

Computer Architecture

November 10, 2024

- 1 Week1/1 intro-to-comp-arch.pdf
- 2 Week1/2 parts-of-computer.pdf
- 3 Week1/3 ic-technology.pdf
- 4 Week1/4 instruction-set-design.pdf
- 5 Week2/1 instruction-encoding.pdf
- 6 Week2/2 function-call-support.pdf
- 7 Week2/3 hll-code-to-process.pdf
- 8 Week3/1 arithmetic-in-mips.pdf

- 9 Week3/2 comp-perf-quant.pdf
- 10 Week3/3 hw-impl-prelims.pdf
- 11 Week3/4 single-cycle-impl.pdf
- 12 Week4/1 single-cycle-impl B9luzhA.pdf
- 13 Week4/2 single-cycle-extn.pdf
- 14 Week4/3 single-cycle-analysis.pdf
- 15 Week5/1 intro-to-pipelining.pdf
- 16 Week5/2 structural-hazards-pipelined-datapath.pdf

- 17 Week5/3 data-hazards.pdf
- 18 Week6/1 control-hazards.pdf
- 19 Week7/1 pipeline-control.pdf
- 20 Week7/2 pipeline-exceptions.pdf
- 21 Week8/1 one-system-to-know-them-all.pdf
- 22 Week8/2 the-memory-system-a-hierarchy-of-caches.pdf
- 23 Week8/3 cache-design-a-beginning.pdf
- 24 Week8/4 associative-caches.pdf

- 25 Week9/1 cache-performance-analysis.pdf
- 26 Week9/2 prog-perf-with-cache.pdf
- 27 Week10/1 virtual-memory.pdf
- 28 Week11/1 virtual-memory-and-caches.pdf
- 29 Week11/2 hw-os-for-vm.pdf
- 30 Week11/3 input-output-intro.pdf
- 31 Week11/4 intro-to-buses.pdf

Week1/1 intro-to-comp-arch.pdf

CS305

Computer Architecture

What is Computer Architecture?

Why Study Computer Architecture?

Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Computer Architecture

- “Architecture”
 - The art and science of designing and constructing buildings
 - A style and method of design and construction
 - Design, the way components fit together
- Computer Architecture
 - The overall design or structure of a computer system, including the hardware and the software required to run it, especially the internal structure of the microprocessor

Pre-Requisites

- Data Structures and Algorithms (CS213)
 - Arrays, pointers, stack, queue
- Logic Design (CS210)
 - Switching theory
 - Number systems, computer arithmetic
 - Logic circuits, combinatorial logic, K-maps
 - Finite state machines in hardware
 - Arithmetic unit, control unit design
 - CAD, FPGA, VHDL

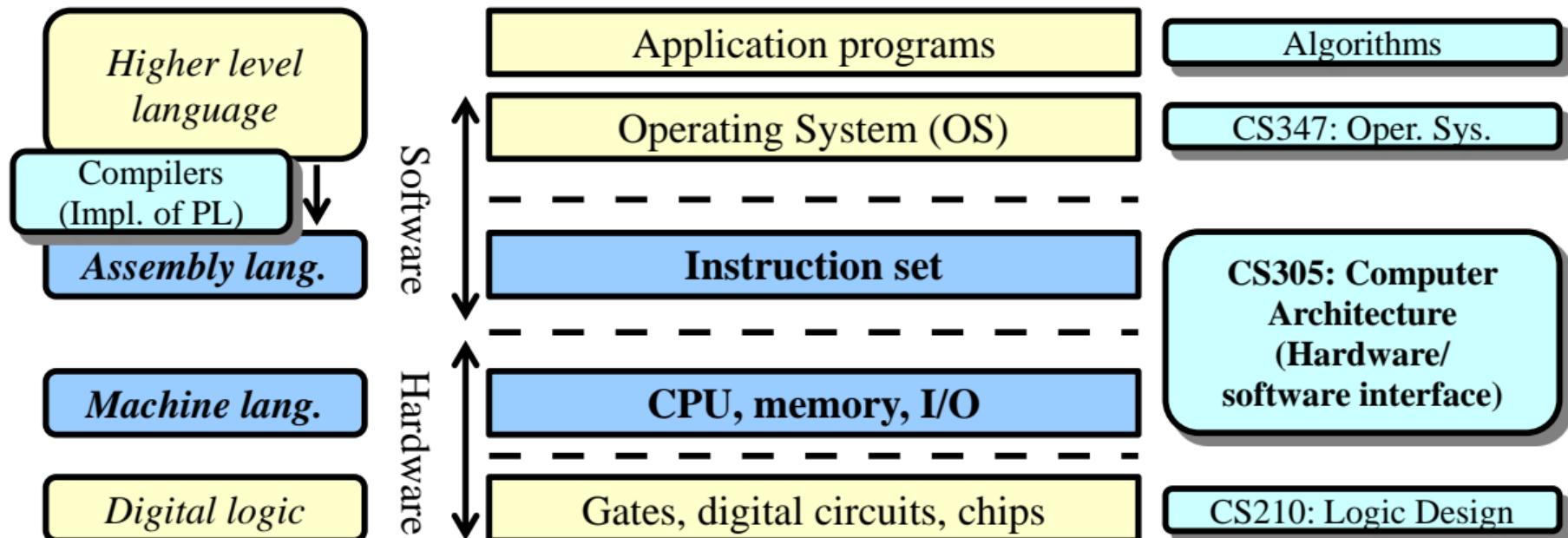
Course Contents

- Computer organization, von Neumann arch.
- Instruction set design
- Measuring performance, Amdahl's law, CPI
- Datapath and control path
- Pipelining, hazards

Course Contents (continued)

- Memory hierarchy, cache design, cache performance
- Disk storage
- RAID
- Error correction codes, Hamming codes
- I/O Buses

Relation to Other Topics/Courses



Text Book References

4th edn: ARM

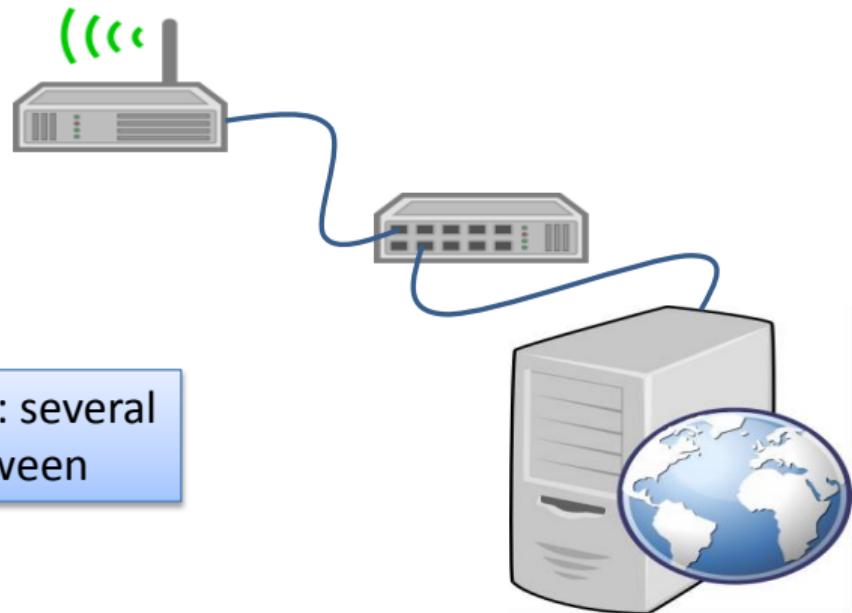
- “Computer Organization and Design: The MIPS Hardware/Software Interface”, 3rd edition, David A. Patterson and John L. Hennessy, Elsevier (Restricted South Asia Edition).
 - 5th edition available, ok to follow, I'll follow 3rd edn. closely
- “Computer Architecture and Organization”, John P. Hayes, 3rd edition, McGraw Hill.
- Low-price editions, e-books available on amazon/flipkart, buy them, no piracy please!
- Notes from other computer architecture courses

Why Study Computer Architecture?

Q: Why do you think Computer Architecture is important (or unimportant)?

Identify Computer Architecture around you

Example-1: This Video



Watching this video on a computer: several computing devices involved in-between

Example-2: Cell-Phones to PCs



A variety of personal devices: the continuum between cell-phones and PCs

Example-3: Servers, Data Centers, Cloud Computing

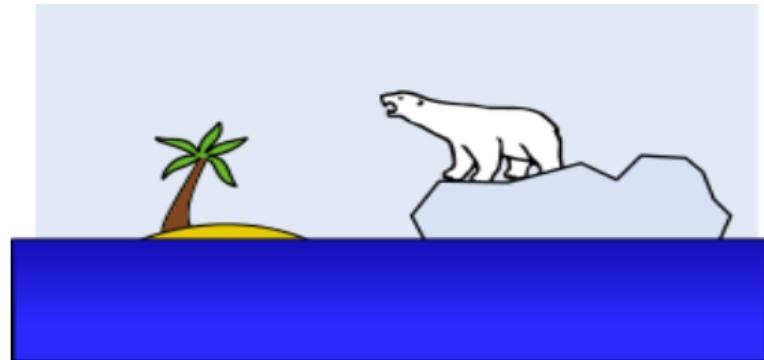
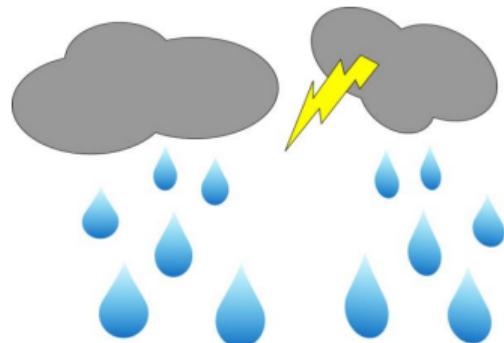


Data storage and computing in the cloud: backbone of major Internet services

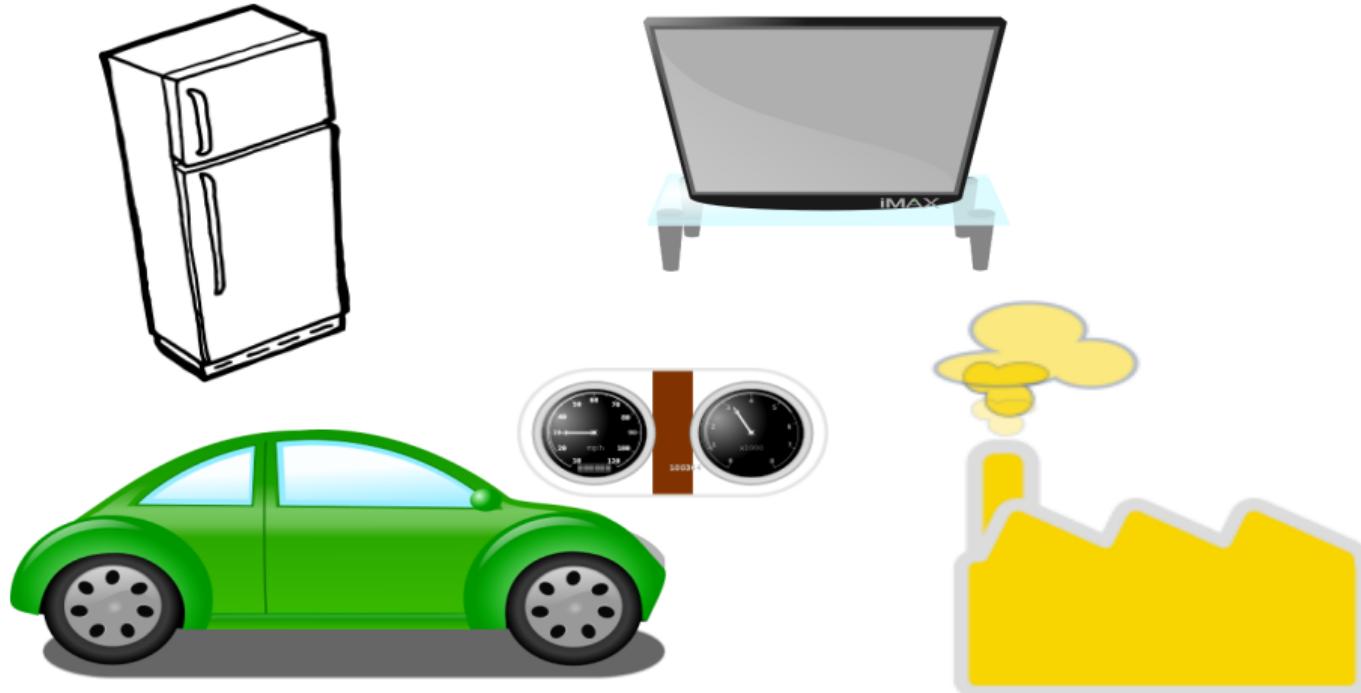
Example-4: Supercomputers



Specialized but important applications, high-end research



Example-5: Embedded Computers

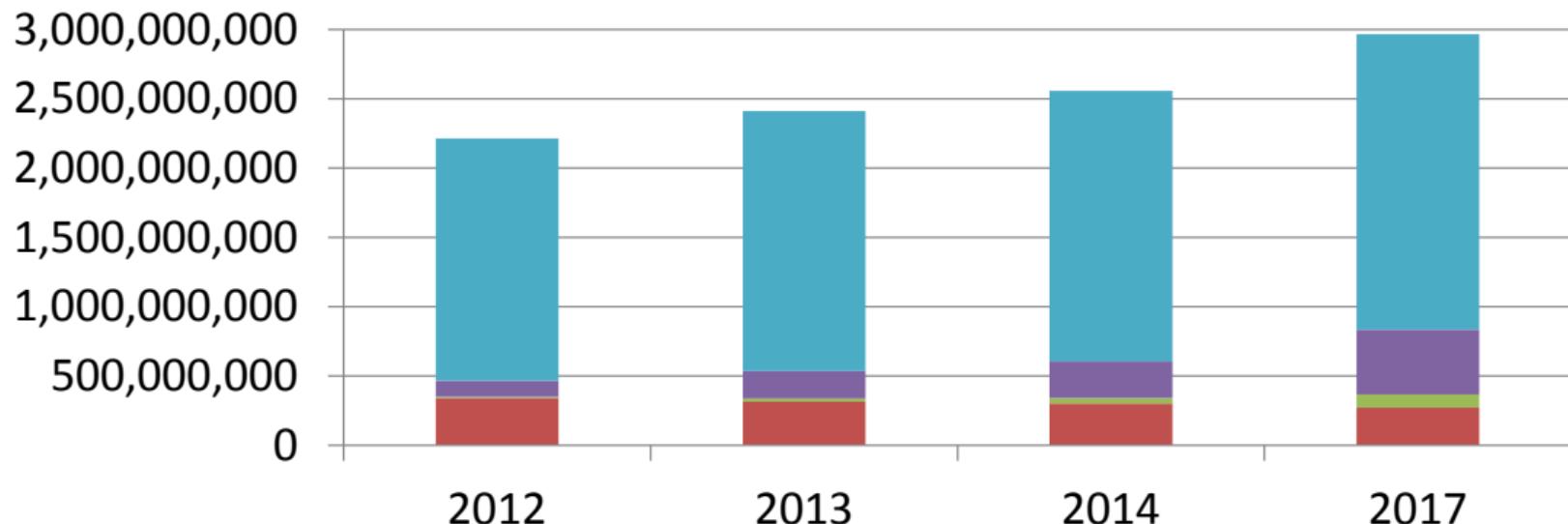


Small but large in number, very critical roles
Home appliances, vehicles, industry automation

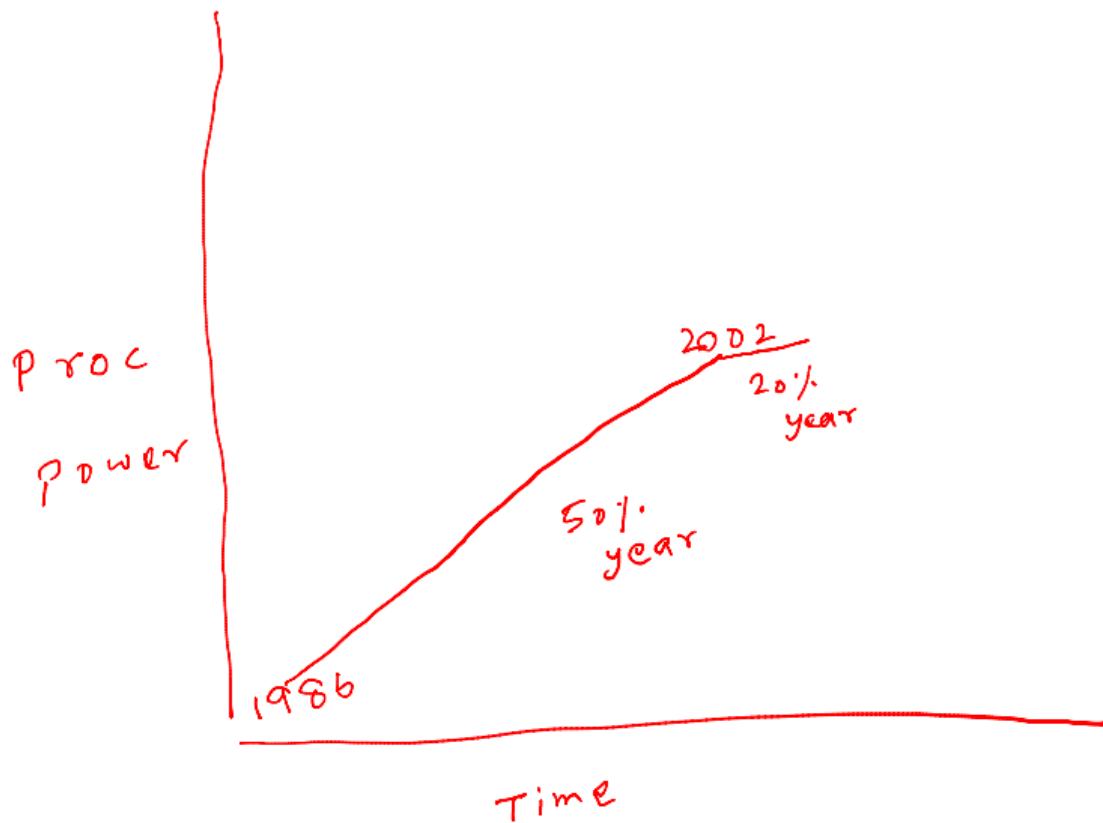
Personal Computing Devices in Numbers

Source: Gartner study, Apr 2013

- PCs (Desktops+Notebooks) ■ Ultramobiles
- Tablets ■ Mobile Phones



Growth in Processing Power



Moore's Law



Summary: Why Study Computer Architecture?

- Computing central to *information age*
- Computer systems range from very small to very large, low-end to super-computers
- New computing devices, end-user devices
 - How are they designed?
 - What affects their performance?
 - What are the performance optimization metrics?
 - How to optimize these metrics?

Week1/2 parts-of-computer.pdf

CS305

Computer Architecture

Parts of a Computer

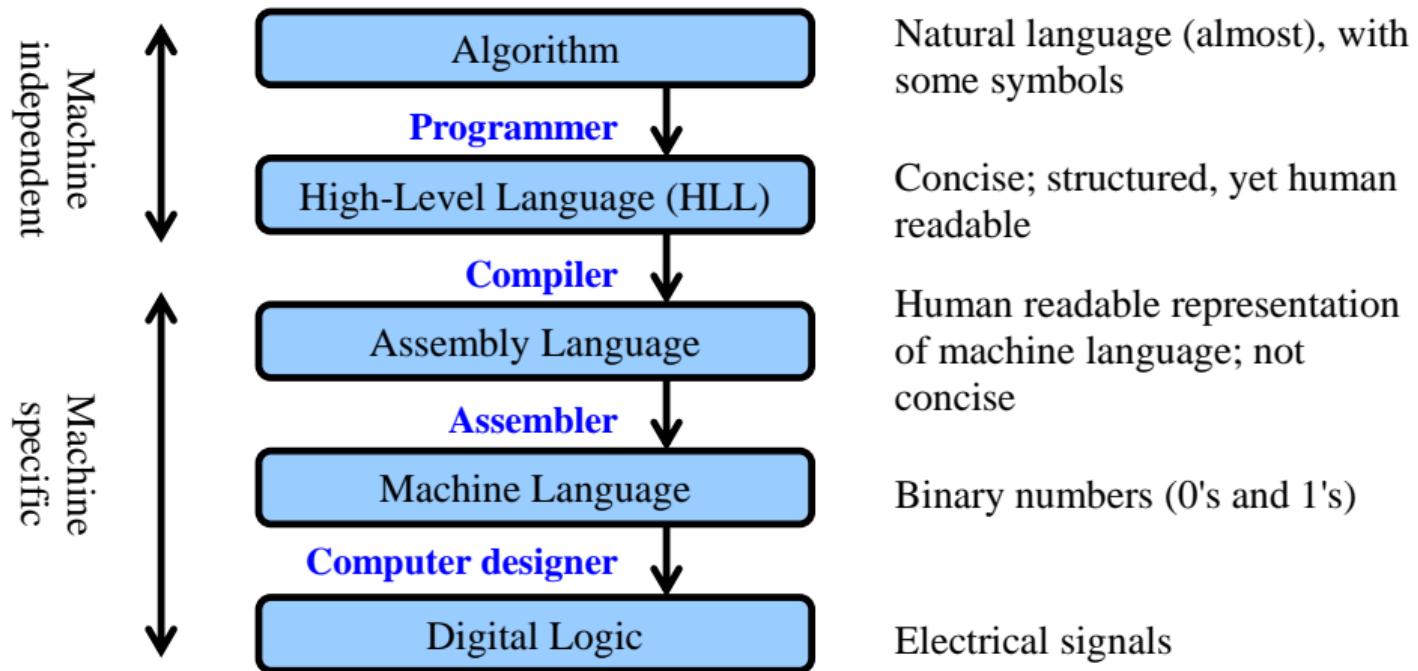
Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

A Hierarchy of Languages



An Example

Algorithm:
Compute the
sum of...

C code:
 $a = b + c;$
 $d = e + f;$

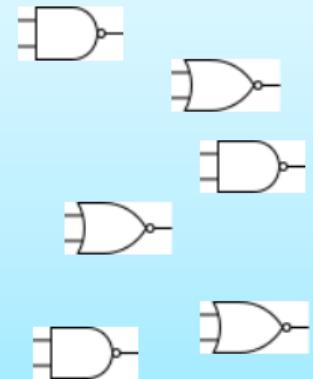
Assembly code:

```
lw    $s1, 4($s0)
lw    $s2, 8($s0)
add  $s3, $s1, $s2
sw    ($s0), $s3
lw    $s3, 16($s0)
lw    $s4, 20($s0)
add  $s5, $s3, $s4
sw    12($s0), $s5
```

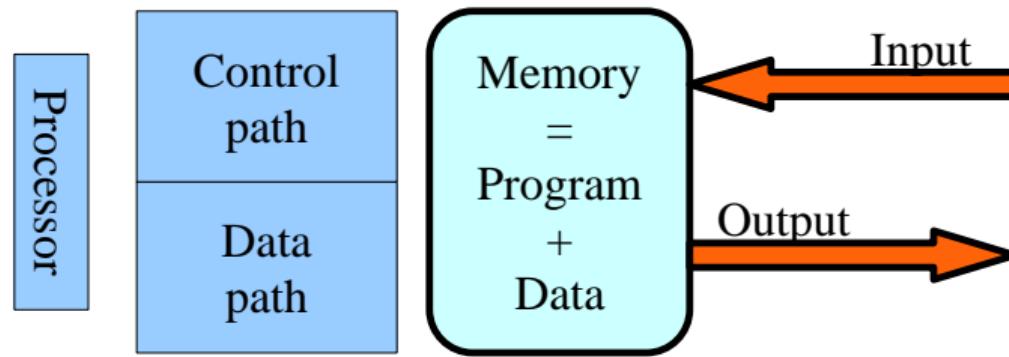
Machine code:

```
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
```

Digital logic:



Von Neumann Architecture

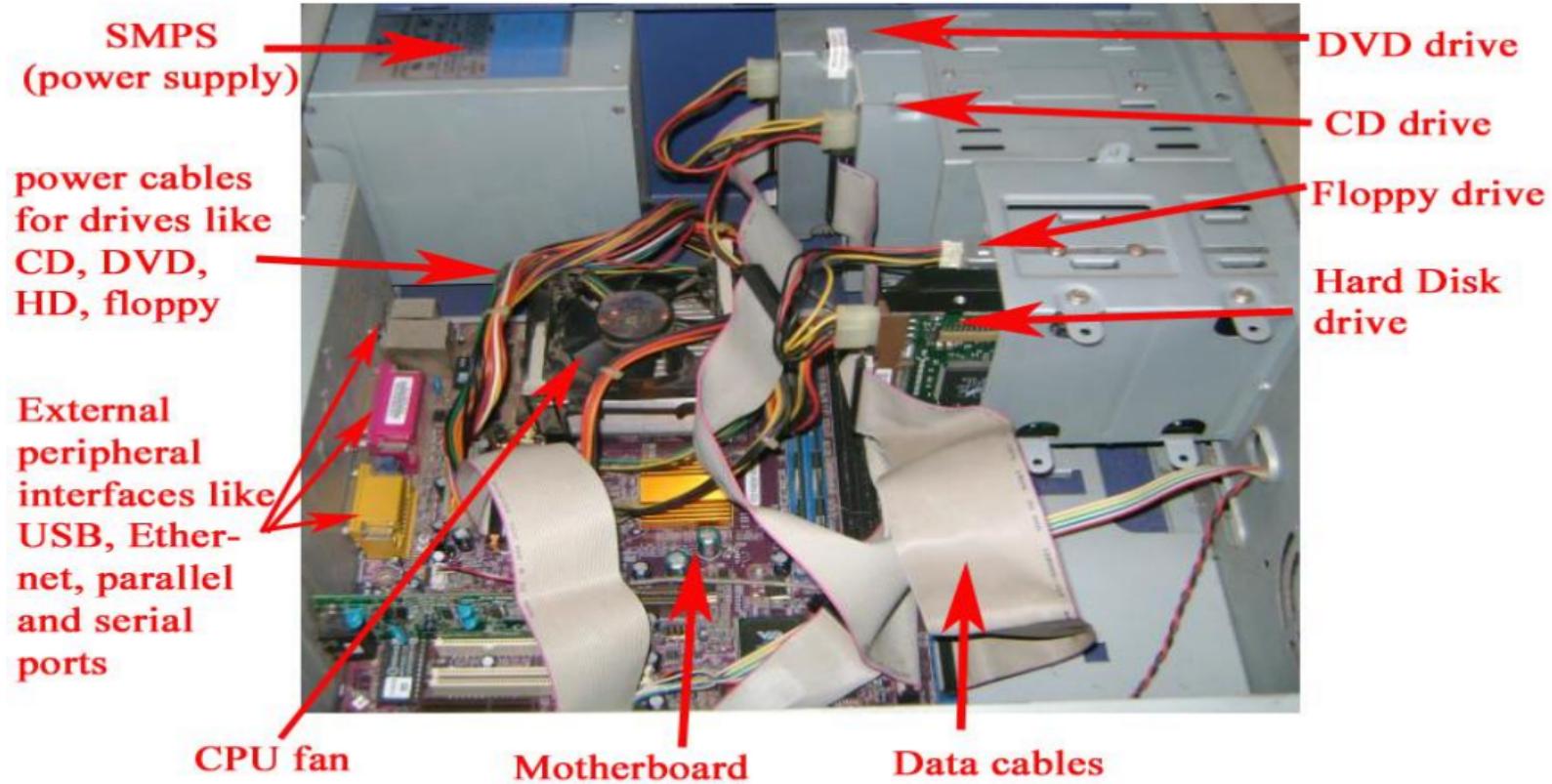


- The stored program concept: program (instructions) as well as data are stored in memory
- Processor *fetches* instructions from memory, and *executes* them on *data* (also fetched from/to memory)
 - Example from previous slide: LW and SW instructions

The Five Components

- All computers have these five components: input, output, memory, [data path + control path = processor]
- Underlined aspects: topics in this course
- Input: keyboard, mouse; also disk, network
- Output: monitor; also disk, network
- Memory: different kinds of memory
- Data path + control path = processor

Inside a Computer...

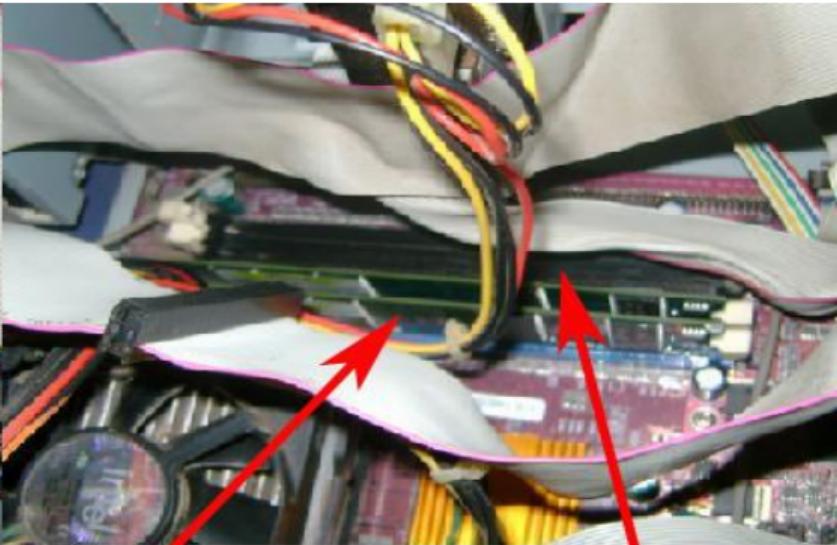


Inside a Computer (continued...)



External modem card

PCI slots for expansion
(add-on cards)



main memory
(RAM) chips

empty slots for RAM
expansion in future

Magnetic Tape



Inside a Computer: Summary

- Integrated circuits, or chips:
 - Flat and black
 - Processor (CPU), main memory, cache memory, etc.
- Motherboard:
 - Houses the various chips
 - Also has many I/O interfaces (PCI, USB, Serial, etc.)
- Secondary memory: non-volatile
 - Magnetic disks, optical (CD/DVD), tape, flash-based (e.g. USB pen-drives, CF cards), floppies (obsolete)

Week1/3 ic-technology.pdf

CS305

Computer Architecture

SS Integrated Circuit (IC) Technology: An Overview

MSI
LSI - 10^4

VLSI

ULSI

Bhaskaran Raman

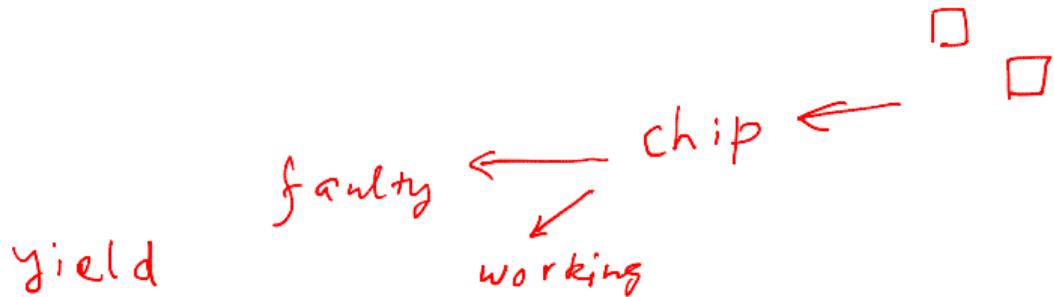
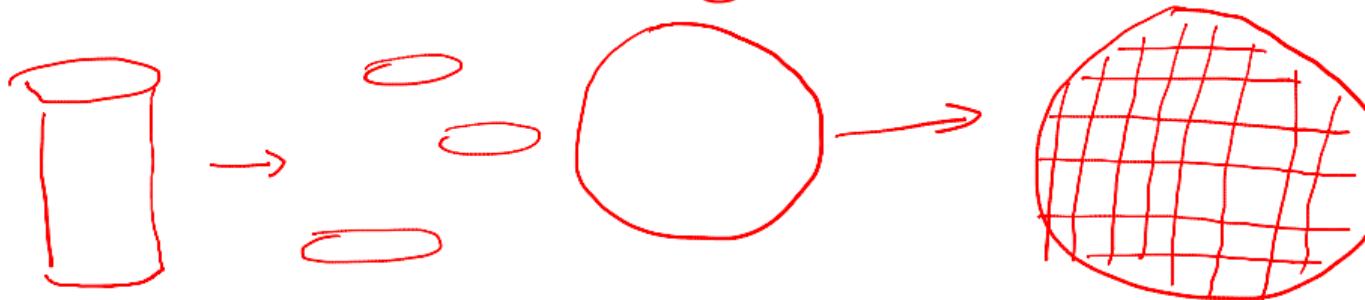
Room 406, KR Building

Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

The IC Manufacturing Process

conductors
insulators
transistors
Silicon



IC Cost

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

Straightforward
algebra

$$\text{Dies per wafer} \approx \frac{\text{Area of wafer}}{\text{Area of die}}$$

Approximation

$$\text{Yield} = \frac{1}{[1 + (\text{Defects per area} \times \text{Area of die}/2)]^2}$$

From experience

- Unit cost of chip decreases with volume *of production*
 - Fixed costs amortised: design, masks in chip manufacture
 - Tuning to improve yield

Limits to IC Density

- Fundamental physical dimension limits
- Power consumption
 - $\sim 2.6 \text{ cm}^2$, 4 GHz Intel Core i7, 88W power
- Fan needed to sink the heat
- Frequency scaling employed

Week1/4 instruction-set-design.pdf

CS305

Computer Architecture

Instruction Set Design

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Instruction Set: What and Why

HLL code examples:

C/C++

f()->next = (g() > h()) ? k() : NULL;

Perl

\$line =~ s/abc/xyz/g;

- Simple for programmers to write
- But, too complex to be implemented *directly* in hardware
- **Solution:** express complex statements as a sequence of simple statements
- **Instruction set:** the set of (relatively) simple instructions using which higher level language statements can be expressed

A Simple Example

C code:

```
a = b + c;  
d = e + f;
```

Compiler

Assembly code:

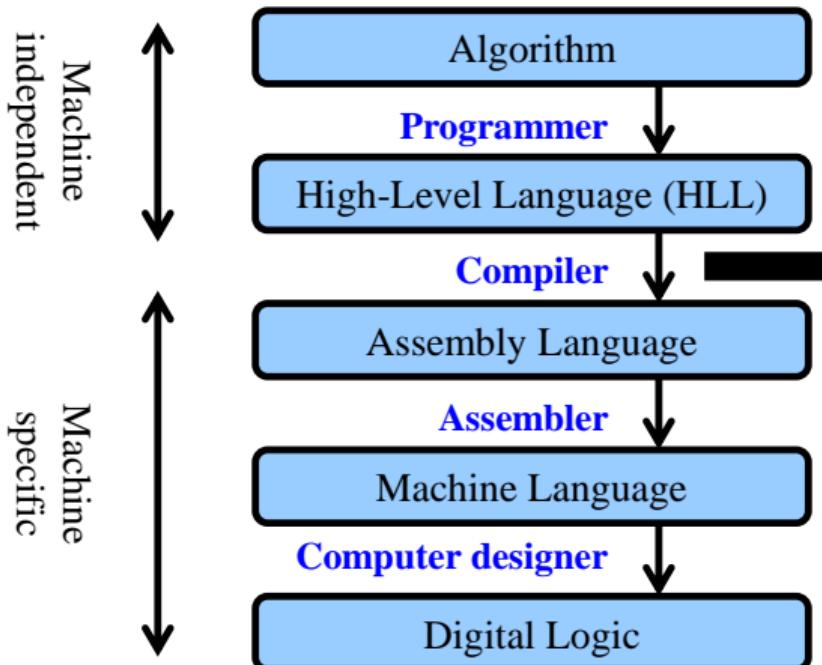
```
lw    $s1, 4($s0)  
lw    $s2, 8($s0)  
add  $s3, $s1, $s2  
sw    ($s0), $s3  
lw    $s3, 16($s0)  
lw    $s4, 20($s0)  
add  $s5, $s3, $s4  
sw    12($s0), $s5
```

Assembler:
instruction
encoding
(straight-
forward)

Machine code:

```
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...
```

Instruction Set



Instruction set is the interface between hardware and software

Interface: instruction set

Interface design:

- Central part of any system design
- Interface before implementation!
- Allows abstraction, independence
- Challenge: should be easy to use by the layer above

Instruction Set Defines a Machine

HLL code examples:

C/C++

```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

```
$line =~ s/abc/xyz/g;
```

MIPS's
instruction
set

x86's
instruction
set

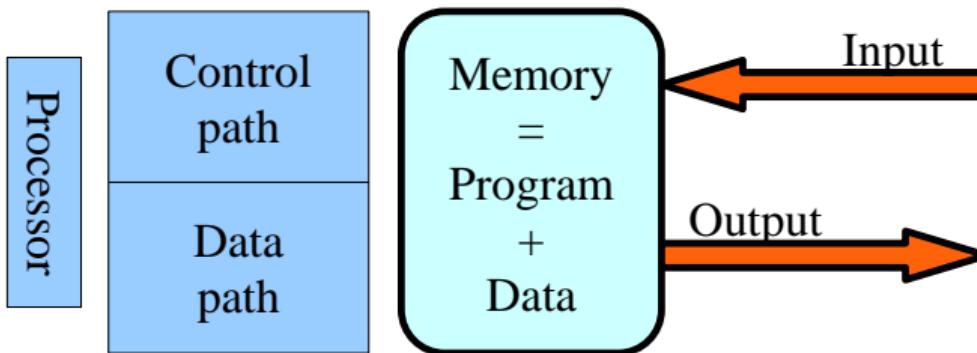
ARM's
instruction
set

Assembly code
(machine code)
for MIPS

Assembly code
(machine
code)
for x86

Assembly code
(machine
code)
for ARM

Instruction Set and the Stored Program Concept

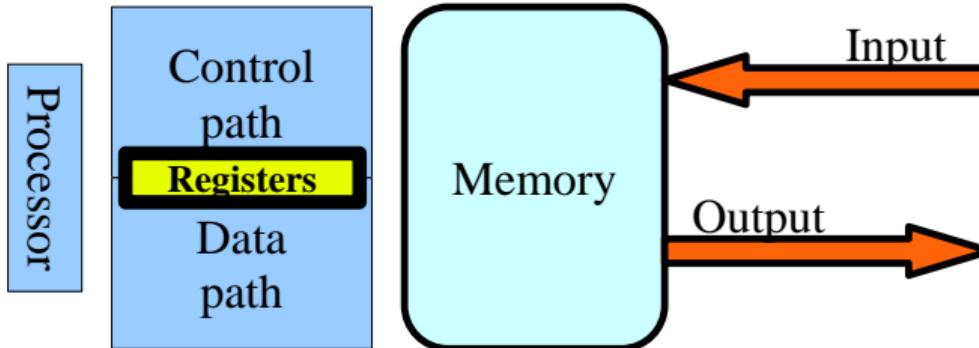


- At the processor, two steps in a loop:
 - Fetch instruction from memory
 - Execute instruction
 - May involve data transfer from/to memory

The Two Aspects of an Instruction

- Instruction: operation, operand
 - Example: $a := b + c$
 - Operation is addition
 - Operands are b , c , a
 - For our discussion: “result” is also considered an operand
- What should be the instruction set? ==
 - What set of operations to support?
 - What set of operands to support?
- We'll learn these in context of: the MIPS instruction set

Registers: Very Fast Operands



- Registers: very small memory, inside the processor
 - Small ==> fast to read/write
 - Small also ==> easy to encode instructions (as we'll see)
 - Integral part of the instruction set architecture (i.e. the hardware-software interface) [NOT a cache]
- MIPS has 32 registers, each of 32-bits

Some Terminology

- 32-bits = 4-bytes = 1-word
- 16-bits = 2-bytes = half-word
- 1-word is the (common-case) unit of data in MIPS
 - 32-bit architecture, also called MIPS32
 - 64-bit MIPS architecture also exists: MIPS64
 - 32-bit & 64-bit architectures are common
 - Low end embedded platforms: 8-bit or 16-bit architectures

Your First MIPS Instruction

add <res>, <op1>, <op2>

Example:

add \$s3, \$s1, \$s2

- The **add** instruction has exactly 3 operands
 - Why not support more operands? Variable number?
 - Regularity ==> simplicity in hardware implementation
 - Simplicity ==> fast implementation
- All 3 operands are registers
 - In MIPS: 32 registers numbered 0-31
 - \$s0-\$s7 are assembly language names for register numbers 16-23 respectively (why? answered later)

\$0 - \$31

Constant or Immediate Operands

```
addi <res>, <op1>, <const>
```

Example:

```
addi $s3, $s1, 123
```

- HLL constructs use immediate operands frequently
- Design principle: *make the common case fast*
 - Most instructions have a version with immediate operands
- Question: common case use of constant addition in C?

Memory Operations: Load and Store

lw <dst_reg>, <offset>(<base_reg>)

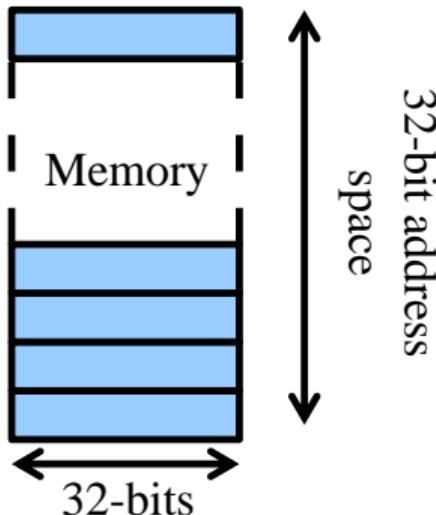
sw <offset>(<base_reg>), <src_reg>

Example:

lw \$s1, 4(\$s0)

sw 12(\$s0), \$s5

0xFFFF FFFC



- Load and store in units of 1-word: terminology w.r.t. CPU
- Also called data transfer instns: memory \leftrightarrow registers
- Address: 32-bit value, specified as **base register + offset**
- Question: why is this useful?
- **Alignment restriction:** address has to be a unit of 4 (why? answered later)

Test Your Understanding...

- Is a **subi** instruction needed? Why or why not?
- Is **sub** instruction needed? Why or why not?

Test Your Understanding (continued)...

- Translate the following C-code into assembly lang.:
 - Ex1: $a[300]=x+a[200];$ // all 32-bit int
 - What more information do you need?
 - Ex2: $a[300]=x+a[i+j];$ // all 32-bit int
 - Can you do it using instructions known to you so far?

```
# a in s0, x in s1
lw    $t0, 800($s0)
add   $t1, $t0, $s1
sw    1200($s0), $t1
```

Registers \$t0-\$t9
usually used for
temporary values

```
# a in s0, x in s1
# i in s2, j in s3
add   $t2, $s2, $s3
muli  $t2, $t2, 4
add   $t3, $t2, $s0
lw    $t0, 0($t3)
add   $t1, $t0, $s1
sw    1200($s0), $t1
```

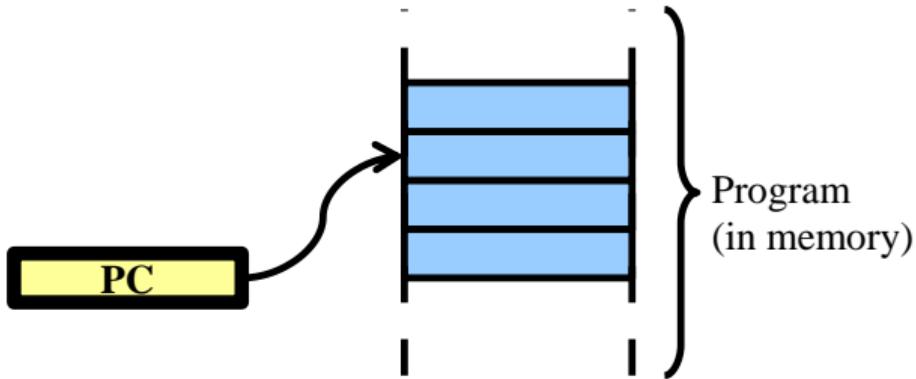
Notion of Register Assignment

- Registers are *statically assigned* by the compiler to registers
- Register management during code generation: one of the important jobs of the compiler
- Example from previous exercise...

Instructions for Bit-Wise Logical Operations

Logical Operators	C/C++/Java Operators	MIPS Instructions
Shift Left	<<	sll
Shift Right	>>	srl
Bit-by-bit AND	&	and, andi
Bit-by-bit OR		or, ori
Bit-by-bit NOT	~	nor

The Notion of the Program Counter



- In MIPS: (only) special instructions for PC manipulation
- PC not part of the register file
- In some other architectures: arithmetic or data transfer instructions can also be used to manipulate the PC

- The program is fetched and executed instruction-by-instruction
- Program Counter (PC)
- A special 32-bit register
- Points to the **current instruction**
- For sequential execution: $\text{PC} += 4$ for each instruction
- Non-sequential execution implemented through manipulation of the PC

Branching Instructions

- Stored program concept: usually *sequential* execution
- Many cases of non-sequential execution:
 - If-then-else, with nesting
 - Loops
 - Procedure/function calls
 - Goto (bad programming normally)
 - Switch: special-case of nested if-then-else
- Instruction set support for these is required...

Conditional and Unconditional Branches

- Two conditional branch instructions:
 - `beq <reg1>, <reg2>, <branch_target>`
 - `bne <reg1>, <reg2>, <branch_target>`
- An unconditional branch, or jump instruction:
 - `j <jump_target>`
- Branch (or jump) target specification:
 - In assembly language: it is a **label**
 - In machine language, it is a **PC-relative offset**
 - Assembler computes this offset from the program

Using Branches for If-Then-Else

```
if(x == 0) { y=x+y; } else { y=x-y; }
```

Convention in my slides:

s0, s1... assigned to variables# in order of appearance

```
# s0 is x, s1 is y
bne    $s0, $zero, ELSE
add    $s1, $s0, $s1
j      EXIT
$0
ELSE:
sub    $s1, $s0, $s1
EXIT:
# Further instructions below
```

Note: use of \$zero register (make the common case fast)

Using Branches for Loops

```
while(a[i] != 0) i++;
```

```
# s0 is a, s1 is i
BEGIN:
    sll    $t0, $s1, 2
    add    $t0, $t0, $s0
    lw     $t1, 0($t0)
    beq    $t1, $zero, EXIT
    addi   $s1, $s1, 1
    j      BEGIN
EXIT:
# Further instructions below
```

Q: is \$t1 really needed above?

Testing Other Branch Conditions

- **slt <dst>, <reg1>, <reg2>**
 - slt == set less than
 - <dst> is set to 1 if <reg1> is less than <reg2>, 0 otherwise
- **slti <dst>, <reg1>, <immediate>**
- Can be followed by a **bne** or **beq** instruction
- How about \leq or $>$ or \geq comparisons?
- Note: register 0 in the register file is always ZERO
 - Denoted **\$zero** in the assembly language
 - Programs use 0 in comparison operations very frequently
- Why not single **blt** or **blti** instruction?

For Loop: An Example

```
for(i = 0; i < 10; i++) { a[i] = 0; }
```

```
# s0 is i, s1 is a
addi    $s0, $zero, 0
BEGIN:
    slti    $t0, $s0, 10
    beq    $t0, $zero, EXIT
    sll    $t1, $s0, 2
    add    $t2, $t1, $s1
    sw     0($t2), $zero
    addi   $s0, $s0, 1
    j      BEGIN
EXIT:
# Further instructions below
```

Q: min # temporary registers needed for above code secn?

Week2/1 instruction-encoding.pdf

CS305

Computer Architecture

Instruction Encoding

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

A Simple Example

C code:

```
a = b + c;  
d = e + f;
```

Compiler

Assembly code:

```
lw    $s1, 4($s0)  
lw    $s2, 8($s0)  
add  $s3, $s1, $s2  
sw    ($s0), $s3  
lw    $s3, 16($s0)  
lw    $s4, 20($s0)  
add  $s5, $s3, $s4  
sw    12($s0), $s5
```

Assembler:
instruction
encoding
(straight-
forward)

Machine code:

```
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...  
...0...1...
```

Instruction Encoding

- **Encoding:** representing instructions as numbers/bits
 - Recall: instructions are also stored in memory!
 - Encoding == (assembly language → machine language)
- MIPS: all instructions are encoded as **32 bits** (why?)
- Also, all instructions have *similar* format (why?)

Regularity ⇒ simplicity ⇒ efficient implementation

MIPS Instruction Format

add
sub

	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
--	---------------	-----------	-----------	-----------	--------------	--------------

R-type instruction: register-register operations

addi

opcode (6)	rs (5)	rt (5)	immediate/constant or offset (16)
---------------	-----------	-----------	--------------------------------------

PC+4

I-type instruction: loads, stores, all immediates, conditional branch, jump register, jump and link register

PC+4
1 2 3 4
26 2
4

opcode (6)	offset relative to PC+4 word (26)
---------------	--------------------------------------

J-type instruction: jump, jump and link, trap and return

Pseudo-direct addressing

Test Your Understanding...

- What format is used by the **slt** instruction?
- What instruction format is used by **beq** ?

Week2/2 function-call-support.pdf

CS305

Computer Architecture

Function Call Support in the Instruction Set

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Recall: MIPS Instructions So Far

Arithmetical: add, addi, sub

Logical: and, or, nor

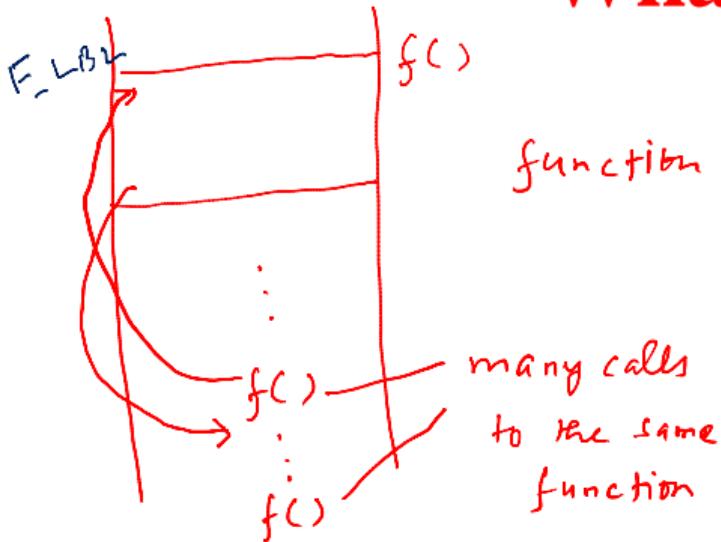
sll, srl $\$0 - \31

memory: $l_{N, SW}$ $\$S0 - \$S7$

branch: beq, bne, j \$t0-\$t9

\$300

Basic Function Call Support: What is Needed?



many calls
to the same
function

function call , return
j is sufficient to remember the return address

jal F_LBL
part of MAPS set of regs
① \$ra = PC+4
② transfer ctrl to F_LBL
PC = F_LBL

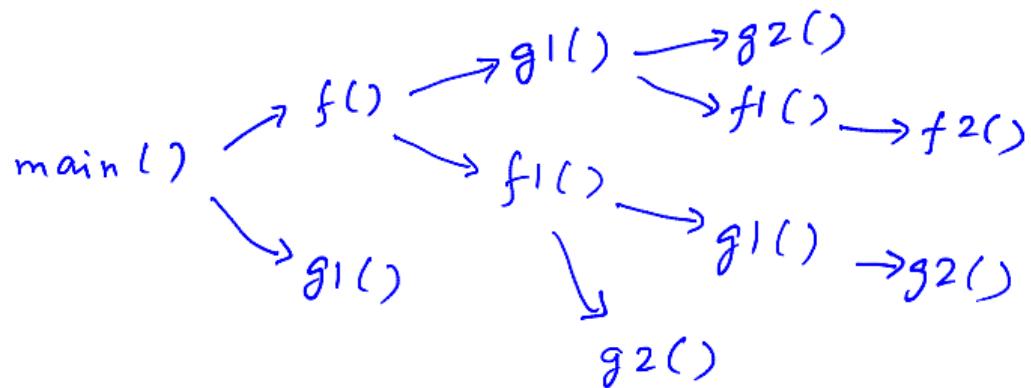
jr \$ra
jump register can be any of the 32 regs

Arguments and Return Values

among {
 \$ a0, \$ a1, \$ a2, \$ a3 - arguments
 \$ v0, \$ v1 - return values
 the
 32
 MIPS
 tregs

What About Nested Function Calls?

`main() → f() → g()`
\$ra gets overwritten



Arbitrary nesting
of function calls
- where to store \$ra?

Need for a Stack Data Structure

- before making a function call, store own return address on to stack
 - restore \$ra from stack after function call

e.g. main → f() ↗
g1()
↗ g2()

$f(\cdot)$:

we gra onto stick

jal g 1

restore \$ra from stack

Sra. ONTO STICK

jal 92

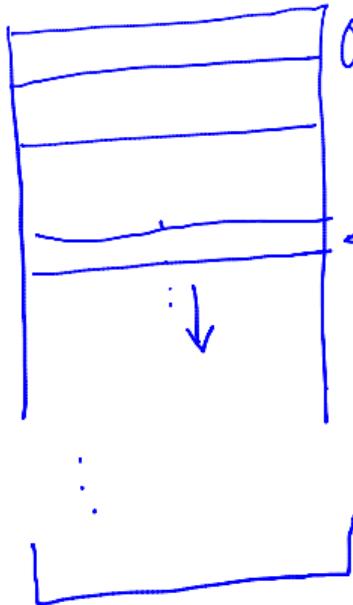
restore \$79 from stack

f():
Save \$ra onto stack

more efficient ↗ jal gl

Restore \$ra from stack

Stack Implementation



\$SP top of stack
part of MIPS
32 regs

why does stack grow down?

Q: write code to push \$ra onto stack

```
addi $SP, $SP, -4  
sw O($SP), $RA
```

Q: write code to pop \$ra from stack

```
lw $RA, O($SP)  
addi $SP, $SP, 4
```

Arguments and Return Values in Nested Calls

main() → f(...) → g(...)

\$a0-\$93, \$v0,\$v1 - caller has to save and restore
if caller cares

What if arg/retval cannot be
accommodated within given regs?

- use stack for these

Caller vs Callee Saving Conventions

main() → f() → g()
caller caller caller caller

caller saved
(if it cares)

f().
: unpreserved
use \$a2
save \$a1:
; al, g
restore \$a2
use \$a2

vs

callee saved
(if it uses)

\$ra

preserved

\$so-\$s7
\$sp

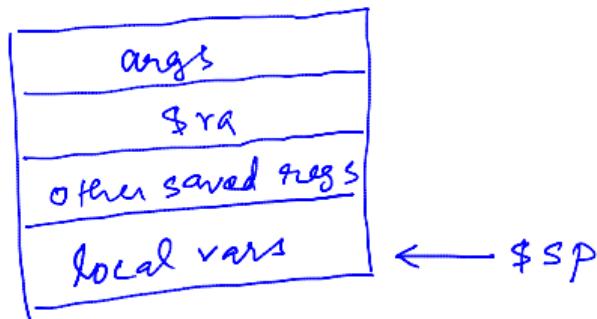
Stack Frame, or Call Activation Record

More arguments than 4×32 bit? Use stack

More return value than 2×32 bit? Use stack

Saving registers? Use stack

Local variables? Use stack



The Frame Pointer



\$fp - makes it easier for compiler

f()

int x;

L1 → x used here

for(int i=0; ...) {

L2 → x used here

}

} // End f()

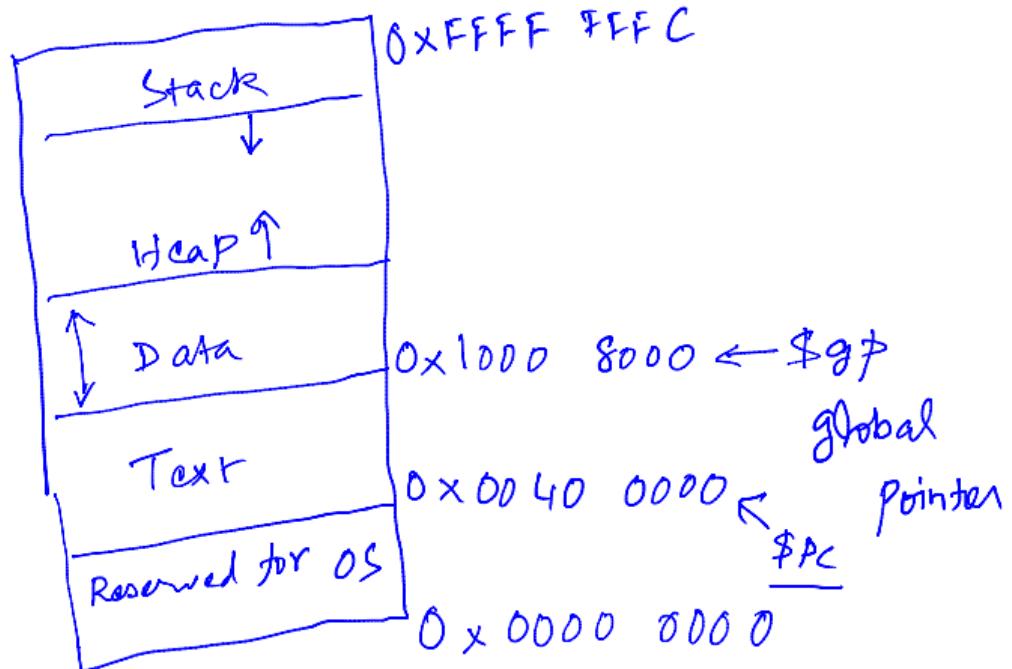
Ref to x at L2 and L1

will be same w.r.t \$fp

will be different w.r.t \$sp

\$fp - preserved or
callee saved

Memory Organization: Stack, Heap, Text



Recursion Example

```
int factorial(int n) {  
    if(n == 0)  
        return 1;  
    else  
        return n*factorial(n-1);  
}
```

FACT:

```
addi $sp, $sp, -8  
sw 4($sp), $ra # $ra in stack  
sw 0($sp), $a0 # $a0 in stack  
bne $a0, $zero, ELSE  
addi $v0, $zero, 1 # base case  
j EXIT
```

ELSE:

```
addi $a0, $a0, -1  
jal FACT # recursive call  
lw $a0, 0($sp) # restore $a0  
mul $v0, $v0, $a0
```

EXIT:

```
lw $ra, 4($sp) # restore $ra  
addi $sp, $sp, 8 # restore $sp  
jr $ra # return to caller
```

A Few More Details: Pseudo-Instructions, Assembler Temporary, Dealing with Bytes/Half-Words

li — addi
ori

bez beq \$zero
bnz bne

lb, lh lbu, lhu
sb, sh sbu?, shu?

beq \$s0, 10, EXIT
addi \$at, \$zero, 10
beq \$s0, \$at, EXIT



\$s0, 10, EXIT

assembler temporary

\$at, \$zero, 10

\$s0, \$at, EXIT

j FAR-AWAY

lui \$at, something
ori \$at, \$at, something
jr \$at

Summary So Far

- MIPS instruction set design
- Instruction encoding
- Instructions for function call support

Week2/3 hll-code-to-process.pdf

CS305

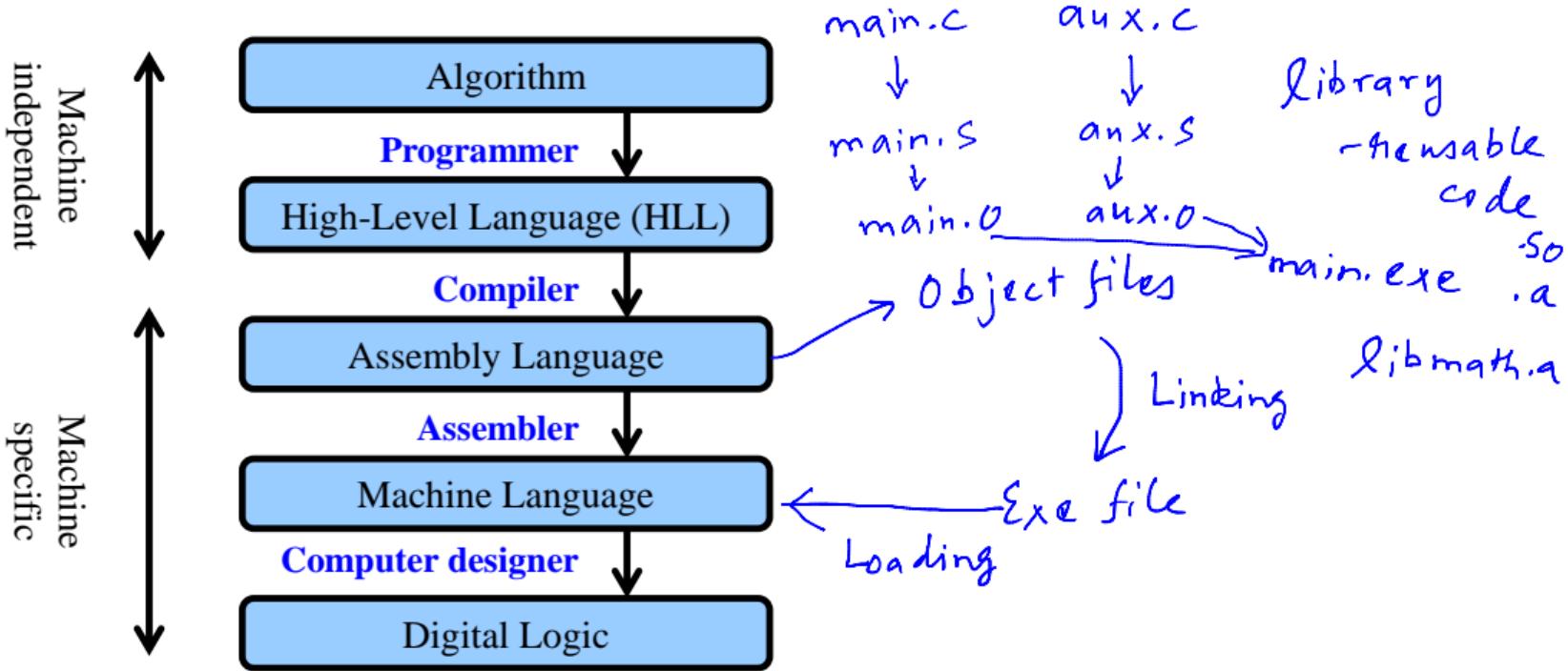
Computer Architecture

**From HLL Code to Process:
Object Files, Linking, Loading**

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Steps from HLL Code to Running Process



Contents of an Object File

1. Header
2. Text segment ✓
3. Static data ✓
4. Relocation table: list of unresolved instructions ✓
5. Symbol table: table of unresolved symbols ✓
6. Debugging information

Object Files: An Example

main.s

```
.data Y  
main:  
...  
jal F          # I1  
...  
lw $s0, X      # I2  
...  
G:  
...  
lw $s1, Y      # I3
```

aux.s

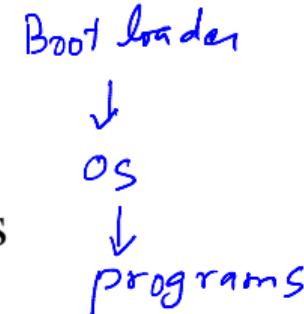
```
.data X  
F:  
...  
jal G          # I4  
...  
lw $s2, X      # I5  
...  
...  
sw Y, $s3      # I6
```

Relocation table: I1, I2, I3
Symbol table: main, F, G, X, Y

Relocation table: I4, I5, I6
Symbol table: F, G, X, Y

Functionalities of Linker, Loader

- Linker:
- generate exe file
 - Organize text and data as it would appear in memory
 - Resolve symbols
 - Rewrite instructions referred to in relocation table
- Loader:
 - Place text and data in memory
 - Init stack, other regs, including args
 - Call main



Dynamic Linking

- Link files on demand
- Why?
 - Smaller exe files, better use of memory
 - Newer libraries can be used seamlessly
- How?
 - Rewrite indirect jump address
 - Use jump table of function pointers
 - **Note:** `jalr` instruction needed to support function pointers

Summary

- HLL code → Compiler → Assembler → Linker → Loader → Executing Process
- Object file contents
- Dynamic linking

Week3/1 arithmetic-in-mips.pdf

CS305

Computer Architecture

Arithmetic in MIPS

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Recall Integer Representation

- Possibilities
 - Sign-magnitude
 - 1's complement
 - 2's complement
- Which representation does MIPS use? Why?

Implications of Number Representation for ISA

- Unsigned versions of **lb**, **lh**: **lbu**, **lhu**, (**lwu**?)
- Many instructions have unsigned versions
 - **add** → **addu**, **addi** → **addiu**, **sub** → **subu**
 - **slt** → **sltu**, **slti** → **sltiu**
- Questions:
 - Is result if **add** and **addu** always the same?
 - So why separate **addu** instruction ?
 - Using single **sltu** or **sltiu** to check array bounds

Integer Multiplication and Division in MIPS

- Use of special registers **hi**, **lo**
 - **mult**, **multu** store 64-bit result in **hi**, **lo**
 - **div**, **divu** store quotient in **lo**, remainder in **hi**
- How to get result from **hi**, **lo** ?
 - Special instructions: **mfhi <reg>**, **mflo <reg>**
 - Why not instructions for moving values to **hi**, **lo** ?

Recall Floating Point Representation

- Normalized scientific notation
 - Sign, significand, exponent (biased 2's complement)
 - Single precision: $32 = 1 + 8 + 23$
 - Double precision: $64 = 1 + 11 + 52$
 - Subnormal numbers, NaN
 - IEEE 754 standard
 - Supported in MIPS, same as in C float/double

MIPS Floating Point Instructions

- Arithmetic: +, -, **x**, / add.s, add.d
 sub.s/d mul.s/d div.s/d
- Comparisons: **eq**, **ne**, **lt**, **le**, **gt**, **ge**
 - Set a special bit c.eq.s c.le.d
 - To be used in next instruction: bc1t, bc1f co-processor
- Memory operations: **lwcl**, ldcl, **swcl**, sdc1
- MIPS has separate 32 x 32-bit FP registers
 - Why don't we need additional bit in instruction encoding?
 - Can be considered as 16 x 64-bit FP registers for double

Common Programming Bugs

- Signed versus unsigned
- Not handling overflow
- Assuming FP associativity
- Assuming FP precision

$$a \times (b \times c) \neq (a \times b) \times c$$

$x = y$

Week3/2 comp-perf-quant.pdf

CS305

Computer Architecture

Computer Performance Quantification

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Aspects of Computer Performance

Who cares?

Customer

Vendor

Computer designer (us)

Metrics

time to deliver

reliability $\sim \%$ past failed attempts

$\sim \%$ past delays

cost, per unit weight

max weight

delivery confirmation

online tracking

destinations on offer
pickup service

Computer Performance Equation

customer = one executing program

Sysad: throughput

metric - execution time ✓

multi-user environment

response time Linux: time

$$\text{Prog. execn. time} = \# \text{cycles} \times \text{clock-cycle-time}$$

$$= \# \text{cycles} / \text{clock-frequency}$$

$$\text{Prog. execn. time} = \# \text{instrns} \times [\# \text{cycles}/\text{instrn}] \times \text{clock-cycle-time}$$

executed \downarrow
average across
executed instructions

Mnemonic:

$$\frac{\text{Time}}{\text{Prog}} = \frac{\text{Instructions}}{\text{Prog}} \times \frac{\# \text{cycles}}{\text{instrn}} \times \frac{\text{Time}}{\text{cycle}}$$

Use of the Computer Performance Equation: Example-1

Should MIPS support **blt** instruction?

Option-A: yes

Implications: #instructions = 5 million, 20% higher cycle time

Option-B: no

Implications: 10% instructions are blt, need to be replaced with 2 instructions

Execn time in A: $5 \text{ million} \times \text{CPI} \times (1.2 t)$ *slower*

Execn time in B: $5.5 \text{ million} \times \text{CPI} \times t$ *faster*

Use of the Computer Performance Equation: Example-2

Two implementations of the same instruction set:

Implementation-1: 2GHz, CPI=1.5

Implementation-2: 2.4GHz, CPI=2

Which is faster and by how much?

$$\text{Execn. time}_{\text{Option-1}}: I \times 1.5 \times \frac{1}{(2 \times 10^9)}$$

$$\text{Option-2: } I \times 2 \times \frac{1}{(2.4 \times 10^9)} \rightarrow \frac{20}{18}$$

Use of the Computer Performance Equation: Example-3

Intel instruction set supports memory as operand in add:

→ Option-1: implement add as 3 cycles

Implication: #cycles increases from 2 million to 2.4 million

→ Option-2: additional hardware support

Implication: cycle length increases by 10%

Which option is better?

$$ET_1: (2.4 \times 10^6) \times t$$

$$ET_2: (2 \times 10^6) \times 1.1t \text{ faster}$$

Measuring the Factors in the Performance Equation

$$\# \text{instrns} \times CPI \times \text{cycle time}$$



instruction mix

executed
profiling
Simulators
hw counters

$$CPI_{ave} = \sum CPI_i \times f_i$$

Types of instrns:

0.25 x 2	
arithmetic	+ 0.75 x 1
memory	= 1.25
branch	

Factors Affecting Performance

Factor	Aspects Affected
Algorithm	#instructions, sometimes CPI
Programming Language	#instructions, CPI
Compiler	#instructions, CPI
ISA	#instructions, CPI, cycle time
Hardware implementation	CPI, cycle time

Algorithm HLL Compiler ISA H/w impl.

Compiler Design Decision Example

CPI for branch instructions: 2

CPI for lw/sw: 3

CPI for reg-reg: 1

Code-sequence-1: 8 branches, 8 loads, 2 stores, 8 reg-reg

Code-sequence-2: 2 branches, 14 loads, 2 stores, 8 reg-reg

Which is better? By what factor?

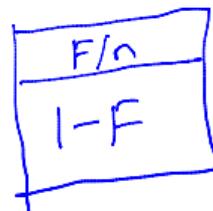
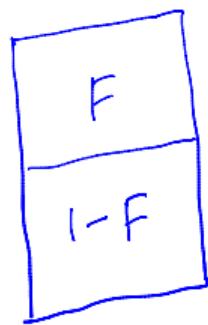
$$ET_1: (8 \times 2 + 8 \times 3 + 2 \times 3 + 8 \times 1) \times t \quad \text{faster: } \frac{60}{54}$$
$$ET_2: (2 \times 2 + 14 \times 3 + 2 \times 3 + 8 \times 1) \times t$$

Workload, Benchmark

- Which program to use for performance analysis?
- Benchmark: special or real ?
- SPEC: System Performance Evaluation Corporation
 - Since 1989
 - SPEC-2000 in textbook (includes gzip, gcc)
- How to summarize performance?
 - Think in terms of reproducibility of results
 - Give all possible details, e.g. input to program

Amdahl's Law

Performance improvement is limited by the fraction of program you are improving



$$\frac{1}{1-F+F/n} < \frac{1}{1-F}$$

Amdahl's Law: An Example

Intel wants to improve its CPU chip

Option-1: memory speedup 10x

Option-2: ALU speedup 2x

$$F_{alu} = 0.5, F_{mem} = 0.2, F_{other} = 0.3$$

Speedup 1 : $\frac{1}{1 - 0.2 + \frac{0.2}{10}} \approx 1.22$

Speedup⁻² : $\frac{1}{1 - 0.5 + \frac{0.5}{2}} \approx 1.33 \checkmark$

Summary

- Computer performance quantification
 - Execution time is the primary metric
 - Helps answer various design questions quantitatively
- Role of benchmarks
- Amdahl's law

CS305

Computer Architecture

Hardware Implementation of MIPS: Preliminaries

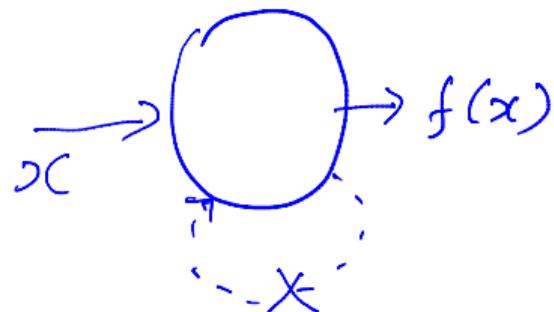
Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Recall: Combinatorial vs Sequential Circuits

Combinatorial circuit

Combinational circuit



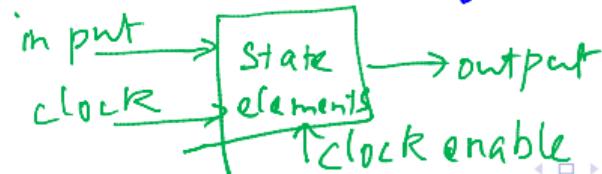
Sequential circuit

State elements

asynchronous

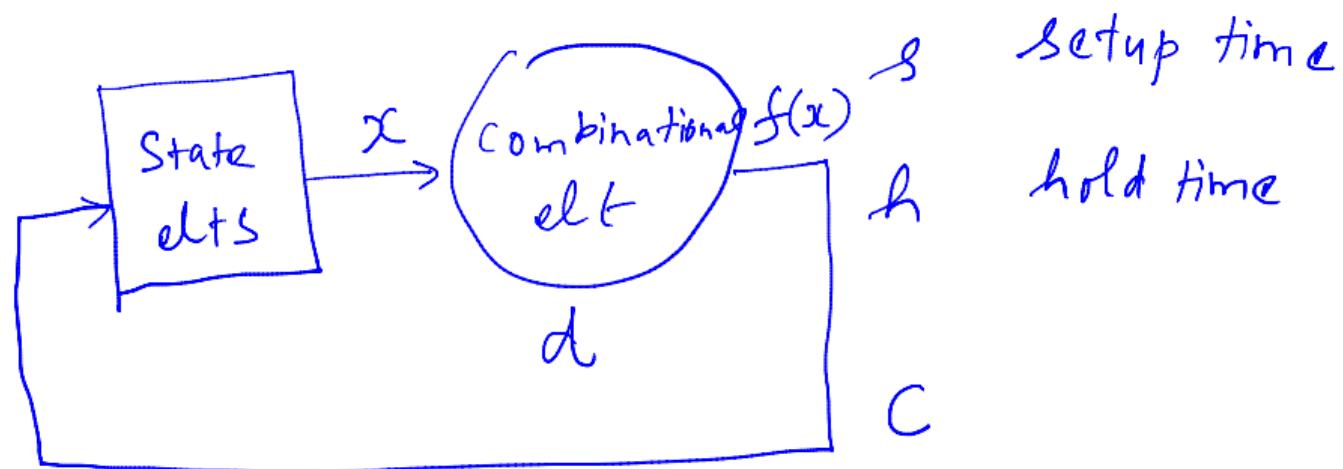
Synchronous
clock

level-triggered



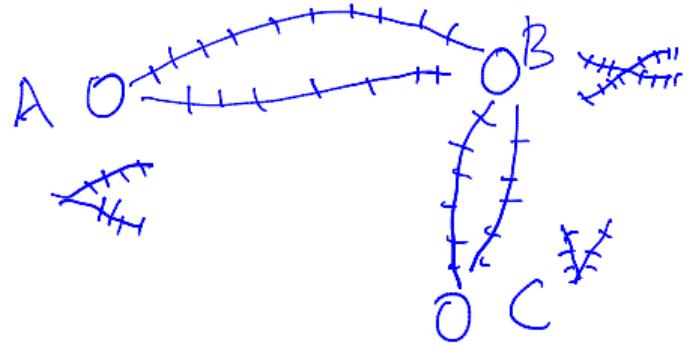
edge-triggered

Sequential + Combinational Circuit



$$c > d + s + h$$

Control Path, Data Path Analogy

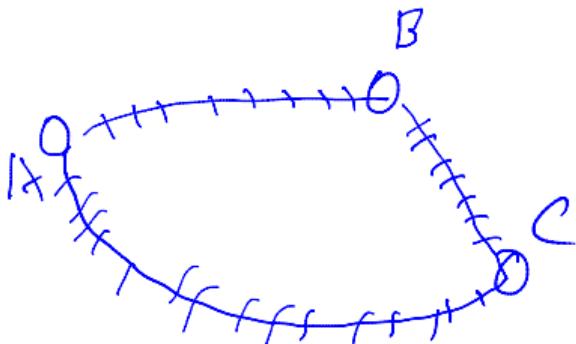


Data path

- platforms
- rail lines

Control path / signals

- trains should not collide
- no deadlock



Control Path, Data Path in MIPS Implementation

Data path

- state dts
- combinational dts
- interconnections

Control path

- when to write
- what to write

Summary

- Synchronous (clocked) sequential circuit, edge-triggered
- Increasing complexity:
 - Single-cycle implementation
 - Multi-cycle implementation
 - Pipelined implementation
- **DRAW** circuit diagrams to **LEARN** effectively
 - Not enough to look at drawings

Week3/4 single-cycle-impl.pdf

CS305

Computer Architecture

Single Cycle Implementation of MIPS ISA (Subset)

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

MIPS ISA Subset

- Reg-reg: **add**, **sub**, **and**, **or**, **slt**
- Memory: **lw**, **sw**
- Branch: **beq** (and **j** later)
- Subset → keep it simple, understand techniques

Before Proceeding, Test Your Understanding

- What kind of arguments does **add** take ?
- How many registers need to be specified in **lw** ?
- How many registers need to be specified in **sw** ?
- What is the least integer value of offset in **lw** ?
- What is the largest integer value of offset in **sw** ?
- What instruction format is used by **beq** ?
- What is the role of the immediate operand in **beq** ?

Recall: MIPS Instruction Format

opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
---------------	-----------	-----------	-----------	--------------	--------------

R-type instruction: register-register operations

opcode (6)	rs (5)	rt (5)	immediate/constant or offset (16)
---------------	-----------	-----------	--------------------------------------

I-type instruction: loads, stores, all immediates, *conditional branch, jump register, jump and link register*

opcode (6)	offset relative to PC (26)
---------------	-------------------------------

J-type instruction: *jump, jump and link, trap and return*

Steps in Program Execution



add, sub, and, or, slt

lw, sw

beq

✓ (2a) read register file

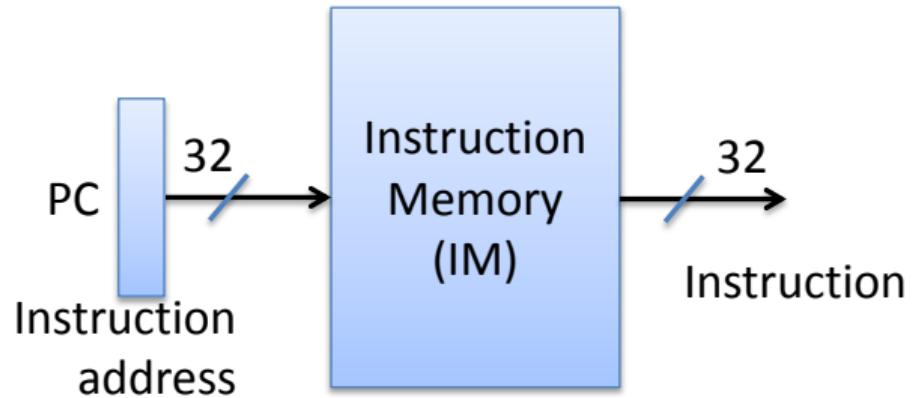
(2b) Specific to instruction

✓ (2c) $PC = \underbrace{PC + 4}$

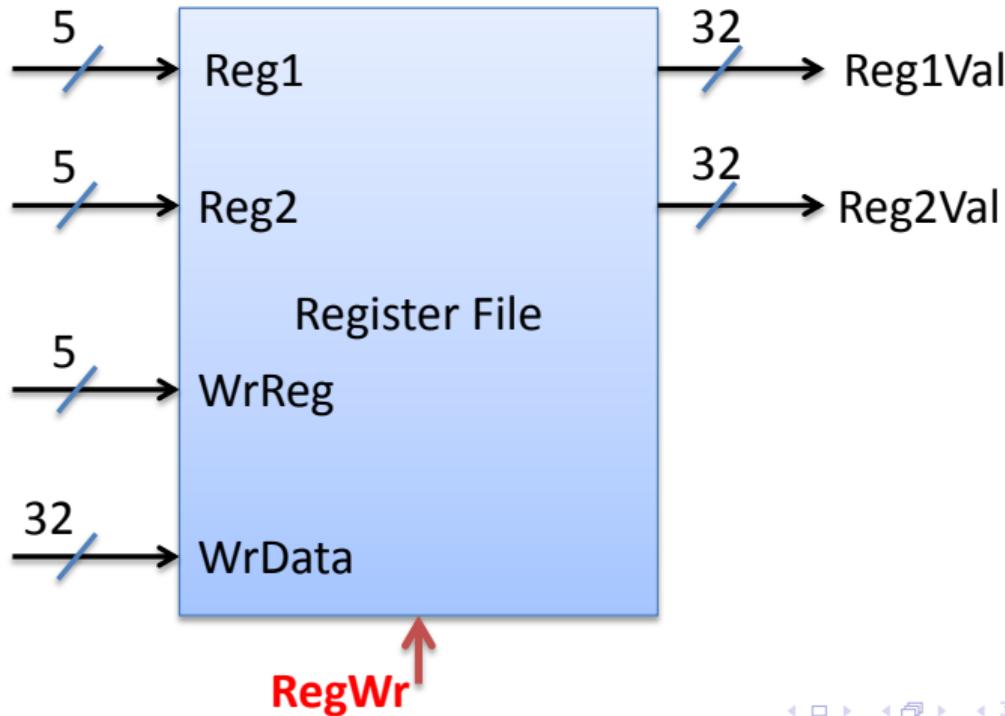
Hardware components

- put them together

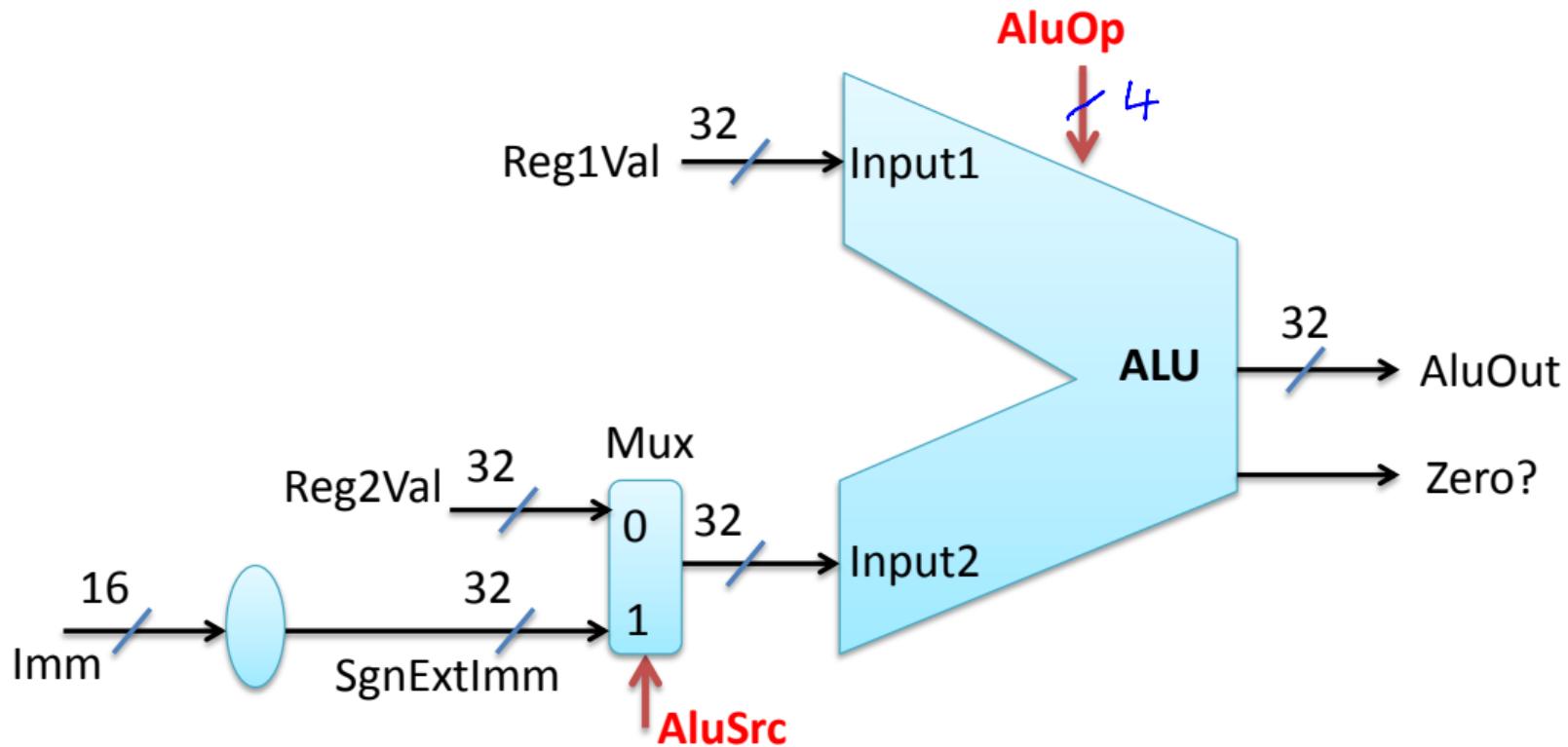
Elements for Instruction Fetch



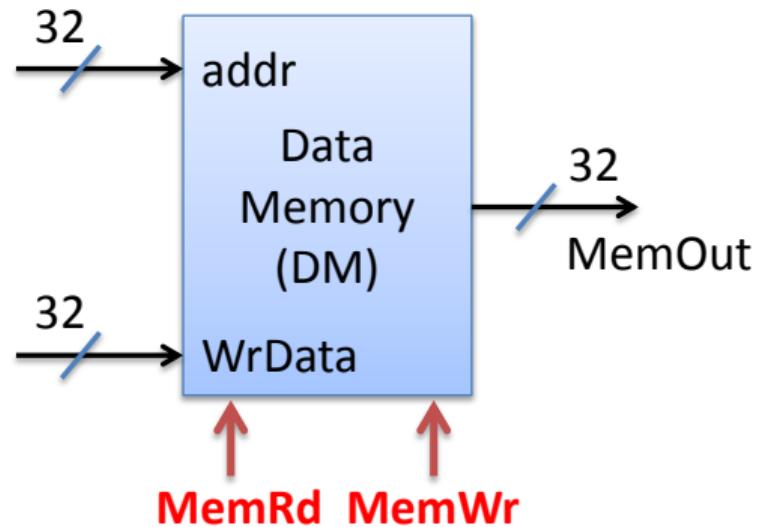
Element for Register Read/Write: The Register File



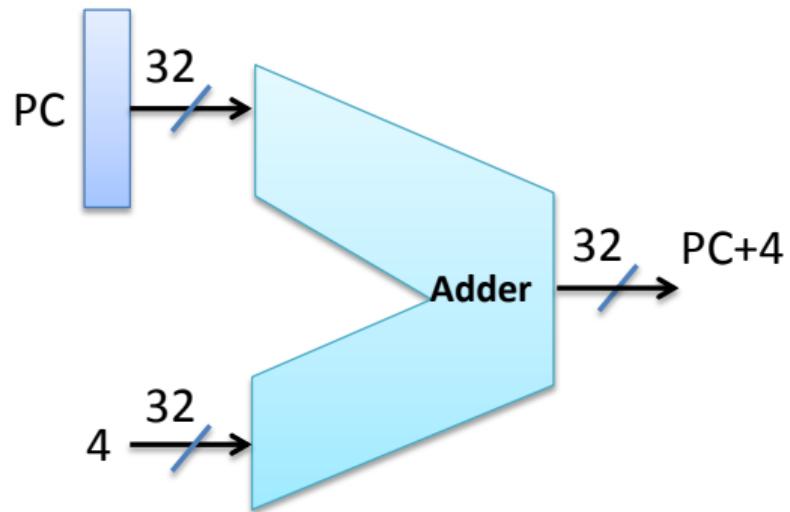
ALU: Arithmetic Logic Unit



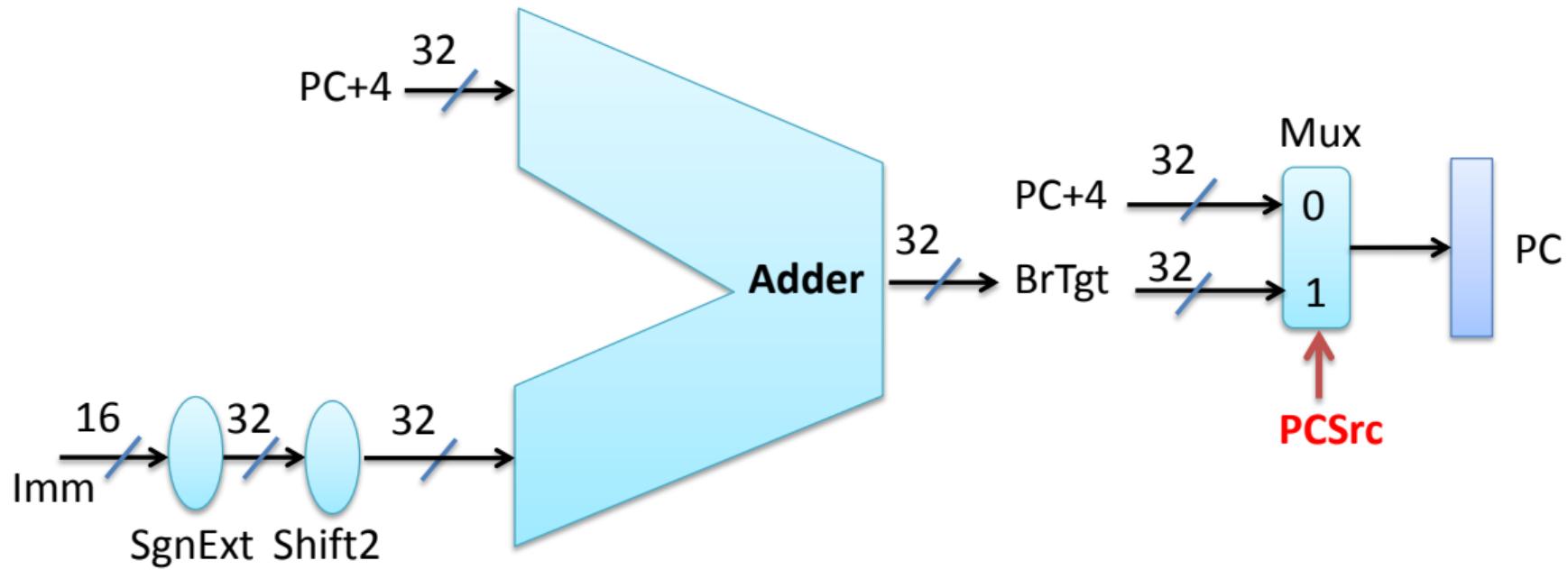
Data Memory



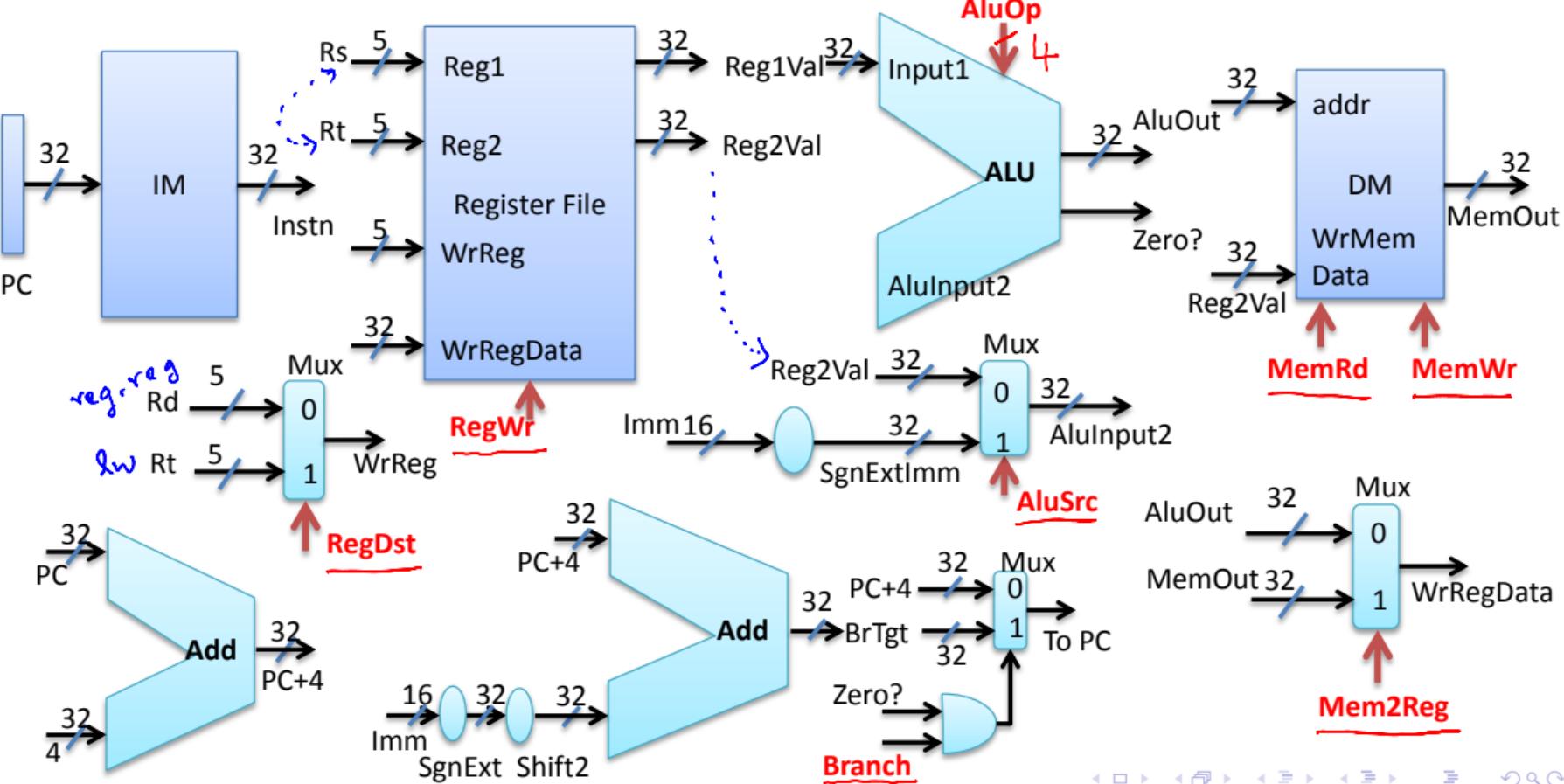
Element for PC+4 Computation



Additional Elements to Implement beq



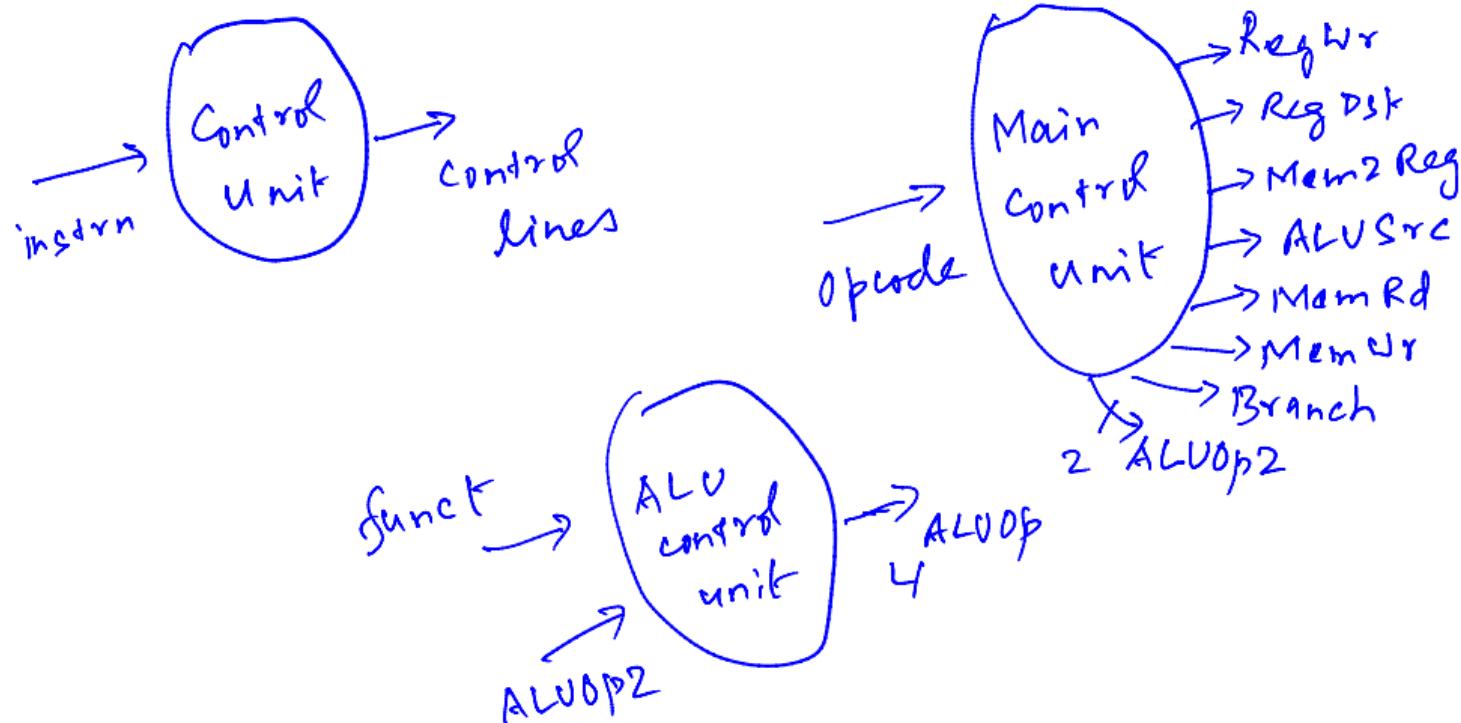
Putting it All Together



Summary of Control Lines

- RegDst (1): to decide Rd vs Rt
- RegWr (1): should register file be written?
- ALUSrc (1): to decide Rt vs SignExtImm
- MemRead (1): should data memory be read?
- MemWrite (1): should data memory be written?
- Mem2Reg (1): to decide ALUOut vs MemOut
- Branch (1): is this a **beq** instruction?
- AluOp (4): which ALU operation to perform

Main Control Unit, ALU Control Unit



Truth Table for Main Control Unit

	RegDst	RegWr	ALUSrc	MemRd	MemWr	Mem2Reg	Branch	ALUOp2
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00
beq	x	0	Reg2Val (0)	0	0	x	1	01

- Can produce optimized combinational circuit to implement truth table
- Similar truth table for ALU control unit as well
- Q: why is MemRd always explicitly enabled or disabled?

Summary

- Single cycle implementation of MIPS ISA subset
 - Sequential, combinational components for different instructions
 - Put together in a datapath
 - Control lines define the control path
 - Control lines generated from opcode + funcode
- Next: extending the implementation to support other instructions

CS305

Computer Architecture

Single Cycle Implementation of MIPS ISA (Subset)

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

MIPS ISA Subset

- Reg-reg: **add**, **sub**, **and**, **or**, **slt**
- Memory: **lw**, **sw**
- Branch: **beq** (and **j** later)
- Subset → keep it simple, understand techniques

Before Proceeding, Test Your Understanding

- What kind of arguments does **add** take ?
- How many registers need to be specified in **lw** ?
- How many registers need to be specified in **sw** ?
- What is the least integer value of offset in **lw** ?
- What is the largest integer value of offset in **sw** ?
- What instruction format is used by **beq** ?
- What is the role of the immediate operand in **beq** ?

Recall: MIPS Instruction Format

opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
---------------	-----------	-----------	-----------	--------------	--------------

R-type instruction: register-register operations

opcode (6)	rs (5)	rt (5)	immediate/constant or offset (16)
---------------	-----------	-----------	--------------------------------------

I-type instruction: loads, stores, all immediates, *conditional branch, jump register, jump and link register*

opcode (6)	offset relative to PC (26)
---------------	-------------------------------

J-type instruction: *jump, jump and link, trap and return*

Steps in Program Execution



add, sub, and, or, slt

lw, sw

beq

Hardware components

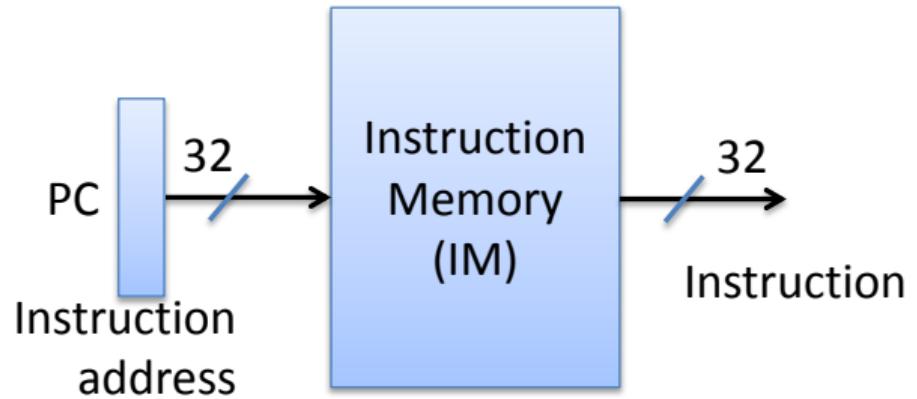
- put them together

✓ (2a) read register file

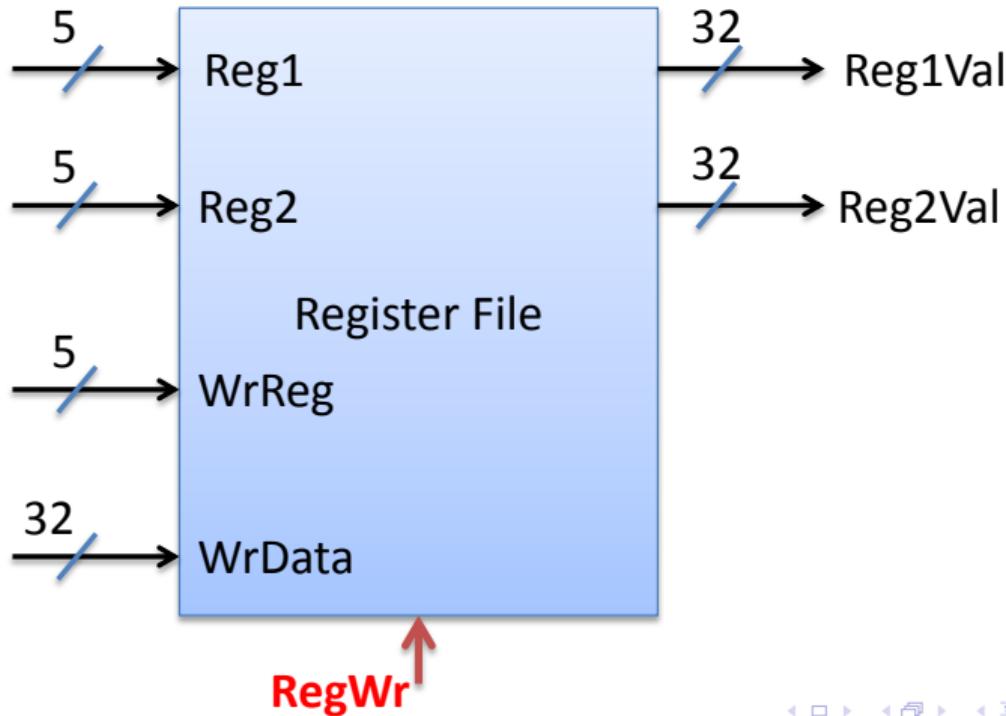
(2b) Specific to instruction

✓ (2c) $PC = \underbrace{PC + 4}$

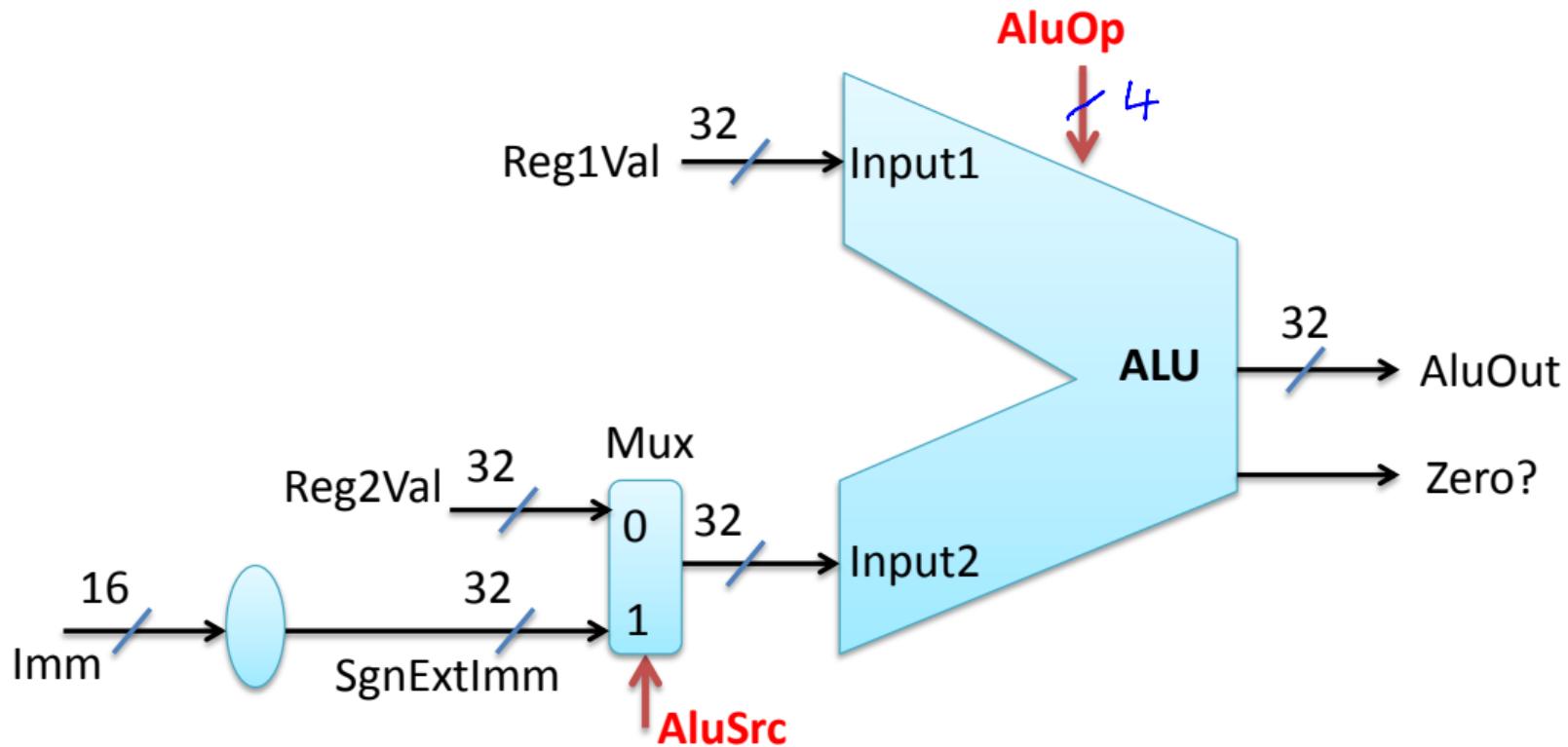
Elements for Instruction Fetch



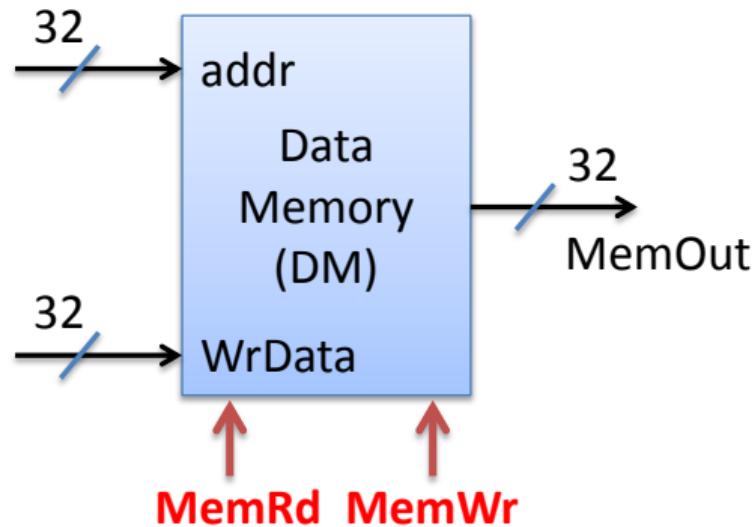
Element for Register Read/Write: The Register File



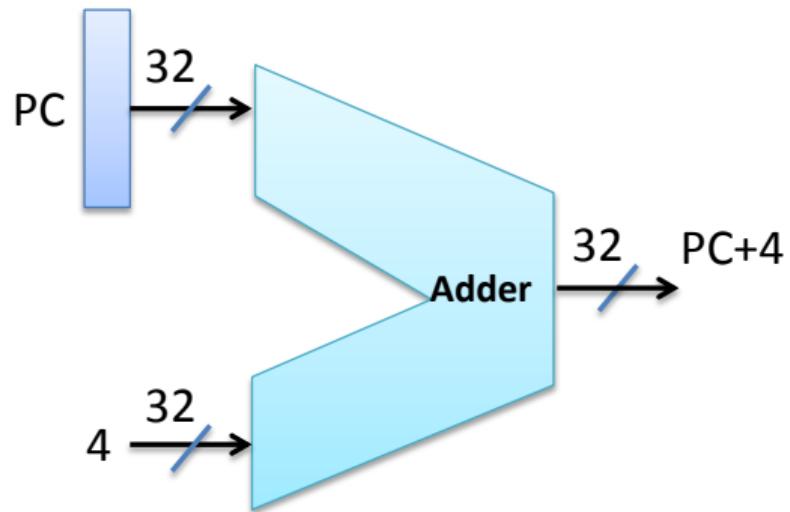
ALU: Arithmetic Logic Unit



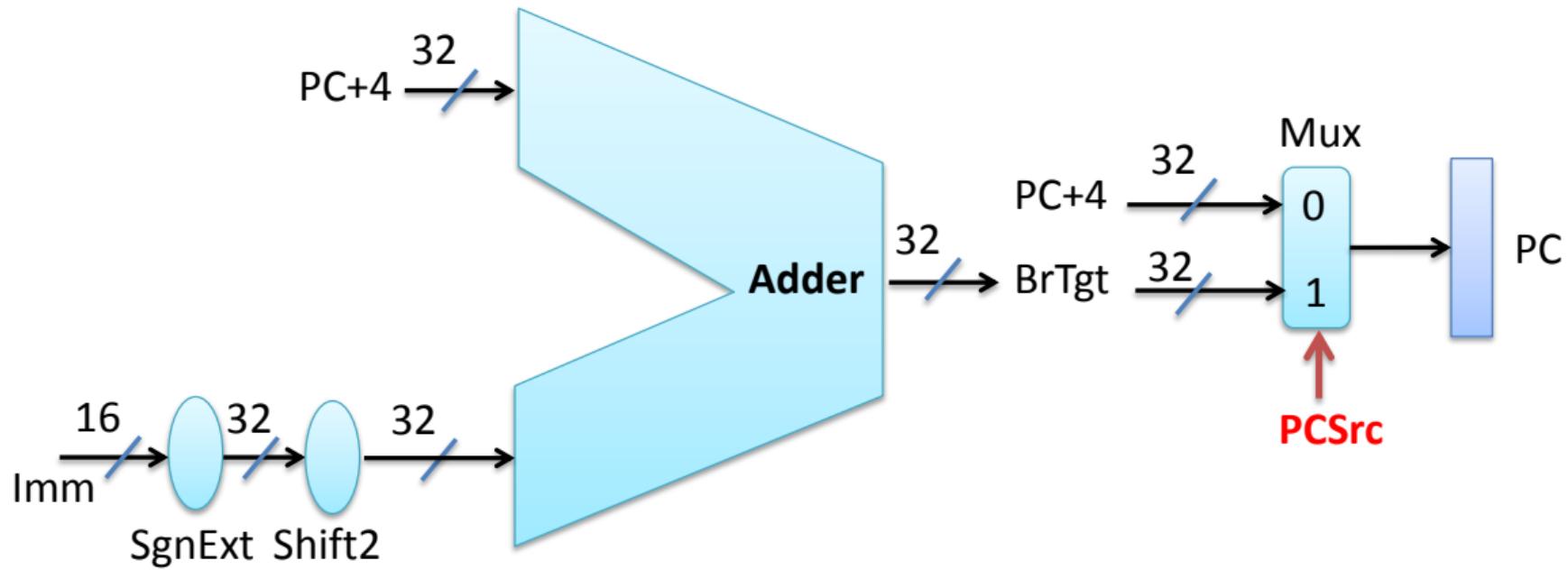
Data Memory



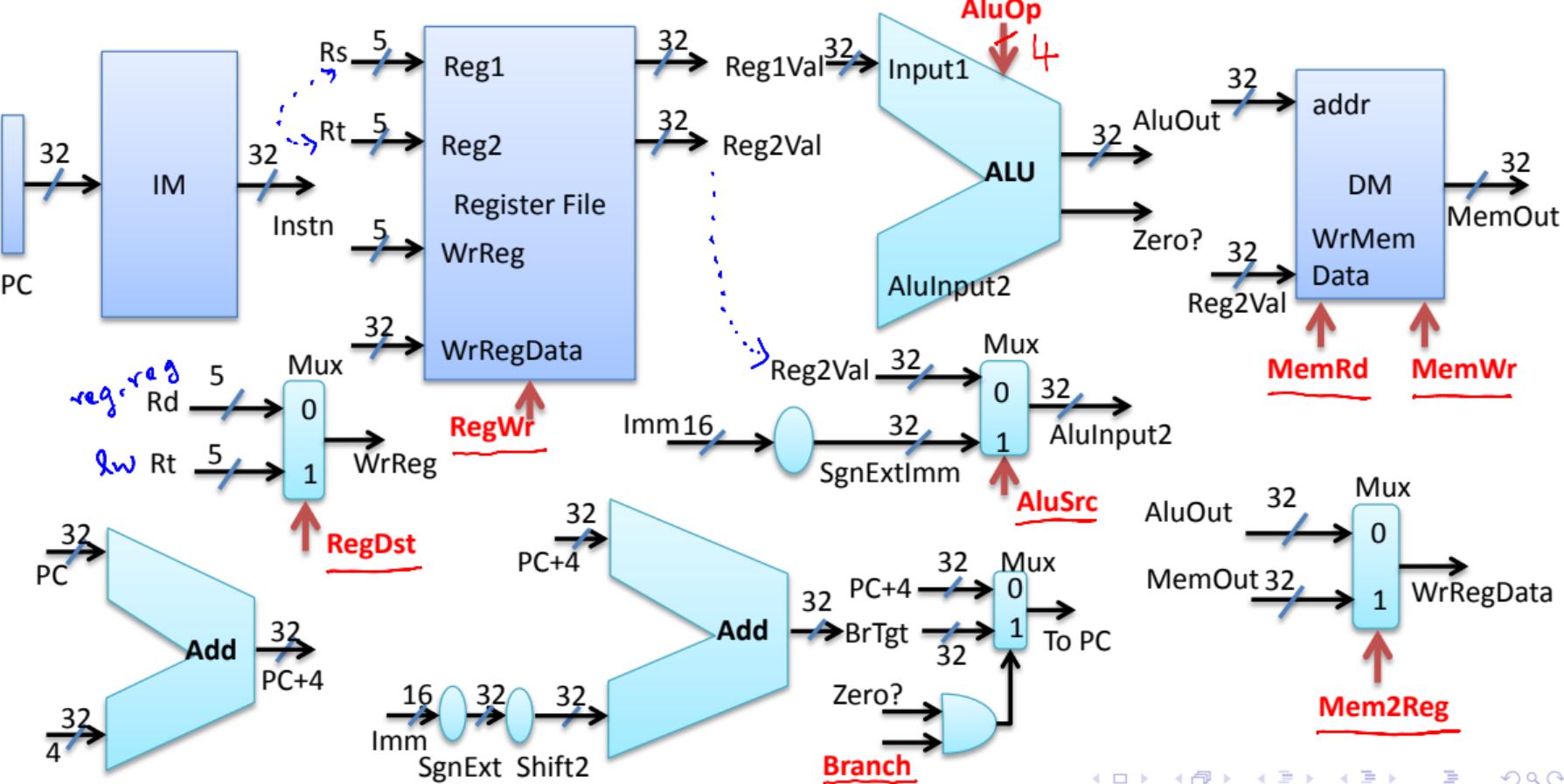
Element for PC+4 Computation



Additional Elements to Implement beq



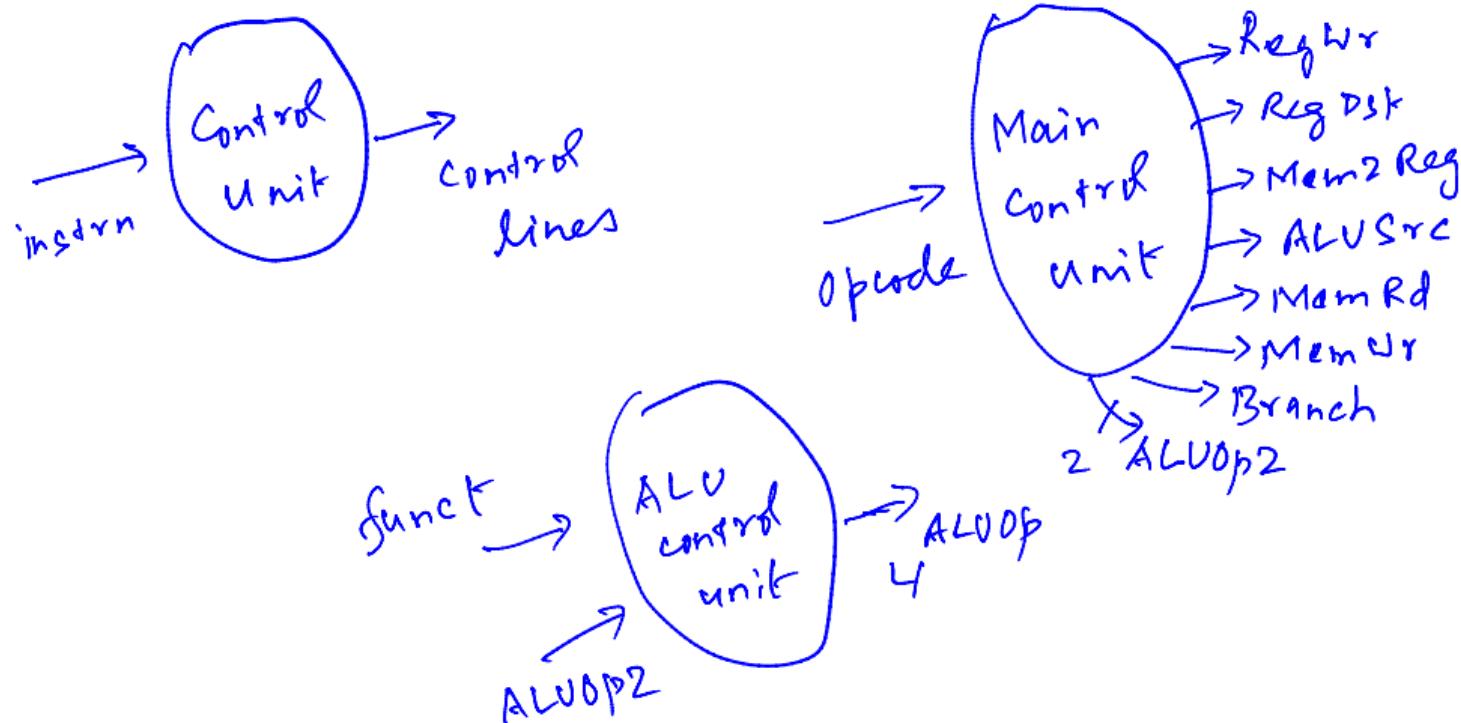
Putting it All Together



Summary of Control Lines

- RegDst (1): to decide Rd vs Rt
- RegWr (1): should register file be written?
- ALUSrc (1): to decide Rt vs SignExtImm
- MemRead (1): should data memory be read?
- MemWrite (1): should data memory be written?
- Mem2Reg (1): to decide ALUOut vs MemOut
- Branch (1): is this a **beq** instruction?
- AluOp (4): which ALU operation to perform

Main Control Unit, ALU Control Unit



Truth Table for Main Control Unit

	RegDst	RegWr	ALUSrc	MemRd	MemWr	Mem2Reg	Branch	ALUOp2
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00
beq	x	0	Reg2Val (0)	0	0	x	1	01

- Can produce optimized combinational circuit to implement truth table
- Similar truth table for ALU control unit as well
- Q: why is MemRd always explicitly enabled or disabled?

Summary

- Single cycle implementation of MIPS ISA subset
 - Sequential, combinational components for different instructions
 - Put together in a datapath
 - Control lines define the control path
 - Control lines generated from opcode + funcode
- Next: extending the implementation to support other instructions

Week4/2 single-cycle-extn.pdf

CS305

Computer Architecture

Extending the Single Cycle MIPS Implementation

reg-reg

lw, sw

beq

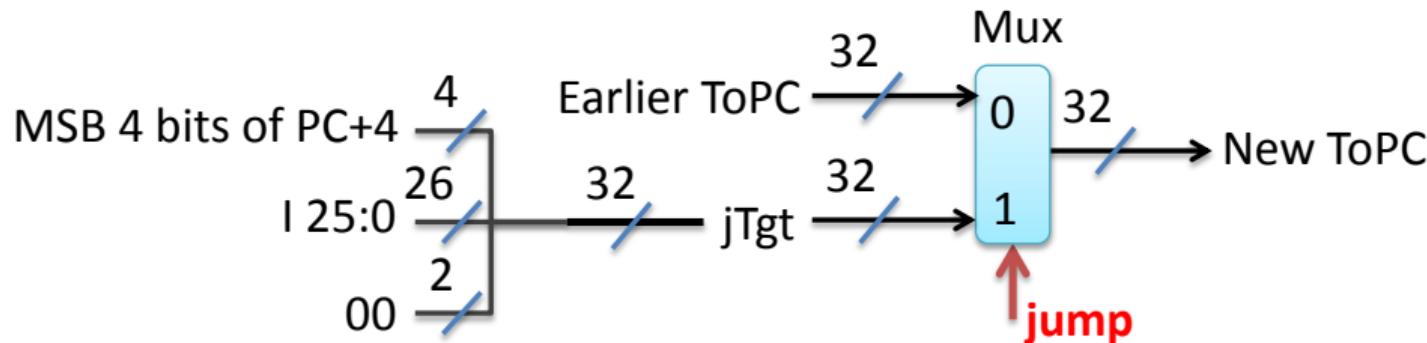
Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

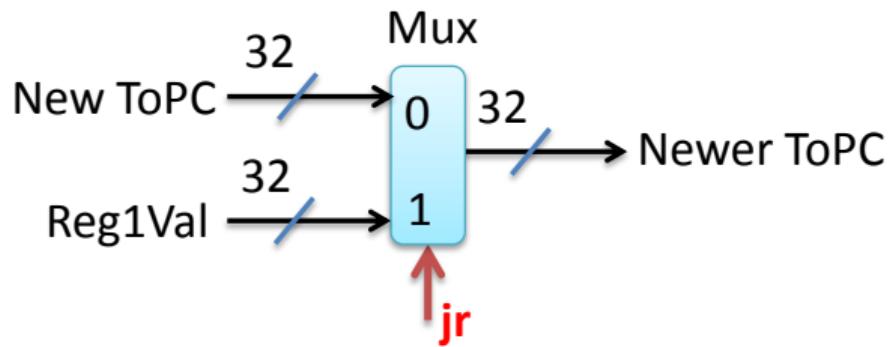
Data Path, Control Path Extensions to Support j



Main Control Unit Truth Table Enhancement

	RegDst	RegWr	ALUSrc	Mem-Rd	Mem-Wr	Mem2-Reg	Branch	ALU-Op2	Jump
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10	0
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00	0
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00	0
beq	x	0	Reg2Val (0)	0	0	x	1	01	0
j	x	0	x	0	0	x	x	x	1

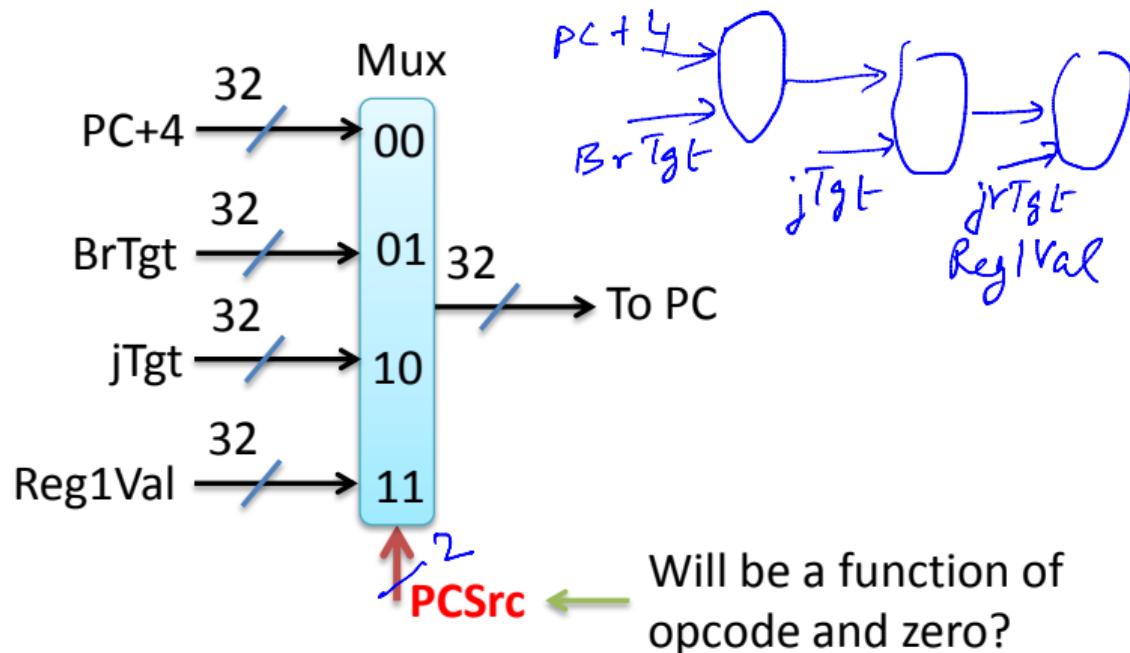
Further Data Path, Control Path Extensions to Support jr



Main Control Unit Truth Table Enhancement

	Reg-Dst	Reg-Wr	ALUSrc	Mem-Rd	Mem-Wr	Mem2-Reg	Branch	ALU-Op2	Jump	Jr
Reg-Reg	Rd (0)	1	Reg2Val (0)	0	0	ALUOut (0)	0	10	0	0
lw	Rt (1)	1	SgnExt-Imm (1)	1	0	MemOut (1)	0	00	0	0
sw	x	0	SgnExt-Imm (1)	0	1	x	0	00	0	0
beq	x	0	Reg2Val (0)	0	0	x	1	01	0	0
j	x	0	x	0	0	x	x	x	1	0
jr	x	0	x	0	0	x	x	x	x	1

Alternate Data Path, Control Path Modification to Support j , jr



Summary

- The data-path and control-path can be extended to support further instructions
 - In some cases, original data-path, control-path must be modified
- Understand general principle before proceeding
 - Identify hardware component(s) for instruction
 - String them together, enhance data-path, control-path
 - Enhance/modify main control unit (truth table)

Week4/3 single-cycle-analysis.pdf

CS305

Computer Architecture

Analysis of the Single Cycle MIPS Implementation

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Single Cycle Implementation: When to Activate Each Element?

- Edge triggered versus level triggered?
 - Different (sequential) elements must trigger at different delays with respect to the start of an instruction fetch+execution
 - This can happen only in a level triggered implementation
- Uncontrolled state change with level triggering?
 - Make **PC-write** level triggered with **latter half of cycle**
 - Also make **RegWr** level triggered with **latter half of cycle**
 - Also make **MemWr** level triggered with **latter half of cycle**



Single Cycle Implementation is Inefficient

- Additional **delay** in clock cycle
 - Each instruction must be as long as instruction with longest delay!
 - E.g. **j** versus **lw**
- **Cost:** several additional pieces of hardware
 - I vs D memory separation
 - Adder for PC+4 is separate
 - Adder for BrTgt is separate

Illustrating the Inefficiency

- Main components:
 - Instruction memory
 - Register file
 - ALU
 - Data memory
- Say, each component takes 100 pico-sec
- What is the max clock frequency?

$$0.35 \times 400 + 0.25 \times 400 + 0.2 \times 500 + 0.2 \times 300$$

Components involved in:

- Register-register instructions
- **sw**
- **lw**
- **beq**
- **j**

lw is longest: $5 \times 100 \text{ ps} = 0.5 \text{ ns}$

Max clock frequency = 2 GHz

Suppose we (magically) have variable clock cycles?

Instruction mix: R 35%, sw 25%, lw 20%, beq 20%

Avg. time per instrn. = 400 ps

Summary

- Single cycle implementation
 - Higher cost than what looks optimal
 - Slower than what looks optimal
- Multi-cycle implementation addresses both
 - Reuse of hardware components
 - Each instruction takes only as long as needed

Week5/1 intro-to-pipelining.pdf

CS305

Computer Architecture

Introduction to Pipelining

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Some Fun Videos to Watch

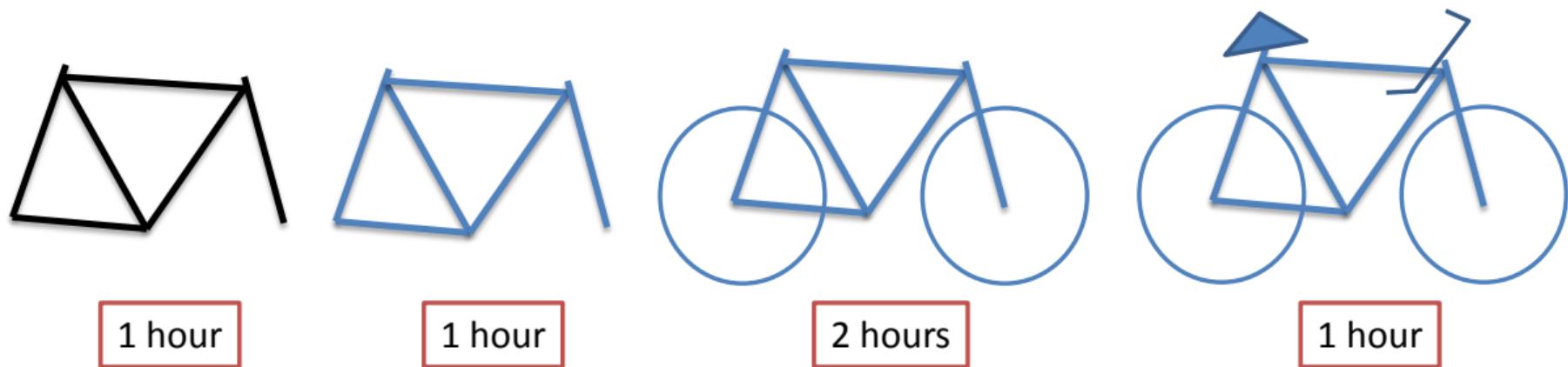
- Charlie Chaplin's critique of industrialization
 - https://www.youtube.com/watch?v=a-FbVF1x1_U
- Modern Times, Assembly Line with Charlie Chaplin
 - <https://www.youtube.com/watch?v=QdwH84AT5fU>
- I Love Lucy – The Candy Wrapping Job
 - <https://www.youtube.com/watch?v=Clr6Zyij7iI>

Lessons Learnt

- Pipelining is natural (at least for machines)
- Overall speed only as good as the slowest unit
- Non-uniformity is bad for pipeline
- Exceptions are really bad

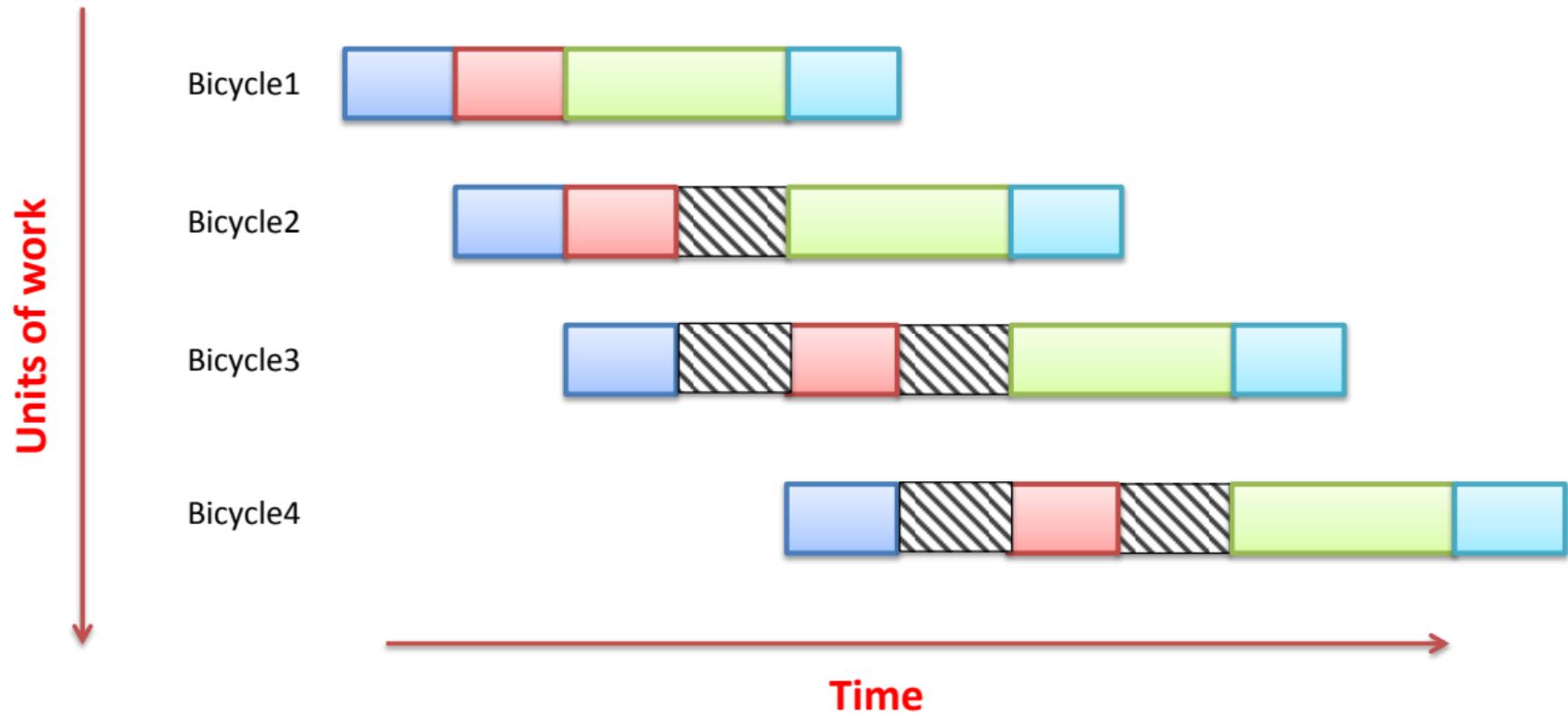
Example Task: Bicycle Manufacture

Make frame → Paint frame → Fit wheels → Fit accessories

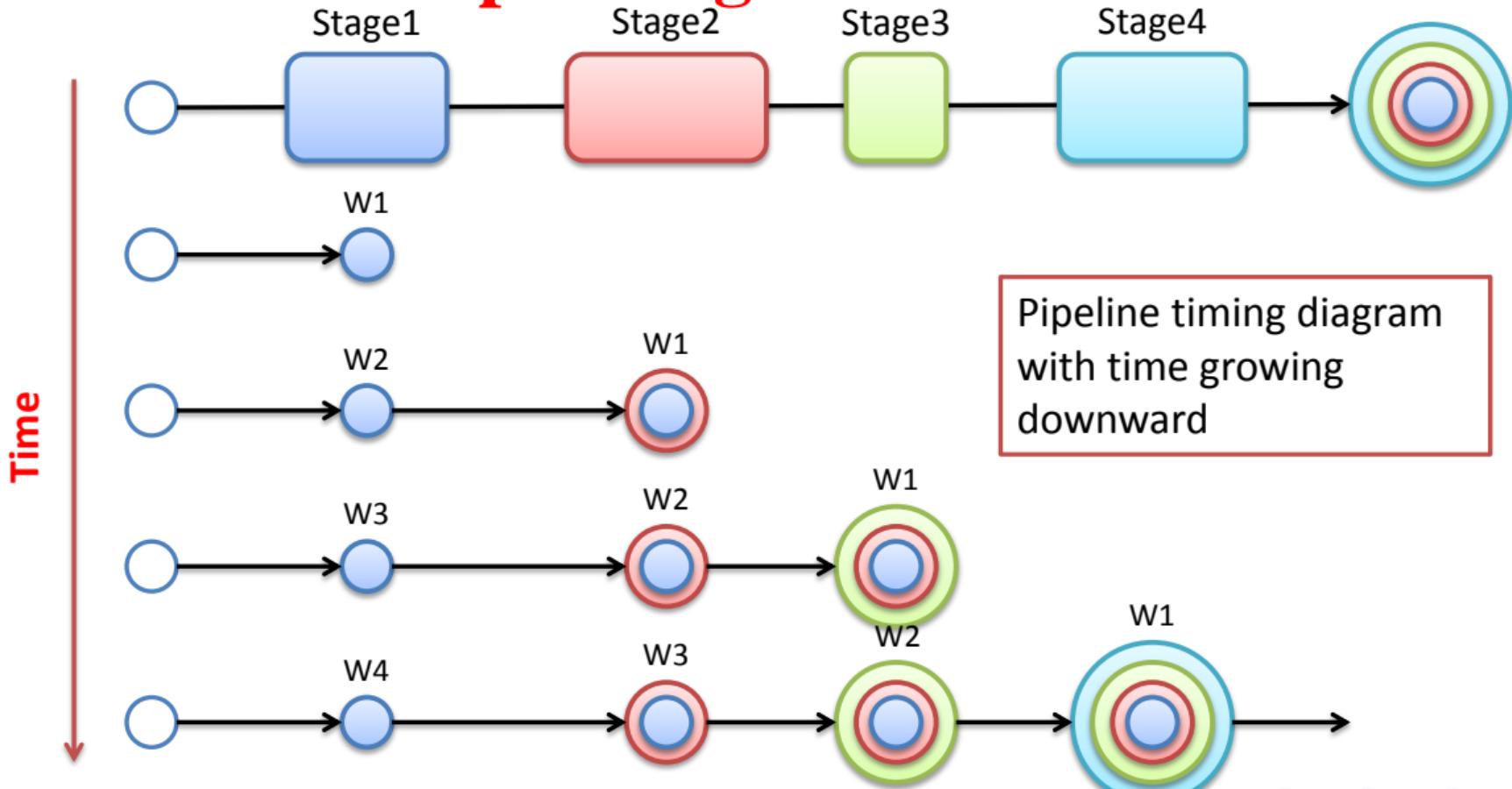


- Time to make 1 bicycle?
- Time to make 10 bicycles?
 - With pipelining? Without pipelining?
 - Pipelining:
 - “Make frame” for 2nd bicycle, when “Paint frame” for 1st bicycle
 - “Make frame” for 3rd bicycle, “Paint frame” for 2nd bicycle, when “Fit wheels” for 1st cycle
 - And so on...

Pipeline Timing Diagram

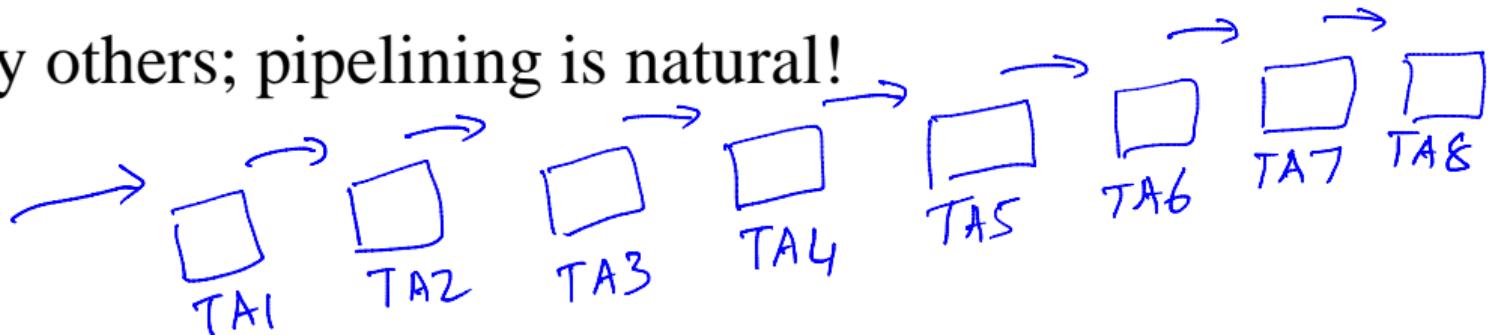
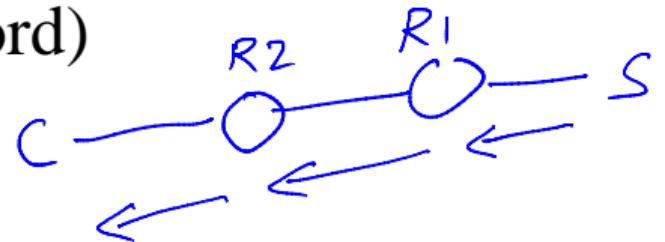


Pipelining in Abstract



Other Examples of Pipelining

- Water pipeline (origin of word)
- Network packets
- Course project evaluation
- Many others; pipelining is natural!



Pipeline Speedup

- Time taken to manufacture N bicycles
 - Without pipelining: $5N$
 - With pipelining: $5 + (N-1) \times 2$
 - Speedup = $5/2$ for large N
- Suppose “Fit wheels” also takes 1 hour?
 - Speedup = $4N/(4+N-1) = 4$ for large N
- Suppose “Fit wheels” takes 2 hours, but is broken into “Fit front wheel”, “Fit rear wheel” each of 1 hour?
 - Speedup = $5N/(5+N-1) = 5$ for large N

Pipeline Speedup (continued)

- In general, k stages, each of time t
- Time for N units:
 - Without pipeline: $N \times kt$
 - With pipelining: $kt + (N-1)t$
 - Pipeline setup time = time to fill pipeline = time to first output
 - Is a significant component, for small N
 - Speedup = k for large N
 - This is the **ideal** speedup

Ideal Pipeline Speedup

- Ideal pipeline speedup = number of pipeline stages
- Necessary conditions:
 - All stages are of equal length in time
 - Enough hardware/hands: e.g. same spanner/person/machine cannot be used for “fit wheels” and “fit accessories”
 - Many, many more conditions (stated later)

Latency versus Throughput

- Latency: from perspective of single unit of work
- Throughput: rate at which work completes

Pipelining in MIPS

Stage-1:

Instruction fetch, PC=PC+4 (all instructions)

IF

Stage-2:

Reg read, branch target computation (all instructions)

ID

Stage-3:

ALU operation (reg-reg)

Memory address computation (lw, sw)

Branch condition computation (beq)

EX

Stage-4:

Memory access (lw, sw)

Reg write back (reg-reg)

MEM

Stage-5:

Reg write back (lw)

WB



Pipelining for Sequence of lw

lw \$t0, 0(\$sp)



lw \$t1, 4(\$sp)



lw \$t2, 8(\$sp)



Such pipelined execution of MIPS instructions is *potentially* possible

Pipelining for Sequence with Other Instructions

add \$t0, \$t1, \$t2



ori \$t1, \$s0, 15



lw \$t2, 8(\$sp)



Effect of moving WB to stage-5 of reg-reg ?
Latency of those instructions increases
Instruction throughput not affected!

Summary

- Pipelining a natural idea for speed-up
- Latency versus throughput
- MIPS: uniformity of instructions allows pipelining
- Issues with pipelining: hazards
 - Charlie Chaplin wants to scratch under his arm, take a break
 - Gets slowed down by buzzing bee
 - Pipeline stops → startup delays

Week5/2 structural-hazards-pipelined-datapath.pdf

CS305

Computer Architecture

Structural Hazards, Pipelined Datapath

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Issues with Pipelining: Hazards

- Hazard: condition or situation which does not allow the pipeline to operate “normally”
- Three kinds:
 - Structural hazards
 - Data hazards
 - Control hazards

Structural Hazard: Hardware Limitation

- Ideally, no hardware limitation
 - Separate spanners for “fit wheels” and “fit accessories”
 - IM and DM must be separate
 - Separate adders for PC+4, branch target computation
- If hardware limitation: structural hazard
 - E.g. same memory for IM and DM
 - Can still use pipelining, as much as possible

Structural Hazard → Stall Pipeline

lw \$t0, 0(\$sp)



lw \$t1, 4(\$sp)



lw \$t2, 8(\$sp)



lw \$t3, 0(\$fp)



Bubble in pipeline = 1 Stall = 1 cycle wasted

Performance Evaluation in Pipelined Implementation

- CPI in ideal pipeline = 1
 - CPI is cycles per instruction completion
- CPI in pipeline with stalls = $1 + F_{stall}$

Example:

Multi-cycle implementation has 9ns cycle time

Pipelined implementation has 10ns cycle time (due to extra datapath overheads)

Instruction mix:

Reg-reg 65%, beq 15%, lw 15%, sw 5% $CPI = 4$ $CPI = 1$ (ideal) $\frac{4 \times 9 \text{ ns}}{1 \times 10 \text{ ns}} = 3.6$

What is the ideal speed-up?

What is the speed-up with structural hazard in memory?

$$\frac{4 \times 9}{1.2 \times 10} = 3$$

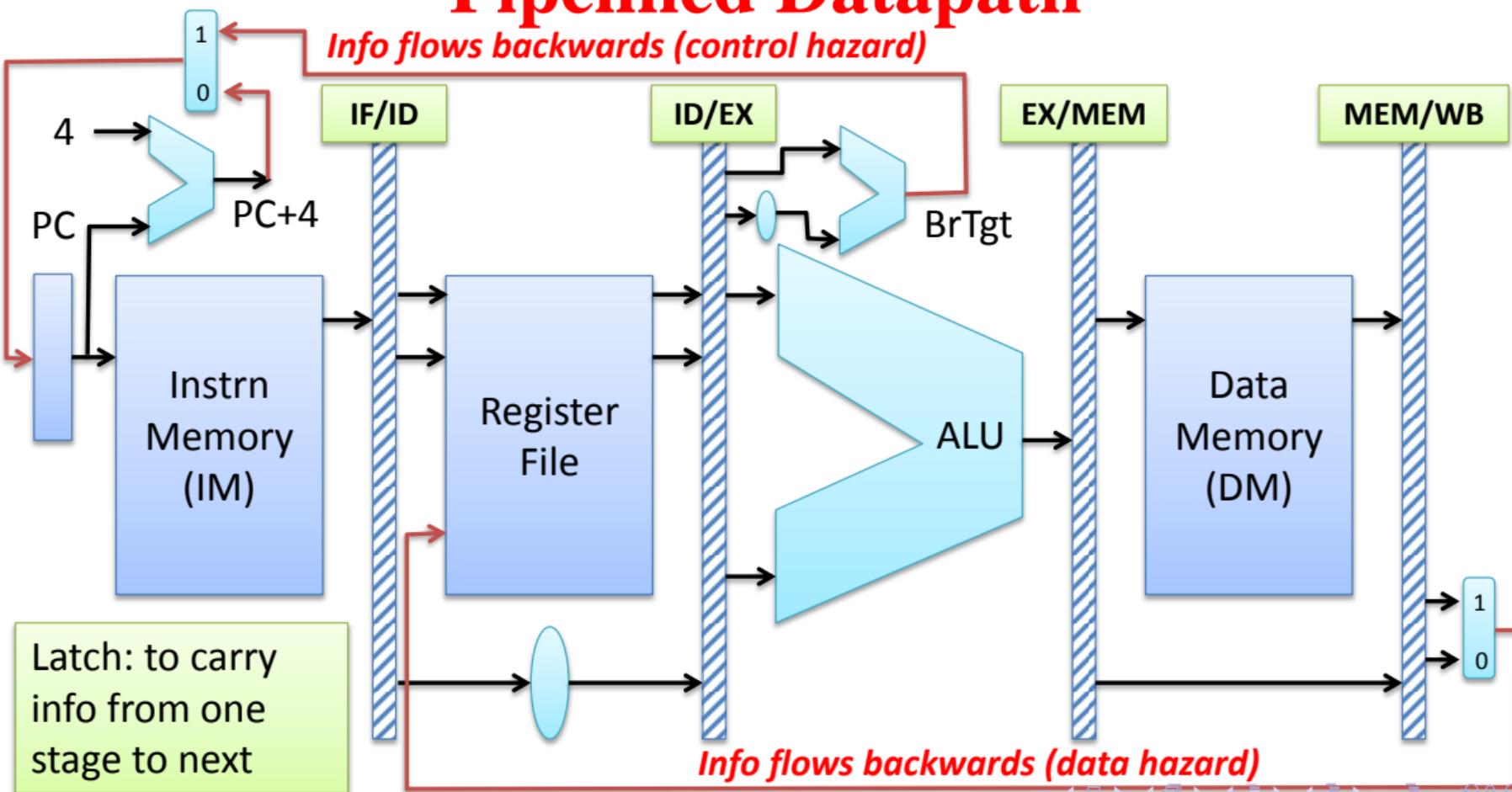
Handling Structural Hazard in the Register File

- Too expensive to stall
- A large fraction of instructions write registers
- Solution:
 - Write RegFile on rising edge (first half)
 - Read RegFile on falling edge (second half)
 - Why this order? Has to do with handling of data hazards

Pipelined Datapath, Control

- Modifications from multi-cycle
 - Simple modifications to datapath
 - Control becomes much more complex
- Same pattern as: single-cycle → multi-cycle

Pipelined Datapath



Summary

- Pipeline hazards:
 - Structural hazard
 - Data hazard
 - Control hazard
- Dealing with hazards: pipeline stall, affects ideal CPI
- Pipelined datapath: minor modifications
- Pipelined control: much more complex!

Week5/3 data-hazards.pdf

CS305

Computer Architecture

Data Hazards in the Pipeline

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

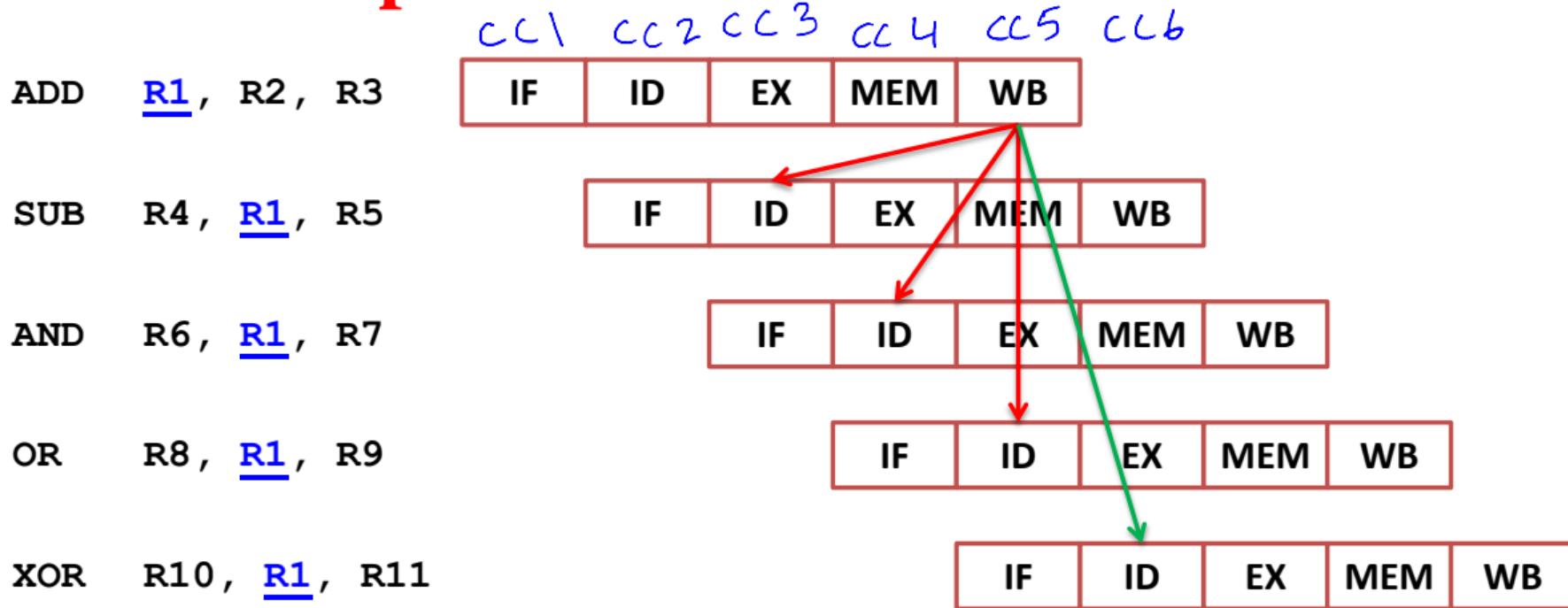
*Structural
→ Data
control*

<http://www.cse.iitb.ac.in/~br>

Data Hazards

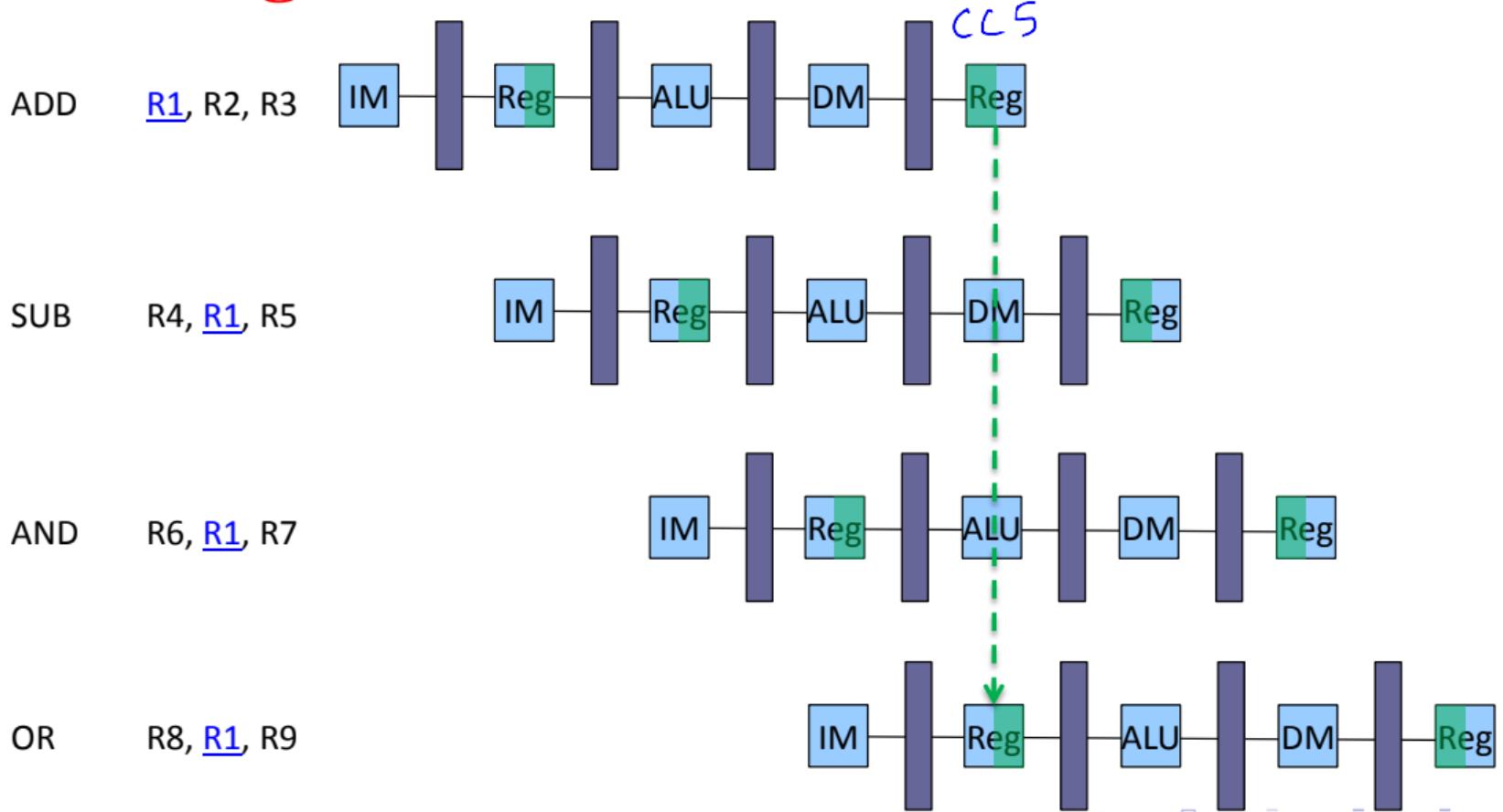
- Data dependence across instructions → data hazard
 - Register dependence
 - Memory dependence
- Example (DLX instruction set):
 - **ADD R1, R2, R3**
 - **SUB R4, R1, R5**
 - **AND R6, R1, R7**
 - **OR R8, R1, R9**
 - **XOR R10, R1, R11**
- All instructions after ADD depend on **R1**

Pipeline With Data Hazards

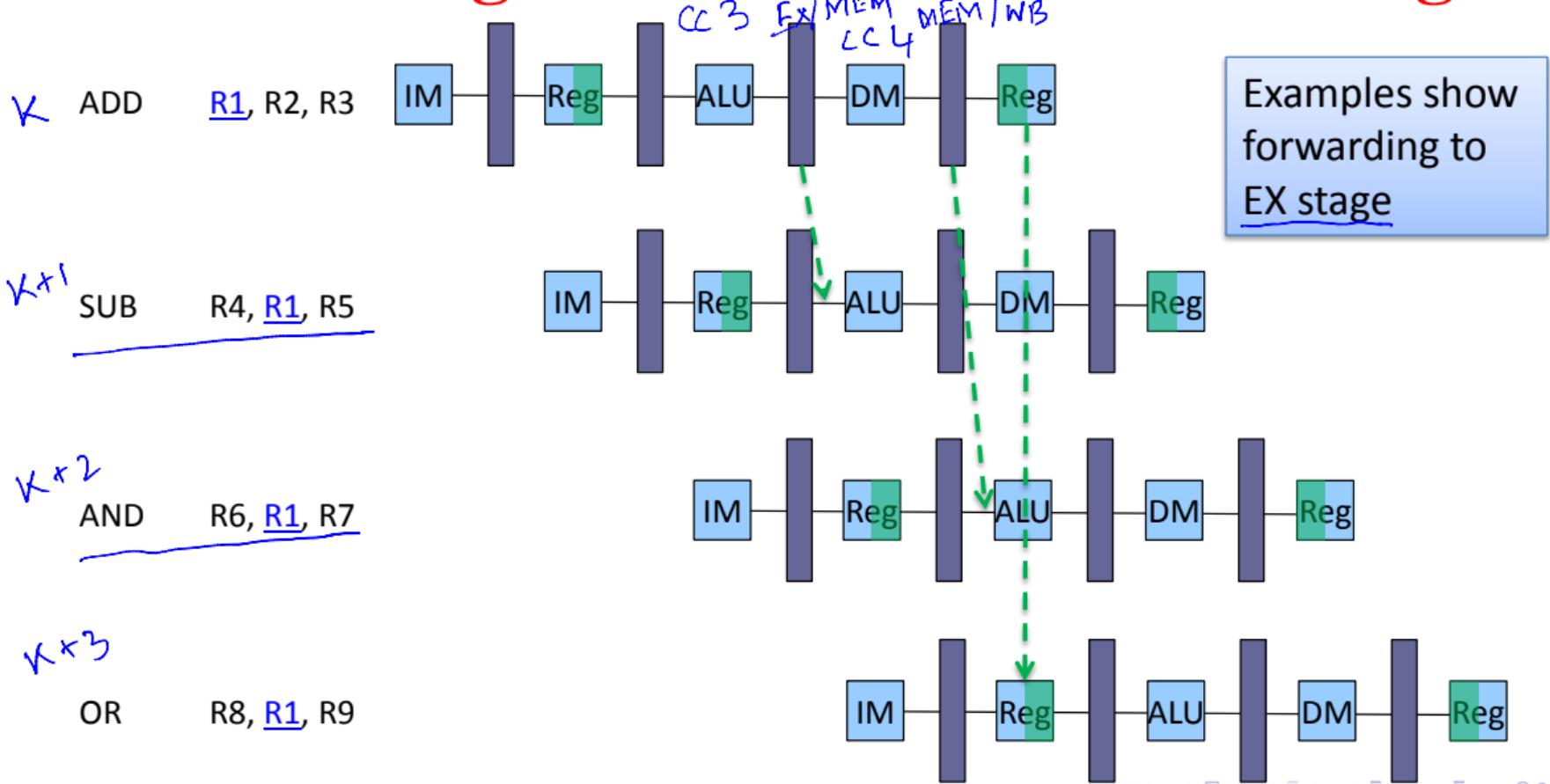


Data hazard: normal pipelined execution will produce **WRONG** result!
Solution? Can stall. Can we do better?

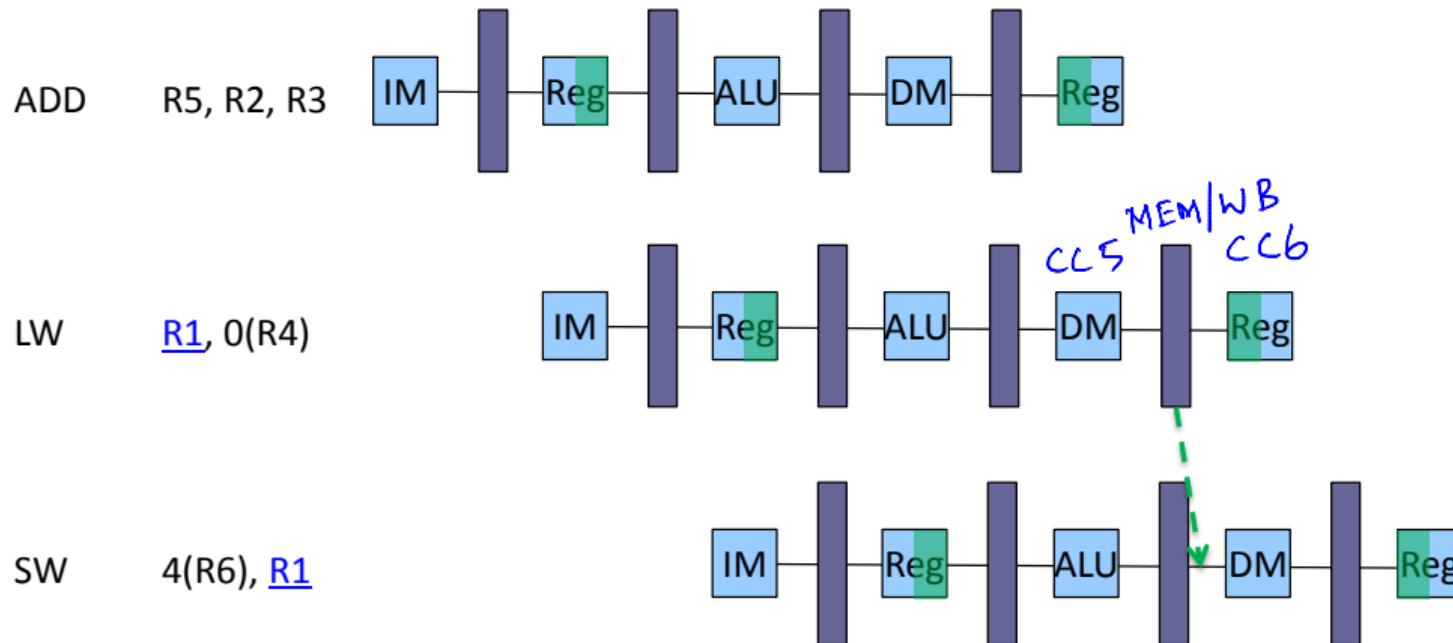
Register File: Reads after Writes



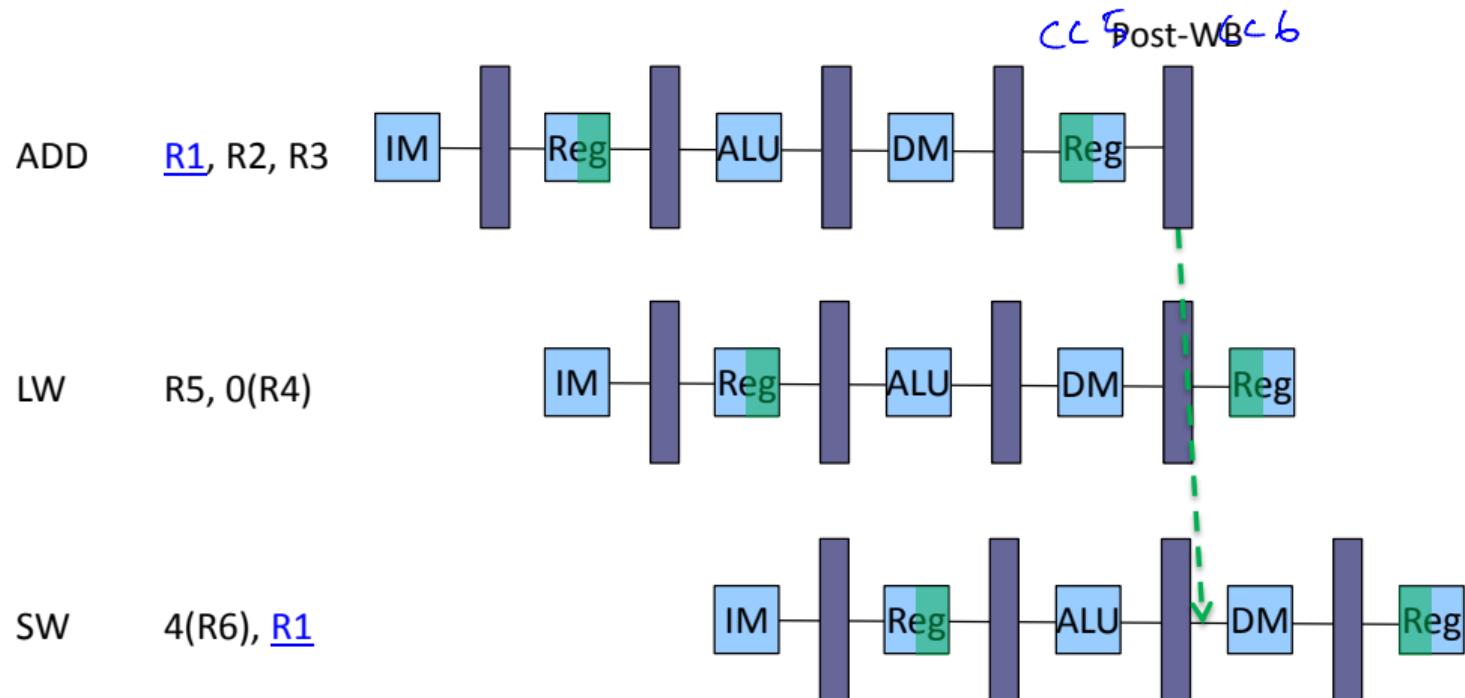
Minimizing Stalls via Data Forwarding



Data Forwarding to MEM Stage



Data Forwarding to MEM Stage (continued)



Memory Data Hazards

- Not possible in MIPS
 - Memory operations always happen in order

$\rightarrow SW \quad R1, \cancel{X}$
 $\rightarrow LW \quad R2, \cancel{X}$

Data Hazard Classification

- **Read after Write (RAW):** use data forwarding to overcome
- **Write after Write (WAW):** arises only when writes can happen in different pipeline stages

	CC1	CC2	CC3	CC4	CC5	CC6
LW <u>R1</u> , 0 (<u>R2</u>)	IF	ID	EX	MEM1	MEM2	WB
ADD <u>R1</u> , R2, R3		IF	ID	EX	WB	

– Has other problems as well: structural hazards

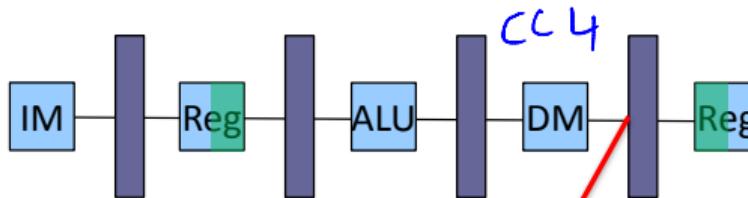
- **Write after Read (WAR):** rare

	CC1	CC2	CC3	CC4	CC5	CC6
SW R1, 0 (<u>R2</u>)	IF	ID	EX	MEM1	MEM2	WB
ADD <u>R2</u> , R4, R3		IF	ID	EX	WB	

Data Dependence Requiring a Stall

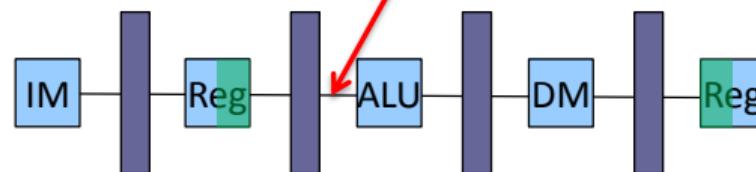
LW

R1, 0(R4)



ADD

R2, R1, R3



All data forwarding, stalling require control logic
(reason for complexity in control)

Compiler's Role in Avoiding Stalls

```
a = b + c;  
d = e + f;
```

Naïve compiler

```
LW R1, b  
LW R2, c  
ADD R4, R1, R2  
SW a, R4  
LW R10, e  
LW R11, f  
ADD R12, R10, R11  
SW d, R12
```

Clever compiler

```
LW R1, b  
LW R2, c  
LW R10, e  
ADD R4, R1, R2  
LW R11, f  
SW a, R4  
ADD R12, R10, R11  
SW d, R12
```

CPI without clever code scheduling = $1 + F_{\text{loads-causing-stalls}}$

Summary

- Data hazard: data dependence
 - MIPS: only RAW dependence in registers
 - Still, can have stall
- Need control logic for:
 - Data forwarding
 - Stalling

CS305

Computer Architecture

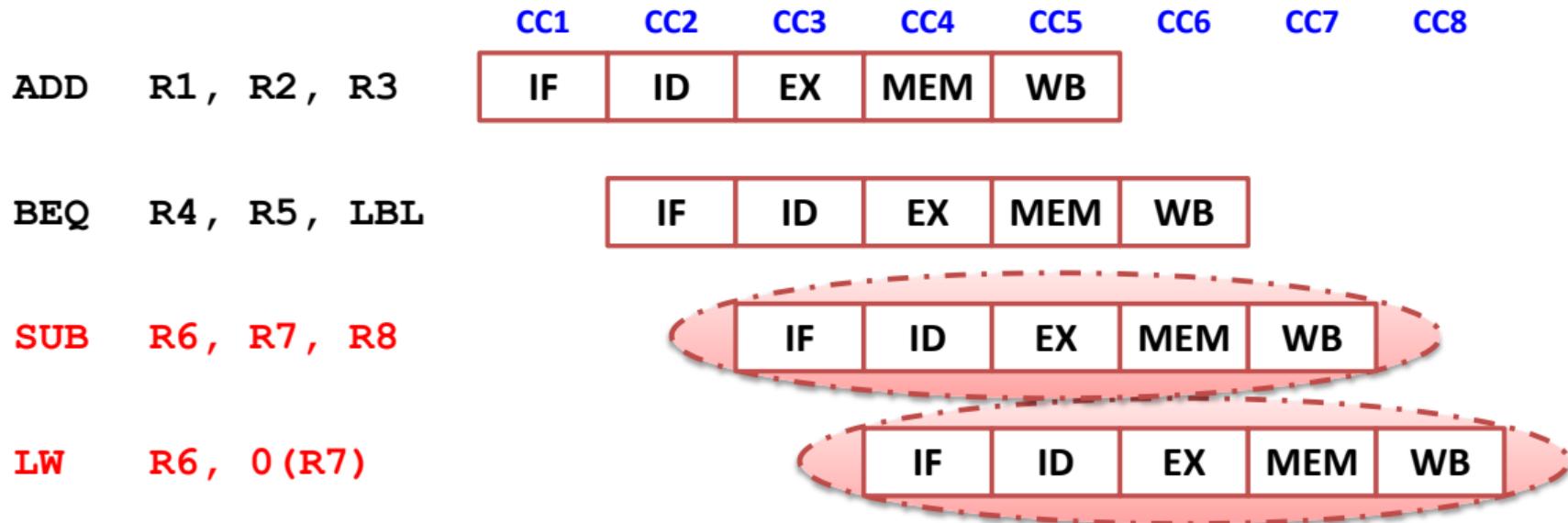
Control Hazards in the Pipeline

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

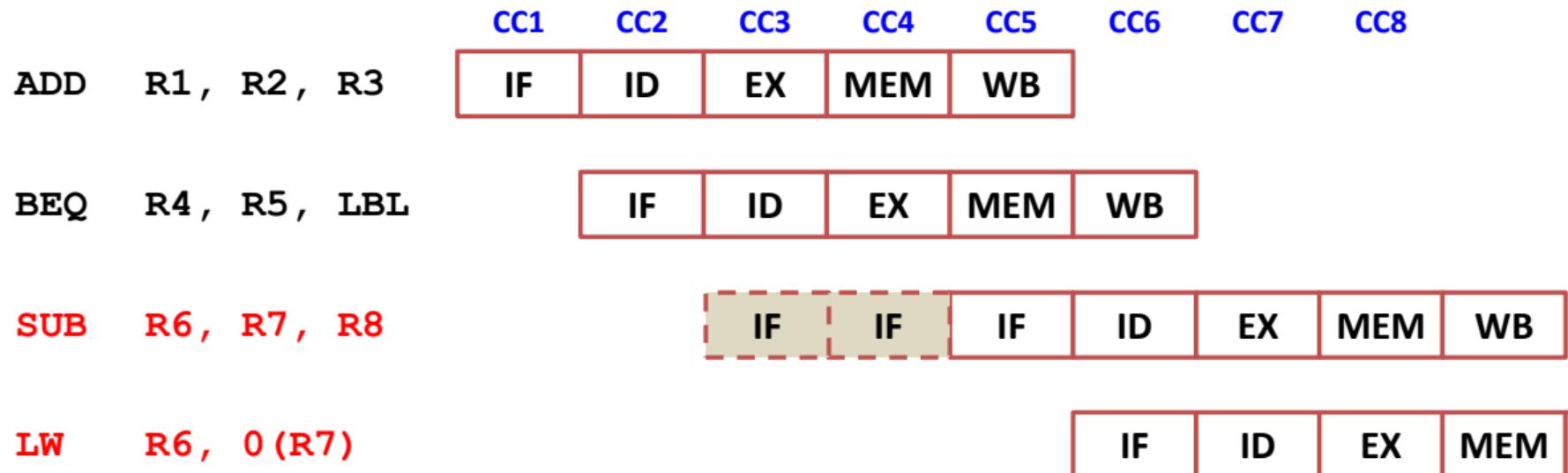
Control Hazards

- Control hazard: pipeline cannot operate normally due to ((possibility of) non-sequential) control flow



Execution of SUB and LW should depend on branch condition: known only in CC4

Stalling “Solution” for Control Hazard



Q: Suppose 10% instructions are branches, what is the performance implication?

A: CPI will increase from 1 to 1.2 due to control stalls: **EXPENSIVE**

Techniques to Reduce Branch Penalty

- 2-stage branch completion
- Assume branch not taken
- Branch prediction
- Delayed branches
- Many advanced techniques (not in this course):
 - Correlating predictors
 - Branch target buffer
 - Special instructions for branch delay slot

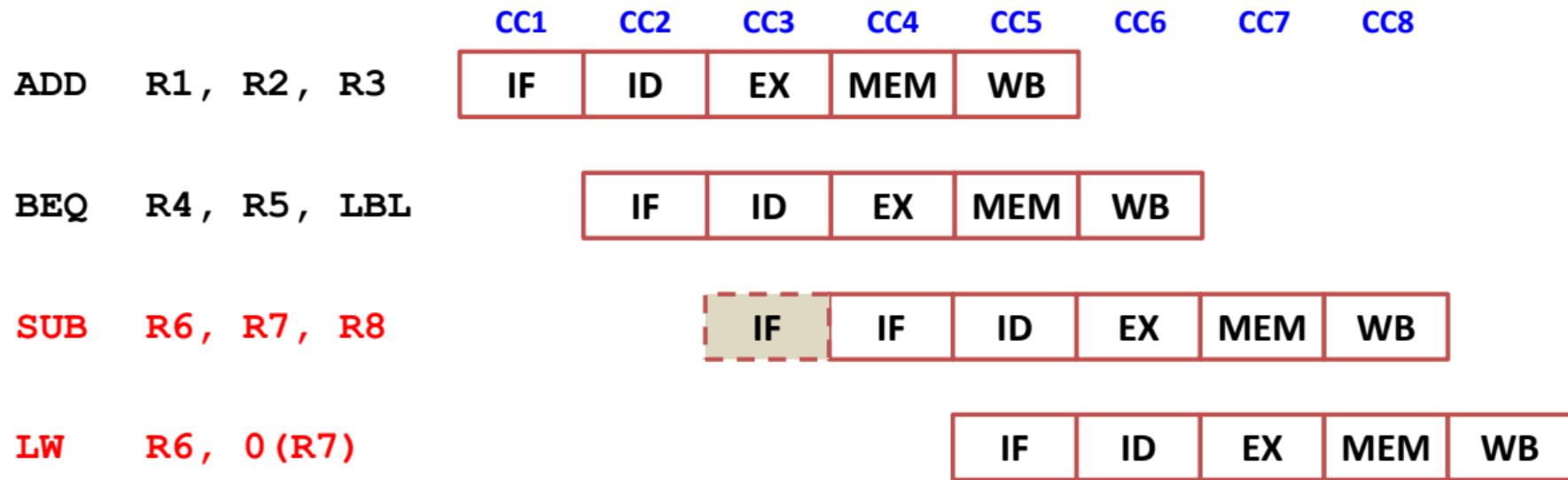
Techniques to Reduce Branch Penalty

- 2-stage branch completion
- Assume branch not taken
- Branch prediction
- Delayed branches
- Many advanced techniques (not in this course):
 - Correlating predictors
 - Branch target buffer
 - Special instructions for branch delay slot

Reducing Stalls: 2-Stage Branch Completion

- Extra hardware required:
 - Comparator in stage-2
 - Needs to complete in half a cycle!
- Data hazard implications:
 - Extra forwarding required: forwarding to ID stage, for branch instruction
 - **beq** itself may stall due to dependence on earlier instruction!

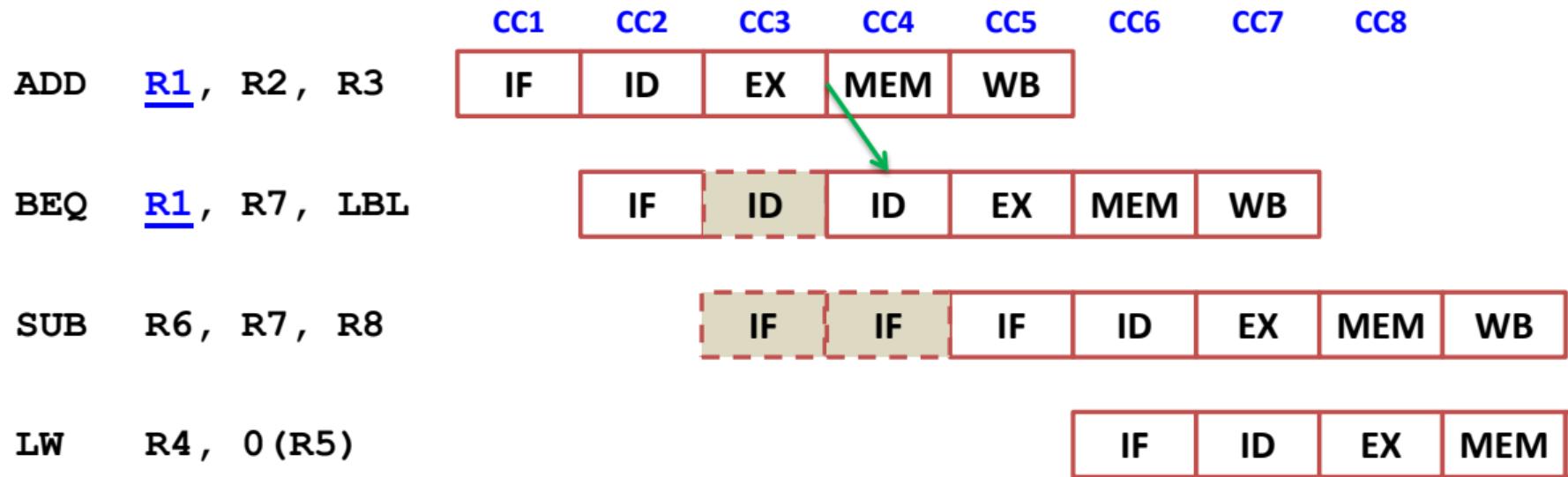
Implication-1 of 2-Stage beq completion



Q: Suppose 10% instructions are branches, what is the performance implication?

A: CPI will increase from 1 to 1.1 due to control stalls: **still not so good**

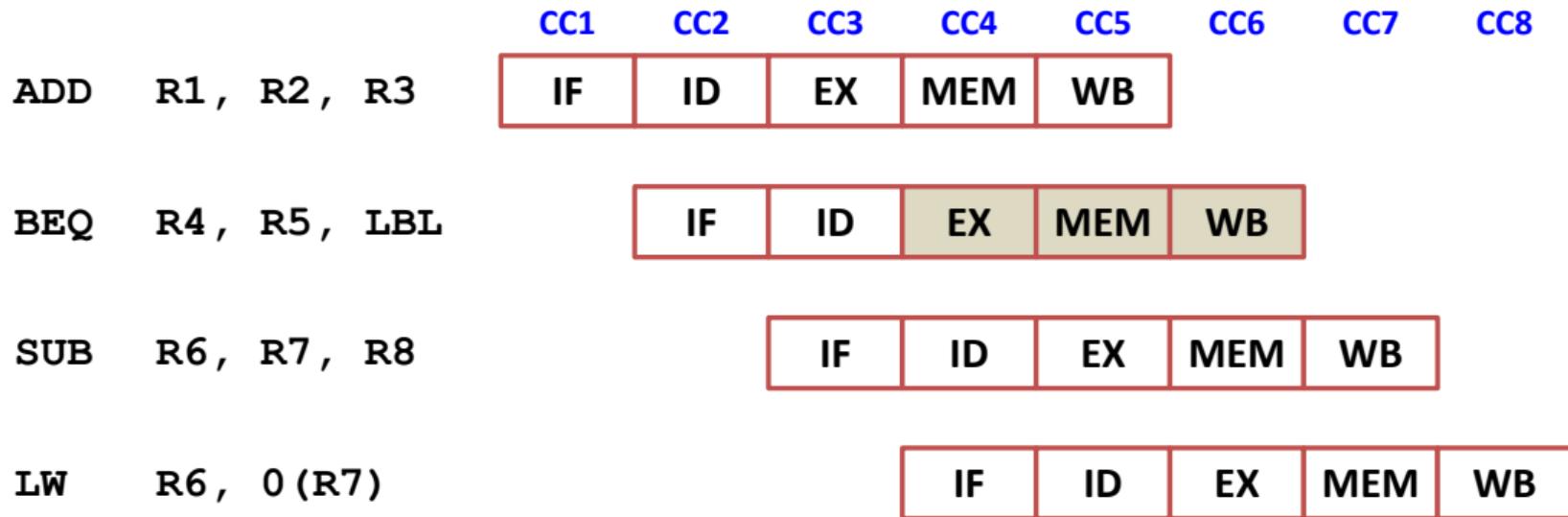
Implications-2,3 of 2-Stage beq completion



Techniques to Reduce Branch Penalty

- 2-stage branch completion
- Assume branch not taken
- Branch prediction
- Delayed branches
- Many advanced techniques (not in this course):
 - Correlating predictors
 - Branch target buffer
 - Special instructions for branch delay slot

Reducing Stalls: Assume Branch Not Taken



In CC4, cancel out the instruction in ID stage, if branch taken

Does not help much for loops: branch taken in most cases

Price paid in this approach: increased control complexity

Techniques to Reduce Branch Penalty

- 2-stage branch completion
- Assume branch not taken
- **Branch prediction**
- Delayed branches
- Many advanced techniques (not in this course):
 - Correlating predictors
 - Branch target buffer
 - Special instructions for branch delay slot

Reducing Stalls: Branch Prediction

- Idea: remember whether branch was taken last time

Last few bits of PC	Prediction (taken=1, not taken=0)
...	1
...	0

- Only last few bits of PC needed: need to deal with branch mis-prediction anyway
- Single-bit predictor: loops are mis-predicted twice
- 2-bit predictor: predict based on last two bits

Techniques to Reduce Branch Penalty

- 2-stage branch completion
- Assume branch not taken
- Branch prediction
- **Delayed branches**
- Many advanced techniques (not in this course):
 - Correlating predictors
 - Branch target buffer
 - Special instructions for branch delay slot

Reducing Branch Penalty: Delayed Branch, Branch Delay Slot

- Change semantic of branch in the ISA
 - Instruction after branch WILL BE executed, even if branch is taken
 - Such an instruction is said to be in the branch delay slot

	1	2	3	4	5
ADD	IF	ID	EX	MEM	WB
BEQ	IF	ID	EX	MEM	WB
SUB	IF	ID	EX	MEM	WB

Q: Who will fill the branch delay slot?

A: Compiler has to do it: yet another important role for the compiler

Filling the Branch Delay Slot

```
AND R1, R2, R3  
SLL R5, R4, 2  
OR  R2, R3, R4  
BEQ R2, R10, LBL
```

```
AND R1, R2, R3  
OR  R2, R3, R4  
BEQ R2, R10, LBL
```

```
ADDI R5, R2, 4  
SUB R6, R2, R7  
...  
LBL:  
ADDI R2, R5, 4  
ADD R6, R5, R2
```

```
AND R1, R2, R3  
OR  R2, R3, R4  
BEQ R2, R10, LBL
```

```
ADDI R5, R2, 4  
SUB R6, R2, R5  
...  
LBL:  
ADDI R2, R5, 4  
ADD R6, R5, R7
```

From before branch

From branch fall through From branch target

- Desirable to fill delay slot from before branch: why? A: always executed
- It may not always be possible to fill delay slot: fill nop

Techniques to Reduce Branch Penalty

- 2-stage branch completion
- Assume branch not taken
- Branch prediction
- Delayed branches
- Many advanced techniques (not in this course):
 - Correlating predictors
 - Branch target buffer
 - Special instructions for branch delay slot

Q: Which of these techniques are complementary? That is, can be used together with one another?

Summary

- Control hazard: non-sequential control flow in program causes disruption in normal pipeline
- Stalling is expensive
- A variety of solutions to reduce branch penalty
 - 2-stage branch
 - Branch prediction
 - Delay slot: filled with compiler's help
 - Many, many advanced techniques!

CS305

Computer Architecture

Pipeline Control

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Controlling the Pipeline

- Without hazards: simple ☺
- Principle: carry your work with you!

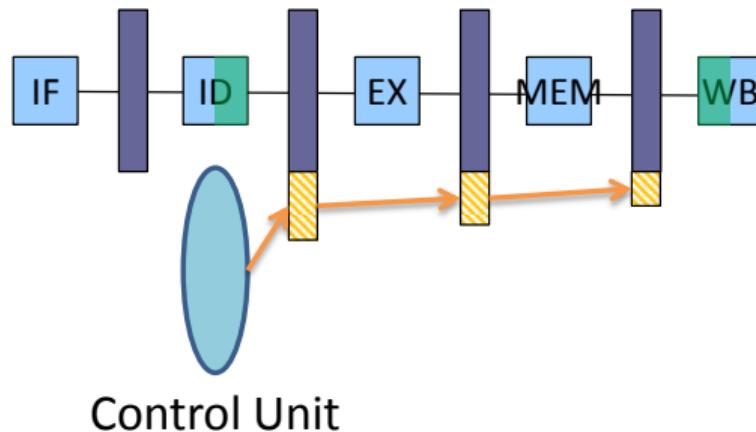
Carry Your Work With You

Single cycle control lines:

ALUSrc, ALUIP4: for EX stage branch

MemRd, MemWr: for MEM stage

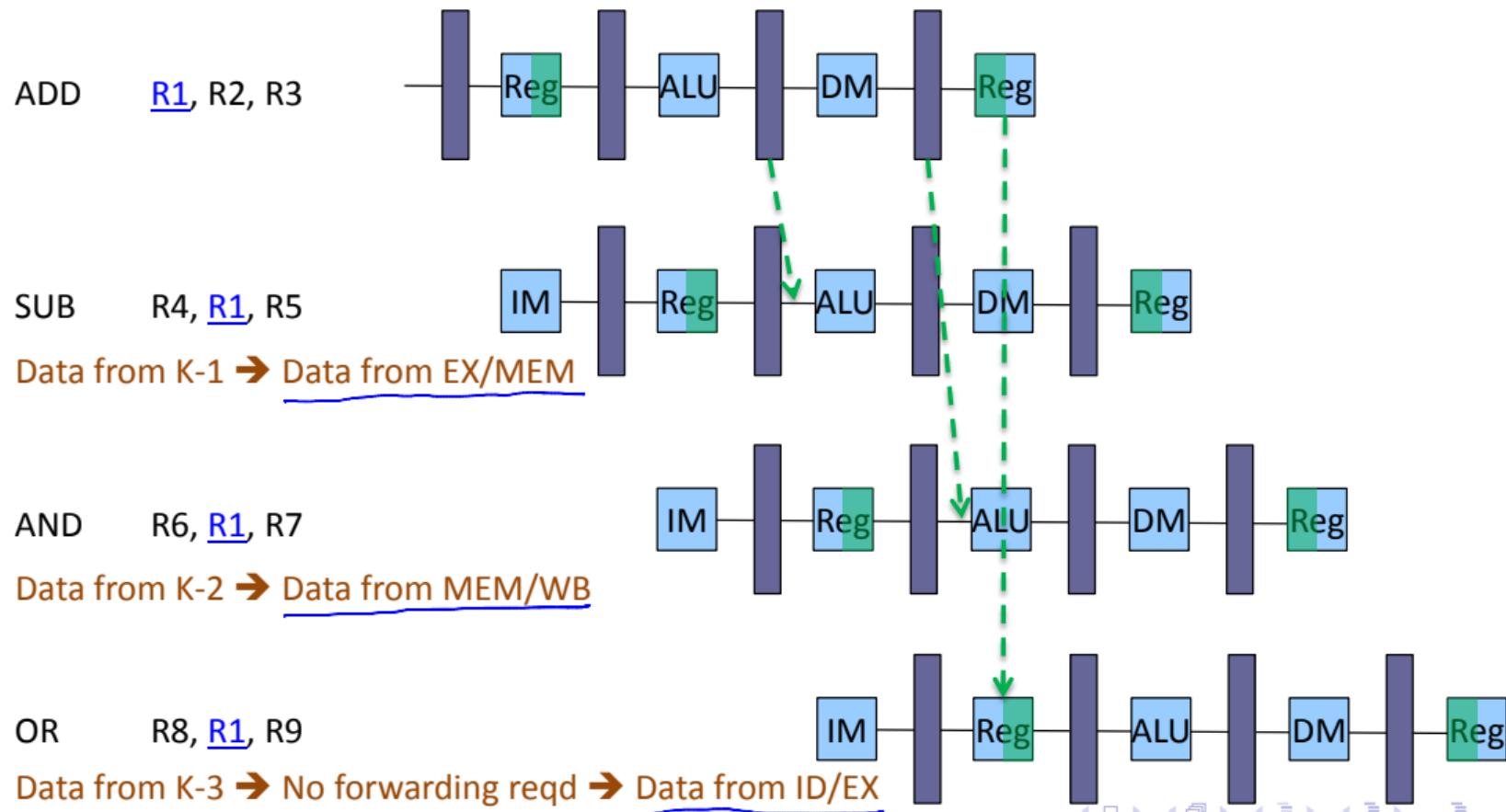
RegWr, RegDst, Mem2Reg: for WB stage



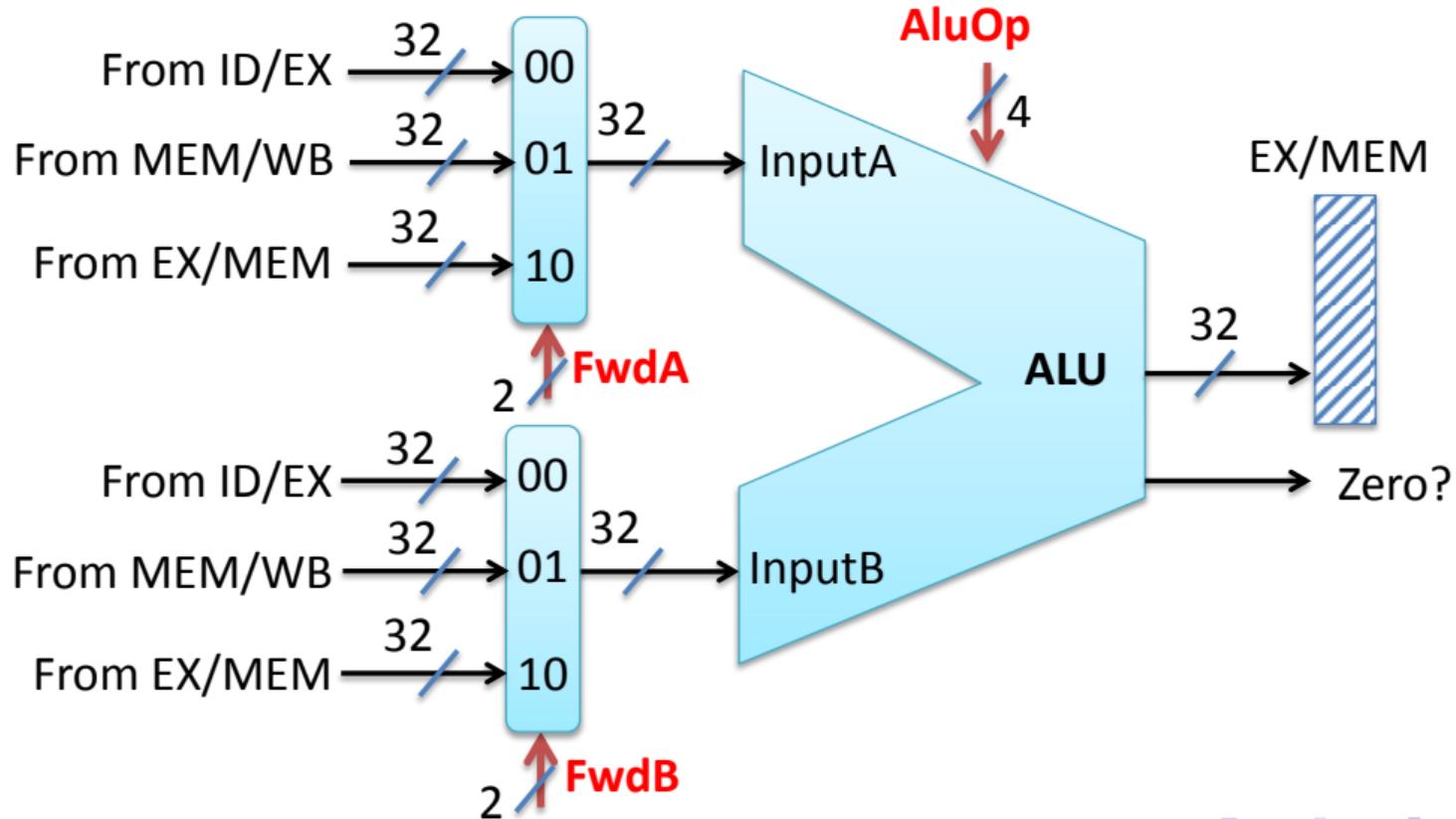
Pipeline Control to Handle Hazards

- Single-cycle control: combinational circuit
- Multi-cycle control: state machine
- Pipeline control: micro-code (Turing machine)
- Recall: micro-code is code executed by a small/simple processor within the main processor (to generate control lines for the pipelined main processor)

Pipeline Control for Data Forwarding to EX



Forwarding to EX: Data-path Changes



Micro-coded Control for Forwarding to EX

Q: What control lines to generate?

A: FwdA, FwdB

Q: When should these be generated?

A: In ID stage of dependent instrn.

Q: What variables to use?

A: Latches, parts of latches

Q: What is the logic to implement?

A: Cases of no fwd, fwd from K-1, K-2

ADD	<u>R1</u> , R2, R3
SUB	<u>R1</u> , R4, R5
SLT	R6, <u>R1</u> , R0

```
// FwdA: micro-coded control  
// Case-1: no forwarding  
FwdA = 00;  
  
// Case-2: fwd from K-1 ←  
if((IF/ID.Rs==ID/EX.Rd) &&  
(ID/EX.RegWr==1) &&  
(ID/EX.MemRd==0)) Swap order  
    FwdA = 10; //frm EX/MEM  
  
// Case-3: fwd from K-2 ←  
if((IF/ID.Rs==EX/MEM.Rd) &&  
(EX/MEM.RegWr==1) &&  
(EX/MEM.MemRd==0))  
    FwdA = 01; //frm MEM/WB
```

Q: Change for FwdB?

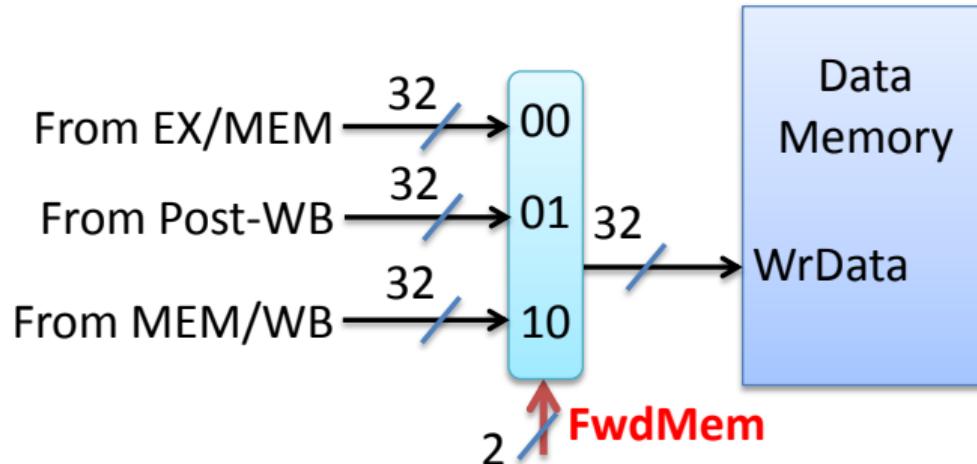
A: Rs → Rt

Why is Pipeline Control Logic Difficult?

- A lot of subtle possibilities → need attention to detail
- Things to keep in mind:
 - Logic runs in ID stage of dependent instruction
 - Actual forwarding happens later
 - Beware of multiple dependences
 - Beware of assumptions
- Exercise: add support for forwarding to EX from **lw**

Forwarding to MEM: Data-path Changes

	CC1	CC2	CC3	CC4	CC5
OR <u>R1</u> , R4, R5	MEM	WB			
SLL <u>R1</u> , R2, 3	EX	MEM	WB		
ADD <u>R1</u> , R2, R3	ID	EX	MEM	WB	
SW <u>R1</u> , 4(R20)	IF	ID	EX	MEM	WB



Forwarding to MEM: Micro-coded Control

```
// FwdMem: micro-coded control
// Case-1: no forwarding
FwdMem = 00;

// Case-2: fwd from K-2
if((IF/ID.Rt==EX/MEM.Rd) && (EX/MEM.RegWr==1) && (EX/MEM.MemRd==0))
    FwdMem = 01; //frm Post-WB
                                         (CtlUnit.MemWr == 1)
                                         (IF/ID.opcode == SW)

// Case-3: fwd from K-1
if((IF/ID.Rt==ID/EX.Rd) && (ID/EX.RegWr==1) && (ID/EX.MemRd==0))
    FwdMem = 10; //frm MEM/WB
                                         (CtlUnit.MemWr == 1)
```

So Far...

- Pipeline control without hazards
- Pipeline control for data forwarding
 - Pseduo-micro-code
- Next: pipeline control for stalling

Stalling Logic: Dependent reg-reg After lw

```
// Stalling: micro-coded control
// Case-1: dependent Rs
if((IF/ID.Rs==ID/EX.Rt) &&
   (ID/EX.MemRd==1))
    STALL // What does this mean?
// Case-2: dependent Rt
if((IF/ID.Rt==ID/EX.Rt) &&
   (ID/EX.MemRd==1))
    STALL // What does this mean?
```

What does STALL mean?

- Do nothing → write nothing
 - **nop** proceeds in the pipeline
 - Zero out control signals, specifically RegWr, MemWr, MemRd
 - No change to machine state
- IF and ID stages must repeat
 - Disable PCWr
 - Disable writing of IF/ID latch

```
// STALL pseudo-micro-code
PCWr = 0;
IF/ID.Wr = 0;
ID/EX latch = 0; // nop (bubble) in pipeline
```

Stalling Logic for Control Hazard

```
if (CtlUnit.branch == 1)
    IF/ID latch = 0; // 1st nop follows branch
if (ID/EX.branch == 1)
    IF/ID latch = 0; // 2nd nop follows branch
```

Q: Changes for 2-stage branch?

A: Second if condition unnecessary

Q: Diff from data hazard stall?

A: nop **FOLLOWS branch**

Summary

- Pipeline control: subtle logic, involving many details
 - Data forwarding logic
 - Control lines generated in ID stage, actual forwarding may happen later
 - Stalling logic: **nop** before stalling instruction, **nop** after branch
- We've seen only bits and pieces
- Very difficult to write control logic without micro-code
- Next: exceptions, the bane of all pipelines

Week7/2 pipeline-exceptions.pdf

CS305

Computer Architecture

Exceptions in the Pipeline

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

How to Handle Exceptions in the Pipeline?

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
SLL R2, R2, 3	IF	ID	EX	MEM	WB			
LW R4, 4(R5)		IF	ID	EX	MEM	WB		
ADD R1, R2, R3			IF	ID	EX	MEM	WB	
SW R4, 4(R20)				IF	ID	EX	MEM	WB

Suppose LW has a misaligned memory address exception

Q: When is the exception detected?

A: End of CC4

Q: What should happen?

A: Flush pipeline, but SLL must be allowed to complete!

Steps in Exception Handling

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
SLL R2, R2, 3	IF	ID	EX	MEM	WB			
LW R4, 4(R5)		IF	ID	EX	MEM	WB		
ADD R1, R2, R3			IF	ID	EX	MEM	WB	
SW R4, 4(R20)				IF	ID	EX	MEM	WB

- Flush → zero out latches
 - IF/ID, ID/EX, EX/MEM; **do not** flush MEM/WB (SLL)
- Load “appropriate” PC value onto EPC
 - PC-8 in this case
 - What if BEQ instead of ADD ?
- Load appropriate value onto Cause register
- PC = 0x8000 0180

Complexity-1: Multiple Exceptions

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
SLL R2, R2, 3	IF	ID	EX	MEM	WB			
LW R4, 4(R5)		IF	ID	EX	MEM	WB		
Invalid opcode			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

Q: Example of multiple exceptions?

A: Invalid opcode following LW

Q: What should happen?

A: Instruction causing earlier exception takes precedence

Complexity arises due to sheer number of such possibilities

Complexity-2: Out-of-Order Completion

	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8
MUL R1, R2, R4	IF	ID	MUL1	MUL2	MUL3	MUL4	MUL5	MEM
ADD R4, R5, R6		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

- MUL exception in CC7
- ADD is done by then!
- Need to “rollback” (Charlie Chaplin in pipeline)
- Solution is well beyond scope of this course

Complexity-3: Partially Changed Machine State

- Classic example: string copy instruction in Intel
- Say, half way through string copy, some other instruction causes exception
- What a mess!

Conclusion

- Exceptions and non-uniformity: enemies of pipelines
- Advanced topics:
 - Out-of-order completion
 - Super-scalar processors: multiple instruction issue per cycle
 - Data driven architectures
- Next: memory systems

Week8/1 one-system-to-know-them-all.pdf

CS305

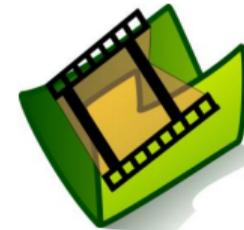
Computer Architecture

One System to Know Them All

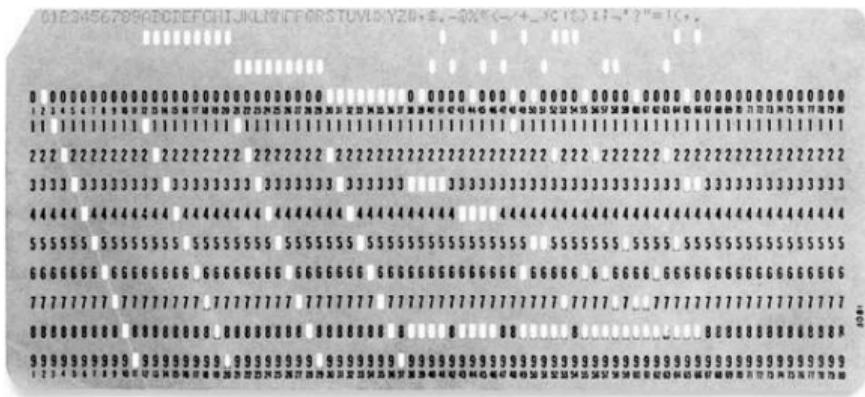
Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

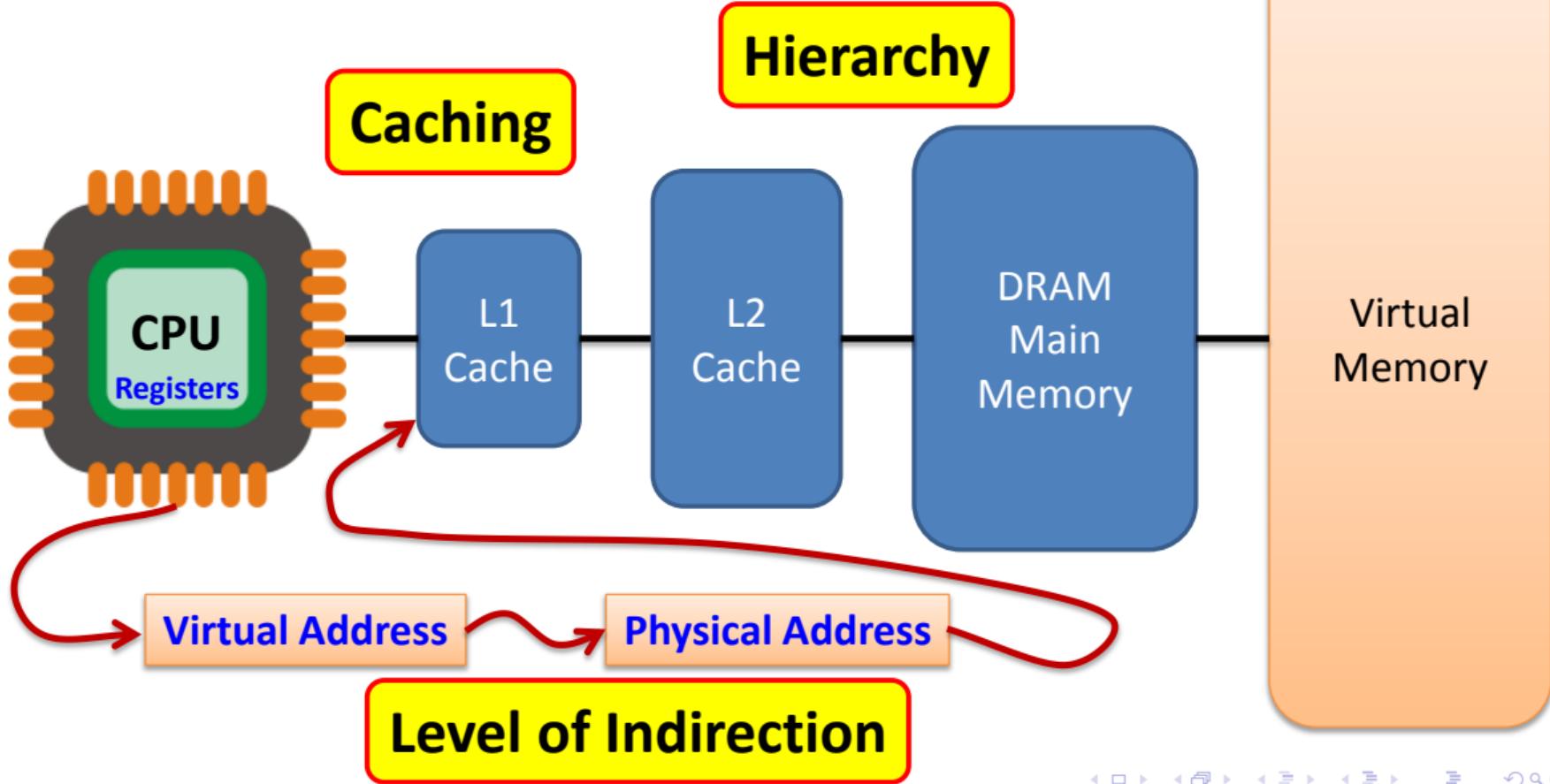
The World is Changed



Much That Once Was...



The Memory System



One System to Know Them All

The world is changed. I feel it in the phones. I feel it in the phablets. I sense it in my apps.

Much that once was is now too slow, and none now live who have use for it.

It all began with the design of the memory system.

The first idea is that of **caching**, for the illusion of memory very vast, yet very fast.

The second is that of **hierarchy**, a great technique to magnify the benefits manifold.

The third is of adding a **level of indirection**, which above all else, provides flexibility.

The three cornerstone ideas of all of Computer Science Systems, they were all of them used, in the design of the memory system. Deep inside every computer, the memory system rules, and dictates the performance of every app you run, and every snap you take. And in this system is steeped, all the three great ideas of Computer Science Systems, their power, their prowess, and their versatility.

One system to know them all.

-- Bhaskaran Raman, 02 Oct 2014

Based on: Galadriel's words in the introduction to the movie "Lord of the Rings: Fellowship of the Ring"

CS305

Computer Architecture

The Memory System: A Hierarchy of Caches

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Memory Systems: Why Important?

- Memory: the second crucial part of a computer
- Today: memory systems dictate performance
 - Processor performance well above memory performance
 - Cannot throw more gates to get faster memory
- Some numbers:
 - Memory access latency: 20+ ns
 - Compare: processor cycle < 1 ns

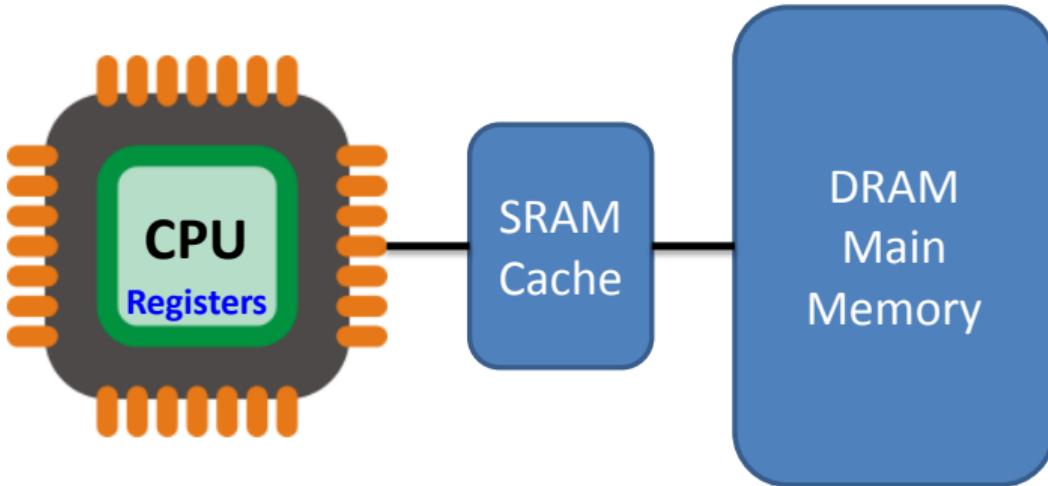
What Programmer Wants vs Reality

- What programmer wants: large memory, fast, cheap
- Reality: large X fast
 - Large memory → slow
 - Large memory → cost per byte is smaller
- Memory system: create **illusion** of large & fast memory
 - Cache memory, main memory, virtual memory, secondary memory (I/O)

What is a Cache?

- **Cache (English):** a safe place to store something
- **Cache (CS):** a temporary place for a copy (usually) of something, for fast, easy, efficient access
- Examples of caching you are aware of ?

Cache in a Computer System



DRAM versus SRAM

Dynamic RAM (DRAM)

- Uses less transistors 1
- Needs to refresh periodically
- More power consumption
- Slower: access latency 20+ ns
- Cycle time can be > access time
 - DRAM needs time to refresh
- Cheaper
- Used for main memory

Static RAM (SRAM)

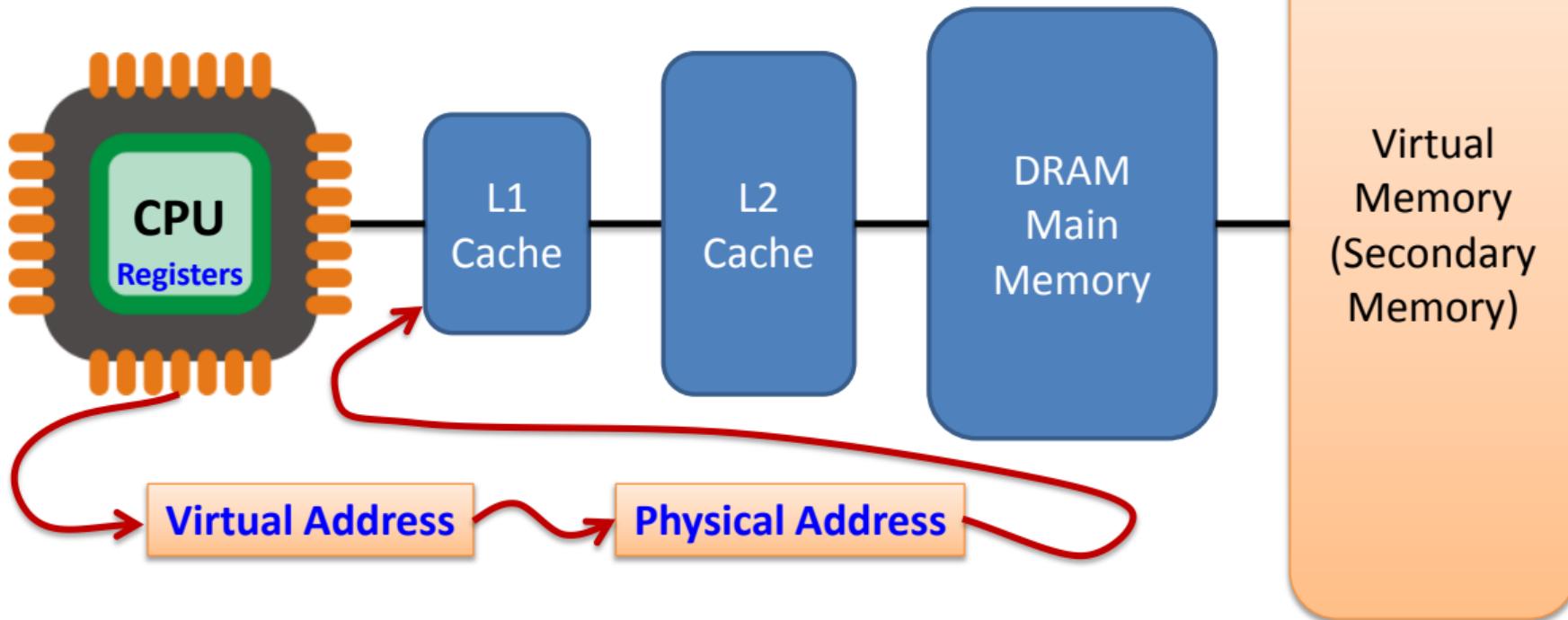
- Uses more transistors 6
- No need to refresh
- Less power consumption ↙
- Faster: access latency ~2 ns
- Cycle time = access time
 - Access one locn. after another
- More expensive
- Used for cache memory

Three reasons why cache is faster: SRAM, smaller, closer to CPU

Why Caches Work: The Principle of Locality

- **Temporal locality:** if X is accessed now, it will likely be accessed again in the near future
- **Spatial locality:** if X is accessed now, locations $X \pm \delta$ will likely be accessed in the near future
- For instructions: sequential execution, loops
- For data: arrays, structures, variables in a function

The Memory Hierarchy



Summary

- Memory system: a hierarchy of caches
- Caching: principle of locality
- Next: cache design

Week8/3 cache-design-a-beginning.pdf

CS305

Computer Architecture

Cache Design: A Beginning

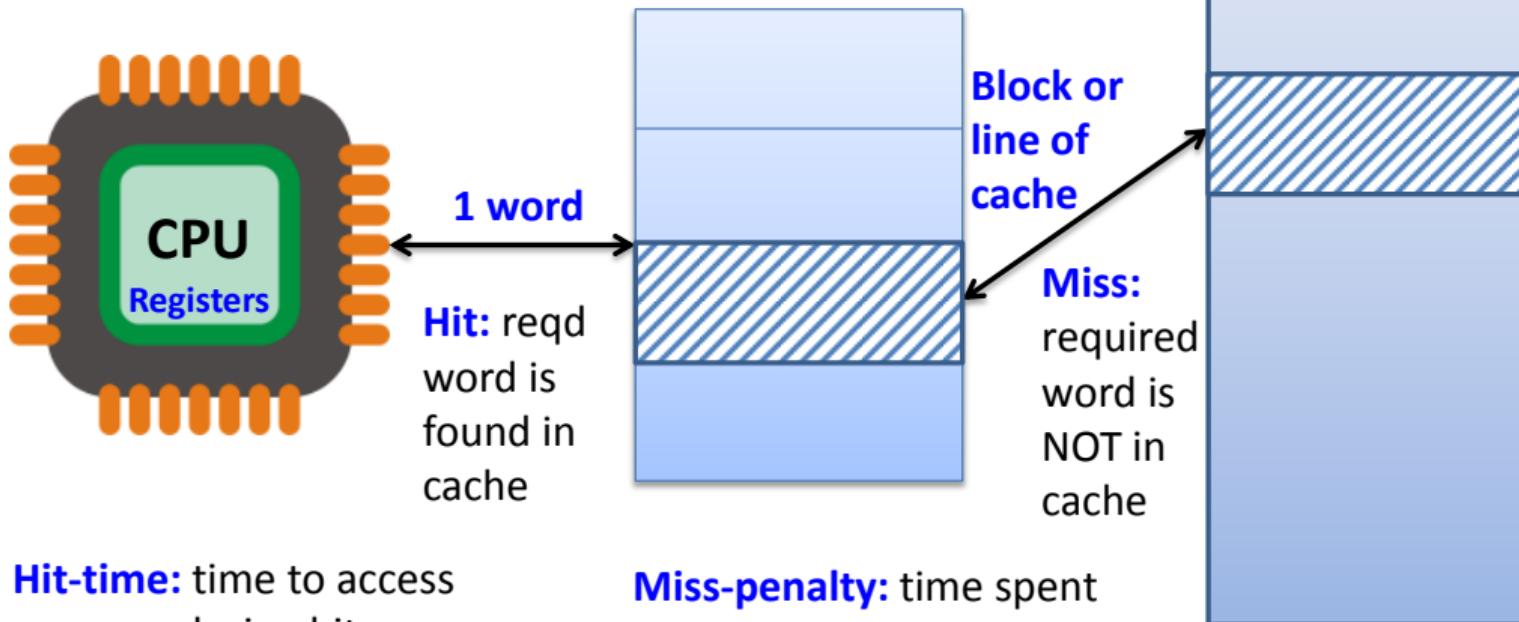
Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Terminology

Hit-ratio/hit-rate: #hits/#accesses

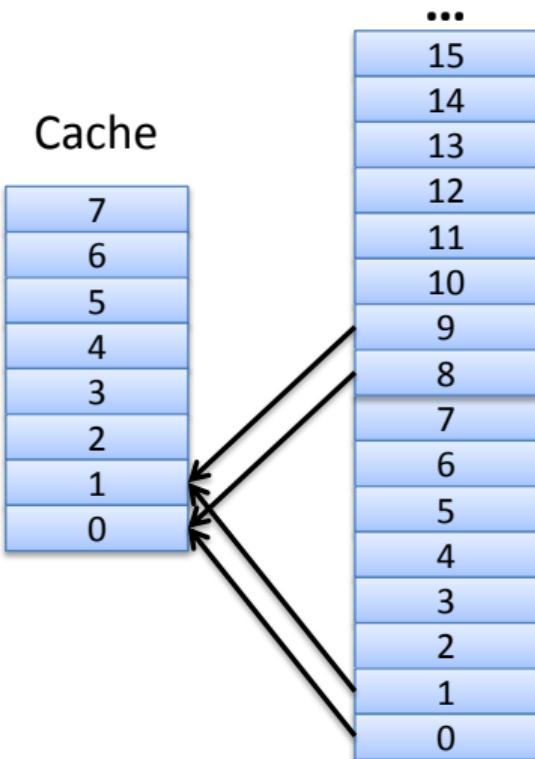
Miss-ratio/miss-rate: #misses/#accesses Cache



Hit-time: time to access memory during hit

Miss-penalty: time spent on a miss (to fetch from main memory)

Simplest Possible Cache Design: Single Word per Block, Direct-Mapped



Single word per block; block size = 1 word

Given memory word,
block # in cache =
(memory block #) MOD (# blocks in cache)

Only one cache location for a given
memory word → **Direct Mapped**

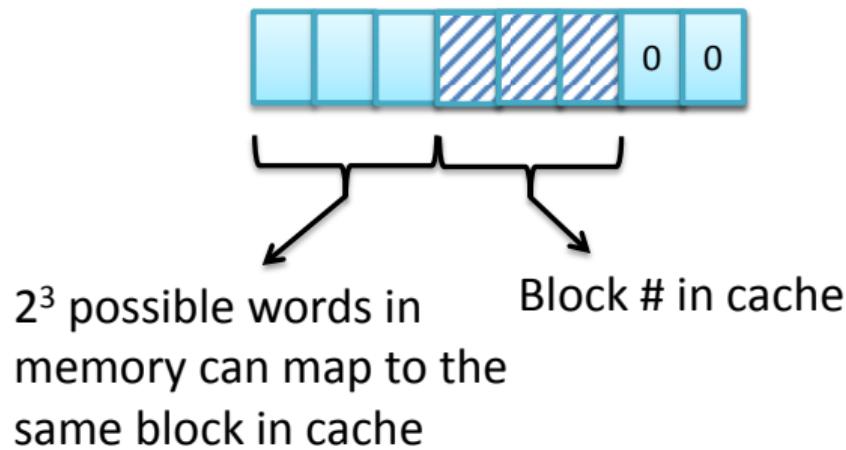
If # blocks in cache is power of 2,
MOD == last few bits of memory word addr

Direct Mapped Cache: A Numeric Example

Memory = 2^6 words, Cache = 2^3 words

Number of bits for memory address = 8

Number of possible states of a cache block = $2^3 + 1$ (can be invalid)



Each cache block, in addition to data (or instruction) has:

- 1 valid bit
- 3 bits of tag (which memory block is in cache currently)

What Happens on a Memory Read? (by Processor)

Given memory address, find cache block:

Cache block # = (Memory block #) MOD (# cache blocks)

Implemented by cache controller

For that cache block:

(a) If valid and cache tag matches tag from memory address)

Cache hit → read from cache

(b) If (invalid or cache tag mismatch)

Cache miss → load appropriate block from memory onto cache, then case (a)

Processor stalls in the meantime!

What happens on memory write? Discussed later...

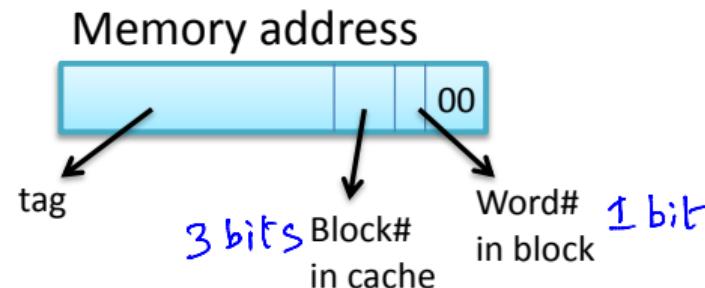
Direct Mapped Cache with Multiple Words Per Block

Cache



To take advantage of spatial locality: block size > 1,
say block size = 2 words

Recall: block is unit of transfer between cache and
memory → on read of any word in a block, entire
block is loaded onto cache

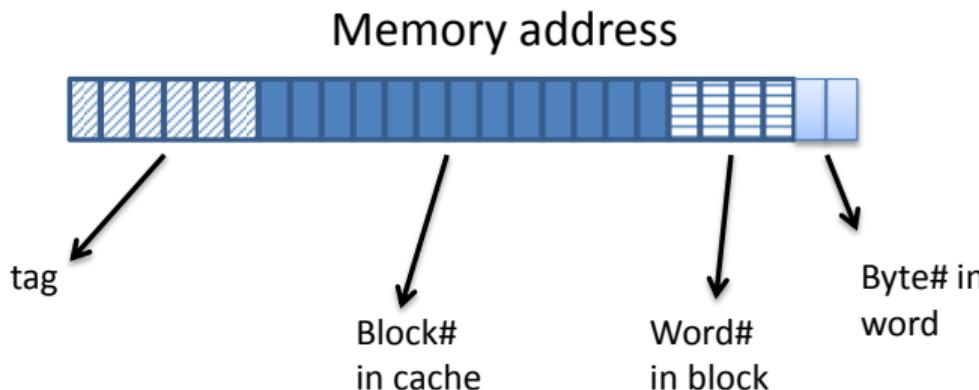


Multiple Words/Block: A Numeric Example

Main memory = 32 MB, Cache = 512 KB,

Block size = 16 words

Draw fields of memory address



$$\frac{512 \text{ KB}}{16 \text{ words}} = \frac{2^{19} \text{ bytes}}{2^6 \text{ bytes}} = 2^{13}$$

bits in cache = $512 \text{ KB} + (2^{13} \times (6+1))$

Cache in Action: An Example

Cache



Block size = 2 words

Cache size = 16 words

Main memory word access sequence:

0, 1, 20, 19, 17, 0

Walk through cache state...

20, 21 in cache 0: Miss
19 in cache 1: Hit
0, 1 in cache x 17, 16 in cache x
 (10~~00~~) 20: Miss
0, 1 in cache (100~~01~~) 19: Miss
 (1000~~11~~) 17: Miss
 0: Miss

Handling Writes: What Happens on Write Hit?

Write-back: write (block) to memory “later”

Need “dirty” bit to know if block needs to be written back, at the time of replacement

Write-through: write (word) to memory now

Can be slow!

Can have a write-buffer to mask delay

Sequence of writes close to one another →
delay will surface

Write Hit: Some Remarks on Performance

In write-back, even write hit can take two cycles!
One to read the tag, one to do the actual write
Can use **store-buffer** to mask latency

Note: can have write-buffer for “write-back” too, to reduce miss penalty
Beware: on read-miss, be sure to check store-buffer and write-buffer as well!

Handling Writes: What Happens on Write Miss?

Write-allocate: fetch block from memory

Write-no-allocate: do not fetch block from memory

Handling Writes: The Four Design Choices

	Write-back	Write-through
Allocate		<i>Possible, makes less sense</i>
No-Allocate	<i>Possible, makes less sense</i>	Write-around

Summary

- Direct mapped: many-to-one mapping
- Memory address =
tag::block#incache::word#inblock::00
- Cache has: block (data), tag, valid bit
- Write policy: four design possibilities

CS305

Computer Architecture

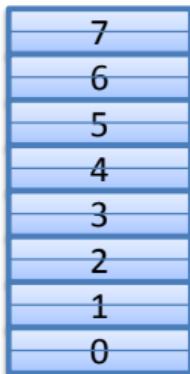
Associative Caches

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Recall Direct Mapped Cache Example

Cache



Block size = 2 words

Cache size = 16 words

Main memory word access sequence:

0, 1, 20, 19, 17, 0

17 evicts 0&1 even though there is plenty of empty space in the cache! **CONFLICT** miss
This causes 0 to miss next. 

Suppose 17 & 0 accessed in a loop: hit-ratio=0 !

Main culprit:

Many-to-one mapping of Direct Mapped cache

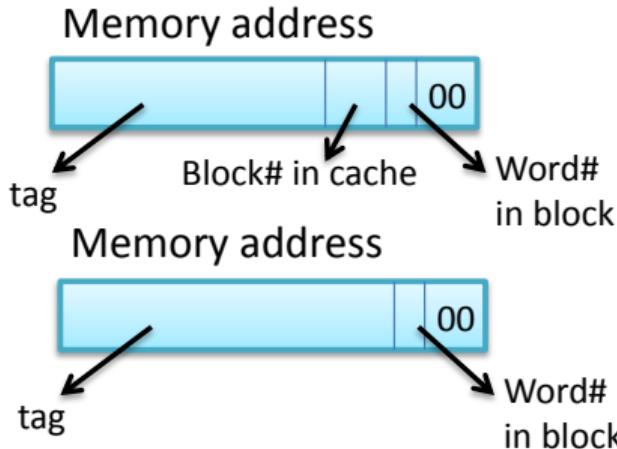
Associative Cache: The Idea

Tag Cache

7	7
6	6
5	5
4	4
3	3
2	2
1	1
0	0

A block of main memory can be in ANY cache block: **fully associative** cache

Main memory block # → Cache block #
Now a many-to-many mapping!



Price to pay: search for block in cache, given memory address → parallel comparator

- (+) Reduced conflict misses
- (-) Comparator cost
- (-) Increased hit time!

Set Associative Cache: The Idea

Direct Mapped

A block of main memory can be in exactly
ONE cache block

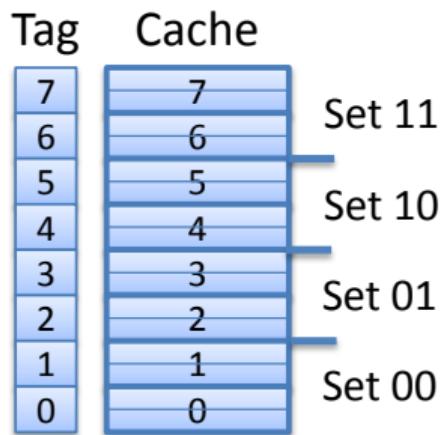
Fully Associative

A block of main memory can be in **ANY**
cache block

Set Associative

A block of main memory can be in **ANY of a
SET** of cache blocks

How a Set Associative Cache Works



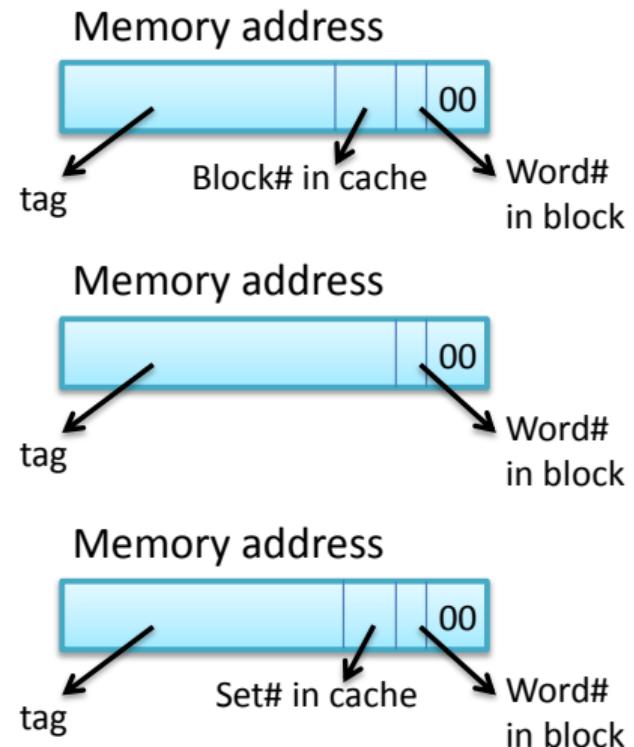
Direct Mapped

Fully Associative

Set Associative

Given memory address, find set#

#parallel comparators = #blocks in set



The Design Space Continuum

Direct Mapped

Set associative cache with set size = 1 block
Called **1-way set associative**

Fully Associative

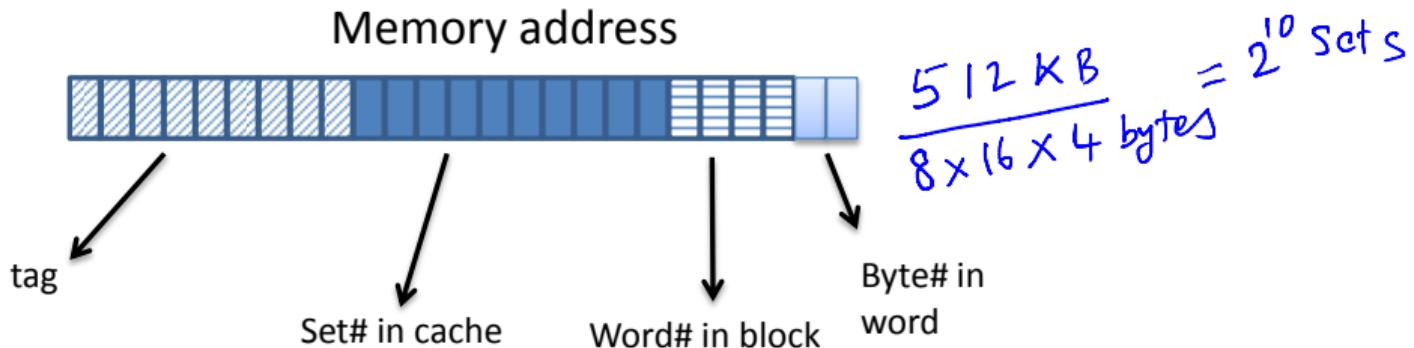
Set associative cache with set size = K
 $K = (\# \text{ blocks in cache})$, **K-way set associative**

Set Associative

Set associative cache with set size S: $1 < S < K$
S-way set associative

Set Associative Cache: A Numeric Example

Main memory = 32MB, Cache size = 512 KB, Block size = 16 words
Show main memory address fields for 8-way set associative cache



Number of comparators required = 8

Number of bits to be compared in each comparator = tag size = 9 bits

Replacement Policy

Tag

Cache

7
6
5
4
3
2
1
0



Set 11
Set 10
Set 01
Set 00

What to do when set is full? Which cache block to replace?

Not relevant for direct mapped cache: no choice

Some possibilities:

- a) Random
- b) First-In-First-Out (FIFO)
- c) Least frequently used (LFU)
- d) Least recently used (LRU)

LRU found most effective in practice: difficult to implement in hardware

What is implemented: **clock algorithm**, an approximation of LRU, works well

More comprehensive treatment of replacement algorithms: in OS course

Summary

- Associativity: helps reduce cache conflict misses
- Fully associative: too expensive (cost, hit time)
- Set associative: get benefits of fully associative and direct mapped
- Block replacement policy: LRU
- Next: facets of cache performance

CS305

Computer Architecture

Cache Performance Analysis

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Outline

- The three kinds of cache misses
- Performance implications of block size
- Joint I+D cache vs separate I, D caches
- Performance implications of associativity
- Design of cache \leftrightarrow main memory interface
- Multi-level caches

The Three Kinds of Cache Misses

Compulsory

The miss caused the first time a block is accessed
Since the cache starts “cold”, “empty”...

Conflict

A miss caused due to insufficient set size
Cannot happen in a fully associative cache (by defn.)

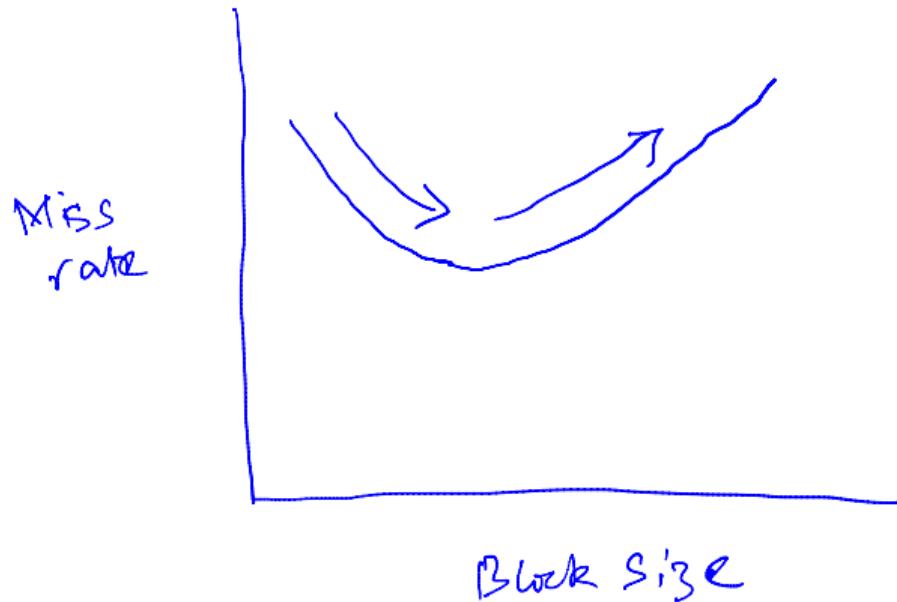
Capacity

A miss caused due to insufficient cache size
All misses after compulsory misses, in fully assoc. caches

Effect of Increase in Block Size

- (+) Increased spatial locality
 - Lesser compulsory misses
- (-) More conflict misses: for same cache size
- (-) Higher miss penalty

Miss Rate vs Block Size



Increased Block Size: Techniques to Reduce Miss Penalty

- Early restart
 - Processor can proceed as soon as required word is in cache
- Critical word first
 - Get the word required by the processor first, then the rest of the block
- Implications: increased complexity in cache controller and/or memory system

Joint I+D Cache or Separate I, D Caches

Joint I+D Cache

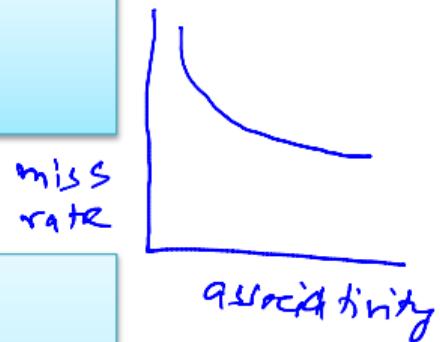
- (+) Better hit rate
- (-) Lower instruction throughput (pipeline stalls)

Separate I, D Caches

- (-) Slightly lower hit rate
- (+) Better instruction throughput

Performance Implications of Associativity

- (+) Reduced conflict misses
- (-) Increased hit time!



Note: decreasing benefits of higher associativity.

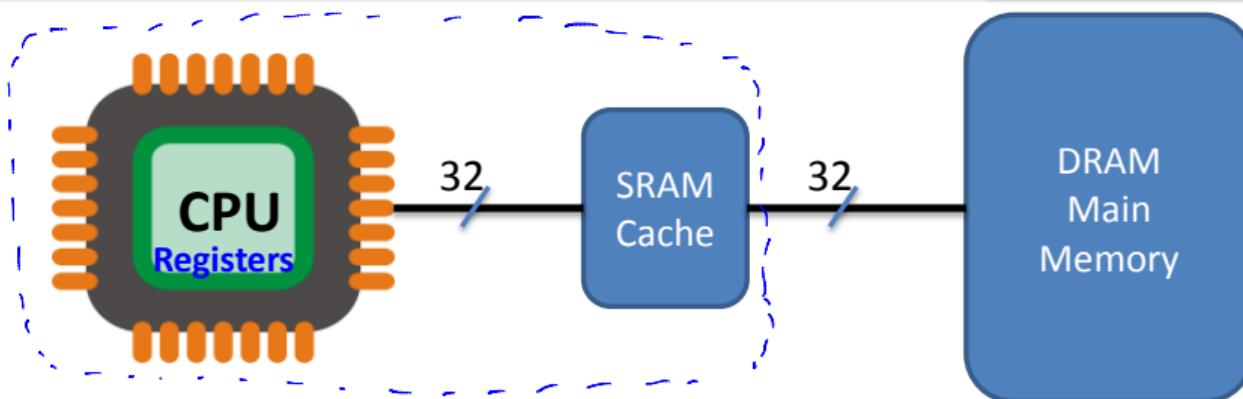
Reason: only a certain number of conflict misses to remove

Design of Cache-to-Main-Memory Interface

What Happens on a Cache Miss?

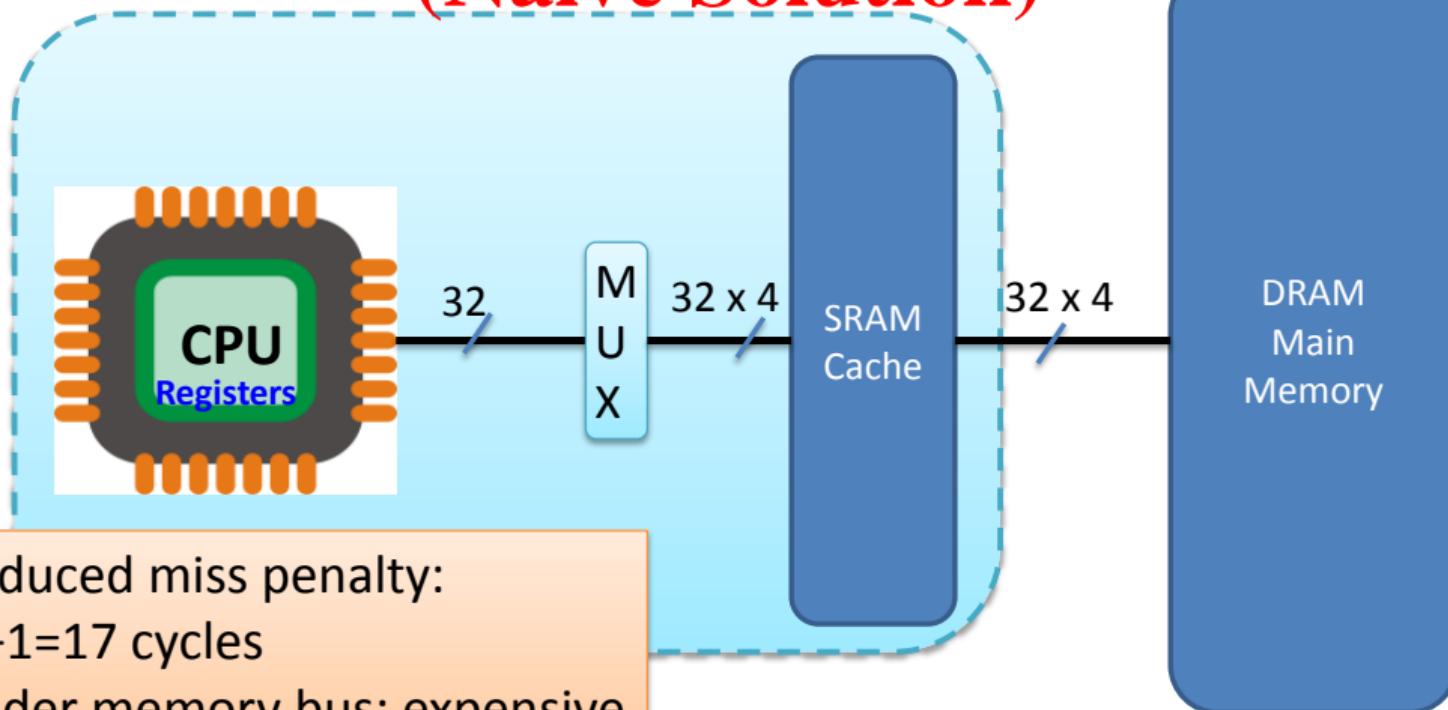
Miss penalty: time to load a block from main memory to cache

- a) Send address to memory → Say 1 cycle
- b) DRAM access initiation latency → Say 15 cycles
- c) Read 1 word of data from memory to cache → Say 1 cycle



Miss penalty for 4-word block = $4 \times (1+15+1) = 68$ cycles

Reducing Miss Penalty: Wide Memory (Naïve Solution)



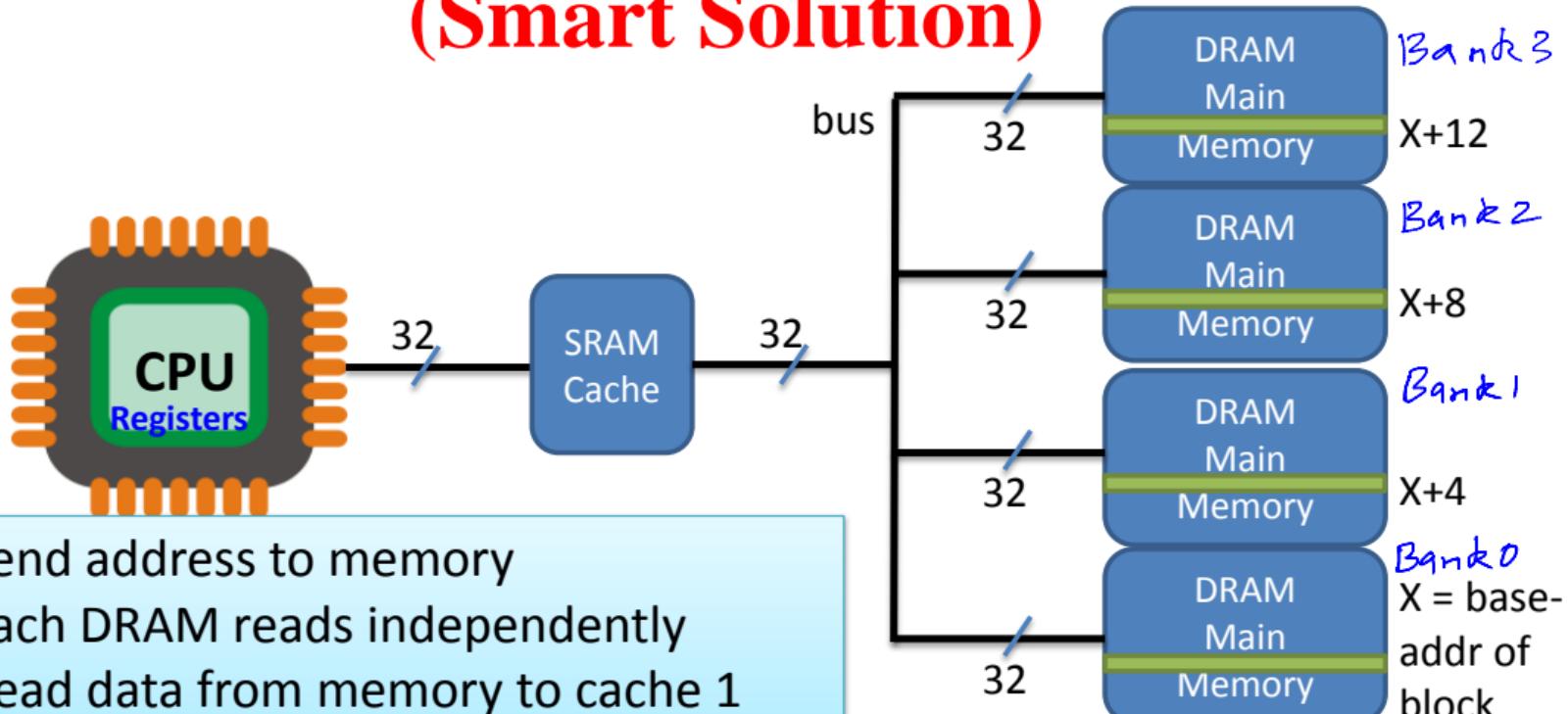
(+) Reduced miss penalty:

$$1+15+1=17 \text{ cycles}$$

(-) Wider memory bus: expensive

(-) Increased hit time!

Reducing Miss Penalty: Interleaved Memory (Smart Solution)

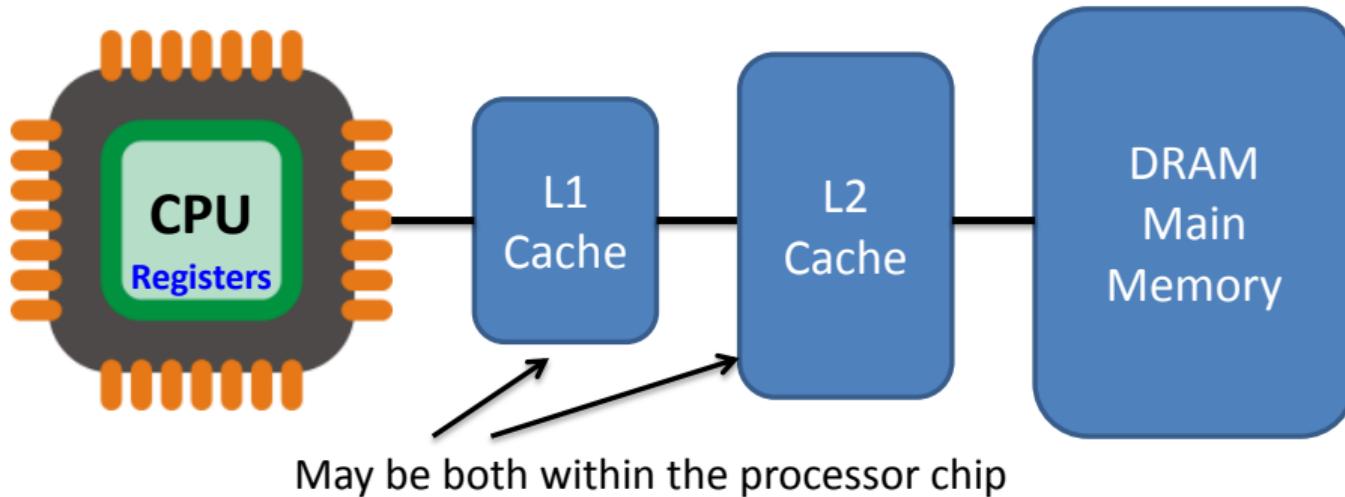


- a) Send address to memory
- b) Each DRAM reads independently
- c) Read data from memory to cache 1 word after another

Miss penalty for 4-word block = $1 + 15 + 4 \times 1 = 20$ cycles

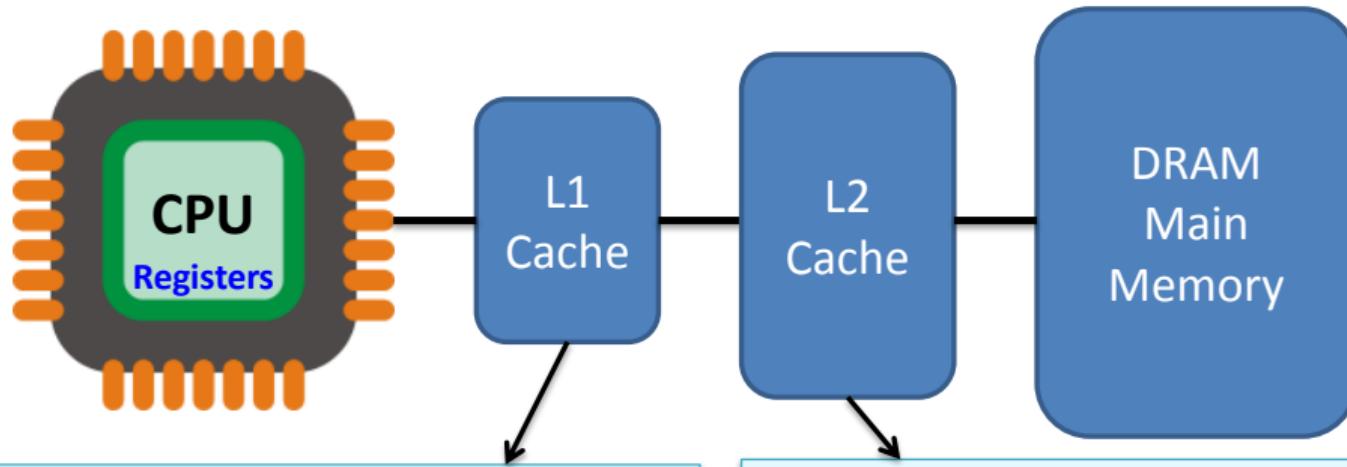
Multi-Level Caches

Reducing Miss Penalty: Multi-Level Caches



L1 thinks L2 to be main memory, L2 thinks L1 is processor
Miss in L1 → see in L2, Miss in L2 → see in main memory

Reducing Miss Penalty: Multi-Level Caches



Optimize for hit-time:
(miss penalty low anyway)
Smaller size, smaller blocks
Direct mapped or low associativity

Optimize for miss-rate:
(hit time does not matter anyway)
Larger size, larger blocks
2, 4, or 8-way associative

L2 Miss Rate: Local vs Global

L2 **local** miss rate: with respect to L2 accesses

L2 **global** miss rate: with respect to memory references by processor

Summary

- The three C's: compulsory, conflict, capacity
- Performance implications of design options: block size, separate vs unified, associativity
- Interleaved memory
- Multi-level caches
- Next: program performance in presence of caches

Week9/2 prog-perf-with-cache.pdf

CS305

Computer Architecture

Program Performance Analysis in Presence of Cache

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Reworking the Performance Equation in Presence of Cache

Cache hit → normal operation

CPU time = CPU time without misses + stalls due to misses

Read stalls: # read misses x read miss penalty

Write stalls: depends on write access scheme

For write-back: read misses will potentially have to write dirty blocks

For write-through: write stalls = # write-buffer stalls

For write-through + write-allocate:

write miss penalty = read miss penalty + % write-buffer stalls

Note: miss rate is given in terms of # memory accesses

Program Performance Analysis in Presence of Cache: An Example

I-cache miss rate: 1%, D-cache miss rate: 5%, % memory instructions = 40%

Miss penalty = 100 cycles

CPI without memory stalls = 3, CPI with memory stalls = ?

CPI with memory stalls = $3 + 1\% \times 100 \text{ cycles} + 40\% \times 5\% \times 100 \text{ cycles} = 6$

Suppose original CPI halved due to better pipelining and data forwarding

New CPI = $1.5 + 3 = 4.5$, effective speedup = $6/4.5 = 4/3$ only, not 2

Suppose CPI further halved (e.g. superscalar architecture), new speedup?

New CPI = $0.75 + 3 = 3.75$, effective speedup = $4.5/3.75 = 6/5$ only, not 2

Program Performance with Multi-Level Caches: An Example

L1 miss rate = 2%, L2 miss rate (global) = 0.5% (both w.r.t. executed instructions)
Miss penalty into L2 = 25 cycles, miss penalty into main memory = 500 cycles
CPI without misses = 2.5, what is the performance improvement due to L2 ?

$$\text{CPI without L2} = 2.5 + 2\% \times 500 = 12.5$$

$$\text{CPI with L2} = 2.5 + 2\% \times 25 + 0.5\% \times 500 = 5.5$$

$$\text{Performance improvement due to L2} = 12.5/5.5$$

Summary

- Program performance with cache: **add to CPI** corresponding to miss rate and miss penalty
- Instance of **Amdahl's law**: improvements to processor almost useless beyond a point
- Octa-core processor versus quad-core processor: what do you think is the performance difference?
- Questions most relevant to program performance: **cache configuration** (cache size, number of levels, on-chip versus off-chip)

Week10/1 virtual-memory.pdf

CS305

Computer Architecture

Virtual Memory

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Programmer's Burden: Program Size > Main Memory Size



Initially Part1, Part2



Then Part1, Part3



Then Part1, Part4

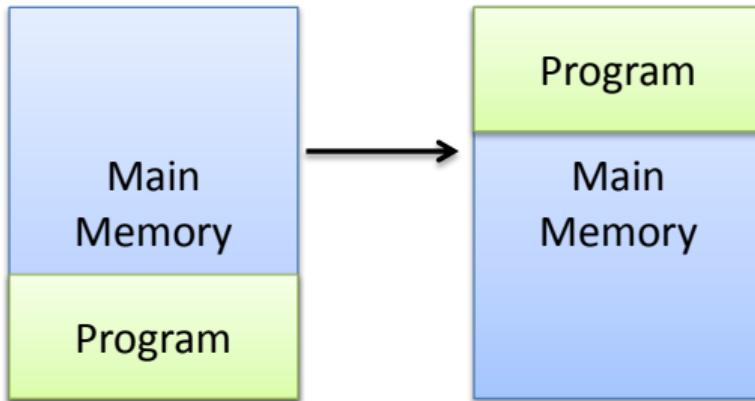


Program had to deal with bringing in relevant parts of the program, into main memory!

Not a problem today: most programs < memory size

Illusion required: memory larger than main memory

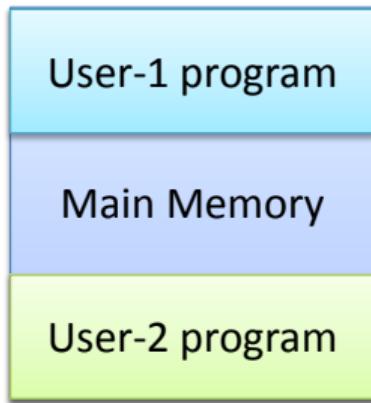
Loader's Burden: Program Relocation



Program, or parts of program need to be constantly rewritten, if loaded to different parts of memory

Illusion required: program at same memory location

OS's Burden: Program Safety



Different user's programs must be protected from one another: malicious or buggy behaviour

Illusion required: each program thinks it has the entire memory to itself, it is the only program

Virtual Memory: Illusions Galore

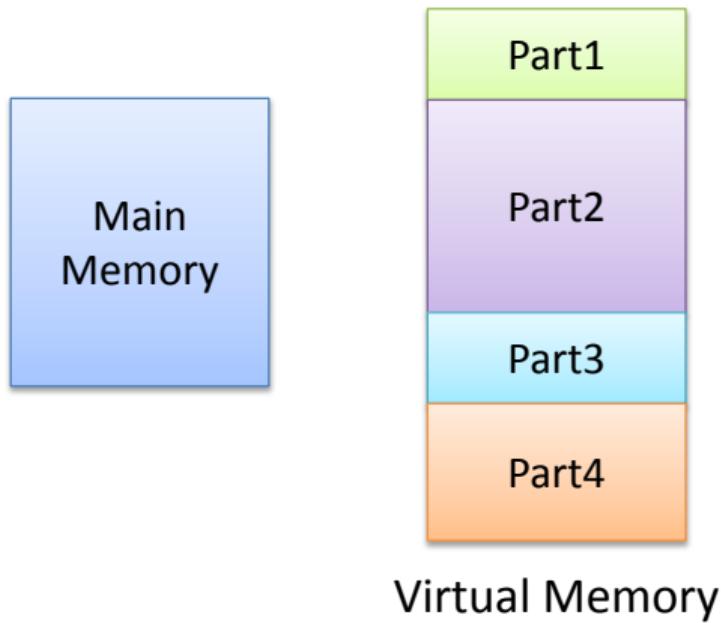
Illusion-1: Larger memory than main memory

Illusion-2: Program at same memory location

Illusion-3: The current program is the only program, with entire memory to itself

Illusion-N... E.g. with shared dynamic libraries, shared memory, etc.

Virtual Memory: The Cache Perspective

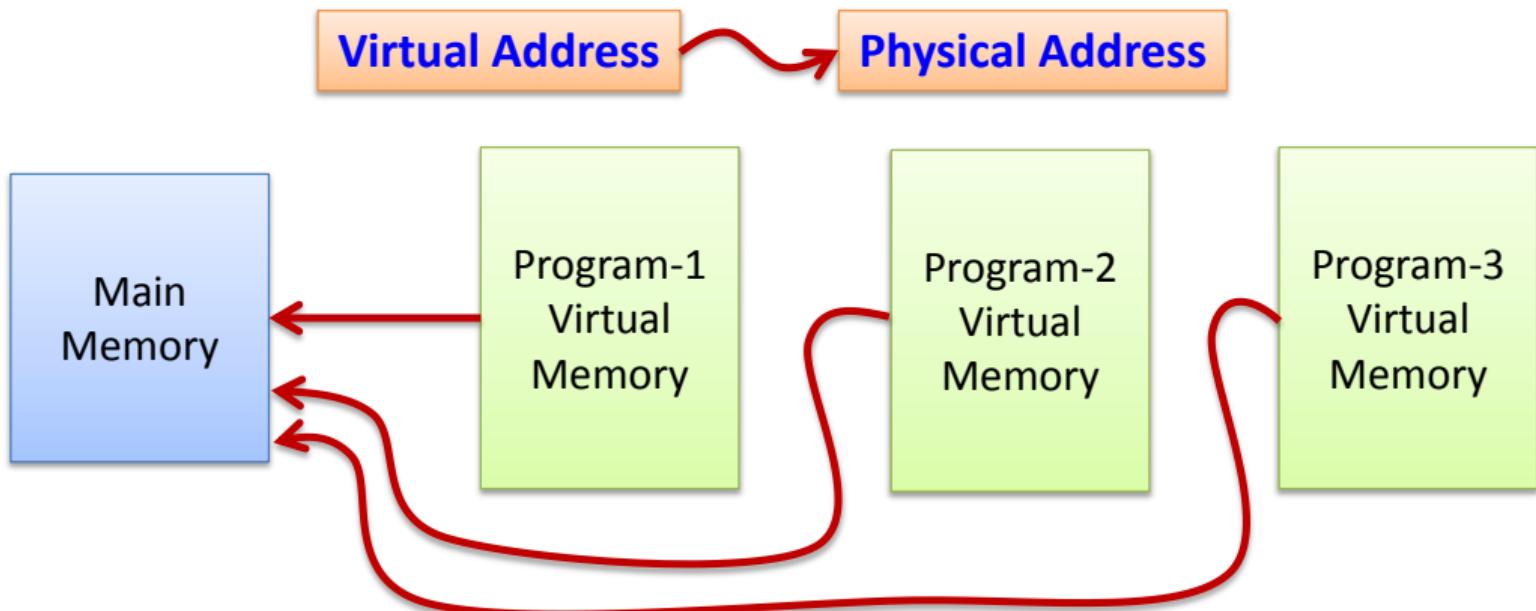


Main memory is a cache for virtual memory

Removes programmer's burden

Incomplete perspective:
e.g. main memory can be 8GB, virtual memory 4GB

Virtual Memory: The Level-of-Indirection Perspective



OS, with hardware support, maintains VA → PA mapping

Level-of-Indirection: The Basis for All Illusions

Virtual address (VA) → physical address (PA) mapping:

Like main memory address → cache location mapping

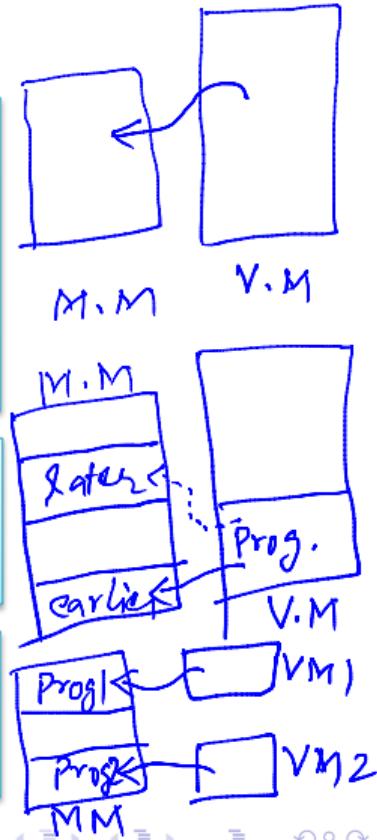
Can support much larger virtual address space than
physical (main) memory

Program relocation: change VA → PA mapping!

Program (in terms of VA) remains unchanged

Program safety: separate VA space for each program

No chance of inadvertent access by another program



The Cache Perspective

L1 as Cache for MM

Unit of transfer = block

Not in cache → miss

Mapping: using MM addr, tags

Miss penalty: a few 100 cycles

MM as Cache for VM

Unit of transfer = page

Not in cache → page fault

Mapping: using page table, other schemes

Miss penalty: to disk → a few ms!
Millions or billions of cycles!

VM Design Decisions in the Cache Perspective

Strategy-1: Reduce miss rate

Page size: large (e.g. 32KB)

Associativity: full

Write-back scheme

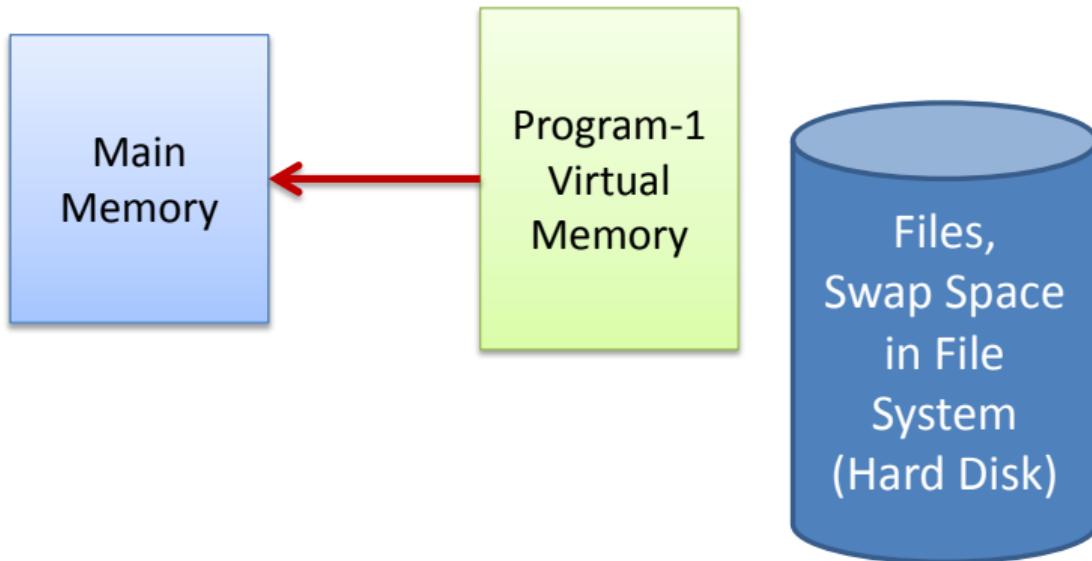
Write-allocate scheme

Page fault: handled in software (OS), sophisticated replacement policy

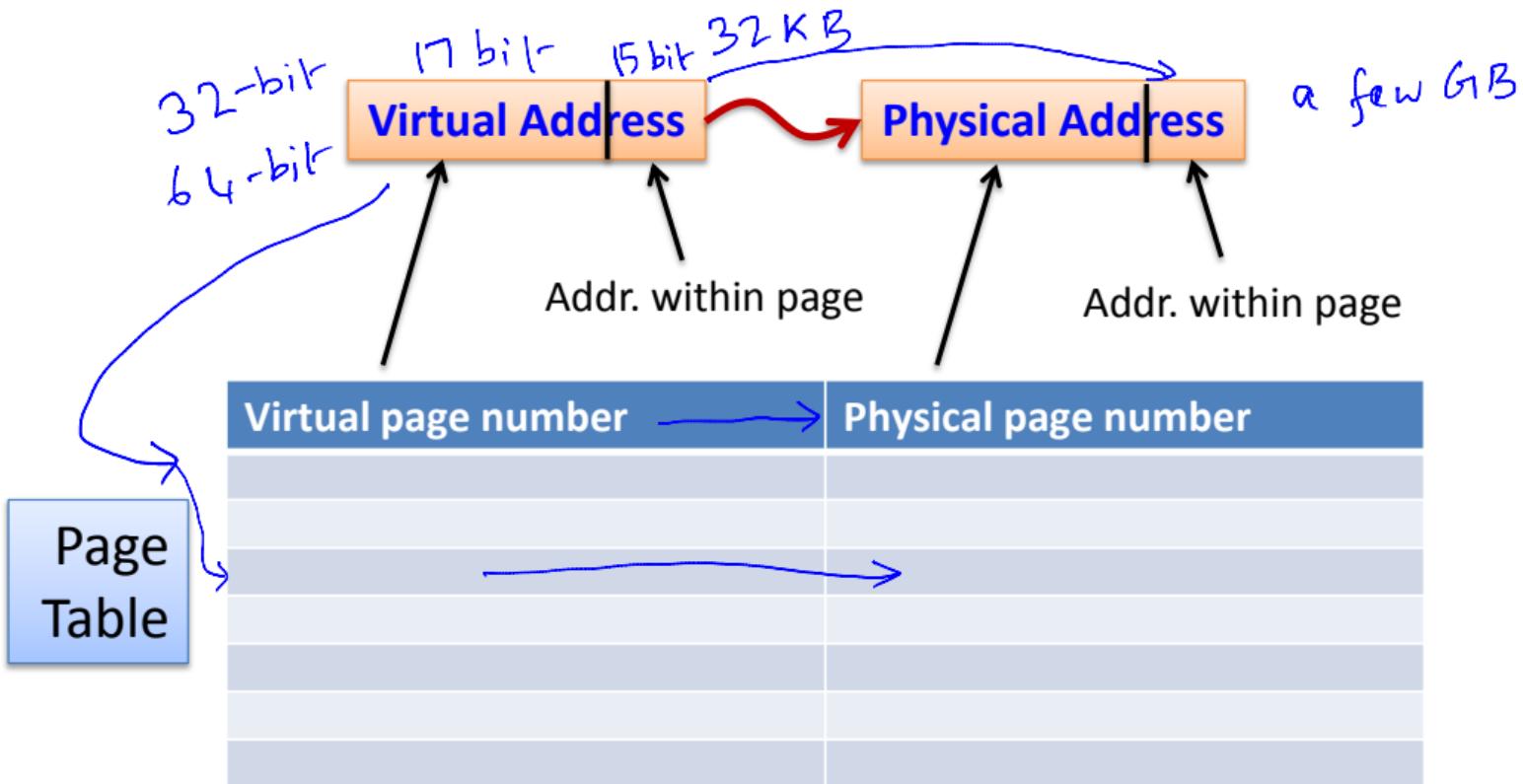
Strategy-2: Multi-tasking

When a program has a miss (page fault), run some other program!

The Swap Space



The VA-to-PA Mapping: Page Table



Page Table: A Numeric Example

32-bit VA space, 8KB pages

Say, 4 bytes per page table entry

Page table size = ?

pages in VM = $2^{32}/2^{13} = 2^{19}$

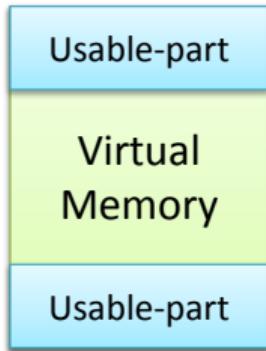
Page table size = 2^{21} bytes = 2MB !

This is per process !

64-bit VA space → page table much larger!

What can we do?

Dealing with Large Mapping Size: Solution-1: Page Table Limits



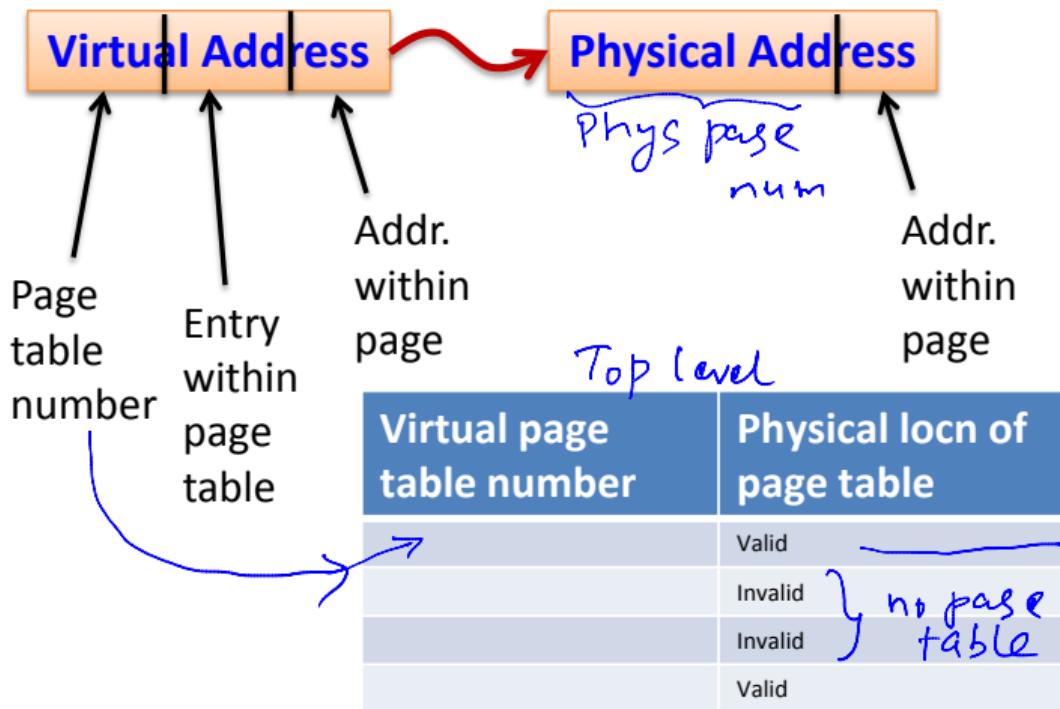
Maintain mapping only for usable parts of virtual memory

Dealing with Large Mapping Size: Solution-2: Inverted Page Table

Virtual page number	Physical page number

- One entry per physical page
 - Entries hashed on virtual page number

Dealing with Large Mapping Size: Solution-3: Multi-level Page Table

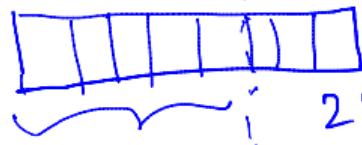


- Main idea: hierarchy
- Idea works since most entries in first level page table will be invalid

Page tables

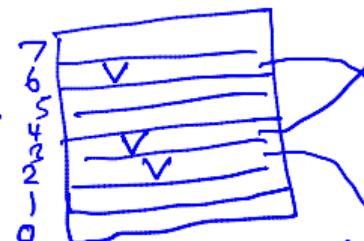
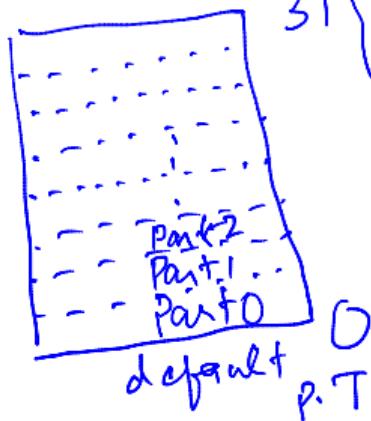
Page table entry number	Physical page number

Multi-Level Page Table: An Example

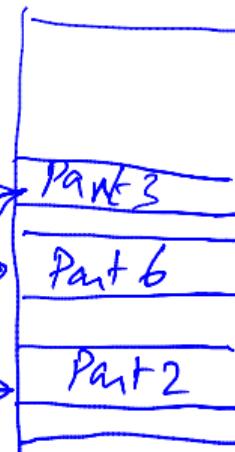


3 + 2

256
8 = 32 pages



First level
P.T



Second level
P.T

M.M

Dealing with Large Mapping Size: Solution-4: Paged Page Tables

Virtual page number	Physical page number

- Main ideas: level of indirection, caching
 - Earlier: page table was in physical memory
 - Now: page table in virtual memory (of a special OS process)
 - Can manage even if page table size is large
 - Page table of OS process pinned to physical memory

Paged Page Tables: An Example

P1

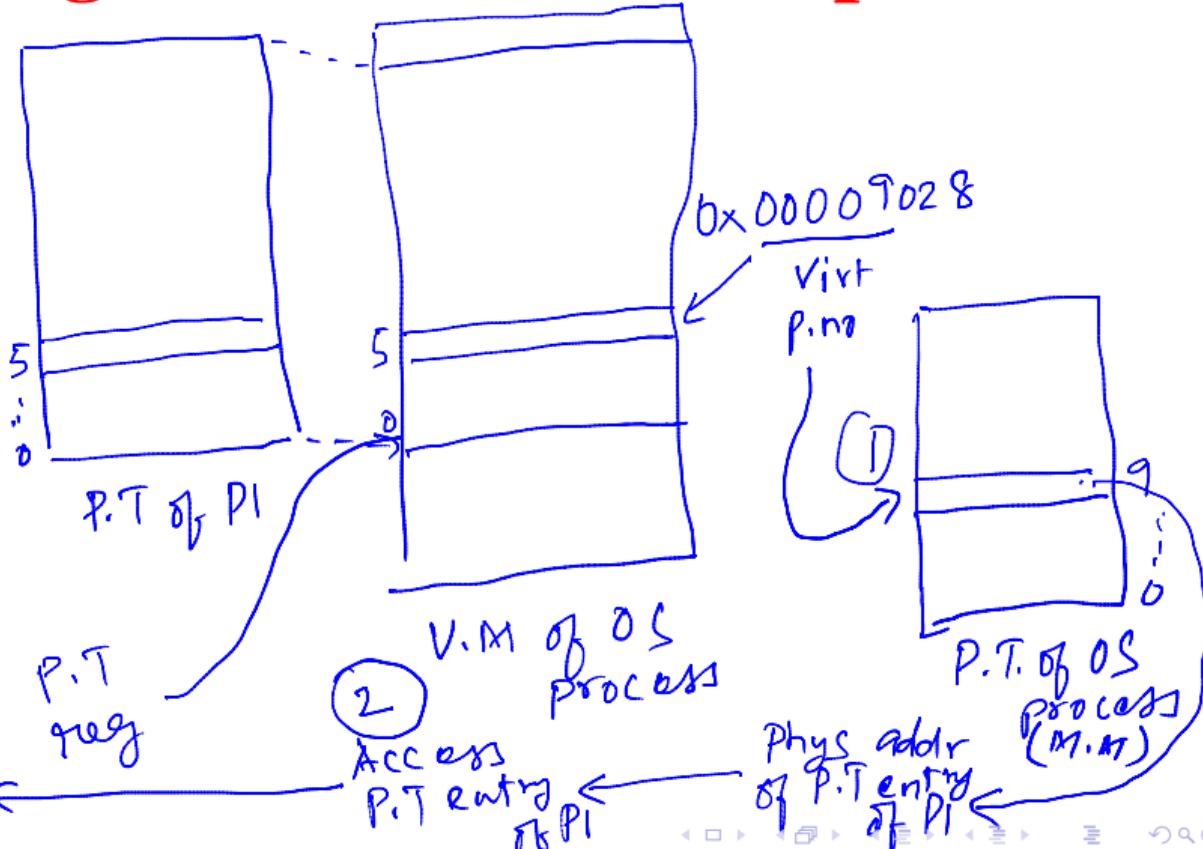
lw R1, 8(R2)



X 0x00005124

Virt p.No

Addr in page



Dealing with Large Mapping Size: Idea Behind the Solutions

- Page table limits: works since most of the page table is empty
- Inverted page table: works since most of the page table is empty
- Multi-level page table: works since most of the page table is empty
- Paged page table: works since there is locality of reference in virtual page references

Summary

- Virtual memory: VA-to-PA mapping is a level of indirection
 - Provides immense flexibility
- Segments: unequal sized blocks
- Managing the mapping: page table
- Ideas to deal with mapping size: page table limits, inverted page table, multi-level page table, paged page table
- Next: putting together VM, caches

Week11/1 virtual-memory-and-caches.pdf

CS305

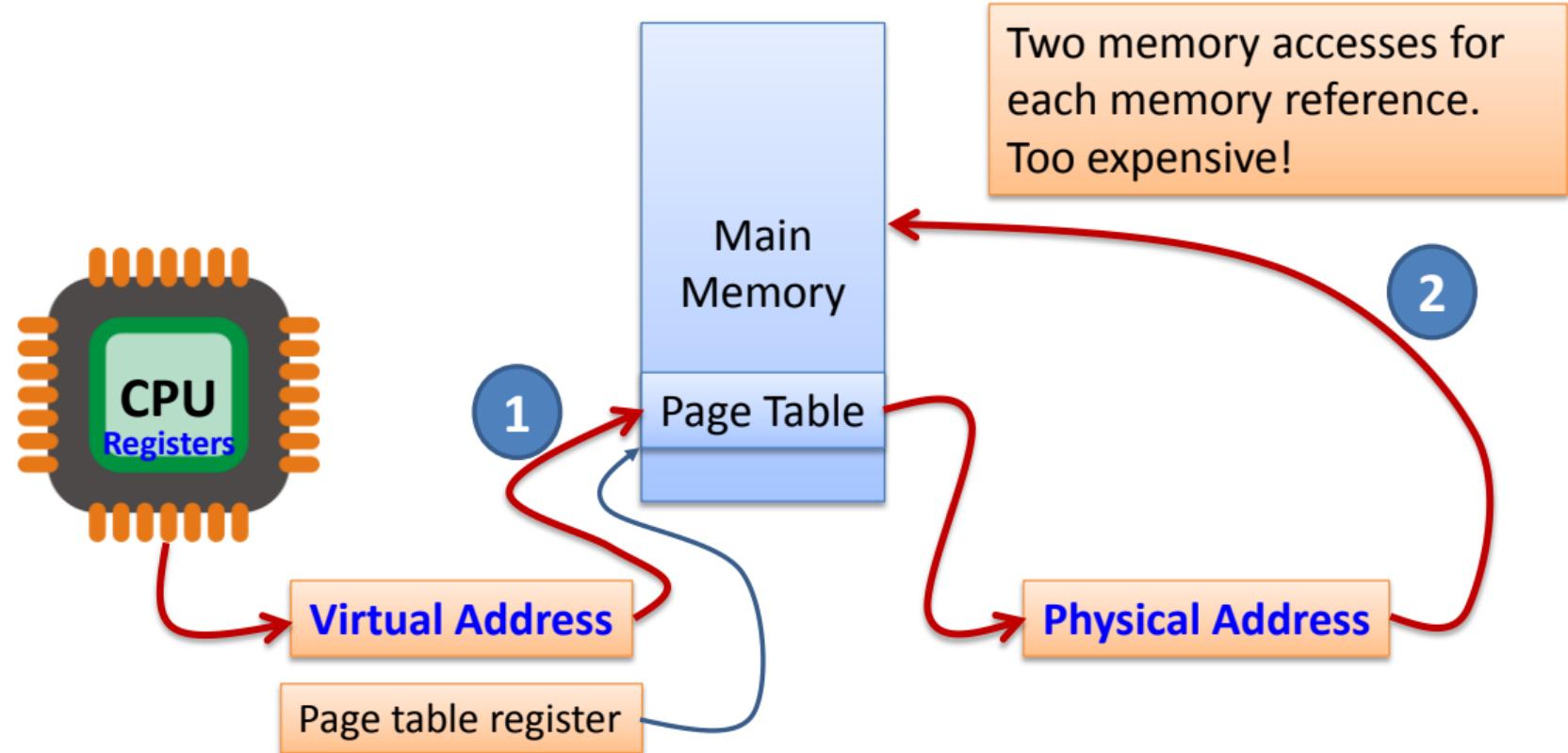
Computer Architecture

Virtual Memory and Caches

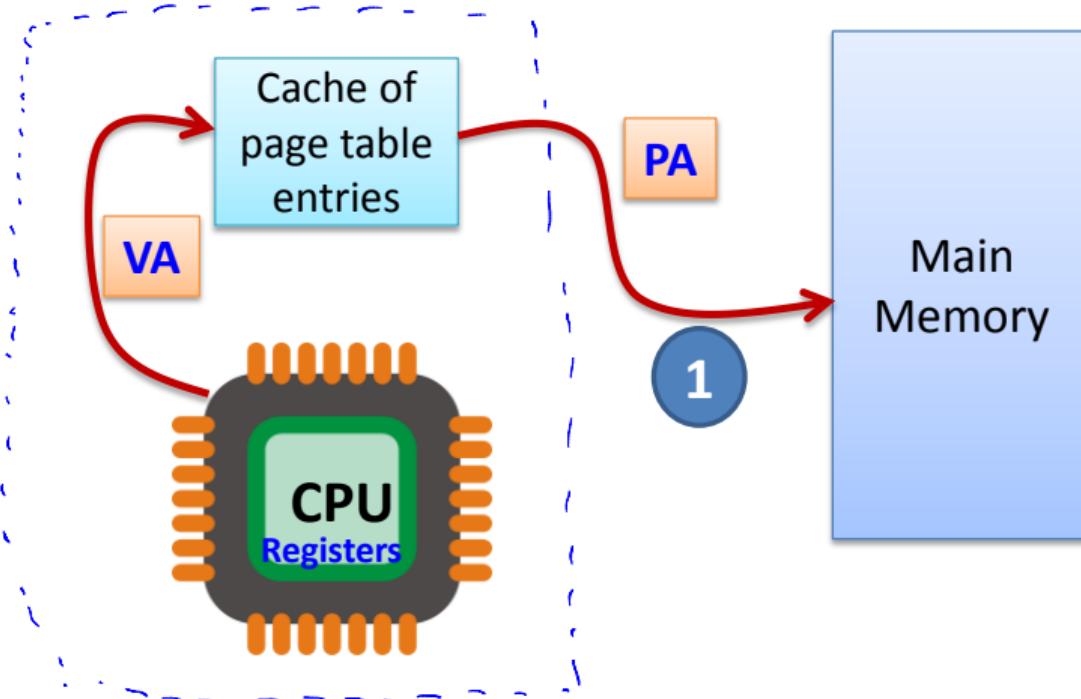
Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

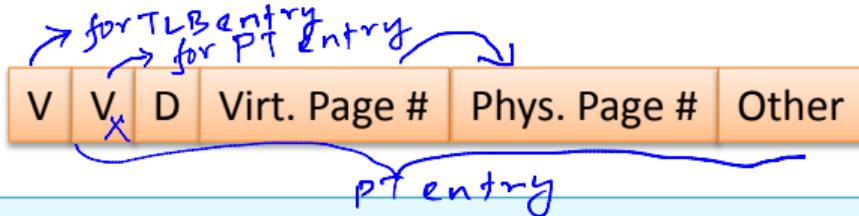
What Happens on a Memory Reference?



Solution: Caching!



Translation Look-aside Buffer (TLB)



- TLB is on-chip
- Unified or separate
- Fully associative and small, or larger with lower associativity
- Cache replacement policy: can't be as sophisticated as for pages
- Some typical parameters:
 - TLB size: 16-512 entries
 - Block size: 1-2 entries (4-8 bytes each)
 - Hit time \leq 1 cycle, TLB access = 1 pipeline stage, if hit time = 1 cycle
 - Miss penalty = 10-100 cycles, miss rate = 0.01-1%
- TLB miss may be handled in hardware or software

Possible Sequence of Events

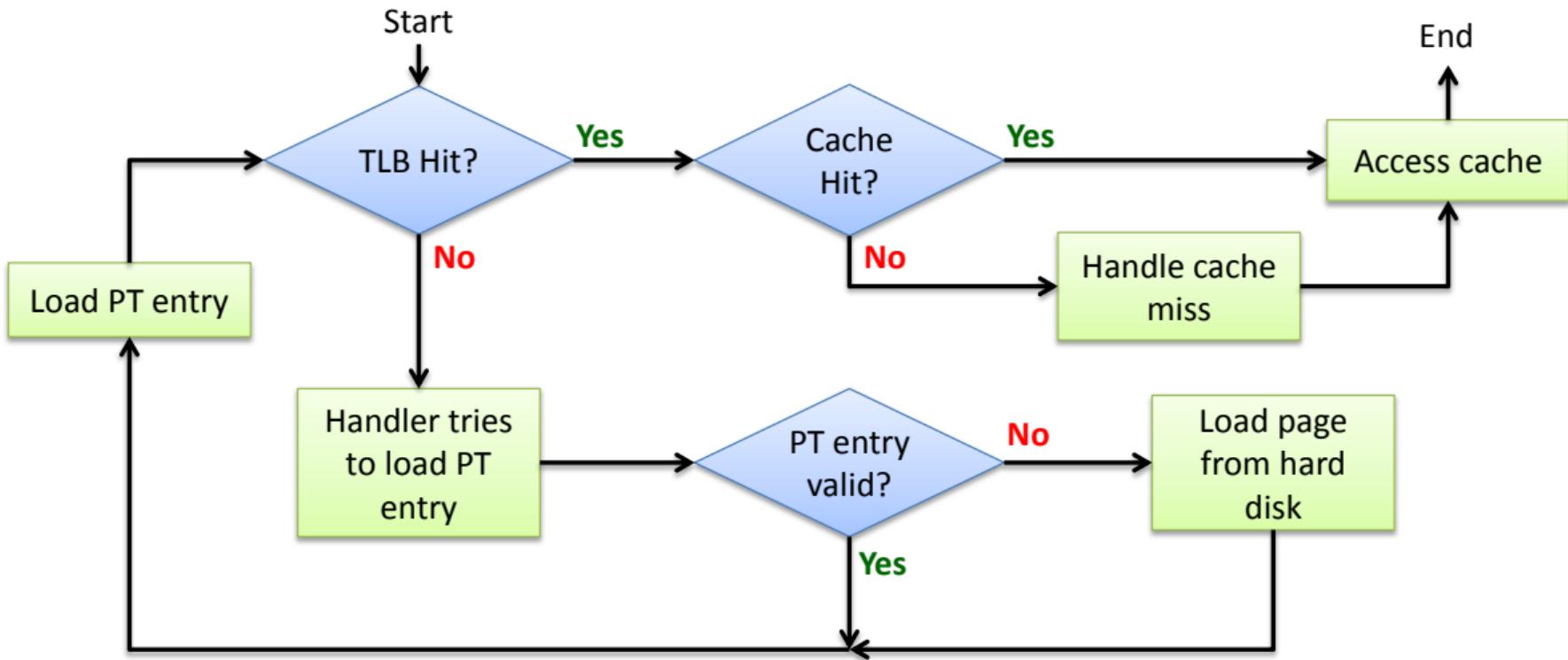


Table of Possibilities

TLB	PT	L1	Remarks
Hit	Valid	Hit	Best case
Hit	Valid	Miss	Only cache miss
Miss	Valid	Hit	Only TLB miss
Miss	Valid	Miss	TLB miss, then cache miss
Miss	Invalid	Miss	Worst case
x	Invalid	Hit	Not possible
Hit	Invalid	x	Not possible

Putting it all Together: A Numeric Example

64-bit VA space, 16GB main memory, 32KB page size

Page table entry: mapping, 16-bit disk addr, bits: valid, used, dirty, write protection

TLB has 16 entries

Q: Compute page table size, indicate fields of VA, PA

A: # virtual pages = $2^{64}/2^{15} = 2^{49}$, # physical pages = $2^{34}/2^{15} = 2^{19}$

Page table entry size = 19 bits for physical page # + 16 + 4 = 39

Page table size = $2^{49} \times 39$ bits

Q: Compute TLB size

A: TLB has 49 bit virt page #, 19 bit phys page #, use bit (for page), use bit (for TLB), valid bit (for TLB), write protect bit (for page), dirty bit (for page), dirty bit (for TLB)

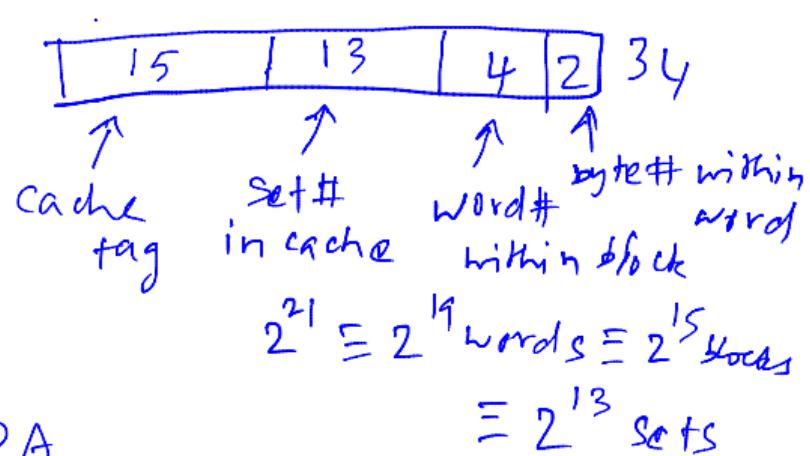
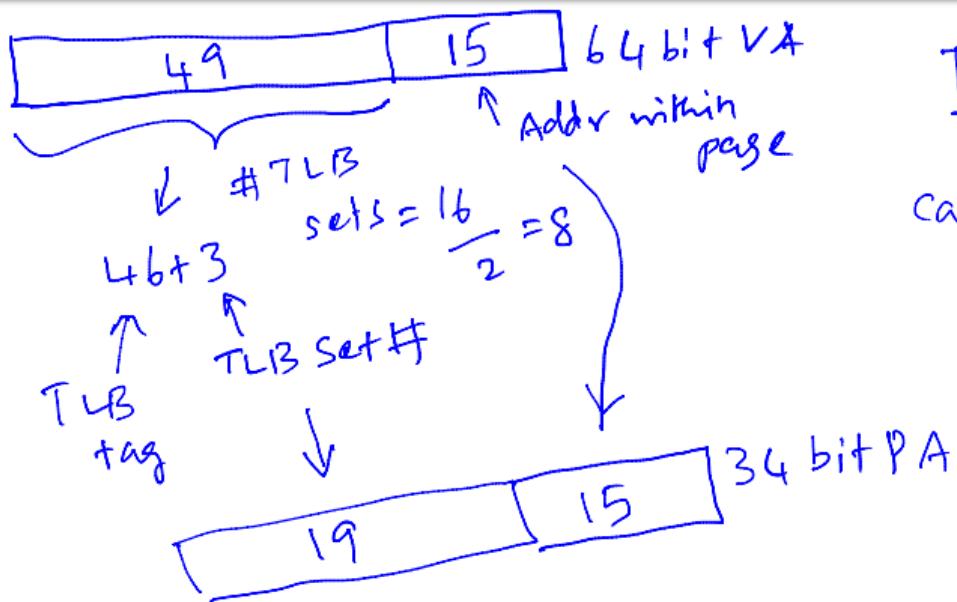
Total = $(49+19+6) \times 16 = 74 \times 16 = 1184$ bits

Numeric Example (Continued)

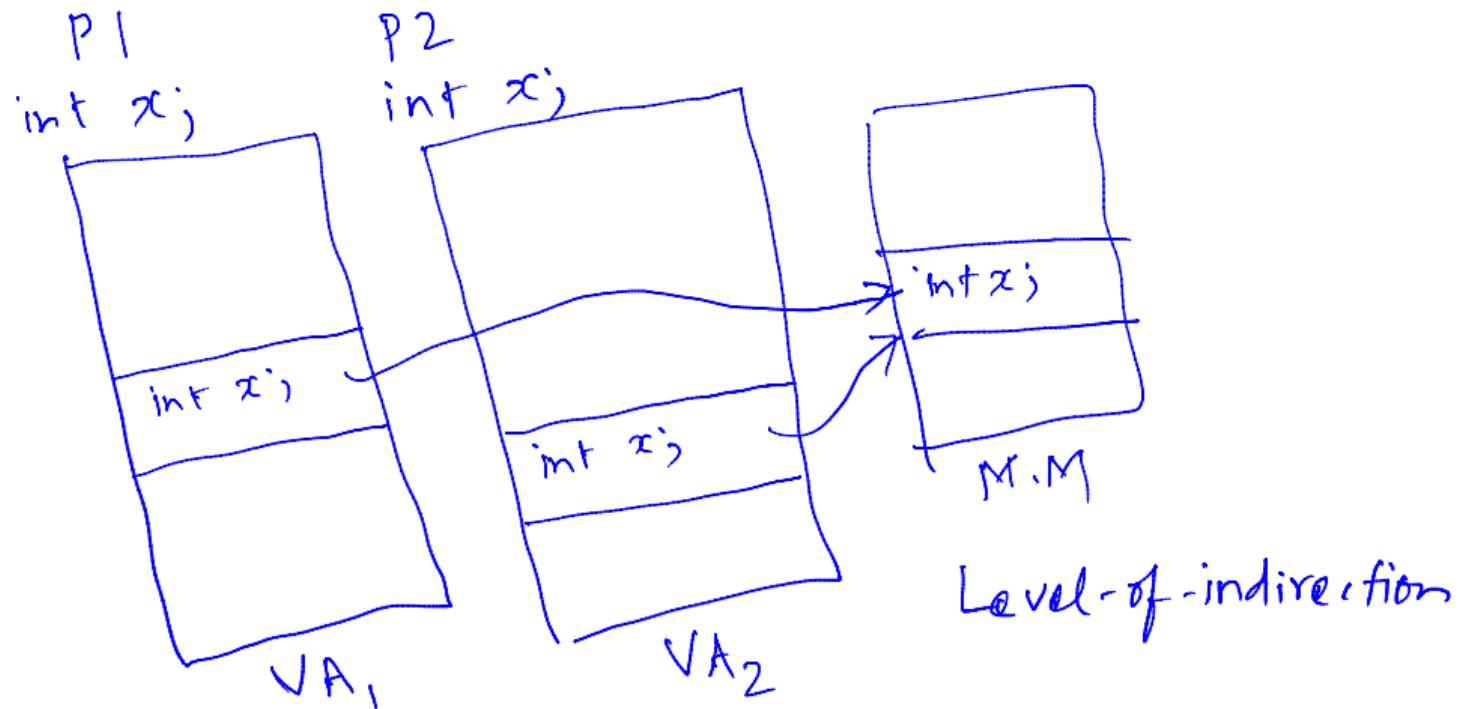
TLB is 2-way set associative

L1 cache: 2MB, 16-word block, 4-way set associative

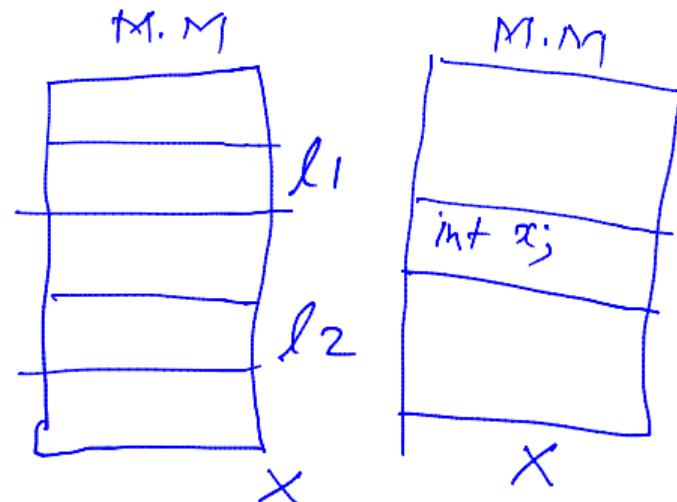
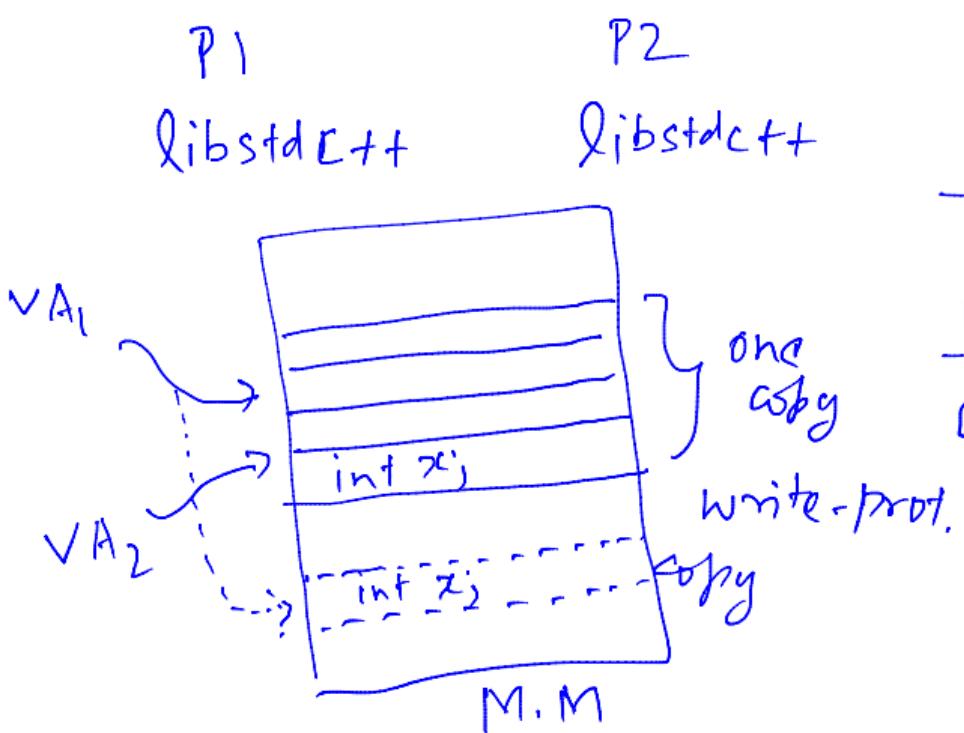
Show bit manipulations during a particular memory access



Shared Memory Using Virtual Memory



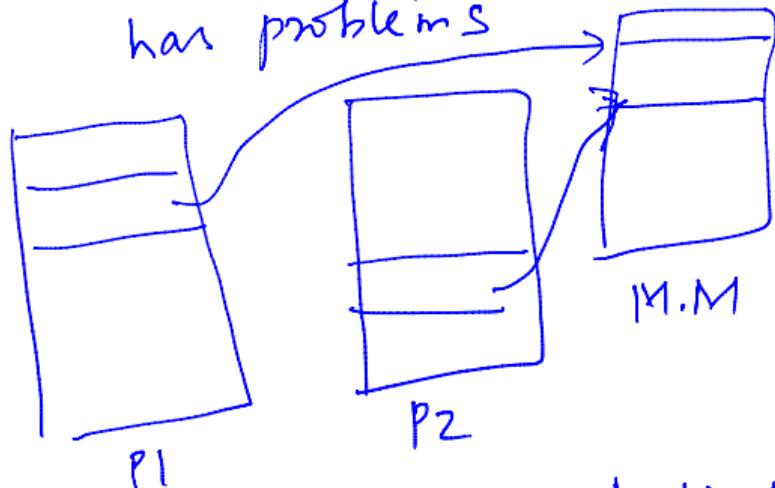
Copy-on-Write Using Virtual Memory



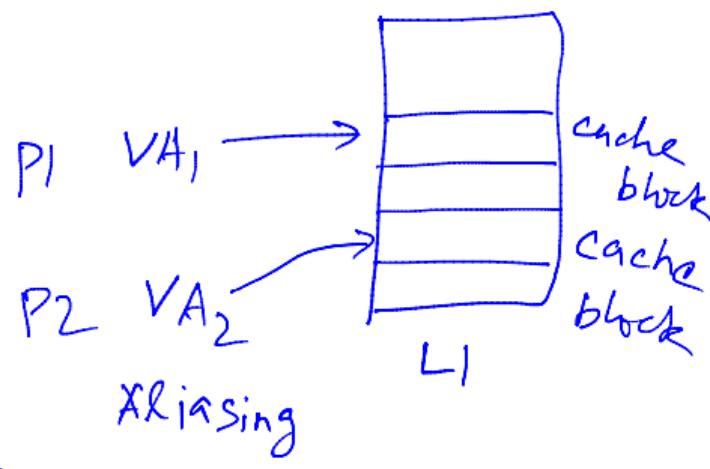
Virtually Addressed Caches

(+) cache hit \Rightarrow VA \rightarrow PA translation
unnecessary

(\rightarrow) shared mem
has problems



Anti-aliasing



Aliasing

Summary

- TLB: cache for PT entries
- Caches in the memory system: TLB, L1, L2, L3, MM
 - Others too: disk is cache for network access, web proxy is cache for server content
- Features using VM: shared memory, copy-on-write
- Next: hardware and OS interaction for VM

Week11/2 hw-os-for-vm.pdf

CS305

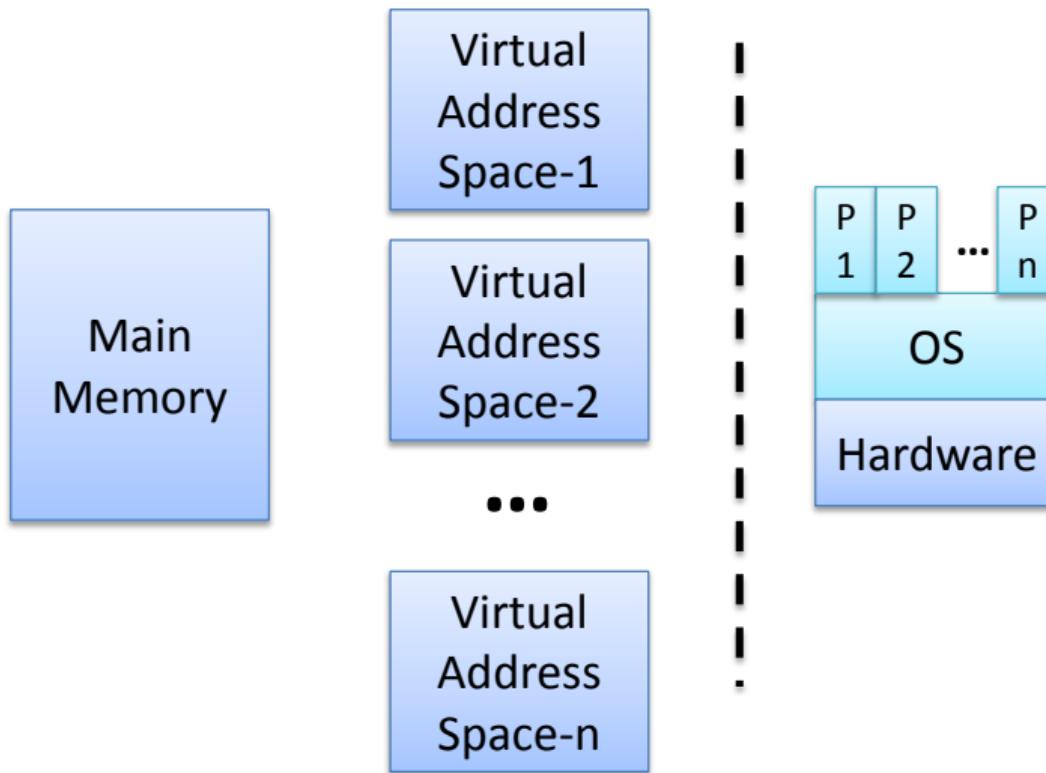
Computer Architecture

Hardware and OS Interaction for Virtual Memory

Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Necessity of Hardware-OS Interaction



- OS manages processes
- Manages their VA spaces
- Hardware-OS interaction crucial for VM
- This is unlike the case of caches (which are mostly transparent to all software)

TLB, PT Management

- PT miss (page fault) handled in software: exception handler
- TLB miss can be handled in hardware or software
 - MIPS handles TLB miss in software: exception handler
 - Need special instructions for TLB access
- What if regular programs write TLB or PT?
 - Need (at least) two processor modes: kernel or supervisor mode, regular user mode
 - TLB, PT writing allowed only in kernel mode

Switching Processor Modes

- Switching modes needs to be controlled
- User-to-kernel:
 - On exception, enter kernel mode automatically
 - `syscall` or `trap` instructions: also called software exceptions
- Kernel-to-user:
 - `eret` (exception return)
- While triggering exception, the hardware:
 - Switches to kernel mode
 - Disables further exceptions (will be enabled at a safe stage)

The MIPS TLB Miss Handler

```
TLB miss exception handler at 0x80000000
mfc0 $k1, Context # spl reg with addr of relevant PT entry
lw   $k1, 0($k1)  # load PT entry (1 word) into reg
mtc0 $k1, EntryLo # prepare to load TLB
tlbwr          # EntryLo --> random locn in TLB
eret           # done handling TLB miss
```

- Invalid PT entries may be loaded onto TLB too!
- TLB miss considered more common than page fault
- Common case is made fast (~ a dozen cycles)

TLB and Multiple Processes

- VA space is per process →
- VA-to-PA mapping is per process →
- TLB entries (cache of this mapping) is per process
- What to do on context switch?
 - Option-1: flush TLB
 - Option-2: have a PID (process ID) tag field in TLB, and process has a PID register (filled by OS)

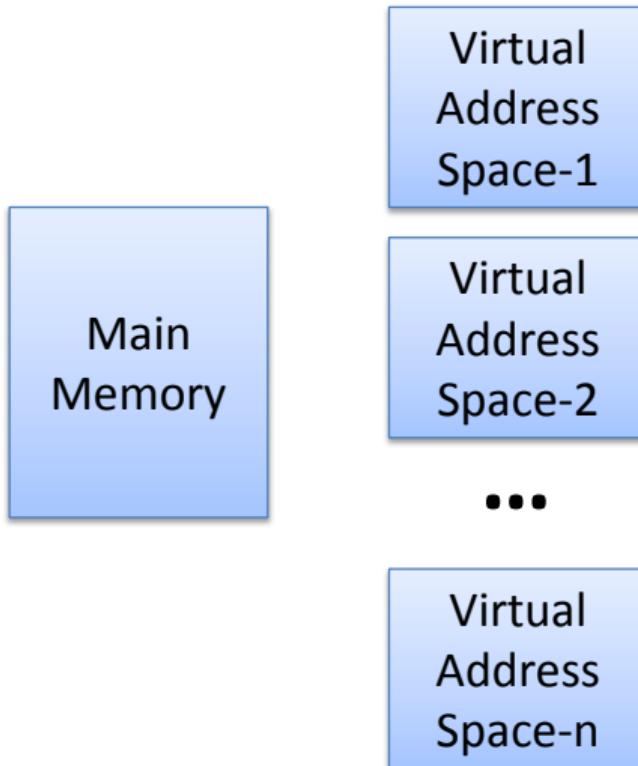
Page Fault Handler

- At 0x8000 0180, separate from TLB miss handler
 - Optimize common case of TLB miss
- Page fault handler has to:
 - Save process state: all GPRs, Hi, Lo onto exception stack
 - Re-enable exceptions
 - Read PT from HD (may need to write dirty page first)
 - Typically, switch to another process which is ready to run

Some Remarks

- Restarting exceptions is NOT easy (e.g. string copy instruction)
- Exception handling itself is not virtual
 - Unmapped memory
 - In MIPS: 0x8000 0000 to 0x8000 FFFF mapped statically to lower portion of physical memory

Thrashing, Working Set



- **Working set:** the set of pages a program or set of “active” programs need in main memory
- **Thrashing:** when working set size exceeds the main memory size
- High page fault rate
- Processor stalls, waiting for disk: terrible performance

Summary

- VM: hardware and OS need to work together
 - Special instructions for TLB access
 - Processor modes
 - Special instructions for switching modes
- Next: Input/Output systems

Week11/3 input-output-intro.pdf

CS305

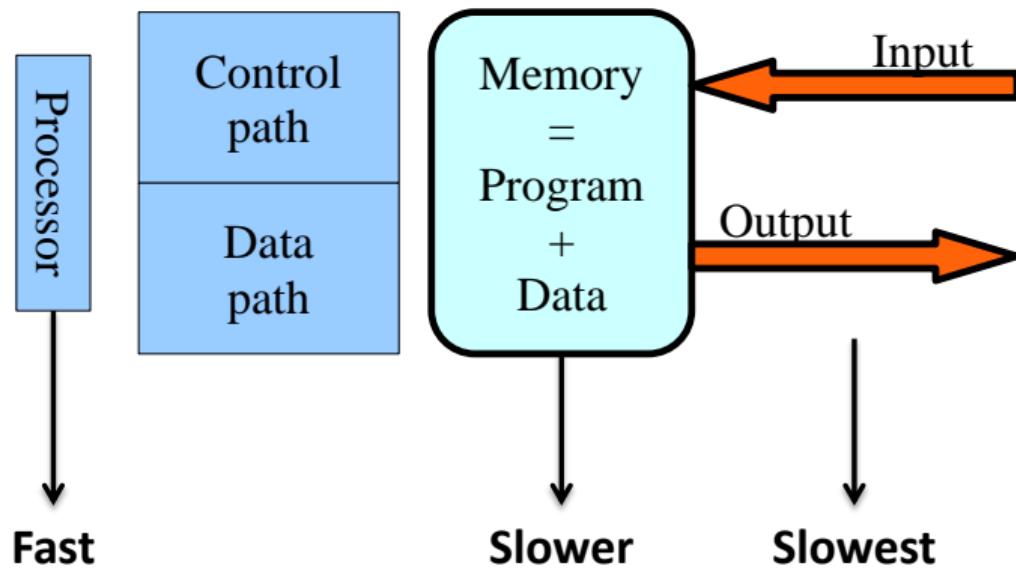
Computer Architecture

Input/Output Systems: An Introduction

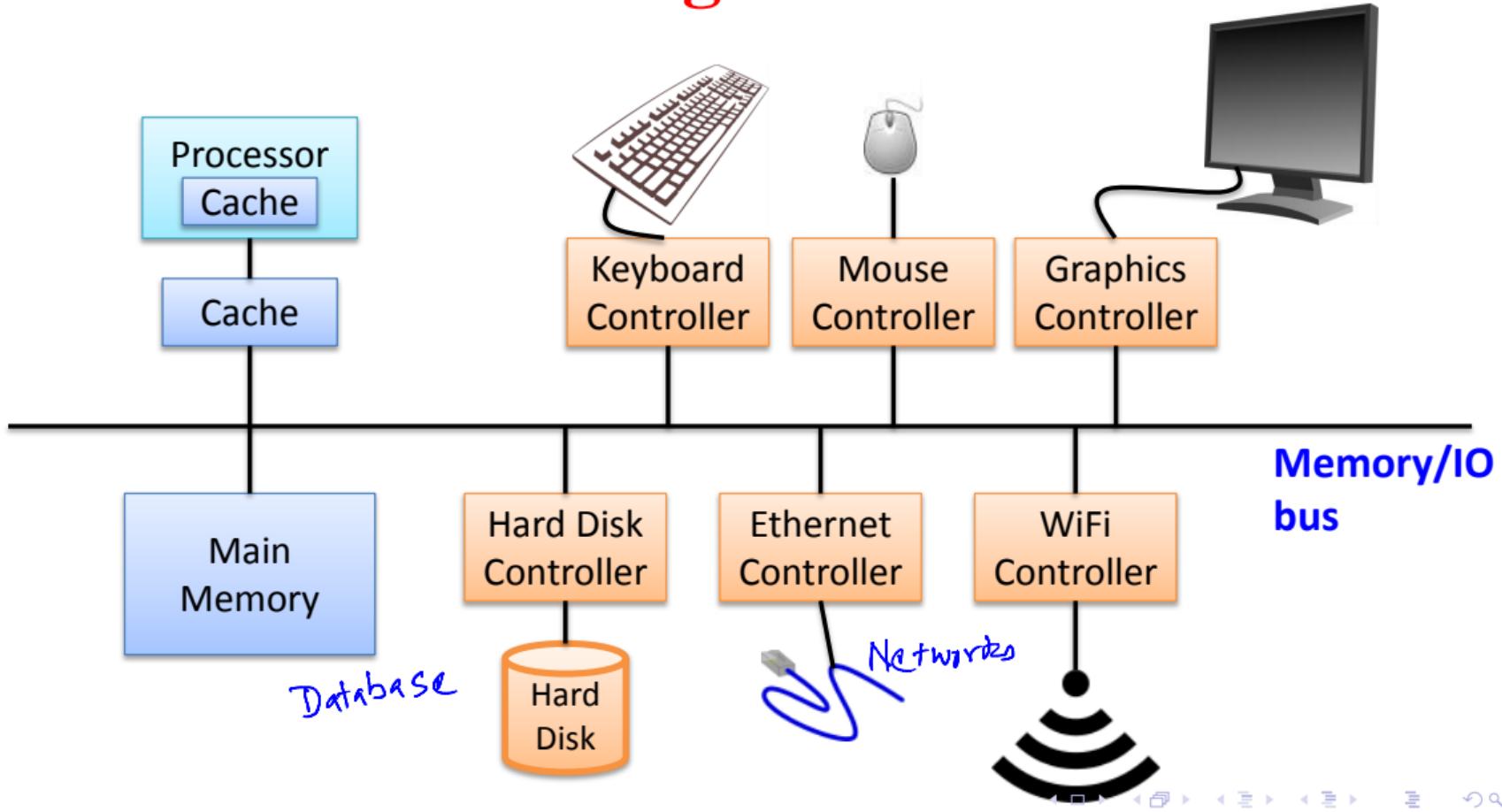
Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Recall: Von Neumann Architecture



Connecting I/O Devices



IO Performance Metrics

- Latency sensitive:
 - Examples: keyboard, mouse
 - Any device with which user interacts directly
- Throughput sensitive:
 - Hard disk, network, graphics
 - Large number of small transactions (e.g. web proxy)
 - Large transactions (e.g. graphics)

Topics in IO Systems

- Magnetic hard disk technology
- RAID: Redundant Array of Inexpensive Disks
- Hamming Codes
- Buses
 - Bus interfacing
 - Bus protocols
 - Bus arbitration

Week11/4 intro-to-buses.pdf

CS305

Computer Architecture

Introduction to Buses

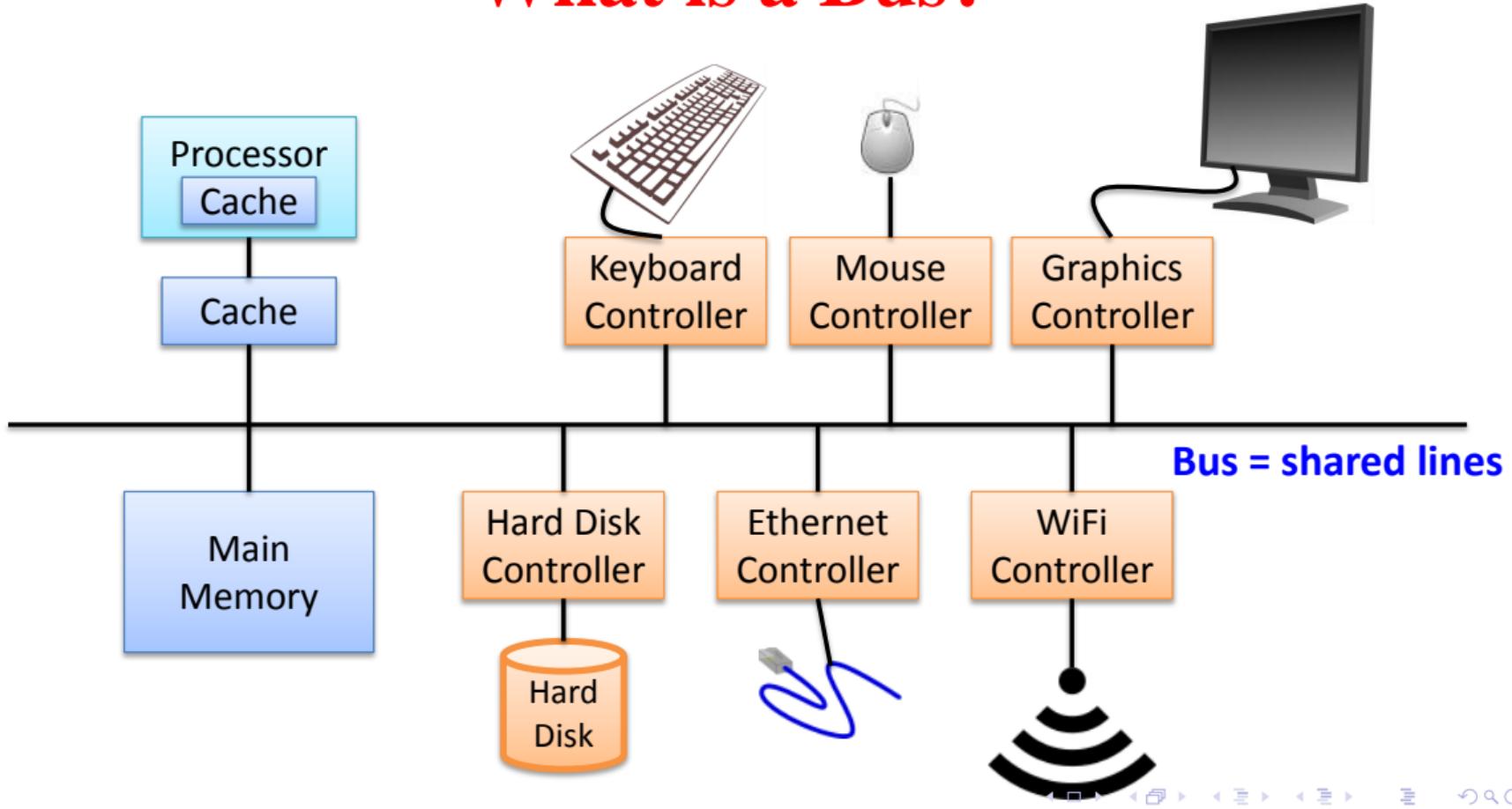
Bhaskaran Raman
Room 406, KR Building
Department of CSE, IIT Bombay

<http://www.cse.iitb.ac.in/~br>

Reference

- “Computer Architecture and Organization”, John P Hayes, McGraw Hill, 3rd Edition

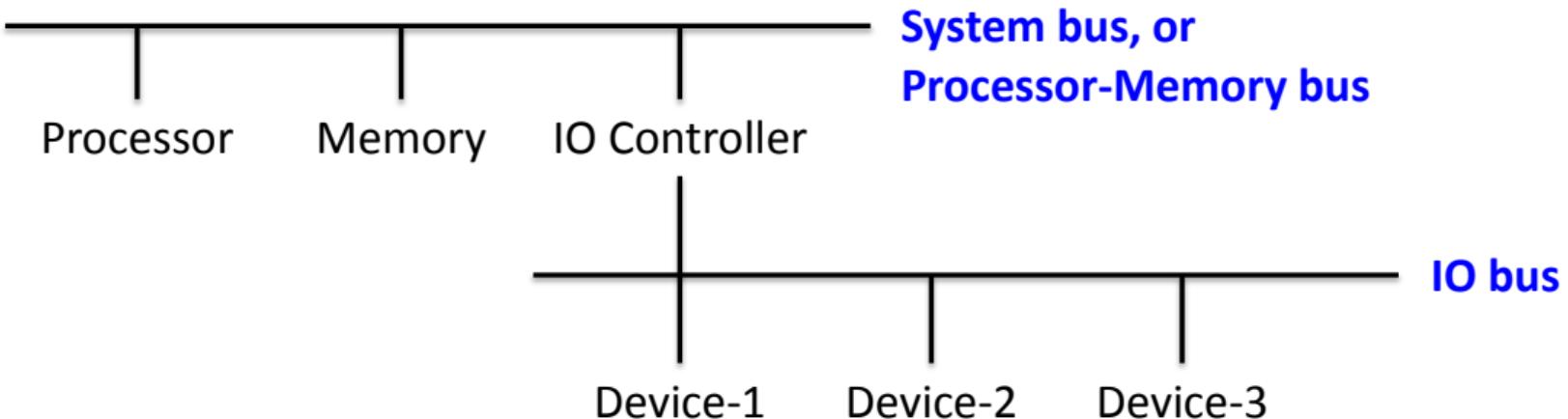
What is a Bus?



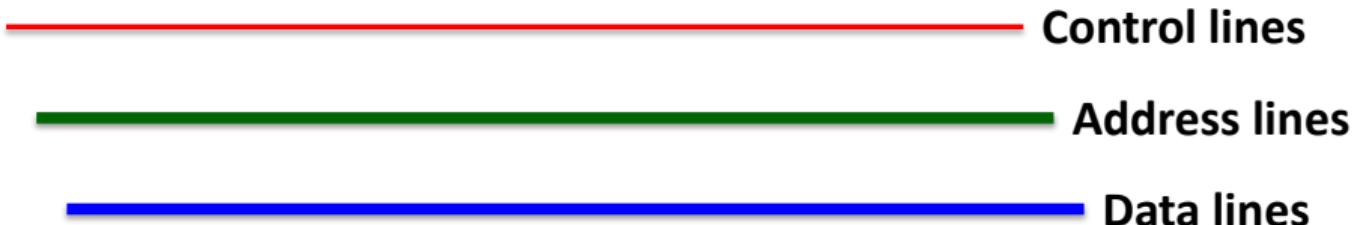
Buses: Merits and Demerits

- Advantages:
 - Versatility or flexibility: Same IO device can be used in different computers
 - Low cost (compared to individual lines betn. proc. & device)
- Disadvantage:
 - Communication bottleneck
 - Max bus speed: limited by length, #devices

System Bus vs IO Bus

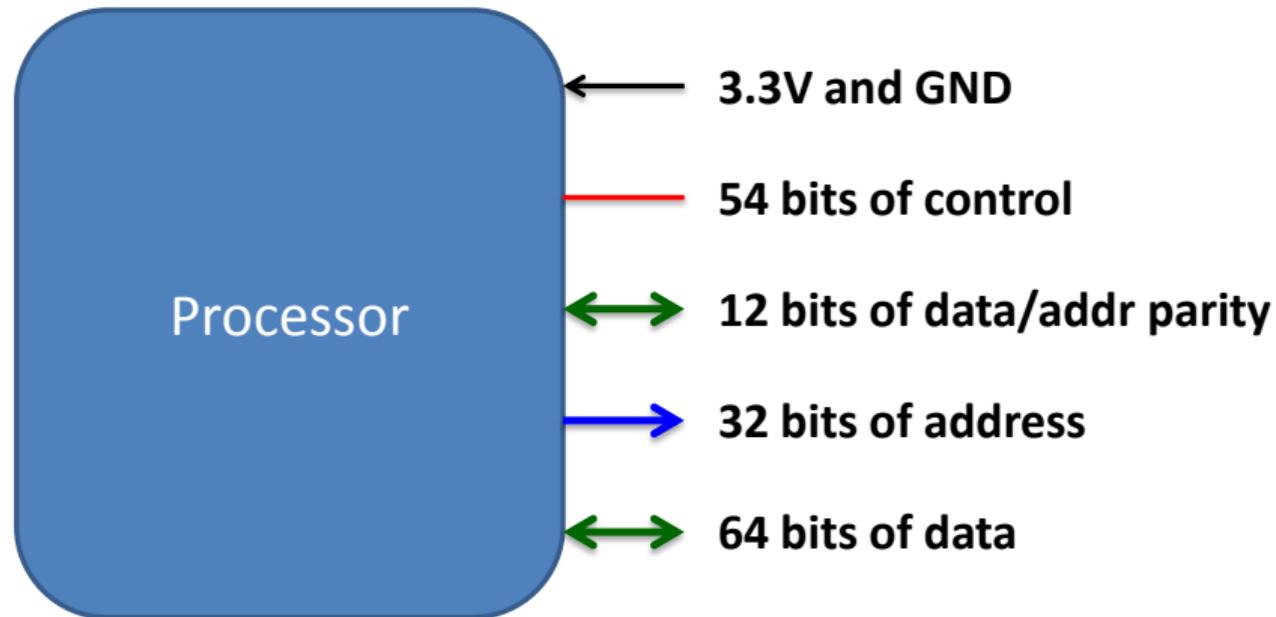


Bus Lines



- Control lines: for bus control
- Address and data lines may be shared in some buses
- Data: up to 4 words in width

Bus Lines in the Power-PC Processor



Bus Standards

- Standards: interoperability
- SCSI: Small Computer System Interconnect
- PCI: Peripheral Component Interconnect
- USB: Universal Serial Bus (hot pluggable)
- IEEE 1394: Firewire (hot pluggable)
- Standards specify: length, frequency, protocol, number of devices, etc
- Our focus: concepts, not standards

Going Forward

- Three bus related concepts:
 - Bus interfacing: how do devices connect to the bus?
 - Bus protocols: how do devices exchange data on the bus?
 - Bus arbitration: who gets access to the bus?