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++

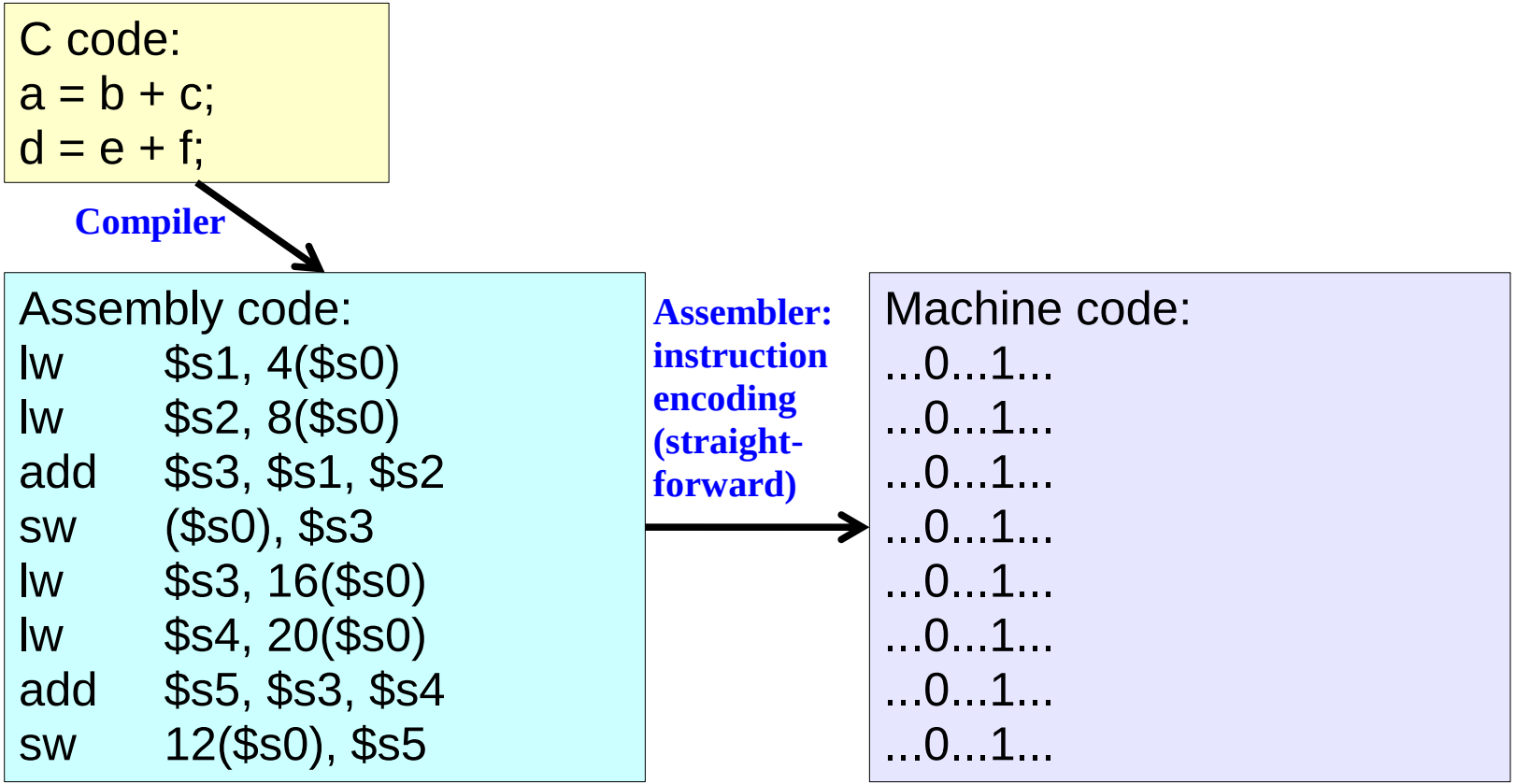
```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

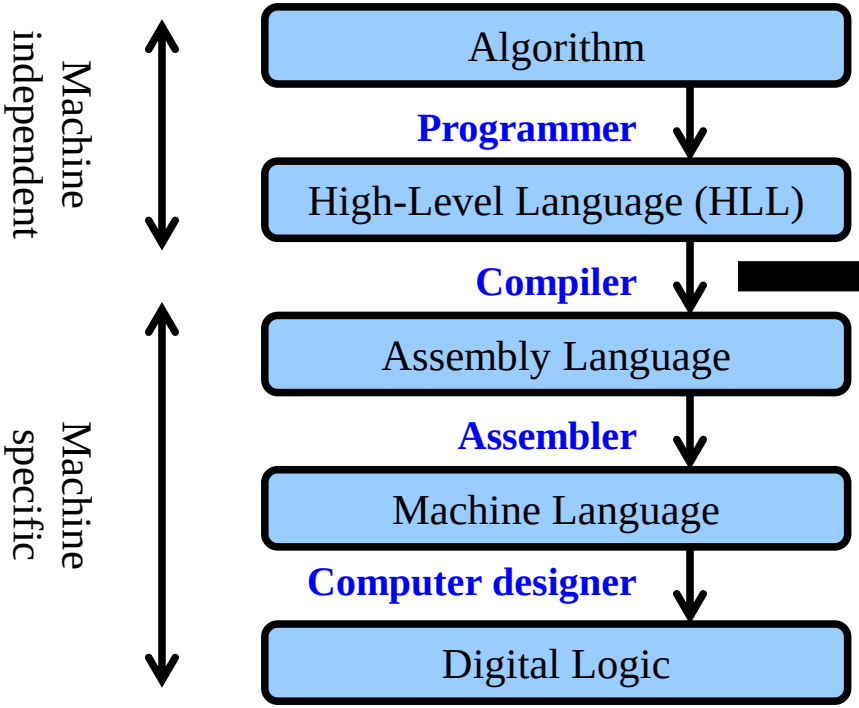
```
$line =~ s/abc/xyz/g;
```

- 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

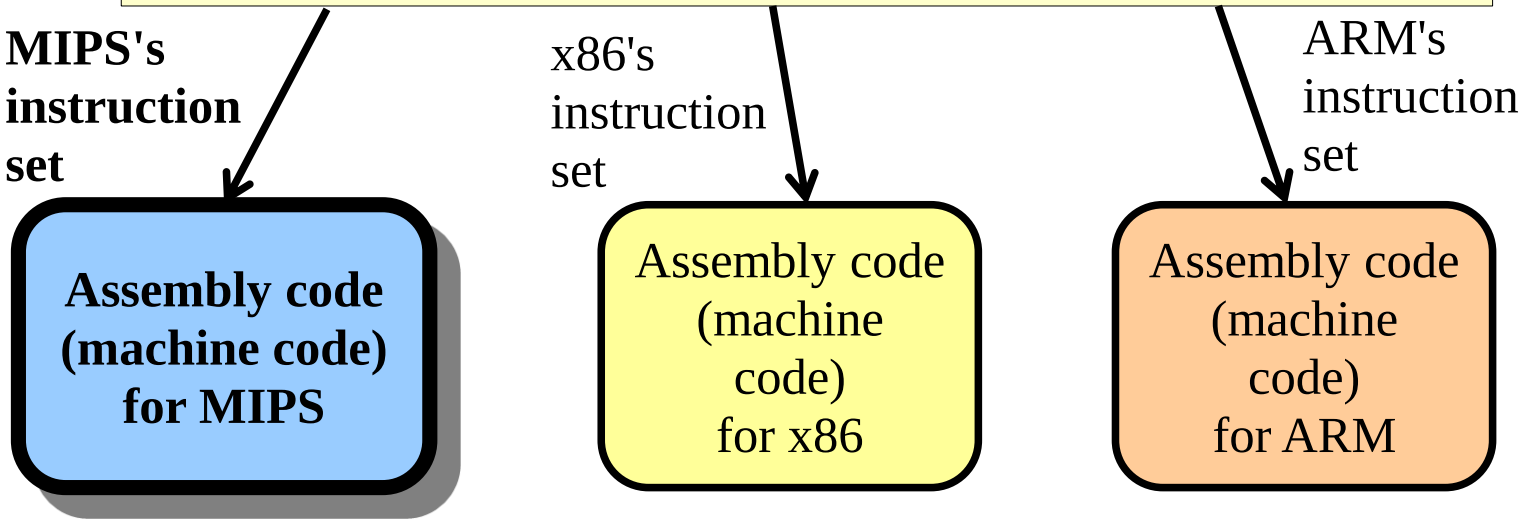
HLL code examples:

C/C++

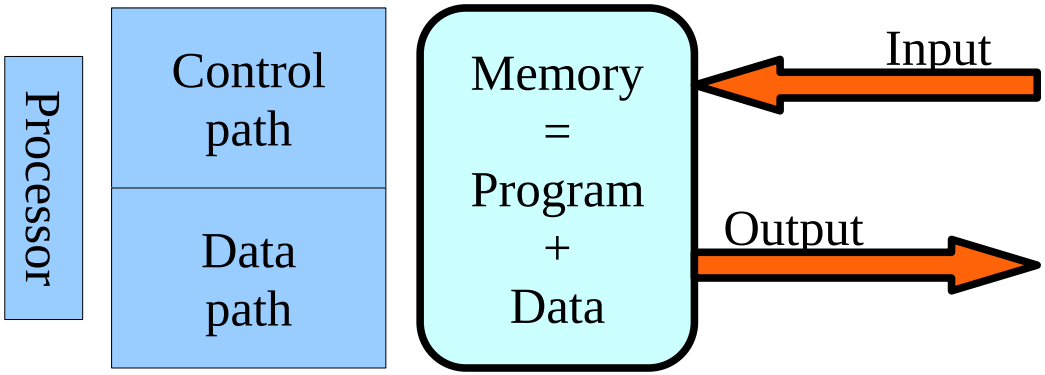
```
f()->next = (g() > h()) ? k() : NULL;
```

Perl

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



Instruction Set and the Stored Program Concept

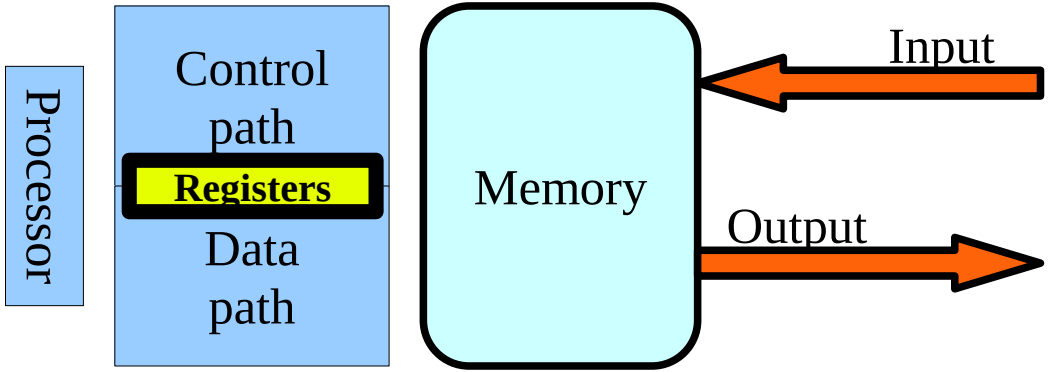


- At the processor, two steps in a loop:
  - Fetch instruction from memory
  - Execute instruction
    - May involve data transfer from/to memory

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)

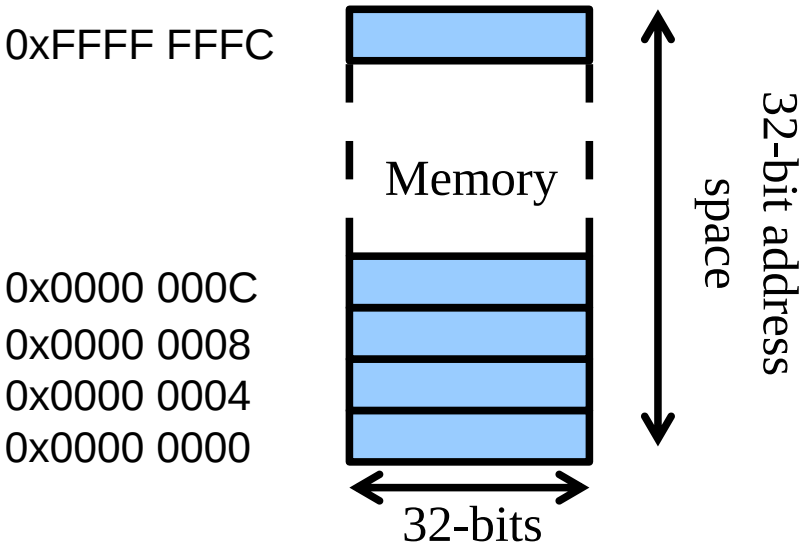
Constant or Immediate Operands

```
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
```

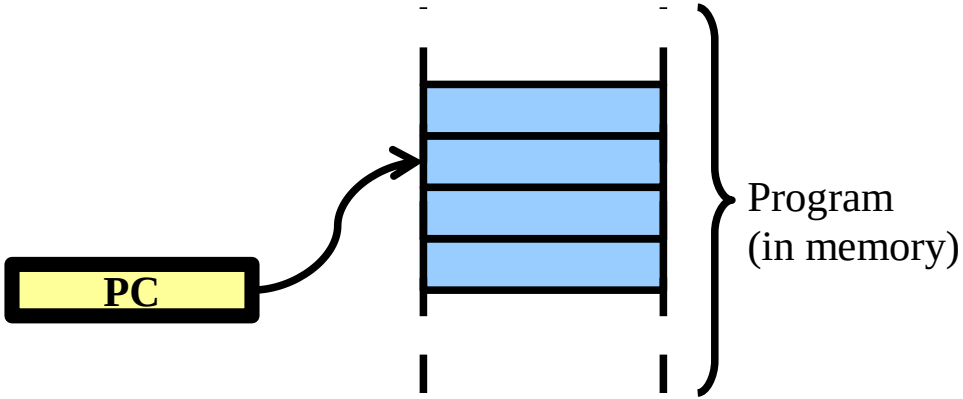
Notion of Register Assignment

- 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?