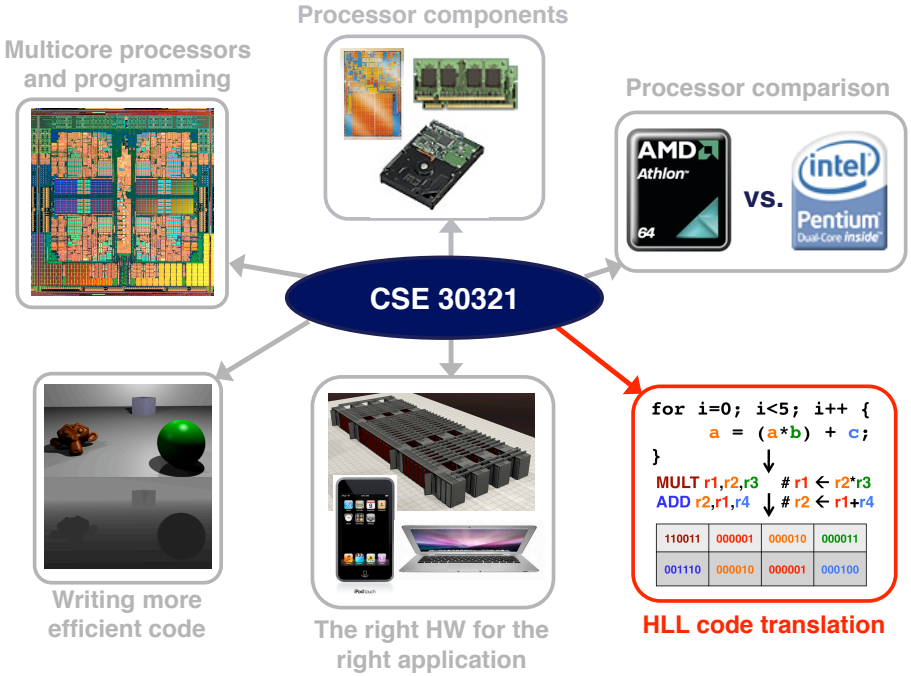
Lecture 05-06
Motivation and Background of MIPS ISA

Recap and Readings

- Context
  - Lecture 01:
    - Introduction to the course
  - Lectures 02-03:
    - Fundamental principals that we will discuss and apply in class using very simplified case study
  - Lecture 04:
    - 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
- Readings
  - H&P: Chapters 2.1-2.3, 2.5-2.7



Motivation: Why MIPS?

- 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

A quick look: more complex ISAs

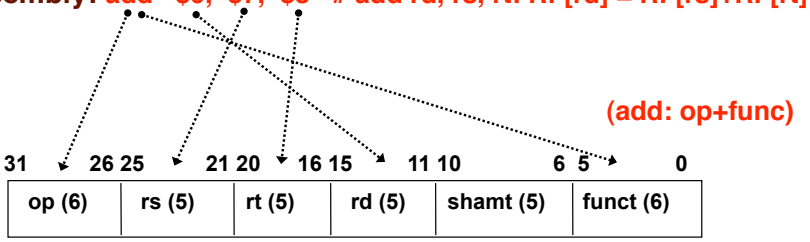
- **6-instruction processor:**

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

Instruction Encoding

A quick look: more complex ISAs

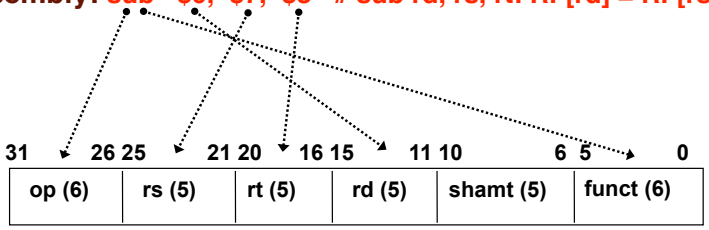
- **6-instruction processor:**

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:

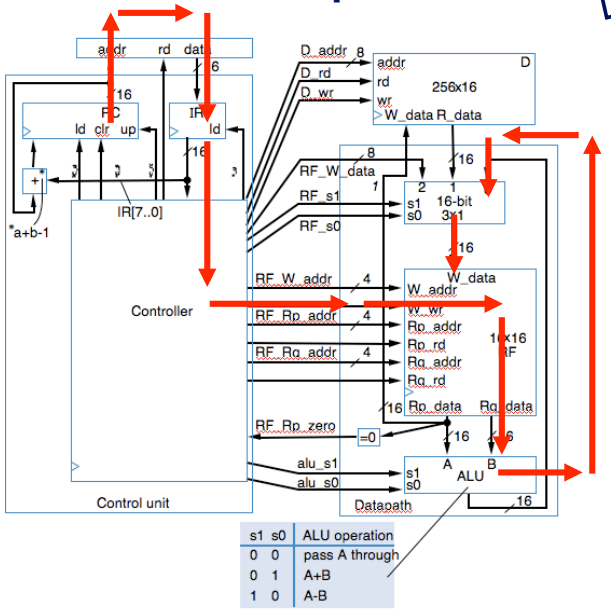
B:	000000	00111	01000	01001	xxxxx	100010
D:	0	7	8	9	x	34

Instruction Encoding

A quick look: more complex ISAs

Datapath

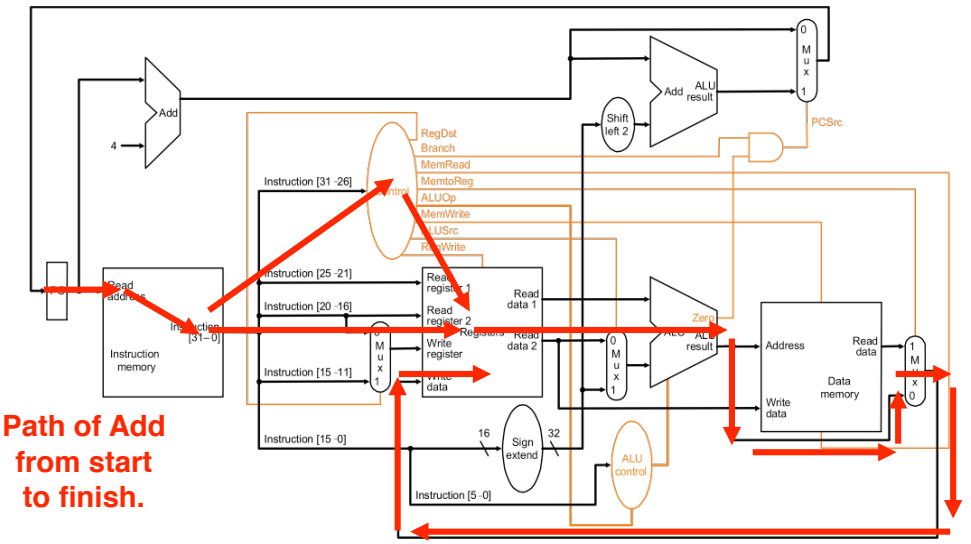
Path of Add from start to finish.



A quick look: more complex ISAs

Datapath

Path of Add from start to finish.



Note:

- In class, we will work with somewhat simplified version of MIPS ISA
  - Reduced instruction set makes HWs, etc. much more manageable
    - However, not dissimilar to early ARMs
  - Taking less sophisticated processing power to places where there was *none* is *significant*

We'll discuss MIPS more in a bit...
...but 1st, a few slides on ISAs in general.

Instructions Sets

- "Instruction set architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine"
  - IBM introducing 360 (1964)
- 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 sytnax (6-instruction vs. MIPS), but can still support the same general types of operations (i.e. Reg-Reg)

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
- Largest difference is in accessing memory
  - 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 — A bit more on this one...
  - 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...
- think about these issues as we discuss aspects of ISAs

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: Add $9, $7, $8 # add 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.

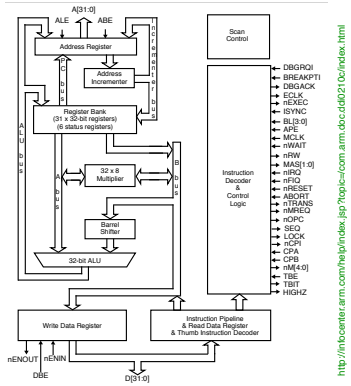
The technology is backwards compatible with existing ARM and Thumb solutions, while significantly extending the features available to the Thumb instructions set. This allows more of the application to benefit from the best in class code density of Thumb.

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.ddi0211c/index.html

Programmability

"Programming" literally sitting down and writing machine code

- 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

over time, a big shift in what "programmability" means

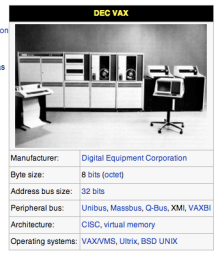
pre-1975: Human Programmability

- 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 Forumla

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

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
Sub R7, R5, R3
Div R7, R7, R4
```

Generally requires more specialized HW *Provides primitives, not solutions*

Present Day: Compiled Assembly

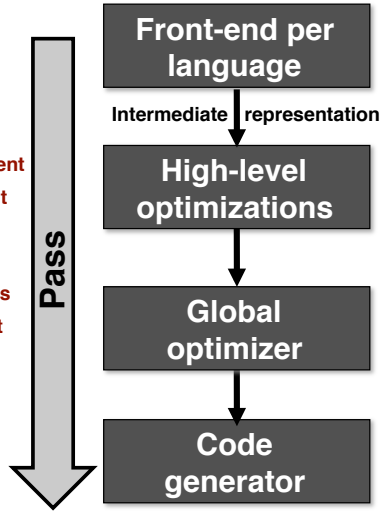
Dependencies:

- Language dependent
- Machine independent

- Somewhat language dependent
- Largely machine independent

- Small language dependencies
- Machine dependencies slight
- (I.e. register counts/types)

- Highly machine dependent
- Language independent



Front-end per language

Intermediate representation

High-level optimizations

Global optimizer

Code generator

Function:

Transform language to common, intermediate form

For example, procedure inlining and loop transformations

Including global and local optimization + register allocation

Detailed instruction selection and machine-dependent optimizations (assembler next?)

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)

Operations

- arithmetic and logical:
  - add, mult, and, or, xor, not
- data transfer:
  - move, load, store
- control:
  - conditional branch, jump, call, return
- system:
  - syscall, traps
- floating point:
  - add, mul, div, sqrt
- decimal:
  - addd, convert (not common today)
- string:
  - move, compare (also not common today)
- multimedia:
  - e.g., Intel MMX/SSE and Sun VIS
- vector:
  - arithmetic/data transfer, but on vectors of data

If no instruction for HLL operation, can "fake it" -- i.e. lots of adds instead of multiply.

Examples...

Internal Storage Model

- choices
  - stack
  - accumulator
  - memory-memory
  - register-memory
  - register-register (also called "load/store")
- running example:
  - add C, A, B (C := A + B)

Storage Model: Stack

```
push A   S[++TOS] = M[A];
push B   S[++TOS] = M[B];
add      T1=S[TOS--]; T2=S[TOS--]; S[++TOS]=T1+T2;
pop C    M[C] = S[TOS--];
```

- operands implicitly on top-of-stack (TOS)
- ALU operations have zero explicit operands
  - (plus) code density (top of stack implicit)
  - (minus) memory, pipelining bottlenecks (why?)
- mostly 1960's & 70's
  - x86 uses stack model for FP
    - (bad backward compatibility problem)
  - JAVA bytecodes also use stack model

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;
```

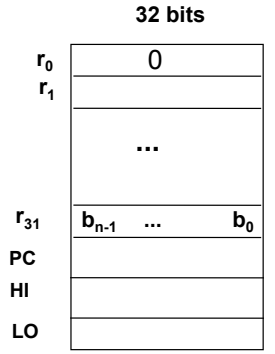
- load/store architecture: ALU operations on regs only
  - (minus) poor code density
  - (plus) easy decoding, operand symmetry
  - (plus) deterministic length ALU operations
  - (plus) fast decoding helps pipelining (later)
- 1960's and onwards
  - RISC machines: Alpha, MIPS, PowerPC (but also Cray)

Introduction to MIPS ISA

- Next:
  - More specifics about:
    - MIPS instruction syntax
    - Register usage

MIPS Registers (R2000/R3000)

- 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 doubl… ion)
- HI, LO, PC
- Sta… Addr, EPC

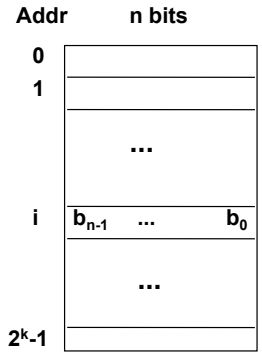
Let's think ahead a bit...

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



Given 1K bit memory, 16 bit word addressable:

How many words?

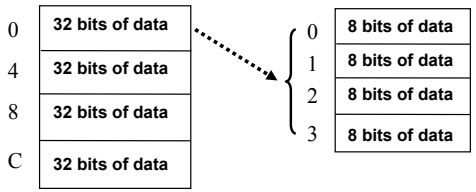
How many address bits?

$(n*2^k)$ bits = $(n*2^{k-3})$ bytes

MIPS uses byte-addressable memory

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*



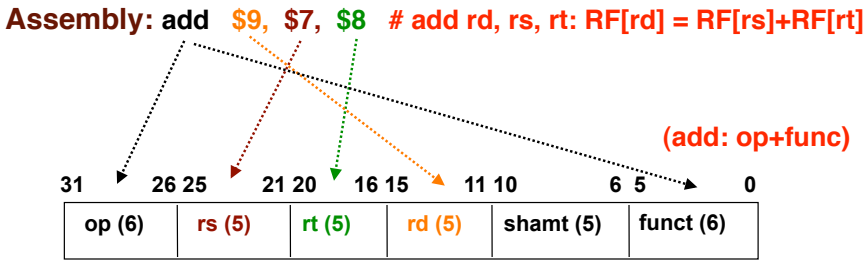
- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned
- What are the least 2 significant bits of a word address?

A View from 10 Feet Above

- Instructions are characterized into basic types
- Each type interpret a 32-bit instruction differently
- 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



Machine:

B:	000000	00111	01000	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.)

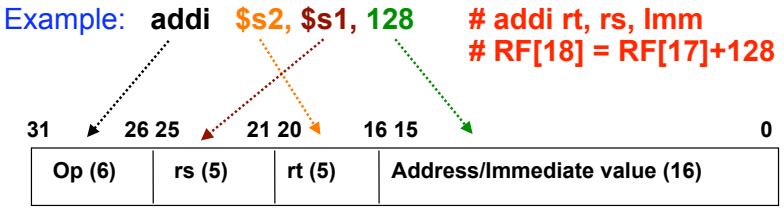
MIPS code: sub $s0, $s1, $s2

Machine code:

- Other R-type instructions
  - addu, mult, and, or, sll, srl, …

I-Type Instructions

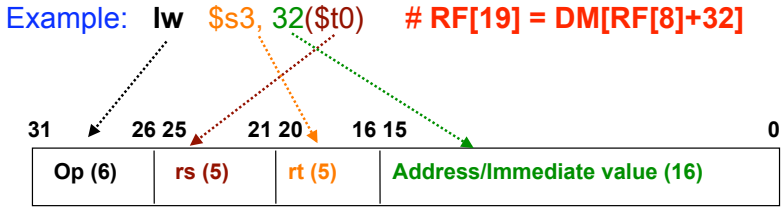
- I-type: One operand is an immediate value and others are in registers



B:	001000	10001	10010	0000000010000000
D:	8	17	18	128

I-Type Instructions: Another Example

- I-type: One operand is an immediate value and others are in registers



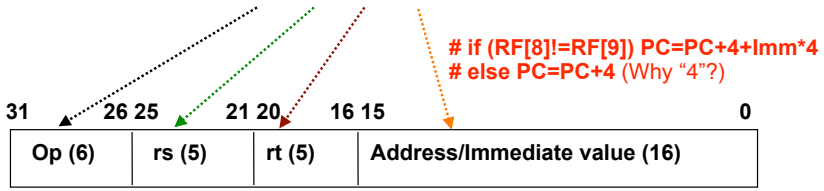
B:	100011	01000	10011	0000000000100000
D:	35	8	19	32

How about load the next word in memory?

I-Type Instructions: Yet Another Example

- I-type: One operand is an immediate value and others are in registers

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



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

PC-relative addressing

J-Type Instructions

- J-type: only one operand: the target address

Example: j 3 # PC = (PC+4)[31:28]||Target||00 (Why "00"?)

B: 000010 00000000000000000000000011
D: 2 3

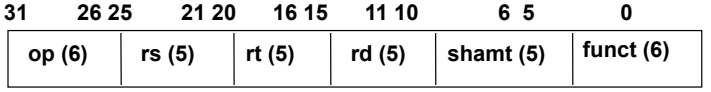
Pseudo-direct Addressing

Example: Memory Access Instructions

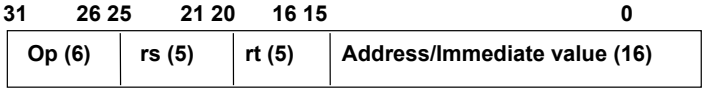
- MIPS is a Load/Store Architecture (a hallmark of RISC)
  - Only load/store type instructions can access memory
- Example: A = B + C;
  - Assume: A, B, C are stored in memory, $s2, $s3, and $s4 contain the addresses of A, B and C, respectively.
    - lw $t0, 0($s3)
      - RF[8]=DM[RF[19]]
    - lw $t1, 0($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?

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?
- How about procedure calls?
- Other features on assembly programming

More Discussion & Examples