



Complex instruction set computers evolved over time

The desire was to close the “semantic gap”

the difference in the way operations are specified in an expressive high-level language such as Java or C++ and their hardware implementation

more sophisticated instructions were included in the instruction set such as:

- string search, string compare, string translation
- automating looping

An instruction that just adds two integers to produce a sum is considered a simple instruction

One that copies an element from one array to another and automatically updates both array subscripts is considered a complex instruction

CISC systems have intricate addressing modes to ease processing high level data structures

A goal was to simplify high level instruction translation by making the CISC machine instructions more closely resemble the high level functions

Microprogramming provides these more complex features
microinstructions carry out the many intricate steps
easier to implement than hardwired logic

A machine instruction is interpreted by a microprogram
may be as many as a dozen or more microinstructions
they direct hardware in performing the required actions

Microprograms are stored in an internal control memory
the same chip as the CPU
take additional time to process compared to hardwired logic

The use of these complex features reached a peak during the 1970s and early 1980s.

Processors of that era following this design approach included:

- the Intel 8086, 80286, 80386 and 80486
- Motorola 68K series
- Digital Equipment (DEC) VAX processors

Memory was very expensive and relatively small in capacity
Using complex instructions made programs smaller

CISC type instructions may use multiple memory operands
each operand may be referenced using a different addressing mode

The instructions support a large and flexible set of operations

They can vary in size from one byte to 15 or more bytes

They must be partly decoded to know how many additional bytes makeup the instruction

This prevents pre-fetching instructions to speed processing

x86 CISC type instructions include:

scas (string scan) - scans a block of memory looking for a match to the contents of a register and automatically updates counter and pointers.

xlat (translate) – performs a table lookup to convert the contents of a register to a number stored in a memory table.

loop – decrements and tests a counter and jumps based on the outcome

VAX CISC type instructions include:

PUSHR #^M<R0, R1, R2, R3, R4, R5>

- which saves a series of registers on the stack based on a specified bit mask.

POPR #^M<R0, R1, R2, R3, R4, R5>

- which restores a series of registers from the stack based on a specified bit mask.

These are functions that would be performed as part of a call to a procedure or in returning from a procedure

A Motorola 68K CISC type instruction is:

CAS (compare and swap) - which compares the value in a memory location with the value in a data register, and copies a second data register into the memory location if the compared values are equal, otherwise copies the contents of the memory location into the first register



The instructions are difficult to implement in hardware and require microprogramming

microcode slows down the system – after fetching each machine instruction, a series of microinstructions must be retrieved and executed to interpret the machine instruction

Compiler writers often avoid using CISC instructions that are unique to one machine in favor of generating groups of more standard simple instructions to perform the same task



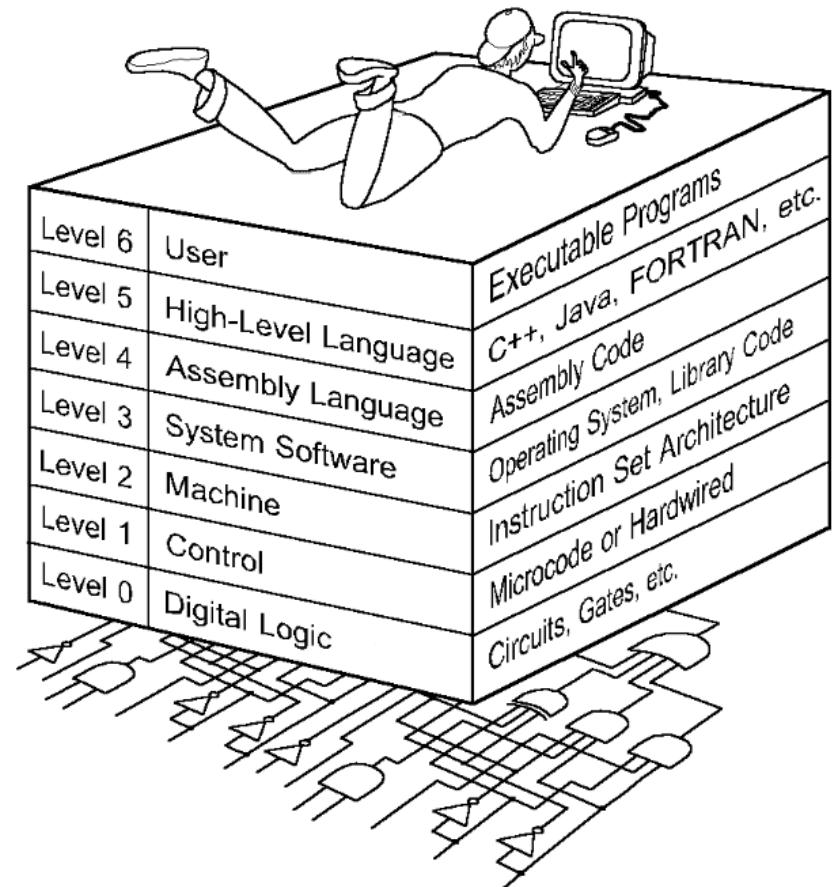
Intricate addressing modes and variable length instructions require a more complex and thus slower control unit

Allowing most instructions to reference memory operands makes the instruction take longer to execute

Complex instructions make pipelining more difficult

- Computers can be viewed at different levels
 - Each layer corresponds to a “*virtual machines*”
 - Each layer provides services to the level above
 - Each layer abstracts away the details of the level below
- “Programs” at each layer can be:
 - translated into the form of the next lower level
 - interpreted by a program at the next lower

- Each virtual machine layer is an abstraction of the level below it.
- The machines at each level execute their own particular instructions, calling upon machines at lower levels to perform tasks as required.
- Computer circuits ultimately carry out the work.



- **Level 4: Assembly Language Level**
 - Acts upon assembly language produced from Level 5, as well as instructions programmed directly at this level.
- **Level 3: System Software Level**
 - Controls executing processes on the system.
 - Protects system resources.
 - Assembly language instructions often pass through Level 3 without modification.



- Level 2: Machine Level
 - Also known as the Instruction Set Architecture (ISA) Level.
 - Consists of instructions that are particular to the architecture of the machine.
 - Programs written in machine language need no compilers, interpreters, or assemblers.



- Level 1: Control Level
 - A *control unit* decodes and executes instructions and moves data through the system.
 - Control units can be *microprogrammed* or *hardwired*.
 - A microprogram is a program written in a low-level language that is implemented by the hardware.
 - Hardwired control units consist of hardware that directly executes machine instructions.



- **Level 0: Digital Logic Level**
 - This level is where we find digital circuits (the chips).
 - Digital circuits consist of gates and wires.
 - These components implement the mathematical logic of all other levels.



Overview of the MIPS Architecture

This course is based on the MIPS R3000 processor as an example of a RISC System



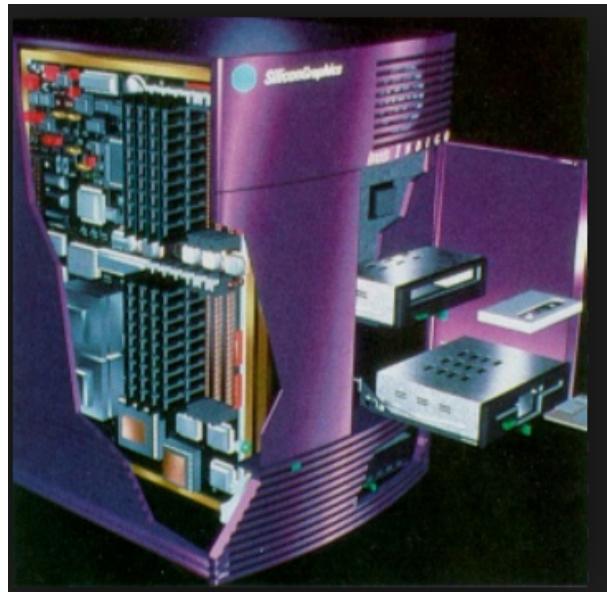
Sample applications of MIPS processors

MIPS processors are commonly found in:

- Set-top boxes
- Game consoles
- VoIP SoCs
- Digital TVs and DVD recorders/players
- Workstations



Sample applications of MIPS processors

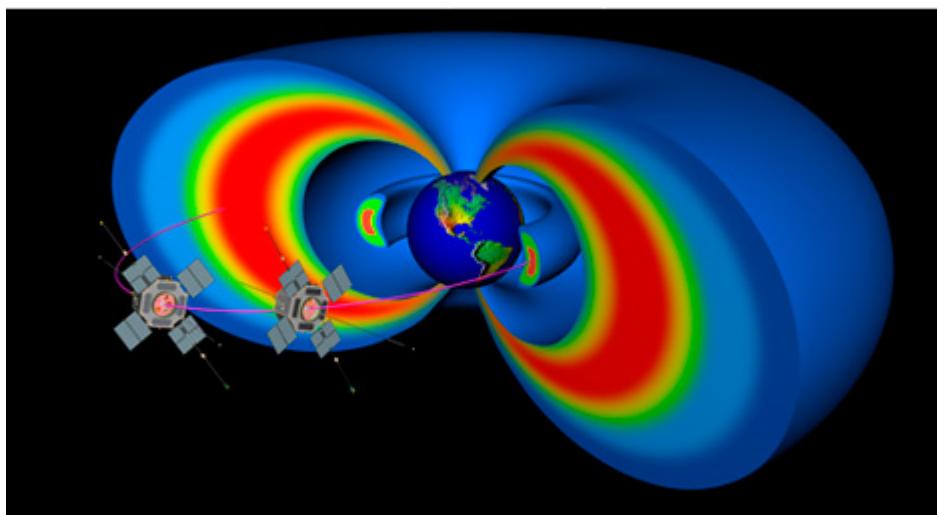
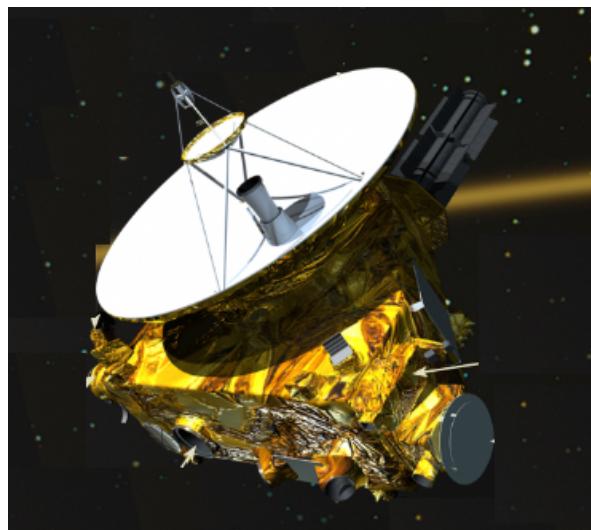


Silicon Graphics (SGI) Workstations were based on MIPS R4000 processors, as were Nintendo 64 game consoles.

The R4000 was the first commercially available 64-bit microprocessor.



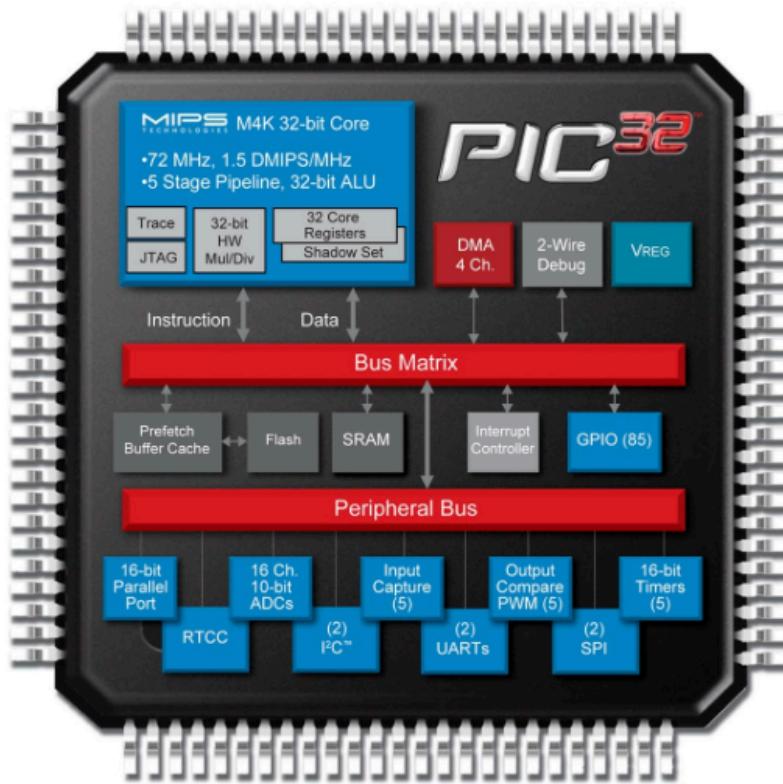
Sample applications of MIPS processors



MIPS processors control the subsystems on the New Horizons Spacecraft to Pluto and were used to implement Software Defined Radios on the twin Van Allen Probes.



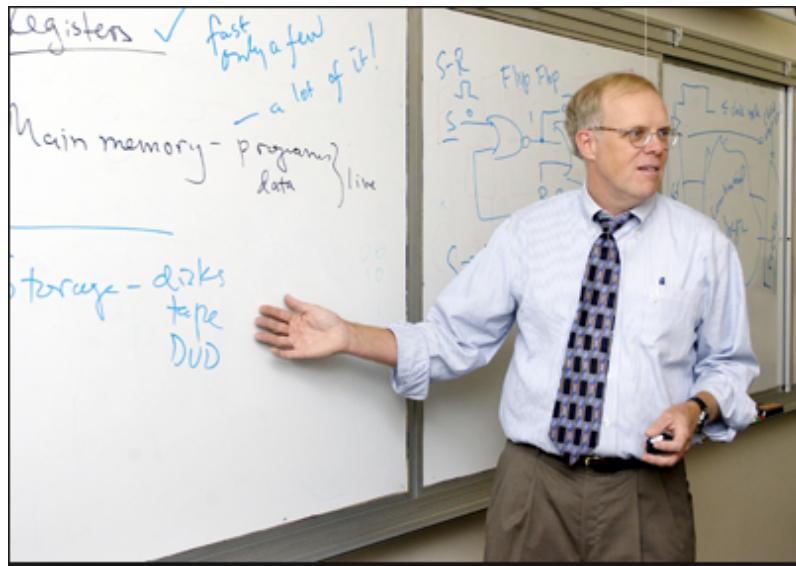
MIPS targets high-performance embedded designs



Used in digital consumer products and has a growing presence in mobile devices



MIPS has become a standard and performance leader in the embedded industry.



The design of the original MIPS processor was developed by Stanford University professor of engineering John Hennessy



The MIPS R3000 processor is a 32-bit machine with the following RISC-like features:

- Fixed size machine instructions (32 bits)
- Only load and store instructions can reference memory (load/store architecture)
- Uses 32-bit addresses
- Pipelined functional units

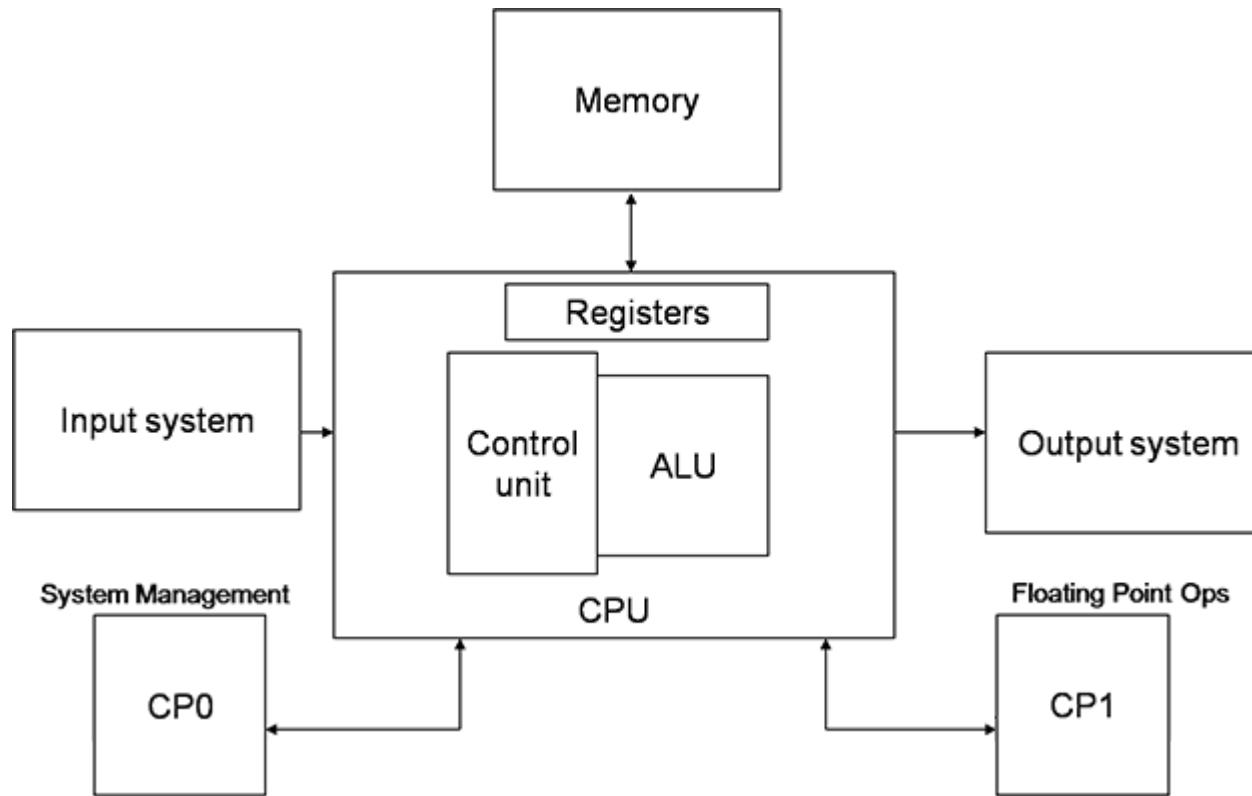


MIPS RISC-like features:

- only three formats for machine instructions, all of which are 32 bits
- only four addressing modes
- 32-bit registers (\$0, \$1, ... , \$31, Hi and Lo)



MIPS Organization





MIPS Registers

- CPU contains 32 registers, each 32 bits wide
- Registers may be used for general purposes
- MIPS calling convention should be adhered to if the user program is to be compatible with library routines and system software.



- Either register names or numbers may be used within assembly language statements
- Format: \$n, where n is the register number (0 – 31)
- Register numbers are encoded in 5-bit fields within machine instructions
- Register names serve as more mnemonic aliases for the register numbers within assembly language statements



MIPS Registers

- Register \$0 (\$zero) is hardwired to zero
- (reads as 0 and cannot be overwritten)
- Serves a convenient source or a zero constant

Example use: add \$3,\$2,\$0 has same effect as
move \$3,\$2 to copy \$2 into \$3

- Register \$1 (\$at) is used by the assembler to implement pseudo-instructions (i.e. synthetic instructions)



MIPS Registers

- Temporary registers are not preserved across function calls:
 - \$t0 - \$t7 (register numbers 8 – 15)
 - \$t8 - \$t9 (register numbers 24 – 25)
- Saved registers must be saved and restored if used within a function:
 - \$s0 - \$s7 (register numbers 16 – 23)



Role of Registers in Procedure Calling

Steps required:

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's function
5. Place result in register for caller
6. Return to place of call



Role of Registers in Procedure Calling

\$a0 – \$a3: arguments (reg's 4 – 7)

\$v0, \$v1: result values (reg's 2 and 3)

\$t0 – \$t9: temporaries (can be overwritten by callee)

\$s0 – \$s7: Must be saved/restored by callee

\$gp: global pointer for static data (reg 28)

\$sp: stack pointer (reg 29)

\$fp: frame pointer (reg 30)

\$ra: return address (reg 31)



MIPS Instruction Types

- R-type employs register operands
- I-type contains a literal (immediate value)
- J-type contains part of the jump address



MIPS Instruction Summary

Instruction type	Examples
arithmetic	add, sub, addu, subu, mult, multu, div, divu
Logical	and, andi, or, ori, xor, xori, nor, sll, srl, sra, sllv, sriv, srav
Data transfer	lw, lh, lb, sw, sh, sb, lui, mfhi, mflo, mthi, mtlo
Conditional branch	beq, bne, bltz, blez, bgtz, bgez
Unconditional branch	b, j, jr, jal
Comparison	slt, slti, sltu, sltiu



Addressing Modes

The MIPS only supports the following addressing modes:

1. Register mode (operands in registers)
2. Immediate mode (literal contained in instruction)
3. Base relative mode (offset + contents of base register give the memory address of the operand)



4. PC-relative: $\text{PC} + (4 * \text{displacement})$ gives the branch target address)
5. Pseudo-direct (rightmost 26 bits with machine instruction is multiplied by 4 and concatenated with upper 4 bits of PC to yield the jump address)



- The 16-bit immediate values are sign-extended to 32 bits by the arithmetic, load/store and branch instructions.
- The logical instructions (e.g. AND, OR, etc.) generate the 32-bit immediate operand by zero-extending the 16-bit immediate field contained in the machine instruction.



Addressing Mode Summary

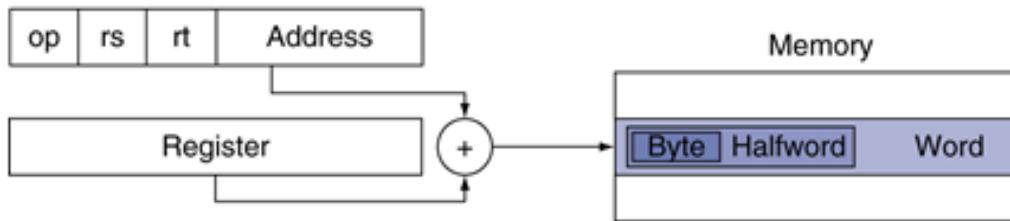
1. Immediate addressing



2. Register addressing



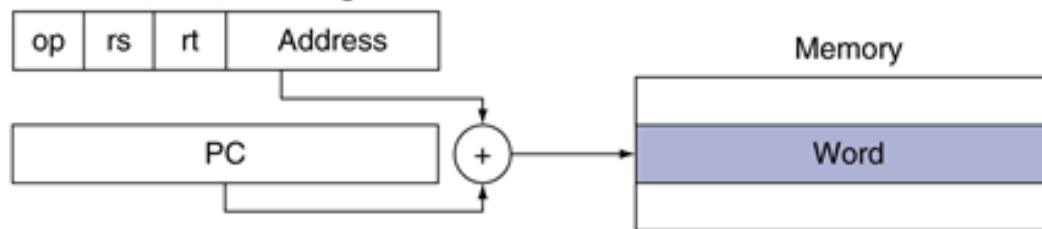
3. Base addressing



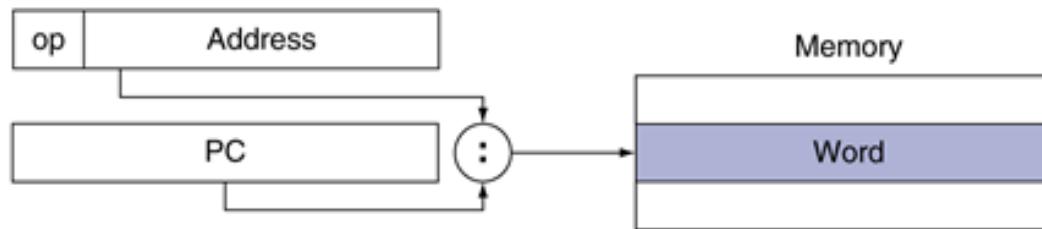


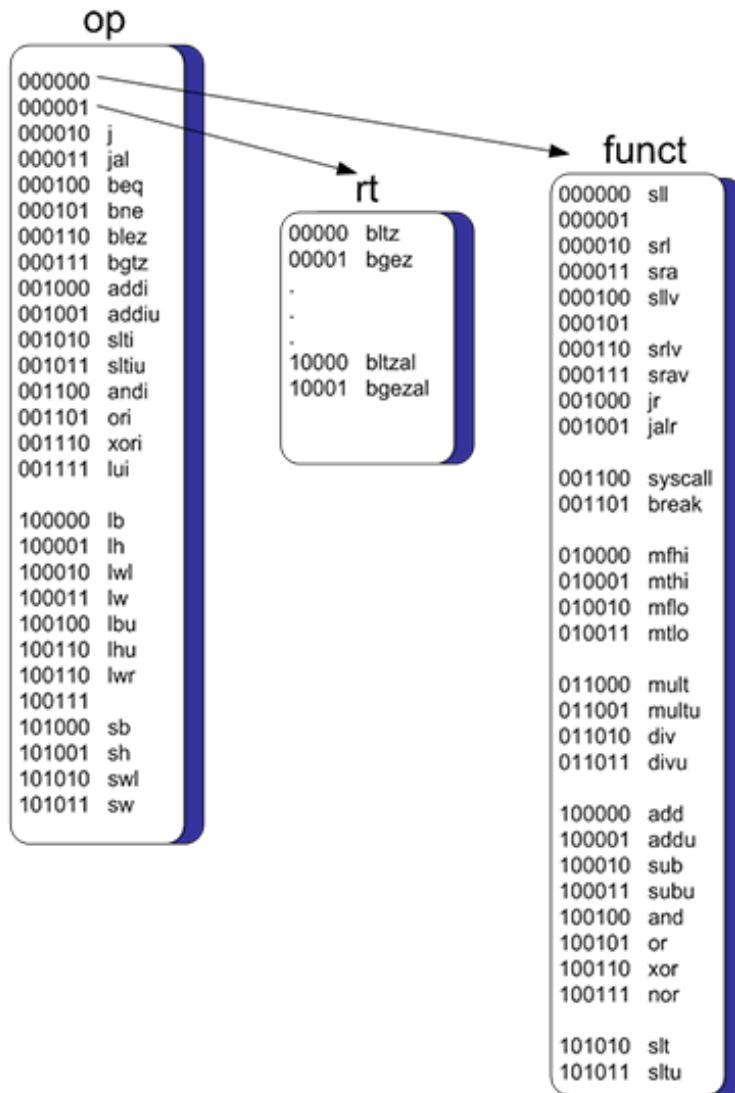
Addressing Mode Summary

4. PC-relative addressing



5. Pseudodirect addressing





Their fixed size and regularity make it easier and faster for the control unit to interpret the machine instructions.

The opcode is always the leftmost 6 bits.

When the opcode is 0, the operation is determined by rt field.

When the opcode is 1, the operation is determined by function code field in the rightmost 6 bits.



Load/Store Architecture

- Compilers that generate code for RISC processors try to maximize and optimize the use of registers
- Only the load and store instructions can access memory operands
- Registers are faster to access than memory
- Less frequently used variables are spilled to memory
- This improves performance and makes the common case fast

Programs are easier to write in high level languages

- examples are C, C++ and Java

Compilers must translate high level language programs

- target programs can be assembly language
- or machine code

Assemblers generate machine code from assembly programs

- which are symbolic representations of machine code

Processors can only execute machine code

The *execution cycle* specifies how instructions are carried out

■ High-level language

- Level of abstraction closer to problem domain
 - Provides for productivity and portability

■ Assembly language

- ## ■ Textual representation of instructions

■ Hardware representation

- Binary digits (bits)
 - Encoded instructions and data

High-level
language
program
(in C)

Assembly language program (for MIPS)

Binary machine
language
program
(for MIPS)

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

Compiler

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

An oval-shaped node with a blue gradient fill and a dark blue border. The word "Assembler" is written in white, bold, sans-serif font inside the node. A vertical black arrow points downwards from the top center of the node to a vertical black arrow pointing downwards at the bottom center.

```
000000000101000010000000000000011000  
0000000000001100000011000000100001  
100011000110001000000000000000000000  
1000110011110010000000000000000000100  
10101100111100100000000000000000000000  
10101100011000100000000000000000000100  
0000001111100000000000000000000000001000
```

The execution cycle consists of one or more steps

Each step requires a clock cycle

cycle time is the duration of a clock cycle (clock period)

Instructions that take fewer cycles execute faster

The shorter the cycle time the faster the execution

The clock rate is the number of cycles per second (Hz)

Clock rate = $1/\text{cycle time}$

Simple RISC type instructions take fewer cycles

CISC type instructions tend to take more cycles

Programs that contain fewer instructions are faster

Programs that use mainly simple instructions are faster

Ways to improve performance include using:

Efficient algorithms

Efficient fast hardware

Fast memory

Parallel operations

Metrics are needed to access performance improvements

- Response time
 - How long it takes to do a task
- Throughput
 - Total work done per unit time
 - e.g., tasks/transactions/... per hour
- How are response time and throughput affected by
 - Replacing the processor with a faster version?
 - Adding more processors?
- We'll focus on response time for now...

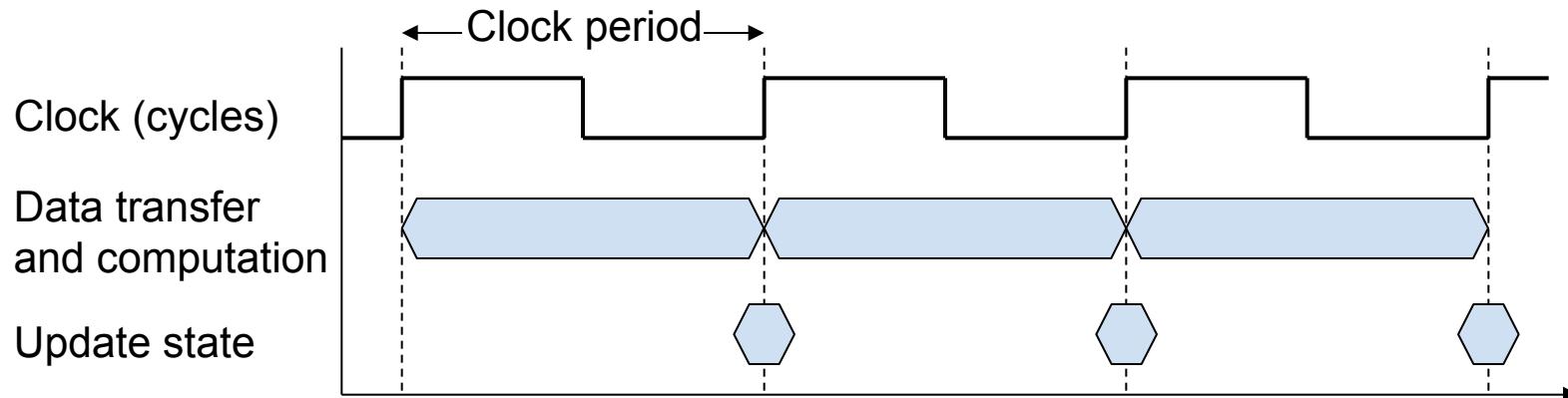
- Define Performance = $1/\text{Execution Time}$

$$\begin{aligned}\text{Performance}_X / \text{Performance}_Y \\ = \text{Execution time}_Y / \text{Execution time}_X = n\end{aligned}$$

- “X is n times as fast as than Y”
- Example: time taken to run a program
 - 10s on computer A, 15s on computer B
 - $\text{Execution Time}_B / \text{Execution Time}_A$
 $= 15s / 10s = 1.5$
 - So A is 1.5 times as fast as B

- Elapsed time
 - Total response time, including all aspects
 - Processing, I/O, OS overhead, idle time
 - Determines system performance
- CPU time
 - Time spent processing a given job
 - Discounts I/O time, other jobs' shares
 - includes user CPU time and system CPU time
 - Different programs are affected differently by CPU and system performance

- Digital hardware is driven by a constant-rate clock



- Clock period:** duration of a clock cycle
 - e.g., $250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$
- Clock frequency (rate):** cycles per second
 - e.g., $4.0\text{GHz} = 4000\text{MHz} = 4.0 \times 10^9\text{Hz}$



$$\begin{aligned} \text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \end{aligned}$$

- Performance can be improved by
 - Reducing number of clock cycles
 - Increasing clock rate
 - Hardware designer must often trade off clock rate against cycle count

- Computer A: 2GHz clock, program takes 10s CPU time
- Designing Computer B
 - Desire is for 6s CPU time
 - Suppose using a faster clock requires $1.2 \times$ clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

Clock Cycles = Instruction Count \times Cycles per Instruction

CPU Time = Instruction Count \times CPI \times Clock Cycle Time

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
 - Determined by program, ISA and compiler
- Average cycles per instruction
 - Determined by CPU hardware
 - If different instructions have different CPI
 - Average CPI is affected by instruction mix

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}\end{aligned}$$

A is faster...

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}\end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

...by this much

- With different instruction classes:

$$\text{Clock Cycles} = \sum_{i=1}^n (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^n \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

Relative frequency

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
 - Clock Cycles
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$
 $= 10$
 - Avg. CPI = $10/5 = 2.0$
- Sequence 2: IC = 6
 - Clock Cycles
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$
 $= 9$
 - Avg. CPI = $9/6 = 1.5$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
 - Algorithm: affects IC, possibly CPI
 - Programming language: affects IC, CPI
 - Compiler: affects IC, CPI
 - Instruction set architecture: affects IC, CPI, T_c

■ MIPS: Millions of Instructions Per Second

- Doesn't account for
 - Differences in ISAs between computers
 - Differences in complexity between instructions

$$\begin{aligned} \text{MIPS} &= \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} \\ &= \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6} \end{aligned}$$

- CPI is the average for a program
- and yields the “*native MIPS*”

- MIPS can be misleading
 - Varies between programs on the same machine
 - Does not reflect the instruction set complexity
 - May vary inversely with performance
- The following example illustrates this last point:

Assume a program is translated by two different compilers for the same machine:

$$\text{MIPS} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Instruction Class	Instruction counts (in millions) for each instruction class		
	A	B	C
Code from compiler 1	4	2	1
Code from compiler 2	9	1	1

$$\text{CPI}_1 = ((4*1 + 2*2 + 1*3)*10^6) / ((4+2+1)*10^6) = 1.57$$

$$\text{Thus MIPS}_1 = 100 \text{ MHz} / 1.57*10^6 = 63.64$$

$$\text{CPU time}_1 = ((4+2+1)*10^6 * 1.57) / 100 * 10^6 = 0.11 \text{ sec}$$

Assume a program is translated by two different compilers for the same machine:

Instruction Class	Instruction counts (in millions) for each instruction class		
	A	B	C
Code from compiler 1	4	2	1
Code from compiler 2	9	1	1

$$\text{CPI}_2 = ((9*1 + 1*2 + 1*3)*10^6) / ((9+1+1)*10^6) = 1.27$$

$$\text{Thus MIPS}_2 = 100 \text{ MHz} / 1.27*10^6 = 78.57$$

$$\text{CPU time}_2 = ((9+1+1)*10^6 * 1.27) / 100 * 10^6 = 0.14 \text{ sec}$$

Yields a higher MIPS rating but takes longer to execute

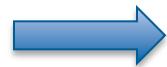
- *Native MIPS* was defined above
- *Peak MIPS* is based on minimum CPI
 - Assumes all instructions in program have minimum CPI
 - Minimum possible CPI = 1
 - All instructions require at least 1 clock cycle
- Relative MIPS is based on a standard machine

$$\text{MIPS}_{\text{relative}} = \frac{\text{Time}_{\text{reference}}}{\text{Time}_{\text{unrated}}} \times \text{MIPS}_{\text{reference}}$$



- *Benchmarks are suites of programs*
 - Chosen to be typical of workloads
 - Systems are rated based on benchmark execution times
 - Average time is used as a measure of performance
- Arithmetic mean is not used as average
 - Outliers can have undue influence on result
 - Geometric mean is used instead (defined below)

Geometric Mean



$$\sqrt[n]{\prod_{i=1}^n \text{Execution time}}$$

- Standard Performance Evaluation Corp (SPEC)
 - Develops benchmarks for CPU, I/O, Web, ...
- SPEC CPU2006
 - Elapsed time to execute a selection of programs
 - Negligible I/O, so focuses on CPU performance
 - Normalize relative to reference machine
 - Summarize as geometric mean of performance ratios
 - CINT2006 (integer) and CFP2006 (floating-point)

- SPEC CPU2017
 - Comprised of a group of 43 programs
 - Organized into 4 suites
- Another possibility is the Harmonic Mean
 - More appropriate for averaging rates such as MIPS
 - Average MIPS for suite gives a single performance measure

Harmonic Mean 
$$\bar{S}_H = \frac{N}{\sum_1^N \frac{1}{S_i}}$$

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

- Example: multiply accounts for 80s/100s
 - How much must multiply be improved to get 5× the overall performance?

$$20 = \frac{80}{n} + 20 \quad \blacksquare \quad \text{Can't be done!}$$

- Corollary: make the common case fast

MIPS measures integer performance

Engineering Applications employ floating point

- So the appropriate metric is Mega-flops (MFLOPS)
- Floating point support varies even more
- trig and exponentiation functions may be available
- In other cases, only simple arithmetic is supported

MFLOPS can be very unreliable in predicting performance

It is best to use execution time to assess performance

The architecture of a computer defines:
the view of the computer from the perspective of an assembly language or machine language programmer.

The instruction set architecture (ISA)
defines the software hardware boundary

It includes:

- the instruction set
- the machine instruction formats
- the available addressing techniques
- the operational register set
- the format of the available data types

The architecture specifies what the computer can do

“Computer Organization” is sometimes used interchangeably with
“Computer Architecture”
but they have different meanings

Computer Organization is also called the microarchitecture
describes how the capability defined by the architecture is
implemented

Architecture may define 32-bit memory word transfers
the organization may internally perform two 16-bit transfers

Computer models that share a common architecture may have
different microarchitectures (i.e., organizations).



The architecture of a clock is defined by the movement of hands on a marked dial to indicate the time

However, one clock may be driven by a wind-up spring, while another, with the same architecture, may be driven by a crystal oscillator

The user may be unaware of the internal timing mechanism

A machine code program can run on different machines with the same architecture without change
organizational details may differ

- what types of operations are available?
(integer, floating point, etc.)
- which instructions are allowed to reference memory?
- do operands have to reside in registers?
or can one or more reside in memory?
- are vector type instructions and vector registers available?
- how many bits do the instruction operands require?
- is a segmented or flat memory model used?
- how many operands can instructions employ?

- are the instructions pipelined?
- is a single-cycle or multi-cycle datapath used?
- how many cache levels are employed?
- is secondary storage provided by magnetic disks or by flash?
- are there multiple buses?
- is the control unit microprogrammed or is hardwired logic used?
- are multiple execution units included?
- how many memory accesses are used to retrieve data or instructions?
- are vector type instructions provided by an array processor or by a vector processor?

Computers with different **organizations**, but with a common shared architecture can execute the same machine code program.

Machine code programs are what the compiler produces when it translates a high level language source program (such as C or C++).

High level languages are designed to abstract away or hide the architecture

The underlying machine architecture is only revealed at the assembly language and machine language levels

Assemblers translate symbolic assembly language into machine code

Assembly instructions are a higher level representation of machine instructions

machine instructions are binary patterns understood by the computer hardware

Assembly language instructions and native built-in machine instructions have a one-to-one correspondence

A set of computers that share a common architecture is referred to as a computer family

Examples of computer families include:
the Intel x86, Motorola 68000 and MIPS processor families

Members of a computer family can all run the same machine language programs although they tend to vary greatly in the speed with which the programs are executed.

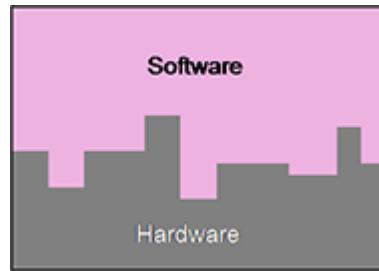
Different technologies may be used to implement the various models within the family.

Computer families usually include lower cost lower performance models as well as more powerful, faster and more expensive models.

New members of a computer family perform better
but execute programs written for older members
this is called “*backwards compatibility*”

Backwards compatibility is desirable
runs existing software base on newer more powerful
models allows for easier hardware upgrades

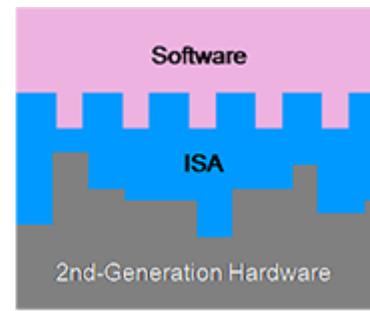
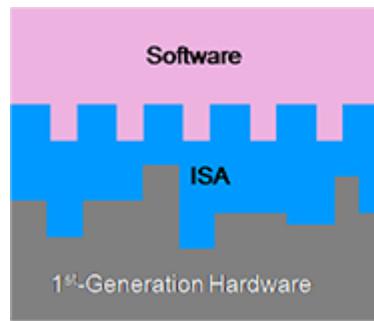
Programs must be re-complied to run on a machine with a
different architecture



Initially programs were written to interact directly with a machine's unique hardware

Each update or improvement in the hardware required rewriting programs that had been previously produced

Machine programs run on all machines with a standard shared ISA



This saves on development time and on cost

manufacturers can innovate and fine-tune the hardware for performance without breaking the existing software base

Von Neumann Machines

Most modern computers are based on this design
Developed by John von Neumann at Princeton
At the Institute for Advanced Studies in the 1940's.

These machines have 3 major components:

- a CPU
- a main-memory system
- an I/O system

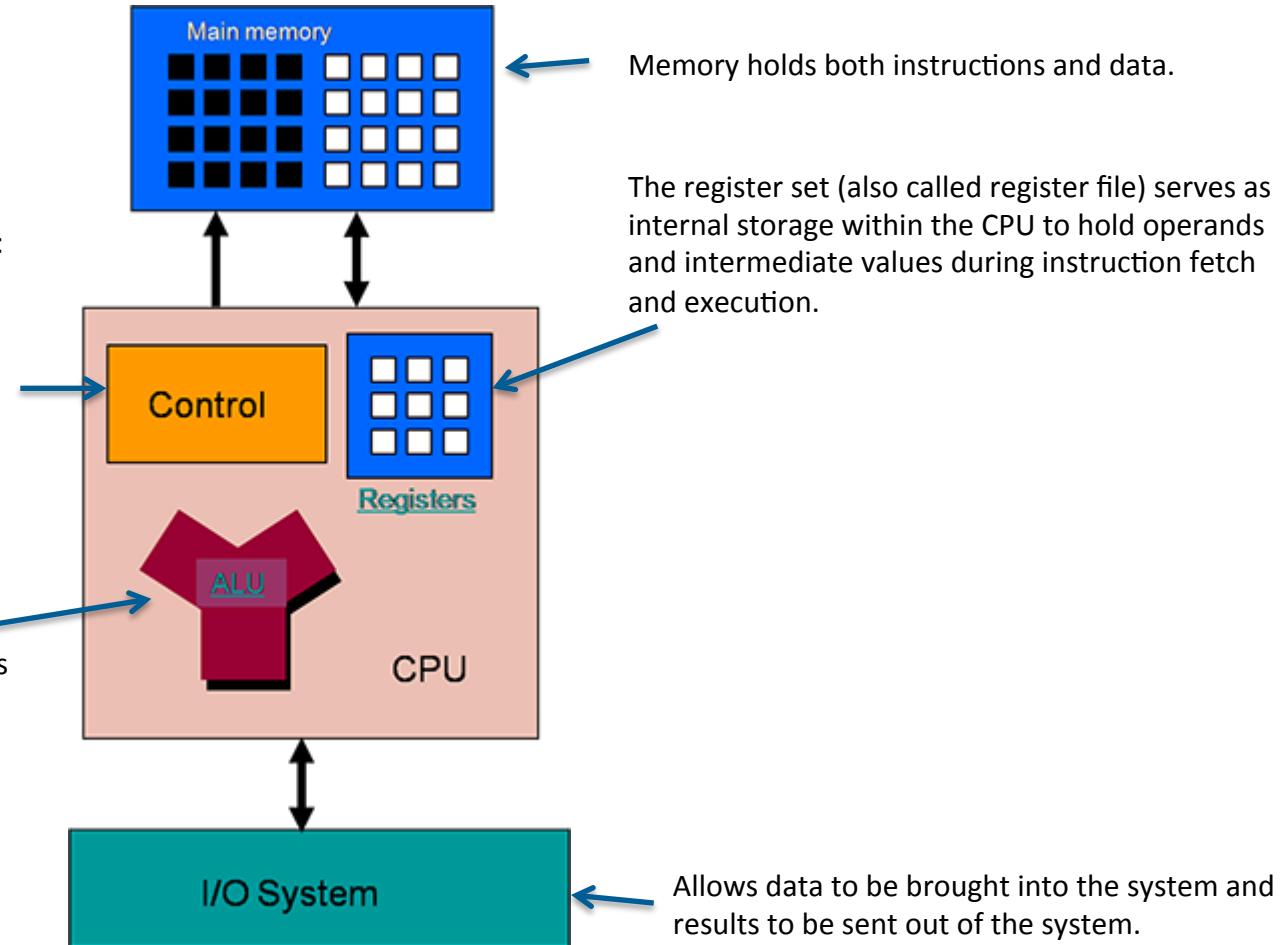
- Both programs and data are stored in a single memory
program instructions can be manipulated like data
- The program counter is used to fetch instructions
data operands are fetched during the execute cycle
based on the operand addressing mode
- Instructions execute sequentially
- flow of control may be altered by a branch type instruction
- CPU accesses memory over a single path
Which is a potential bottle neck
Called the "von Neumann bottleneck"



The diagram below depicts a system that adheres to the von Neumann architecture:

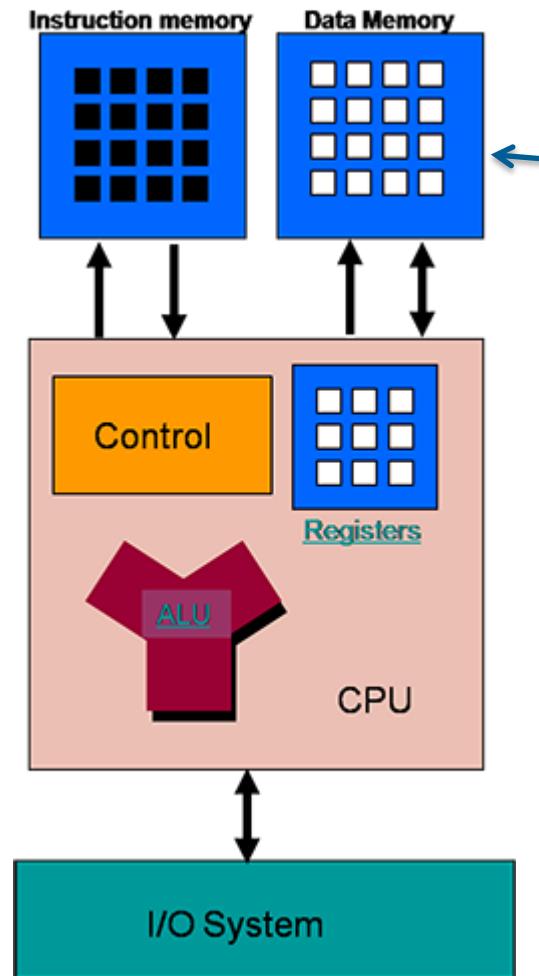
The major subsystems within the CPU are:
The control unit (CU) which selects and interprets machine instructions and coordinates the various parts of the computer in executing these instructions.

The arithmetic logic unit (ALU) which performs arithmetic, logical, and shift operations on the operands and generates the results.





As an alternative, illustrated below is a “Harvard architecture”:



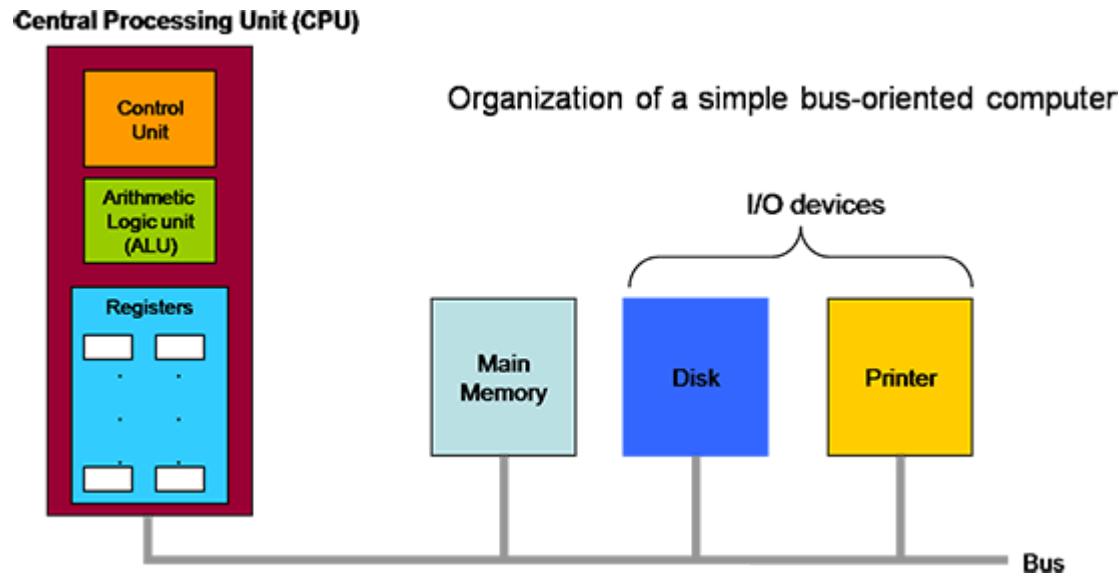
The distinguishing feature is the presence of separate instruction and data memories.

This allows one instruction to be fetched while another stores or reads an operand.

This design was developed at Harvard University by Howard Aiken and others.

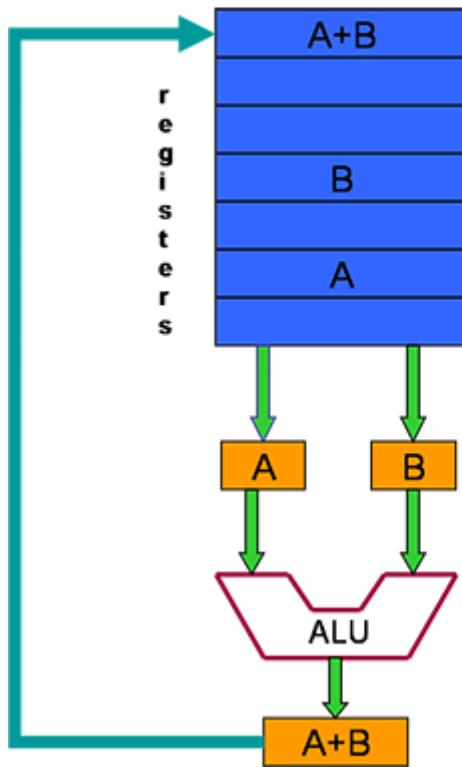


Computer components exchange data and communicate over a “system bus”:



A bus is a collection of parallel wires that carry address, data, and control signals
External buses connect the CPU to memory and I/O devices
Internal buses carry signals between registers, the ALU and the control unit

Different organizations have different datapaths

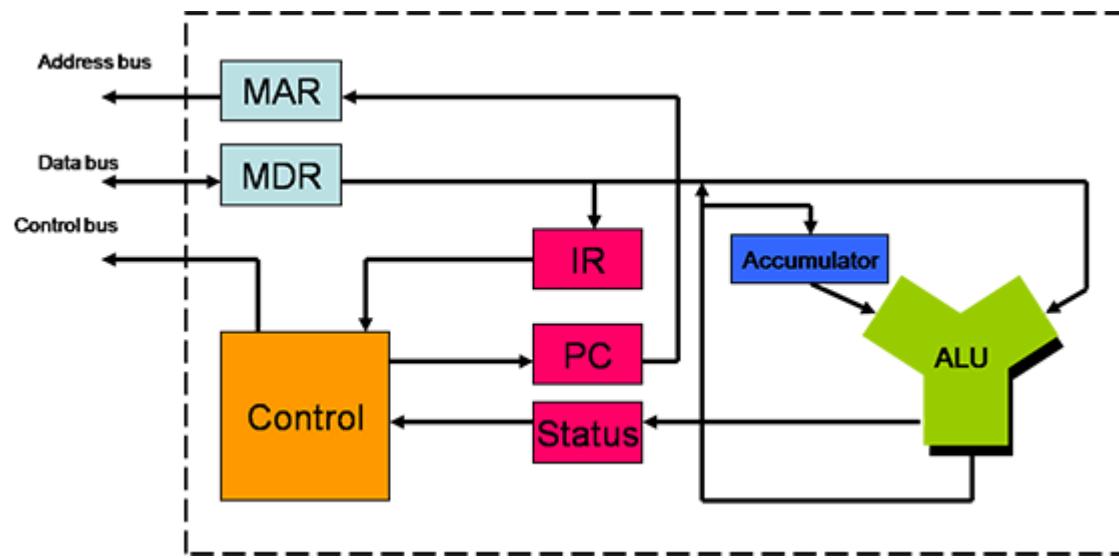


The registers, ALU and interconnecting buses constitute the data path for the machine.

The datapath contains all of the devices and pathways needed to execute instructions.

Typical Data path for executing instructions

Early designs used a special accumulator register to hold an input operand for the instruction and to receive the instruction's result



Such systems employed one-address instructions
a single memory operand
the second implicit operand was the accumulator register

LDA	X	# load the variable X from memory into the accumulator
ADD	Y	# add the memory variable Y to the accumulator
STA	Z	# store the accumulator result into memory variable Z
BGE	P1	# branch to location P1 if result is not negative

Most instructions incur a penalty
the time required to access the memory operand

The alternative is a load/store architecture
allows only a few instructions to use memory operands
all others have to use operands that reside within registers
registers takes a fraction of the memory access time
(better performance)

Accumulators limit the number of high cost CPU registers

Other designs use two or more instruction memory operands
This gives rise to two-address and three-address instructions:

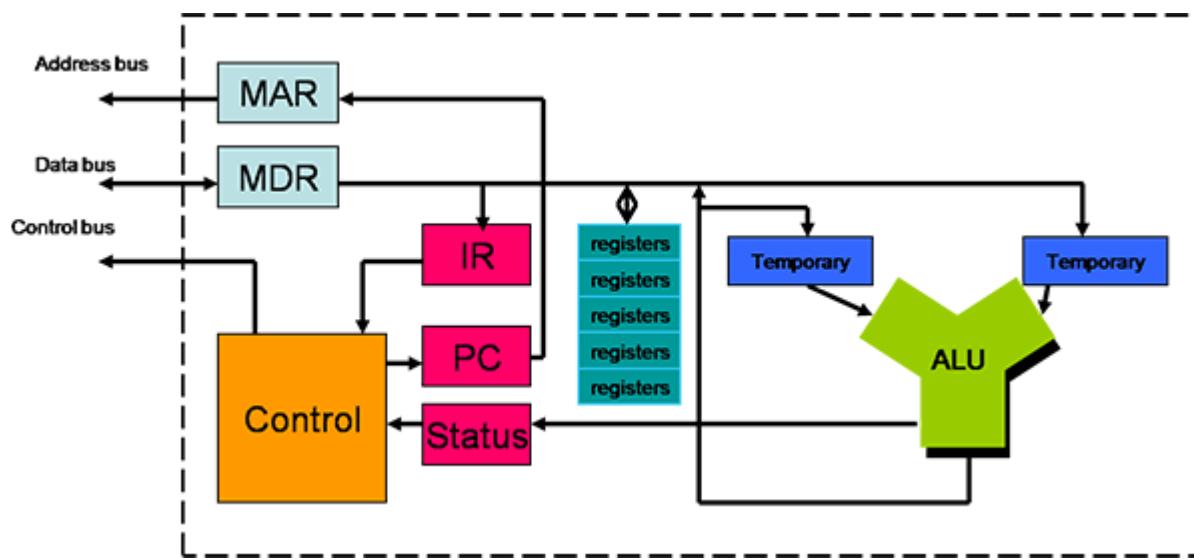
Mul x,y # the variable x is multiplied by y and the product is stored back in x
Add z,w # the sum of z plus w is stored in z

Three-address instructions:

Mul z,x,y # the product of x times y is stored into z
Add w,x,y # the sum of x plus y is stored into w

such instructions use even more memory accesses
therefore take longer to execute.

Over time, register cost declined and reliability increased



More on-chip registers allowed fast register-to-register operations
indexed and register indirect addressing could be used

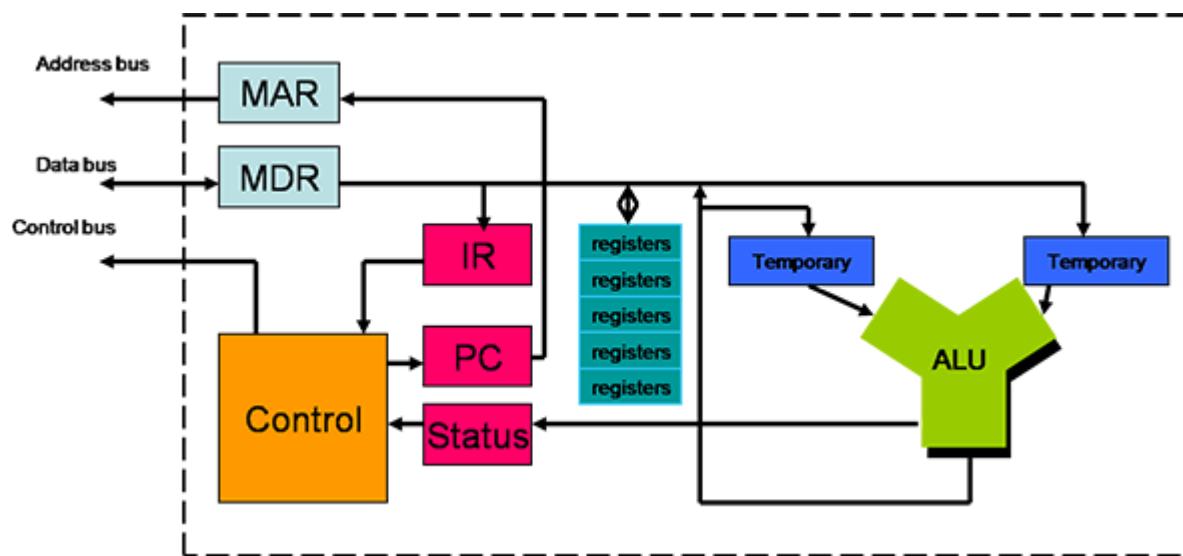
MAR and MDR serve as the CPU to memory interface
MAR (memory address register)

holds the address of the item in memory to be accessed.

MDR (memory data register)

receives the item read from memory

or holds the item to be written into memory

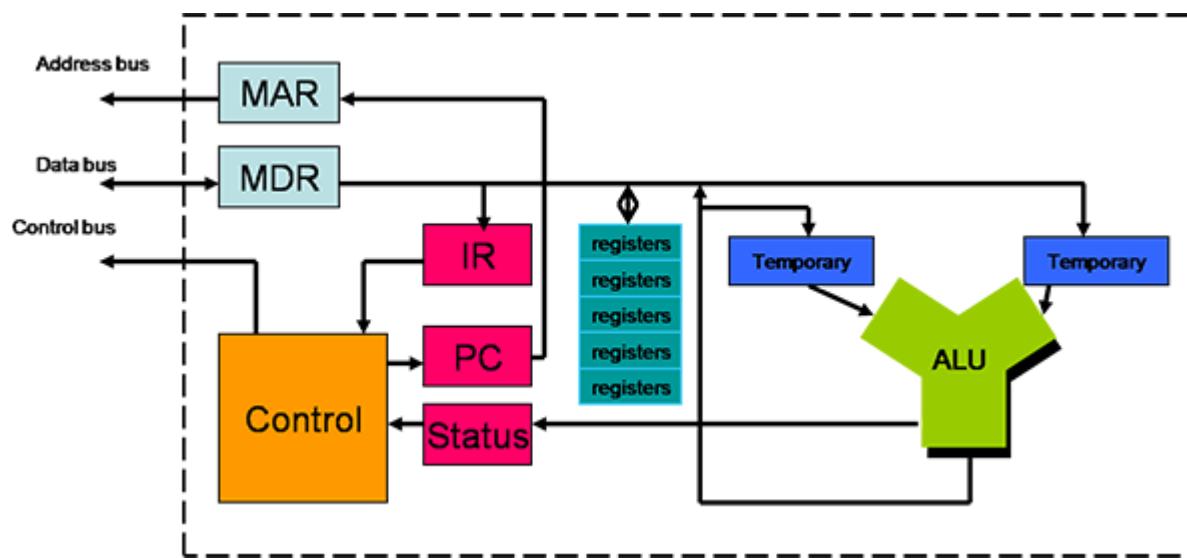


IR is the instruction register

Holds the instruction while the control unit processes it

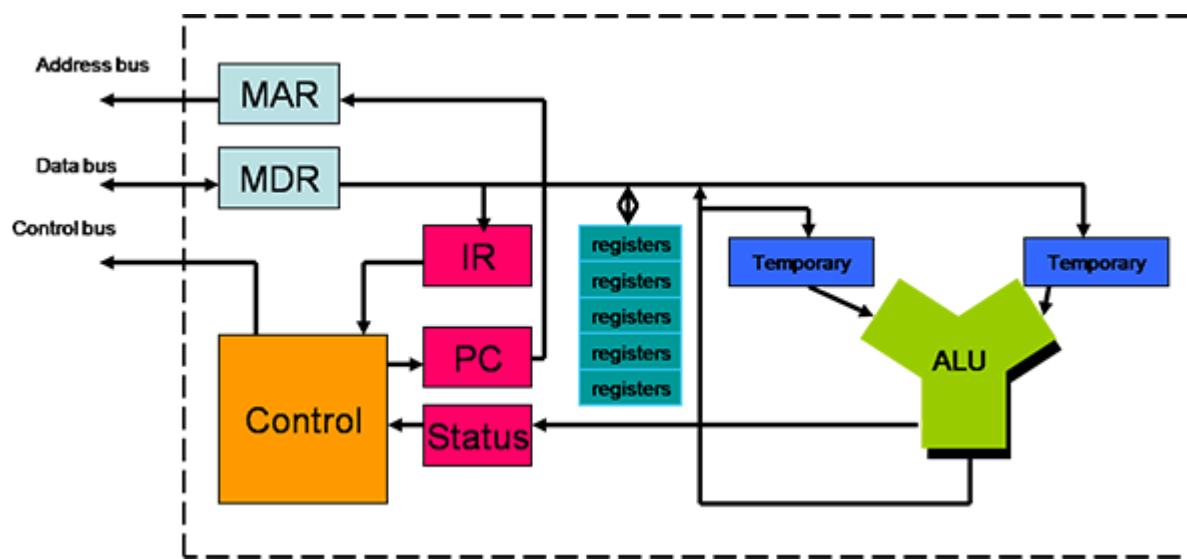
PC is the program counter register

holds the address of the next instruction to fetch from memory



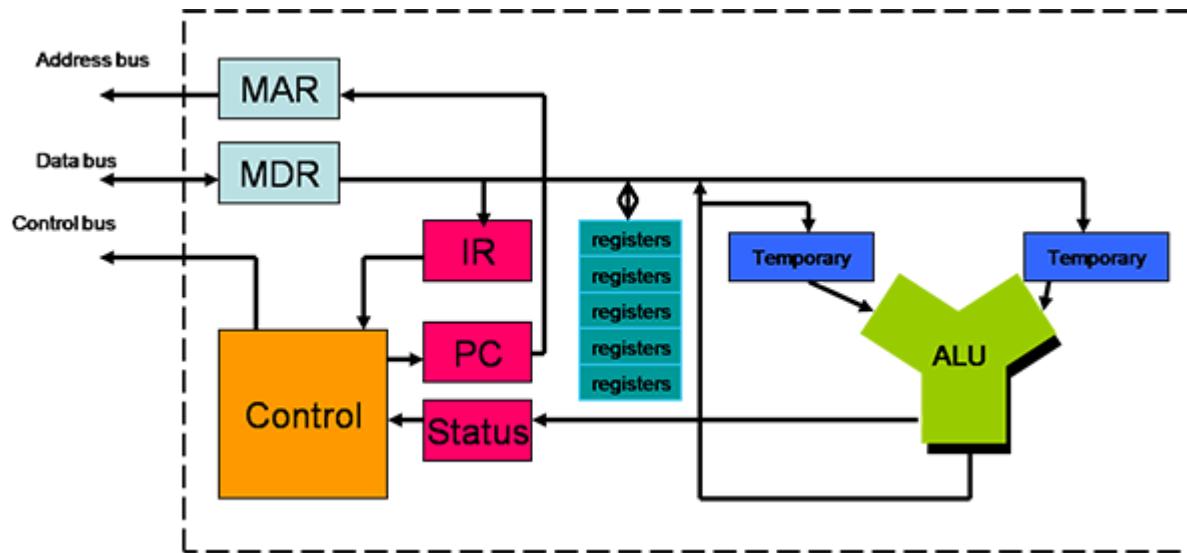
STATUS register holds condition codes
(negative, zero, positive, a carry, etc.)

Registers serve as internal storage cells
hold operands and results for the instructions



Control unit generates signals that direct operations
For example, which registers to use, what ALU operation to perform, etc.

The ALU actually performs the operations
(such as addition, multiplication, shifting, etc.)



Multi-register designs allow explicit use of one or more register operands

Some systems have instructions that use 1 memory operand and 1 or more registers: (for example the Intel x86 systems and their clones)

```
add    eax,m      # add the 32-bit variable m to the 32-bit eax register  
mov    ebx,data1  # copy the contents of the variable data1 into the ebx register
```

These are sometimes called one and a half address instructions.

Some classes of computers use mostly implicit stack operands
These are known as stack-based machines
they use zero-address instructions
the instructions make no explicit reference to operands

Examples of such instructions are:

```
add      # pop the top two stack elements, add them and push the sum onto the stack
dup      # make a copy of the top element (i.e., duplicate it)
sub      # pop the top two elements and push the difference back onto the stack
pop x    # remove the top element and store it into memory variable x
push 5   # push the constant 5 onto the stack
```

these instructions incur a penalty for each operand access
unless the stack is implemented using registers

MIPS machines are reduced instruction set computers (RISC)
require that most instructions use only registers operands
only the load and store instructions use memory operands
That is, they employ a load/store architecture

An example of a MIPS instruction that uses 3 register operands:

```
sub $8,$9,$10 # subtract register $10 from $9 and put the difference in $8
```

- Instruction sets that only included simple straight forward instructions
- Simple addressing modes
- Fixed size machine instructions that can be fetched with one memory access
- Instruction pipelines that overlap the execution of instructions and complete one instruction per clock cycle

- Restricting the use of memory operands to only the load and store type instructions
- Hardwired logic implementation as opposed to microprogramming
- The use of optimizing compilers to generate the code to perform more complex tasks
- May have large instruction sets but the instructions are simple in nature

Require multiple instructions to accomplish what CISC machines perform in a single complex instruction.

The compiler (or assembly language programmer) has to generate a sequence of simple instructions that perform the same actions of the individual CISC instructions.

Can result in code expansion which can be a problem.

Code expansion refers to the increase in size that you get when you take a program that had been compiled for a CISC machine and re-compile it for a RISC machine

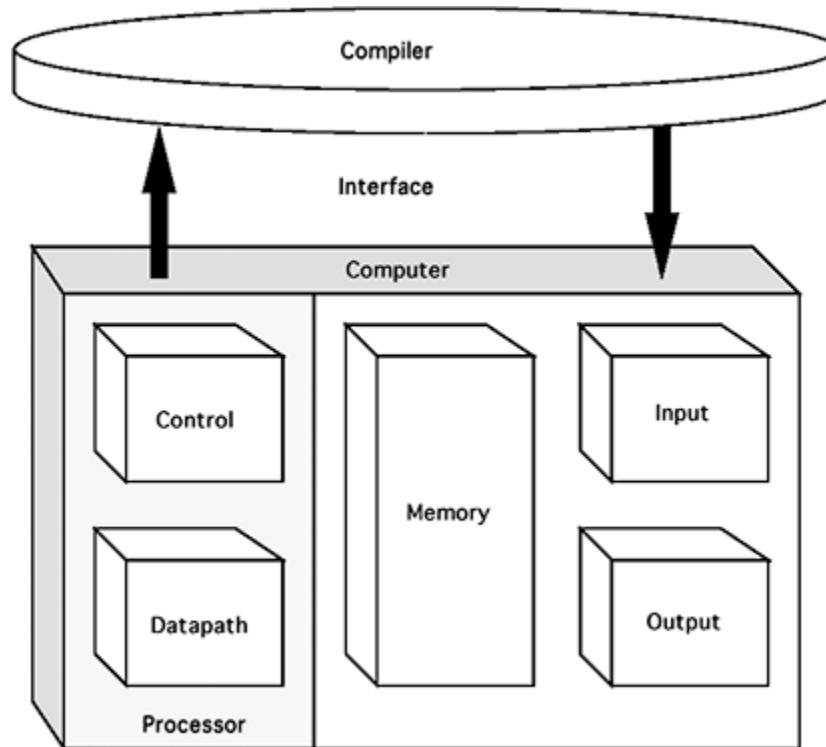
The amount of expansion depends primarily on the compiler
higher quality compilers generate less code
the nature of the machine's instruction set has an effect

Typically, code expansion can range up to 100% or more

- MIPS
- Sparc
- ARM
- PowerPC

The MIPS architecture will be explained in detail later in the course as a prime example of a RISC machine.

much of the complexity is handled by the compiler not the hardware



optimizations such as instruction rearrangement take care of pipeline dependencies



The simpler nature of the RISC control unit leaves space (real estate) on the CPU chip

This extra space can be used to implement additional registers

RISC machines tend to have 32 or more registers

CISC machines may have 8 or fewer registers

Sparc processors have more than a hundred CPU registers organized into logical groups called register windows

RISC	CISC
Simple instructions	Complex instructions
Completes one instruction per cycle	Requires many cycles to complete an instruction
Load/Store Architecture (only load and store instructions can reference memory)	Most instructions can reference memory
Relies heavily on pipelining	Relies less on pipelining
Instructions executed in hardware	Microcode interprets instructions
Fixed size instructions with a small number of formats	Variable size instructions with many formats
Simpler addressing modes	Many intricate addressing modes

Integers are represented as n-bit vectors (series of 1's and 0's)

$$B = b_{n-1} \dots b_1 b_0$$

The unsigned value represented is:

$$V(B) = b_{n-1} \times 2^{n-1} + \dots + b_1 \times 2^1 + b_0 \times 2^0$$

The b 's are the coefficients in a powers of 2 polynomial

The number of bits (n) determines the range that can be covered

$$0 \text{ to } 2^n - 1$$

Unsigned overflow exists if an operation yields a value out of range

The total number of patterns = 2^n where n is the number of bits

This is called the modulus of the system

Incrementing the largest pattern (all 1's) rolls over to 0

Signed systems use some of the patterns for negative values

- Sign-and-magnitude
- One's complement
- Two's complement
- Biased (or excess)

Two's complement is by far the most common

Biased is used for integer exponents in floating point numbers

The first three systems are compared below (using 4 bits):

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

In each system, the MSB is 0 for positive and 1 for negative values
Positive values have identical representations for each system

- Same as for the unsigned value

They differ in how negative values are represented

- Only negative values require complementing

Sign-and-magnitude:

MSB only indicates sign (0 for +, 1 for -)

Remaining bits give the magnitude

One's complement

Invert each bit to get the negative

Same as adding negative value to modulus-1

e.g. -5 is represented as $-5 + (16-1) = 10$

Two's complement

Invert each bit and add 1 to get negative

Same as adding negative value to modulus

e.g. -5 represented as $-5 + 16 = 11$

$b_3 b_2 b_1 b_0$	Sign and magnitude	1's complement	2's complement
0 1 1 1	+ 7	+ 7	+ 7
0 1 1 0	+ 6	+ 6	+ 6
0 1 0 1	+ 5	+ 5	+ 5
0 1 0 0	+ 4	+ 4	+ 4
0 0 1 1	+ 3	+ 3	+ 3
0 0 1 0	+ 2	+ 2	+ 2
0 0 0 1	+ 1	+ 1	+ 1
0 0 0 0	+ 0	+ 0	+ 0
1 0 0 0	- 0	- 7	- 8
1 0 0 1	- 1	- 6	- 7
1 0 1 0	- 2	- 5	- 6
1 0 1 1	- 3	- 4	- 5
1 1 0 0	- 4	- 3	- 4
1 1 0 1	- 5	- 2	- 3
1 1 1 0	- 6	- 1	- 2
1 1 1 1	- 7	- 0	- 1

Sign-and-magnitude properties:

two distinct representations for 0

+0 → 0 sign bit and all 0's for magnitude

-0 → 1 sign bit and all 0's for magnitude

Range = $-(2^{n-1}-1)$ to $+2^{n-1}-1$

One's complement:

two distinct representations for 0

+0 → bits are all 0's

-0 → bits all 1's

Range = $-(2^{n-1}-1)$ to $+2^{n-1}-1$

Two's complement:

Unique representation of 0 (all 0 bits)

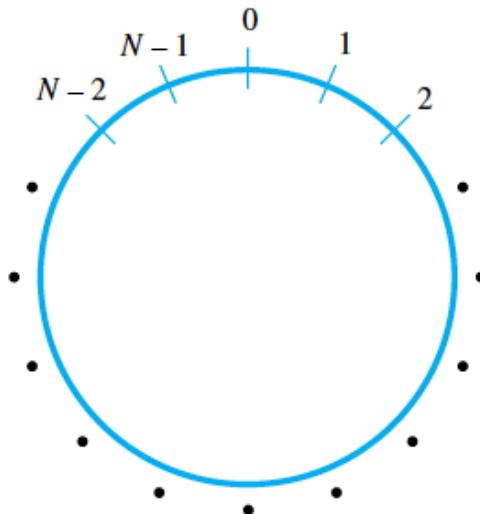
Range = -2^{n-1} to $+2^{n-1}-1$

Extra negative pattern (-2^{n-1}) has MSB=1, all other bits = 0

We will review the algorithms that the ALU must implement

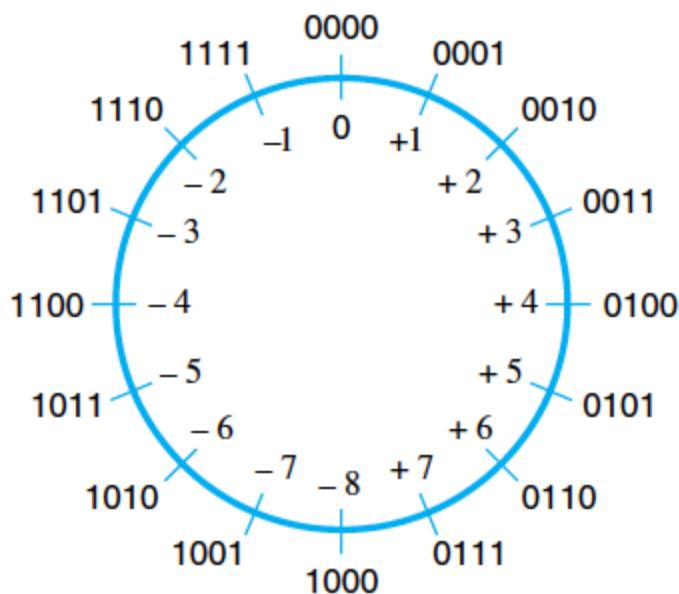
- Addition & subtraction
- Multiplication & division

We will also show why two's complement is preferred



An incremented unsigned integer rolls over from the max value back to 0. (modulo N)

Stepping counter-clockwise
from 0 goes to -1



Stepping clockwise from 0 goes
to +1

In this 4-bit system there are 16
possible values (modulus=16)

To add M to a value, perform M clockwise steps

To subtract M from a value, perform M counter-clockwise steps
this is equivalent to $16-M$ clockwise steps
the same as adding the two's complement of $-M$

Using hex makes dealing with more bits easier

Each hex digit (0,..,9,A,B,C,D,E,F) corresponds to a group of 4 bits

Example: assume 16-bit numbers:

$$\begin{array}{r} -9 \\ +22 \\ \hline 13 \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 0xFFFF \\ +0x0016 \\ \hline 0x000D \end{array} \quad \text{modulus} = 2^{16} = 65536 = 0x10000$$

-9 in two's complement = $65536 - 9 = 65527 = 0xFFFF$

$$\begin{array}{r} 350 \\ +922 \\ \hline 1272 \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 0x015E \\ +0x039A \\ \hline 0x04F8 \end{array}$$

In two's complement addition:

add the two bit patterns

ignore any carry out of the leftmost bit

Subtract by adding the two's complement of the subtrahend

Only negative values need to be complemented

Two's complement is preferred over one's complement

One's complement has both +0 and -0

One's complement addition requires an end-around carry

$$\begin{array}{r} -9 \\ +22 \\ \hline 13 \end{array} \quad \xrightarrow{\hspace{1cm}} \quad \begin{array}{r} 0xFFFF \\ +0x0016 \\ \hline 0x000C \end{array} \quad \text{modulus-1} = 2^{16} - 1 = 65535 = 0xFFFF$$

The carry must be added into the result to obtain $0x000C + 1 = 0x000D = 13$

Using sign and magnitude representation complicates arithmetic
extra operations are required (sign checking & absolute value)

For addition or subtraction:

- check the signs of the two numbers
- if they differ, subtract the smaller number from the larger
- use the sign of the larger as the sign for the result

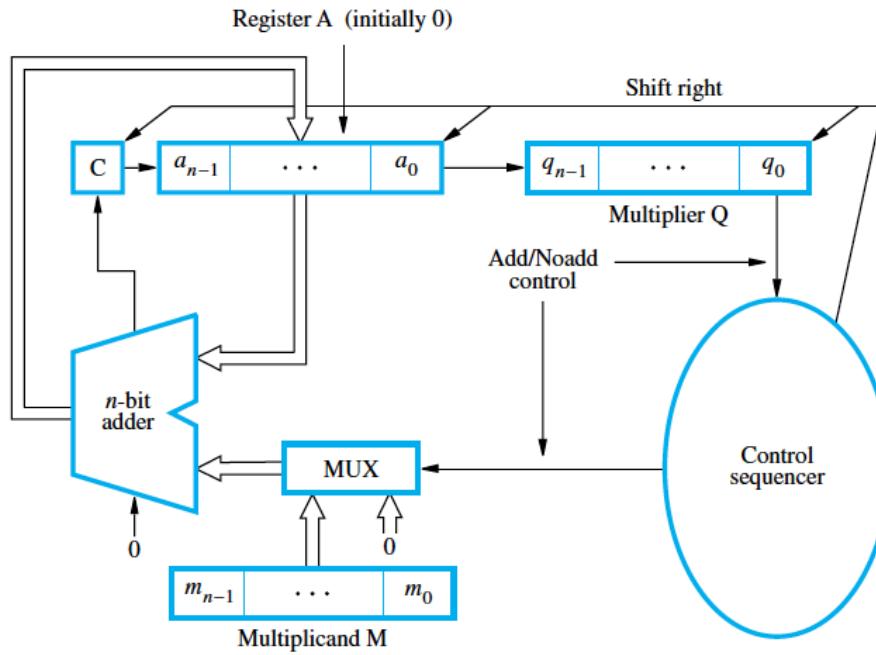
For multiplication or division:

- check the signs of the two numbers
- if they differ, the result is negative
- compute using the absolute values of the two numbers

So two's complement representation and arithmetic is preferred

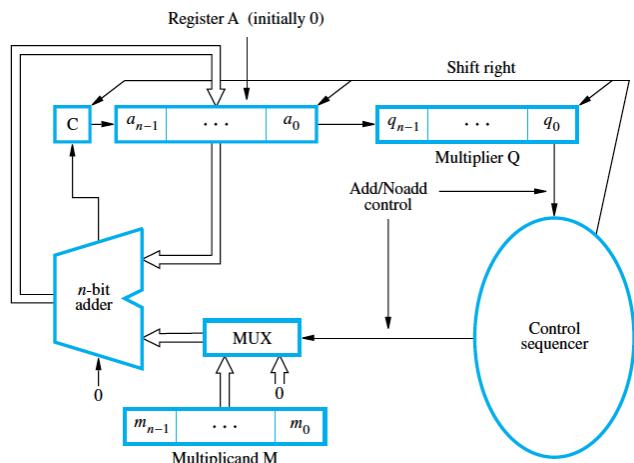
To multiply by 2^n , just shift left n bits

In general, multiplication involves shifting and adding



The product of two n -bit numbers requires $2n$ bits

Configuration for unsigned multiplication



4-bit numbers (-13 x 11) unsigned multiply

		M	Q	
0	C	1 1 0 1	1 0 1 1	Initial configuration
0	A	0 0 0 0		
0		1 1 0 1	1 0 1 1	
0		0 1 1 0	1 1 0 1	
1		0 0 1 1	1 1 0 1	First cycle
0		1 0 0 1	1 1 1 0	Second cycle
0		1 0 0 1	1 1 1 0	Third cycle
0		0 1 0 0	1 1 1 1	Fourth cycle
		Product		
1	0 0 0 1	1 1 1 1		
0	1 0 0 0	1 1 1 1		

Repeat N times (N= 4 here)

Add multiplicand to the A register only if the LSB of Q is 1

Shift for each cycle (C, A and Q together)

Generates an 8-bit product

Compute 13×11 and negate the result to get -143



A more general technique

Works for both positive and negative factors

Reduces the number of operations required

Example: suppose we want to multiply by $30 = 0011110_2$
30 can be regarded as

$$\begin{array}{r} 0100000 \quad (32) \\ - 0000010 \quad (2) \\ \hline 0011110 \quad (30) \end{array} \quad \text{i.e., multiply by } (32 - 2)$$

Add $32 \times$ multiplicand plus $-2 \times$ multiplicand



Multiplier		Version of multiplicand selected by bit i
Bit i	Bit $i - 1$	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

Scan multiplier bits from right to left
(assume initial 0 bit to right of LSB)
subtract multiplicand when moving from 0 to 1
add multiplicand when moving from 1 to 0
neither add nor subtract when moving from 0 to 0 or 1 to 1
shift multiplicand left one bit for each cycle



The original multiply is “*recoded*”

$30 = 0011110_2$ using 7 bits

0 0 1 1 1 1 0

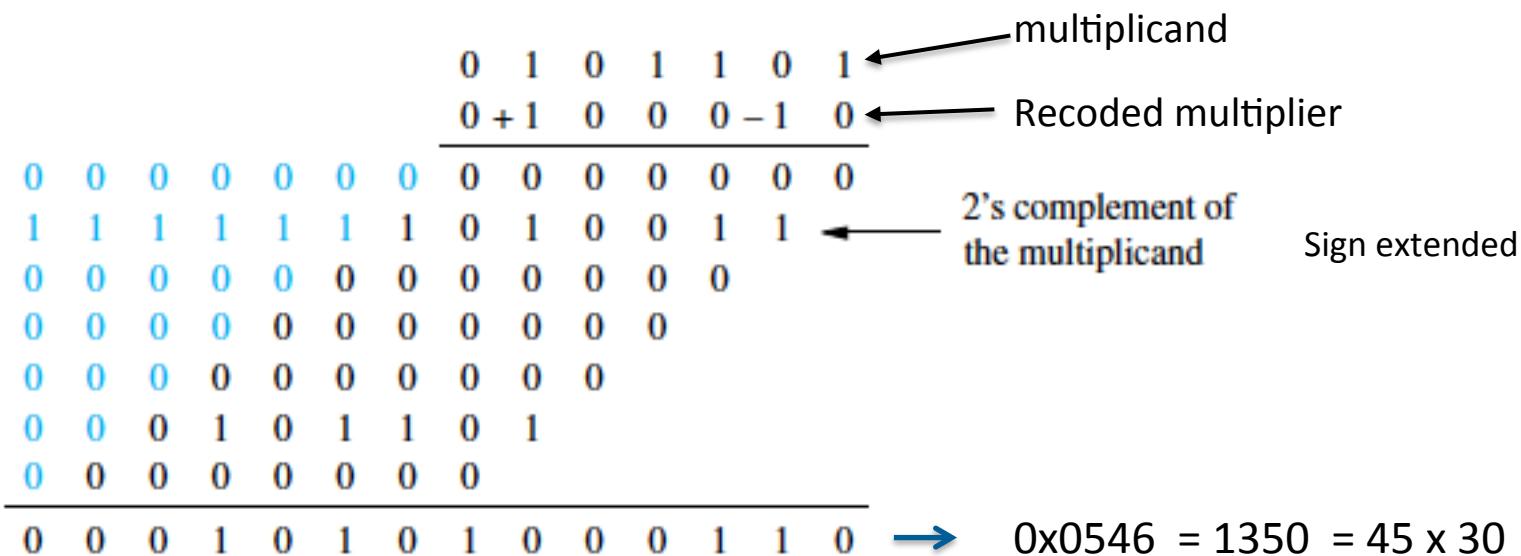


0 +1 0 0 0 -1 0

Subtract multiplicand for each -1 in recoded multiplier
Add multiplicand for each +1 in recoded multiplier
Neither add nor subtract for each 0 in recoded multiplier

45 → 0101101

-45 → 1010011



One addition and one subtraction is required to generate the product
Using 00111110 would have required 5 additions

Assume a 17-bit multiplier:

$$\begin{array}{cccccccccccccccccc} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ & & & & & & & & \downarrow & & & & & & & & & \\ 0 & +1 & -1 & +1 & 0 & -1 & 0 & +1 & 0 & 0 & -1 & +1 & -1 & +1 & 0 & -1 & 0 & 0 \end{array}$$



Longhand division in binary is similar to that in decimal

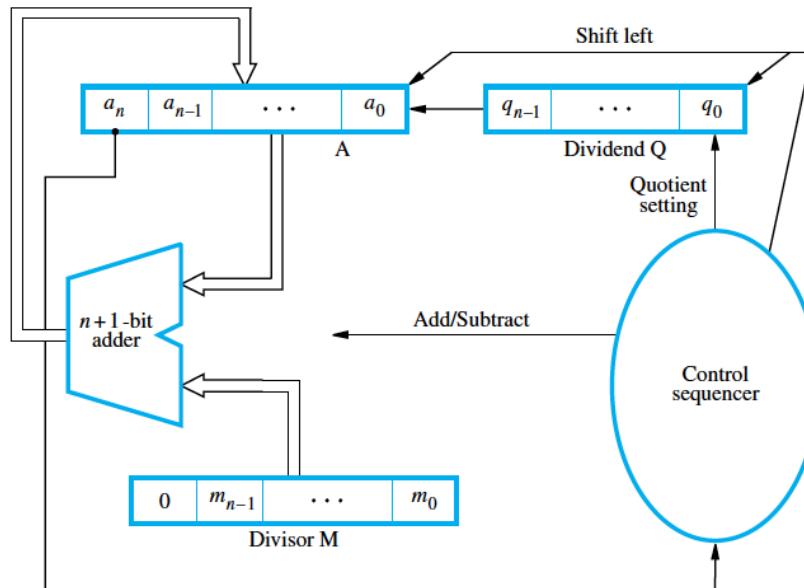
$$\begin{array}{r} 21 \\ 13 \overline{)274} \\ 26 \\ \hline 14 \\ 13 \\ \hline 1 \end{array}$$

$$\begin{array}{r} 10101 \\ 1101 \overline{)100010010} \\ 1101 \\ \hline 10000 \\ 1101 \\ \hline 1110 \\ 1101 \\ \hline 1 \end{array}$$

Both a quotient and a remainder are produced

Handle signed numbers by dividing the positive equivalents
Sign of remainder = sign of dividend (dividend rule)
Quotient is negative if the divisor and dividend differ in sign

Set A to 0
Put divisor in M
Put dividend in Q



Repeat the following 3 steps n times:

1. Shift A and Q left one bit position
2. Subtract M from A
3. If A is negative, set q_0 to 0 and add M back to A (i.e. restore A); otherwise set q_0 to 1

When done, Q contains quotient and A contains the remainder

Assume 4-bit values. Division of 8 by 3

$$\begin{array}{r} 10 \\ 11 \overline{) 1000} \\ 11 \\ \hline 10 \end{array}$$

Initially	0 0 0 0 0	1 0 0 0
Shift	0 0 0 1 1	0 0 0 □
Subtract	1 1 1 0 1	
Set q_0	1 1 1 1 0	
Restore	1 1	
	0 0 0 0 1	0 0 0 0
Shift	0 0 0 1 0	0 0 0 □
Subtract	1 1 1 0 1	
Set q_0	1 1 1 1 1	
Restore	1 1	
	0 0 0 1 0	0 0 0 0
Shift	0 0 1 0 0	0 0 0 □
Subtract	1 1 1 0 1	
Set q_0	0 0 0 0 1	
Shift	0 0 0 1 0	0 0 0 1
Subtract	1 1 1 0 1	0 0 1 □
Set q_0	1 1 1 1 1	
Restore	1 1	
	0 0 0 1 0	0 0 1 0

First cycle

Second cycle

Third cycle

Fourth cycle

2 Remainder Quotient 2



Avoids having to add M back when subtracting makes A<0

Restoring division computes A-M and if negative adds A back before shifting left 1 bit for the next cycle

Shifting A-M left first and then adding M gives:

$$2(A-M)+M = 2A - M \text{ (which is needed in the next cycle)}$$

This avoids the restore step

Stage 1: Repeat the following 2 steps n times:

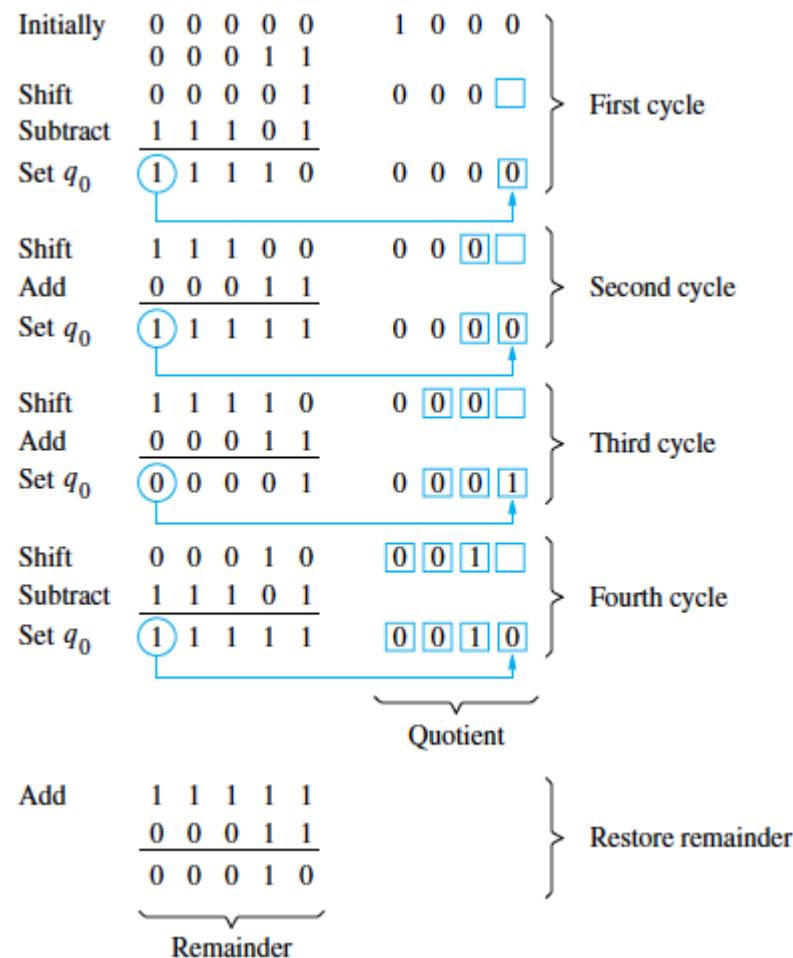
1. If $A < 0$, shift A and Q left 1 bit and add M to A
Else shift A and Q left 1 bit and subtract M from A

2. If $A < 0$, set $q_0 = 0$ else set $q_0 = 1$

Stage 2: If $A < 0$, add M to A

Stage 2 is needed to leave the proper positive remainder in A

Assume 4-bit values. Division of 8 by 3



Use absolute values of the dividend and divisor

Quotient is negative if the signs of divisor & dividend differ

Sign of remainder should match the sign of the dividend
this is called the dividend rule

32-bit integers have an implied number point on the right

Setting the number point in the middle allows fractional values

bits to the left of the point represent non-negative powers of 2
bits to the right of the point represent negative powers of 2

To illustrate assume 8-bit patterns with point in middle:

Implied number point

$$\begin{array}{ccccccccc} 0 & 1 & 1 & 0 & \downarrow & 0 & 1 & 0 & 1 & 0 \\ & \nearrow & \nearrow & \nearrow & & \nearrow & & \nearrow & \\ 2^{+2} & 2^{+1} & 2^{-1} & & & 2^{-1} & 2^{-2} & 2^{-3} \end{array} = 4 + 2 + 0.5 + 0.125 = 6.625$$

$$\text{In hex: } 6.A = 6 * 16^0 + 10 * 16^{-1} = 6 + 10 * 0.0625 = 6.625$$

The position of the number point is set, thus the name “fixed-point”

We would like to be able to represent ranges of values

- Very large integer parts

- Very small fractional parts

This requires that the number point *slide* or *float*

- Representation must specify number point location

Floating Point Representation must include:

- Sign of number
- Significant bits
- Signed exponent for an implied base of 2

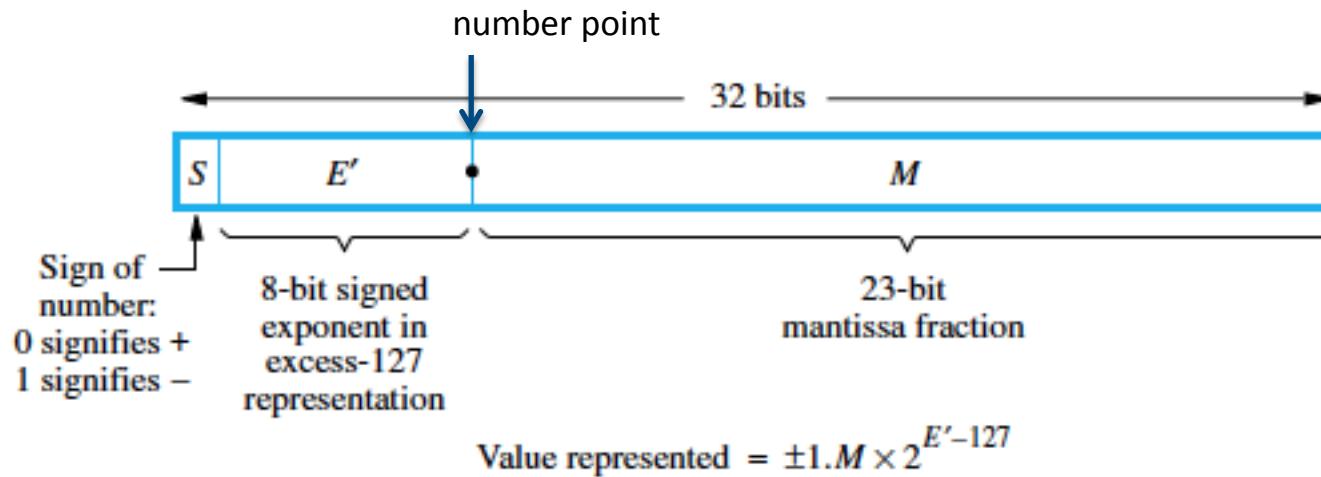
Width of exponent field determines range

Number of significant bits determines precision

IEEE-754 Standard specifies a common floating format

32-bit single precision

64-bit double precision



$E' = \text{exponent} + 127$ (excess-127 form) also called the “characteristic”

“Normalized” numbers have an implied 1 to the left of the number point

With single precision numbers $0 \leq E' \leq 255$

But endpoints (0 and 255) are used for special cases

0 denotes exponent of -126 for denormalized

Denormalized numbers have 0 to left of number point

$E'=0$ and $M \neq 0$ for denormalized numbers

$E'=0$ and $M=0$ for true zero

Denormalized numbers allow gradual underflow

$E'=255$ denotes $\pm\infty$ or NaN (not a number)

$E'=255$ and $M \neq 0$ for NaN (e.g. $0/0$ or $\sqrt{-1}$)

$E'=255$ and $M=0$ for $\pm\infty$

Provides 7 decimal places of precision

With the hidden 1, the significand is essentially 24 bits

X, the number of decimal places would be such that:

$$10^x \approx 2^{24}$$

$$\log(10^x) \approx \log(2^{24})$$

$$X = 24 * \log(2) = 24 * 0.301 = 7.22$$

Approximate range for normalized values is $\pm(10^{\pm 38})$

$\pm 2^{-126}$ to $\pm 2^{+128}$

0x14140000



$$\text{Value represented} = 1.001010\dots 0 \times 2^{-87}$$

$$\text{Exponent} = \text{characteristic } 00101000 - 127 = 40 - 127 = -87$$

How would 763.5_{10} be represented as a single precision IEEE floating point number?

$$763.5_{10} = 2FB.8_{16} = 1011111011.1_2 = 1.0111110111 \times 2^9$$

Hence the sign bit = 0

$$\text{the characteristic} = 9 + 127 = 136_{10} = 10001000_2$$

the 23-bit mantissa is 01111101110000000000000

The corresponding IEEE single precision representation is

s	characteristic	mantissa
0	10001000	01111101110000000000000

This 32-bit pattern can be written in short hand form using 8 hex digits: 443EE000

Converting decimal numbers to floating point format with the aid of a calculator.

Given a decimal value X, if we compute $N = \text{Floor}(\log_2 X)$, this will produce the exponent needed in expressing X in the form $1.M \times 2^N$.

$\text{Floor}(\text{number})$ is defined as the largest integer less than or equal to number. For example:

$$\text{Floor}(4.8) = 4$$

$$\text{Floor}(-3.2) = -4$$

To find the exponent needed for the decimal value 9.56×10^4 , we would compute

$$\begin{aligned}\log_2(9.56 \times 10^4) &= \log_{10}(9.56 \times 10^4) / \log_{10}2 = 4.9805 / 0.301 = 16.5447 \\ N &= \text{Floor}(16.5447) = 16\end{aligned}$$

Now if we divide $9.56 \times 10^4 / 2^{16}$

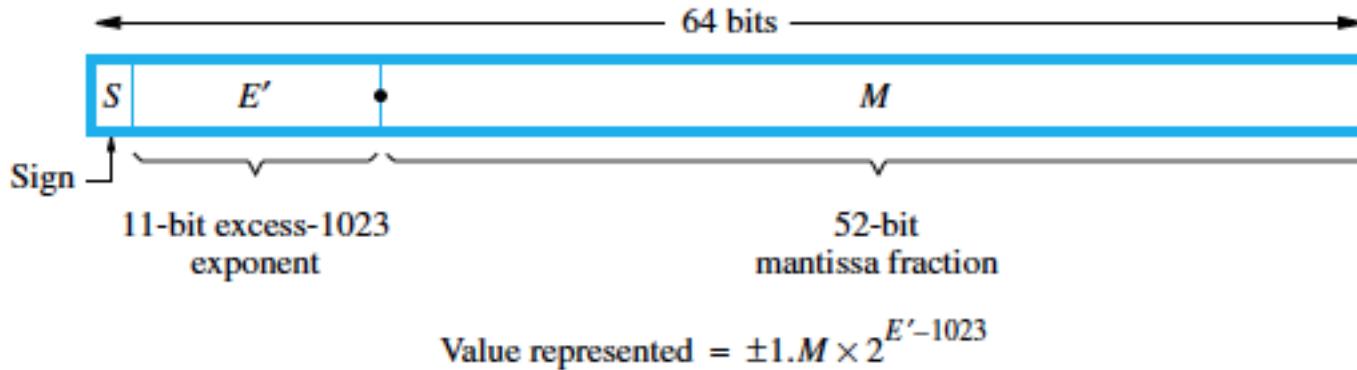
We get 1.4587, hence $9.56 \times 10^4 = 1.4587 \times 2^{16}$

Multiply fraction to convert to 24-bit integer: $0.4587 \times 2^{24} = 7695708$

So $0.4587 = 7695708 \times 2^{-24} = 0x756D5C \times 2^{-24} = 0.756D5C$

Mantissa is 011101010110110101011100

Characteristic = $16 + 127 = 143 = 0x8F$



$$E' \text{ (characteristic)} = \text{exponent} + 1023$$

$E'=0$ and $M \neq 0$ for denormalized numbers

$E'=0$ and $M=0$ for true zero

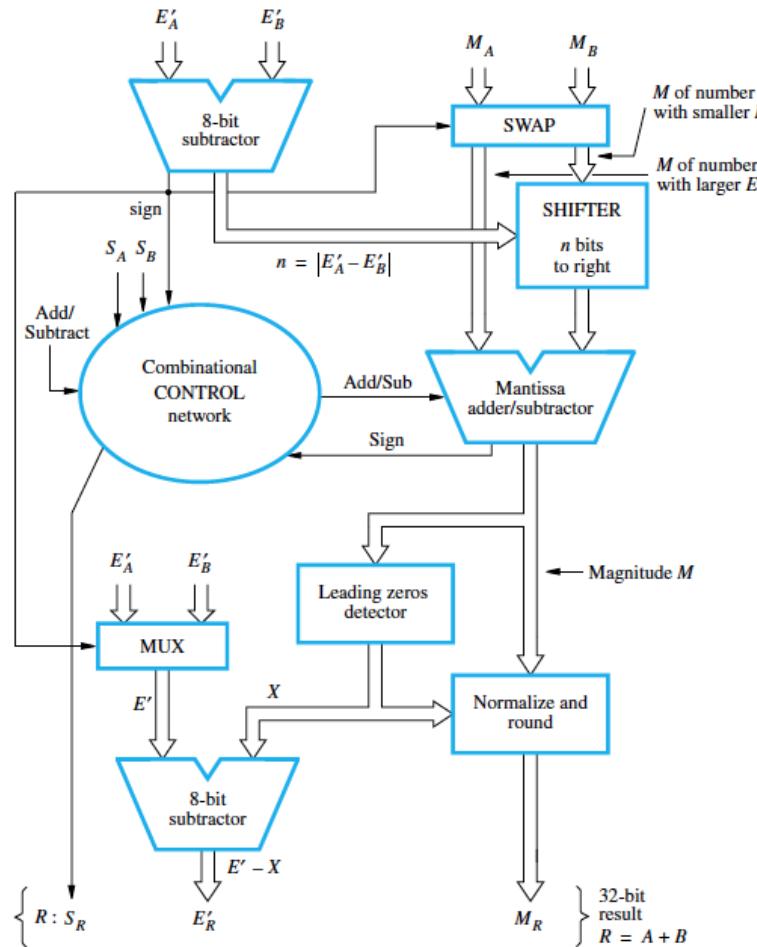
$E'=2047$ and $M \neq 0$ for NaN (e.g. $0/0$ or $\sqrt{-1}$)

$E'=2047$ and $M=0$ for $\pm\infty$

Provides 15 decimal places of precision

Approximate range of $\pm(10^{\pm 308})$

1. Select number with smaller exponent
2. Shift its mantissa right n bits
where n is the difference between the exponents
3. Add/subtract the two mantissas and set sign of result
4. Normalize and round the result if necessary



mantissa is shifted right to align for addition or subtraction

low bits are lost when this is done

Guard, round and sticky bits increase the accuracy of result

sticky bit remains a 1 once a non-zero bit passes through

guard and round retain the two most recent bits shifted out

Rounding of the result can be based on these 3 bits (GRS)

Most systems allow the programmer to specify the rounding mode



Rounding Mode	Description
Round to nearest	Add 1 to significand if GRS > 100 Add 1 to significand if GRS = 100 and LSB of significand = 1
Round towards zero	Truncate (drop bits to right of significand)
Round toward +infinity	Add 1 to significand if the result is positive and either guard or sticky bit = 1
Round toward -infinity	Add 1 to significand if result is negative and either guard or sticky bit =1



Multiply rule:

1. Exponent of product = sum of exponents of factors
(check for exponent overflow)
2. mantissa of product = 1st mantissa times 2nd mantissa
3. Normalize and round result if necessary

Divide rule:

1. Exponent of quotient = dividend exponent – divisor exponent
(check for underflow)
2. Quotient mantissa = dividend mantissa / divisor mantissa
3. Normalize and round result if necessary

- Human readable information must also be represented
 - Characters and text
- ASCII and Unicode code are most common
- Some earlier systems used EBCDIC
- ASCII and EBCDIC are 8-bit codes
 - A single character fits into one byte
 - This allows a maximum of 256 characters
- Unicode uses 2 bytes per symbol
 - Allows many more characters
 - Including user defined symbols

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	:	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	0	5F	_	6F	o		

- Strings are arrays of characters
 - the location of the leading byte is the string address
- MIPS uses lbu (load byte) and sb (store byte)
- MIPS has no instruction to manipulate entire strings
 - Loops containing lbu or sb are used
 - Unlike some CISC machines

- The Unicode code space allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE



- Bytes are the smallest addressable items on MIPS
 - It is a byte-addressable machine
- Access to bits require:
 - Using the byte or word containing the desired bits
 - Using masking operations or shifting to access bits
- Contents of a register or memory word could be interpreted as a series of bits (bit string)

All MIPS machine instructions are 32 bits

Bits are number 0 to 31 from right to left

Three formats are used:

R-type

Arithmetic and logical instructions

I-Type

Instructions using an immediate operand

Same format is used for load word (lw) and store word (sw)

J-type

Contains part of jump address in bits 0 through 25

Used by jump (j) instruction and jal (to call functions)

Each has a 6-bit opcode in bit positions 26 through 31

R-type uses:

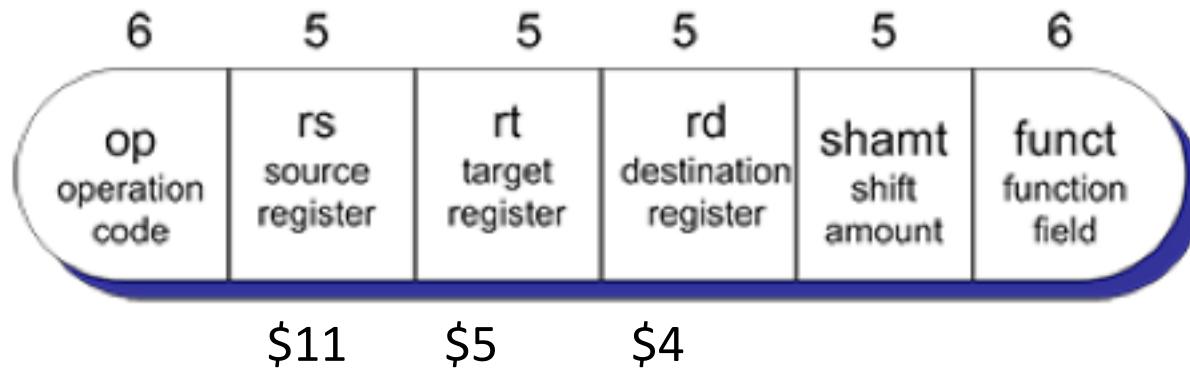
3 register operands

opcode is always 0

function is specified by rightmost 6 bits

shamt field is used with shift instructions

R-type instruction (register operands)



Example: add \$4,\$11,\$5

I-type uses:

immediate operand in rightmost 16 bits

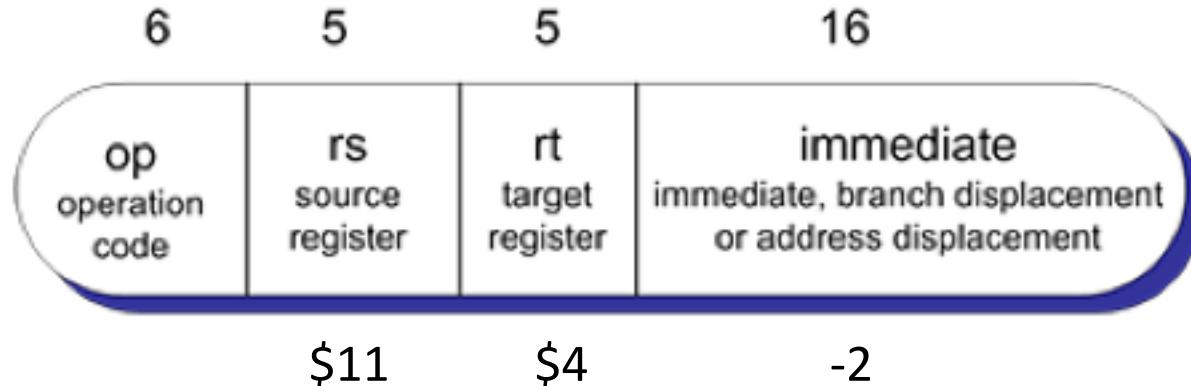
treated as signed operand by arithmetic instructions

treated as unsigned operand by logical instructions

each has a unique opcode

this format is used by lw and sw as well

I-type instruction (immediate)



Example: addi \$4,\$11,-2

Branch instructions also use the I-type format:

rightmost 16 bits = # of instructions to branch

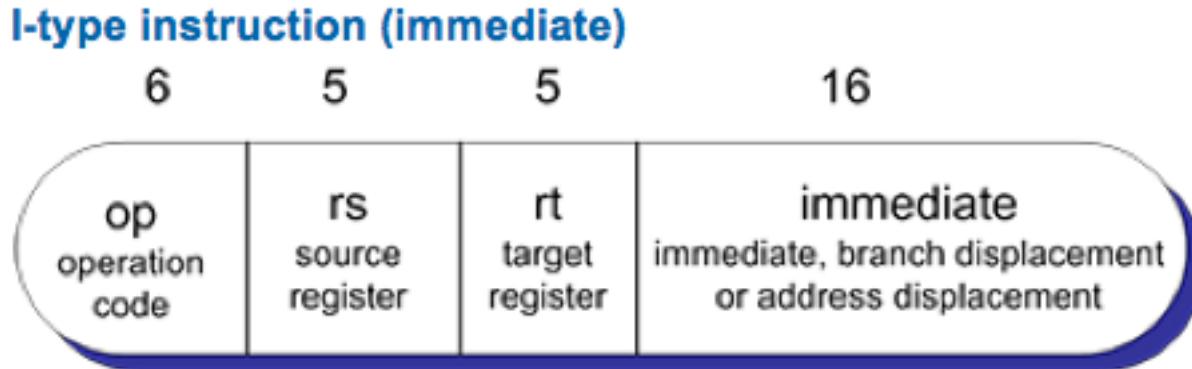
Negative value means branch backwards

Positive value means branch forward

Sign-extended to 32 bits & shifted left 2 bits

then added to the PC to produce branch address

PC points to location following the branch instruction



Example: beq \$4,\$8,exit

J-type uses:

Bits 0 through 25 in generating jump address

Shifted left 2 bits to make it a multiple of 4

Aligned with a 4-byte instruction boundary

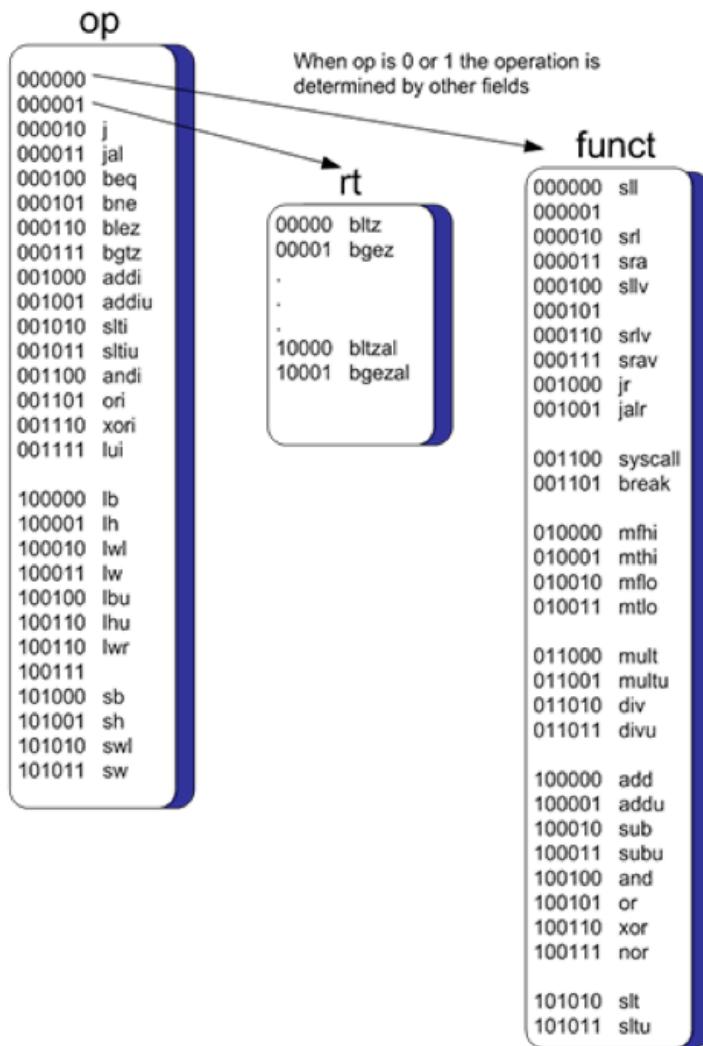
Prepended with 4 upper bits from PC

This yields the full 32-bit jump address

J-type Instruction (jump)



Examples: j exit
jal sqrt



Appendix A contains opcodes for all instructions

All information is represented in binary within the computer

Registers and memory hold information as bit patterns

integers (signed and unsigned)

real numbers (floating point)

instructions

characters

Information is stored using digital devices

These devices can represent 1 (on) or 0 (off)

Hence binary (base 2) is a natural representation

The meaning of the patterns depend on their interpretation

And on the context in which they are used

The same bit pattern can mean entirely different things

MIPS registers and memory words hold 32 bits

The 32-bit pattern can represent:

- a single 32-bit integer (signed or unsigned)
- a single precision floating point number
- a single machine instruction
- a group of 4 ASCII characters

A 32-bit pattern can be written as 8 hex digits:

Example: given the pattern 0x21626364

As an integer it represents **560096100** (decimal)

As floating point it represents **1.7686579 x 2⁻⁶¹**

As a machine instruction it is **addi \$2,\$11,25444**

As ASCII characters it represents **!bcd**

The following videos explain these different encodings

They also review integer and floating point arithmetic

This will serve as a basis for discussing:

The ALU operation and implementation

The design and operation of the floating point hardware

If the representation and bit pattern are given, we know the value

Sign & magnitude

If MSB=0, value = +magnitude ($0110 \rightarrow +110$ i.e. +6)

If MSB=1, value = -magnitude ($1011 \rightarrow -011$ i.e. -3)

One's complement

If MSB=1, flip each bit to get the value ($1011 \rightarrow -0100$ i.e. -4)

same as subtracting from modulus-1 (for 4 bits, modulus-1 = $16-1=15$)

$$(1111 - 1011) = 0100 \text{ or } 15 - 11 = 4$$

If MSB=0, use bit pattern as the value ($0110 \rightarrow +0110$ i.e. +6)

Two's complement

If MSB=1, flip each bit and add 1 to get the value ($1011 \rightarrow -0101$ i.e. -5)

Same as subtracting from modulus ($16 - 11 = 5$)

If MSB=0, use bit pattern as the value ($0111 \rightarrow +0111$ i.e. +7)

Biased is also referred to as excess representation

Representation is value + bias

The threshold (bias) must be specified

The number of available bits (n) & threshold determine the range

Value is recovered by subtracting bias from representation

midpoint (2^{n-1}) as bias gives same range as n -bit two's complement

In general, range = -bias to $(2^n - 1)$ - bias n = number of bits

Not using midpoint as bias gives a lopsided range

All zero bits represent the lower limit $(0 - \text{bias})$

All one bits represent the upper limit $(2^n - 1) - \text{bias}$

Example: if number of bits $n=4$, and bias = $2^3 = 8$ (excess-8)

+5 is represented as $5 + 8 = 13$ (1101)

-3 is represented as $-3 + 8 = 5$ (0101)

Given a bit pattern and the bias, we know the value

Value = representation – bias

1110 → $14 - 8 = +6$ (the pattern is 6 greater than 8)

0110 → $6 - 8 = -2$ (the pattern is 2 less than 8)

Range = -bias to $(2^n - 1)$ -bias (-8 to $15 - 8 = 7$) in this case

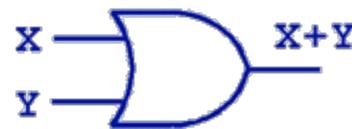
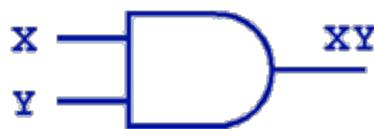


Logic Gates

- A gate is an electronic device that produces a result based on one or more digital input values.
 - In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
 - Integrated circuits contain collections of gates suited to a particular purpose.
- Digital computer circuits employ logic gates to implement Boolean functions.



- The three simplest gates are the AND, OR, and NOT gates.



X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT X

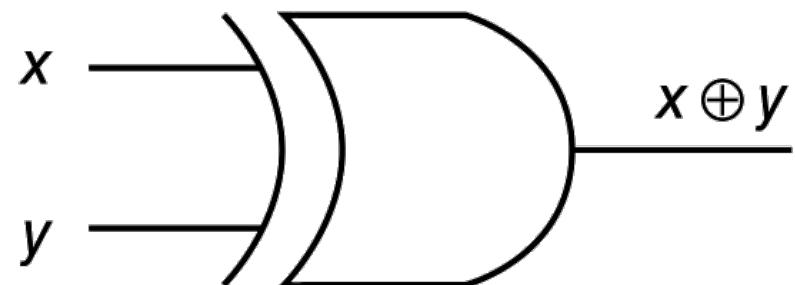
X	X'
0	1
1	0

- They correspond directly to their respective Boolean operations, as shown in their truth tables.



- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.

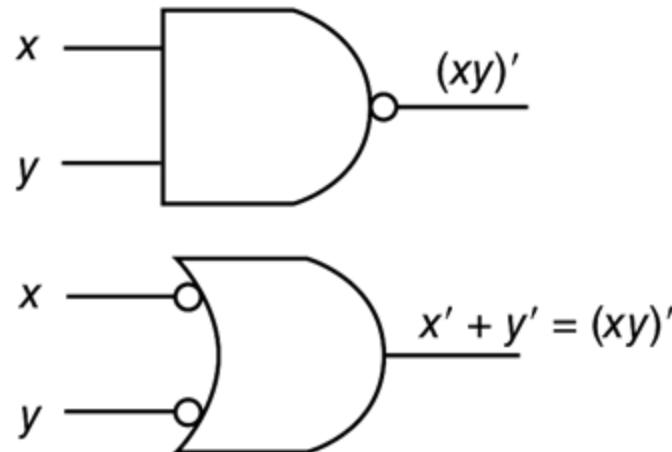
X XOR Y		
X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0



The symbol \oplus denotes the XOR operator.

- Two other important gates are the NAND and NOR gates. The truth table for the NAND gate is shown below:

X NAND Y		
X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

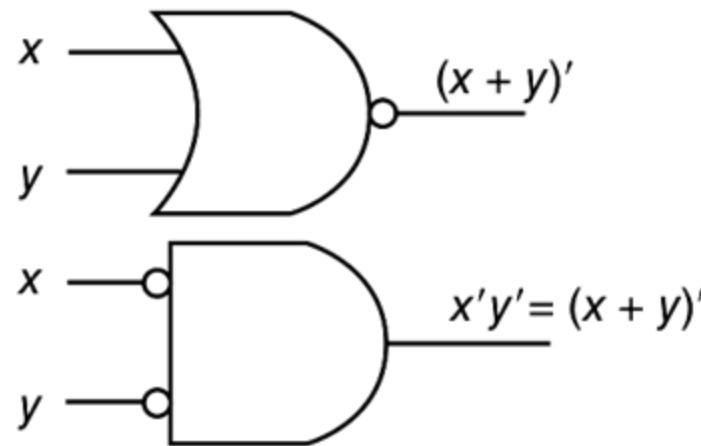


- The open circles are inversion bubbles. The two gates above are equivalent based on DeMorgan's theorem.



- The truth table for the NOR gate is shown below along with two equivalent implementations:

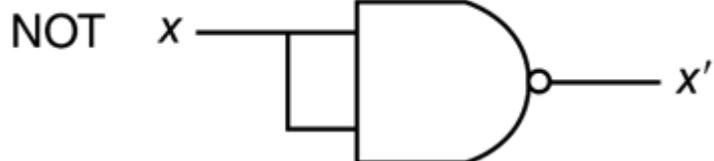
X NOR Y		
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



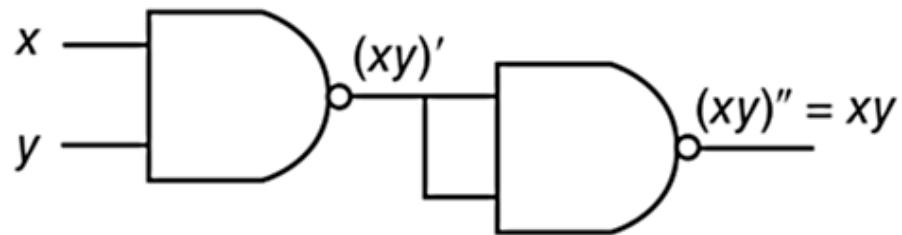
- NAND and NOR gates are said to be *universal* gates because they can be used to implement any logic function.



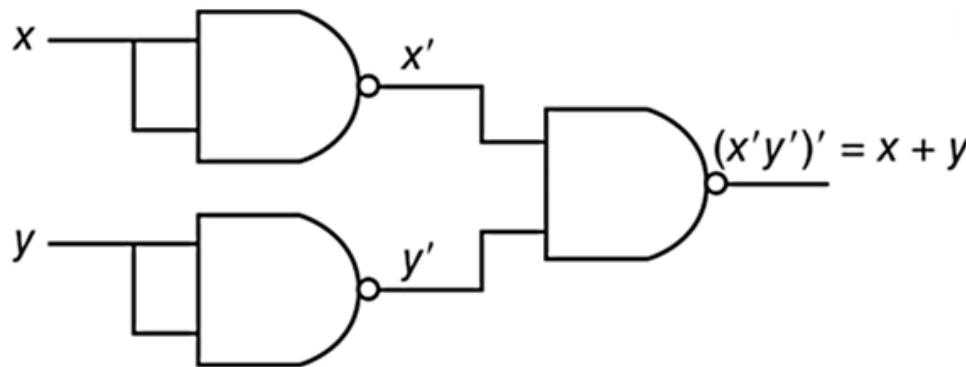
- The examples below show how NAND gates alone can be used to implement NOT, AND, and OR functions:



AND

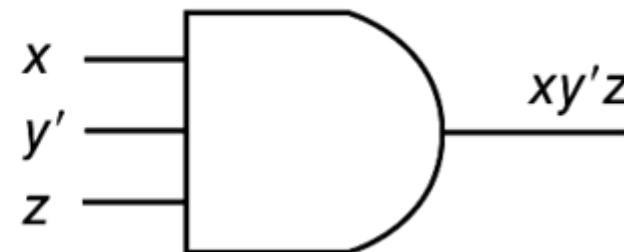
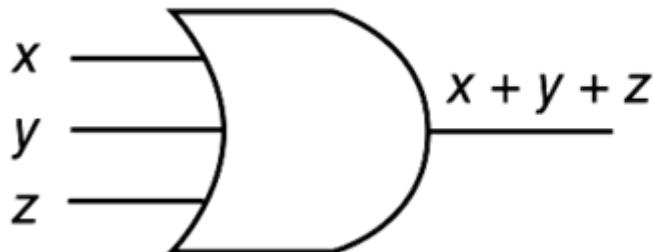


OR

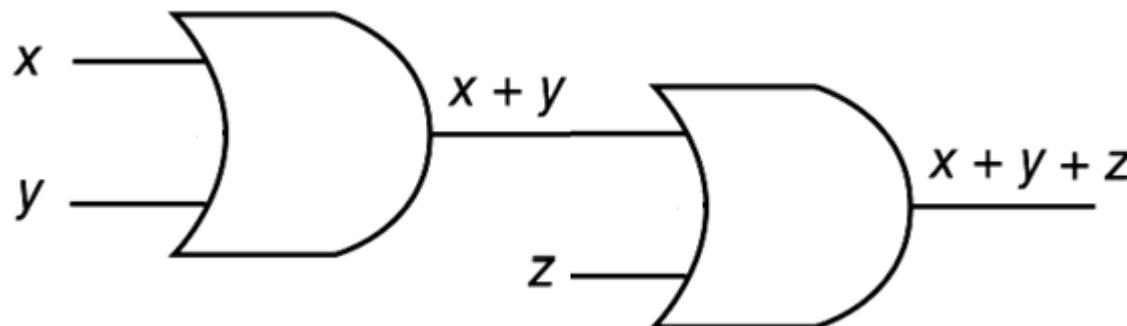




- Here are examples of 3-input gates:

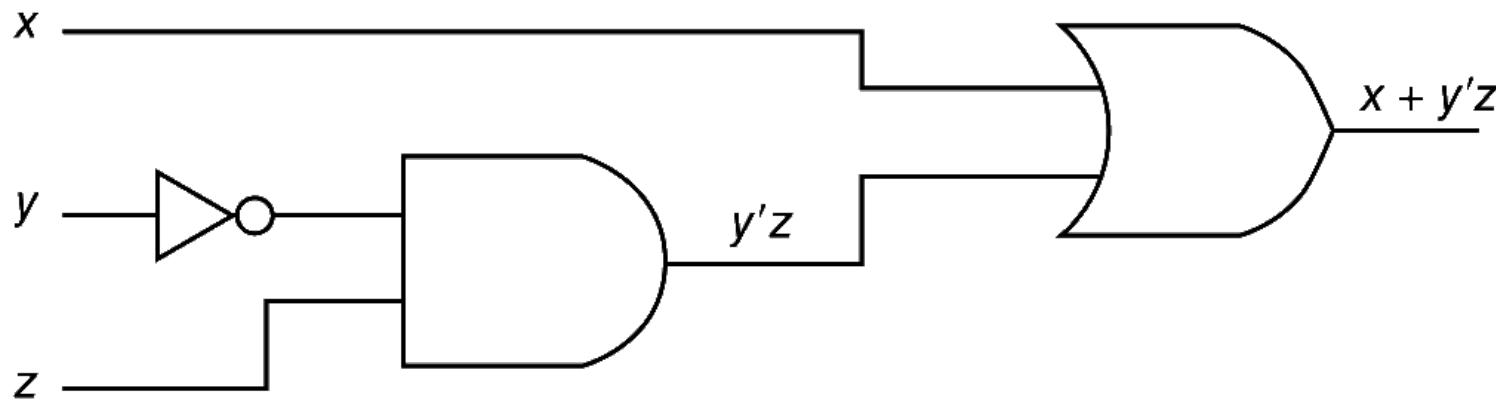


- However the same result can be generated using multiple 2-input gates:



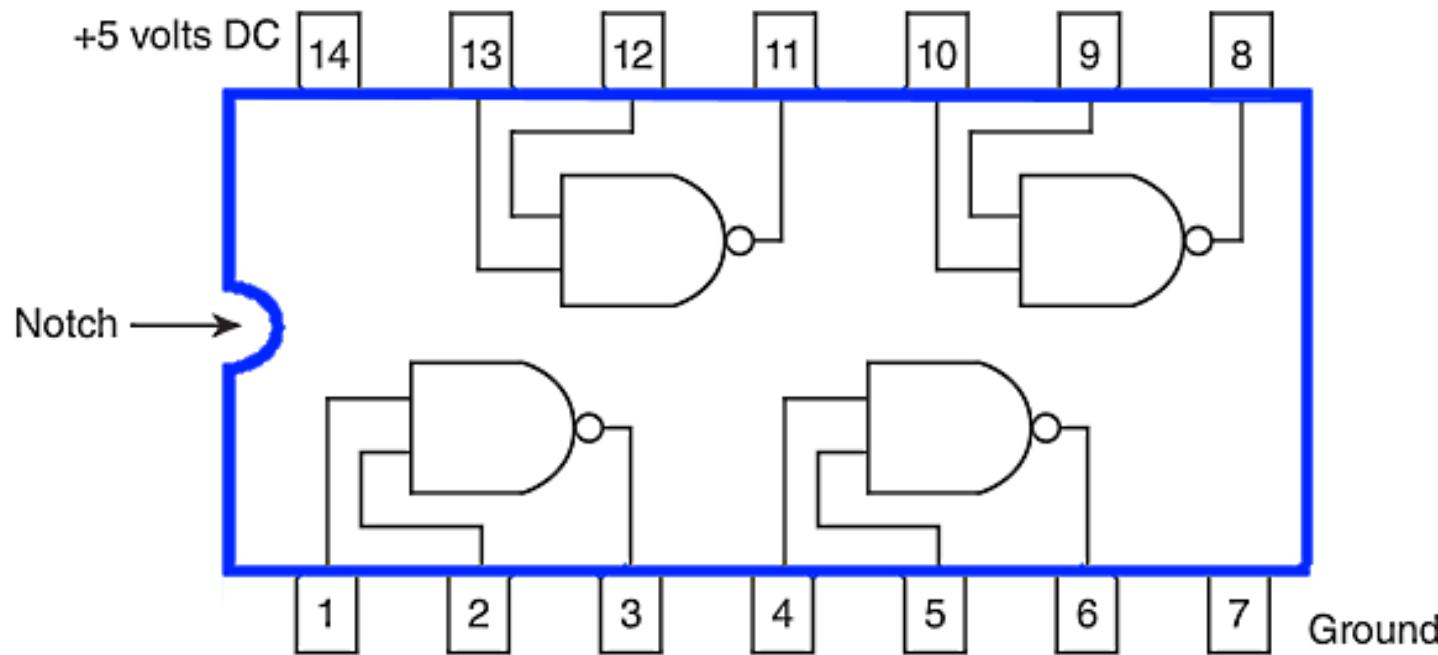


- The circuit below implements the Boolean function $F(x, y, z) = x + y'z$:



- The logic gate implementation of any function can be derived from its truth table. However, the resulting expression should be simplified to minimize the number of gates required.

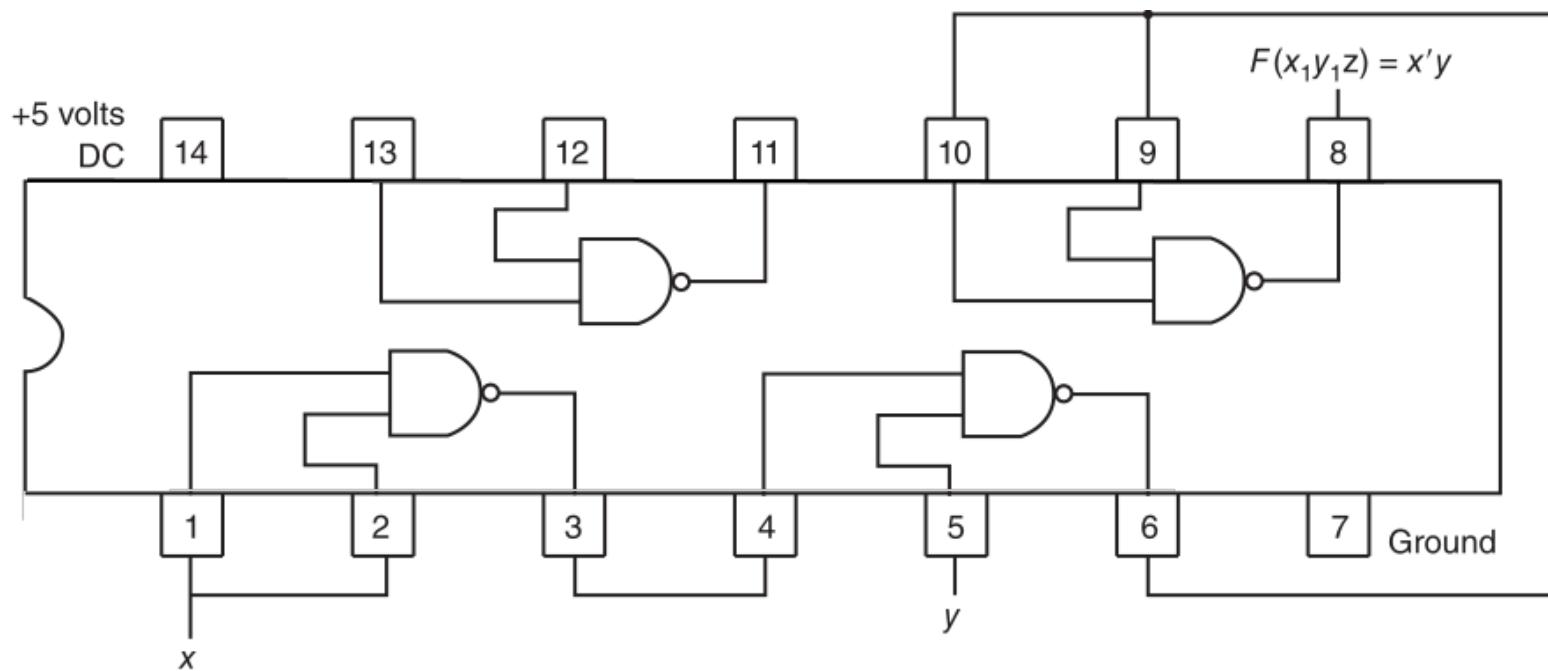
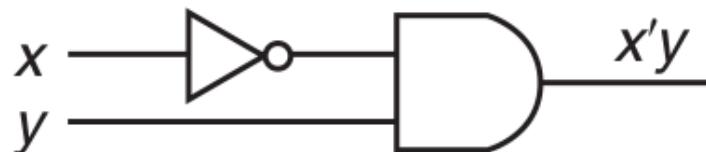
- Standard digital components are combined into single integrated circuit packages.



This is a small scale integrated circuit containing 4 NAND gates.



This chip's pins can be wired as shown below to implement the function:





Suppose we wanted to design a lighting control system that turns lights off when they are not needed.

Assume the lights are to be turned off if it is before 6 PM or if the Sun is out and it is a week day.

The decision would be based on 3 inputs:

$x = 1$ if the time is before 6 PM (or 0 otherwise)

$y = 1$ if the Sun is out (or 0 otherwise)

$z = 1$ if it is a week day (or 0 otherwise)

The output should = 1 if the lights are to be turned off



The truth table for the lights out function is:

Before 6 PM	Sun is out	Week day	Lights out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

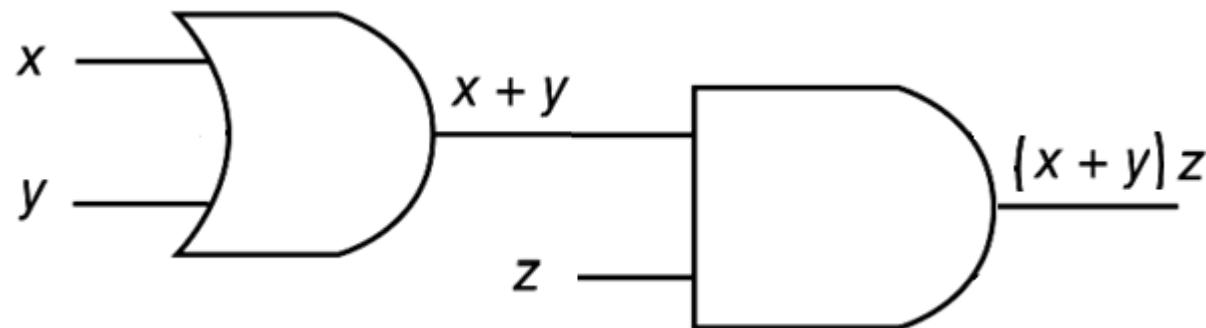
Using the logical sum of the non-zero min-terms we get:

$$\text{lights out} = X' Y Z + X Y' Z + X Y Z$$



Simplifying we get:

$$\begin{aligned}\text{lights out} &= X' Y Z + X Y' Z + X Y Z = (X' Y + X Y' + X Y) Z \\ &= ((X' Y) + X(Y' + Y)) Z = (X' Y + X) Z = (X' + X)(X+Y) Z \\ &= (X + Y) Z\end{aligned}$$

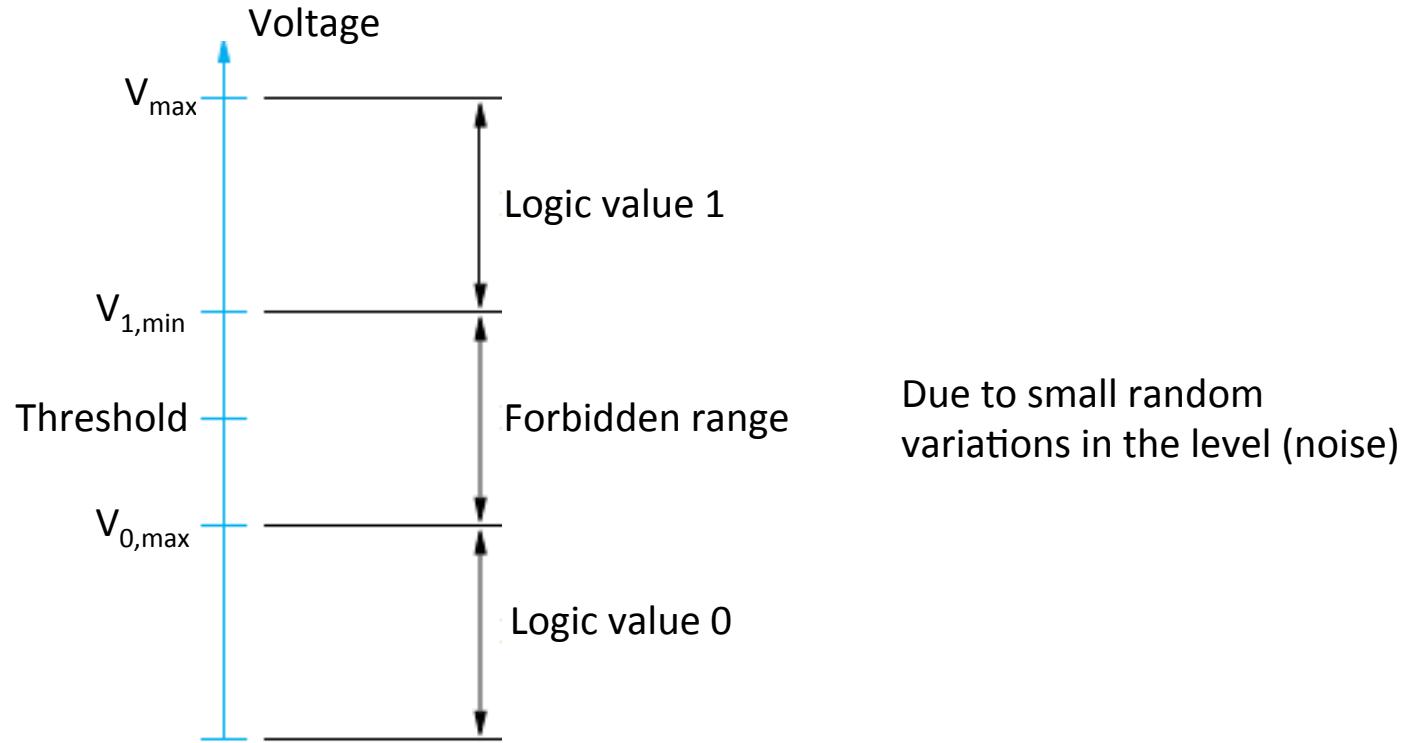


Logic variables have values of 1 and 0, or “high” and “low”

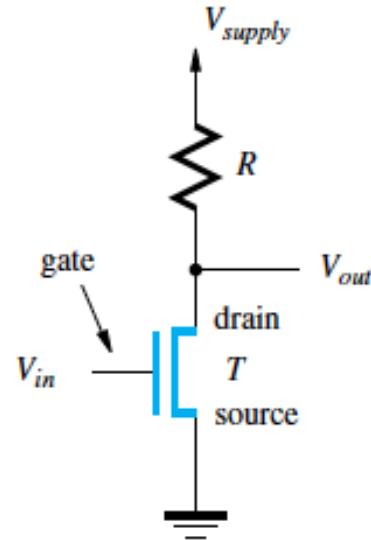
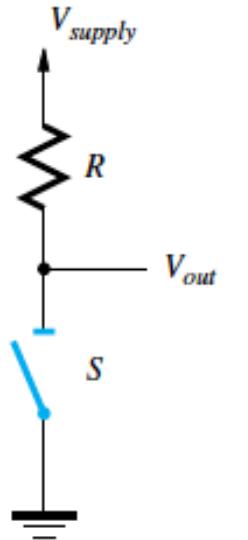
Voltages or currents can represent logic variables

Values above a given threshold represent one state (“on” or “off”)

Values below the threshold represent the opposite state



Digital systems use transistors as switches

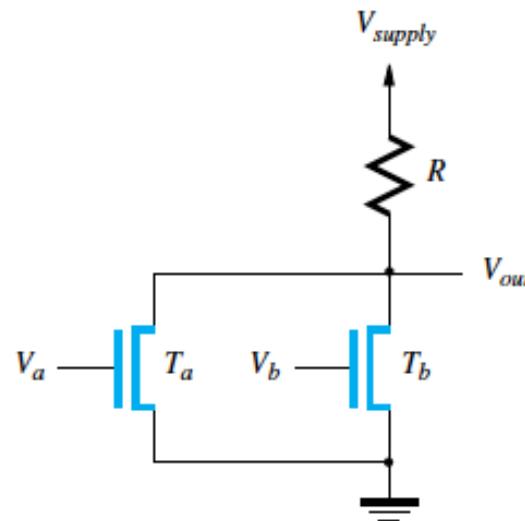
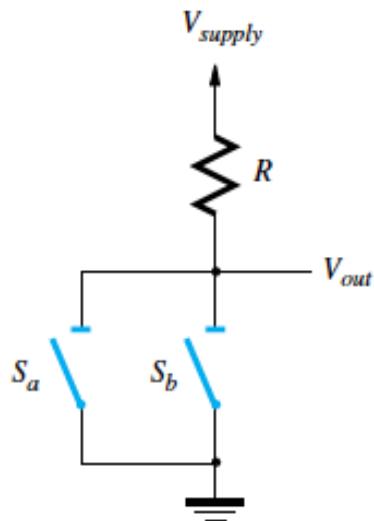


$V_{out} = V_{supply} (1)$ when the switch is open

$V_{out} = 0$ when the switch is closed

Transistor act as an open switch when $V_{in} = 0$

Transistor act as a closed switch when $V_{in} = 1$



Acts as NOR Gate

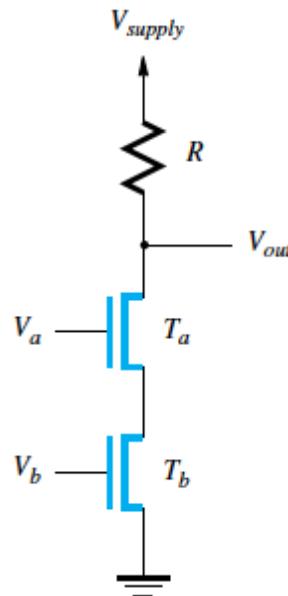
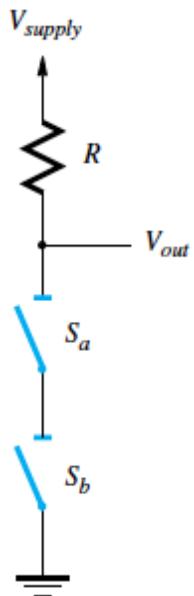
V_a	V_b	V_{out}
0	0	1
0	1	0
1	0	0
1	1	0

$V_{out} = 0$ when either switch is closed

$V_{out} = 1$ when both switches are open

Transistor act as an open switch when $V_{in} = 0$

Transistor act as a closed switch when $V_{in} = 1$



Acts as NAND Gate

V_a	V_b	V_{out}
0	0	1
0	1	1
1	0	1
1	1	0

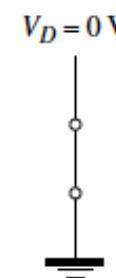
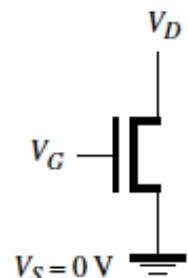
$V_{out} = 1$ when either switch is open

$V_{out} = 0$ when both switches are closed

AND and OR can be implemented from NAND and NOR gates

Just use NOT gate to invert output of NAND or NOR gates

So more transistors are required for AND and OR gates

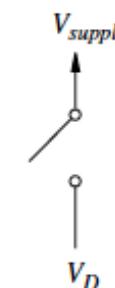
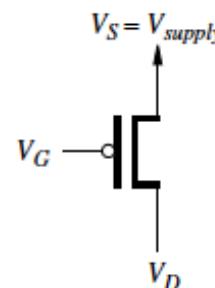


Closed switch
when $V_G = V_{\text{supply}}$

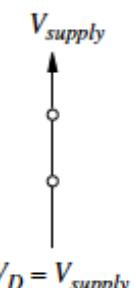


Open switch
when $V_G = 0 \text{ V}$

NMOS Transistor



Open switch
when $V_G = V_{\text{supply}}$



Closed switch
when $V_G = 0 \text{ V}$

PMOS Transistor

Two types of metal-oxide semiconductor (MOS) are available
NMOS-type behave as closed switch when the gate voltage is high
PMOS-type behave as closed switch when the gate voltage is low
Inversion bubble on input denotes PMOS

Source for NMOS transistor is connected to ground (0)
Source for PMOS transistor is connected to V_{supply} (1)

When the switches are closed, current flows

Power is consumed by current flowing through the resistor

More heat results when more power is dissipated

CMOS circuits combine NMOS and PMOS transistors

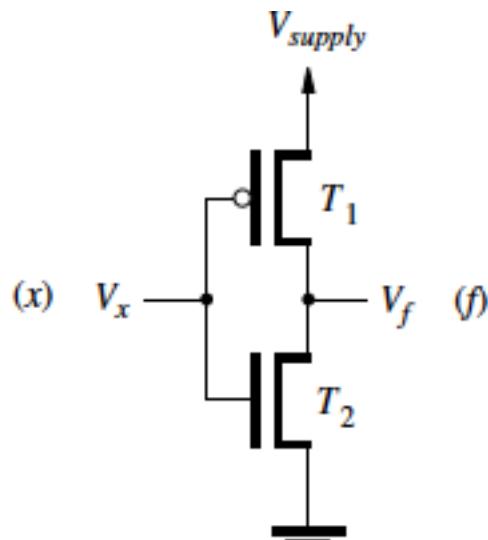
CMOS means complementary metal-oxide semiconductor

CMOS circuits consume less power

MOS transistors occupy a very small area on IC chips

Billions can fit on a single integrated circuit (IC) chip

Smaller transistors can switch at extremely high rates (GHz range)



Circuit

$$(V_f = \text{NOT } V_x)$$

x	V_x	T_1	T_2	V_f	f
0	low	on	off	high	1
1	high	off	on	low	0

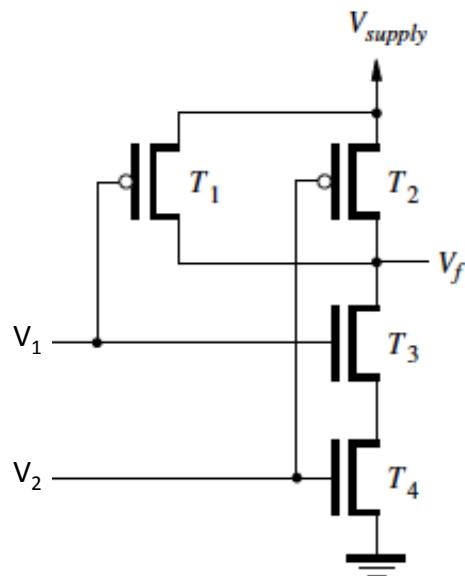
Truth table and transistor states

$V_x=0$ closes T_1 and opens T_2 , pulling V_f up to V_{supply}

$V_x=1$ closes T_2 and opens T_1 , pulling V_f down to 0

So V_f is the complement of V_x

T_1 and T_2 operate in a complementary fashion



Circuit

$$(V_f = V_1 \text{ NAND } V_2)$$

V_1	V_2	T_1	T_2	T_3	T_4	f
0	0	on	on	off	off	1
0	1	on	off	off	on	1
1	0	off	on	on	off	1
1	1	off	off	on	on	0

Truth table and transistor states

If T_1 or T_2 is closed V_f is pulled up to V_{supply}

$V_1 = 0$ closes T_1 ; $V_2 = 0$ closes T_2

If T_3 and T_4 are closed V_f is pulled down to 0

$V_1 = 1$ closes T_1 ; $V_2 = 1$ closes T_2

Dissipates power only when switching



Combinational Circuits

Combinational circuits are those whose output depend only on the current inputs.

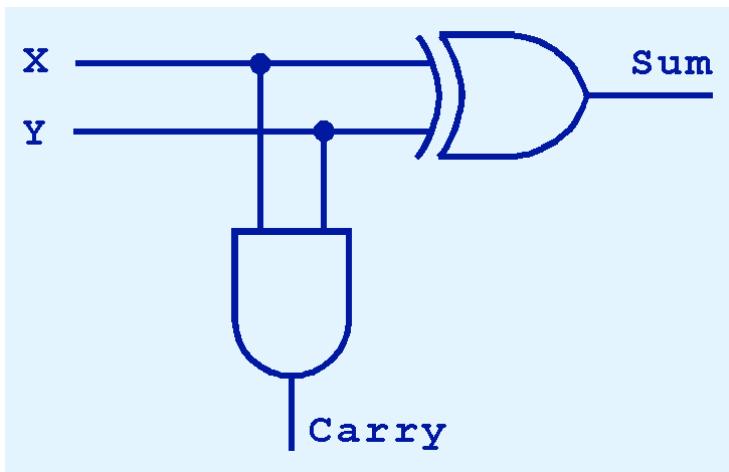
As soon as the inputs change, the output changes (after a short propagational delay)

An example is a half adder :

Inputs		Outputs	
x	y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



- The sum matches the XOR operation and the carry matches the AND operation.



Inputs		Outputs	
X	Y	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This only works for the LSB of the sum.

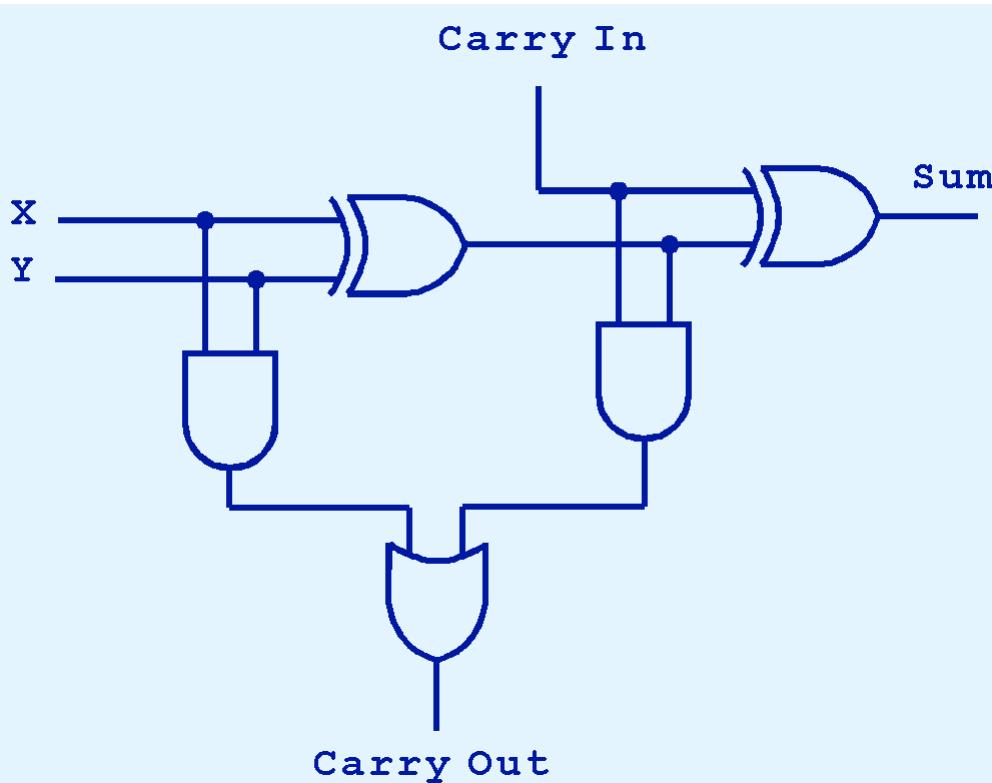


- The half adder becomes a full adder by including gates for processing the carry bit.
- The truth table for a full adder is shown at the right.

Inputs			Outputs		
Carry In			Carry Out		
x	y	Carry In	Sum	Carry Out	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	

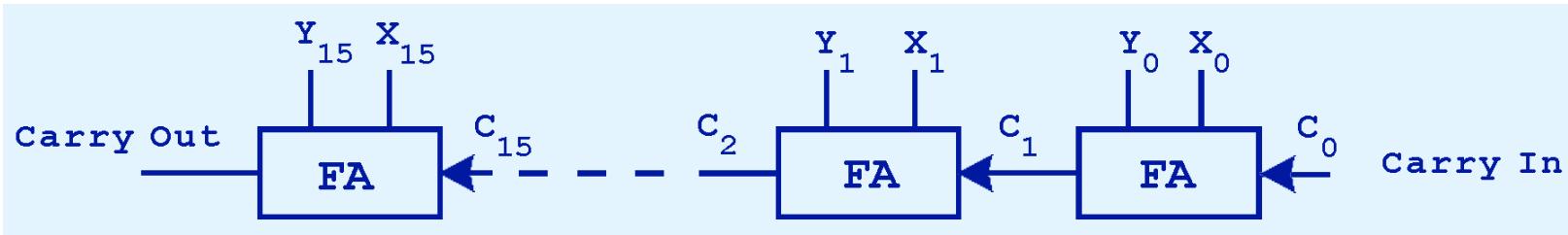


Here's the completed full adder that works for any bit in the sum:



Inputs			Outputs		
X	Y	Carry In	Sum	Carry Out	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	

- Adders of any desired size can be produced by connecting full adders in series.
- The carry bit “ripples” from one adder to the next; hence, this configuration is called a *ripple-carry adder*.



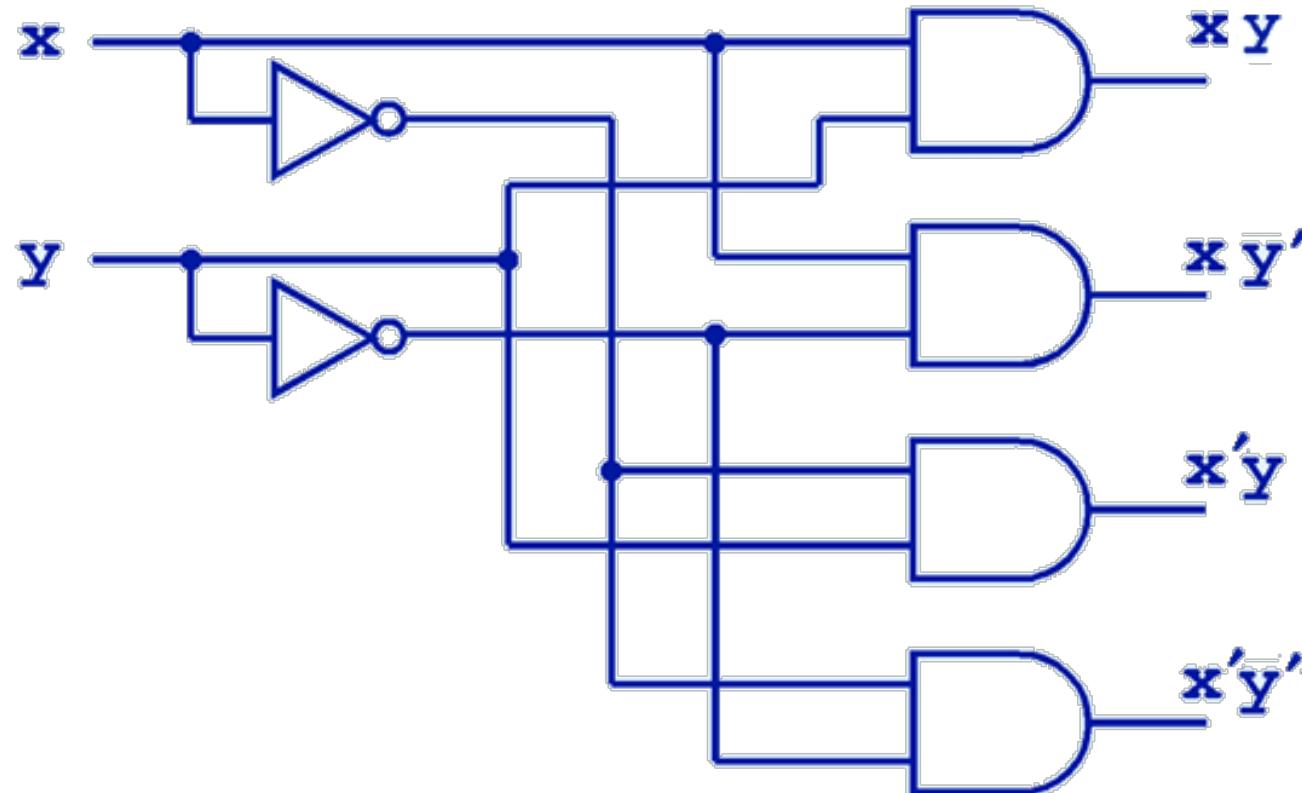
- The full adders must operate sequentially.
- A look-ahead carry adder would be more efficient because it allows the bits in the sum to be computed in parallel.



- Decoders are another important type of combinational circuit.
- Among other things, they are useful in selecting a memory location indicated by a binary value placed on the address lines of a memory bus.
- Address decoders with n inputs can select any of 2^n locations.



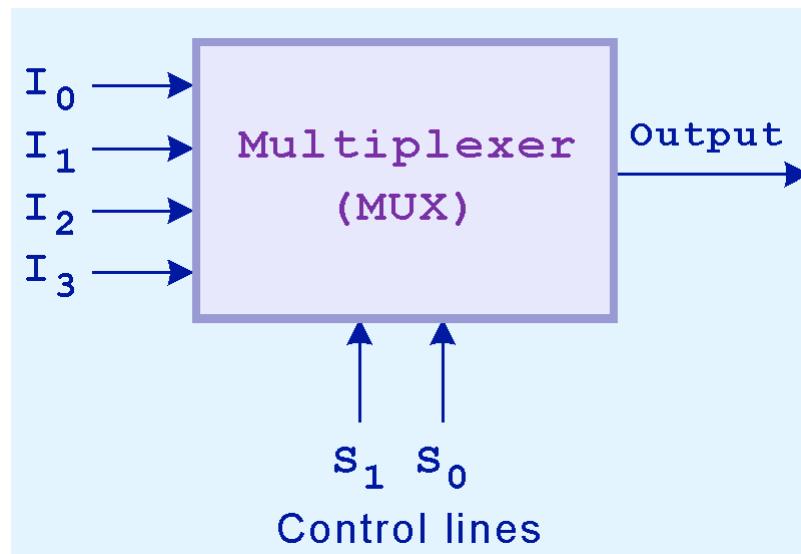
- A 2-to-4 decoder could be implemented as:



The 2-bit number xy is decoded to select one of 4 outputs.



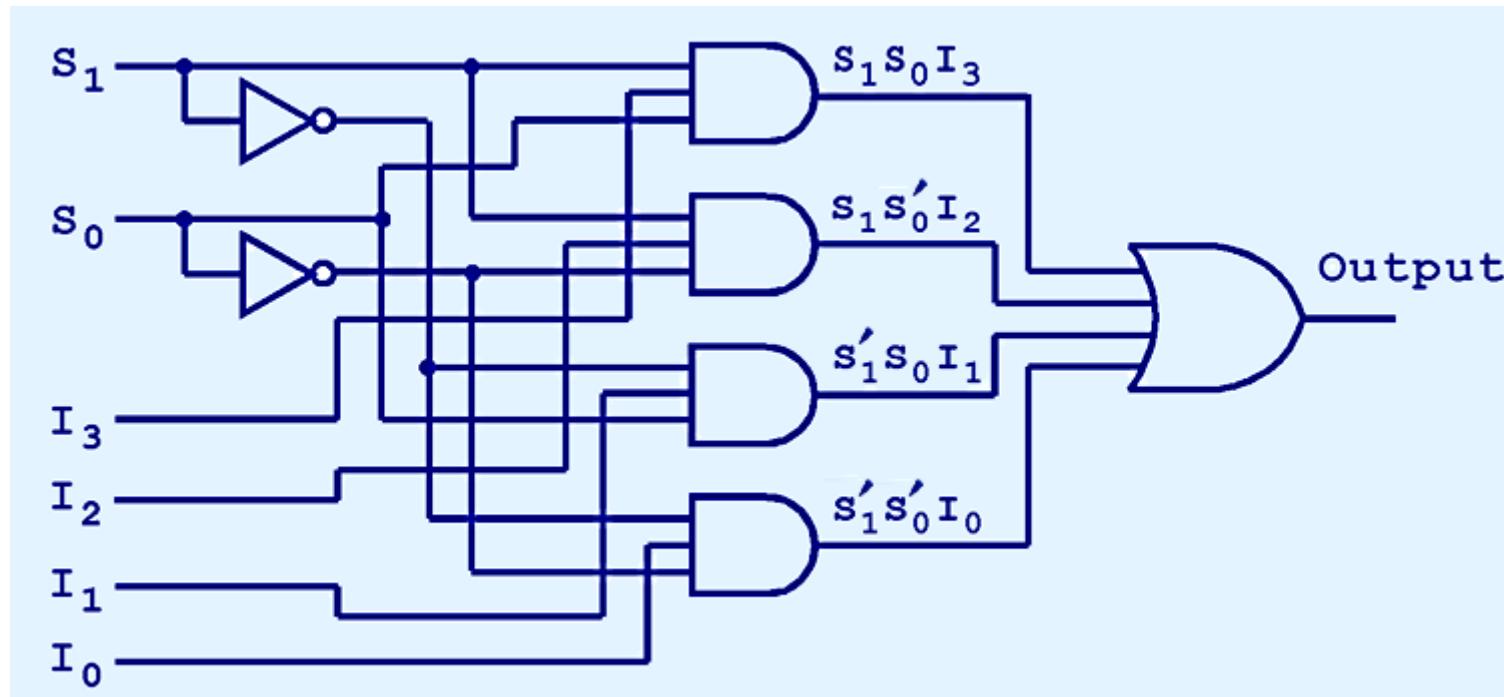
- A multiplexer does just the opposite of a decoder.
- It selects a single output from several inputs.
- The particular input chosen for output is determined by the value of the multiplexer's control lines.
- To be able to select among n inputs, $\log_2 n$ control lines are needed.



**Multiplexers
are also called
selectors.**



- A possible implementation of a 4-to-1 multiplexer is shown below:

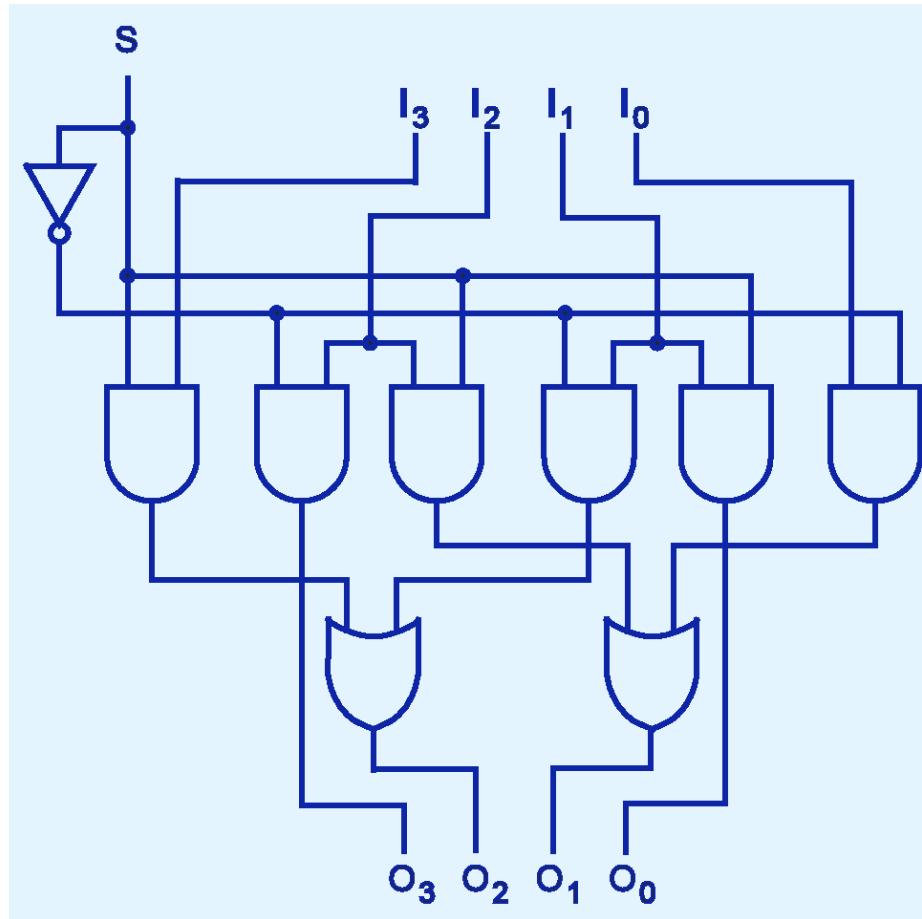


$s_1 s_0$ selects one of 4 inputs I_3 , I_2 , I_1 or I_0 to pass to the output.



- Multiplication and division involve a series of additions, subtractions and shifting operations.
- We have seen how logic gates can add numbers
- Once we see how to use logic gates to perform shifting, we can then compute products, quotients and remainders.
- A shifter moves the bits within a binary pattern one position to the left or right.

One way to implement a shifter is shown below:



$S = 0$ shifts left, $S=1$ shifts right.



- Combinational logic circuits generate outputs that depend only on the current set of inputs.
- *Sequential logic circuits* generate outputs that depend on the previous history of inputs.
 - These circuits have to “remember” their current state.
- *Sequential logic circuits* are used to implement devices such as registers and memories.
- These memory-type devices will be examined later in the course.



Boolean Algebra

Digital computers contain circuits that implement Boolean functions.

Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values (“true” and “false”)

Boolean expressions are created by performing operations on Boolean variables.

Common Boolean operators include AND, OR, and NOT.



Boolean Algebra

In digital systems, these values are “on” and “off,” 1 and 0, or “high” and “low.”

This is why the binary numbering system is a natural basis for digital systems.

Logic circuits take one or more Boolean or digital inputs and generate a digital output.

We will examine how logic circuits can be made up from combinations of basic logic gates.



Boolean Algebra

A Boolean function has:

- At least one Boolean variable,
- At least one Boolean operator, and
- At least one input from the set $\{0, 1\}$.

It produces an output that is also a member of the set $\{0, 1\}$.

Boolean functions can be represented using truth tables and can be implemented using logic gates



The truth table for the Boolean function:

$$F(x, y, z) = xz' + y$$

is shown at the right.

The extra (shaded) columns hold evaluations of subparts of the function.

$$F(x, y, z) = xz' + y$$

x	y	z	z'	xz'	$xz' + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

Like arithmetic operators,
Boolean operators have
rules of precedence.

The NOT operator has
highest priority, followed by
AND and then OR

z' denotes NOT z

xz denotes x AND y

$x + y$ denotes x OR y

$$F(x, y, z) = xz' + y$$

x	y	z	z'	xz'	$xz' + y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1



- The simpler the Boolean functions, the fewer the number of logic gates needed to implement them.
- Simpler circuits are cheaper to build, consume less power, and run faster than complex circuits.
 - Best to reduce our Boolean functions to their simplest form.
- There are a number of Boolean identities that help us to do this.



Most Boolean identities have an AND (product) form as well as an OR (sum) form. We give our identities using both forms. The group below is rather intuitive:

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$



This second group of Boolean identities should be familiar to you from your study of algebra:

Identity Name	AND Form	OR Form
Commutative Law	$xy = yx$	$x+y = y+x$
Associative Law	$(xy)z = x(yz)$	$(x+y)+z = x + (y+z)$
Distributive Law	$x+yz = (x+y)(x+z)$	$x(y+z) = xy+xz$



The next group of Boolean identities are perhaps the most useful.

If you have studied set theory or formal logic, these laws are also familiar to you.

Identity Name	AND Form	OR Form
Absorption Law	$x(x+y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x+y)' = x'y'$
Double Complement Law		$(x)'' = x$



The example below illustrates the use of Boolean identities to simplify a logic expression:

$$F(x,y,z) = xy + x'z + yz$$

$$\begin{aligned} F(x,y,z) &= xy + x'z + yz \\ &= xy + x'z + yz(1) && \text{(Identity)} \\ &= xy + x'z + yz(x + x') && \text{(Inverse)} \\ &= xy + x'z + (yz)x + (yz)x' && \text{(Distributive)} \\ &= xy + x'z + x(yz) + x'(zy) && \text{(Commutative)} \\ &= xy + x'z + (xy)z + (x'z)y && \text{(Associative twice)} \\ &= xy + (xy)z + x'z + (x'z)y && \text{(Commutative)} \\ &= xy(1 + z) + x'z(1 + y) && \text{(Distributive)} \\ &= xy(1) + x'z(1) && \text{(Null)} \\ &= xy + x'z && \text{(Identity)} \end{aligned}$$



- A Boolean expression may have multiple logically equivalent (or “synonymous”) forms
- There are two canonical (i.e. standardized) forms for Boolean expressions:
 - sum-of-products and product-of-sums.
- The Boolean product is the AND operation and the Boolean sum is the OR operation.



In the sum-of-products form, ANDed variables are ORed together.

For example:

$$F(x, y, z) = xy + xz + yz$$

In the product-of-sums form, ORed variables are ANDed together:

For example:

$$F(x, y, z) = (x+y) (x+z) (y+z)$$



To obtain the sum-of-products form using its truth table, we list the values of the variables that result in a true function value (=1).

Each group of variables is then ORed together.

$$\begin{aligned} F(x, y, z) = & (x'yz') + (x'yz) \\ & + (xy'z') + (xyz') + (xyz) \end{aligned}$$

Use the complement of the 0 inputs.
The result is the logical sum of the non-zero “minterms”.

$$F(x, y, z) = xz' + y$$

x	y	z	$xz' + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



To obtain the product-of-sums form using its truth table, we list the values of the variables that result in a false function value (=0).

Each group of variables is then ANDed together.

$$F = (x+y+z)(x+y+z')(x'+y+z')$$

Use the complement of the 1 inputs. The result is the logical product of the Zero “maxterms”.

$$F(x, y, z) = xz' + y$$

x	y	z	$xz' + y$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



It can be more economical to build a circuit using the complement of a function (and complementing its result) than to implement the function directly.

DeMorgan's law provides an easy way of finding the complement of a Boolean function.

Recall DeMorgan's law states:

$$(xy)' = x' + y' \text{ and } (x + y)' = x'y'$$



DeMorgan's law can be extended to any number of variables.

Replace each variable by its complement and change all ANDs to ORs and all ORs to ANDs.

Thus, we find that the complement of:

$$F(x, y, z) = (xy) + (x'y) + (xz')$$

is:

$$\begin{aligned} F' (x, y, z) &= ((xy) + (x'y) + (xz'))' \\ &= (xy)' (x'y)' (xz')' \\ &= (x' + y') (x + y') (x' + z) \end{aligned}$$

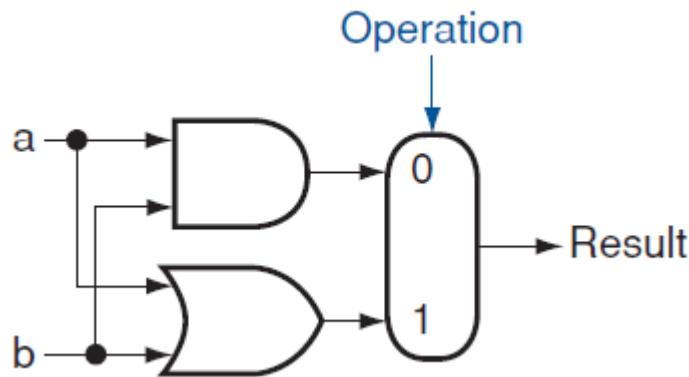


This concludes our examination of Boolean Algebra as a basis for describing the behavior of logic circuits.

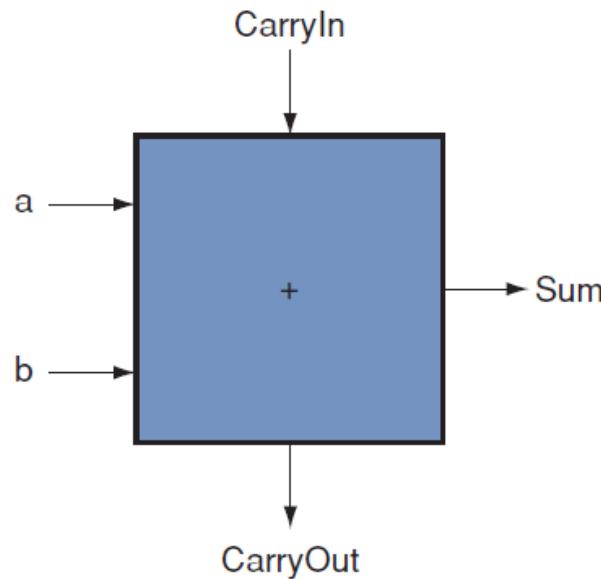
Next we will see how logic gates are used to implement various functions required within the computer.

- The ALU is the brawn of the computer
- Performs integer arithmetic operations
 - Addition and subtraction
 - Multiplication and division
- Performs logical operations
 - AND, OR, XOR
- Acts on commands from control unit

- Multiple 1-bit ALUs can be used to build a 32-bit ALU
- This is called a bit sliced design
- The AND and OR operations map directly to gates



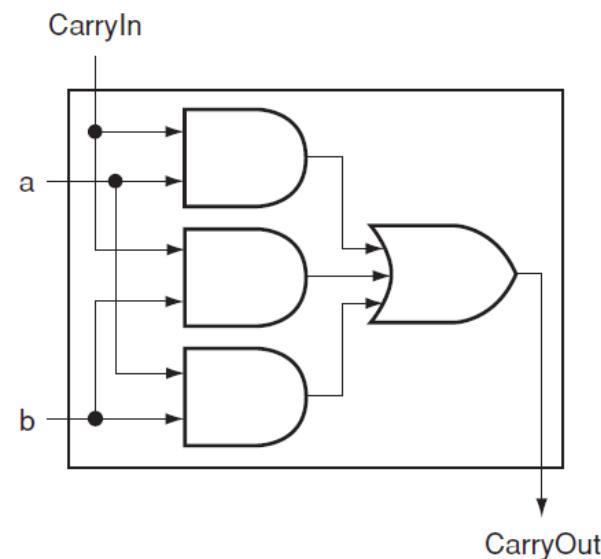
- From module 3 we know how to build a full adder



- The single bit inputs (a and b) together with the Carryin are added to produce the Sum and Carryout

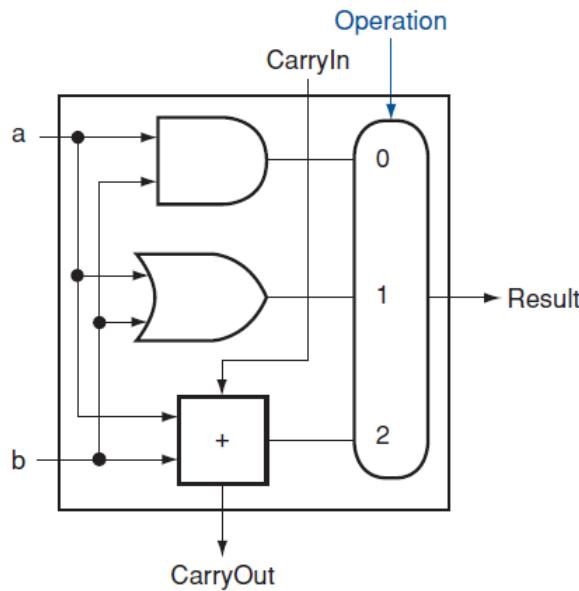
- Truth table and possible circuit to generate CarryOut

Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1



$$\begin{aligned}\text{CarryOut} &= (a \cdot \text{CarryIn}) + (a \cdot b) + (b \cdot \text{CarryIn}) \\ &= (a \cdot b) + (a + b) \cdot \text{CarryIn}\end{aligned}$$

- 1-Bit ALU containing AND gate, OR gate and Full adder



- Full 32-Bit ALU can be built up from multiple copies

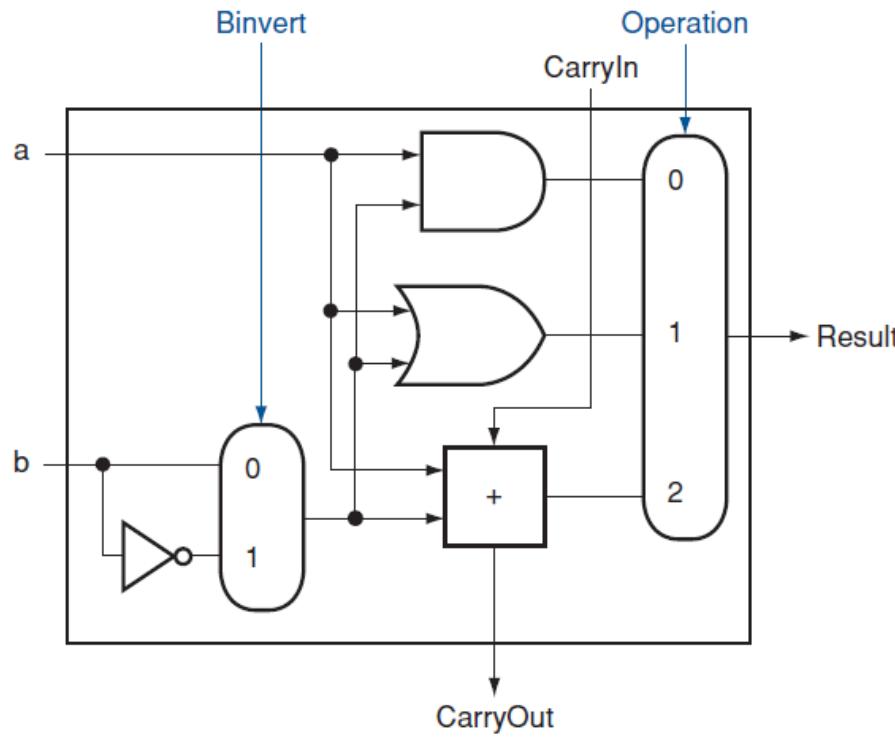
Subtraction can be included by noting that:

$$a + \bar{b} + 1 = a + (\bar{b} + 1) = a + (-b) = a - b$$

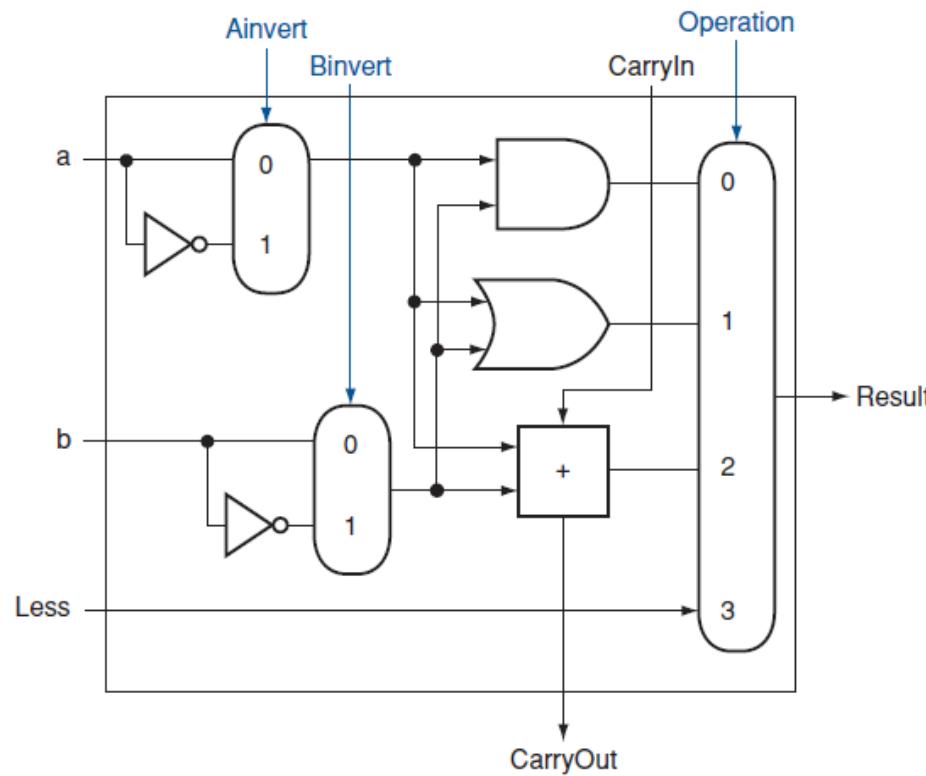
$\bar{b} + 1$ is the negative of b

Subtract b from a by adding the negative of b to a

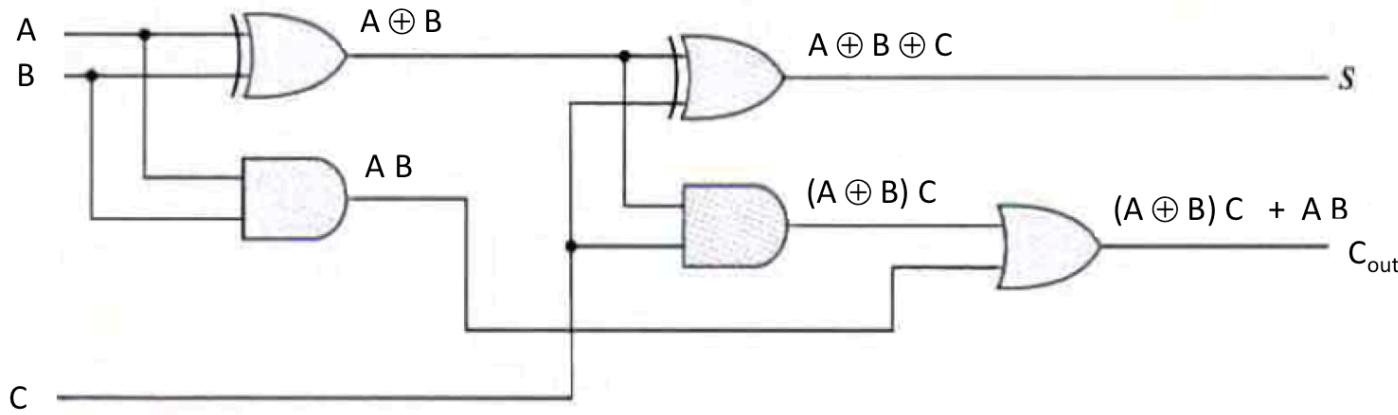
Uses two's complement of b



1-Bit ALU that subtracts, adds and performs AND and OR



Ainvert allows the computation of $(b - a)$ and other functions

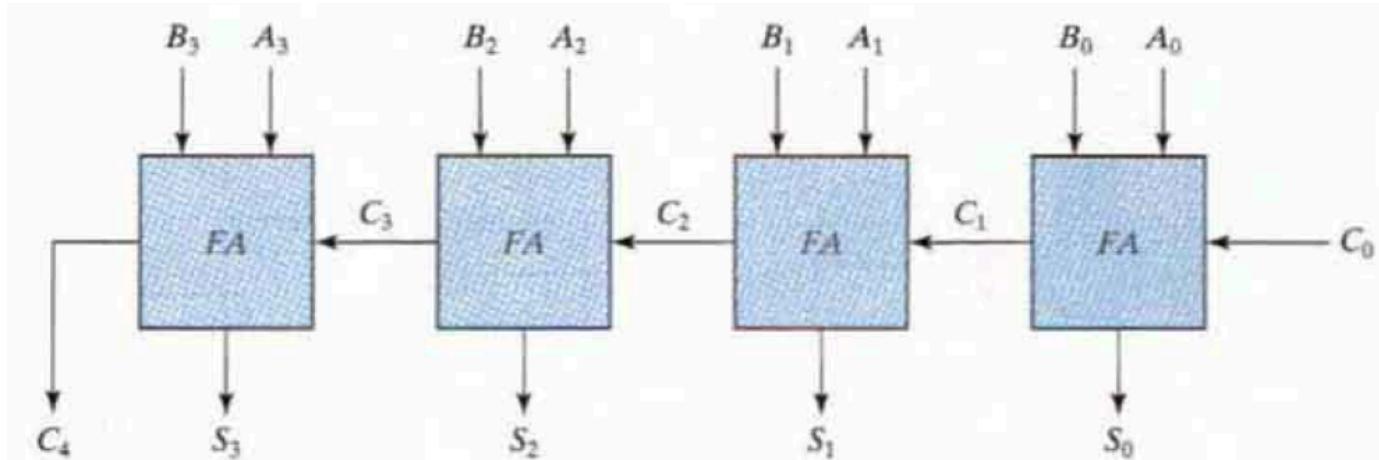


$A \oplus B$ and AB can be generated in parallel for each position
 S is then generated as $A \oplus B \oplus C$ (where C is the carry-in)

Hence S_0 , the LSB of the sum requires 2 gate delays

C_{OUT} is available after 3 gate delays

Other sum bits need the carry from the bit position on the right
 $S = A \oplus B \oplus C$, so another gate delay is needed once C is available
 $C_{out} = (A \oplus B)C + AB$, and requires 2 more gate delays



S_0 is available after 2 delays

C_1 is available after 3 delays

S_1 is available after $1 + 3 = 4$ delays

C_2 is available after $2 + 3 = 5$ delays

S_2 is available after $1 + 5 = 6$ delays

C_3 is available after $2 + 5 = 7$ delays

S_3 is available after $1 + 7 = 8$ delays

C_4 is available after $2 + 7 = 9$ delays

Recall that for the full adder:

$$\text{CarryOut} = (a \cdot b) + (a \oplus b) \cdot \text{CarryIn}$$

If a and b are both 1, they generate a CarryOut when added

If either a or b is 1, Carryin is propagated to CarryOut

In general: $c_{i+1} = (a_i \cdot b_i) + (a_i + b_i) \cdot c_i = g_i + p_i \cdot c_i$

$g_i = a_i \cdot b_i$ and $p_i = a_i \oplus b_i$ the subscript indicates the bit position

C_0 is the input carry for bit 0, the LSB

$C_0 = 0$ for addition

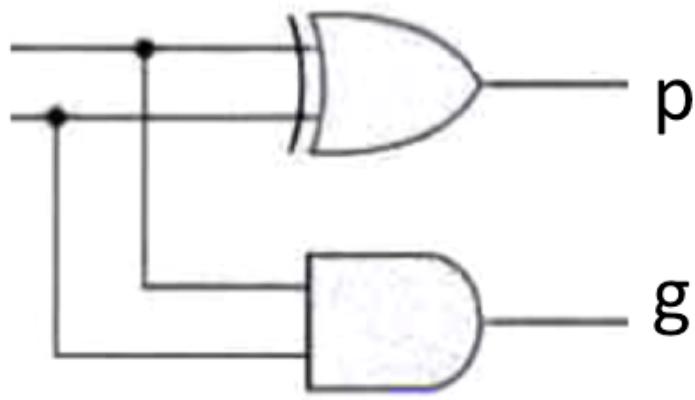
$C_0 = 1$ for subtraction

recurrence relation for all of the output carries:

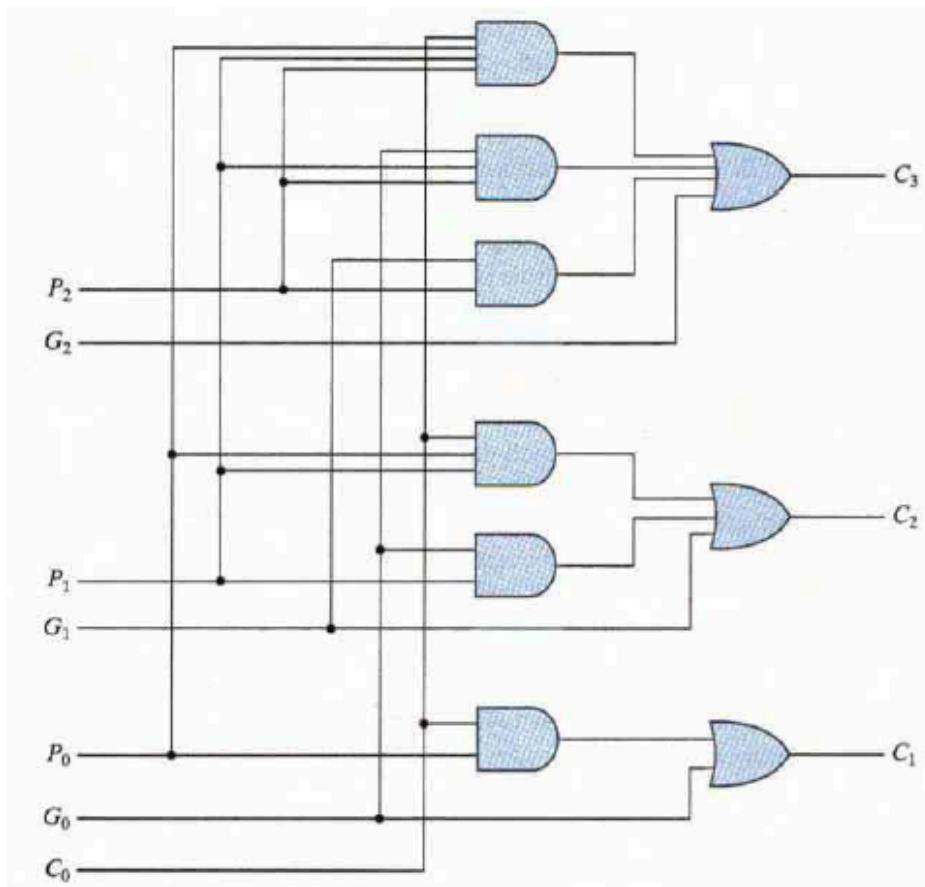
$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1 \cdot c_1$$

$$c_i = g_{i-1} + p_{i-1} \cdot c_{i-1} \quad (\text{for } i > 0)$$



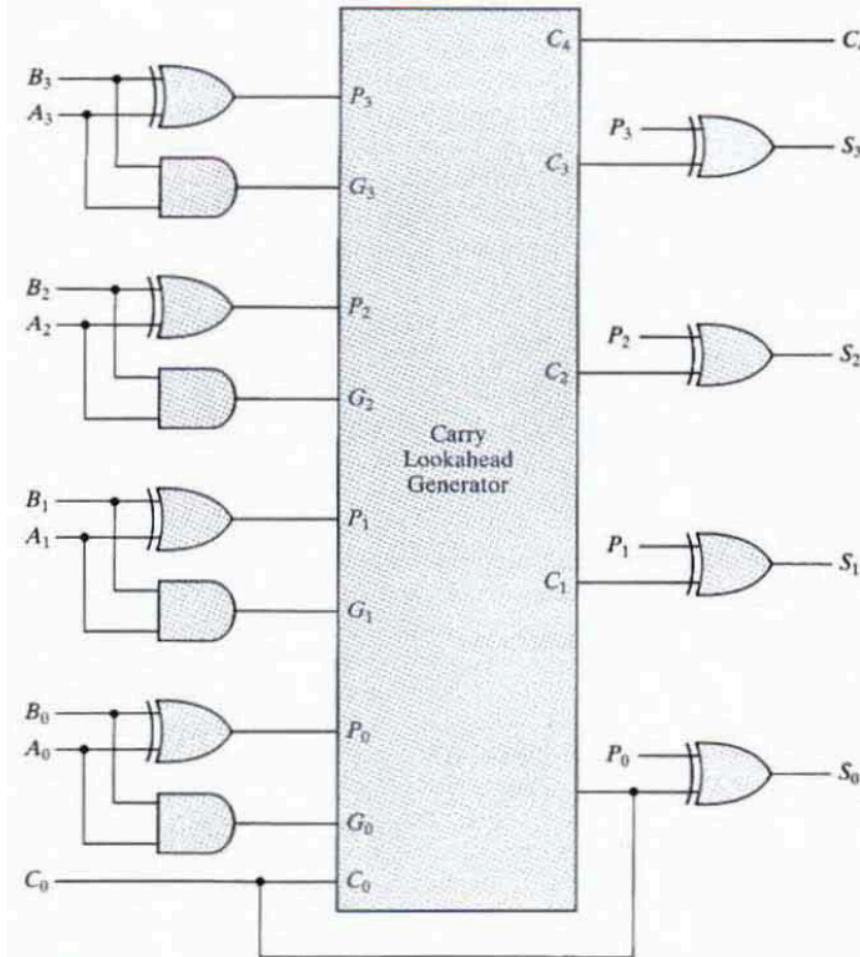
Inputs are the data bits for each position



Propagate & generate bits are used to produce carry bits

Carries require 2 gate delays and are produced in parallel

Lookahead carry generator circuit

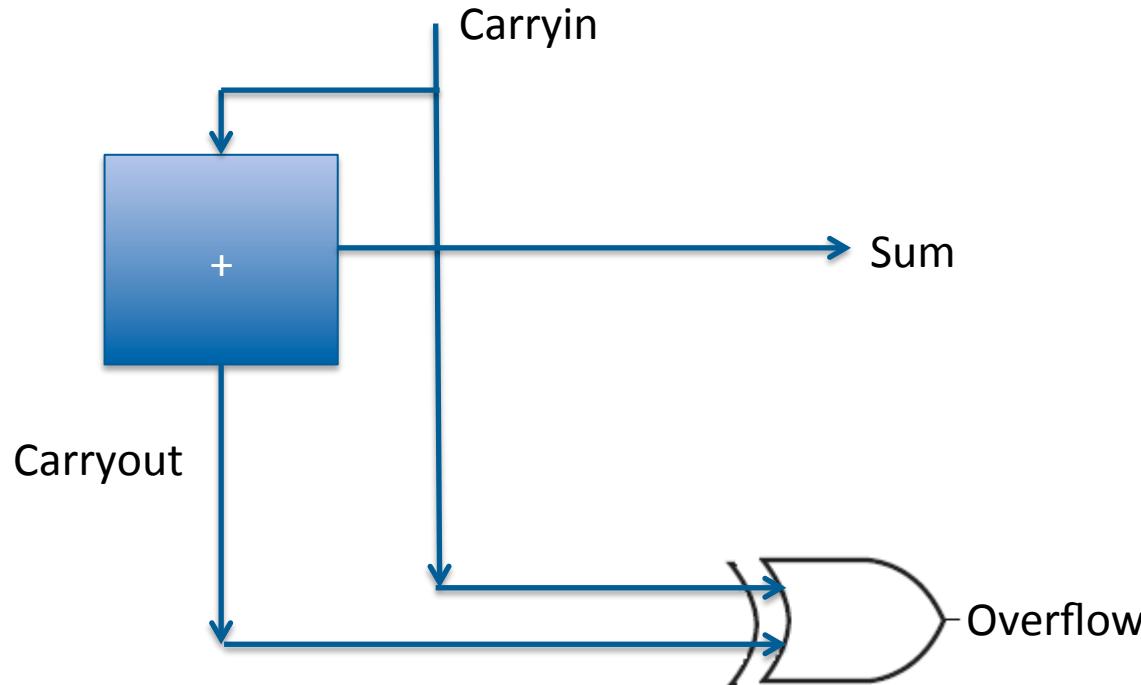


Bits in sum are available after
 $1+2+1 = 4$ gate delays

4-bit adder with lookahead carry

- c_0 and all of the a_i and b_i bits are known up front
- All of the p_i and g_i can be generated in parallel (1 delay)
- All carry bits c_i are generated in parallel (2 more delays)
- All sum bits are produced in parallel (1 more delay)
- This is faster than the ripple carry adder
- Ripple carry adder computes each sum sequentially from LSB to MSB

- Overflow exists if the result is larger than the register
- Mismatching carry-in and carry-out of the MSB indicates overflow
- Carry out alone signals overflow only for unsigned numbers
- Signed arithmetic requires a separate overflow indicator bit
- Negative results have MSB = 1, if no overflow occurs
- Positive results have MSB = 0, if no overflow occurs

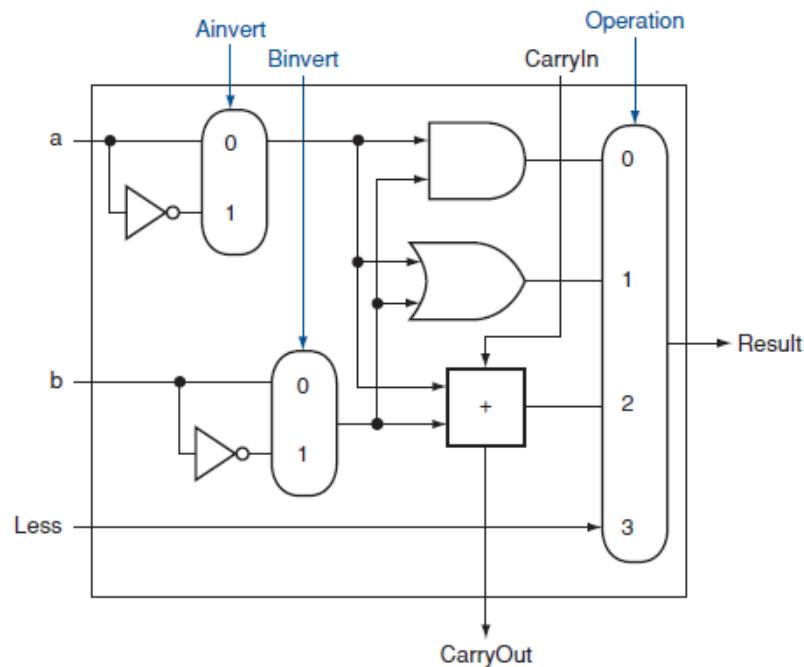


For MSB, compare carryin with carryout

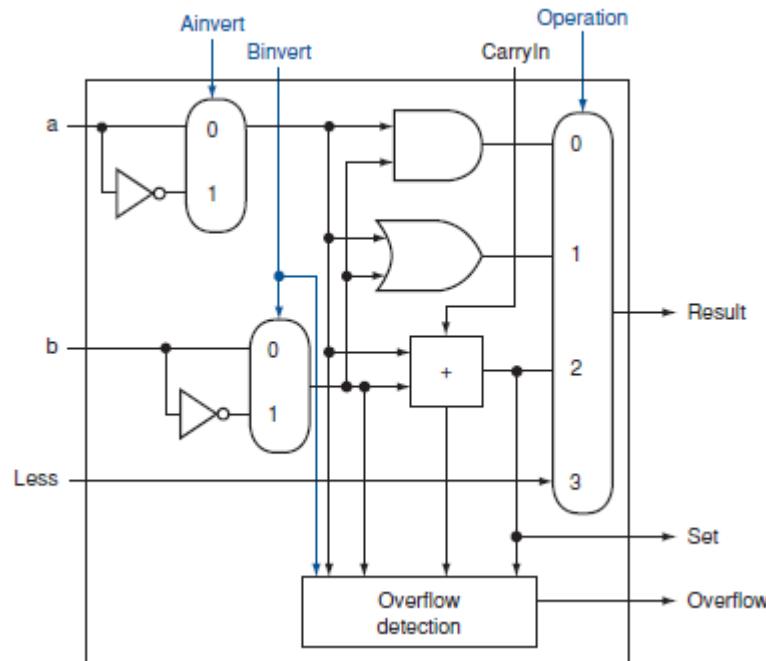
Mismatch indicates signed overflow

- The ALU must also support slt (set on less than) instruction
- The slt instruction generates 1 in the result if $a < b$
- Otherwise, it generates 0
- If $a-b < 0$, then $a < b$ (indicated by sign bit in result)
- Assumes no overflow occur

- A new input, Less, is included for ALU
- Less = 0 for the high order 31 bit positions
- LSB of result = 1 if $a < b$, otherwise LSB = 0
- Logic is only needed in the ALU for the MSB to generate both Overflow and Set (replaces LSB of result)



ALU for all but the MSB position

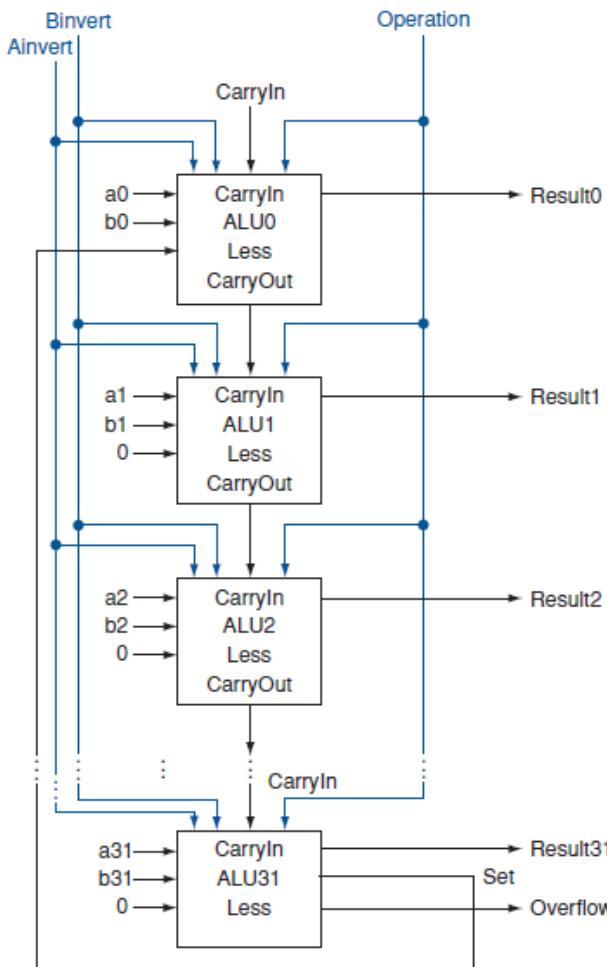


ALU for the MSB position

Includes logic to generate Set bit for slt instruction
Also generate overflow flag

Includes support for slt

Set from MSB routed
back to LSB of result



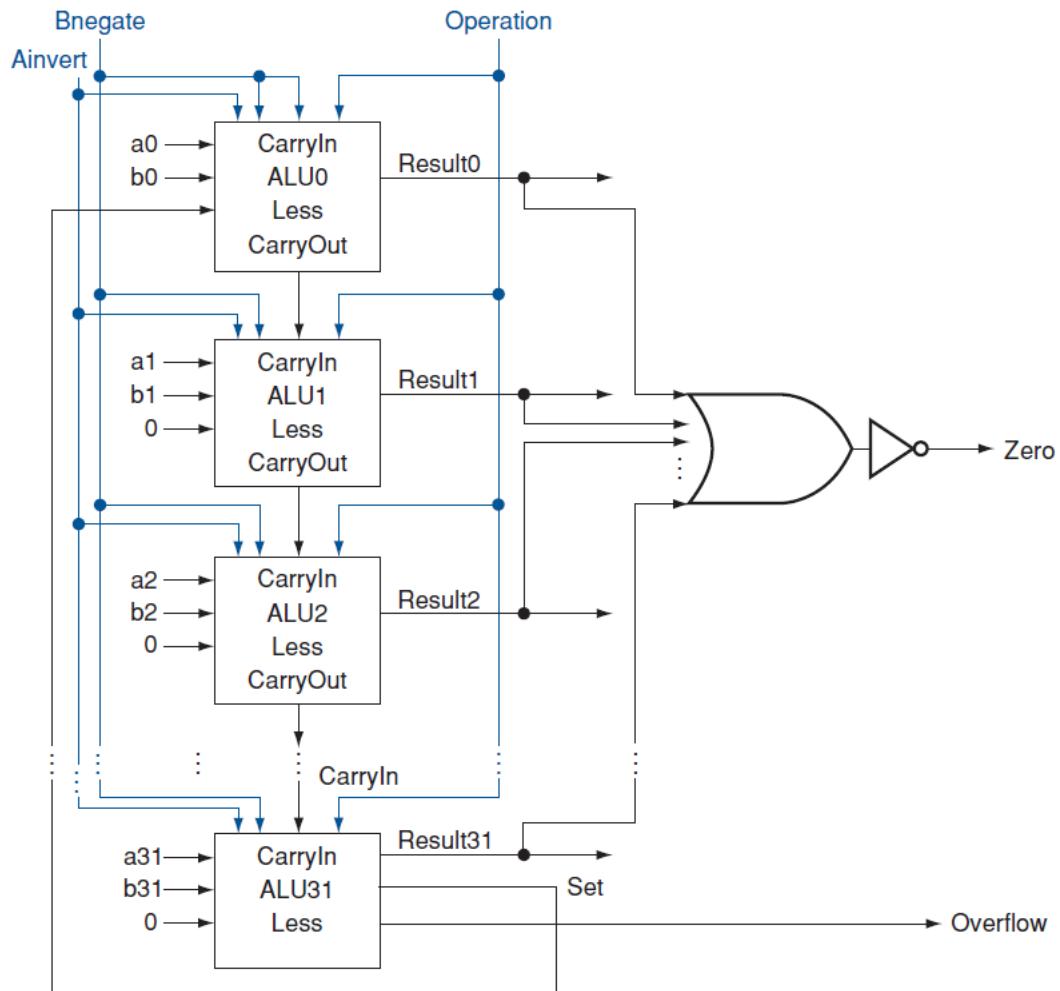
- Branch on equal (beq) requires zero flag output from ALU
- Branch is taken if zero flag = 1 (PC is loaded with target address)
- Otherwise PC+4 is used as address of next instruction
- If $a-b = 0$, a and b are equal, so zero flag is set to 1
- NOR of all result bits yields zero flag

$$\text{Zero} = \overline{(\text{Result}31 + \text{Result}30 + \dots + \text{Result}2 + \text{Result}1 + \text{Result}0)}$$

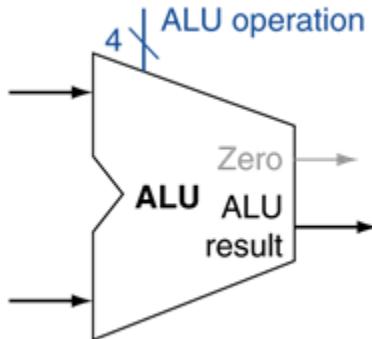
Final 32-bit ALU

Bnegate negates b
(Carryin for LSB = 1 and all
b bits are inverted)

So $a - b$ is computed



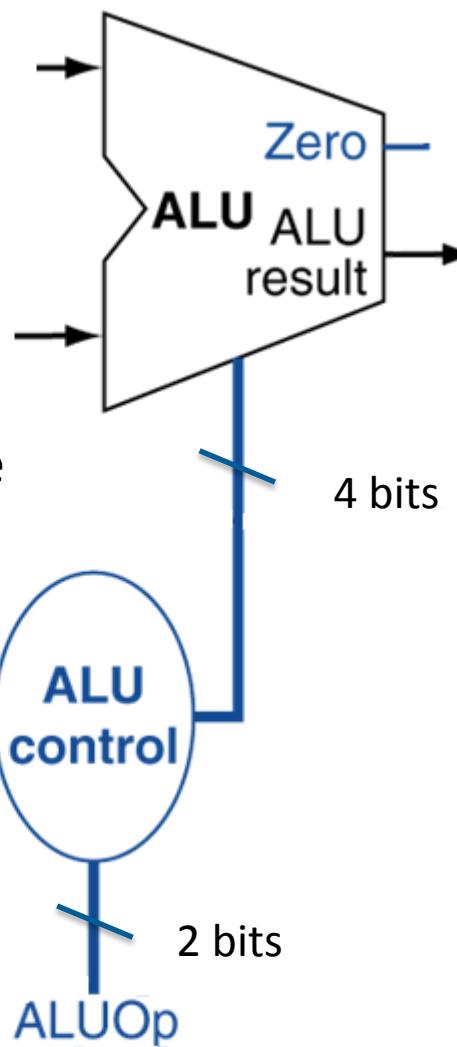
Supports all operations needed for the core MIPS instruction subset



- The ALU takes two input operands
- Generates result as directed by 4-bit control signal
- Sets zero flag if result is 0
- 4-bit control signal derived from opcode & function code

The 4-bit ALU control is derived from ALUOp1 & ALUOp0

If ALUOp = 10, then 6-bit function code is also used





- Instruction type determines ALU operation
 - Load/Store: operation = add
 - Branch: operation = subtract (compare operands)
 - R-type: operation depends on funct field

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

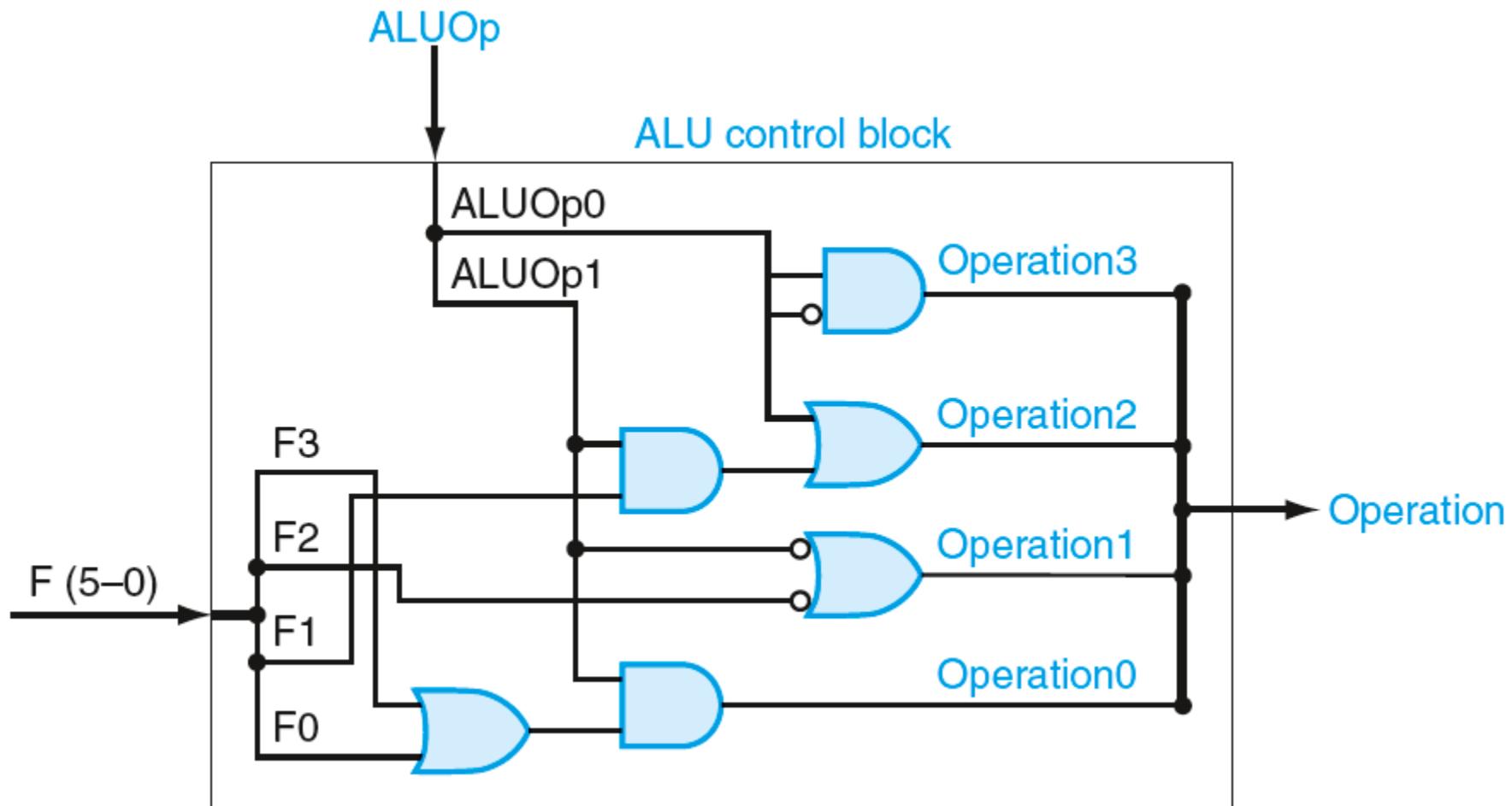
- Combinational logic derives the 2 ALUOp bits from opcode

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111



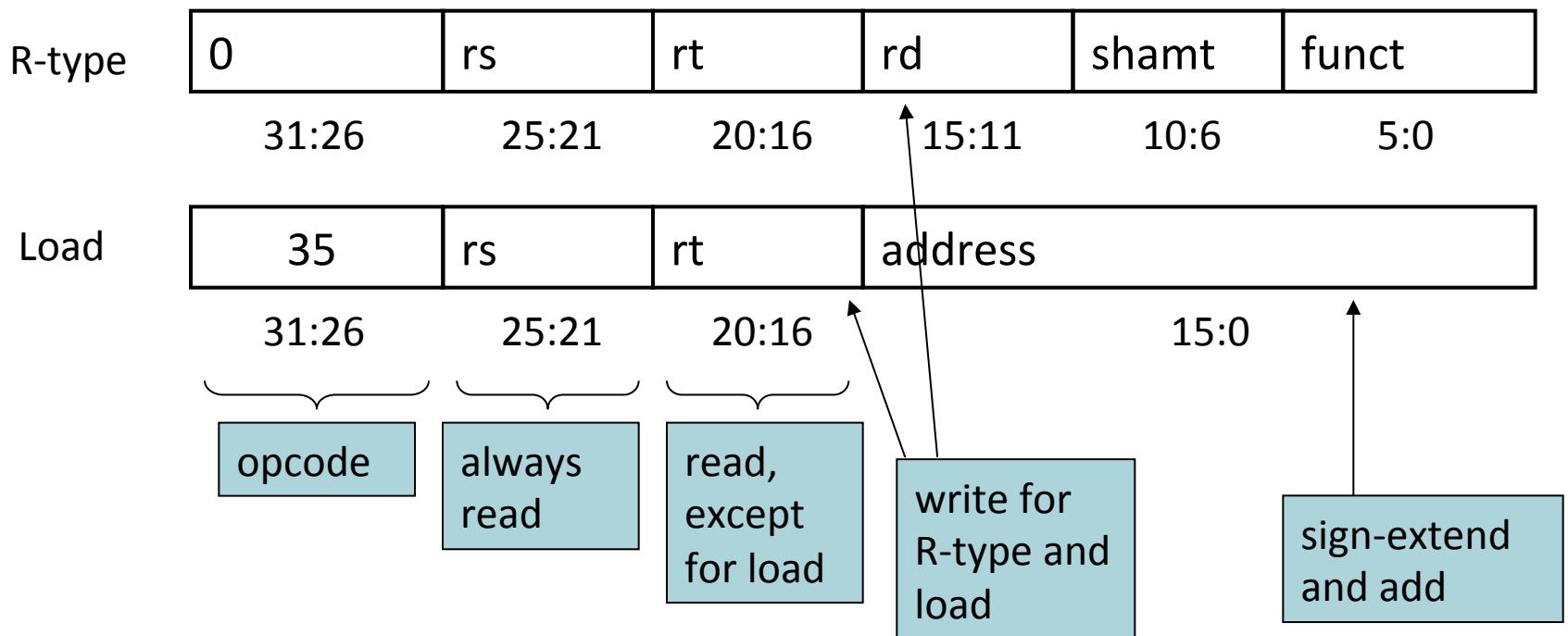
ALUOp		Funct field							Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	0110	
1	X	X	X	0	0	0	0	0010	
1	X	X	X	0	0	1	0	0110	
1	X	X	X	0	1	0	0	0000	
1	X	X	X	0	1	0	1	0001	
1	X	X	X	1	0	1	0	0111	

- Truth table for the 4 ALU control bits (called Operation)
- Depends on ALUOp and instruction function code field (X's represent “don't cares”)



Combinational circuit to generate the 4-bit ALU control signal

- Control signals are derived from the instruction



Store

43	rs	rt	address
----	----	----	---------

31:26 25:21 20:16

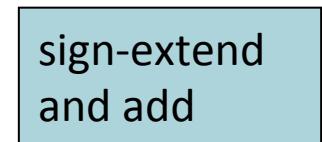
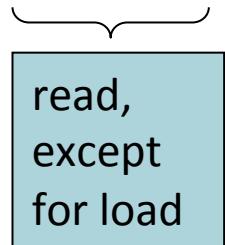
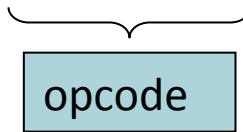
15:0

Branch

4	rs	rt	address
---	----	----	---------

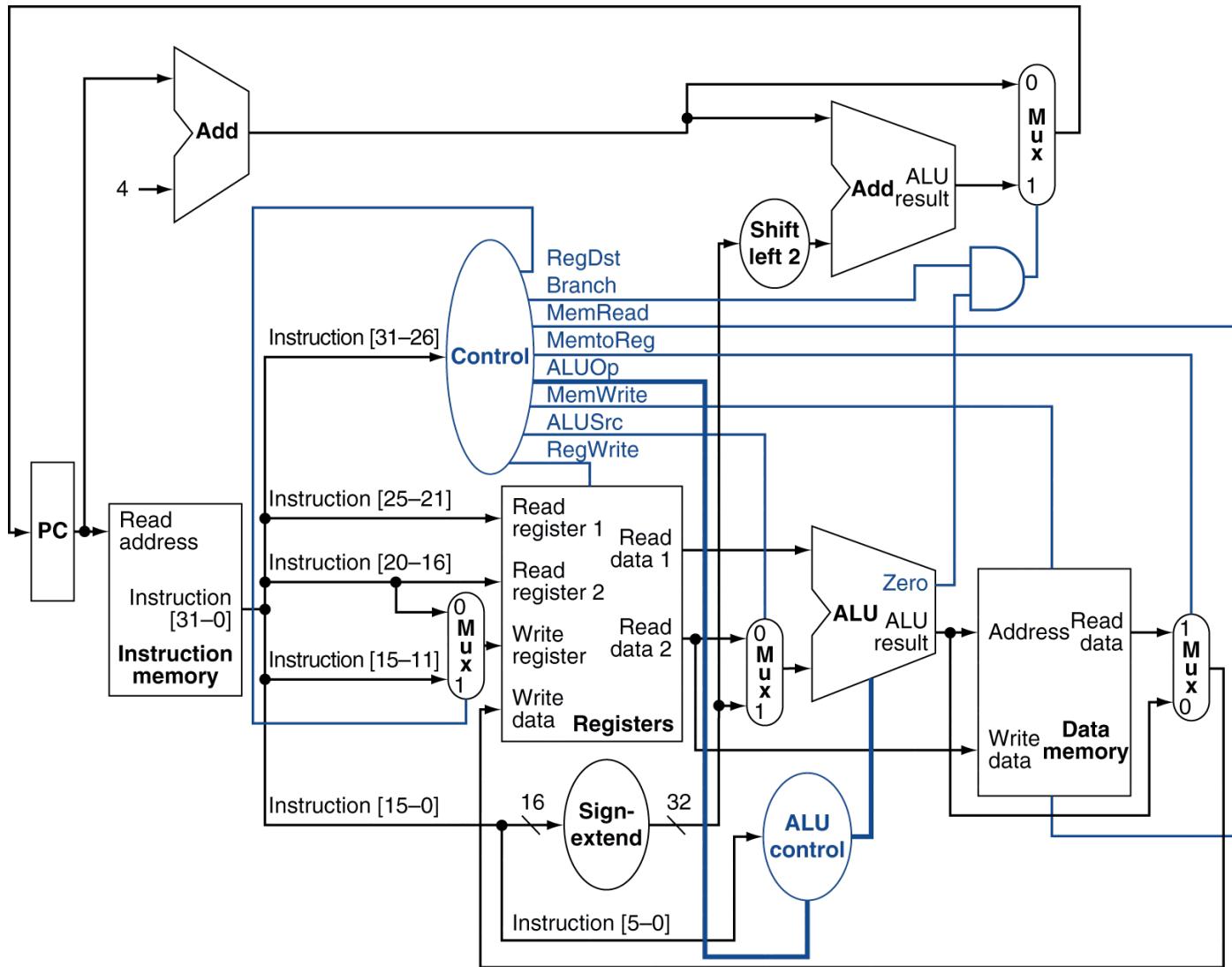
31:26 25:21 20:16

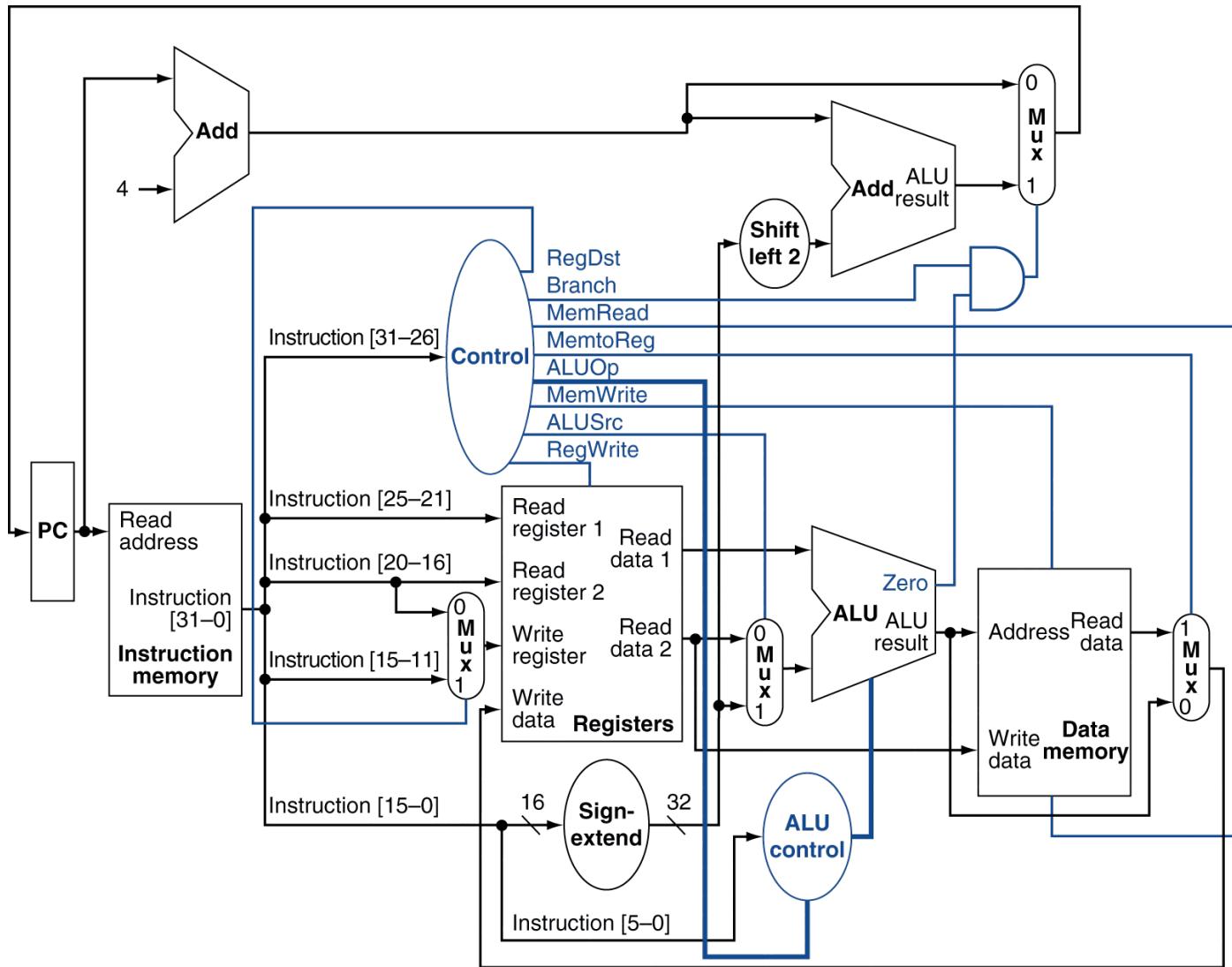
15:0

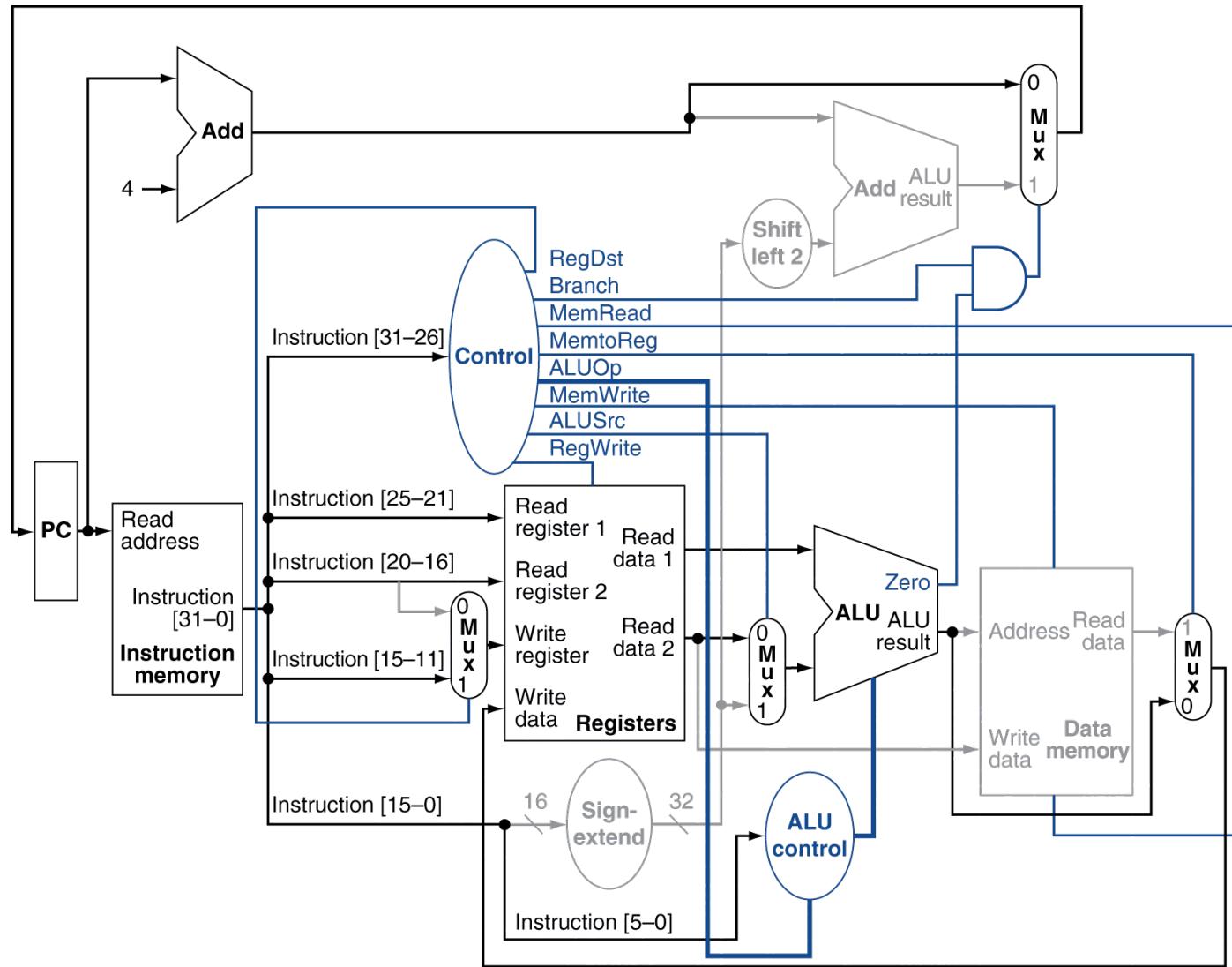


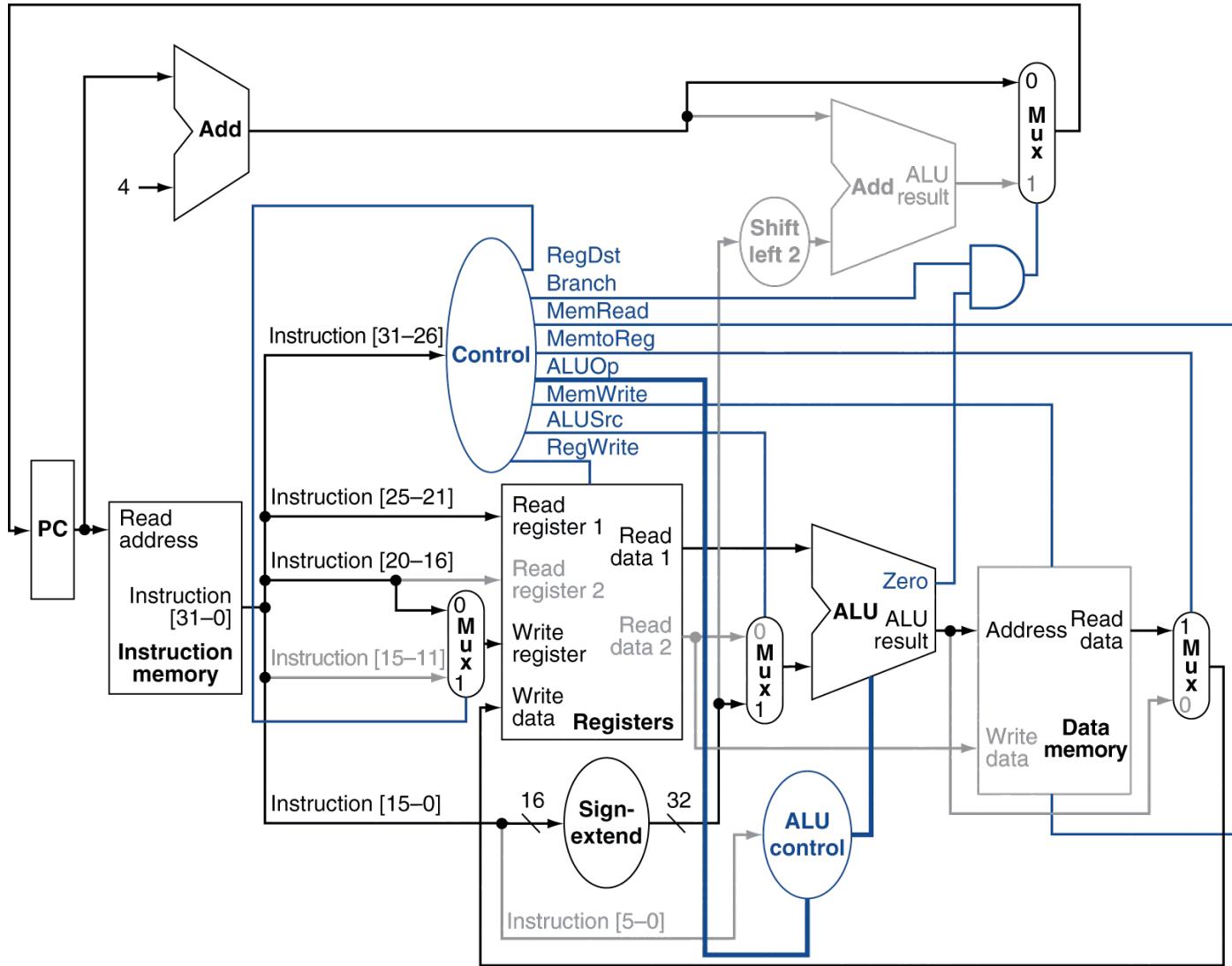
Store copies content of rt register into memory, rt not changed

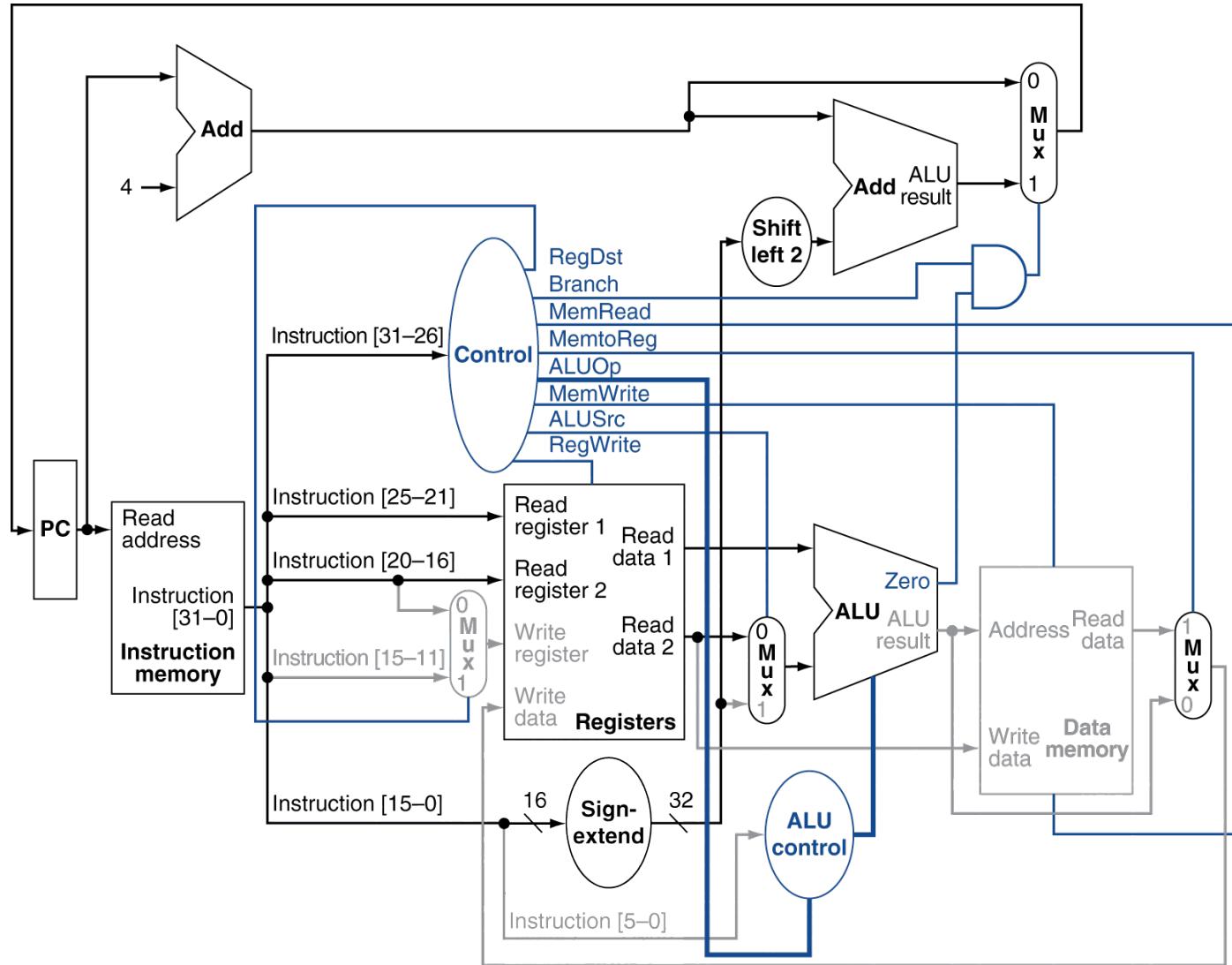
Branch subtracts rs from rt to generate zero flag, rs and rt not changed

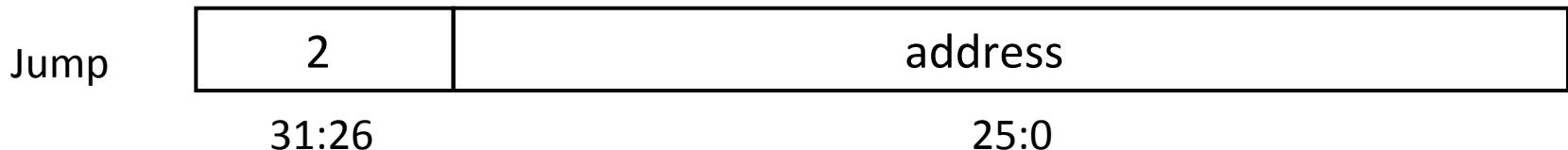




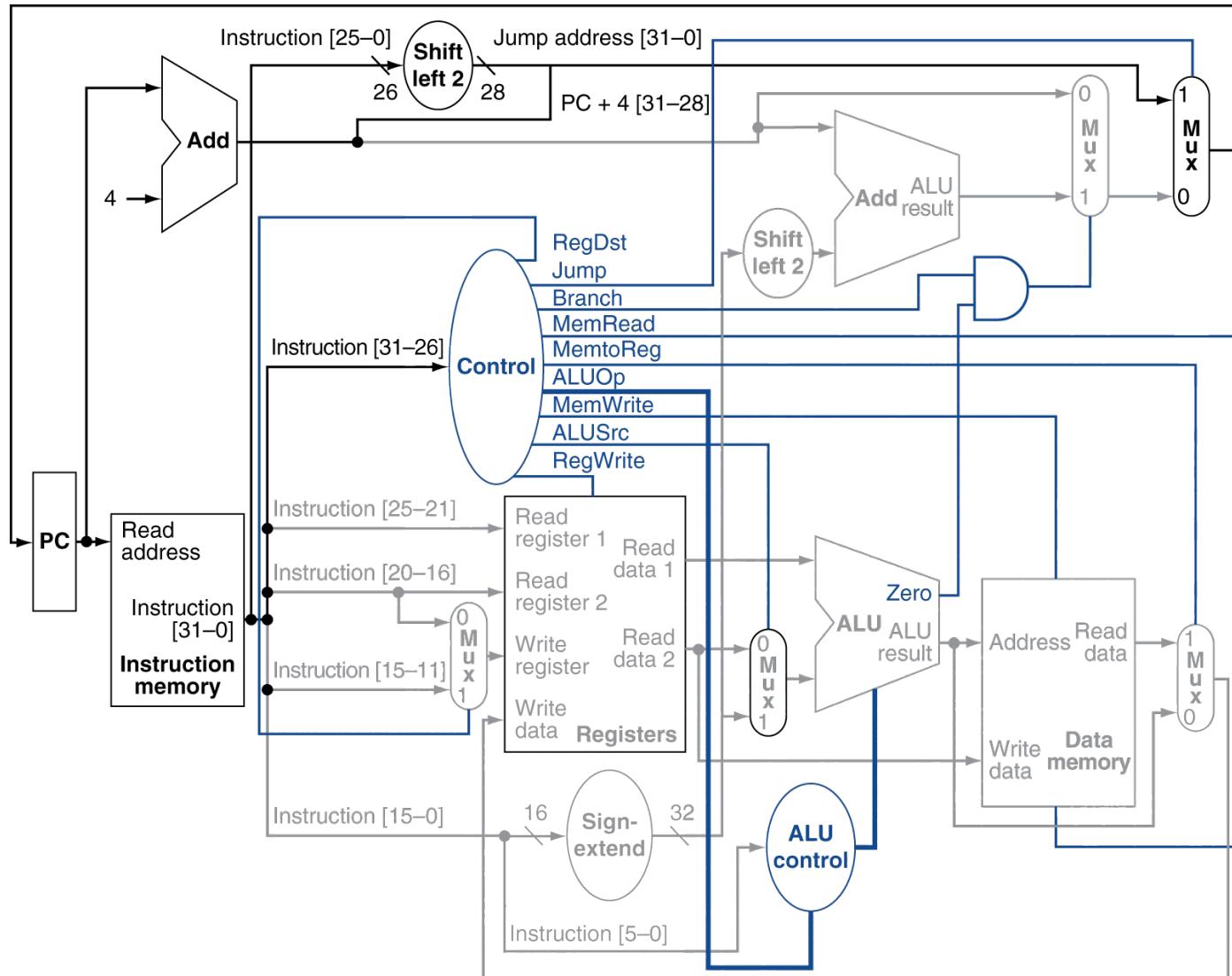








- Jump uses word address (instruction boundary)
- Sets $\text{PC} = (\text{PC} \& 0xF0000000) + 4 * (\text{26-bit jump address})$
- Needs an extra control signal decoded from opcode



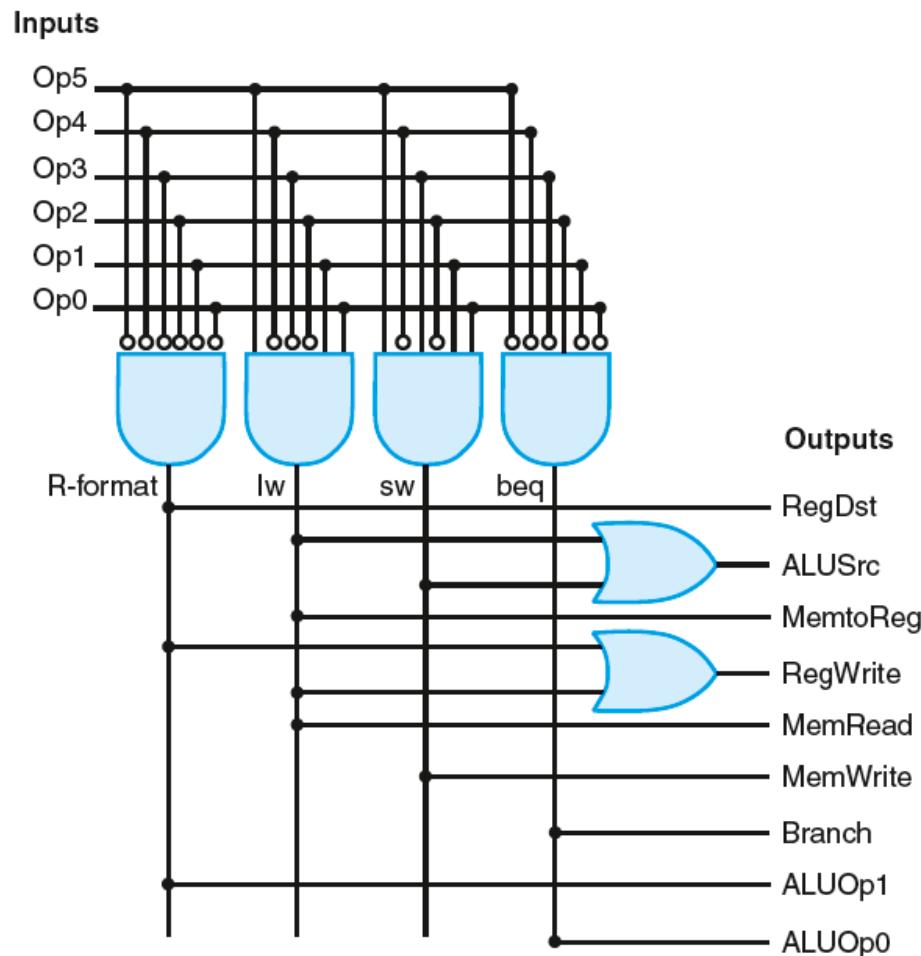
- Use of instruction subset simplifies control
- Only 9 control signals have to be generated

Signal name
RegDst
ALUSrc
MemtoReg
RegWrite
MemRead
MemWrite
Branch
ALUOp1
ALUOp0

- A truth table defines the outputs as function of opcode

Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

- Simple logic circuit derived from truth table



- With this option, instruction executes in a single clock cycle
- Longest instruction determines clock period
 - Critical path: load instruction
 - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
 - Making the common case fast
- We will improve performance by pipelining

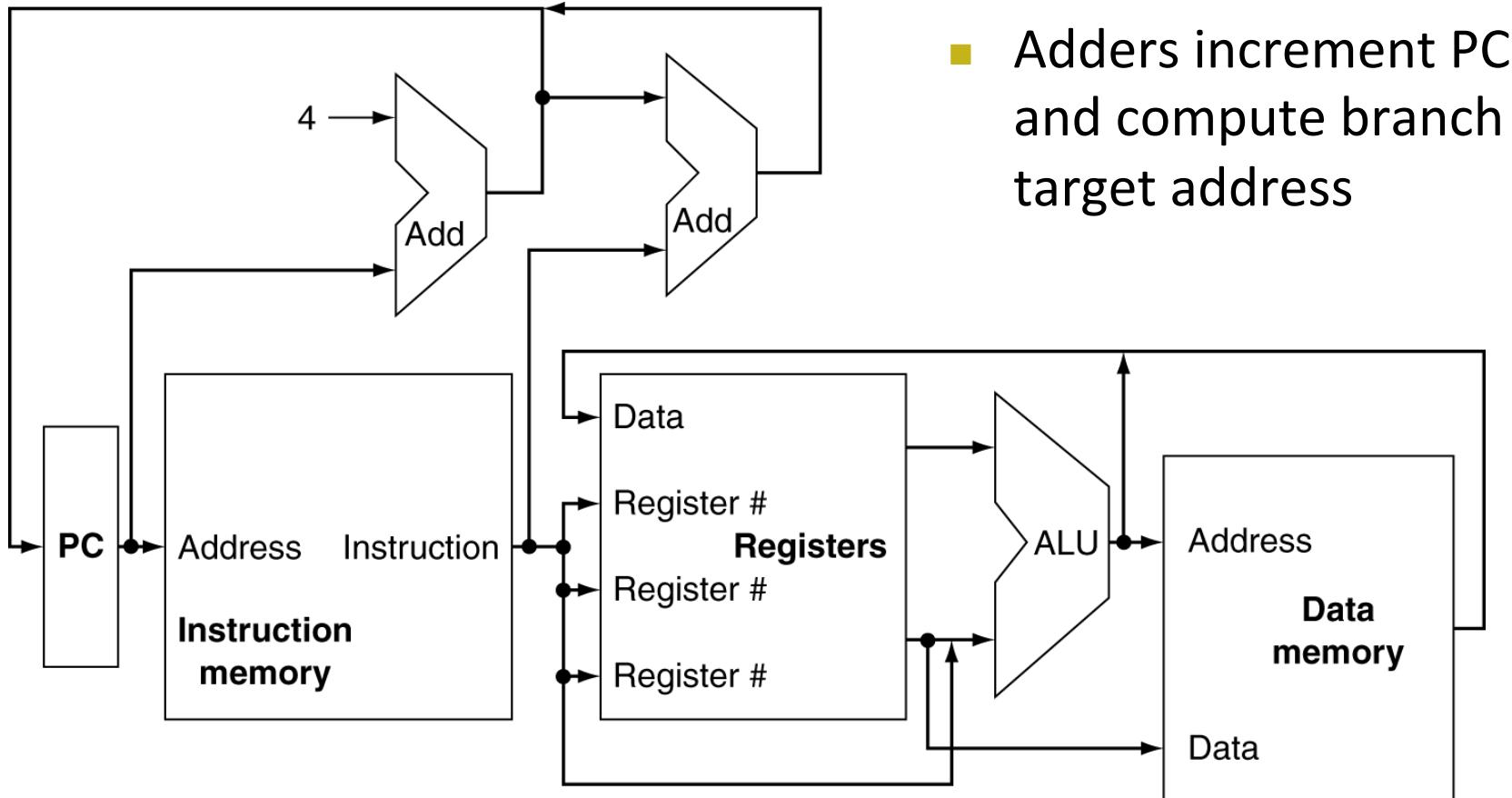


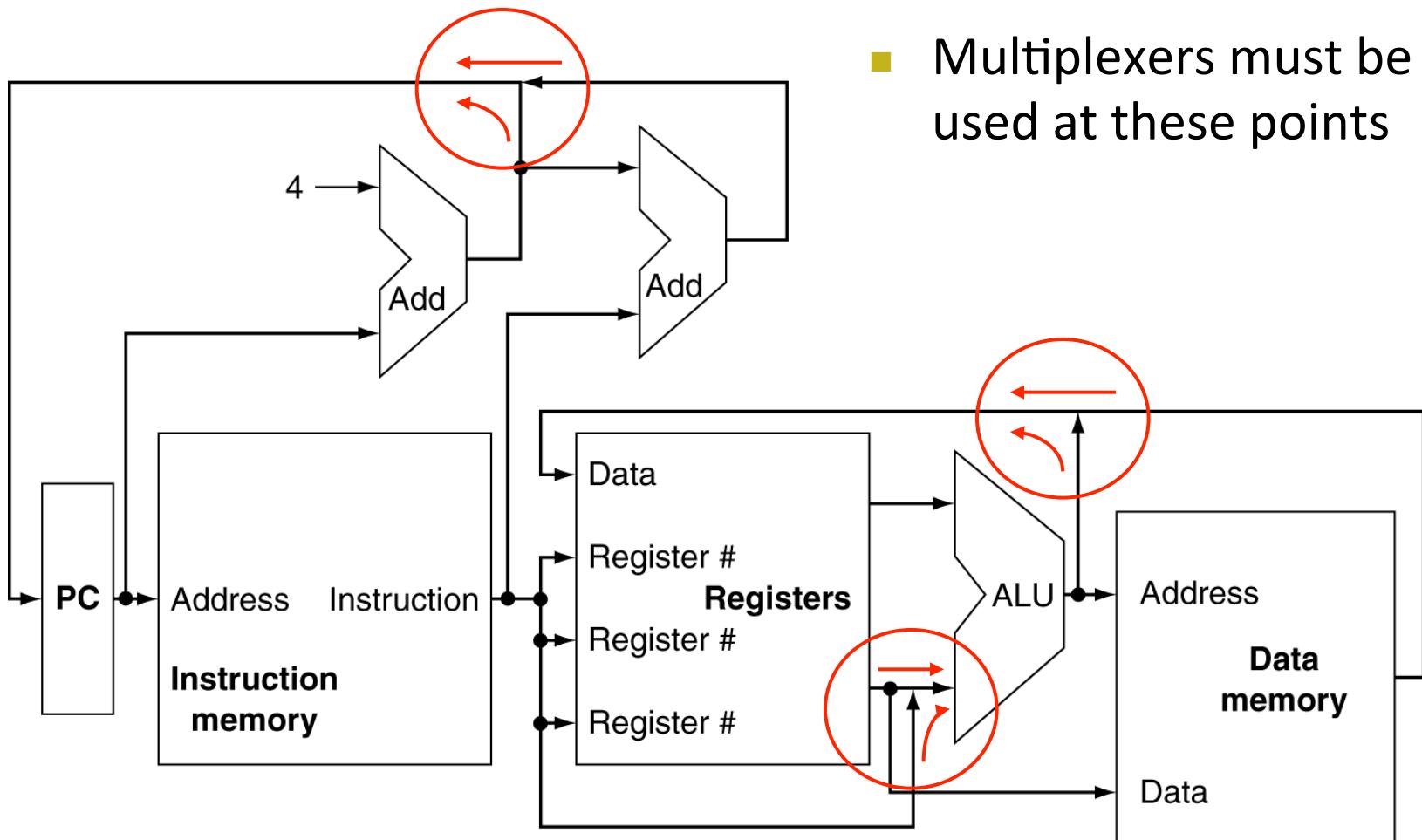
- This module describes how instructions are executed using the MIPS datapath
- The datapath includes the ALU, control unit, register file and the pathways that connect the various components
- The ALU and control unit are implemented using the logic circuits described in the previous module



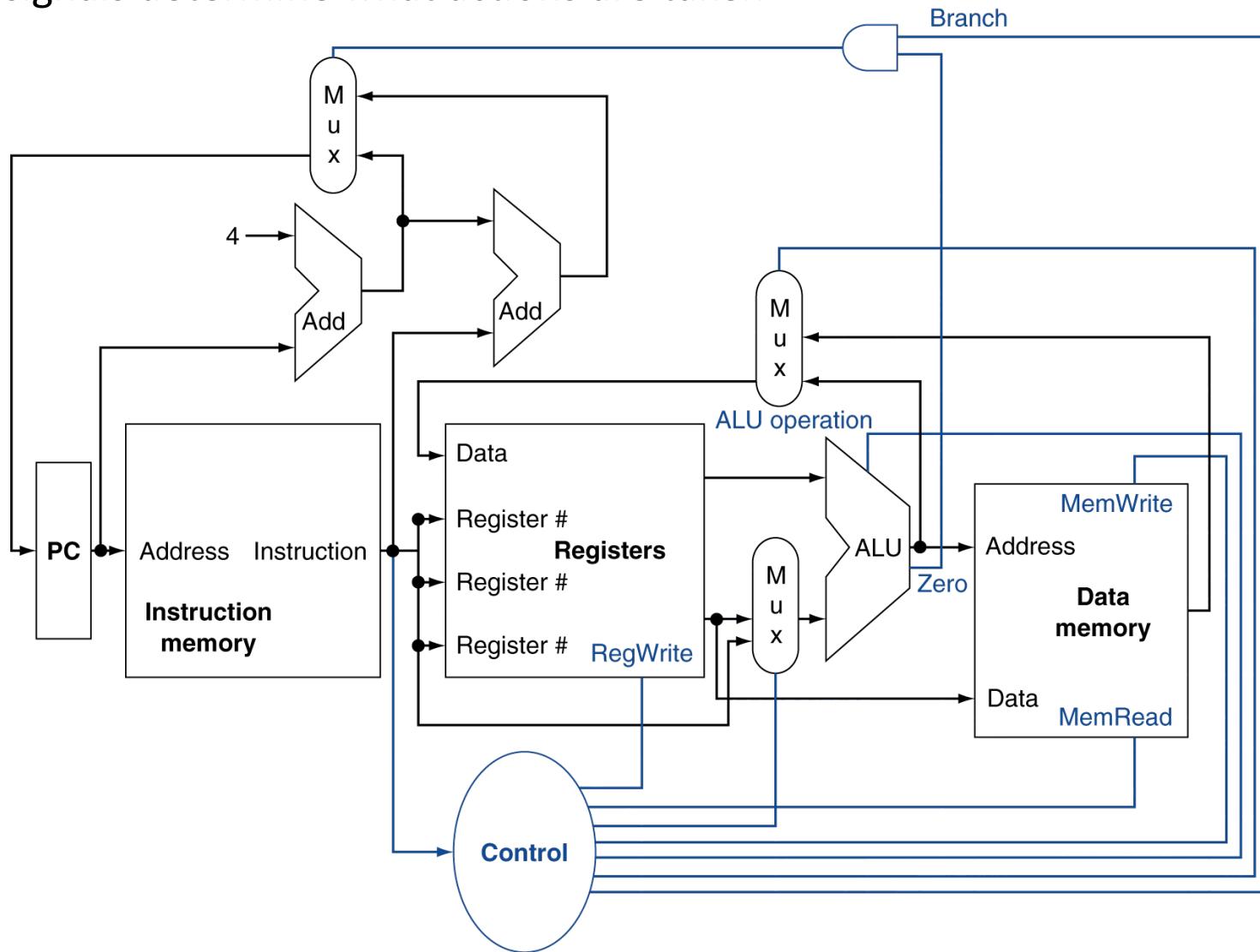
- CPU performance factors
 - Instruction count
 - Determined by ISA and compiler
 - CPI and Cycle time
 - Determined by CPU hardware
- We will examine two MIPS implementations
 - A simplified single-cycle version
 - And later, a more realistic pipelined version
- A simple instruction subset will be used
 - Memory reference: lw, sw
 - Arithmetic/logical: add, sub, and, or, slt
 - Control transfer: beq, j

- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
 - Use ALU to
 - Calculate result
 - Compute memory address for load/store
 - Evaluate branch condition
 - Access data memory for load/store
 - PC ← target address or PC + 4

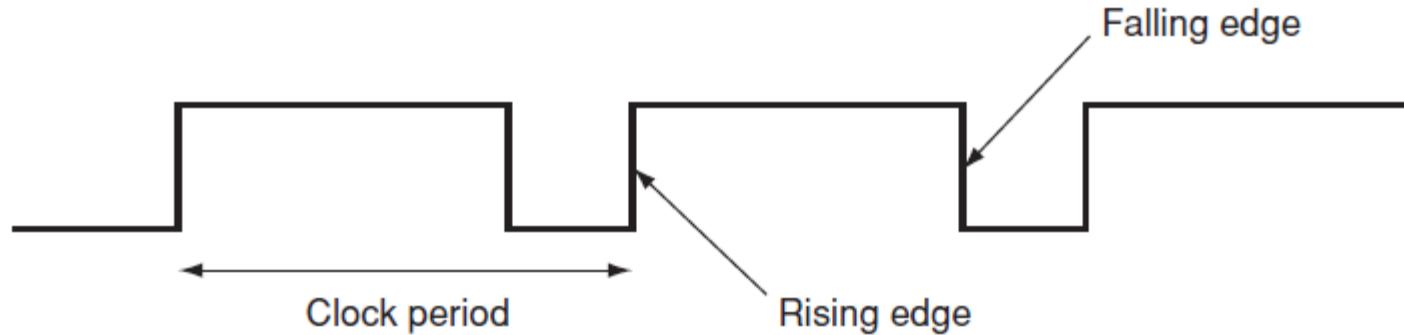




Control signals determine what actions are taken

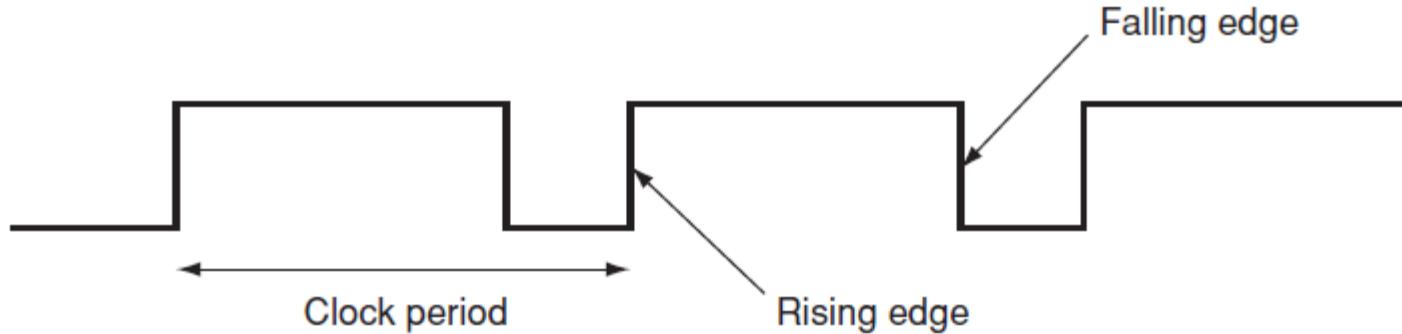


- Operations are synchronized to a clock
 - For example, when a register is written
 - Instructions complete at clock edges



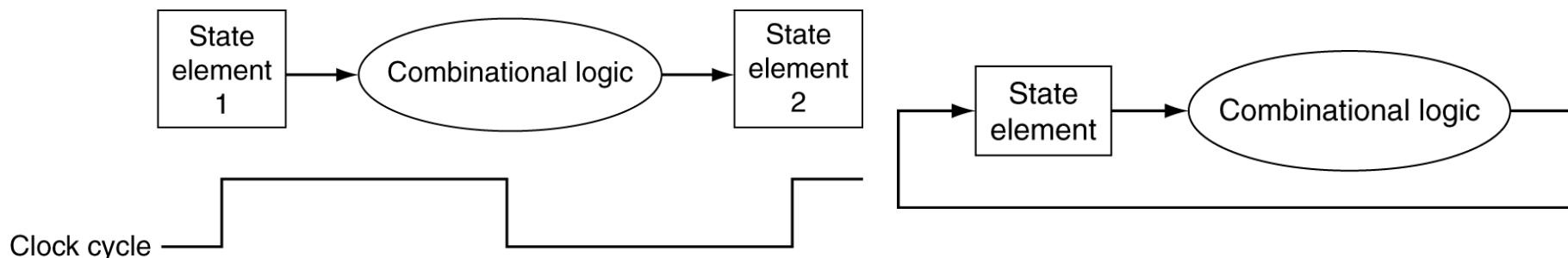
- Clock signal oscillates between high and low values
- Clock period is one full clock cycle
- State changes only on clock edge (either rising or falling edge)

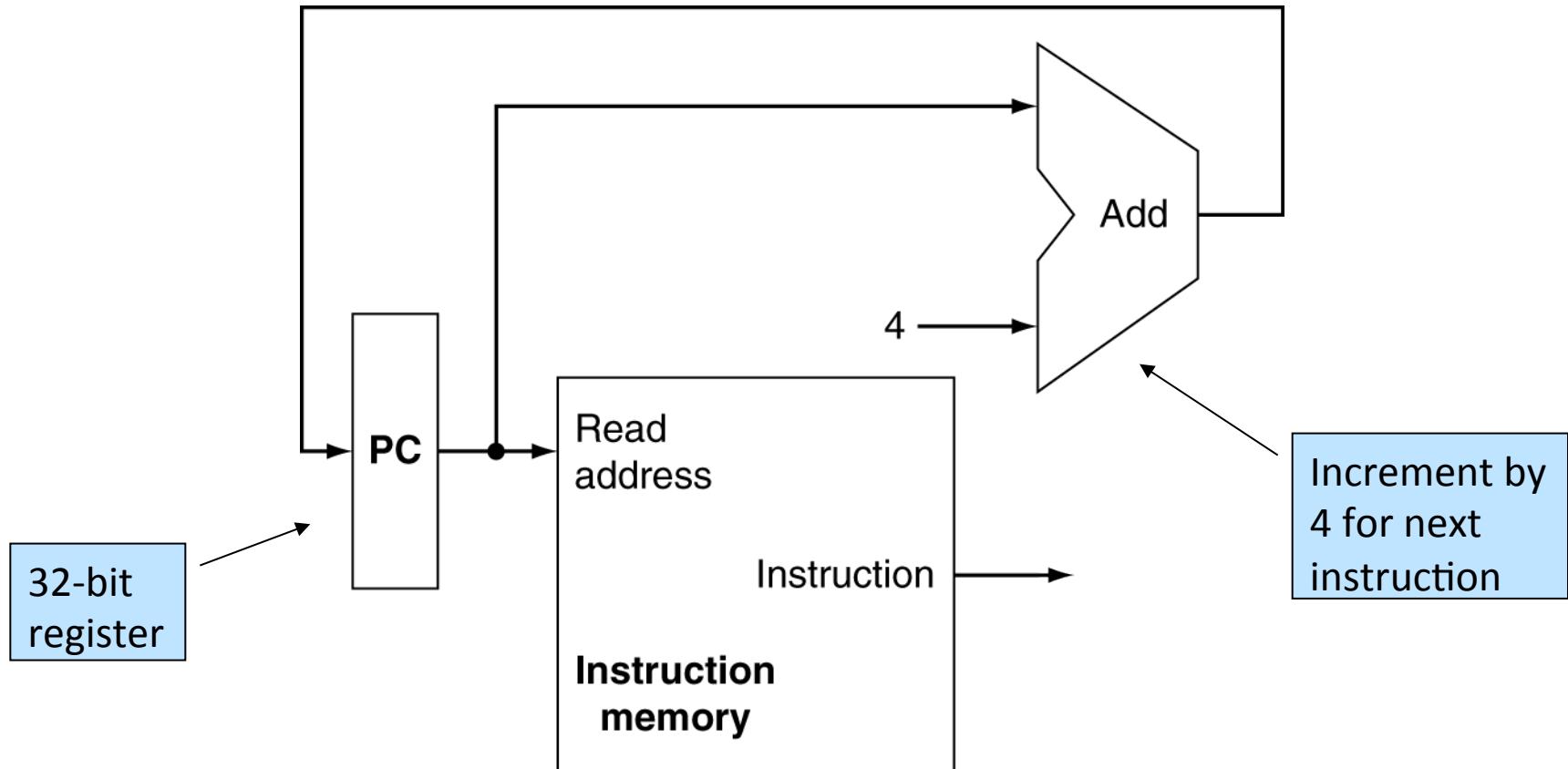
- Operations are synchronized to a clock
 - For example, when a register is written
 - Instructions complete at clock edges



- Clock signal oscillates between high and low values
- Clock period is one full clock cycle
- State changes only on clock edge (either rising or falling edge)

- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Inputs come from state elements
 - Outputs go to state elements
 - Longest delay determines minimum clock period

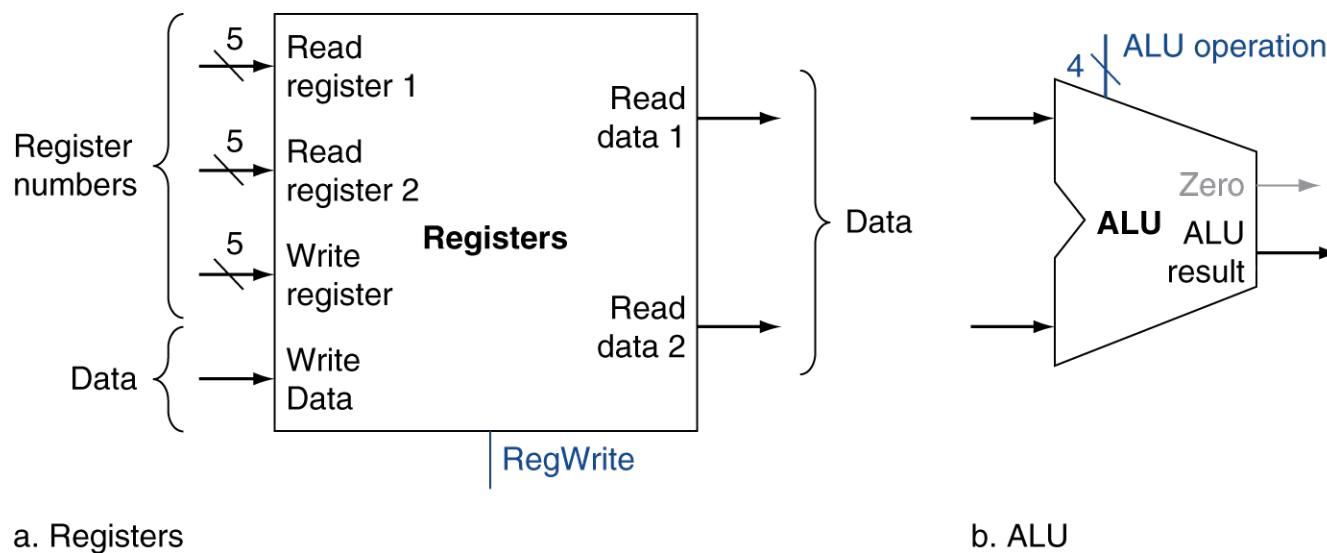




Performs Instruction Fetch

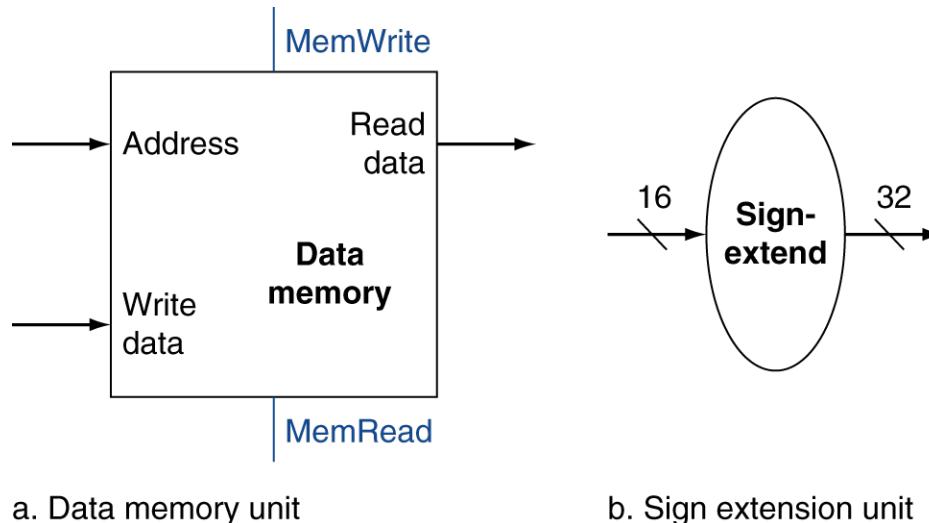
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



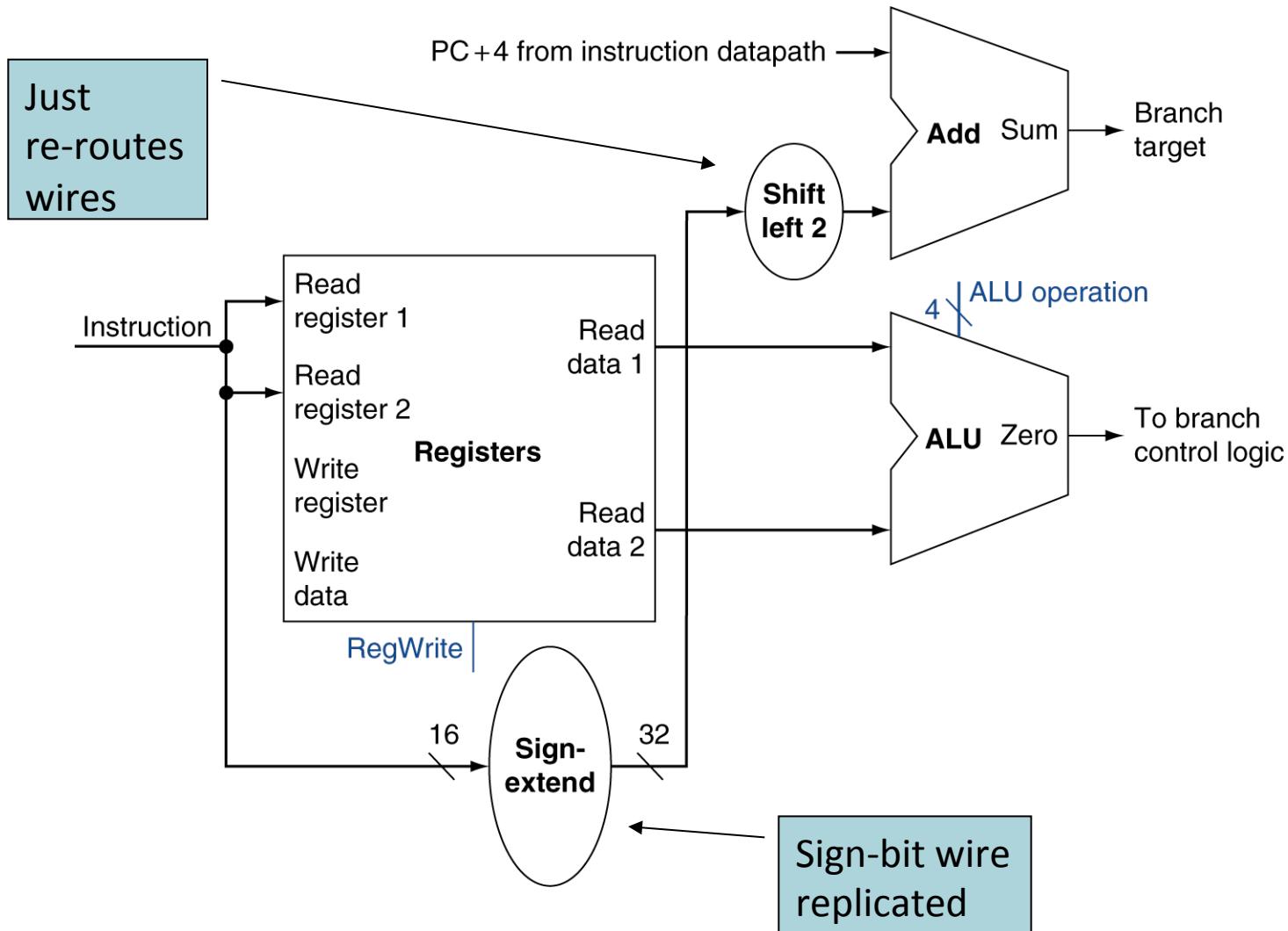
Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory

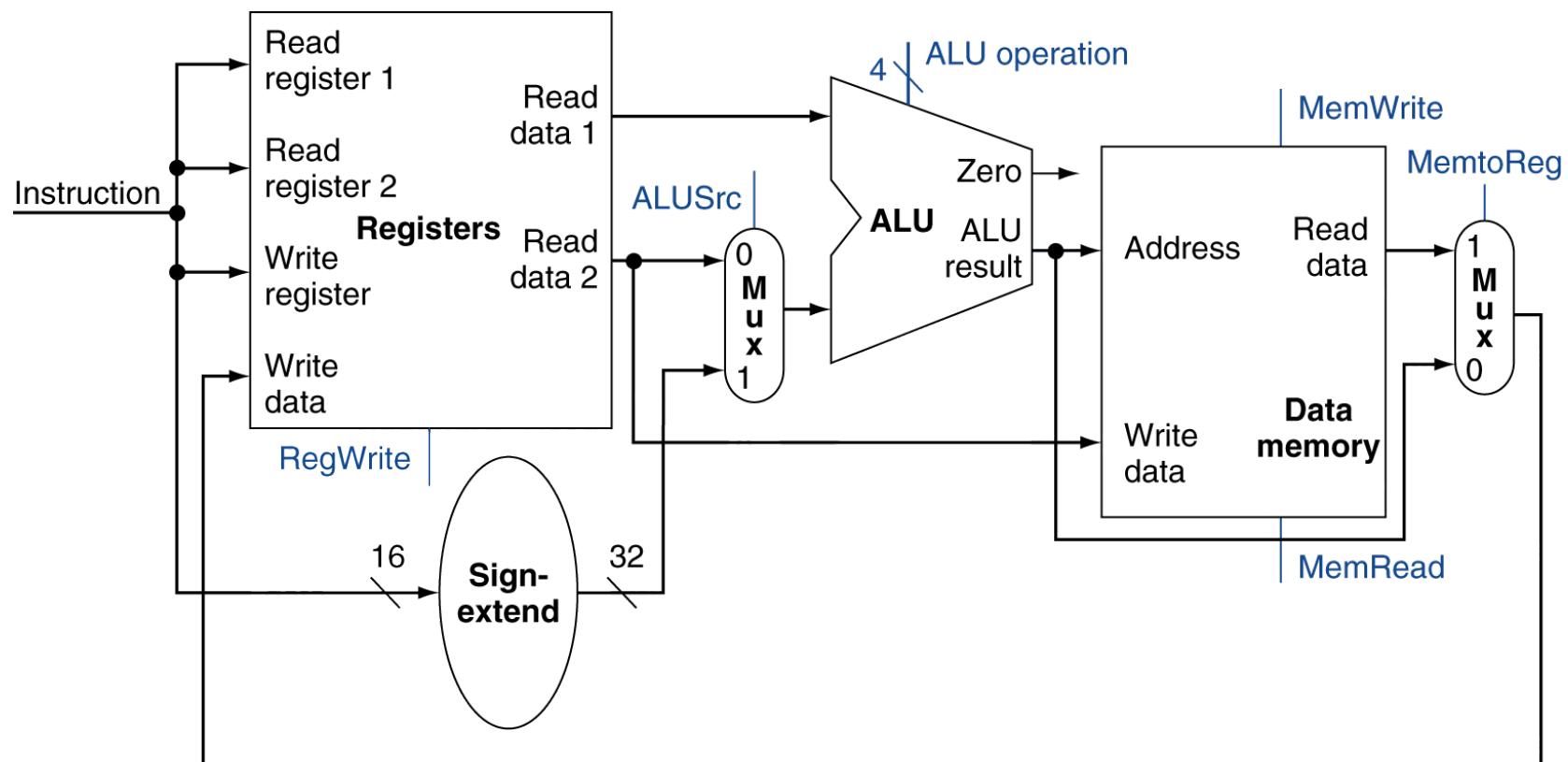


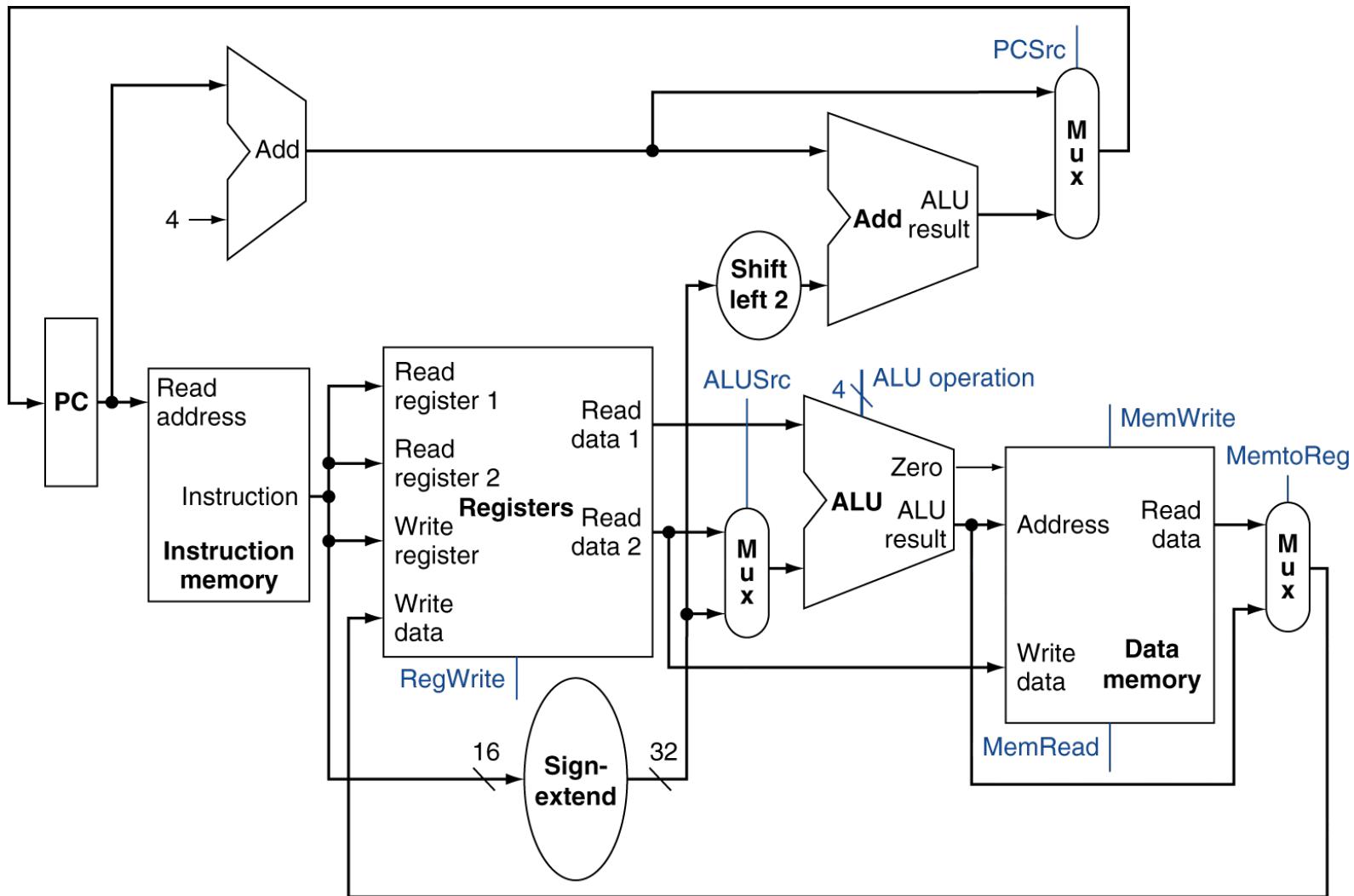
Branch Instructions

- Read register operands
- Compare operands
 - Use ALU, subtract and check Zero output
- Calculate target address
 - Sign-extend displacement
 - Shift left 2 places (word displacement)
 - Add it to PC
 - PC already incremented by instruction fetch



- One option is for the datapath to execute an instruction in one clock cycle
 - Each datapath element can only do one function at a time
 - Hence, we need separate instruction and data memories (Harvard Architecture)
- Use multiplexers where alternate data sources are used for different instructions



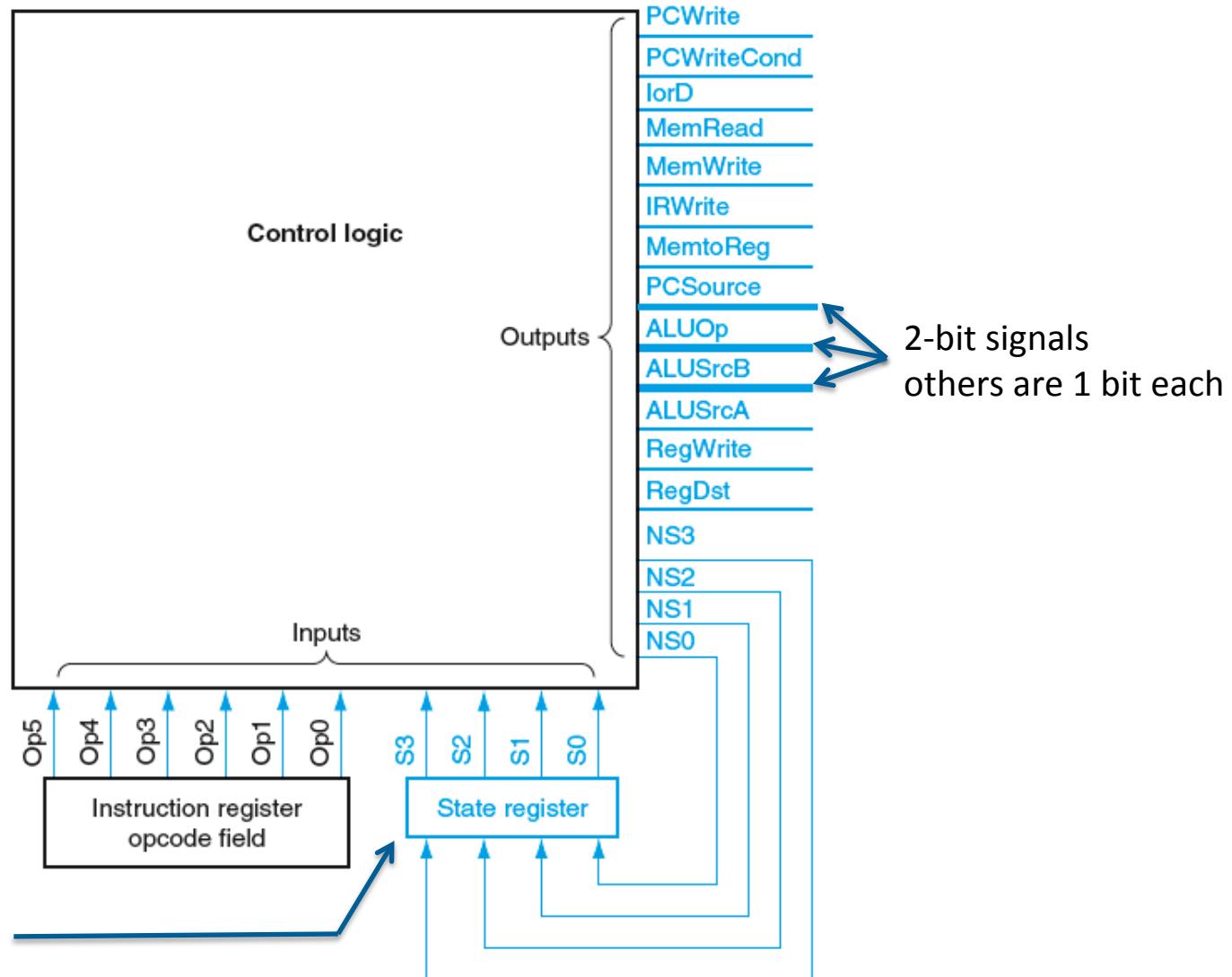




Conclusion

- The datapath components have now been described
- Next, the ALU will be examined in more detail
- We will also see how the control unit produces the required signals

MIPS Control Unit

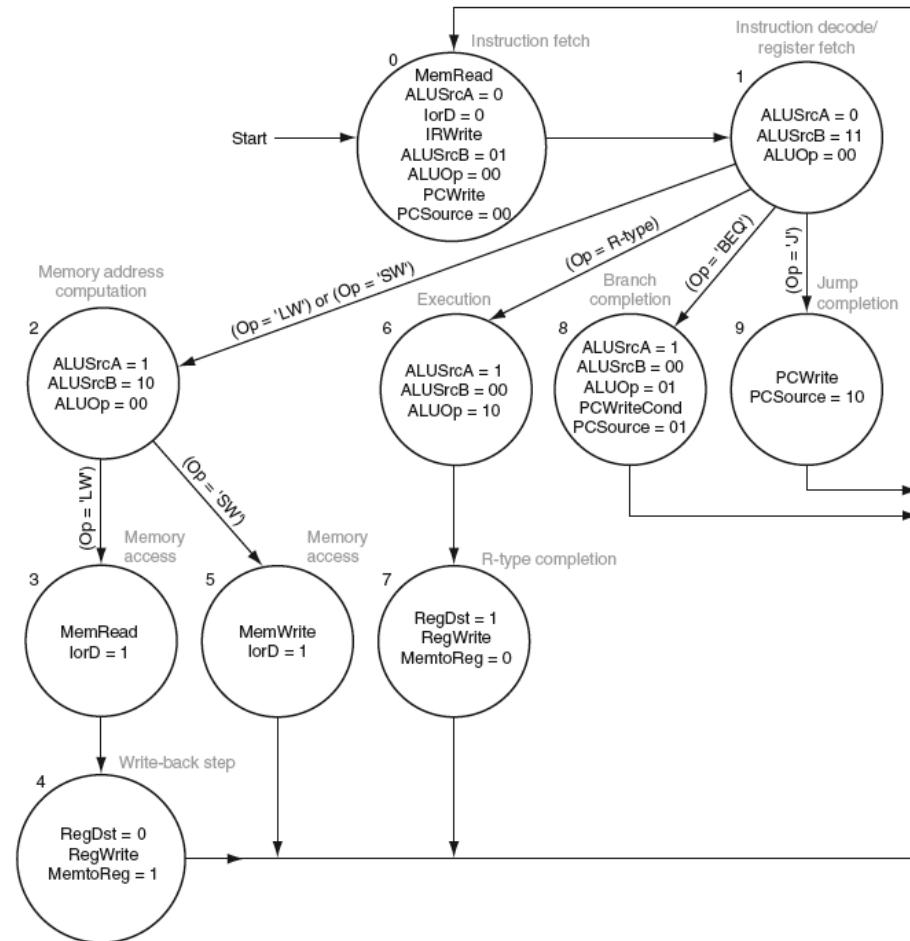


State alone is sufficient to determine output control signals

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 + state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	

State alone determines output control signals

Output	Current states
PCWrite	state0 + state9
PCWriteCond	state8
IorD	state3 + state5
MemRead	state0 + state3
MemWrite	state5
IRWrite	state0
MemtoReg	state4
PCSource1	state9
PCSource0	state8
ALUOp1	state6
ALUOp0	state8
ALUSrcB1	state1 + state2
ALUSrcB0	state0 + state1
ALUSrcA	state2 + state6 + state8
RegWrite	state4 + state7
RegDst	state7



Opcode as well as state are needed to determine next state

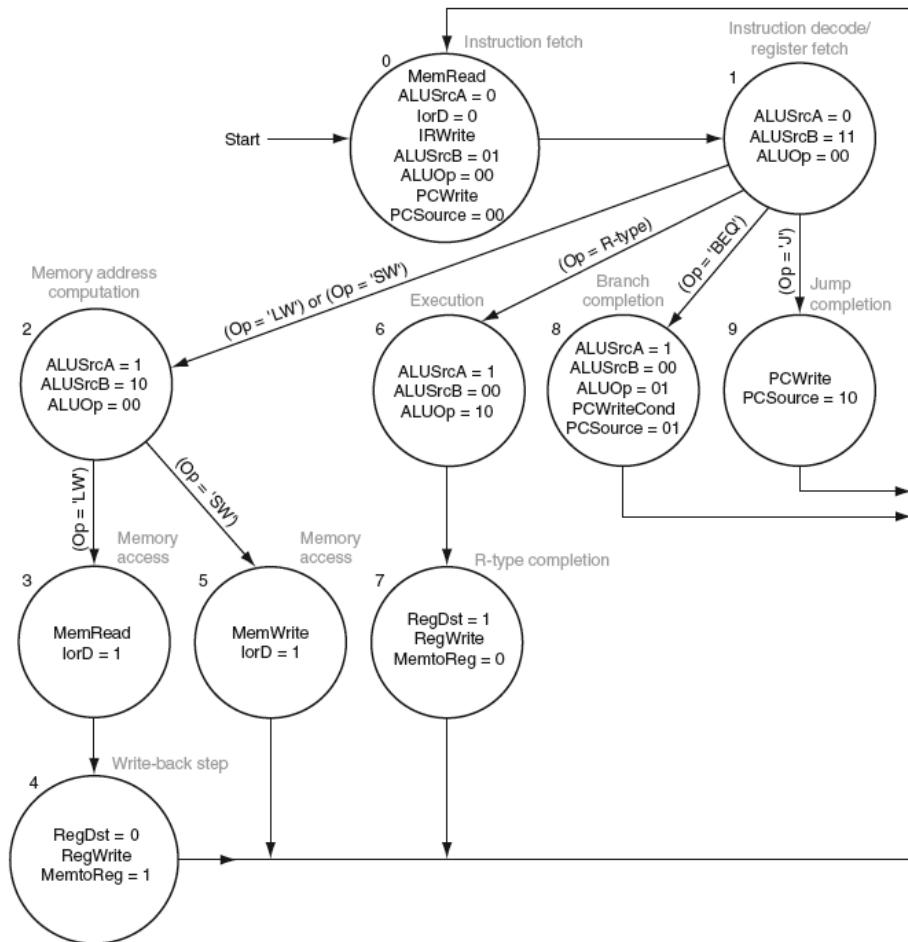
Output	Current states	Op
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

Four bits are needed for state number (since there are 10 states)

Output	Current states	Op
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

Opcode as well as state are needed to determine next state

Four bits are needed for state number (since there are 10 states)



Outputs	Input values (S[3-0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

Truth table gives the 16 control signals as function of 4-bit state

The 16 control signals could be read from a lookup table in ROM

Lower 4 bits of the address	Bits 19–4 of the word
0000	1001010000001000
0001	00000000000011000
0010	00000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	00000000000000011
1000	0100000010100100
1001	1000000100000000

Use state number as index or address in ROM LUT (lookup table)

Use state number (as row) and opcode (as column) to get next state number from second ROM LUT

Current state S[3-0]	Op [5-0]					
	000000 (R-format)	000010 (jmp)	000100 (beq)	100011 (lw)	101011 (sw)	Any other value
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	Illegal
0010	XXXX	XXXX	XXXX	0011	0101	Illegal
0011	0100	0100	0100	0100	0100	Illegal
0100	0000	0000	0000	0000	0000	Illegal
0101	0000	0000	0000	0000	0000	Illegal
0110	0111	0111	0111	0111	0111	Illegal
0111	0000	0000	0000	0000	0000	Illegal
1000	0000	0000	0000	0000	0000	Illegal
1001	0000	0000	0000	0000	0000	Illegal

Undefined state/opcode combinations are “illegal”

- State number alone is sufficient to determine the 16 control bits
- The 4-bit state number and 6-bit opcode form a 10-bit address
- 10-bit address determines control bits as well as next state
- Instruction execution corresponds to a sequence of states
- Control bits output in each state direct datapath actions
- Each state is one step in the execution and takes 1 clock cycle
- Other control unit implementations will be described next.

Lookup Tables can consume large amounts of storage

Using a full 10-bit address (opcode + state) means 1024 entries

Each entry contains 16 control bits and 4 next state bits

Total size = $1024 * 20$ bits (expensive in earlier days of computing)

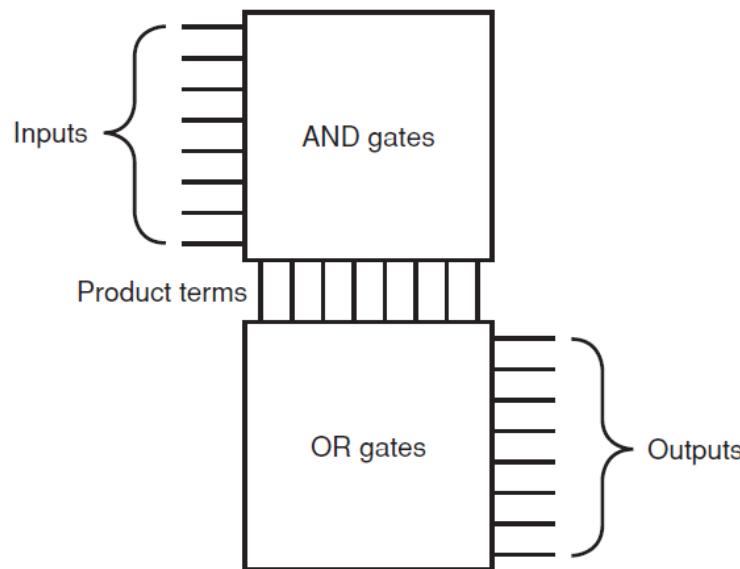
One way to reduce the required storage is to use a PLA

“PLA” means programmable logic array

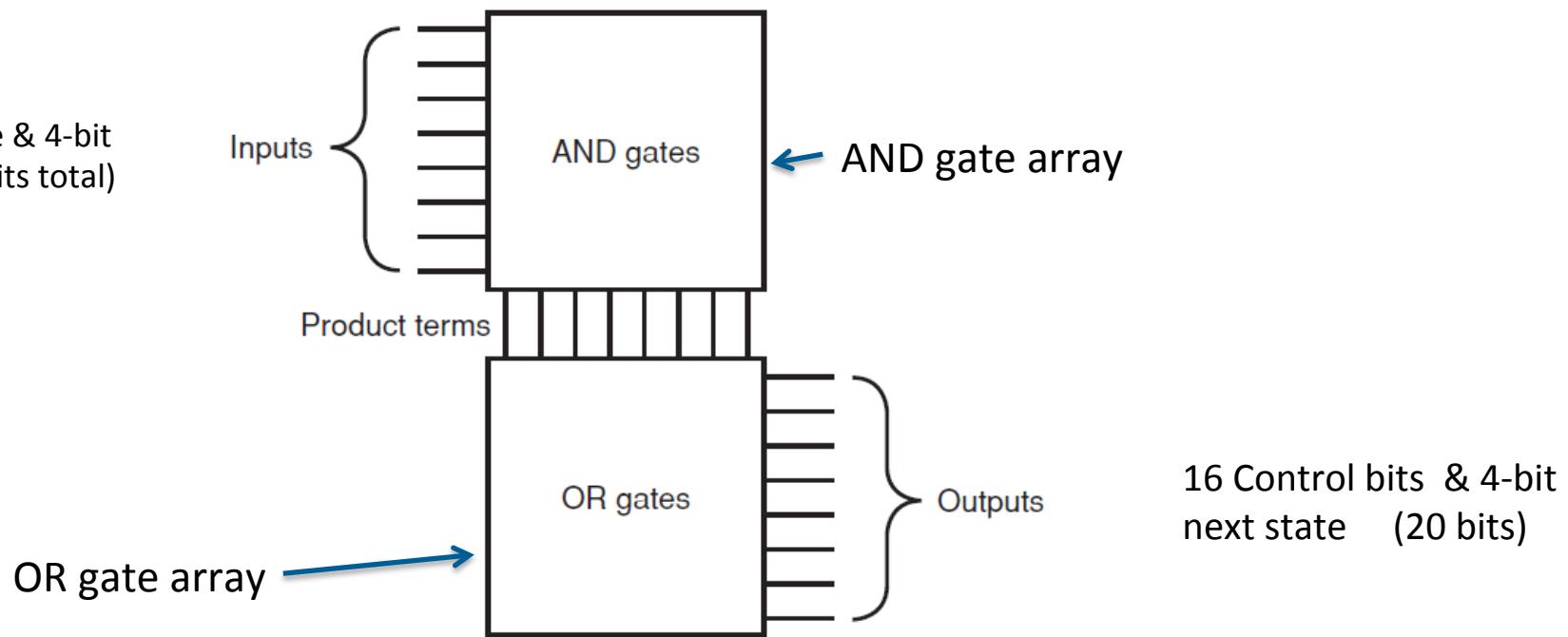
Each PLA output is a logical sum of one or more minterms

Minterm (or product term) is a logical AND of two or more inputs

A minterm corresponds to a single row in a truth table



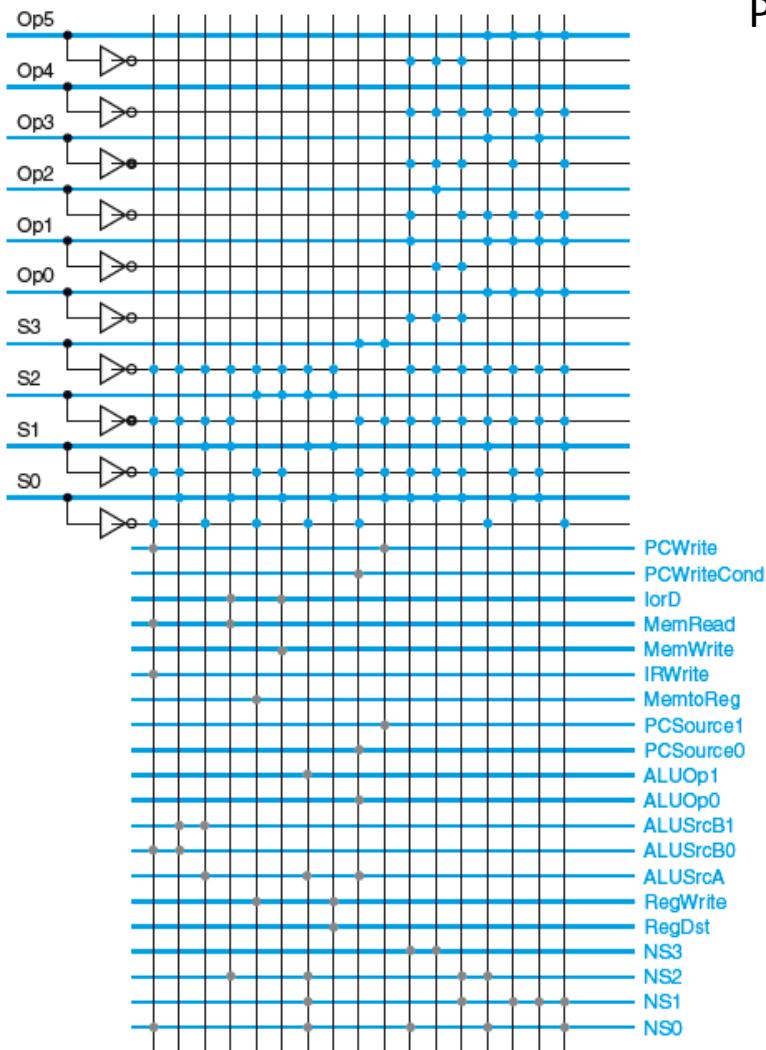
6-bit Opcode & 4-bit state# (10 bits total)



- AND array generates products of inputs or inverted inputs
- OR array generates logical sums of product terms

One vertical line for each of the 17 minterms

Size proportional to
 $(\# \text{inputs} + \# \text{outputs}) * \# \text{product_terms}$

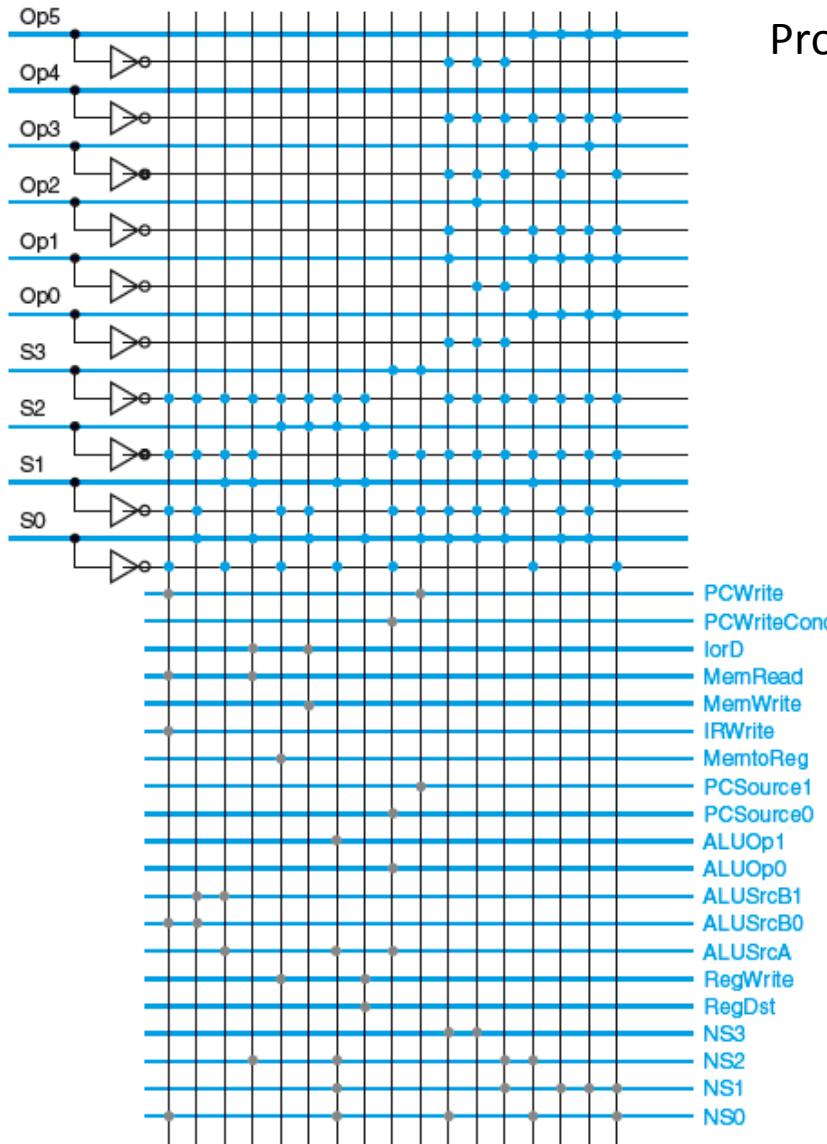


Programmable Logic Array

Horizontal output
lines correspond to
control lines and
next state bits

The 10 Inputs are
opcode and current
state

Size proportional to
 $(10 + 20) * 17 = 510$



Programmable Logic Array

The 20 outputs are
control signals and
next state

Opcode bits and current state# and fed in for each cycle

The PLA outputs the resulting control signals and next state#

Control signals are output for each instruction step or subcycle

The sequence of state numbers identify the steps or transitions

This PLA supports our MIPS core instruction subset

Each vertical line in the previous PLA is a minterm

The leftmost 10 depend only on the state

The remaining 7 depend on the state and opcode

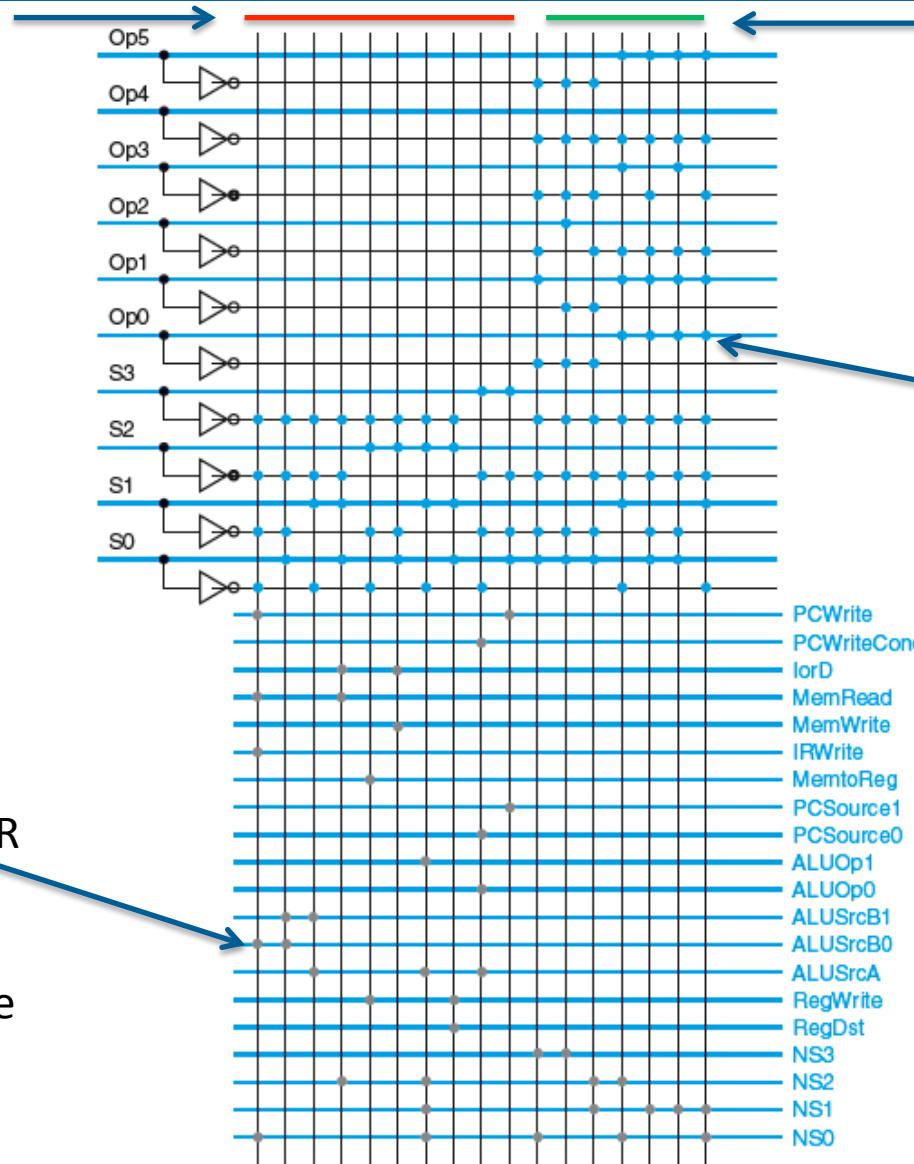
The top half of the figure is the AND plane that computes minterms

Dots in top half indicate which inputs are fed into AND gate

The bottom half is the OR plane that sums the minterms

Dots in lower half indicate which minterms are fed into OR gate

These 10 minterms
depend only on
state#



Black dots identify OR
gate inputs

OR gates produce the
sum of products

These 7 minterms
depend on opcode
as well as state#

Blue dots identify
AND gate inputs

Previous PLA can be replaced by two smaller PLAs (PLA1 & PLA2)

PLA1

produces 10 minterms from 4 inputs (state)
outputs the 16 control signal bits
 $\text{Size} = (4 + 16) * 10 = 200$

PLA2

produces 7 minterms from 10 inputs (opcode & state)
outputs 4-bit next state number
 $\text{Size} = (10 + 4) * 7 = 98$

Together the two consume less space than the single larger PLA

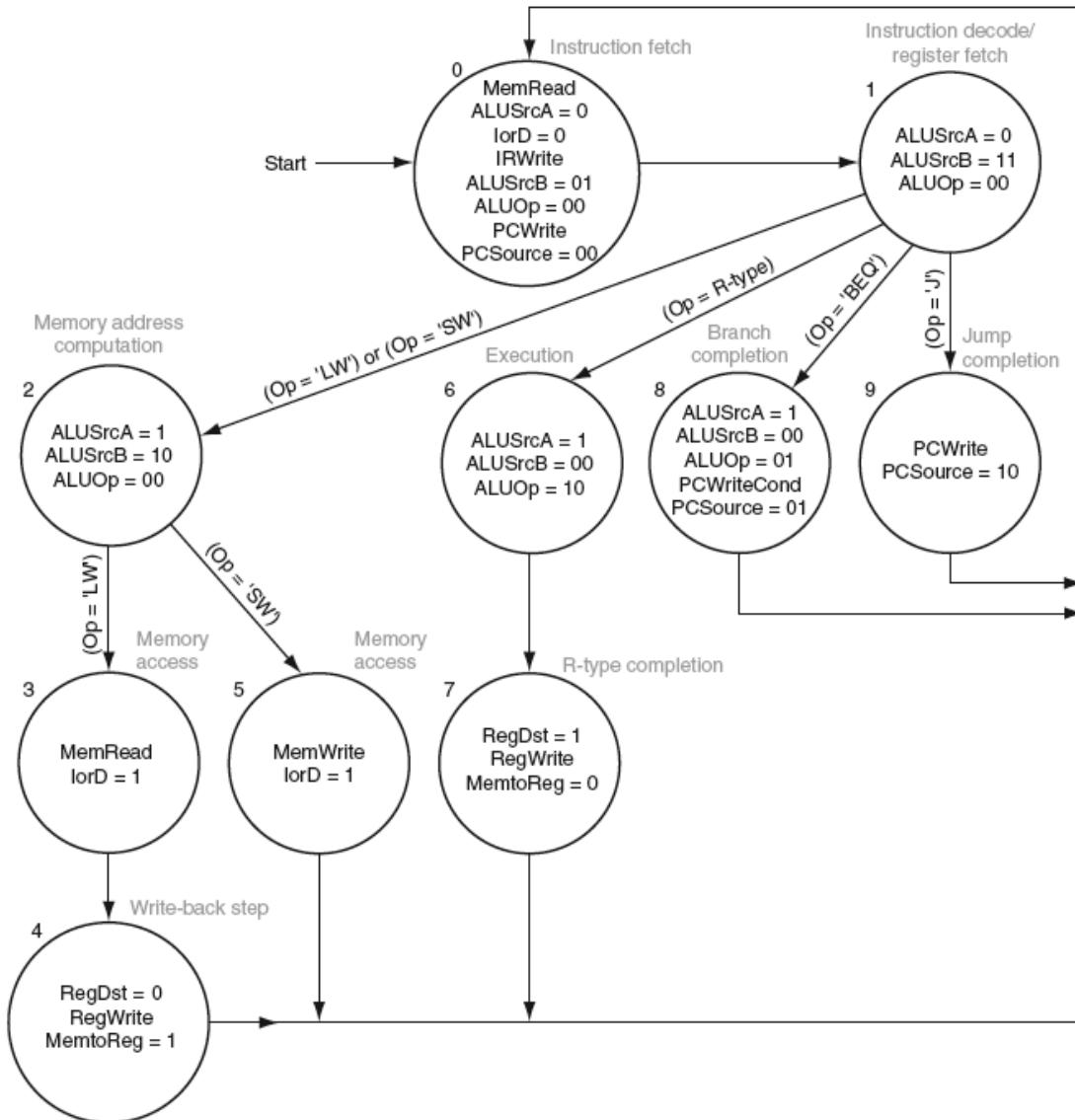
- Most of the control logic is needed just for the next state number
- Our core MIPS system only has 10 states
- More realistic systems would have many more states
- Often the next state is just the previous state + 1
- The start state always comes after the final state in each instruction
- The start state begins the next instruction

State1 always comes after state0

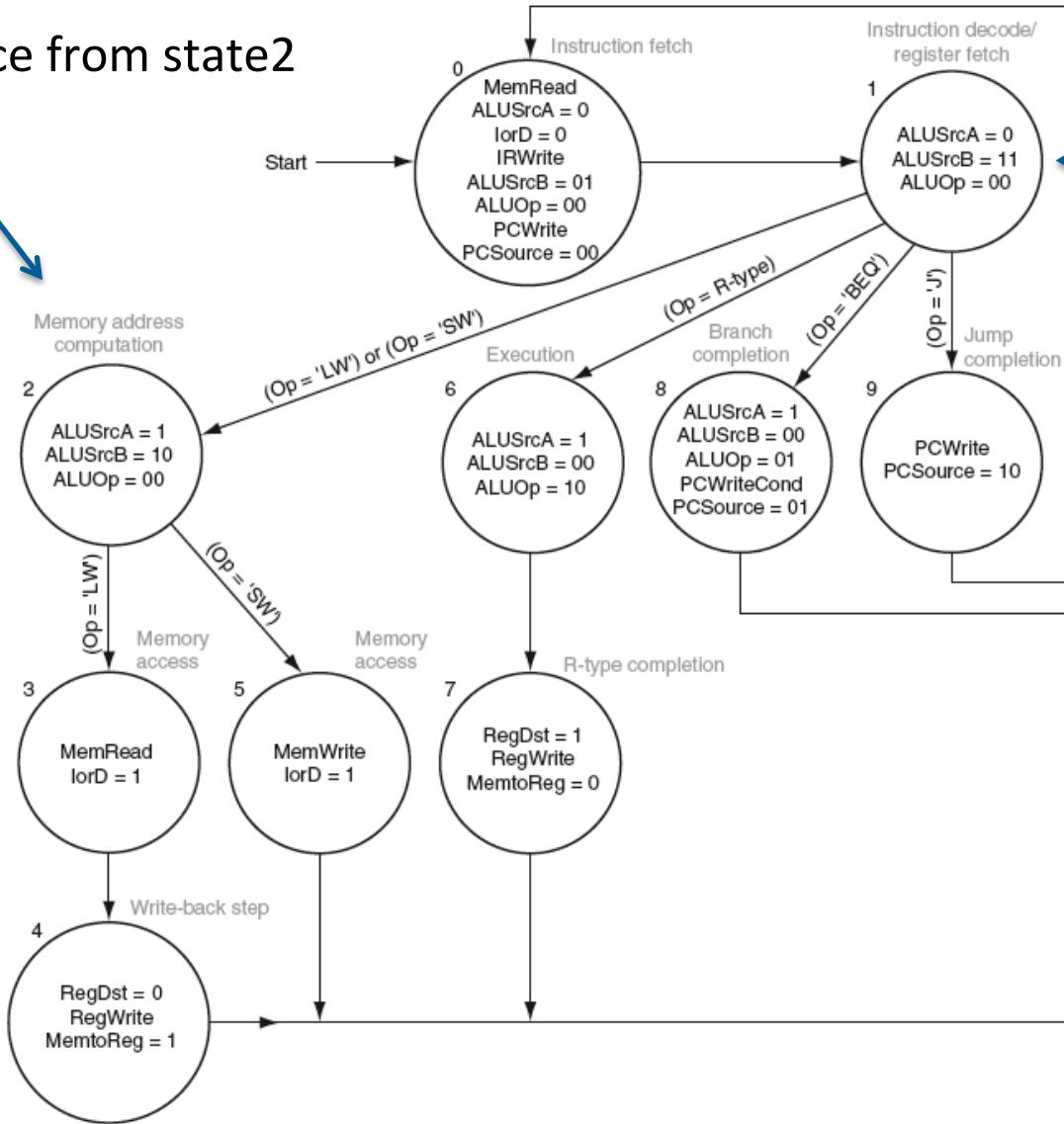
State4 always comes after state3

State7 always comes after state 6

State0 always comes after states 4, 5, 7, 8 and 9



2-way choice from state2

4-way choice
from state1

To go to the next sequential state, just increment the current state

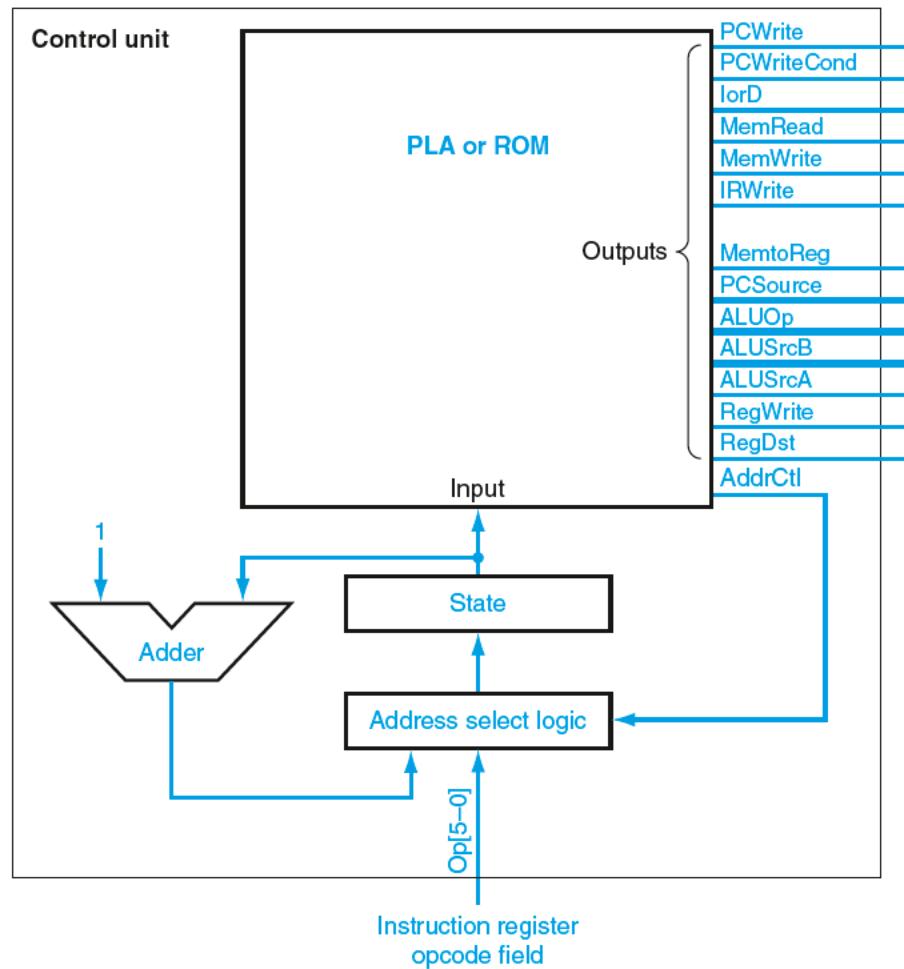
Set the state to 0 to go back to the initial state

Small lookup tables can be used to choose a non-sequential state

The choice from state 1 and from state2 is based on the opcode

Two ROMs (addressed by opcode) can be used

A 2-bit control signal can be used with a MUX for 4 possible choices



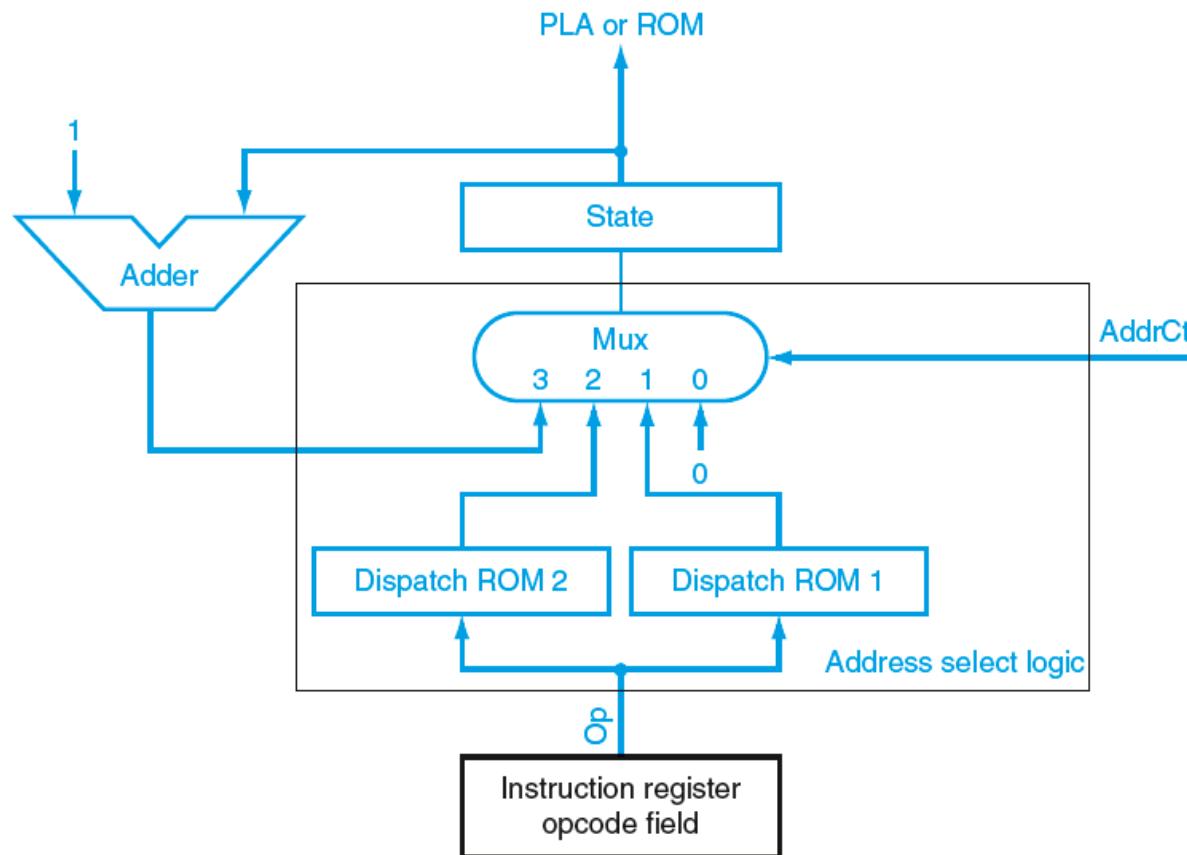
2-bit AddrCtl causes Address select logic to output the next state number

AddrCtl is defined as follows:

AddrCtl value	Action
0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state

In state1, next state is looked up in dispatch ROM1

In state2, next state is looked up in dispatch ROM2



AddrCtl selects from: 0, ROM1, ROM2 or Output of Adder as next state

Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

6-bit opcode as index implies up to $2^6 = 64$ entries
Only 5 entries are used from ROM1
Only 2 entries are used from ROM2

AddrCtl is determined just by the state number

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

State number determines AddrCtl as well as datapath control bits

AddrCtl as well as datapath control bits can be stored in a control memory

State number	Control word bits 17-2	Control word bits 1-0
0	1001010000001000	11
1	00000000000011000	01
2	00000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

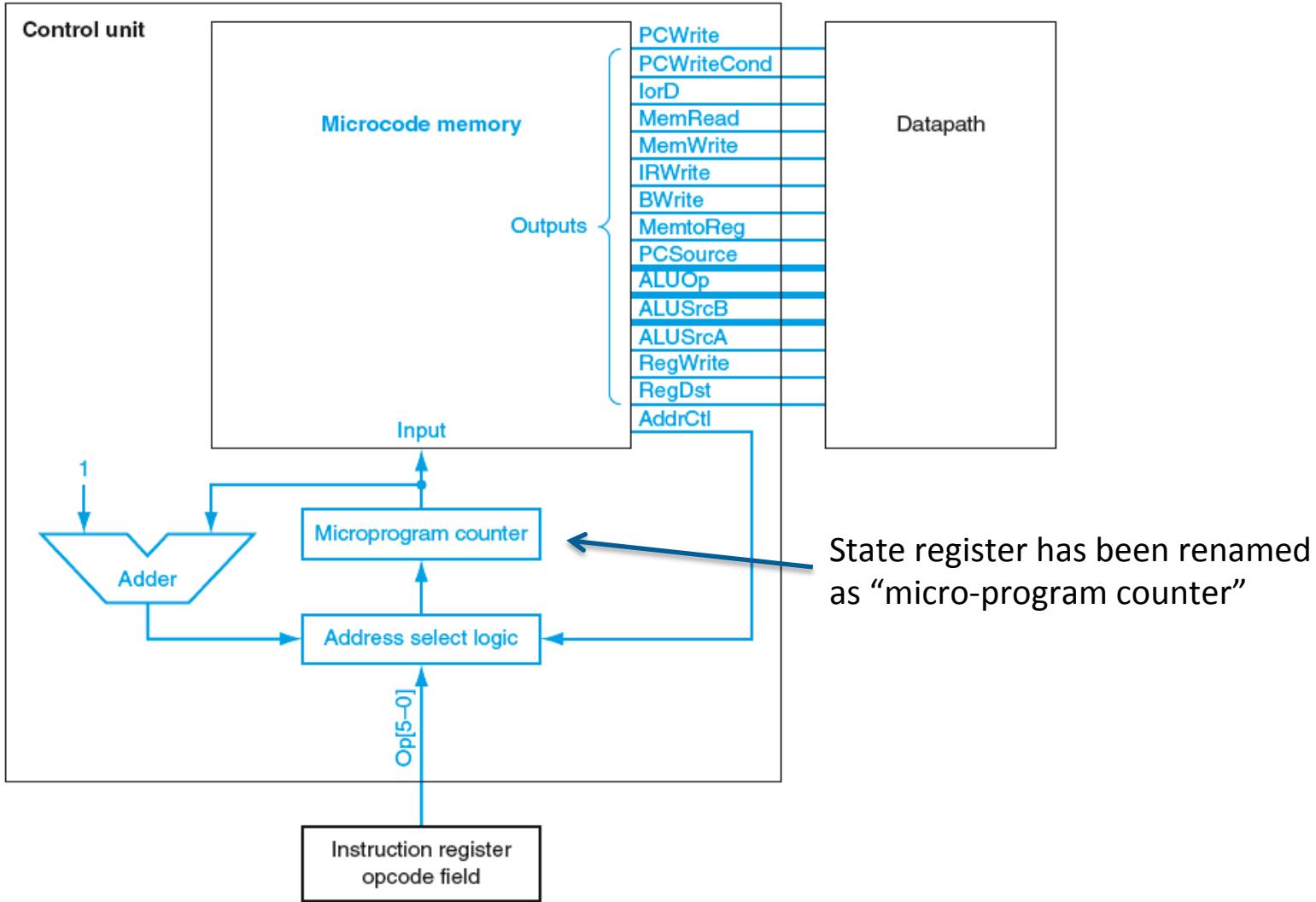
State number serves as address of all 18 bits (16 control + 2 AddrCtl)

Each 18-bit pattern can be viewed as a single “micro-instruction”

Micro-instructions are contained in the control store (ROM)

State number	Control word bits 17-2	Control word bits 1-0
0	1001010000001000	11
1	00000000000011000	01
2	00000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

Each micro-instruction takes one clock cycle



Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.

Field name	Value	Signals active	Comment
Memory	Read PC	MemRead, lrd = 0, IRWrite	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lrd = 1	Read memory using ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lrd = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00, PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	Jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

- Each micro-instruction performs one step in a machine instruction
- Each micro-instruction takes one clock cycle
- A machine instruction corresponds to a “micro-program”
- A micro-program is a sequence of micro-instructions
- Control memory containing micro-programs is on CPU chip
- Extra time is needed to retrieve and execute micro-programs

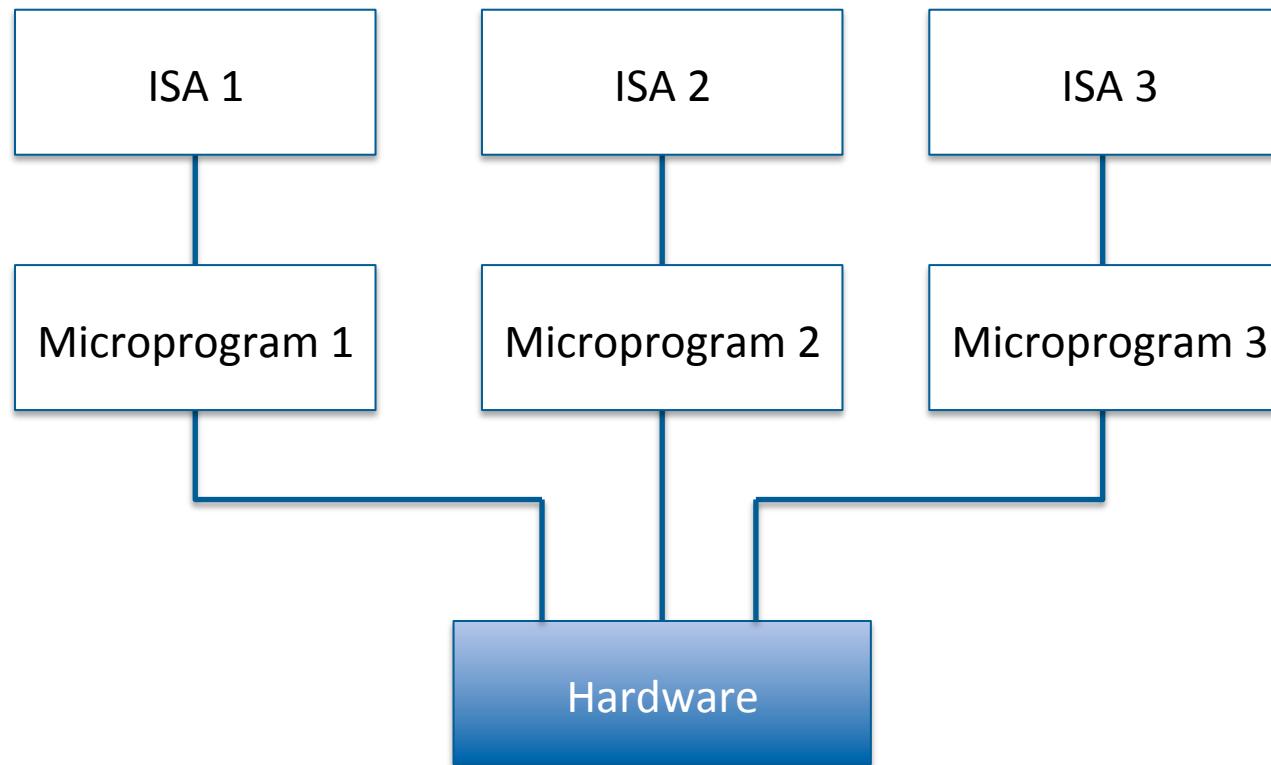
- Time required to retrieve micro-programs slows the system
- Micro-programming provides flexibility
- Complex instructions are easier to implement as micro-code
- CISC systems employ micro-programming
- Changing micro-code changes behavior of the system
- RISC systems use faster hardwired logic instead

- Each micro-instruction in our control ROM is 18 bits wide
- None of the fields within the micro-instructions are encoded
- This is fast since no decoding is required
- Systems of this type are said to be “minimally encoded”
- This is also called “horizontal micro-code”
- Micro-code is sometimes called firmware
- Firmware is harder to change than software
- Hardware is more difficult to change than firmware

- A writeable control store employs RAM for micro-code
- More realistic systems are more complex than our core MIPS
- Such systems could require very wide micro-instructions
- Encoding fields within these micro-instructions would save bits
- However these take longer to process due to need for decoding
- These systems employ maximally encoding (“vertical micro-code)

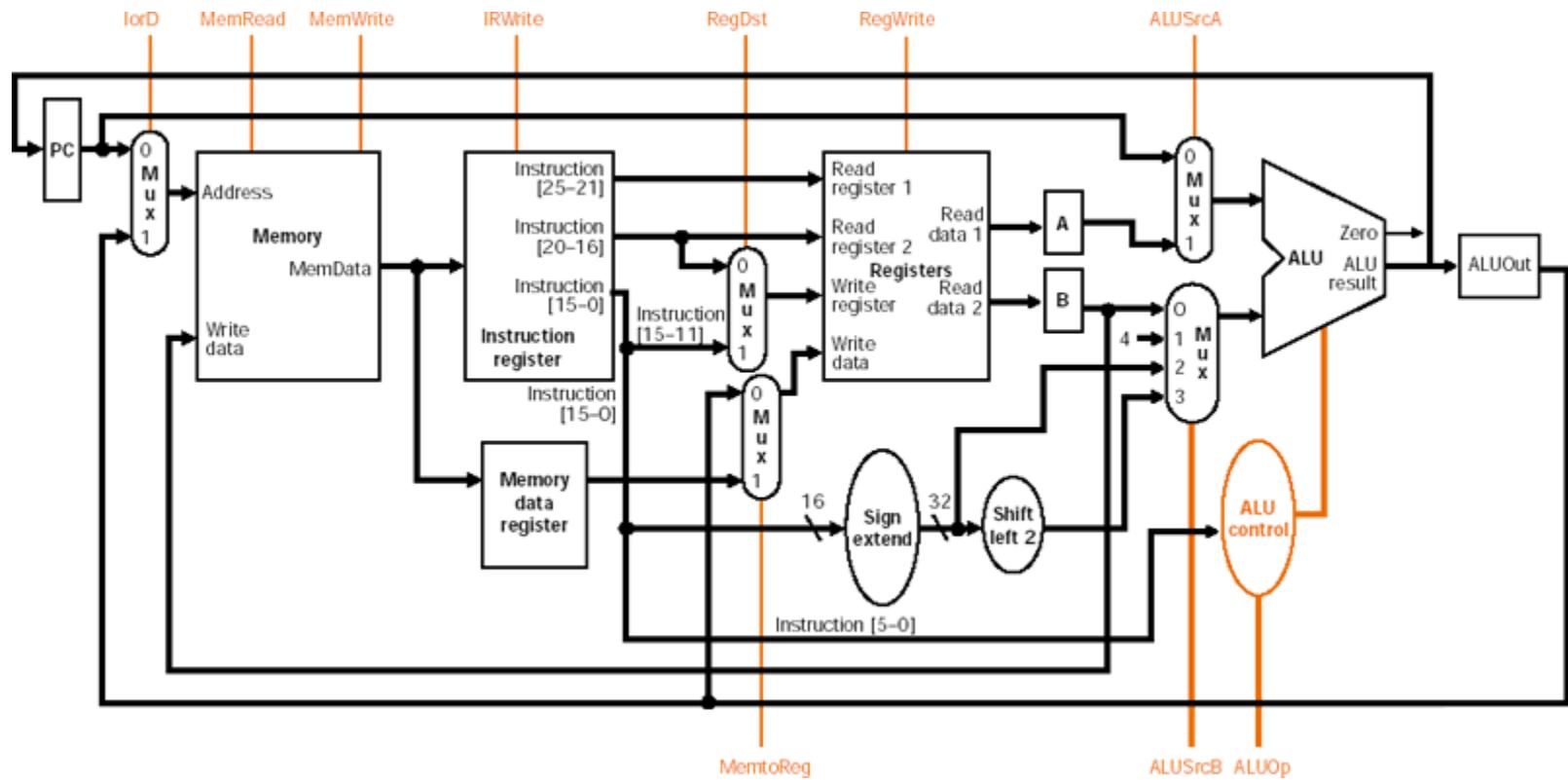
- Micro-programming enables one system to behave like another
- This is called “emulation”
- It allows the same hardware to be used with different ISAs
- ISA is the instruction set architecture
- Different machine code can execute on the same hardware
- The micro-code is said to interpret the machine instructions

Emulation allows the hardware to run different machine programs



- Instructions can be executed in multiple steps
- Longest step determines clock period
 - Critical path: memory access time
 - Each step takes one clock cycle
- Different control signals are generated in different clock cycles
- Change in state occurs with each cycle
- Sequential logic contains state
- Finite-state machine can model behavior

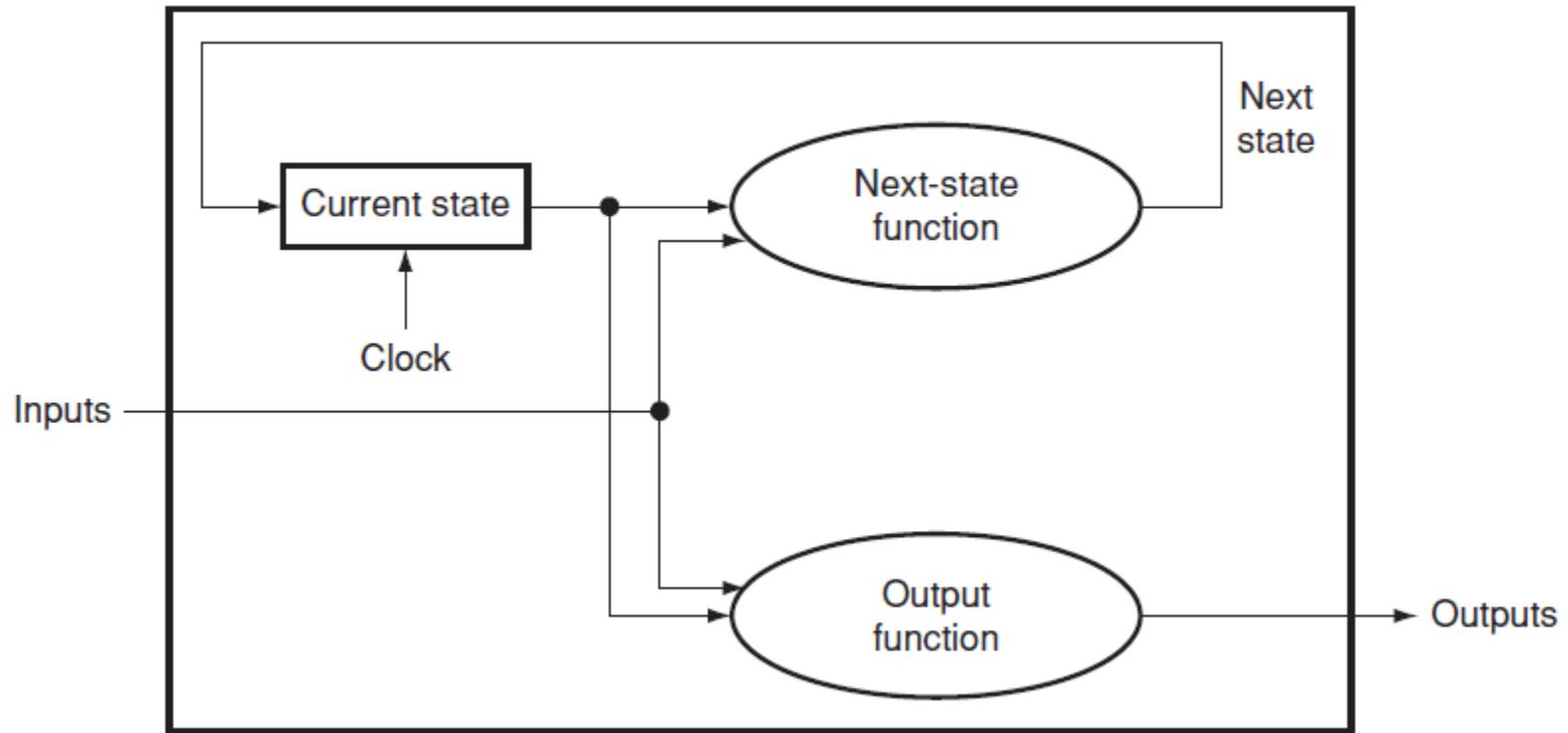
Multi-cycle Datapath



- Component re-use eliminates duplication and reduces costs

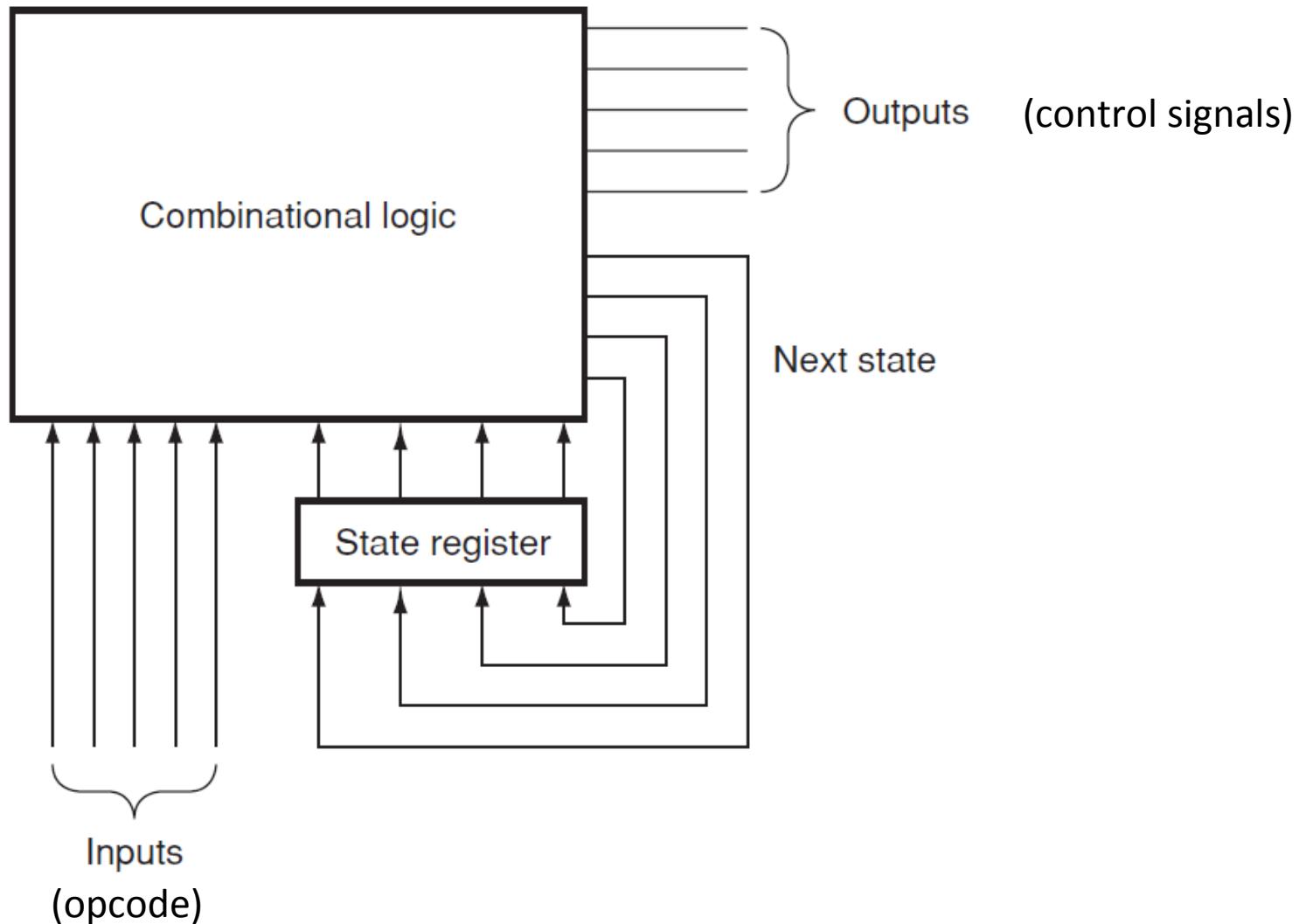
Also called “state machine” or FSM and contains:

- Set of states (one of which is the initial start state)
- Set of inputs
- Set of outputs
- Next-state function maps current state and inputs to a new state (state transition)
- Output function maps current state to set of outputs
 - Outputs may also depend on inputs
 - “Moore machine” outputs depend only on state
 - “Mealy machine” outputs depend on state and inputs

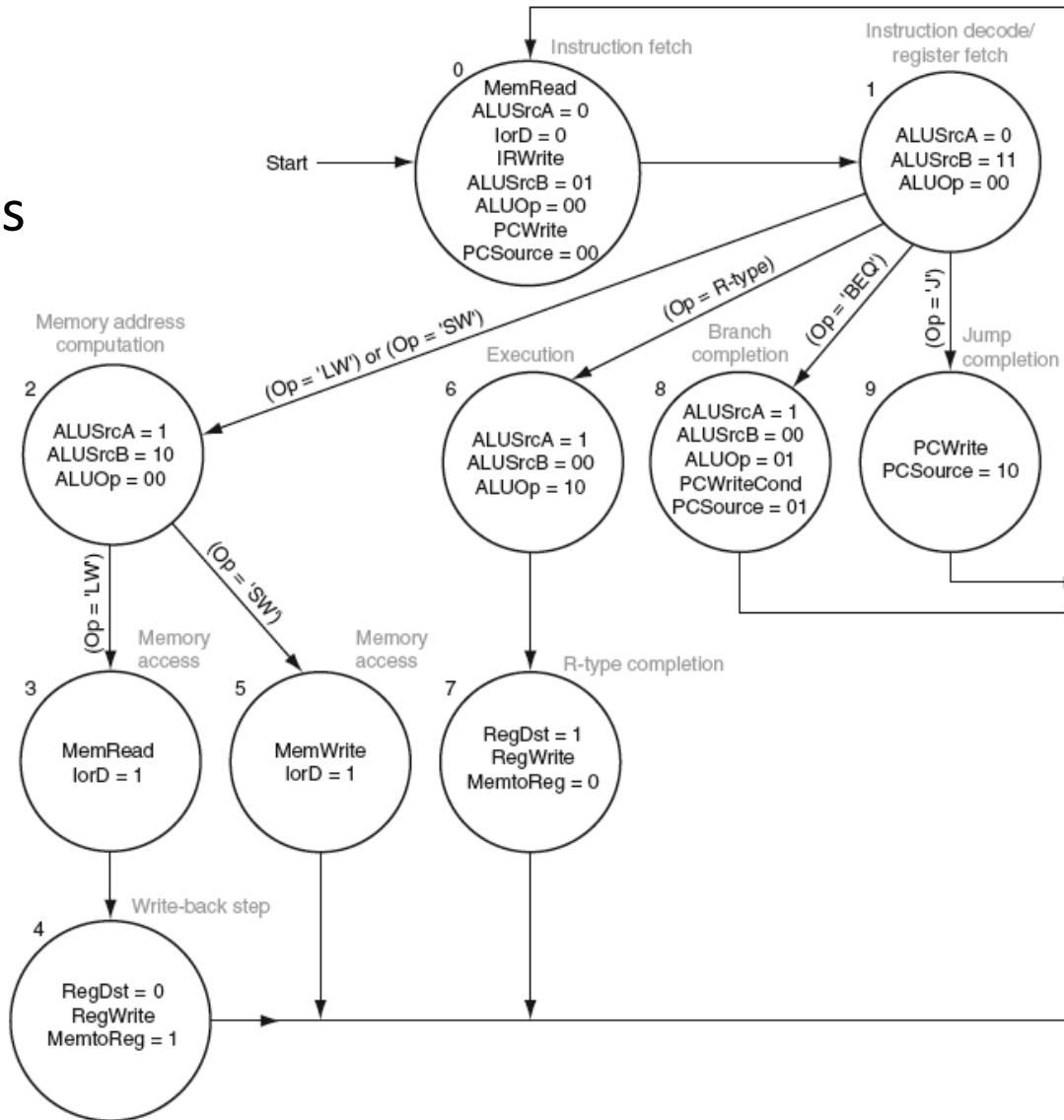


Sequential logic, internal storage contains the state information

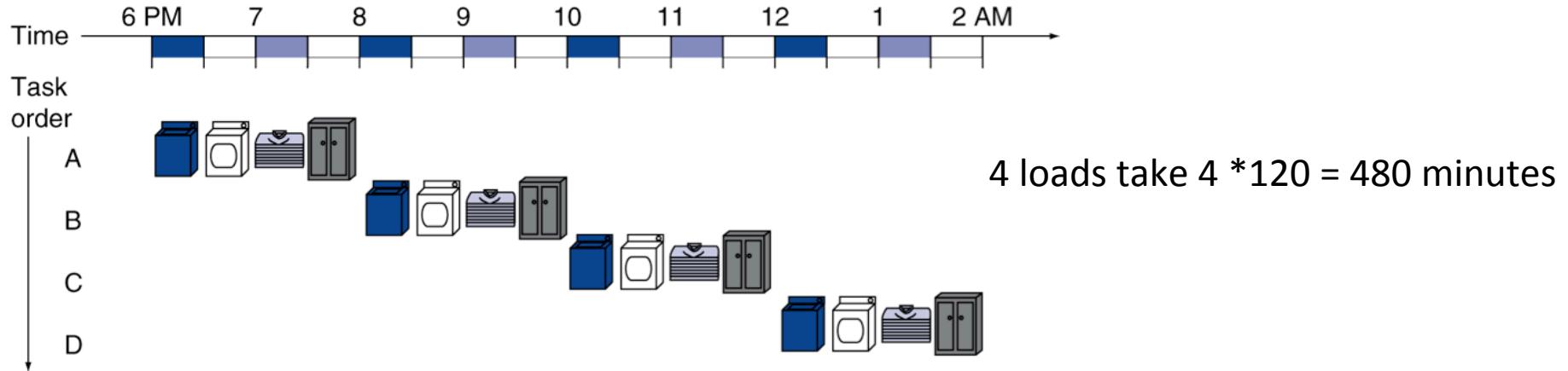
- Set of inputs are the opcode bits
- Set of outputs are the control signals
- Next-state function is implemented with combinational logic
- New state is computed synchronously with clock cycle



MIPS core instruction subset requires 10 states

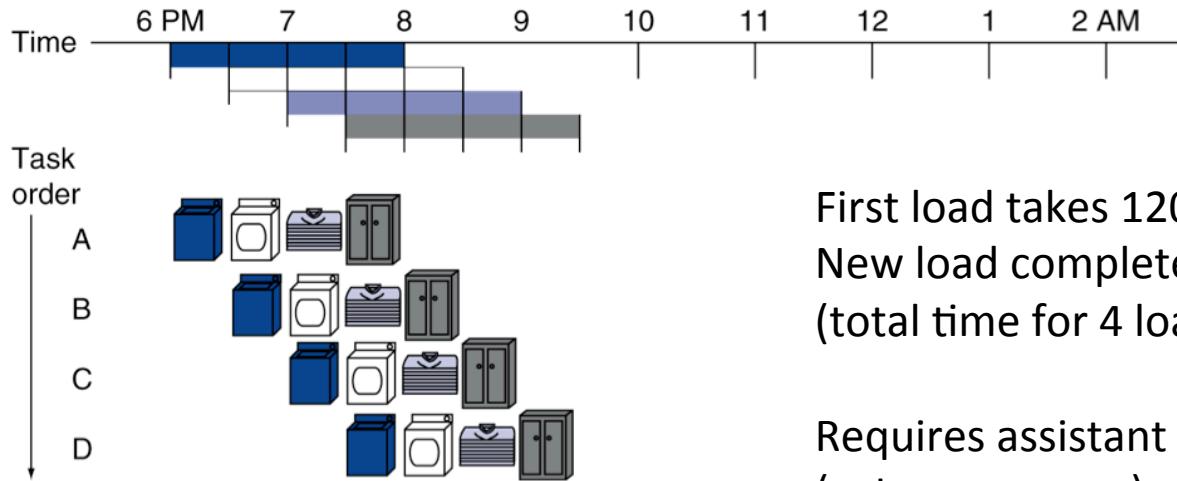


Doing multiple loads of laundry



1. Use washer for next dirty load of clothes (30 min.)
2. When done, use dryer to dry clothes (30 min.)
3. When done, fold the clean dry clothes (30 min)
4. When done, put the clothes away (30 min)

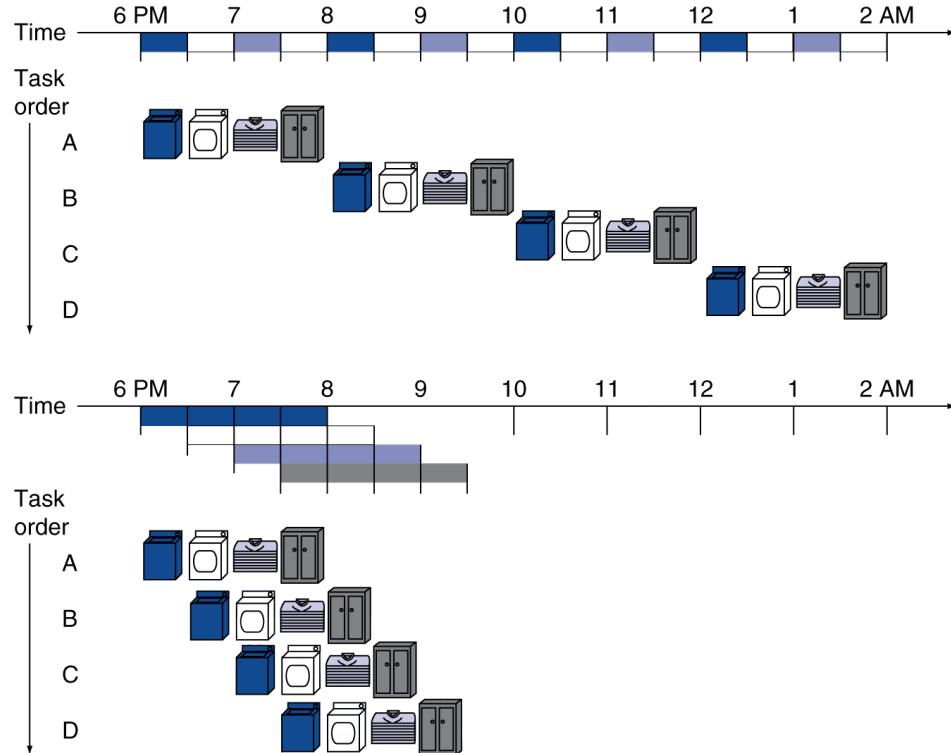
Parallelism improves performance by overlapping



First load takes 120 minutes
New load completes every 30 min thereafter
(total time for 4 loads = $120 + 90 = 210$ min)

Requires assistant to work in parallel
(extra resources)

1. Wash 1st load
2. When done place 1st into dryer and wash 2nd
3. When done, fold 1st load, dry 2nd and wash 3rd
4. When done, put away 1st, fold 2nd, dry 3rd and wash 4th



- Four loads:
 - Speedup
 $= 480/210 = 2.3$
- Non-stop:
 - Speedup
 $= 120n/(120+30n) \approx 4$
= number of stages

Each load still takes 120 minutes (2 hours)
But more loads are completed per hour

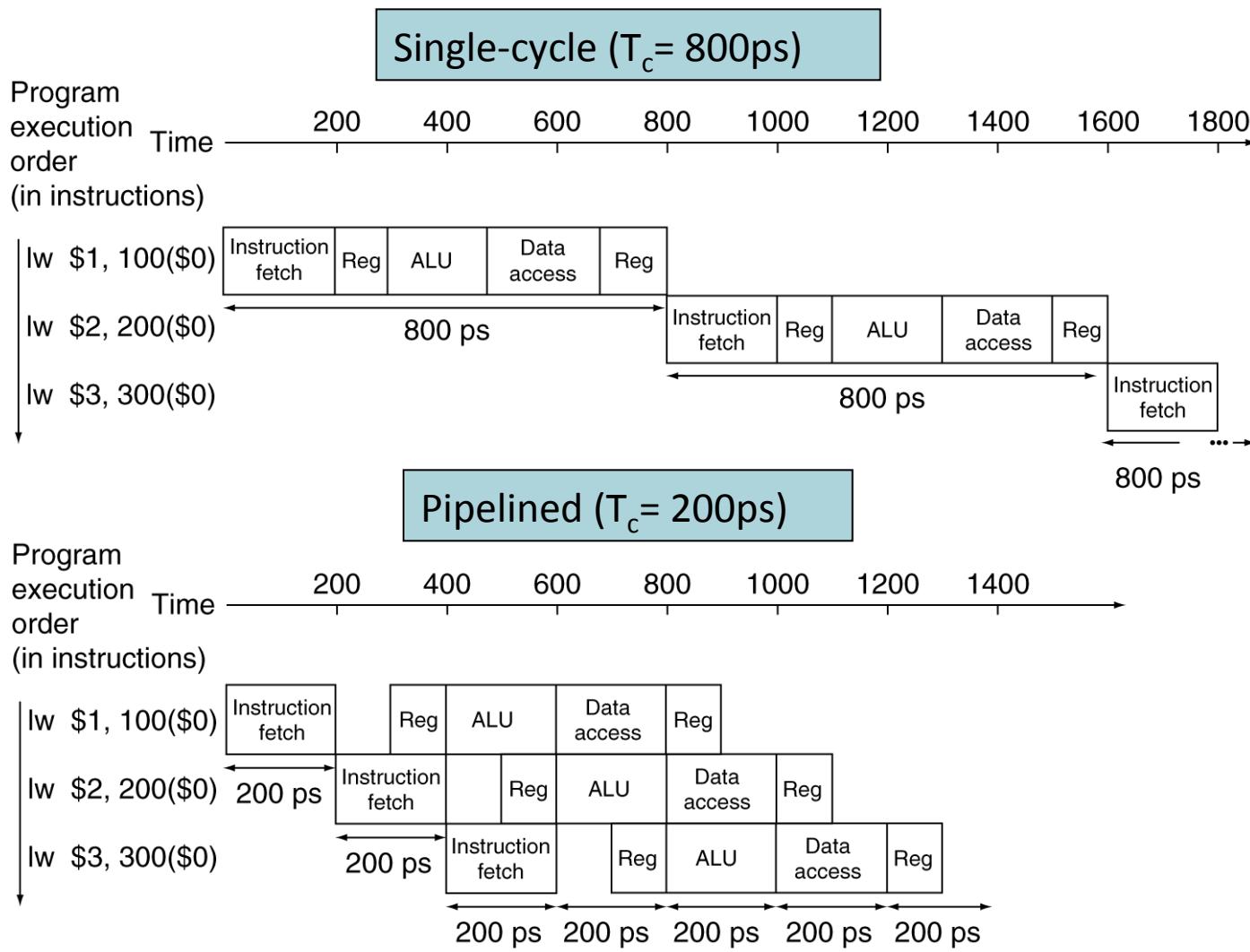
Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

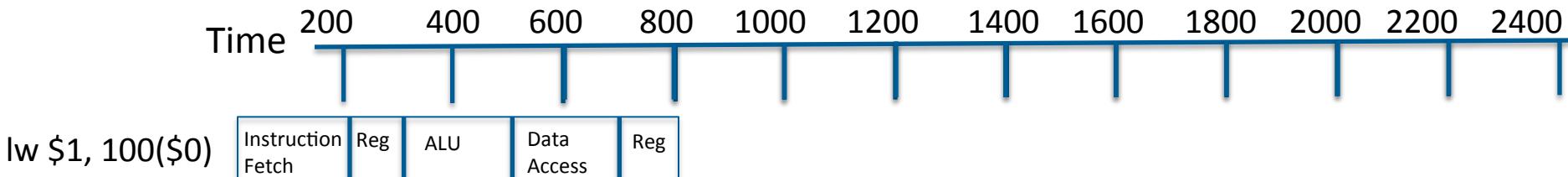
- All instructions are 32-bits
 - Easier to fetch and decode in one cycle
 - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
 - Can decode and read registers in one step
- Load/store addressing
 - Can calculate address in 3rd stage, access memory in 4th stage
- Alignment of memory operands
 - Memory access takes only one cycle

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps



Single-Cycle Performance



add \$2, \$1, \$1



sw \$1, 100(\$0)

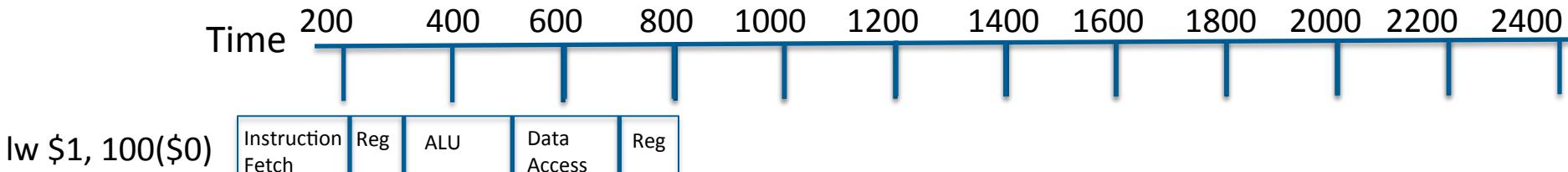


Single-cycle ($T_c = 800\text{ps}$)
 $T_{total} = 2400 \text{ ps}$

Cycle time matches longest instruction



Multi-Cycle Performance



add \$2, \$1, \$1



sw \$1, 100(\$0)



Multi-cycle with variable cycle time

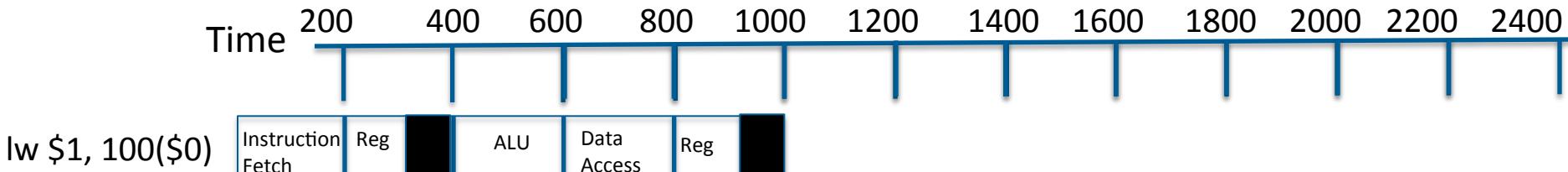
$$T_{\text{total}} = 800 + 600 + 700 = 2100 \text{ ps}$$

Multi-cycle with fixed cycle time (200ps)

$$T_{\text{total}} = 1000 + 800 + 800 = 2600 \text{ ps}$$

Cycle time matches longest step

Pipelined Performance



add \$2, \$3, \$3



sw \$2, 100(\$0)

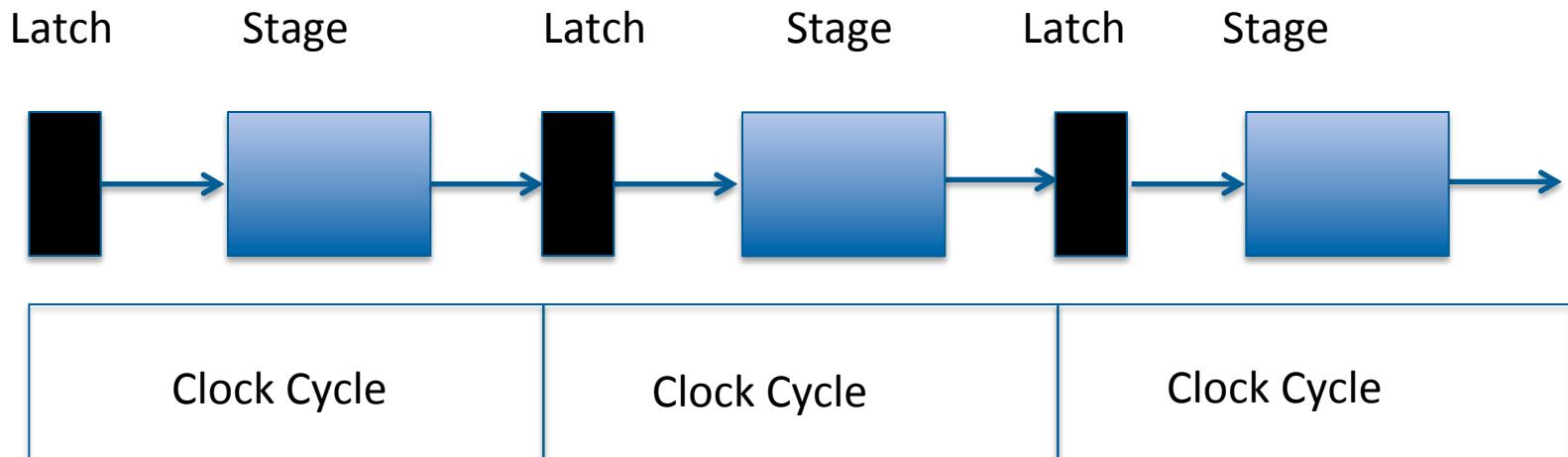


Pipelined-cycle ($T_c = 200\text{ps}$)
 $T_{total} = 1000 + 200 + 200 = 1400 \text{ ps}$

Cycle time matches longest step



Pipelining



- Although it is possible to divide the work into approximately equal stages, they are unlikely to be exactly equal
- Latches are added to synchronize the stages
- The latches themselves may add a small latency to each stage (folded into cycle time)

$$T_{\text{pipelined}} = T_{\text{nonpipelined}} \div \text{Number of stages}$$

Speedup $\approx N$ assuming:

- independent instructions

- balanced pipeline stages

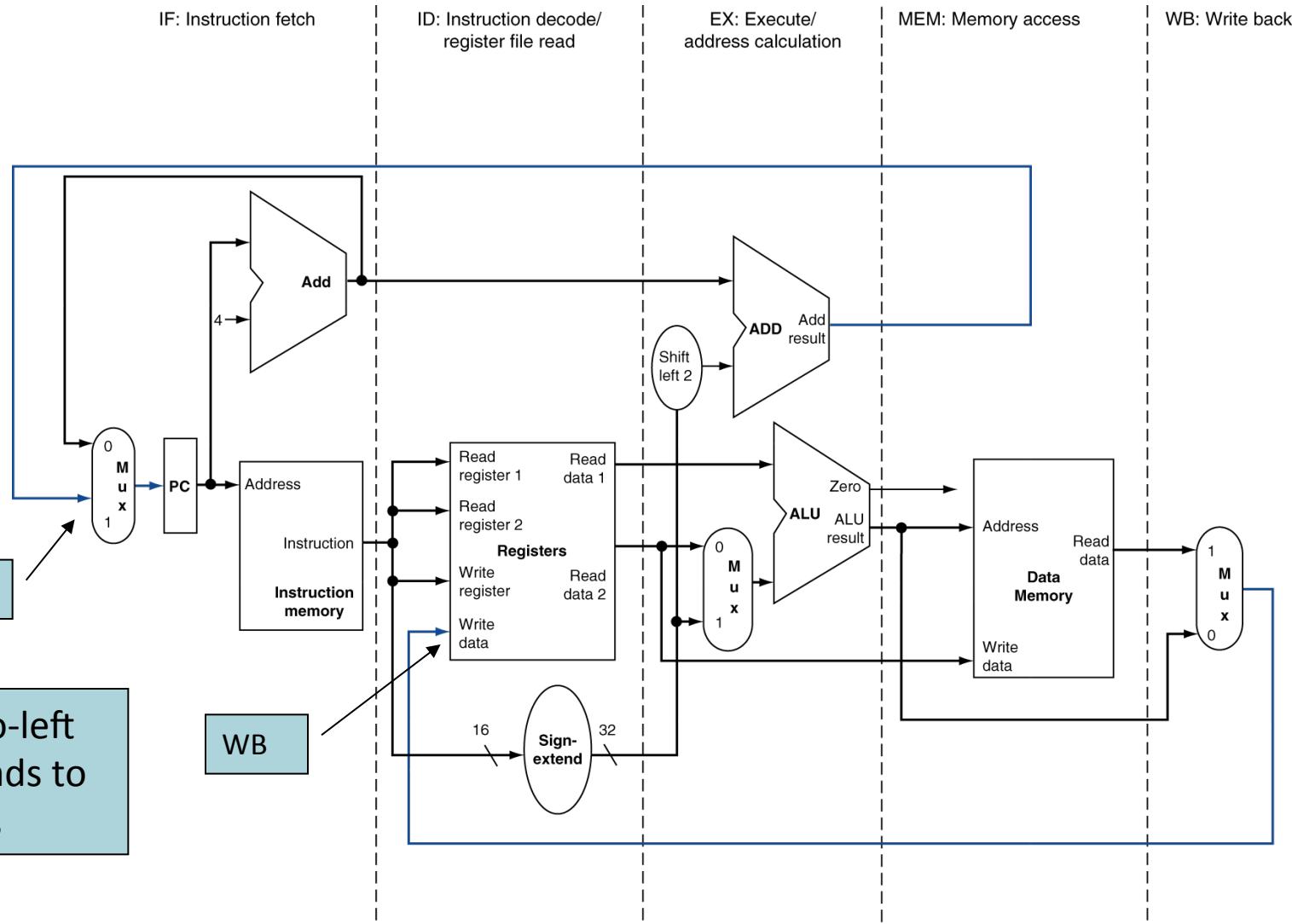
Dependencies cause stalls and reduce speedup

It is throughput that is increased

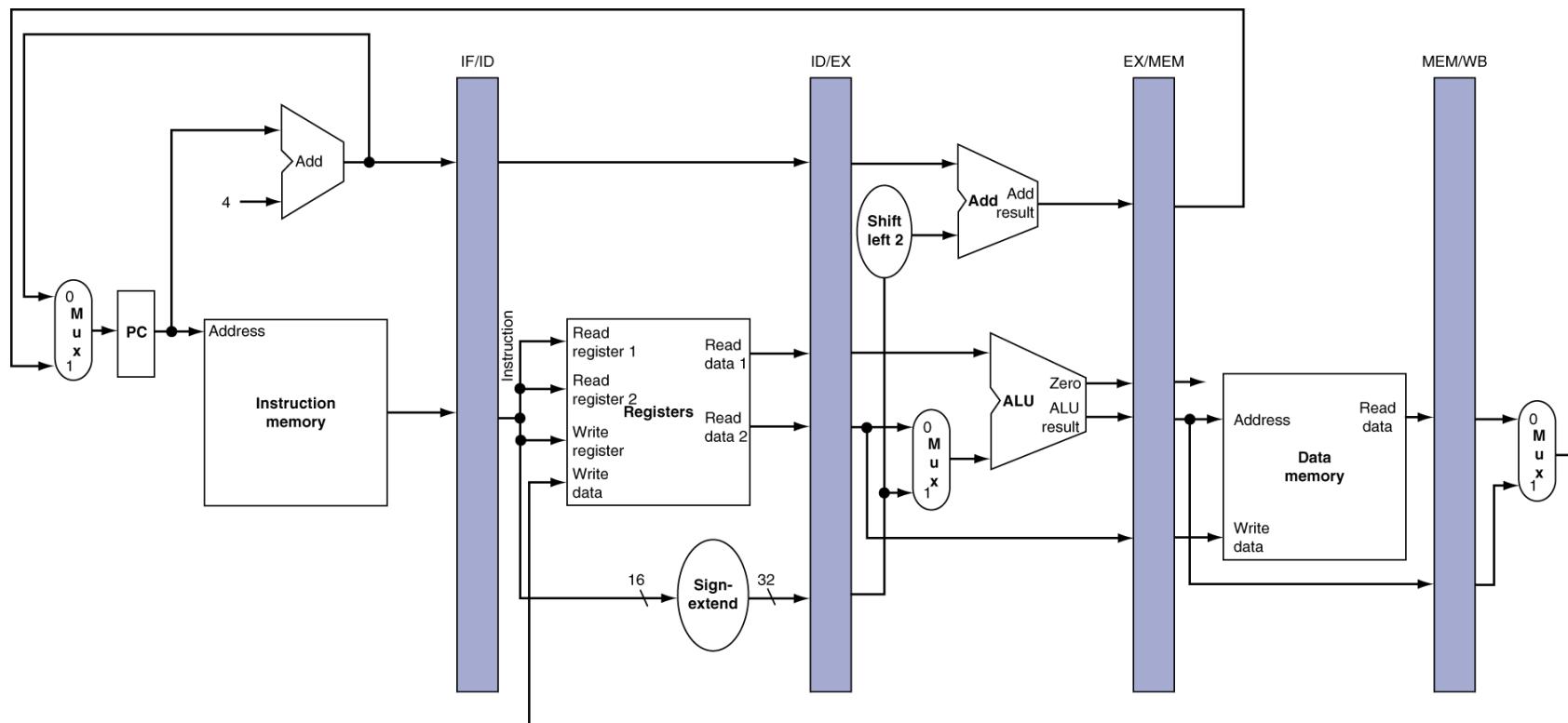
Latency or fill time is 5 cycles

- A new instruction completes for each cycle thereafter

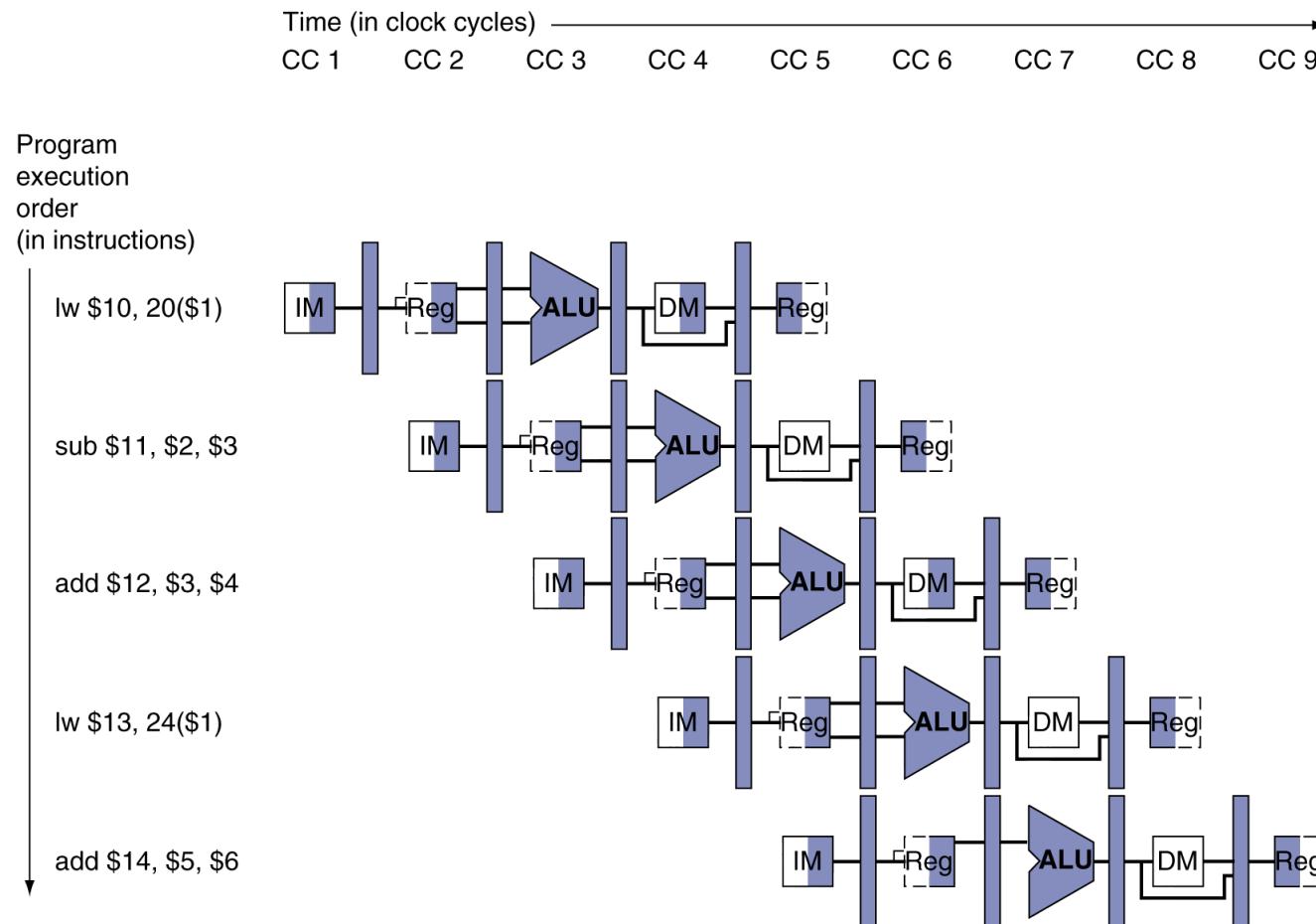
- Each instruction still takes 5 cycles



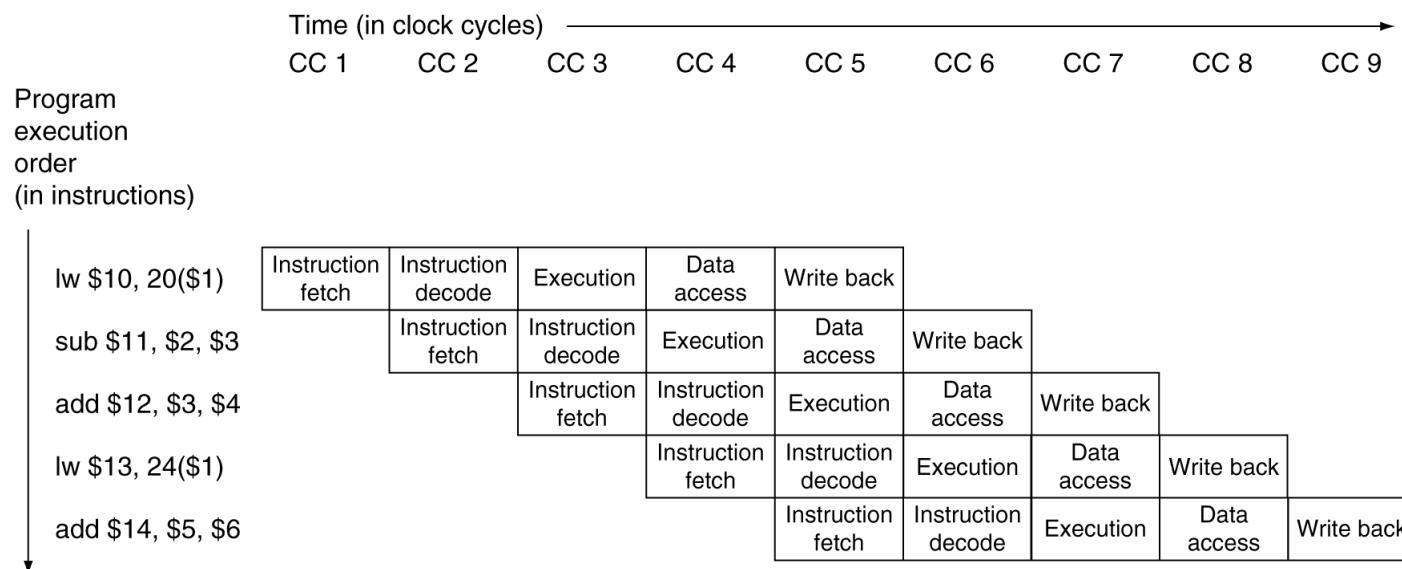
- Need registers between stages
 - To hold information produced in previous cycle



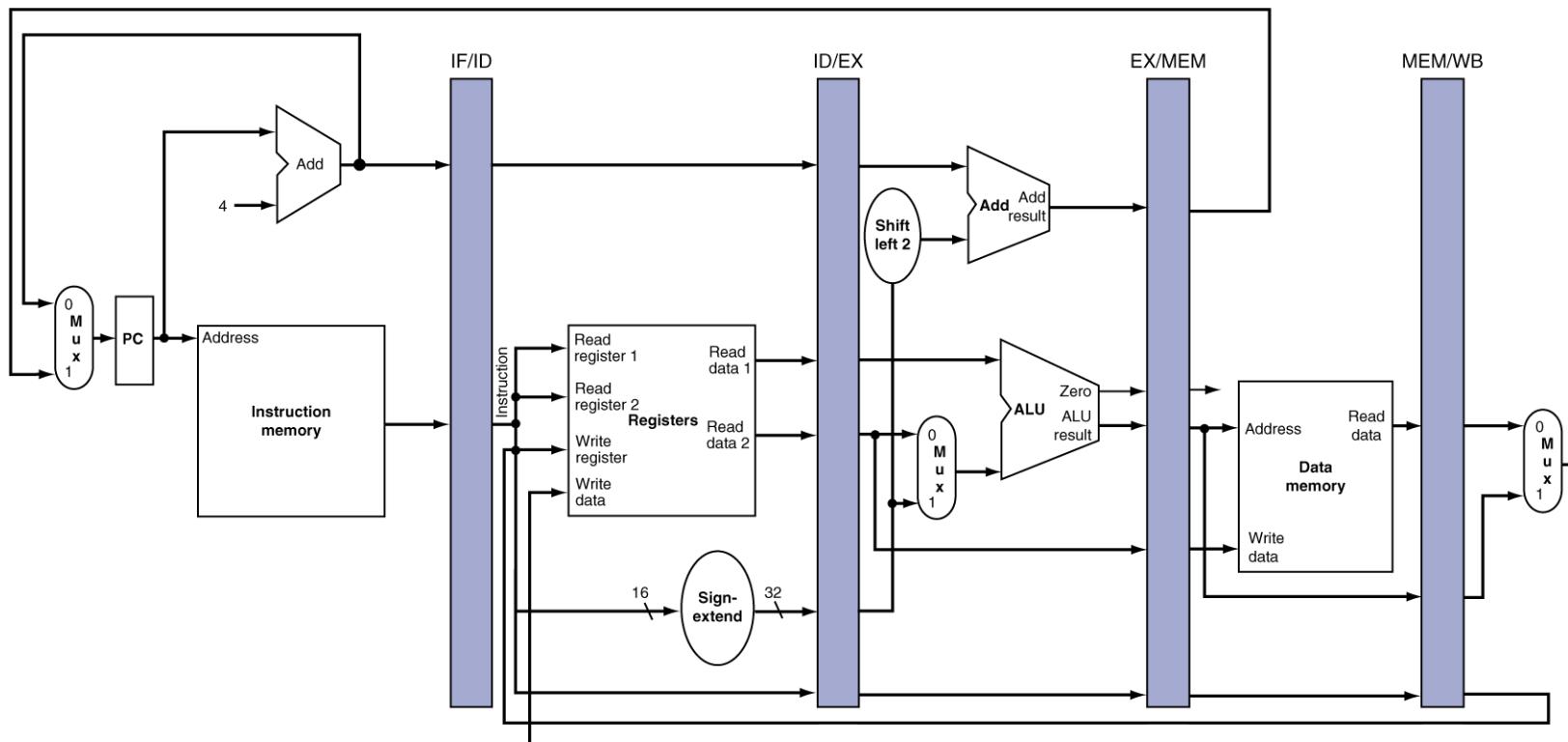
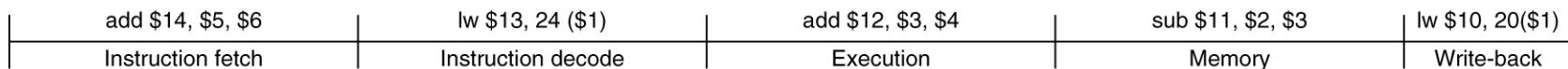
Form showing resource usage

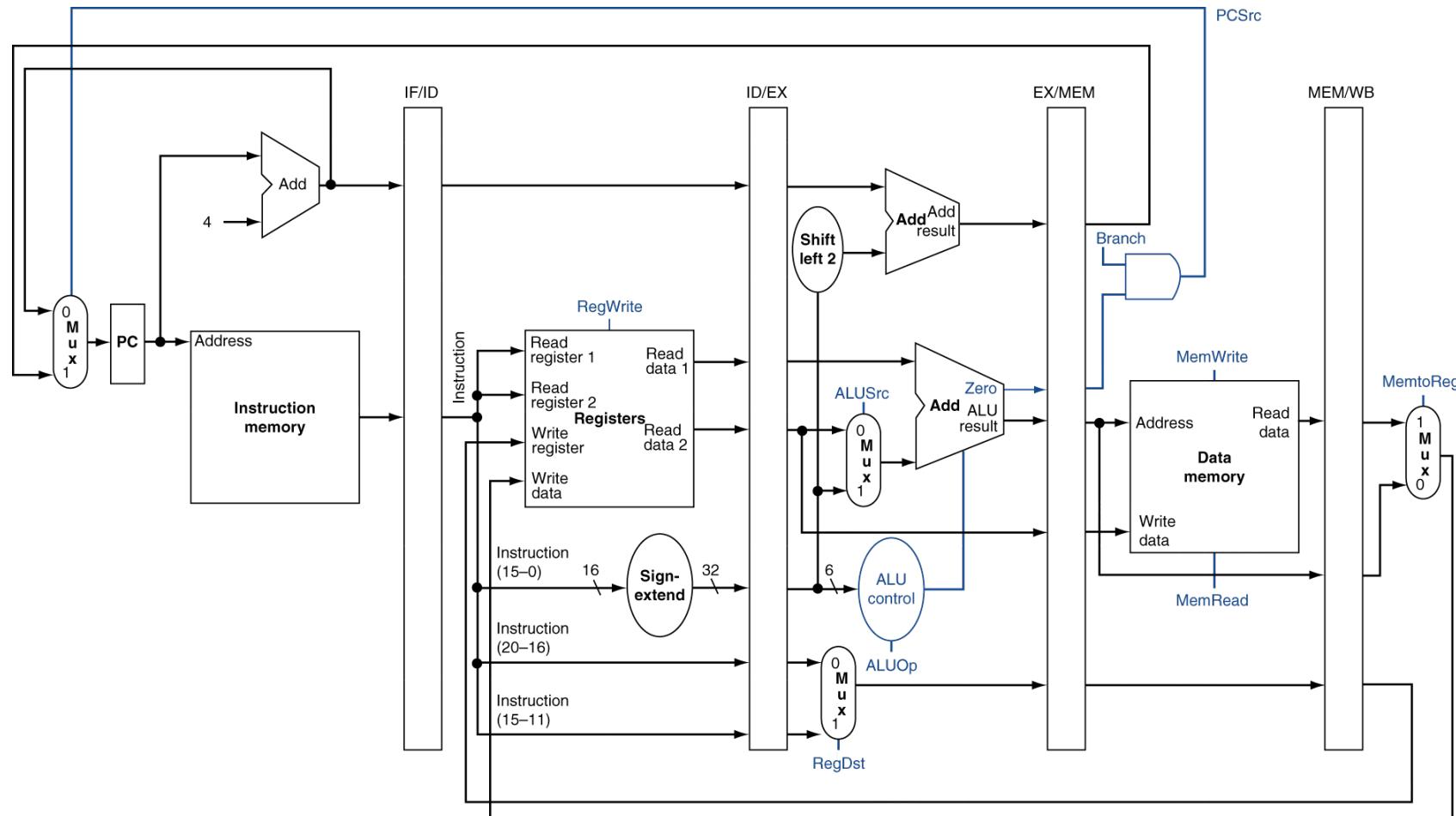


■ Traditional form

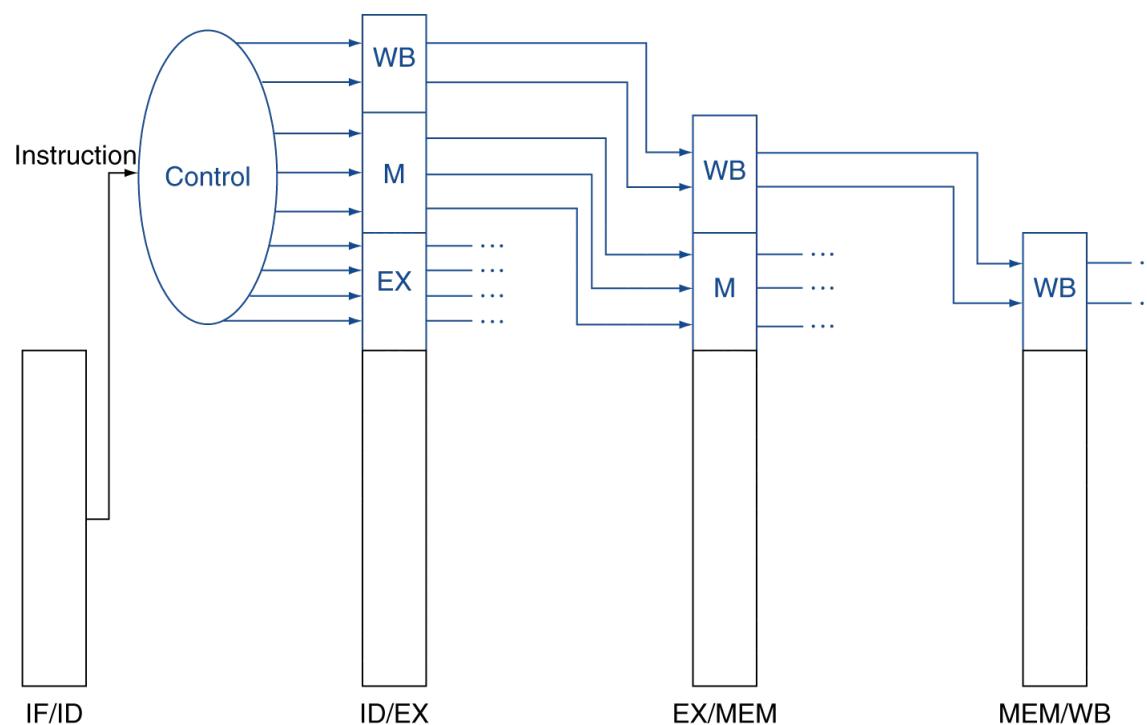


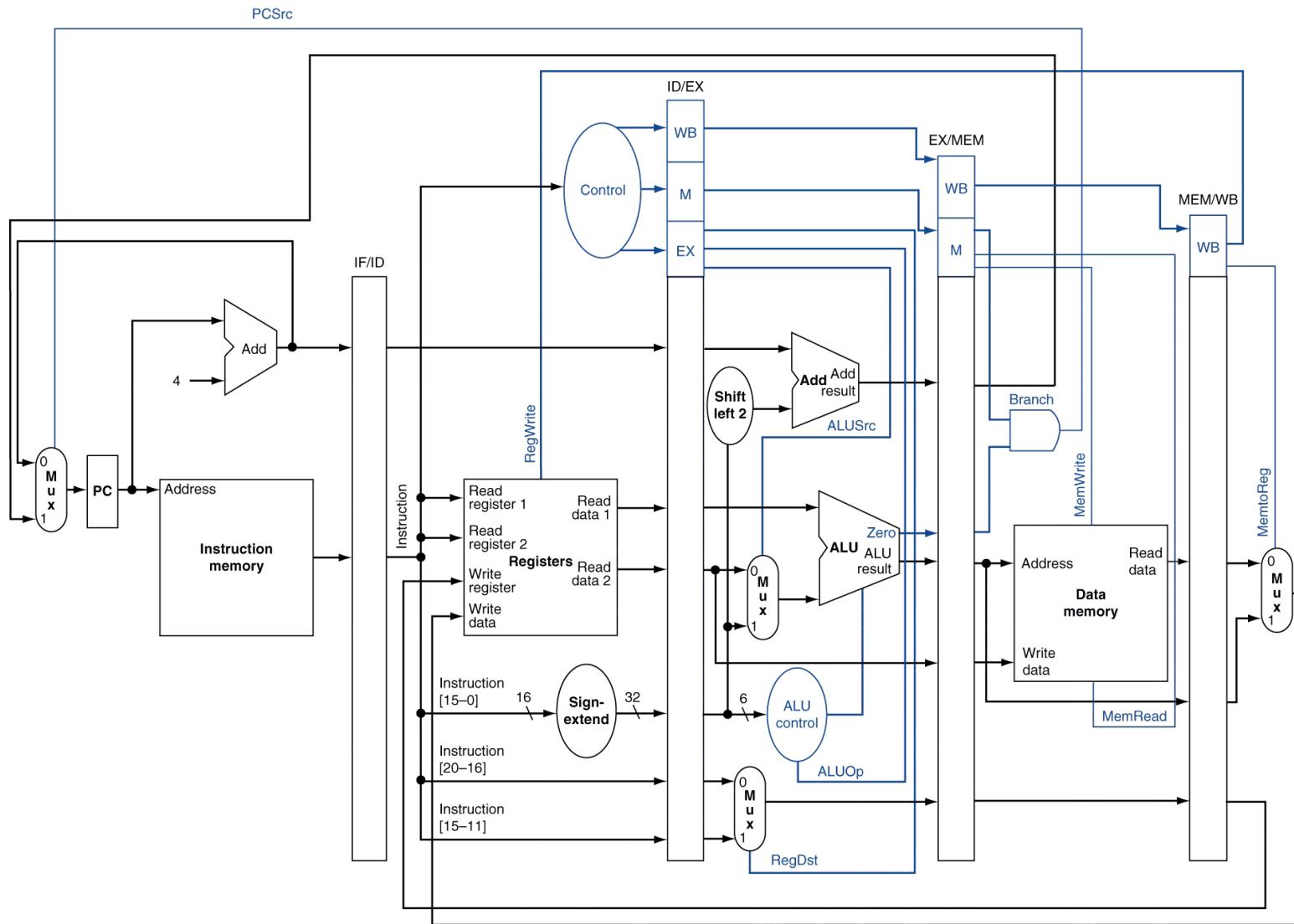
State of pipeline in a given cycle





- Control signals derived from instruction
 - As in single-cycle implementation



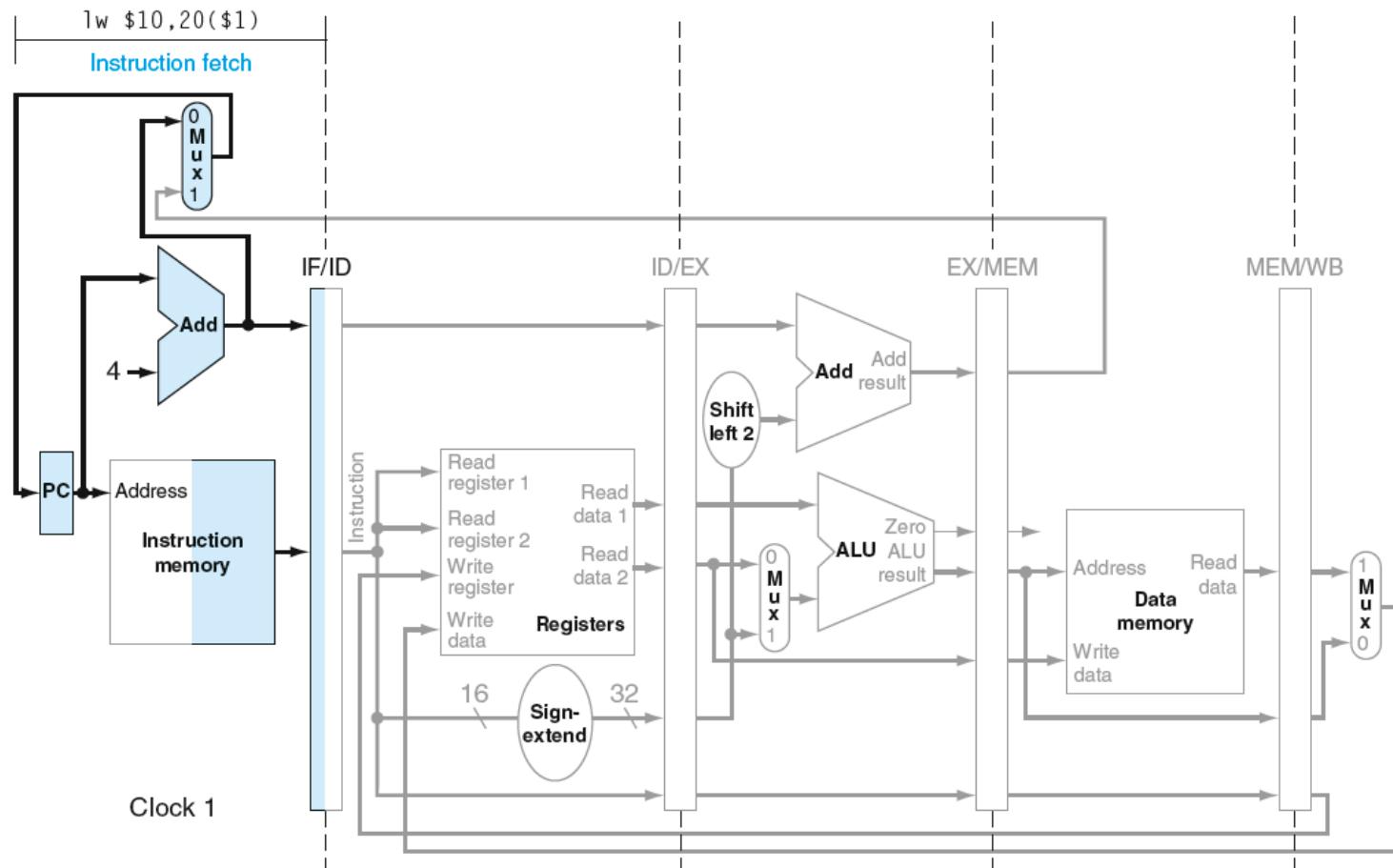


Pipelining the instructions below produces the expected results

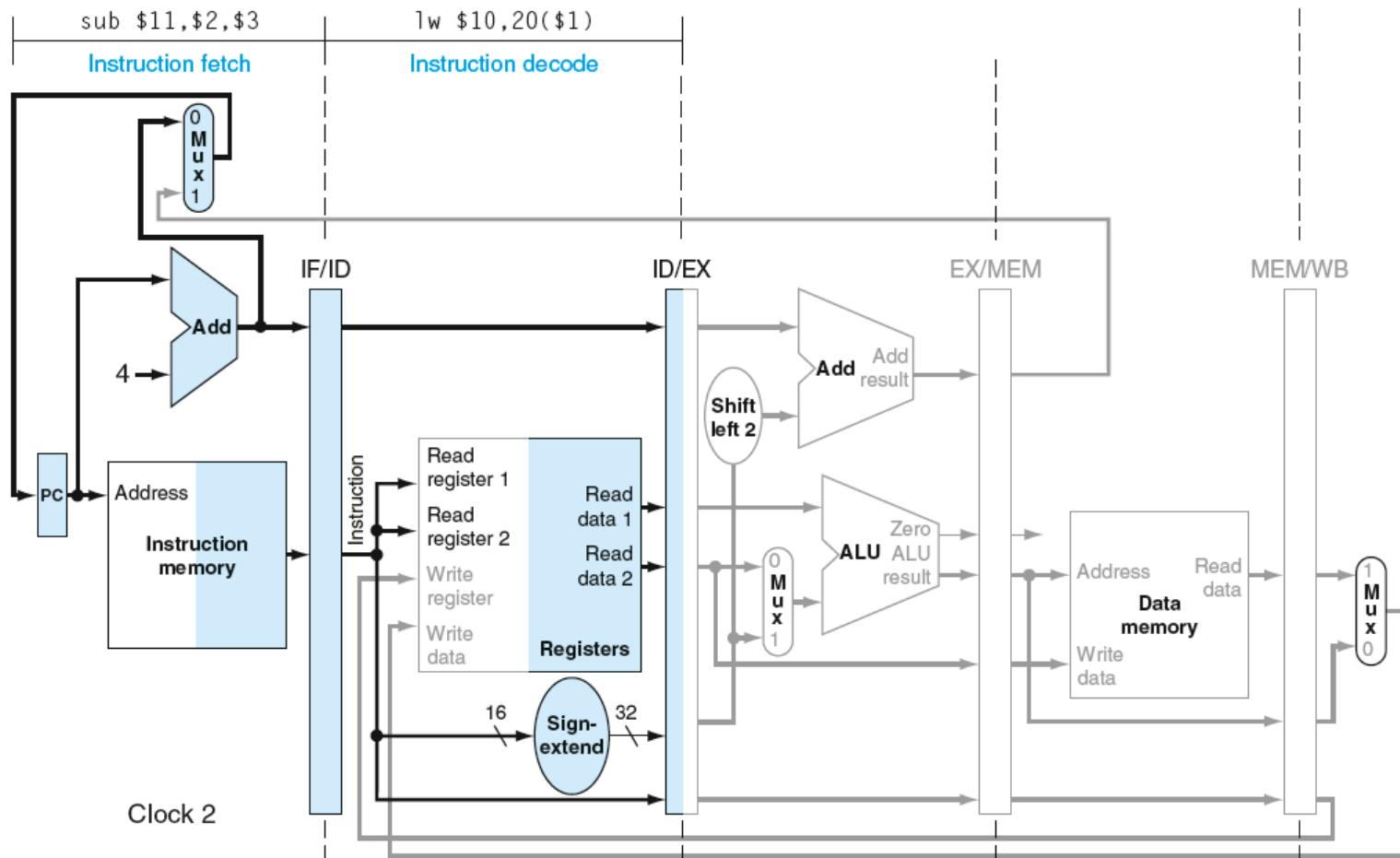
lw	\$10, 20(\$1)
sub	\$11, \$2, \$3
add	\$12, \$3, \$4
lw	\$13, 24(\$1)
add	\$14, \$5, \$6

There are no dependencies, so no stalls are needed

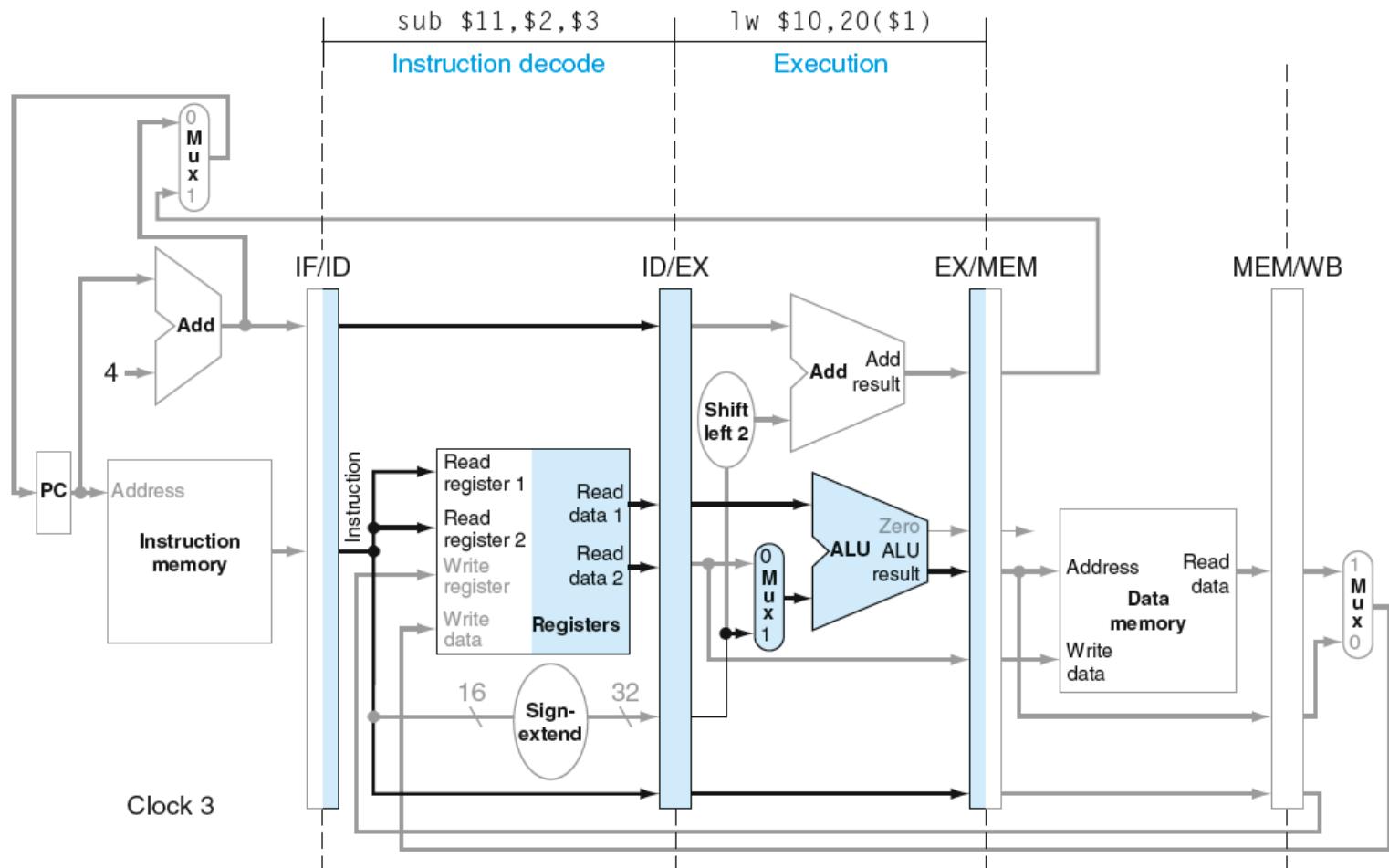
As a simple example, we can trace the first 2 instructions



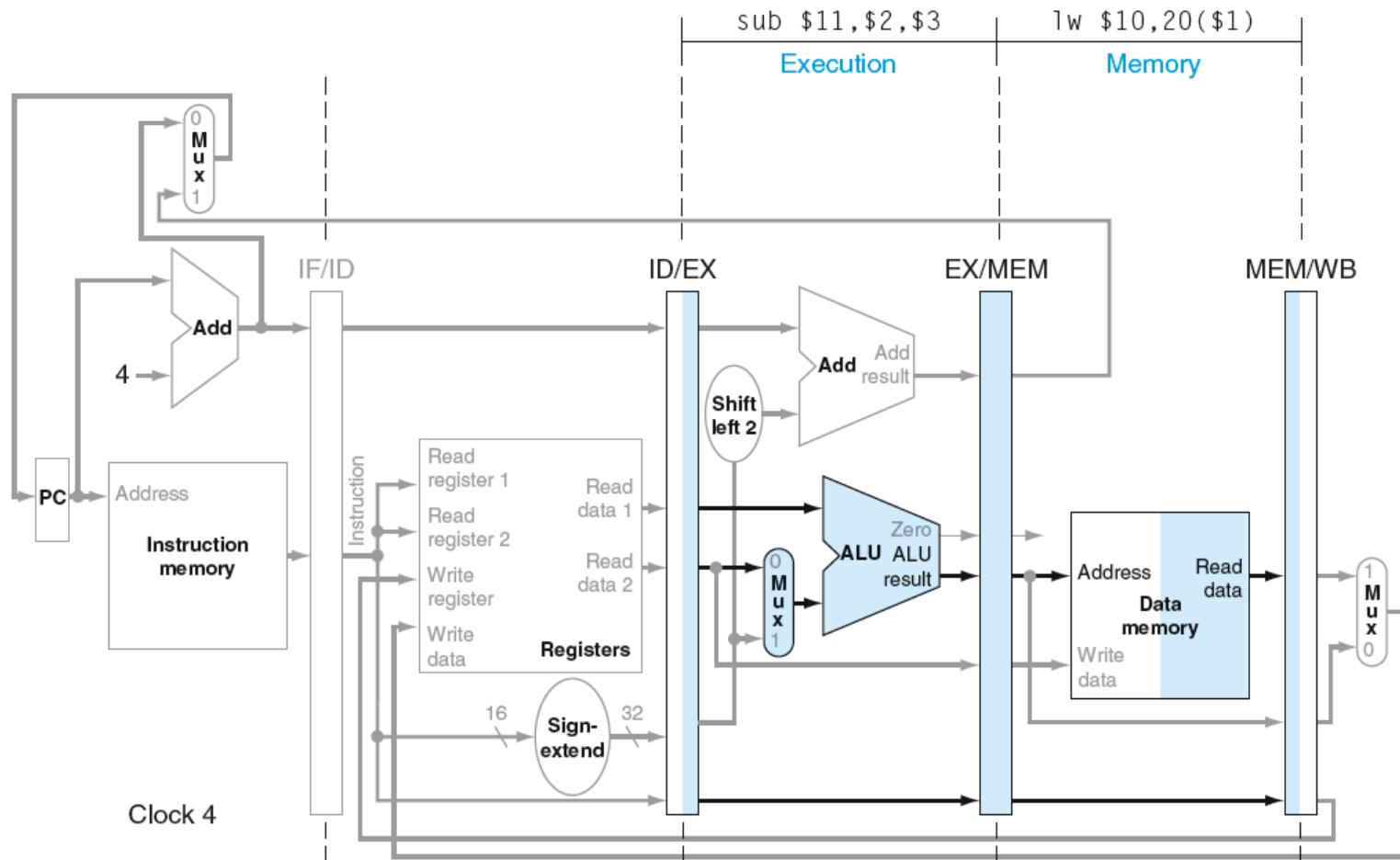
LW is fetched in cycle 1



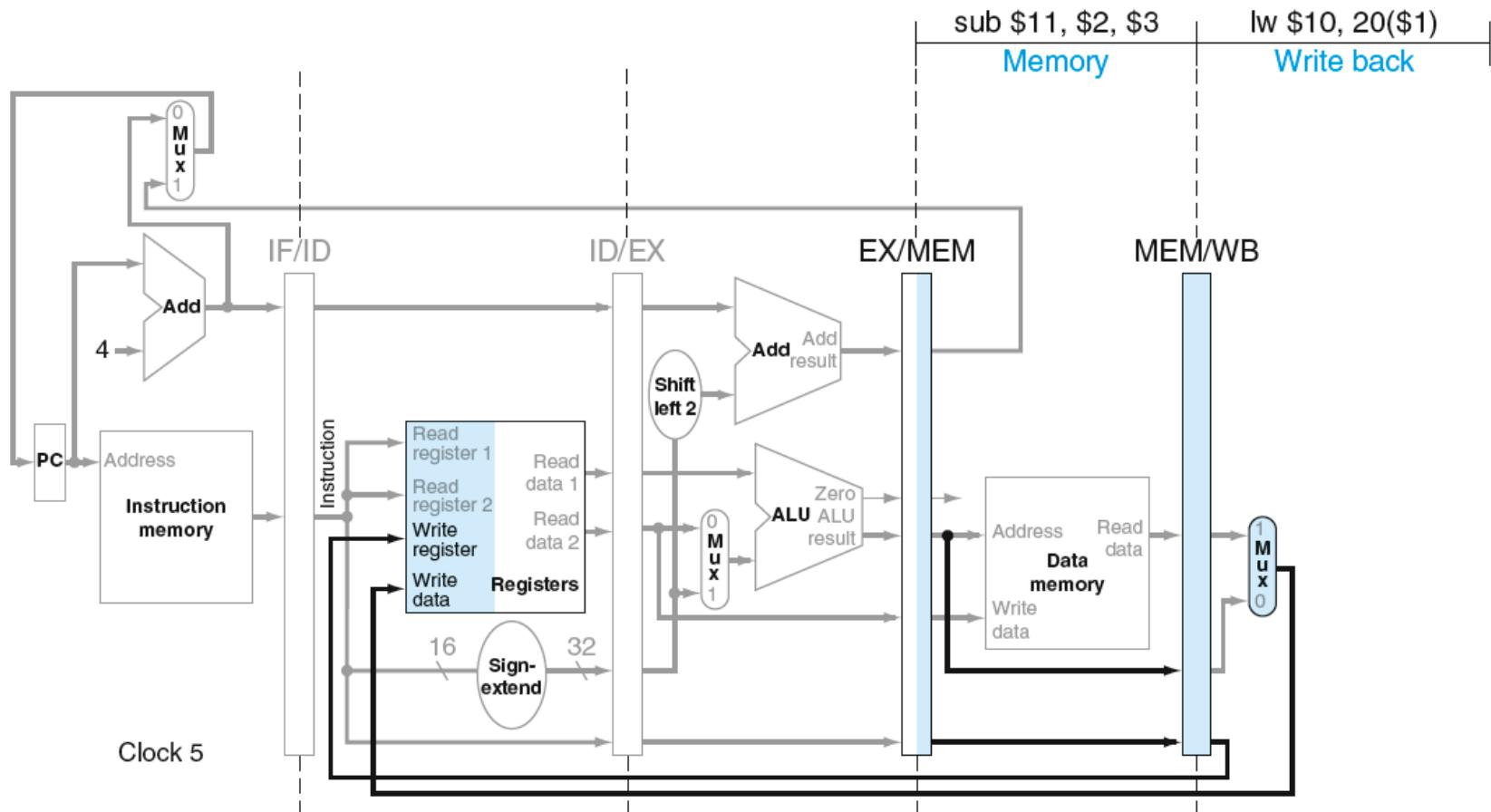
In cycle 2, LW advances to the decode stage and SUB is fetched



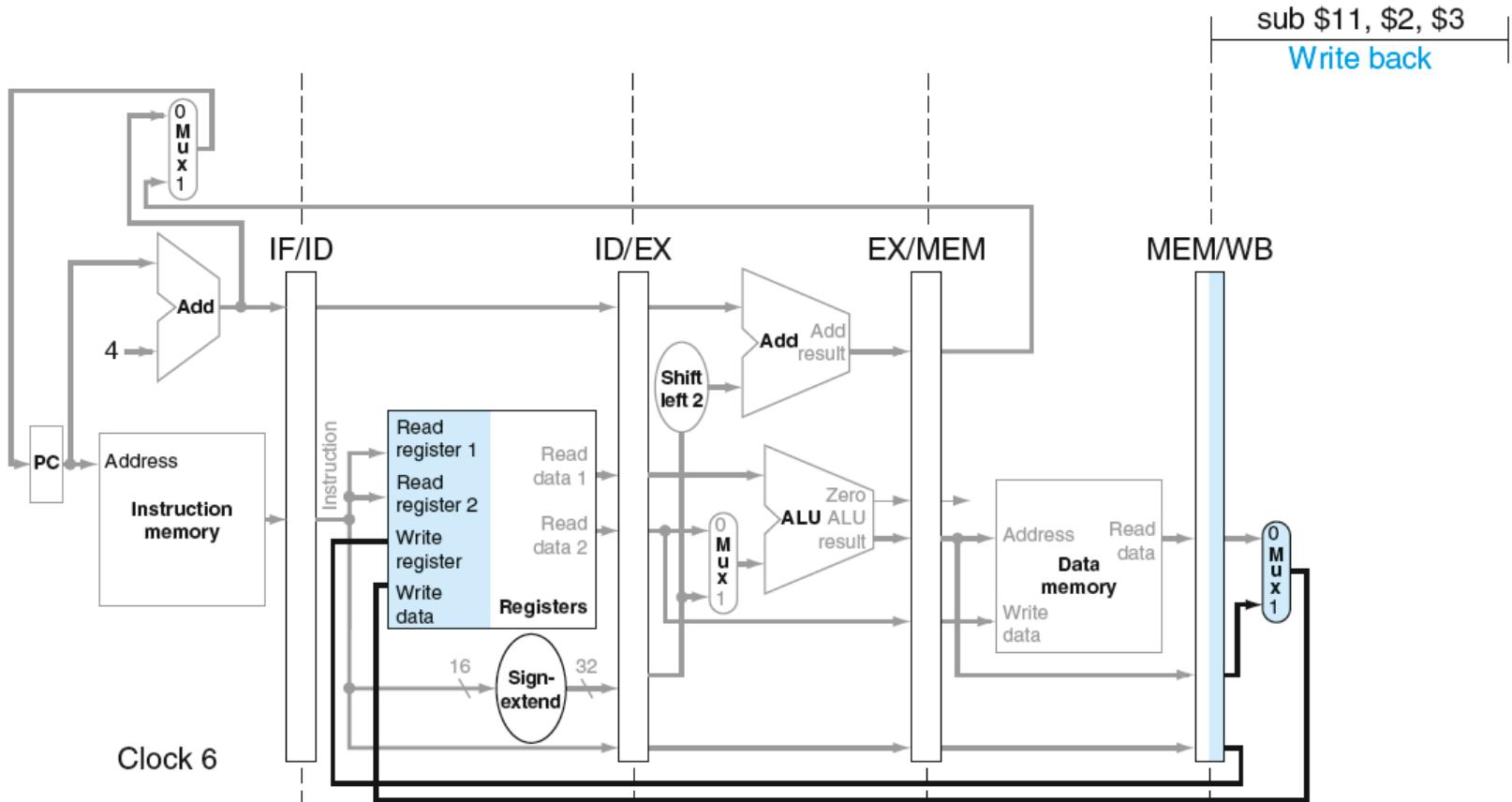
By cycle 3, LW is in the execute stage and SUB is in the decode stage



LW reads from the data memory in cycle 4, while SUB executes in stage 3



LW completes in cycle 5 by writing its result into \$10
SUB simply idles in stage 4 since it does not access memory



SUB completes in cycle 6, writing its result into \$11
The two instructions require a total of 6 cycles to complete

- Pipelining provides a higher throughput
 - More instructions complete per unit time
 - Each instruction takes 5 cycles
 - But instructions are overlapped
- On the non-pipelined multi-cycle system
 - lw takes 5 cycles
 - sub takes 4 cycles
 - Total for this example = 9 cycles instead of 6
 - Each instruction completes before the next starts

This example traces 5 instructions through the pipeline:

lw	\$10, 20(\$1)
sub	\$11, \$2, \$3
and	\$12, \$4, \$5
or	\$13, \$6, \$7
add	\$14, \$8, \$9

There are 5 stages, so it takes 5 instructions to fill the pipeline
The result registers \$10 through \$14 are not used as inputs,
so there are no dependencies or data hazards

Five instructions are required to fill the pipeline

The first instruction completes after 5 cycles

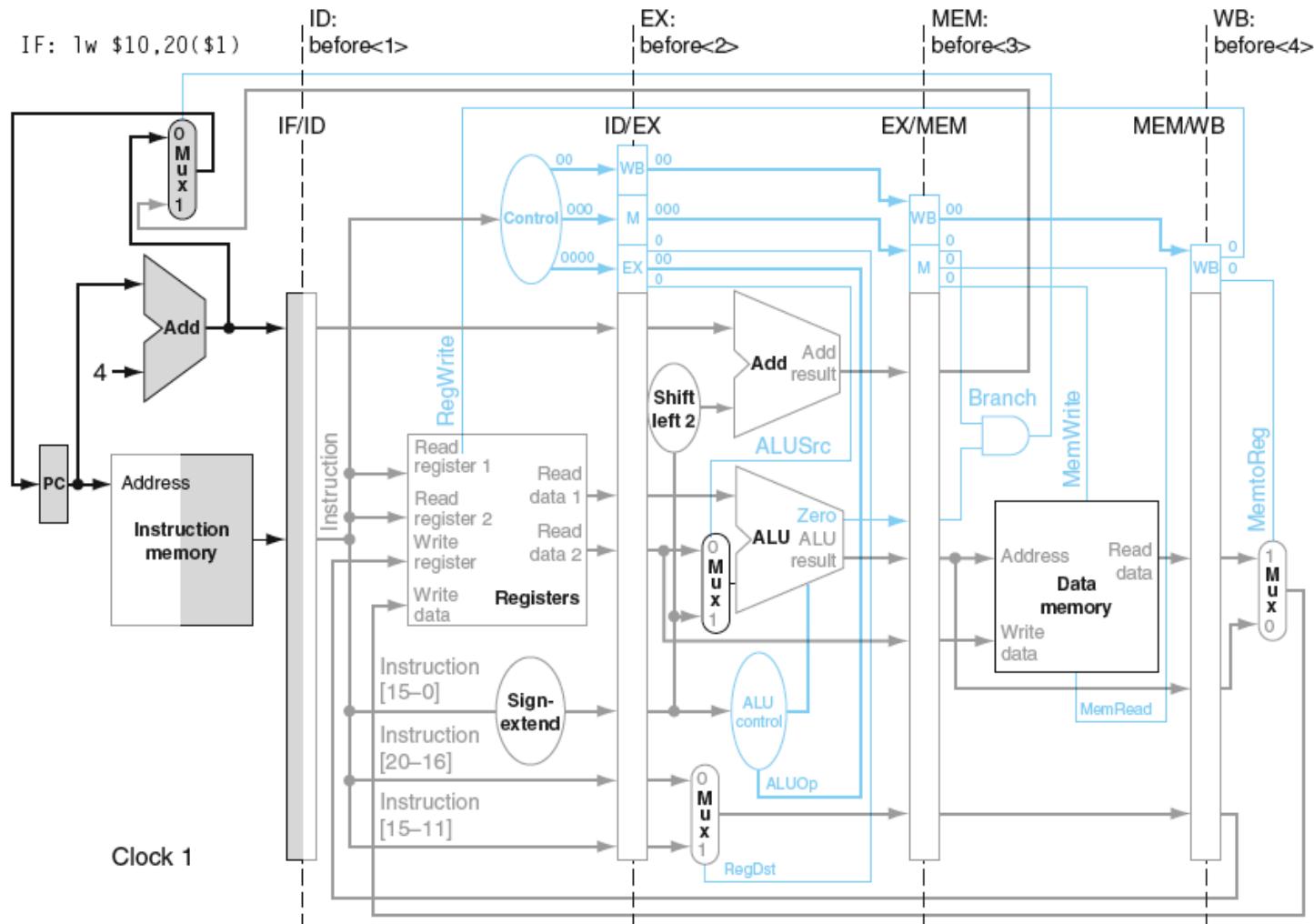
- Hence the pipeline *latency* is 5 cycles
- This is also called the *fill time*

One instruction completes for each subsequent cycle

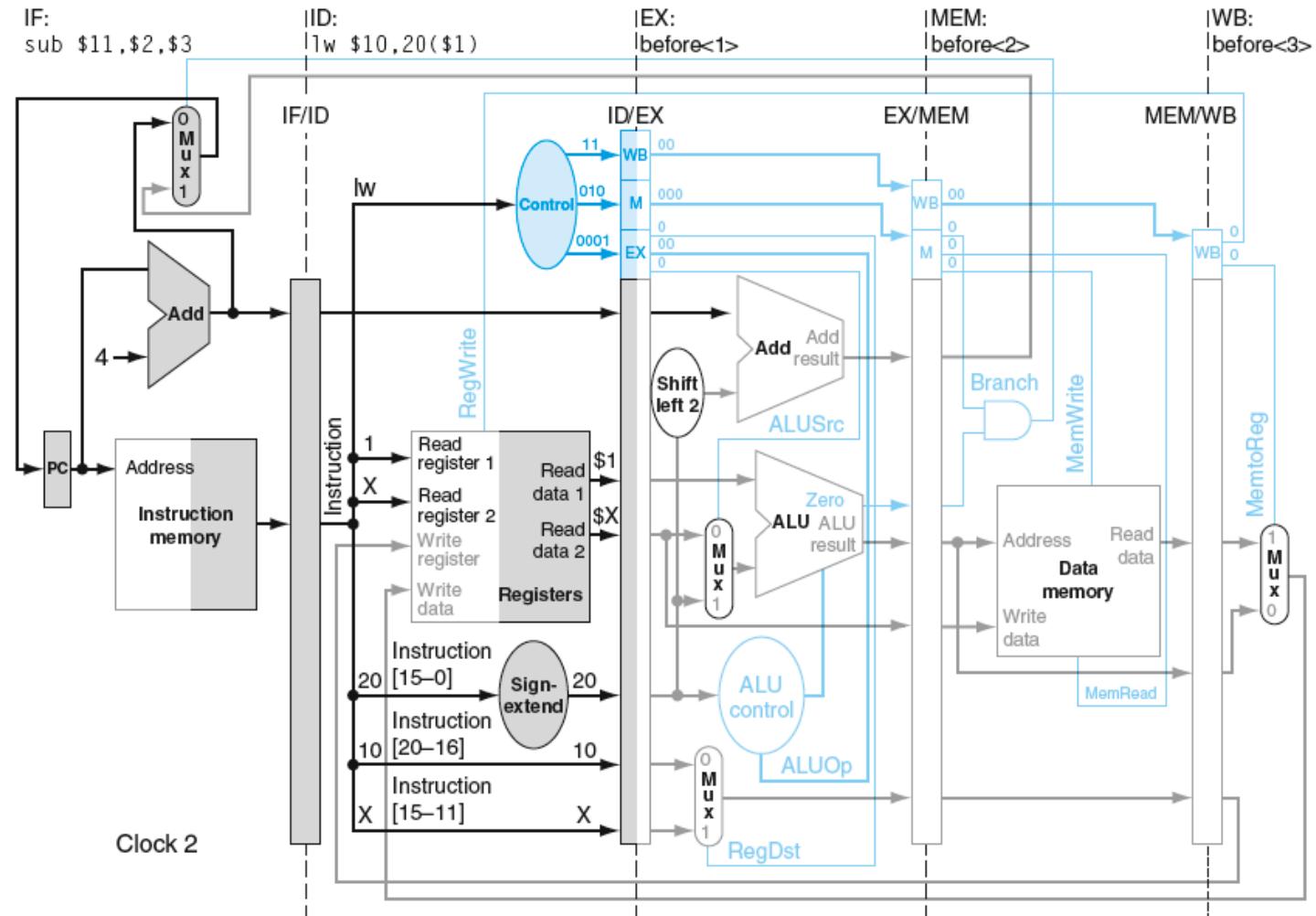
- pipeline stalls would be required if there were hazards

“before $<i>$ ” refers to the i^{th} instruction before those in the example 5-instruction sequence

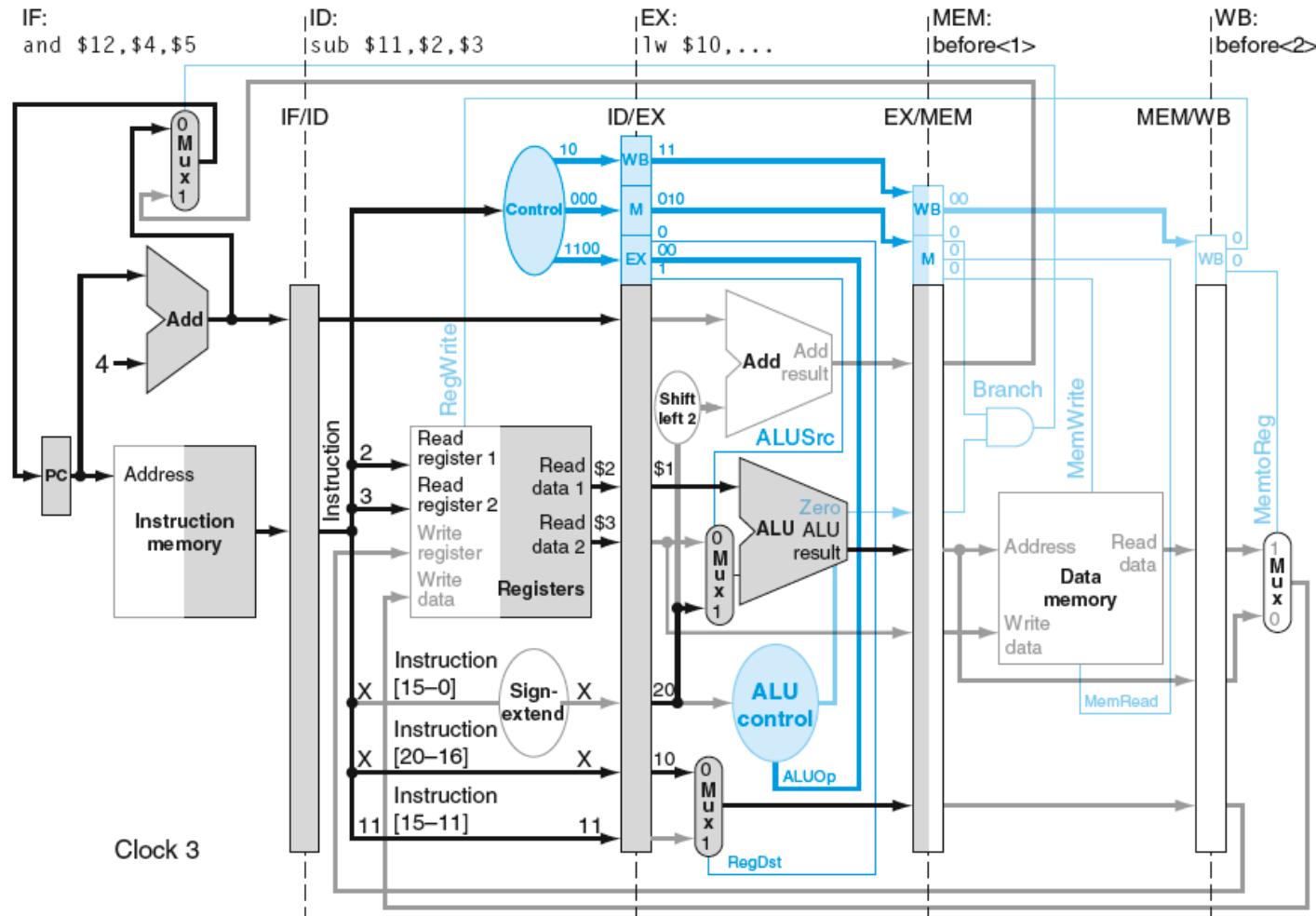
“after $<i>$ ” will refer to the i^{th} instruction following those in the example sequence



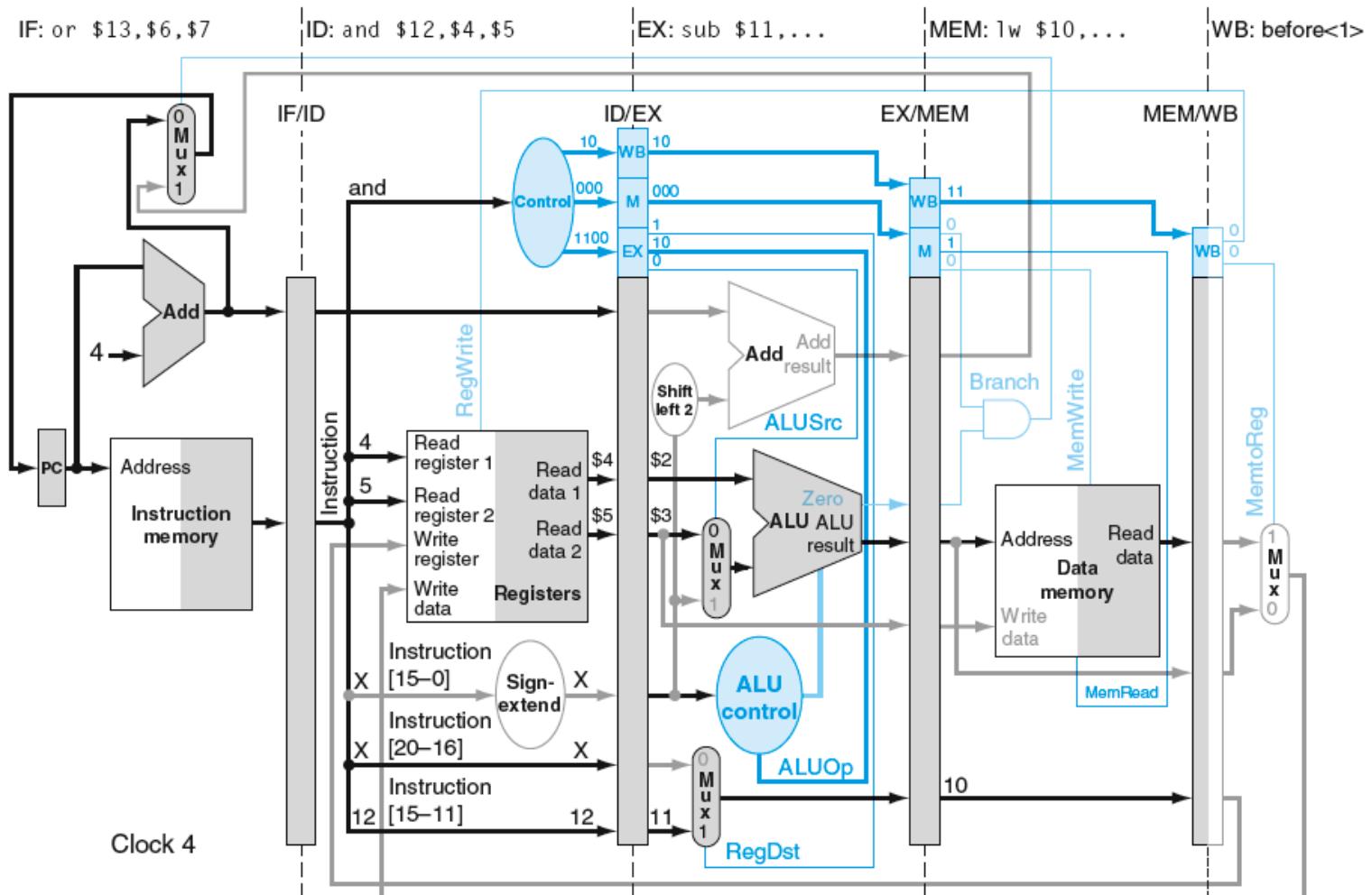
LW is fetched in cycle 1



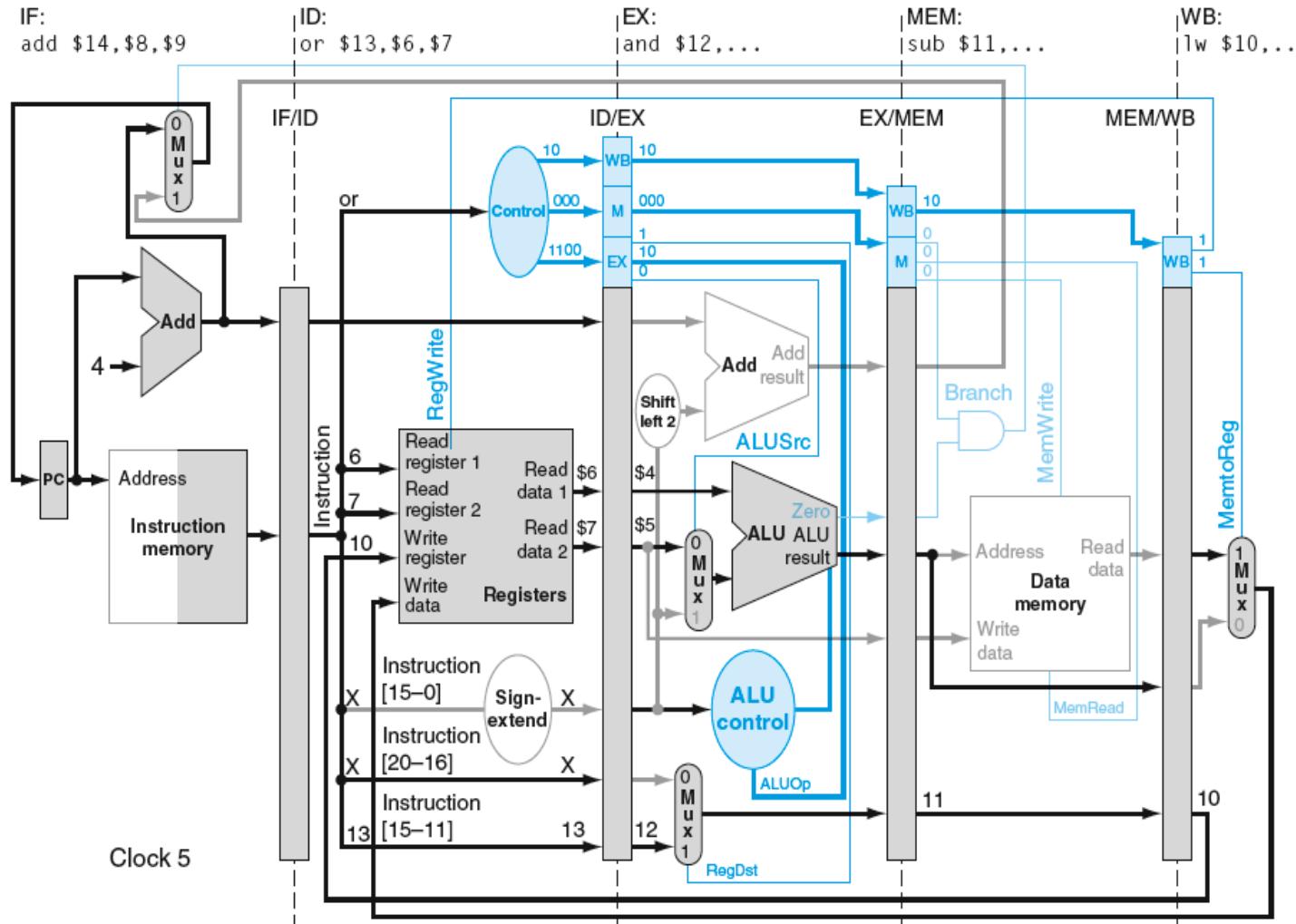
In cycle 2, the control signals for LW are generated and SUB is fetched



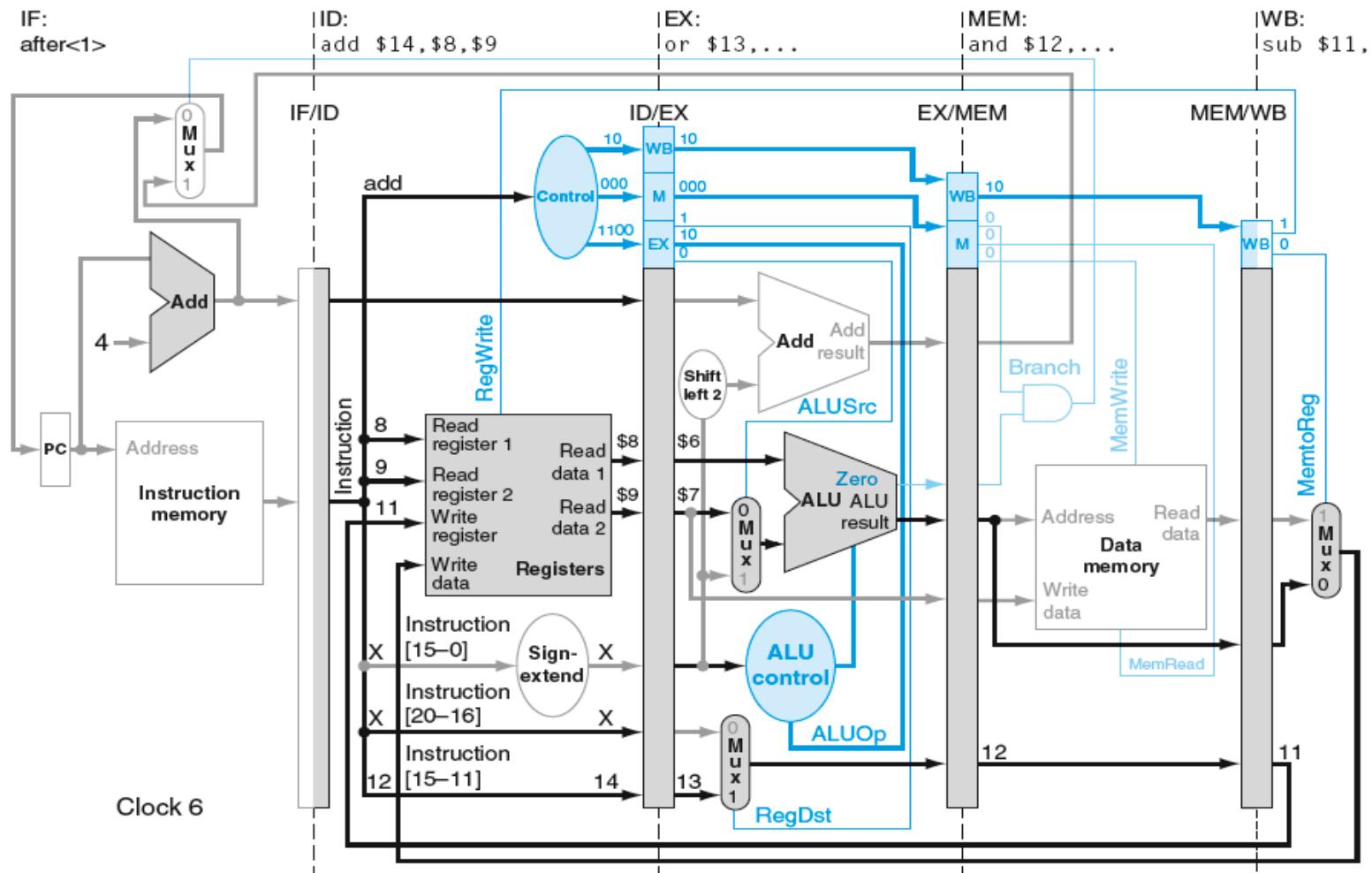
Each instruction reads its input registers in the decode stage
In cycle 3, LW executes, SUB is decoded while AND is fetched



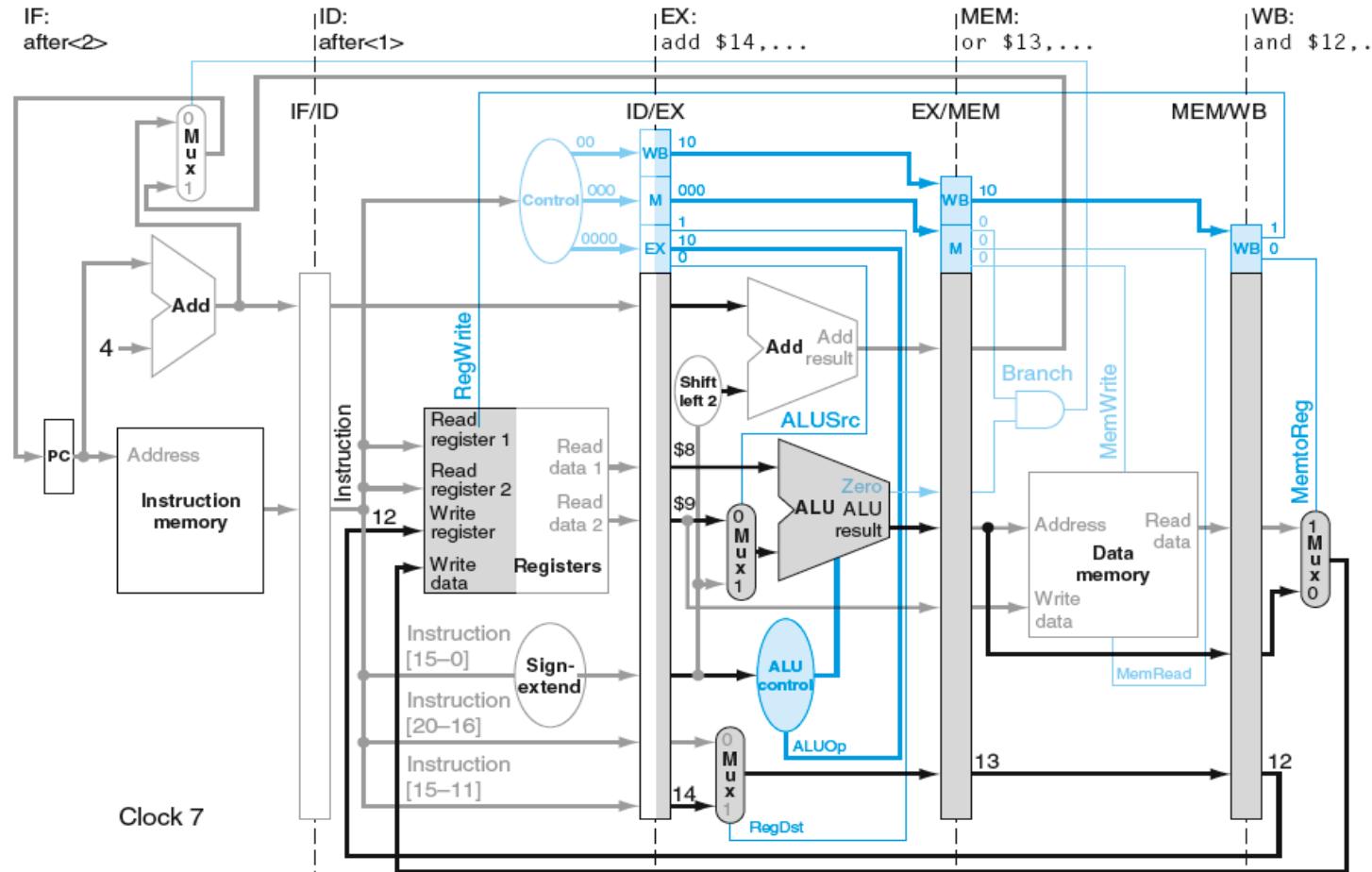
LW reads from the data memory using the address it computed in the execute stage
SUB computes \$2 - \$3, AND is decoded and reads its input registers (\$4 and \$5)



LW completes after writing \$10, SUB idles, AND executes, OR is decoded and reads \$6 & \$7



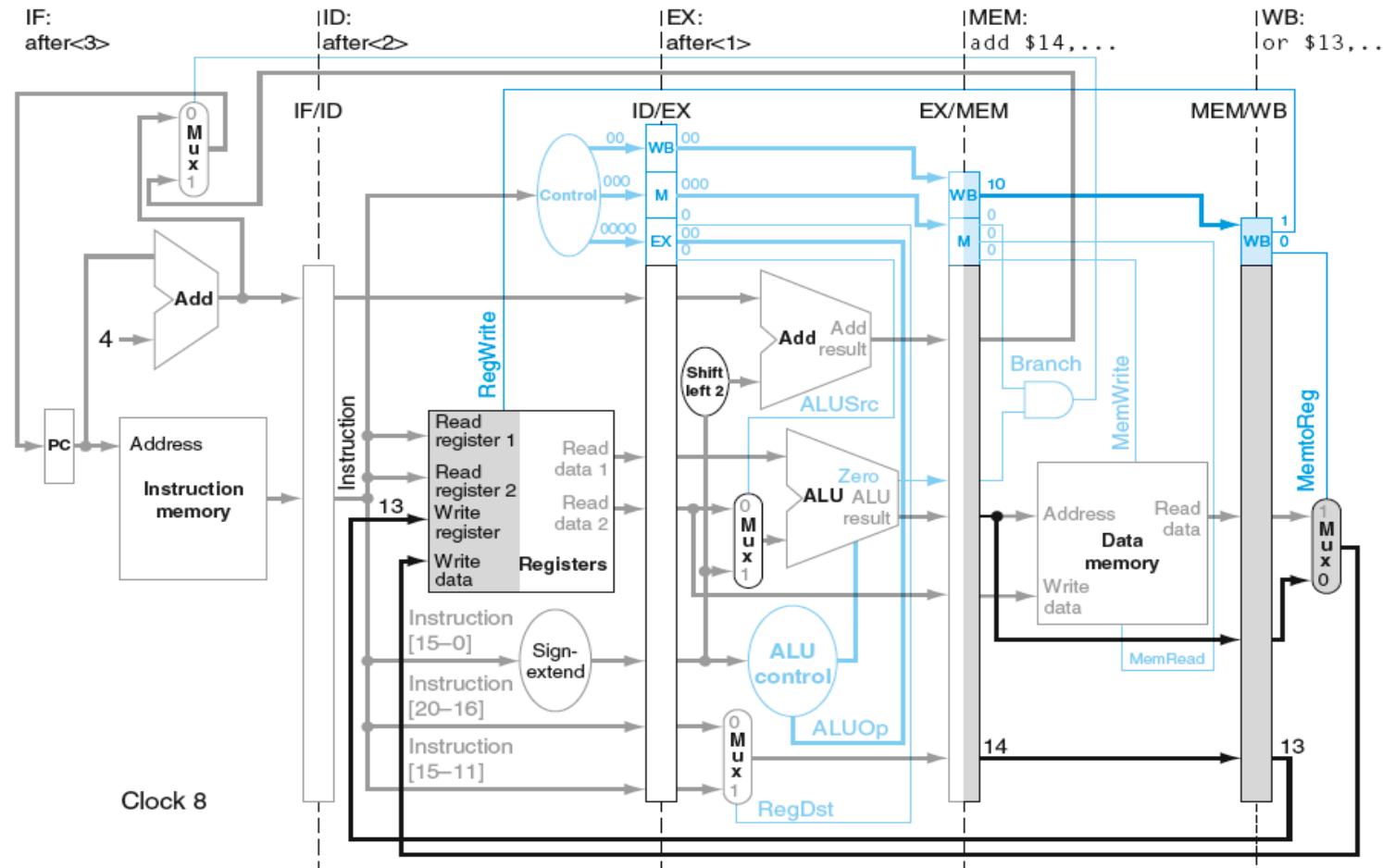
SUB writes \$11 and completes, OR executes, ADD is decoded while \$8 & \$9 are read



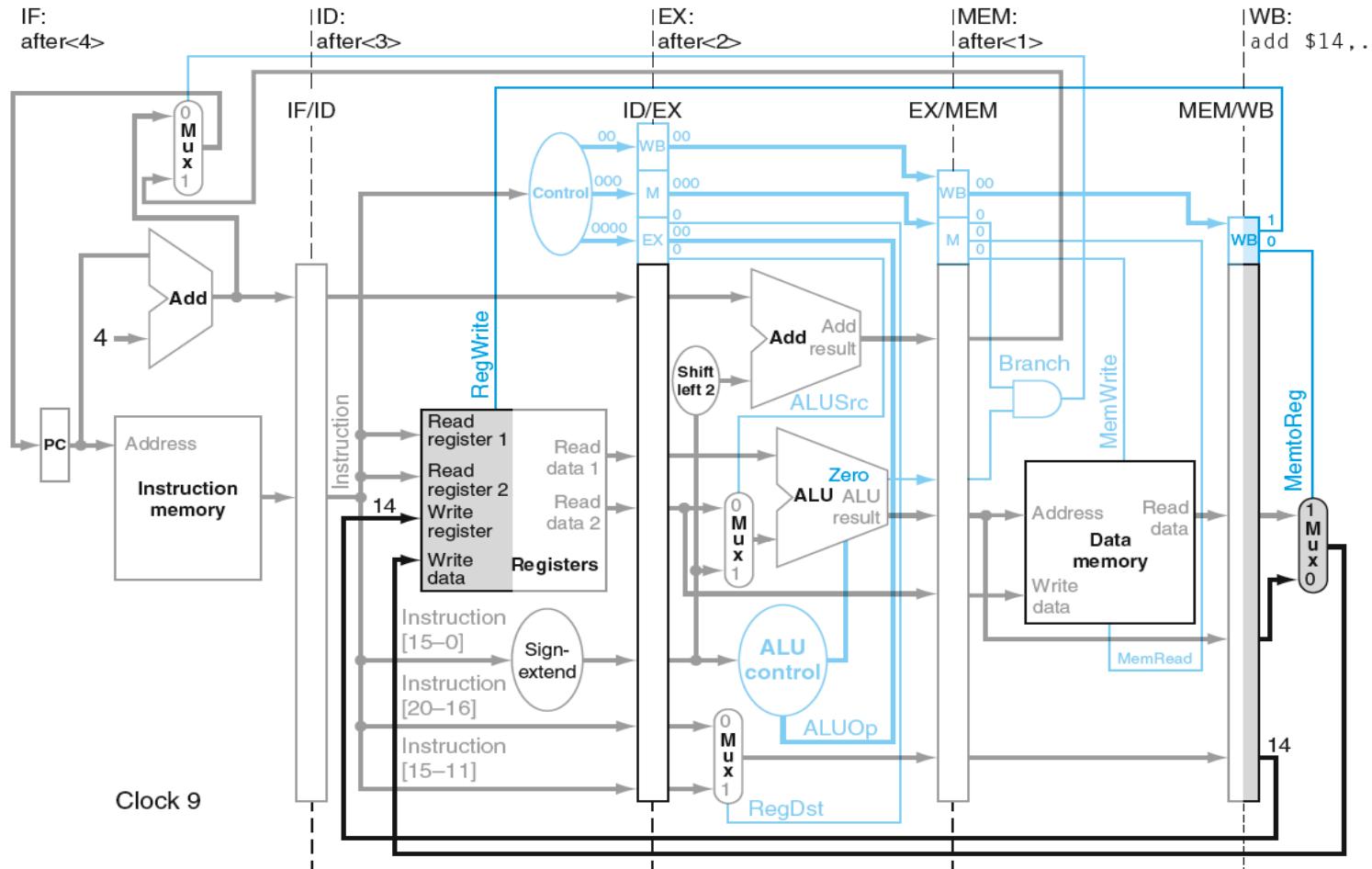
Every instruction must pass through each pipeline stage

Instructions that do not access the data memory idle in stage 4

Hence all instructions take 5 cycles to complete



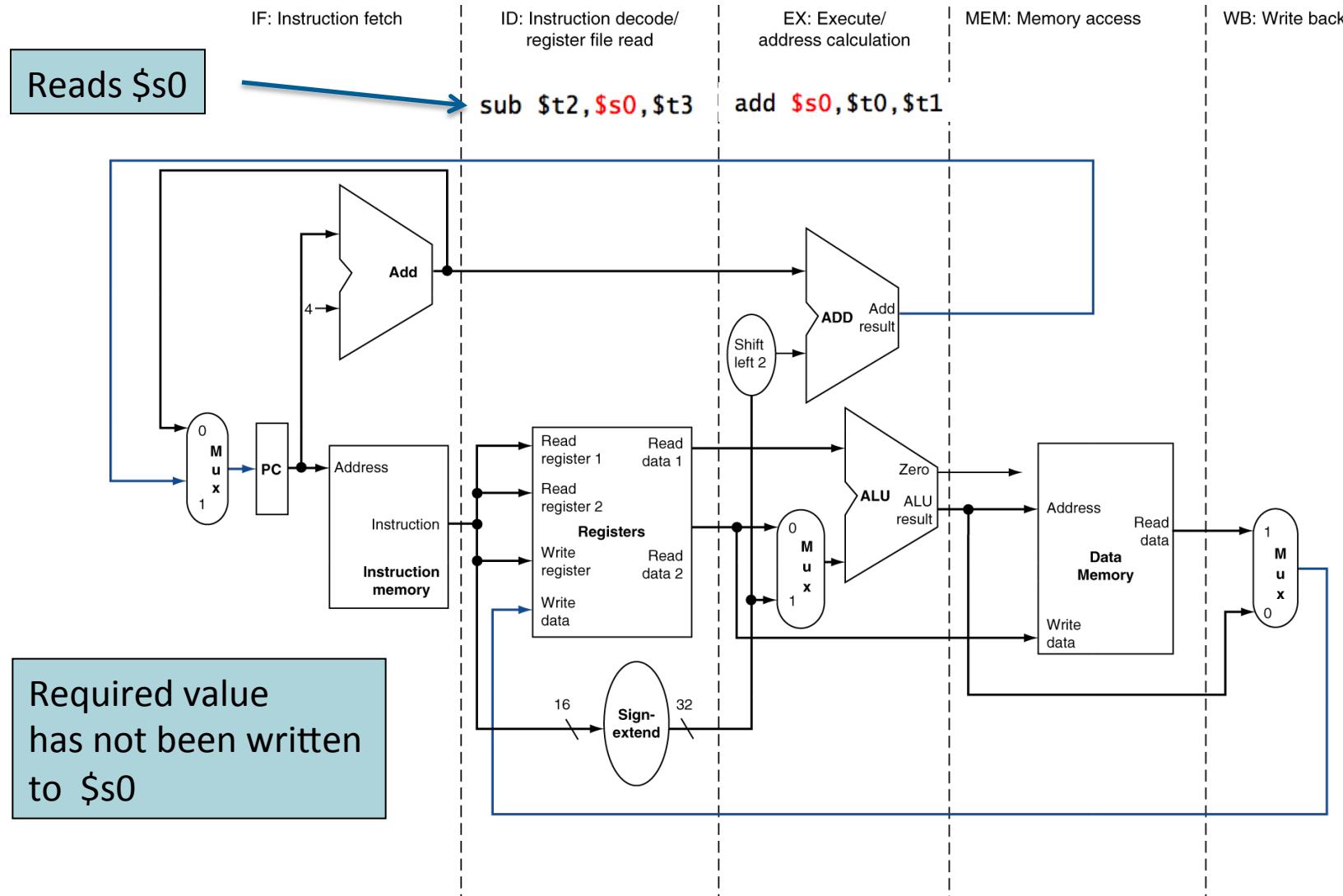
Result registers are written in stage 5 (the write-back stage)
Instructions such as SW and BEQ, do not produce results
In cycle 8, the OR instruction completes

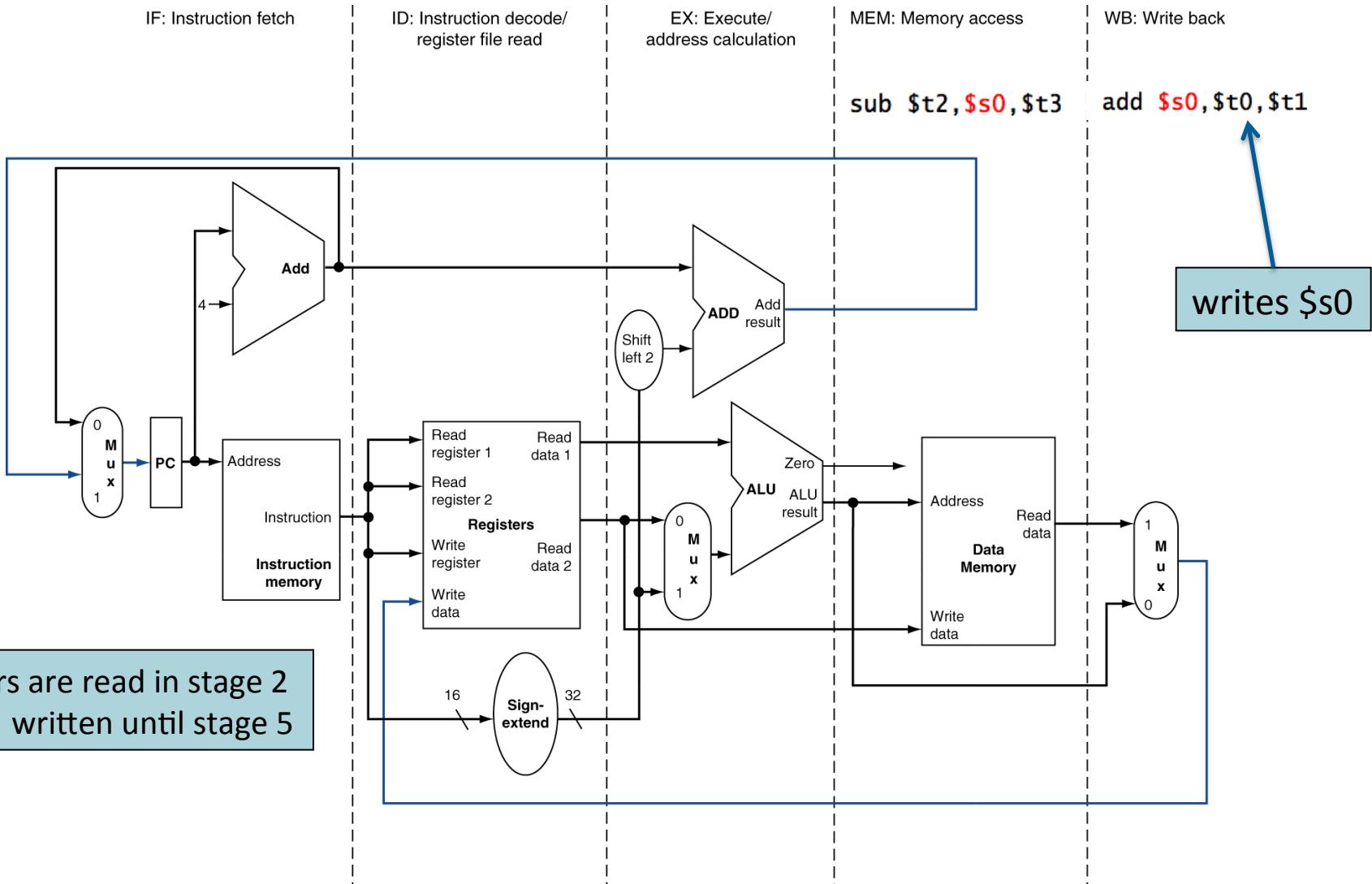


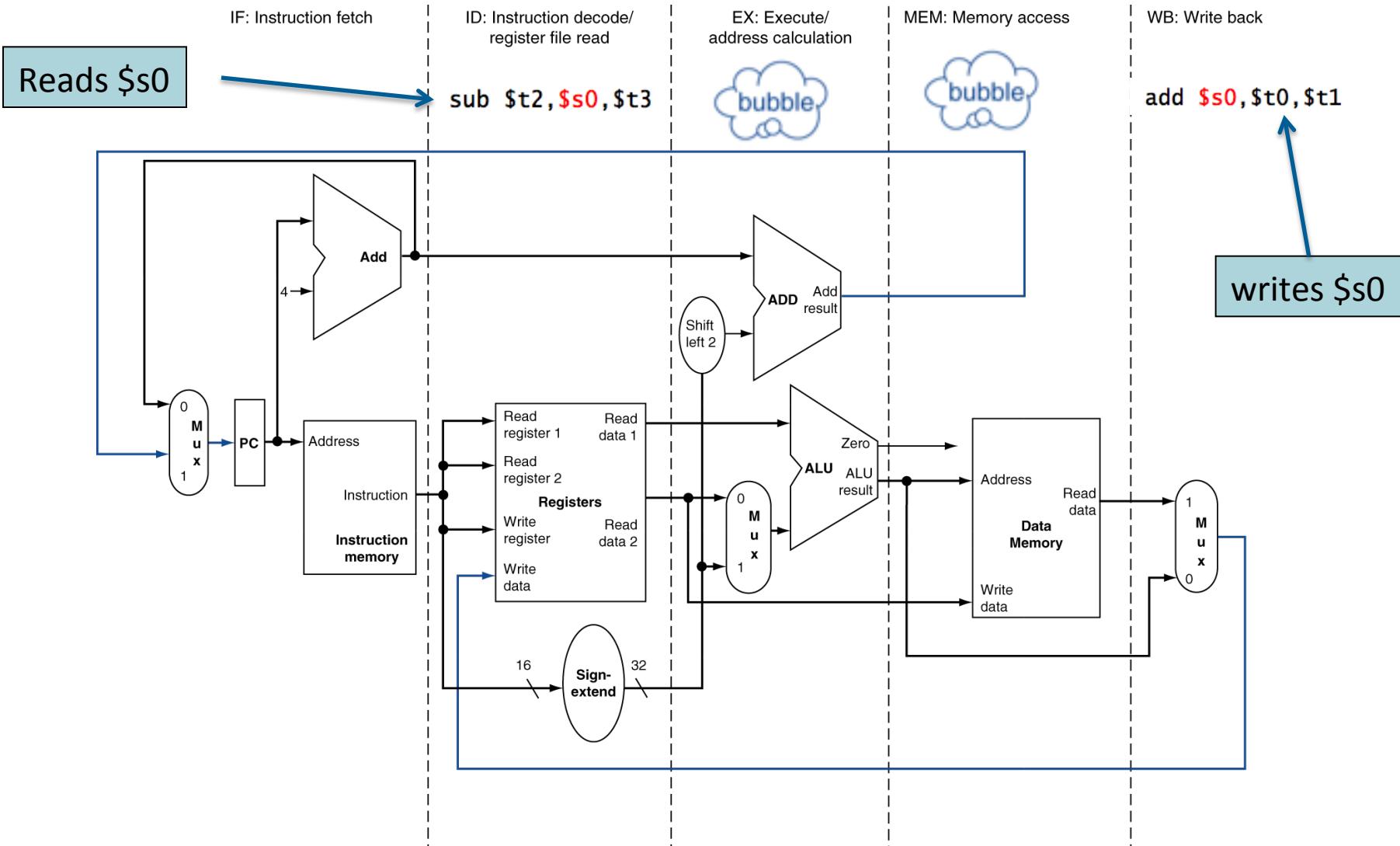
The final instruction in the 5-instruction sequence completes in cycle 9

- The 5-instruction sequence completes in 9 cycles
- On the non-pipeline multi-cycle system
 - LW takes 5 cycles
 - SUB, AND, OR & AND each takes 4 cycles
 - Total = 21 cycles without pipelining
- Pipelining provides a speedup
 - In this case speedup = $21/9 = 2.33$
 - Stalls to cope with data hazards would reduce the speedup

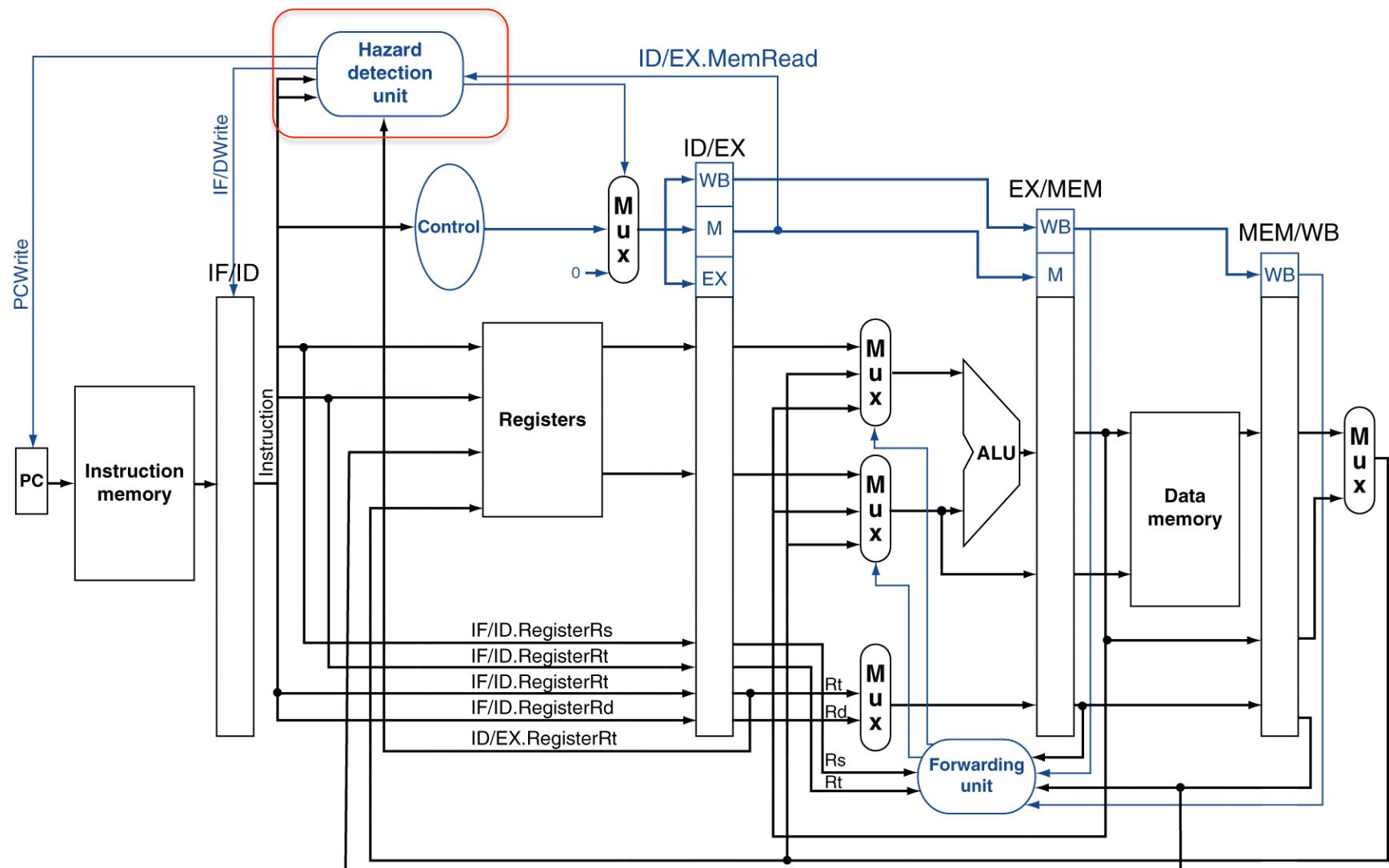
- An instruction depends on a result from a previous instruction
 - add \$s0, \$t0, \$t1
 - sub \$t2, \$s0, \$t3
- 







- Stalling is also known as *pipeline interlock*
 - Instructions in front continue to advance
 - Creates empty stages or *bubbles*
 - Consumes 2 extra clock cycles
- Without stalling, dependent instructions use stale data
 - Extra hardware is required to implement stalls
 - Hazard detection unit controls when stalls occur



Instruction in decode may be dependent

- If either of its 2 input registers matches:
 - reg written by instruction in EXE or in MEM stage
 - Hazard detection unit outputs:
 - 0 to allow normal control signals to be passed
 - 1 to instead substitute zero control signals (i.e. bubble)
- Instructions must write a result to a cause data hazard
 - Neither sw nor beq write a result

If (EX/MEM.RegWrite or MEM/WB.RegWrite)

- Does instruction in stage 4 or 5 write a register?

If (EX/MEM.Rd == ID/EX.Rs or EX/MEM.Rd == ID/EX.Rt) or

If (MEM/WB.Rd == ID/EX.Rs or MEM/WB.Rd == ID/EX.Rt)

- Result reg. match an input reg. for instruction in stage 2?

Hazard exits if both conditions are true

```
add $s0,$t0,$t1
sub $t2,$s0,$t3
slt $t4,$t2,$0
```

- Hazard detection unit compares register numbers
- Sub must be stalled until add writes \$s0
- Slt must be stalled until sub writes \$t2
- Registers are written during the first half of a cycle
- Registers are read during the second half of a cycle
 - otherwise 3 rather than 2 bubbles would be required

Cycle	IF	ID	EX	MEM	WB
1	add \$s0,\$t0,\$t1				
2	sub \$t2,\$s0,\$t3	add \$s0,\$t0,\$t1			
3	slt \$t4,\$t2,\$0	sub \$t2,\$s0,\$t3	add \$s0,\$t0,\$t1		
4	slt \$t4,\$t2,\$0	sub \$t2,\$s0,\$t3	bubble	add \$s0,\$t0,\$t1	
5	slt \$t4,\$t2,\$0	sub \$t2,\$s0,\$t3	bubble	bubble	add \$s0,\$t0,\$t1
6		slt \$t4,\$t2,\$0	sub \$t2,\$s0,\$t3	bubble	bubble
7		slt \$t4,\$t2,\$0	bubble	sub \$t2,\$s0,\$t3	bubble
8		slt \$t4,\$t2,\$0	bubble	bubble	sub \$t2,\$s0,\$t3
9			slt \$t4,\$t2,\$0	bubble	bubble
10				slt \$t4,\$t2,\$0	bubble
11					slt \$t4,\$t2,\$0

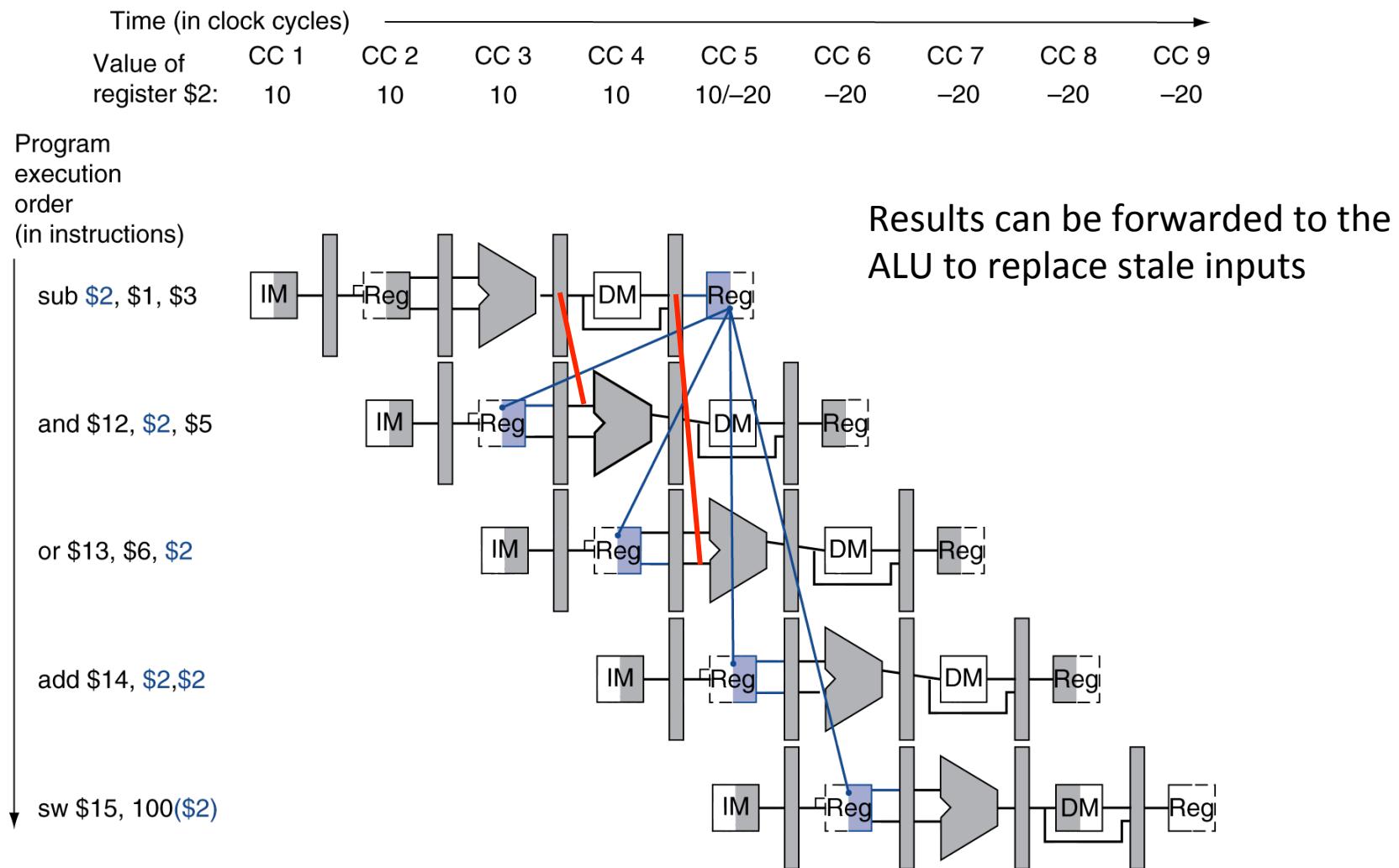
Stalls add 4 extra cycles to the time required for these 3 instructions.

■ Software

- Compiler inserts useful instructions
 - 2 independent instructions between dependent instructions (code rearrangement)
 - 2 NOP instructions if useful ones can't be found

■ Hardware

- Required value is forwarded to dependent instruction
- Just-in-time replacement of stale ALU inputs

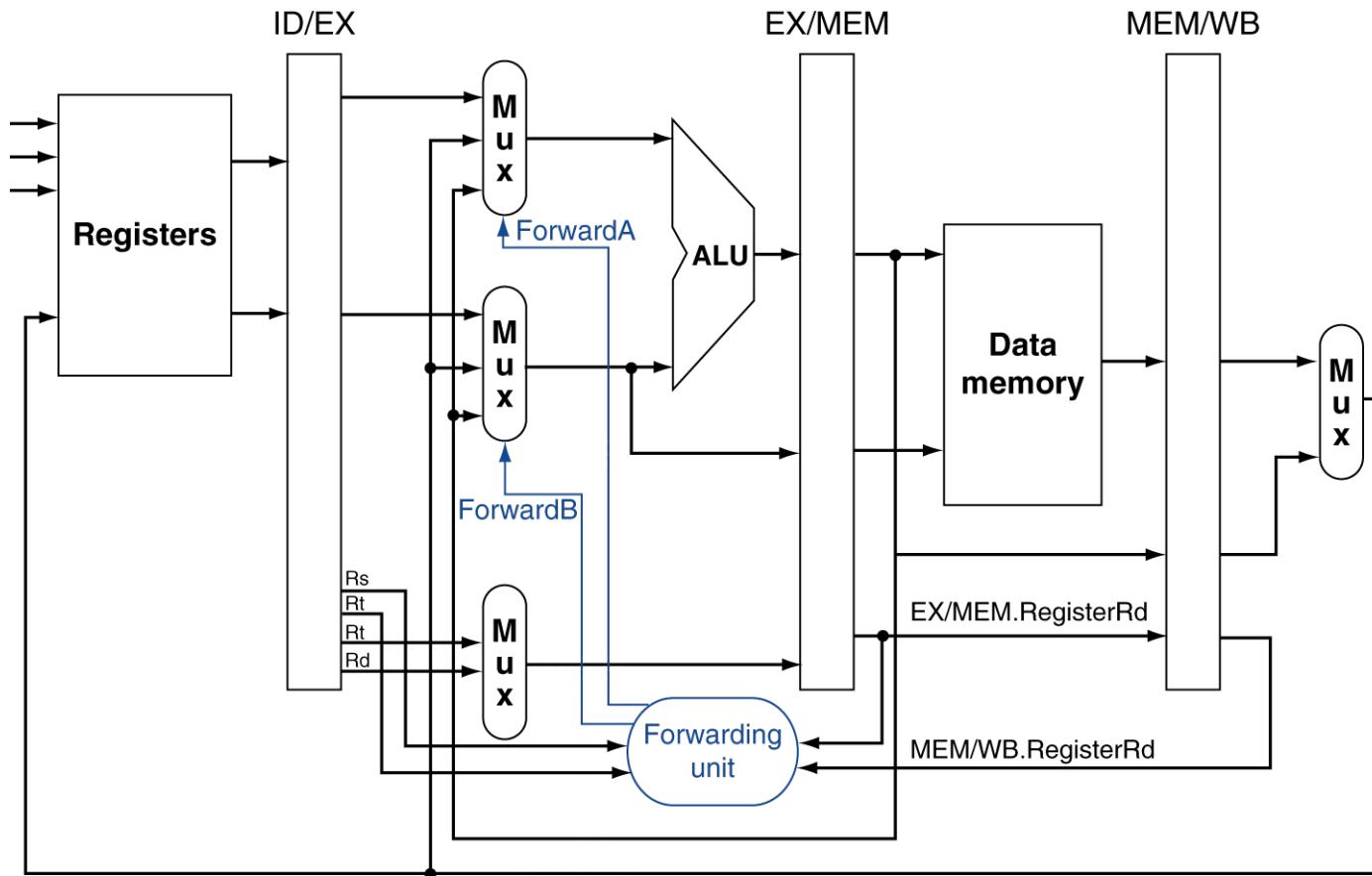


- Pass register numbers along pipeline
 - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
 - ID/EX.RegisterRs, ID/EX.RegisterRt
- Data hazards when
 - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
 - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
 - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
 - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

- Only if forwarding instruction writes a register!
 - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not \$zero
 - EX/MEM.RegisterRd \neq 0,
MEM/WB.RegisterRd \neq 0



Forwarding replaces stale ALU inputs

- EX hazard
 - 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
- MEM hazard
 - 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

Mux control	Source	Explanation
ForwardA = 00	ID/EX	First ALU operand comes from the register file.
ForwardA = 01	MEM/WB	First ALU operand is forwarded from data memory or earlier ALU result.
ForwardA = 10	EX/MEM	First ALU operand comes from the prior ALU result.
ForwardB = 00	ID/EX	Second ALU operand comes from the register file.
ForwardB = 01	MEM/WB	Second ALU operand is forwarded from data memory or earlier ALU result.
ForwardB = 10	EX/MEM	Second ALU operand comes from the prior ALU result.

- Consider the sequence:
 - add \$1, \$1, \$2
 - add \$1, \$1, \$3
 - add \$1, \$1, \$4
- Both hazards occur
 - Want to use the most recent
- Revise MEM hazard condition
 - Only fwd if EX hazard condition isn't true

- MEM hazard
 - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.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 not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01

This sequence will demonstrate how forwarding works

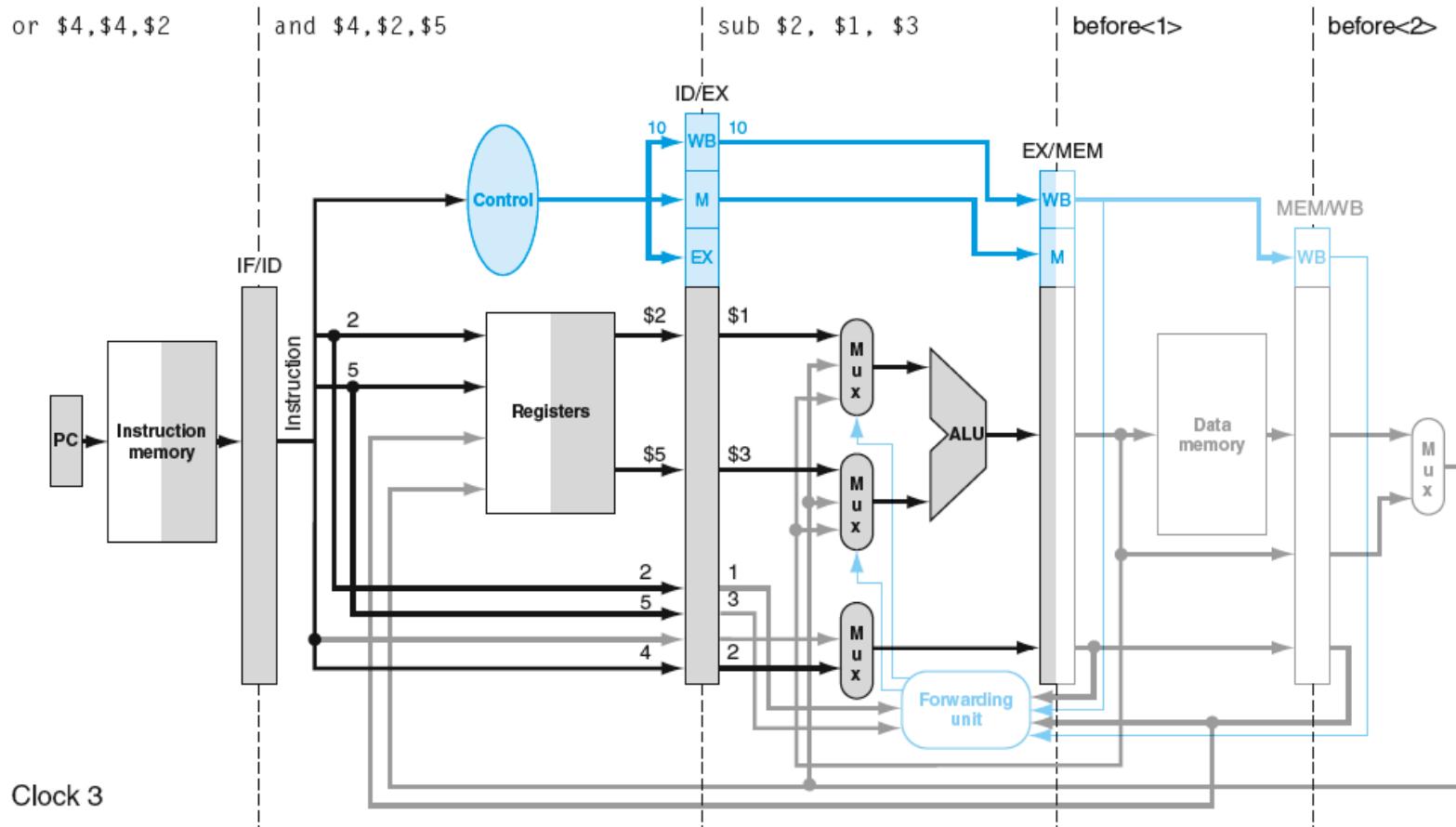
sub	\$2, \$1, \$3
and	\$4, \$2, \$5
or	\$4, \$4, \$2
add	\$9, \$4, \$2

The SUB produces a result in \$2 needed by the AND

The result in \$4 produced by AND is needed by the OR

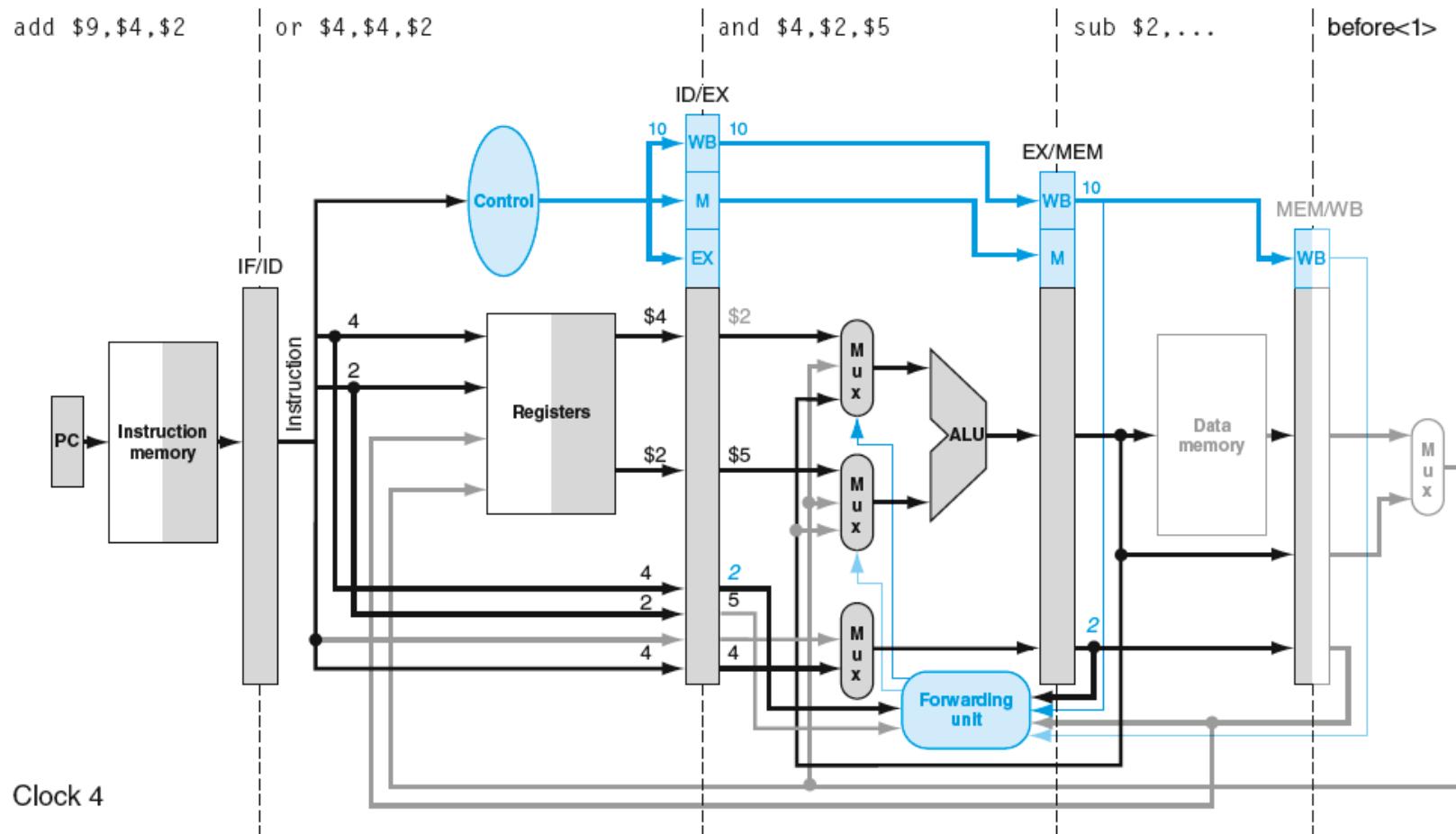
OR updates \$4 which is then used by ADD

By cycle 3, the pipeline contains:

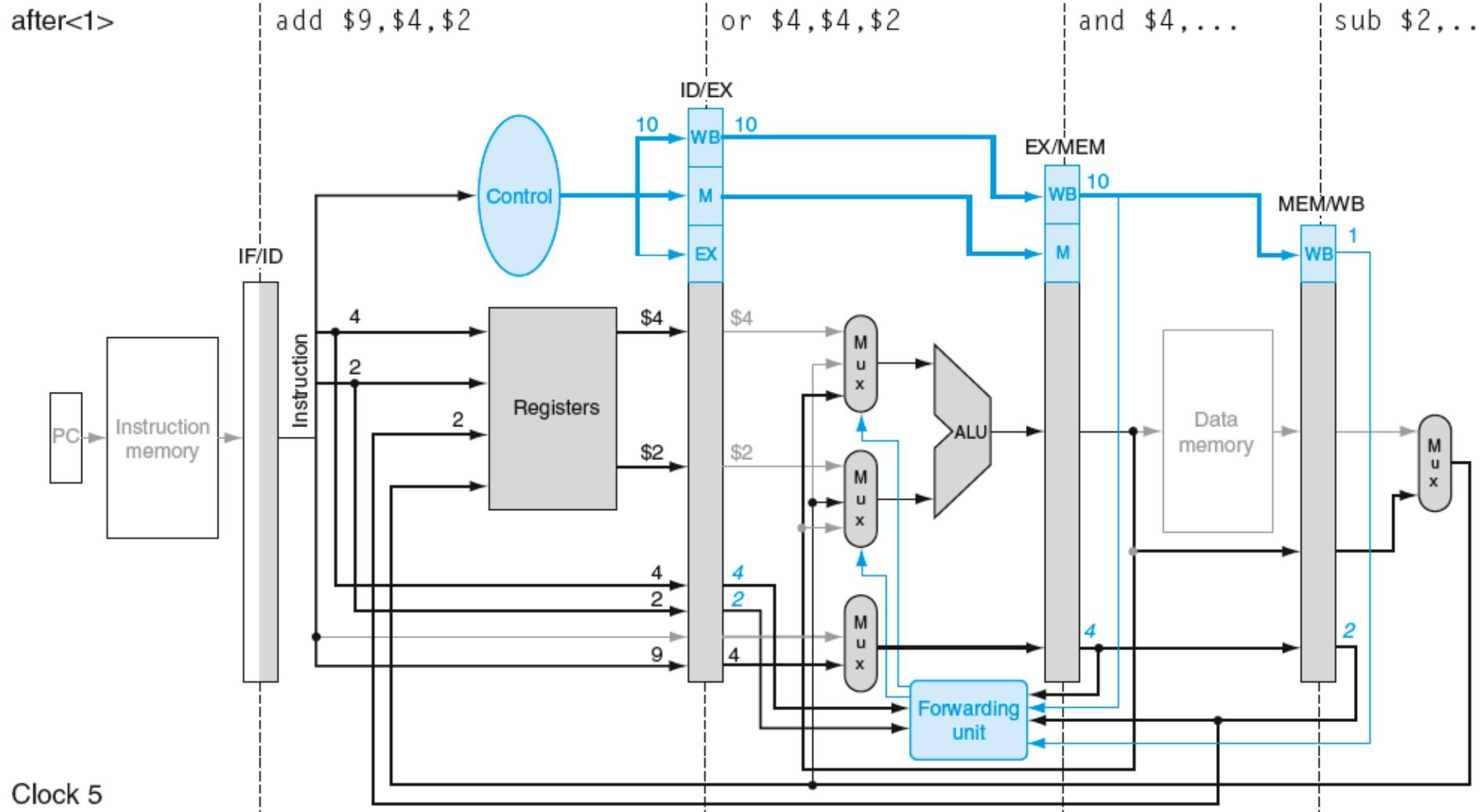


The decode stage is where instructions read their input registers
The write-back stage is where instructions write their result registers

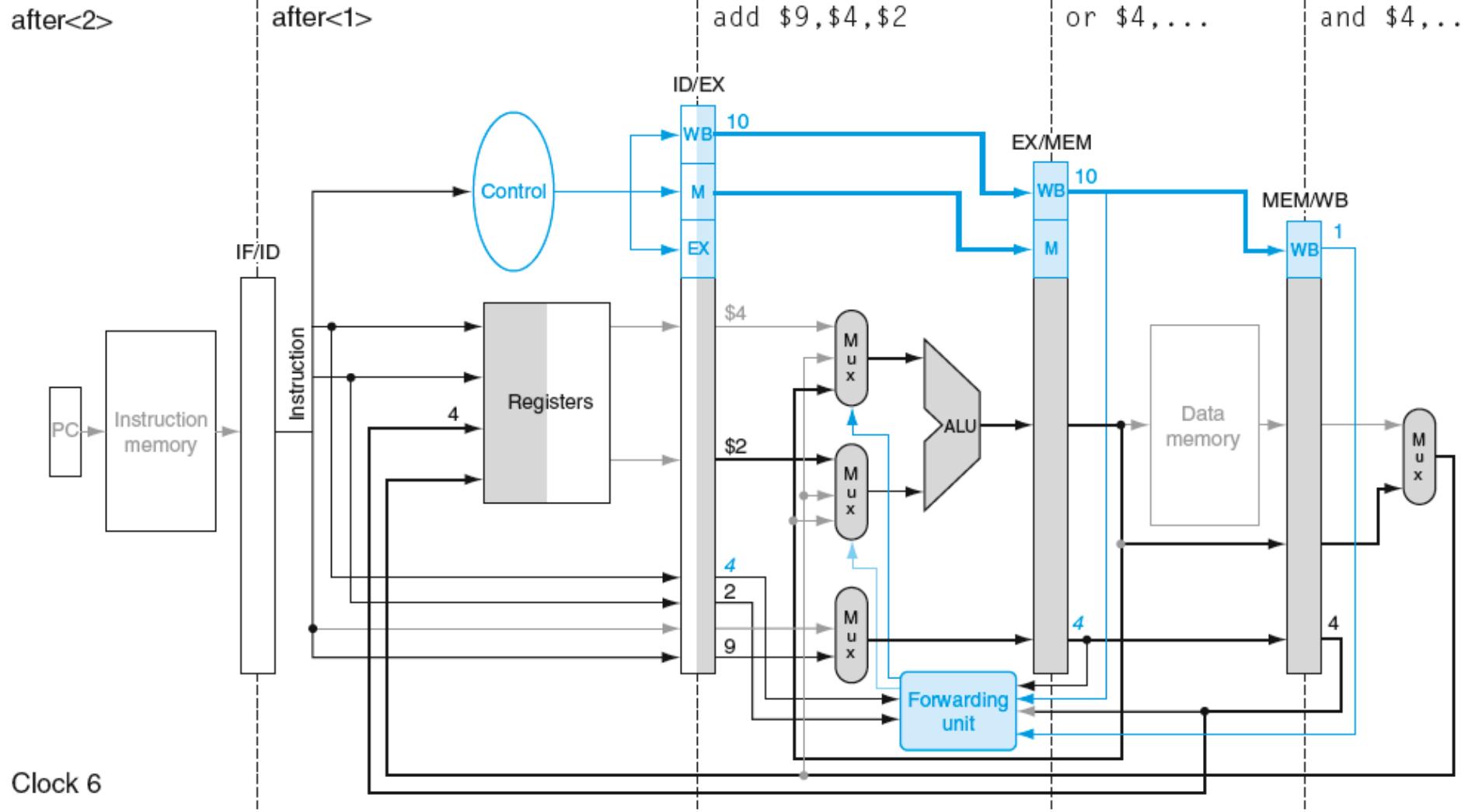
By cycle 4, the pipeline contains:



The upper ALU input for the AND is replaced by the forwarded value from EX/MEM (\$2)



The upper ALU input for the OR is replaced by the forwarded value from EX/MEM



The upper ALU input for the ADD is replaced by the forwarded value from EX/MEM

With forwarding, fewer cycles are consumed

- The above sequence completes in 8 cycles
- It takes 14 cycles without forwarding

Without forwarding, stalls are needed (bubbles)

- Dependent instruction is held in decode stage
- Instruction producing result must reach stage5
- Register writes occur in first half of clock cycle
- Register reads occur in second half of clock cycle
- Otherwise one extra stall would be needed

Forwarding avoids having to stall dependent instructions

- By just-in-time replacement of stale ALU inputs

Some instructions cannot cause data hazards

- These instruction don't write a result register
- Sw writes to memory, not to a register
- Beq compares 2 registers without writing either

Forwarding avoids having to stall dependent instructions

- Except for values read from memory
- *Delay slot* is the slot following a load instruction
- Instructions in the delay slot can't use the load result
- otherwise they must be stalled for one cycle

Hazard detection unit is still required for this special case

This sequence illustrates the effect of a load delay:

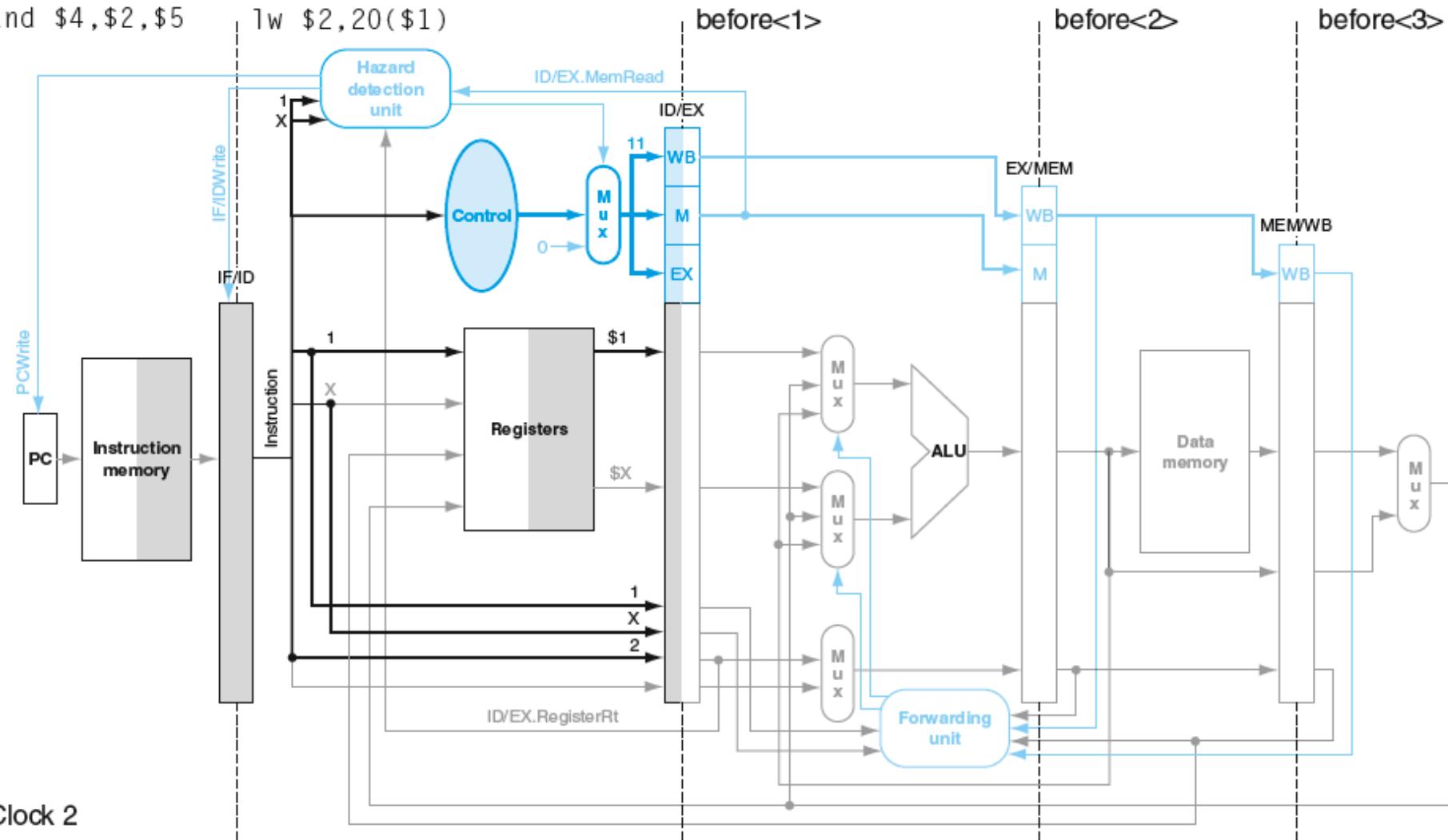
lw	\$2, 20(\$1)
and	\$4, \$2,\$5
or	\$4, \$4,\$2
add	\$9, \$4,\$2

The AND needs the result in \$2 read from memory by LW

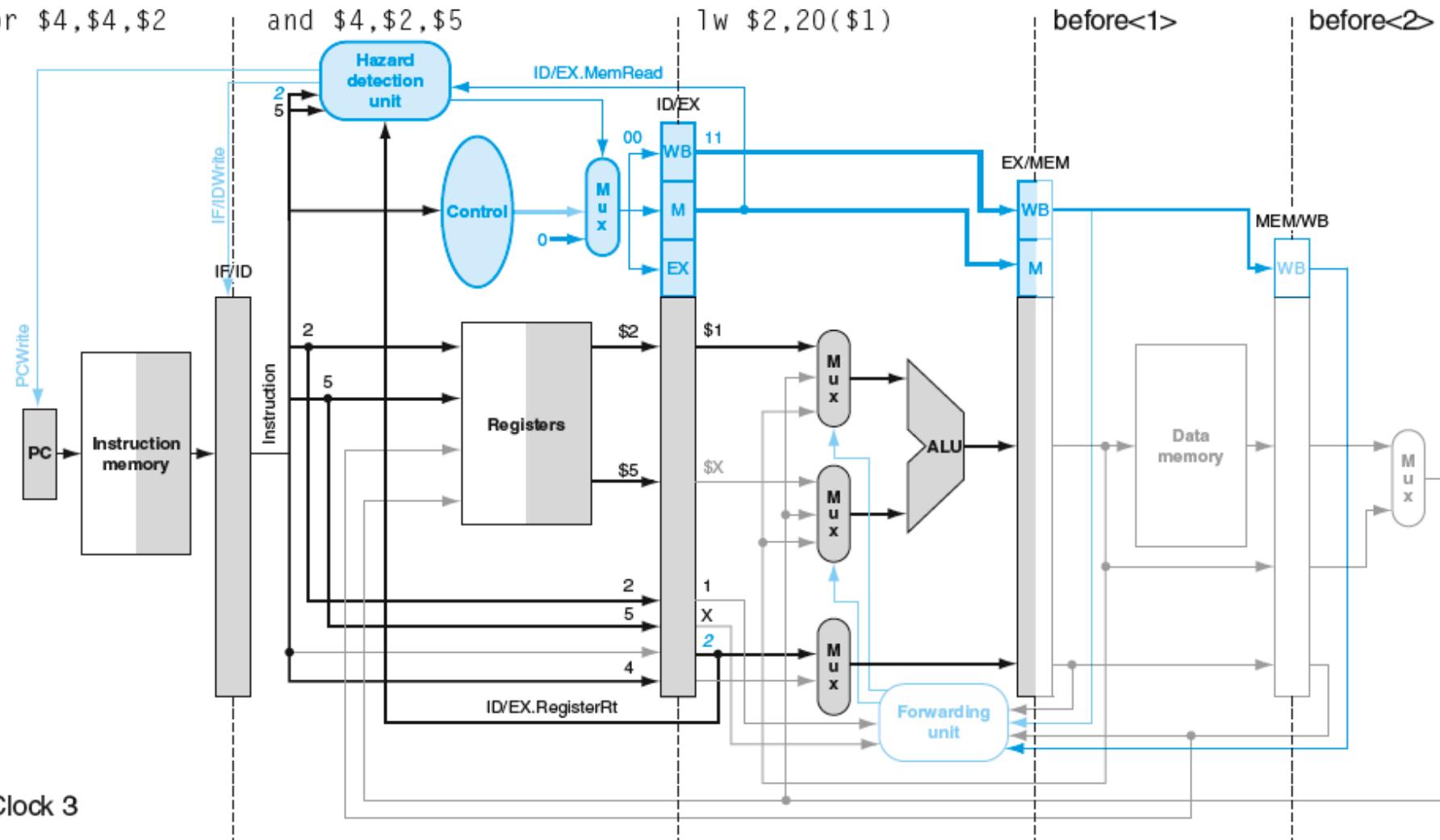
The result in \$4 produced by AND is needed by the OR

OR updates \$4 which is then used by ADD

and \$4,\$2,\$5



or \$4,\$4,\$2



AND uses \$2, so it must be stalled to allow LW time to read from the data memory

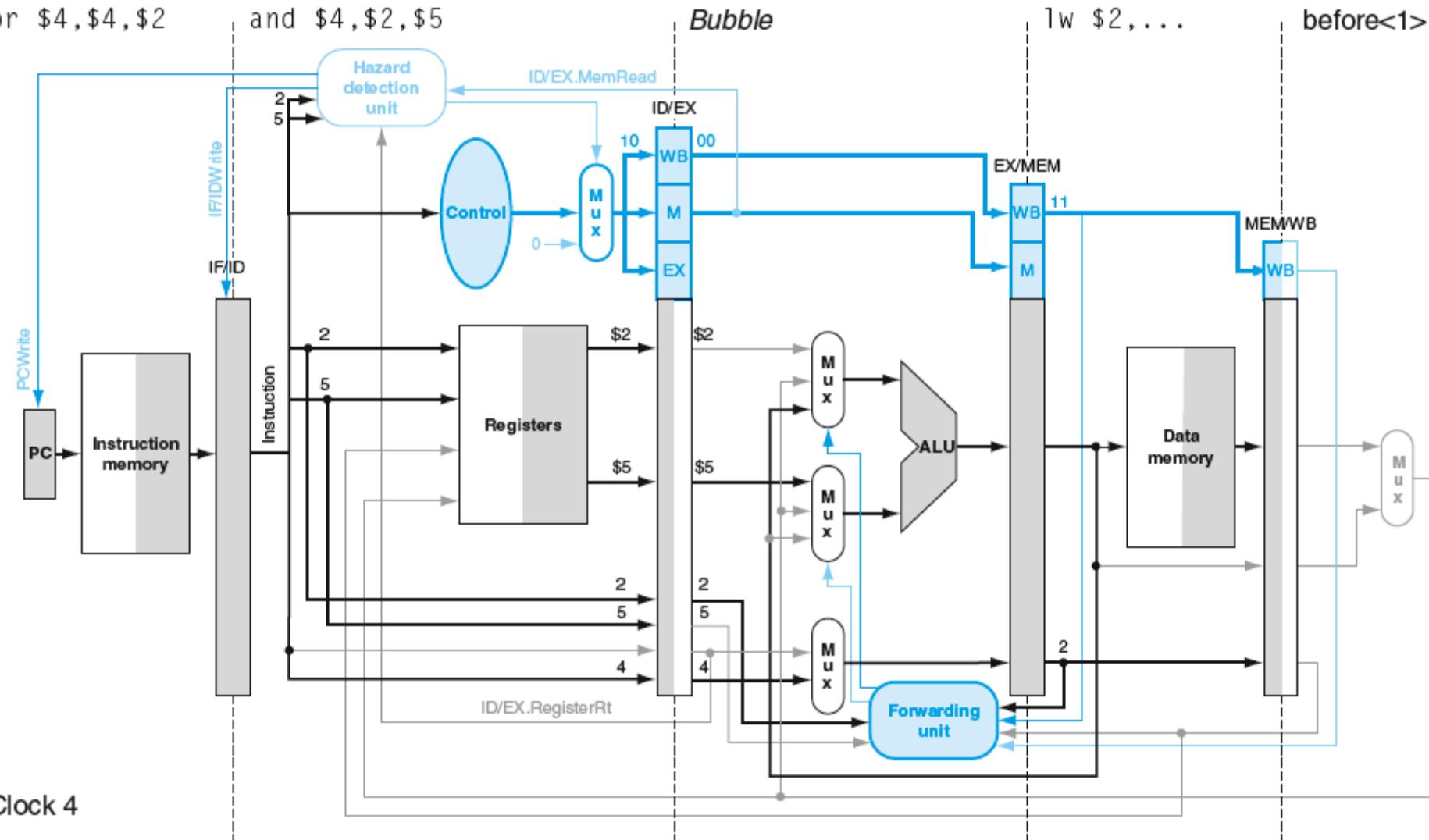
or \$4,\$4,\$2

and \$4,\$2,\$5

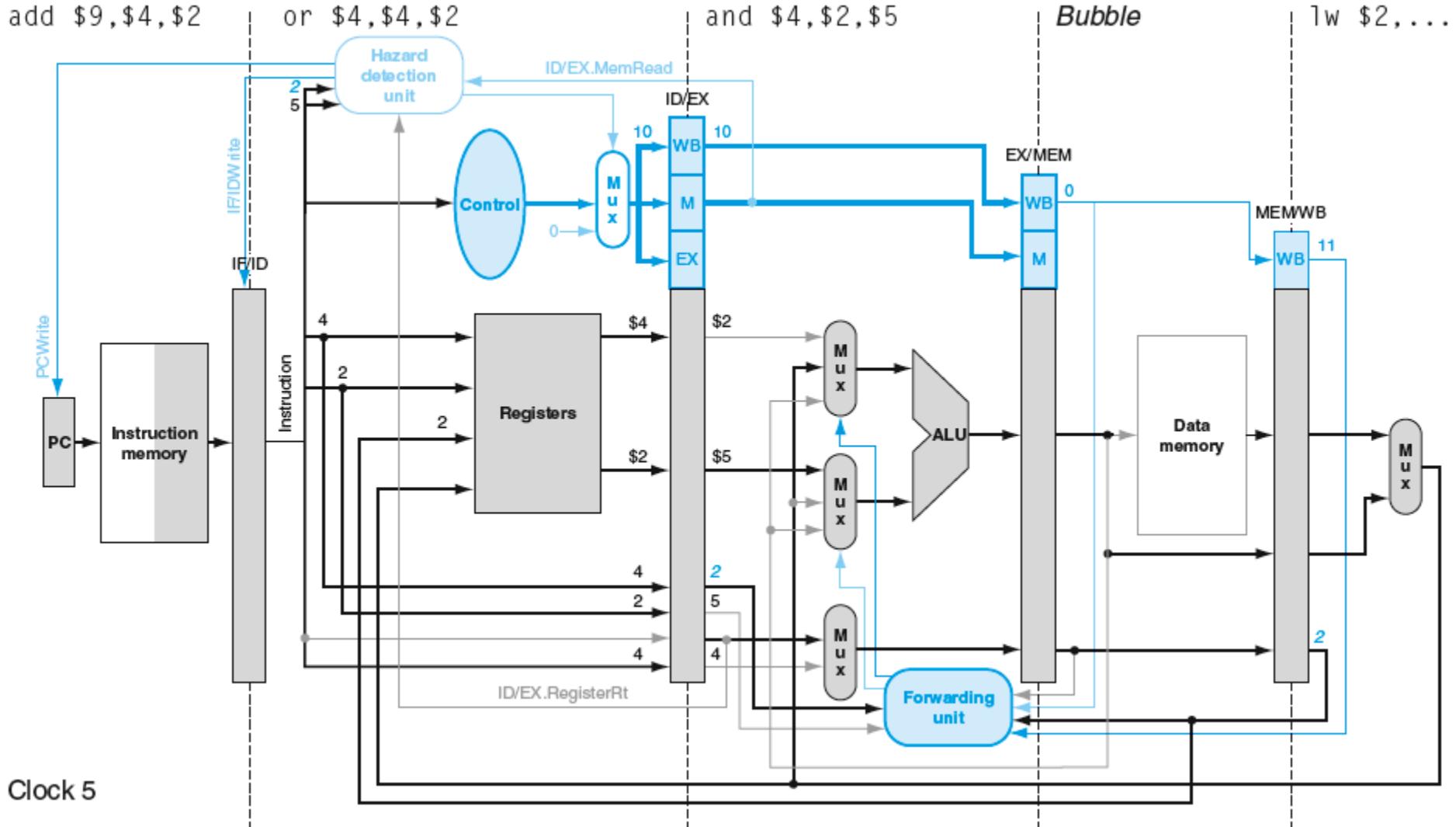
Bubble

lw \$2,...

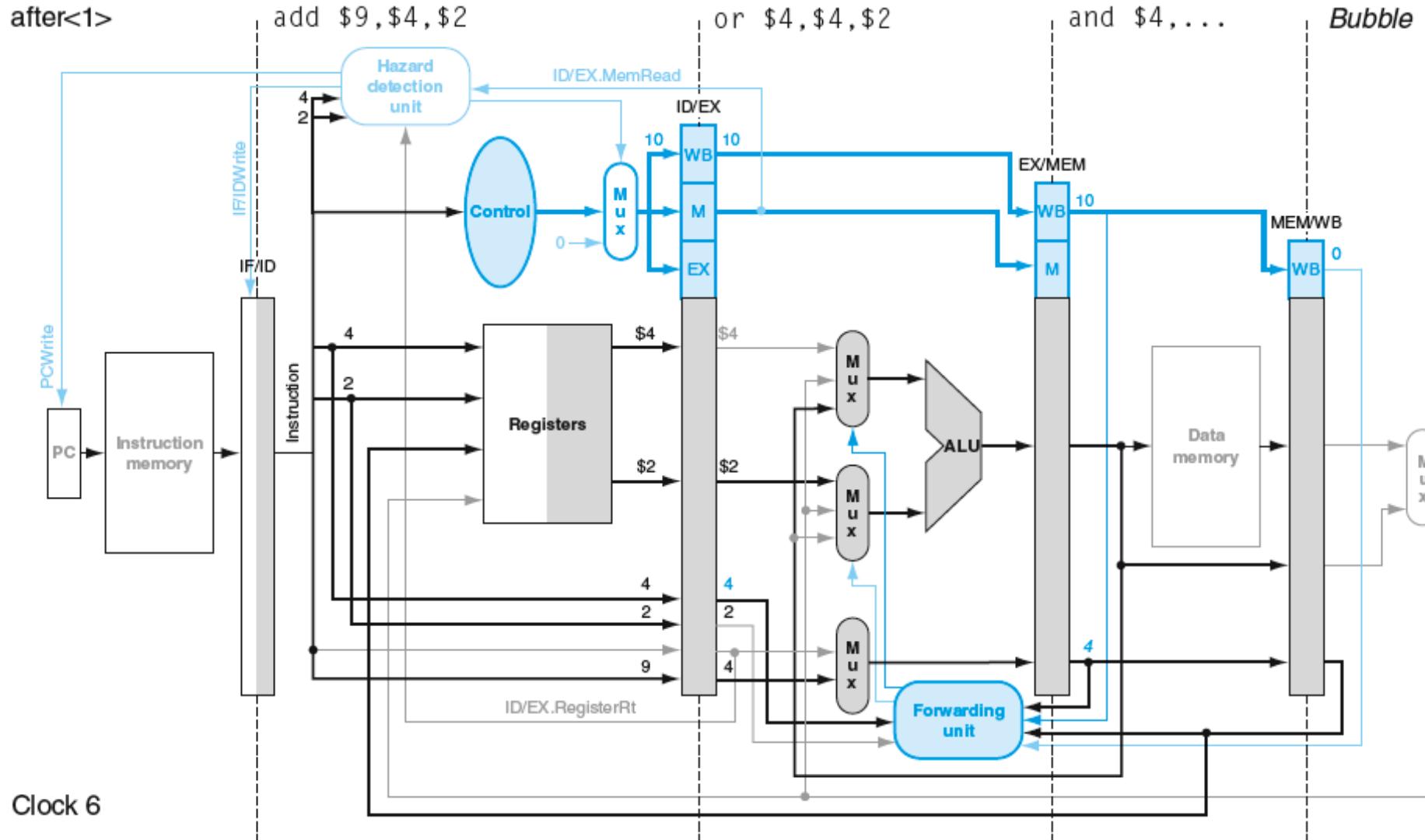
before<1>



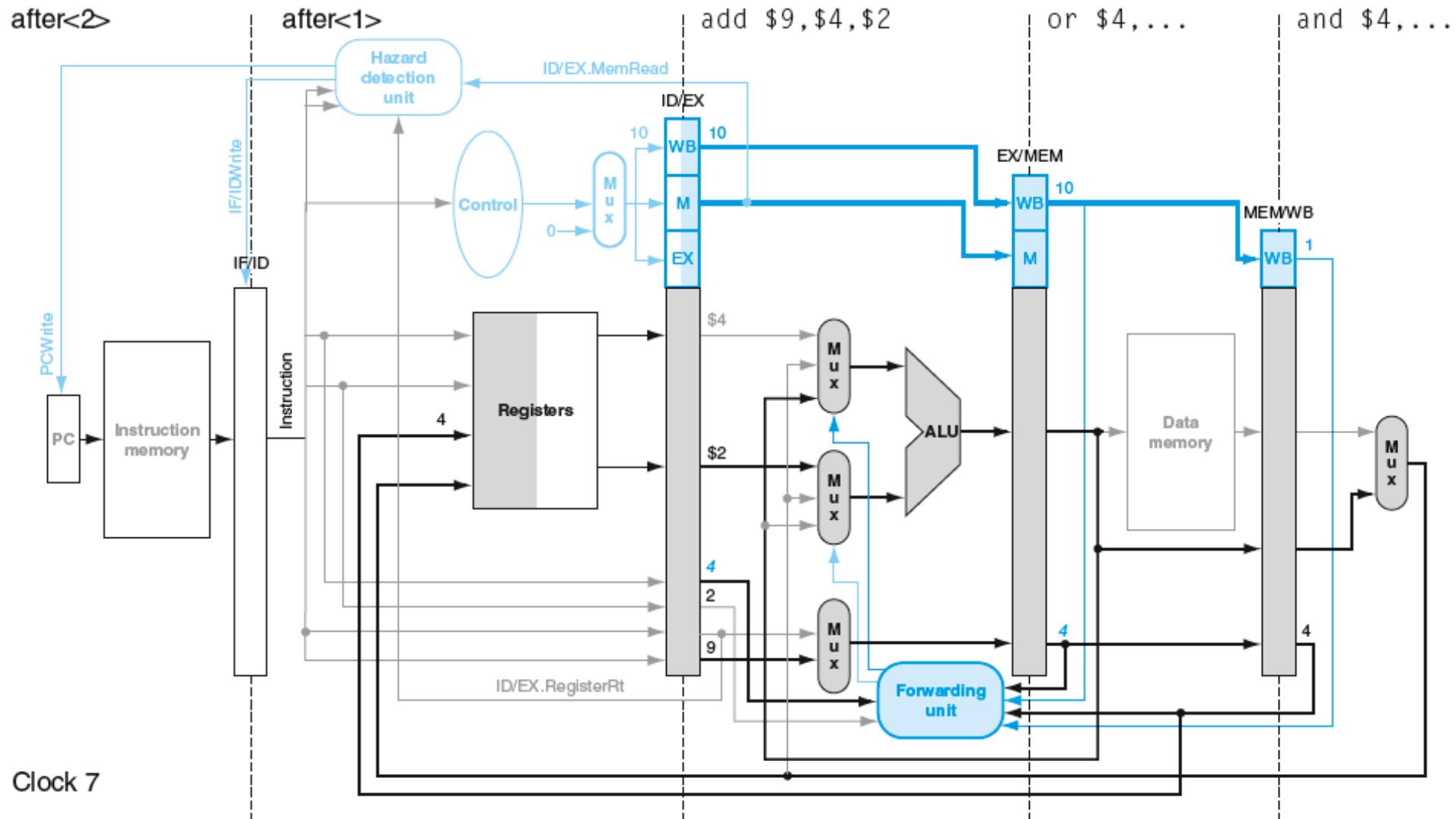
Bubble occupies stage 3 while LW reads from the data memory



AND receives forwarded value from MEM/WB pipeline register in cycle 5



Forwarding takes care of the dependencies for the OR and ADD instructions.



All instructions complete by cycle 9.

The stall due to load delay added one extra cycle

Some compilers rearrange machine instructions

- This can avoid extra cycles due to stalls
- Rearrangement must not change program behavior
- Assembly language programmers can fill delay slot
- NOP instruction can be used as last resort

Branches disrupt the flow of instructions in the pipeline

- The branch target address is computed in stage 3

- The branch condition is evaluated in stage 3

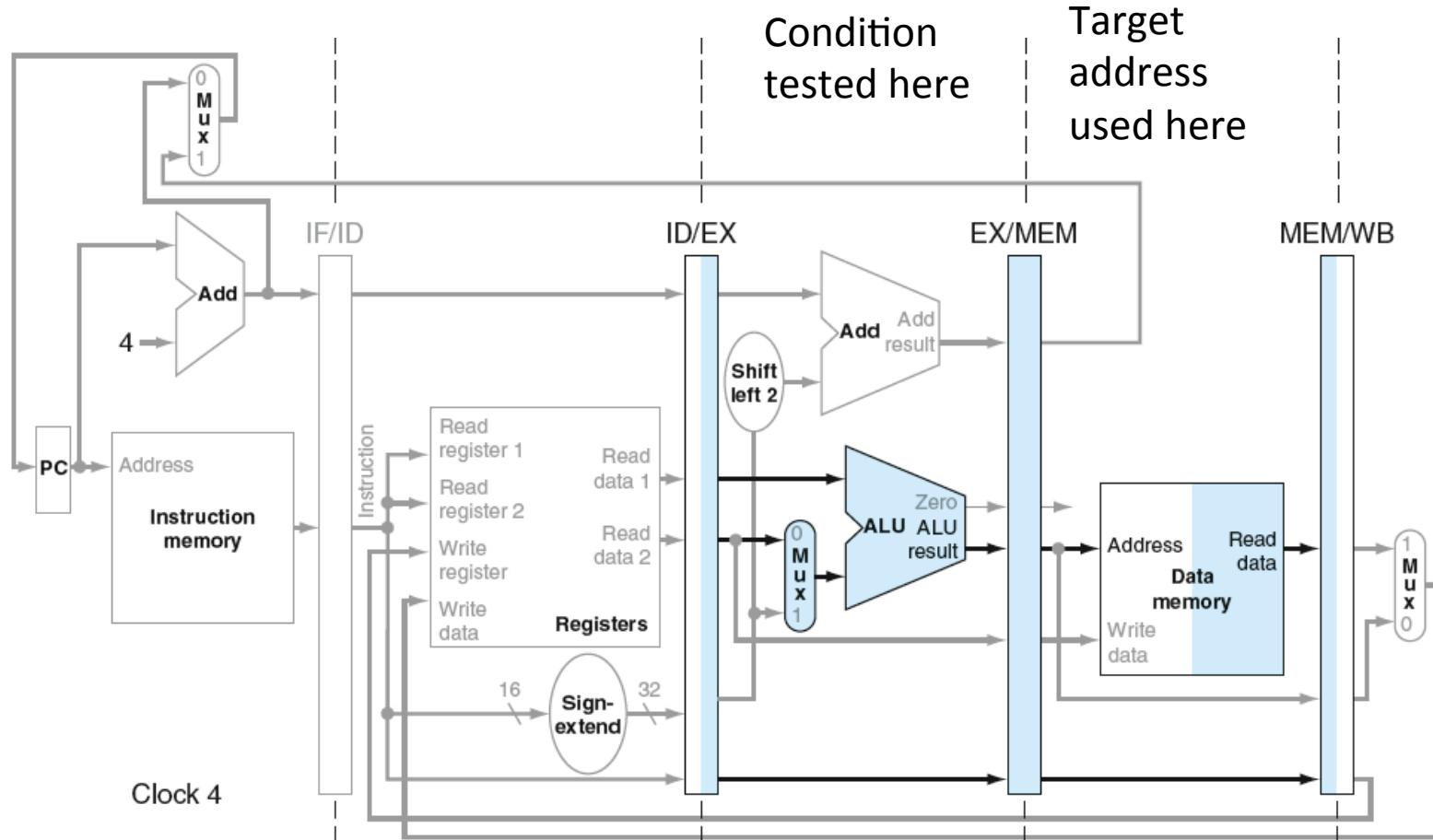
- The branch takes effect in stage 4

Instructions behind a taken branch must not complete

- Stages 1 through 3 must be flushed

- This creates 3 bubbles (a 3-cycle penalty)

The effect is called a control hazard



If condition is true, the PC is loaded with the target address when beq is in stage 4

sub \$11,\$6,\$5

beq \$11,\$0,skip

add \$4,\$7,\$3

sw \$9,4(\$7)

add \$9,\$2,\$9

.

.

.

skip: or \$8,\$4,\$0

If \$11 = 0, the 3 instructions that follow beq should not complete.
The next instruction to execute should be or \$8,\$4,\$0 instruction

Cycle	IF	ID	EX	MEM	WB
1	sub \$11,\$6,\$5				
2	beq \$11,\$0,skip	sub \$11,\$6,\$5			
3	add \$4,\$7,\$3	beq \$11,\$0,skip	sub \$11,\$6,\$5		
4	sw \$9,4(\$7)	add \$4,\$7,\$3	beq \$11,\$0,skip	sub \$11,\$6,\$5	
5	add \$9,\$2,\$9	sw \$9,4(\$7)	add \$4,\$7,\$3	beq \$11,\$0,skip	sub \$11,\$6,\$5
6	or \$8,\$4,\$0				beq \$11,\$0,skip

Flushing adds 3 extra clock cycles in this case

This is known as the branch penalty

1. Delay fetching instructions until branch behavior is known

Still causes 3-cycle penalty (outcome is known in stage 4)
2. Employ delayed branches

Fill delay slots with instructions needing to execute in any case
Compiler or programmer fills delay slots
Use NOPs if useful instructions cannot be identified
3. Evaluate the branch condition early

Requires computing target address in stage 2
Requires comparator in stage 2
4. Predict the behavior of the branch instruction

Requires recording previous branch behavior
Flushing is still required if prediction is wrong



The behavior of an unconditional branch is known

- The branch is decoded in stage 2

- The instruction following the branch is in stage 1

- This instruction could be flushed

Allowing the instruction to execute avoids a bubble

- The instruction is said to occupy the “branch delay slot”

- Compilers can move an instruction into the delay slot

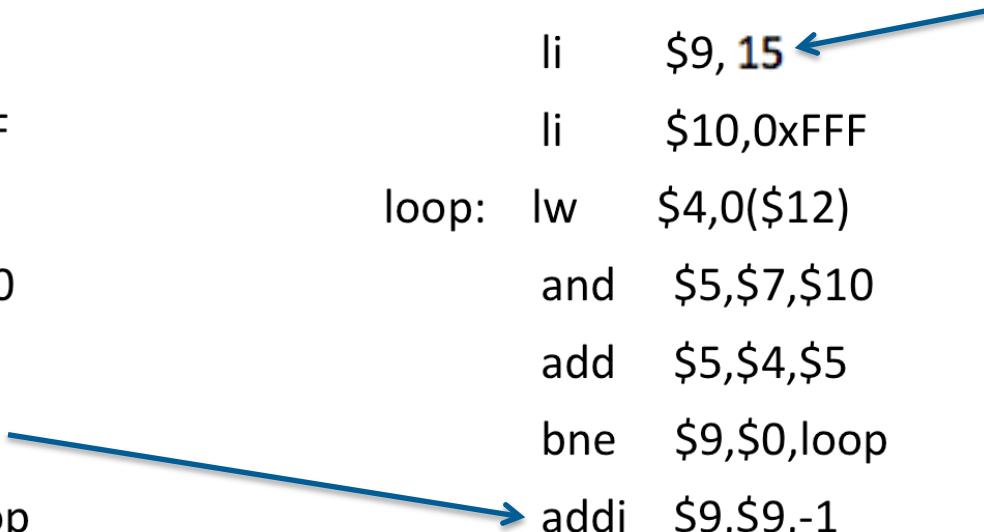
- It is easier to fill a single slot than to fill 3

li	\$9,16	li	\$9,16
li	\$10,0xFFFF	li	\$10,0xFFFF
loop:	lw \$4,0(\$12)	loop:	lw \$4,0(\$12)
	and \$5,\$7,\$10		and \$5,\$7,\$10
	add \$5,\$4,\$5		add \$5,\$4,\$5
	addi \$9,\$9,-1	bne \$9,\$0,loop	
	bne \$9,\$0,loop	addi \$9,\$9,-1	
	nop		



Branch delay slot instruction executes whether branch is taken or not
A cycle is saved by filling the slot with a useful instruction

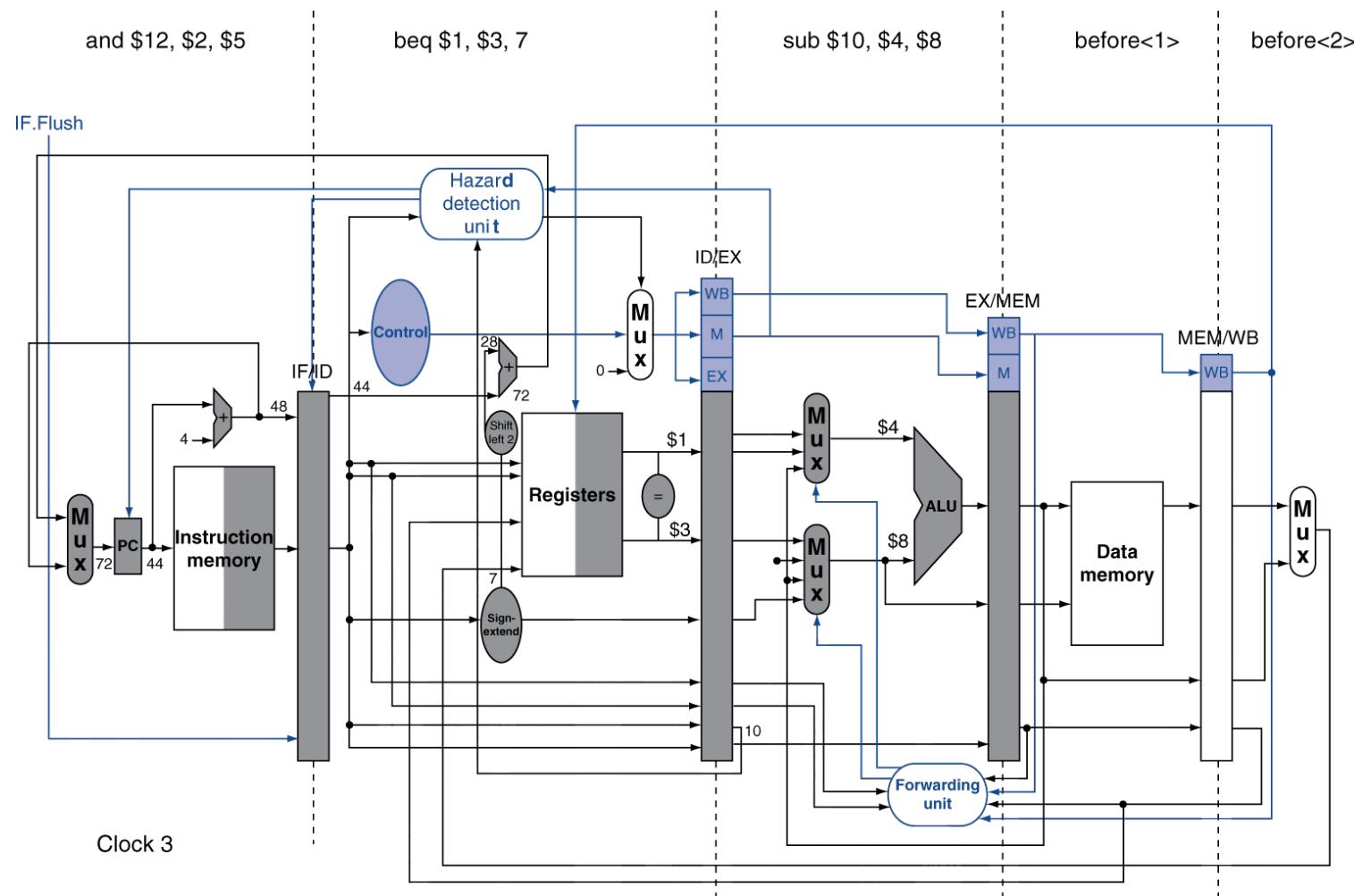
li	\$9,16	li	\$9, 15
li	\$10,0xFFFF	li	\$10,0xFFFF
loop:	lw \$4,0(\$12)	loop:	lw \$4,0(\$12)
	and \$5,\$7,\$10		and \$5,\$7,\$10
	add \$5,\$4,\$5		add \$5,\$4,\$5
	addi \$9,\$9,-1	bne \$9,\$0,loop	
	bne \$9,\$0,loop		addi \$9,\$9,-1
	nop		



The initial value in the loop control register has been set to 15 so that the number of loop iterations will be the same as for the original code.

- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken

```
36: sub $10, $4, $8
40: beq $1, $3, 7
44: and $12, $2, $5
48: or $13, $2, $6
52: add $14, $4, $2
56: slt $15, $6, $7
...
72: lw $4, 50($7)
```



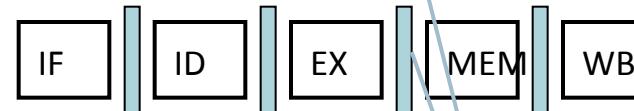
Delay slot is still needed since condition is tested in stage 2

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

add \$1, \$2, \$3

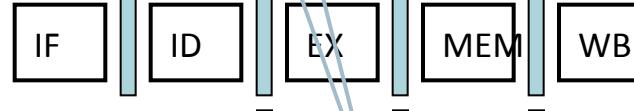


add \$4, \$5, \$6



...

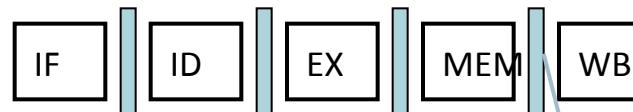
beq \$1, \$4, target



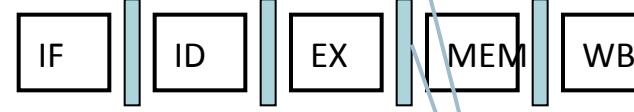
- Can resolve using forwarding

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
 - Need 1 stall cycle

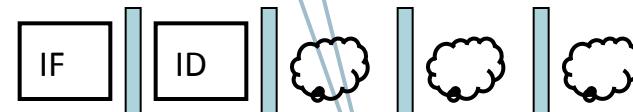
lw \$1, addr



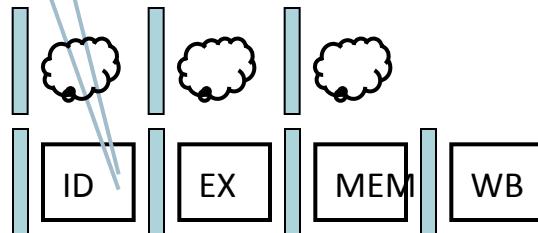
add \$4, \$5, \$6



beq stalled



beq \$1, \$4, target



- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles

lw \$1, addr



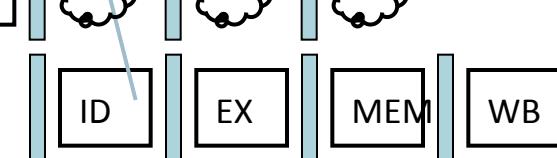
beq stalled



beq stalled



beq \$1, \$0, target



Branches have more impact on deeper and superscalar pipelines

More stages may have to be flushed

Superscalar pipelines process multiple instructions per stage

Branch prediction will be examined as another technique

Studies have shown:

20% to 30% of program instructions involve branching

About 65% of branches are taken

Instructions along the predicted path are speculative
the work done must be undone if the prediction is wrong

There are two options for branch prediction:

static prediction & dynamic prediction

Static prediction can be done by the compiler

Dynamic prediction requires extra hardware

Static (fixed) Prediction

The prediction is always the same (taken or not taken)

Forward branches may be predicted not taken

Backwards branches may be predicted taken
e.g. at end of loops

Once actual behavior is known, work may have to be undone

Dynamic Branch Prediction Increases accuracy of prediction

Previous history of branch behavior is recorded

Shows if branch was taken or not when last encountered

Branch prediction buffer

Branch History table (BHT)

Decode History table (DHT)

These tables are high-speed buffers (caches)

address of branch instruction is used to access them

BHT is accessed as soon as PC is updated

- Hits only occur for branch instructions already in the table

- Hits provide the predicted target address

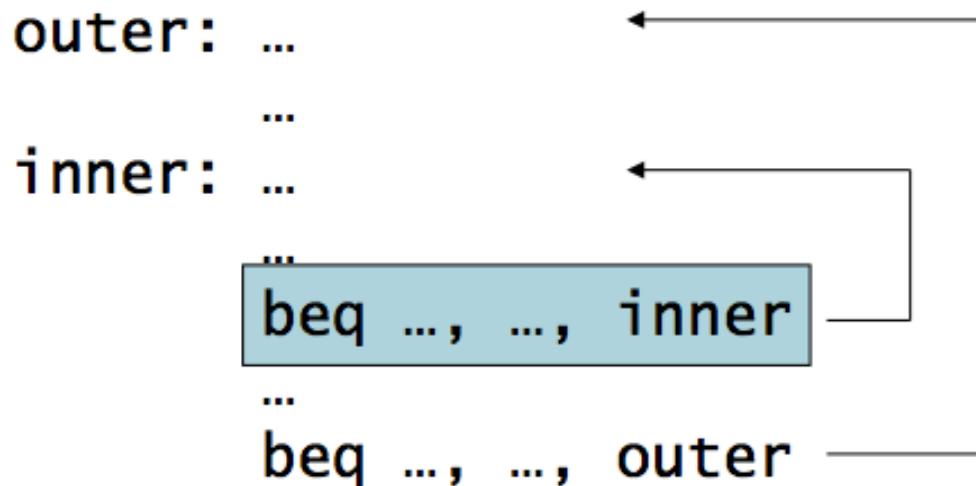
- Non-branch instructions cause misses

- Can be accessed before branch enters the pipeline

DHT is only accessed after branch is decoded

- It is not checked for non-branch instructions

- hits provide the predicted target instruction



Let's base prediction on a recorded bit (=0 if not taken, =1 if taken)

Assume 9 iterations of inner loop, 1st prediction & 9th are wrong

For each iteration of outer loop, inner beq is mispredicted twice

A 2-bit predictor would be better

2-bit predictor	Meaning
00	Strongly not taken
01	Weakly not taken
10	Weakly taken
11	Strongly taken

On 1st encounter bits change from 00 to 01 (for previous example)

On 2nd encounter bits change from 01 to 10

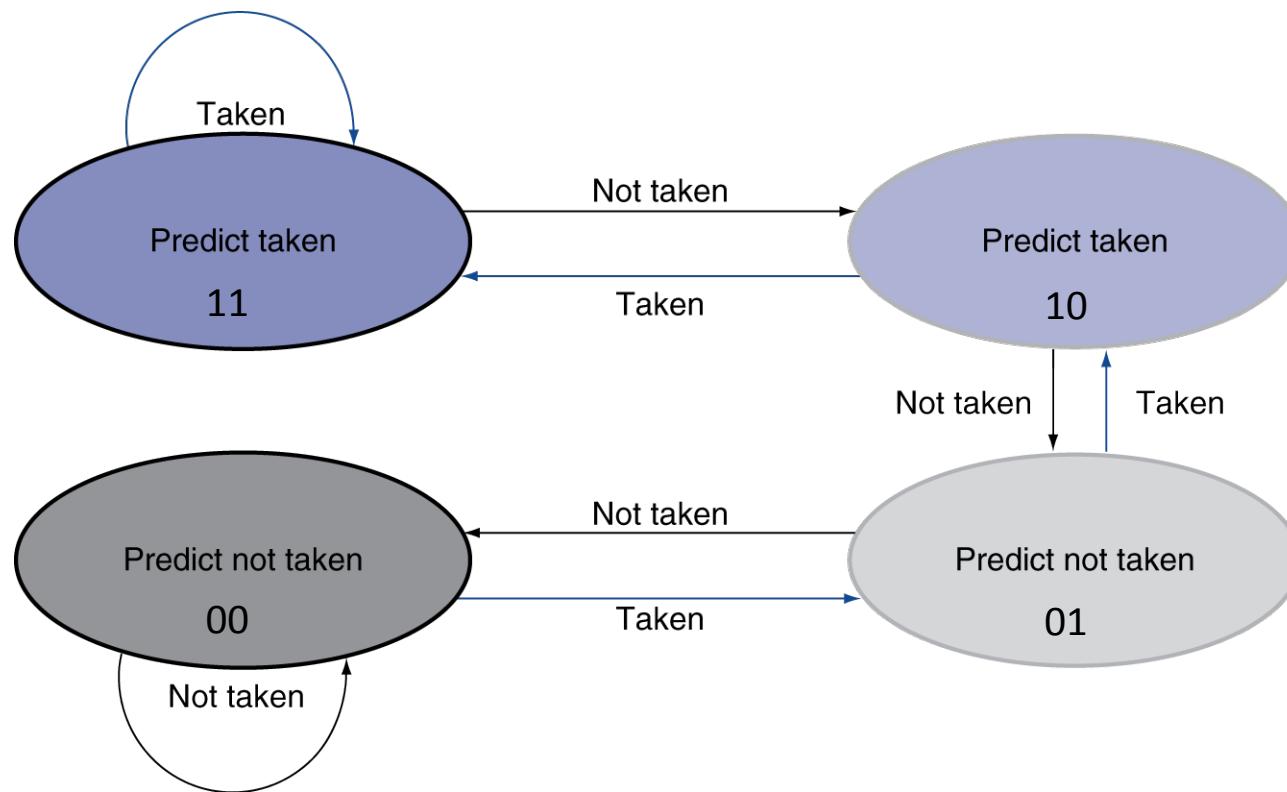
On 3rd encounter bits change from 10 to 11

On 9th encounter bits change from 11 to 10

Inner prediction is wrong three times for 1st iteration of outer loop

But wrong only once for each remaining outer loop iteration

- Only change prediction on two successive mispredictions



Exceptions affect the pipeline like a function call
control is diverted to the exception handler
interrupts are a particular type of exception

Exceptions are triggered by unexpected events
detecting invalid instructions during decode
arithmetic overflow
memory errors
syscall
interrupts from external I/O controllers

System Coprocessor (CP0) manages MIPS exceptions

EPC is the exception program counter (CP0 \$14)

Holds address of offending (or interrupted) instruction

Cause register (CP0 \$13) indicates problem

Control is transferred to handler at 0x800000018

handler takes the appropriate action

resumes program using address in EPC

terminates program if problem cannot be resolved

Transfer to exception handler causes a control hazard
program instructions are flushed from pipeline
this creates pipeline bubbles

The effect on pipeline is similar to a mispredicted branch

- Exception on add in

```
40      sub    $11,   $2,   $4
44      and    $12,   $2,   $5
48      or     $13,   $2,   $6
4C      add    $1,    $2,   $1
50      s1t    $15,   $6,   $7
54      lw     $16,  50($7)
```

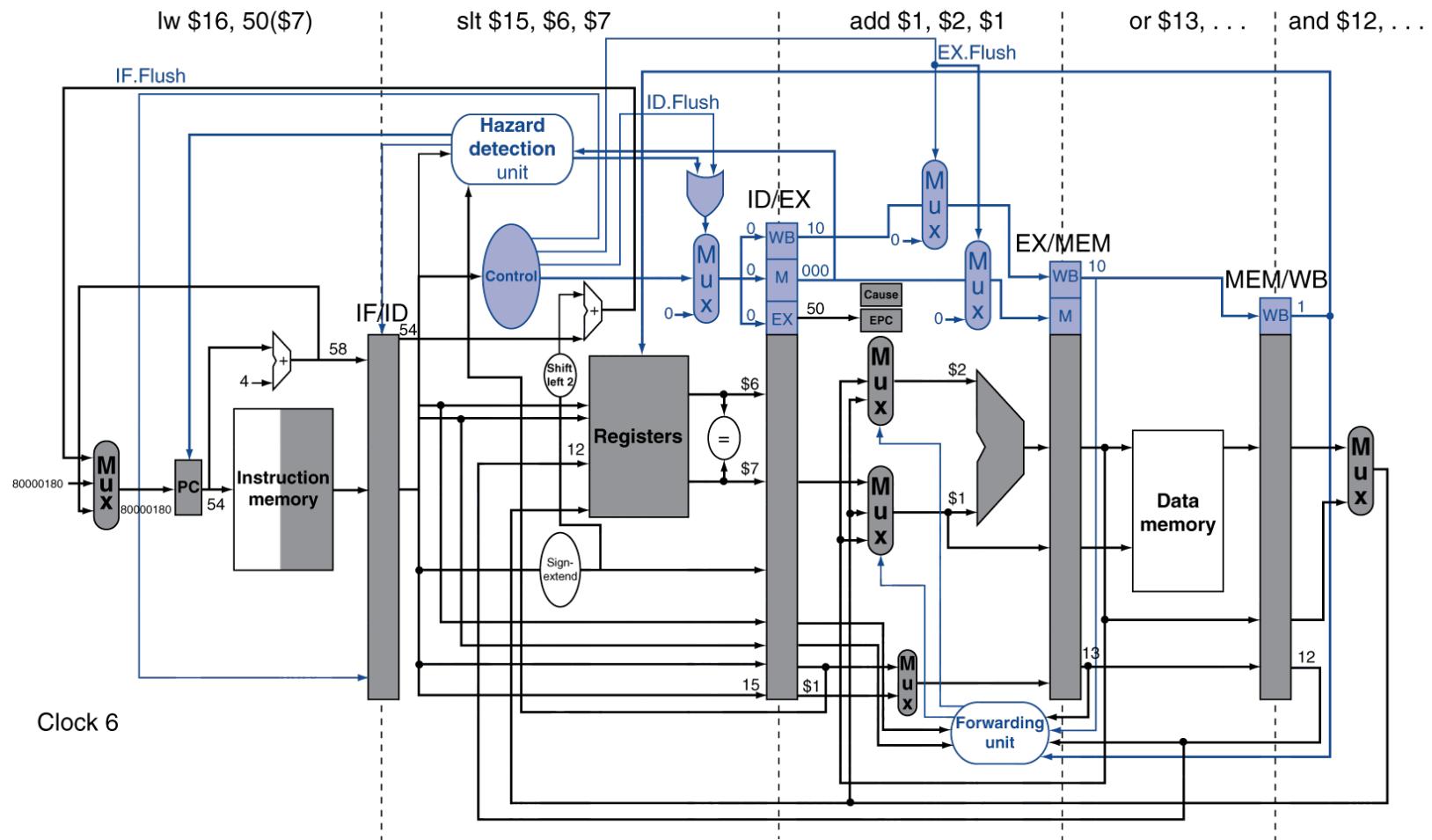
...

- Handler

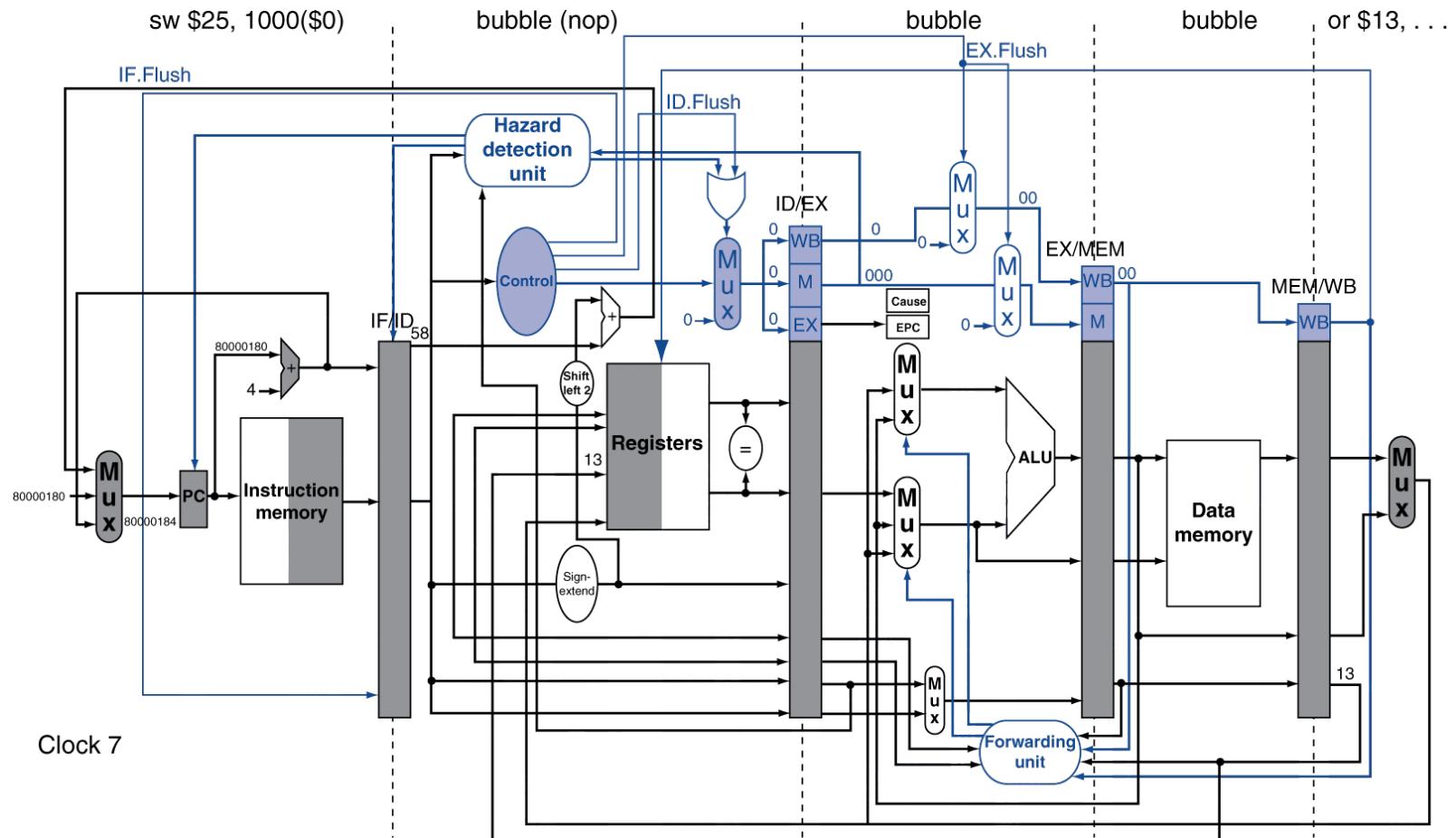
```
80000180      sw     $25,  1000($0)
80000184      sw     $26,  1004($0)
```

...

Exception Example



Flush control signal inserts bubbles



Flush control signal inserts bubbles

Some systems support precise exceptions
no register writes after the offending instruction
leaves the system in a consistent state

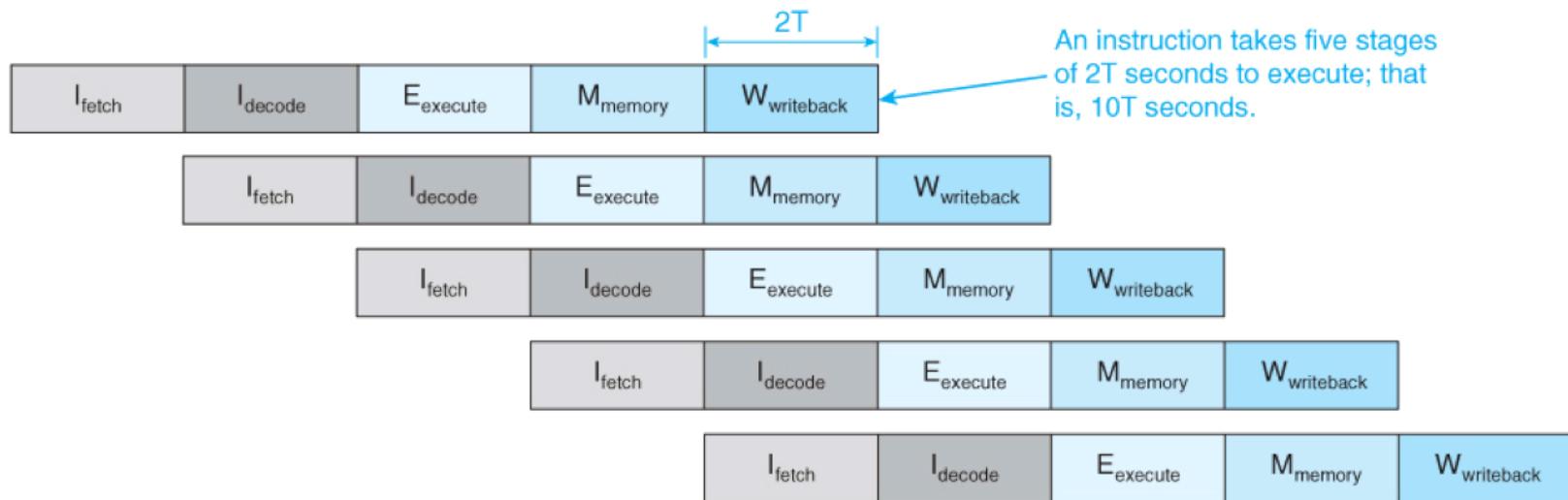
Imprecise exceptions
instruction causing exception is identified
but succeeding instructions may complete
allows the system to be in an inconsistent state

Superpipelining allows more instructions to be overlapped

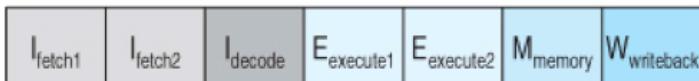
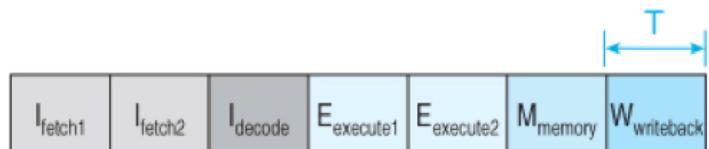
Operations may not all require a complete clock cycle
register reads or writes
check for hits in cache
decoding an opcode

The pipeline can run faster than the external clock rate

Superpipelines have some stages subdivided
a different instruction is in each phase with a stage



Conventional pipeline with cycle time = $2T$



The fetch stage has been divided into two pipelined stages.



An instruction takes five stages of $2T$, T , $2T$, T , T seconds, that is, $7T$ seconds.

The writeback stage requires only one cycle.



$$\text{Speedup} = 10/7 = 1.43$$

Superpipeline with cycle time = T (twice the rate)

Superpipelines use finer granularity
instruction throughput increases
the cost is a higher clock rate

Other parts of the datapath may require a slower rate
precludes running the entire system at a higher clock rate

Hazards & mispredictions have a greater impact
more stages have to be flushed

More stages cause more interstage delays
more pipeline registers

Superpipelines overlap more instructions

Superpipelining provides a modest performance increase

Superscalar provides a greater benefit

Superscalar systems can include superpipelining

Pipelining's goal is a throughput of 1 instruction per cycle
data hazards & control hazards make this difficult

Scalar systems handle 1 instruction per stage
each stage requires one clock cycle
only one instruction is started each cycle
all instructions go through all stages
some instructions may take longer than without the pipeline
however the instruction throughput is increased

Superscalar systems aim for 2 or more instructions per cycle
extra hardware executes multiple instruction per cycle
multiple pipelines can operate in parallel
or each stage can process multiple instructions at one time

Multiple instructions are fetched at one time
requires a wider CPU-to-memory bus

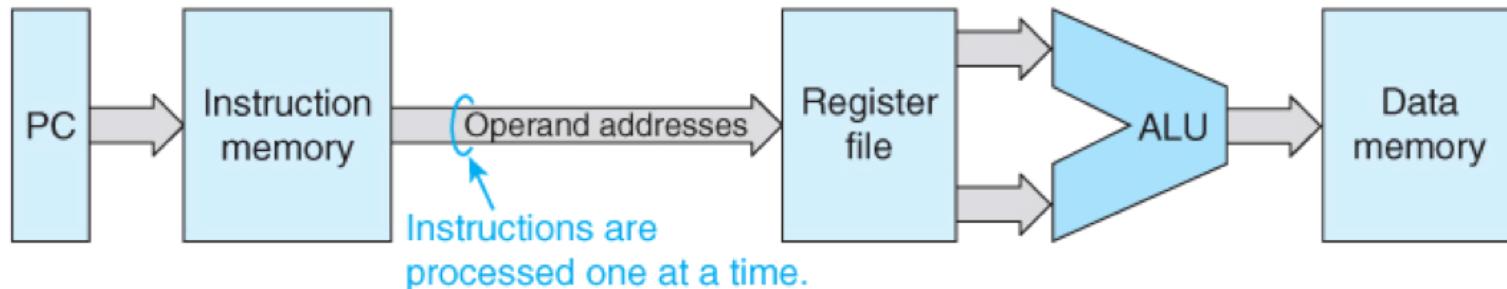
Multiple instructions are decoded together
dependencies between the instructions are detected
independent instructions are issued to their execute units

Multiple execute units process instructions at the same time
multiple ALUs for integer operations
multiple floating point units
load/store unit (to compute memory addresses)
branch unit to analyze branch conditions & make predictions

Resource Hazards are possible with superscalar systems
the required unit may not be available for next instruction
required registers may already be in use

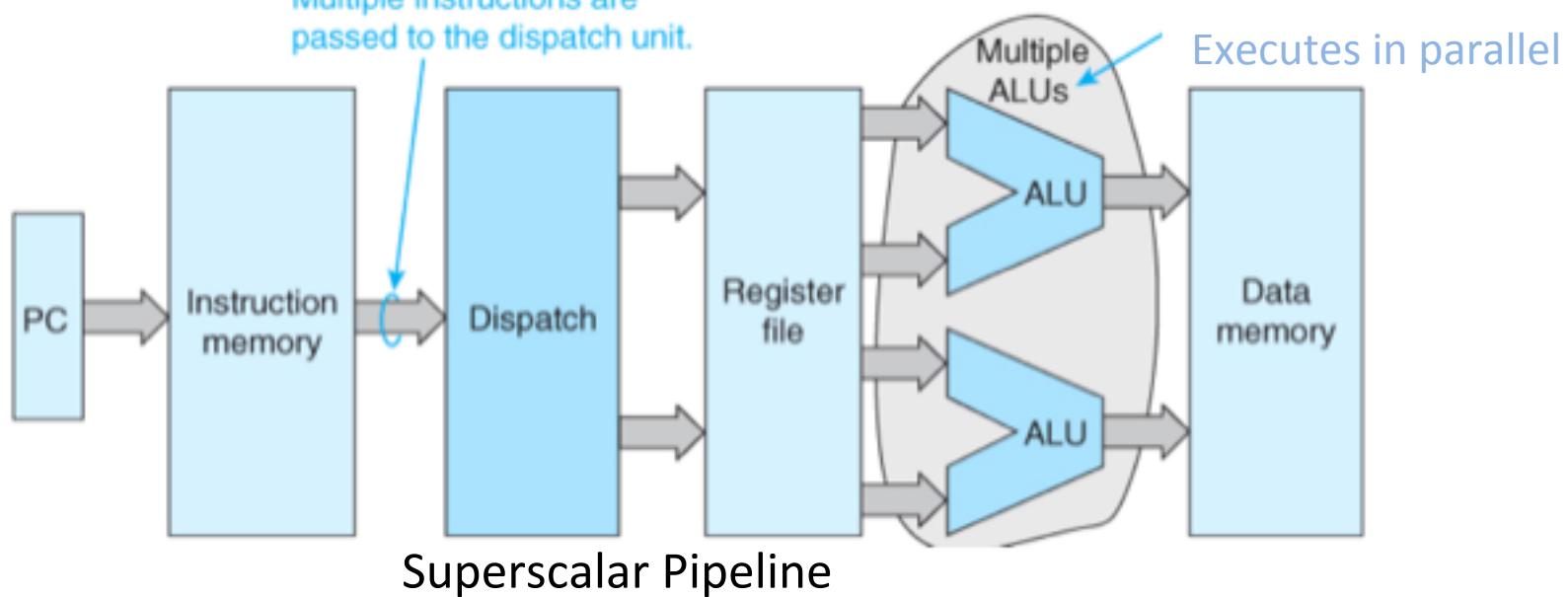
Out-of-order execution may be needed for good performance
instructions may start in a different order than in program
later instructions may complete before earlier ones

Scalar Pipeline



Multiple instructions are passed to the dispatch unit.

Executes in parallel



An m-way superscalar system has m parallel pipelines
m-fold increases in performance are seldom achieved

The Pentium P5 had 2 integer pipelines
the 2nd pipeline could only be used about 30% of the time

The original Pentium used superscalar operation
two 5-stage integer pipelines U and V
U pipe included a shifter not in the V pipe
a 6-stage floating point unit

Compilers generate machine instructions in program order

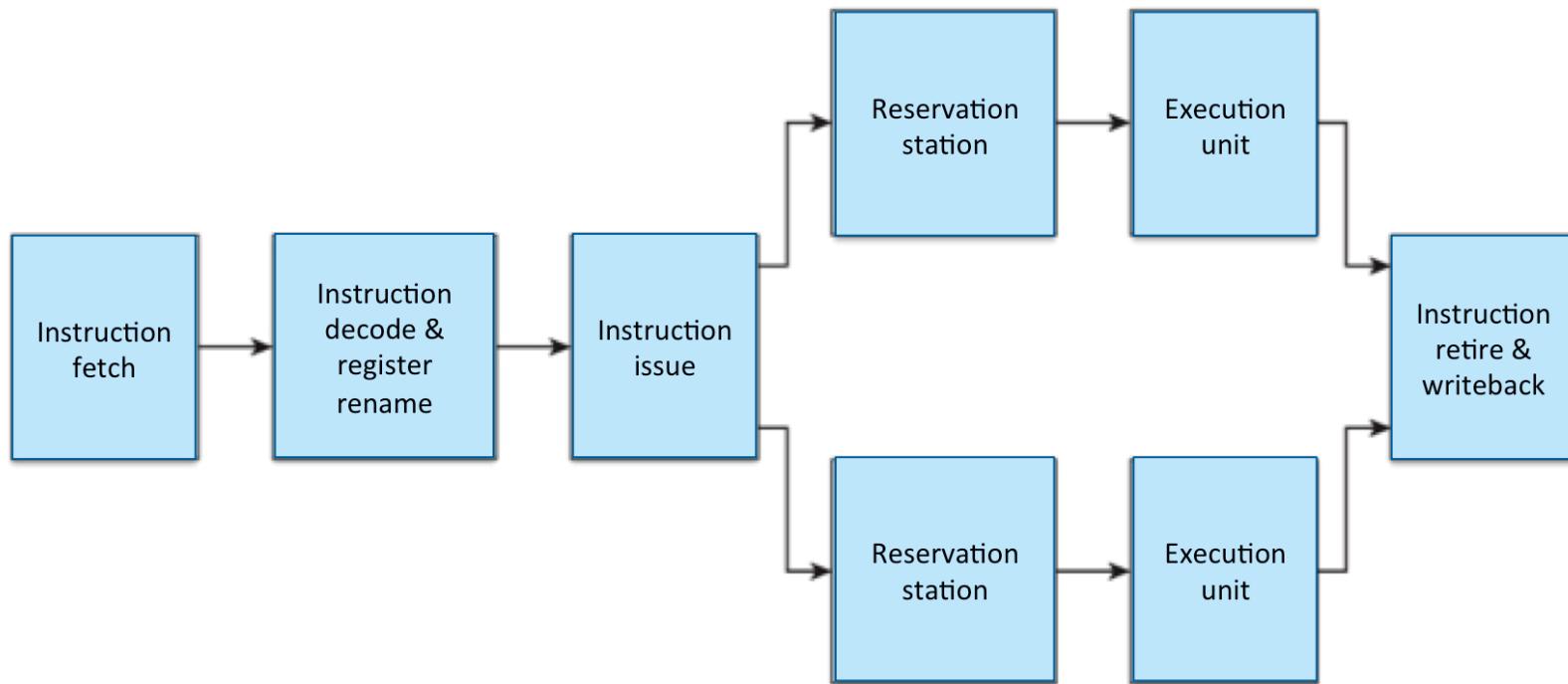
Superscalar processors *dispatch* multiple instructions in parallel
operands are obtained from multiported register files
Instructions that have operands can execute in parallel
The execute units needed must be available

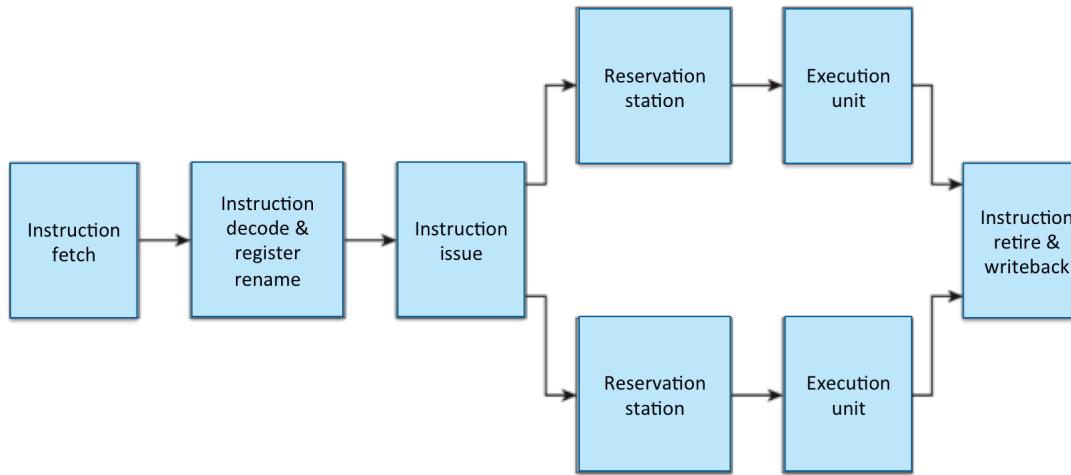
The program semantics (meaning must be preserved)

	Sequence 1	Sequence 2
add	\$11,\$12,\$13	\$11,\$12,\$13
add	\$14,\$11,\$13	\$15,\$16,\$17
add	\$15,\$16,\$17	\$14,\$11,\$13

Different instruction order but same meaning or net effect

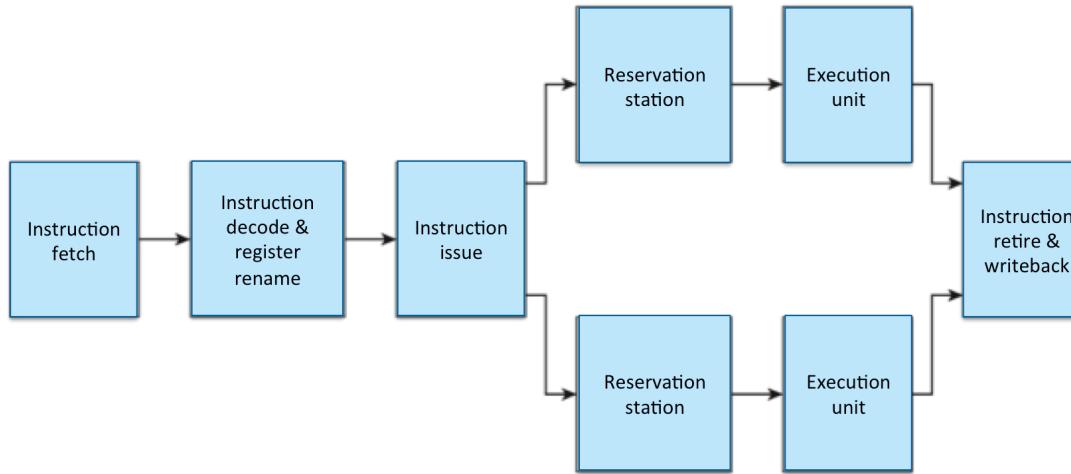
A more realistic view of a superscalar system is:





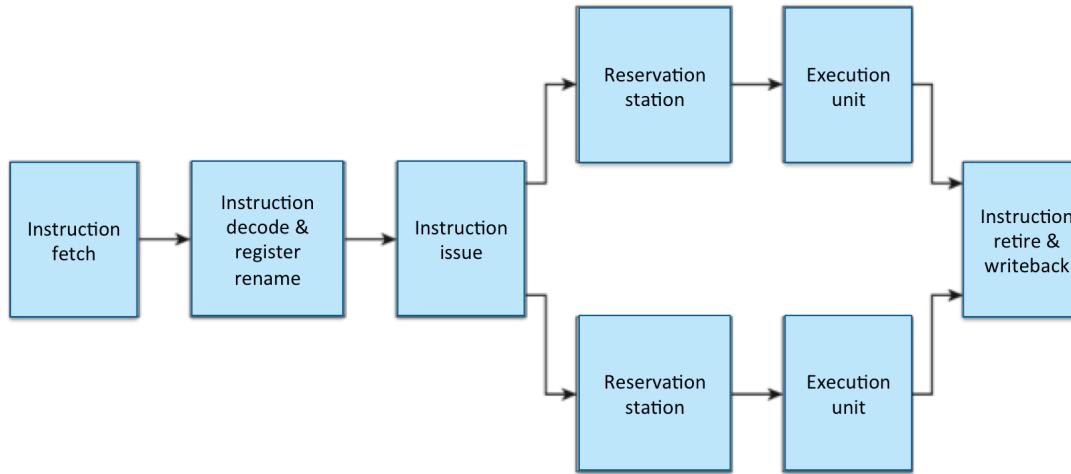
Instruction fetch
obtains instructions from cache or memory

Instruction Decode
interprets opcodes
substitutes temporary registers to avoid unnecessary stalls



Instruction issue

ensures as many instructions as possible execute in parallel
sends instructions to reservation stations (*issues* them)
Instructions are *dispatched* from the reservation stations
once the required input operands are available



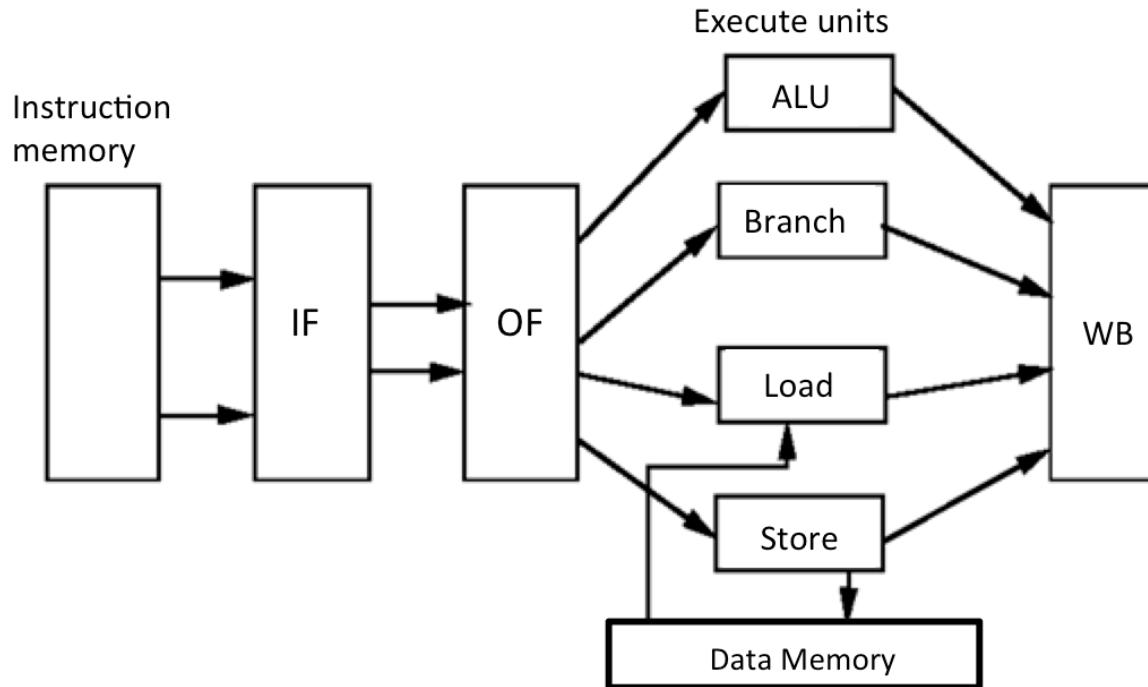
Reservation stations

Serve as front end buffer to execution units

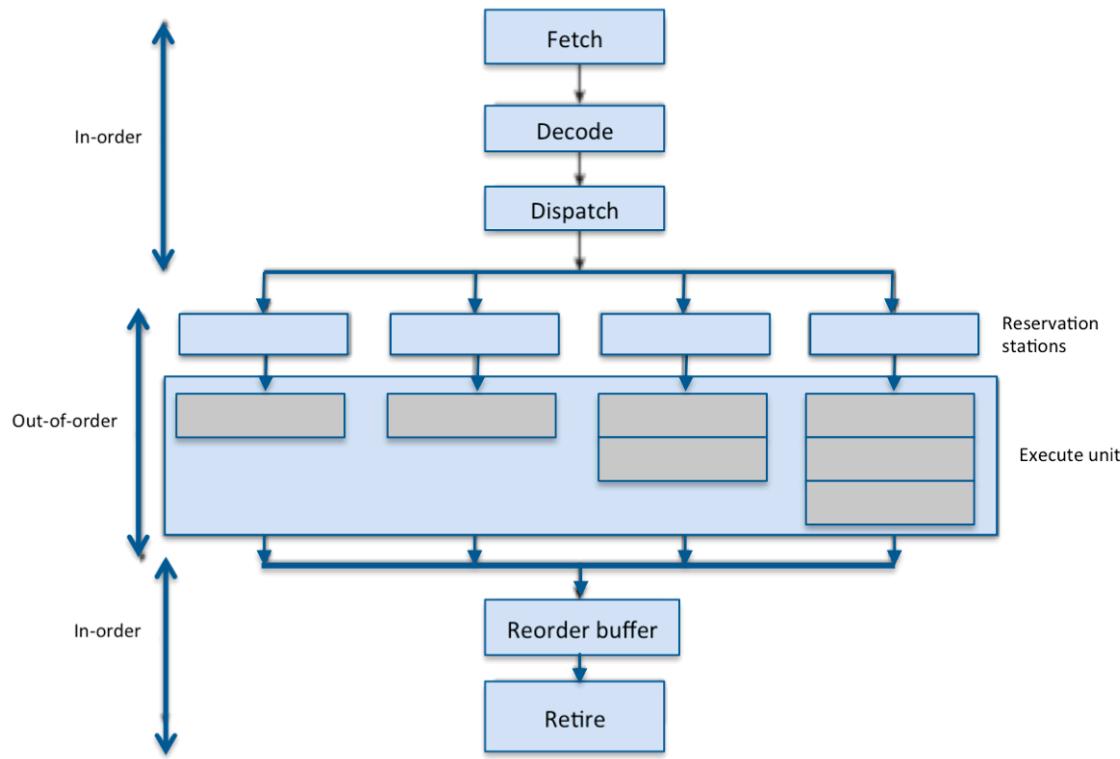
Hold instructions that use the corresponding execution unit

Instruction retire

Writes the results to the destination registers (*commits* them)
Tells reservation stations when resources are available



Even with in-order issue, instructions may complete out-of-order
e.g. An add or shift may be issued after a lw or floating point
instruction, but may finish executing first.



Reorder buffers hold completed instructions
ensure that the program meaning or outcome is preserved
instructions that complete out-of-order are retired in-order
retire means writing the result to the correct register

Possible issue/completion options:

in-order issue & in-order completion

 used in scalar pipeline

in-order issue & out-of-order completion

out-of-order issue & in-order completion

out-of-order issue & out-of-order completion

Changing execution order can cause other types of data dependencies

1. True data dependency (RAW) read after write

Sub \$3, \$2, \$4

Add \$5,\$3,\$4 # must read \$5 after it is written by sub

2. Antidependency (WAR) write after read

or \$2, \$4, \$3

sub \$4, \$5,\$6 # can't write \$4 until after the or reads \$4

The or may be stalled waiting on a resource or on \$3

3. Output dependency (WAW) write after write

sub \$3,\$2,\$4

add \$5,\$6,\$5

slt \$3,\$7,\$0 # must write \$3 after sub writes \$3

The use of registers can create apparent dependencies

1. or \$3, \$3,\$5
2. add \$4,\$3,\$2
3. add \$3, \$5, \$2
4. sub \$7,\$3,\$4

WAW exists between 3. and 1. (both write \$3)

WAR exists between 3. and 2. (3. overwrites \$3 input to 2.)

RAW exists between (1. & 2.), (2. & 4.), (3. & 4.)

The use of registers can create apparent dependencies

1. or \$3_a, \$3_a, \$5
2. add \$4, \$3_a, \$2
3. add \$3_b, \$5, \$2
4. sub \$7, \$3_b, \$4

3_a and 3_b are unrelated, so a different register can be used for 3_b

Eliminates false dependencies

Register renaming performs this dynamic substitution

Register renaming adapts to larger register sets

Applications use more registers without having to be recompiled

Recompilation is required without register renaming feature

Instruction-level Parallelism (IPL)
the ability to execute multiple instructions together

Pipelining provides IPL by overlapping instruction execution

Superscalar systems replicate hardware to provide IPL
multiple instructions can be sent through separate pipelines
multiple units can execute instructions simultaneously

Multiple issue means starting more than 1 instruction at a time

Static multiple issue is based on:

compiler deciding which instruction can execute together

Independent instructions are grouped into packets or bundles

This packaging occurs prior to execution (compile time)

Bundles or packets are assigned to issue slots

These are also called very long instruction words (VLIW)

Packets are issued within a single clock cycle

Packed instructions must map to separate resources

Instruction mix may be restricted

Dynamic multiple issue is based on:

Control unit deciding which instruction can execute together

Decisions are made by hardware during execution

Based on hazards (data, resources, etc.)

Compiler may assist by reordering machine instructions

Consider a 2-issue MIPS processor

Two instructions are fetched and executed together

Requires 64-bit CPU-to-memory bus

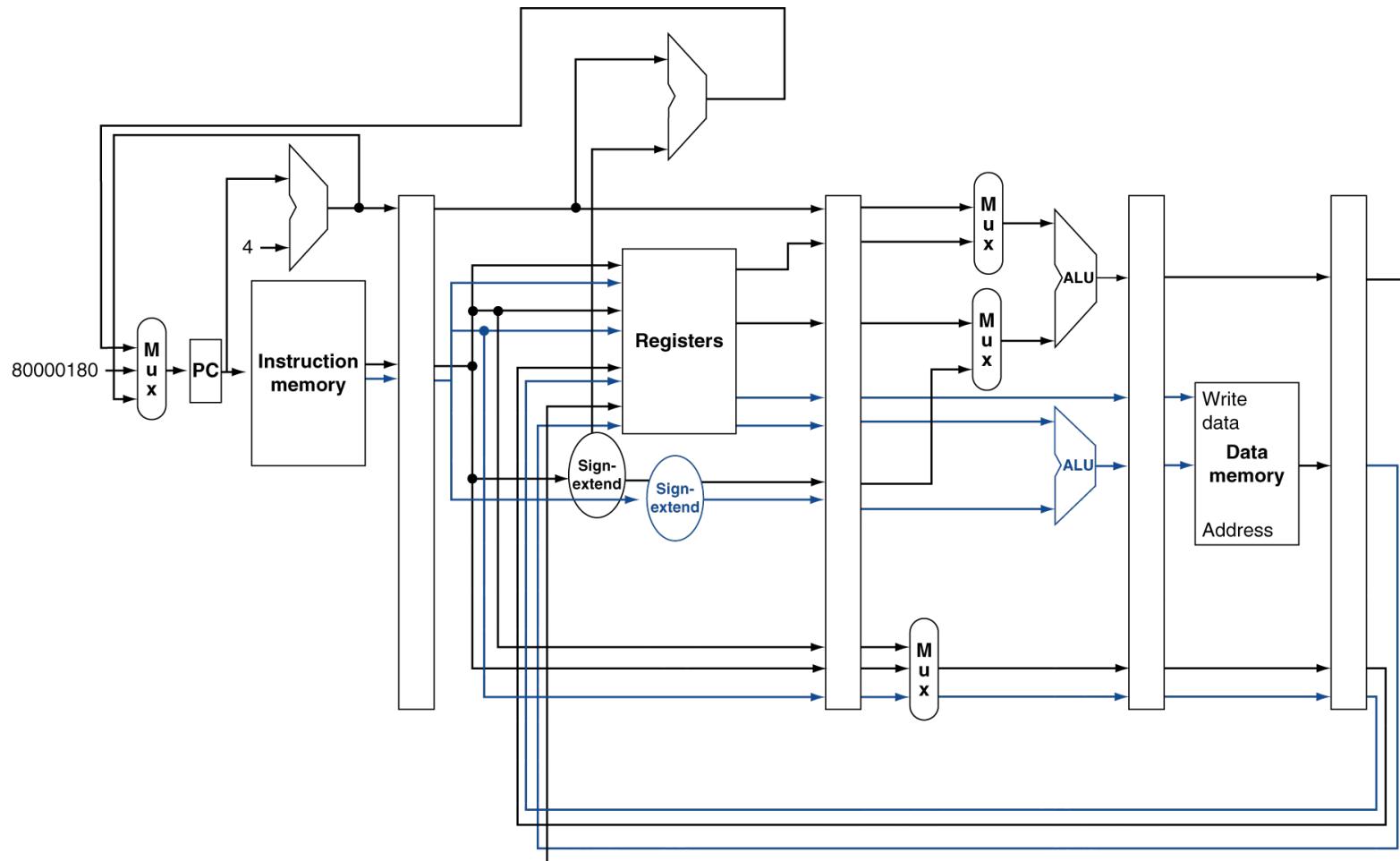
One can be an integer or branch instruction

The other can be a load or store instruction

Instructions are always issued in pairs

Nop instruction replaces an unused slot within pair

Required stalls holdup both instructions in a pair



Extra resources: updated register file, unit to compute memory address,
2nd sign extension unit

- More instructions executing in parallel
- EX data hazard
 - Forwarding avoided stalls with single-issue
 - Now can't use ALU result in load/store in same packet
 - add \$t0, \$s0, \$s1
 - load \$s2, 0(\$t0)
 - Split into two packets, effectively a stall
- Load-use hazard
 - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

```
Loop: 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, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1,-4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)



- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- Use different registers per replication
 - Called “register renaming”
 - Avoid loop-carried “anti-dependencies”
 - Store followed by a load of the same register
 - Aka “name dependence”
 - Reuse of a register name

	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1,-16	lw \$t0, 0(\$s1)	1
	nop	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, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

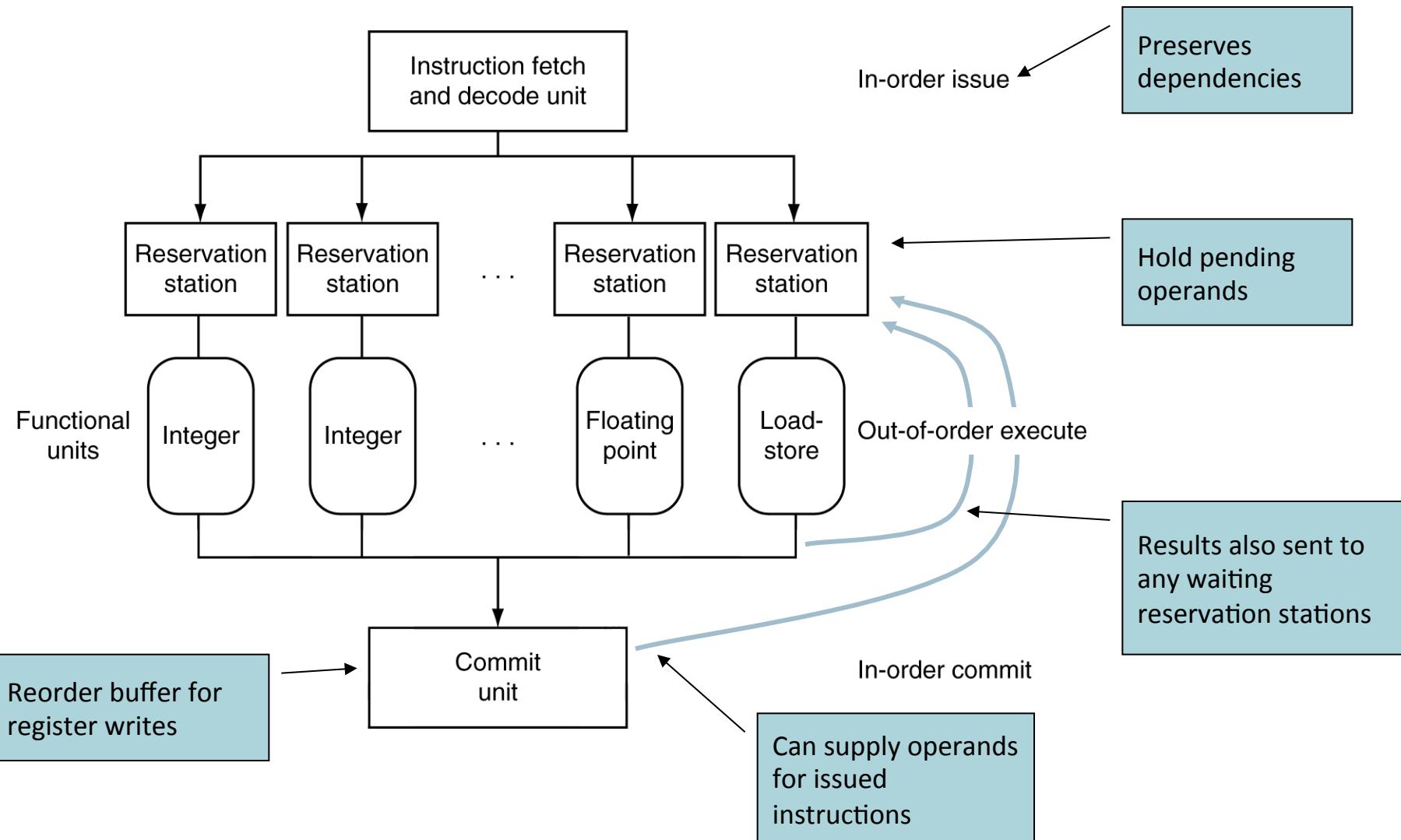
- $IPC = 14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size

- “Superscalar” processors
- CPU selects instructions to execute each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though compiler may still help
 - Code semantics ensured by the CPU

- Allow the CPU to execute instructions out of order to avoid stalls
 - But commit result to registers in order
- Example

```
lw      $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slt   $t5, $s4, 20
```

- Can start sub while addu is waiting for lw



- Register renaming via reservation stations (RS) & ROB
 - ROB is reorder buffer
- On instruction issue to reservation station
 - copy available operand to reservation station
 - from register file or from ROB
 - Once done, register can be overwritten
 - When available, operands are provided to RS
 - Come from the function unit producing the result
 - Register update may not be required


```
Loop: 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, $zero, Loop  # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw \$t0, 0(\$s1)	1
	addi \$s1, \$s1,-4	nop	2
	addu \$t0, \$t0, \$s2	nop	3
	bne \$s1, \$zero, Loop	sw \$t0, 4(\$s1)	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)



- Replicate loop body to expose more parallelism
 - Reduces loop-control overhead
- Use different registers per replication
 - Called “register renaming”
 - Avoid loop-carried “anti-dependencies”
 - Store followed by a load of the same register
 - Aka “name dependence”
 - Reuse of a register name

Loop:

```
lw    $t0, 0($s1)
addu $t0, $t0, $s2
sw    $t0, 0($s1)
addi $s1, $s1,-4
lw    $t0, 0($s1)
addu $t0, $t0, $s2
sw    $t0, 0($s1)
addi $s1, $s1,-4
lw    $t0, 0($s1)
addu $t0, $t0, $s2
sw    $t0, 0($s1)
addi $s1, $s1,-4
lw    $t0, 0($s1)
addu $t0, $t0, $s2
sw    $t0, 0($s1)
addi $s1, $s1,-4
lw    $t0, 0($s1)
addu $t0, $t0, $s2
sw    $t0, 0($s1)
addi $s1, $s1,-4
bne  $s1, $zero, Loop
```

Loop:

```
addi $s1, $s1,-16
lw    $t0, 16($s1)
addu $t0, $t0, $s2
sw    $t0, 16($s1)
lw    $t1, 12($s1)
addu $t1, $t1, $s2
sw    $t1, 12($s1)
lw    $t2, 8($s1)
addu $t2, $t2, $s2
sw    $t2, 8($s1)
lw    $t3, 4($s1)
addu $t3, $t3, $s2
sw    $t3, 4($s1)
bne  $s1, $zero, Loop
```



	ALU/branch	Load/store	cycle
Loop:	addi \$s1, \$s1,-16	lw \$t0, 0(\$s1)	1
	nop	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, \$t4, \$s2	sw \$t1, 12(\$s1)	6
	nop	sw \$t2, 8(\$s1)	7
	bne \$s1, \$zero, Loop	sw \$t3, 4(\$s1)	8

(a 0 displacement & the original value in \$s1 are used in the first lw instruction)

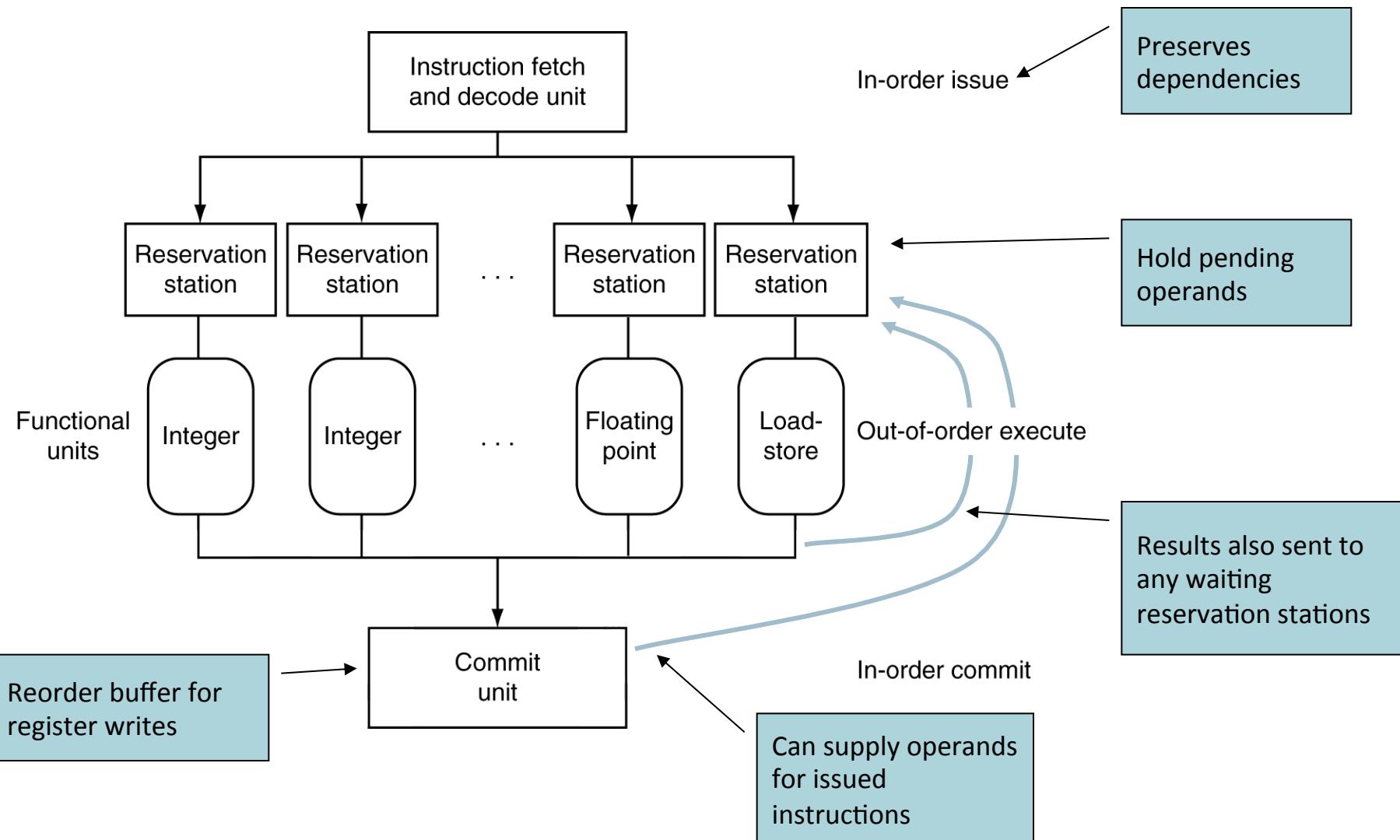
- $\text{IPC} = 14/8 = 1.75$
 - Closer to 2, but at cost of registers and code size

- “Superscalar” processors
- CPU selects instructions to execute each cycle
 - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
 - Though compiler may still help
 - Code semantics ensured by the CPU

- Allow the CPU to execute instructions out of order to avoid stalls
 - But commit result to registers in order
- Example

```
lw      $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slti   $t5, $s4, 20
```

- Can start sub while addu is waiting for lw

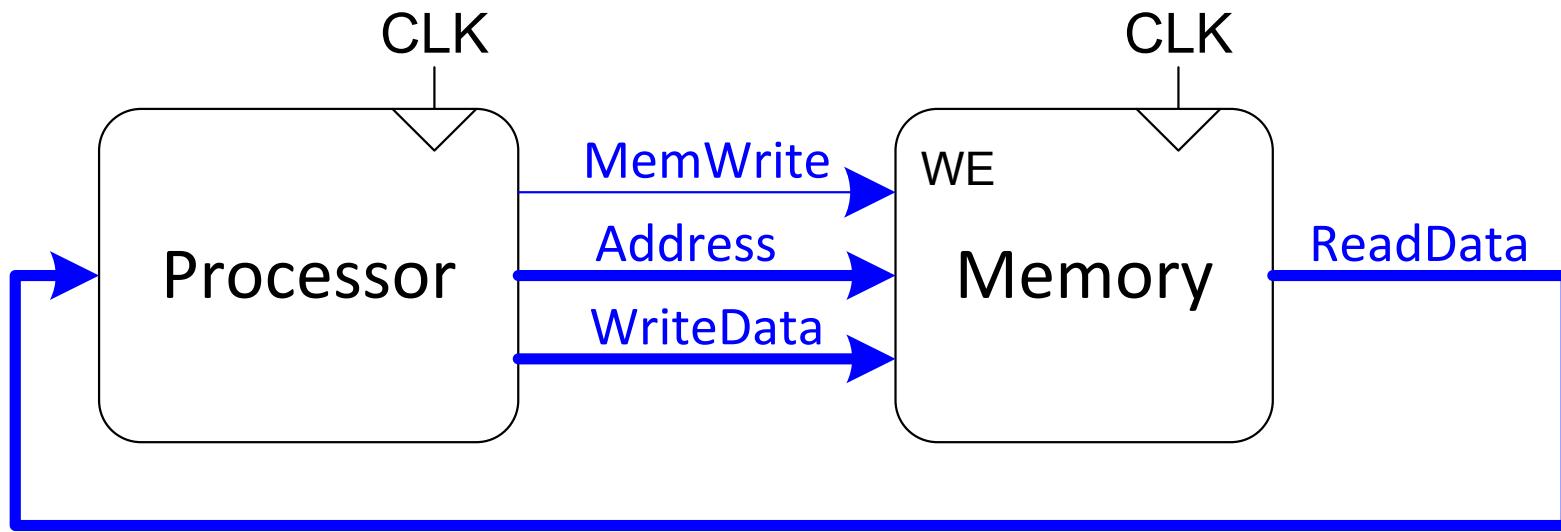


- Register renaming via reservation stations (RS) & ROB
 - ROB is reorder buffer
- On instruction issue to reservation station
 - copy available operand to reservation station
 - from register file or from ROB
 - Once done, register can be overwritten
 - When available, operands are provided to RS
 - Come from the function unit producing the result
 - Register update may not be required

- Provides storage for instructions and data
- Large storage capacity eases the task of developing programs
- Greater speed improves performance and reduces the need to stall the CPU
- Reduced cost makes the overall system more economical
- All of these goals cannot be achieved at the same time
- Techniques such as caching and virtual memory can give the illusion of greater speed and capacity

- There are two basic types of memory
 - Read/write can be changed or updated (RAM)
 - Read-only can be read but not changed (ROM)
- Memory access is a read or write operation
 - Location to access must be specified
 - Read obtains copy of contents
 - Write replaces contents with specified data
- Each location is assigned a unique number (address)
- Any location in RAM or ROM can be accessed directly
- Storage capacity is measured in units of 8-bit bytes

Example: Processor writes to memory by specifying the address, the data, and the write control signal.



The clock signal (CLK) synchronizes the interactions.

- Most systems are “byte addressable”
 - Individual bytes can be accessed
 - Actual transfer size matches bus width
- Multi-byte items usually must reside on proper boundary
 - Word (4 bytes) address must be multiple of 4
 - Half word (2 bytes) address must be even
 - Address of aligned data item is a multiple of its size
 - Unaligned items may require multiple transfers
 - Unaligned accesses cause exceptions on MIPS

- MIPS memory features
 - Employs 32-bit addresses (4 GB address space)
 - Byte-addressable
 - Enforces memory alignment
- Amount of physical memory dictates number of address bits
- Width of pathway (bus) dictates number of bytes in a transfer
- Usually, at most 1 read or write can occur at a time

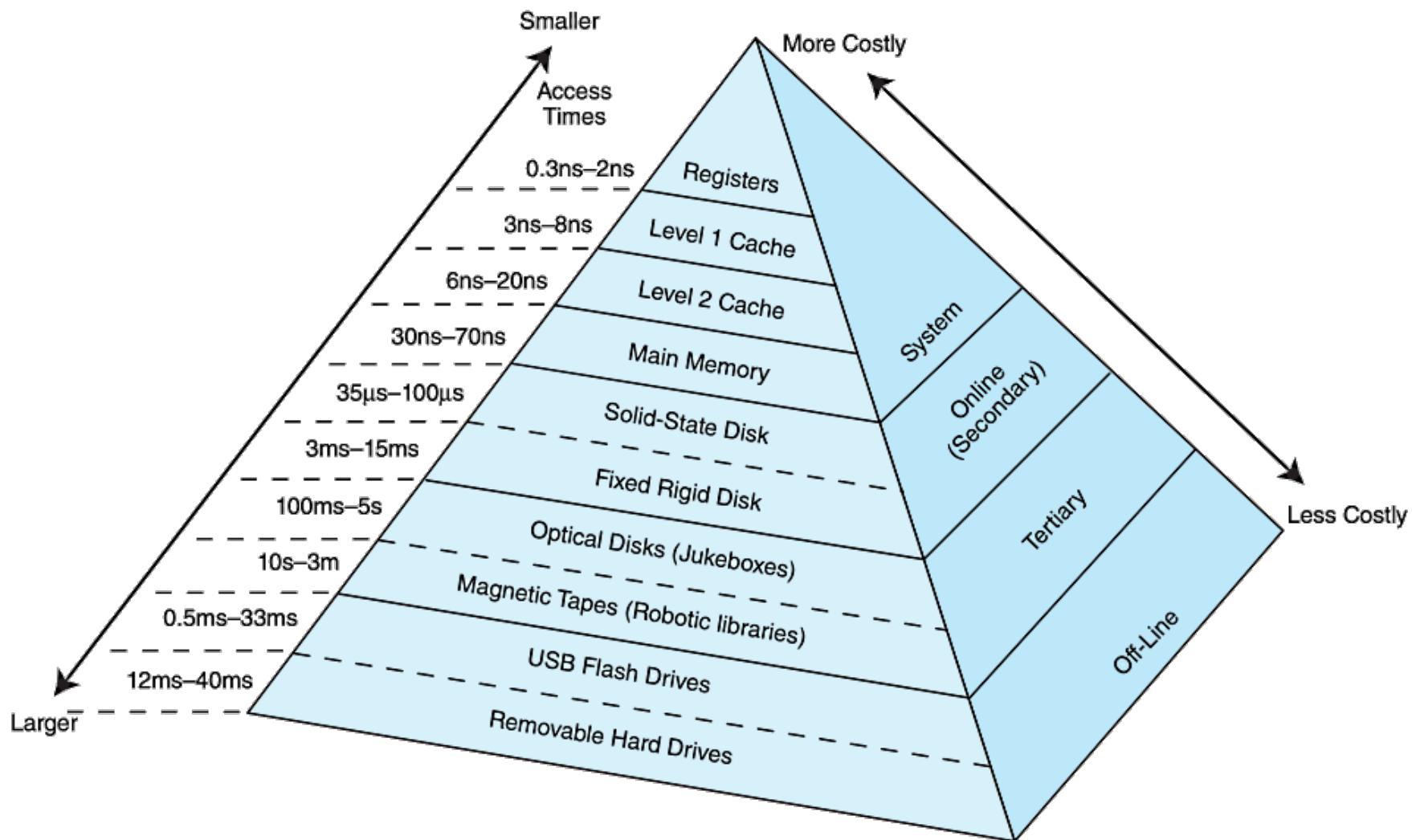
- “Access time”
 - time between read request and return of data
- “Memory cycle time”
 - minimum time between consecutive reads
 - Includes setup time, access time and recovery time
- Addresses are sent over the address bus
- Data bits are sent over the data bus
- Read/write request signals are sent over the control bus

- There are two types of RAM
 - Dynamic RAM (DRAM)
 - Static RAM (SRAM)
- DRAM stores charge to represent 0 or 1
 - Charges leaks off overtime
 - Requires periodic refresh to restore charge
 - Must be charged before a read occurs
 - Reads are destructive (must rewrite to restore)
 - Relatively inexpensive
 - Allows more bits per unit area (more dense)
 - “Volatile” contents lost when power is off

- SRAM uses switches (gates) to store bits
 - Provides much shorter access time than DRAM
 - Contents remain stable as long as power is on
 - Reads are non-destructive
 - More expensive than DRAM
 - Consumes more area per bit than DRAM
 - Used for high speed memory (cache)
 - Volatile, contents lost when power is off

- ROM needs no refresh
 - Used to store permanent or semi-permanent data
 - Contents remains intact even when power is off
 - Reads are non-destructive
 - “Non-volatile” contents persists when power is off
- Other memory types
 - PROM (programmable) may be written once
 - EPROM (erasable PROM) exposed to UV to erase
 - EEPROM (electrically erasable) erase/rewrite in place
 - FLASH (entire blocks must be erased)

- In general, faster memory is more expensive than slower memory
- Different types of memory can be arranged in a hierarchy
- Small fast storage elements (registers) are kept in the CPU
- Cache is slightly slower than registers and kept close to the CPU
- Larger slower main memory is accessed through the bus
- Even larger and much slower storage (disks, tapes, network drives) are farther from the CPU



- Most systems are byte addressable
 - Each 8-bit byte is assigned a unique number (address)
 - Addresses range from 0 to some maximum
 - Consecutive byte addresses differ by 1
- Maximum address depends on address register width
- Larger storage units consist of multiple bytes
 - Words (4 bytes), half words (2 bytes)
 - Word size matches the CPU register size

- Some systems may be word addressable
 - Each word contains multiple bytes
 - Consecutive word addresses differ by 1
- MIPS processor uses 32-bit registers and addresses
 - Addresses range from 0 to $2^{32} - 1$
 - Registers and words contain 4 bytes

- SW instruction copies a register into memory
 - Leftmost byte is stored at the lowest address
 - Next lower byte is stored at the next higher address
 - This “big endian” storage order is used by the MIPS
- Example: if 0x12345678 is stored at address 200:

Address	200	201	202	203
contents	0x12	0x34	0x45	0x78

- Other systems use “little endian” memory storage
 - Rightmost byte is stored at the lowest address
 - Next higher byte is stored at next higher address
 - Intel systems use this memory storage order
- Example: if 0x12345678 is stored at address 200:

Address	200	201	202	203
contents	0x78	0x56	0x34	0x12

- Byte order matters when exchanging data
 - Network order is big endian
 - Little endian systems must reorder network bytes received
- Registers always contain bytes in high to low order
 - High byte on left, low byte on right
- Character strings are arrays of bytes
 - Individual bytes are accessed
 - Characters in string are ordered from first to last
 - Address of string = address of leading character
- Byte order matters when accessing multi-byte items



- We will examine registers, cache, main memory, and virtual memory.
- Registers are accessed directly by the processor
 - Instructions contain register numbers (rs, rt and rd)
 - Registers are contained within the CPU
- Virtual memory extends the address space from RAM (main memory) to the secondary storage (hard drive)
- Virtual memory provides more space: Cache memory provides speed

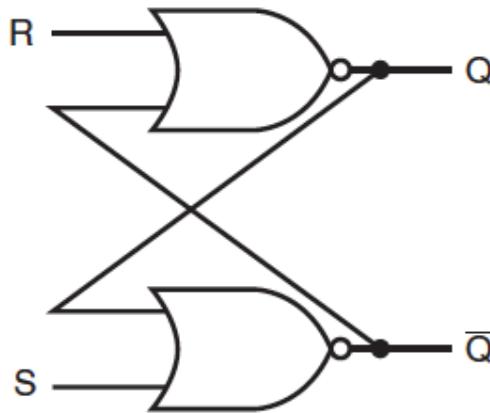
- Combinational logic circuits need state elements to:
 - provide inputs
 - store the output
- State elements are sequential logic devices
 - Their output depends on the history of the inputs
- Combinational logic output depends only on current inputs



- State changes occur at clock edges



A 1-bit memory device must capture and store its input



Cross coupled NOR gates

S and R are the inputs

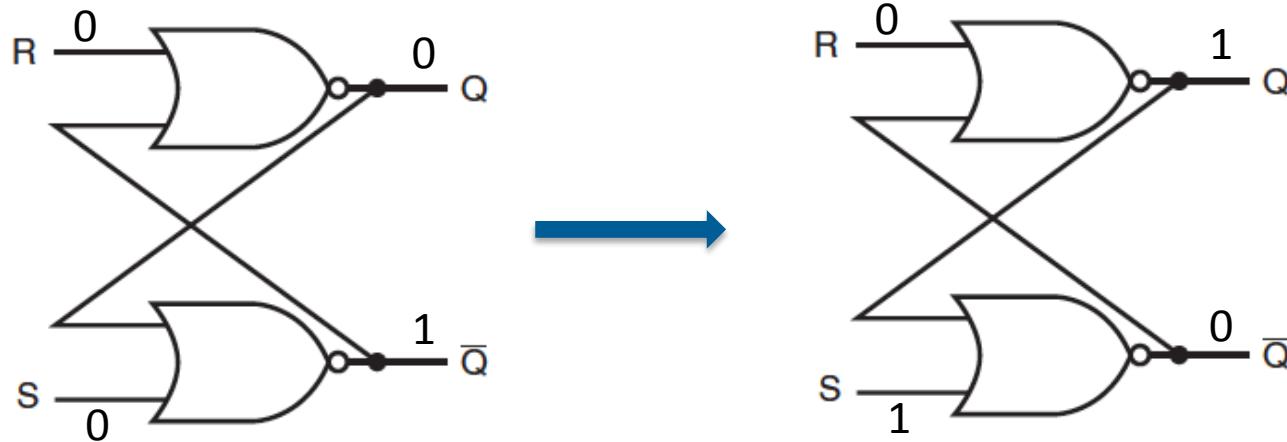
The value of the output Q defines the state (0 or 1)

\overline{Q} (NOT Q) is the other output

Outputs do not change as long as S and R are both 0



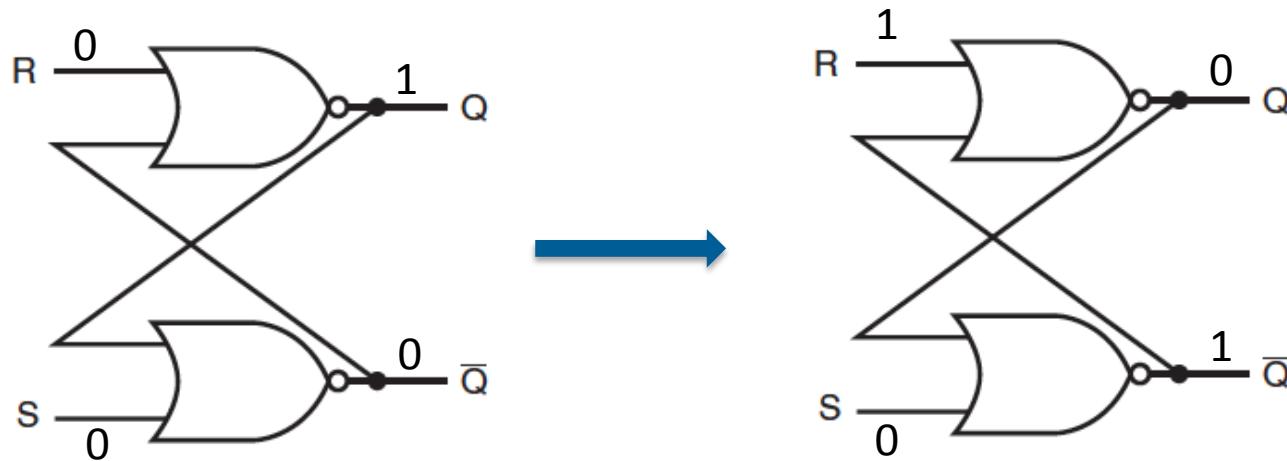
In state 0, if S changes from 0 to 1, Q becomes 1



When S goes back to 0, Q retains its new value of 1



In state 1, if R changes from 0 to 1, Q resets to 0



When R goes back to 0, Q retains its new value of 0



Behavior of the S-R Latch is described by a *characteristic table*
Defines output at $t+1$ (next cycle) as function of inputs at t

S	R	$Q(t+1)$
0	0	$Q(t)$ no change
0	1	0
1	0	1
1	1	? unpredictable

Differs from truth table which shows output for current cycle



S-R Latch

Sets $Q = 1$ if S is asserted while clock is asserted

Sets $Q = 0$ if R is asserted while clock is asserted

S-R flip-flop

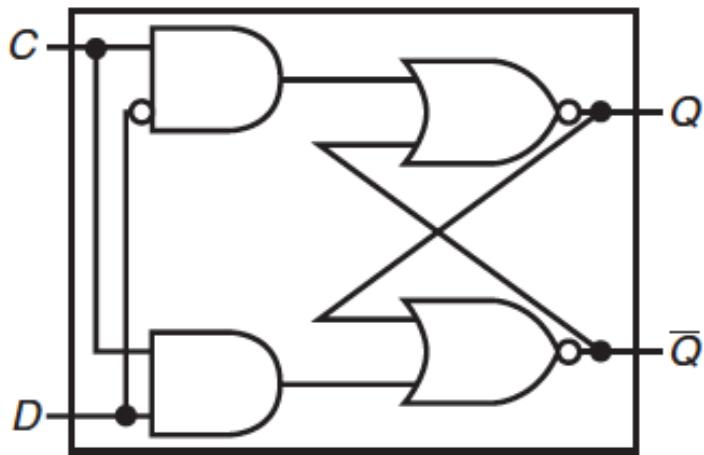
Sets $Q = 1$ if S is asserted at clock edge

Sets $Q = 0$ if R is asserted at clock edge



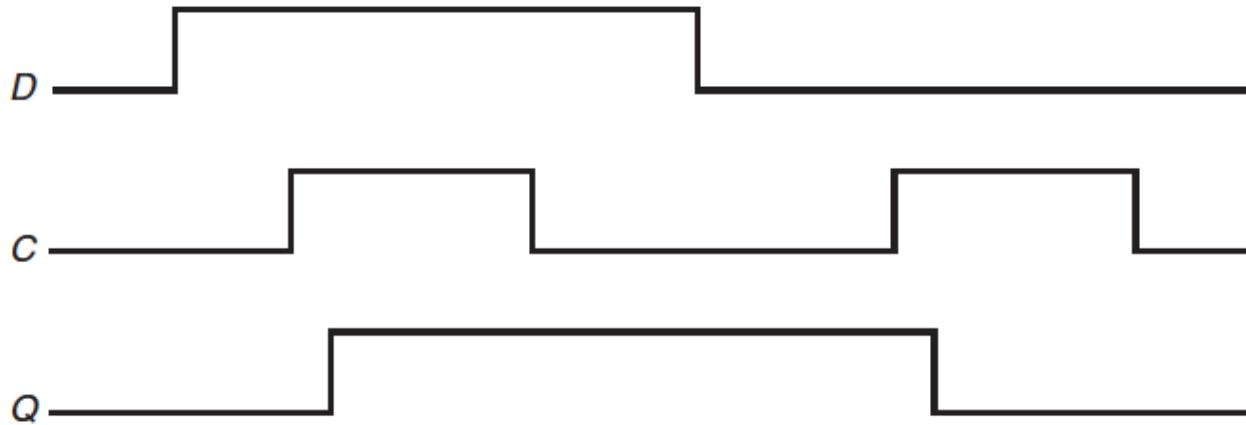
What is needed for single bit storage device is a D latch or flip-flop

D Latch has single data input and a clock input



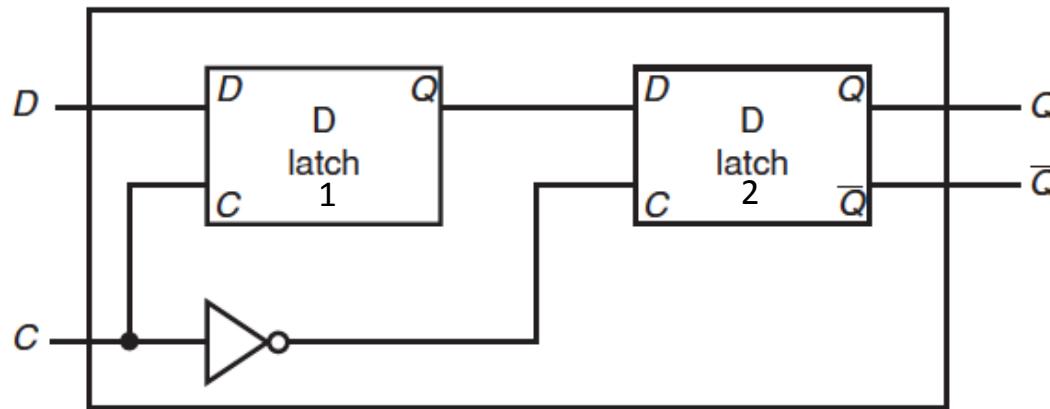
The two AND gates open when C (clock) is high

Setting D=1 has same effect as S=1 and R=0 with S-R latch
Setting D=0 has same effect as R=1 and S=0 with S-R latch



When C, the clock, is high, Q takes on the same value as D

D flip-flop (with falling edge trigger)

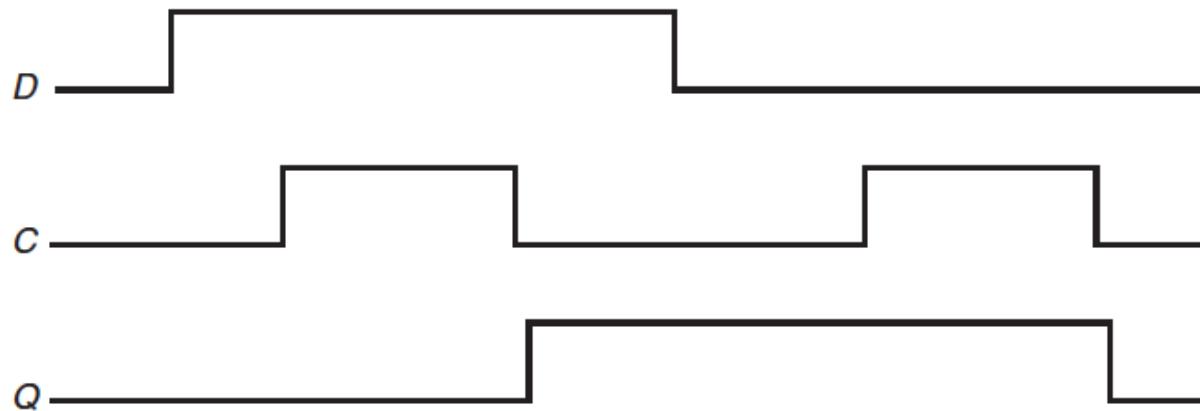


When C is high, latch 1 passes its input (D) to its output

When C goes low:

- output from latch 1 is retained
- Latch 2 passes output from latch 1 on to latch 2's output

Q only changes when clock goes high and back low



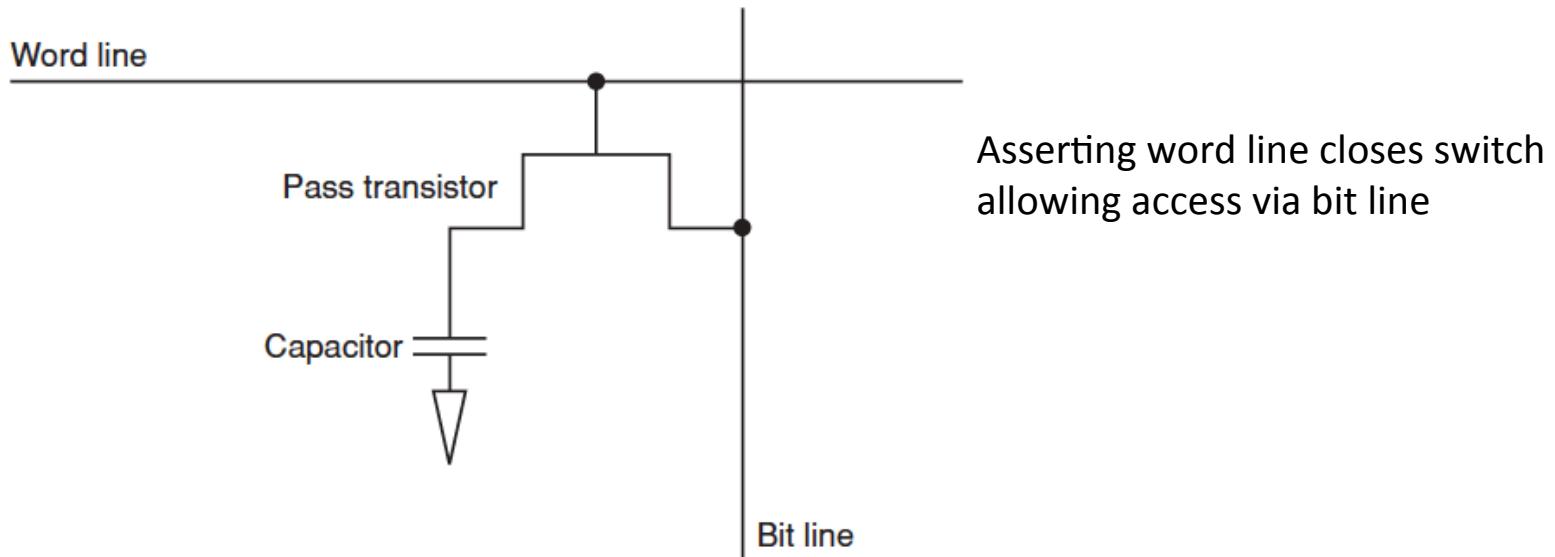
Output Q changes at trailing clock edge in this example

The alternative would be leading edge trigger

As long as power is applied, the stored bit is retained



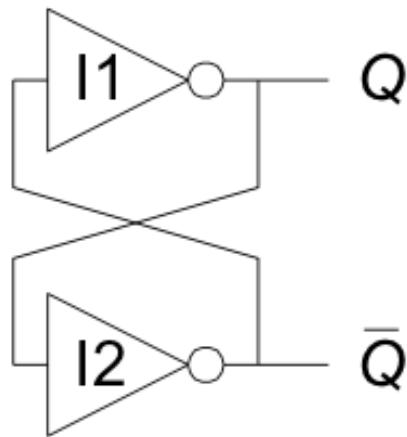
With dynamic RAM, the bit is stored as charge on a capacitor
A transistor switch allows the bit to be read or written



Presence of charge represents 1
Absence of charge represents 0
Must refresh every few milliseconds to maintain the charge

Storage cells must have 2 stable states: 0 and 1 (*bi-stable*)

Another possible implementation uses cross-strapped inverters:

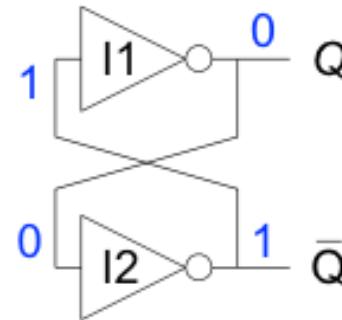


Stable output Q defines the state
State does not change



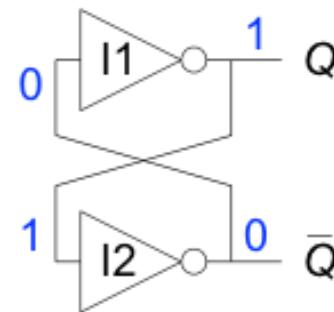
$Q = 0:$

then $\bar{Q} = 1, Q = 0$ (consistent)



$Q = 1:$

then $\bar{Q} = 0, Q = 1$ (consistent)

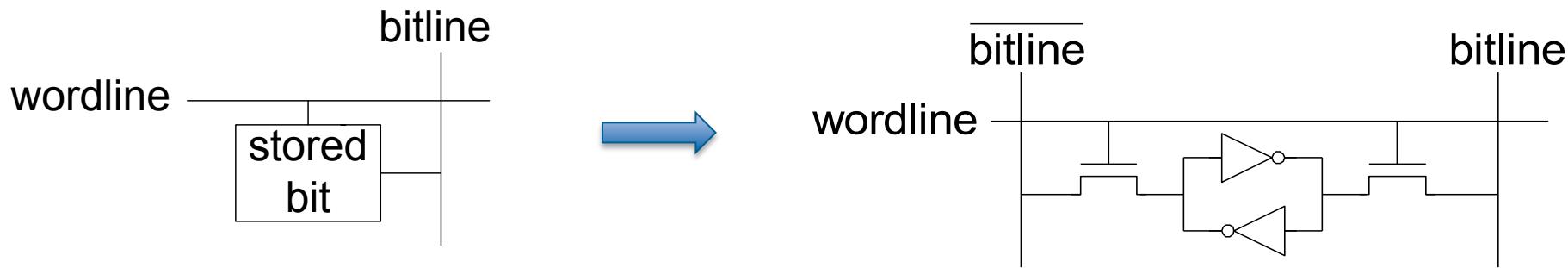


Stores 1 bit of state in the state variable, Q (or \bar{Q})

No refresh is required



Wordline and bitline allow access to the stored bit



Both switches turn on when wordline is asserted

Stored bit can be transferred to or from bitline or bitline

Larger storage cells can be built using these single-bit devices

Flip-flops require more transistors to build

Cost, power and area consumed increases with more transistors

Memory Type	Transistors per bit	Latency
Flip-flop	20 or more	fast
SRAM	6	medium
DRAM	1	slow

Latency = time required to perform a read or write

Throughput = number of bits that can be accessed per unit time

Flip-flops have very short latencies and high throughput

SRAM has higher throughput and lower latency than DRAM

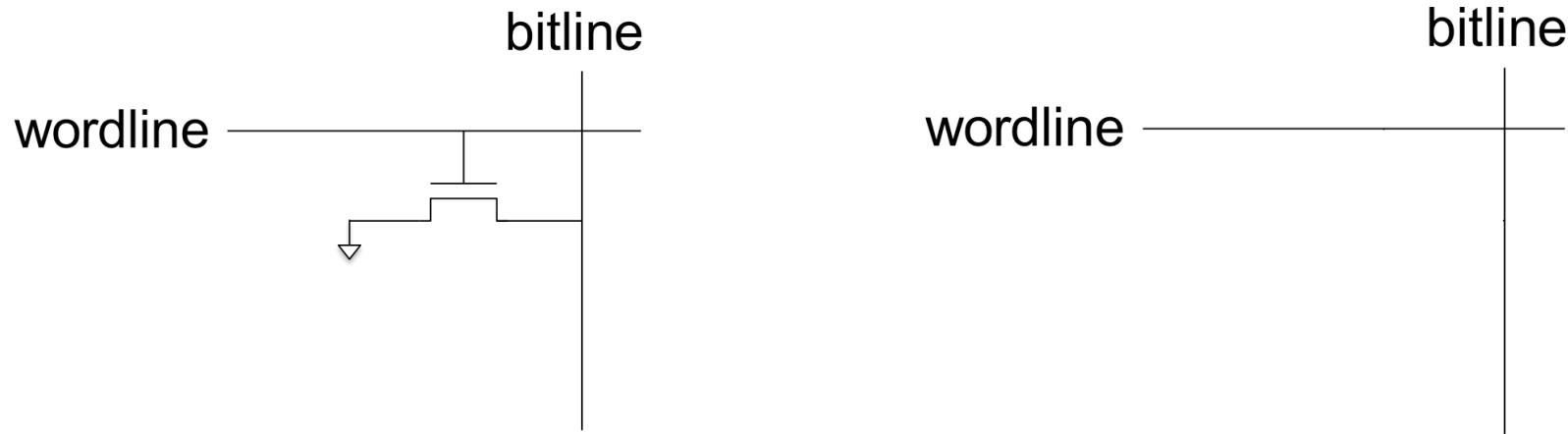
DRAM must wait for charge to move to bitline

DRAM must be refreshed periodically and after each read

In general, latency increases with larger memory sizes



Bits are stored as the presence or absence of a transistor
Bitline is weakly pulled HIGH, then wordline is turned on



Transistor pulls bitline LOW if present (represents 0)
If transistor is absent, bitline remains HIGH (represents 1)
Nonvolatile, does not change if power is turned off



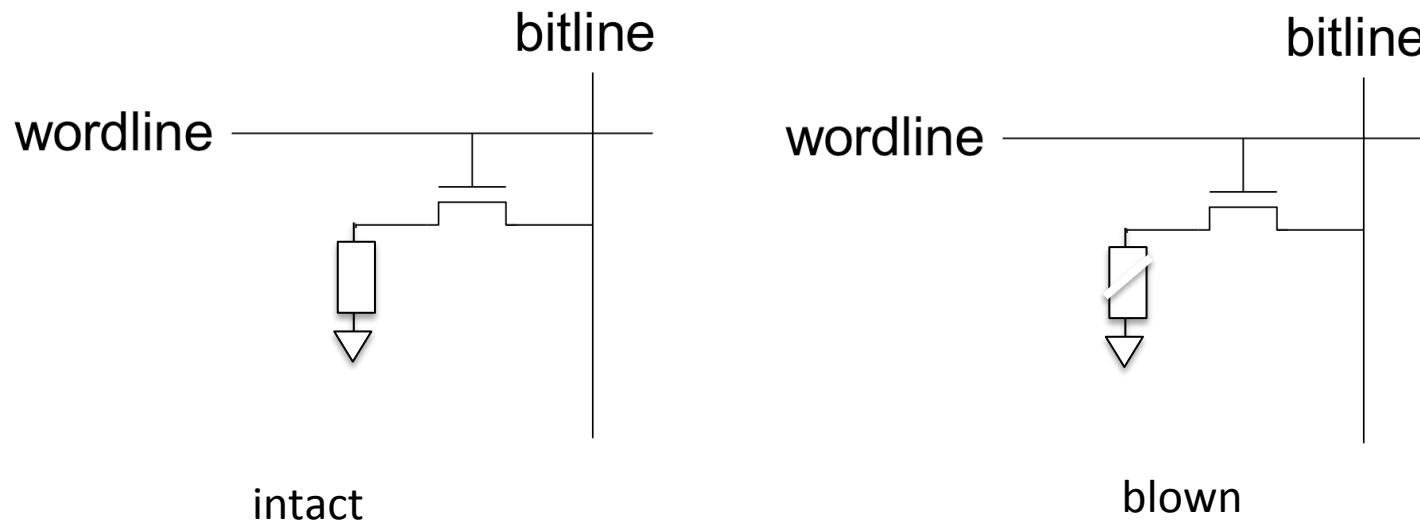
Contents of ROM bit cells can be set during manufacturing by including or omitting a transistor in various cells

These are sometimes called “masked” ROMs

PROMs (programmable ROM) have a transistor in every bit cell provides a way to connect or disconnect each transistor to ground



High voltage is applied to selectively blow the fuse links
Bitline is pulled low if link is in place, otherwise it is high



Transistor pulls bitline LOW if fuse is present (represents 0)
blown fuse disconnects transistor from ground (bitline=1)
These are sometimes called “*one-time programmable*”



Some types of PROM are reprogrammable

Transistors can be reversibly connected or disconnected to ground

Erasable PROMs (EPROM) use floating-gate transistors

Electron tunnelling turns on the transistors when voltage is applied

Exposure to UV light is used to erase the PROMs

This requires removing the memory chip from its socket

Electrically erasable PROMs (EEPROMs) are erased in place

EEPROM includes on-chip erasure circuitry

EEPROM bit cells are individually erasable

Flash memory is similar to EEPROM

Flash memory erases larger blocks rather than individual cells

Fewer erasing circuits are required for Flash

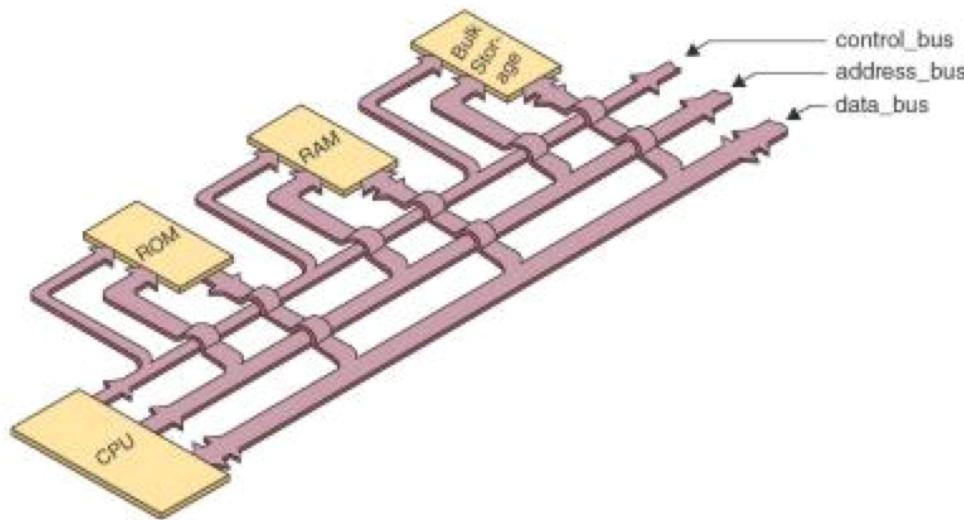
This makes Flash less expensive than EEPROM

Flash is a popular way of storing large amounts of data

Used in portable battery-powered cameras and music players

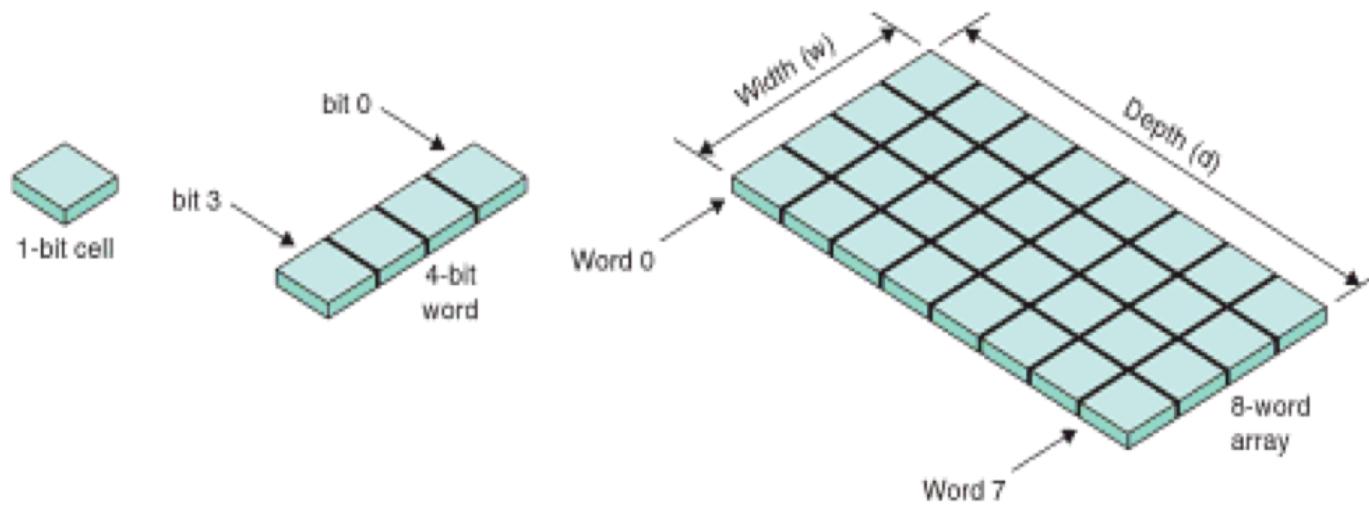
The various types of ROM take longer to write than does RAM

- The CPU obtains instructions and data from memory
 - Sends address over address bus
 - Sends or receives data over data bus
 - Sends read/write and other control signals over control bus



- A bus is a set of wires or electrical traces

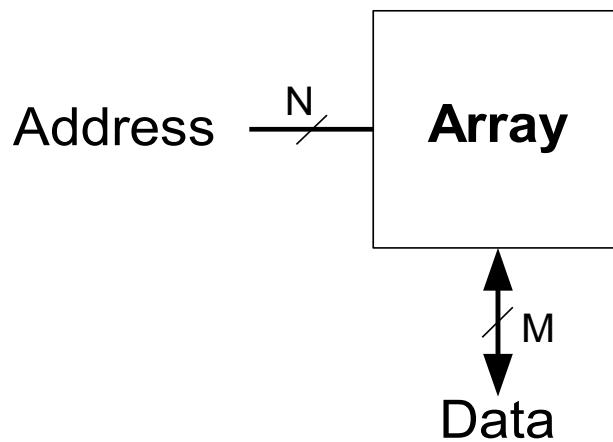
- A memory word is a group of 1-bit storage cells
- Each word has a unique address (from 0 up to some maximum)



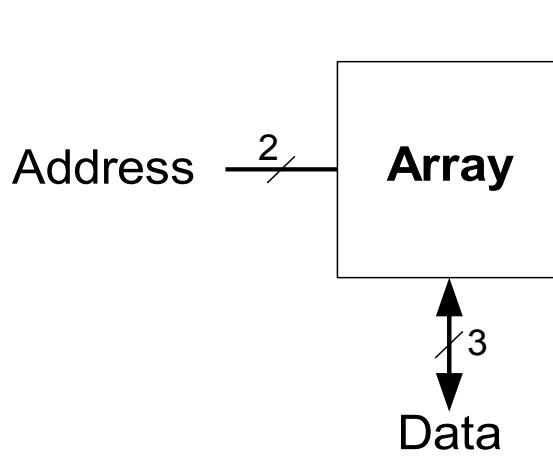
- Number of bits per word defines its width
- Number of words in memory device defines its depth or height



- The previous diagram shows a memory array
- The array contains 3-bit rows (words)
- In general, the array size = $2^N \times M$
 - N is the number of bits in the address
 - M is the number of bits per word



- $2^2 \times 3$ -bit array
- Number of words: 4
- Word size: 3-bits
- For example, the 3-bit word stored at address 10 is 100



	Address	Data
11		0 1 0
10		1 0 0
01		1 1 0
00		0 1 1

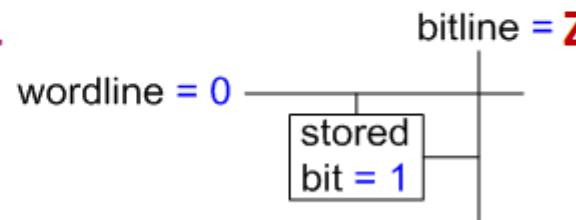
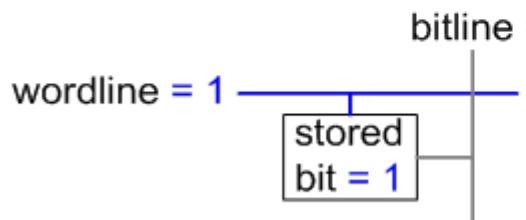
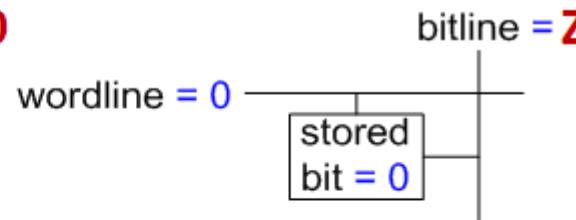
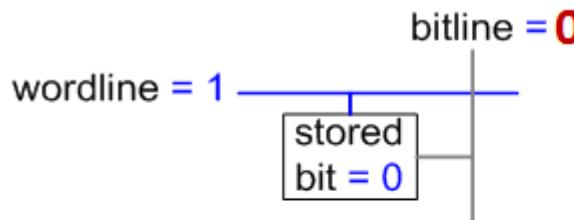
The table illustrates a 4x3 memory array. The columns represent the data width (3 bits), and the rows represent the address depth (4 addresses: 00, 01, 10, 11). The data values are: Address 00 contains 011; Address 01 contains 110; Address 10 contains 100; and Address 11 contains 010. A double-headed arrow below the table is labeled "width", and a double-headed arrow to the right is labeled "depth".

Wordline selects all bit cells within the specified word

Stored bit is transferred to or from bitline if wordline=1

Bits cells disconnected from bitline if wordline=0

Z represents the disconnected state

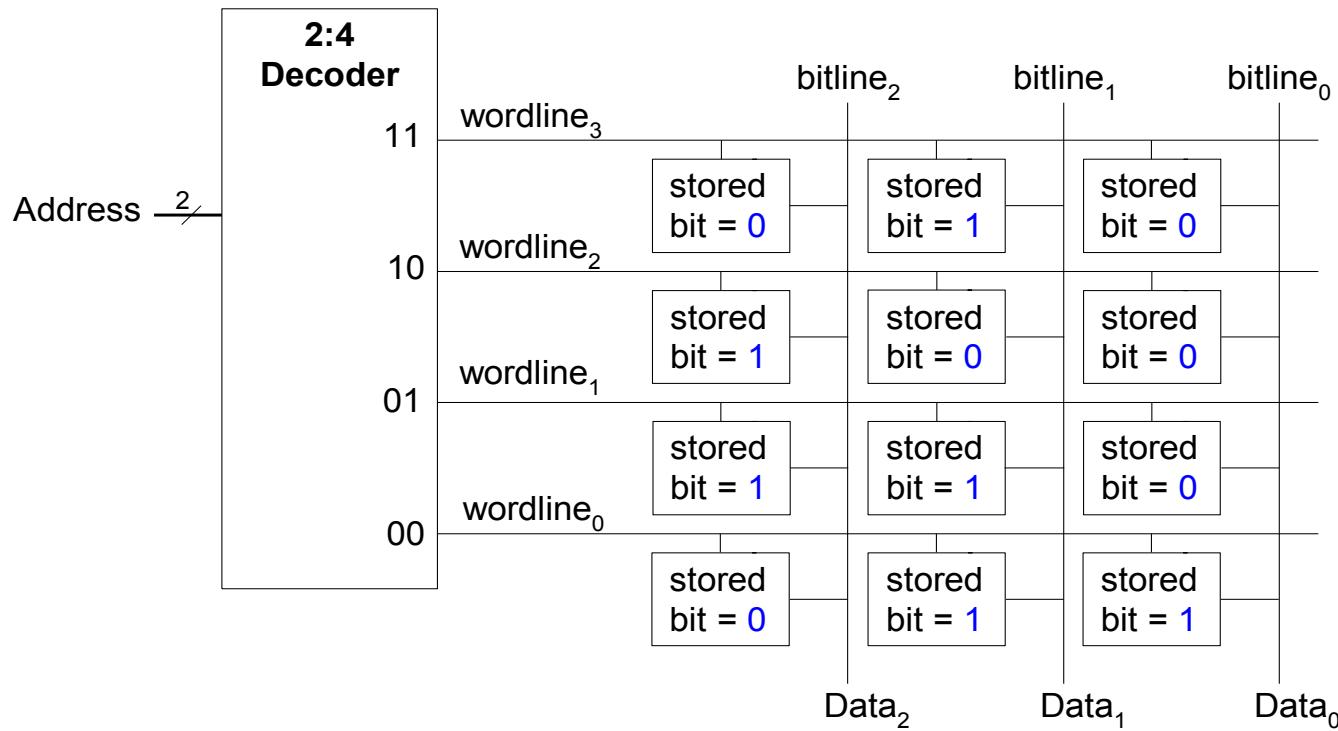


(a)

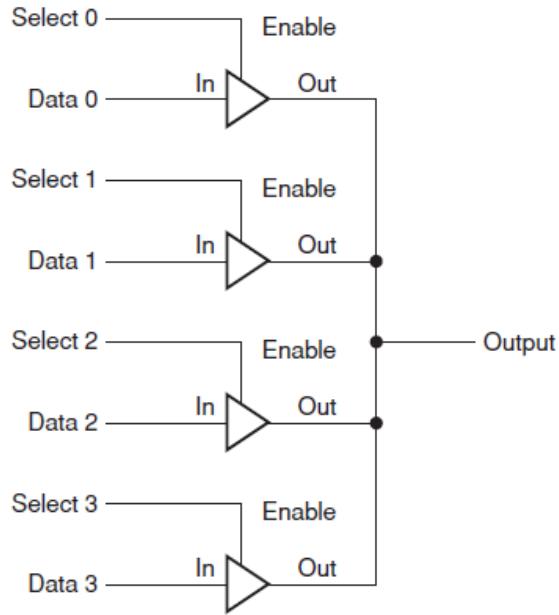
(b)

- **Wordline:**

- like an enable
- single row in memory array read/written
- corresponds to unique address
- only one wordline HIGH at once

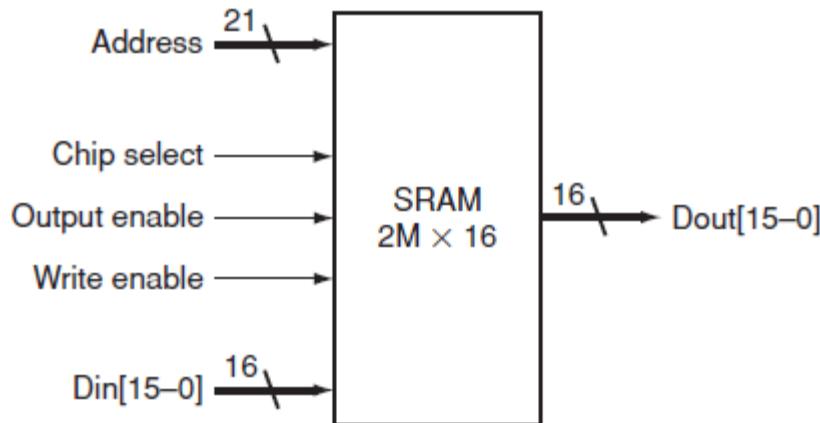


The unselected words are detached from the bitlines



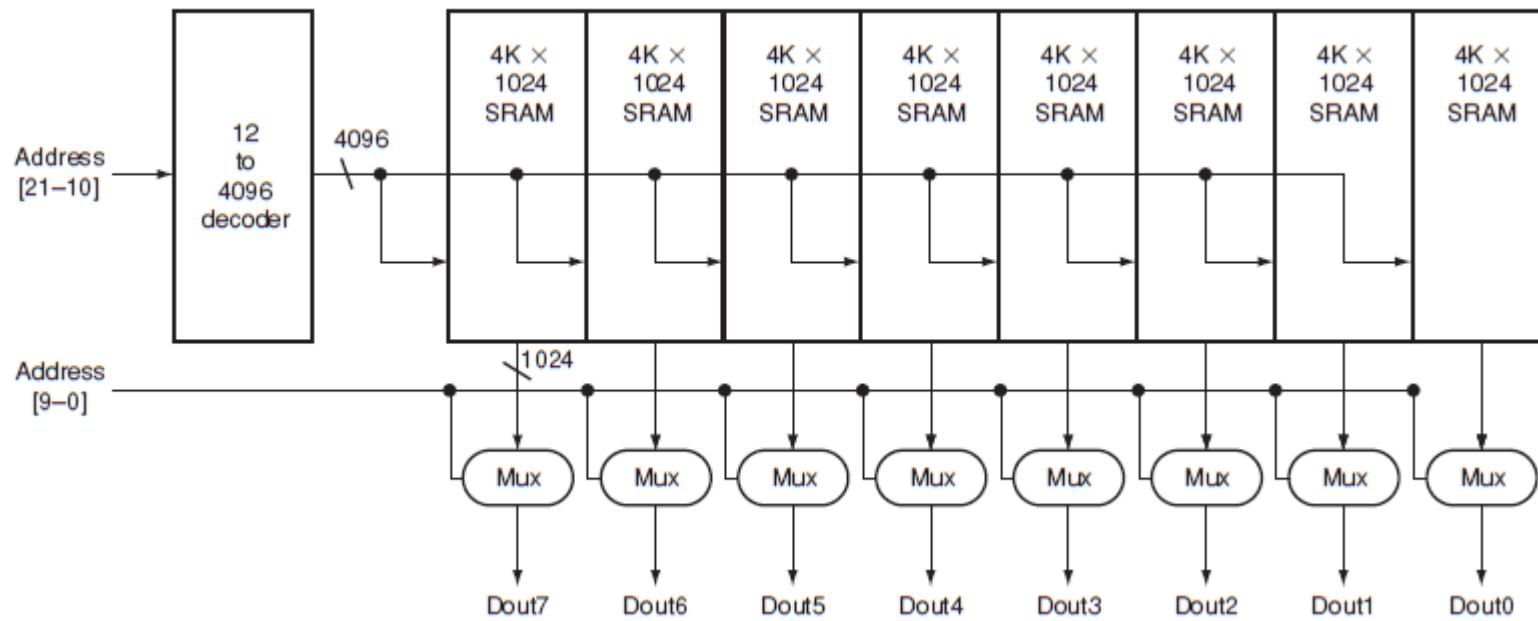
When enabled the tri-state buffer passes the data input through
When disabled, the tri-state buffer has a high impedance output
The output is thus disconnected from the bitline

- Each memory array is contained in a separate chip
- Multiple chips are grouped to provide the desired memory size
- Chip select signals determine which chip to access
- The read/write enables determine the direction of data transfer
- The address determines which word or row to access
- Write data goes through data-in port
- Read data is copied out through data-out port
- Each chip performs only one read or write at a time



- $2M (2^{21})$ words, each 16 bits (2 bytes) wide
- Total size = $2097152 * 2 = 4194304$ bytes
- A decoder could map the address onto the proper wordline

- A 21-to-2097152 address decoder is needed in the previous case
- A more practical approach uses a 2-step decoding scheme

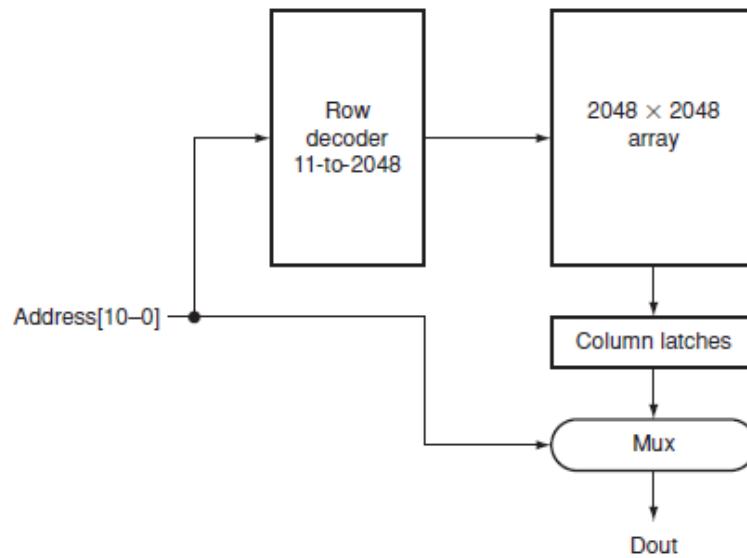


- 12-to-4096 decoder selects same row in every array
- 1024-to-1 Mux selects bit from each of the columns



- DRAMs use a two-level decoding scheme to save on cost
- Same lines carry the row address and later the column address
- Row address determines which wordline is asserted
- Column address selects the data from the column latches
- Uses Row address (RAS) and Column address (CAS) strobes
- Refresh reads columns into latches and rewrites same values
- Entire row is refreshed in one cycle
- Memory controller handles refresh independently of CPU

4M x 1 DRAM



11-bit row address selects one of 2048 rows (RAS)
Entire 2048-bit wide row is read into column latches
11-bit column address then selects one of 2048 latches (CAS)



SDRAM and SSRAM transfer data in bursts

Burst is defined by starting address and a length

A clock is used to transfer successive bits in the burst

Multiple transfers occur without having to update the address

Significantly improves the rate of data transfer

SDRAM is the most popular choice for central memory

DDRAM stands for double data rate RAMs (DDR)

DDRAM transfers data on both the rising and falling clock edge

DDR was standardized in 2000 and ran at 100 to 200 MHz

Later standards use increasingly higher speeds (≥ 1 GHz)

DDR2, DDR3 and DDR4

A single memory module can perform 1 read or 1 write at a time

This limits performance, especially for narrow widths

Interleaving refers to how consecutive locations are distributed

High order interleaving

consecutive locations are in the same module

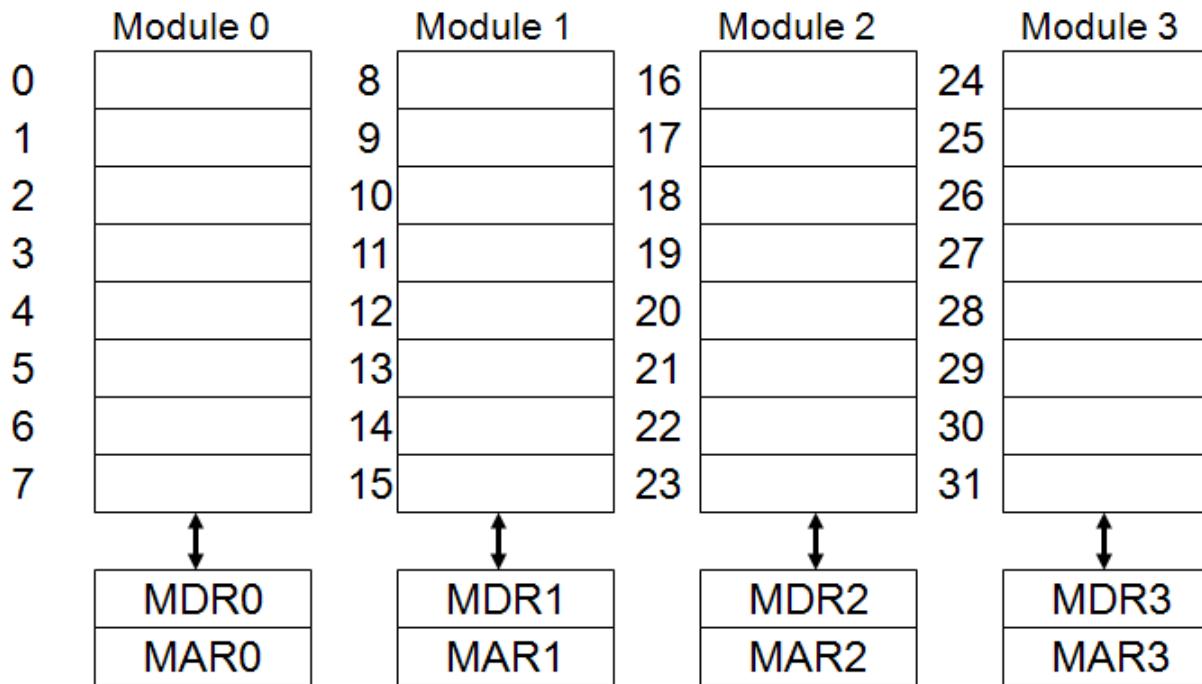
the high order address bits indicate the module number

Module number

Offset within module

Consecutive locations must be accessed sequentially

Example: if each module is 8 bits wide, reading a 4-byte word could require 4 separate reads from the same module.



MAR is memory address register.

MDR is memory data register.

Low order interleaving

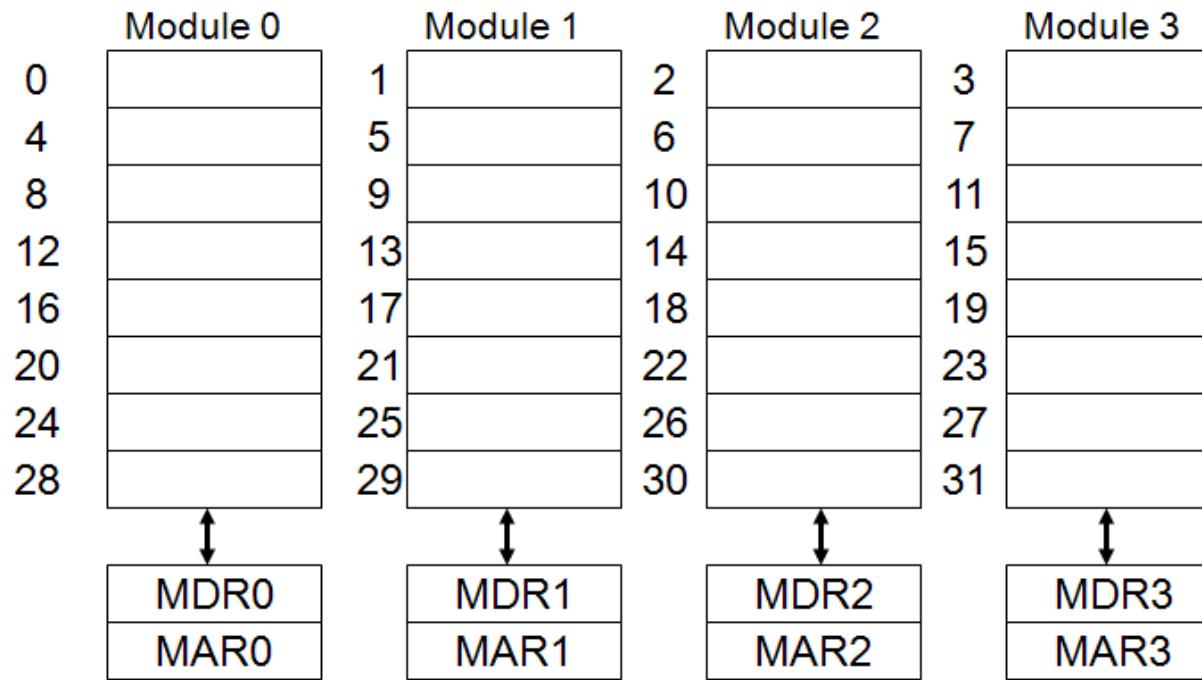
- consecutive locations are in different modules
- low order address bits indicate the module number
- high order address bits give offset into module



Consecutive locations can be accessed in parallel

High performance systems favor low order interleaving

Low order interleaving example:



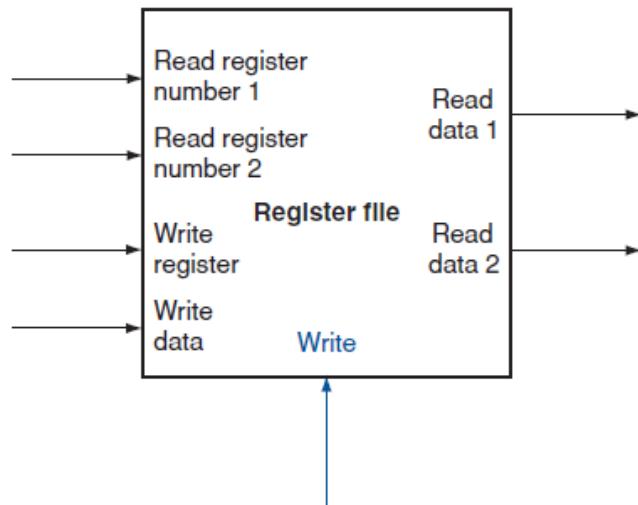
Bytes within each word have the same offset

All four modules are read in parallel to obtain the word

A set of registers each of which can be specified by a number

MIPS register file has two read ports and one write port

D flip-flops are used to construct the registers



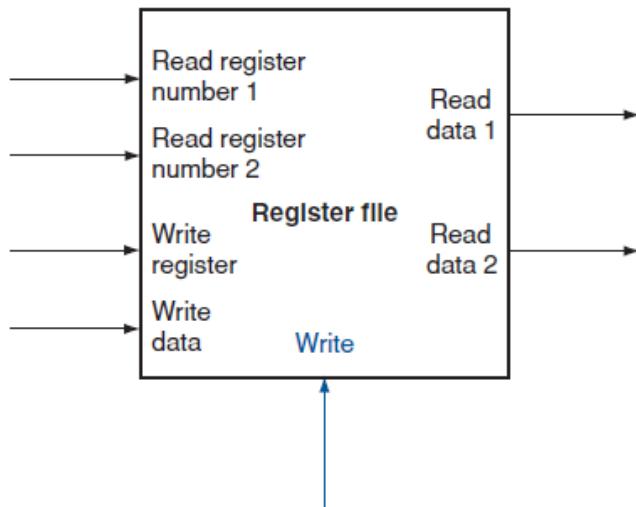
Write control line causes write data to be placed into write register

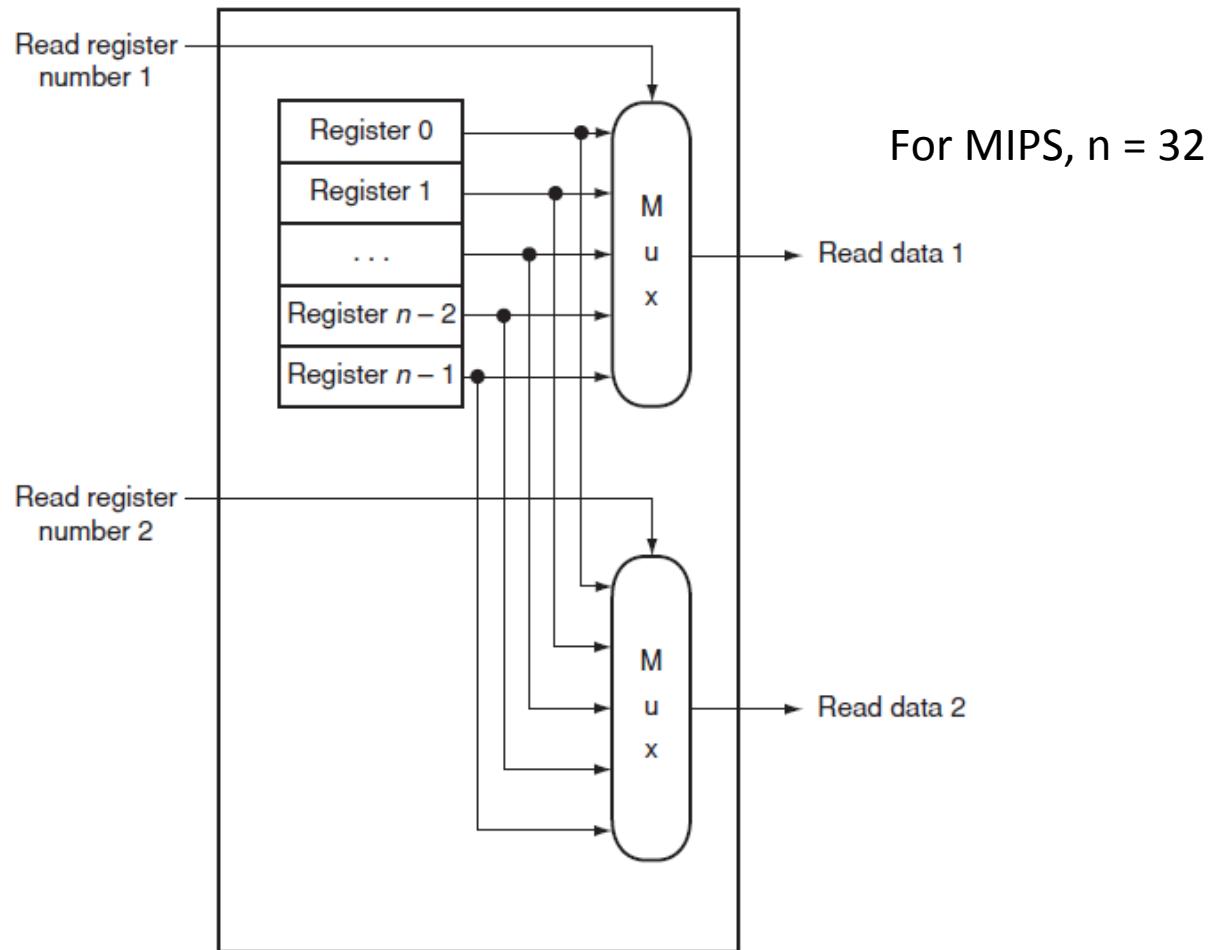
5-bit read reg. number 1 identifies the rs register

5-bit read reg. number 2 identifies the rt register

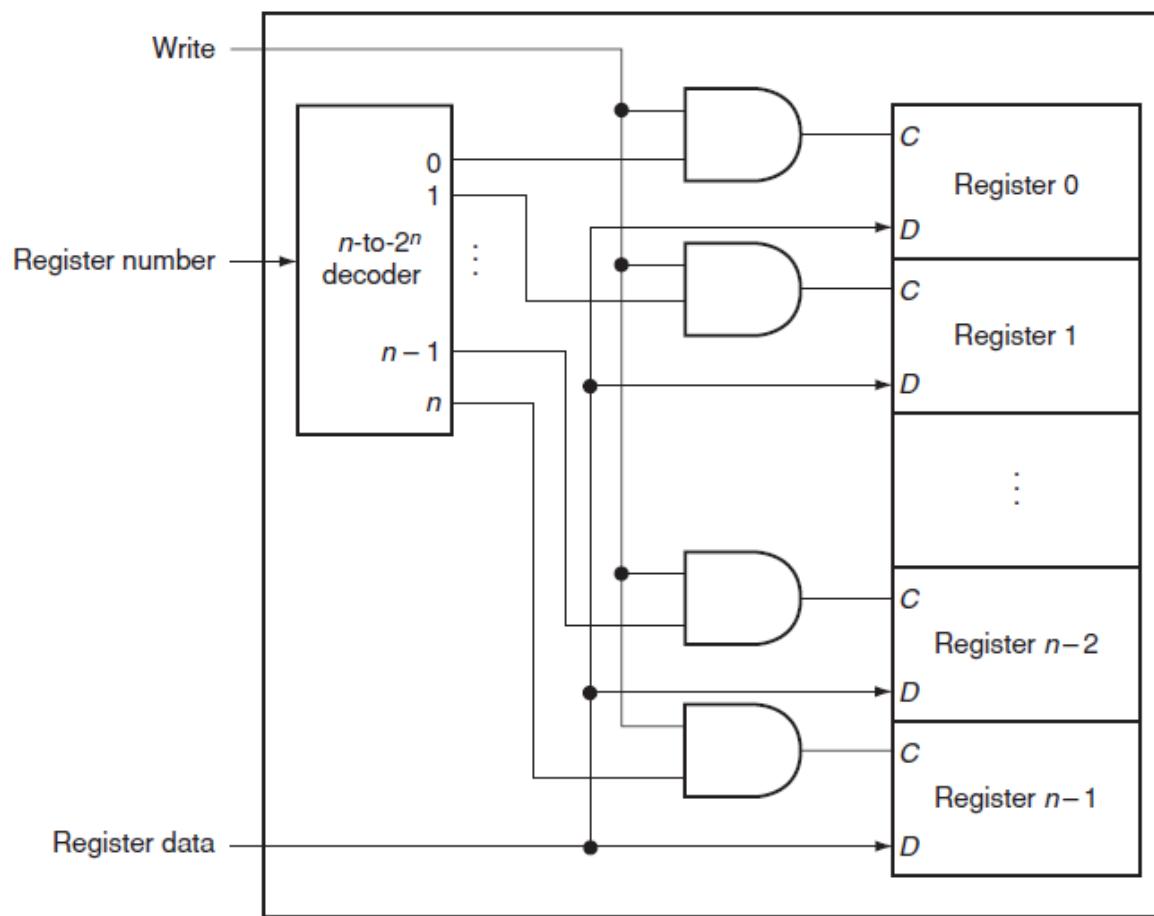
5-bit write reg. identifies register to write (rd or rt)

Read data 1, read data 2 and write data are each 32-bit values





Register read number controls which input the Mux allows to pass through



For MIPS, $n = 32$

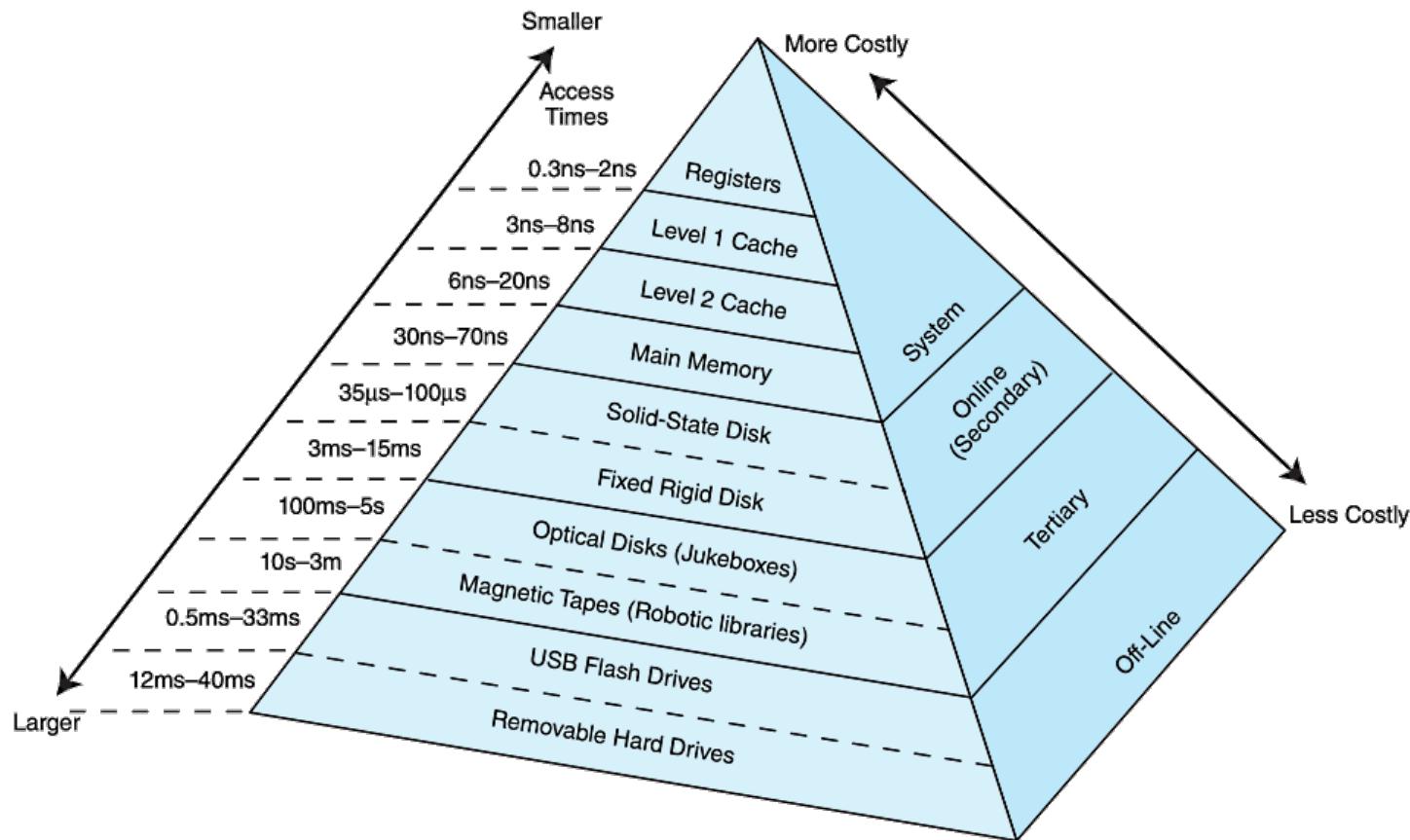
Writes to register 0 have no effect

5-to-32 decoder selects register to be written

Data input supplies value to write into selected register

- Most programs exhibit a property known as “locality”
 - Addresses referenced tend to cluster
- Addresses within a relatively small area are reused
 - repetitive loops
 - fairly small functions
 - access to items within arrays and structures
- *Spatial locality*
 - the use of limited areas within the address space
 - these areas change or migrate slowly over time

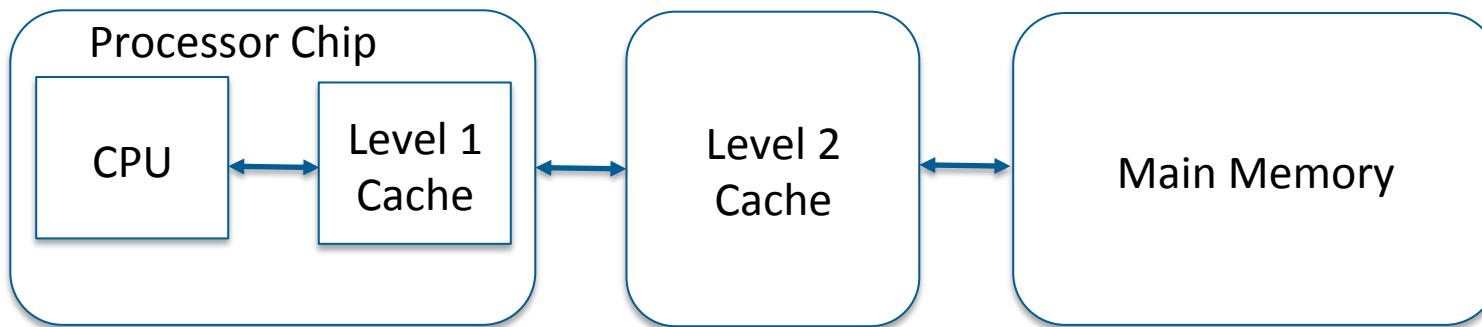
- *Sequential locality*
 - Instructions tend to be accessed sequentially
 - The PC is incremented to point to the next instruction
 - Branches and function calls alter this sequential flow
- *Temporal locality*
 - Reuse of items recently accessed
 - Previously executed instructions
 - Previously accessed data operands
- Locality can be exploited to improve performance



Keep recently used items at the higher levels within the memory hierarchy

- Use registers for operands
 - RISC systems tend to have more registers
 - Registers are within the CPU
- Use a high speed “cache” memory
 - Constructed from SRAM
 - Much faster than DRAM (central memory)
 - Holds instructions and data
- Use multiple cache levels
 - Accommodates more instructions and data
 - Reduces frequency of access to central memory

- L1 or primary cache is on the same chip as the CPU
 - Takes longer to access than registers
 - Much shorter access time than main memory
- L2 or secondary cache is not on same chip as CPU
 - Takes longer to access than L1 cache
 - Much larger size than L1 cache



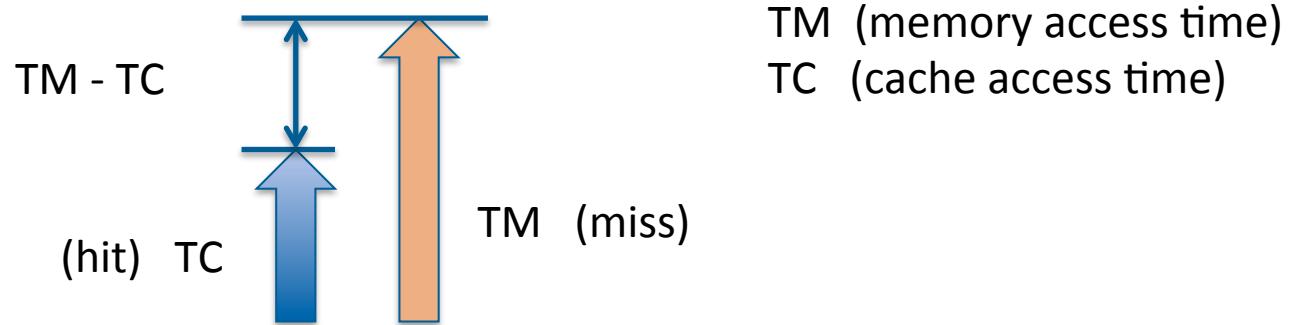
- L1 is checked first, then L2, then main memory

- Operands not in registers should be kept in cache
- Block containing a desired item is loaded into cache
 - This takes advantage of spatial locality
 - Blocks may contain hundreds of bytes
- Blocks are copied from main memory into L2 cache
- L1 contains a subset of the information in L2
- Main or central memory is accessed as a last resort
 - The extra time needed to access memory is a penalty

- Cache related terms:
 - A *hit* is when data is found at a given memory level
 - A *miss* is when it is not found
 - The *hit rate* = % of references found at a given memory level
 - Hit ratio = (Number of hits) / (total number of references)
 - The *miss rate* = % of references not found at that level
 - Miss ratio = 1 - hit ratio

- Cache related terms:
 - *hit time* = time needed to access data at a given memory level
 - The *miss penalty* = time required to process a miss
 - includes block load and/or replacement time
 - includes time it takes to extract and deliver the item
 - The *effective access time* = average access time per reference
 - Depends on hit ratio, hit time and miss penalty
 - *Cache line* holds a block read in from main memory
 - Line size and block size are the same

- These allow overlapped accesses
- Accesses to cache and to memory occur in parallel
- Memory access is cancelled if hit in cache occurs
- Tends to lower the effective access time
- Increases CPU-to-memory traffic



- Consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit ratio of 99%.
- Suppose access to cache and main memory occurs concurrently. (The accesses overlap.)
- The EAT is:

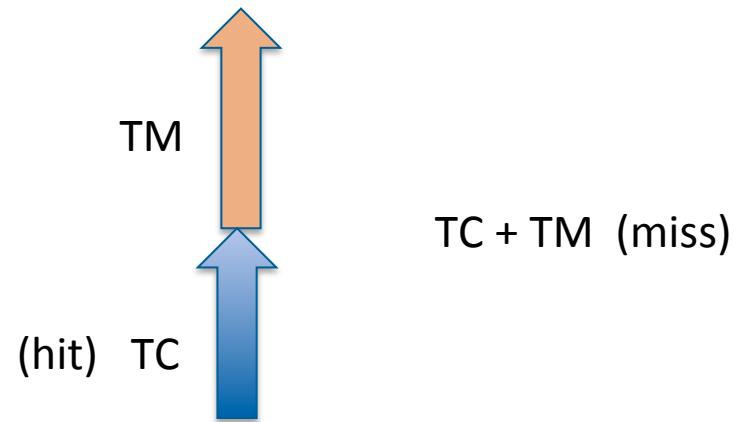
$$0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}$$

$$\text{EAT} = \text{hit ratio} * \text{TC} + (1 - \text{hit ratio}) * \text{TM}$$

- With these, the cache is checked first
- Next level is only checked if miss occurs
- Tends to increase the effective access time
- Avoids unneeded CPU-to-memory traffic

$$EAT = h * TC + (1 - h) * (TC + TM)$$

where h is the hit ratio



- Consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%
- If the accesses do not overlap, the EAT is:

$$\begin{aligned}0.99(10\text{ns}) + 0.01(10\text{ns} + 200\text{ns}) \\= 9.9\text{ns} + 2.01\text{ns} = 12\text{ns}\end{aligned}$$

- This equation for determining the effective access time can be extended to any number of memory levels

- Caching is beneficial if programs exhibit good locality
 - Some object-oriented programs have poor locality owing to their complex, dynamic structures
 - Arrays stored in column-major rather than row-major order can be problematic for certain cache organizations
- With poor locality, caching can actually cause performance degradation rather than performance improvement

- Write Policy must be considered as well
 - Two types: *write through* and *write back* (or *copy back*)
- Write through updates cache and main memory
 - Memory is updated in parallel with cache for every write
- Write back only updates the cache copy of the item
 - Main memory is updated when item is removed from cache
 - Must record whether cache line was written (*dirty*)
 - May cause issues with concurrent access
 - Shared memory multiprocessors
 - I/O devices using DMA (direct memory access)

- Cache contains only a fixed number of lines
- Lines are replaced to make room for new blocks
 - Required if all lines are already occupied
- Replacement Policy selects “victim” to be evicted
 - Dirty lines must be written back to memory
- Possible replacement policies
 - Random
 - FIFO (oldest)
 - LRU (least recently used)

- *Unified* or *integrated* caches contain instructions & data
- The alternative is a *split cache* system
 - Instruction (I-cache) and data (D-cache) are separate
 - This is called a *Harvard* cache
 - Allows instruction fetch in parallel with data access
- The separation of data from instructions provides better locality, at the cost of greater complexity
 - Simply making the cache larger provides about the same performance improvement without the complexity.

- Many modern systems use multiple levels of cache
- L2 is on the same die as the CPU in 3-level systems
- Level 3 cache is between the CPU and main memory
- Some systems allow multiple copies of information
 - data and instructions are in more than one cache level at a time
 - These are called inclusive cache systems
- Exclusive caches permit only one copy of data

- Compulsory Misses
 - Also called “cold start” misses
 - Unavoidable due to initially empty cache
- Capacity Misses
 - Due to limited size of cache
 - Cache too small to hold all referenced blocks
- Conflict Misses
 - Due to different blocks mapping to the same set or line

- Next we will consider cache mapping schemes
 - Determines the line into which to load a block
 - Depends on the cache organization
 - Affects the line replacement policy
- We will exam three organizations
 - Fully associative
 - Set associative
 - Direct mapped

Cache makes memory appear to be faster

- It is closer to the CPU than is main memory
- It holds recently accessed instructions and data
- Cache is much smaller than main memory
- Its access time is a fraction of that of main memory
- Main memory is accessed by address
- Cache is accessed by content
 - it is often called *content addressable memory*
- A single large cache isn't always desirable–
 - it takes longer to search.

- We will exam three cache organizations:
 - Fully associative
 - Set associative
 - Direct mapped
- Each is based on the idea of memory blocks
 - A program's address space is subdivided into blocks
 - Each block is like an element in an array
 - Each is assigned a unique block number (B)
 - The block address = $B * \text{block size}$
(e.g. if block size = 256, block 0 begins at address 0,
block1 begins at address 256, etc.)

- Addresses of items map to blocks
- Address can be interpreted as:



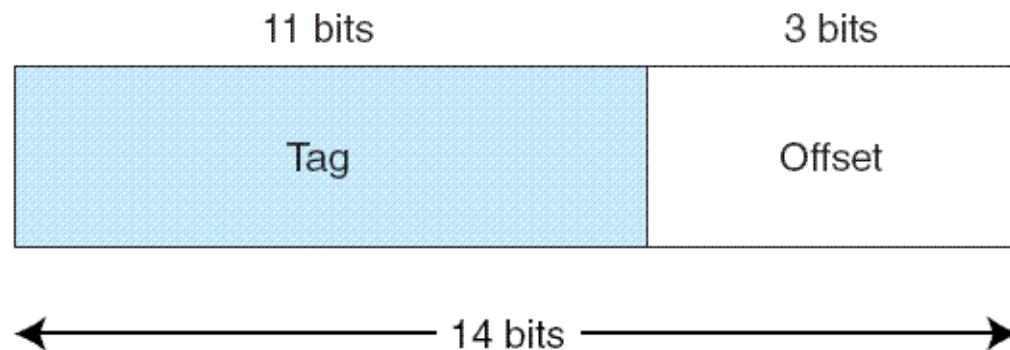
- Block number = address / block size
- Number of bits in offset = \log_2 block size
- Example: for 256-byte blocks, the address 0x0400ACE8 falls within block:
 $0x0400ACE8 / 256 = 0x00400AC$
Offset width = $\log_2 256 = 8$ bits

- Cache can hold some number of blocks
 - The blocks within the cache are called lines
 - Each line is the same size as a memory block
 - A line can hold one or more blocks
 - Mapping determines which block resides in which line
- Number of lines is much less than number of blocks
 - $(\text{Memory size}) / (\text{block size}) = \text{number of blocks}$
- Cache size determines the number of lines
 - Example: If cache size = 8 KB and line size = 256 bytes, then number of lines = $8 \text{ KB} / 256 = 32$

- A block is the basic unit of transfer between memory and cache
- A miss results in copying the entire block to cache
 - Takes advantage of spatial locality
- Cache is organized into some number of sets
- *Associativity* is the number of blocks per set
 - N-way associative means there are N blocks per set

- Fully Associative is the easiest to describe
- Any memory block can go into any empty cache line
- All lines are checked to detect cache hits or misses
- Requires hardware to compare all lines in parallel
 - A valid bit for each line is set if the line is occupied
 - Block loaded into vacant cache line if miss occurs
 - If no line is vacant, a replacement must occur

- Mapping is based on dividing address into 2 fields:
 - Tag field and offset

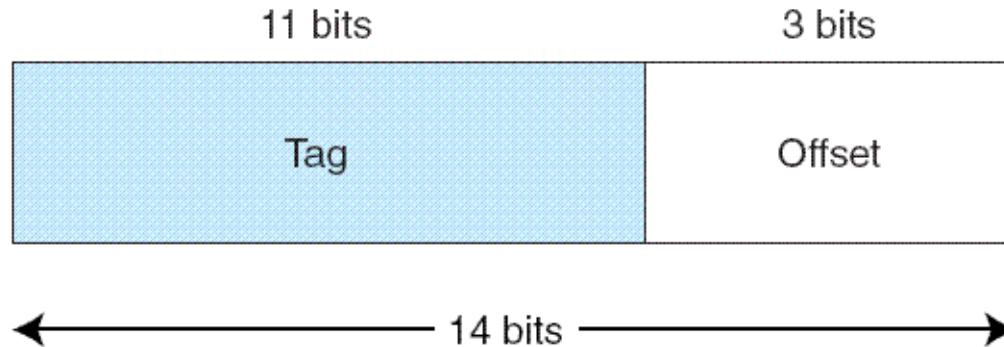


Example: 14-bit addresses and a cache containing 16 lines. Each line is 8 bytes in size.

A separate tag is stored for each cache line

If a stored tag matches the tag field in the address and the line is valid, there is a hit.

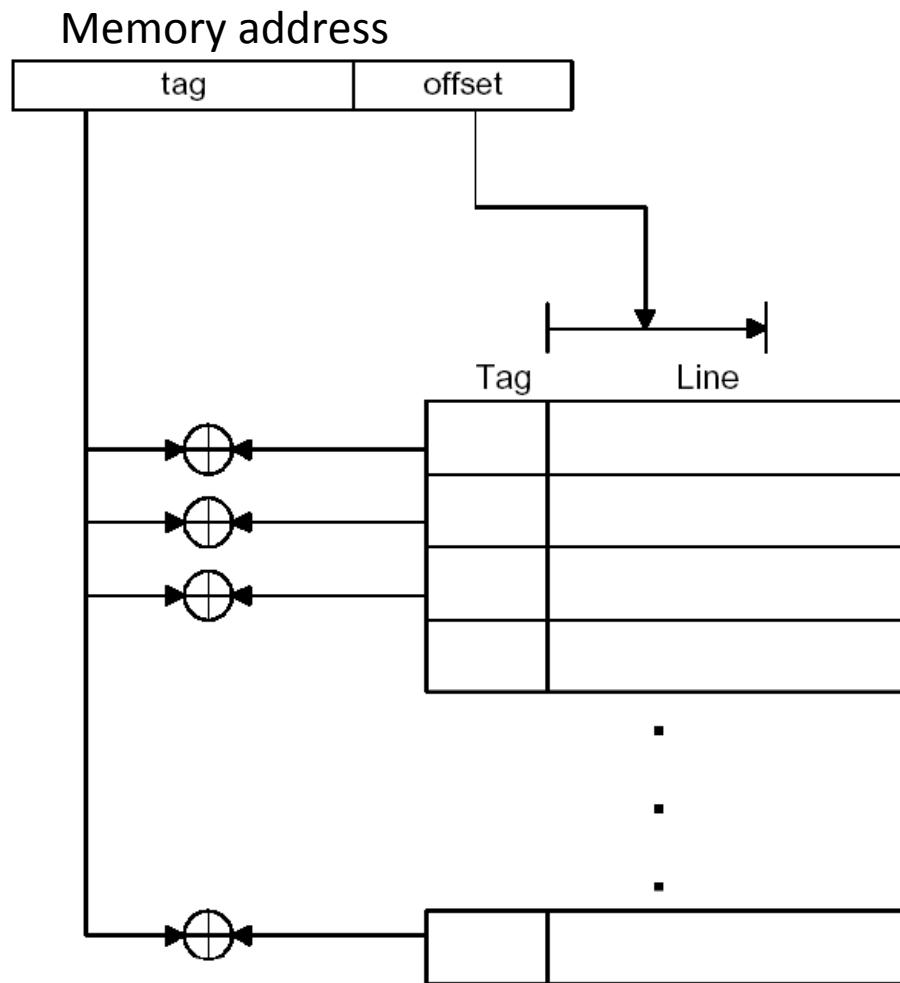
A separate tag is stored for each cache line



Only lines for which valid bit is set are checked

Match between a stored tag and address tag indicates a hit

Miss means none of the stored tags match the address tag



Separate comparator is needed for each cache line

- Assume a read and a block size of 256 bytes
- Address 0x0400ACE8 is interpreted as:

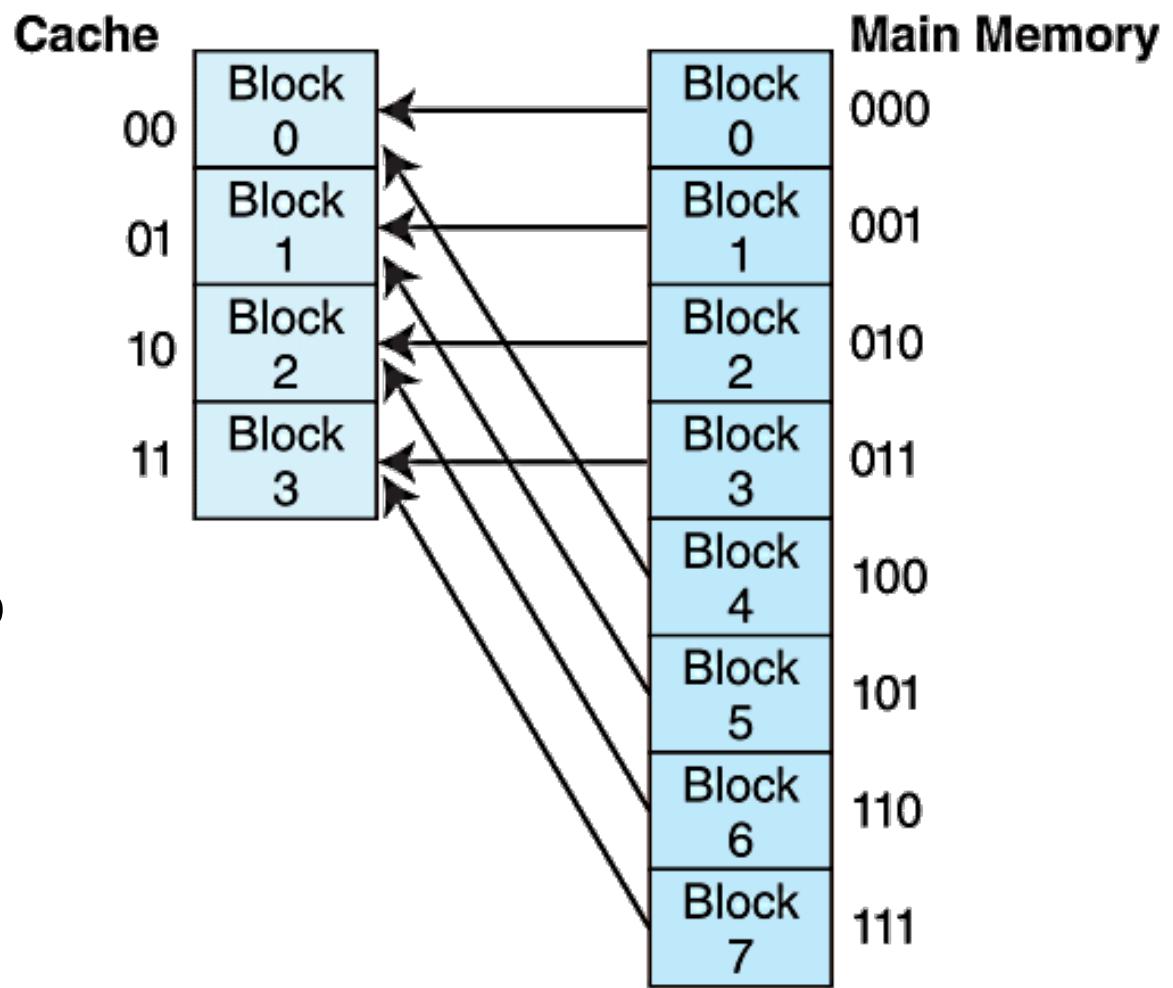


- Block number = address / block size = tag
 $0x0400ACE8 / 256 = 0x00400AC$
- a hit occurs if some cache line has a matching tag
- for a miss, the block is loaded into any empty line
- if all lines are full, a line is replaced

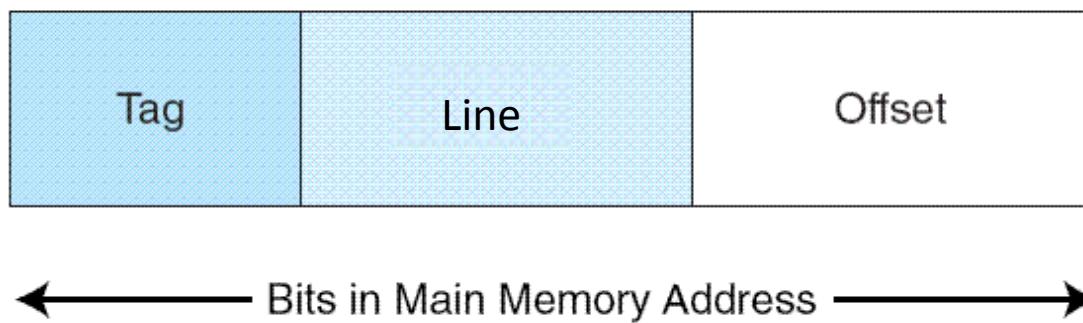
- Fully Associative is one extreme
 - A separate comparator is required for every cache line
- Direct Mapped is the other extreme
 - only one comparator is required for entire cache
- With a direct mapped cache with N lines:
 - block B of memory maps to line $L = B \bmod N$
 - B is the block number, not the address

Example: direct mapped cache with 10 lines,
Line 6 may hold blocks 6, 16, 26, 36, ...

- With direct mapped cache consisting of N lines of cache, block X of main memory maps to cache line $Y = X \bmod N$.

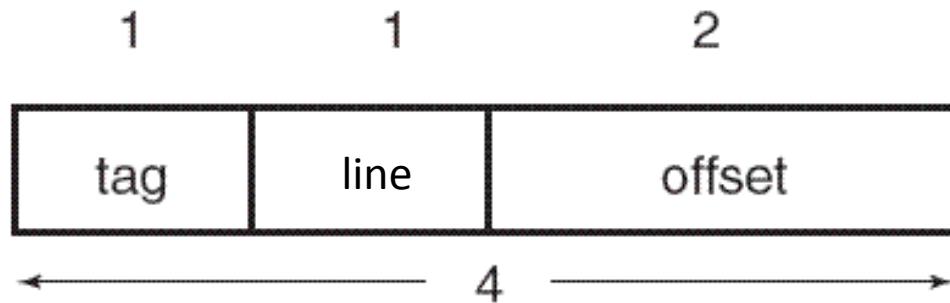


- Address is split into 3 fields for direct mapping
 - The *offset* field identifies location within line
 - The *line* field selects a unique line of cache
 - The *tag* field is whatever is left over.



- Offset width OW = \log_2 line size
- Line# width LW = \log_2 number of lines
- Tag width = (Bits in address) - LW - OW

- EXAMPLE: byte-addressable main memory consisting of 4 blocks with 4-byte block size. Cache contains 2 lines.
- Block 0 and 2 of main memory map to line 0 of cache, and Blocks 1 and 3 of main memory map to line 1 of cache.
- Address format is:

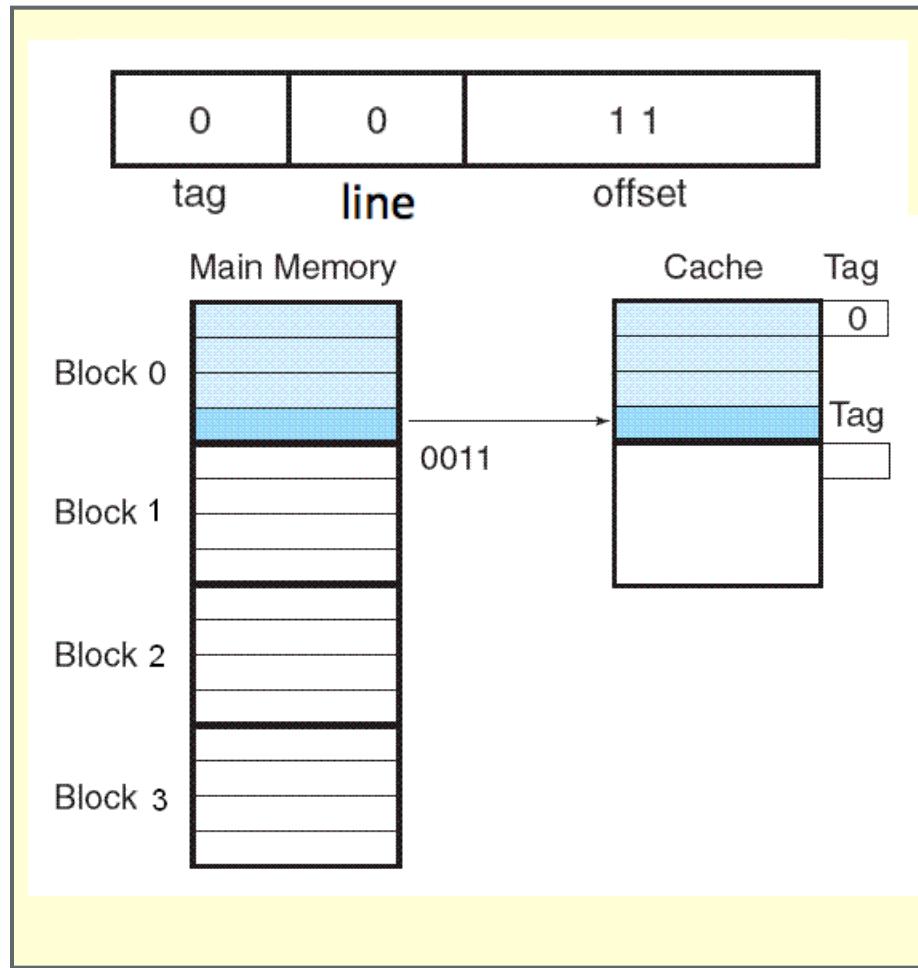


Address 3 is in memory block 0 and maps to cache line 0

$$\text{Block\#} = 3 / 4 = 0$$

$$\text{line\#} = 0 \bmod 2 = 0$$

16-byte memory
Requires 4-bit
address

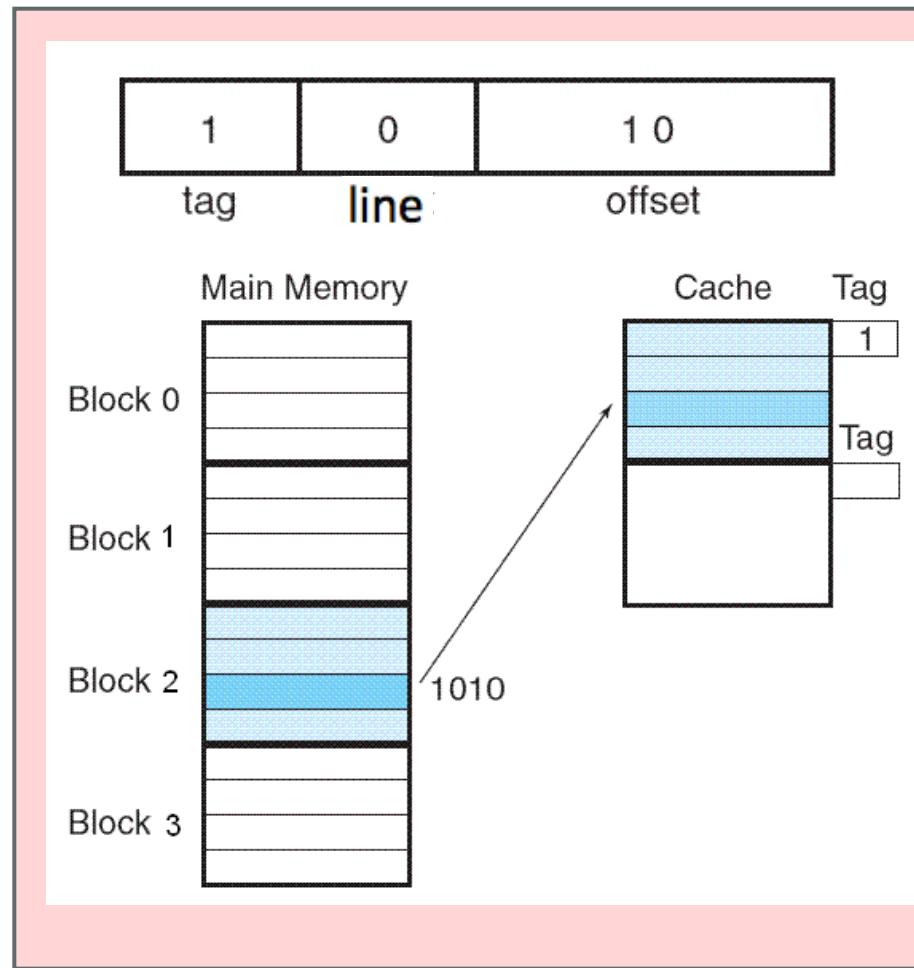


Cache has 2 lines:
line0 and line 1

Address 10 is in memory block 2 and maps to cache line 0

$$\text{Block\#} = 10 / 4 = 2$$

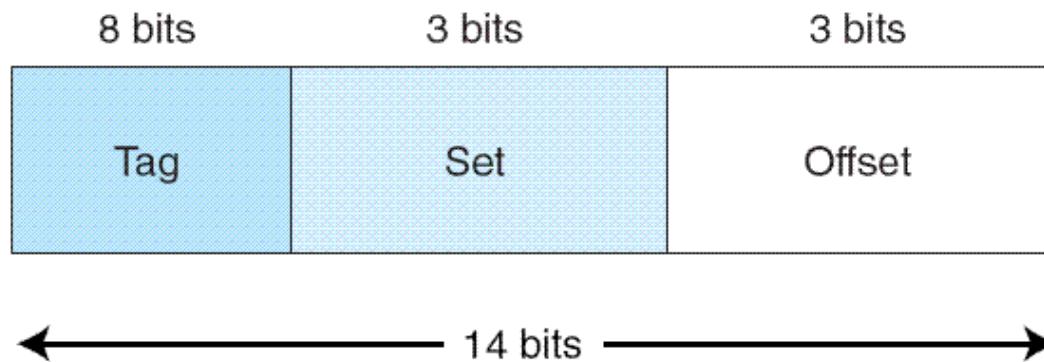
$$\text{line\#} = 2 \bmod 2 = 0$$



- Multiple addresses with same line number cause conflicts
 - cache line can hold only one candidate block
 - other vacant lines may be unused
 - causes increased miss ratio
- Can hurt performance
 - 0 hit ratio if alternating between conflicting addresses

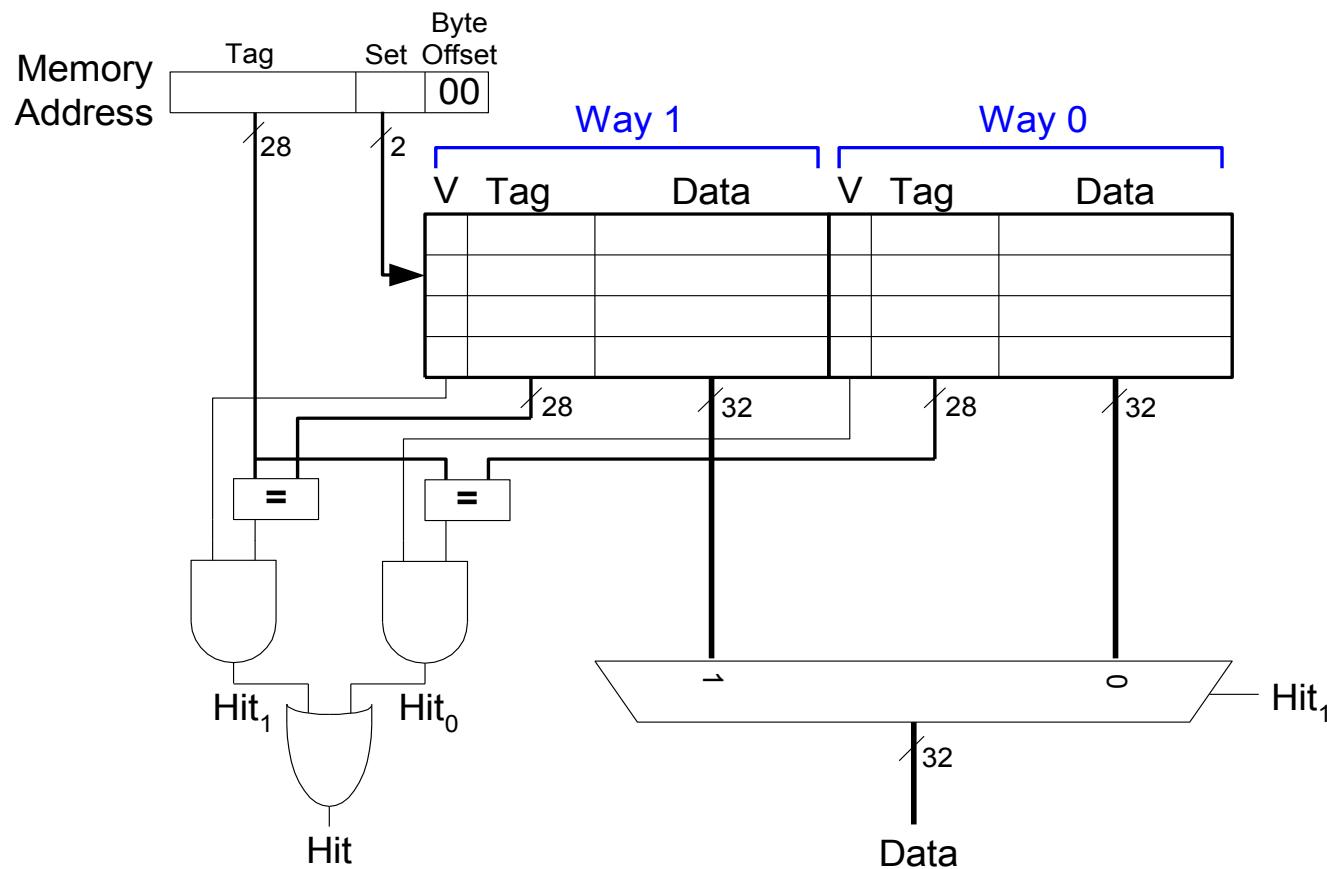
- Set associative cache mapping reduces conflicts
- Sets of blocks map to the same set of lines
- Addresses are divided into three fields:
 - tag, set, and offset
- set field determines set (group of lines) to which the memory block maps
- tag field identifies which line within set contains block
- offset field chooses the location within the line

- EXAMPLE:
- 2-way set associative cache (2 lines per set)
- 2^{14} bytes of main memory (14-bit addresses)
- Cache with 16 lines, 8 bytes per line, sets = $16/2 = 8$



- Address $0x12CF = 01001011\textcolor{red}{0}01\textcolor{green}{1}11$ as 14-bit binary
- Maps to set **1**
- Hit if tag for either line in set 1 = $01001011 = 0x4B$
- Line must be valid (i.e. contain valid data)

Example: 32-bit address, 4 sets, 2 lines per set, 4 bytes per line



- New block can be loaded into any vacant line in set
- Replacement is required if set is full
- Most common choice is LRU (least recently used)
- *Optimal* replacement policy
 - replaces line not needed in the future
- Optimal policy can't be implemented
- Used as benchmark for assessing other schemes
- Replacement policy requires extra bits for each set

- Replacement policies try to optimize temporal locality
- *First-in, first-out* (FIFO) is a popular replacement policy
 - oldest line in the set is evicted
 - ignores when it was last used
- *Random* replacement policy
 - Replaces line at random from the set
 - Can evict a line that will be needed often or soon
 - it never *thrashes*
- Trashing refers to repeatedly replacing the wrong line

- *Least recently used (LRU)* is most popular
- Keeps track of the last time a line was accessed
- Line unused for the longest period of time is evicted
- Disadvantage is its complexity
- Maintaining access history for each line, slows down the cache
- Can be approximated to save bits

- *True LRU replacement*
 - Always correctly identifies the actual least recently used line
 - Requires a number of bits per set (depends on associativity)
 - Can be excessive in terms of overhead (storage & speed)
- *Pseudo LRU replacement*
 - Identifies least recently used line most of the time
 - Can make wrong choice at times
 - requires fewer bits per set
 - Faster than true LRU

Example: Scheme used on the Pentium and other processors with 4-way associative cache

3 bits (B2 B1 B0) are stored for each set

Bits are updated based on way (line) accessed:

Way accessed	Effect on LRU bits
0	1->B1, 1->B0
1	0->B1, 1->B0
2	1->B2, 0->B0
3	0->B1, 0->B0

Reading, writing or replacing a line is considered an “access”

LRU bits are updated for each access

New block is loaded into next available line
Replacement occurs if the set is full

LRU bits select the victim when a replacement is required:

B2 B1 B0	Way to replace
000	0
001	2
010	1
011	2
100	0
101	3
110	1
111	3

Correct selection >90% of the time

Occasionally does not select the actual LRU way



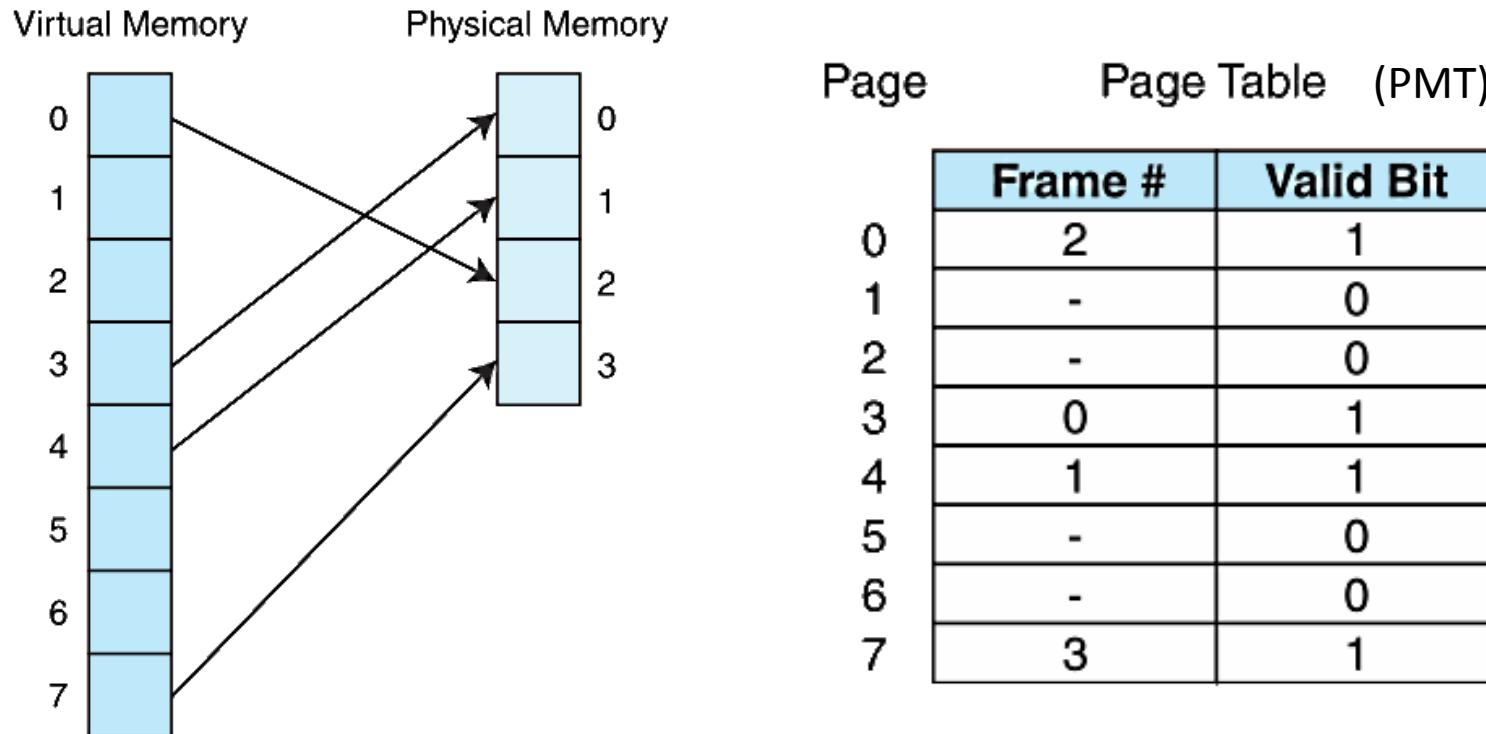
- Set associative is the most general case
 - Number of lines per set = associativity
- Direct mapped is equivalent to a 1-way set associative cache
 - Each line is a separate single-element set
- Fully associative has 1 set with N elements
 - N = number of lines in the cache

- Cache is used to make memory appear to be faster
- Virtual memory makes main memory appear larger
- Virtual memory encompasses the secondary storage
 - The currently used portion of the program is in memory
 - The remaining parts stay on disk until needed

- Most virtual memory systems employ paging
 - Programs use a logical address space
 - Logical address space is partitioned into pages
 - Physical memory is partitioned into frames (same size as page)
- Main memory and virtual memory are divided into equal sized pages
- Pages are allocated to a process or program
 - Pages do not need to be stored contiguously-- either on disk or in memory

- Addresses used by the program map to pages
 - Page Map Tables are used to do the mapping
 - References to pages not in memory cause “page faults”
- There are many more pages than frames
 - This gives the illusion of a larger memory

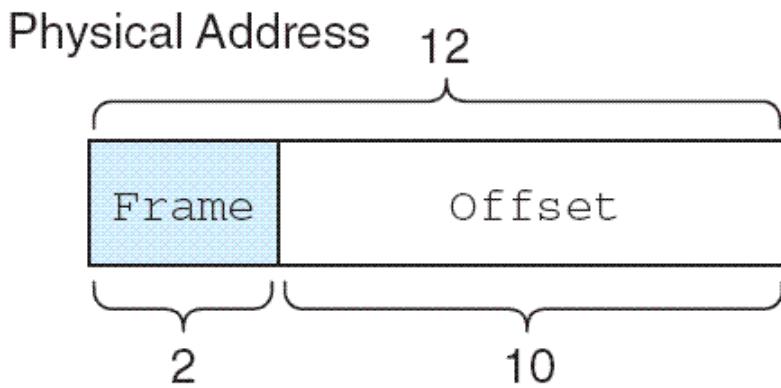
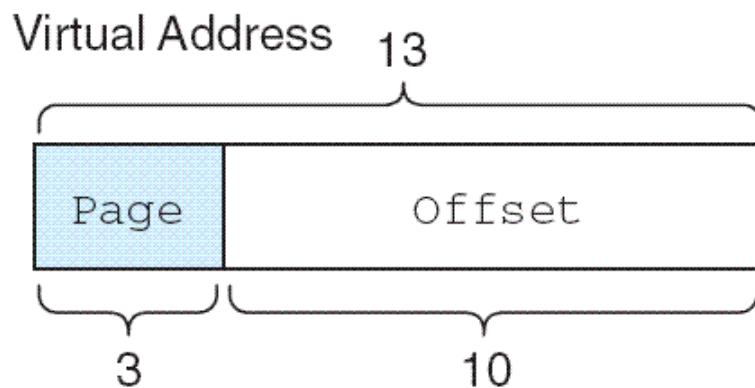
There is one page table for each active process



If valid bit=1, the page is in memory, and the page table entry (PTE) indicates which frame contains the page

- Operating systems translate virtual addresses
- A virtual address is divided into two fields:
 - *page number* field, and
 - *offset* field
- page number indexes into the PMT to select PTE
 - If valid bit = 0 within PTE, we have a page fault
 - If valid bit = 1, the frame number in PTE is read
 - Frame number concatenated with offset = physical address

- Example:
- a byte addressable system has an 8K virtual address space, a 4K physical address space with 1K frames
- We have $2^{13}/2^{10} = 2^3 = 8$ virtual pages
- A virtual address has 13 bits ($8K = 2^{13}$) with 3 bits for the page field and 10 bits for the page offset
- A 4K physical memory address requires 12 bits, the first two bits for the frame and the trailing 10 bits for the offset

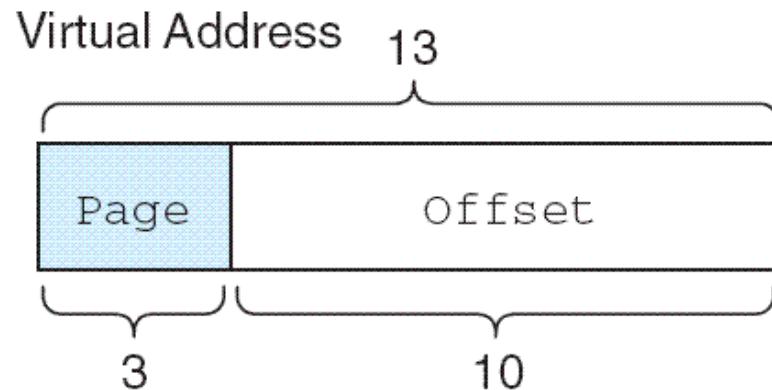


- Assuming the PMT contents shown below, what happens for CPU address $5459_{10} = 1010101010011_2 = 0x1553$?

Page = 5

Page	Frame	Valid Bit
0	-	0
1	3	1
2	0	1
3	-	0
4	-	0
5	1	1
6	2	1
7	-	0

Page	Addresses		
	Base 10	Base 16	
0	0 - 1023	0 -	3FF
1	1024 - 2047	400 -	7FF
2	2048 - 3071	800 -	BFFF
3	3072 - 4095	C00 -	FFF
4	4096 - 5119	1000 -	13FF
5	5120 - 6143	1400 -	17FF
6	6144 - 7167	1800 -	1BFFF
7	7168 - 8191	1C00 -	1FFF



The high-order 3 bits of the virtual address, 101 (5_{10}), provide the page number in the page table

The address 1010101010011_2 is converted to physical address $010101010011_2 = 0x553$ because the page field **101** is replaced by frame number **01** through a lookup in the page table (PTE 5)

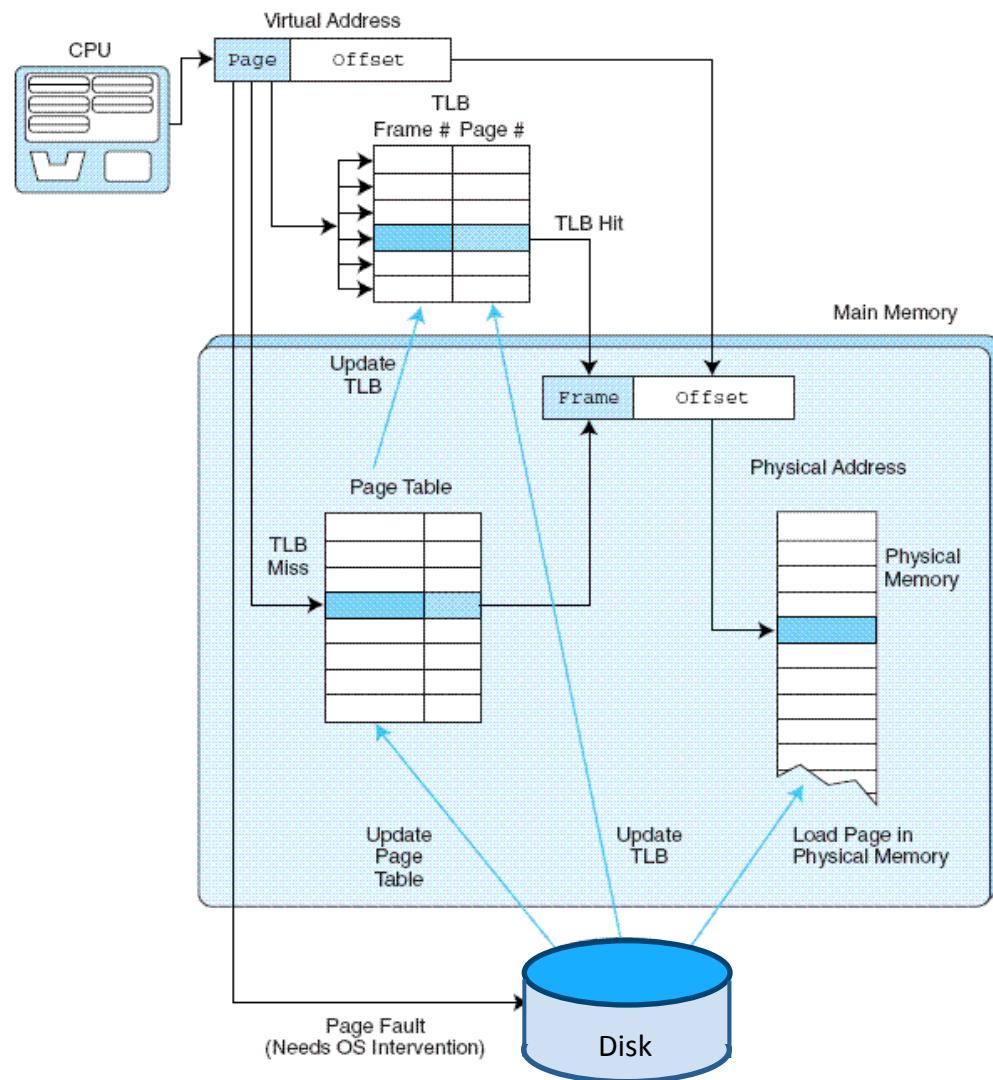
Page	Page Table		Addresses			
	Frame	Valid Bit	Page	Base 10	Base 16	
0	-	0	0	0 - 1023	0 -	3FF
1	3	1	1	1024 - 2047	400 -	7FF
2	0	1	2	2048 - 3071	800 -	BFF
3	-	0	3	3072 - 4095	C00 -	FFF
4	-	0	4	4096 - 5119	1000 -	13FF
5	1	1	5	5120 - 6143	1400 -	17FF
6	2	1	6	6144 - 7167	1800 -	1BFF
7	-	0	7	7168 - 8191	1C00 -	1FFF

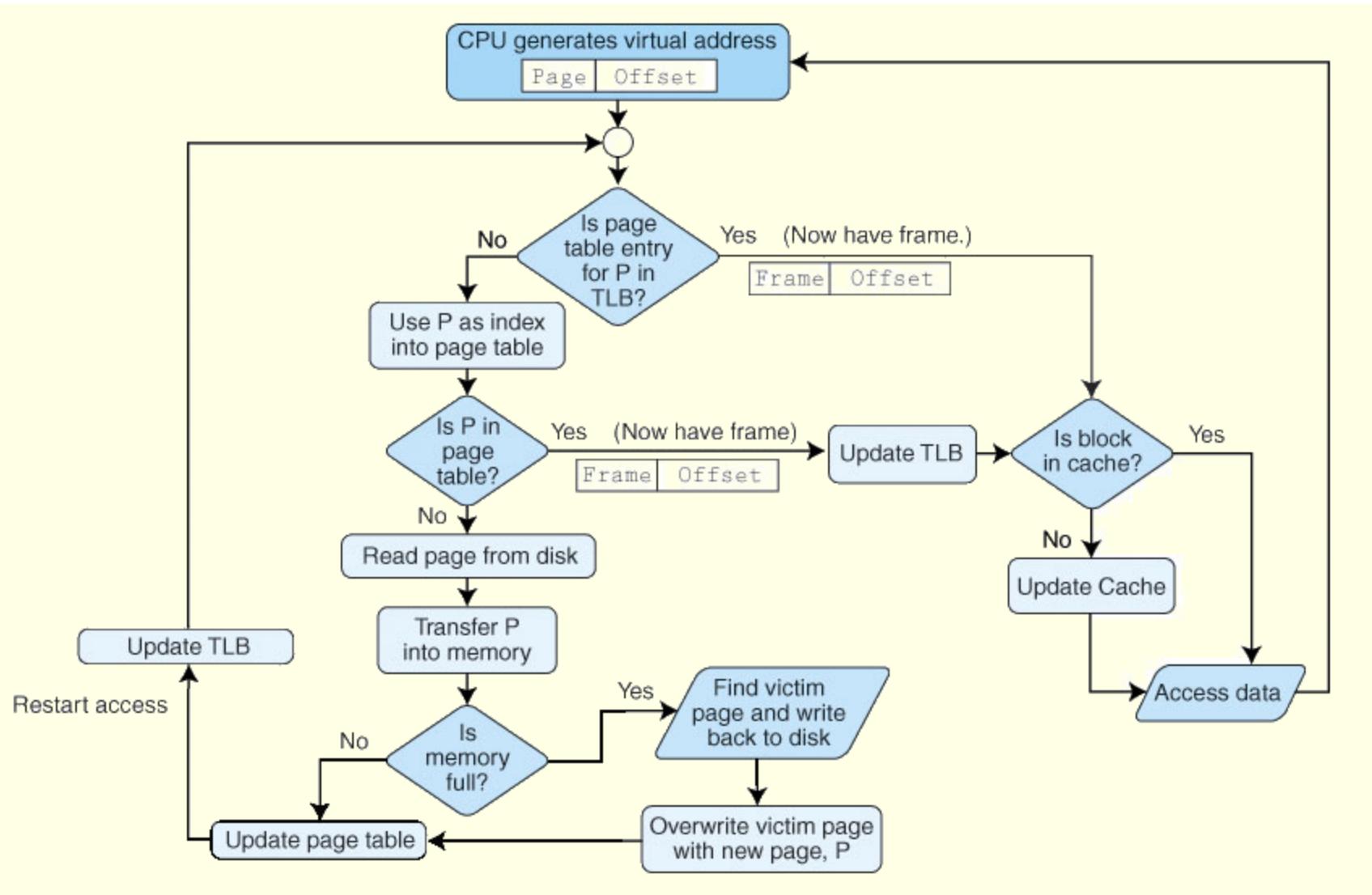
- Note that the use of the PMT causes 2 accesses
 - The first access is to the PTE to get the frame
 - The resulting physical address is then accessed
 - Ex: `lw $8,40($5)` now accesses memory twice to load the operand
- Suppose main memory access time = 200ns, the page fault rate is 1%, and it takes 10ms to load a page from disk.
Effective access time is:
$$\text{EAT} = 0.99(200\text{ns} + 200\text{ns}) + 0.01(10\text{ms}) = 100,396\text{ns.}$$

Even with 0% page faults, $\text{EAT} = 400\text{ns}$

- Cache can be used to speed up the translation
 - Recent translations can be saved in cache
 - This type of cache is a translation look-aside buffer (TLB)
 - Can be implemented as an associative cache
 - Its hit ratio is high due to the locality property
- All TLB entries are checked in parallel for the page
- A TLB hit provides the frame number immediately
- A TLB miss means that the PMT must be checked

1. Extract page# and offset from the virtual address.
2. If TLB hit, combine frame# from TLB with offset to yield the physical address
3. Else use page# to check the page table entry (PTE) in the page map table (PMT).
If valid bit is set, combine frame# with offset to yield the physical address.
4. Else generate a page fault and load page from disk.
5. Restart the access once page is in memory.







The Operating System must identify free frames

Bit string with one bit per frame (0=free, 1=occupied)

Linked list with one node per frame

Replacement occurs if all frames are full

Algorithm can be implemented in software

LRU (access bit periodically cleared by OS)

Random

Only modified pages are written back to disk

Page faults can result from writes as well

Pages are loaded then updated for write misses

Write-through is never used due to slow disk access

Modify bit (dirty bit) is set for each write to a page

A bit within the PTE is also set whenever a page is accessed

- The access bit is cleared periodically by the OS
- Access bit = 0 if page has not been recently used

The OS insures that frames are assigned exclusively

- Each process is assigned a different set of frames
- PTEs contain ID indicating owner of the page

Processes can reference same virtual address without conflict

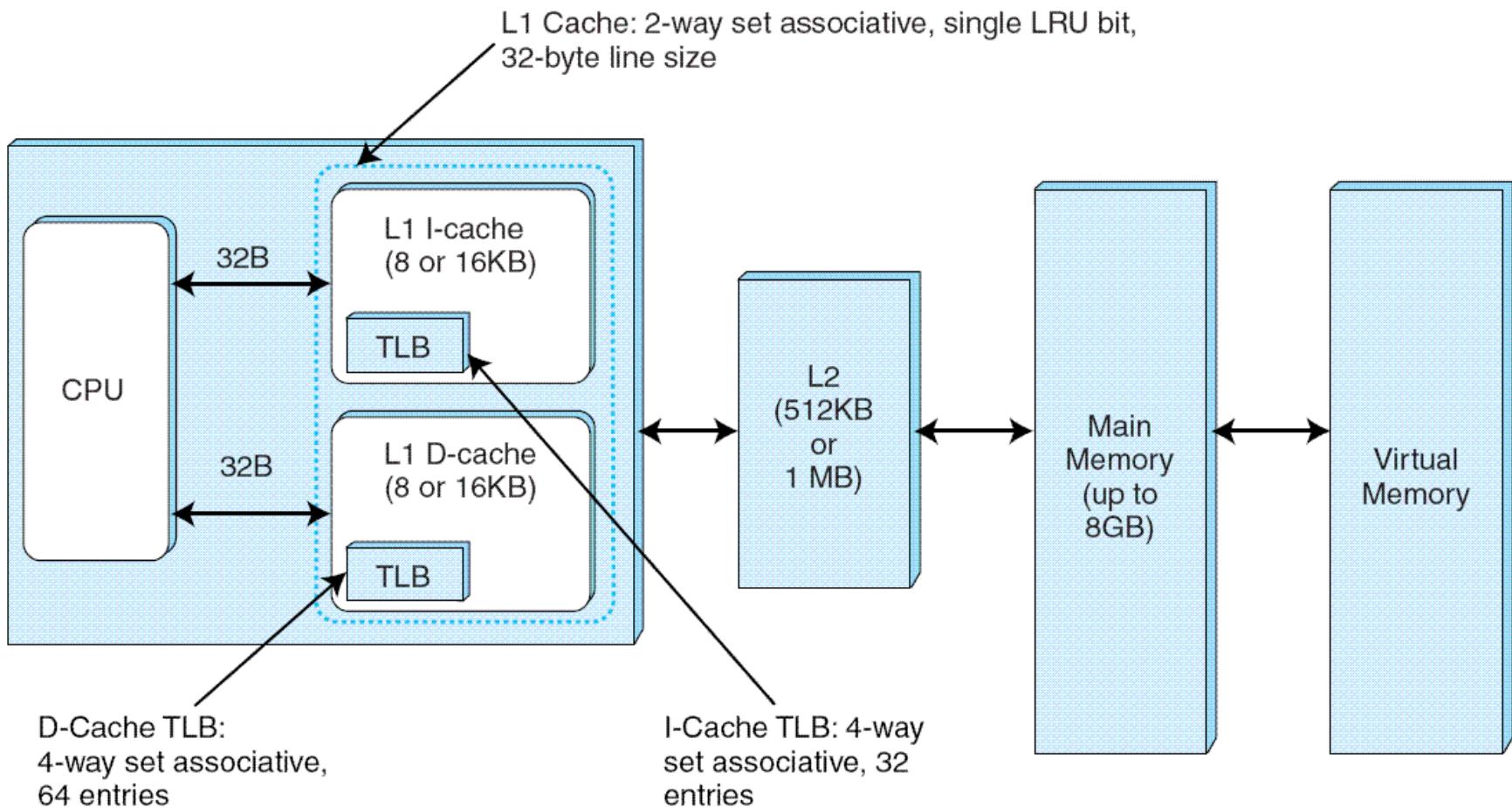
Different tasks can request to share pages

- Requires OS assistance

Hardware supports protection

- Privileged supervisor mode
- Privileged instructions
- PMT is only accessible in supervisor mode

Pentium System with multiple caches and virtual memory



- PMTs contain an entry for each possible page
- Address space may correspond to millions of pages
- Some programs only use a subset of the pages
 - Only pages actually used need to be in memory

- Multi-level Page Tables can reduce the required space
 - Only the portion of the PMT that map the pages used
 - The unused portions can reside on the disk
 - The PMT itself can be paged
- By paging the PMT, less memory is used
- This works by partitioning the page number field

- A two-level system (page table & page offset):
 - Page table field indicates entry in first level table
 - Page table offset indicates entry in second level table
 - Page offset indicates location within selected page
- Three levels could also be used
- The first level table is kept in memory
 - Lower level tables are read in on demand
 - The final level contains the actual frame numbers

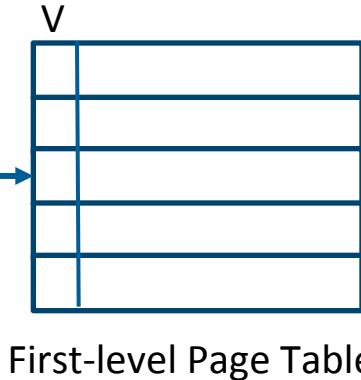
32-bit Virtual Address is split into three fields



9

10

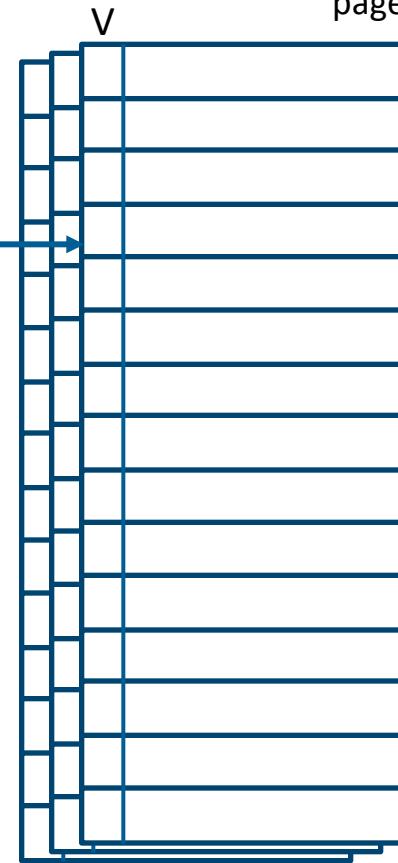
Entries contain address of required second level table if valid, or indicate that second level table is on disk.



First-level Page Table

$$2^9 = 512 \text{ entries}$$

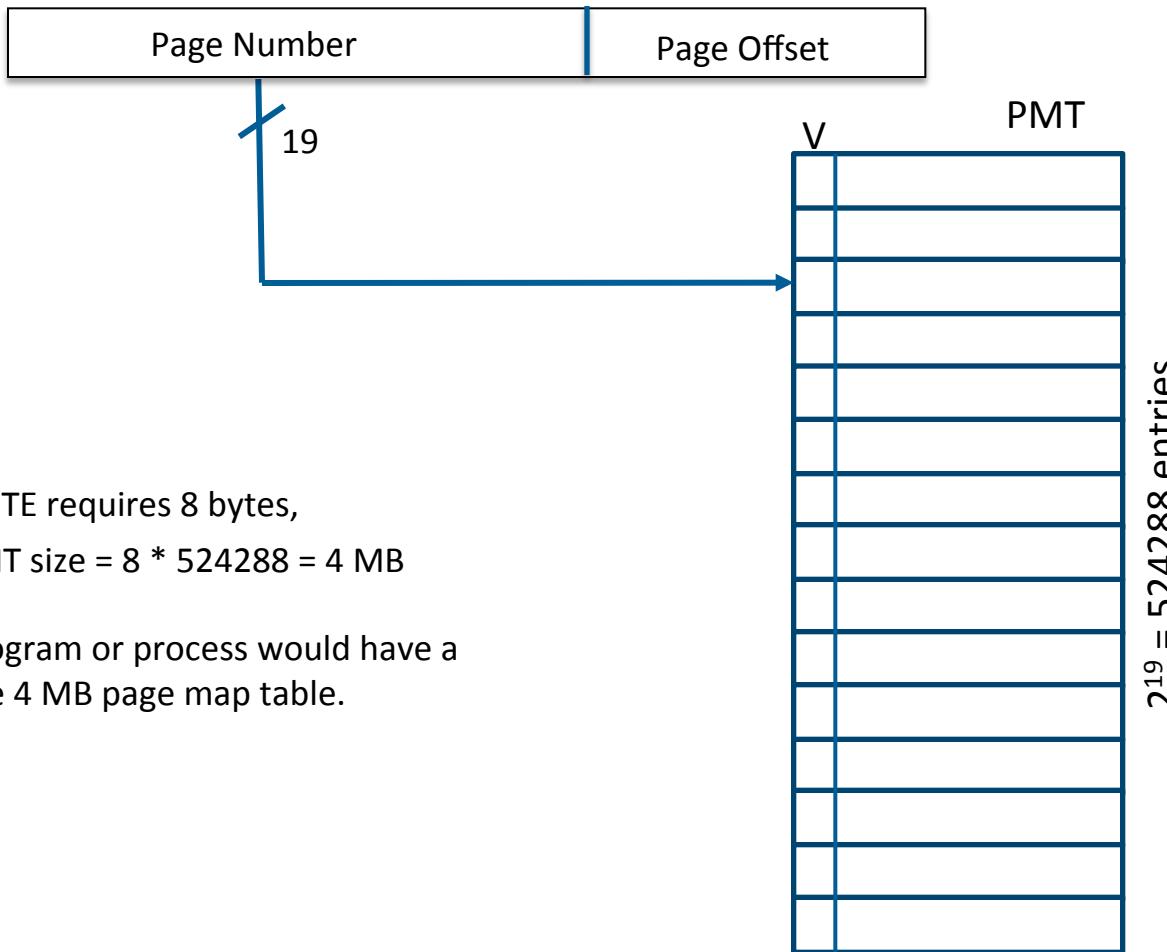
Entries contain frame# if valid, or indicate that page is on disk.



Second-level Page Table

$$2^{10} = 1024 \text{ entries}$$

Using equivalent single PMT would require 2^{19} table entries



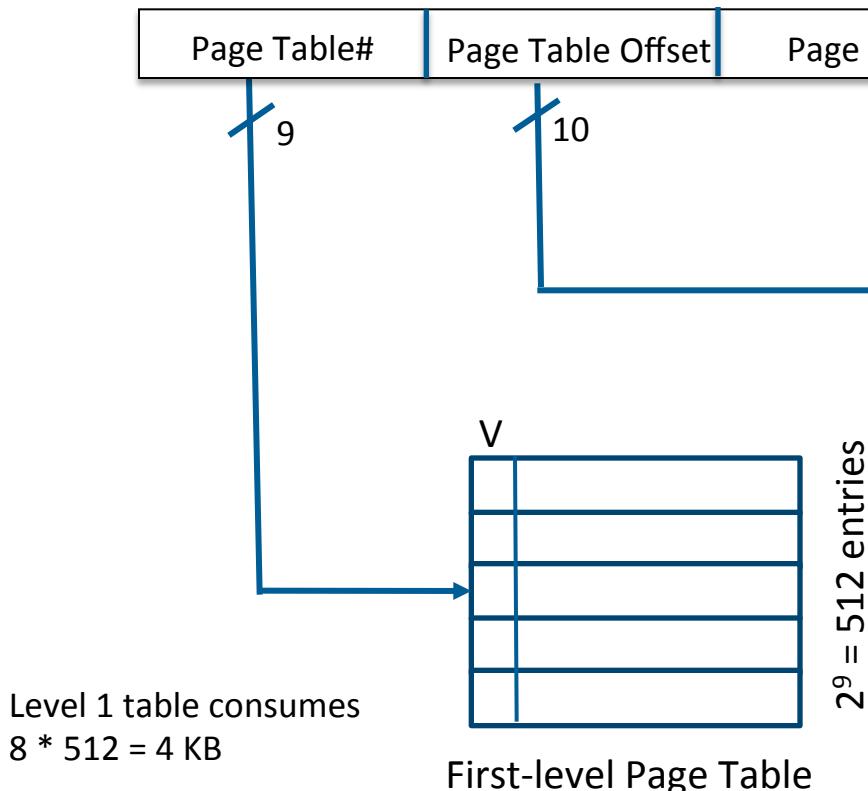
Entire PMT is in
memory, even if not all
entries are used.

If each PTE requires 8 bytes,
total PMT size = $8 * 524288 = 4 \text{ MB}$

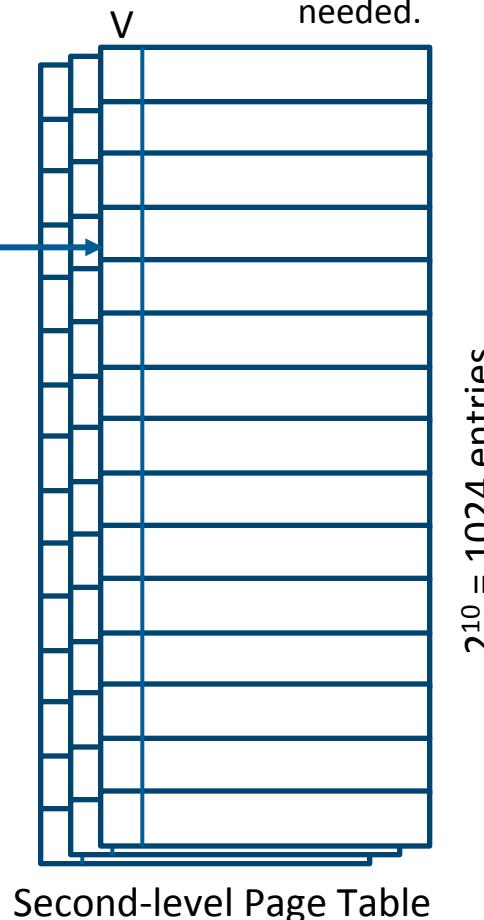
Each program or process would have a
separate 4 MB page map table.

Example 2-level System

The first level table is kept in memory.



If no more than 1024 pages are used, only one level 2 table is needed.

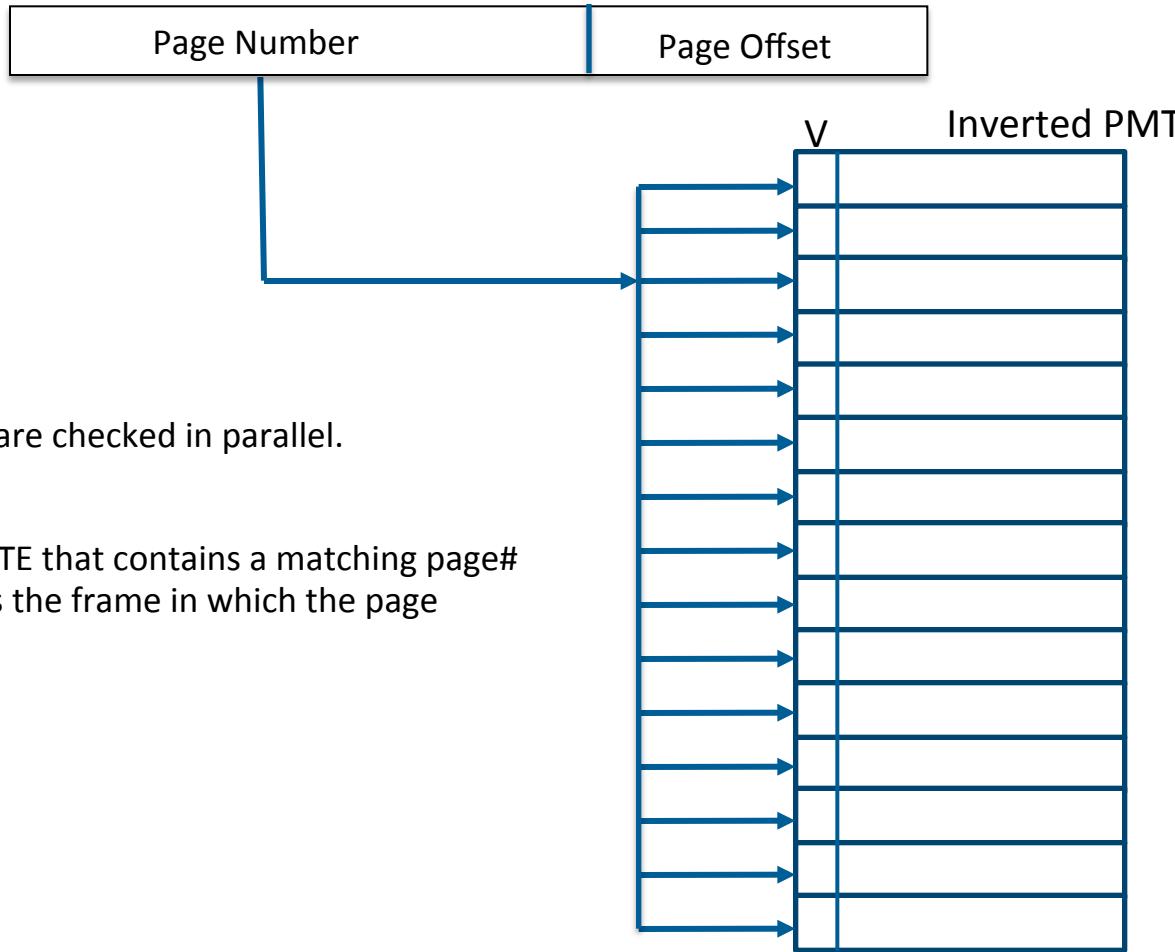


- Multi-level PMTs increase the effective access time
 - *Each level requires a memory access*
 - *All but the first level could cause a page fault*
- Bits to left of offset must be split into N fields
 - One field for each of the levels
 - Virtual address translation takes longer

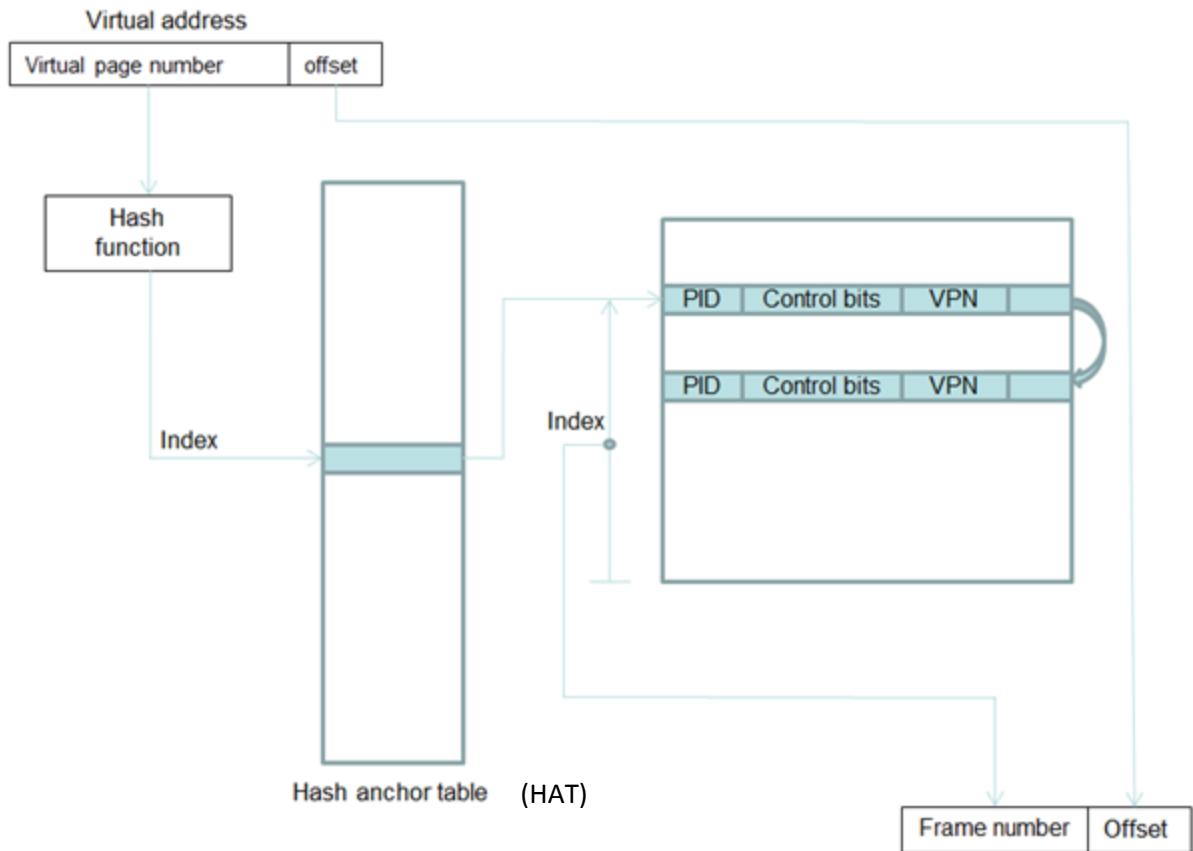
- PMTs described above, are “*forward*” page tables
- *Inverted* PMTs can reduce the required space
- A virtual address is divided into two fields:
 - *page number* field, and
 - *offset* field

- There is one PTE per memory frame
 - Number of PTEs depends on physical memory size
 - If valid, the PTE identifies the page in the frame
 - Fault occurs if no PTE contains a matching page number
- There is one system wide inverted PMT
 - Instead of one PMT per process as with forward tables

To translate address, one option is to perform associative search of the inverted PMT.



- There are alternatives to the associative PMT search
- Hash function may be used instead
 - *The page number is hashed into a table index*
 - *However, collisions must be handled*
 - *Happens if more than one page number hashes to same PTE*
- Collisions can be handled by using:
 - secondary hash function
 - Rehashes the page# into a different index
 - Chaining
 - Links entries that map to the same index



First entry
contains link to
next entry
resulting from
collisions.

The HAT contains pointers to the heads of the chains for each page number. This allows more entries without large increases in the table size.

- This module begins an overview of I/O
 - How computers handle I/O transactions
 - The I/O system infrastructure
 - Device controllers
 - I/O bus systems
- Functionality as well as performance are important
 - Access to networks and the internet
 - The use of a rich variety of devices
 - Digital cameras
 - Music players
 - Video display devices
 - Printers
 - Storage devices

- I/O affects the overall system performance
 - I/O devices are even slower than central memory
 - Overlapping I/O with computation can hide the slowness
- Amdahl's Law applies to I/O as well
 - Limiting the amount of I/O boosts performance
- I/O is defined as a subsystem of components
 - These components move coded data
 - The exchange is between external devices and a host system

- I/O performance is measured in two ways:
 - I/O throughput or bandwidth
 - I/O transactions per unit time
- Bandwidth (bytes per second)
 - Depends on clock rate and width of data pathways
 - Important when large amounts of data need to be exchanged
- Transactions per second (TPS)
 - Important when numerous small data exchanges are made

- Overall system performance depends on the interaction of all of its components
- Improving the most heavily used components is most effective
- This idea is quantified by Amdahl's Law:

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

S is the overall speedup;
 f is the fraction of work performed by a faster component; and
 k is the improvement of the faster component

Example: if a component is made 100% faster, then $k=2$.



Facts:

Processes spend 70% of their time running on the CPU and 30% of their time waiting for disk service

Options:

make the CPU 50% faster for \$10,000

make the disk drives 150% faster for \$7,000

Question:

Which option would be better based on benefit and cost?

- The processor option offers a 30% speedup:

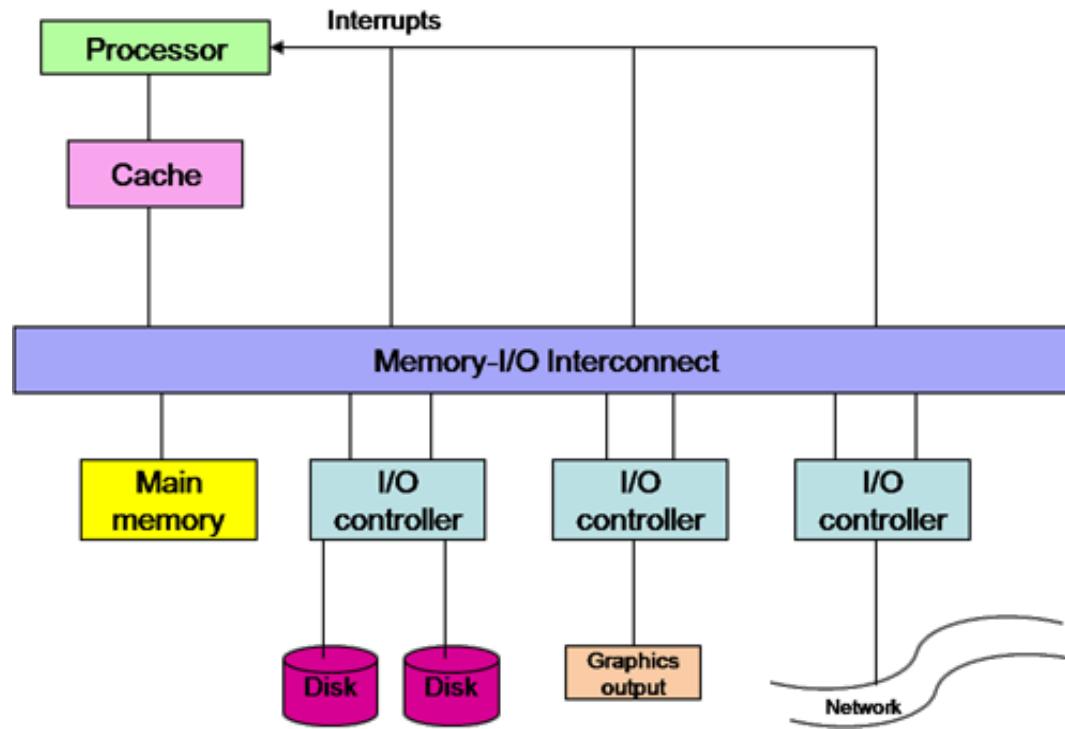
$$f = 0.70, \quad S = \frac{1}{(1 - 0.7) + 0.7/1.5} = 1.30$$
$$k = 1.5$$

- And the disk drive option gives a 22% speedup:

$$f = 0.30, \quad S = \frac{1}{(1 - 0.3) + 0.3/2.5} = 1.22$$
$$k = 2.5$$

- Each 1% of improvement for the processor costs \$333 ($10000/30$), and for the disk a 1% improvement costs \$318 ($7000/22$).

I/O controllers manage communication between I/O devices and the CPU or main memory.

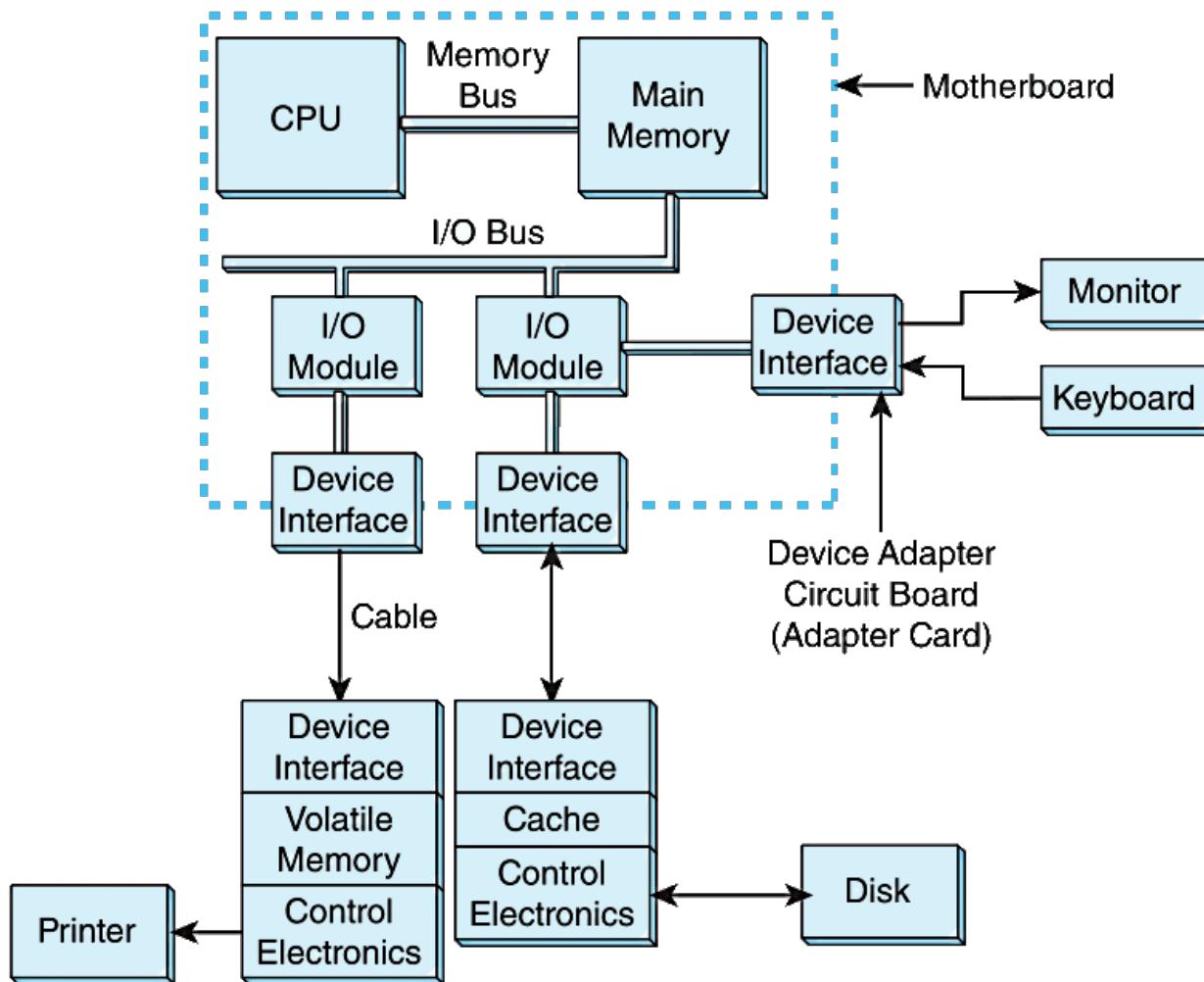


I/O transactions adhere to well-defined rules (*protocol*)



I/O subsystems include:

- Blocks of main memory that are devoted to I/O functions.
- Buses that move data into and out of the system.
- Control modules in the host and in peripheral devices
- Interfaces to external devices such as keyboards and disks.
- Cabling or communications links between the host system and its peripherals.



Various types of buses allow for communication among components.

- Devices can be accessed in one of two ways:

- 1 *Memory mapped*

- One or more registers are assigned to each I/O device
- The registers correspond to specified memory addresses
- Part of the address space is thus reserved for I/O
- Memory access instructions can also perform I/O
- RISC systems tend to use this approach

- 2 *Port mapped or Isolated I/O*

- Each device is assigned one or more port numbers
- Special I/O instructions transmit data via the ports
- Access must be identified as memory or I/O accesses
- CISC systems tend to use this approach

MIPS example:

```
lui    $t0,0xFFFF  # base address for keyboard  
lw     $t1,4($t0)  # read the next input character
```

Pentium example:

```
KB_DATA    EQU  60H  # port number for keyboard  
  
in     AL,KB_DATA  # read character from port into AL register
```

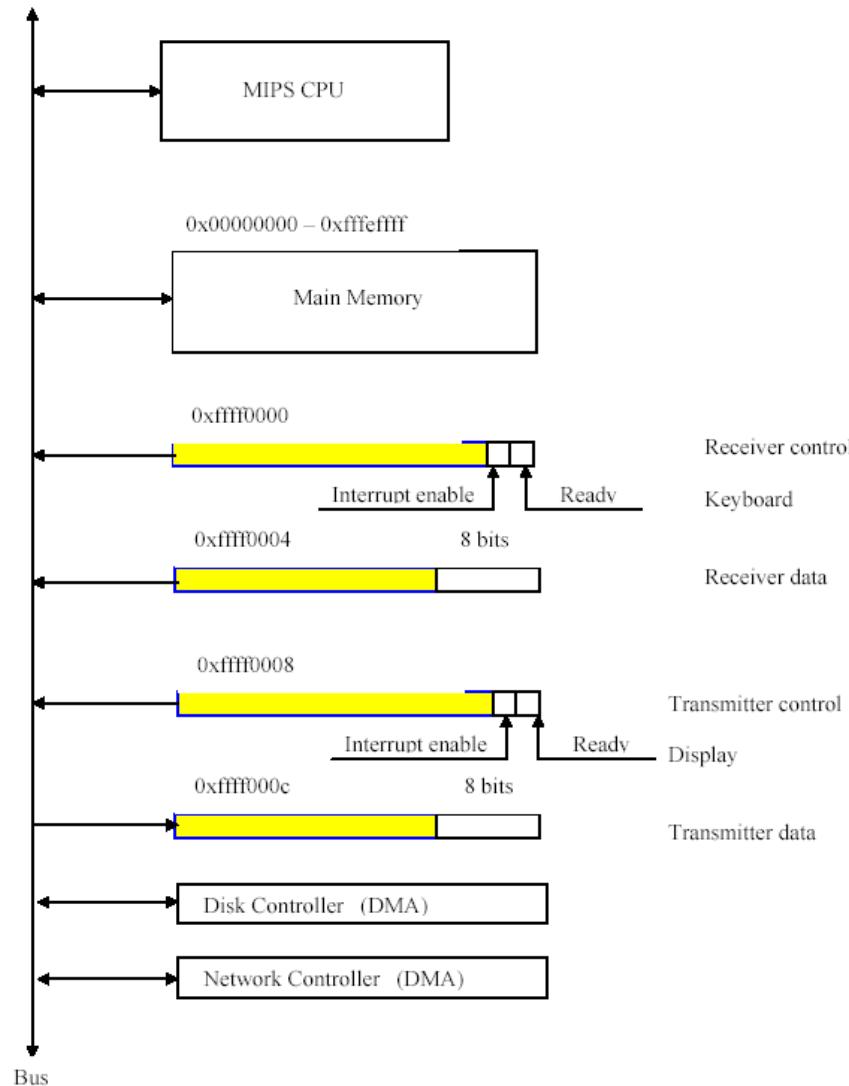


Next we will examine more details of low level I/O

- Programmed I/O
- Interrupt Driven I/O
- DMA

- Direct program controlled I/O uses polling
 - Device status registers are repeatedly checked
 - CPU is kept busy by the polling operations
- Each device is assigned one or more registers
 - Status register
 - Control register
 - Data register
- Simple devices have fewer registers
 - Keyboard
 - Console display
- Complex devices have more registers

Example memory-mapped system.



Detecting key presses using polling:

```
# device register base address = 0xFFFF0000
    lui      $t3,0xFFFF    # KB status address
poll_CR: lw       $t1,0($t3)    # read status register
          andi     $t1,$t1,1    # does LSB=1?
          beqz    $t1,poll_CR   # keep checking if not
          lw       $t0,4($t3)    # else read character into $t0
```

Time between keystrokes can be considerable

The status bit is automatically cleared when the data register is read

Output to the console display using polling:

```
Poll_XR:    lui      $t3,0xFFFF      # device reg base address
             li       $t0,'>'      # ASCII code for output char
             lw       $t1,8($t3)    # Console Cntrl reg offset=8
             andi    $t1,$t1,1      # does LSB=1?
             beqz   $t1,poll_XR    # keep checking if not
             sw       $t0,12($t3)    # display the character
```

Status register is polled to see if the console is ready

Ready bit goes to 0 when data register is written

Ready bit goes back to 1 once the character has been displayed

Polling keeps the CPU busy

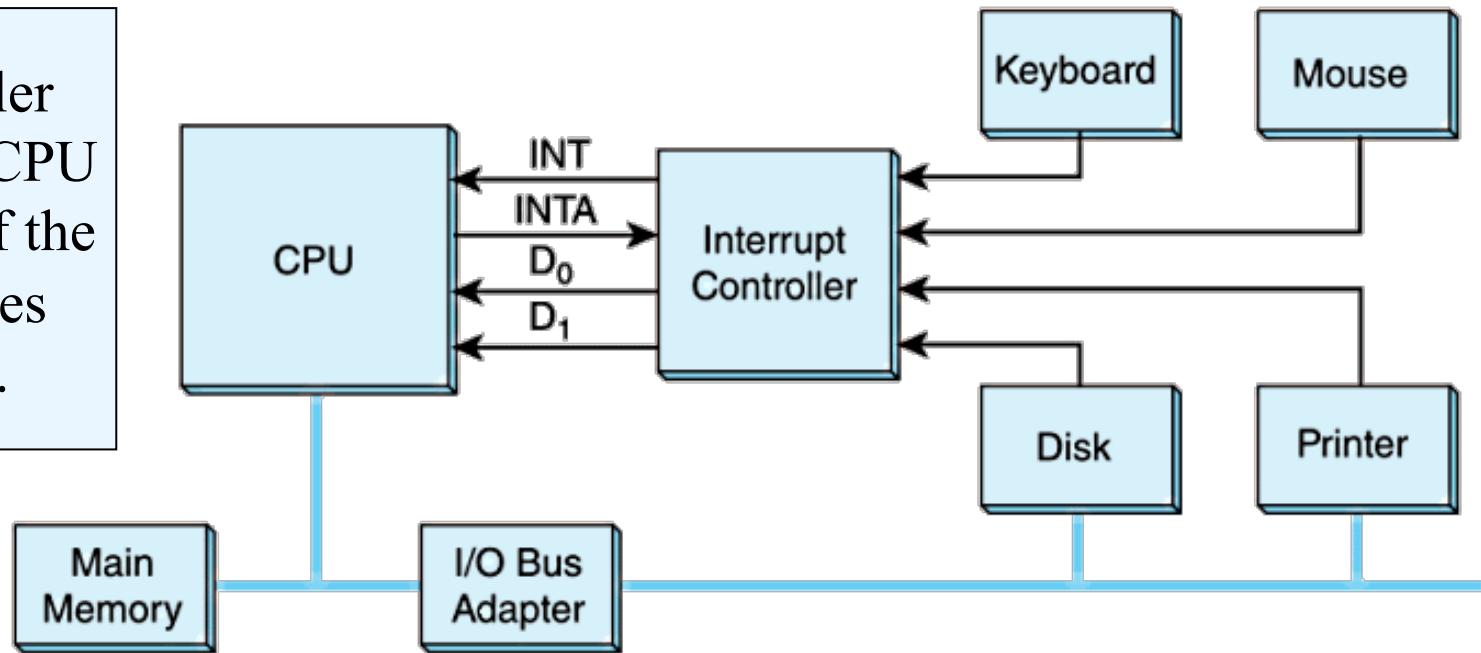
Using interrupts instead allows the CPU to do other useful work

- CPU responds only when I/O request is made
 - Each device is assigned an ID
 - Each may have a different priority level
 - Interrupt signals are sent to notify the CPU of I/O completion
- Some systems employ vectored interrupts
 - Each corresponds to a different interrupt handler address
 - The Interrupt service routine (ISR) is called via the supplied address
- The MIPS uses a single vector address
 - Control goes to this vector address for all exceptions
 - MIPS interrupts are a particular type of exception

This is an idealized I/O subsystem that uses interrupts.

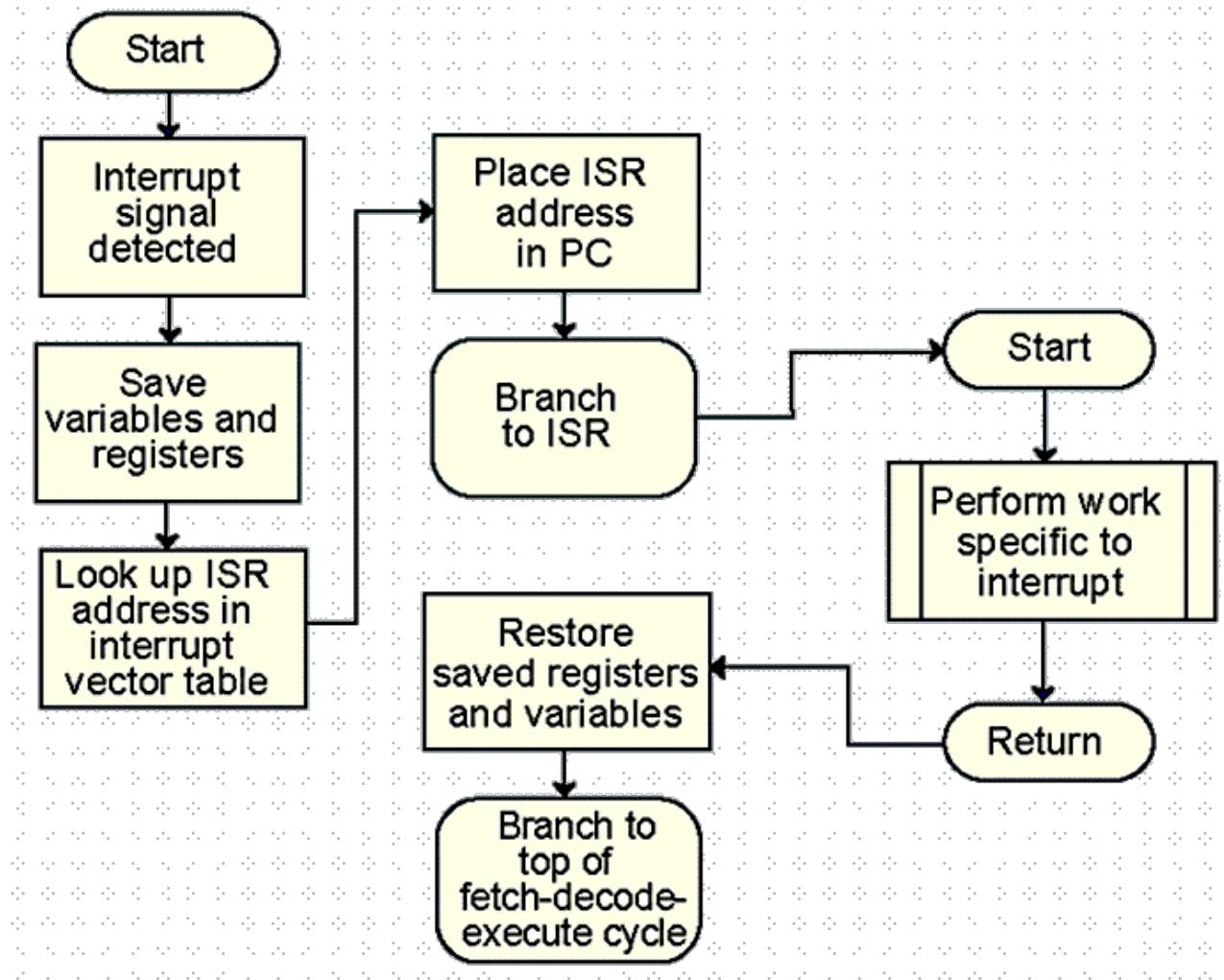
Each device connects its interrupt line to the interrupt controller.

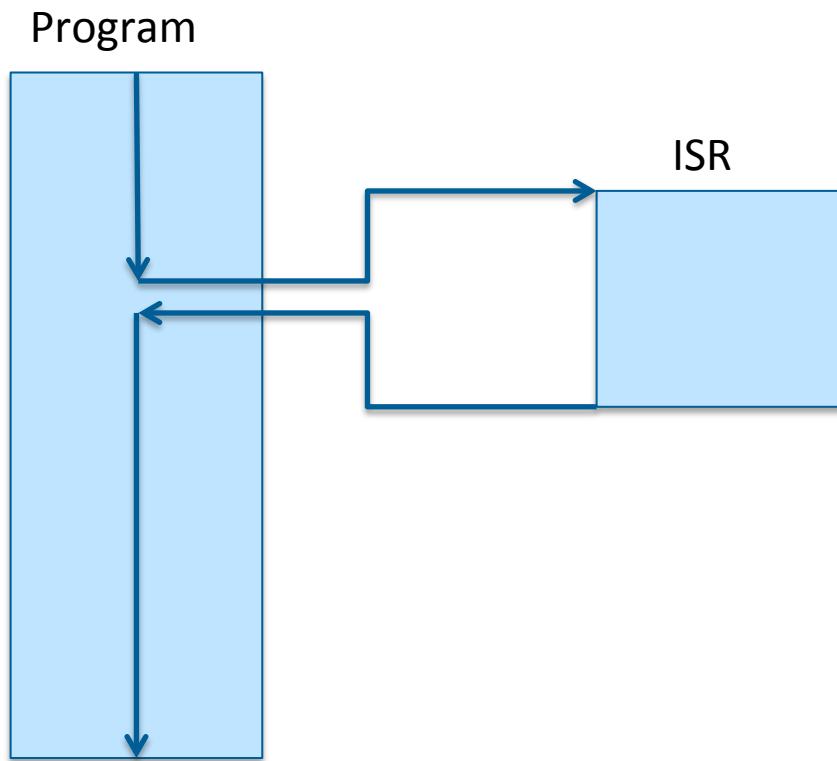
The controller signals the CPU when any of the interrupt lines are asserted.



The system's state is saved before the interrupt service routine is executed and is restored afterward (e.g. all registers)

Interrupt Driven I/O

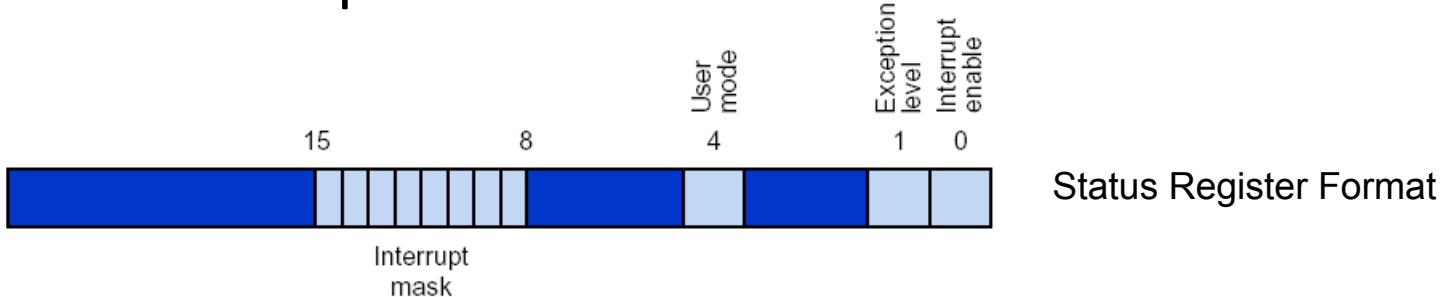




Interrupts are like unsolicited procedure calls
The program may be unaware that an interrupt occurred
Interrupts are delayed until the current instruction completes

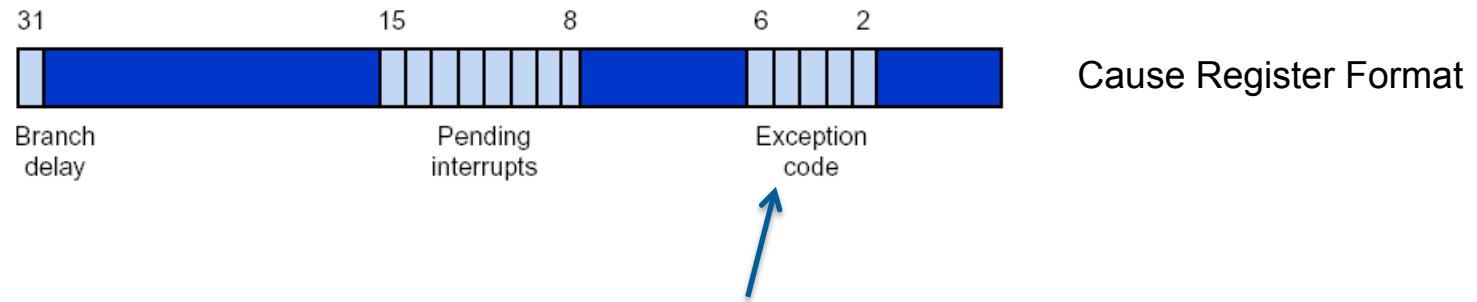
Support for interrupts on the MIPS Machine

CP0 register \$12



Status Register Format

CP0 register \$13



Cause Register Format

Exception code = 0 for interrupts

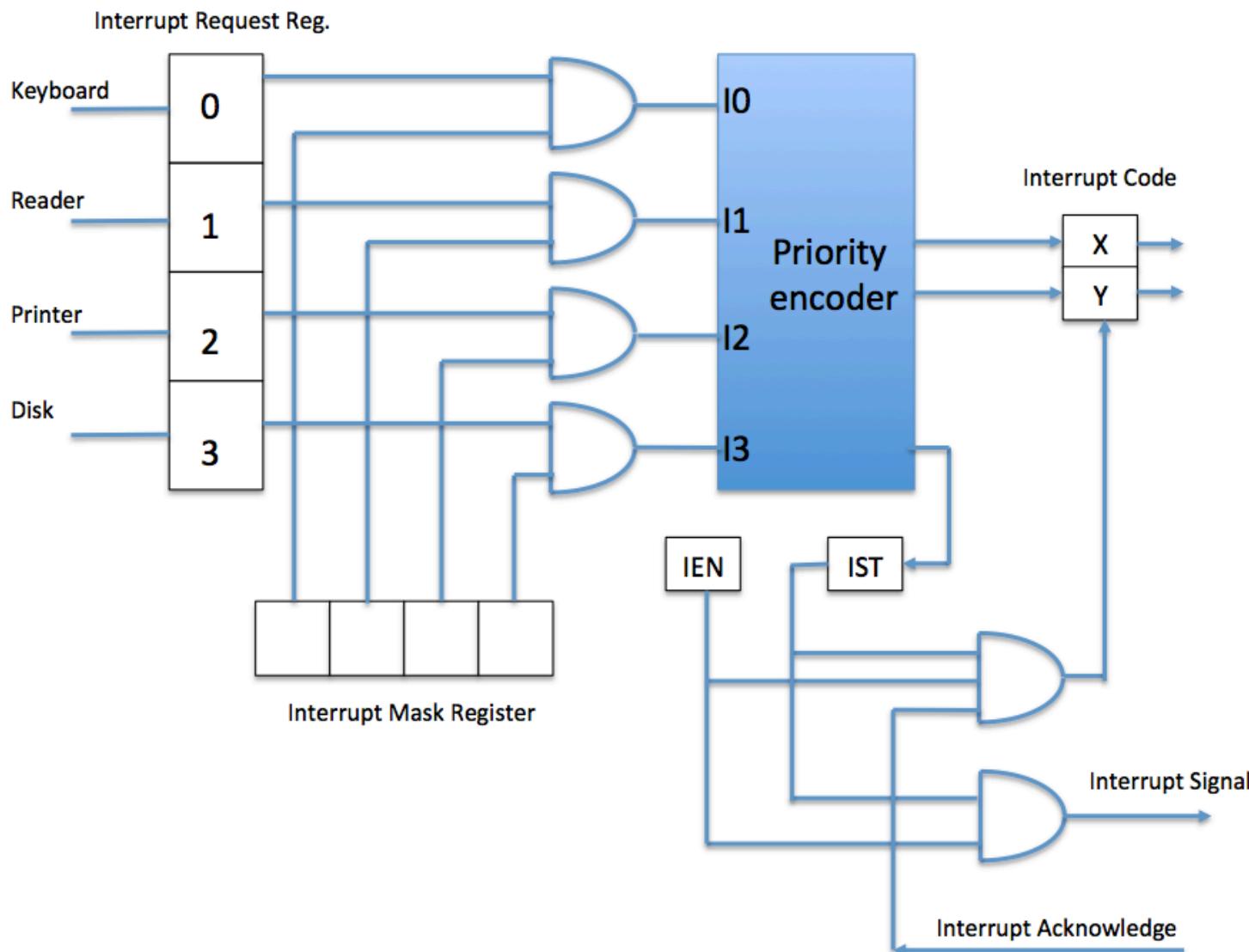
Non-zero value indicates other types of exceptions

CP0 register \$14



EPC exception program counter

Interrupt Controller

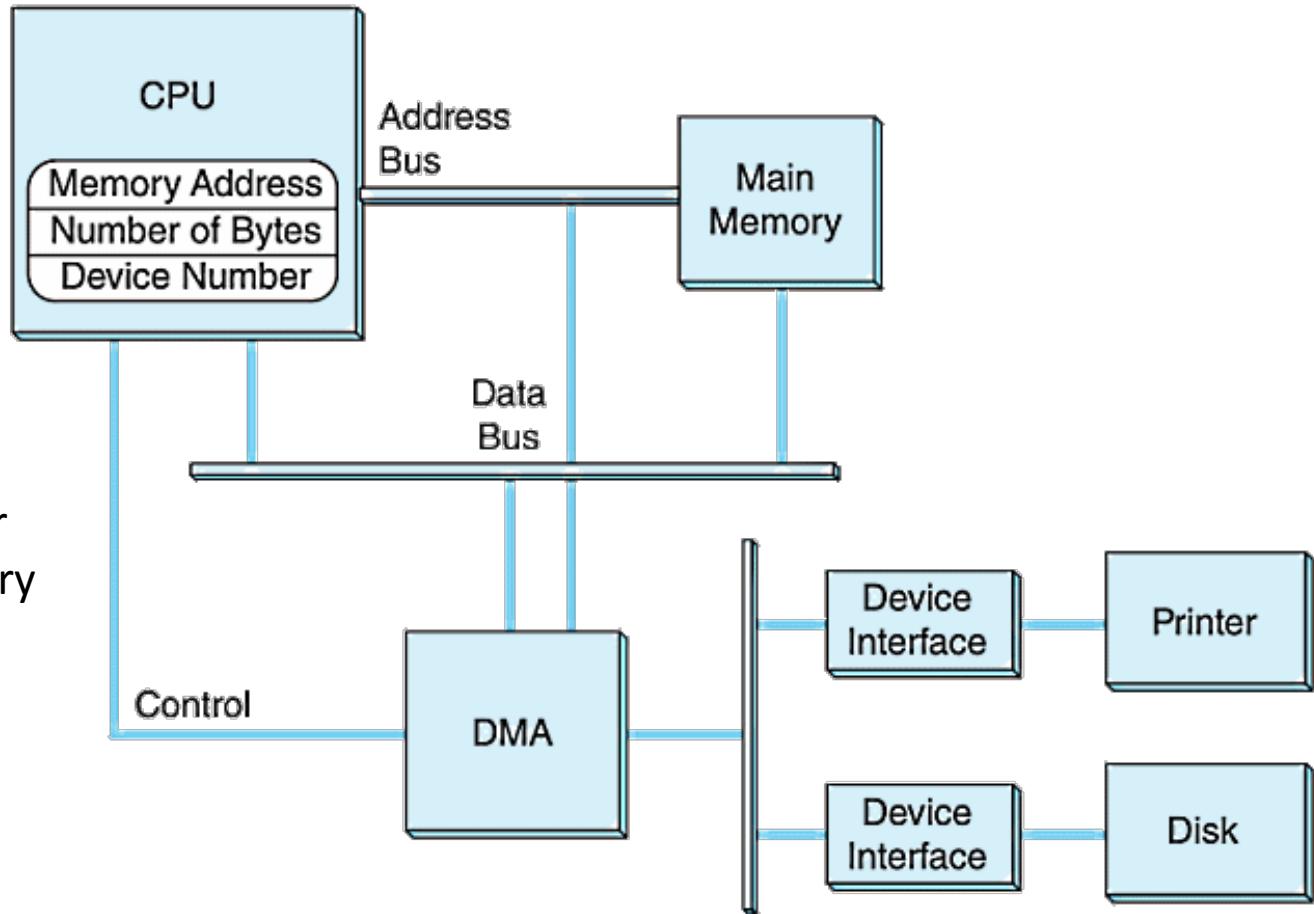


- Some devices can directly access memory
 - Bypasses the CPU and goes directly to memory
- The CPU is only involved at the beginning and at the end
 - DMA Device ID or address
 - Memory address (source or destination of data)
 - Size of block transfer
 - Direction (input or output)
- A single interrupt occurs at the end of the transaction
 - DMA controller decrements count and adjusts pointer to memory
 - Consumes less CPU time than with interrupt after each byte or word

This is a DMA configuration.

Notice that the DMA and the CPU share the bus.

The DMA runs at a higher priority and steals memory cycles from the CPU.



The DMA does not go through the cache, so the cache contents could be made stale.

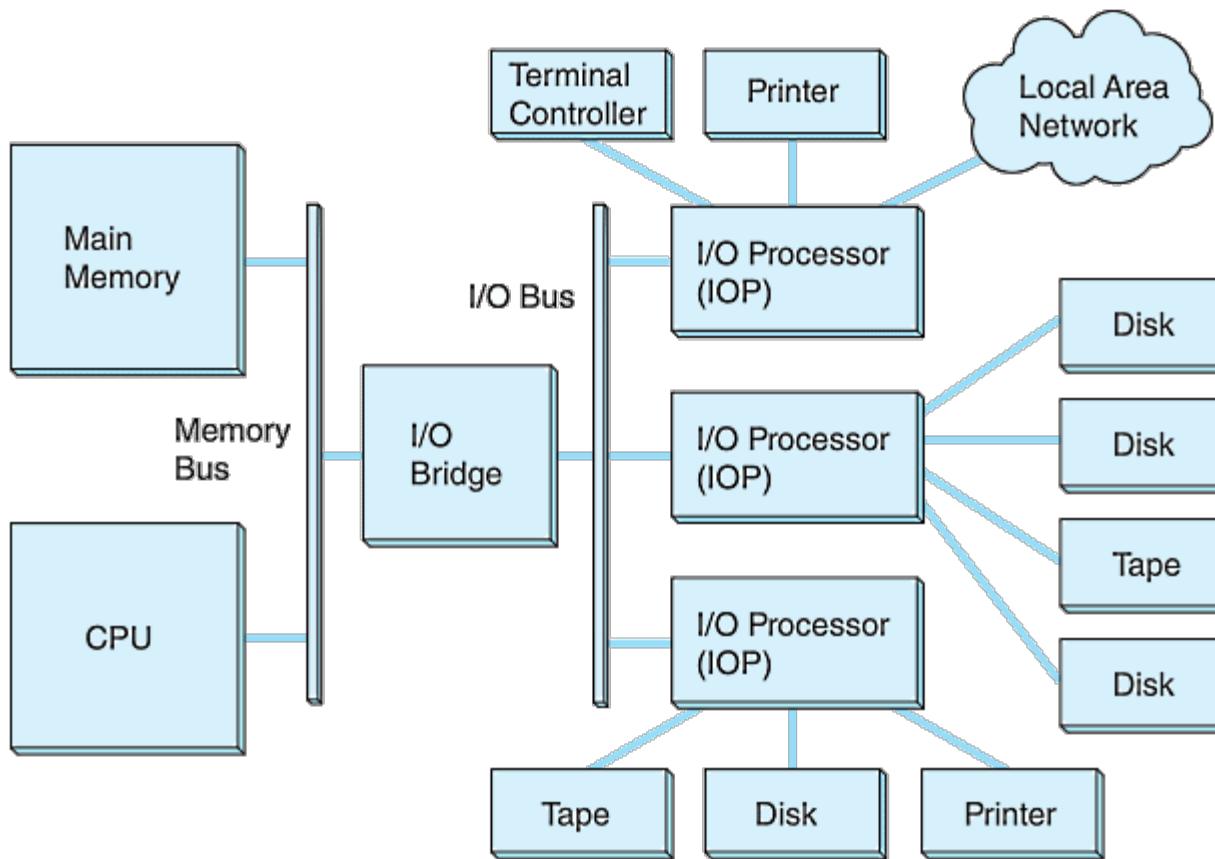
Ways CPU and DMA device can share memory bus:

- Transparent mode
 - Bus is used for DMA only when not needed by CPU
 - This is the ideal theoretical case
 - Can't really predict when CPU will not need the bus
- Cycle stealing mode
 - CPU frees bus long enough for DMA of one data unit
 - Cycles are given up by CPU to allow DMA transfers
- Burst mode
 - CPU relinquishes bus long enough for a block transfer
 - Feasible if CPU is actively getting cache hits

- IOPs are also known as I/O channels
 - They have their own instruction set tailored for I/O
 - They can carryout a series of multi-step transactions
 - They run in parallel with and independent of the CPU
 - They transfer entire files or groups of files
- “channel command” IBM’s term for I/O instruction
- Intel 8086 had a companion 8089 I/O processor
- Relieves the CPU of most I/O duties
 - Less burden on CPU than interrupts or DMA

- IOPs have more intelligence than DMA controllers
 - They negotiate protocols
 - issue device commands
 - translate storage coding to memory coding
 - transfer entire files or groups of files independently of the host CPU
- The host CPU creates the I/O program
 - These are I/O specific instructions
 - CPU tells IOP where to find the I/O program

- This is a channel I/O configuration.

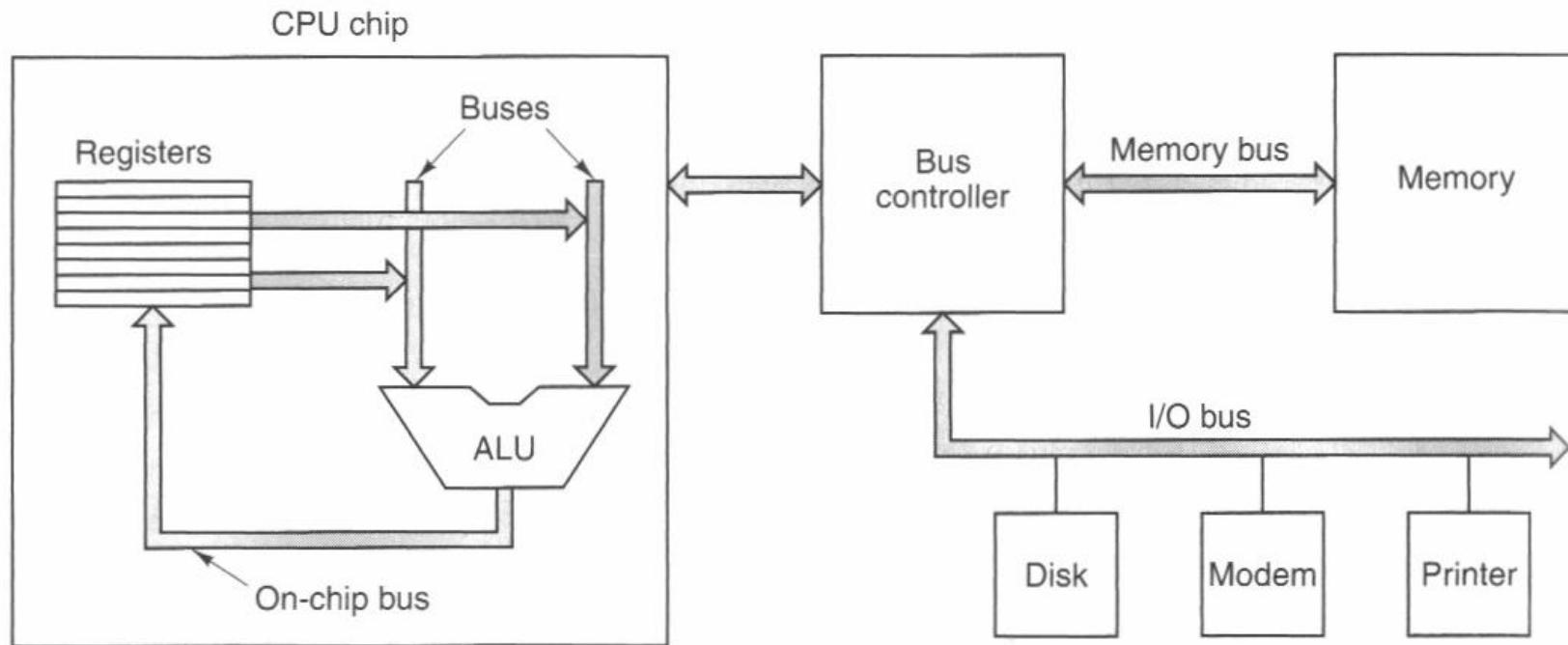


- Memory mapped device registers are accessed like any other data items.
 - Copies of the registers in the cache may be stale
 - CPU accesses go through the cache
 - I/O device accesses do not go through the cache
- DMA controllers use physical addresses
 - Network packet buffers could be a problem if cached
 - DMA transfer goes directly to memory
 - If the buffer maps to cache, the CPU may be unaware

- Some CPUs have a cache flush instruction
 - This invalidates one or more the cache lines
 - References to the items then go directly to memory
- An alternative is to use uncached areas
 - Specified address ranges used by the CPU do not go through cache
 - I/O device registers and I/O buffers could be assigned to these uncached areas

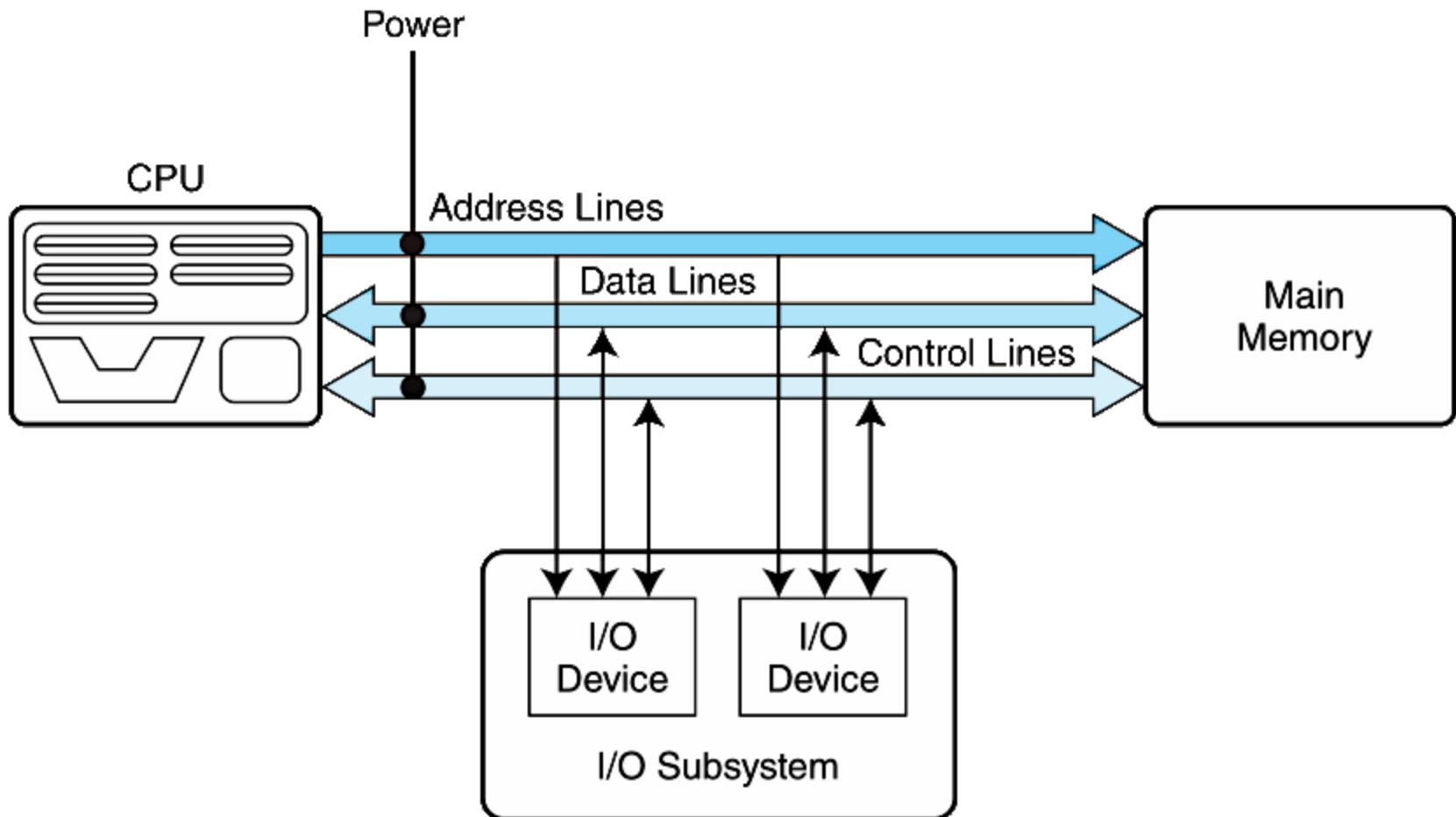
- Next we will examine bus systems in more detail
 - Buses are required to access memory and I/O devices
- We will also examine disk arrays (RAID systems)

- Components exchange information via buses
 - A bus is a set of shared lines or electrical traces
 - Chips have pins that connect to these shared lines
- Computers use a hierarchy of buses
 - Some are internal to the CPU
 - External buses are used for I/O
- Bus protocols are rules for using the bus
 - specify timing techniques
 - Arbitration rules to resolve conflicting requests
 - Synchronous versus asynchronous communication



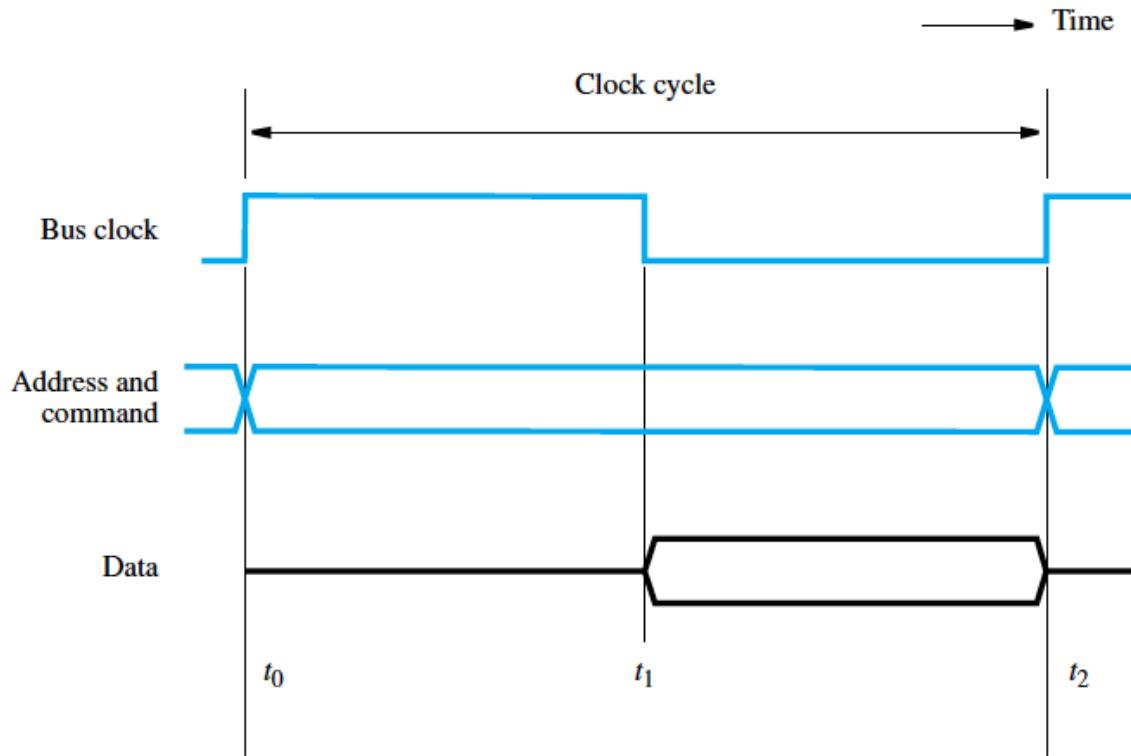
- CPU-to-memory buses are parallel synchronous buses
 - Operate at higher rates than I/O buses
 - Clock signals synchronize the exchanges
- I/O buses can be parallel or serial
 - Operate at rates that match slow I/O device speeds
 - Communicate asynchronously using handshake signals

- Buses consist of:
 - data lines
 - control lines
 - address lines
- Data lines convey bits from one device to another
- Signals on control lines determine:
 - the direction of data flow
 - when each device can access the bus
- Address lines determine the location of the source or destination of the data



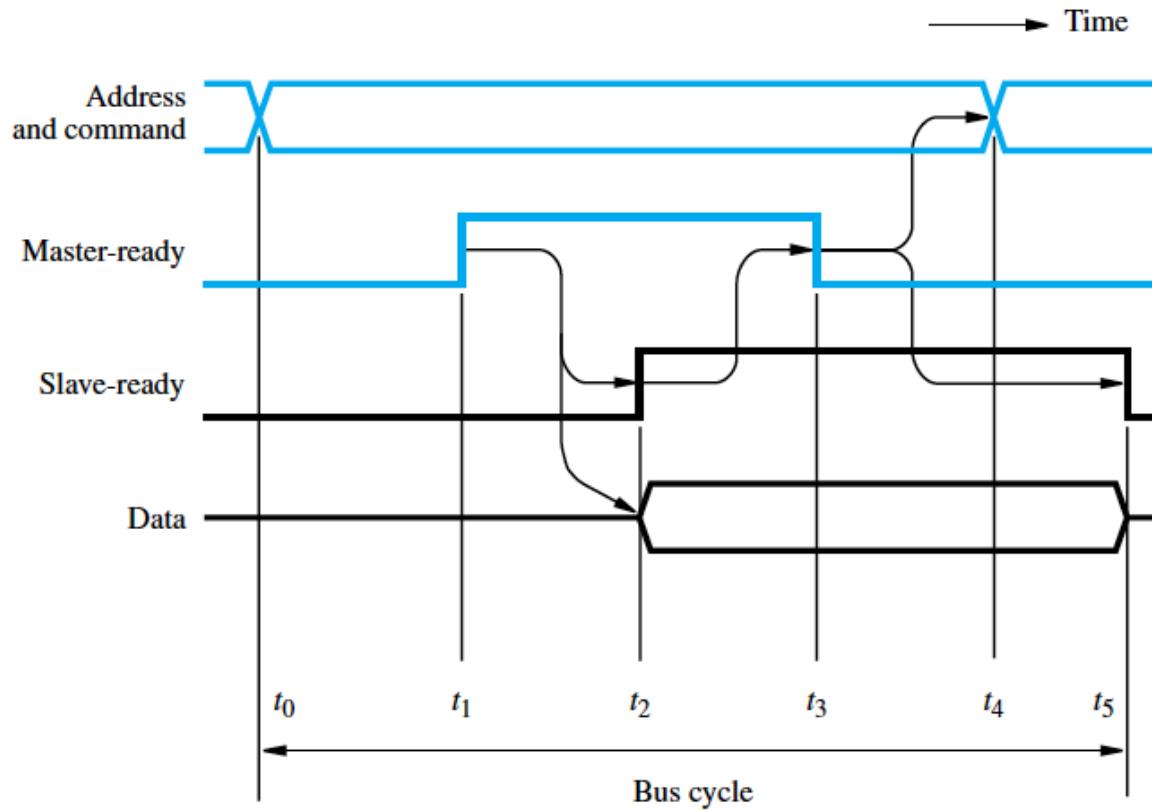
- Buses may be serial or parallel
 - Bits are sent simultaneously along each parallel bus line
 - Bits are sent one after the other along a serial bus line
- Bus bandwidth = amount of data transferred per unit time
 - Depends on width (number of data lines)
 - Depends on cycle time and number of cycles used
- A common clock signal is used by *synchronous* buses
 - Transactions adhere to a bus schedule
 - The schedule must accommodate the slowest device
 - Slowest device dictates the cycle time
 - Clock skew can be a problem
 - Different devices may not see the clock signal at the exact same time

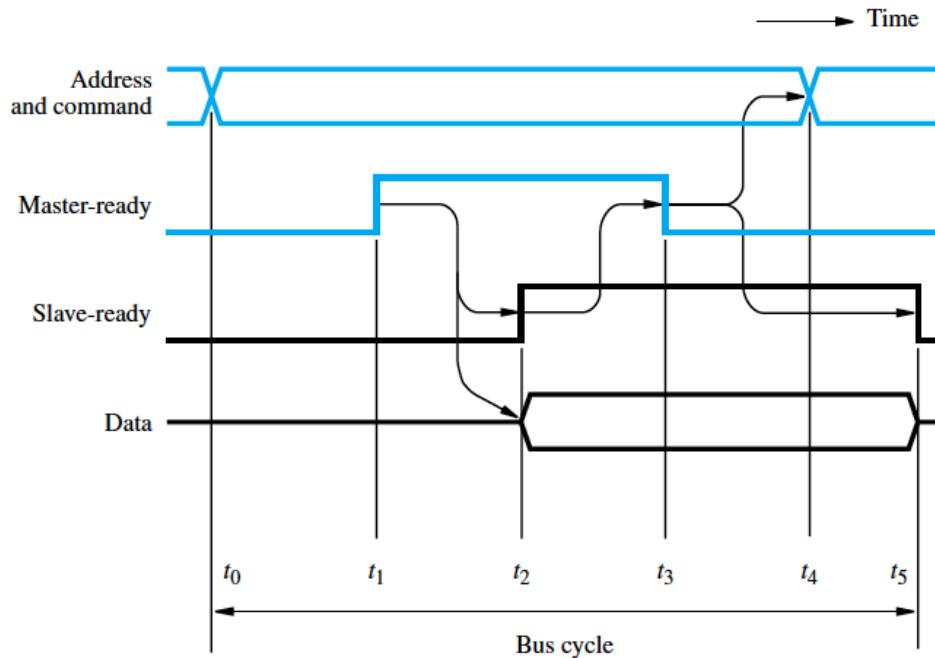
Synchronous Bus Read



Data requested from the specified address is available at t_1
If data cannot be provided in time, “wait states” are inserted
Wait states are additional complete clock cycles

- **Asynchronous** buses exchange handshake signals
 - Each step in the transaction requires a signal or response
 - transaction speed varies with the devices involved (self-paced)





Master-ready is asserted to signal valid address and command

Slave-ready is asserted when requested data is available

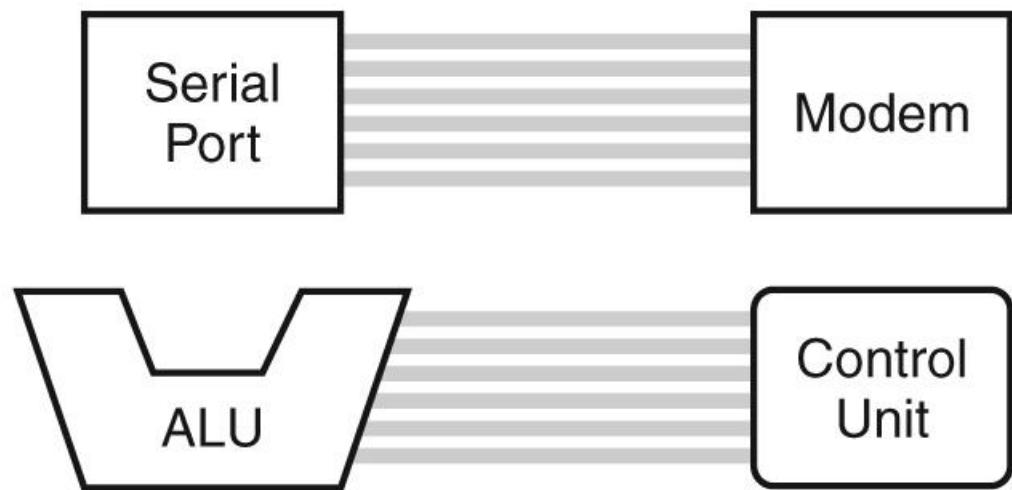
master-ready is de-asserted in response to receiving the data

Slave-ready is de-asserted to allow next transaction to begin

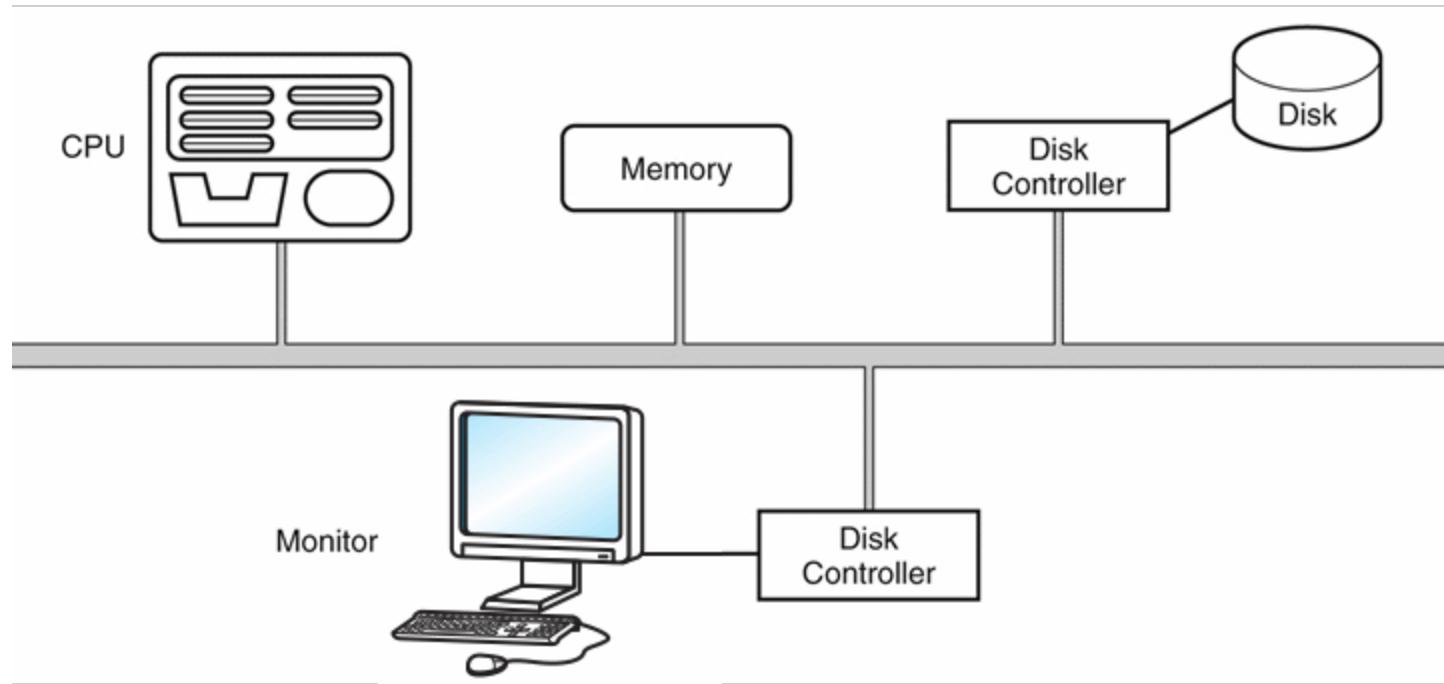
A positive action is required to proceed to the next step

- Two types of buses are commonly found in computer systems: *point-to-point*, and *multipoint* buses.

These are point-to-point buses:

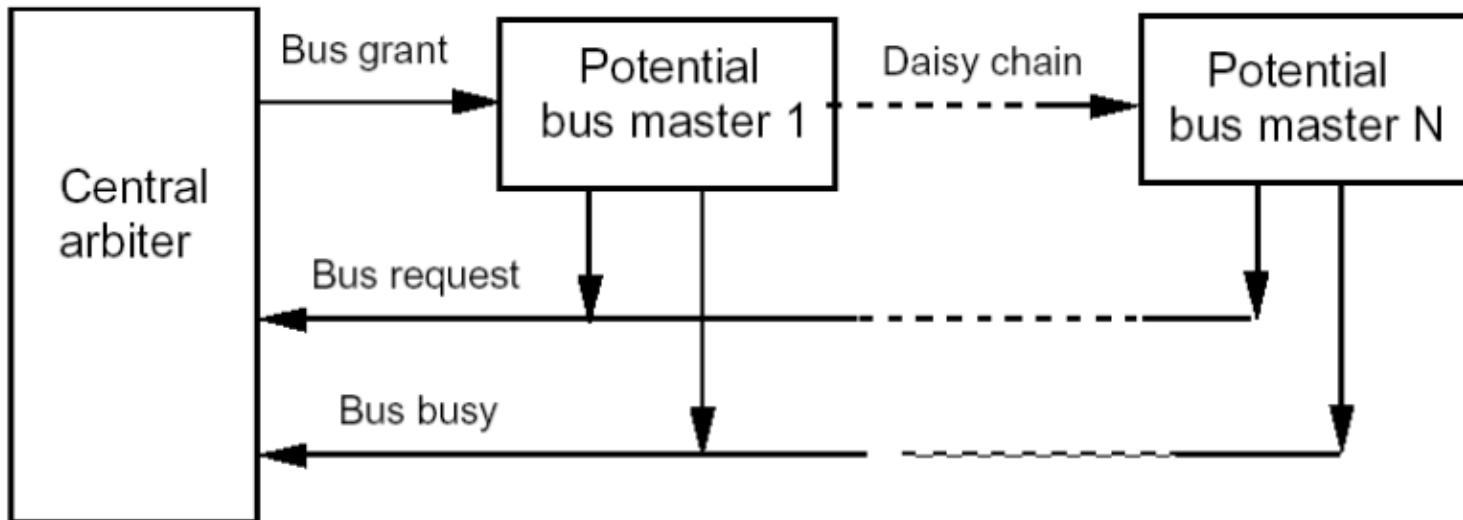


- A multipoint bus is a shared resource
- Protocols can be built into the hardware
 - They control access to the bus



- Bus transactions:
 - All the actions needed to exchange information
 - Occurs in a “bus cycle”
 - May correspond to multiple clock cycles
- Only one device at a time controls the bus
 - Called an “initiator” or “master”
 - Initiator or master communicates with “target” or “slave”
- Arbitration resolves competing requests for control

Daisy chain: Permission passed along the chain of devices



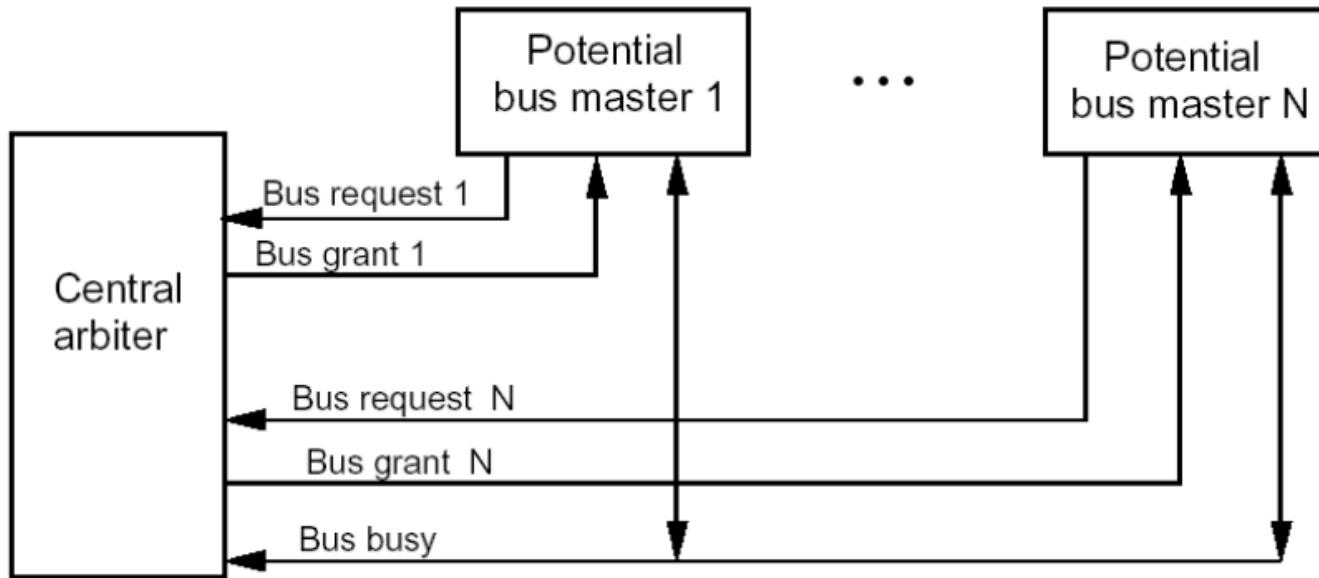
Priority is based on proximity to arbiter

Malfunctioning device can lockout others that follow

Requires relatively few bus lines

Centralized parallel:

Each device is directly connected to the arbiter

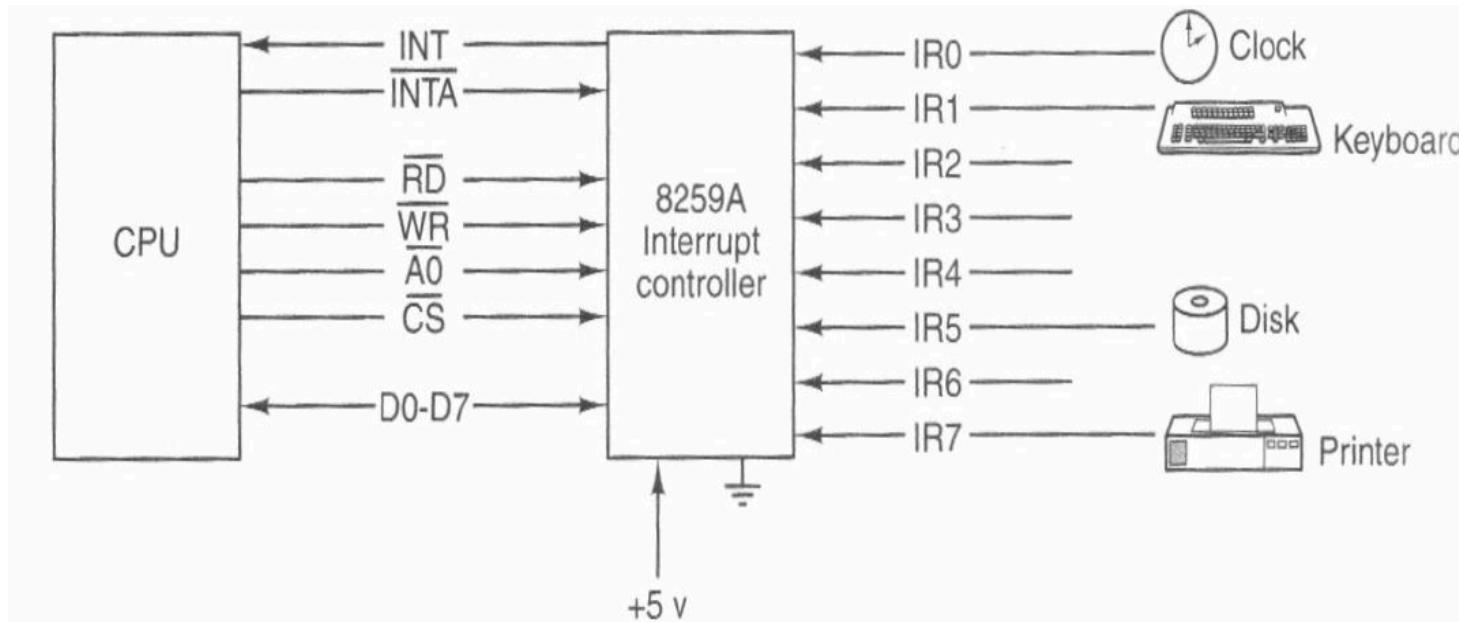


Arbiter assigns priority based on separate grant lines

Malfunctioning devices can be ignored

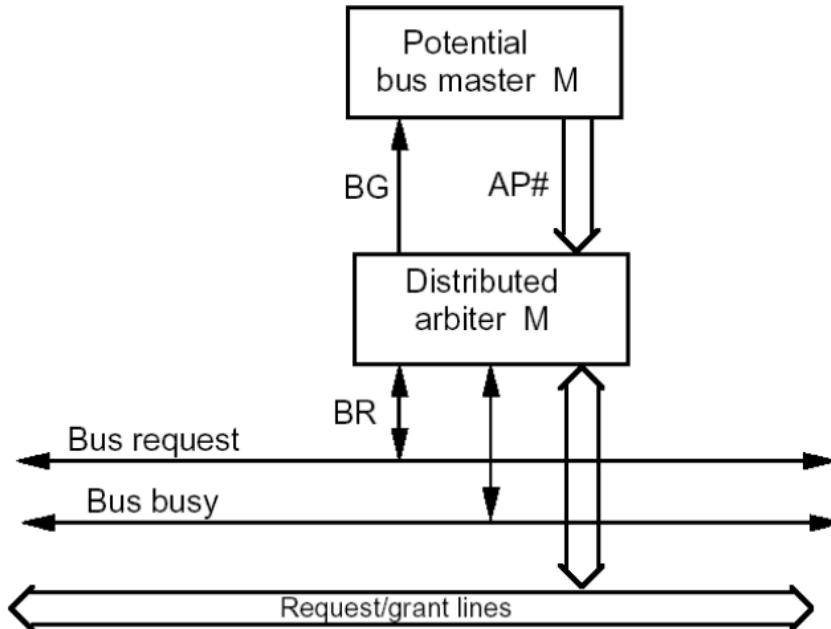
Requires more bus lines than daisy chained alternative

Centralized parallel example:



Up to 8 I/O controller chips can be connected
Using the 8 IRQ (interrupt request) lines of the 8259A controller
Used on early PC systems (Intel)

- Distributed arbitration using self-detection:
 - Devices decide among themselves who gets the bus
- Arbitration priority numbers (AP#) determine order
 - Requester AP# is compared with that of other current requesters
 - Requests with lower AP# are removed
 - Remaining request is granted
- Distributed arbitration using collision-detection:
 - Any device can try to use the bus
 - Waits and tries again if a collision occurs



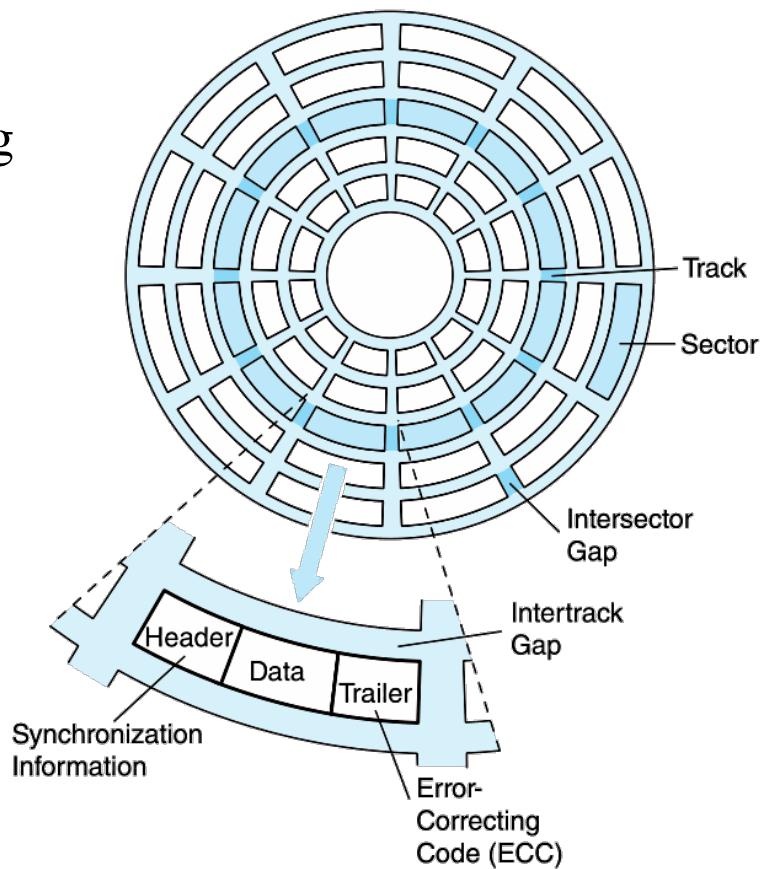
- Starvation can occur with priority based arbitration
 - Very frequent high priority requests can lockout others
- Fairness can be imposed by centralized or distributed arbiter
 - Limit number of consecutive grants to high priority devices

Option	Higher performance	Lower cost
Type	Separate address, data and control lines operate in parallel	Multiplexed lines carry address and data at different times
Transfer size	Block transfers have less overhead	each data unit incurs overhead for individual transfers
Bus masters	Multiple requires arbitration but increases flexibility	Single master requires no arbitration
Transaction type	Split transactions allow interleaving of transfers	Continuous connection may tie up lines while waiting
Width	More lines allow more data per transfer	Fewer lines reduce cost but limit amount of data in a transfer
Clocking	Synchronous is faster but may be limited by the slowest device	Asynchronous requires handshake signals (more overhead) but is not limited by the slowest device

- Processor speeds far outpace the speed of disks
 - Instructions execute in nanoseconds
 - Disks still require milliseconds to access
- Parallel access can hide the slowness of disks
 - Separate blocks are read from different disks at the same time
- The seek and access delays are not reduced
 - The beginning of the data must still be determined
 - Multiple pieces of data are accessed in parallel
 - The multiple pieces are buffered and reassembled
 - The complete file can then be transferred into memory

Disk tracks are numbered from the outside edge, starting with zero.

A sector is the smallest unit of transfer.



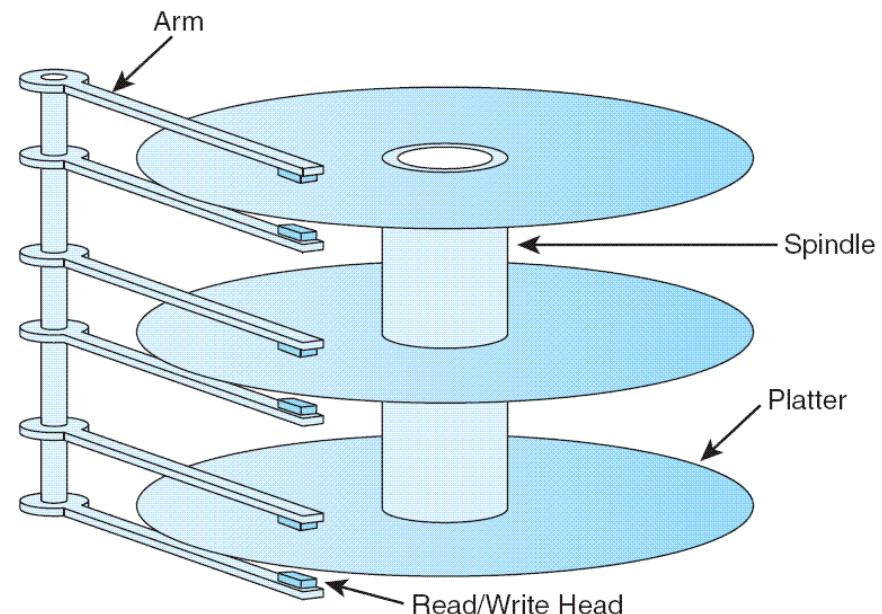
Hard disk platters are mounted on spindles.

Read/write heads are mounted on a comb that swings radially to read the disk.

The rotating disks form a logical cylinder beneath the read/write heads.

Data blocks are addressed by their cylinder, surface, and sector

A disk can perform one read or one write at a time.



Seek time is the time that it takes for a disk arm to move into position over the desired cylinder.

Rotational delay is the time that it takes for the desired sector to move into position beneath the read/write head.

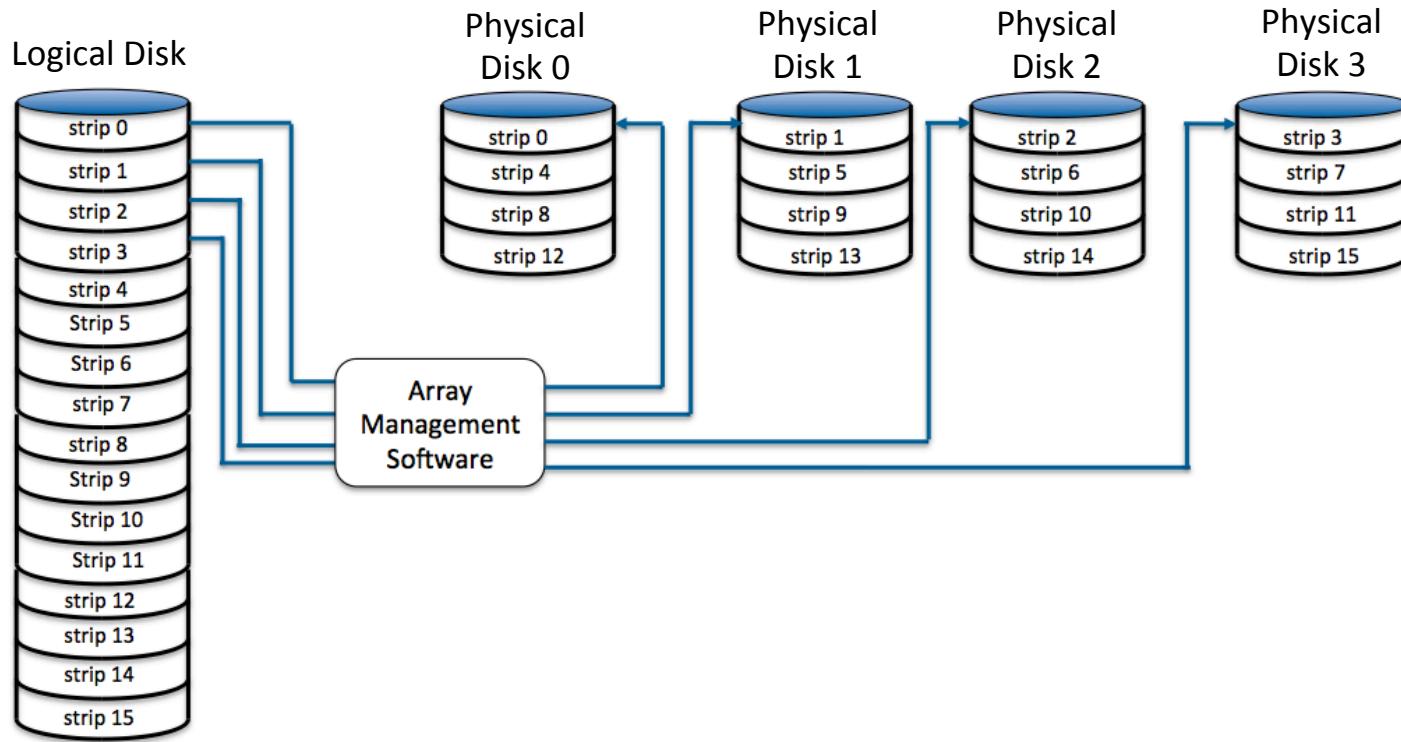
Seek time + rotational delay = *access time*.

The data transfer time is also a factor that must be considered.



- RAID is an acronym:
 - Redundant array of inexpensive disks
 - Most disks are now inexpensive
 - So “inexpensive” was replaced by “independent”
- Different level numbers do not imply ranking
 - They only distinguish one category from another
 - Originally, levels 0 through 5 were identified
- Not all levels duplicate data
- Not all levels provide redundancy

- Data Striping can improve performance
 - Files are split into smaller pieces (blocks)
 - The pieces are stored on different disks
 - For reads, all disks access their portions of the data
- Multiple categories or levels have been defined
 - Files are split into separate parts
 - The separate parts are on different disks
 - Multiple disks can be accessed at the same time
- Using multiple disks can yield greater reliability
 - Duplicate copies of data can be stored
 - Error correcting information can be included
 - Contents of failing disks can be reconstructed



RAID 0 provides no redundancy (low reliability)

Speeds access to large amounts of logically contiguous data

Supports multiple transactions that map to separate strips or blocks



Redundancy provided by duplicating each data disk

- Each data disk has a mirror image or shadow disk

If a disk fails, its mirror image is used

- copied to replacement disk

Disk spindles are not synchronized

Reads obtain data from first available copy

Writes must update both mirror images

Main disadvantage is the cost of fully duplicating the data

Levels 2 through 5 do not fully duplicate data

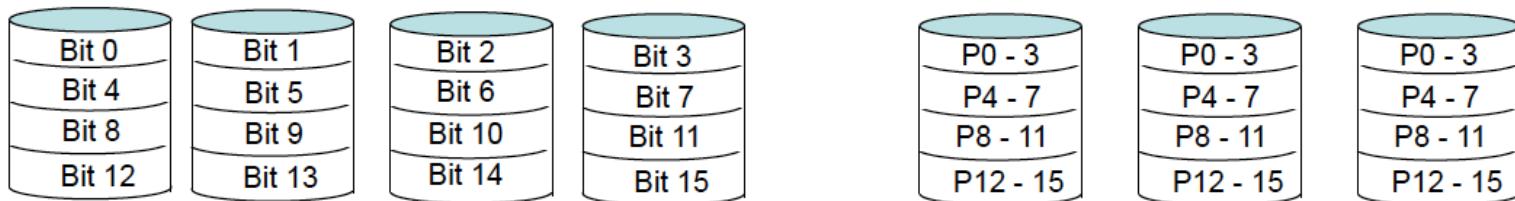
- Redundancy is provided by:
 - Error correcting information or
 - Parity

Levels 2 and 3 employ parallel access

- All disks in the array are accessed at the same time
- Heads on each disk move in unison
- Heads on each disk are at the same relative position

Levels 4 and 5 use independent access

- They differ in how the parity is distributed
- Spindles are not synchronized



Each access involves all disks

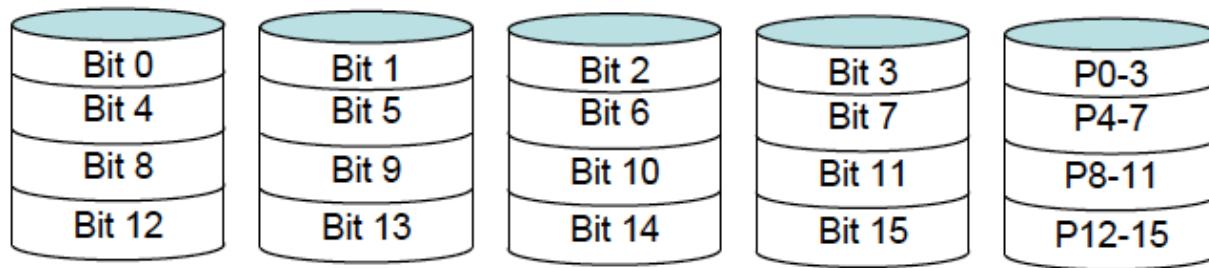
- Parity disks must be written along with data

Employs Hamming code for error correction

- Uses almost as much overhead as level 1
- # of parity disks is function of data disks
- For example: 3 error disks for 4 data disks

Nibble, byte or bit level striping

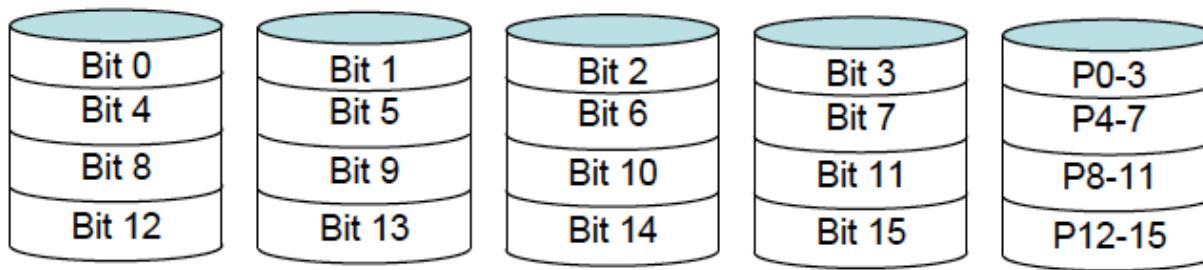
Not used commercially



Uses a single parity drive

- Parity based on simple XOR function
- Parity disks must be written along with data
- Parity drive creates a bottleneck
- Parity only used to reconstruct data
- Failures often involve just one disk

All strips within the same “row” constitute a stripe



Suppose drive containing bit3 fails:

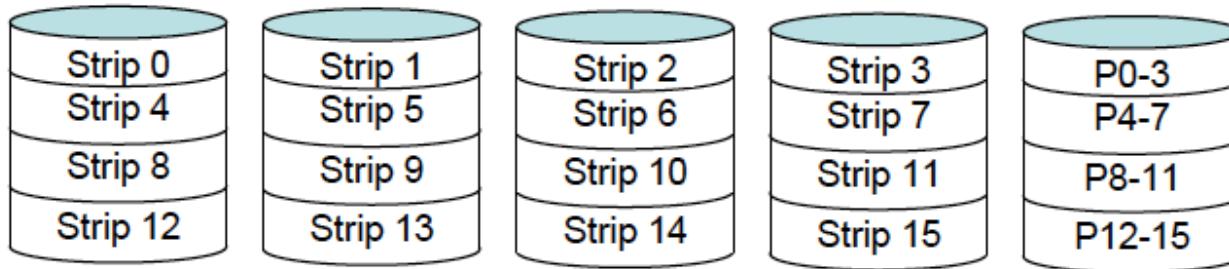
$$P0-3 = \text{bit0} \wedge \text{bit1} \wedge \text{bit2} \wedge \text{bit3} \quad (\wedge \text{ denotes XOR operator})$$

$$\text{bit3} \wedge P0-3 \wedge P0-3 = \text{bit0} \wedge \text{bit1} \wedge \text{bit2} \wedge \text{bit3} \wedge \text{bit3} \wedge P0-3$$

$$\text{bit3} = \text{bit0} \wedge \text{bit1} \wedge \text{bit2} \wedge P0-3$$

So bit3 can be reconstructed from the parity and remaining data disks
(entire contents of failed disk can be reconstructed this way)

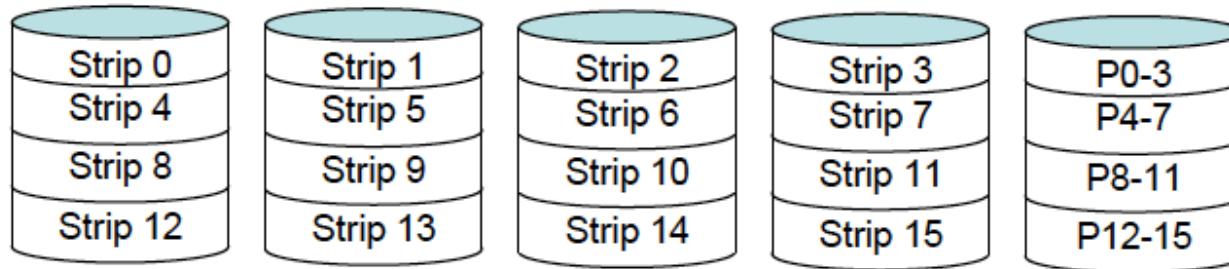
Writes or reads map to one stripe at a time



RAID4 uses block level striping (unlike RAID3 strips)
Access heads move independently (unlike RAID 3)

Example: strip0 contains bits 0 – 1023
strip1 contains bits 1024 – 2047, etc.

First bit of P0-3 = XOR of bits 0, 1024, 2048, etc.
(bitwise XOR of data strips)

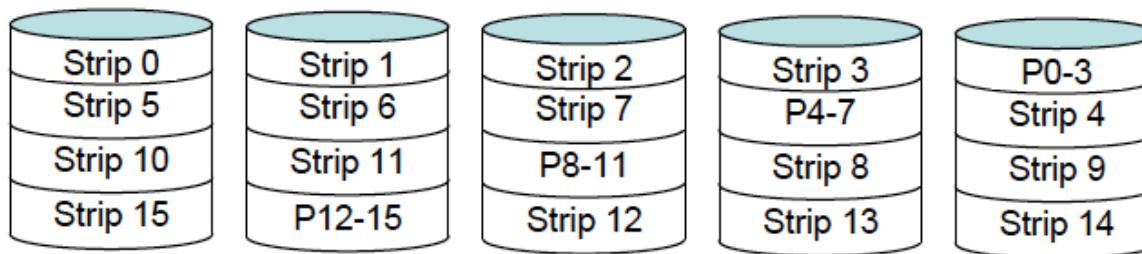


Parity disk must be written if any of the data strips are written

With parity on only one disk, the parity disk is a bottleneck

Tolerates at most one failing disk at a time

If a second disk fails before a replacement occurs, the system is inoperable



Uses block-interleaved distributed parity

Different stripes can be updated in parallel (no bottleneck)

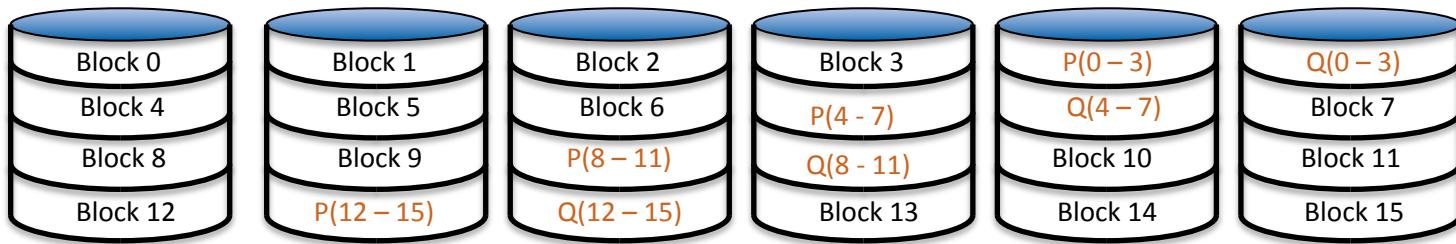
- Parity and data blocks are on separate drives

Most popular system

- good reliability
- good performance

RAID5 can only tolerate one failing disk at a time

Failed disk must be replaced before a second disk fails



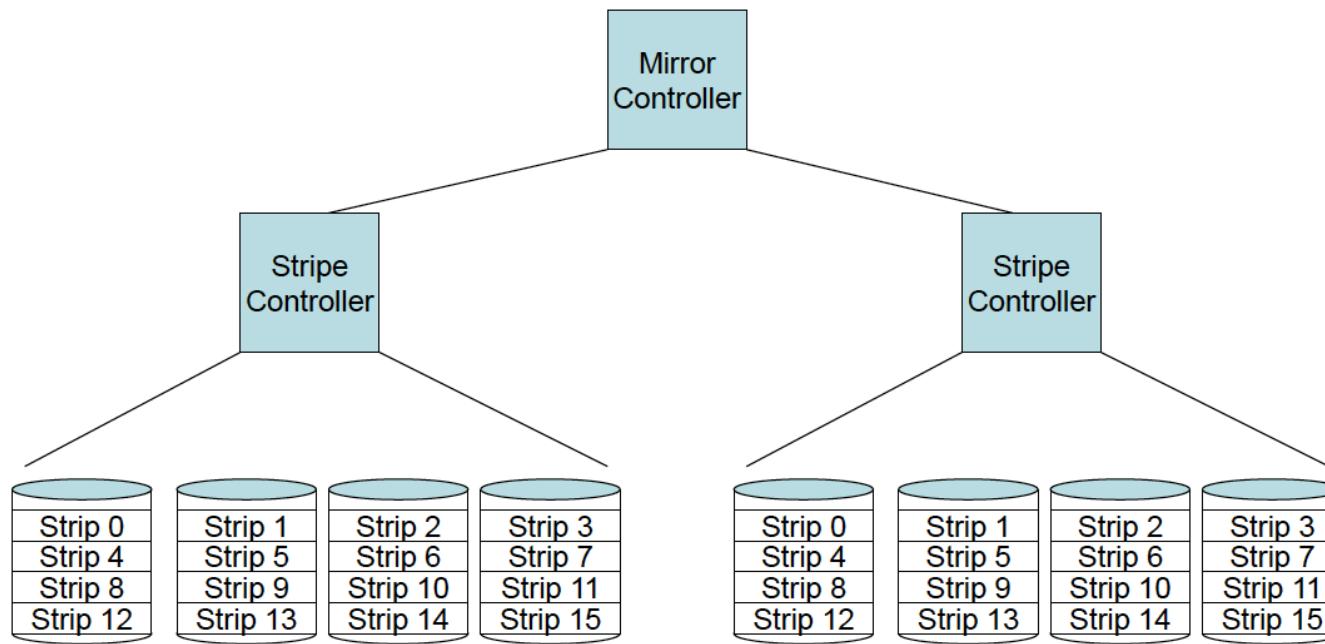
Uses dual-redundancy

- P is the same XOR function used in RAID5 and RAID4
- Q is based on a different error check scheme
 - Such as Reed-Solomon code
- missing data can be reconstructed from either parity

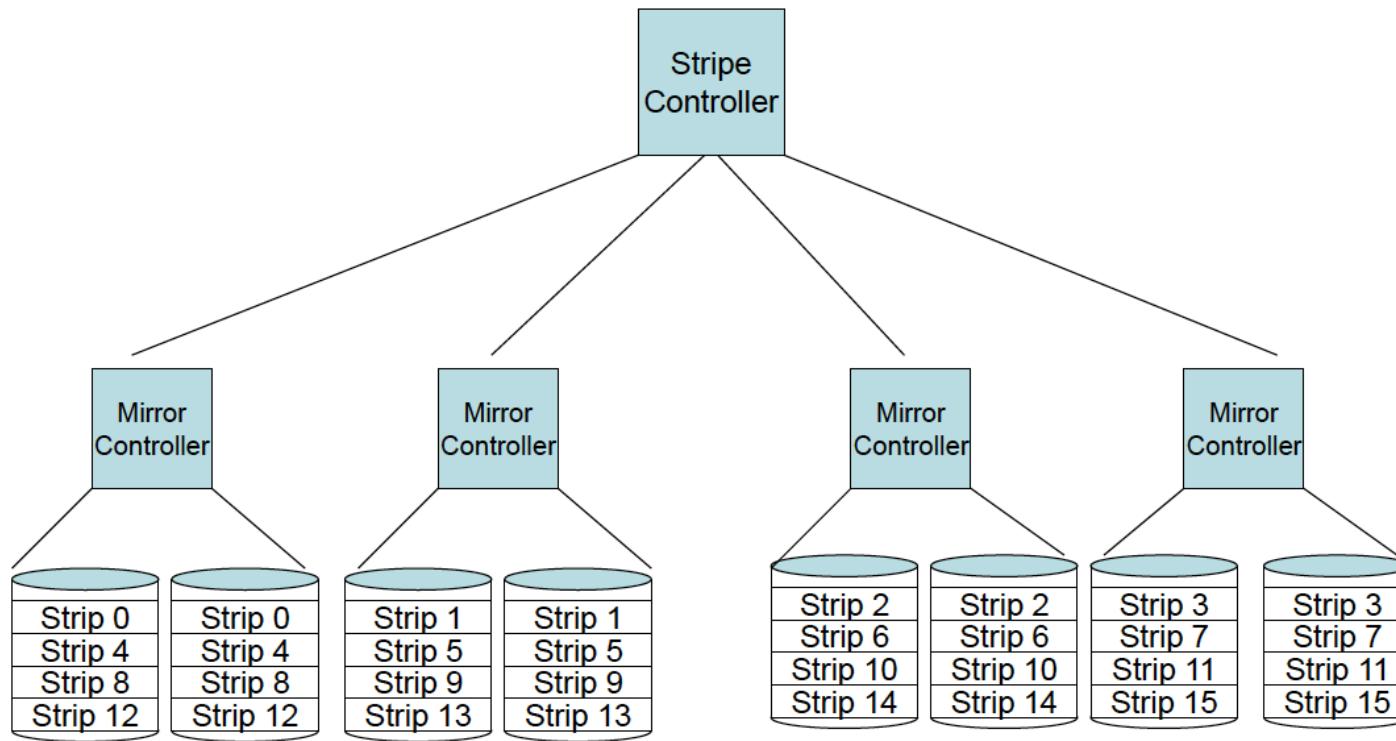
Tolerates up to two failing disks

Access heads on separate disks operate independently

I/O requests to separate disks can be handled in parallel



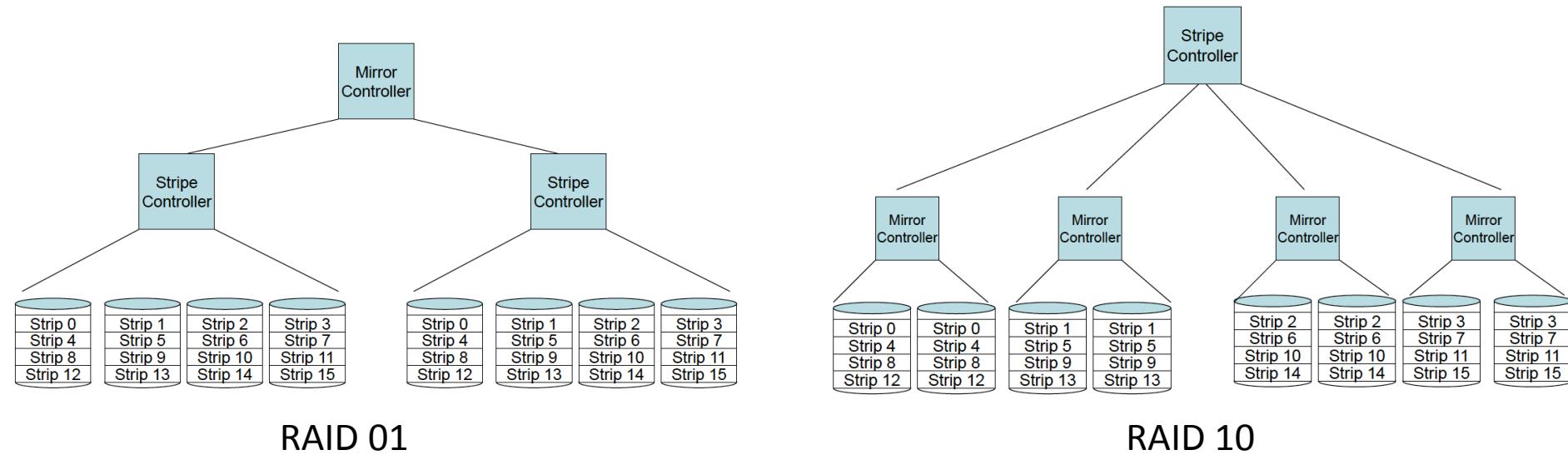
More recent systems combine levels: e.g. Mirrored RAID0 systems (01)
Each stripe controller makes its 4 disks appear as a single disk to mirror controller
mirror controller makes the mirrored pair appear as a single disk to the I/O system



Stripped Mirrored RAID0 systems (01)

Each mirror controller makes its 2 disks appear as a single disk to the stripe controller

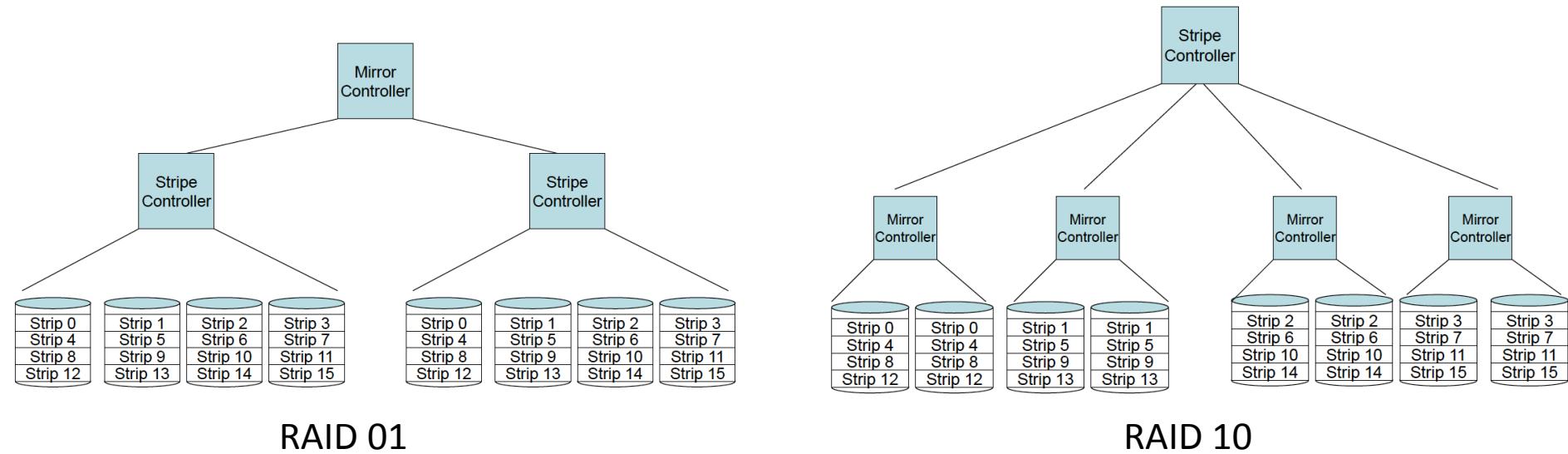
Stripe controller makes the 4 striped disks appear as a single disk to the I/O system



Expense:

3 controllers are needed for the level 01 system

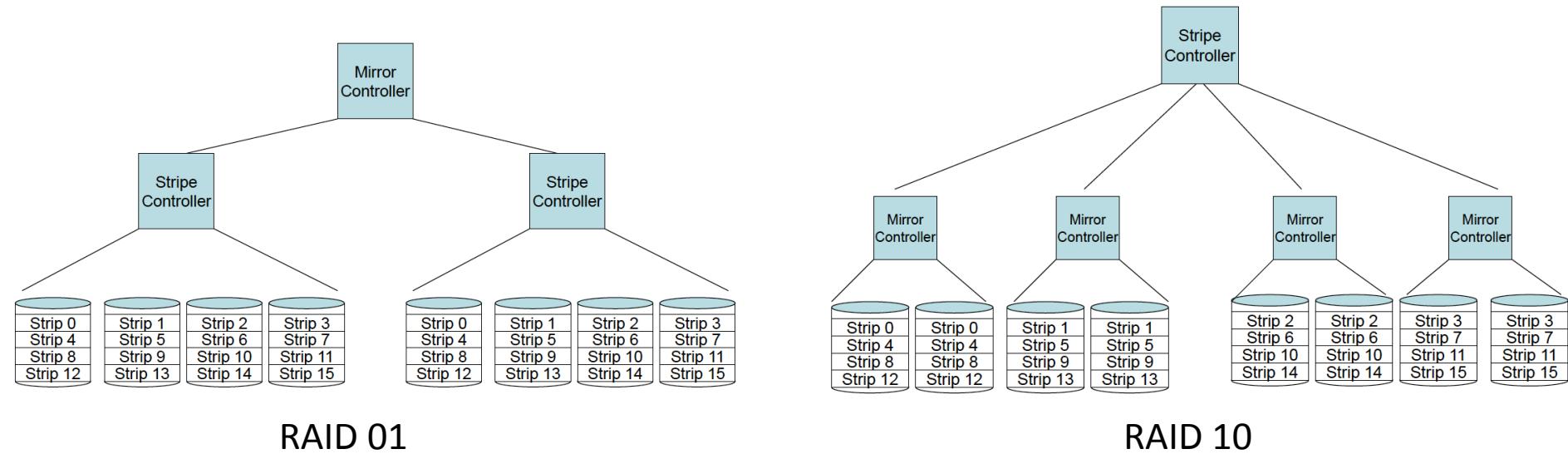
5 controllers are needed for the level 10 system



Expense:

3 controllers are needed for the level 01 system

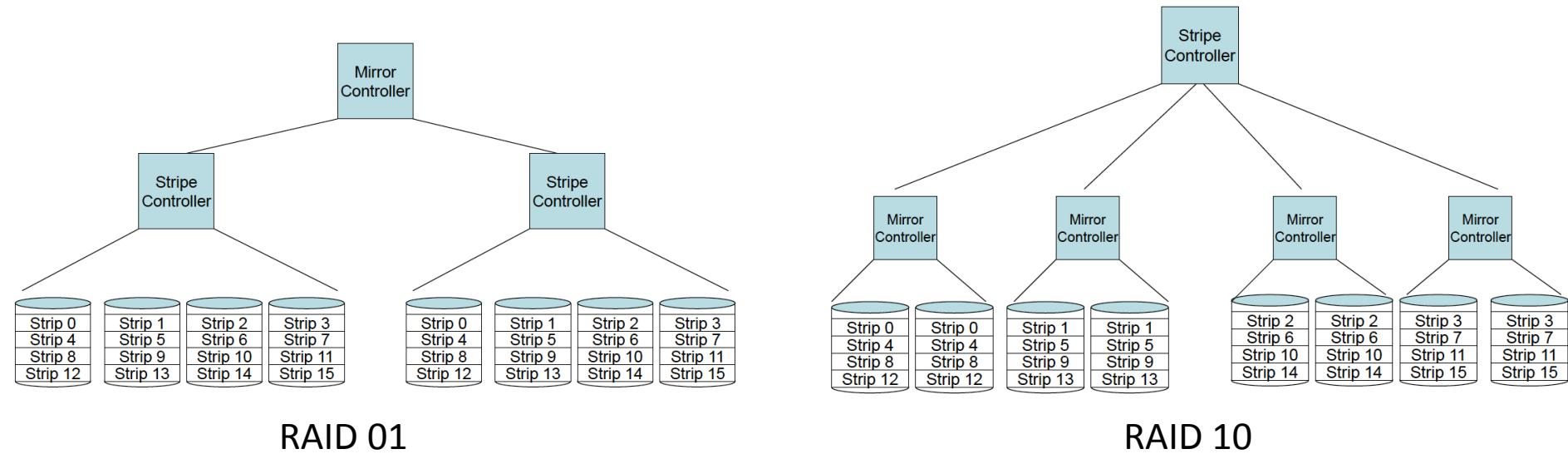
5 controllers are needed for the level 10 system



Reliability:

RAID01 is inoperable if a disk fails in each striped group
even if the two disks are not mirror images

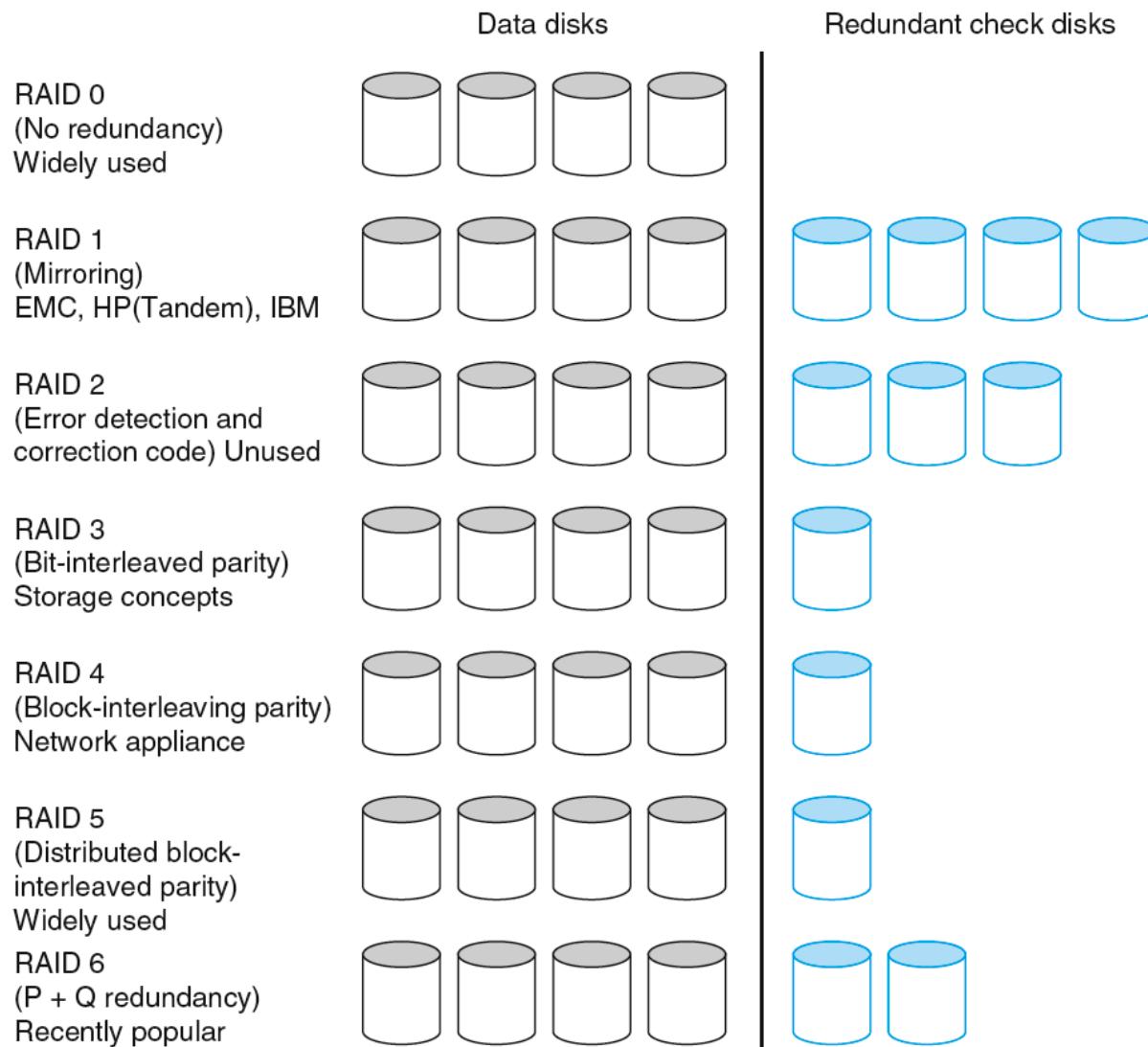
Both mirror images must fail for RAID10 to stop operating



Time required for restoration

All disks within a group would be re-written for RAID01

Only mirror image of failed disk would be re-written for RAID10



- Solid State Drives (SSD) employ Flash Memory
 - They mimic the electrical interfaces of hard drives
 - SSD's can plug into hard disk sockets
 - They provide higher performance,
 - lower weight & less power consumption
 - Greater tolerance to shock
 - They are more costly and have a lower storage capacity



- SSD's have no moving parts as with HDDs
 - They are truly random access
 - No rotational latency or seek delay
 - Response times are shorter
 - Yield higher input/output operations per second (IOPS)
 - Generate less heat
 - Provide quiet operation
 - Cost more per gigabyte of storage than HDDs
 - Fragmentation is not a problem as with hard drives

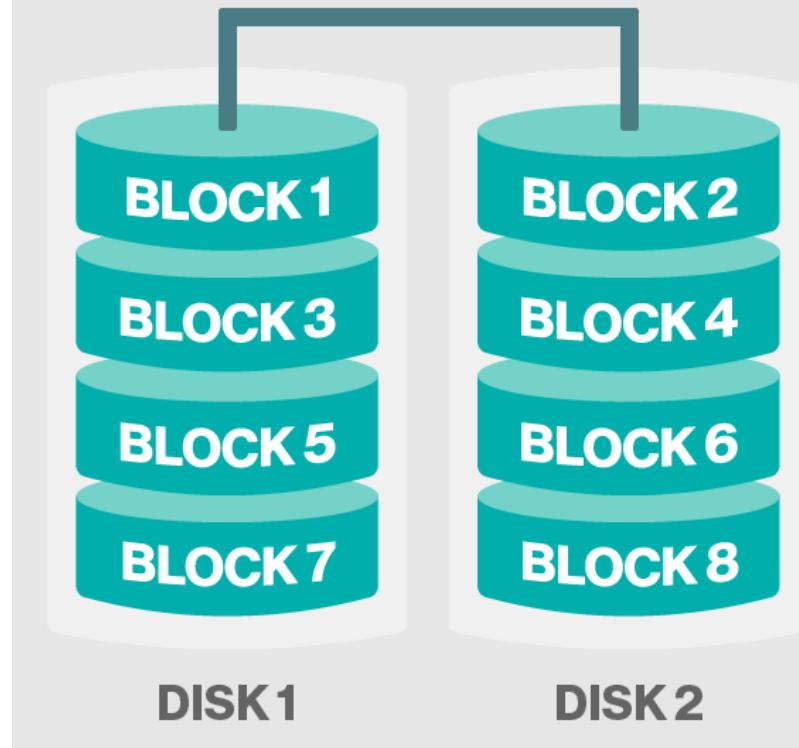
- Hard disks have effectively equal read, write and erase times
- SSD's use different read and write mechanisms
 - Writes take longer than reads
 - Write performance can be up to 50% lower than read performance
 - Blocks cannot simply be overwritten
 - they must first be erased and then written
- There is a limit to how many write cycles can be performed
- The cells fail after tens of thousands of write cycles
- Repeated write cycles may cause cells to become stuck at 0
- Wear leveling techniques are distributed the writes

- Wear leveling
 - Requested blocks are mapped onto physical block addresses
 - Controller monitors how often physical blocks are used
 - Adjusts mapping table to ensure all blocks share the load
 - This is required when SSD's are used as secondary storage
-
- Wear leveling is less of an issue for other applications
 - BIOS
 - Digital cameras
 - Music or video players

- Management of free space
 - Unused blocks that are erased can be used immediately
 - Unused blocks containing unneeded data must first be erased
- Entire blocks are erased (e.g. 4KB)
- SSD use is expanding with the declining cost of flash memory
- Their greater reliability means less frequent replacement

RAID 0

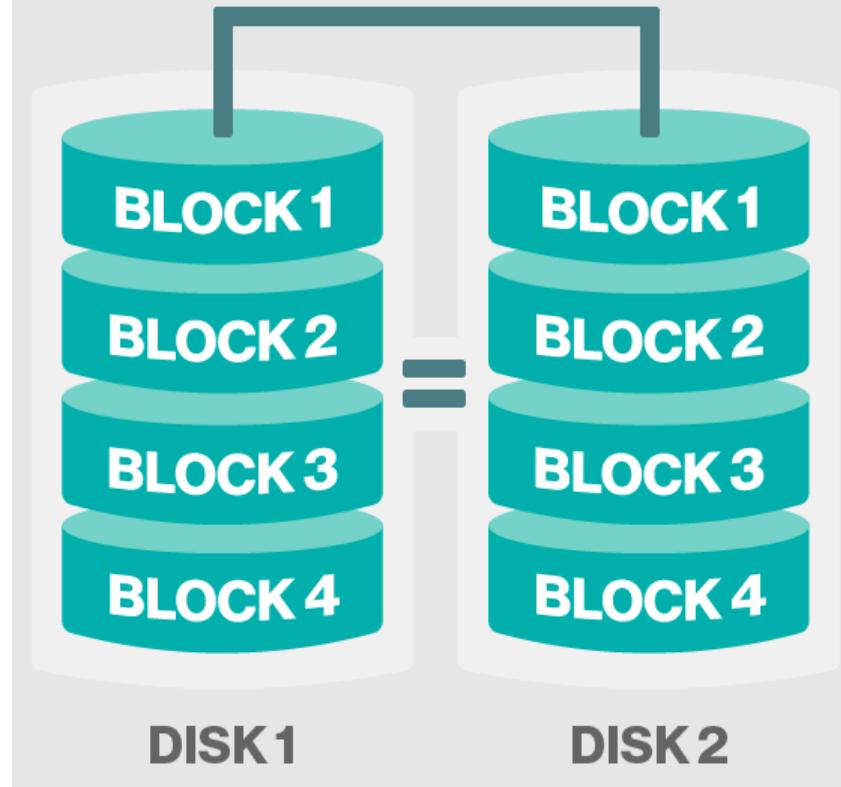
Striping



RAID 0: This configuration has striping but no redundancy of data. It offers the best performance but no fault-tolerance.

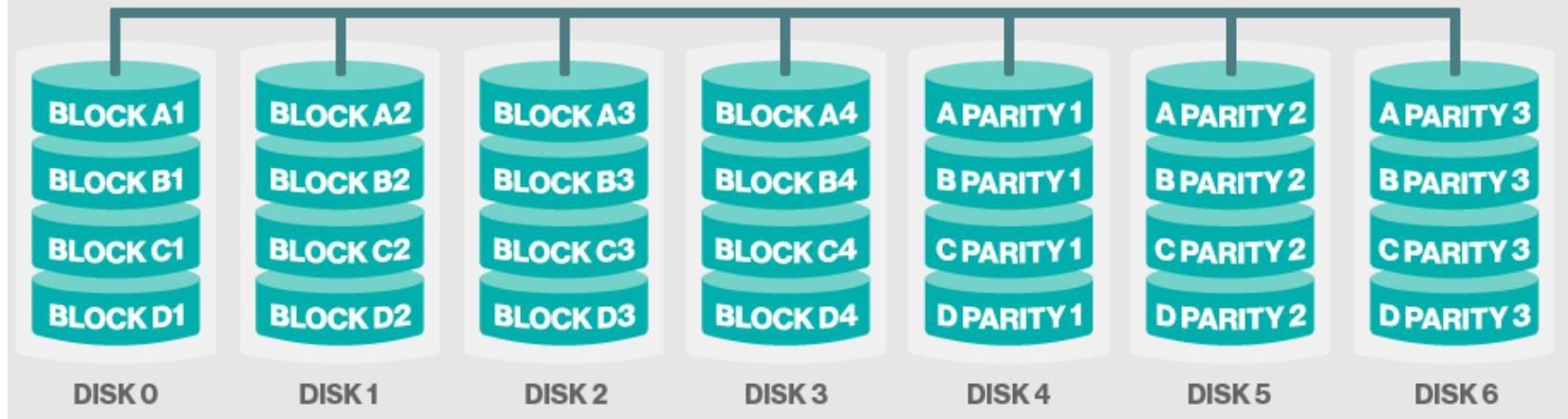
RAID 1

Mirroring



RAID 1: Also known as *disk mirroring*, this configuration consists of at least two drives that duplicate the storage of data. There is no striping. Read performance is improved since either disk can be read at the same time. Write performance is the same as for single disk storage.

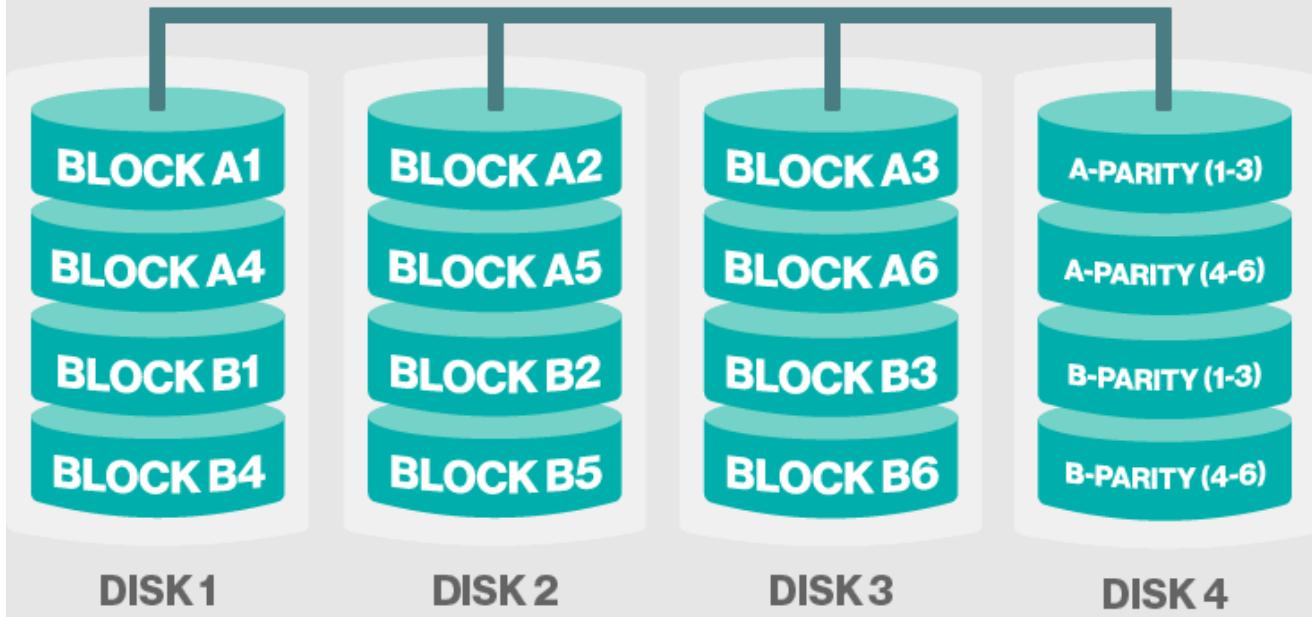
RAID 2



RAID 2: This configuration uses striping across disks with some disks storing error checking and correcting ([ECC](#)) information. It has no advantage over RAID 3 and is no longer used.

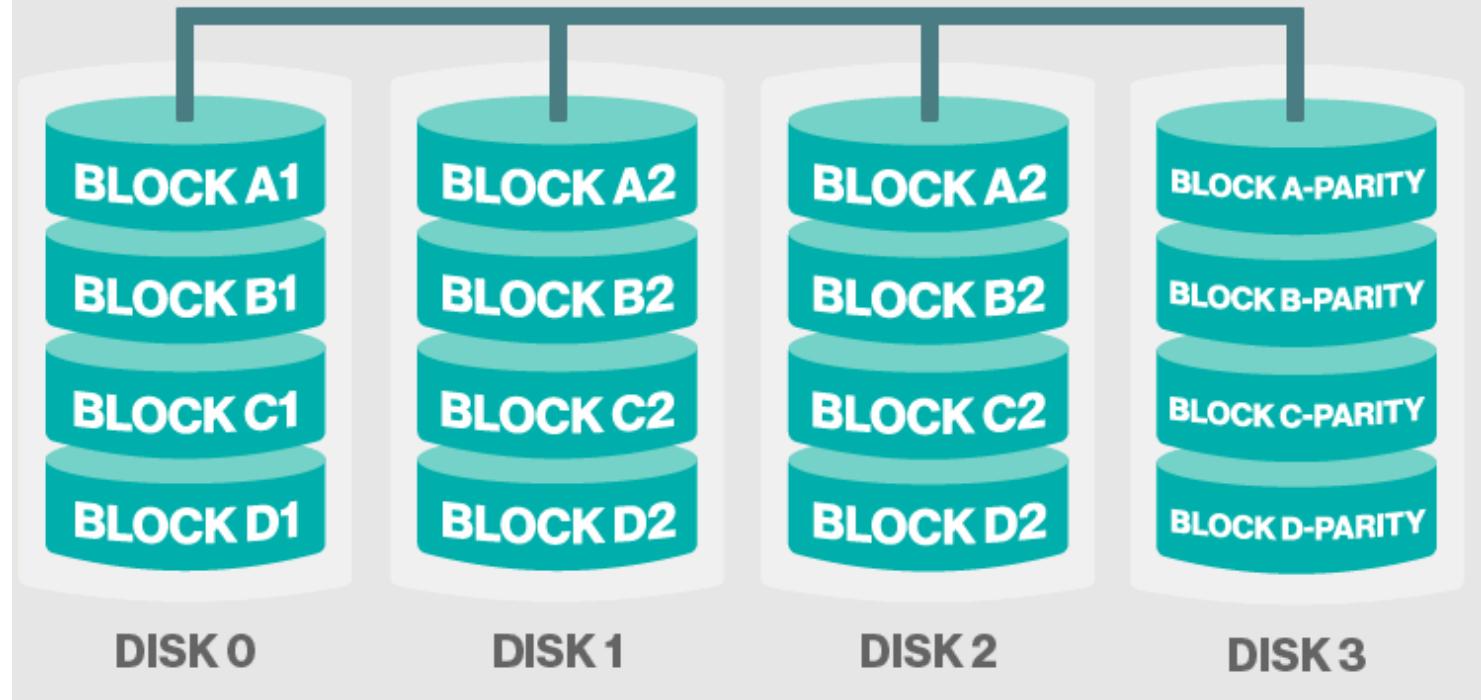
RAID 3

Parity on separate disk



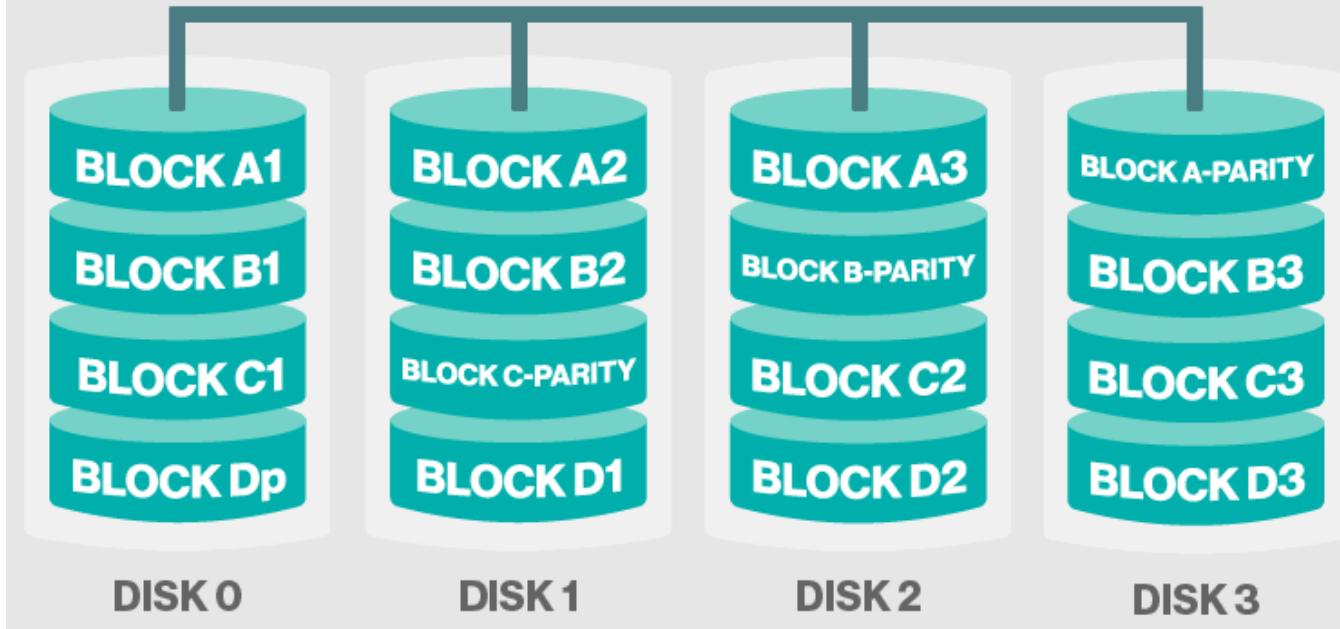
RAID 3: This technique uses striping and dedicates one drive to storing parity information. The embedded ECC information is used to detect errors. Data recovery is accomplished by calculating the exclusive OR (XOR) of the information recorded on the other drives. Since an I/O operation addresses all drives at the same time, RAID 3 cannot overlap I/O. For this reason, RAID 3 is best for single-user systems with long record applications.

RAID 4



RAID 4: This level uses large stripes, which means you can read records from any single drive. This allows you to use overlapped I/O for read operations. Since all write operations have to update the parity drive, no I/O overlapping is possible. RAID 4 offers no advantage over RAID 5.

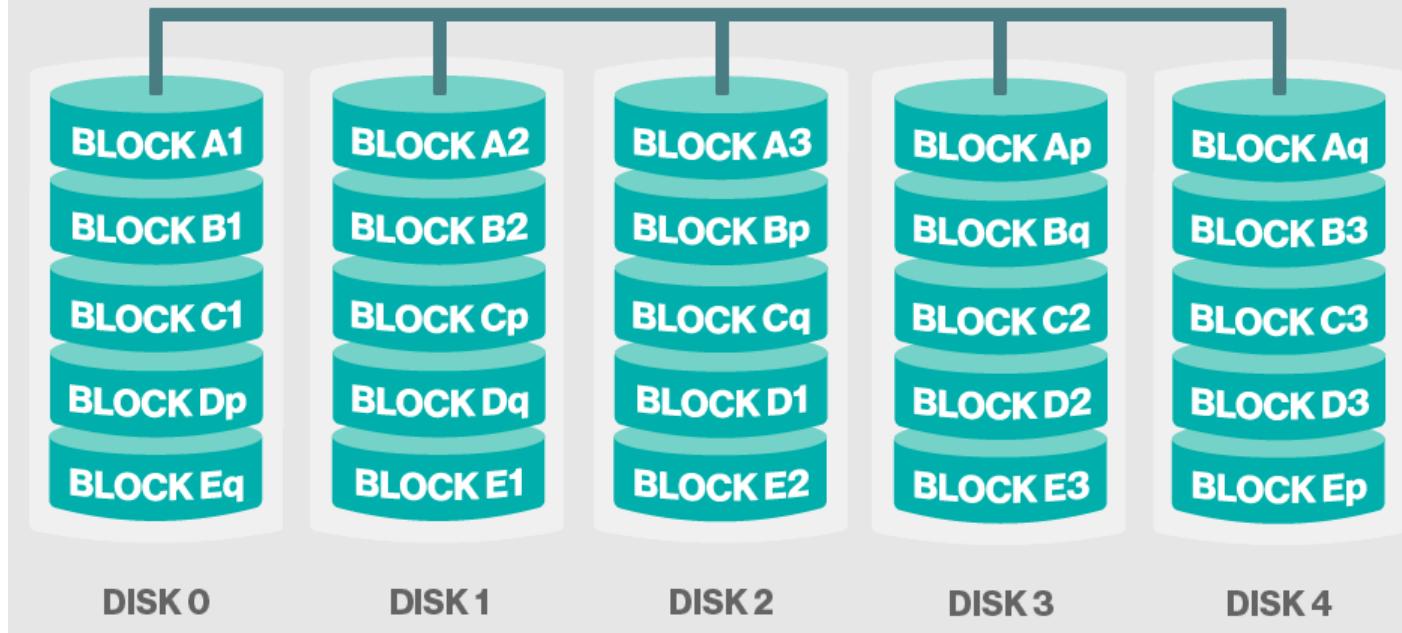
RAID 5



RAID 5: This level is based on [block](#)-level striping with parity. The parity information is striped across each drive, allowing the array to function even if one drive were to fail. The array's architecture allows read and write operations to span multiple drives. This results in performance that is usually better than that of a single drive, but not as high as that of a RAID 0 array. RAID 5 requires at least three disks, but it is often recommended to use at least five disks for performance reasons.

RAID 5 arrays are generally considered to be a poor choice for use on write-intensive systems because of the performance impact associated with writing parity information. When a disk does fail, it can take a long time to rebuild a RAID 5 array. Performance is usually degraded during the rebuild time and the array is vulnerable to an additional disk failure until the rebuild is complete.

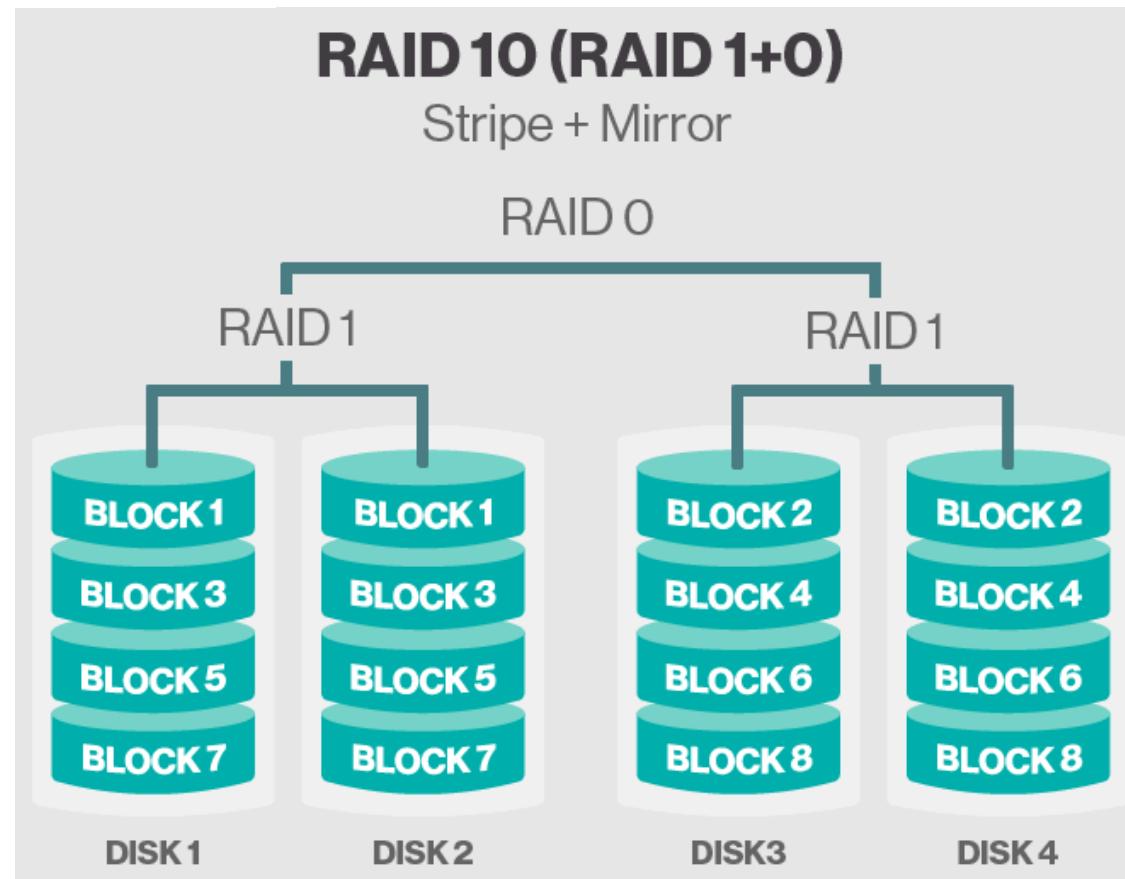
RAID 6



RAID 6: This technique is similar to RAID 5 but includes a second parity scheme that is distributed across the drives in the array. The use of additional parity allows the array to continue to function even if two disks fail simultaneously. However, this extra protection comes at a cost. RAID 6 arrays have a higher cost per gigabyte ([GB](#)) and often have slower write performance than RAID 5 arrays.

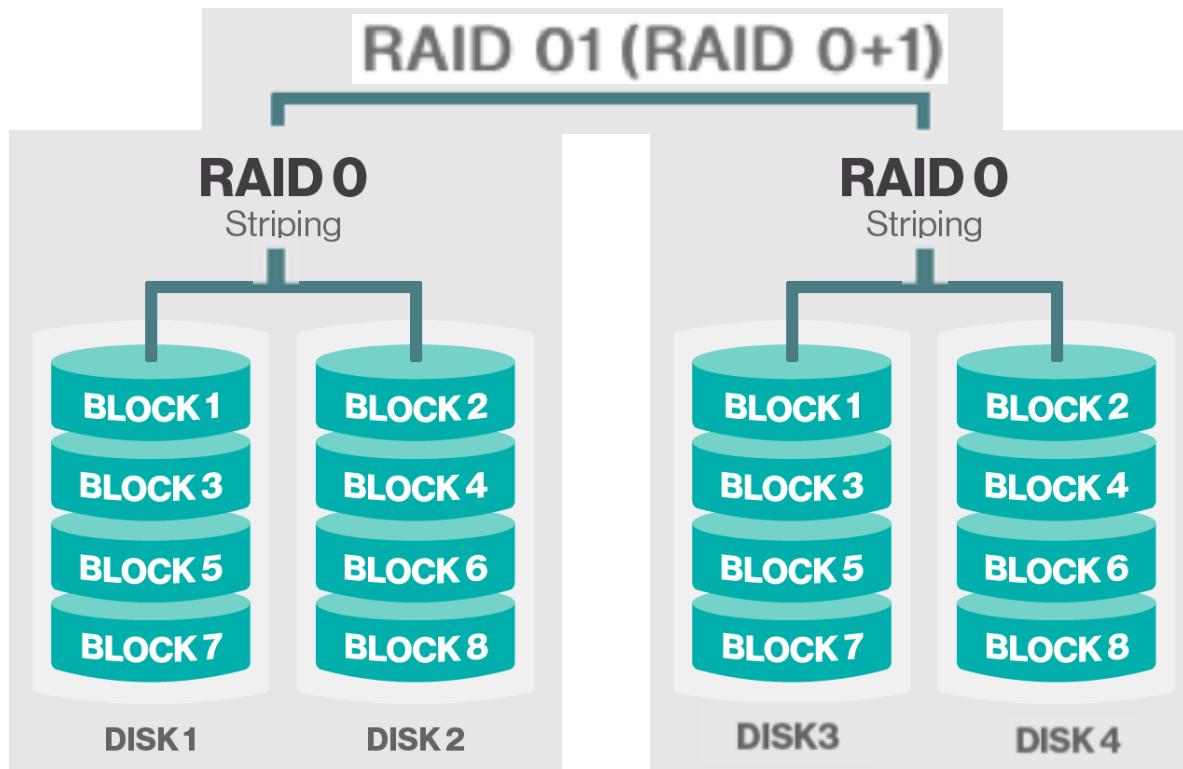
Nested RAID levels

Some RAID levels are referred to as *nested RAID* because they are based on a combination of RAID levels.



RAID 10 (RAID 1+0): Combining RAID 1 and RAID 0, this level is often referred to as RAID 10, which offers higher performance than RAID 1 but at a much higher cost. In RAID 1+0, the data is mirrored and the mirrors are striped.

Nested RAID levels



RAID 01 (RAID 0+1): RAID 0+1 is very similar to RAID 1+0, except the data organization method is slightly different. Rather than creating a mirror and then stripping the mirror, RAID 0+1 creates a stripe set and then mirrors the stripe set.



Parallel Systems perform multiple actions at the same time

Multitasking systems execute programs in parallel

Example: surfing the web while playing music files
or editing a document while updating a spreadsheet

Switching frequently between sequentially executing tasks

Gives the illusion of continuous execution

Tasks execute concurrently rather than simultaneously

The tasks may run on the same processor

Pipelining overlaps multiple instructions

Example of fine grained instruction level parallelism (ILP)

Overlaps instruction steps (fetch, decode, execute, etc.)

Scalar pipeline contains just one execution unit

Superscalar processors allow true parallel processing

Takes place within the CPU

Executes separate instructions simultaneously

Invisible to the user

Compiler assists in supplying more instructions

Separate execute units may each be pipelined

Hyperthreading is another form of parallelism

Single execute unit switches between threads

Registers & PC are replicated (one set per thread)

Thread switches cause a different register set to be used

Hides memory latency caused by cache misses

True parallel processing uses multiple execution units

The units run at the same time

Each executing a separate thread

These are called multiprocessor systems

Multi-core systems have 2 or more processors on a single chip
required resources are shared:

Memory interface

Cache control

Interconnect systems

More cost effective than single sophisticated processors
Increase performance without greater complexity
Use lower clock rates, thus less power

In parallel systems, groups of processors work together
topology defines the way the processors are interconnected

The collaborating processors must communicate

There are two options:

Shared bus

Shared interconnection network

Bus-based multiprocessors share memory

Processors can access another's memory directly

Message-passing multiprocessors use communication links such as ethernet or other proprietary high speed links

The degree of coupling differs for bus-based vs. message-passing
Coupling is far higher for bus-based systems
Bus-based systems have high bandwidth but are expensive
Message-passing systems are less complicated

Bus-based systems are harder to scale
access to the shared memory becomes harder to manage
These are referred to a “tightly coupled”
Versus the “loosely coupled” message-passing systems

Memory systems greatly affect multiprocessor performance

Uniform memory access (UMA)

- Equally accessible by all processors

- Used with smaller tightly coupled bus-based systems

NUMA (non-uniform memory access)

- memory is not homogeneous

- not all memory is equally accessible to all processors

- Used in large message passing multiprocessors

- Referred to as loosely coupled

Using N processors may not provide a speedup of N

Amdahl's law still applies

Code must contain independent parts

Each independent part can run on a different processor

Sequential part does not benefit from extra processors

The size of the problem or task makes a difference

Best to use more processors on larger problems

Processors will be idle unless they have work to do

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?

$$T_{\text{old}} = T_{\text{sequential}} + T_{\text{parallelizable}}$$

$$T_{\text{new}} = T_{\text{sequential}} + \frac{T_{\text{parallelizable}}}{100}$$

$$\text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{1}{\frac{T_{\text{new}}}{T_{\text{old}}}} = \frac{1}{\frac{T_{\text{sequential}}}{T_{\text{old}}} + \frac{T_{\text{parallelizable}}}{T_{\text{old}} * 100}}$$

$$\text{Speedup} = \frac{1}{f_{\text{sequential}} + \frac{f_{\text{parallelizable}}}{100}}$$

$$\text{Speedup} = \frac{1}{(1 - f_{\text{parallelizable}}) + f_{\text{parallelizable}} / 100}$$

$$\text{Speedup} = \frac{1}{(1 - f_{\text{parallelizable}}) + f_{\text{parallelizable}} / 100} = 90$$

$$\text{Speedup} = \frac{1}{1 - 0.99 * f_{\text{parallelizable}}} = 90$$

$$f_{\text{parallelizable}} = \frac{1 - \frac{1}{90}}{0.99} = 0.999$$

So sequential part can only be 0.1% of the total.

An SMP system contains 8 processors.

A program consists of a startup sequential section
that produces results used in remaining parallel part

Desired speedup = 8/3
relative to executing the program on a single processor

Parallel part must be what percent of the total code?

Let f = fraction of the code corresponding to parallel part
Based on definition of speedup:

$$\frac{1}{(1-f) + \frac{f}{8}} = \frac{8}{3} \quad \longrightarrow \quad 1 - \frac{7f}{8} = \frac{3}{8}$$

$$\frac{7f}{8} = \frac{5}{8} \quad \longrightarrow \quad f = \frac{5}{7} = 0.7143$$

71.43% of the code must be parallelizable

Strong Scaling:

using more processors on a given size problem

Weak Scaling:

Increasing the number of processors with problem size

Good speedup is more difficult with strong scaling

Extra processors may sit idle unless problem size grows

Example:

A program computes the sum of two 10element vectors
and the sum of two 10x10 matrices

Each addition takes 1 cycle

10 processors are available

The vector sum is computed by 1 processor

The matrix sum is split among 10 processors

The potential speedup is a factor of 10

Total time using 1 processor = $10 + 100 = 110$ cycles

Time using 10 processors = $10 + 100/10 = 20$ cycles

Speedup = $110/20 = 5.5$

Achieves 55% of the potential speedup

Example:

Suppose 40 processors are used instead

The potential speedup is a factor of 40

Total time using 1 processor = $10 + 100 = 110$ cycles

Time using 40 processors = $10 + 100/40 = 12.5$ cycles

Speedup = $110/12.5 = 8.8$

Achieves only 22% of the potential speedup

Example:

Suppose matrix size grows to 20x20

Total cycles using 1 processor = $10 + 400 = 410$

Cycles using 10 processors = $10 + 400/10 = 50$

Cycles using 40 processors = $10 + 400/40 = 20$

Speedup with 10 processors = $410/50 = 8.2$

Achieves 82% of the potential speedup

Speedup with 40 processors = $410/20 = 20.5$

Achieves 51.25% of the potential speedup

The size of the problem affects the speedup

Amdahl's Law for multi-processors

Let T_1 be the execution time for a program on a single processor

f = fraction of time due to the parallel part split N ways

T_N is the execution time using N processors

$$T_N = [(1 - f) + \frac{f}{N}] T_1$$

Adding extra cores only improves the parallel part

$$\text{Speedup} = \frac{T_1}{T_N} = \frac{1}{(1 - f) + \frac{f}{N}} < \frac{1}{(1 - f)}$$

Indicates an upper limit on the speedup (strong scaling)

Gustafson's Law applies when N increases with problem size

More processors can be used with larger workloads

TN is the execution time using N processors

f = fraction of TN used for parallel part on N-processor system

Gustafson's law states:

$$\text{Speedup} = (1 - f) + f * N$$

Speedup varies linearly with f (weak scaling)

As workload expands, parallel part becomes a larger fraction of TN

Classification based on instruction and data streams

- SISD – single instruction & single data stream
 - MISD – multiple instruction streams & one data stream
 - SIMD – single instruction stream & multiple data streams
 - MIMD – multiple instruction streams & multiple data streams
-
- Flynn introduced this scheme in a 1972 paper
 - Does not cover topology or interconnection techniques



An instruction stream is a sequence of instructions
Fetched using a single program counter register

- SISD systems execute instructions sequentially
- Pipelining or multiple execute units may be used

Data operands are obtained one at a time from memory
as a single stream of values

A single instruction stream is applied to multiple data

- Includes vector type instructions
- MMX multi-media extensions (Intel processors)
- A single control unit fetches and decodes instructions
- Multiple PEs act on separate operands in parallel
 - PEs are processing elements

Operands can be sub-sets of bits within a word or register



Multiple instruction streams act on one data stream

- Few, if any, systems follow this model

A possible use is in highly fault tolerant systems

- Multiple processors produce results from the same data
- Majority vote determines the result used
- Example use is in fly-by-wire aircraft



Multiple instruction streams act on multiple data streams

- Multiple instructions are fetched and decoded in parallel
- Instructions may be vector or scalar type
- Each control unit directs one or more execute units (PEs)

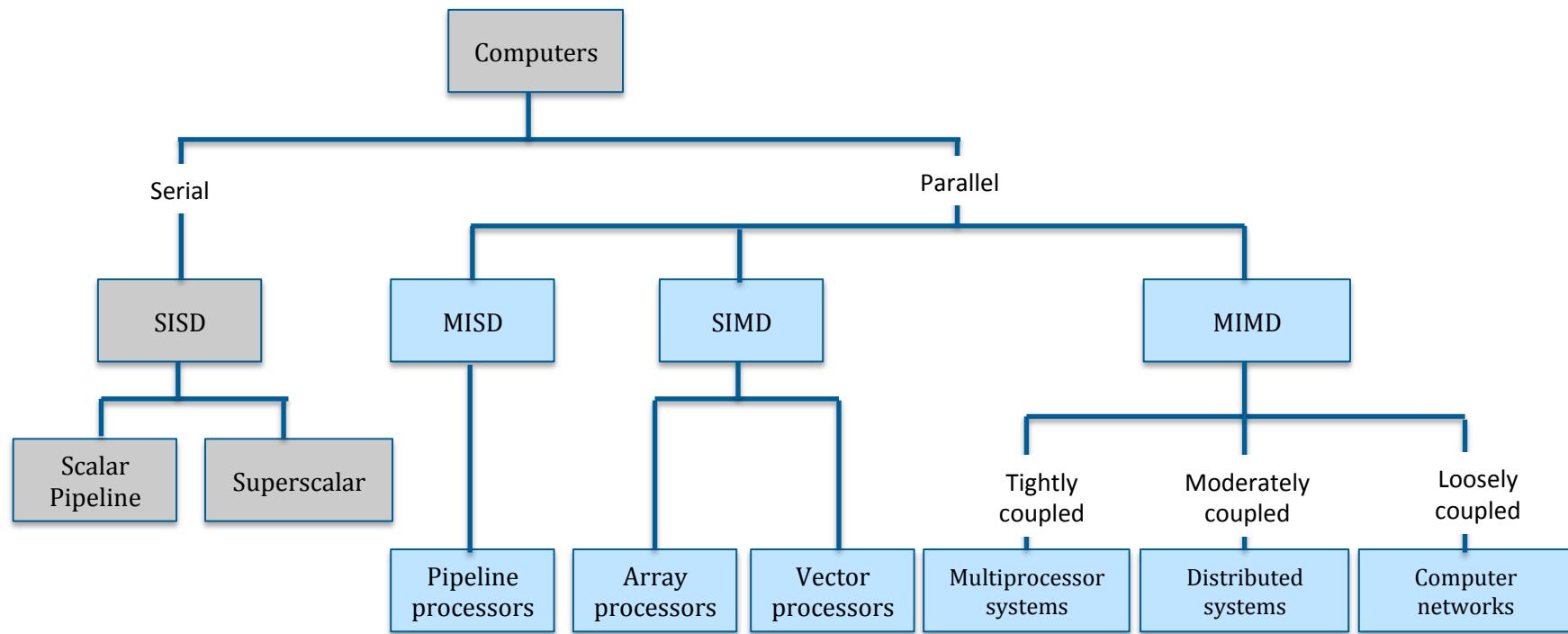
Based on shared memory or on message passing

- May employ board level or chip level multiprocessors
- Chip level multiprocessors are also called multi-core
- Distributed systems require a high speed interconnect
- The interconnection scheme determines the topology

Collaborating programs can run on different processors

- One program can run on all processors of a MIMD system
- Conditional statements determine when different processors should execute different sections of the program
- This is known as SPMD (single program multiple data)

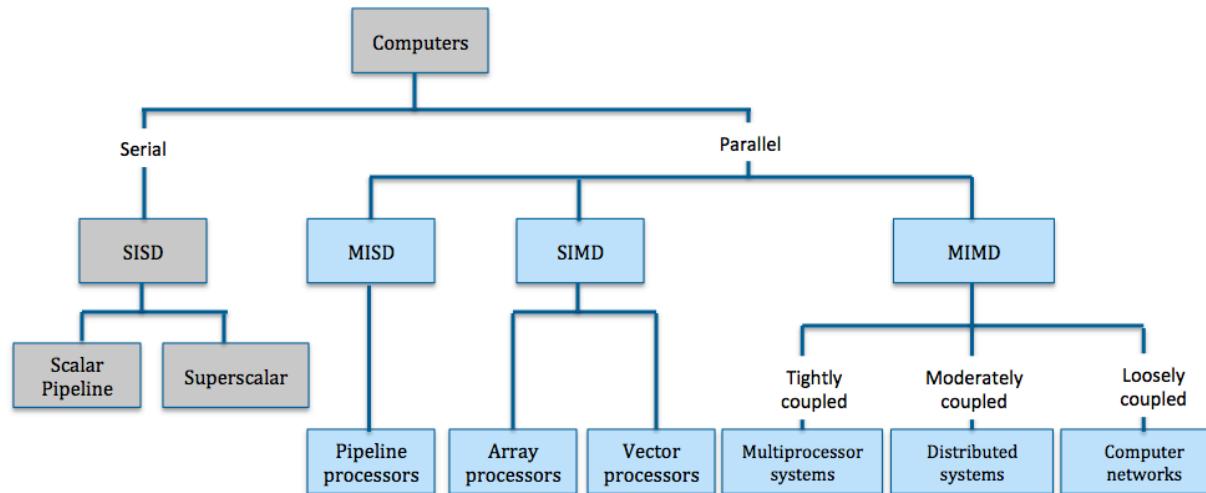
		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Core i7	Windows Vista Operating System running on an Intel Core i7



Multiprocessors can contain thousands of PEs

- Typically many fewer PEs are used due to coupling issues
- Goal is to efficiently increase performance
 - Avoid extremely high clock rates
 - Reduce heat and power dissipation

Coupling describes the degree of interconnection



- Loosely coupled systems employ message passing
 - Exchange data slowly relative to CPU clock speed
 - Connecting via the internet is one extreme example
 - Granularity of data is high (entire files)
- Tightly coupled systems use shared memory
 - Provide high data bandwidth and low latency
 - Use fine grained granularity
 - Processors can operate on the same data structure
 - For example: on different elements within an array

- There are issues related to sharing memory
 - Synchronization
 - Cache consistency
 - Access order
- These issues will be addressed later

A vector computer is a SIMD machine

Single vector instructions operate on multi-element data items

Commands are broadcast to the processing elements (PE)

PE's operate in lock-step fashion performing the same operation

Example: $Z = s*X + Y$ where Z, X and Y are arrays or vectors
s is a scalar constant or variable

Intel-based PC's have MMX, SSE & AVX instructions

Multi-media extensions (integer)

Streaming SIMD extensions (floating point)

Advanced Vector Extensions (AVX)

Operates on four 64-bit floating point numbers in parallel

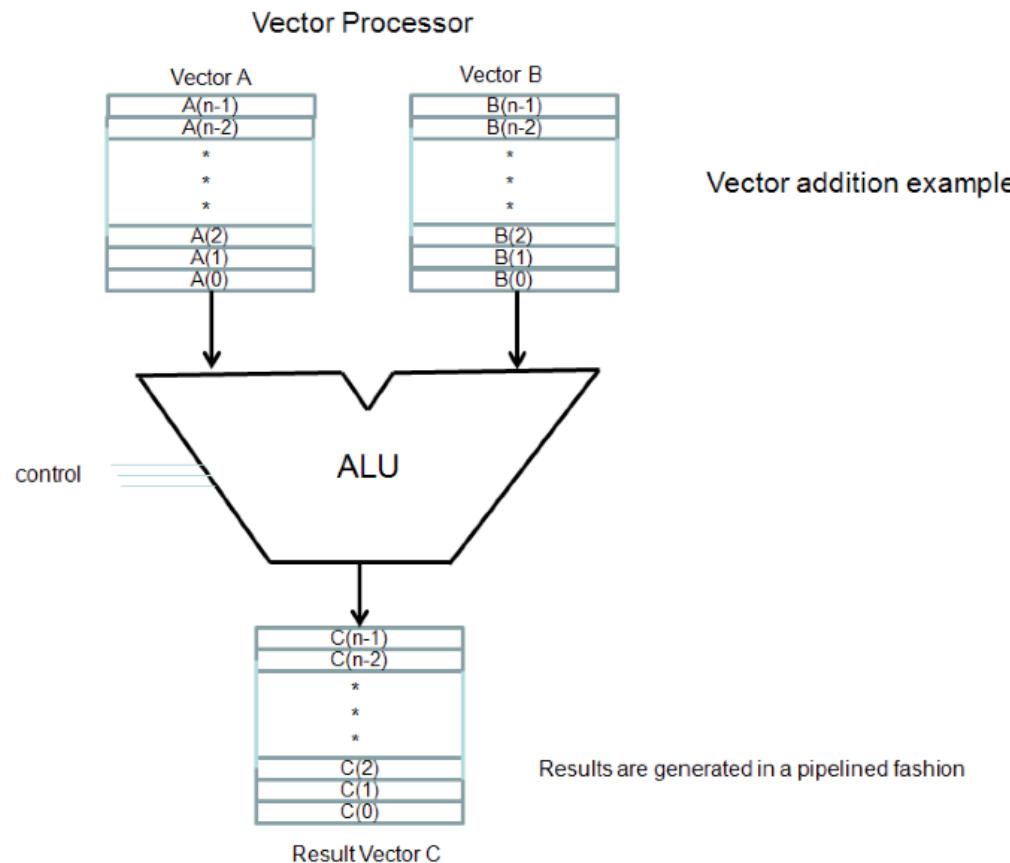
Only supercomputers had vector instructions in the past
the Cray-1 designed by Seymour Cray in the 70s

- Highly pipelined functional units
- Data get streamed to and from vector registers
 - Data collected from memory into registers
 - Results stored from registers to memory

Basic idea:

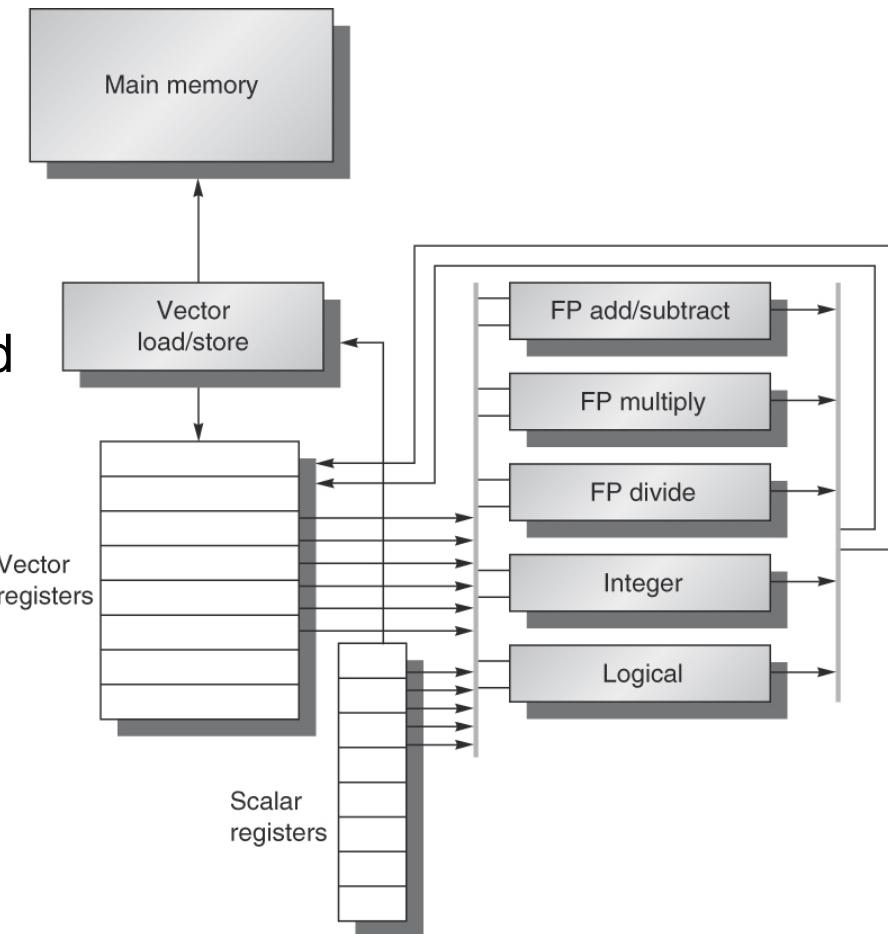
- Read sets of data elements into “vector registers”
- Operate on those registers
- Disperse the results back into memory

High memory bandwidth is required to quickly fill registers

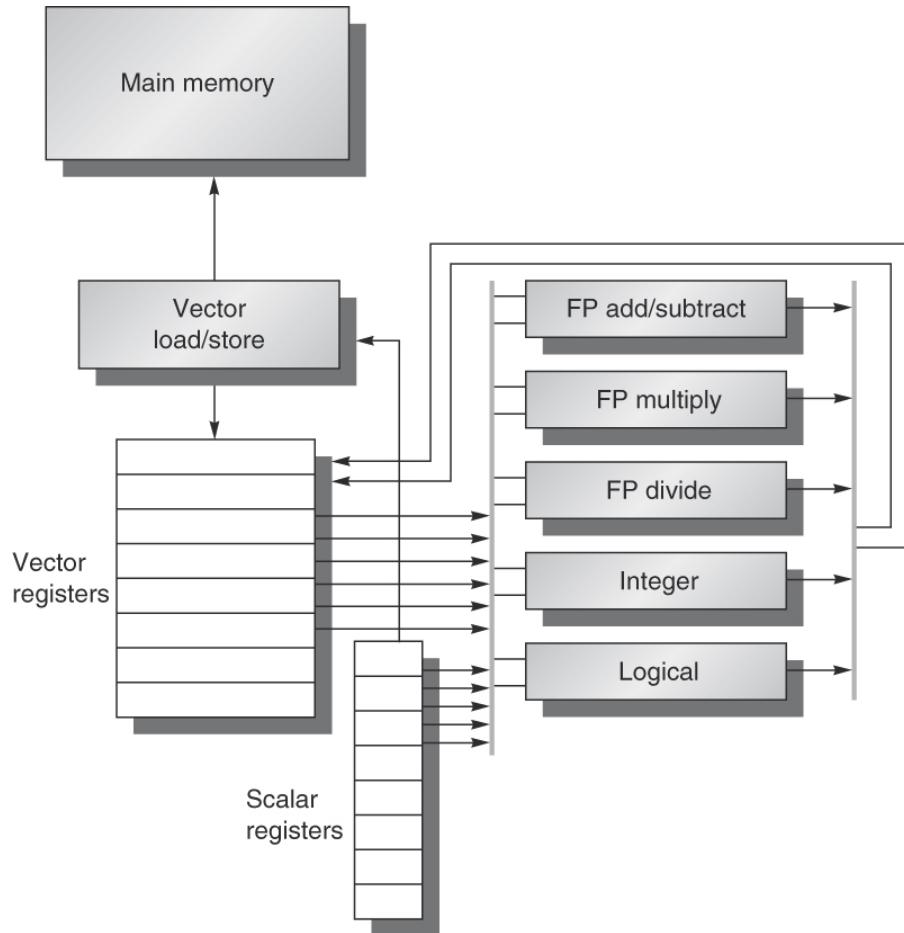


- Vector registers

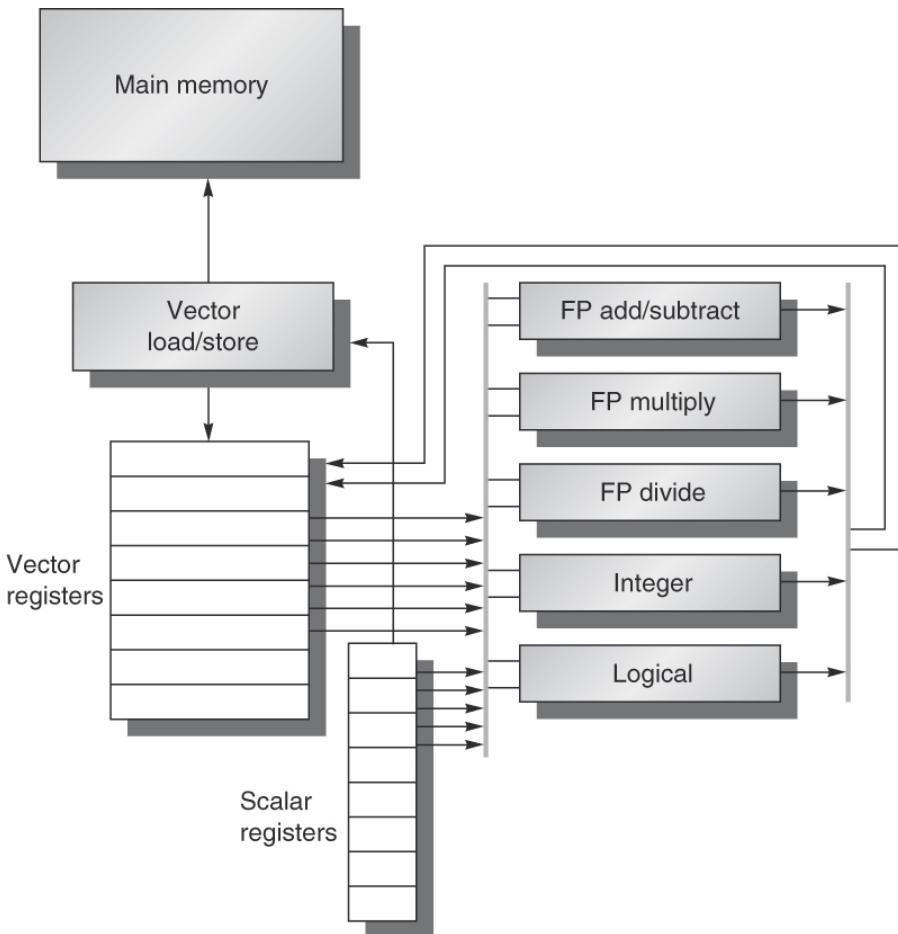
- Each register holds a 64-element, 64 bits/element vector
- Register file has 16 read ports and 8 write ports



- Vector functional units
 - Fully pipelined
 - Data and control hazards are detected



- Vector load-store unit
 - Fully pipelined
 - Words move between registers
 - One word per clock cycle after initial latency
- Scalar registers
 - 32 general-purpose registers
 - 32 floating-point registers



```
# C code
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];
```

# C code	# Scalar Code	# Vector Code
	LI R4, 64	LI VLR, 64
	loop:	LV V1, R1
	L.D F0, 0(R1)	LV V2, R2
	L.D F2, 0(R2)	ADDV.D V3, V1, V2
	ADD.D F4, F2, F0	SV V3, R3
	S.D F4, 0(R3)	
	DADDIU R1, 8	
	DADDIU R2, 8	
	DADDIU R3, 8	
	DSUBIU R4, 1	
	BNEZ R4, loop	

- Example: DAXPY

```
L.D      F0,a      ;load scalar a
LV       V1,Rx      ;load vector X
MULVS.D V2,V1,F0  ;vector-scalar mult
LV       V3,Ry      ;load vector Y
ADDVV   V4,V2,V3  ;add
SV       Ry,V4      ;store result
```

- In MIPS Code

- ADD waits for MUL, SD waits for ADD

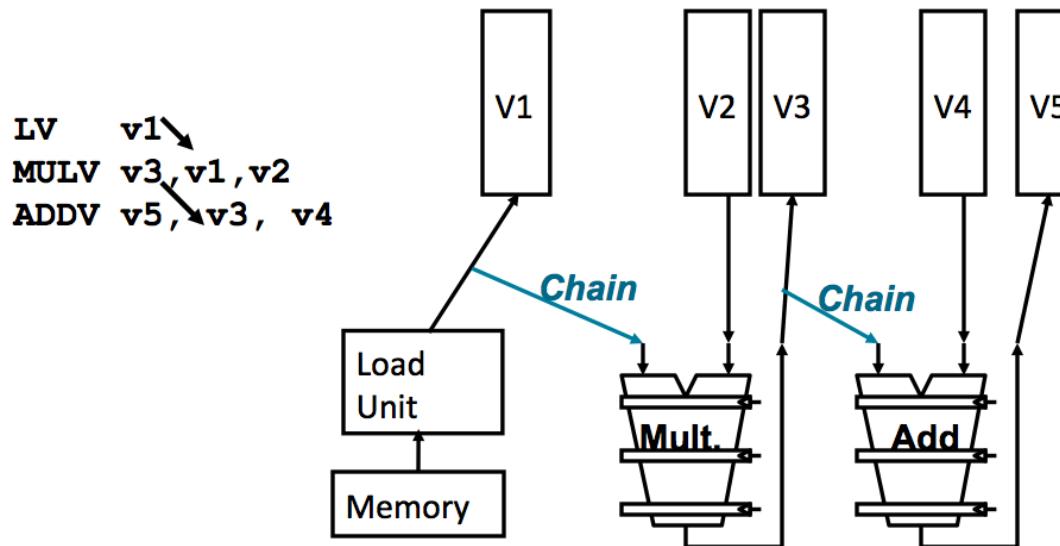
- In VMIPS

- Stall once for the first vector element, subsequent elements will flow smoothly down the pipeline.
 - Pipeline stall required once per vector instruction!

- Execution time depends on three factors:
 - Length of operand vectors
 - Structural hazards
 - Data dependencies
- VMIPS functional units consume one element per clock cycle
 - Execution time is approximately the vector length

Input operands are used as soon as they arrive from memory
Averts stalling until the entire vector register is filled

Results can be sent from one functional unit directly to another
A vector version of register bypassing and forwarding



Chaining overlaps the loading of operands into vector registers
The next operands are read in parallel with the use of the previous set

The vector length register (VLR) specifies number of reads

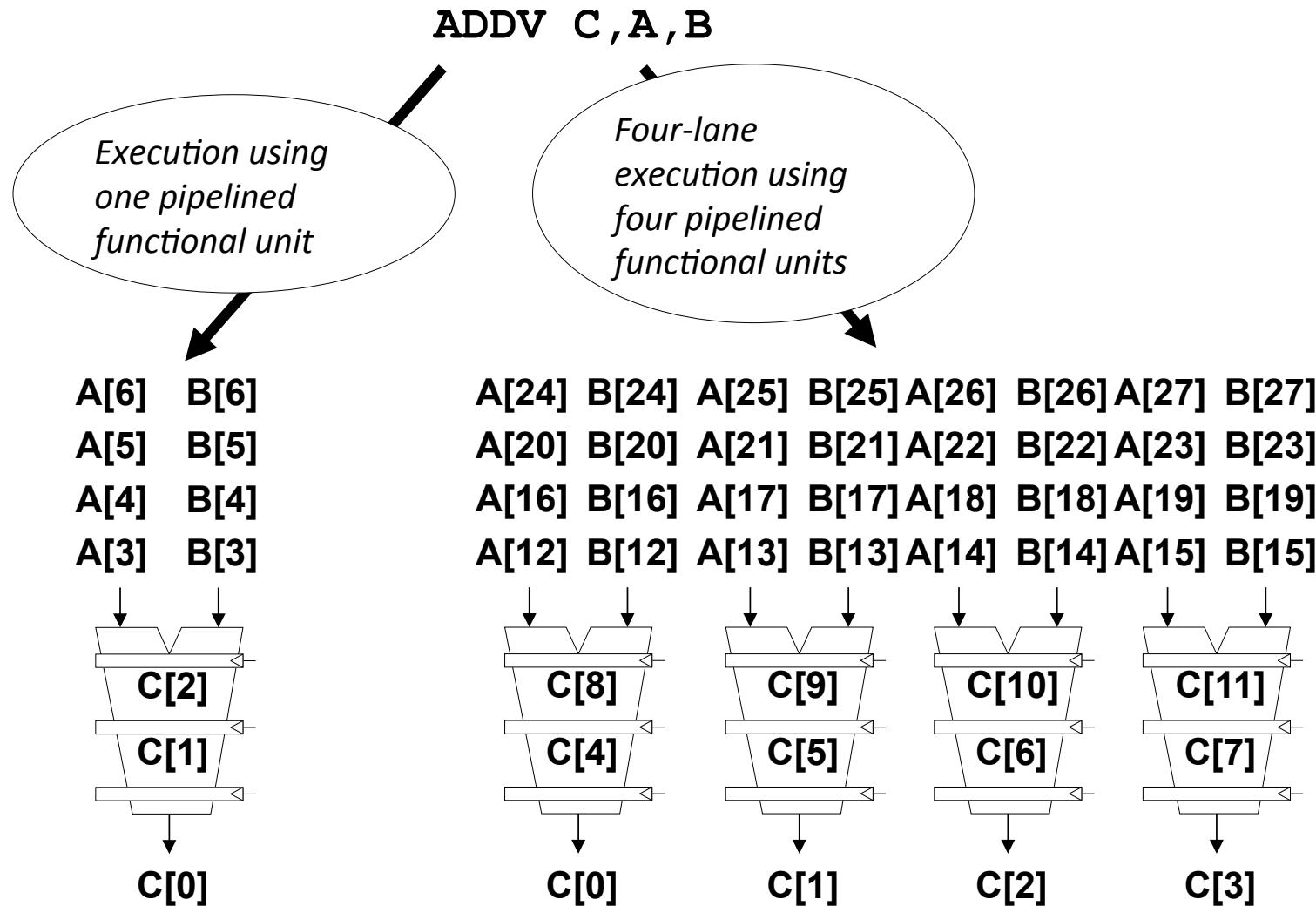
Vector elements do not have to occupy adjacent memory cells
Unlike with multi-media extensions (MMX, SSE and AVX)
Stride = amount by which elements are separated in memory

Results can be chained back to memory as they are produced

Some vector processors use registers to locate data items
vector registers need not be loaded from adjacent locations
These indexed loads are called a *gather* operation

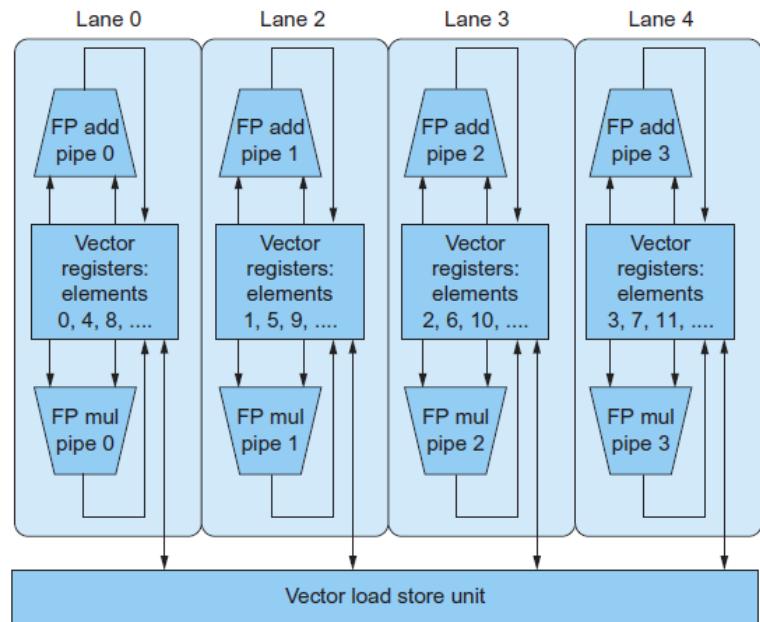
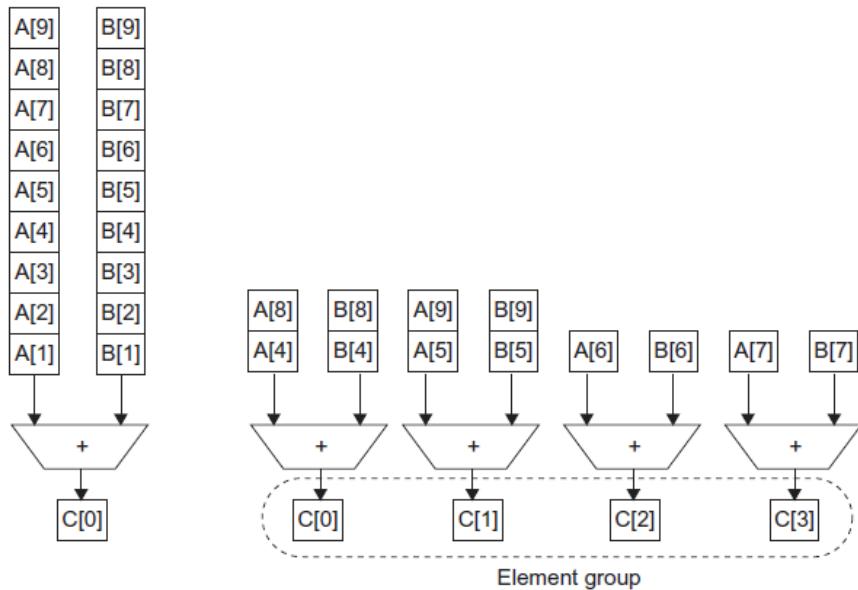
Indexing via registers can also be used to store vector results
Elements in vector registers are dispersed to non-adjacent locations
This is known as a *scatter* operation

Strip mining compensates for mismatch in vector register length
E.g. if vector is of length 60 and register is of length 32
Vector register is loaded with first 32 elements
And VLR (vector length register) is adjusted to load the next 28



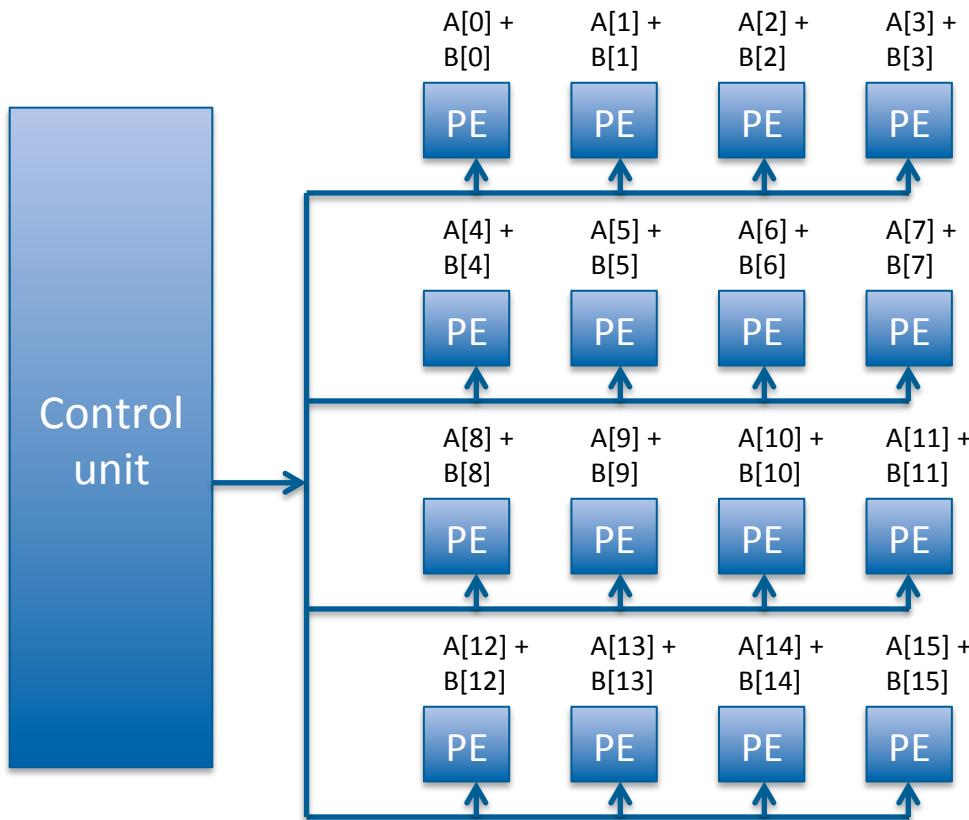
- Element n of vector register A is “hardwired” to element n of vector register B

- Allows for multiple hardware lanes
- No communication between lanes
- Little increase in control overhead
- No need to change machine code



Adding more lanes allows designers to tradeoff clock rate and energy without sacrificing performance!

- An alternative in an array processor (SIMD)
- Separate processing elements could add corresponding array elements



One control unit broadcasts the same command to multiple processing elements that operate in lock-step fashion.

Symmetric Multiprocessors are said to be tightly coupled

Also called shared memory multiprocessor

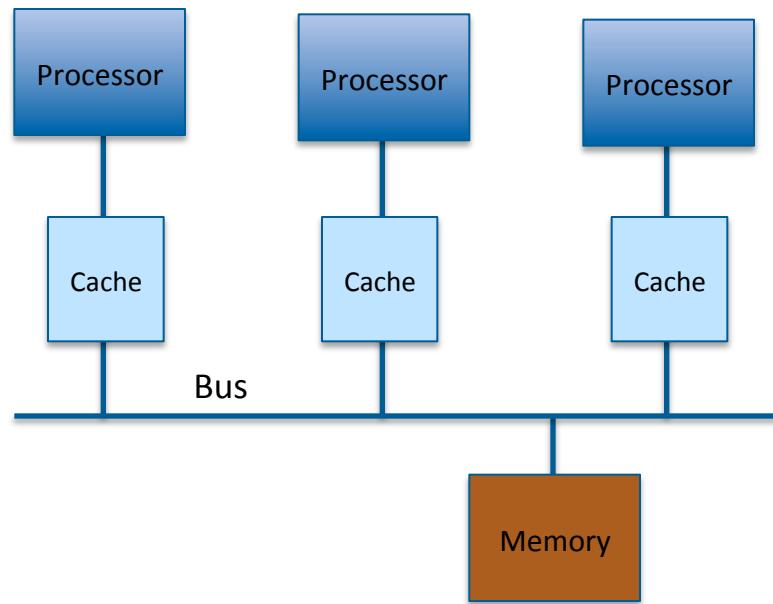
A type of MIMD system

An SMP architecture treats all processors equally

- *Symmetric* implies the processors are logically interchangeable
- The OS divides and distributes the work to processors

Programs can be written to run faster on SMPs

Programs optimized for SMP will suffer if run on uniprocessor



SMPs use a common shared bus

The bus may become a bottleneck

Processors must make good use of their internal caches

Cache misses cause stalls and contention for the bus

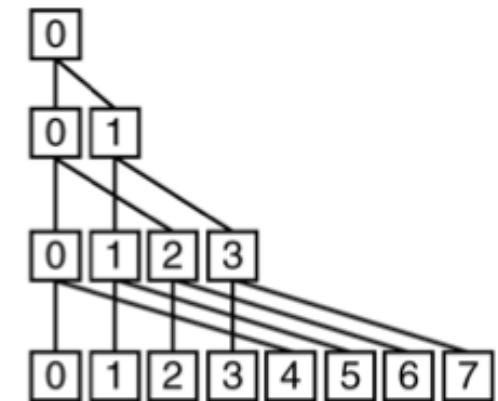
- To illustrate the idea, assume an 8-element array
- Assume there are 4 processors
- Each will add 2 elements

$$\text{Sum}[P_0] = A[0] + A[1]$$

$$\text{Sum}[P_1] = A[2] + A[3]$$

$$\text{Sum}[P_2] = A[4] + A[5]$$

$$\text{Sum}[P_3] = A[6] + A[7]$$



Then use half the processors (2)

$$\text{Sum}[P_0] = \text{Sum}[P_0] + \text{Sum}[P_2]$$

$$\text{Sum}[P_1] = \text{Sum}[P_1] + \text{Sum}[P_3]$$

Finally use one processor: $\text{Sum}[P_0] = \text{Sum}[P_0] + \text{Sum}[P_1]$

- Sum 100,000 numbers on 100 processor UMA
 - Each processor has ID: $0 \leq P_n \leq 99$
 - Partition 1000 numbers per processor
 - Initial summation on each processor

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

Sum[0] = A[0] + ... + A[999]

.

.

.

Sum[99] = A[99000] + ... + A[99999]

- Now need to add these 50 partial sums
 - Reduction: divide and conquer
 - Half the processors add pairs, then quarter, ...
 - 50, then 25, then 12, then 6, then 2, then 1
 - Using $\frac{1}{2}$ may yield an odd number
 - P0 takes care of the left over value
 - Need to synchronize between reduction steps

```
half = 100;  
do  
    synch();  
    if (half%2 != 0 && Pn == 0)  
        sum[0] = sum[0] + sum[half-1];  
        /* Conditional sum needed when half is odd;  
           Processor0 gets missing element */  
    half = half/2; /* dividing line on who sums */  
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];  
while (half > 1); // exit with final sum in sum[0]
```

synch() insures that all required partial sums have been produced
Private variables, such as *half* or a loop index, are local to each processor

- Clusters are loosely coupled
- Network of independent computers
 - Each has private memory and OS
 - Connected using I/O system
 - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
 - Web servers, databases, simulations, ...
- High availability, scalable, affordable

- SMPs
 - Processors are identical
 - Controlled by a single operating system
 - Cost to administer is about the same as a uniprocessor
 - Communicate via memory bus
 - More difficult to scale
 - Higher cost to purchase

- Clusters

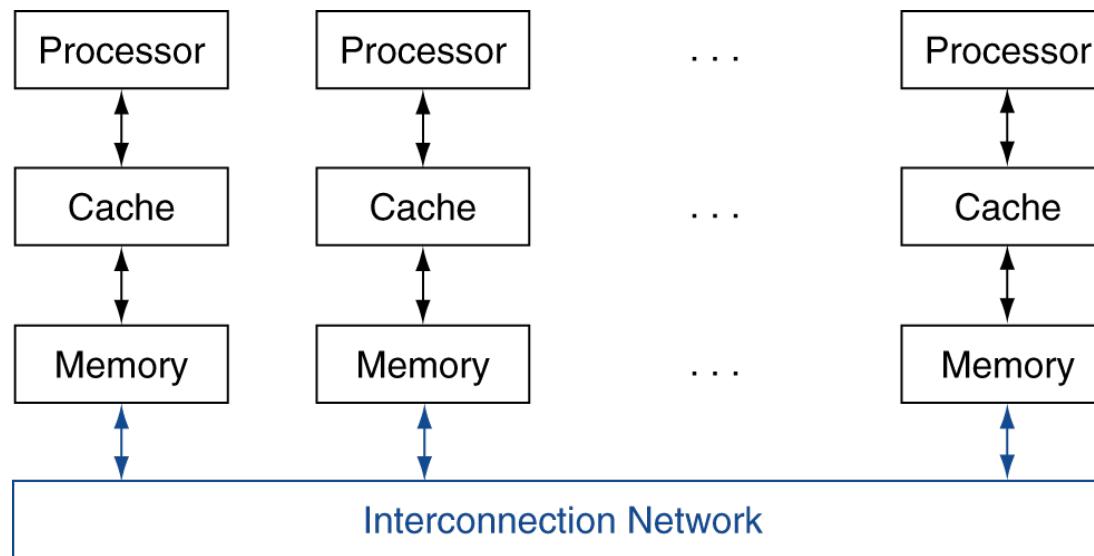
- Each node is a complete computer
- Nodes can employ different processors
- Cost to administer N-node system is about the same as for N separate computer systems
- Communicate via I/O bus at a slower rate
- Easier to scale
- Nodes may be low cost COTS (commodity off the shelf) computers

- Identical commodity-grade computers networked into a LAN
- Originally referred to a computer built in 1994 by Thomas Sterling and Donald Becker at NASA
- Normally use a unix-like operating system
- One server node and one or more client nodes connected via ethernet

Beowulf Cluster



- Each processor has private physical address space
- Hardware sends/receives messages between processors



- Sum 100,000 on 100 processors
- First distribute 1000 numbers to each
 - The do partial sums

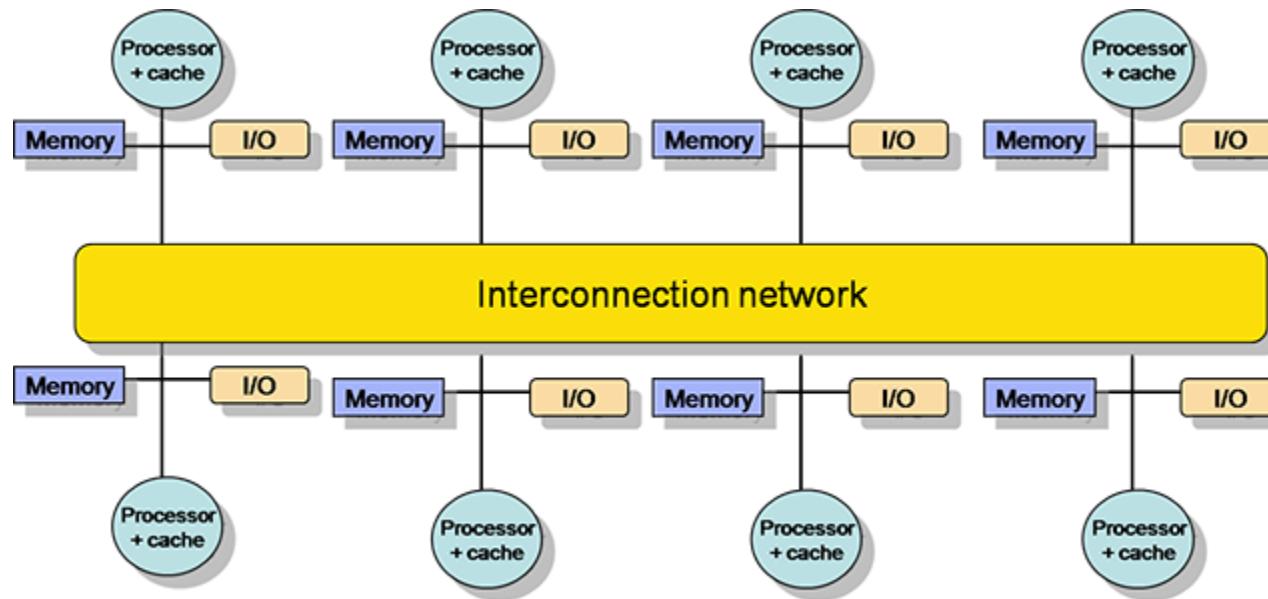
```
sum = 0;  
for (i = 0; i<1000; i = i + 1)  
    sum = sum + AN[i];
```

- Reduction
 - Half the processors send, other half receive and add
 - Then $\frac{1}{4}$ send, & $\frac{1}{4}$ receive and add, ...
 - Send/receive also provide synchronization
 - Assumes send/receive take similar time to addition

- Given send() and receive() operations

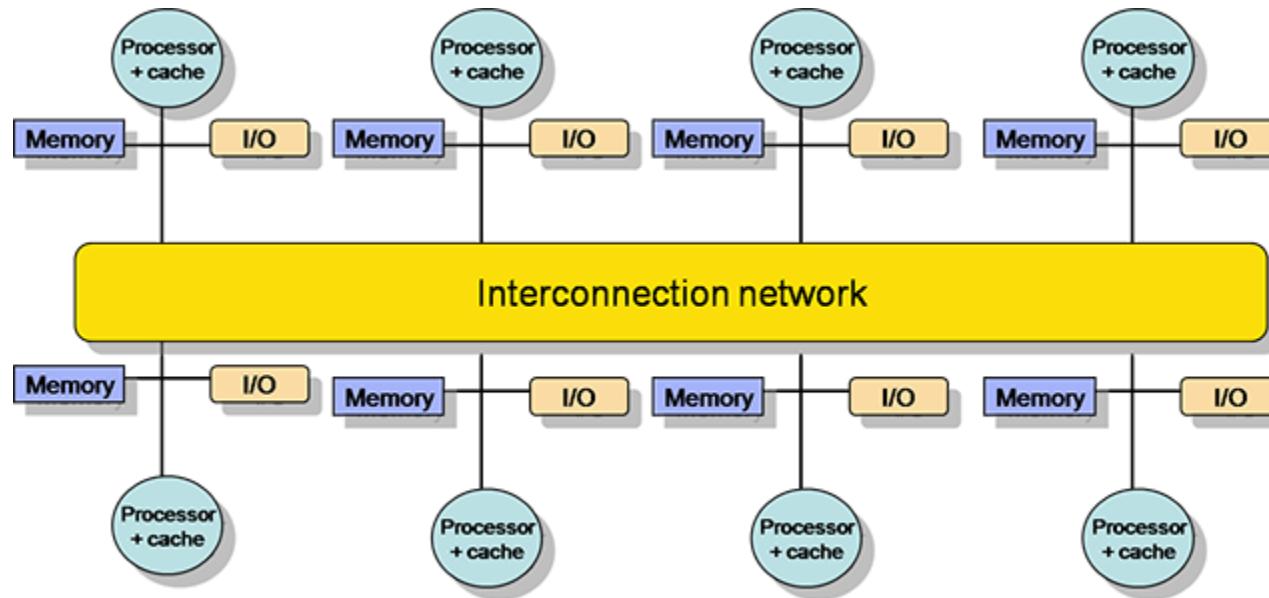
```
limit = 100; half = 100; /* 100 processors */
repeat
    half = (half+1)/2; /* send vs. receive
                           dividing line */
    if (Pn >= half && Pn < limit)
        send(Pn - half, sum);
    if (Pn < (limit/2))
        sum = sum + receive();
    limit = half; /* upper limit of senders */
until (half == 1); /* exit with final sum */
```

NORMA (no remote memory access) systems



Each processor has a separate private memory & address space

Processors only have direct access to their local memory

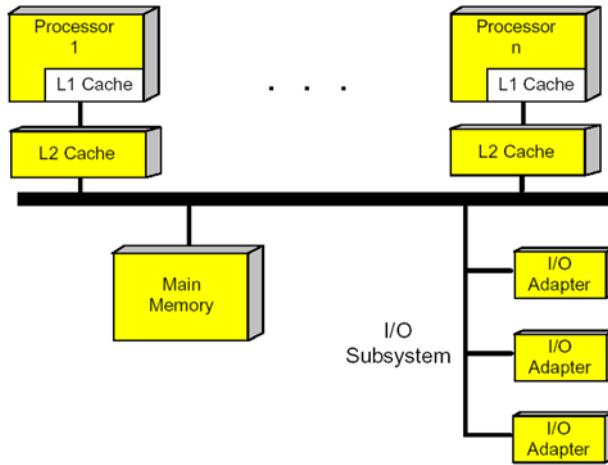


Message exchange is used to obtain data from remote modules
Clusters fit this loosely coupled model

These are easier to scale than the SMP tightly coupled counterpart

Using a single global memory unit

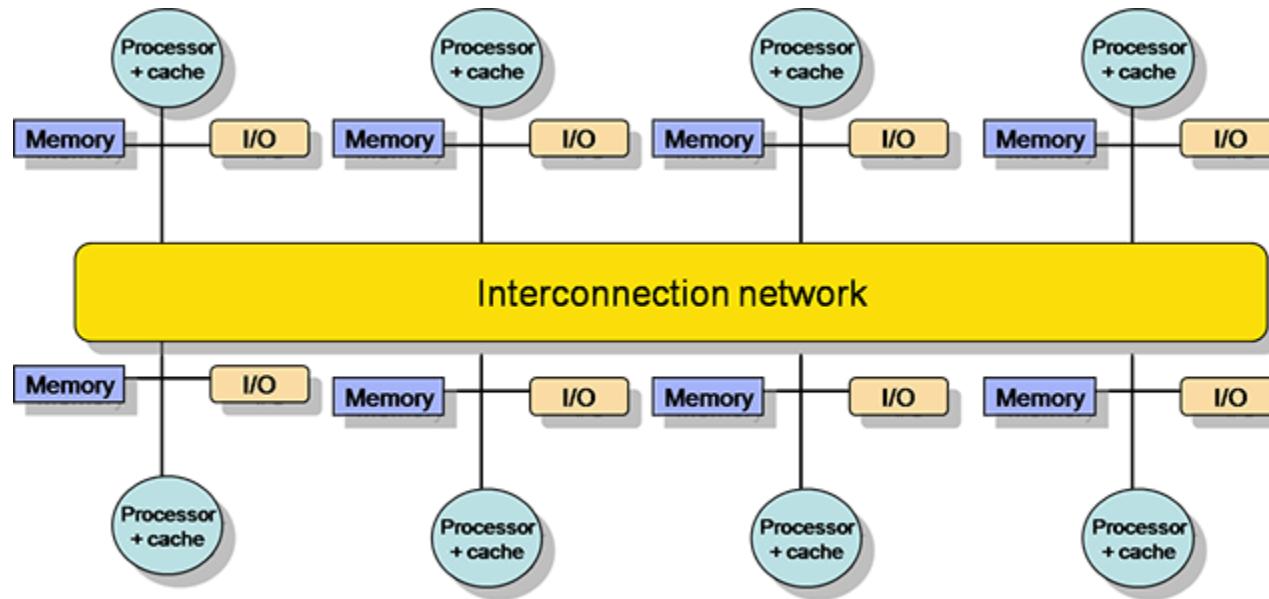
Uniform memory access (UMA) for all processors



Only one processor at a time can access memory
Competition for the CPU-to-memory bus will be high
Cache can be used to reduce the conflicts
This would be too limiting

Distributed memory modules work better

Non-uniform memory access (NUMA) system



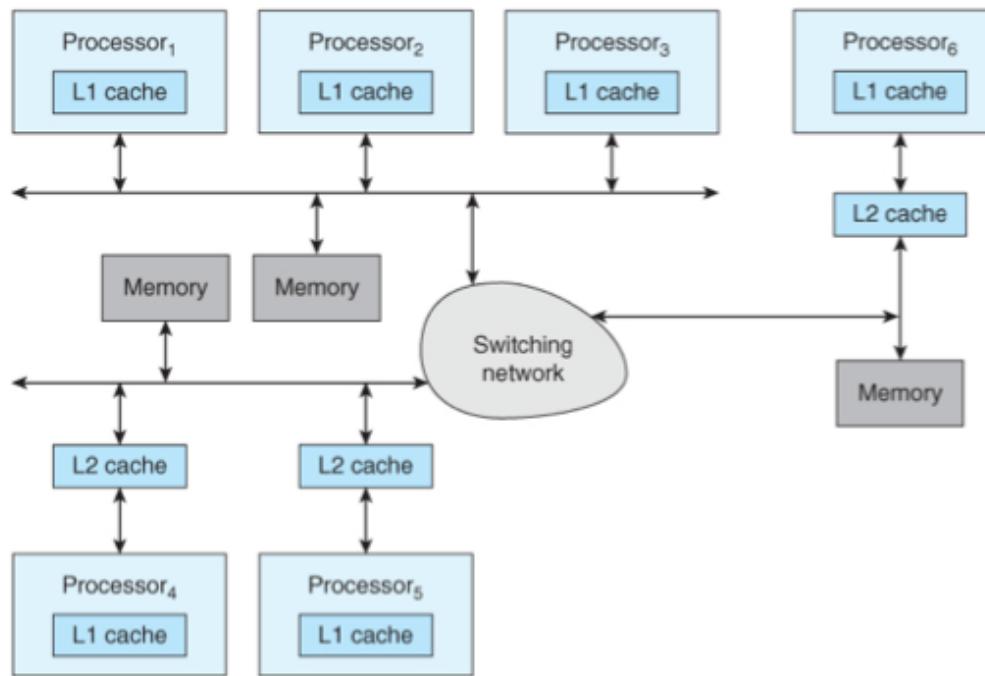
Local module access is faster than to a remote module

CPU addresses map to a particular memory module

Single shared address space spans all modules

Each processor has a cache system

Another design for a NUMA system is shown below:



Memory is accessed less often when cache is used
But shared data must be kept consistent (i.e., coherent)

Correct program order must be preserved

Each processor must read the most recent version of data

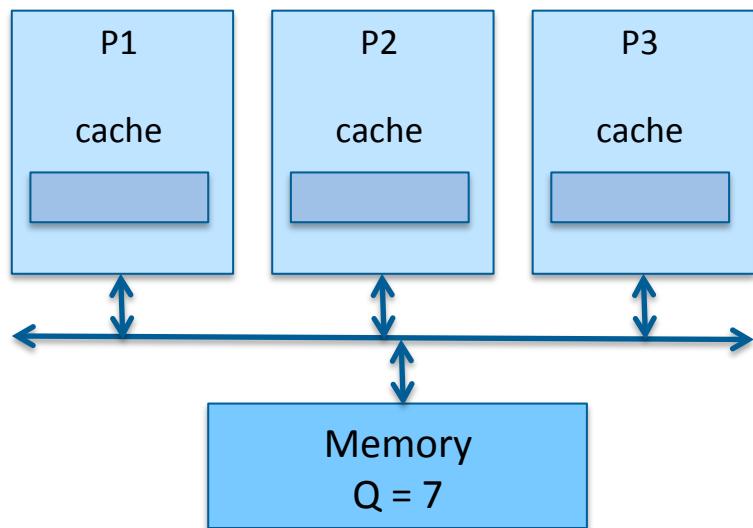
All writes must be visible to all processors

If P1 writes x and then P2 reads x, P2 must obtain value written by P1

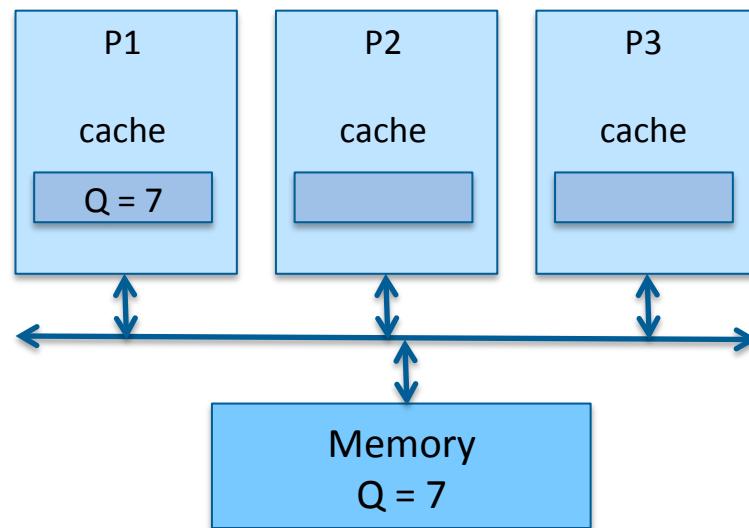
Order must be preserved

Example: P1 sets x to 1, P2 reads x and if $x==1$, sets it to 2

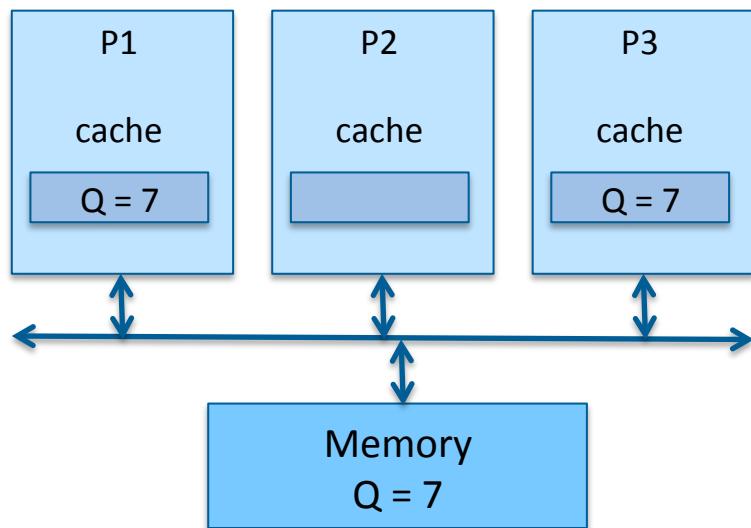
If P3 then reads x, it should see 2 and not the old value 1



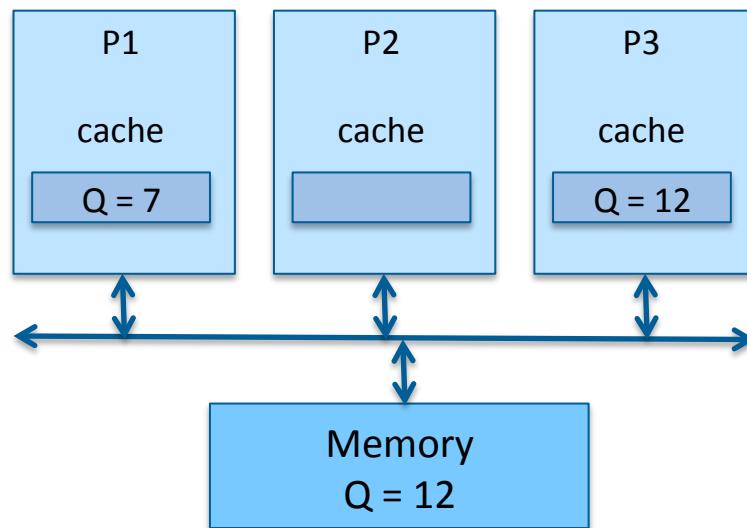
a) Initially $Q = 7$ in memory



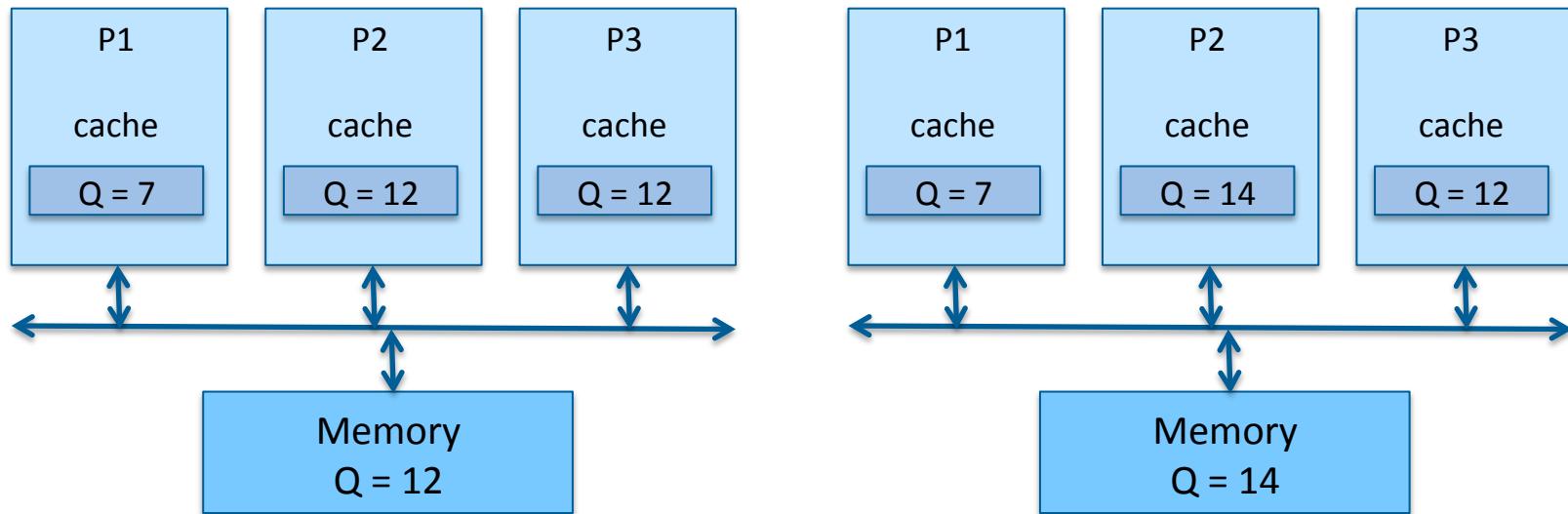
b) P1 reads Q and caches it locally



c) P3 reads Q from memory and caches it.



d) P3 writes $Q=12$ to cache and to memory



e) P2 reads Q from memory and caches it

f) P2 computes $Q + 2$, caches it locally, and writes it back to memory

There are now three different cached values of Q
This happened because cache coherency was not maintained.

MESI is a protocol used by Intel and others

Each cache line can be in one of 4 states:

M – modified (in one cache & was altered since read from memory)

E – exclusive (in one cache & matches what's in memory)

S – shared (in at least 2 caches and matches what's in memory)

I – invalid (never filled or is outdated)

Lines change state based on accesses that are made

From local processor or from other processors over the bus

Cache controller listens (snoops) on the bus for accesses

A snoopy protocol

Writes are broadcast on the bus

All caches observe the write

Writes may invalidate copies of lines in other caches

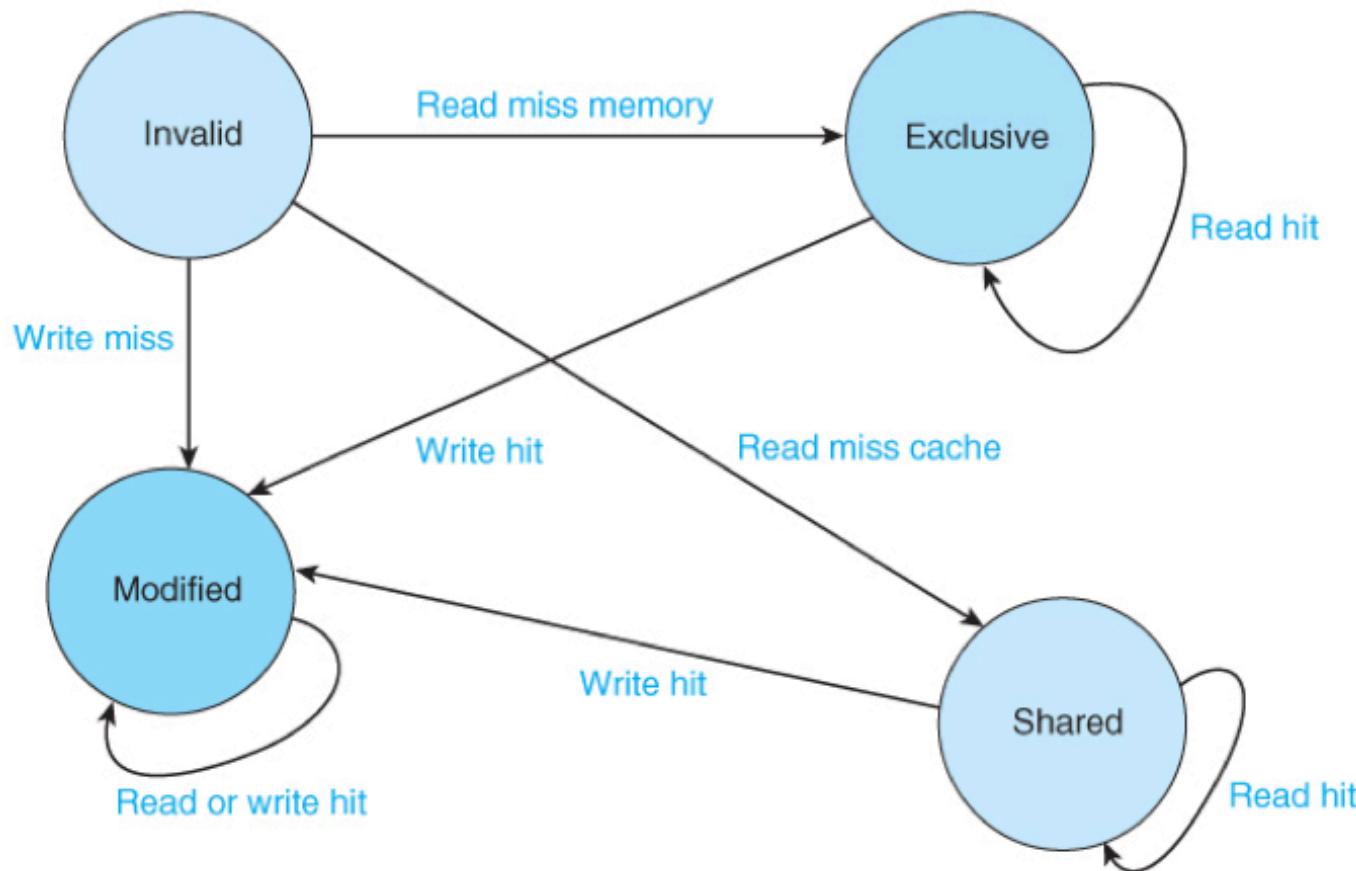
Does not scale beyond about 128 processors

The bus traffic due to snooping becomes too high

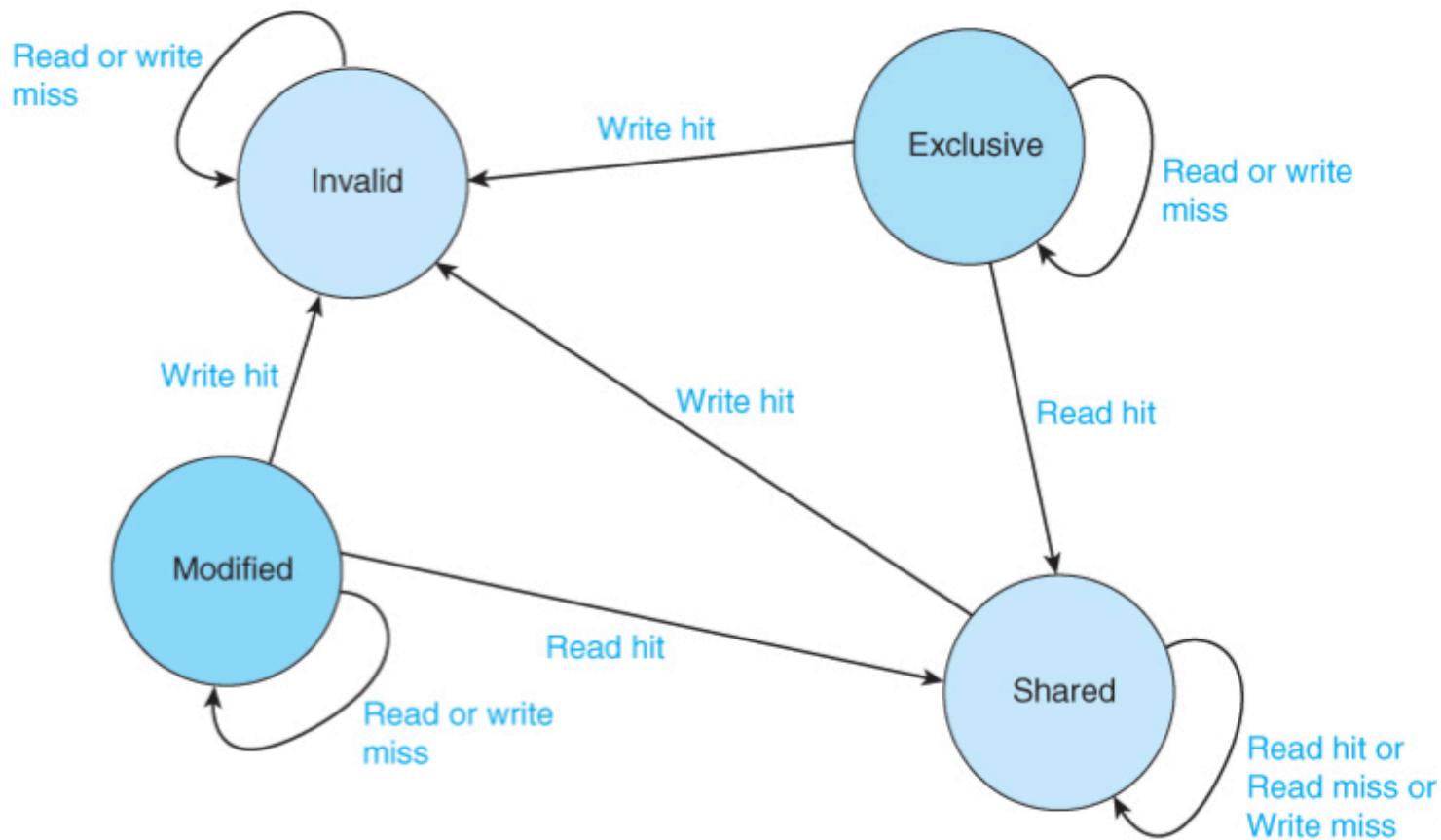
Exclusive and modified states reduces traffic

Writes to modified or exclusive lines need not be broadcast

MESI state diagram as seen from the local CPU bus



MESI state diagram as seen from the system bus



Bus Transaction	Action by local cache
Read hit	$E \rightarrow S, M \rightarrow S$ or no change if S
Read miss	no change
Write hit	$E \rightarrow I, S \rightarrow I, M \rightarrow I$
Write miss	no change

Snoopy caches work well when connected to a single bus

Large shared-memory multiprocessors use interconnects

These are networks such as rings or meshes

Broadcasting cache operations to all processors would be inefficient

Cache directories can be used as an alternative to snooping

Each memory module would have a directory

This approach scales better than snooping on a shared bus

Directories identify which nodes contain a copy of a block

The state of each cache line containing a copy is recorded

Accesses to a module are intercepted by the directory

The directory determines the action to take:

Reads are forwarded to the cache containing the copy

Writes are sent just to the nodes whose copies are affected

Misses cause the memory module to be accessed

Superscalar and VLIW systems are based on ILP
Instruction level parallelism
The processor fetches and executes bundles of instructions
Hardware executes as many instructions as possible in parallel
Compilers combine instructions into long words on VLIW systems

Thread level parallelism overlaps streams of instructions
Processes or tasks are split into separate threads
When one thread stalls, control is switch to another
This hides latencies due to cache misses or I/O
All threads appear to be running at the same time

Tasks & threads differ in scale or granularity

Tasks involve longer streams of instructions than do threads

- Multitasking involves context switches and more overhead

- Must complete pending instructions, save registers & flush cache

- Triggered, for example, by a page fault

Thread switching is more efficient

Hardware has multiple register sets

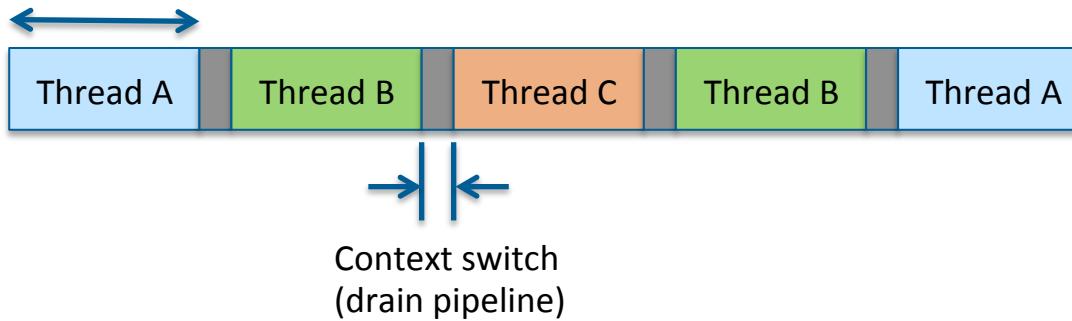
- no saving or restoring is needed when switching

Multithreading can be coarse-grained or fine-grained

Coarse-grained

Switch occurs after a group of instructions are executed

Pipeline is drained each time a switch takes place



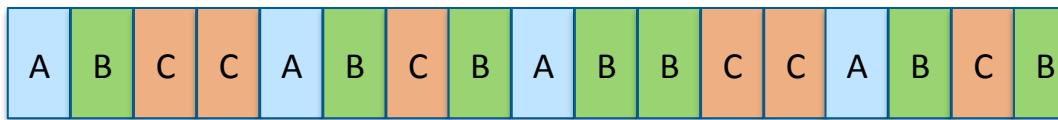
Expiration of a time slice or having to wait for I/O to complete trigger the switching

Fine-grained

Switch occurs at the end of each cycle

Each thread has its own set of registers

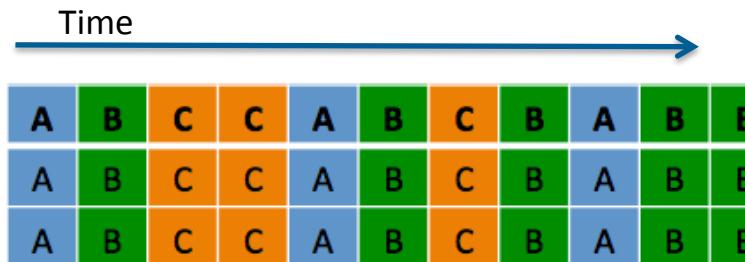
No need to drain the pipeline



- Instructions from multiple threads are interleaved within the pipeline
- Having 2 or more processors allows several threads to execute in parallel
- To be efficient, there must be enough threads to keep the processors busy

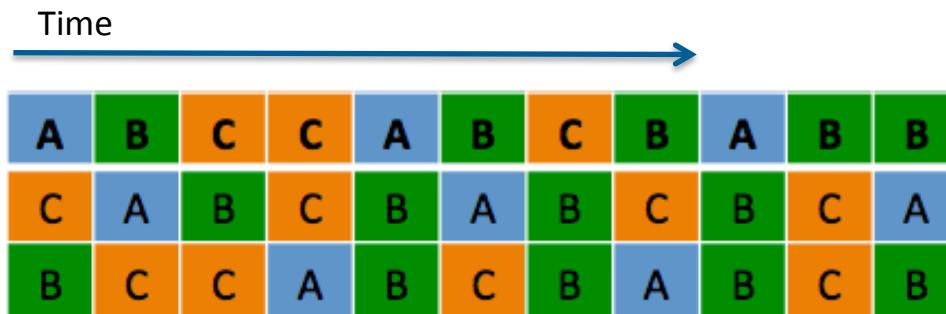


Multiprocessors may also be superscalar



(each column corresponds to a clock cycle)

Fine-grained multithreading on a processor with three execute units which are available to only one thread at a time



Fine-grained simultaneous multithreading (SMT)
Execute units are available to all threads



Some execute units may not be used in some cycles

Data hazards and stalls may prevent the use of some units

Lack of proper instruction type may also cause idle units

Cycle	1	2	3	4	5	6	7	8	9	10	11
	A	B	C	C	A		C		A	B	B
	C		B		B	A	B		B		A
	B	C	C		B	C	B		B	C	B

(empty boxes represent idle units)

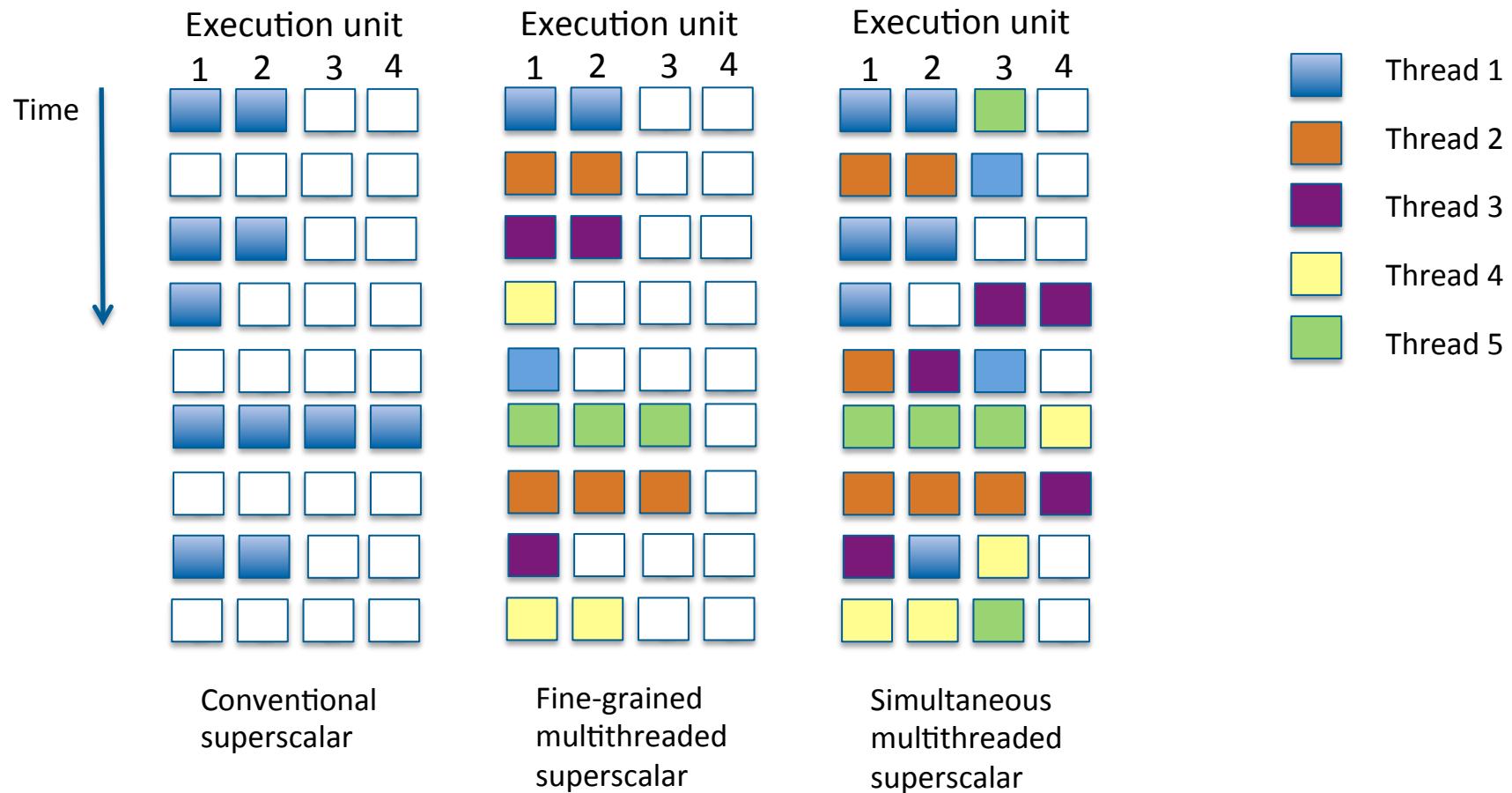
For example, with 2 integer units and 1 floating point unit, a bubble occurs if there is no floating point instruction available (cycle 2)

The need to stall may idle all 3 units (cycle 8)

SMT requires that each thread have its own register set and PC

Each thread must have its own resources

Instructions are tagged with the thread number or thread ID



Hyperthreading (HT) is another name for SMT

Intel first used HT in its Xeon processor

HT is also used in the Pentium 4 & Core i7

HT provides two logical processors for each physical processor

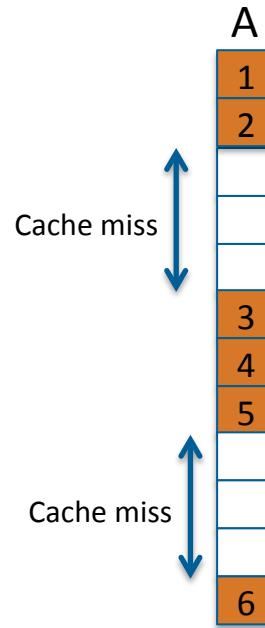
Increased Xeon's performance by 30%

The cost in increased chip area was about 5%

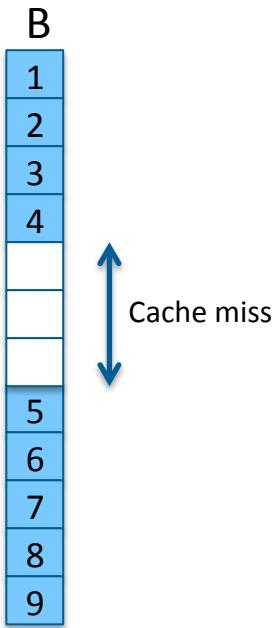
Requires reproducing registers:

Architectural, machine state and control registers

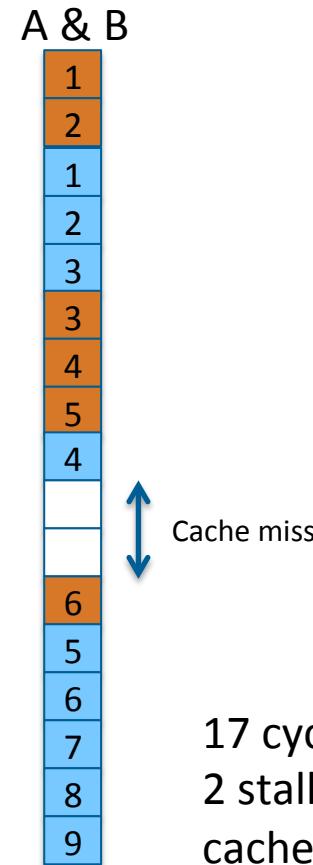
All other resources are shared by logical processors including:
caches, execute units, branch predictors, control logic & buses



12 cycles total
6 stall cycles due
to cache misses
Efficiency = 6/12



12 cycles total
3 stall cycles due
to cache misses
Efficiency = 9/12



17 cycles total
2 stall cycles due to
cache misses
Efficiency = 15/17
Latency for B increases
Now 17 rather than 12

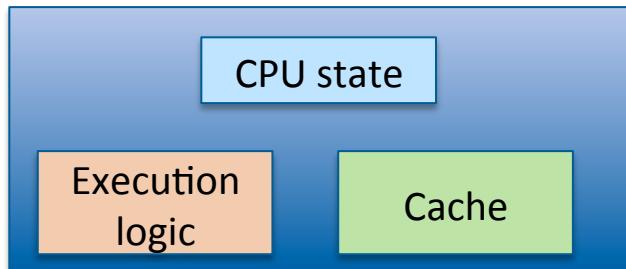
- Performance can be improved in different ways
 - Higher clock rates
 - Power consumption varies as the square of clock rate
 - Superscalar operation with multiple execute units
 - Requires more complex control units
 - To schedule instructions
 - To avoid dependencies
 - Uses many more transistors
 - Consumes more power and dissipates more heat
 - The degree of improvement reaches a limit

- Multiple less sophisticated processors can be used
 - Provide similar performance at a lower cost
 - Allows slower clock rates
 - Consumes less power
 - Generates less heat
- Coprocessors provide a form of multiprocessing
 - Asymmetric in nature
 - MIPS examples:
 - CP1 for floating point
 - CP0 for managing exceptions
- Vector & array coprocessors may be used

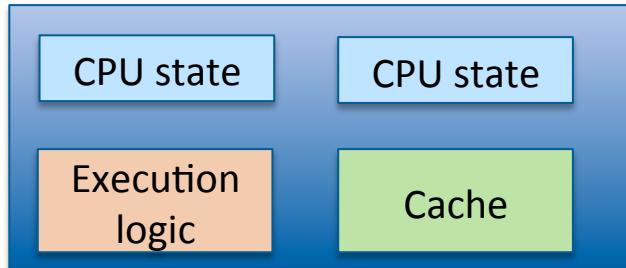
- Multi-processing Systems
 - Contain more than one processor
 - The processors are identical in an SMP (symmetric multi-processor)
 - Also known as homogeneous
- Each processor plugs into a different socket
 - These are board level multiprocessors
 - A bus connects the processors
 - Provides additional performance on a single computer system
 - To benefit, software must take advantage of the additional processors
 - Inclusion of more processors in an SMP is a choice

- “Core” logic is replicated on the same chip
 - Multiple cores are presented to the OS
 - Cores share data through on-chip logic or shared caches
 - Homogenous systems use identical cores
 - Heterogeneous systems mix cores having different:
 - ISA
 - Chip area
 - Performance
 - Power dissipation
 - These are chip level multiprocessors
 - Each core has a separate set of registers
 - The number of cores is predetermined
 - Issues relating to shared memory must be handled

- Multi-core concepts & designs developed over time

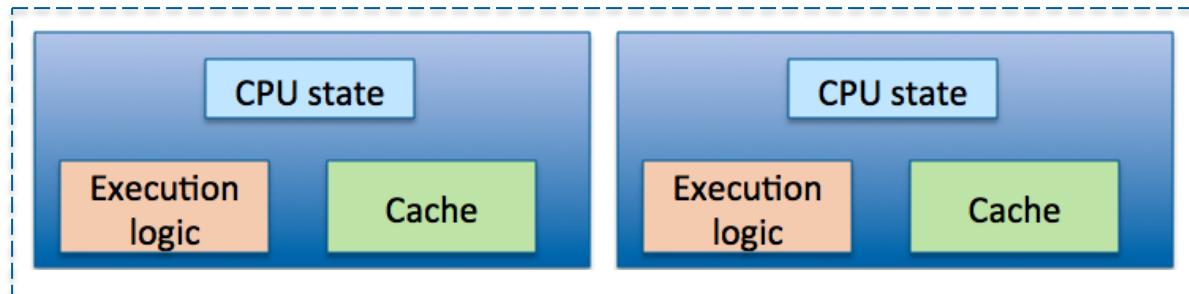


Basic single-core processor



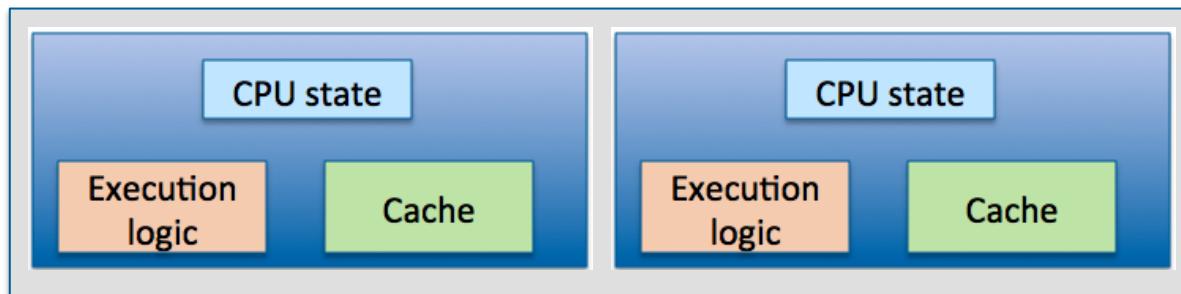
Single-core processor with hyperthreading

Hyperthreading also known as simultaneous multithreading uses additional hardware registers to allow one physical processor to act as two virtual processors



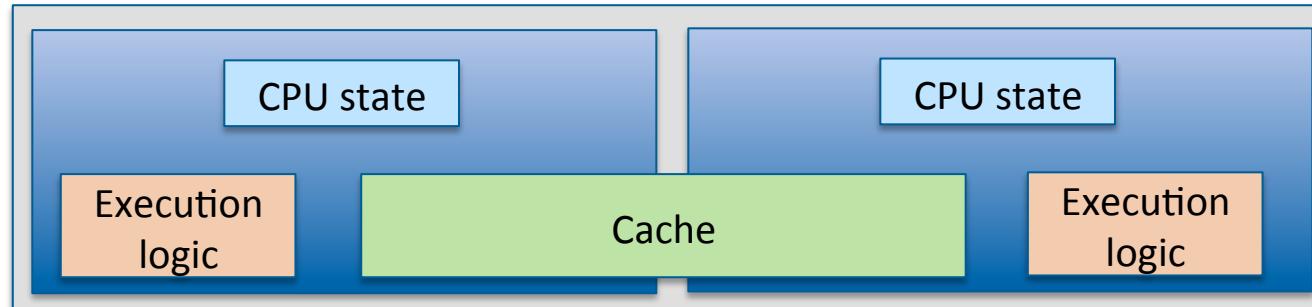
Two or more
separate processors

The processors share memory and the same environment (motherboard)
The dashed lines mean they are in the same system but not on the same chip

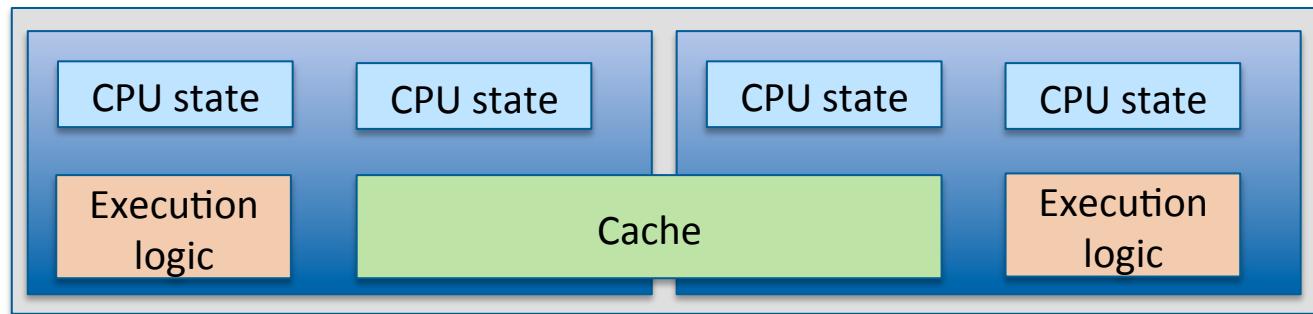


Multiple cores

Multi-core processor with two or more cores housed in the same package.



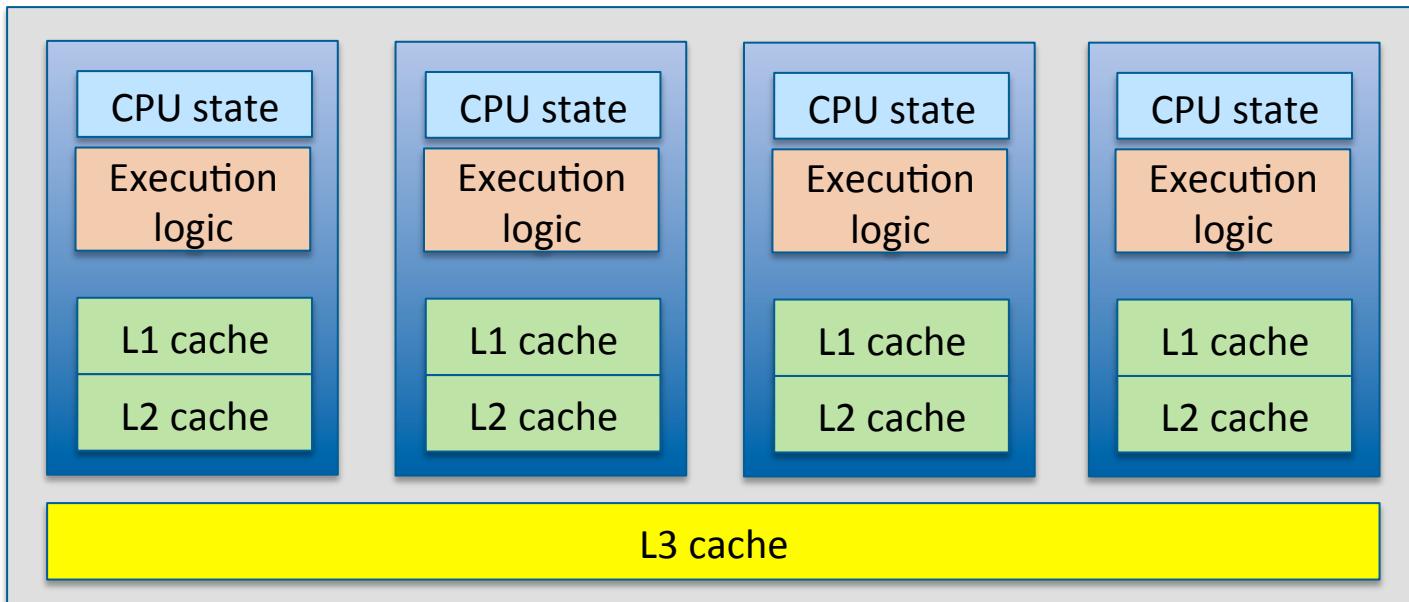
Cores share a common cache



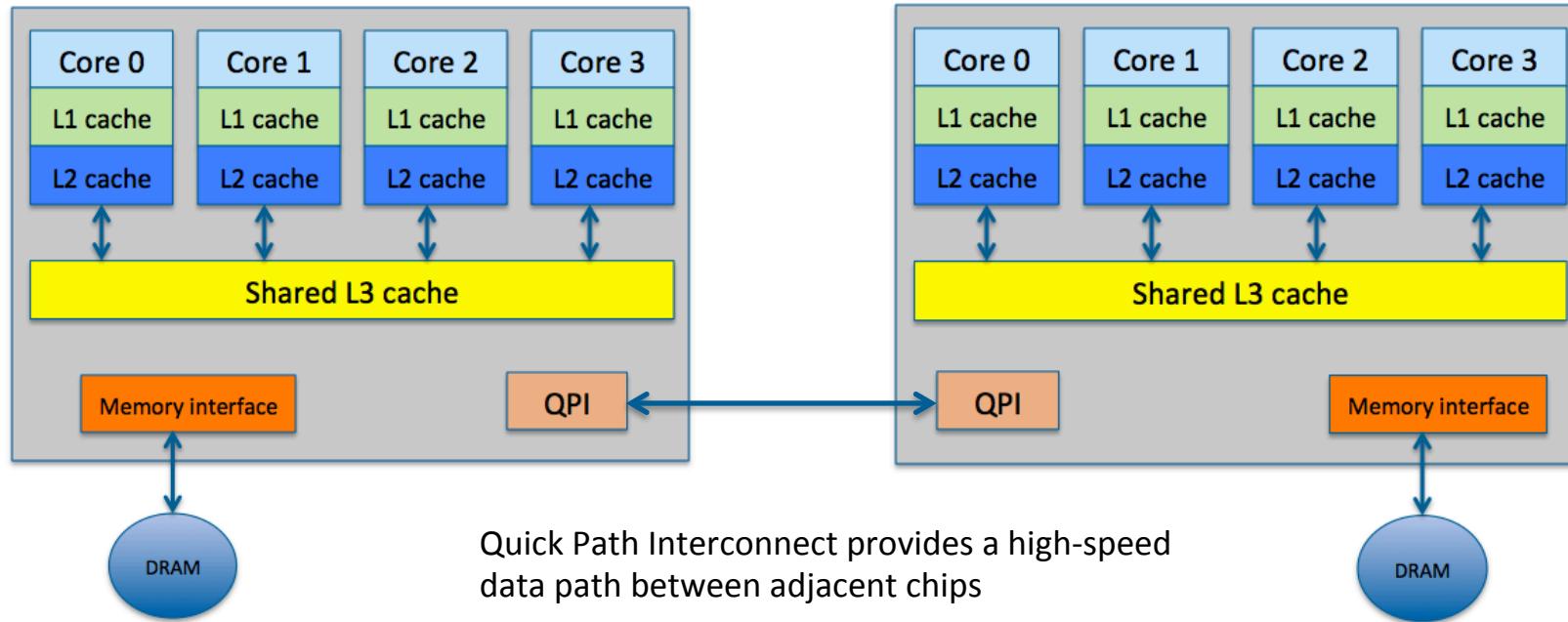
Dual-core processor with hyperthreaded cores

Sharing the cache increases the degree of coupling between the processor cores

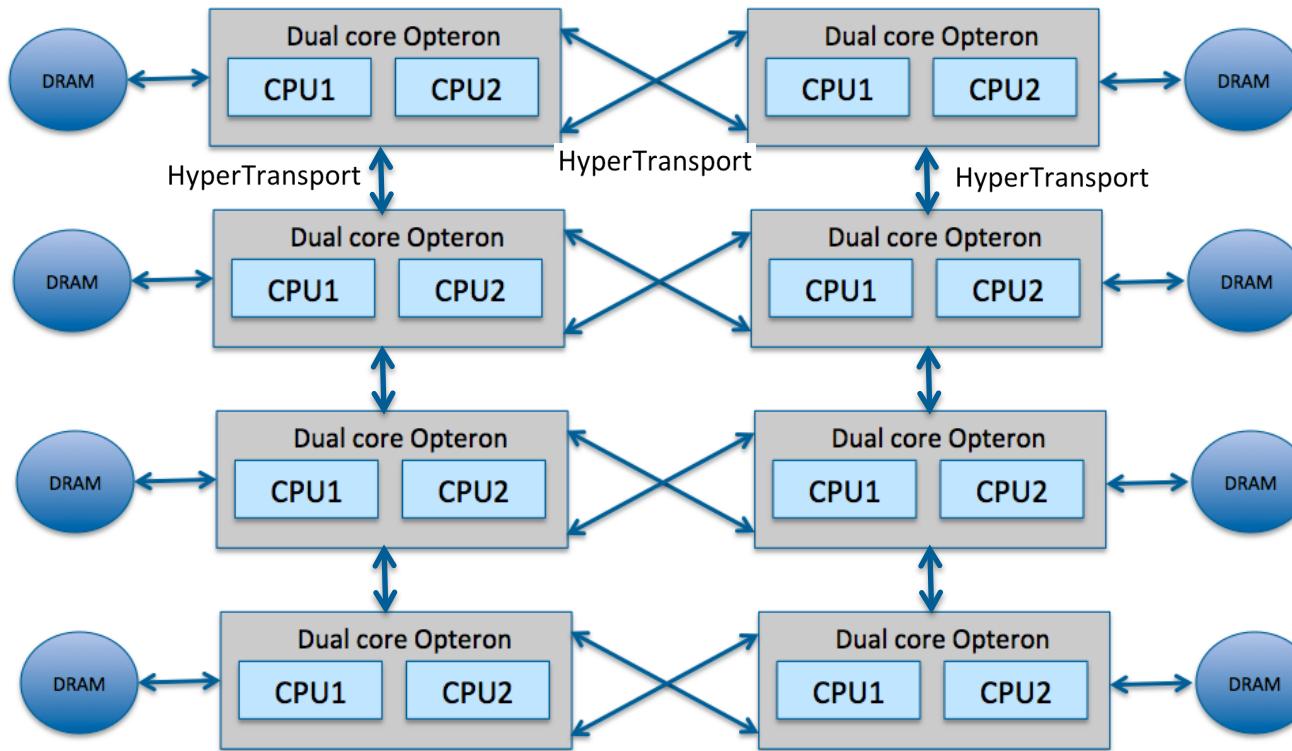
Quad-core system



Each processor has a private L1 and L2 cache
All processors share a much larger L3 cache

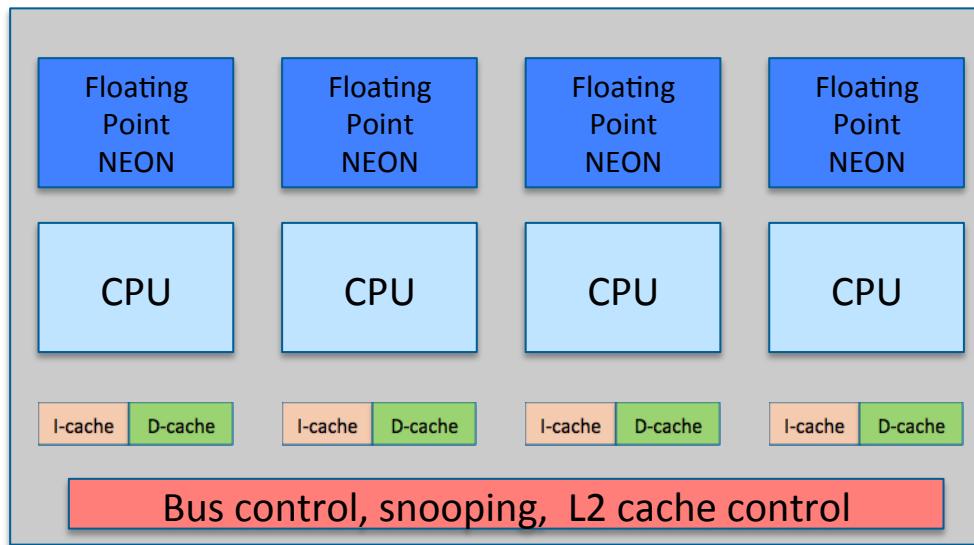


- Introduced in 2008 with the Core i7
 - The QPI interface transfers data on both the rising and falling edge of the clock (at 3.2 GHz) for a bandwidth of 25.6GB/s.

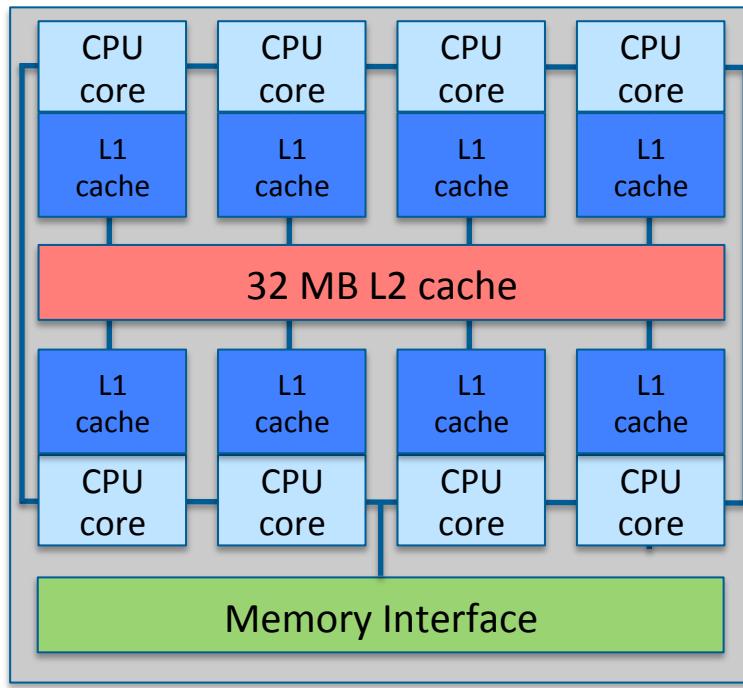


Has 8 dual-core processors linked via HyperTransport

- A NUMA system with a high speed cross-linked network



A general-purpose processor intended for use in mobile devices
Available with 1, 2 , 3, or 4 superscalar cores
Can directly execute Java bytecode
No L2 cache, but includes on chip bus snooping control



Eight cores, can be turned off individually to save energy

Cores contain 2 integer ALUs & 2 floating point units

Also includes a decimal floating-point (DFP) unit for financial applications

Hardware DFP is 100 to 1000 times faster than software decimal arithmetic

- Advanced RISC Machines (ARM)
 - Employs RISC-style architecture with some CISC-style features
 - 32-bit registers and addresses
 - Byte-addressable memory with 8-bit bytes, 16-bit halfwords and 32-bit words.
 - Enforces memory alignment for words and half words.
 - Supports both big-endian and little-endian memory storage order

■ RISC features

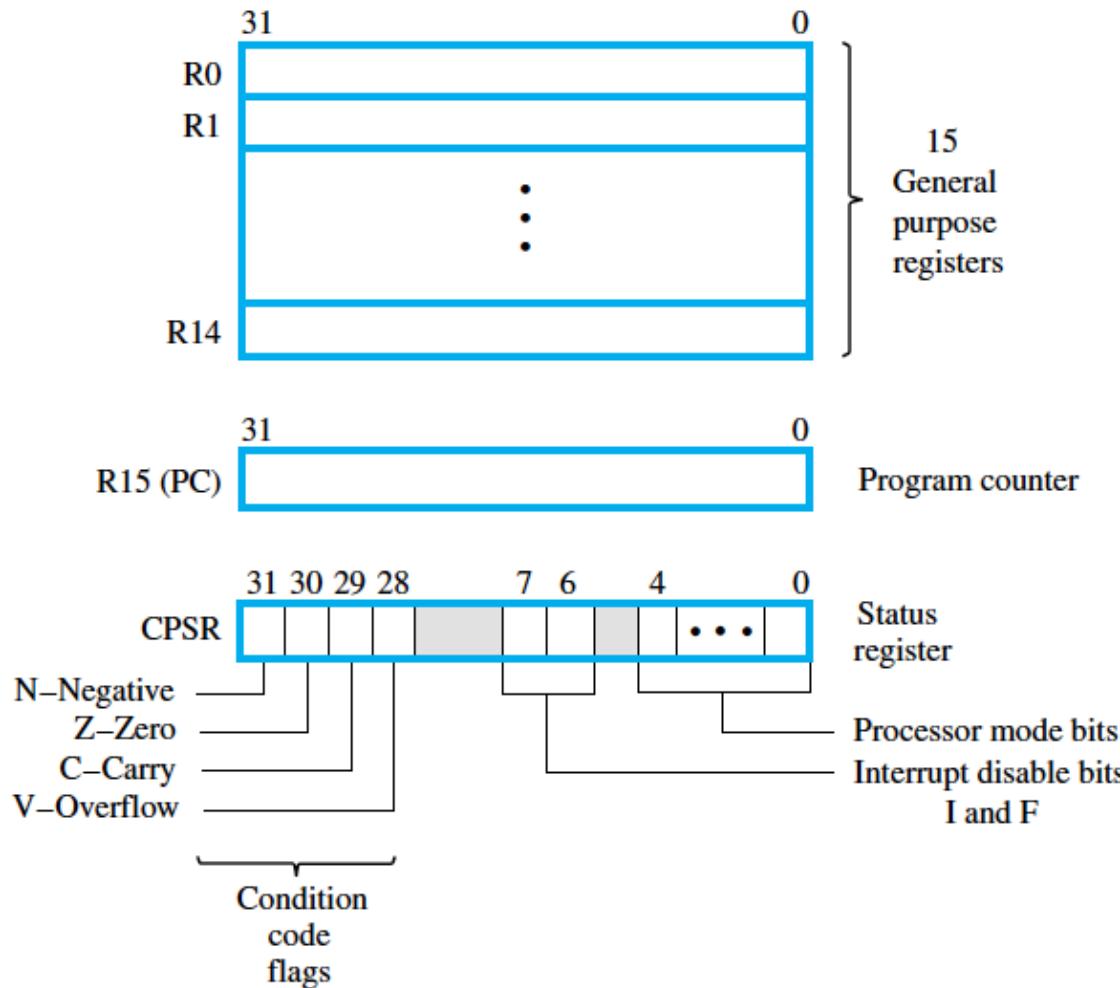
- Fixed length 32-bit instructions
- Load/store architecture
- All arithmetic and logic instructions use only register operands

■ CISC features

- Supports autoincrement, autodecrement & PC-relative addressing modes
- Condition codes (N,Z,V,C) are used for branching & conditional execution
- Multiple registers can be loaded from or stored into a block of consecutive memory words

- Unusual features
 - All instructions are conditionally executed
 - conditions specified by 4-bit field within each instruction
 - Can be set to always execute
 - Permits shorter routines than RISC machines using many branch instructions
 - No divide or explicit shift instructions
 - Division must be done in software
 - Immediate or register operands can be shifted by a prescribed amount before being used

- 16 registers (R0 – R15)



- R15 used as the program counter (PC)
- R14 is the subroutine linkage register
- R13 is the stack pointer
- “Banked registers” facilitate context switches
 - Additional copies of R0 – R14
 - Reduces the need to save and restore registers.

■ ARM addressing modes

- Indexed addressing mode
- Register mode
- Immediate mode
- Indirect mode
- Absolute mode
- Auto-increment & auto-decrement
- PC relative mode

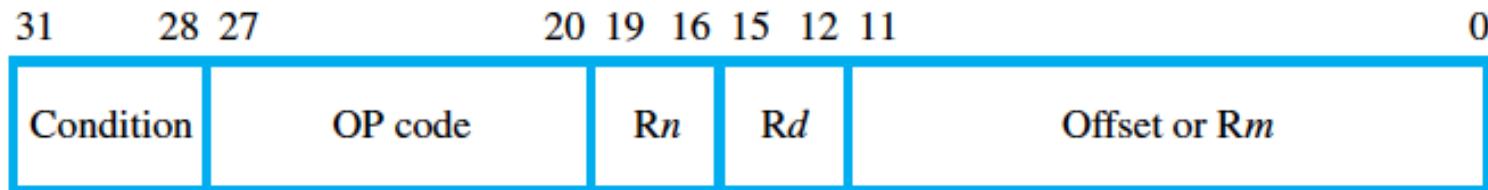
■ Other ARM features

- Employs memory mapped I/O
 - Direct programmed controlled
 - Interrupt driven
- Supports coprocessors
 - Floating point
 - Signal and video coprocessors
 - Exchange data between CPU and coprocessor registers
 - Memory transfers to/from coprocessor registers

- Supports THUMB ISA
 - Used in low-cost & low-power embedded systems
 - Mobile devices such as phones
 - 16-bit machine instructions (reduced code space)
 - Fetched and expanded to 32 bits at run time
 - T bit within CPSR indicates when in Thumb mode
 - Programs can contain a mix of Thumb and standard routines.
 - Many Thumb instructions use a two-operand format
 - Conditional execution applies to all standard instructions but only to branches in Thumb set

■ Basic Indexed Mode

- Pre-indexed – effective address (EA) = contents of base register plus a signed offset
- Load (LDR) & store (STR) instructions will be used to illustrate addressing modes



Format for Load and Store instructions.

High 4 bits in all instructions specify a condition that determines whether the instruction is executed.

■ Pre-indexed mode

- A bit within the opcode indicates whether the offset is in the low 12 bits or whether the offset is in a register indicated by the low 4 bits of the instruction.
- Offset is treated as an unsigned value (a bit within the opcode indicates sign for offset).
- Examples:
 - LDR Rd,[Rn, #offset] ; $Rd \leftarrow [[Rn] + \text{offset}]$
 - LDR Rd,[Rn, Rm] ; $Rd \leftarrow [[Rn]+[Rm]]$
 - LDR Rd,[Rn] ; $Rd \leftarrow [[Rn] + 0]$

■ Relative Addressing Mode

- The PC is used as the base register.
- Programmer uses label and assembler determines offset
- Offset = operand address - PC+8
- Examples:
 - LDR R1,ITEM ; loads contents of memory location ITEM into R1.

- Pre-indexed with writeback
 - Computed EA overwrites Rn
- Post-indexed
 - $EA = [Rn]$
 - Rn is then overwritten with $EA + \text{offset}$
- Offset in register may be scaled by power of 2
 - shifted right or left a specified amount (0 – 31)
 - direction & shift amount are encoded in Rm field

■ Example:

- LDR R0,[R1, -R2,LSL #4]!
 - $R0 \leftarrow [R1] - 16 * [R2]$
 - R1 is overwritten by EA (! specifies writeback)

ARM indexed addressing modes.

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[Rn, #offset]	$EA = [Rn] + \text{offset}$
Pre-indexed with writeback	[Rn, #offset]!	$EA = [Rn] + \text{offset};$ $Rn \leftarrow [Rn] + \text{offset}$
Post-indexed	[Rn], #offset	$EA = [Rn];$ $Rn \leftarrow [Rn] + \text{offset}$
With offset magnitude in Rm:		
Pre-indexed	[Rn, ± Rm, shift]	$EA = [Rn] \pm [Rm] \text{ shifted}$
Pre-indexed with writeback	[Rn, ± Rm, shift]!	$EA = [Rn] \pm [Rm] \text{ shifted};$ $Rn \leftarrow [Rn] \pm [Rm] \text{ shifted}$
Post-indexed	[Rn], ± Rm, shift	$EA = [Rn];$ $Rn \leftarrow [Rn] \pm [Rm] \text{ shifted}$
Relative (Pre-indexed with immediate offset)	Location	$EA = \text{Location}$ $= [PC] + \text{offset}$
EA = effective address offset = a signed number contained in the instruction shift = direction #integer where direction is LSL for left shift or LSR for right shift; and integer is a 5-bit unsigned number specifying the shift amount ±Rm = the offset magnitude in register Rm can be added to or subtracted from the contents of base register Rn		

■ Register mode

- Used for arithmetic & logic instructions
 - 2 source registers and a result register

■ Absolute mode

- If base register contains 0, 12-bit offset = absolute address

■ Immediate mode

- Provided via pseudo-instructions
- Format: LDR Rd,=value
- Equal sign indicates immediate value
 - Examples:
 - LDR R2,=127 ; replaced by MOV R2,#127
 - LDR R2,=&ABCD3456 ;replaced by LDR R2,MEMLOC
 - where MEMLOC contains hex ABCD3456
 - “&” prefix denotes a hex value

■ Load 32-bit addresses

- Examples:
 - ADR Rd,LOCATION ; loads 32-bit address into Rd
 - Assembler computes offset from current PC value
 - If LOCATION is in forward direction
 - ADD Rd,R15,#offset ; is substituted
 - If LOCATION is in backward direction
 - SUB Rd,R15,#offset ; is substituted
 - In either case, offset is an unsigned 8-bit number
 - Rotating the 8-bit number can give larger offsets

- General features
 - All instructions are encoded as 32-bit words
 - Only load and store instructions can access memory
 - Arithmetic and logic instructions use register operands
- Load and Store instructions
 - LDR & STR read and write 32-bit memory words
 - LDRB & LDRH read 8-bit or 16-bit values into low part of register
 - High bits within register are zero filled
 - LDRSB & LDRSH read 8-bit or 16-bit values into low part of register
 - High bits within register are filled with copies of sign bit

■ Other Store instructions

- STRB stores the low byte of a register into memory
- STRH stores the low 16 bits of a register into memory
 - Must use half-word aligned address (i.e., multiple of 2)

■ Block Transfer Instructions

- Transfer a block of consecutive memory words to or from a subset of general purpose registers
 - List of registers must appear in increasing order

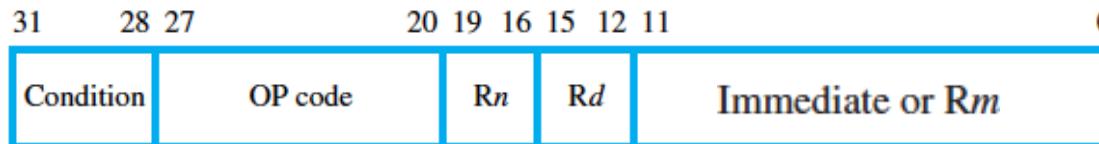
Example: base register R10 contains 1000 initially

LDMIA R10!,{R0, R2, R5, R7}

Loads contents of locations 1000,1004,1008 and 1012 into registers R0, R2, R5 and R7. Suffix IA means “increment after” (i.e., post increment) So R10 is incremented by 4 after each word transfer and final value 1016 is written to R10 due to the “!” writeback indicator.

■ Arithmetic instructions

- Use registers or immediate operands
- Assembly syntax is OP Rd,Rn,Rm
- Machine code format:



Addition and Subtraction

The instruction

ADD R0, R2, R4

performs the operation

$$R0 \leftarrow [R2] + [R4]$$

The instruction

SUB R0, R6, R5

performs the operation

$$R0 \leftarrow [R6] - [R5]$$

The second source operand can be specified in the immediate mode. Thus,

ADD R0, R3, #17

performs the operation

$$R0 \leftarrow [R3] + 17$$

The immediate operand is an 8-bit value contained in bits b_{7-0} of the encoded machine instruction. It is an unsigned number in the range 0 to 255. The assembly language allows negative values to be used as immediate operands. If the instruction

ADD R0, R3, #-17

is used in a program, the assembler replaces it with the instruction

SUB R0, R3, #17

■ Shifting of second source register operand

- Register second operand can be shifted before use
 - LSL (logical shift left)
 - LSR (logical shift right)
 - ASR (arithmetic shift right)
 - ROR (rotate right)
- Specified after the register name:

ADD R0,R2,R4,LSL #4

- R4 is shifted left 4 bits ($R4 * 16$), then added to R2 & sum is placed into R0.

- Shifting of second source immediate operand
 - Contained in low byte of machine instruction (0 – 255)
 - Expanded into a limited number of 32-bit values
 - Programmer specifies value
 - Assembler zero-extends 8-bit value to 32 bits & rotates an even number of bits to generate the value
 - The shift amount and 8-bit value are encoded in the low-order 12 bits of the machine instruction

■ Multiple-Word Operands

- Carry flag, C, is used to perform multiple precision addition & subtraction
- ADC (add with carry) & SBC (subtract with carry)

Example - to add the 64-bit number in the register pair R2,R3 to the 64-bit number in the register pair R4,R5:

ADDS R7,R3,R5 ;add low parts

ADC R6,R4,R2 ;add high parts & include carry

The S suffix causes the ADD to set the condition flags

■ Multiplication

- Two basic forms of multiply instruction
- MUL R0,R1,R2 ; R0 \leftarrow [R1] \times [R2]
 - Only the low 32 bits of 64-bit product are retained
- MLA R0,R1,R2,R3 ; R0 \leftarrow ([R1] \times [R2]) + [R3]
 - Called multiply and accumulate
 - Often used in signal-processing applications
- Other versions of MUL and MLA are available
 - Generate 64-bit results
 - Handle signed and unsigned operands

- No provision for shifting or rotating operands in multiplication
- No divide instructions
 - Division must be done in software

■ Move Instructions

- Copies an immediate value or contents of a register into a destination register
- **MOV Rd,Rm ; Rd ← [Rm]**
- **MOV Rd, #value ; Rd ← value (8-bit immediate)**
- **MVN R0,#4 ; moves one's complement ; representation of 4 into R0**
 - To move 2's-complement representation of c into a register, MVN can be used with c-1 as immediate operand
 - E.g. MVN R0,#4 places 2's-complement representation of -5 into R0 (0xFFFFFFF5 is -4 as 1's-comp, -5 as 2's-comp)
 - Assembler treats MOV R0,#-5 as pseudo-instruction and substitutes MVN R0,#4

■ Implementing shift and rotate instructions

- There are no explicit ARM shift or rotate instructions
- Shifting or rotating source register in Move instructions produces the same effect.
 - `MOV Rd,Rm, LSL #4` shifts [Rm] left 4 bits and places result into Rd
 - `MOV R3, R3, ROR #16 ; swap left & right halfwords in R3`

■ Logic instructions

- AND, OR, XOR (bit-wise AND, OR, XOR)
- Bit-Clear
 - BIC R0, R0, R1 ; where there is a 1 bit in R1, the ; corresponding bit in R0 is cleared
 - Works by ANDing 1's-complement of [R1] with [R0]

■ Bit Test instructions

- TST R2,#1 ; sets Z flag = 0, if LSB of R2 is 1 (non-zero)
 - ANDs [R2] with immediate value
- TEQ R2,#6 ; sets Z = 1, if [R2] = 6
 - XOR's the immediate value with the register

■ Compare instructions

- CMP Rn,Rm ; sets condition flags based on $[Rn] - [Rm]$
 - Result is discarded
 - Second operand may be an immediate value
- CMN Rn,Rm ; sets condition flags based on $[Rn] + [Rm]$
 - Compare negative
 - Second operand may be an immediate value
- Second operand may be shifted before use

■ Branch instructions

- Conditional branch instructions contain a 24-bit two's complement branch offset
 - offset is shifted left 2 bits & sign-extended to 32 bits
 - Offset + updated PC = branch target address
 - Instruction condition field (bits 31 – 28) indicate test to be performed (branch is taken if condition is met)
- Conditions are defined below:

Condition field encoding in ARM instructions.

Condition field $b_{31} \dots b_{28}$	Condition suffix	Name	Condition code test
0 0 0 0	EQ	Equal (zero)	$Z = 1$
0 0 0 1	NE	Not equal (nonzero)	$Z = 0$
0 0 1 0	CS/HS	Carry set/Unsigned higher or same	$C = 1$
0 0 1 1	CC/LO	Carry clear/Unsigned lower	$C = 0$
0 1 0 0	MI	Minus (negative)	$N = 1$
0 1 0 1	PL	Plus (positive or zero)	$N = 0$
0 1 1 0	VS	Overflow	$V = 1$
0 1 1 1	VC	No overflow	$V = 0$
1 0 0 0	HI	Unsigned higher	$\overline{C} \vee Z = 0$
1 0 0 1	LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
1 0 1 0	GE	Signed greater than or equal	$N \oplus V = 0$
1 0 1 1	LT	Signed less than	$N \oplus V = 1$
1 1 0 0	GT	Signed greater than	$Z \vee (N \oplus V) = 0$
1 1 0 1	LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
1 1 1 0	AL	Always	
1 1 1 1		not used	

■ Subroutine Linkage instructions

- Branch and Link instruction is used to call subroutines
 - BL SQRT ; call routine with name SQRT
 - Return address (address of instruction after BL) is loaded into R14 (the link register)
 - Link register must be saved on stack before a nested call
 - R13 is used as the processor stack pointer
- STMD is used to save (push) registers onto stack
 - Suffix FD means predecrement R13 toward lower addresses
- LDMD is used to restore (pop) contents of registers
- Restoring PC (R15) returns to calling routine
 - MOV R15,R14 ; copy link register into PC to return

■ Supports 7 Modes

- System Mode

- Privileged mode for exception handling
- Uses same registers as user mode
- Can only be entered from another exception

- User Mode

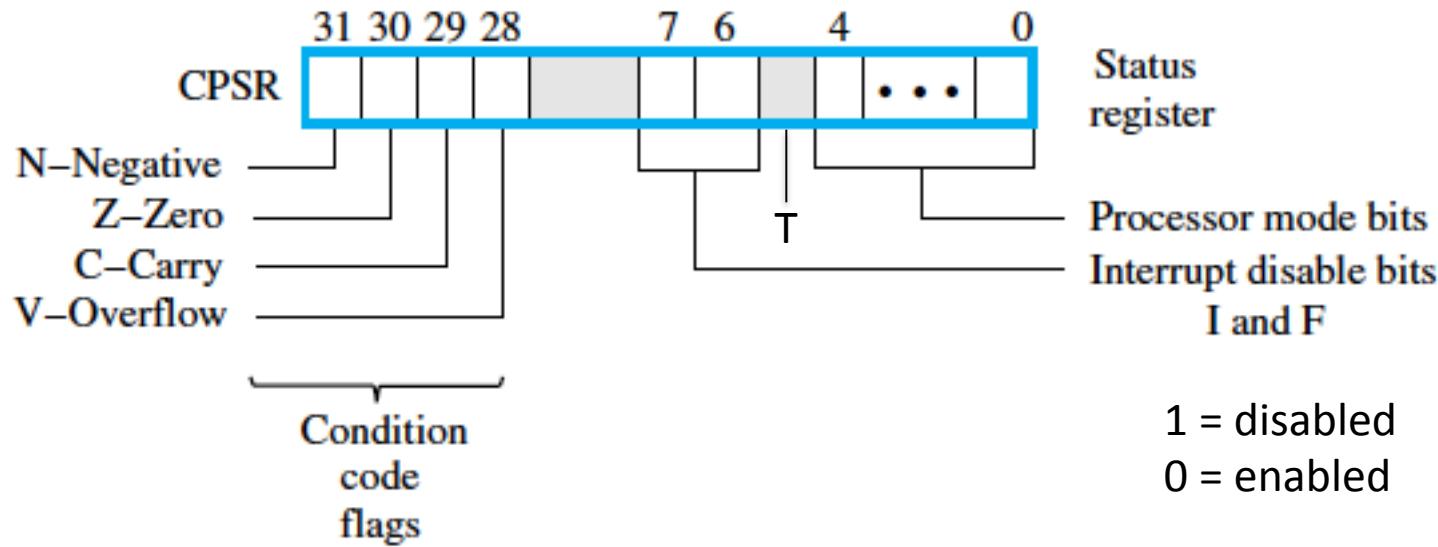
- For user applications

- Five exceptions modes

■ Exception Types

- Fast interrupt (FIQ) mode is entered when an external device raises a fast-interrupt request to obtain urgent service.
- Ordinary interrupt (IRQ) mode is entered when an external device raises a normal interrupt request.
- Supervisor (SVC) mode is entered on powerup or reset, or when a user program executes a Software Interrupt instruction (SWI) to call for an operating system routine to be executed.
- Memory access violation (Abort) mode is entered when an attempt by the current program to fetch an instruction or a data operand causes a memory access violation.
- Unimplemented instruction (Undefined) mode is entered when the current program attempts to execute an unimplemented instruction.

System Mode & exception modes are privileged modes.
Access to CPSR is allowed so I and F bits can be changed



User mode is unprivileged, so instructions that change CPSR are not available.

■ Exception Handling

- Vector table in low memory maps exception code

Exceptions and processor modes.

Exception	Processor mode entered	Vector address	Priority (Highest = 1)
Fast interrupt	FIQ	28	3
Ordinary interrupt	IRQ	24	4
Software interrupt	Supervisor (SVC)	8	–
Powerup/reset	Supervisor (SVC)	0	1
Data access violation	Abort	16	2
Instruction access violation	Abort	12	5
Unimplemented instruction	Undefined	4	6

■ Banked Registers

- Exceptions switch from user mode to one of 5 exception modes
 - Extra (banked) registers are substituted for some of the 16 normal registers used in System or User mode
 - Replaced registers are left unchanged (no need to save)
 - There is a set of banked registers for each exception mode
-
- “*Banked*” registers are used in exception handling

Accessible registers in different modes of the ARM processor.

User/System	FIQ	IRQ	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq		R8	R8	R8
R9	R9_fiq		R9	R9	R9
R10	R10_fiq		R10	R10	R10
R11	R11_fiq		R11	R11	R11
R12	R12_fiq		R12	R12	R12
R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15	R15	R15	R15	R15	R15

Processor status register



■ Actions taken when exception occurs

1. The contents of the Program Counter (R15) are loaded into the banked Link register (R14_mode) of the exception mode.
2. The contents of the Status register (CPSR) are loaded into the banked Saved Status register (SPSR_mode).
3. The mode bits of CPSR are changed to represent the appropriate exception mode, and the interrupt-disable bits I and F are set appropriately.
4. The Program Counter (R15) is loaded with the dedicated vector address for the exception, and the instruction at that address is fetched and executed to begin the exception-service routine.

■ Return from exception

- Operating System initializes R13_mode to point to top of stack area for each exception mode
- Return from an exception handler is performed by copying the mode link register (R14_mode) into the program counter and copying the SPSR-mode register into the CPSR
 - Example: SUBS PC,R14_irq,#4
 - Sets PC = R14_irq – 4
 - The S suffix means copy SPSR_irq into CPSR
 - MOVS PC,R14_svc returns from software interrupt

Address correction during return from exception.

Exception	Saved address*	Desired return address	Return instruction
Undefined instruction	PC+4	PC+4	MOVS PC, R14_und
Software interrupt	PC+4	PC+4	MOVS PC, R14_svc
Instruction Abort	PC+4	PC	SUBS PC, R14_abt, #4
Data Abort	PC+8	PC	SUBS PC, R14_abt, #8
IRQ	PC+4	PC	SUBS PC, R14_irq, #4
FIQ	PC+4	PC	SUBS PC, R14_fiq, #4

*PC is the address of the instruction that caused the exception. For IRQ and FIQ, it is the address of the first instruction not executed because of the interrupt.

	Memory address label	Operation	Addressing or data information
Assembler directives		AREA ENTRY	CODE
Statements that generate machine instructions	LOOP	LDR LDR MOV LDR ADD SUBS BGT STR	R1, N R2, POINTER R0, #0 R3, [R2], #4 R0, R0, R3 R1, R1, #1 LOOP R0, SUM
Assembler directives	SUM N POINTER NUM1	AREA DCD DCD DCD DCD END	DATA 0 5 NUM1 3, -17, 27, -12, 322

Assembly-language source program

- Evolved from the x86 architecture
 - Uses 32-bit memory addresses
 - 32-bit registers
 - Supports 8-bit, 16-bit, 32-bit and 64-bit operands
 - Byte-addressable memory
 - Allows unaligned memory accesses
 - Employs little-endian memory storage order
 - Presents a CISC architecture to programmer
 - Breaks down CISC instructions into micro-ops
 - Micro-ops execute internally like RISC instructions

■ Memory Organization

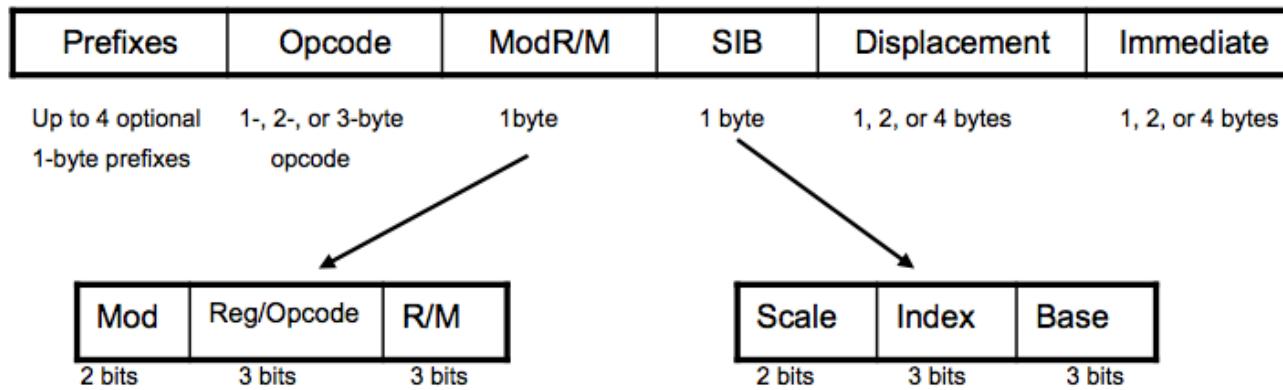
- Early x86 machines defined “word” as 16 bits
- IA-32 “double-word” is a 32-bit item
- “Quad-word” is a 64-bit item
- Uses 32-bit memory addresses
- Earlier x86 machines used 20-bit addresses
 - Derived from 16-bit segment registers and an offset
 - For code, stack and four data segments
- IA-32 architecture supports flat address space
 - Maps all segments to a common 4 GB space
 - Shared by code, stack and data areas

- IA-32 instructions vary in length
 - Unlike fixed-length instructions on RISC processors
 - Instruction encoding is complex
- Only 8 general purpose register
 - Can be treated as 8-bit or 16-bit parts
 - Maintains compatibility with older x86 instructions
- Supports CISC instructions
 - These do the work of an entire routine on RISC system
 - Such as string search and manipulation
 - Arithmetic/logic instructions can use memory operands
 - Not a load/store architecture

- Floating point unit uses separate set of 8 registers
 - Registers are organized as a stack ST(0) – ST(7)
 - Internal 80-bit extended precision float format is used
 - Converts IEEE 754 floats on the way to/from memory
- Supports SIMD vector type operations
 - MMX for integer operands
 - Multi-Media Extensions
 - SSE for floating point operands
 - Streaming SIMD Extensions

- Performs I/O using special instructions
 - IN reads from an I/O device port
 - OUT sends data to an I/O device port
 - I/O ports are mapped to separate I/O space
 - Memory-mapped I/O can also be used
 - Using the normal memory access instructions
 - Like the only option for I/O on RISC systems

- IA-32 Machine Instruction format is complex
 - Vary in size from 1 to 14 bytes
 - Complicates prefetching instructions

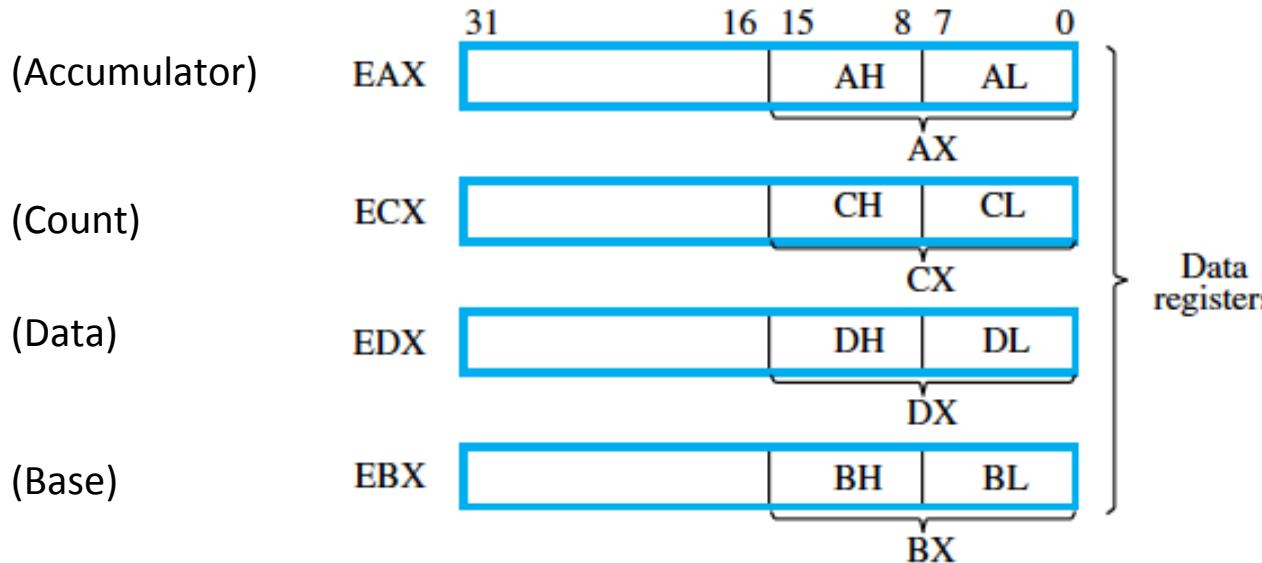


- Opcode may be 1, 2 or 3 bytes
- Other fields are optional and depend on the instruction type and the addressing mode used

- Eight 32-bit General Purpose Registers
 - 4 Data Registers
 - 2 Pointer Registers
 - 2 Index Registers
- 32-bit Program counter
 - called Instruction pointer (EIP)
- 32-bit Status Register
 - Contains condition flags
- Register names, not numbers, are used

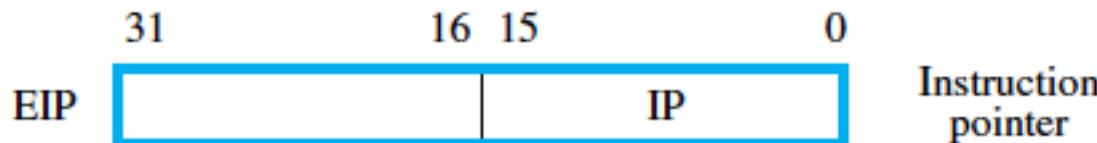
- 8-bit Intel processors used 8-bit registers
 - Data registers were called A, B, C, and D.
- Later 16-bit processors used 16-bit registers
 - Data registers were called AX, BX, CX and DX.
- IA-32 Architecture uses 32-bit registers
 - Data registers are called EAX, EBX, ECX and EDX
 - E-prefix indicates 32-bit “extended” versions

■ Data Registers

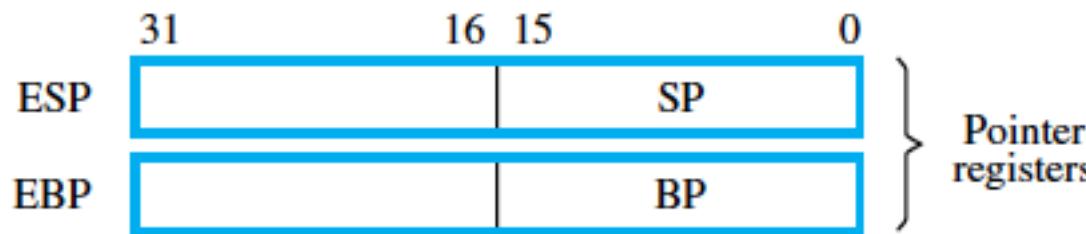


- EAX, ECX, EDX & EBX are 32-bit (extended)
- AX, CX, DX, and BX used for 16-bit items
- AH, AL, CH, CL, DH, DL, BH, BL are for 8-bit items
 - Maintains compatibility with earlier x86
 - Correctly runs 16-bit code on IA-32 processors

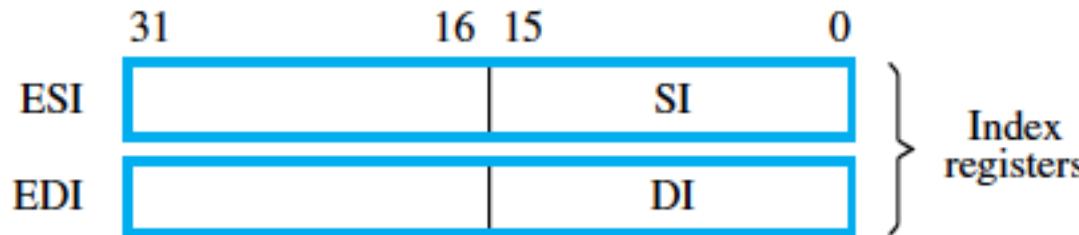
- Points to next instruction to execute
 - IP refers to the 16-bit instruction pointer
 - Limits code segment to 64 KB
 - EIP is the 32-bit extended instruction pointer
 - Code segment as large as 4 GB

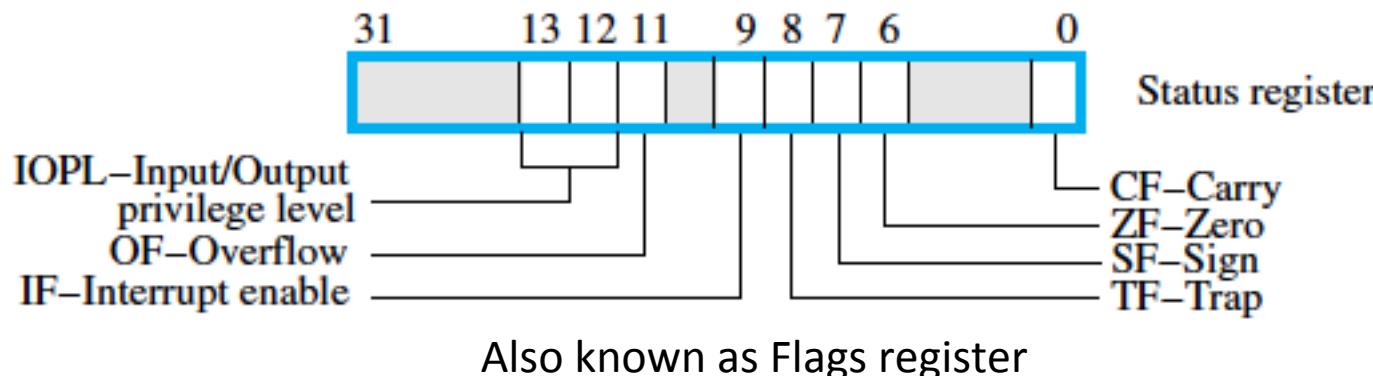


- ESP & SP are 32-bit & 16-bit stack pointers
 - Points to stack in memory
- EBP & BP are 32-bit & 16-bit base registers
 - Used for call frames and data



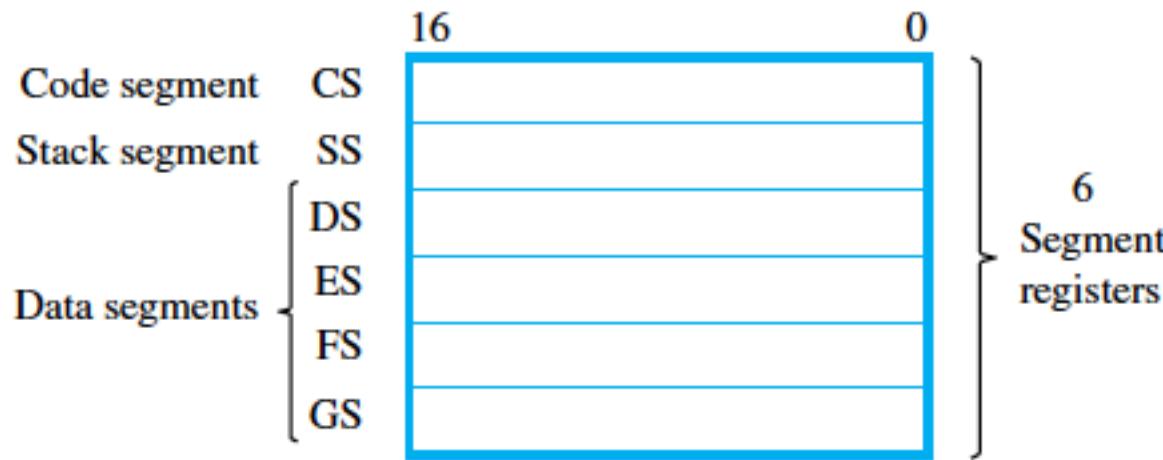
- ESI & SI are 32-bit & 16-bit source index
- EDI & DI are 32-bit & 16-bit destination index
 - Used to reference memory operands
 - Array and string indices





- CF, ZF, SF, & OF are condition code flags
 - Reflect the result of arithmetic operations
- IOPL, IF, TF for I/O operations & interrupts
 - IF is interrupt enable/disable
 - TF is for single stepping through code
 - IOPL is set by the OS to control which operations are allowed

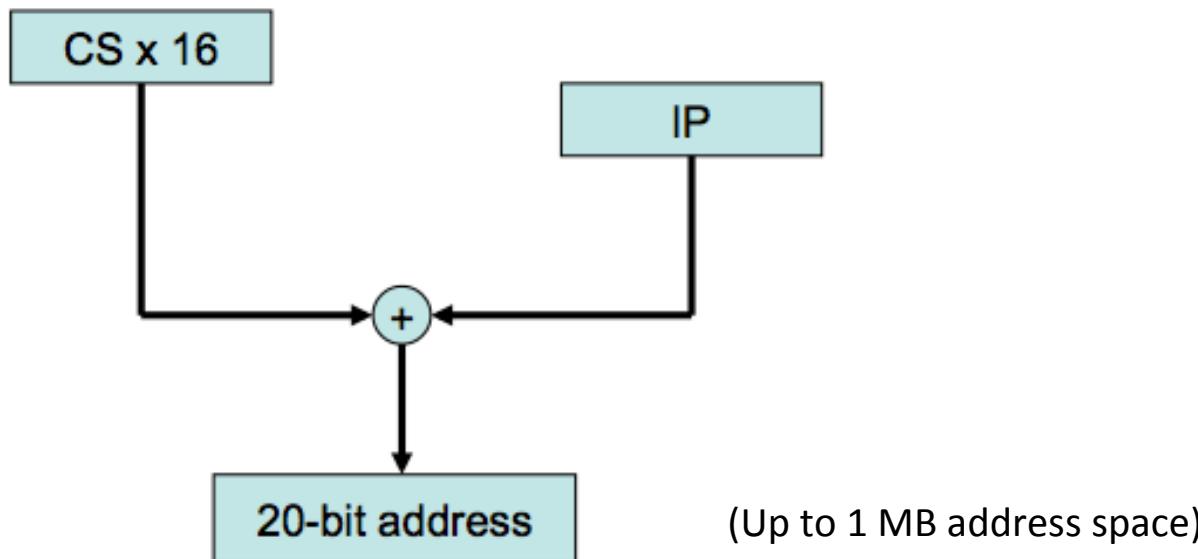
- Earlier x86 processors used memory segment
 - Segment starting address was held in a register
 - Segment registers are 16-bit
 - Segments were limited to 64 KB size



- Memory address were specified as segment/offset
 - Segment_reg_contents:offset_value (xxxx:yyyy)

Address Generation Example:

The instruction Pointer register (IP) contains the offset within the current code segment from which the next instruction is fetched. On the 8086 this was a 16-bit register which is combined with the code segment register as shown below to produce the 20-bit memory address:



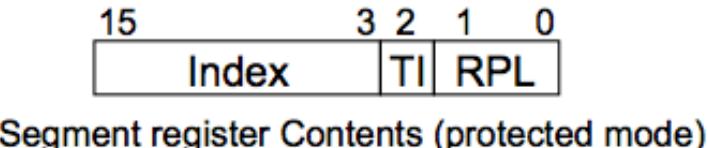
Memory operand addresses are generated in a similar way, but one of the other segment registers is combined with a 16-bit offset.

- Use of registers is based on Processor Mode
 - Real Mode
 - Segment registers are used the same as on x86 processors
 - Addresses are 20 bits
 - Protected Mode uses more complex segmentation
 - Segment registers contain index into descriptor table
 - 1 Local Descriptor table (LDT) for each task
 - Global Descriptor table (GDT) shared by all tasks
 - Interrupt Descriptor table (IDT) used by O.S.
 - Table entries contain 32-bit base address
 - Start address + 32-bit offset = 32-bit linear address
 - Overlapping segments provides a flat address space
 - Segments can be up to 4 GB in size

Protected Mode Segment Registers

Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.
Segment selector	Segment base address, size, access rights, etc.

In protected mode, every segment register has a “visible” part and an “invisible” part. The visible part is referred to as the segment selector. The invisible part is automatically loaded by the processor from a descriptor table.

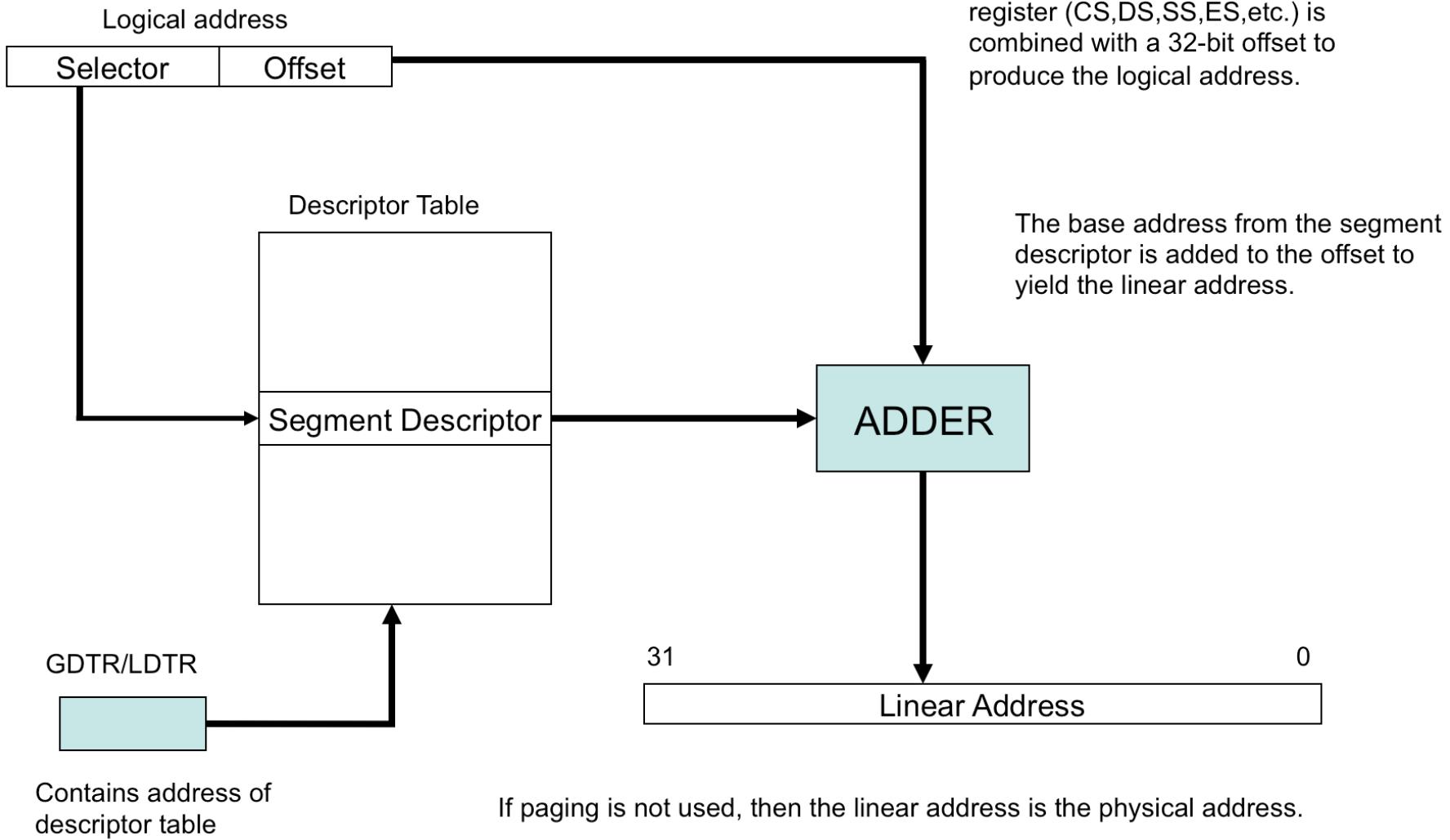


RPL – request privilege level

TI - table indicator 0=GDT, 1=LDT

Index – points to descriptor in table

Protected Mode Address Translation



- IA-32 addressing modes are CISC like
 - Large number of flexible addressing modes
 - Register Mode – operands are in registers
 - Immediate Mode – instruction contains operand
 - Direct mode – instruction contains operand address
 - Register indirect – operand address in register
 - Base with displacement – $[reg] + \text{displacement} = \text{op adrs}$
 - Index with displacement – $[reg] * \text{scale} + \text{disp.} = \text{op adrs}$
 - Base with index – $\text{op adrs} = [breg] + [ireg] * \text{scale}$
 - Base with index & displacement
 - $\text{op adrs} = [\text{base_reg}] + [\text{index_reg}] * \text{scale} + \text{displacement}$

- IA-32 addressing modes are CISC like
 - Large number of flexible addressing modes
 - Register Mode – operands are in registers
 - Immediate Mode – instruction contains operand
 - Direct mode – instruction contains operand address
 - Register indirect – operand address in register
 - Base with displacement – $[reg] + \text{displacement} = \text{op adrs}$
 - Index with displacement – $[reg] * \text{scale} + \text{disp.} = \text{op adrs}$
 - Base with index – $\text{op adrs} = [breg] + [ireg] * \text{scale}$
 - Base with index & displacement
 - $\text{op adrs} = [\text{base_reg}] + [\text{index_reg}] * \text{scale} + \text{displacement}$

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] × S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value = an 8- or 32-bit signed number

Location = a 32-bit address

Reg, Reg1, Reg2 = one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register.

Disp = an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.

S = a scale factor of 1, 2, 4, or 8

- Instructions can have 0, 1 or 2 operands
 - Two-operand syntax: OP destination,source
 - Examples based on MOV instruction:
 - MOV EBP,EAX copies EAX reg into EBP reg
 - MOV EAX,25 copies 32-bit constant into EAX
 - MOV AX,320 copies 16-bit constant into AX
 - MOV AL,125 copies 8-bit constant into AL
 - MOV EAX,LOC1 copies 32 bits at LOC1 into EAX
 - MOV EBX, OFFSET LOC1
 - Puts address LOC1 into EBX
 - MOV EAX,[EBX]
 - EAX = 32-bit contents of location whose address is in EBX

■ Base with displacement examples

- Assume that EBP contains 2000
- MOV EAX,[EBP+60]
 - Copies contents of doubleword (32 bits) at 2060 into EAX
- MOV AL,[EBP+60]
 - Copies contents of byte (8 bits) at 2060 into AL
- MOV [EBP+67],AH
 - Copies contents of AH into byte at address 2067
- MOV [EPB+100],28 size of constant is unclear
 - MOV BYTE PTR [EBP+67],28 for 8-bit
 - MOV WORD PTR [EBP+67],28 for 16-bit
 - MOV DWORD PTR [EBP+67],28 for 32-bit

- Base with index & displacement example
 - Assume that [EBP] = 2000 & [ESI] = 0
 - MOV EAX,[EBP + ESI*4 + 100]
 - Copies contents of doubleword (32 bits) at 2100 into EAX
 - To copy contents of doublewords at 2100, 2104, 2108, etc. place in loop and increment ESI by 1 for each iteration
 - E.g., to step through array of 32-bit elements
 - Scale factor of 2 for 16-bit elements
 - Scale factor of 1 for 8-bit elements or characters

- Instructions can have 0, 1 or 2 operands
 - Zero operand examples:
 - PUSHAD pushes all 8 data regs onto stack
 - POPAD restores all 8 regs from stack (popping)
 - One operand examples:
 - INC EDI adds 1 to EDI register
 - DEC EBX subtracts 1 from EBX register
 - Two operand examples:
 - ADD EAX,EBX
 - MUL EBX,511

■ Load-effective-address instruction

- LEA EAX,LOC1
 - Puts address corresponding to LOC1 into EAX
 - Address is part of the instruction
- MOV EAX,OFFSET LOC1 has same effect
 - Address of LOC1 is known by assembler
- LEA EBX,[EBP + 8]
 - Puts address = (contents of EBP) + 8 into EBX at runtime
 - Assembler can't know what EBP will contain

■ Arithmetic instructions

- May use all register operands
- one operand may be in memory
- Operands can be 8-bit or 32-bit
- Examples:
 - ADD EAX,EBX
 - ADC EAX,EBX
 - SUB EBX,[EAX+4]
 - SBB EAX,EBX
 - CMP [EBX + 10],AL

■ Multiply instructions

- IMUL EBX implicit multiplicand is EAX
 - Computes $[EAX]*[EBX]$
 - Puts 64-bit product into EDX,EAX (high,low)
- IMUL EBX,[EBP]
 - Puts low 32 bits of product dest*src into dest
 - OF flag = 1 if high half of 64-bit product is nonzero
- IMUL does signed multiply
- MUL does unsigned multiply
- Source can be immediate, register or in memory
- Destination must be a register

■ Division instructions

- IDIV *src*
 - Divides EDX,EAX pair by src
 - 32-bit value in EAX must be sign extended to 64-bits
 - CDQ converts EAX into 64 bits in EDX,EAX
 - Sets EAX=quotient and sets EDX=remainder
 - Division by zero causes an exception
- DIV does unsigned divide
- Source can be immediate, register or in memory

Conditional Jump instructions

- All branches are called “jumps” with IA-32
- Tests condition codes to decide

Mnemonic	Condition name	Condition test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF \vee ZF = 0
JBE	Unsigned below or equal	CF \vee ZF = 1
JGE	Signed greater than or equal	SF \oplus OF = 0
JL	Signed less than	SF \oplus OF = 1
JG	Signed greater than	ZF \vee (SF \oplus OF) = 0
JLE	Signed less than or equal	ZF \vee (SF \oplus OF) = 1

\vee denotes OR

\oplus denotes XOR

E.g.: JG EXIT

■ Conditional Jump instructions

- Assembler generates signed offset relative to next location
 - One-byte offset if in the range -128 to +127
 - Otherwise a 4-byte offset is generated
- IP + Offset = target address

■ Loop Instruction

```
        MOV    ECX,NUM_PASSES
START:   .
        .
        .
        DEC    ECX
        JG     START
```

```
        MOV    ECX,NUM_PASSES
START:   .
        .
        .
        LOOP   START
```

LOOP instruction implicitly decrements ECX and branches if > 0

- JMP is the unconditional jump
 - Uses 1-byte or 4-byte signed offset
 - May also use other addressing modes for target address
 - Example: JMP [JUMPTBLE + ESI * 4]
 - Uses ESI as Index into table of jump addresses
 - Implements high-level language Case statement

■ Logic Instructions

- AND, OR, XOR
 - All use 2 operands and put result into destination
 - Example: AND EBX,EAX
- NOT uses a single operand
 - Takes 1's complement (i.e. flips each bit)
 - Example: NOT EAX
- NEG negates it's single operand (takes 2's complement)
- Test Instruction Format: TEST dst,src
 - Sets ZF=0 in any matching bits are set in src and dst
 - Operands are ANDed but neither operand is modified
 - destination may be a register or in memory
 - source may be a register, in memory or immediate

■ Bit Test Instructions

- BT op1,n Sets CF = bit n within op1
- BTC op1,n Sets CF = bit n within op1 & flip bit n within op1
- BTR op1,n Sets CF = bit n within op1 & clear bit n within op1
- BTS op1,n Sets CF = bit n within op1 & set bit n within op1
- The operand op1 may be a register or in memory

- Other examples of available CISC type instructions include:
- BSF & BSR bit scan forward and bit scan reverse
- SCANSB, SCANSW, SCANSO for scanning strings in search of a particular value (direction of scan is controlled by DF flag in status register).

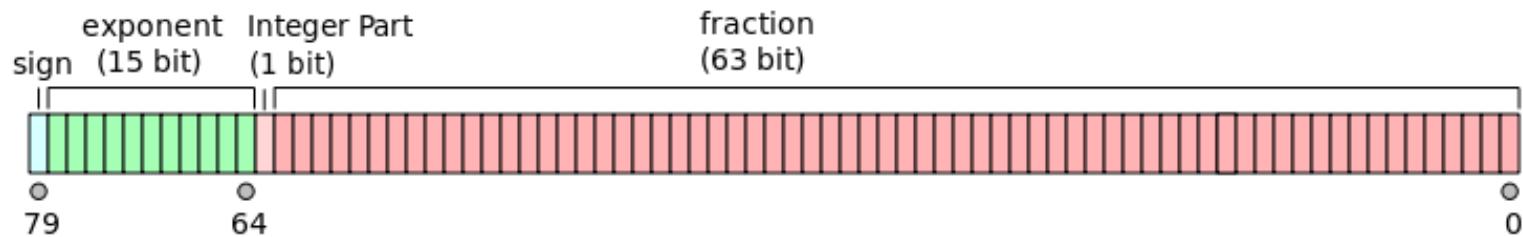
- Shift Instructions
 - SHL (shift left logical)
 - SHR (shift right logical)
 - SAL (shift left arithmetic; same as SHL)
 - SAR (shift right arithmetic)
- Rotate Instructions
 - ROL (Rotate left without the carry flag CF)
 - ROR (Rotate right without the carry flag CF)
 - RCL (Rotate left including the carry flag CF)
 - RCR (Rotate right including the carry flag CF)
- Shift & Rotate instruction format: OP dst, count
- dst is shifted (Any addressing mode can be used)
- Count must be an 8-bit immediate or in the 8-bit CL register (low byte of ECX)

■ Subroutine Linkage

- The CALL instruction is used to call subroutines
- Example: CALL ROUTINE
 - The address of the instruction following the CALL is the return address
 - The return address is pushed onto the stack before the transfer
 - ESP register points to the top of stack (TOS)
 - The stack grows downward toward lower addresses
 - PUSHAD can be used to save all 8 general purpose registers before call
 - The subroutine uses POPAD to restore the registers before returning
 - PUSH & POP may be used to handle individual registers or data items
 - RET returns control to caller by popping TOS into EIP
 - EBP register points to call frame on stack
 - LIFO stack facilitates nested subroutine calls

- Floating Point Unit (FPU) executes F.P. instructions

- Has its own set of 8 registers
 - FPU & CPU execute their instructions concurrently
 - FPU uses 80-bit internal format for all operations
 - Converts IEEE-754 formatted operands to/from 80-bit format



Bit 79 is the sign bit (s)

Bits 64 – 78 give the 15-bit excess-16383 exponent (e)

Bit 63 = 1 for normalized values, = 0 for denormalized (i)

Bits 0 – 62 give the 63-bit fraction (f) [there is no hidden bit]

Number point is between bits 62 & 63

Value represented is $(-1)^s \times i.f \times 2^{e-16383}$

■ Floating Point Stack

- The 8 FPU registers are organized as a stack
 - ST(0) to ST(7) from top to bottom
 - Modulo-8 pointer defines top of stack
- FLD copies its single memory operand onto stack ST(0)
- FST writes contents of ST(0) into memory
 - FSTP does the same but also pops ST(0) from stack
- FI^IST converts ST(0) to 32-bit integer & stores in memory
 - FI^ISTP does same but also pops ST(0)

■ Floating Point Load/Store examples

- **FLD DWORD PTR[EAX]**
 - Converts 32-bit float in memory into 80-bit float in ST(0)
 - 80-bit value is pushed onto FPU stack
- **FST QWORD PTR [EDX + 8]**
 - converts 80-bit float in ST(0) into 64-bit float in memory
- **FST writes contents of ST(0) into memory**
 - FSTP does the same but also pops ST(0) from stack
- **FISTP [EDX + 8]**
 - Pops ST(0) converts to 32-bit integer and stores in memory

■ Floating Point Arithmetic Instructions

- FADD QWORD PTR[EAX]
 - Converts 32-bit float in memory into 80-bit float & adds to ST(0)
- FSUBP ST(1),ST(0)
 - puts ST(1)-ST(0) into ST(1) , pops ST(0), so result is now ST(0)
- FSUBR ST(2),ST(0)
 - Puts $ST(0) - ST(2)$ into ST(2) [reverse subtract]
 - FDIVR is similar
- FISUB DWORD PTR [EDX + 8]
 - Converts 32-bit integer, sets $ST(0) = ST(0) - \text{float equivalent}$
 - FIADD, FIMUL & FIDIV are similar

■ Floating Point Compare Instructions

- FCOMI ST(0),ST(4)

- Compares register with ST(0) and sets condition codes
 - ST(0) must be destination operand

- FCOMIP ST(0),ST(2)

- Similar to FUCOMI but also pops stack

- Additional Float Instructions

- FCHS changes sign of implicit operand ST(0)
 - FABS takes absolute value
 - FSQRT computes square root of ST(0)
 - FSIN & FCOS compute sine and cosine of ST(0)
 - FLDZ & FLD1 push 0.0 & 1.0 as 80-bit float
 - FLDPI pushes π as 80-bit float

- IA-32 has Vector or SIMD instructions
 - Images can be represented as matrices of pixels
 - Color & brightness are encoded in each element
 - Multiple pixels can be packed into a single elements
 - Simple arithmetic & logic operations are applied
 - SIMD instructions act on multiple pixels in parallel

- IA-32 has multimedia extensions (MMX)
 - Multiple data elements are packed into 64-bit quadwords
 - Operands can be in memory or in FPU registers
 - FPU registers correspond to MM0 – MM7
 - Lowermost 64 bits within each 80-bit register are used
 - MMX registers are not managed as a stack
 - Example MMX instructions
 - MOVQ MM0,[EAX] loads 64 bits from memory
 - MOVQ MM3,MM4 copies MM4 into MM3

- Example MMX arithmetic instructions

- PADDB MM2,[EBX] add packed bytes
 - adds 8 memory bytes to corresponding bytes in MM2
 - The sums are computed in parallel
 - B suffix indicates 8 bytes
 - W indicates four 16-bit words
 - D indicates two 32-bit doublewords
 - Q indicates single 64-bit quadword
 - Other operations are also available:

- | | |
|---------|--------|
| ■ PSUB | ■ PAND |
| ■ PMUL | ■ POR |
| ■ PMADD | ■ PXOR |

- Streaming SIMD extensions (SSE)
 - Handle packed 128-bit double quadwords
 - 8 additional 128-bit registers XMM0 to XMM7
 - MOVAPS & MOVUPS transfer between memory and registers
 - PS suffix indicates packed single-precision float values
 - A or U indicates aligned or unaligned (16-bit word aligned)
- Examples:
 - MOVUPS XMM3,[EAX]
 - Copies 4 32-bit floats from memory into XMM3
 - MOVUPS XMM4,XMM5 copies XMM5 into XMM4
 - ADDPS XMM0,XMM1
 - Adds 4 corresponding pairs of 32-bit floats
 - SUBPS, MULPS & DIVPS are also available

■ Memory mapped Input/Output

- The code below is an example illustrating the use of mapped I/O
- Polls keyboard to echo characters until CR (carriage return) is hit

	LEA EBX, LOC	Initialize register EBX to point to the address of the first location in main memory where the characters are to be stored.
READ:	BT KBD_STATUS, 1 JNC READ	Wait for a character to be entered in the keyboard buffer KBD_DATA.
	MOV AL, KBD_DATA	Transfer character into AL (this clears KIN to 0).
	MOV [EBX], AL	Store the character in memory and increment pointer.
	INC EBX	
ECHO:	BT DISP_STATUS, 2 JNC ECHO	Wait for the display to become ready.
	MOV DISP_DATA, AL	Move the character just read to the display buffer register (this clears DOUT to 0).
	CMP AL, CR	If it is not CR, then
	JNE READ	branch back and read another character.

- Port Mapped I/O (isolated I/O) is common with IA-32

- Requires special Input/Output instructions
 - Examples:

IN AL,DX ; reads one byte from the devices whose port number is in DX

IN AX,DX ; reads two bytes of data from the port

IN EAX,DX ; reads 4 bytes

OUT 61h,AX ; sends two bytes to output port number 61h

OUT DX,AL ; sends one byte to the port whose number is in DX

Port mapped scheme uses I/O address space separate from the program address space.

- IA-32 Interrupts and Exceptions
 - Two interrupt lines
 - NMI non-maskable is always accepted by processor
 - INTR user interrupt is maskable
 - Enabled/disabled using IF flag within status register
 - Accepted if priority > privilege level of currently running program
 - Events other than external interrupts cause exceptions
 - Vector number is assigned to each interrupt or exception
 - This index identifies an IDT (interrupt descriptor table) entry
 - Table entry contains starting address of corresponding handler
 - I/O devices are connected through an APIC
 - Advanced Programmable Interrupt Controller
 - Controller implements priority scheme and sends vector number to CPU

■ Actions taken for Interrupts or Exceptions

1. Push status register, code segment register and EIP onto processor stack
2. For exceptions due to abnormal condition, push cause of exception
3. Disable further interrupts of the same type
4. Use vector number to load appropriate handler address from IDT into EIP

When finished, the handler uses the IRET instruction to resume the program.

IRET pops EIP, code segment register and status register from stack

Handler must undo any changes it made to stack pointer before executing IRET

- ❑ **SPARC** stands for **S**calable **P**rocessor **A**rchitecture.
- ❑ developed by Sun Microsystems in the 1980s.
- ❑ based on the RISC II designed at UC Berkeley in early 1980s
- ❑ "Scalable" from embedded systems to servers
- ❑ RISC design that shares many features with the MIPS system
- ❑ There are some major differences between Sparc V8 & MIPS
- ❑ Sparc V8 is the only version examined here

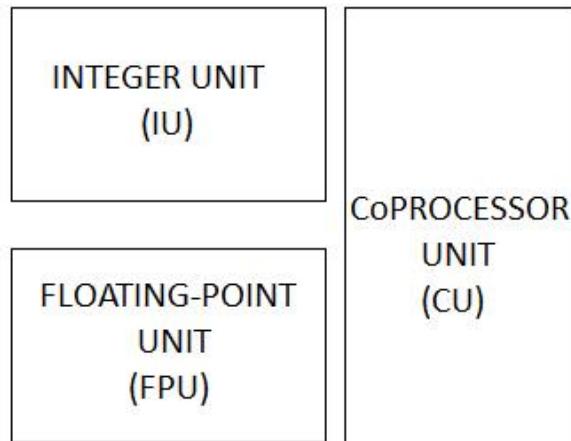
- Features Shared with MIPS:

- 32-bit registers and addresses
 - Register 0 is hardwired to 0
- Byte-addressable memory
 - 8-bit bytes, 16-bit halfwords and 32-bit words
- Load/store architecture
- Enforces memory alignment
- Employs big-endian memory storage order
- Delayed branches
- Passes arguments via registers
 - Using stack when needed

- Features that differ from MIPS:

- May have hundreds of registers
- Registers are grouped into “windows”
 - 32 registers are visible at one time
 - Windows slide and overlap to pass arguments
- Branch delay slot can be “annulled”
- Has next PC (NPC) as well as PC (program counter)
 - PC points to current instruction
 - NPC points to instruction to be executed next
- Instructions can set condition codes or not

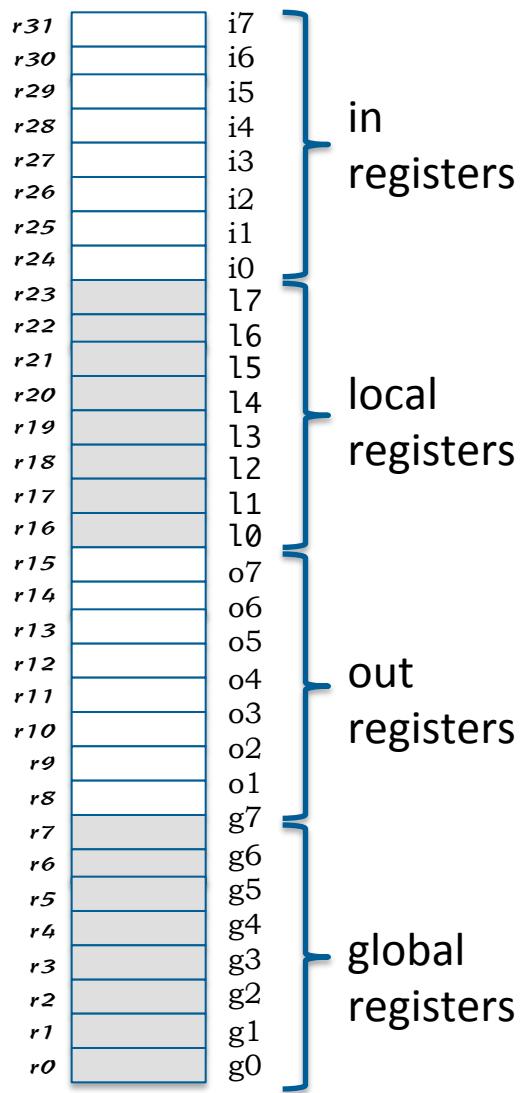
□ Sparc V8 Organization



- Units have separate register sets
- Units operate in parallel
- FPU has 32 floating point registers (not windowed)

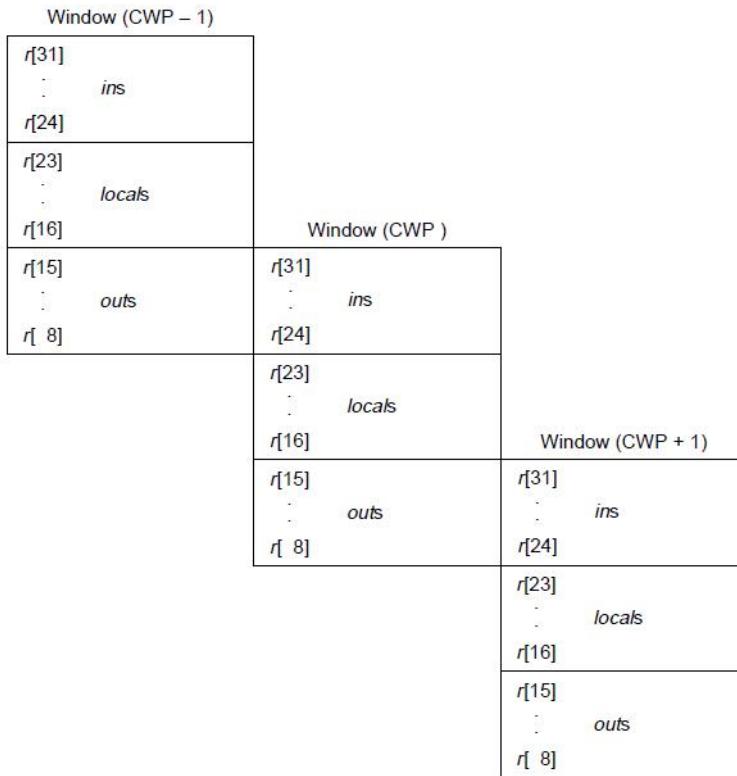
Names of registers visible to a user program at any one time:

Register	Alias	Usage
%g0	%r0	Hardwired to zero
%g1	%r1	The next seven are global registers
%g2	%r2	
%g3	%r3	
%g4	%r4	
%g5	%r5	
%g6	%r6	
%g7	%r7	
%o0	%r8	First of six registers for local data and subroutine arguments
%o1	%r9	
%o2	%r10	
%o3	%r11	
%o4	%r12	
%o5	%r13	
%sp	%r14, %o6	Stack pointer
%o7	%r15	Linkage register containing subroutine return address
%l0	%r16	First of eight registers for local variables
%l1	%r17	
%l2	%r18	
%l3	%r19	
%l4	%r20	
%l5	%r21	
%l6	%r22	
%l7	%r23	



Caller	Callee	Usage
%o0	%i0	first argument
%o1	%i1	second argument
%o2	%i2	third argument
%o3	%i3	fourth argument
%o4	%i4	fifth argument
%o5	%i5	sixth argument
%o6	%i6	frame pointer
%o7	%i7	return address

- Current window pointer (*CWP*)



There can be up to 32 register windows

Each window contains 24 registers

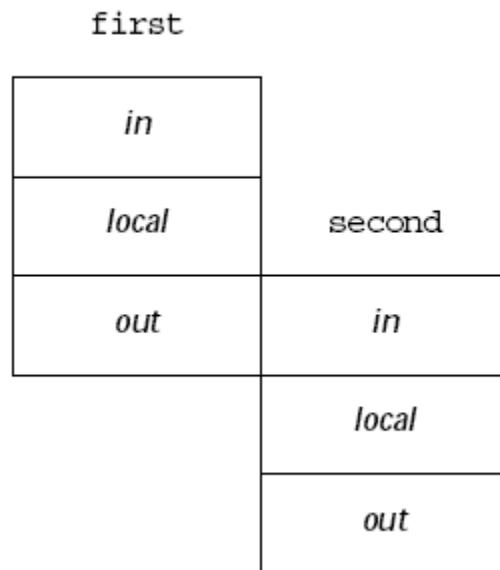
Windows can slide to pass arguments

Windows overlap

CWP identifies current window

- Example: function first calls second:

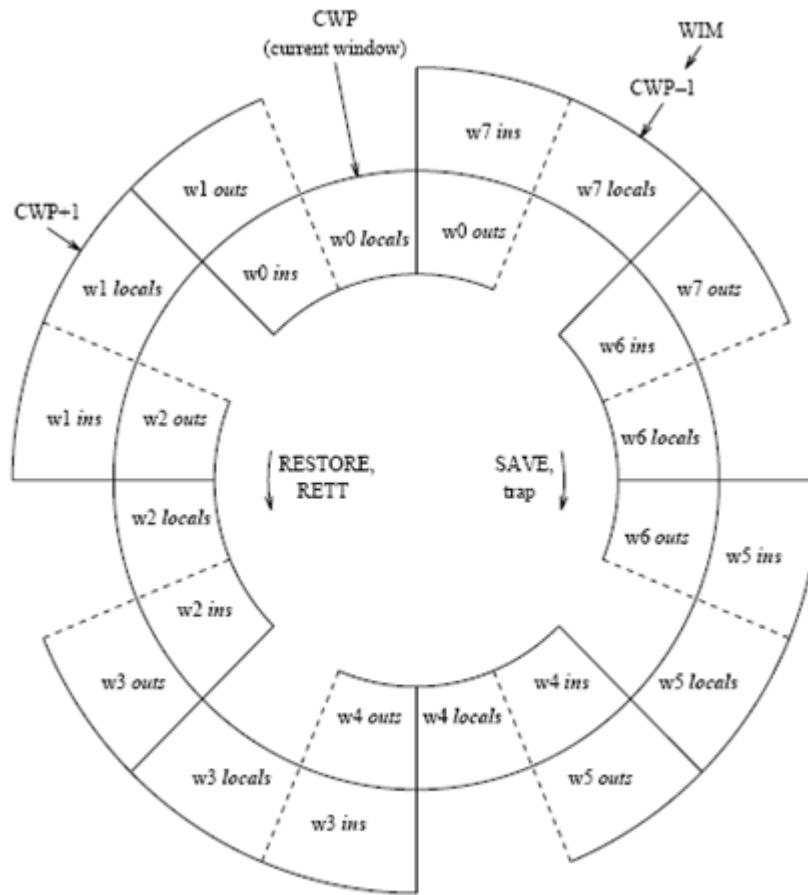
```
first()
{
    ...
    second();
    ...
}
```



second:

```
save    %sp, -80, %sp
```

restore instruction is used to unwind stack and slide window back

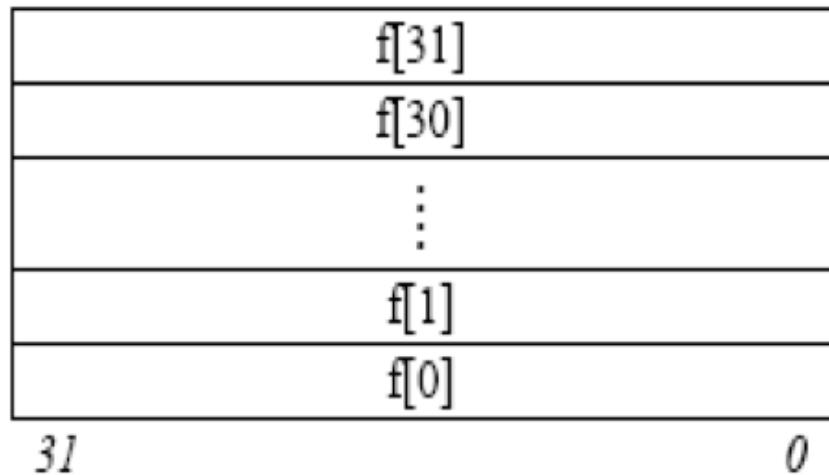


Special-purpose registers

PC	program counter	Holds the address of the currently executing instruction
nPC	next PC	Holds address used to fetch next instruction to be executed
PSR	processor state register	Holds state of CPU, condition codes, mode, trap enable, bit, etc.
WIM	window-invalid mask	Indicates when window underflow or overflow should occur
TBR	trap base register	Holds address of trap table and indicates trap type
Y	Y register	High part of product for mul ; high part of dividend for div
FSR	floating point state	Holds floating point mode and status, controls rounding and floating point traps

Floating point unit has a single set of 32 floating point registers

The f Registers



Each float register is 32 bits wide.

Supported Modes:

- Source operands are on left, result on right in assembly instructions
- Register mode (all operands in registers)
 - add %g2,%o4,%o1 $[%g2] + [\%o4] \rightarrow \%o1$
- Immediate mode
 - sub %o2, 23, %g4 $[\%o2] - 23 \rightarrow \%g4$
- Base register with displacement
 - ld $[%g2 + 8], \%o3$ $[%g2 + 8] \rightarrow \%o3$
- Register indirect with index
 - stb %o4, [%g4 + %o2] $[%o4] \rightarrow [%g4 + \%o2]$
 - stb %o4, [%g4 + %g0] $[%o4] \rightarrow [%g4]$ (%g0 always 0)

Loading a 32-bit constant or address into a register

To load an address into %g2:

sethi %hi(X), %g2	high 22 bits of address
or %g2, %lo(X), %g2	merge in low 10 bits

To load the constant 0x4A3C4098 into %o2:

0100101000111100010000 0010011000

sethi 0x128F10, %o4	high 22 bits
or %o4, 0x98, %o4	low 10 bits

Synthetic instructions:

set X, %g2
set 0x4A3C4098, %o4

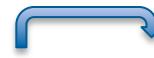
%hi() and %lo() are implemented by the assembler

Instructions fall into following categories :

- Load/store
- Arithmetic/logicalshift
- Control transfer
- Read/write control register
- Floating-point/Coprocessor operate

Load Integer Instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
LDSB	001001	Load Signed Byte
LDSH	001010	Load Signed Halfword
LDUB	000001	Load Unsigned Byte
LDUH	000010	Load Unsigned Halfword
LD	000000	Load Word
LDD	000011	Load Doubleword


ld [%sp - 4], %o2

Store Integer Instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
STB	000101	Store Byte
STH	000110	Store Halfword
ST	000100	Store Word
STD	000111	Store Doubleword


sh %o4, [%fp + 8]

SETHI Instruction

<i>opcode</i>	<i>op</i>	<i>op2</i>	<i>operation</i>
SETHI	00	100	Set High-Order 22 bits

sethi 0x4A3, %g2
sethi %hi(Z), %o3

SETHI zeroes the least significant 10 bits of “r[rd]”, and replaces its high-order 22 bits with the value from its *imm22* field.

A SETHI instruction with *rd* = 0 and *imm22* = 0 is defined to be a NOP

Shift Instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
SLL	100101	Shift Left Logical
SRL	100110	Shift Right Logical
SRA	100111	Shift Right Arithmetic

sll %g2, 4, %g2
sra %o4,%g4,%g2

Logical Instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	000011	Exclusive Or
XORcc	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNORcc	010111	Exclusive Nor and modify icc

and %g2, 4, %g2
xorcc %o4,%g4,%g2

13-bit immediate operands
are sign extended to 32 bits

Add and subtract instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

addcc %g2, 4, %g2
addxcc %o4,%g4,%g2

<i>opcode</i>	<i>op3</i>	<i>operation</i>
SUB	000100	Subtract
SUBcc	010100	Subtract and modify icc
SUBX	001100	Subtract with Carry
SUBXcc	011100	Subtract with Carry and modify icc

sub %g2, 4, %g2
subx %o4,%g4,%g2

Multiply Instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
UMUL	001010	Unsigned Integer Multiply
SMUL	001011	Signed Integer Multiply
UMULcc	011010	Unsigned Integer Multiply and modify icc
SMULcc	011011	Signed Integer Multiply and modify icc

smul %g2, 4, %g2
umul %o4,%g4,%g2

Result=Reg * sign-extended 13-bit immediate
Or = reg1 * reg2
Low part of product goes to rd
High part of product goes to Y register

Divide Instructions

<i>opcode</i>	<i>op3</i>	<i>operation</i>
UDIV	001110	Unsigned Integer Divide
SDIV	001111	Signed Integer Divide
UDIVcc	011110	Unsigned Integer Divide and modify icc
SDIVcc	011111	Signed Integer Divide and modify icc

udiv %g2, 4, %g2
sdiv %o4,%g4,%g2

Y:reg1 ÷ sign-extended 13-bit immediate
Or Y:reg1 ÷ reg2
32-bit quotient goes into rd
Remainder is discarded

wry instruction writes Y register
rdy instruction reads Y register

FPU instructions employ separate float registers

Floating point arithmetic	(fadds, fsubs, fmuls, fmuld, fdivs, fdivd, fsqt)
Conversion between float and integer	(fitos, fitod, fstoi, fdtoi)
Floating point comparison	(fcmps, fcmpd)
Load & store floating point operands in memory	(ldf, lddf, stf, stdf)
Copy between float registers	(fmovs, fnegs, fabss)

PC and NPC are used in executing and fetching instructions

1. The instruction pointed to by the PC executes while the next is fetched using NPC
2. Contents of NPC get copied into PC and NPC is updated
 - NPC is incremented by 4
 - For taken branches, NPC is overwritten with branch target address

Branches use PC-relative addressing

Target address = $PC + 4 * \text{sign-extended(disp22)}$

NPC = target address

Control is transferred by copying NPC into PC

Branch Instructions

<i>opcode</i>	<i>cond</i>	<i>operation</i>	<i>icc test</i>
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	not Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	not (Z or (N xor V))
BLE	0010	Branch on Less or Equal	Z or (N xor V)
BGE	1011	Branch on Greater or Equal	not (N xor V)
BL	0011	Branch on Less	N xor V
BGU	1100	Branch on Greater Unsigned	not (C or Z)
BLEU	0100	Branch on Less or Equal Unsigned	(C or Z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	not C
BCS	0101	Branch on Carry Set (Less than, Unsigned)	C
BPOS	1110	Branch on Positive	not N
BNEG	0110	Branch on Negative	N
BVC	1111	Branch on Overflow Clear	not V
BVS	0111	Branch on Overflow Set	V

Conditional branching is based on condition codes

Delayed branching is used

instruction in delay slot always executes unless it is annulled

Annulment (controlled by bit29, the annul bit, in machine instruction)

Branches have a single delay slot (contains the delay instruction)

For conditional branches that are taken, the delay instruction is always executed (independent of the annul bit).

For conditional branches that are NOT taken, a=1 annuls delay instruction
(i.e., the instruction in delay slot is not executed)

E.g.: bne,a %g2,done

Unconditional branches are always taken, a=1 annuls the delay instruction
(if the a bit = 0, the delay instruction is executed for unconditional branches)

E.g.: ba,a exit

Two instructions support calling subroutines: call and jmpl

call func1 overwrites pc with the address corresponding to func1
address of call instruction is written into %o7 (link register)
target address = pc + 30-bit signed-displacement*4

return address = %o7 + 8 to get past the instruction = the delay slot

call func1 has same effect as jmpl %o2, %o7 if %o2 contains address of func1
using register as function pointer allows for dynamic addresses (e.g., jump table)

jmpl %o7+8, %g0 same as ret (return) synthetic instruction (%g0 is read-only)

Caution: if function executes a save instruction, a new register window appears
(%o7 becomes %i7)

Window Management

Subroutines and functions can use the save instruction to slide the register window
new registers become visible (no need to save registers on stack)
may also allocate space on stack (stack frame)

Example: save %sp, -512, %sp gets new register window and allocates 512 bytes

return address = %i7 + 8 to get past the instruction = the delay slot

The restore instruction slides window back to previous registers (restores %sp)

Example: ret same as jmpl %i7+8, %g0
 restore executes in delay slot of the ret instruction

- There are 3 machine instruction formats

Format 1

01	disp30			
----	--------	--	--	--

Call

Format 2a

00	a	cond	op2	disp22	
----	---	------	-----	--------	--

Branches

Format 2b

00	rd	op2	imm22		
----	----	-----	-------	--	--

sethi

The bit a =1 to annul the instruction in the branch delay slot

Format 3a

op	rd	op3	rs1	opf		rs2
----	----	-----	-----	-----	--	-----

Floating Point

Format 3b

op	rd	op3	rs1	0	asi	rs2
----	----	-----	-----	---	-----	-----

Data movement

Format 3c

op	rd	op3	rs1	1	simm13	
----	----	-----	-----	---	--------	--

ALU

10
11



Register or immediate type

The leftmost 2 bits indicate to which of the 3 groups the instruction belongs.

Format 1

01

disp30

Call

The CALL instruction contains a 30-bit PC-relative displacement to the target location.

$npc = (\text{instruction}\langle 29:0 \rangle \ll 2) + pc$ generates address of function to be called

Using the pc-relative displacement instead of the direct address makes the CALL instruction position-independent.

Format 2a



Branches

Branch instructions contains a 22-bit PC-relative displacement to the target location.

Limits branch range to $\pm 2^{21}$ instructions from program counter

Op2 = 010 indicates a conditional branch

4-bit cond field indicates condition that causes branch to be taken

Annul bit (a) prevents execution of instruction in delay slot when branch is not taken

Format 2b



sethi

The sethi instruction has op2 = 100 & contains a 22-bit immediate value

Value is loaded into upper 22 bits of rd register and low 10 bits are cleared to 0

Can be combined with “or” instruction to fill rd with a 32-bit constant

facilitated by %hi(x) and %lo(x) assembler operators

sethi %hi(x), %o1 ; puts upper 22 bits of address x into %o1 reg.

or %o1, %lo(x), %o1 ; merges in low 10 bits of address x

Arithmetic/logic instructions use one of two formats.

Format 3b

10	rd	op3	rs1	0		rs2
10	rd	op3	rs1	1	simm13	

Format 3c

ALU

Rd is destination result register

Op3 indicates the operation

1st source register is rs1

If bit13 = 0 the rs2 is the 2nd source register

If bit13 = 1 the second source operand is a 13-bit signed immediate value

Load & store instructions access memory

Format 3b

11	rd	op3	rs1	0		rs2
----	----	-----	-----	---	--	-----

Data movement

Format 3c

10	rd	op3	rs1	1	simm13
----	----	-----	-----	---	--------

Op3 indicates type of data movement instruction (6 bits)

Loads read from memory into the rd register

Stores write contents of rd register into memory

Memory address accessed = $[rs1] + [rs2]$ if bit13=0

= $[rs1] + \text{simm13}$ if bit13 = 1 ($-4096 \leq \text{offset} \leq 4095$)

if rs2 = %g0, this is effectively register indirect addressing for the memory operand

Floating Point machine instruction format

Format 3a

10	rd	op3	rs1	opf	rs2
----	----	-----	-----	-----	-----

Op3 = 110100 indicates Floating Point instruction

Opf indicates floating point operation

FP result register is rd, source registers are rs1 and rs2

Sparc V8 Traps

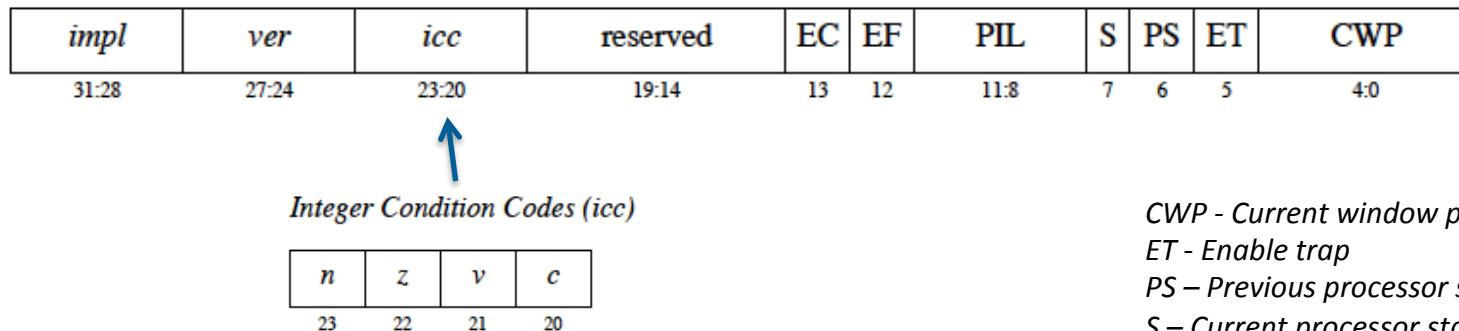
Traps are the mechanism for responding to events during program execution. Such events include:

- Attempts to execute privileged or unimplemented instructions
- Exceptions such as divide by zero or overflow
- Attempted access to restricted or reserved memory areas
- Operating system service calls
- External interrupts

To understand trap handling, the processor state register must first be understood.

Processor State Register

The processor state register is accessed using privileged read/write instructions (rdpsr & wrpsr).



CWP - Current window pointer
ET - Enable trap
PS – Previous processor state
S – Current processor state
PIL – Processor interrupt Level
EF - Enable FPU
EC – Enable Coprocessor

Sparc V8 Traps

A trap is a vectored transfer of control to supervisor code through a 256-entry trap table.

The trap base address (TBA) is held in the trap base register (TBR).

TBA	tt	zero
(31:12)	(11:4)	(3:0)

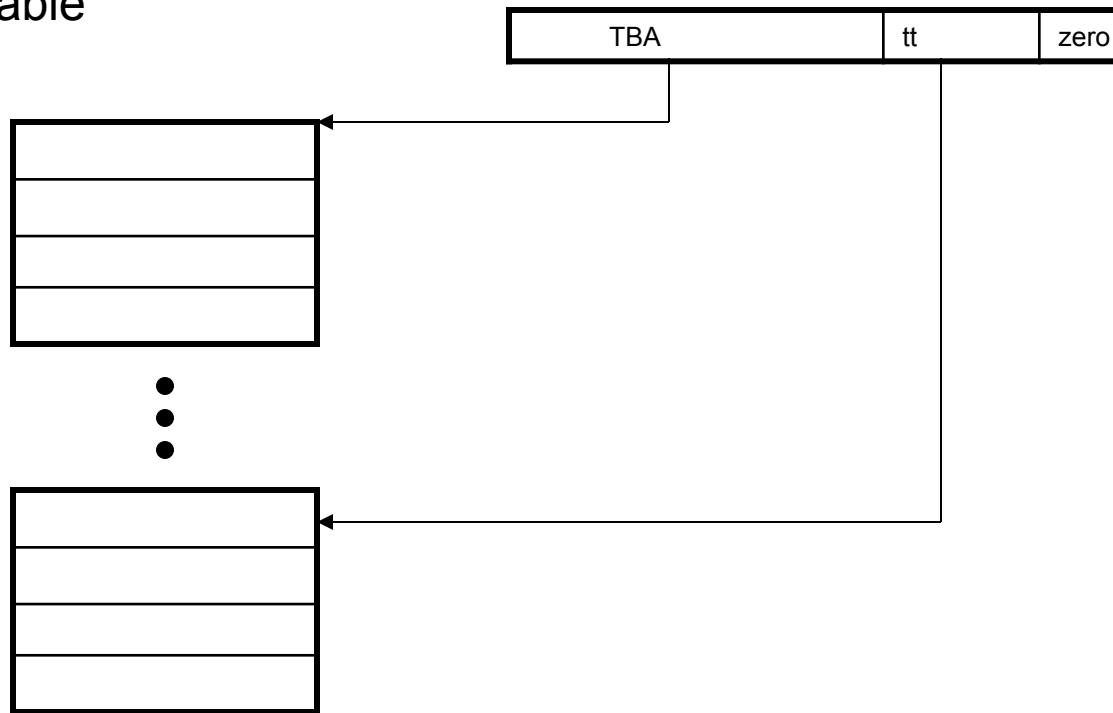
TBA = the most significant 20 bits of the trap base address

(tt) = an 8-bit number set by hardware when a trap occurs to identify the trap type

(zero) = Bits 3 through 0 are zeroes since each trap table entry contains 16 bytes (the first 4 instructions of the handler)

The TBA field within the TBR can be written by the privileged instruction WRTBR (write TBR).

Trap Vector Table



Each entry is 4 words in size and contains the first four instructions of the corresponding trap handler.

Recall that the MIPS employs a single exception vector address through which all exceptions are funneled and examines the cause register to determine the action to take.

Traps are like unsolicited or unexpected procedure calls

New register window is obtained when a trap occurs

pc, npc and psr are saved

copied into local registers within the new window

For traps other than external reset, hardware generates a trap ID

trap ID is written into the tt field within the TBR

ET bit is cleared within PSR (preventing further traps)

Control is transferred into the trap table whose address is in the TBR

Trap Table Entries

The tt field (in TBR) can specify 256 distinct types of trap

- half for hardware traps & half for software traps

ticc instruction generates software trap (for system service, etc.)

operand is `software_trap#`. Example: `ta 12 trap always`

<i>opcode</i>	<i>cond</i>	<i>operation</i>	<i>icc test</i>
TA	1000	Trap Always	1
TN	0000	Trap Never	0
TNE	1001	Trap on Not Equal	not Z
TE	0001	Trap on Equal	Z
TG	1010	Trap on Greater	not (Z or (N xor V))
TLE	0010	Trap on Less or Equal	Z or (N xor V)
TGE	1011	Trap on Greater or Equal	not (N xor V)
TL	0011	Trap on Less	N xor V
TGU	1100	Trap on Greater Unsigned	not (C or Z)
TLEU	0100	Trap on Less or Equal Unsigned	(C or Z)
TCC	1101	Trap on Carry Clear (Greater than or Equal, Unsigned)	not C
TCS	0101	Trap on Carry Set (Less Than, Unsigned)	C
TPOS	1110	Trap on Positive	not N
TNEG	0110	Trap on Negative	N
TVC	1111	Trap on Overflow Clear	not V
TVS	0111	Trap on Overflow Set	V

Trap Table Entries

Since the low 4 bits of the TBR are zero, table entries are on 16-byte boundaries

Each 4-word (16-byte) entry contains the first 4 of the trap handler's instructions

These 4 instructions can be used to call the corresponding handler

A reset will be performed if a trap occurs while the ET bit is 0.

A reset trap causes a transfer of control to address 0. (boot code)

Returning from Traps (rett instruction)

The code sequence below returns to the instruction that caused the trap:

jmpl %l1, %g0 Load PC with saved PC from local register 1

rett %l2 Load NPC with the saved NPC

Alternative return to instruction following the one causing the trap:

jmpl %l2, %g0 Load PC with saved NPC

rett %l2 + 4 Load NPC with saved NPC+4

Memory mapped I/O is used and is based on interrupts or on polling