Architecture and Assembly

Author: Jared Moore

Edited by Nathan Bowman

Topics

1. Computer Architecture
2. Assembly Language Basics
3. Mapping to Hardware

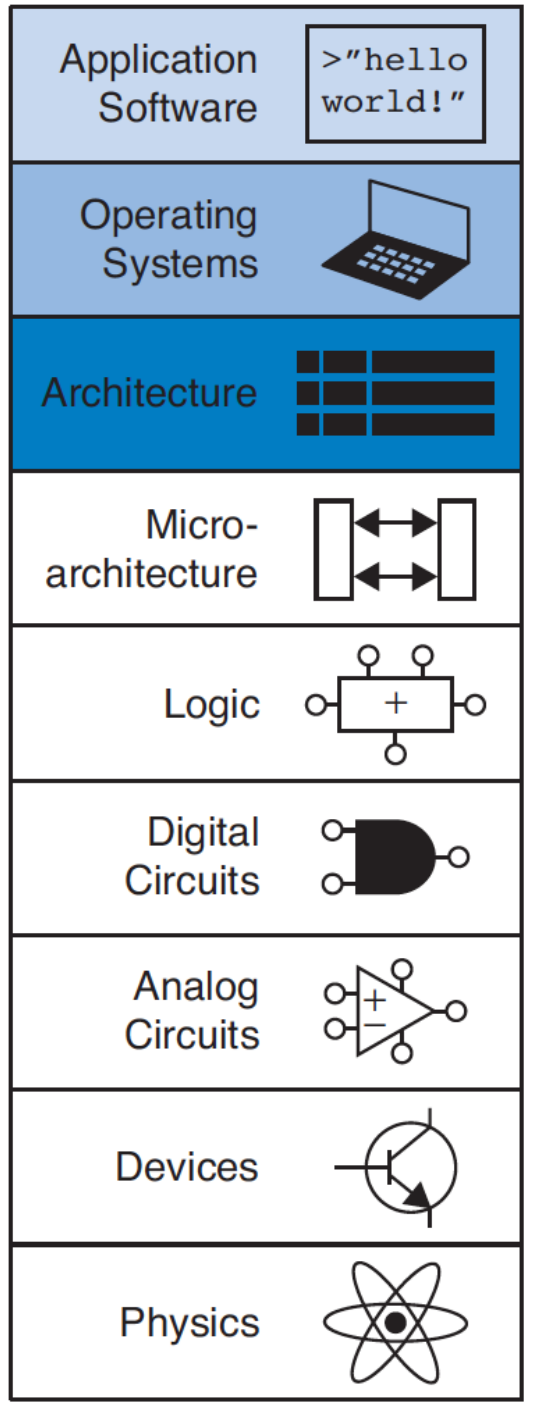
Architecture

Programmer's view of a computer

Abstracts away from physical hardware

Defined by the instruction set (language) and locations (registers and memory)

x86, MIPS, SPARC, PowerPC



Architecture is …

- the commands we can pass directly to the hardware (as in the handout), or
- a set of operations and storage locations, or
- the lowest-level API possible

An architecture is all of these more-or-less equivalent things

Architecture is not…

a specific arrangement of hardware.

It is more like specifying a truth table – we know the inputs and outputs, but we could create an infinite number of circuits to do the same thing.

Instructions

Instruction Set: computer's vocabulary

(All instructions available to it.)

All programs on a computer use the same instruction set!

Basic Commands: Add, Subtract, Jump, etc

Unlike instructions in, e.g., Java, these are things a computer can do

Instructions

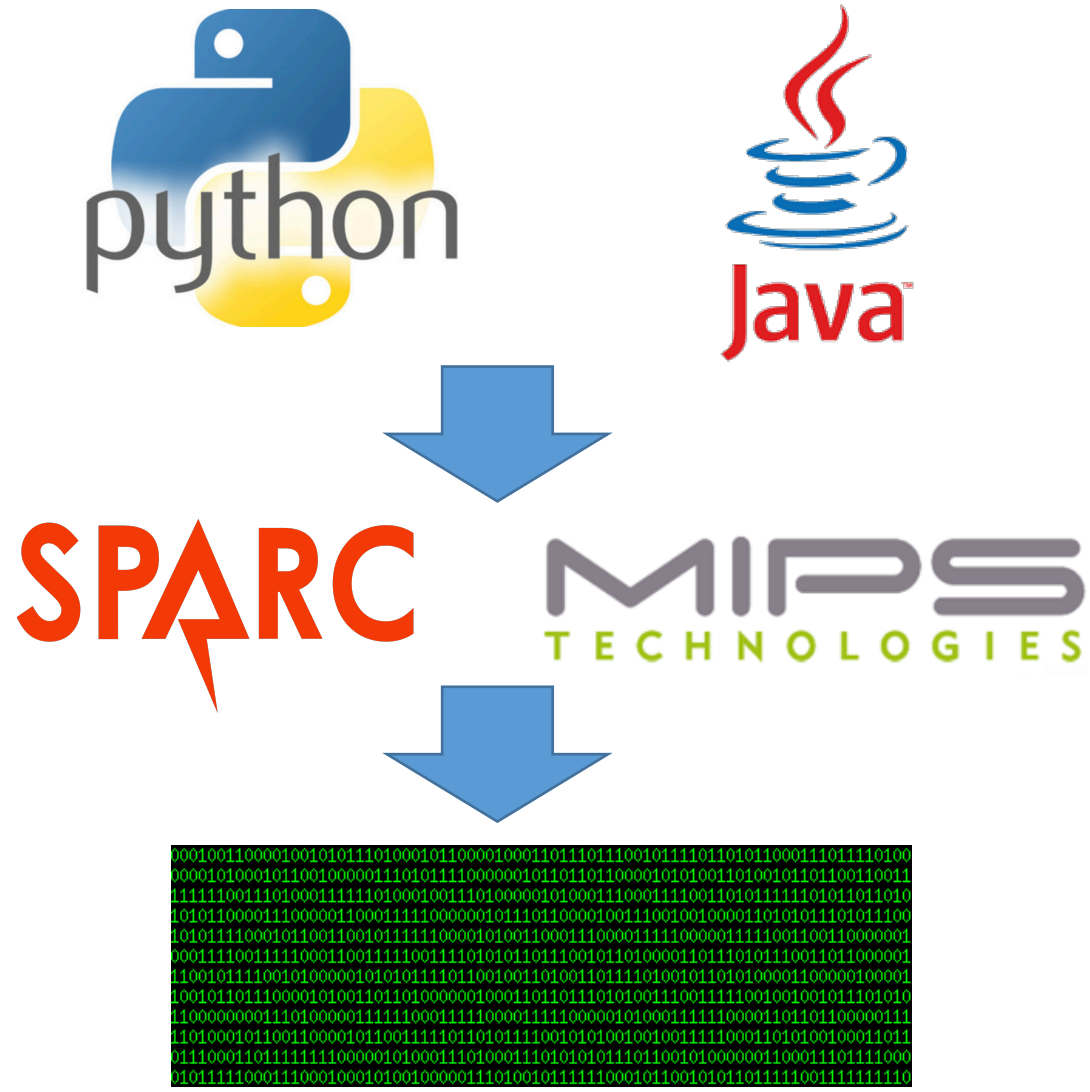
- add $s0, $s1, $s2
- sub $t1, $s1, $t0

Machine Language

Computer's speak in 1's and 0's

Instructions are thus binary numbers

Microprocessors read and execute machine language.



An actual MIPS instruction

- 00000010001100101000000000100000
- add $s0, $s1, $s2

Hardware Implementation

Architecture does not specify hardware implementation!

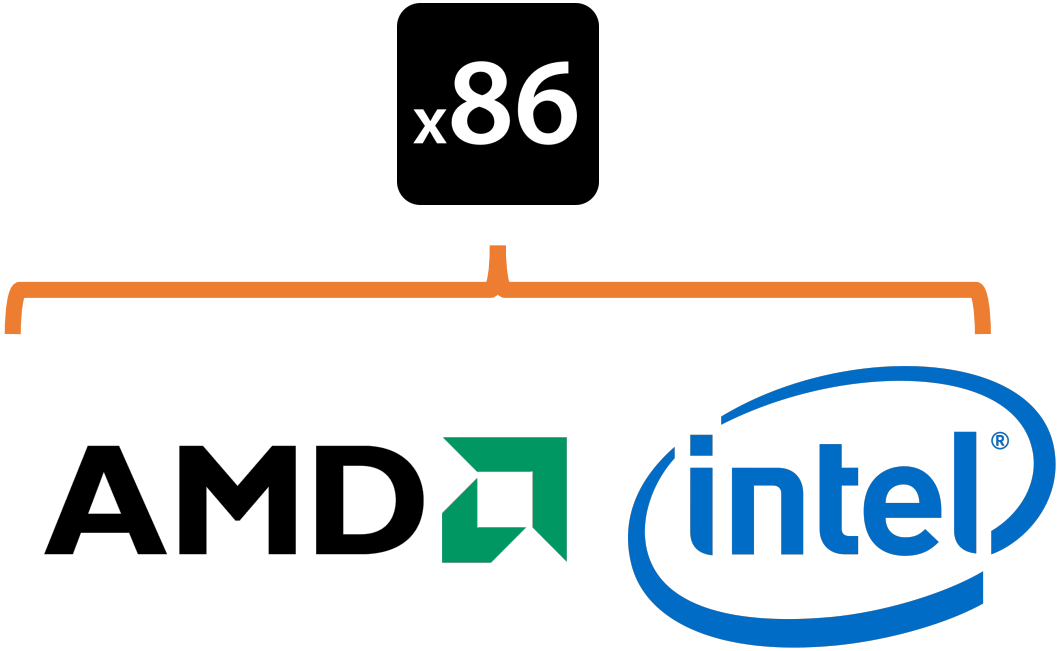
x86

AMD

intel

Microarchitecture

Specific arrangement of registers, memories, and ALUs.
Many different microarchitectures for a single architecture.



Instructions

Mnemonic – which operation to perform.

Source Operands – what operands provide data

Destination Operand – where is the result written to

Classified based on their operands

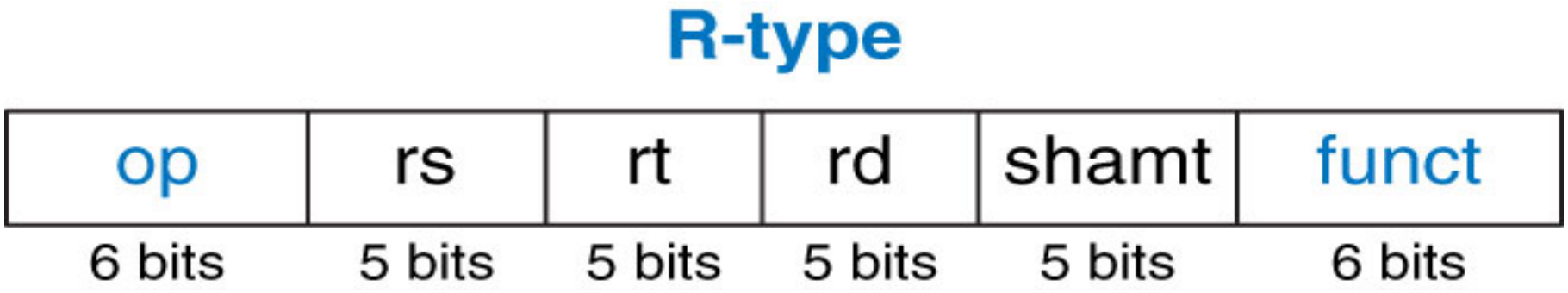
Principle #1

Simplicity favors Regularity.

Instructions have a consistent number of operands.

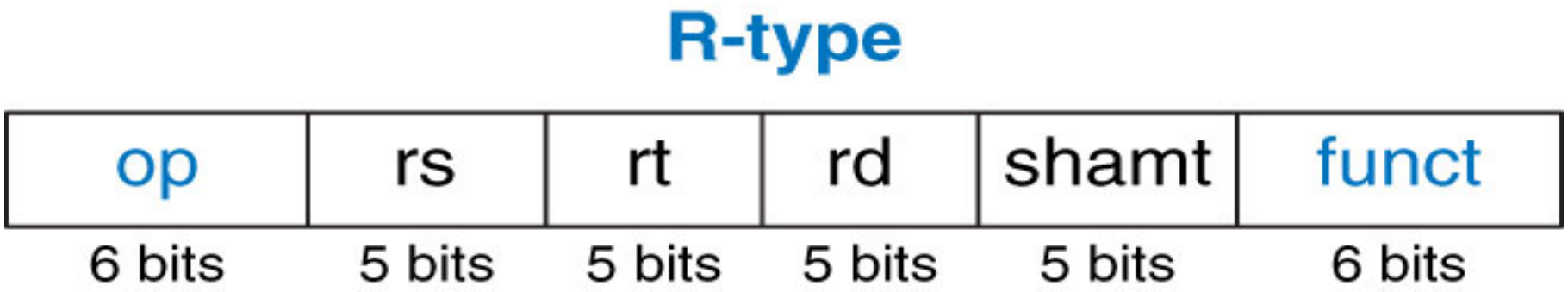
(two sources, one destination)

Complex high-level code might require multiple MIPS instructions.



R-Type Instructions

- Register-type MIPS Instructions
  - 2 source registers, 1 destination register

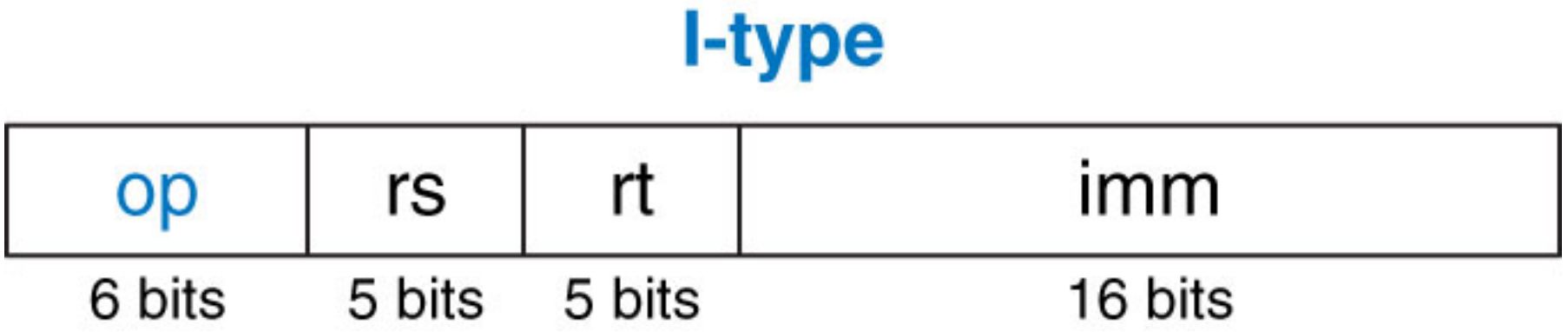


Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

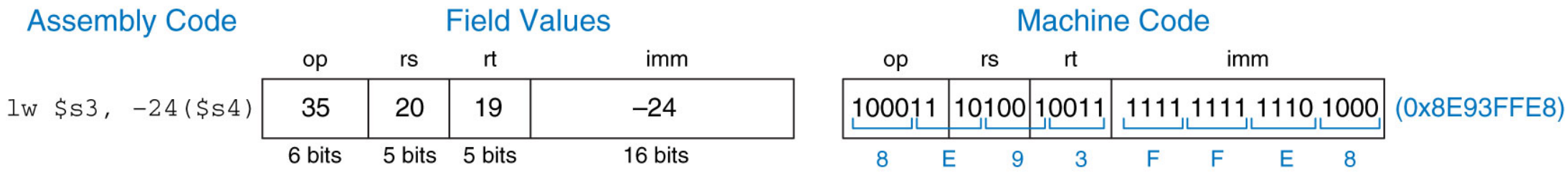
Assembly Code	Field Values						Machine Code						
	op	rs	rt	rd	shamt	funct	op	rs	rt	rd	shamt	funct	
`add $s0, $s1, $s2`	0	17	18	16	0	32	000000	10001	10010	10000	00000	100000	(0x02328020)
`sub $t0, $t3, $t5`	0	11	13	8	0	34	000000	01011	01101	01000	00000	100010	(0x016D4022)
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

I-Type Instructions

Two register operands and an immediate.



Immediate


Assembly Code
Field Values
Machine Code
lw $s3, –24($s4)
op
rs
rt
imm
35
20
19
–24
6 bits
5 bits
5 bits
16 bits
op
rs
rt
imm
100011
10100
10011
1111 1111 1110 1000
(0x8E93FFE8)
8
E
9
3
F
F
E
8

Interpreting Machine Language

1. Convert hex to binary.
2. Find the function
    1. If first 6 bits are 0, R-type
    2. Else I-type or J-type
3. Break rest of instruction down based on type and function.

We already practiced step 3. Step 2 is new because we added a second type of command.

Translate the Following to Assembly

0x2237FFF1

0x02F34022

R-type

I-type

Instruction	Opcode/Function	Syntax	Operation
add	100000	f $d, $s, $t	$d = $s + $t
addu	100001	f $d, $s, $t	$d = $s + $t
addi	001000	f $d, $s, i	$d = $s + SE(i)
addiu	001001	f $d, $s, i	$d = $s + SE(i)
and	100100	f $d, $s, $t	$d = $s & $t
andi	001100	f $d, $s, i	$t = $s & ZE(i)
sub	100010	f $d, $s, $t	$d = $s - $t

Summary

1. Architecture defines what the programmer sees.
2. Assembly language allows us to interact with the system without writing in pure machine language.
3. R-type instructions use registers to operate on data.
4. I-type instructions allow the use of hard-coded constants.