

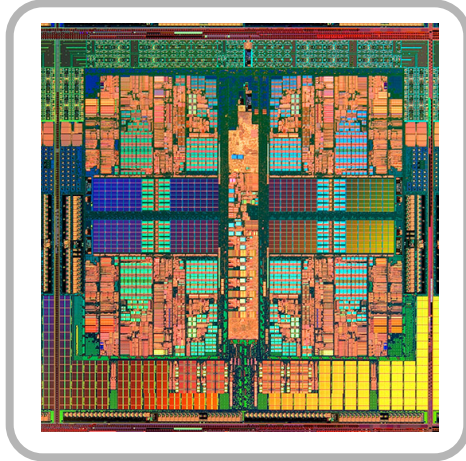
Lecture 05-06

Motivation and Background of MIPS Instruction Set Architecture (ISA)

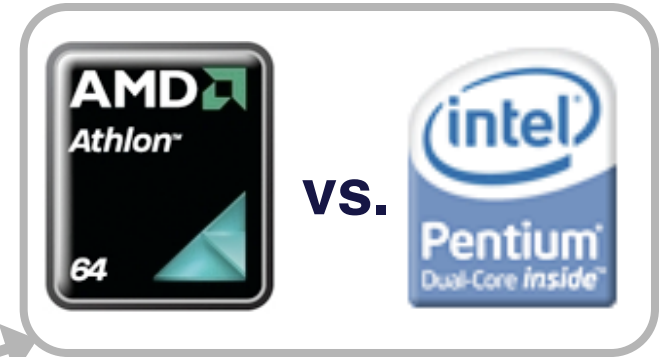
Suggested reading:
(HP Chapter 2.1-2.3 & 2.5-2.7)
(do not need to read HP Chapter 2.4)

Processor components

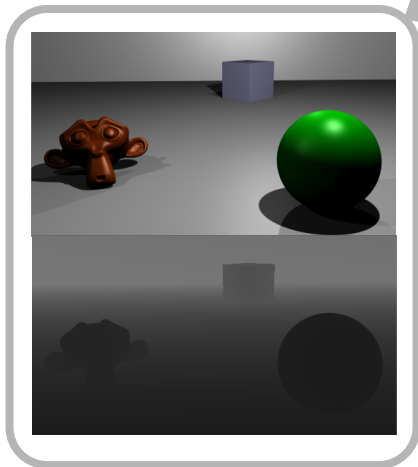
Multicore processors and programming



Processor comparison



CSE 30321



```

for i=0; i<5; i++ {
    a = (a*b) + c;
}

```

↓

```

MULT r1,r2,r3 # r1 ← r2*r3
ADD r2,r1,r4  ↓ # r2 ← r1+r4

```

110011	000001	000010	000011
001110	000010	000001	000100

Writing more efficient code

The right HW for the right application

HLL code translation

Fundamental lesson(s)

- **Today I'll explain what an ISA in a typical, modern microprocessor looks like**
 - **How memory references are handled / encoded, etc. in the MIPS ISA (our example) are fairly representative of others too (e.g. the ARM ISA).**

Why it's important...

- In this lecture, you'll get a very good sense as to what kind of assembly code is generated when you compile some HLL code
- Later in the semester, I'll show you what HLL code you write can **SIGNIFICANTLY** impact its execution time
 - **Should already start to see this in lab**
- To really take advantage of this, need to understand how HLL code gets mapped to assembly code + how assembly suggests how HW actually performs a computation

Quick recap

- **Context**
 - **Lecture 01:**
 - Introduction to the course
 - **Lectures 02-03:**
 - Introduction to programmable processors
 - (6-instruction + some ARM ISA)
 - **Lecture 04 (and part of Lecture 05):**
 - How to quantify impact of design decisions
 - **Lecture 05: (MIPS ISA)**
 - Apply / revisit ideas introduced in Lectures 02, 03, but use context of modern ISA
 - Use benchmark techniques from Lecture 04 with this material and throughout the rest of the course

A more sophisticated ISA

- **Shortcomings of the simple processor**
 - Only 16 bits for data and instruction
 - Data range can be too small
 - Addressable memory is small
 - Only “room” for 16 instruction opcodes
- **MIPS ISA: 32-bit RISC processor**
 - A representative RISC ISA
 - (RISC – Reduced Instruction Set Computer)
 - A fixed-length, regularly encoded instruction set and uses a load/store data model
 - Used by NEC, Cisco, Silicon Graphics, Sony, Nintendo...
 - ...and more

Most modern microprocessors are RISC-like including ARMs



6-instruction vs. MIPS

Instruction
Encoding

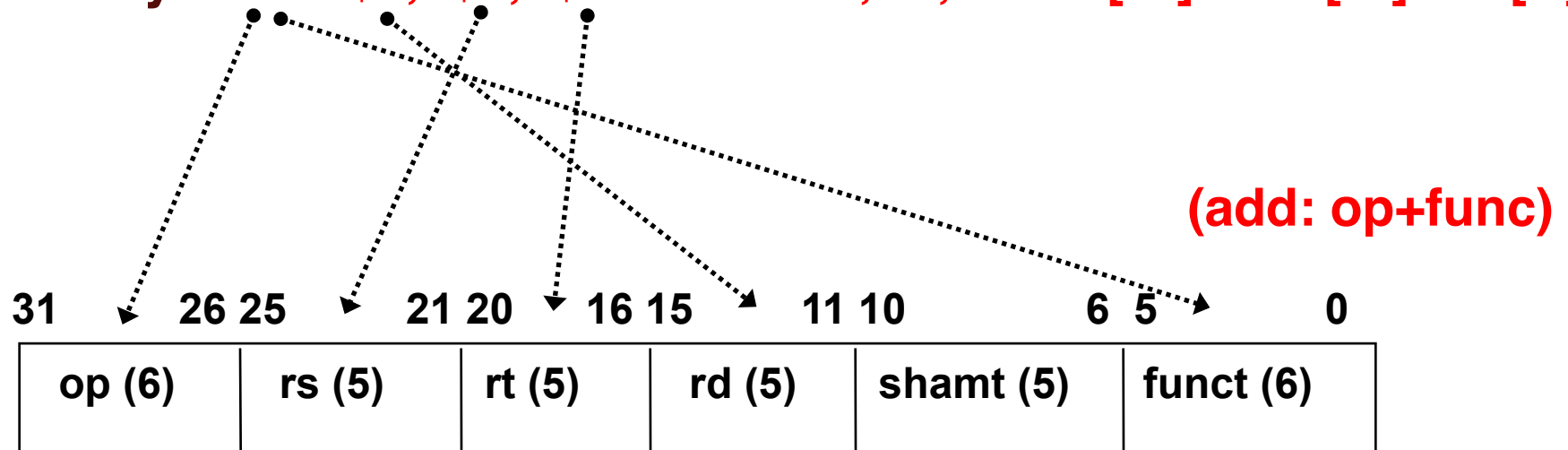
6-instruction processor:

Add instruction: **0010** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

Add **Ra, Rb, Rc**—specifies the operation $RF[a]=RF[b] + RF[c]$

MIPS processor:

Assembly: **add \$9, \$7, \$8 # add rd, rs, rt: $RF[rd] = RF[rs]+RF[rt]$**



Machine:

B: 000000 00111 01000 01001 xxxxx 100000
D: 0 7 8 9 x 32

6-instruction vs. MIPS

Instruction
Encoding

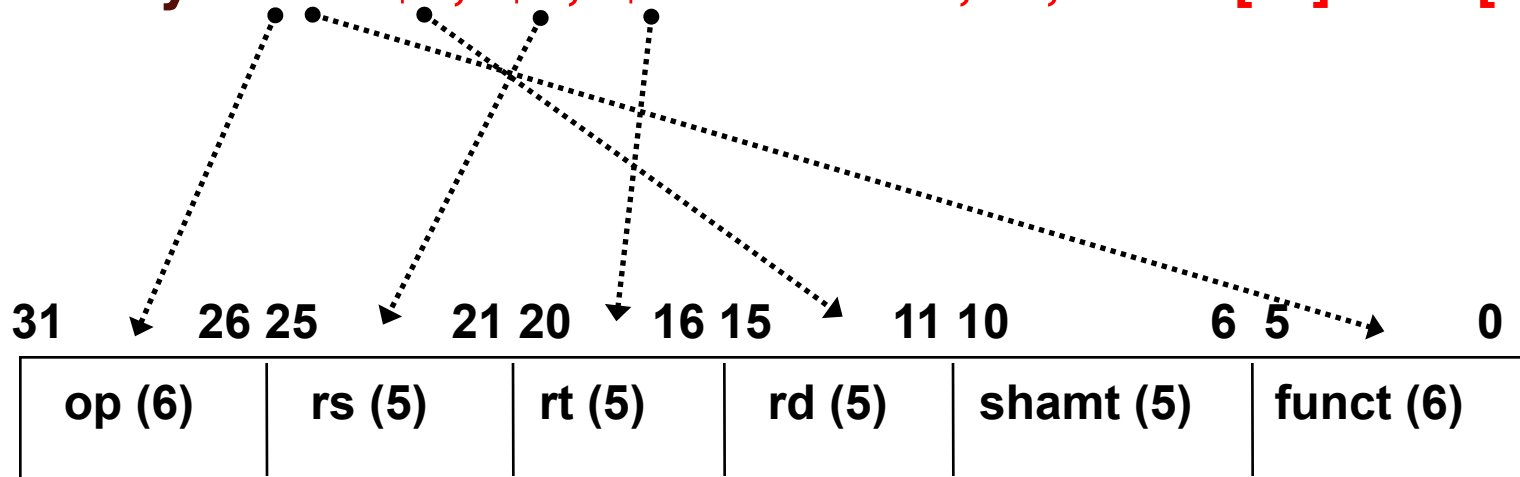
6-instruction processor:

Sub instruction: **0111** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

SUB Ra, Rb, Rc—specifies the operation $RF[a]=RF[b]-RF[c]$

A MIPS subtract

Assembly: **sub \$9, \$7, \$8 # sub rd, rs, rt: $RF[rd] = RF[rs]-RF[rt]$**

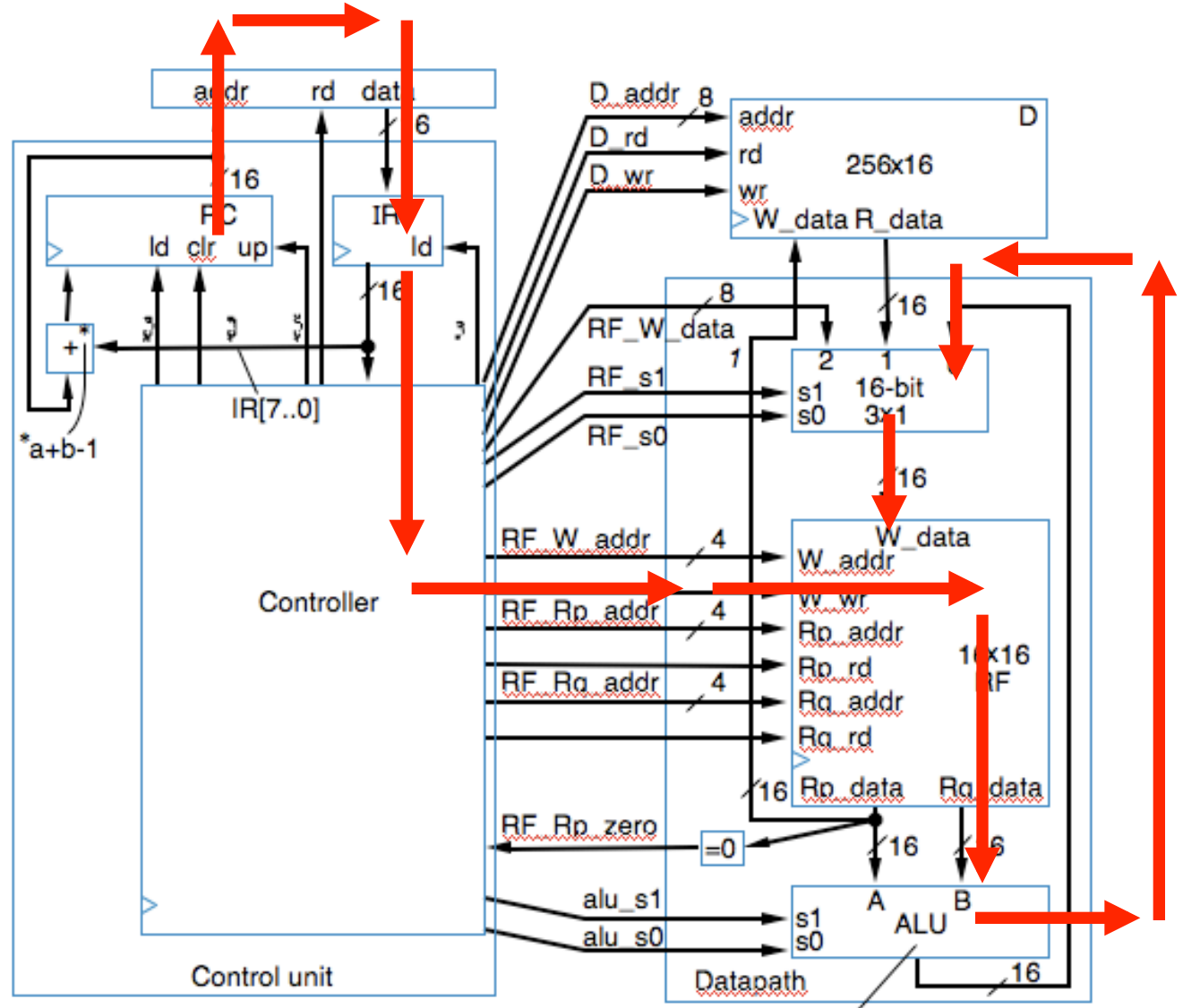


Machine:

B: 000000 00111 01000 01001 xxxxx 100010
D: 0 7 8 9 x 34

6-instruction vs. MIPS

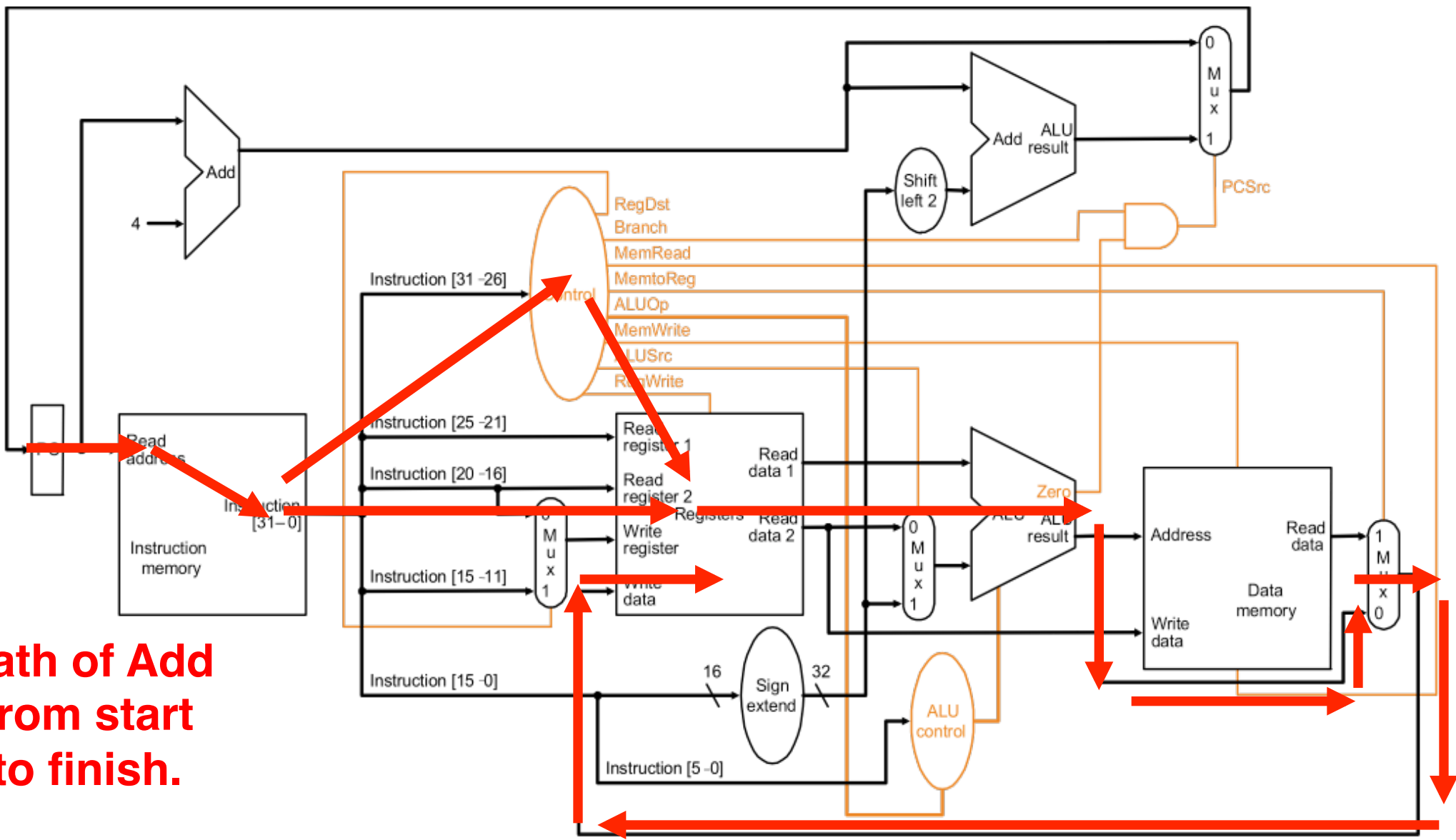
**Path of Add
from start
to finish.**



s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

6-instruction vs. MIPS

Datapath



Path of Add from start to finish.

Note:

We'll discuss the specifics of the MIPS ISA in more detail shortly...

...but first, I'll go through a few slides on how MIPS-like (i.e. RISC) ISAs came to be.

Instructions Sets

- **An instruction set specifies a processor's functionality**
 - **what operations it supports**
 - **what storage mechanisms it has & how they are accessed**
 - **how the programmer/compiler communicates programs to processor**
- **ISA: “interface” between HLL and HW**
- **ISAs may have different syntax (6-instruction vs. MIPS), but can still support same general types of operation (i.e. register-register)**

Instruction Set Architecture

- **Instructions must have some basic functionality:**
 - **Access memory (read and write)**
 - **Perform ALU operations (add, multiply, etc.)**
 - **Implement control flow (jump, branch, etc.)**
 - **I.e. to take you back to the beginning of a loop**
- **Significant difference often how memory, data addressed**
 - **Operand location**
 - **(stack, memory, register)**
 - **Addressing modes**
 - **(computing memory addresses)**
 - **(Let's digress on the board and preview how MIPS does a load)**
 - **(Compare to 6-instruction processor?)**

What makes a good instruction set

- **implementability**
 - **supports a (performance/cost) range of implementations**
 - **implies support for high performance implementations**
- **programmability**
 - **easy to express programs (for human and/or compiler)**
- **backward/forward compatibility**
 - **implementability & programmability across generations**
 - **e.g., x86 generations: 8086, 286, 386, 486, Pentium, Pentium II, Pentium III, Pentium 4...**

Example: range of cost implementations

32-bit ARM
not unlike
32-bit MIPS

ARM and Thumb-2 Instruction Set
Quick Reference Card

Operation	§	Assembler	S updates	Action
Multiply	Multiply	MUL{S} Rd, Rm, Rs	N Z C*	Rd := (Rm * Rs)[31:0]

MIPS: Mul \$9, \$7, \$8 # mul rd, rs, rt: RF[rd] = RF[rs]*RF[rt]

Simplified (16-bit) ARMs available too

» Improved Code Density with Performance and Power Efficiency

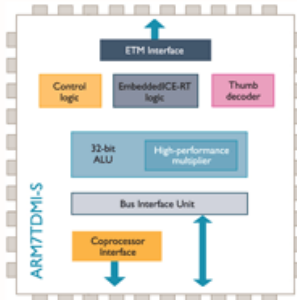
Thumb-2 technology is the instruction set underlying the ARM Cortex architecture which provides enhanced levels of performance, energy efficiency, and code density for a wide range of embedded applications.

Thumb-2 technology builds on the success of Thumb, the innovative high code density instruction set for ARM microprocessor cores, to increase the power of the ARM microprocessor core available to developers of low cost, high performance systems.

The technology is backwards compatible with existing ARM and Thumb solutions, while significantly extending the features available to the Thumb instructions set. This allows more of the application to benefit from the best in class code density of Thumb.

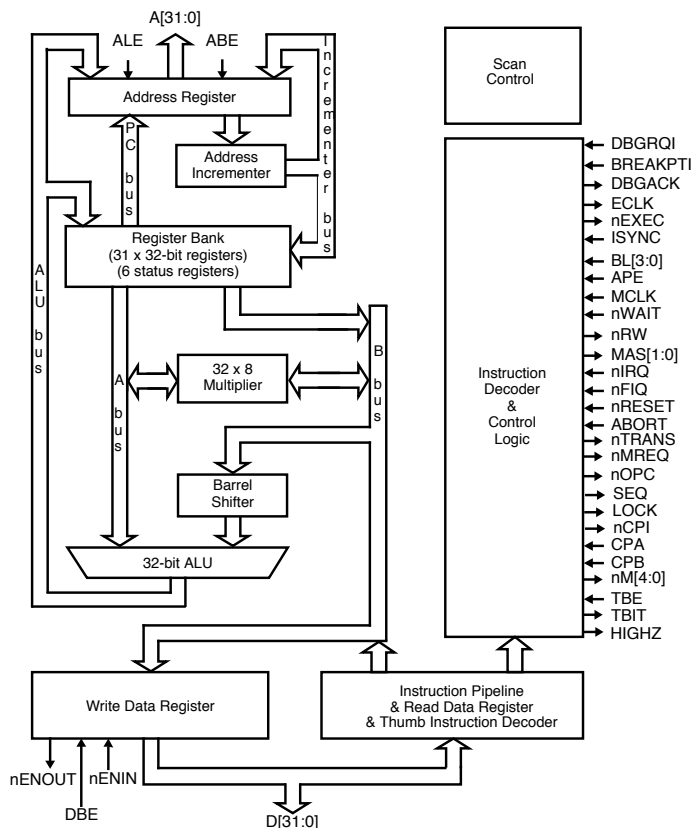
For performance optimised code Thumb-2 technology uses 31 percent less memory to reduce system cost, while providing up to 38 percent higher performance than existing high density code, which can be used to prolong battery-life or to enrich the product feature set. Thumb-2 technology is featured in the processor, and in all ARMv7 architecture-based processors.

<http://www.arm.com/products/processors/technologies/instruction-set-architectures.php>



<http://www.arm.com/images/pro-A7TDMI-s.gif>

http://www.arm.com/images/armpp/nook2_%281%29.jpg




<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/index.html>

Programmability

- **a history of programmability**

- **pre - 1975: most code was hand-assembled**
- **1975 – 1985: most code was compiled**
 - **but people thought that hand-assembled code was superior**
- **1985 – present: most code was compiled**
 - **and compiled code was at least as good as hand-assembly**



**“Programming”
literally sitting
down and
writing machine
code**

over time, a big shift in what “programmability” means

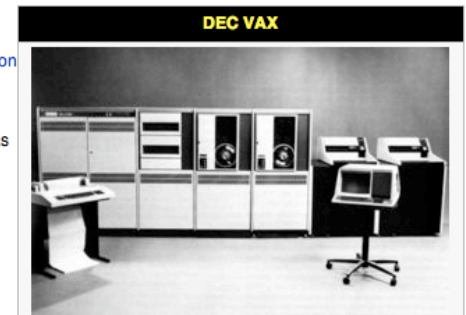
pre-1975: Human Programmability

- focus: instruction sets that were easy for humans to program
 - ISA semantically close to high-level language (HLL)
 - closing the “semantic gap”
 - semantically heavy complex instructions
 - e.g., the VAX had instructions for polynomial evaluation
 - people thought computers would someday execute HLL directly
 - never materialized

Let's look at an example

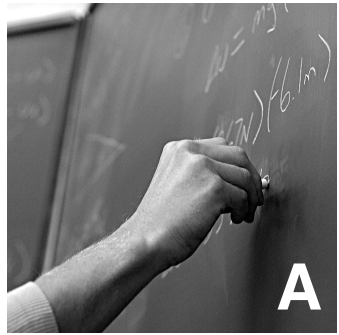
VAX is a 32-bit computing architecture that supports an orthogonal instruction set (machine language) and virtual addressing (i.e. demand paged virtual memory). It was developed in the mid-1970s by Digital Equipment Corporation (DEC). DEC was later purchased by Compaq, which in turn was purchased by Hewlett-Packard.

The VAX has been perceived as the quintessential CISC processing architecture, with its very large number of addressing modes and machine instructions, including instructions for such complex operations as queue insertion/deletion and polynomial evaluation.^[citation needed]



Manufacturer:	Digital Equipment Corporation
Byte size:	8 bits (octet)
Address bus size:	32 bits
Peripheral bus:	Unibus, Massbus, Q-Bus, XMI, VAXBI
Architecture:	CISC, virtual memory
Operating systems:	VAX/VMS, Ultrix, BSD UNIX

The Quadratic Formula



$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Approach 1:

QUAD_Plus X1, a, b, c
QUAD_Minus X2, a, b, c

or

QUAD X1, X2, a, b, c

Approach 2:

Mult R1, b, b
Mult R2, a, c
Multi R2, R2, 4
Sub R3 R1, R2
Sqrt R3, R3
Mult, R4, a, 2
Mult R5, b, -1
Add R6, R5, R3
Div R6, R6, R4 # result 1
Sub R7, R5, R3
Div R7, R7, R4 # result 2

Generally requires more specialized HW

Provides primitives, not solutions

Storage Model: Register-Register (Ld/St)

```
load R1,A      R1 = M[A];  
load R2,B      R2 = M[B];  
add R3,R1,R2   R3 = R1 + R2;  
store C,R3     M[C] = R3;
```

Again, typical of a modern ISA
and the focus in this course...

- **load/store architecture: ALU operations on regs only**
 - (minus) poor code density
 - (plus) easy decoding, operand symmetry
 - (plus) deterministic length ALU operations
 - **(plus) fast decoding helps pipelining (later)**
- **1960's and onwards**
 - **RISC machines: Alpha, MIPS, PowerPC, ARM**

Instruction Formats

- **fixed length (most common: 32-bits)**
 - (plus) easy for pipelining (e.g. overlap) and for multiple issue (superscalar)
 - don't have to decode current instruction to find next instruction
 - (minus) not compact
 - Does the MIPS add “waste” bits?



- **variable length**
 - (plus) more compact
 - (minus) hard (but do-able) to efficiently decode
 - (important later)



Present Day: Compiled Assembly

Consider a representative set of passes

Function:

Transform HLL to common, intermediate form

Representative examples:
Procedure in-lining, loop transformations, etc.

Representative example:
Register allocation

Detailed instruction selection and machine-dependent optimizations

Front end language specific

High-level optimizations
(things any HLL code might benefit from)

Global optimizer

Code generator

Dependencies:

Language dependent, machine independent

Somewhat language dependent, largely machine independent

Small language dependencies, some machine dependencies (i.e. register counts)

Highly machine dependent, language independent

More specifics about:

MIPS instruction syntax

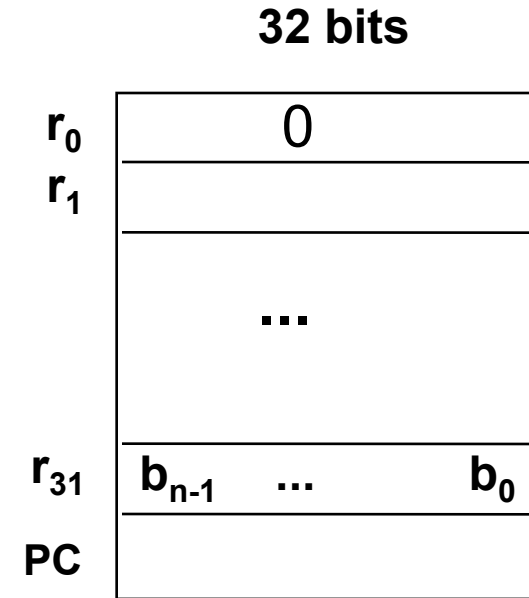
Register usage

NOW: INTRODUCTION TO MIPS ISA

MIPS registers

□ 32x32-bit GPRs (General purpose registers)

- \$0 = \$zero (therefore only 31 GPRs)
- \$1 = \$at (reserved for assembler)
- \$2 - \$3 = \$v0 - \$v1 (return values)
- \$4 - \$7 = \$a0 - \$a3 (arguments)
- \$8 - \$15 = \$t0 - \$t7 (temporaries)
- \$16 - \$23 = \$s0 - \$s7 (saved)
- \$24 - \$25 = \$t8 - \$t9 (more temporaries)
- \$26 - \$27 = \$k0 - \$k1 (reserved for OS)
- \$28 = \$gp (global pointer)
- \$29 = \$sp (stack pointer)
- \$30 = \$fp (frame pointer)
- \$31 = \$ra (return address)

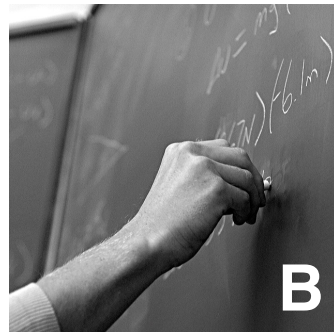


- 32x32-bit floating point registers (paired double precision)
- Program counter
- Status, Cause, BadVAddr, EPC

Let's think ahead a bit...

Board digression

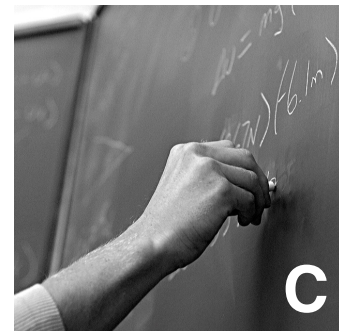
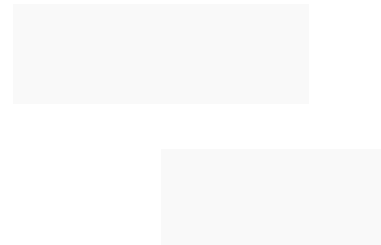
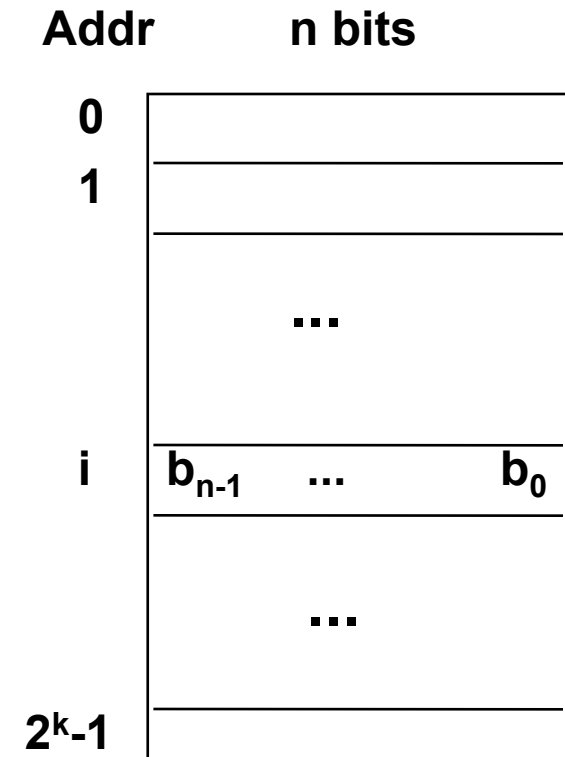
- Programmer visibility
- Procedure calls



Memory Organization

□ Addressable unit:

- smallest number of consecutive bits (word length) can be accessed in a single operation
- Example, $n=8$, byte addressable



Effect of Byte Addressing

MIPS: Most data items are contained in *words*, a word is 32 bits or 4 bytes. *Registers hold 32 bits of data*

Word 0	Byte 0011	Byte 0010	Byte 0001	Byte 0000
Word 1	Byte 0111	Byte 0110	Byte 0101	Byte 0100
Word 2	Byte 1011	Byte 1010	Byte 1001	Byte 1000
Word 3	Byte 1111	Byte 1110	Byte 1101	Byte 1100

Assume PC=0,
want to read word
of data (4 bytes)

To get next data
word, need to
increment PC by 4.

Address of this byte is 1010 (or 9 in base 10)

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned
- What are the least 2 significant bits of a word address?



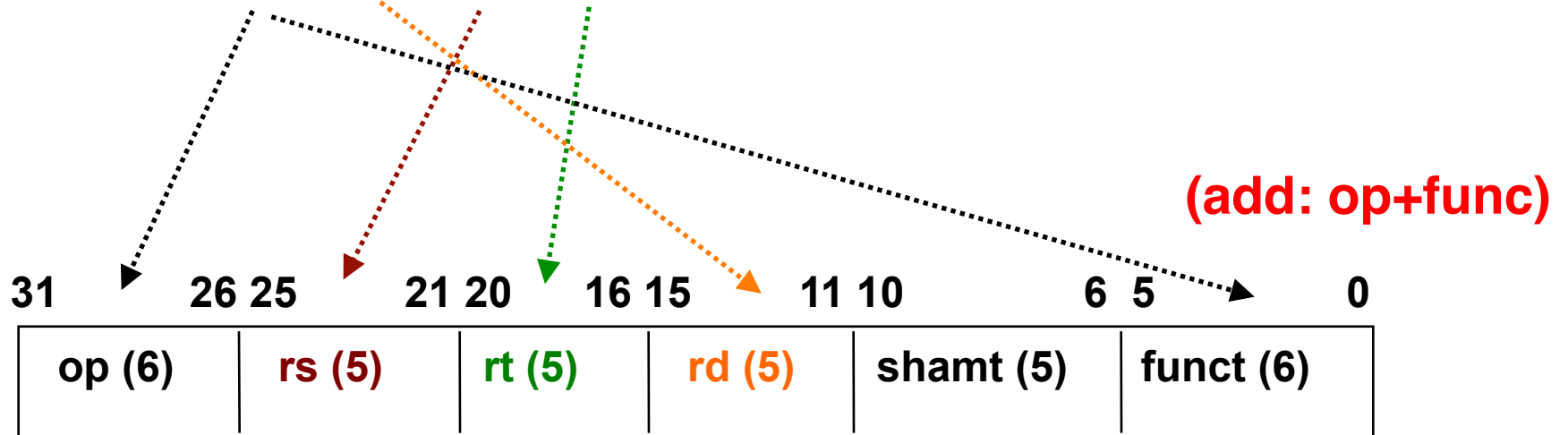
A View from 10 Feet Above

- Instructions are characterized into basic types
- With each type, 32 bits of instruction encoding are interpreted differently...
 - Generally a good idea not to have *too many* types.
 - Why?
- 3 types of instructions:
 - R type
 - I type
 - J type
- Look at both assembly and machine code

R-Type: Assembly and Machine Format

- R-type: All operands are in registers

Assembly: **add \$9, \$7, \$8** # add rd, rs, rt: $RF[rd] = RF[rs] + RF[rt]$



Machine:

B:	000000	001111	010000	01001	xxxxx	100000
D:	0	7	8	9	x	32

R-type Instructions

- ❑ All instructions have 3 operands
- ❑ All operands must be registers
- ❑ Operand order is fixed (destination first)
- ❑ Example:

C code: $A = B - C;$

(Assume that A, B, C are stored in registers s0, s1, s2.)

MIPS code: `sub $s0, $s1, $s2`

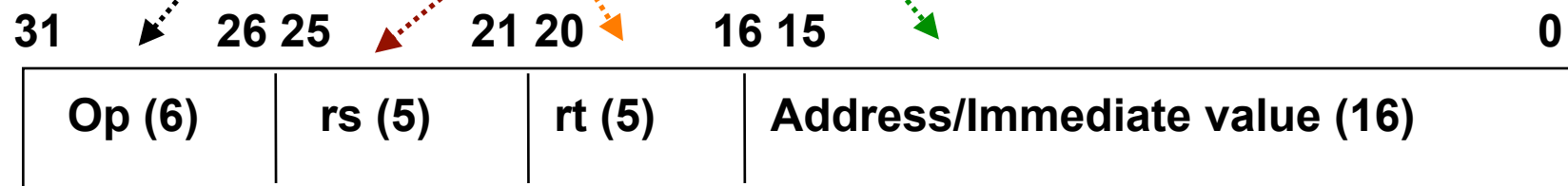
Machine code: 

- ❑ Other R-type instructions
 - `addu, mult, and, or, sll, srl, ...`

I-Type Instructions – example 1

I-type: one of two source operands is an “immediate value” and the other is in a register; destination = a register

Example: `addi $s2, $s1, 128` # `addi rt, rs, Imm`
`RF[18] = RF[17]+128`

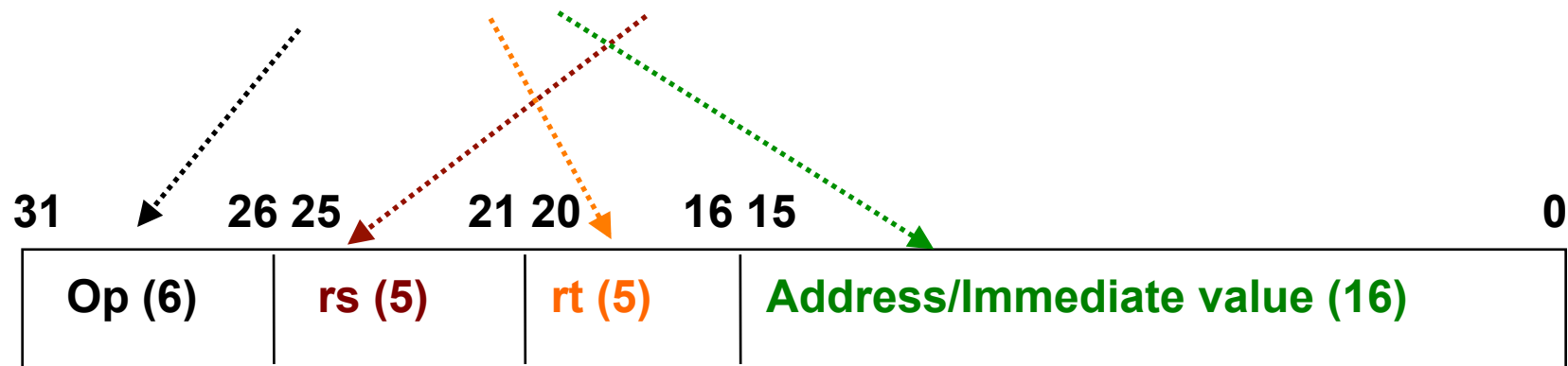


B: 001000 10001 10010 000000010000000
D: 8 17 18 128

I-Type Instructions – example 2

I-type: one of two source operands is an “immediate value” and the other is in a register; destination = a register

Example: **lw** \$s3, 32(\$t0) # RF[19] = DM[RF[8]+32]



B: 100011 01000 10011 0000000000100000
 D: 35 8 19 32

How about load the next word in memory?

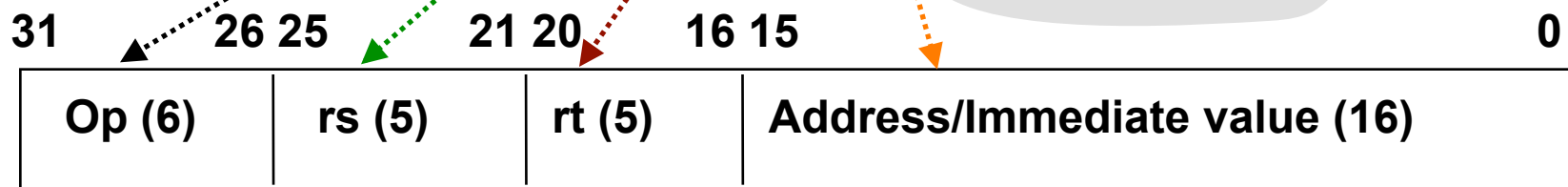
I-Type Instructions – example 3

I-type: one of two source operands is an “immediate value” and the other is in a register; destination = a register

Example: Again: **bne \$t0, \$t1, Again**

Can always just use label in HW, labs, exams

if (RF[8]!=RF[9]) PC=PC+4+Imm*4
else PC=PC+4



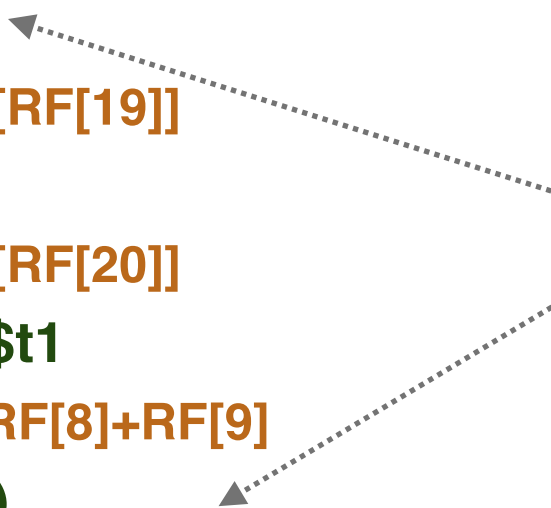
B: 00101 01000 01001 1111111111111111
D: 5 8 9 -1

PC-relative addressing

Example: Memory Access Instructions

- MIPS is a Load/Store Architecture (a hallmark of RISC)
 - Only load/store type instructions can access memory
- Example: $A = B + C$;
 - Assume: A, B, C are stored in memory, \$s2, \$s3, and \$s4 contain the addresses of A, B and C, respectively.
 - lw \$t0, 0(\$s3)
 - $RF[8]=DM[RF[19]]$
 - lw \$t1, 0(\$s4)
 - $RF[9]=DM[RF[20]]$
 - add \$t2, \$t0, \$t1
 - # $RF[10]=RF[8]+RF[9]$
 - sw \$t2, 0(\$s2)
 - $DM[RF[18]]=RF[10]$
- sw has destination last
- What is the instruction type of sw?

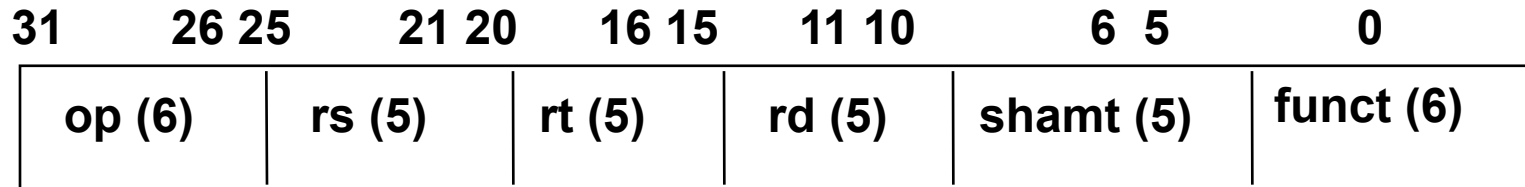
Take note of ordering...
different than 6-instruction
processor example



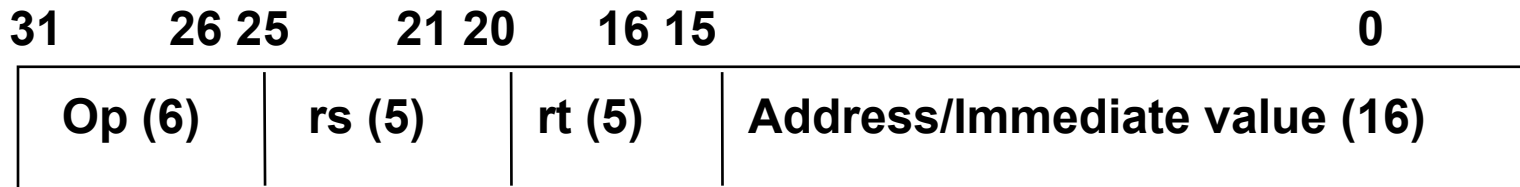
Summary of MIPS Instruction Formats

All MIPS instructions are 32 bits (4 bytes) long.

R-type:



I-Type:



J-type



More on MIPS ISA

- **How to get constants into the registers?**
 - **Zero used *very* frequently => \$0 is hardwired to zero**
 - if used as an argument, zero is passed
 - if used as a target, the result is destroyed
 - **Small constants are used frequently (~50% of operands)**
 - A = A + 5; (addi \$t0, \$t0, 5)**
 - slti \$8, \$18, 10**
 - andi \$29, \$29, 6**
 - ori \$29, \$29, 4**
 - **What about larger constants?**

More Discussion & Examples

