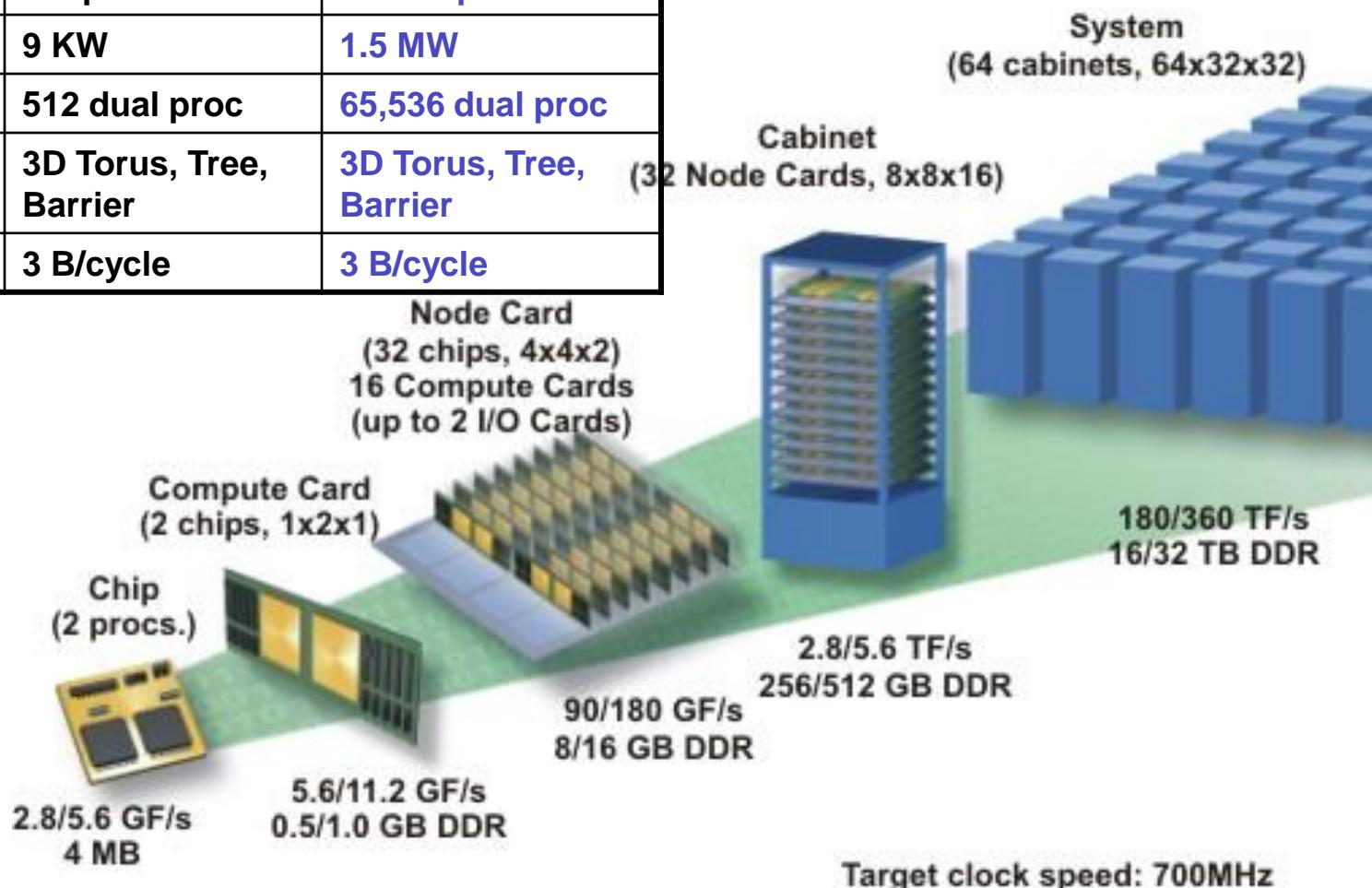
- ❑ For performance reasons we want to allow the shared data to be stored in caches
- ❑ Once again have multiple copies of the same data with the same address in different processors
 - bus snooping **won't work**, since there is no single bus on which all memory references are broadcast
- ❑ **Directory-base protocols**
 - keep a **directory** that is a repository for the state of every block in main memory (records which caches have copies, whether it is dirty, etc.)
 - directory entries can be distributed (sharing status of a block always in a **single** known location) to reduce contention
 - directory controller sends explicit commands over the IN to each processor that has a copy of the data

Network Connected Multiprocessors

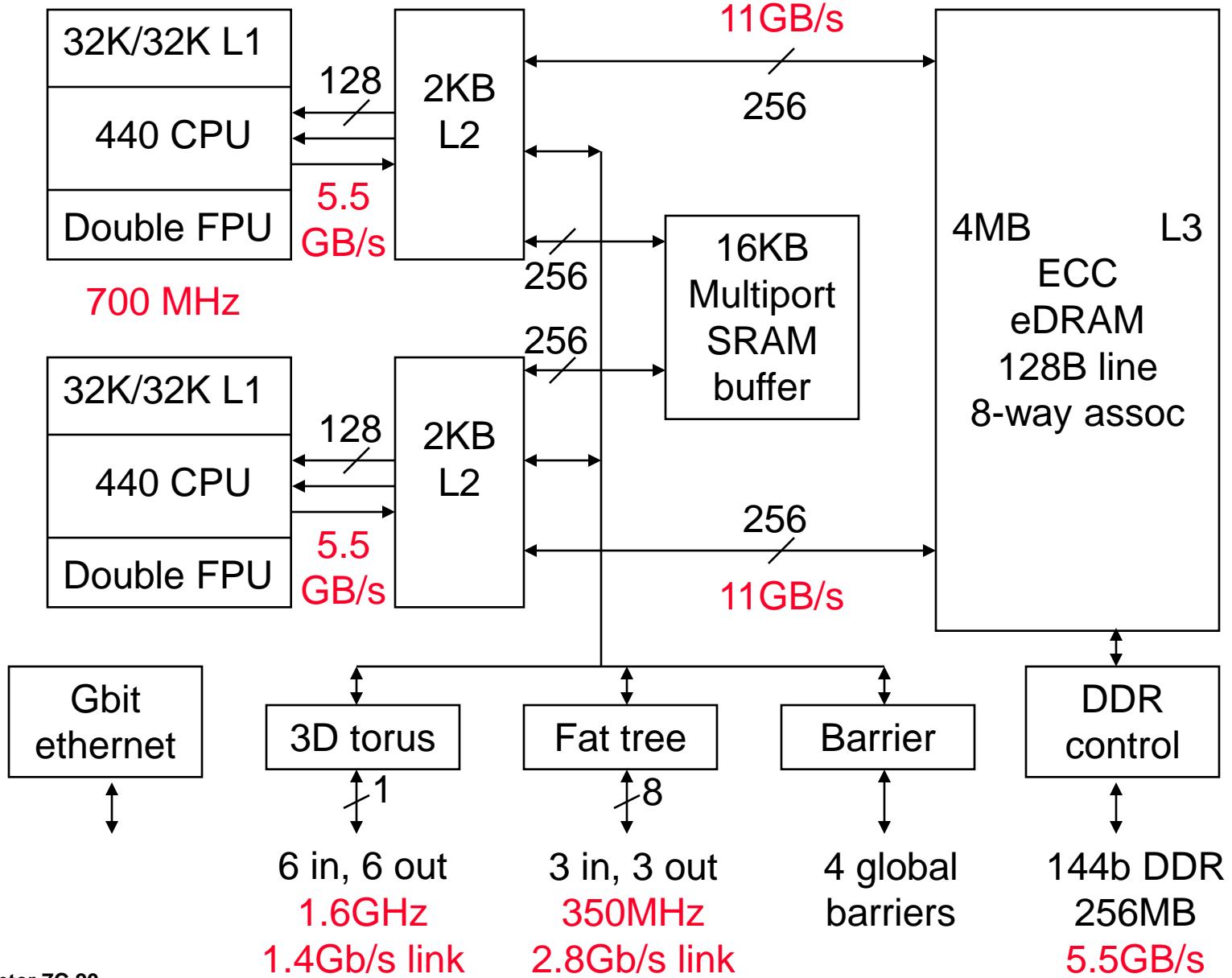
| | Proc | Proc Speed | # Proc | IN Topology | BW/link (MB/sec) |
|----------------|----------------|------------|----------|-----------------------------------|------------------|
| SGI Origin | R16000 | | 128 | fat tree | 800 |
| Cray 3TE | Alpha 21164 | 300MHz | 2,048 | 3D torus | 600 |
| Intel ASCI Red | Intel | 333MHz | 9,632 | mesh | 800 |
| IBM ASCI White | Power3 | 375MHz | 8,192 | multistage Omega | 500 |
| NEC ES | SX-5 | 500MHz | 640*8 | 640-xbar | 16000 |
| NASA Columbia | Intel Itanium2 | 1.5GHz | 512*20 | fat tree, Infiniband | |
| IBM BG/L | Power PC 440 | 0.7GHz | 65,536*2 | 3D torus, fat tree, barrier | |

IBM BlueGene

| | 512-node proto | BlueGene/L |
|--------------|-------------------------|-------------------------|
| Peak Perf | 1.0 / 2.0 TFlops/s | 180 / 360 TFlops/s |
| Memory Size | 128 GByte | 16 / 32 TByte |
| Foot Print | 9 sq feet | 2500 sq feet |
| Total Power | 9 KW | 1.5 MW |
| # Processors | 512 dual proc | 65,536 dual proc |
| Networks | 3D Torus, Tree, Barrier | 3D Torus, Tree, Barrier |
| Torus BW | 3 B/cycle | 3 B/cycle |



A BlueGene/L Chip

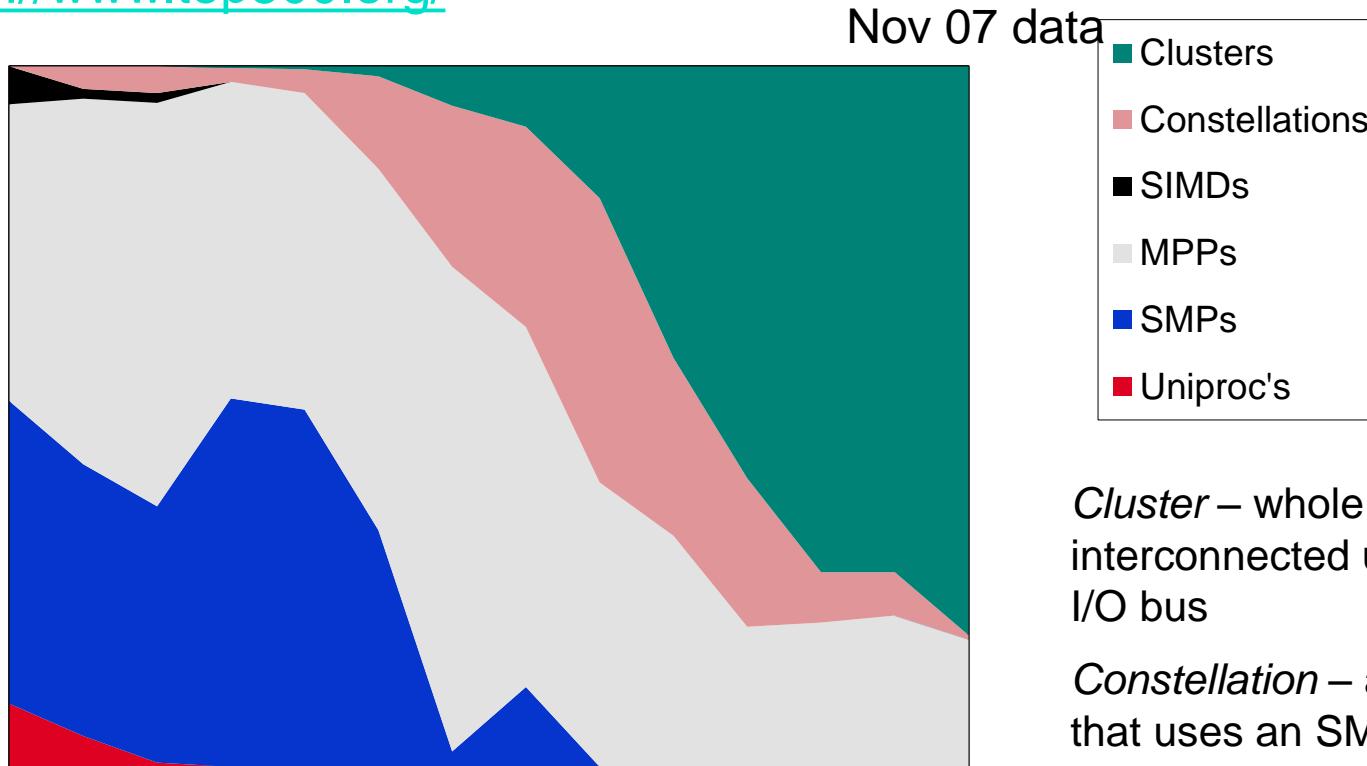


Multiprocessor Benchmarks

| | Scaling | Description |
|--------------------------------|-------------------|---|
| Linpack | Weak | Dense matrix linear algebra |
| SPECrate | Weak | Independent job parallelism |
| SPLASH 2 | Strong | Independent job parallelism (both kernels and applications, many from high-performance computing) |
| NAS Parallel | Weak | Five kernels, mostly from computational fluid dynamics |
| PARSEC | Weak | Multithreaded programs that use Pthreads and OpenMP. Nine applications and 3 kernels – 8 with data parallelism, 3 with pipelined parallelism, one unstructured |
| Berkeley Design Patterns | Strong or Weak | 13 design patterns implemented by frameworks or kernels |

Supercomputer Style Migration (Top500)

<http://www.top500.org/>



Cluster – whole computers interconnected using their I/O bus

Constellation – a cluster that uses an SMP multiprocessor as the building block

- ❑ Uniprocessors and SIMDs disappeared while Clusters and Constellations grew from 3% to 80%. Now its 98% Clusters and MPPs.

Reminders

❑ Reminders

- Assignment due in two weeks.
- Hints over the page...

❑ Next lecture

- FPGA architecture

❑ Assignment 1 marks are available online from 2pm today.

- If you want your report back, email me.

Assignment 2 Hints

- ❑ Options if you are worried about completing in time:
 - Make the cache write through.
 - Don't use write buffers.
 - It is arguable whether you need to write back the cache at the end.
 - Use an approximate least recently used overwrite strategy for set associative caches.
- ❑ If computation is taking a long time:
 - Run one cache configuration per program execution.
 - Use a compiled language. (C/C++/C#)
 - Turn off debug and turn on optimization switches (if available)
 - Bit-wise operations are your friend.
 - Use inline function calls or don't call functions within your main loop. (Remember the overhead associated with function calls.)
 - Use a fast desktop computer.

Assignment 2 Hints

- ❑ If running out of memory for cache:
 - Use dynamic memory allocation (new/malloc) and error check your pointers.

- ❑ If you are wondering what to show in your report:
 - Start with computation time for each configuration
 - Hit/Miss ratios are important

ECE4074

Computer Architecture

Semester 2 2012

FPGA Architecture

[Adapted from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK]

Introduction

❑ Primitives

- The building blocks of your custom logic

❑ Clock management

- Control clock frequencies and phase offsets

❑ Routing

- Connecting logic blocks together

❑ Clock distribution

- Minimize the effect of clock skew

❑ Programming

- How to get your design onto the FPGA

Primitives

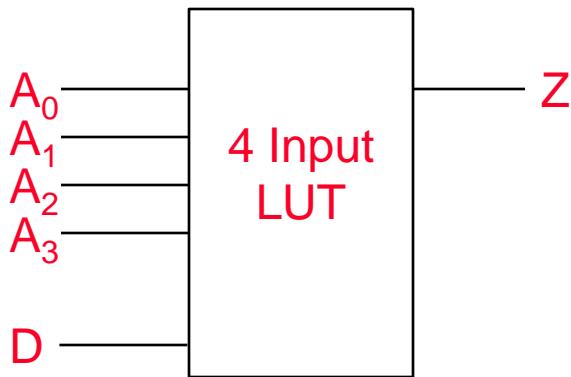
- ❑ FPGA is built out of many components
- ❑ The smallest programmer accessible components are called primitives
- ❑ They can be directly instantiated by the programmer by instancing the primitive module name.
- ❑ All modern FPGAs have some combination of:
 - Look Up Table (LUT)
 - Flip Flops/Latches
 - Logic Elements
 - High Density Configurable RAM
 - Multipliers/Multiply Accumulate Units
 - Configurable IOs
 - Clock Management Primitives

Look Up Tables

- ❑ Look up tables form the configurable logic “gates” of an FPGA.
- ❑ Generally accept a small number of inputs (4 is most common, 5 and 6 input are possible in recent generations)
- ❑ They are a 1 bit wide, 4 bit address memory block/table.
- ❑ Your 4 bit logic input is the address to “Look Up”:
 - There are 16 different addresses and by setting the value in the table for each address, any 4 bit logic function can be created.
 - For any 4 bit function, no logic minimization is needed.
 - Any 4 bit function has the same delay (approximately)

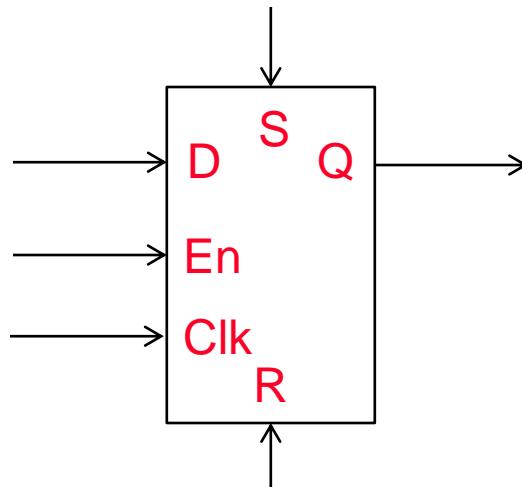
Look Up Table

- ❑ If LUT are just memory, why not also allow the programmer to use them as memory:
 - Most FPGA allow this.
 - 1 additional write input is needed.



Flip Flop

- ❑ General purpose memory elements are needed for many designs.
- ❑ FPGAs contain a configurable Flip Flop/Latch structure.
- ❑ These structures can be configured to be:
 - Level sensitive latches
 - RS latches
 - D-Flip Flops

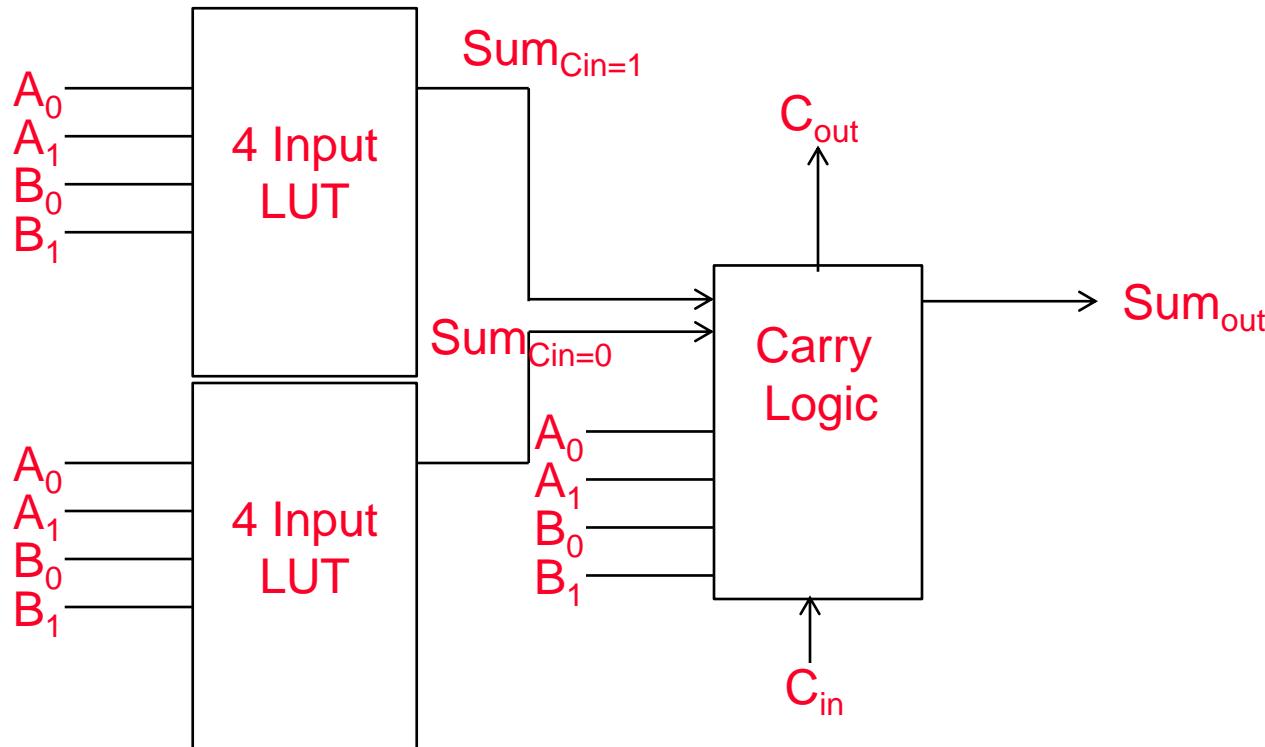


Adder Logic

- ❑ Addition is a common operator used in FPGA designs
- ❑ 4 Input LUTs would be under utilized for a full addition
 - $A + B + C_{in} = \text{Sum}$
 - $A.B + B.C_{in} + A.C_{in} = C_{out}$
 - Only 3 inputs required
- ❑ 3 input LUTs would require greater levels of logic for 4 input functions.
- ❑ For wide additions C_{out} needs to be calculated quickly to be fast.

Adder Logic

- To solve the problem of LUT utilization and slow addition, dedicated carry logic is created.
- This logic varies between vendor and generation but the general format is below (Carry look ahead)

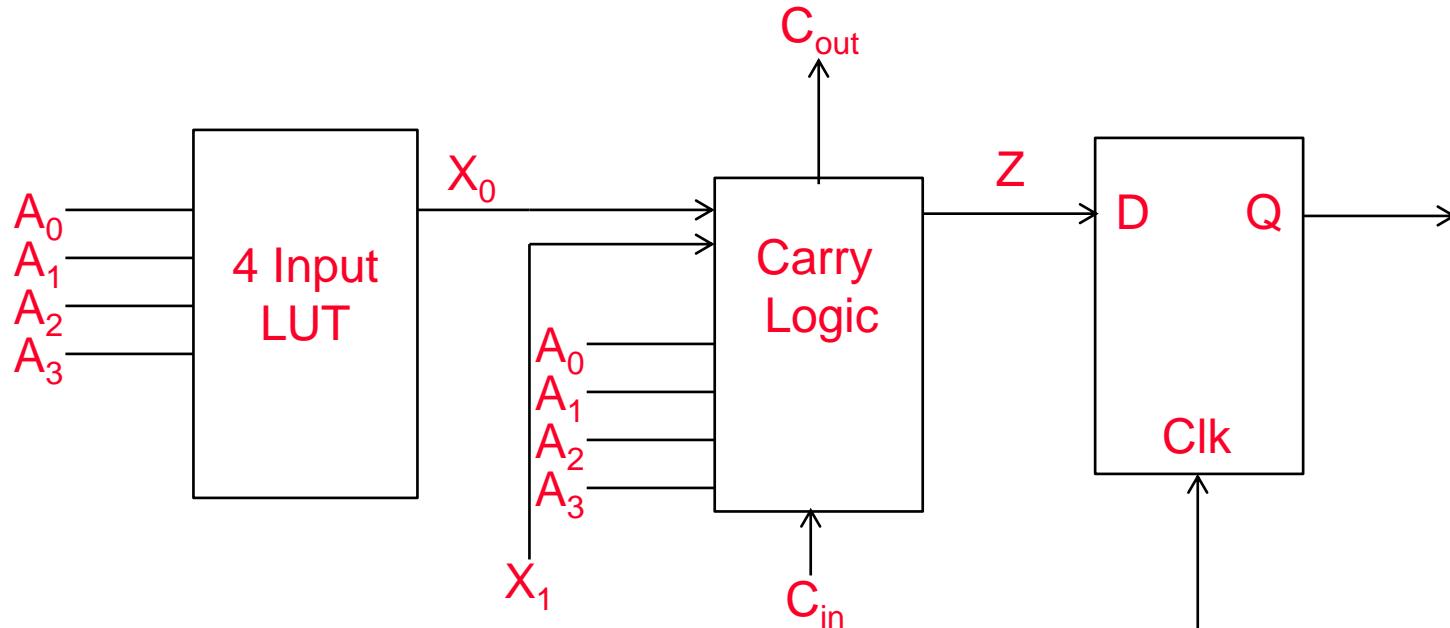


Adder Logic

- ❑ This structure varies a lot between generations and vendors.
 - Newer devices have 2 output LUTs that share inputs, not requiring a full second LUT.
 - Many devices have more inputs into the carry logic.
 - Some devices use pairs of LUTs to calculate the two possible carries, rather than dedicated logic.
 - Some devices have two or more different types of logic element on the same FPGA.
- ❑ The carry logic is always optimized to be very fast!
 - In general there is little point to designing your own adder on an FPGA.

Logic Elements

- ❑ FPGA = Field Programmable Gate **Array**
- ❑ The combination of LUTs, Flip Flops and Carry Logic create a single **element** of the array. A “Logic Element”.

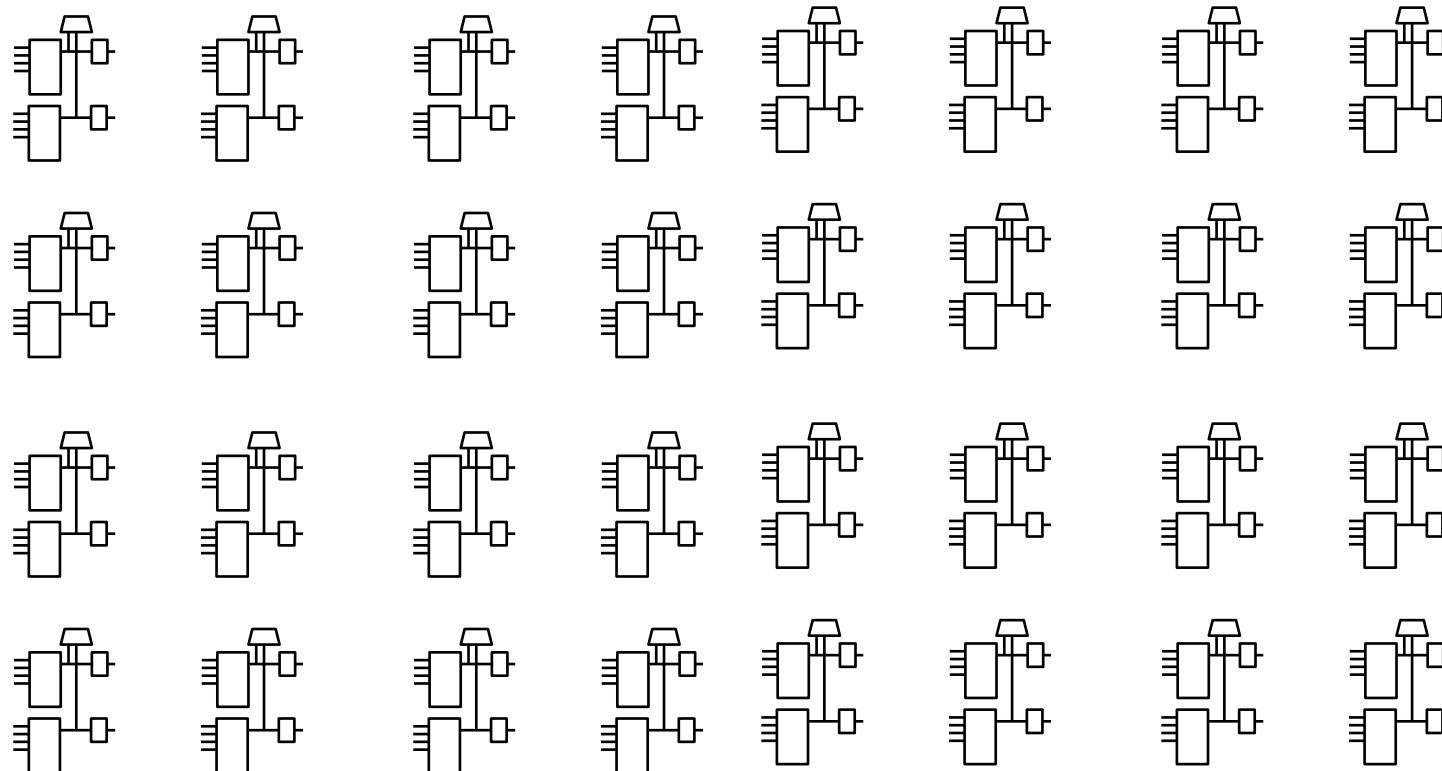


Logic Elements

- ❑ A logic element represents the smallest logic device on chip.
- ❑ It is also designed to be a pipeline stage, the fastest clock frequency possible on a device is generally dictated by the speed of a LUT and the setup time of the flip flop.
- ❑ For processing, FPGAs generally work best for heavily pipeline-able algorithms.

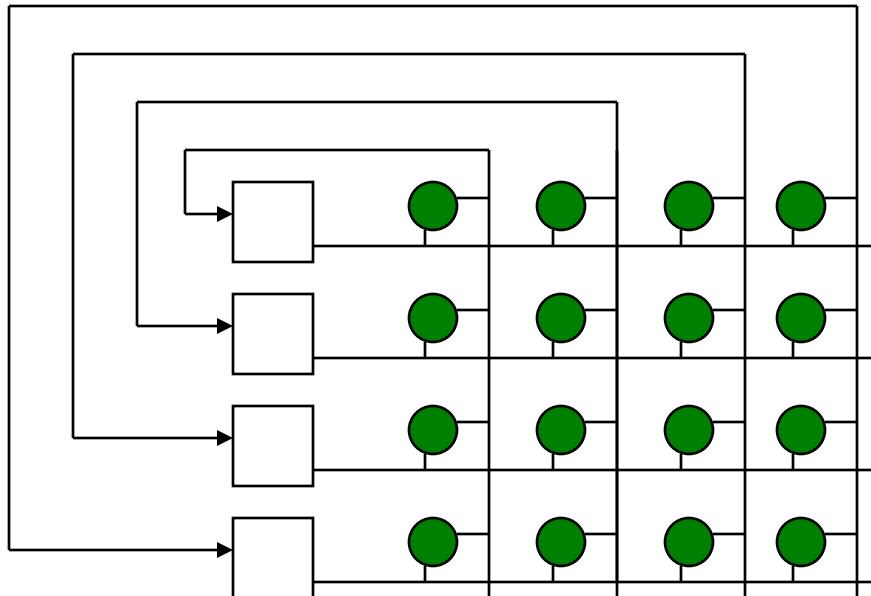
Logic Elements

- These elements are instantiated many times on an FPGA.



Routing

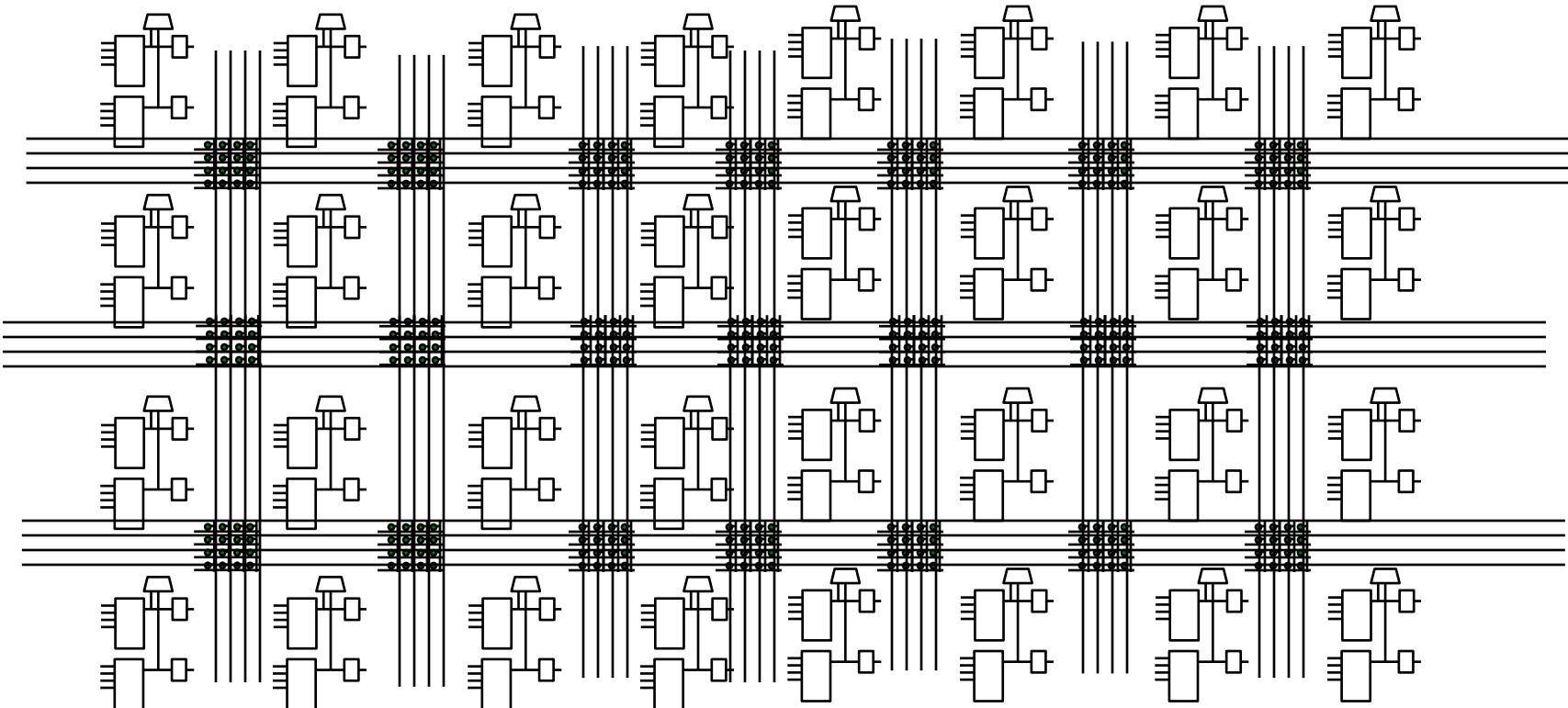
- ❑ How do you connect these together?
- ❑ Remember the crossbar switch:



- ❑ By using cross bar switches (or similar structures) many logic elements can be connected via predefined wires. Programming sets the cross bar switch connections.

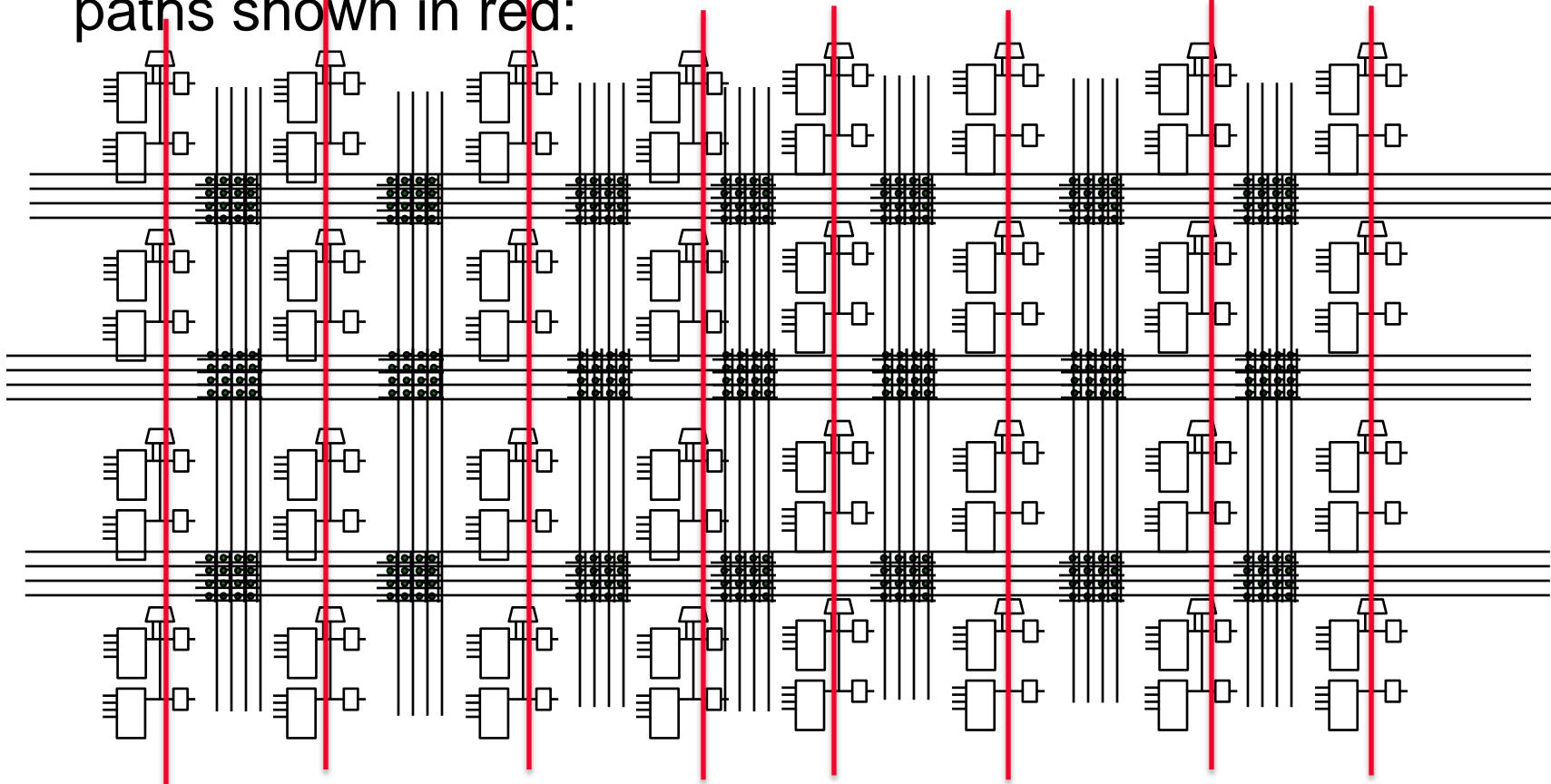
Routing

- By combining with logic element, we get the “Array”.



Routing

- The carry signals of an adder have their own dedicated paths shown in red:

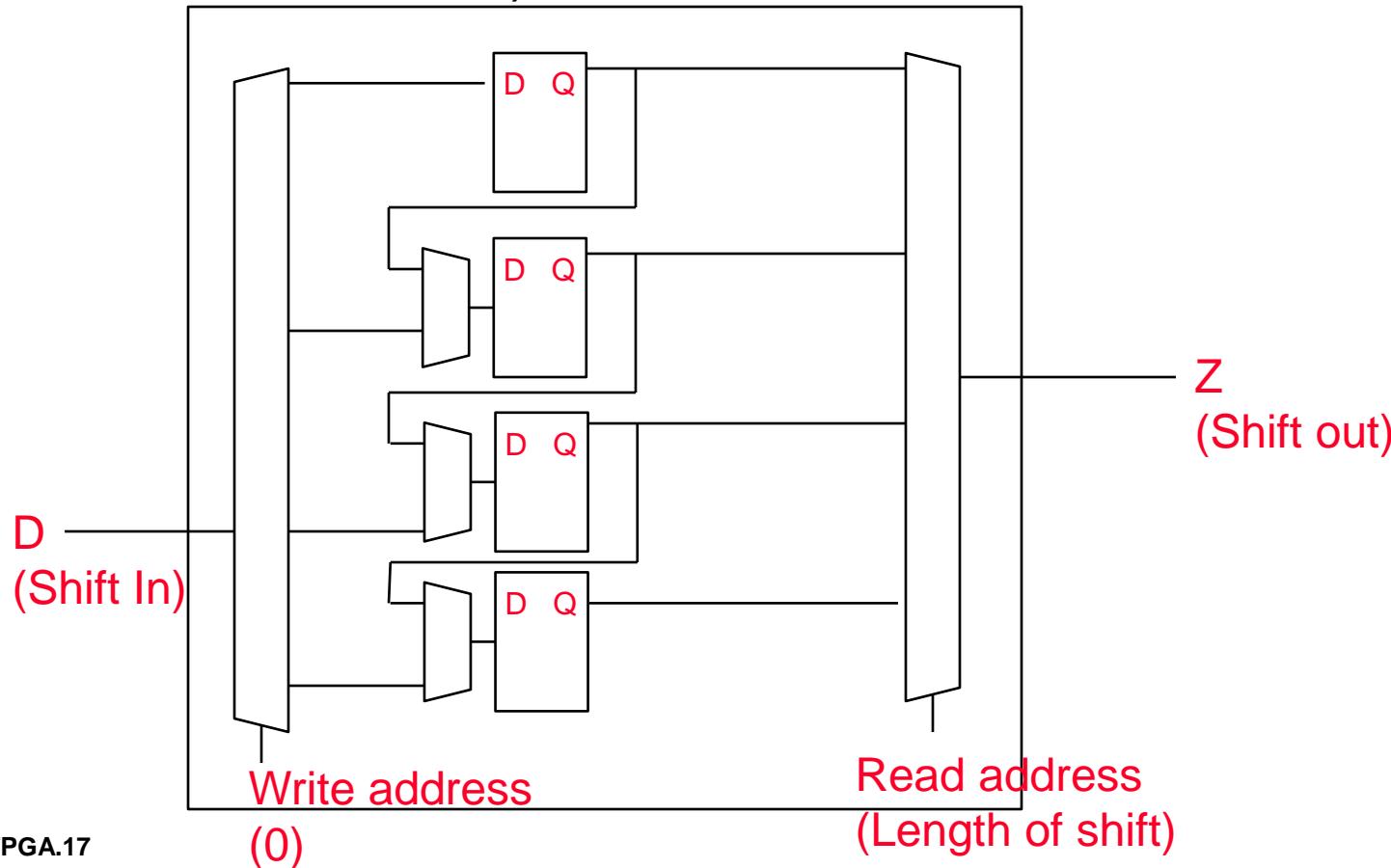


Logic Elements

- ❑ Some tasks require shift register with no logic between edge triggered flip-flops.
- ❑ To implement a 16 bit shift register using the system we have seen would require:
 - 16 LE flip flops
 - 16 cross bar switches
- ❑ Routing like this increases parasitic delays within the device.
- ❑ Larger RCs need to be driven, increasing power consumption.

Logic Element

- ❑ The solution that most FPGAs is to reconfigure the LUT as a shift register.
- ❑ This also makes programming and testing easier (which we will see later)



Multipliers

- ❑ As seen in earlier lectures, multiplying requires many additions.
- ❑ Since LEs are general purpose element they are slower than dedicated logic.
- ❑ Many modern FPGAs create dedicated multipliers so that multiplication can complete in similar delays to addition.
- ❑ LEs are on devices in the order of 10s of 1000s
- ❑ Multipliers are on devices in the order of 10s to 100s.
- ❑ Multipliers have optional pipelined stages to increase clock frequency.

High Density RAM

- ❑ Building memory out of flip flops is generally wasteful.
- ❑ A D-type flip flop is at least 3-4 times bigger than an SRAM bit.
- ❑ Memory blocks built out of higher density components are put on chip.
- ❑ Common Features:
 - Synchronous
 - Configurable initial state
 - Multi port (ie 2 read, 2 write)
 - Variable read and write width
 - Accept different widths and clocks for each port
 - High speed (close to LUT delays for 1000s of bit memories)

Configurable IOs

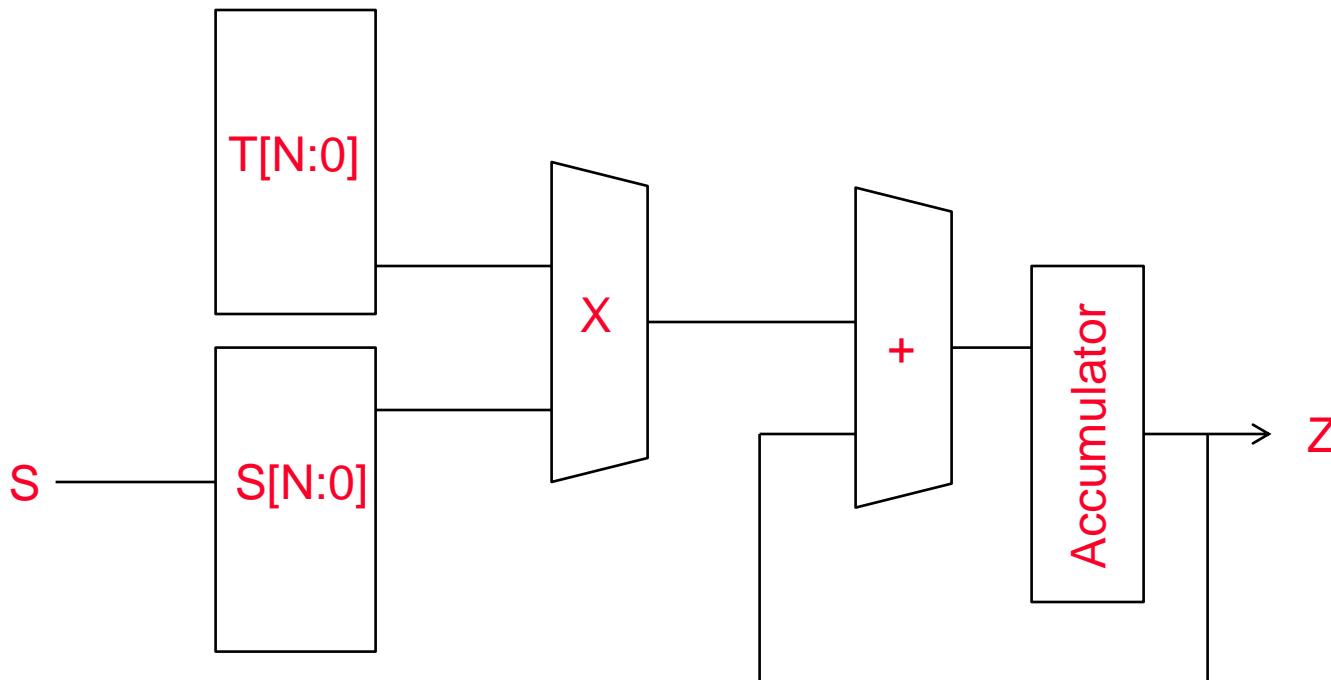
- ❑ Logic levels have many different standards
- ❑ External circuitry has many different parasitics
- ❑ Configurable IOs allow the designer to optimise for these.
 - Faster switching (harder driven) IOs can produce more noise (cross coupling) on adjacent tracks but can toggle faster.
 - Slower switching produces less noise but toggles slower.
 - Rule of thumb, choose the minimum drive needed for the operating frequency.
 - IO logic can run off different supplies (1V8, 2V5, 3V3, 5V0)

Multiply Accumulate Units

- ❑ FPGAs are commonly used for Digital Signal Processing (DSP).
- ❑ Many DSP algorithms perform the following task for each new sample:
 - $Z = T[0]*S[n] + T[1]*S[n-1] + T[0]*S[n-2] + \dots + T[n]*S[0]$
- ❑ This requires storage of both $T[n:0]$ and $S[n:0]$, $n+1$ multiplies and n adds.
- ❑ Multiply accumulate units are dedicated logic units designed to perform this task.
- ❑ Common on recent gen devices (but not all)
- ❑ Generally heavily pipelined can and can be clock faster than other logic on the FPGA.

Multiply Accumulate Unit

- ❑ Input and output rate are related by N
- ❑ Rate of S and Z are N times slower than the frequency of operation.
- ❑ Output is usually accompanied by a “Ready” signal.

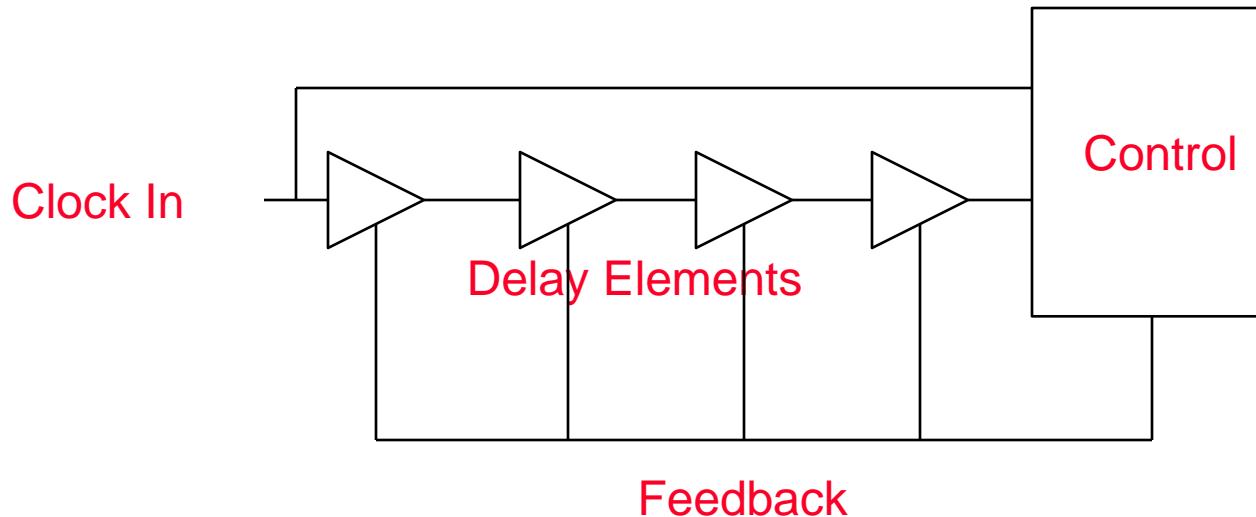


Clock management

- ❑ As FPGAs are general purpose, single clock frequency on a development board is not optimal.
- ❑ Clock management primitives allow you to increase/decrease frequencies and align clock with each other so they are in phase.
- ❑ Commonly use a phase lock loop (PLL) or delay lock loop (DLL) structure.

Phase Lock Loop

- ❑ Clock is delayed with a tuneable delay.
- ❑ When the delay matches the frequency of the clock, the PLL can lock.
- ❑ By knowing the period, the clock frequency can be multiplied.



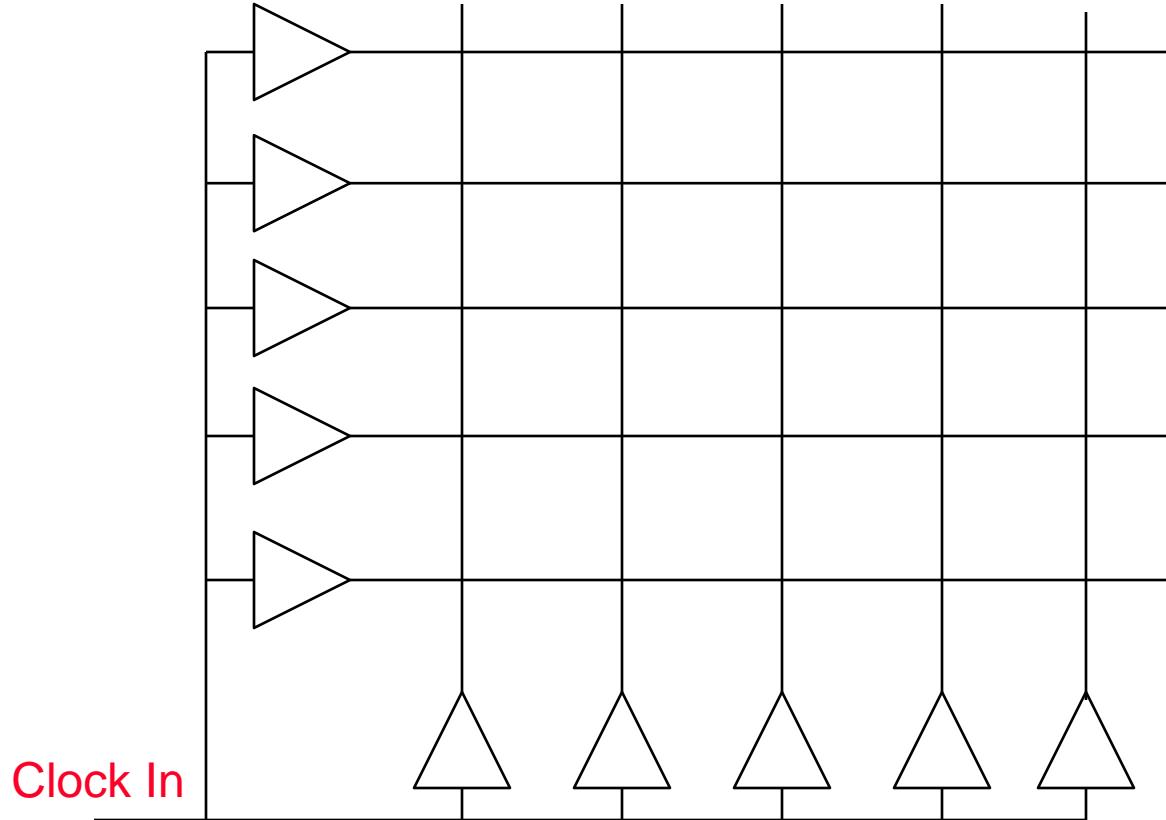
- ❑ Feedback is tuned to match the frequency.

Clock Distribution

- ❑ Logic large scale logic circuits have two main methods for clock distribution:
 - Clock Trees
 - Clock Grids
- ❑ Clock trees are small and good for most designs, but clock skew generally forms no regular pattern.
- ❑ Clock grids form low skew networks for large designs and are predictable and regular provided load is regular.
- ❑ Grids take longer to design and simulate (full spice simulation of the clock network), but once created and characterized are very predictable.
- ❑ Input characteristics (drive and slew) need to be known accurately, which is why custom logic doesn't drive clock.

Clock Distribution

- ❑ Clock grids are big and power hungry, usually only a small number on chip (2-8)



Programming

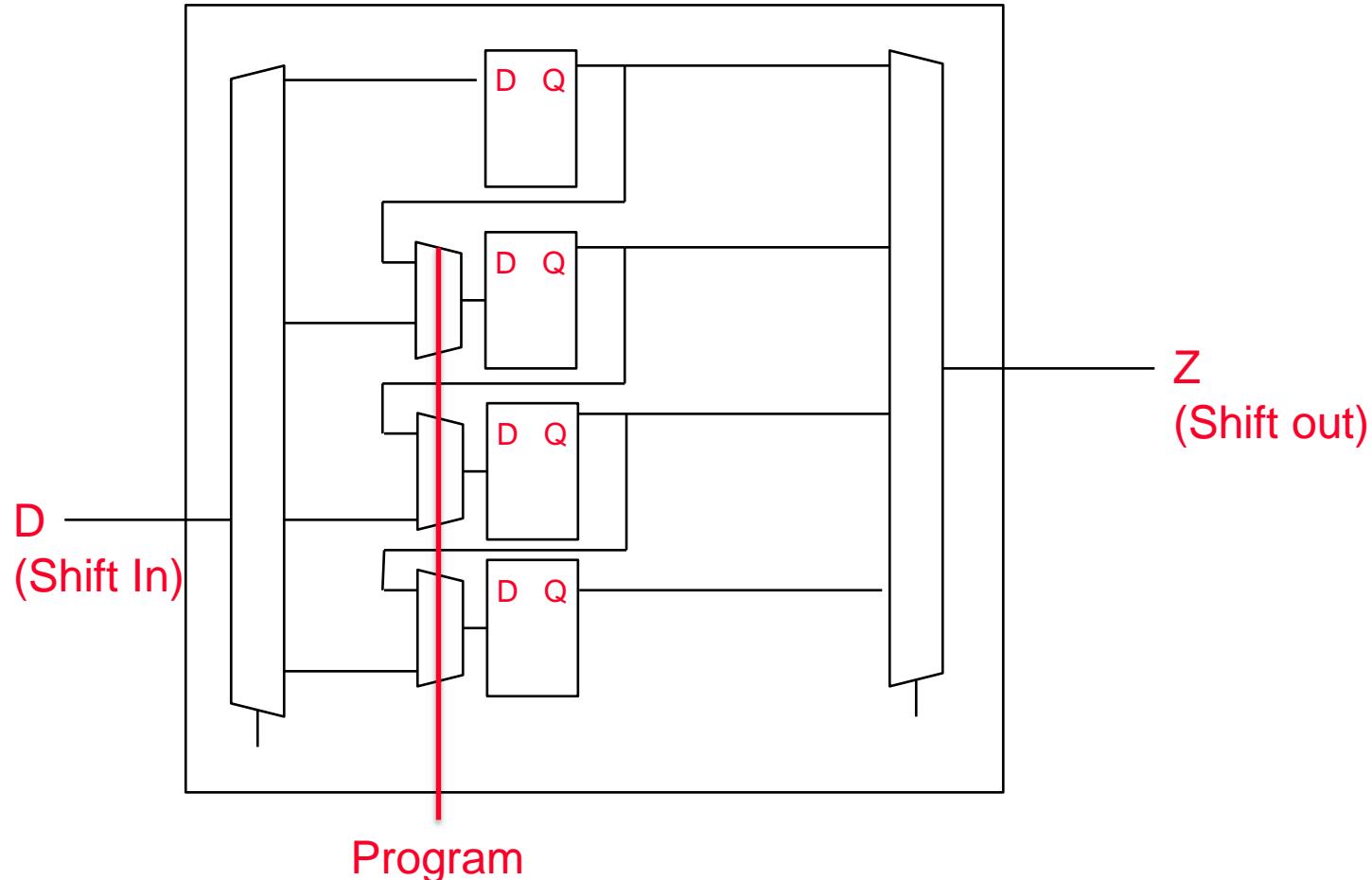
- ❑ Parallel programming of LUTs would require far too many wires.
- ❑ We also don't want to dedicate too many extra resources to programming if possible.
- ❑ Programming can take time, if doesn't have to happen fast.
- ❑ Solution: Serial programming via “Boundary Scan”

Programming

- ❑ Boundary scan is a method used to test logic circuits.
- ❑ It involves adding multiplexors to each flip flop's "D" input to create a "scan chain"
- ❑ The multiplexor is either set to receive its normal logic function input, or the output of the previous flip flop in the "chain".
- ❑ By enabling this "chain" the device becomes one big shift register, whereby each flip flop can be individually set.
- ❑ Since LUTs are already able to be configured as shift registers, little extra hardware needs to be added.

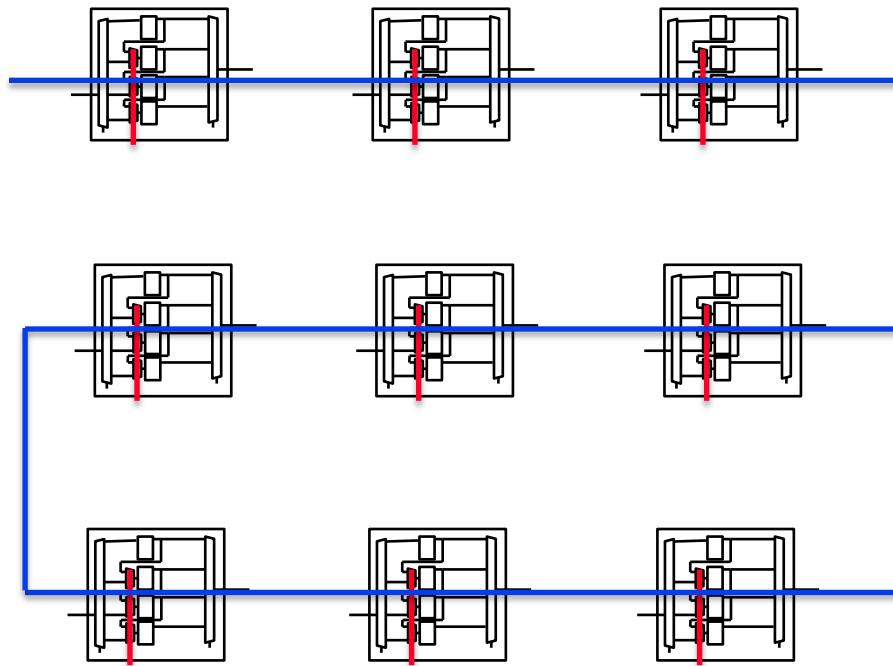
Programming

- By setting “Program” to 1 the chain is formed. Each LUT then feeds into the next.



Programming

- ❑ The path of the chain is shown in blue



Programming

- ❑ Latches, RAM and routing switches can be configured in the same way. Though it's a bit more complicated.
- ❑ Boundary Scan hardware is already in 99% of devices for circuit testing purposes. Adds about 5% extra logic to the device in exchange for unrestricted access to all memory bits. It can also be used to read out bits.
- ❑ Only requires 3 pins (and two can usually double as normal functional pins already in the design)

Reminders

- ❑ No more lectures until the last week of semester.
- ❑ Assignment 3 will be released mid this week (probably Wednesday night).
- ❑ Exam:
 - No calculators
 - 3 hours
 - 100 marks (A good student should be able to complete the exam in approximately 100 mins)
 - There are multiple questions for EVERY lecture
 - Make sure you review your notes and slides