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

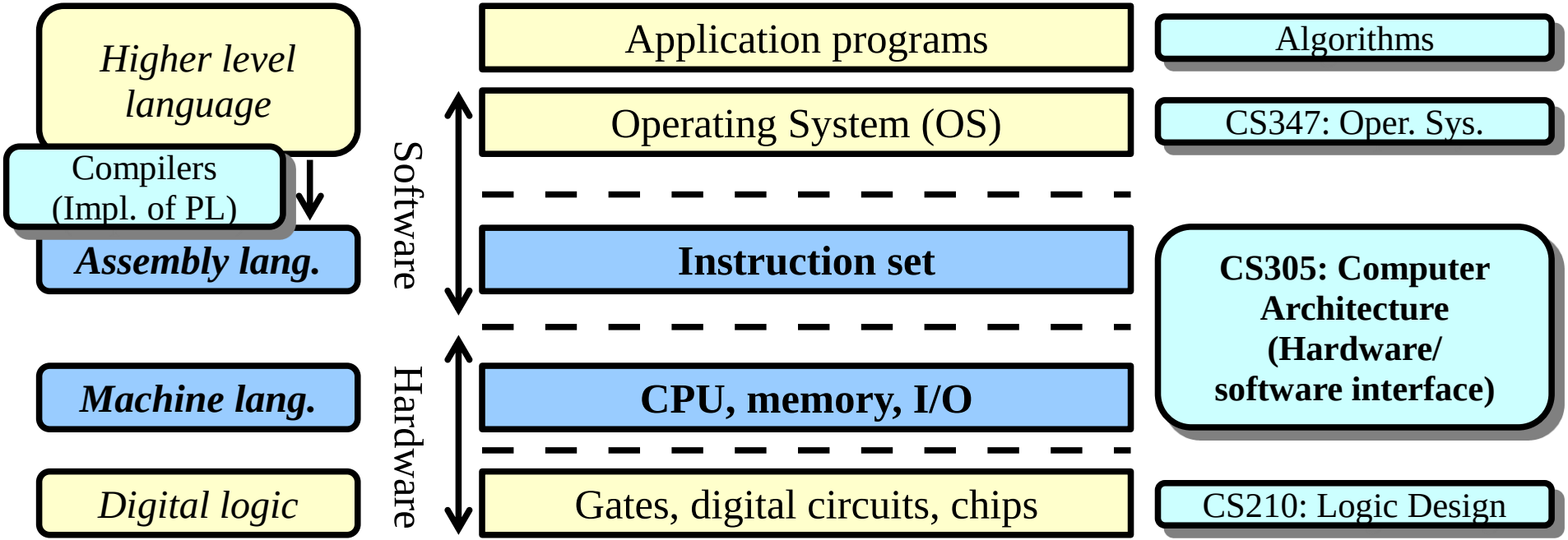
Course Contents

- 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
MIPS

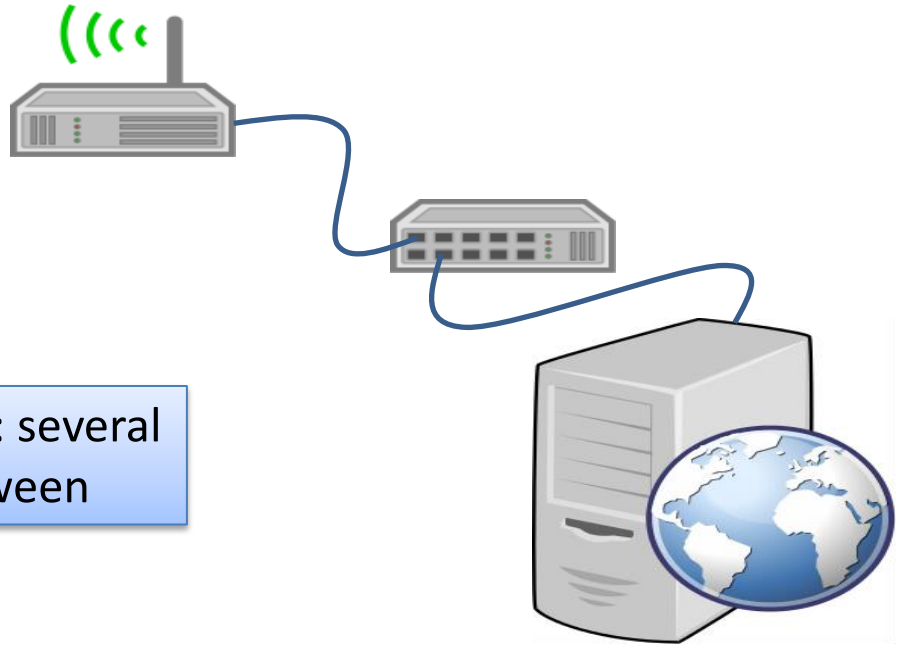
- **"Computer Organization and Design: The Hardware/Software Interface"**, 3rd edition, David A. Patterson and John L. Hennessy, Elsevier (Restricted South Asia Edition).
  - 5th edition available, ok to follow, I'll follow 3rd edn. closely
- **"Computer Architecture and Organization"**, John P. Hayes, 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

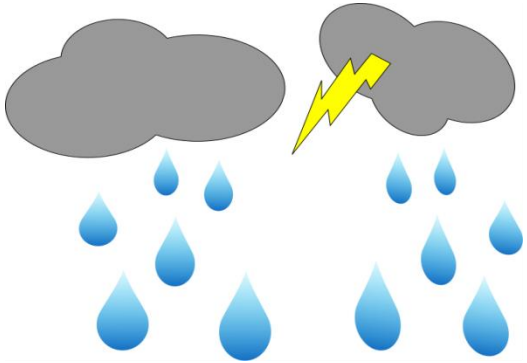
A variety of personal devices: the continuum between cell-phones and PCs

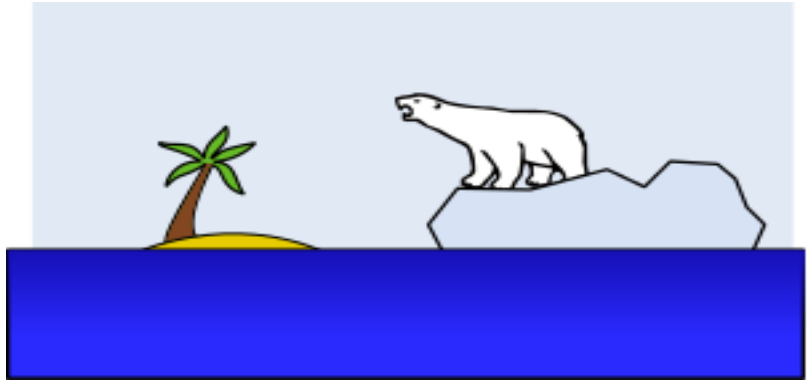
Example-3: Servers, Data Centers, Cloud Computing

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

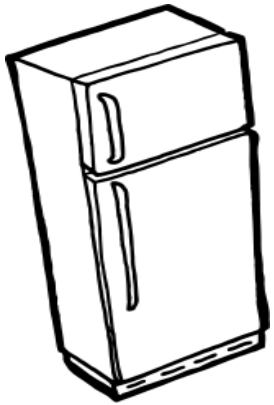
Example-4: Supercomputers

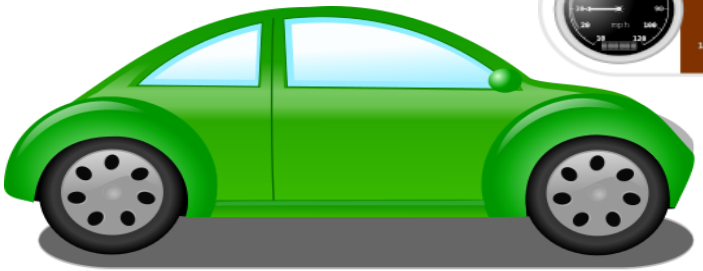
Specialized but important applications, high-end research





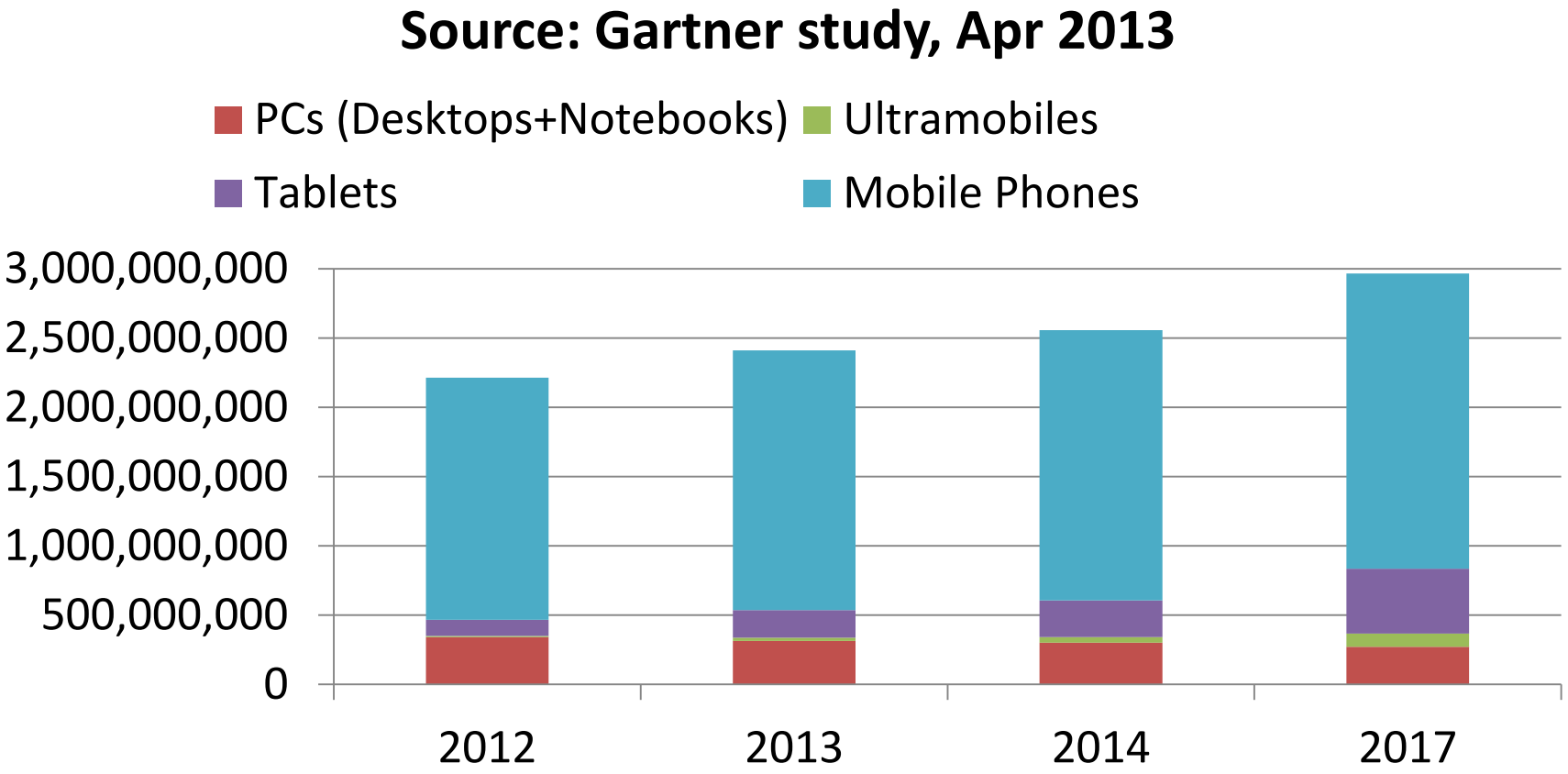
Example-5: Embedded Computers





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

Personal Computing Devices in Numbers



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?

CS305
Computer Architecture

Parts of a Computer

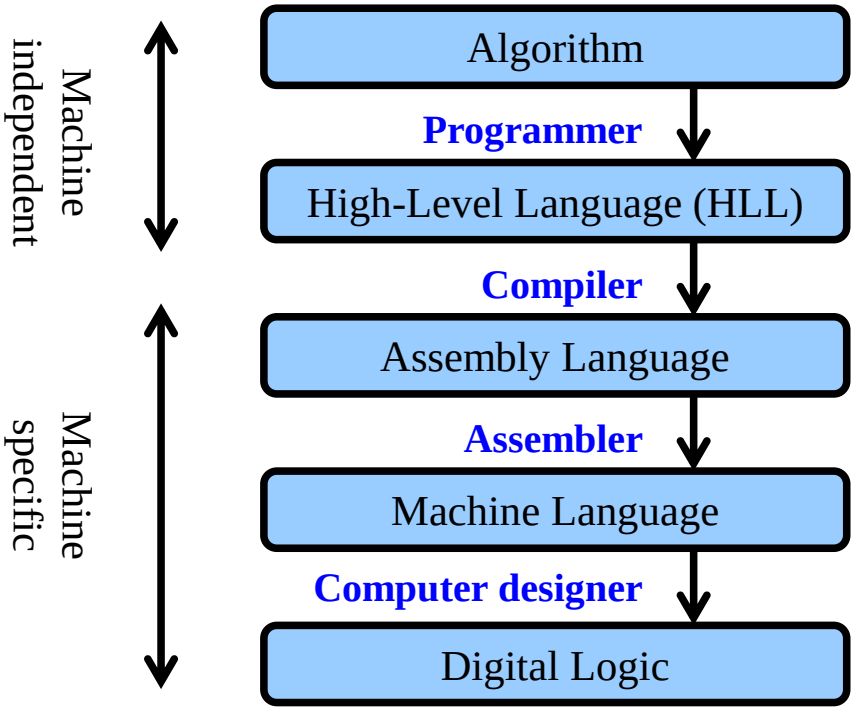
Bhaskaran Raman

Room 406, KR Building

Department of CSE, IIT Bombay

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

A Hierarchy of Languages



Natural language (almost), with some symbols

Concise; structured, yet human readable

Human readable representation of machine language; not concise

Binary numbers (0's and 1's)

Electrical signals

An Example

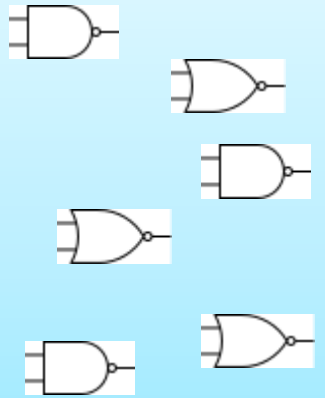
Algorithm:
Compute the sum of…

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

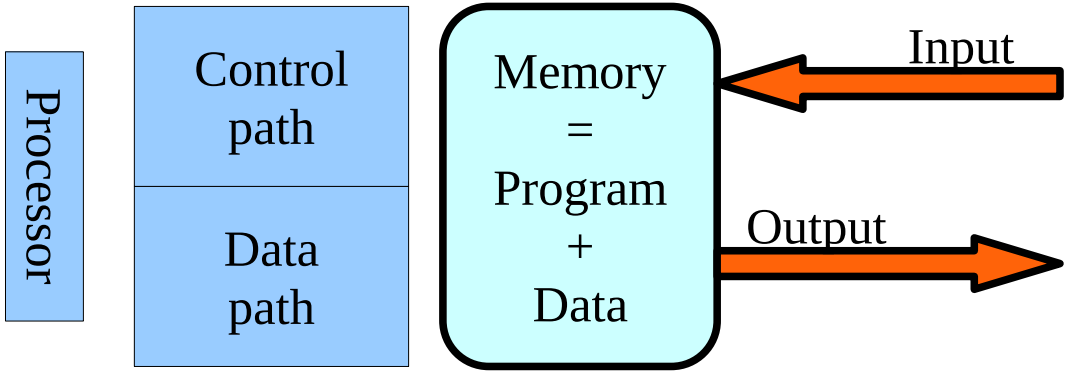
Assembly code:
```
lw      $s1, 4($s0)
lw      $s2, 8($s0)
add     $s3, $s1, $s2
sw      ($s0), $s3
lw      $s3, 16($s0)
lw      $s4, 20($s0)
add     $s5, $s3, $s4
sw      12($s0), $s5
```

Machine code:
```
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
...0...1...
```

Digital logic:



Von Neumann Architecture



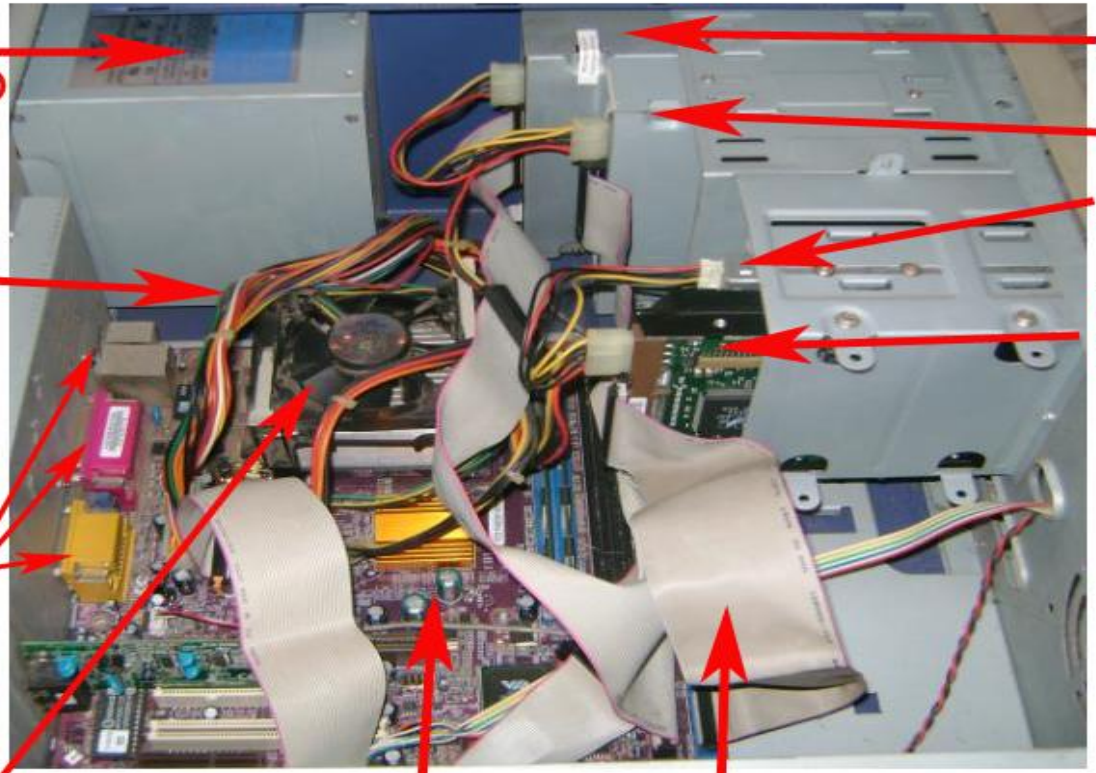
- The stored program concept: program (instructions) as well as data are stored in memory
- Processor *fetches* instructions from memory, and *executes* them on *data* (also fetched from/to memory)
  - Example from previous slide: LW and SW instructions

The Five Components

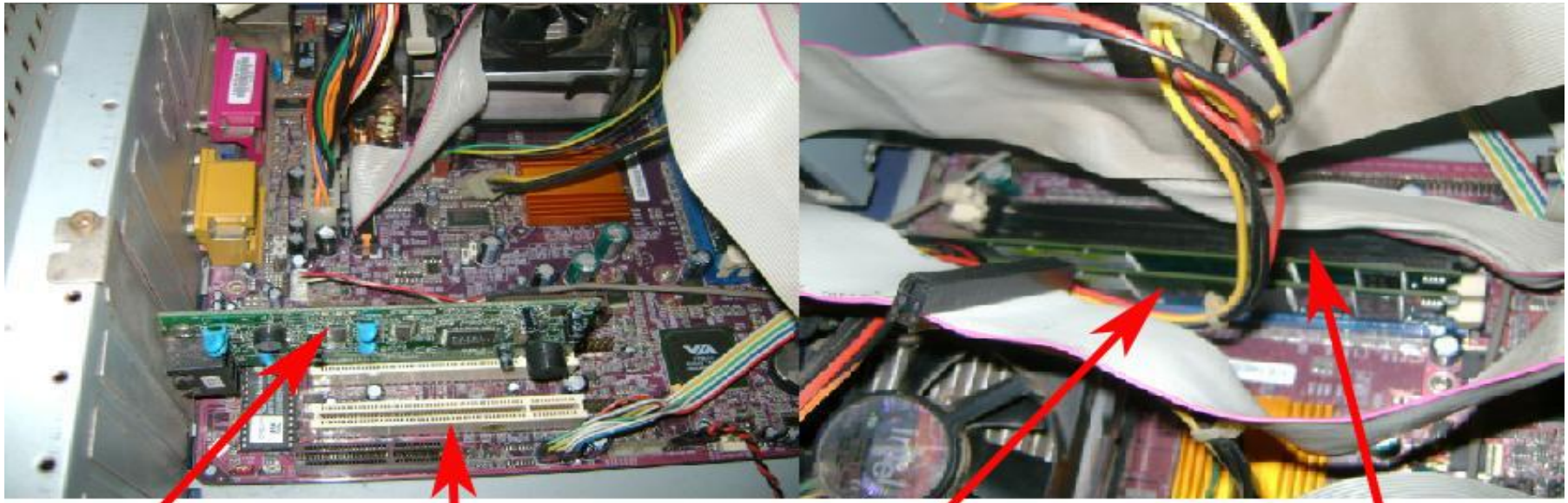
- All computers have these five components: input, output, memory, [data path + control path = processor]
- Underlined aspects: topics in this course

- 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)

CS305
Computer Architecture

Integrated Circuit (IC) Technology: An Overview

SSI
MSI
LSI - 10^4
VLSI
ULSI

Bhaskaran Raman

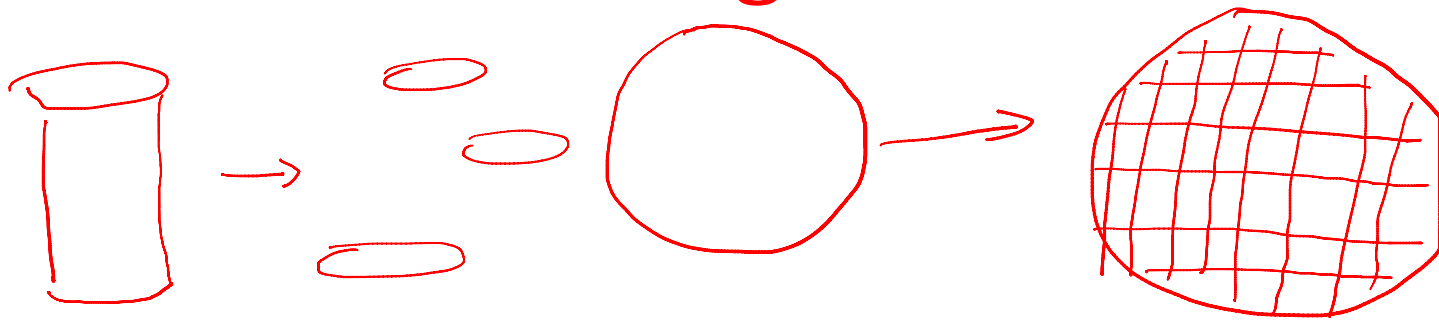
Room 406, KR Building

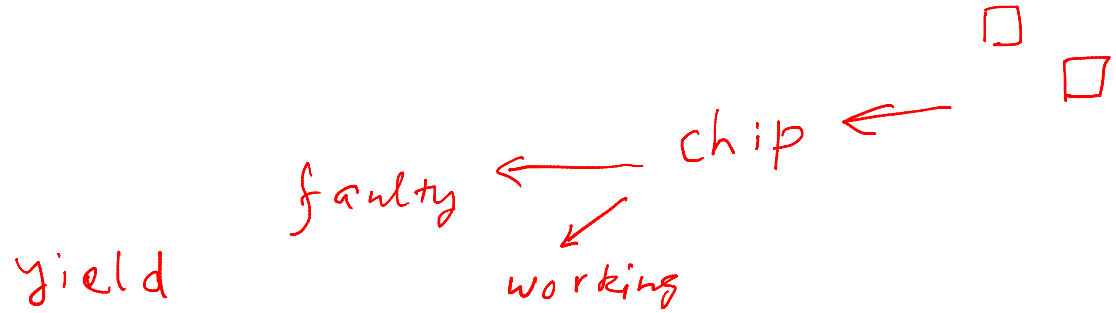
Department of CSE, IIT Bombay

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
  - ~ 2.6 cm^2, 4 GHz Intel Core i7, 88W power
- Fan needed to sink the heat
- Frequency scaling employed

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++

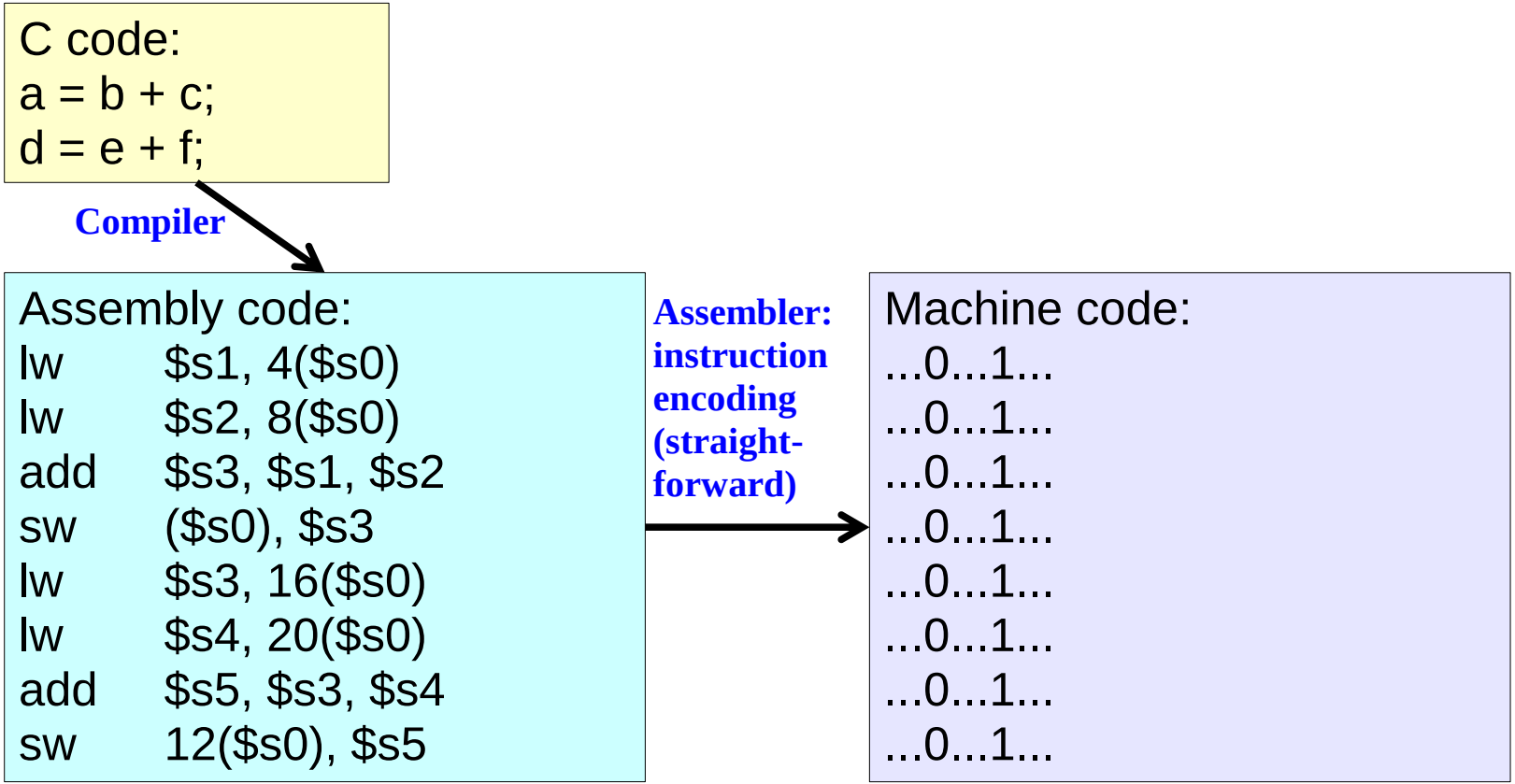
```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

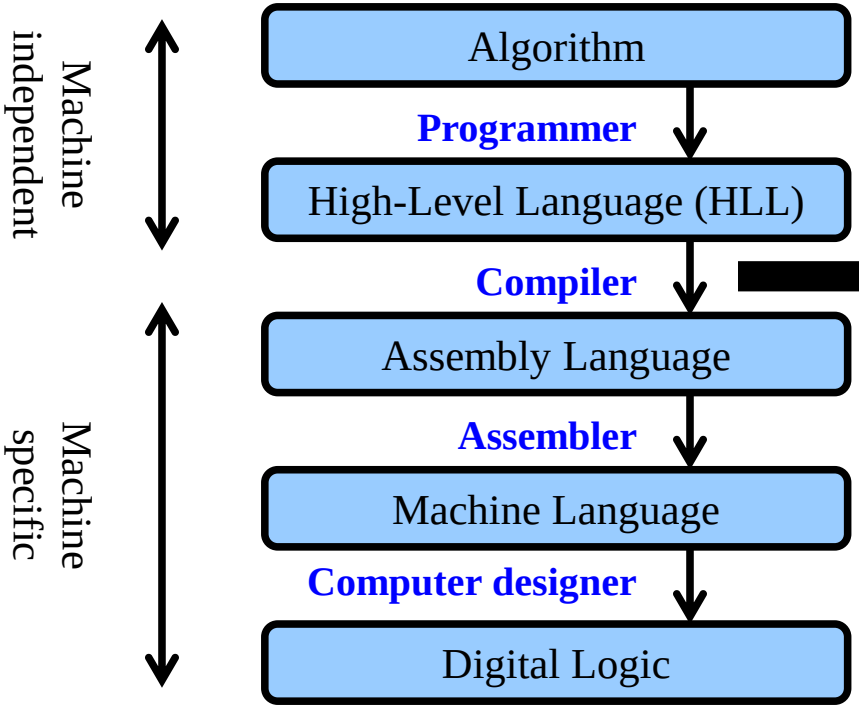
```
$line =~ s/abc/xyz/g;
```

- 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



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

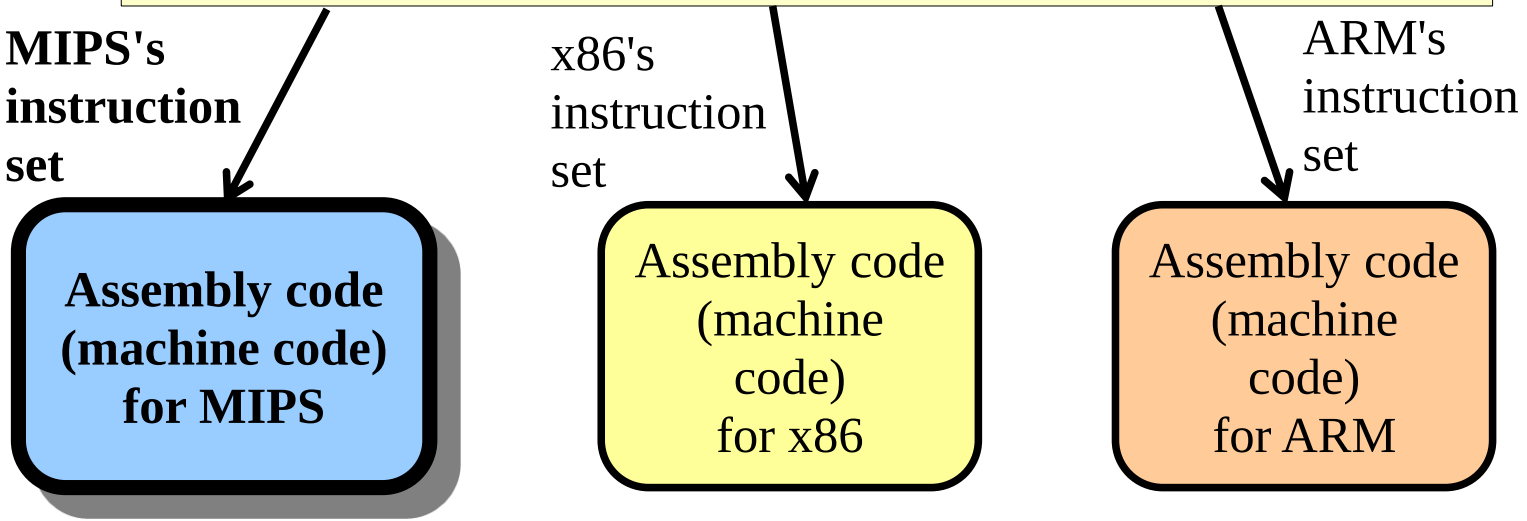
HLL code examples:

C/C++

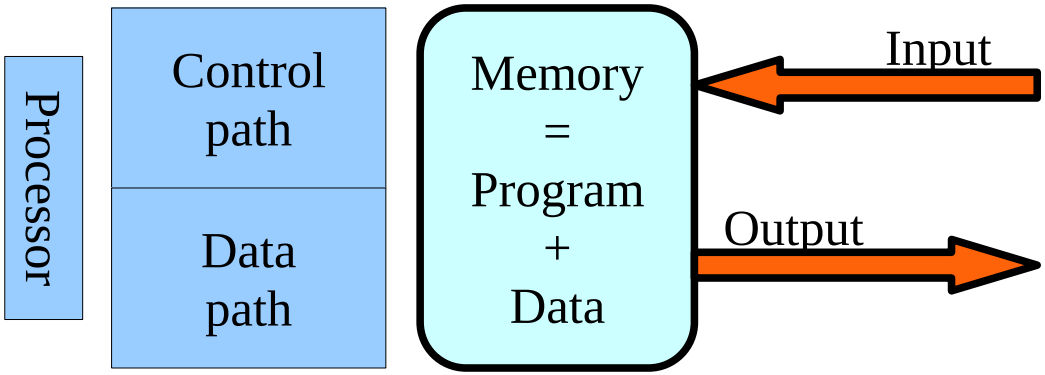
```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

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



Instruction Set and the Stored Program Concept

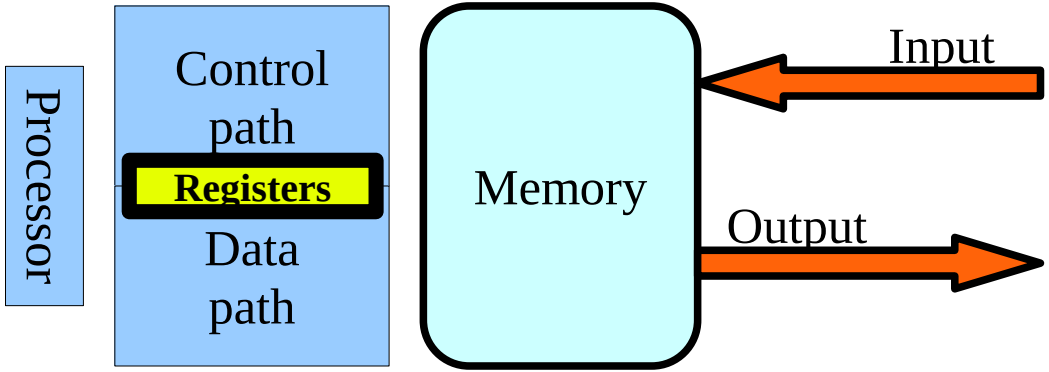


- At the processor, two steps in a loop:
  - Fetch instruction from memory
  - Execute instruction
    - May involve data transfer from/to memory

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

$0 - $31

  - In MIPS: 32 registers numbered 0-31
  - $s0-$s7 are assembly language names for register numbers 16-23 respectively (why? answered later)

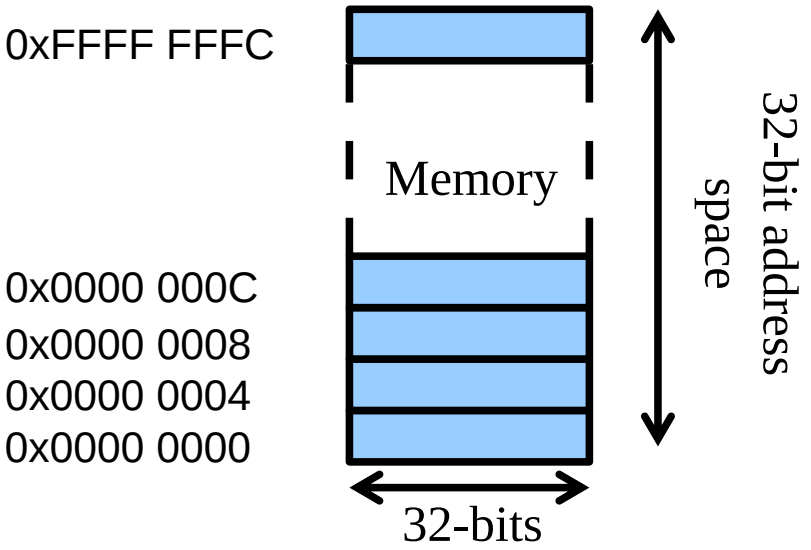
Constant or Immediate Operands

```
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
```



- Load and store in units of 1-word: terminology w.r.t. CPU
- Also called data transfer instns: memory ↔ 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
```

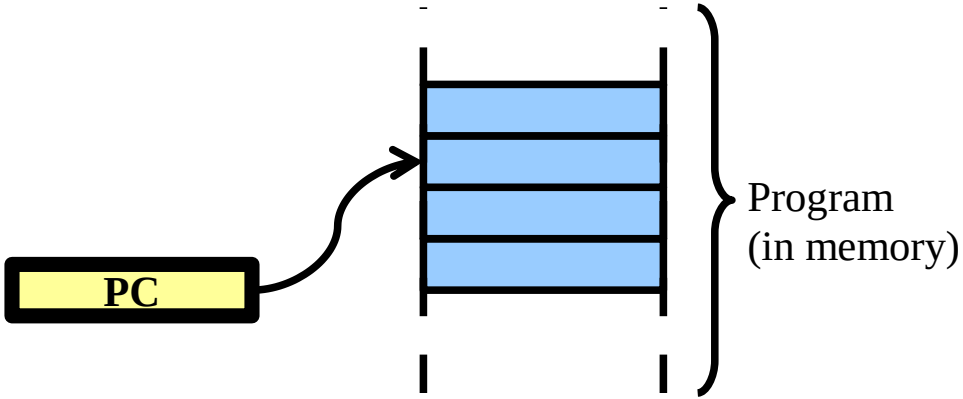
Notion of Register Assignment

- 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: 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
ELSE:
sub    $s1, $s0, $s1
EXIT:
# Further instructions below
```

$0

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 <= or > or >= 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?

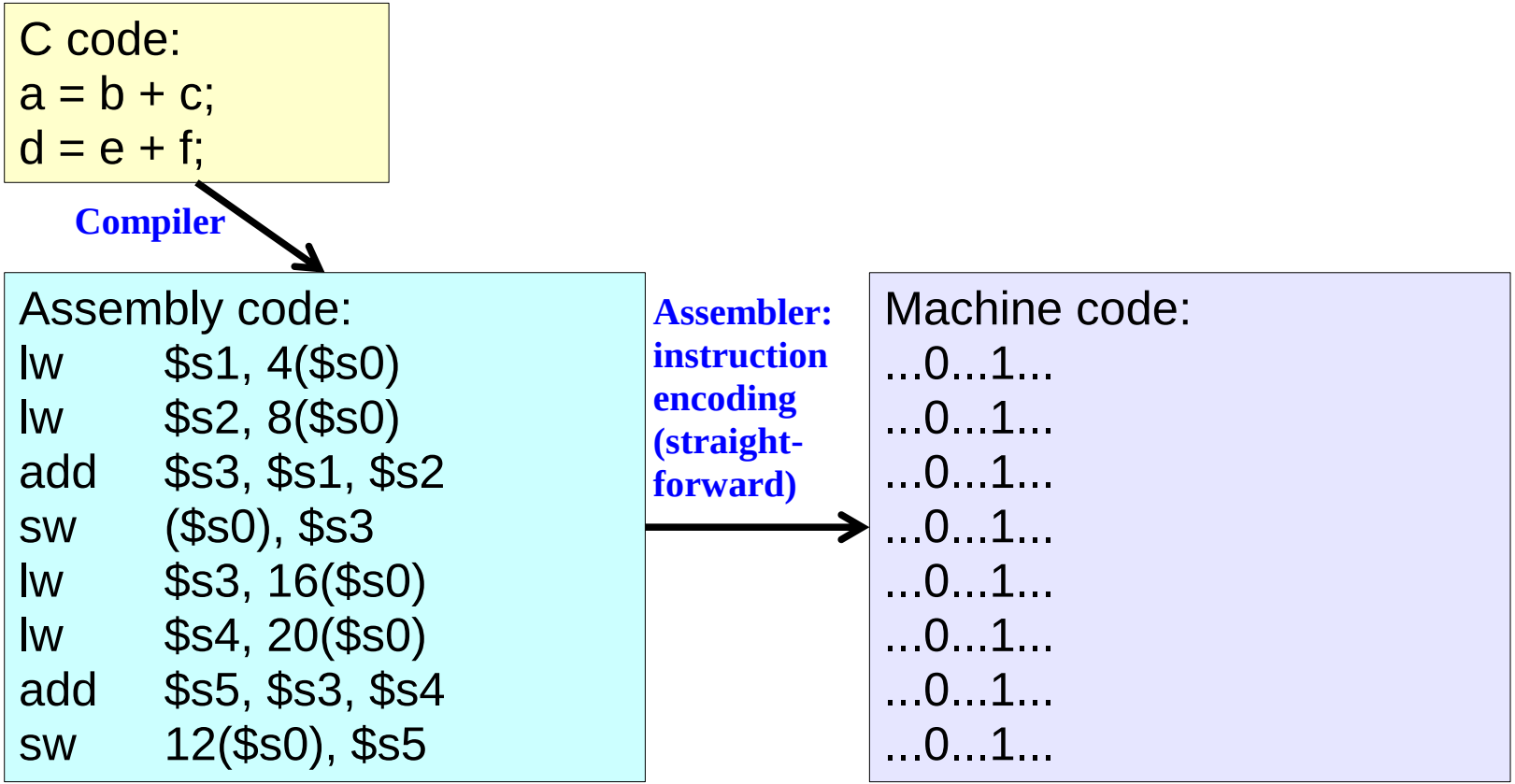
CS305
Computer Architecture

Instruction Encoding

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

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

A Simple Example



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

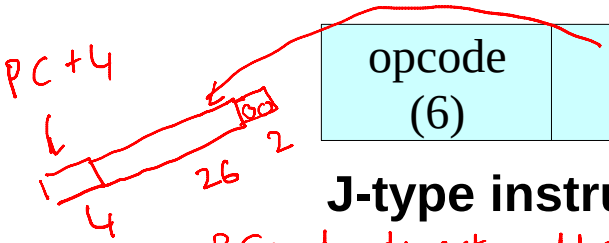
addi

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



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 ?

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

Arithmetic: add, addi, sub

Logical: and, or, nor

sll, srl

memory: lw, sw

branch: beq, bne, j

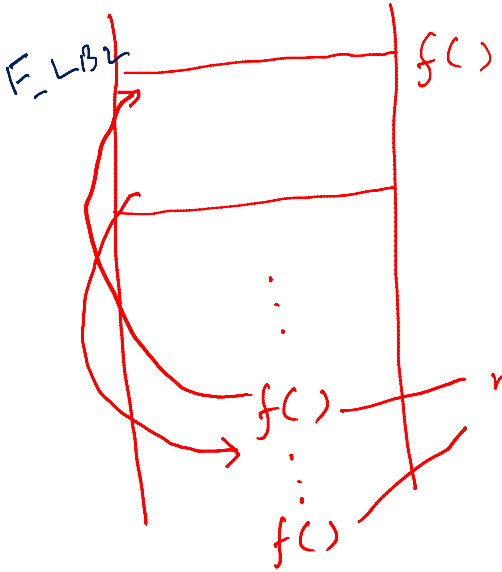
$0-$31

$s0-$s7

$t0-$t9

$zero

Basic Function Call Support: What is Needed?



many calls to the same function

function call, return

j is sufficient

remember the return address

jal F_LBL

part of MIPS 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 the 32 MIPS regs:

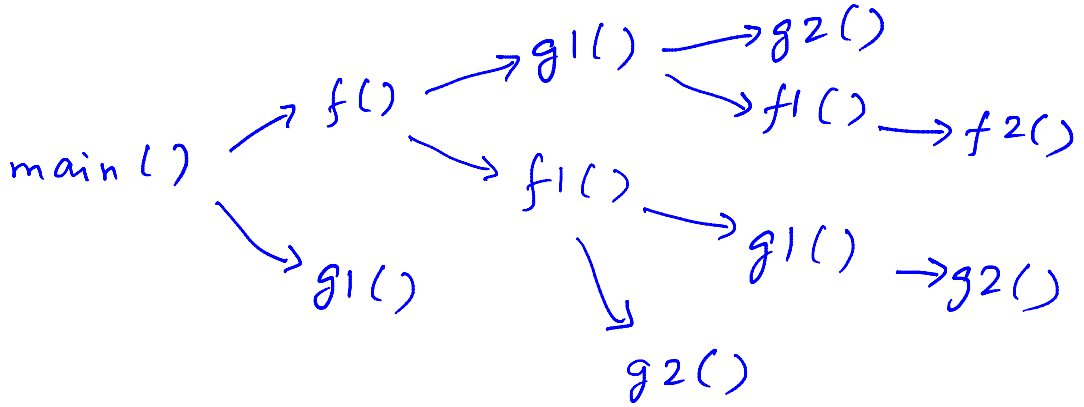
- $a0, $a1, $a2, $a3 – arguments
- $v0, $v1 – return values

Convention – not enforced by MIPS h/w

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 addr on to stack
- restore $ra from stack after function call

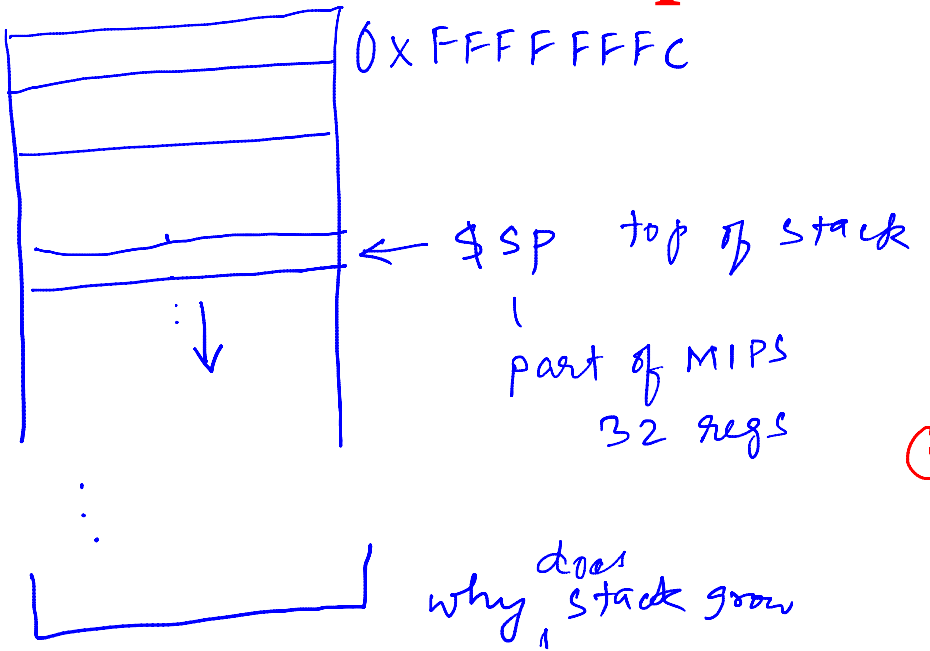
e.g. main → f() → g1()
f() → g2()

```
f():
    :
    save $ra onto stack
    jal g1
    restore $ra from stack
    :
    save $ra onto stack
    jal g2
    restore $ra from stack
    :
```

more efficient

```
f():
    save $ra onto stack
    :
    jal g1
    :
    jal g2
    :
    restore $ra from stack
```

Stack Implementation



Q: write code to push $ra onto stack

```
addi  $sp, $sp, -4
sw    0($sp), $ra
```

Q: write code to pop $ra from stack

```
lw    $ra, 0($sp)
addi  $sp, $sp, 4
```

Arguments and Return Values in Nested Calls

main() → f(...) → g(...)

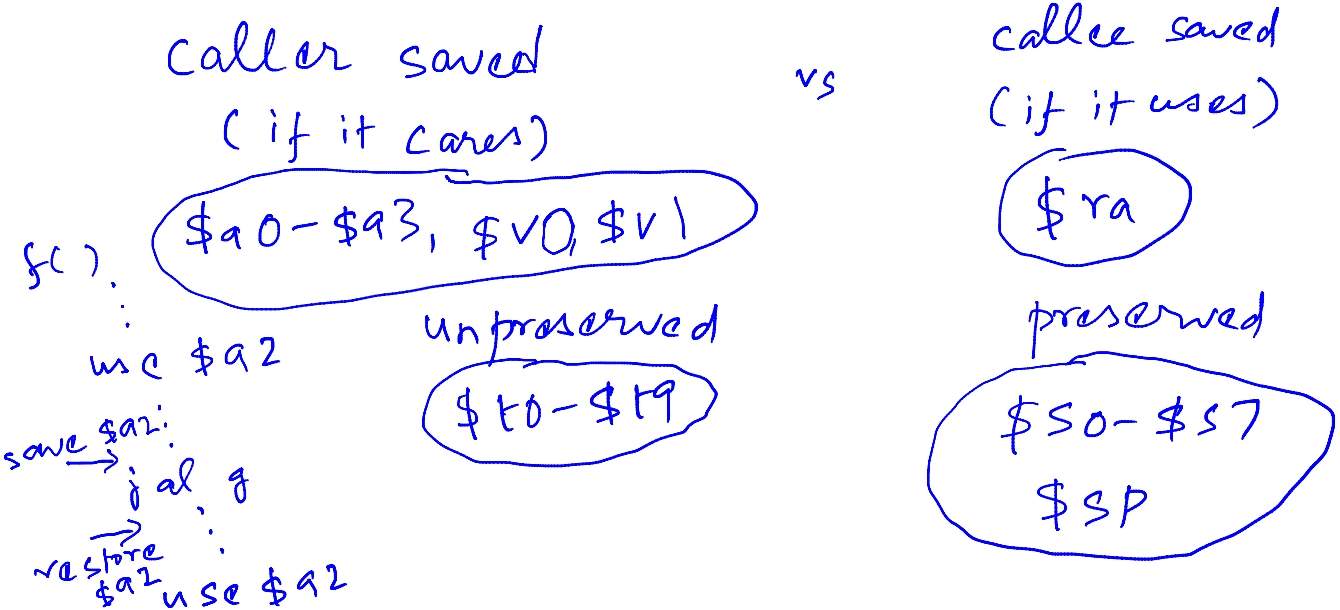
$a0-$a3, $v0, $v1 – caller has to save and restore if caller cares

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

- use stack for these

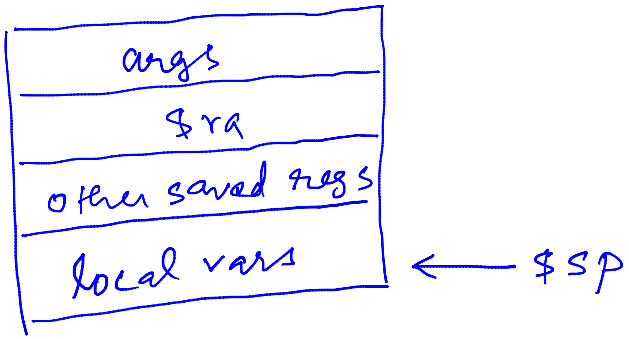
Caller vs Callee Saving Conventions

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

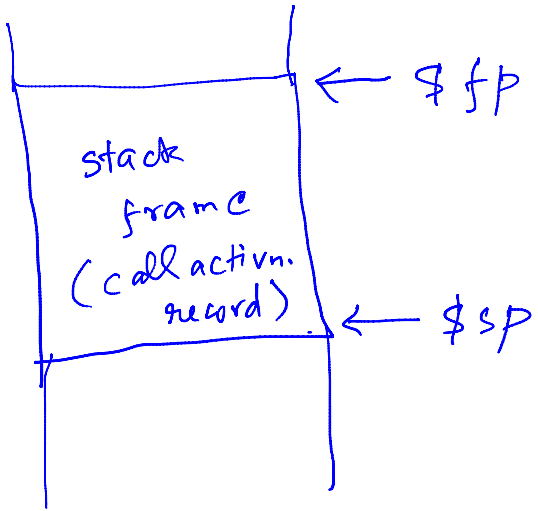


Stack Frame, or Call Activation Record

More arguments than 4 x 32 bit? Use stack
More return value than 2 x 32 bit? Use stack
Saving registers? Use stack
Local variables? Use stack



The Frame Pointer



← $fp – makes it easier for compiler

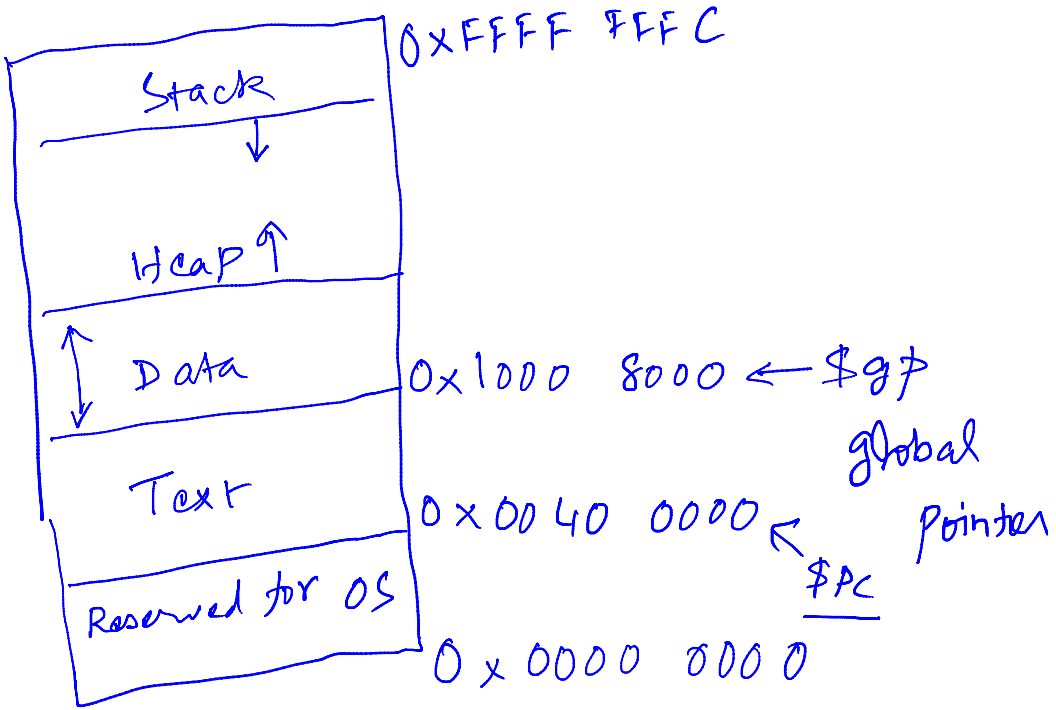
← $sp

```
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

beq $zero
beq bne
beq bnz

beq $s0, 10, EXIT

↓

assembler temporary

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

j FAR-AWAY

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

lb, lh
sb, sh

lbu, lhu
sbu? shu?

Summary So Far

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

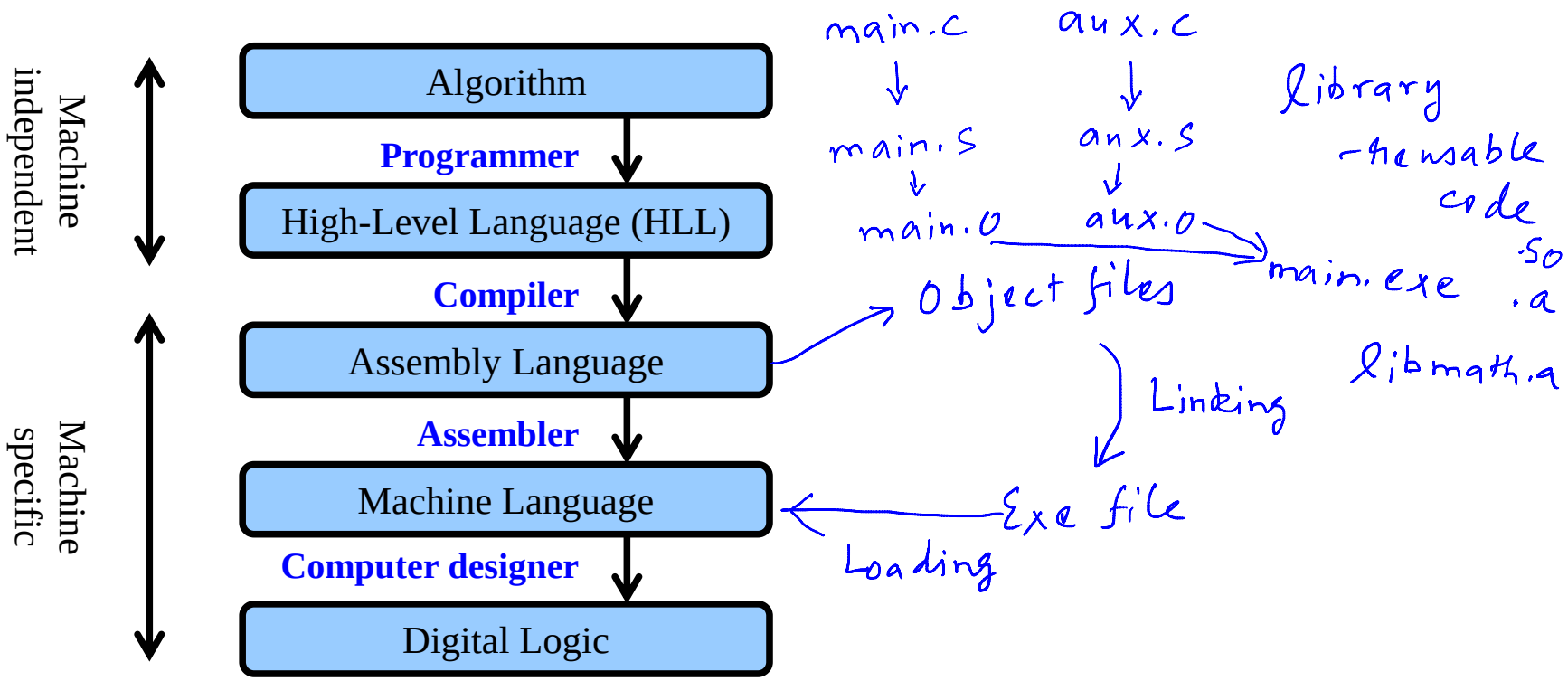
CS305
Computer Architecture

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

Boot loader
↓
OS
↓
programs

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

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: **lwc1, ldc1, swc1, 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

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

$x == y$

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 — % past failed attempts
reliability — % past delays
cost, per unit weight
max weight

delivery confirmation
online tracking
#destinations on offer
pickup service

Computer Performance Equation

customer = one executing program

metric - execution time ✓
response time Linux: time

Sysad: throughput
multi-user environment

Prog. execn. time	= #cycles x clock-cycle-time
	= #cycles / clock-frequency
Prog. execn. time	= #instrns x [#cycles/instrn] x clock-cycle-time

executed

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 million x CPI x (1.2 t) slower

Execn time in B: 5.5 million x CPI x 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?

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

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?

ET1: $(2.4 \times 10^6) \times t$

ET2: $(2 \times 10^6) \times 1.1t$ faster

Measuring the Factors in the Performance Equation

#instrns x CPI x cycle time

↓ executed
- profiling
- Simulators
- h/w counters

↘ instruction mix

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

Types of instrns:
- arithmetic
- memory
- branch

$0.25 \times 2 + 0.75 \times 1 = 1.25$

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?

ET1: $(8\times2 + 8\times3 + 2\times3 + 8\times1) \times t$

ET2: $(2\times2 + 14\times3 + 2\times3 + 8\times1) \times t$

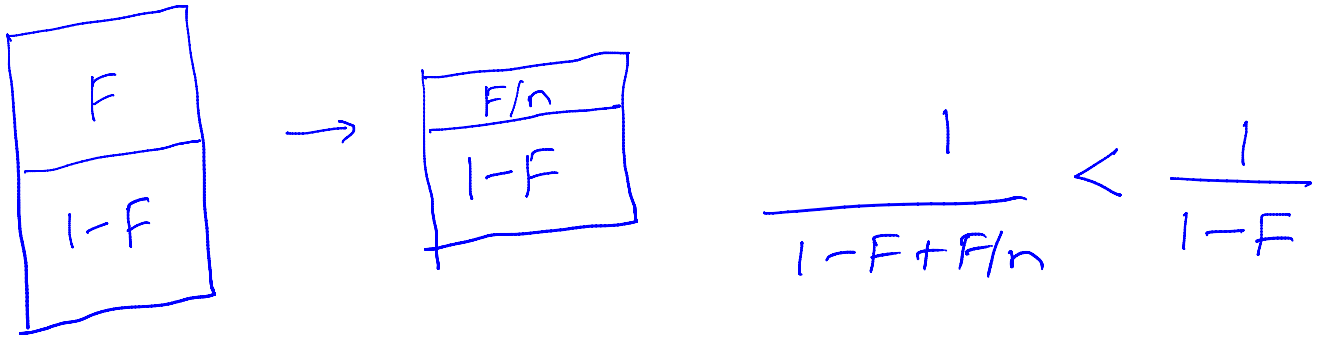
faster: $\frac{60}{54}$

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



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

Speedup 2 : $\frac{1}{1-0.5+\frac{0.5}{2}} \simeq 1.33$ ✓

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

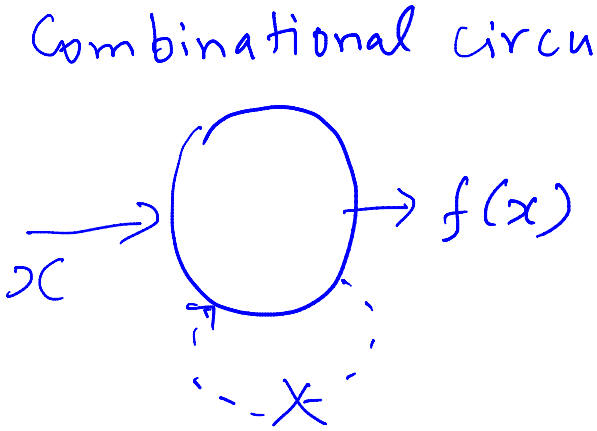
Hardware Implementation of MIPS: Preliminaries

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

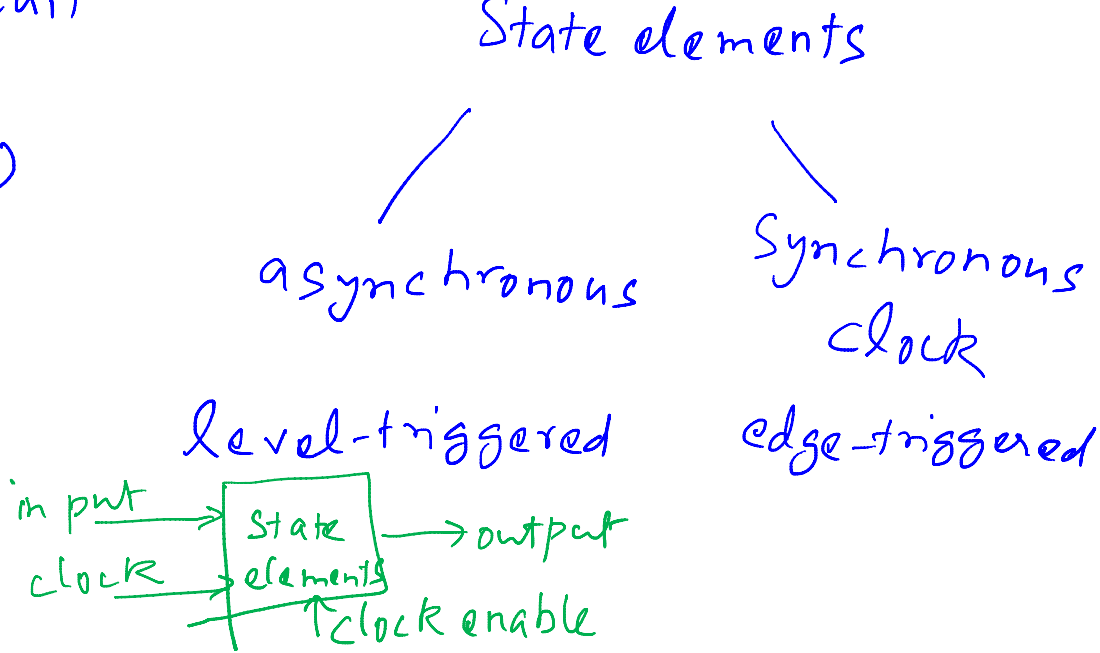
http://www.cse.iitb.ac.in/~br

Recall: Combinatorial vs Sequential Circuits

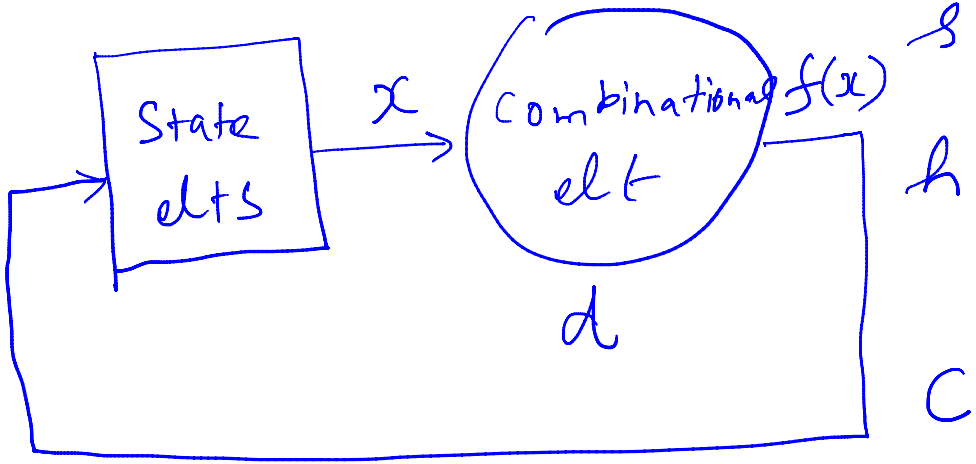
Combinatorial circuit



Sequential circuit



Sequential + Combinational Circuit

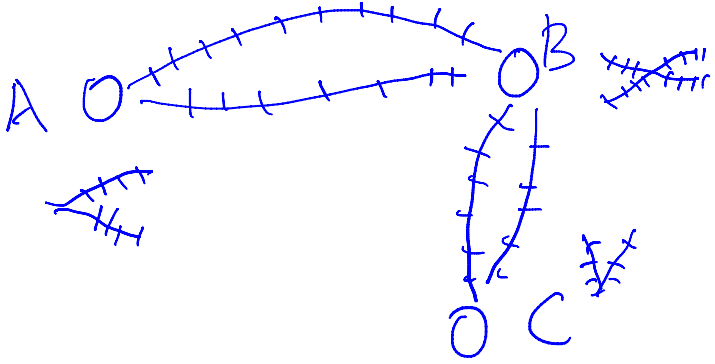


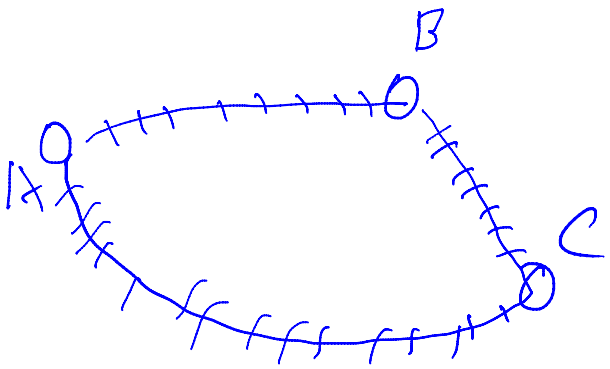
Setup time

hold time

$C > d + s + h$

Control Path, Data Path Analogy





Datapath

- platforms
- rail lines

Control path / signals

- trains should not collide
- no deadlock

Control Path, Data Path in MIPS Implementation

Data path

- state elts
- combinational elts
- 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

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

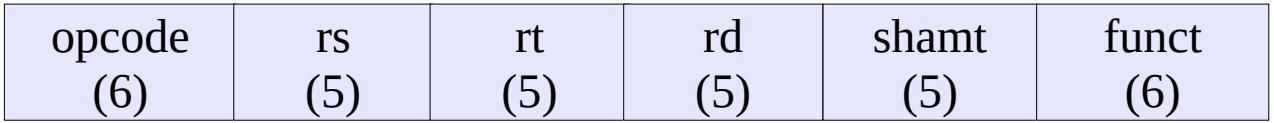
MIPS ISA Subset

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

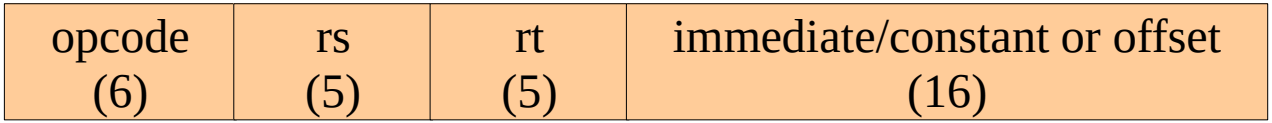
Before Proceeding, Test Your Understanding

- 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`** ?

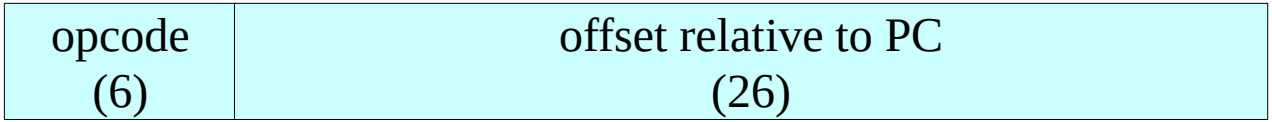
Recall: MIPS Instruction Format



R-type instruction: register-register operations

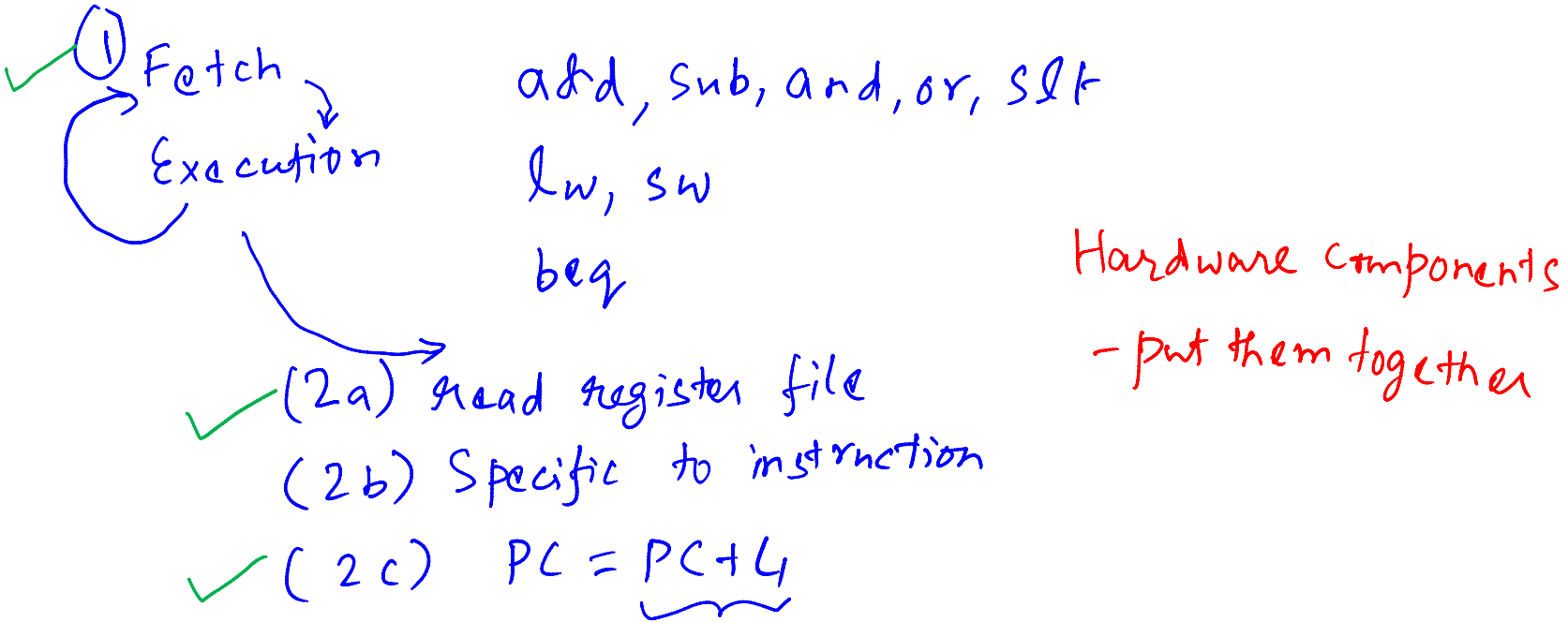


I-type instruction: loads, stores, all immediates, *conditional branch, jump register, jump and link register*

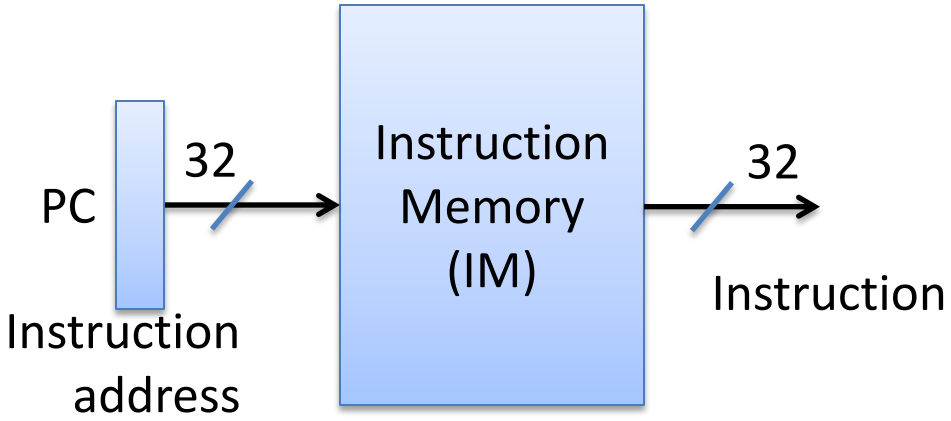


J-type instruction: *jump, jump and link, trap and return*

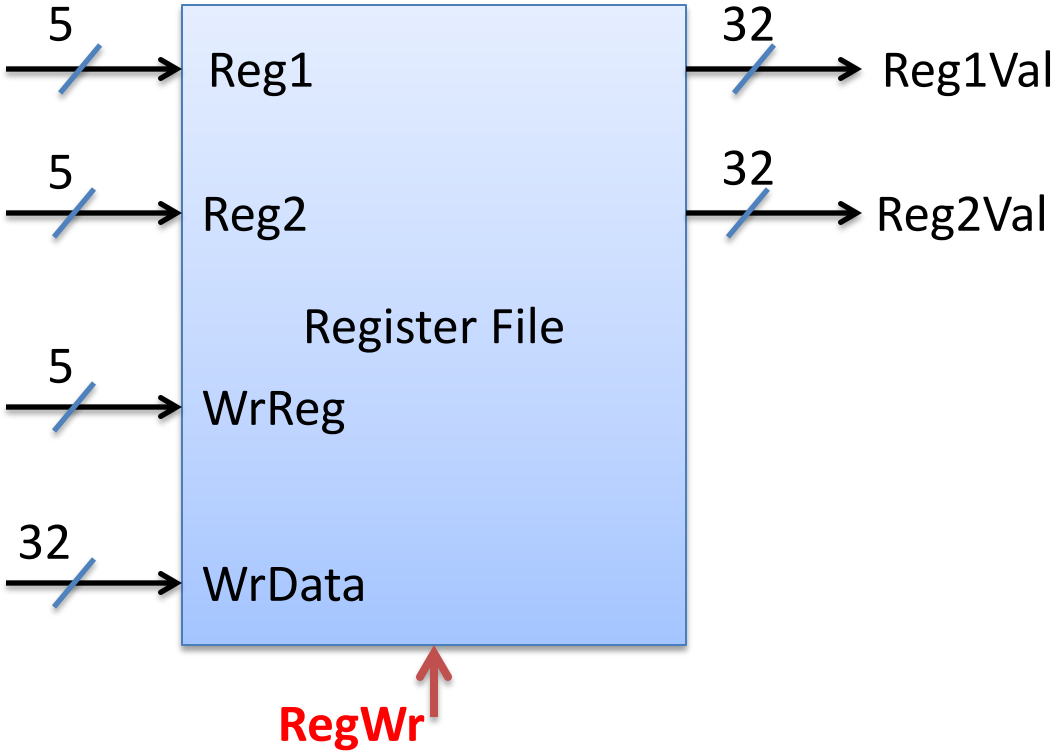
Steps in Program Execution



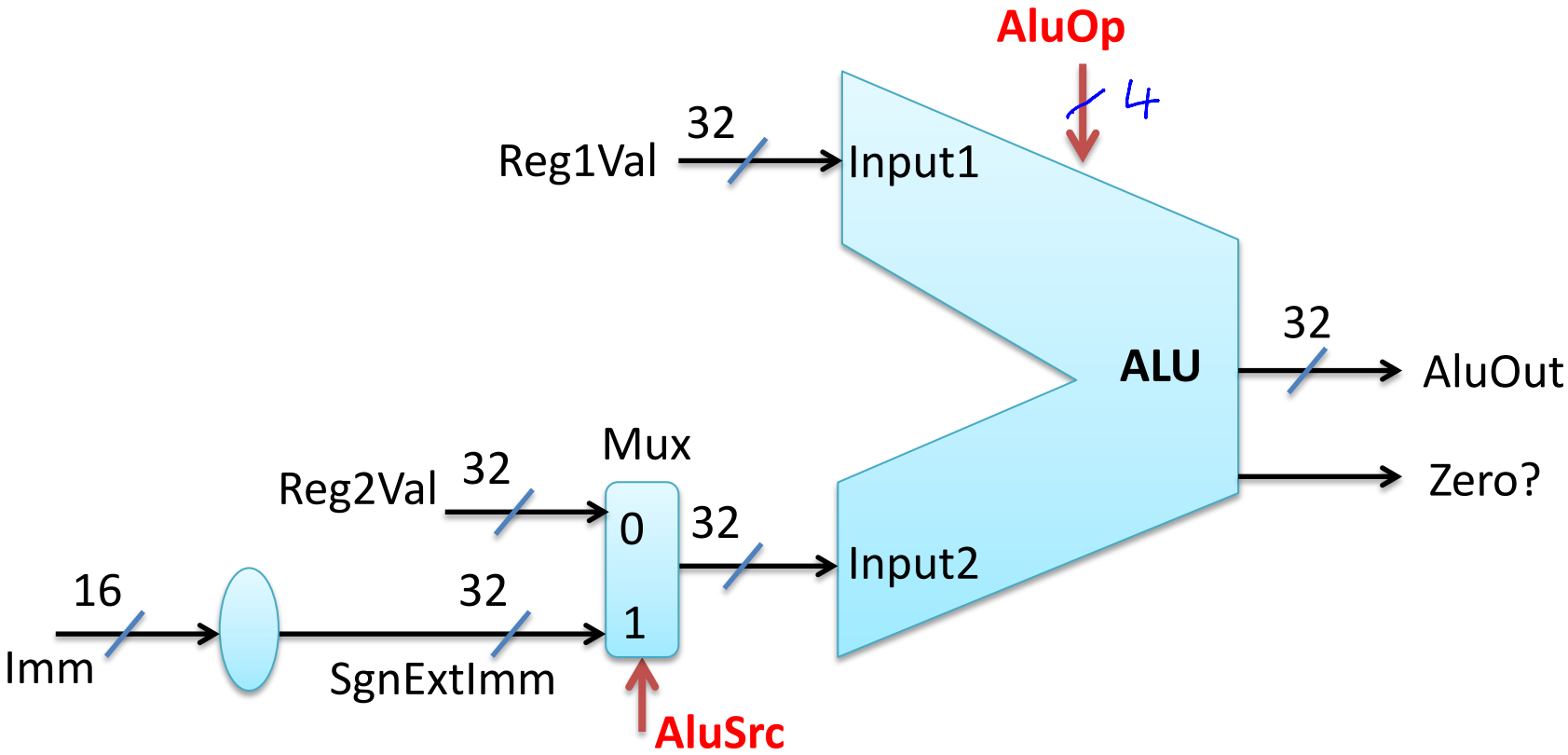
Elements for Instruction Fetch



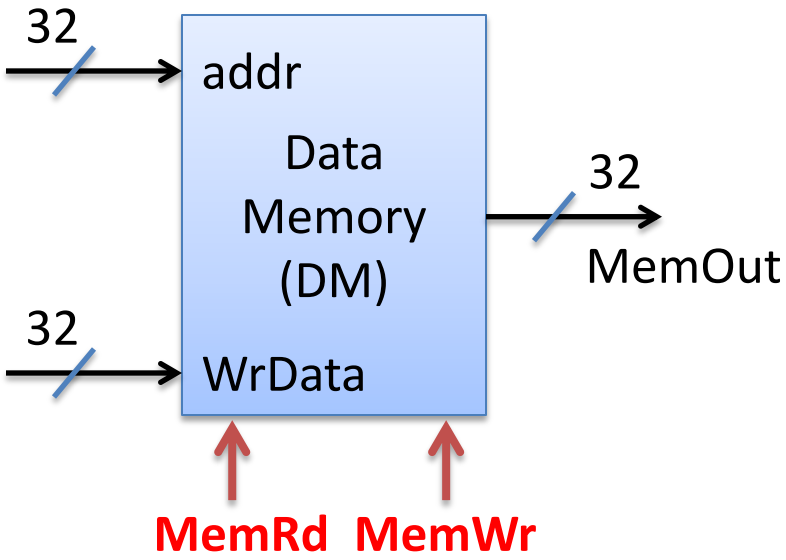
Element for Register Read/Write: The Register File



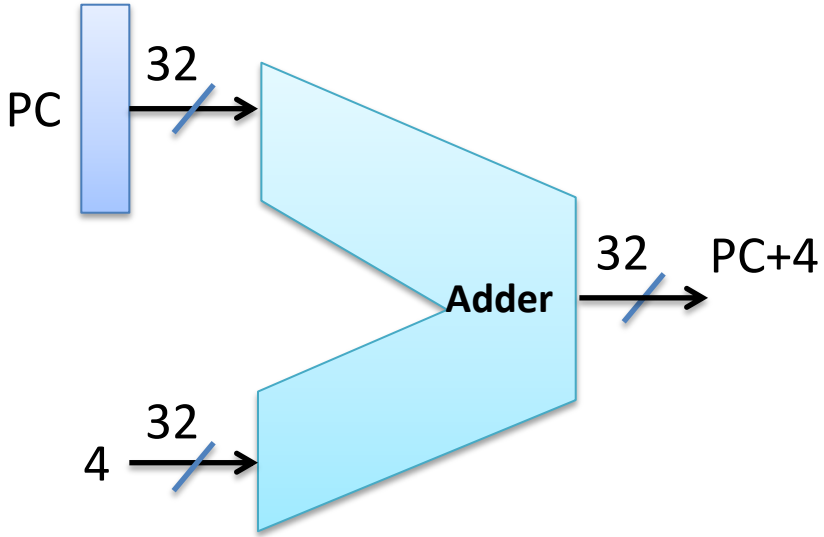
ALU: Arithmetic Logic Unit



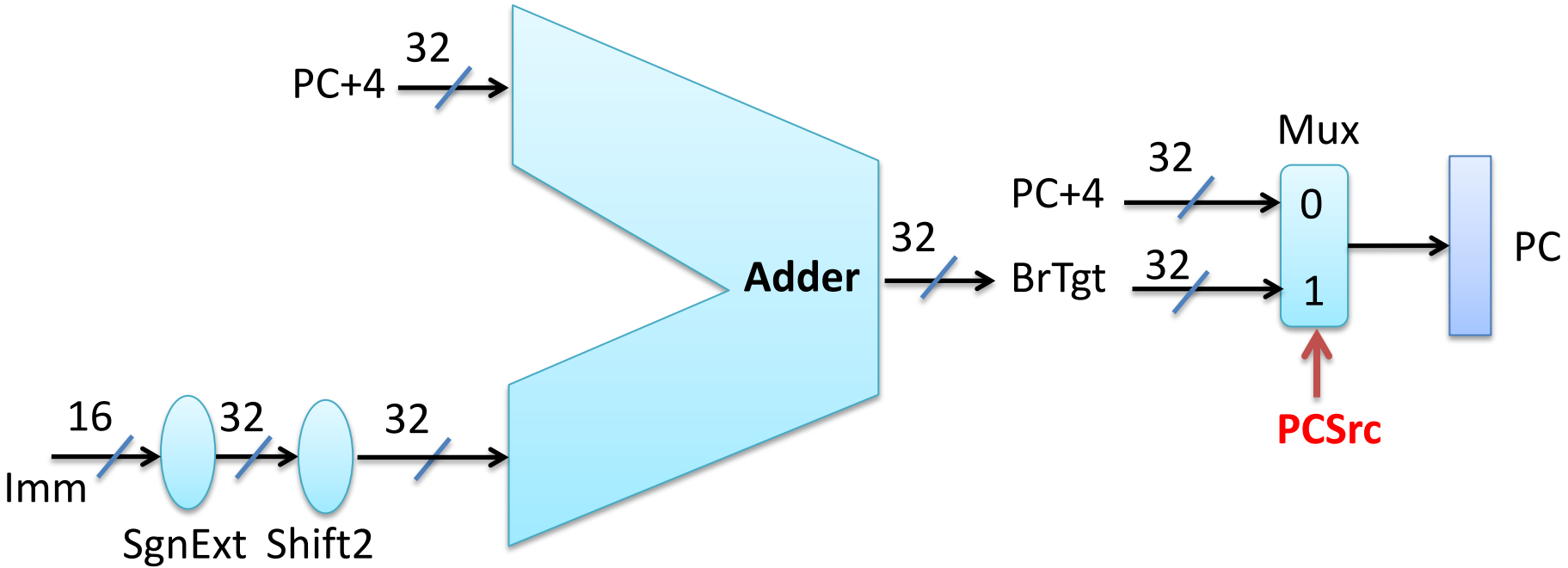
Data Memory



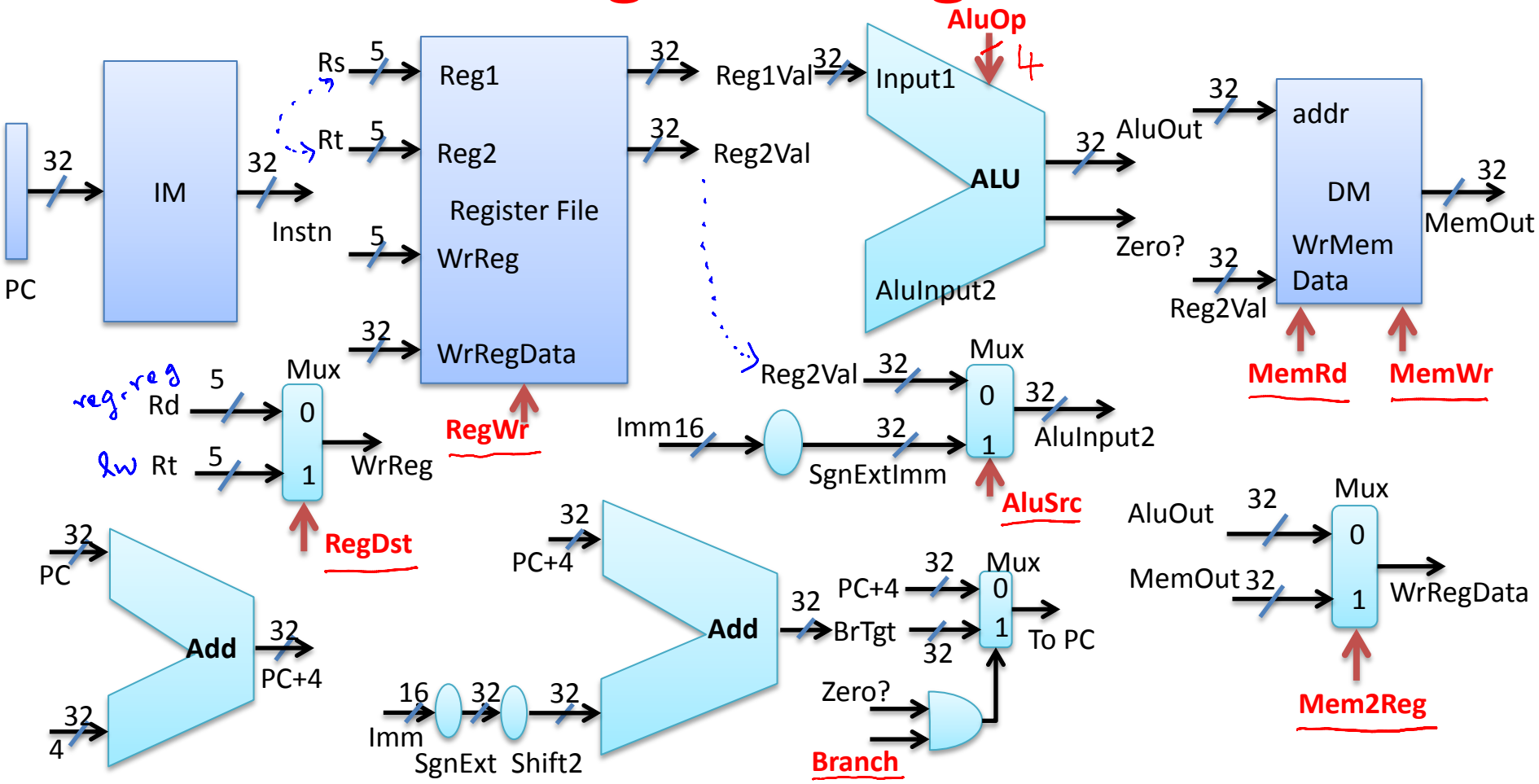
Element for PC+4 Computation



Additional Elements to Implement beq



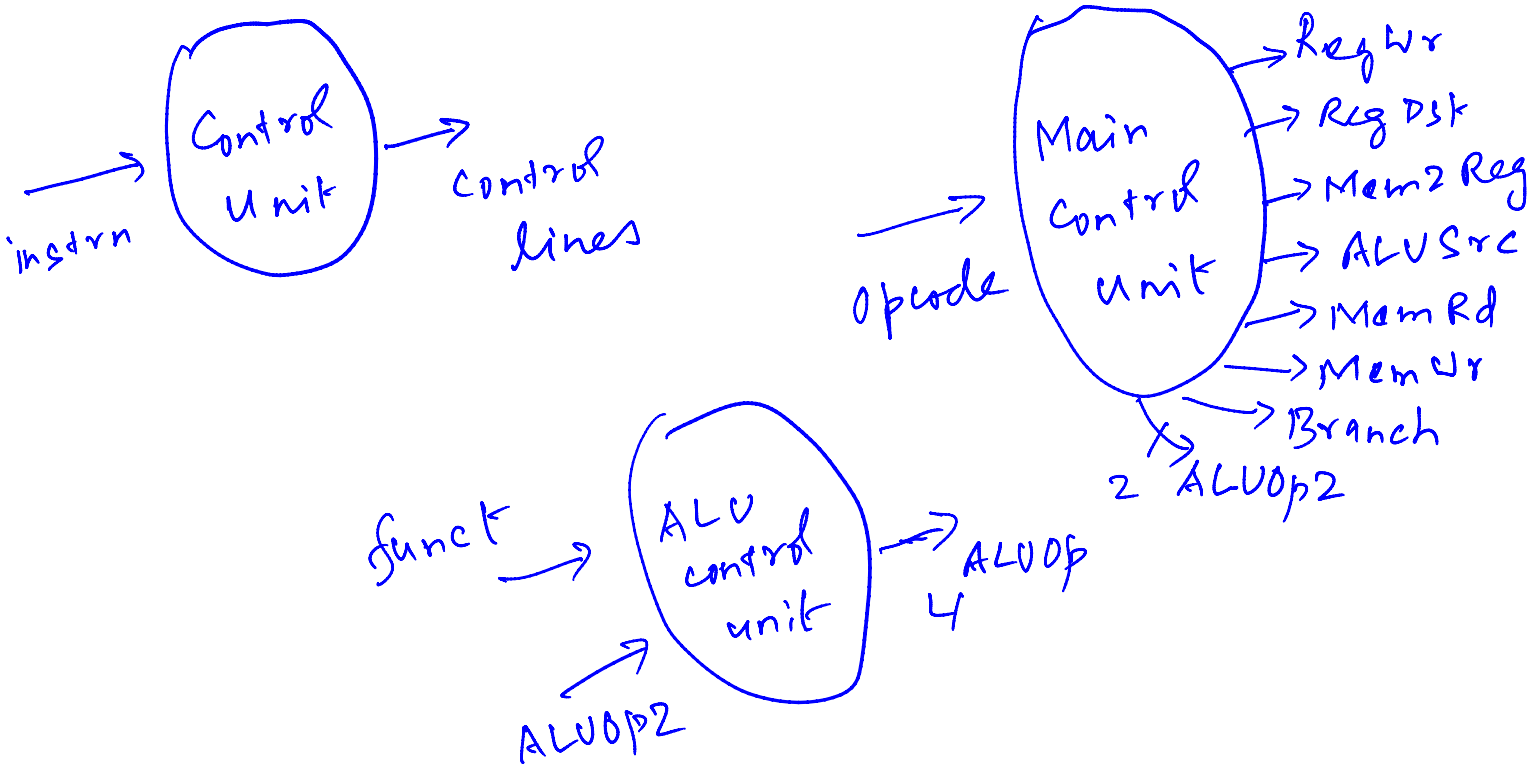
Putting it All Together



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 + funccode
- 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

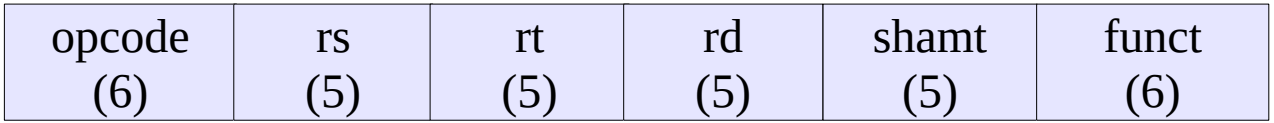
MIPS ISA Subset

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

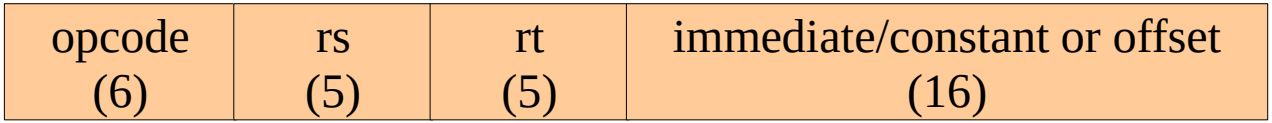
Before Proceeding, Test Your Understanding

- 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`** ?

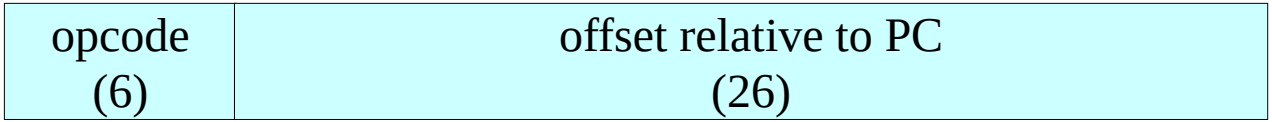
Recall: MIPS Instruction Format



R-type instruction: register-register operations

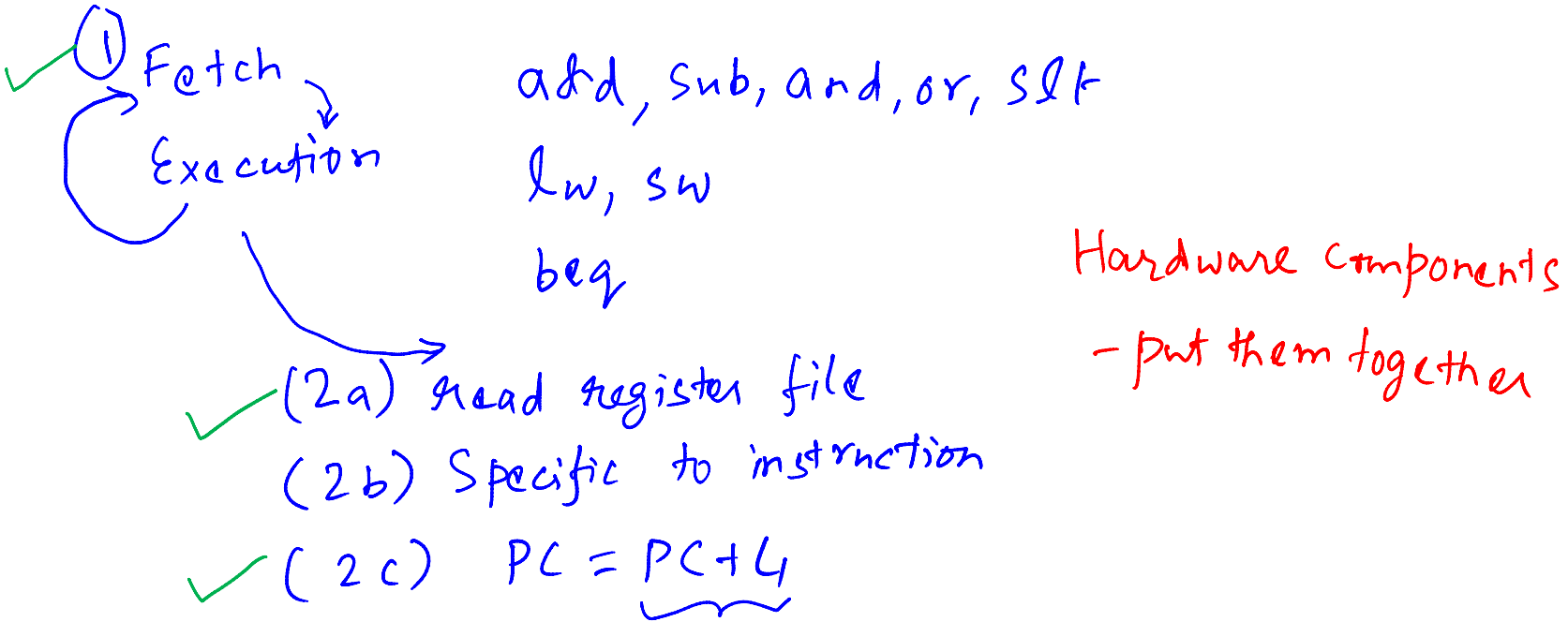


I-type instruction: loads, stores, all immediates, *conditional branch, jump register, jump and link register*

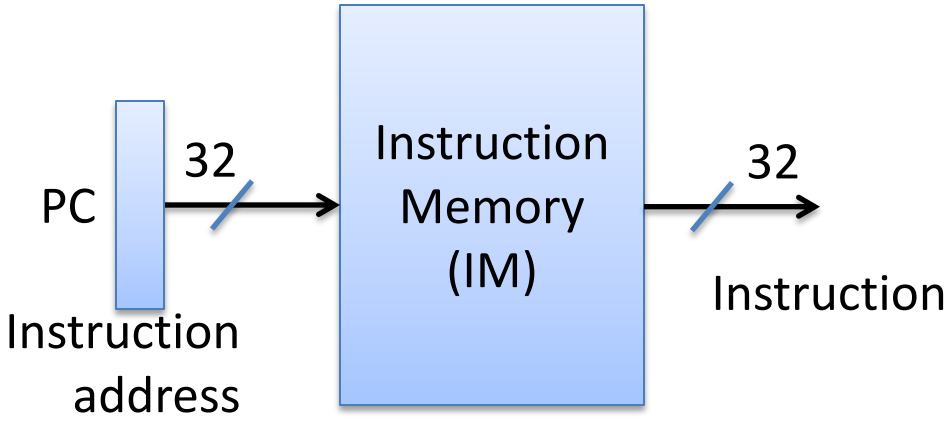


J-type instruction: *jump, jump and link, trap and return*

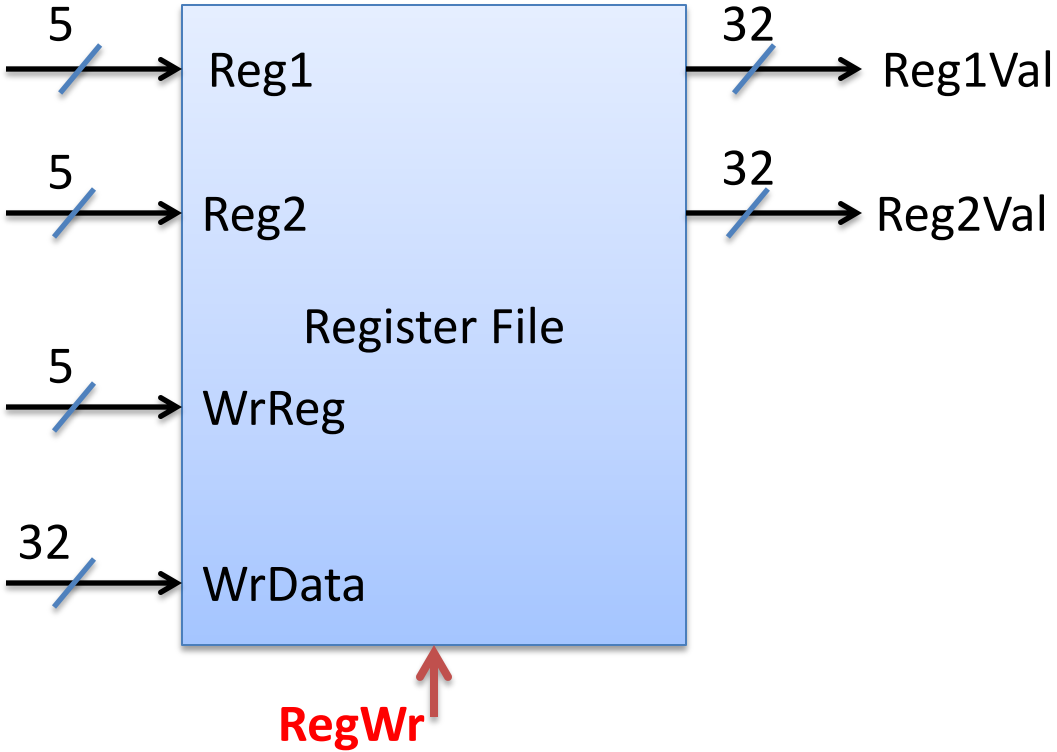
Steps in Program Execution



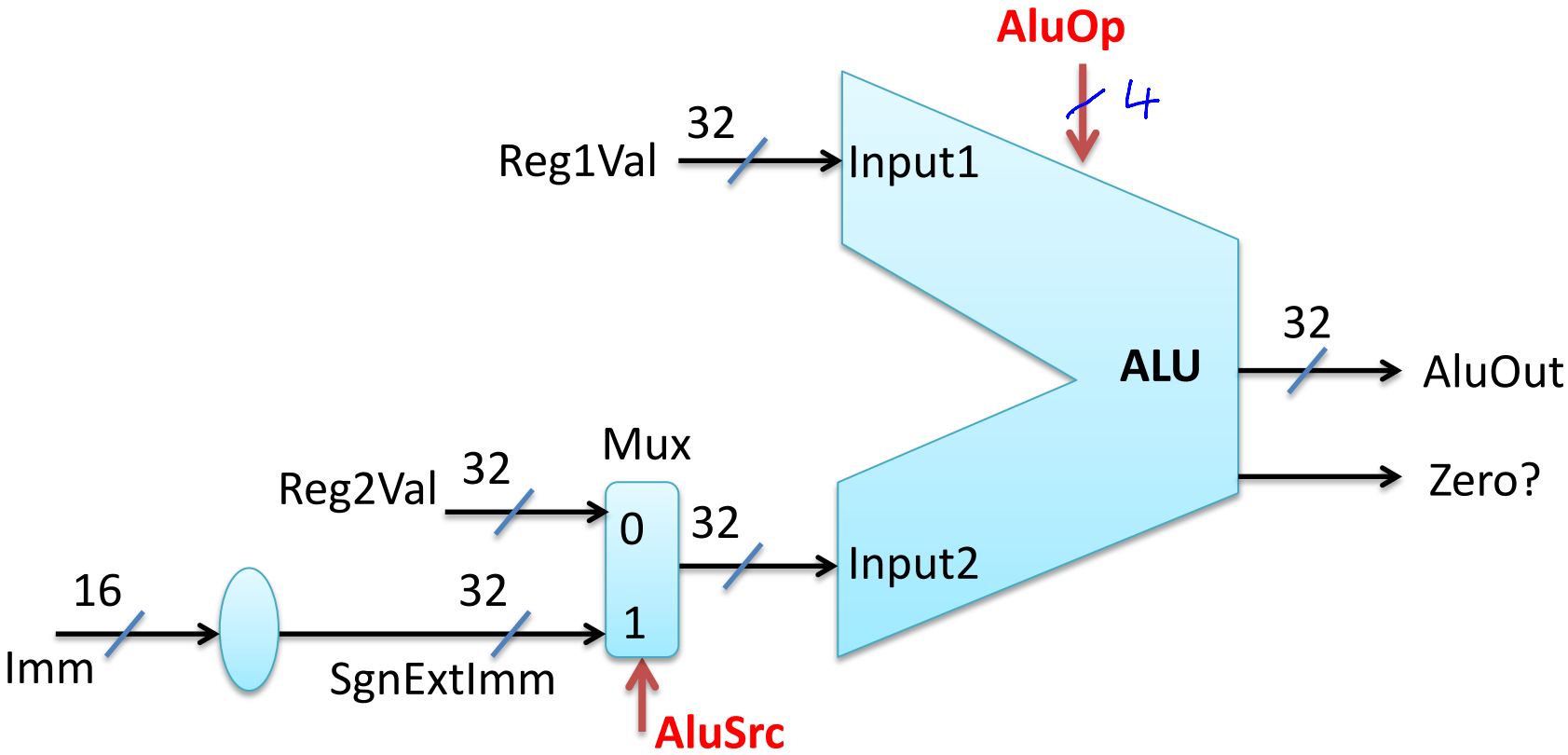
Elements for Instruction Fetch



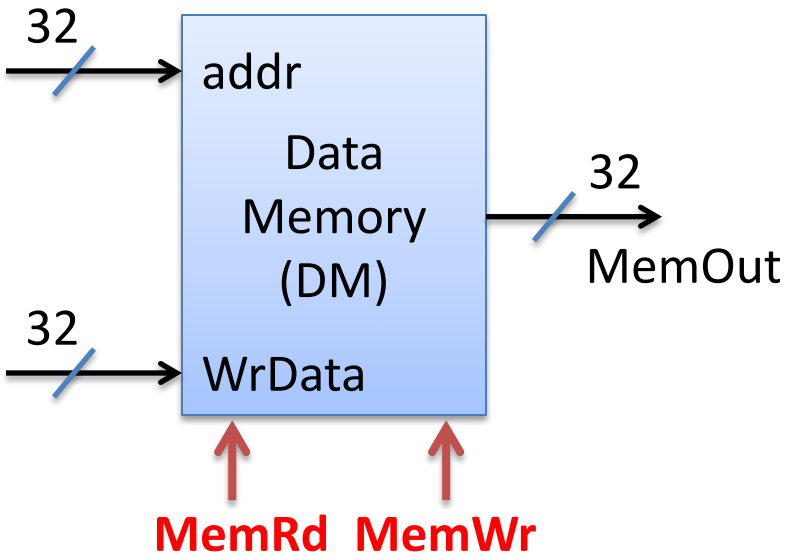
Element for Register Read/Write: The Register File



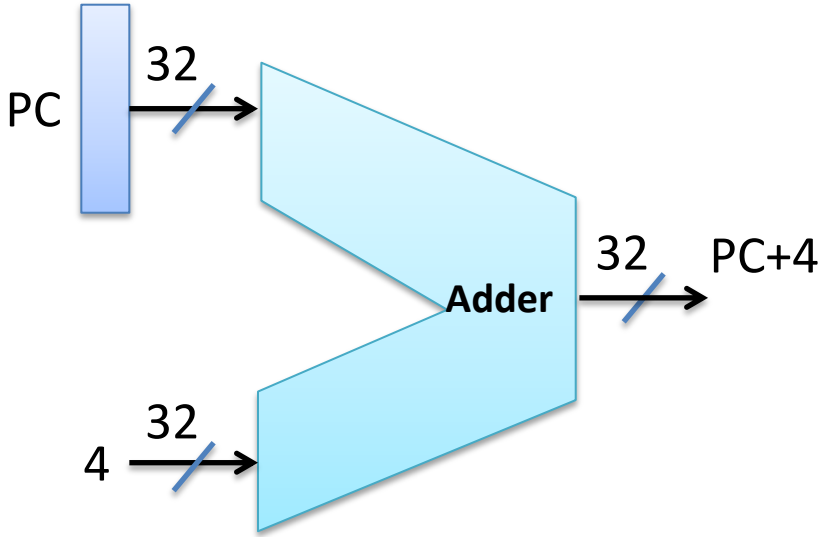
ALU: Arithmetic Logic Unit



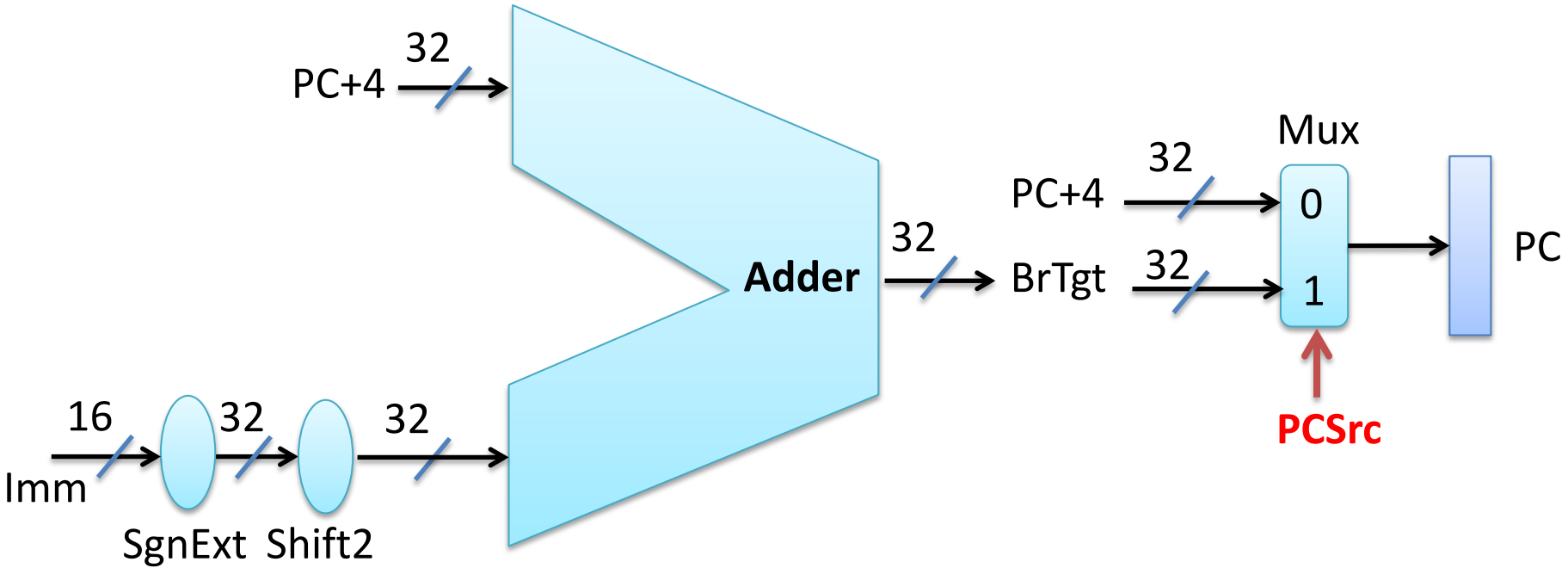
Data Memory



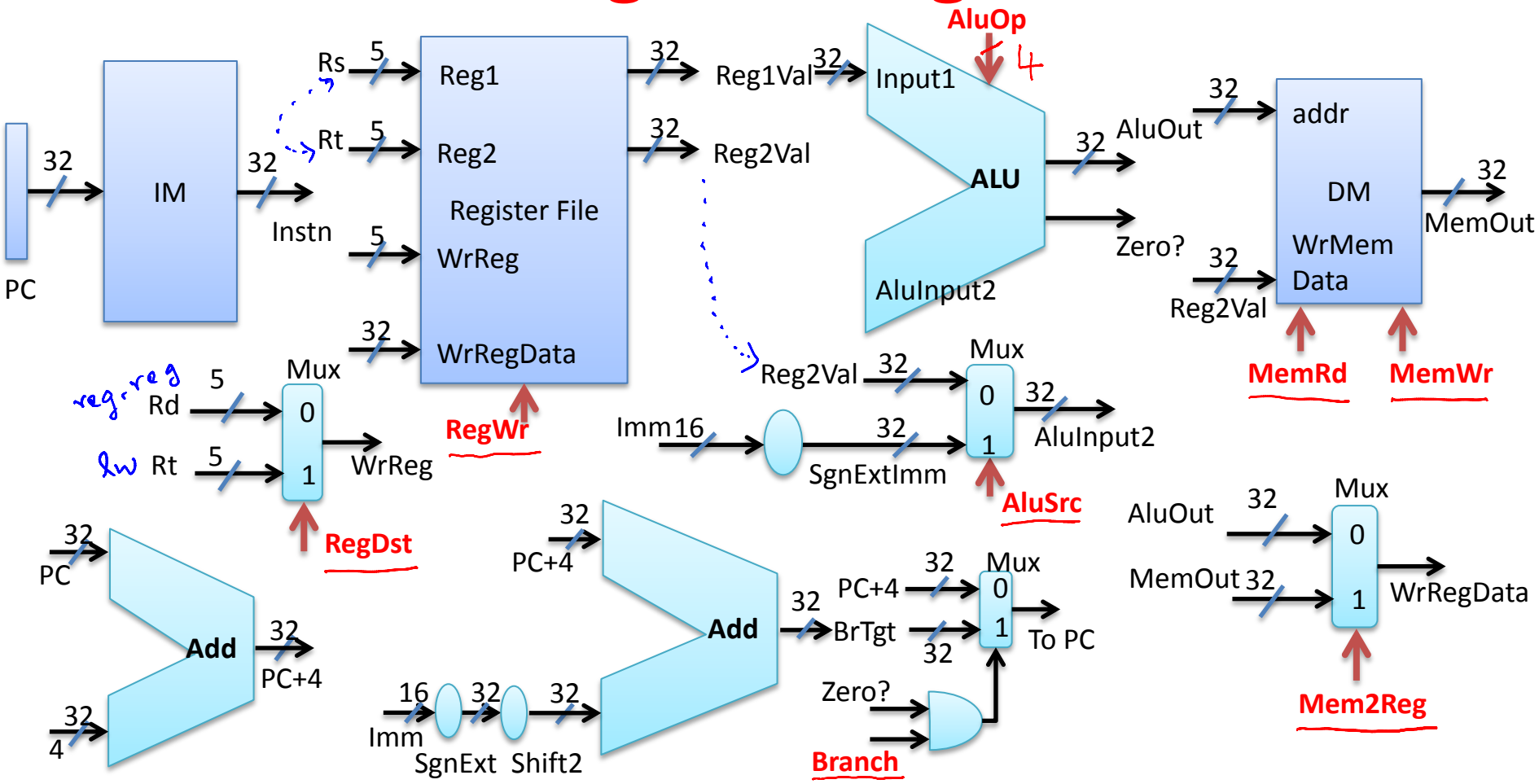
Element for PC+4 Computation



Additional Elements to Implement beq



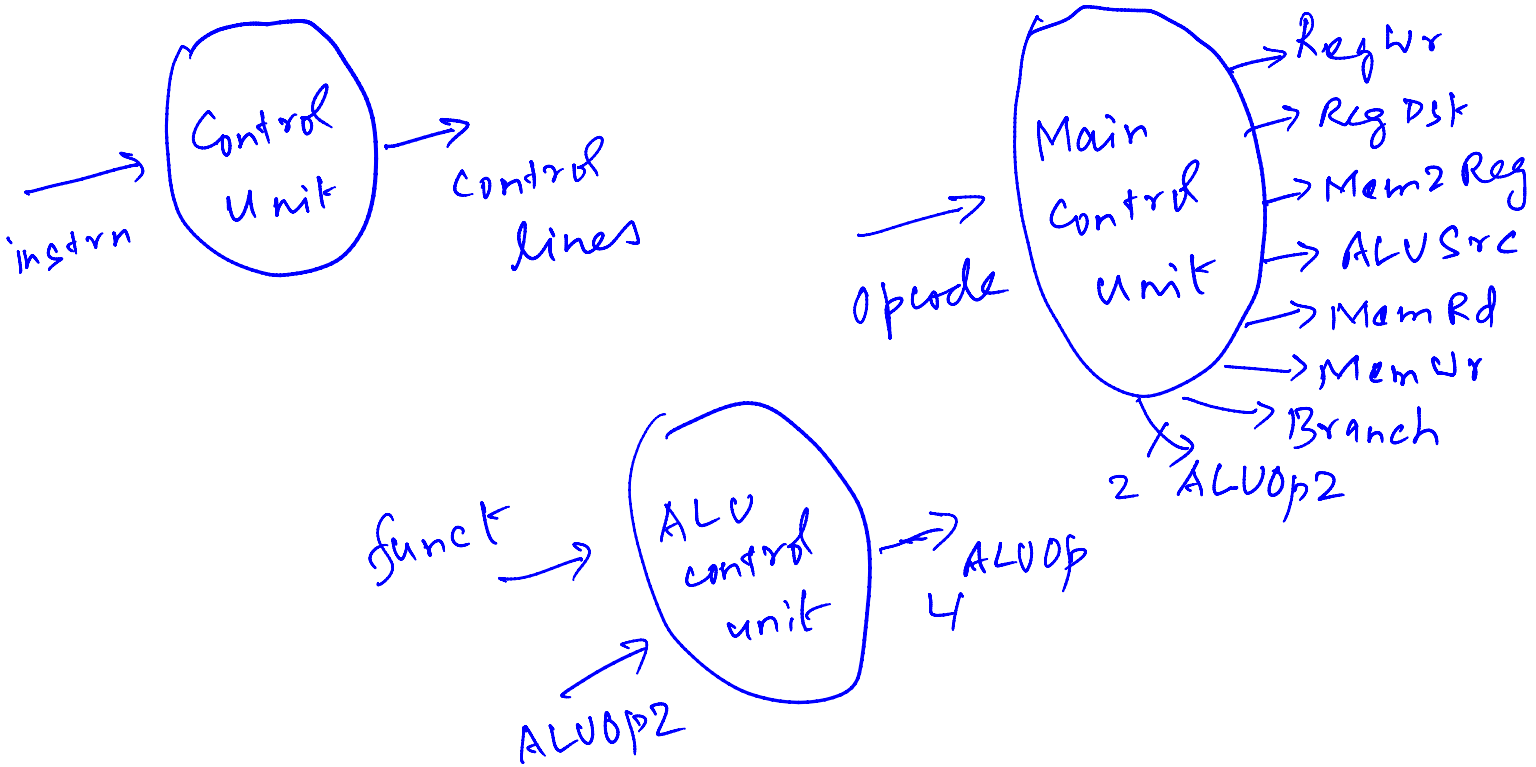
Putting it All Together



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 + funccode
- Next: extending the implementation to support other instructions

CS305
Computer Architecture

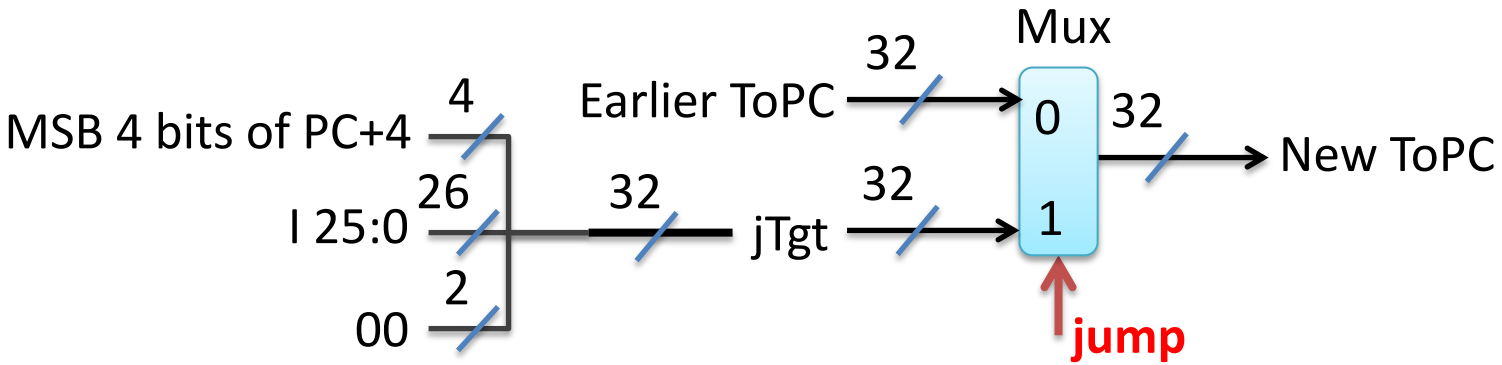
Extending the Single Cycle MIPS Implementation

reg-reg
lw, sw
beq

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

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

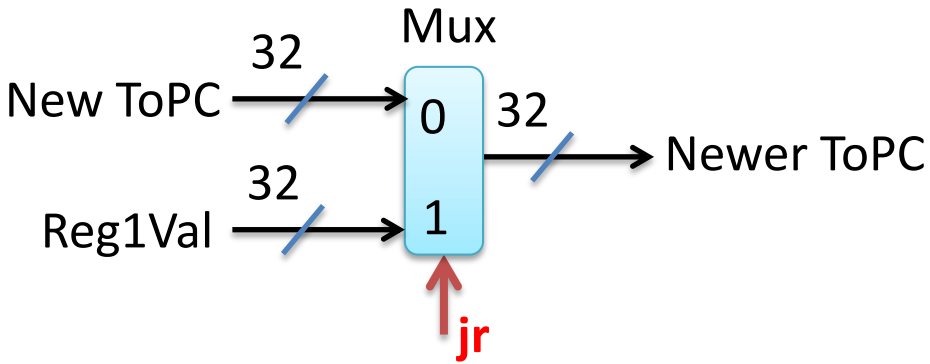
Data Path, Control Path Extensions to Support `j`



Main Control Unit Truth Table Enhancement

	RegDst	RegWr	ALUSrc	Mem-Rd	Mem-Wr	Mem2-Reg	Bran-ch	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

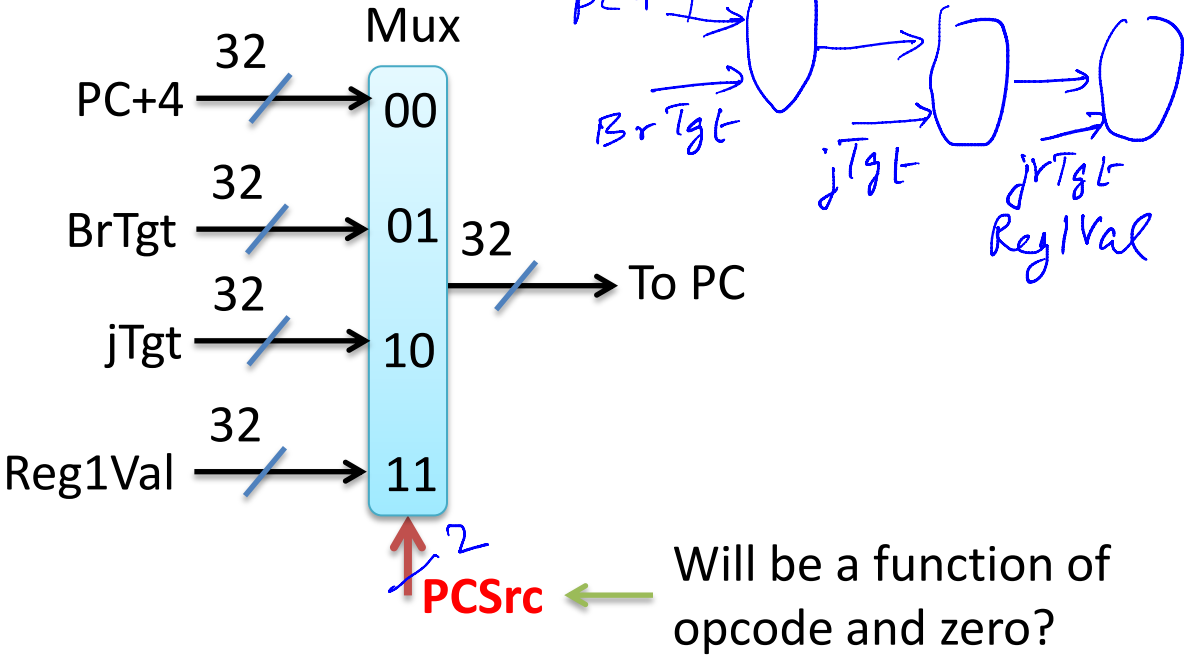
Further Data Path, Control Path Extensions to Support `jr`



Main Control Unit Truth Table Enhancement

	Reg-Dst	Reg-Wr	ALUSrc	Mem-Rd	Mem-Wr	Mem2-Reg	Bran-ch	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)

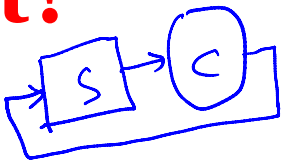
CS305
Computer Architecture

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 x 400 + 0.25 x 400 + 0.2 x 500 + 0.2 x 300

Components involved in:
- Register-register instructions
- **`sw`**
- **`lw`**
- **`beq`**
- **`j`**

`lw` is longest: 5 x 100 ps = 0.5 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