ECE 250 / CPS 250
Computer Architecture

Instruction Set Architecture Assembly Language

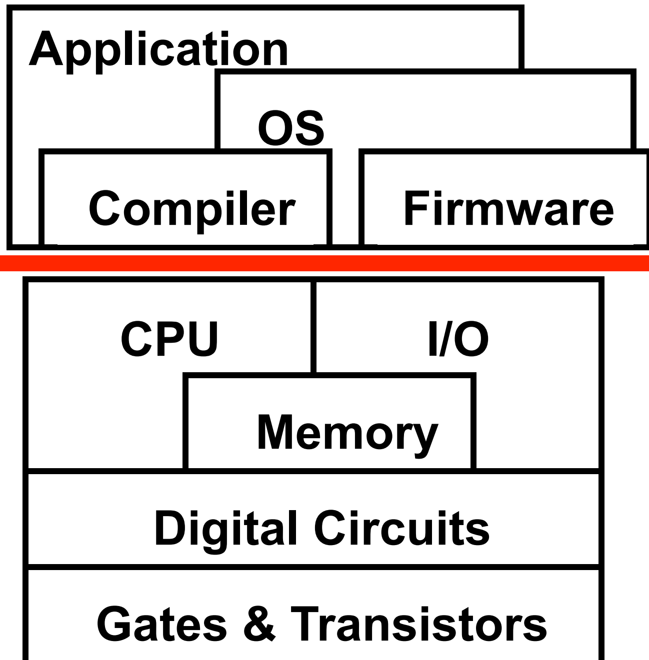
Benjamin Lee

Slides based on those from

Andrew Hilton (Duke), Alvy Lebeck (Duke)
Benjamin Lee (Duke), and Amir Roth (Penn)

© 2012 Daniel J. Sorin
from Roth and Lebeck

Instruction Set Architecture (ISA)



- ISAs in General
  - Using MIPS as primary example
- MIPS Assembly Programming
- Other ISAs

© 2012 Daniel J. Sorin
from Roth and Lebeck

Readings

- Patterson and Hennessy
  - Chapter 2
    - Read this chapter as if you'd have to teach it
  - Appendix A (reference for MIPS instructions and SPIM)
    - Read as much of this chapter as you feel you need

© 2012 Daniel J. Sorin
from Roth and Lebeck

What Is a Computer?

- Machine that has storage, which holds instructions and data, and executes instructions
- Storage (as seen by each running program)
  - Memory:
    - 2^{32} bytes for 32-bit machine
    - 2^{64} bytes for 64-bit machine *[[impossible! mystery for later…]]*
  - Registers: some (e.g., 32) 32-bit (or 64-bit) storage elements
    - Live inside processor core
- Instructions
  - Move data from memory to register or from register to memory
  - Compute on values held in registers
  - Switch to instruction other than the next one in order
  - Etc.

© 2012 Daniel J. Sorin
from Roth and Lebeck

What Is an ISA?

- ISA
  - The **“contract”** between software and hardware
  - If software does X, hardware promises to do Y
    - **Functional definition** of operations, modes, and storage locations supported by hardware
    - **Precise description** of how software can invoke and access them
  - Strictly speaking, ISA is the architecture, i.e., the interface between the hardware and the software
    - Less strictly speaking, when people talk about architecture, they’re also talking about how the the architecture is implemented

© 2012 Daniel J. Sorin
from Roth and Lebeck

How Would You Design an ISA?

- What kind of interface should the hardware present to the software?
  - Types of instructions?
  - Instruction representation?
  - How do we get from instruction 1 to 2 (or to 7 instead)?
  - Software's view of storage? Where do variables live?
  - Does the hardware help to support function/method calls? If so, how?
  - Should the hardware support other features that are specific to certain HLLs (e.g., garbage collection for Java)?

© 2012 Daniel J. Sorin
from Roth and Lebeck

Microarchitecture

- ISA specifies what hardware does, not how it does it
  - No guarantees regarding these issues:
    - How operations are implemented
    - Which operations are fast and which are slow
    - Which operations take more power and which take less
  - These issues are determined by the microarchitecture
    - Microarchitecture = how hardware implements architecture
    - Can be any number of microarchitectures that implement the same architecture (Pentium, Core i7 are almost the same architecture, but are very different microarchitectures)

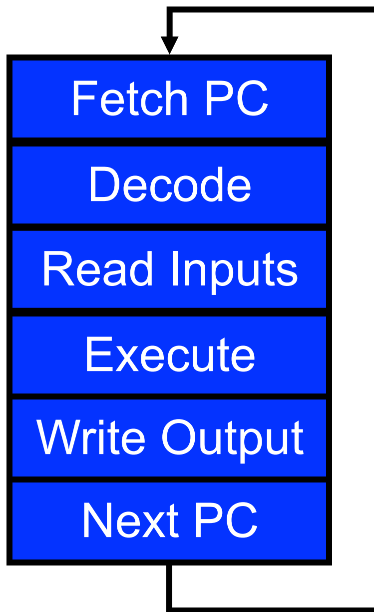
© 2012 Daniel J. Sorin
from Roth and Lebeck

Aspects of ISAs

1. The Von Neumann (pronounced NOY-muhn) model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding
3. Operations
4. Operand model
   - Where are operands stored and how do address them?
5. Datatypes and operations
6. Control

- Running example: MIPS
  - MIPS ISA designed to match actual pattern of use in programs

1. The Von Neumann Model



- Implicit model of modern computing
  - Often called Von Neumann
- Basic feature: the **program counter (PC)**
  - Defines **total order** of dynamic instructions
    - Next PC is PC++ unless insn says otherwise
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before insn X+1 starts

© 2012 Daniel J. Sorin
from Roth and Lebeck

Aspects of ISAs

1. The Von Neumann (pronounced NOY-muhn) model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding

© 2012 Daniel J. Sorin
from Roth and Lebeck

2. Instruction Format

- **Length**
  a. Fixed length
    - 32 or 64 bits (depends on architecture)
    - \+ Simple implementation: compute next PC using only this PC
    - – Code density: 32 or 64 bits for a NOP (no operation) insn?

  b. Variable length
    - – Complex implementation
    - \+ Code density

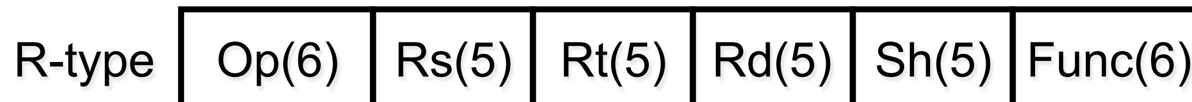
  c. Compromise with two lengths
    - Example: $MIPS_{16}$

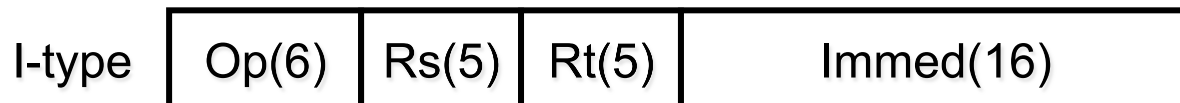
- **Encoding**
  - A few simple encodings simplify decoder implementation

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Format

- Length
  - 32-bits
  - $MIPS_{16}$: 16-bit variants of common instructions for density
- Encoding
  - 3 formats, simple encoding, 6-bit opcode (type of operation)
  - ICQ: how many operation types can be encoded in 6-bit opcode?





© 2012 Daniel J. Sorin
from Roth and Lebeck

Aspects of ISAs

1. The Von Neumann (pronounced NOY-muhn) model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding
3. Operations

© 2012 Daniel J. Sorin
from Roth and Lebeck

3. Operations

- Operation type encoded in instruction opcode
- Many types of operations
  - Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  - FP arithmetic: add, sub, mul, div, sqrt
  - Bit-wise/integer logical: and, or, xor, not, sll, srl, sra
  - Packed integer: padd, pmul, pand, por… (saturating/wraparound)

- More operation types == better ISA??
- DEC VAX computer had LOTS of operation types
  - E.g., instruction for polynomial evaluation (no joke!)
  - But many of them were rarely/never used (ICQ: Why not?)
  - We'll talk more about this issue later …

© 2012 Daniel J. Sorin
from Roth and Lebeck

Aspects of ISAs

1. The Von Neumann (pronounced NOY-muhn) model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding
3. Operations
4. Operand model
   - Where are operands stored and how do address them?

© 2012 Daniel J. Sorin
from Roth and Lebeck

4. Operations Act on Operands

- If you're adding, you need at least 3 operands
  - Two source operands, one destination operand
  - Note: operands need not be unique (e.g., A = B + A)

- Question #1: Where can operands come from?
- Question #2: And how are they specified?
- Running example: A = B + C

- Criteria for evaluating operand models
  a. **static code size**
  b. **data memory traffic**
  c. **instruction execution latency**

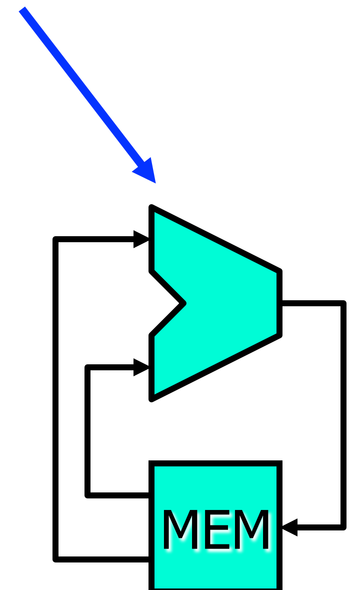
© 2012 Daniel J. Sorin
from Roth and Lebeck

Operand Model I: Memory Only

- **Memory only**

  `add A,B,C    //` mem[A] = mem[B] + mem[C]

international symbol for **Arithmetic Logic Unit (ALU)** – a piece of logic that performs arithmetic, bitwise logic, shifts, etc.



© 2012 Daniel J. Sorin
from Roth and Lebeck

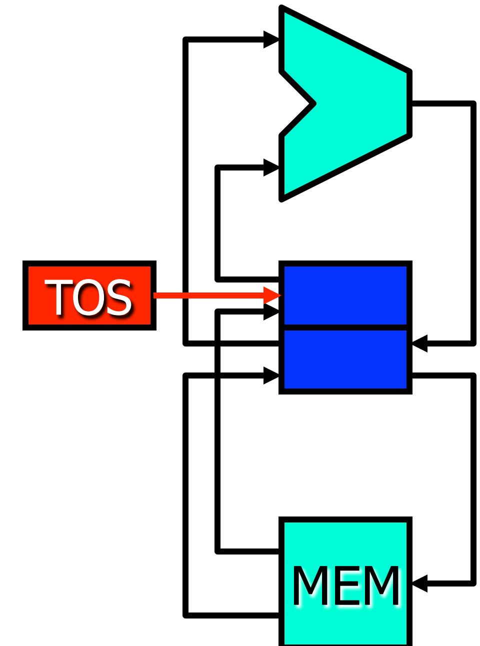
Operand Model II: Stack

- **Stack**: top of stack (TOS) is implicit operand in all insns

```
push B    //  stack[TOS++] = mem[B]
push C    //  stack[TOS++] = mem[C]
add       //  stack[TOS++] = stack[--TOS] + stack[--TOS]
pop A     //  mem[A] = stack[--TOS]
```

++x increments value of x, then returns x

x++ returns x, then increments value of x

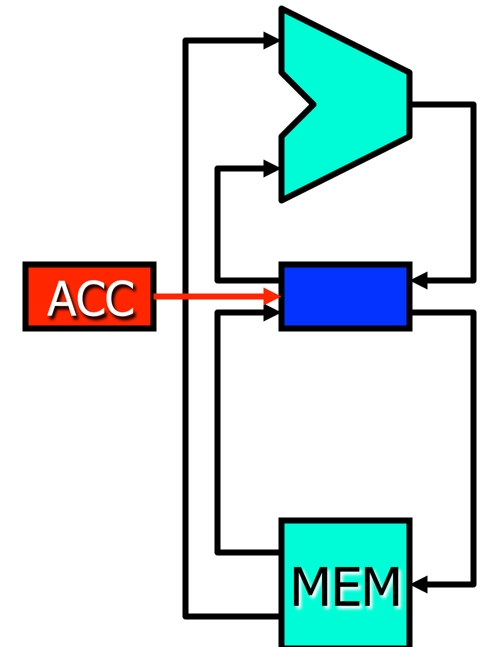


© 2012 Daniel J. Sorin
from Roth and Lebeck

Operand Model III: Accumulator

- **Accumulator**: implicit single-element stack

```
load B     // ACC = mem[B]
add C      // ACC = ACC + mem[C]
store A    // mem[A] = ACC
```



© 2012 Daniel J. Sorin
from Roth and Lebeck

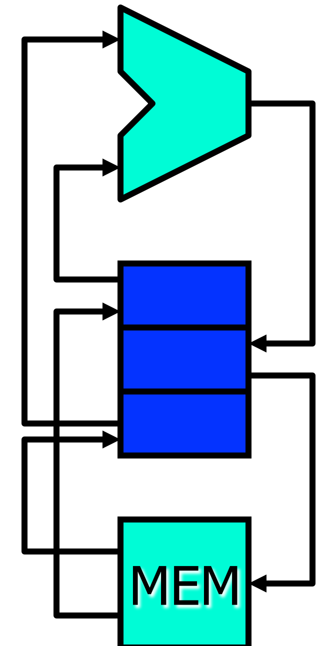
Operand Model IV: Registers

- **General-purpose registers**
- multiple explicit accumulators

```
load R1,B       //   R1 = mem[B]
add R1,C        //   R1 = R1 + mem[C]
store A,R1      //   mem[A] = R1
```

- **Load-store**
- GPR and only loads/stores access memory

```
load R1,B       //   R1 = mem[B]
load R2,C       //   R2 = mem[C]
add R3,R2,R1    //   R3 = R1 + R2
store A,R3      //   mem[A] = R3
```



© 2012 Daniel J. Sorin
from Roth and Lebeck

Operand Model Pros and Cons

- Metric I: **static code size**
  - Number of instructions needed to represent program, size of each
  - Want many implicit operands, high level instructions
  - Good → bad: memory, stack, accumulator, load-store
- Metric II: **data memory traffic**
  - Number of bytes moved to and from memory
  - Want as many long-lived operands in on-chip storage
  - Good → bad: load-store, accumulator, stack, memory
- Metric III: **instruction latency**
  - Want low latency to execute instruction
  - Good → bad: load-store, accumulator, stack, memory

- Upshot: many current ISAs are load-store

© 2012 Daniel J. Sorin
from Roth and Lebeck

How Many Registers?

- Registers faster than memory
- → use as many as possible? No!
  - One reason registers are faster is that there are **fewer of them**
    - Smaller storage structures are faster (hardware truism)
  - Another is that they are **directly addressed** (no address calc)
    - More registers → larger specifiers → fewer regs per instruction
  - **Not everything can be put in registers**
    - Structures, arrays, anything pointed-to
    - Although compilers are getting better at putting more things in
  - More registers means **more overhead**
    - Saving/restoring registers at procedure calls, context switches

- Upshot: more registers 8(IA-32)→32(MIPS) →128(IA-64)

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Operand Model

- MIPS is load-store
  - 32 32-bit integer registers
    - Actually 31: r0 is hardwired to value 0
    - Also, certain registers conventionally used for special purposes
  - 32 32-bit FP registers
    - Can also be treated as 16 64-bit FP registers
  - HI,LO: destination registers for multiply/divide

- Integer register conventions
  - Specifies guidelines for using registers
  - Allows separate function-level compilation, fast function calls
  - We'll discuss this more when we get to procedure calls

© 2012 Daniel J. Sorin
from Roth and Lebeck

Memory Operand Addressing

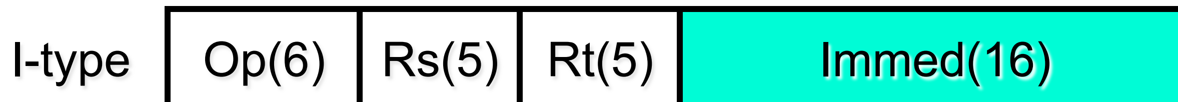
- ISAs assume **"virtual" address size**
  - Addresses are 32-bit or 64-bit
  - Program can address 2^{32} bytes (4GB) or 2^{64} (16EB)
  - Instruction is 32 bits, cannot accommodate 32+ bits for address
- **Addressing mode**: way of specifying address
  - **(Register) Indirect:** `ld R1,(R2)` R1=mem[R2]
  - **Displacement:** `ld R1,8(R2)` R1=mem[R2+8]
  - **Index-base:** `ld R1,(R2,R3)` R1=mem[R2+R3]
  - **Memory-indirect:** `ld R1,@(R2)` R1=mem[mem[R2]]
  - **Auto-increment:** `ld R1,(R2)+` R1=mem[R2++]
  - **Scaled:** `ld R1,(R2,R3,32,8)` R1=mem[R2+R3*32+8]
- ICQ: What HLL program idioms are these used for?

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Addressing Modes

- MIPS implements displacement addressing only
  - Why? Experiment on VAX (ISA with every mode) found distribution
  - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
  - 80% use displacement or register indirect (displacement=0)

- I-type instructions: 16-bit displacement
  - Is 16-bits enough?
  - Yes! VAX experiment showed 1% accesses use displacement >16



© 2012 Daniel J. Sorin
from Roth and Lebeck

Addressing Issue: Endian-ness

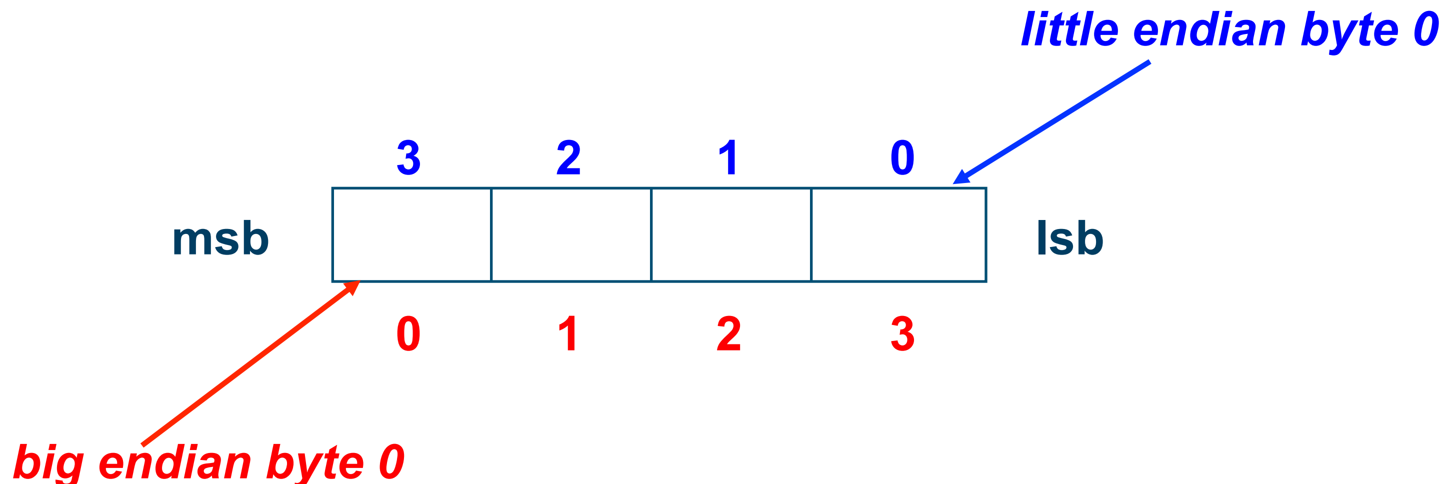
Byte Order

- Big Endian: byte 0 is 8 most significant bits

  IBM 360/370, Motorola 68k, MIPS, SPARC, HP PA-RISC

- Little Endian: byte 0 is 8 least significant bits

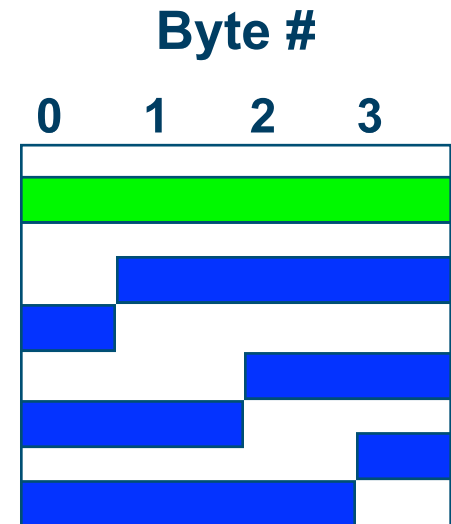
  Intel 80x86, DEC Vax, DEC/Compaq Alpha



© 2012 Daniel J. Sorin
from Roth and Lebeck

Addressing Issue: Alignment

- Alignment: require that objects fall on address that is multiple of their size
- 32-bit integer
  - Aligned if address % 4 = 0 [% is symbol for "mod"]
  - Aligned: `lw @XXXX00`
  - Not: `lw @XXXX10`
- 64-bit integer?



- What to do with (uncommon) unaligned accesses?
  - Support in hardware? Makes all accesses slow
  - Trap to software routine? Possibility
  - MIPS? Support unaligned access with 2 instructions
    `lw @XXX010 = lwl @XXX010; lwr @XXX100`

© 2012 Daniel J. Sorin
from Roth and Lebeck

Aspects of ISAs

1. The Von Neumann (pronounced NOY-muhn) model
   - Implicit structure of all modern ISAs
2. Format
   - Length and encoding
3. Operations
4. Operand model
   - Where are operands stored and how do address them?
5. Datatypes and operations

© 2012 Daniel J. Sorin
from Roth and Lebeck

5. Datatypes

- Datatypes
  - Software view: property of data
  - Hardware view: data is just bits, property of operations
  - Instruction specifies interpretation of bits (add $r3, $r2, $r1)

- Hardware datatypes
  - Integer: 8 bits (byte), 16b (half), 32b (word), 64b (long)
  - IEEE754 FP: 32b (single-precision), 64b (double-precision)
  - Packed integer: treat 64b int as 8 8b int’s or 4 16b int’s

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Datatypes (and Operations)

- Integer operations read/write 32-bits
- Integer operations specify immediate, (un)signed variants
- Immediate variants for all instructions

  `add, addu, addi, addiu`

- Load / store instructions specify word, byte, half variants

  `lw, sw, lb, lbu, lh, lhu, sb, sh`

- Loads sign-extend (or not) byte/half into 32-bits

- **Regularity**: all variants for all operations
  - Makes compilation easier

© 2012 Daniel J. Sorin
from Roth and Lebeck

Aspects of ISAs

1. The Von Neumann (pronounced NOY-muhn) model
    - Implicit structure of all modern ISAs
2. Format
    - Length and encoding
3. Operations
4. Operand model
    - Where are operands stored and how do address them?
5. Datatypes and operations
6. Control

6. Control Instructions

- Testing Condition

  Does PC = PC++?

  Depends on condition.

- Computing Target

  If PC != PC++, then what is it?

  Depends on target.

- Procedure Calls

  Calling procedure: PC = address of the called procedure

  Returning from procedure: PC = return address after completion

© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Instructions I: Testing Condition

- <u>Option I</u>: **compare and branch instructions** (not used by MIPS)

```
blti $1,10,target // if $1<10, goto target
```

  + Simple, – two ALUs: one for condition, one for target calculation

- <u>Option II</u>: **implicit condition codes (CCs)**

```
subi $2,$1,10   // sets "negative" CC
bn target  // if negative CC set, goto target
```

  + Condition codes set "for free", – implicit dependence is tricky

- <u>Option III</u>: **condition registers, separate branch insns**

```
slti $2,$1,10  // set $2 if $1<10
bnez $2,target  // if $2 != 0, goto target
```

  – Additional instructions, + one ALU per, + explicit dependence

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Conditional Branches

- MIPS uses combination of options II and III
  - Compare 2 registers and branch: **`beq`**, **`bne`**
    - Equality and inequality only
    - \+ Don't need adder for comparison
  - Compare a register to zero and branch: **`bgtz`**, **`bgez`**, **`bltz`**, **`blez`**
    - Greater / less than comparisons
    - \+ Don't need adder for comparison
  - Set explicit condition registers: **`slt`**, **`sltu`**, **`slti`**, **`sltiu`**, etc.

- Why?
  - 86% of branches in programs are (in)equalities or comparisons to 0
  - OK to take two insns to do remaining 14% of branches
  - Make the common case fast (MCCF)!

© 2012 Daniel J. Sorin
from Roth and Lebeck

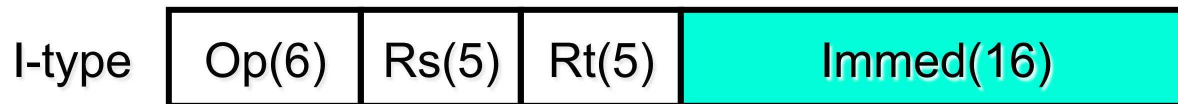
Control Instructions II: Computing Target

- Option I: **PC-relative**
  - Specify target relative to current PC
  - Used for branches and jumps within a procedure
- Option II: **Absolute**
  - Specify target with an absolute address
  - Used for procedure calls
- Option III: **Indirect** (target found in register)
  - Needed for jumping to dynamic targets
  - Used for returns, dynamic procedure calls, switches
- How far do you need to jump?
  - Typically not so far within a procedure
  - Further from one procedure to another

© 2012 Daniel J. Sorin
from Roth and Lebeck

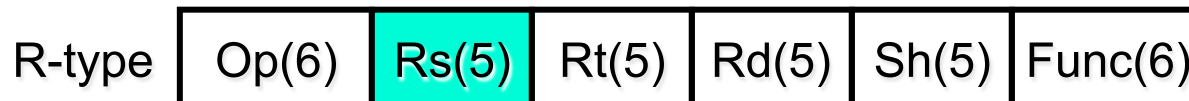
MIPS Control Instructions

- PC-relative → conditional branches: **`bne`**, **`beq`**, **`blez`**, etc.
  - 16-bit relative offset, <0.1% branches need more
  - PC = PC+4+Concat {$\text{Immed}_{15\text{-}0}$, 00} if true, PC+4 otherwise



- Absolute → unconditional jumps: **`j target`**
  - PC = Concat{$\text{PC}_{31\text{-}28}$, $\text{Target}_{25\text{-}0}$, 00}

- Indirect → Indirect jumps: **`jr $rs`**



© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Instructions III: Procedure Calls

- We "link" (i.e., remember) address of calling instruction + 4 (i.e., current PC + 4) so we can return to it after procedure

- MIPS
  - Call procedure with direct jump-and-link: **`jal address`**
    → $ra = PC+4; PC = address
  - Return from call with: **`jr $ra`**
  - Implicit return address register is **`$ra(=$31)`**

  - Call procedure with indirect jump-and-link-reg: **`jalr $rd, $rs`**
    → $rd = PC+4; PC = $rs // explicit return address register
  - Return from call with: **`jr $rd`**

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Register Naming Conventions

0	zero constant	
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		
15	t7	
16	s0	callee saves
. . .		
23	s7	
24	t8	temporary (cont' d)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	return address

© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: If-Then-Else

- Understanding programs helps with architecture
  - Know what common programming idioms look like in assembly
  - Why? How can you MCCF if you don't know what CC is?
- First control idiom: **if-then-else**

```
if (A < B) A++;       // assume A in register $s1
else B++;             // assume B in $s2
```

```
        slt  $s3,$s1,$s2     // if $s1<$s2, then $s3=1
        beqz $s3,else        // branch to else if !condition
        addi $s1,$s1,1
        j    join            // jump to join
else:   addi $s2,$s2,1
join:
```

ICQ: assembler converts "else" target of beqz into immediate → what is the immediate?

© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: Arithmetic For Loop

- Second idiom: **for loop with arithmetic induction**

```
int A[100], sum, i, N;
for (i=0; i<N; i++){           // assume: i in $s1, N in $s2
  sum += A[i];                 // &A[i] in $s3, sum in $s4
}
        sub   $s1,$s1,$s1      // initialize i to 0
  loop: slt   $t1,$s1,$s2      // if i<N, then $t1=1
        beqz  $t1,exit         // test for exit at loop header
        lw    $t1,0($s3)       // $t1 = A[i]  (not &A[i])
        add   $s4,$s4,$t1      // sum = sum + A[i]
        addi  $s3,$s3,4        // increment &A[i] by sizeof(int)
        addi  $s1,$s1,1        // i++
        j loop                 // backward jump
  exit:
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: Pointer For Loop

- Third idiom: **for loop with pointer induction**

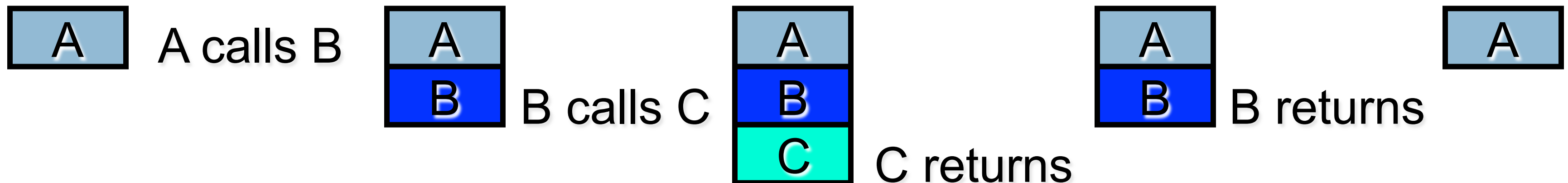
```
struct node_t { int val; struct node_t *next; };
struct node_t *p, *head;
int sum;
for (p=head; p!=NULL; p=p->next)    // p in $s1, head in $s2
    sum += p->val                   // sum in $s3

        add $s1,$s2,$0      // p = head
loop:   beq $s1,$0,exit     // if p==0 (NULL), goto exit
        lw $t1,0($s1)       // $t1 = *p = p→val
        add $s3,$s3,$t1     // sum = sum + p→val
        lw $s1,4($s1)       // p = *(p+1) = p→next
        j loop
exit:
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: Procedure Call

- In general, procedure calls obey **stack discipline**
  - Local procedure state contained in **stack frame**
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses
- Procedure stack is **in memory**
  - Distinct from operand stack which is not addressable
- Procedure linkage **implemented by convention**
  - Called procedure ("callee") expects frame to look a certain way
    - Input arguments and return address are in certain places
  - Caller "knows" this



© 2012 Daniel J. Sorin
from Roth and Lebeck

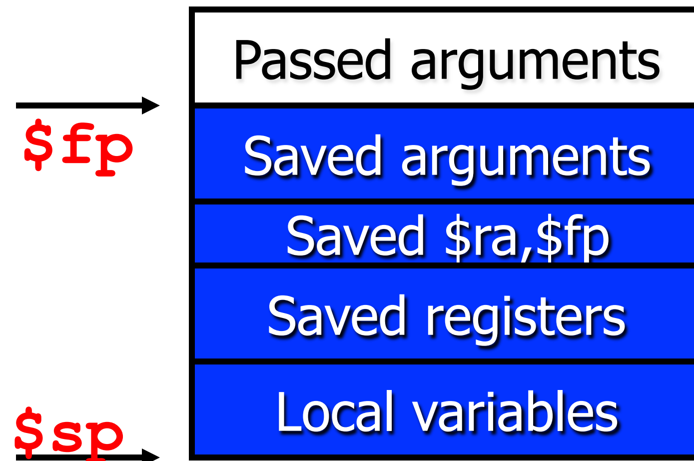
MIPS Procedure Calls

- Procedure stack implemented in software
  - No ISA support for frames: set them up with conventional stores
  - Stack is linear in memory and grows down (convention)
  - One register reserved for stack management
    - **Stack pointer** (`$sp`=`$29`): points to bottom of current frame
    - Sometimes also use **frame pointer** (`$fp`=`$30`): top of frame

- Frame layout
  - Access contents with $sp

```
sw $ra,24($sp)
```

  - Displacement addressing



© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Procedure Call: Factorial (Naïve version)

```
fact:  addi $sp,$sp,-128      // open frame (32 words of storage)
       sw $ra,124($sp)        // save 31 registers
       sw $1,120($sp)
       sw $2,116($sp)
       …
       lw $s0,128($sp)        // read argument from caller's frame
       subi $s1,$s0,1
       sw $s1,0($sp)          // store (argument-1) to frame
       jal fact               // recursive call
       lw $s1,-4($sp)         // read return value from frame
       mul $s1,$s1,$s0        // multiply
       …
       lw $2,116($sp)         // restore all 32 registers
       lw $1,120($sp)
       lw $ra,124($sp)
       sw $s1,124($sp)        // return value below caller's frame
       addi $sp,$sp,128       // collapse frame
       jr $ra                 // return
```

Note: code ignores base case of recursion (should return 1 if arg==1)

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Calls and Register Convention

- Some inefficiencies with basic frame mechanism
  - **Registers**: do all need to be saved/restored on every call/return?
  - **Arguments**: must all be passed on stack?
  - **Returned values**: are these also communicated via stack?
  - No! Fix with **register convention**

    `$2,$3($v0,$v1)`: expression evaluation and return **v**alues

    `$4-$7($a0-$a3)`: function **a**rguments

    `$8-$15,$24,$25($t0-$t9)`: caller saved **t**emporaries

      - A saves before calling B only if needed after B returns

    `$16-$23($s0-$s7)`: callee **s**aved

      - A needs after B returns, B saves if it uses also
  - We'll discuss complete set of MIPS registers and conventions soon

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Factorial: Take II (Using Conventions)

```
fact: addi $sp,$sp,-8      // open frame (2 words)
      sw $ra,4($sp)        // save return address
      sw $s0,0($sp)        // save $s0
      …
      add $s0,$a0,$0       // copy $a0 to $s0
      subi $a0,$a0,1       // pass arg via $a0
      jal fact             // recursive call
      mul $v0,$s0,$v0      // value returned via $v0
      …
      lw $s0,0($sp)        // restore $s0
      lw $ra,4($sp)        // restore $ra
      addi $sp,$sp,8       // collapse frame
      jr $ra               // return, value in $v0
```

+ Pass/return values via **$a0-$a3** and **$v0-$v1** rather than stack
+ Save/restore 2 registers (**$s0,$ra**) rather than 31 (excl. $0)

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Swap

```
int equal(int a1, int a2
{
   int tsame;
   tsame = 0;
   if (a1 == a2)
       tsame = 1;
   return(tsame);
}
main()
{
   int x,y,same;
   x = 43;
   y = 2;
   same = equal(x,y);
   // other computation
}
```

Address	Instruction
0x10000	addi $a0,$0,43
0x10004	addi $a1,$0,2
0x10008	jal 0x30408
0x1000c	??

Address	Instruction
0x30408	addi $v0,$0,0
0x3040c	bne $a0,$a1,4
0x30410	addi $v0,$0,1
0x30414	jr $ra

PC	$ra=$31
0x10000	??
0x10004	??
0x10008	??
0x30408	0x1000c
0x3040c	0x1000c
0x30410	0x1000c
0x30414	0x1000c
0x1000c	0x1000c

© 2012 Daniel J. Sorin
from Roth and Lebeck

Procedure Call Gap

<u>ISA Level</u>

- Call and return instructions

<u>C/C++ Level</u>

- Local name scope
  - Change tsame to same
- Recursion
- Arguments and return value (functions)

<u>Assembly Level</u>

- Must bridge gap between HLL and ISA
- Supporting local names
- Passing arguments (arbitrary number?)

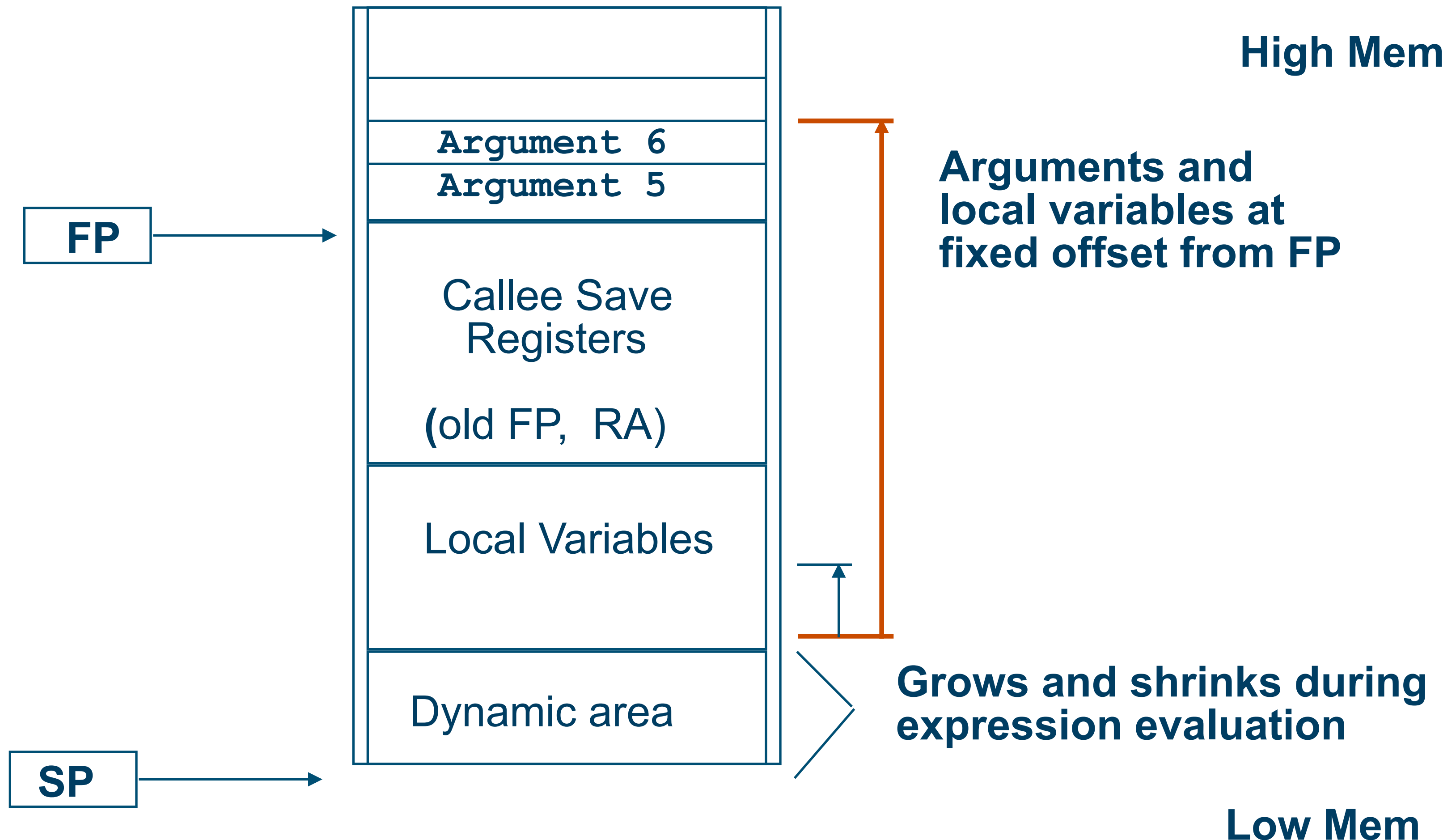
© 2012 Daniel J. Sorin
from Roth and Lebeck

Stack Frame

- Procedures use a frame in the stack to:
  - Hold values passed to procedures as arguments
  - Save registers that the callee procedure may modify, but which the procedure's caller does not want changed
  - To provide space for local variables
    (variables with local scope)
  - To evaluate complex expressions

© 2012 Daniel J. Sorin
from Roth and Lebeck

Stack Frame & Call-Return Linkage



© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Register Naming Conventions

0	zero constant	
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		
15	t7	

16	s0	callee saves
. . .		
23	s7	
24	t8	temporary (cont' d)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	Stack pointer
30	fp	frame pointer
31	ra	return address

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS/GCC Procedure Calling Conventions

Calling Procedure

- Step-1: Pass the arguments
  - First four arguments (arg0-arg3) are passed in registers $a0-$a3
  - Remaining arguments are pushed onto the stack
    (in reverse order, arg5 is at the top of the stack)

- Step-2: Save caller-saved registers
  - Save registers $t0-$t9 if they contain live values at the call site

- Step-3: Execute a jal instruction

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS/GCC Procedure Calling Conventions (cont.)

Called Procedure (i.e., Callee) @ Start

- Step-1: Establish stack frame
  - Subtract the frame size from the stack pointer

    subiu $sp, $sp, <frame-size>
  - Typically, minimum frame size is 32 bytes (8 words)
- Step-2: Save callee saved registers in the frame
  - Register $fp is always saved (by convention)
  - Register $ra is saved if routine makes a call
  - Registers $s0-$s7 are saved if they are used
- Step-3: Establish frame pointer
  - Add the stack <frame-size> - 4 to the address in $sp

    addiu $fp, $sp, <frame-size> - 4

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS/GCC Procedure Calling Conventions (cont.)

Called Procedure (i.e., Callee) @ End

- Step-1: Put returned values in registers $v0 and $v1 (if values are returned)
- Step-2: Restore callee-saved registers
  - Restore $fp and other saved registers: $ra, $s0 - $s7
- Step-3: Pop the stack
  - Add the frame size to $sp

    ```
    addiu $sp, $sp, <frame-size>
    ```
- Step-4: Return
  - Jump to the address in $ra

    ```
    jr      $ra
    ```

MIPS Factorial: Take II (Using Conventions)

```
fact: addi $sp,$sp,-8      // open frame (2 words)
      sw $ra,4($sp)        // save return address
      sw $s0,0($sp)        // save $s0
      ...
      add $s0,$a0,$0       // copy $a0 to $s0
      subi $a0,$a0,1       // pass arg via $a0
      jal fact             // recursive call
      mul $v0,$s0,$v0      // value returned via $v0
      ...
      lw $s0,0($sp)        // restore $s0
      lw $ra,4($sp)        // restore $ra
      addi $sp,$sp,8       // collapse frame
      jr $ra               // return, value in $v0
```

+ Pass/return values via **`$a0-$a3`** and **`$v0-$v1`** rather than stack
+ Save/restore 2 registers (**`$s0,$ra`**) rather than 31 (excl. $0)

© 2012 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: Call by Reference

- Passing arguments
  - **By value**: pass contents [$sp+4] in $a0

```
int n;                    // n in 4($sp)
foo(n);
       lw $a0,4(sp)
       jal foo
```

  - **By reference**: pass address $sp+4 in $a0

```
int n;                    // n in 4($sp)
bar(&n);
       add $a0,$sp,4
       jal bar
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Instructions and Pseudo-Instructions

- Assembler helps give compiler illusion of regularity
  - Processor does not implement **all** possible instructions
  - Assembler accepts all insns, but some are **pseudo-insns**
    - Assembler translates these into native insn (insn sequences)
  - MIPS example #1

```
sgt $s3,$s1,$s2  // set $s3=1 if $s1>$s2

slt $s3,$s2,$s1  // set $s3=1 if $s2<$s1
```

  - MIPS example #2

```
div $s1,$s2,$s3  // want div to put result in $s1

div $s1,$s2,$s3   // div puts result in $lo
mflo $s1          // move it from $lo to $s1
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Outline

- ISAs in General
- MIPS Assembly Programming
- Other Instruction Sets

© 2012 Daniel J. Sorin
from Roth and Lebeck

But first: SPIM

- SPIM is a program that simulates the behavior of MIPS32 computers
  - Can run MIPS32 assembly language programs
  - You will use SPIM to run/test the assembly language programs you write for homeworks in this class
- Two flavors of same thing:
  - spim: command line interface
  - xspim: xwindows interface

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Assembly Language

- One instruction per line
- Numbers are base-10 integers or Hex with leading 0x
- Identifiers: alphanumeric, _, . string starting in a letter or _
- Labels: identifiers starting at the beginning of a line followed by ":"
- Comments: everything following # until end-of-line
- Instruction format: Space and "," separated fields
  - [Label:] <op> reg1, [reg2], [reg3]   [# comment]
  - [Label:] <op> reg1, offset(reg2)   [# comment]
  - .Directive [arg1], [arg2], . . .

© 2012 Daniel J. Sorin
from Roth and Lebeck

MIPS Pseudo-Instructions

- Pseudo-instructions: extend the instruction set for convenience
- Examples
  - `move $2, $4                # $2 = $4, (copy $4 to $2)`

    Translates to:

    `add $2, $4, $0`
  - `li $8, 40                  # $8 = 40, (load 40 into $8)`

    `addi  $8, $0, 40`
  - `sd $4, 0($29)              # mem[$29] = $4; Mem[$29+4] = $5`

    `sw $4, 0($29)`

    `sw $5, 4($29)`
  - `la $4, 0x1000056c          # Load address $4 = <address>`

    `lui $4, 0x1000            # load upper immediate (lui)`

    `ori $4, $4, 0x056c        # or immediate (ori)`

© 2012 Daniel J. Sorin
from Roth and Lebeck

Assembly Language (cont.)

- Directives: tell the assembler what to do
- Format "."<string> [arg1], [arg2] . . .

- Examples

```
.data [address]          # start a data segment
.text  [address]         # start a code segment
.align n                 # align segment on 2^n  byte boundary
.ascii <string>          # store a string in memory
.asciiz <string>         # store null-terminated string in memory
.word w1, w2, . . . , wn # store n words in memory
```

Let's see how these get used in programs …

© 2012 Daniel J. Sorin
from Roth and Lebeck

A Simple Program

- Add two numbers x and y:

```
   .text               # declare text segment
   .align 2            # align it on 4-byte (word) boundary
main:                  # label for main
   la   $3, x          # load address of x into R3 (pseudo-inst)
   lw   $4, 0($3)      # load value of x into R4
   la   $3, y          # load address of y into R3 (pseudo-inst)
   lw   $5, 0($3)      # load value of y into R5
   add  $6, $4, $5     # compute x+y
   jr   $31            # return to calling routine

   .data               # declare data segment
   .align 2            # align it on 4-byte boundary
x: .word 10            # initialize x to 10
y: .word 3             # initialize y to 3
```

Note: program doesn't obey register conventions

© 2012 Daniel J. Sorin
from Roth and Lebeck

Another example: The C / C++ code

```
#include <iostream.h>

int main ( )
{
   int i;
   int sum = 0;
   for(i=0; i <= 100; i++)
       sum = sum + i*i ;
   cout << "The answer is " << sum << endl;
}
```

Let's write the assembly …

© 2012 Daniel J. Sorin
from Roth and Lebeck

Assembly Language Example 1

```
    .text
    .align    2
main:
    move $14, $0 # i = 0
    move $15, $0 # tmp = 0
    move $16, $0 # sum = 0
loop:
    mul $15, $14, $14  # tmp = i*i
    add $16, $16, $15  # sum = sum + tmp
    addi $14, $14, 1   # i++
    ble $14, 100, loop # if i < 100, goto loop

# how are we going to print the answer here?
# and how are we going to exit the program?
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

System Call Instruction

- System call is used to communicate with the operating system and request services (memory allocation, I/O)
  - syscall instruction in MIPS
- SPIM supports "system-call-like"

1. Load system call code into register $v0
   - Example: if $v0==1, then syscall will print an integer
2. Load arguments (if any) into registers $a0, $a1, or $f12 (for floating point)
3. syscall

- Results returned in registers $v0 or $f0

© 2012 Daniel J. Sorin
from Roth and Lebeck

SPIM System Call Support

code	service	ArgType	Arg/Result
1	print	int	$a0
2	print	float	$f12
3	print	double	$f12
4	print	string	$a0 (string address)
5	read	integer	integer in $v0
6	read	float	float in $f0
7	read	double	double in $f0 & $f1
8	read	string	$a0=buffer, $a1=length
9	sbrk	$a0=amount	address in $v0
10	exit		

© 2012 Daniel J. Sorin
from Roth and Lebeck

Echo number and string

```
.text
main:
   li   $v0, 5         # code to read an integer
   syscall             # do the read (invokes the OS)
   move $a0, $v0       # copy result from $v0 to $a0

   li   $v0, 1         # code to print an integer
   syscall             # print the integer

   li   $v0, 4         # code to print string
   la   $a0, nln       # address of string (newline)
   syscall

# code continues on next slide …
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Echo Continued

```
    li   $v0, 8         # code to read a string
    la   $a0, name      # address of buffer (name)
    li   $a1, 8         # size of buffer (8 bytes)
    syscall

    la   $a0, name      # address of string to print
    li   $v0, 4         # code to print a string
    syscall

    jr   $31            # return

    .data
    .align 2
name:   .word 0,0             # initialize two words to zero
nln:    .asciiz "\n"          # initialize an ascii character
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Example 2

```
# Program to add together list of 9 numbers
        .text                       # Code
        .align  2
        .globl  main
main:                               # MAIN procedure Entrance
        subu    $sp, 40             #\ Push the stack
        sw      $ra, 36($sp)        # \ Save return address
        sw      $s3, 32($sp)        #  \
        sw      $s2, 28($sp)        #   > Entry Housekeeping
        sw      $s1, 24($sp)        #  / save registers on stack
        sw      $s0, 20($sp)        # /
        move    $v0, $0             #/ initialize exit code to 0
        move    $s1, $0             #\
        la      $s0, list           # \ Initialization
        la      $s2, msg            # /
        la      $s3, list+36        #/
```

Example 2 (cont.)

```
#                       Main code segment

again:                                  #   Begin main loop
        lw      $t6, 0($s0)             #\
        addu    $s1, $s1, $t6           #/  Actual "work"
                                        #    SPIM I/O
        li      $v0, 4                  #\
        move    $a0, $s2                # >  Print a string
        syscall                         #/
        li      $v0, 1                  #\
        move    $a0, $s1                # >  Print a number
        syscall                         #/
        li      $v0, 4                  #\
        la      $a0, nln                # > Print a string (eol)
        syscall                         #/

        addu    $s0, $s0, 4               #\ index update and
        bne     $s0, $s3, again           #/  end of loop
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Example 2 (cont.)

```
#                      Exit Code

        move    $v0, $0         #\
        lw      $s0, 20($sp)    # \
        lw      $s1, 24($sp)    #  \
        lw      $s2, 28($sp)    #   \ Closing Housekeeping
        lw      $s3, 32($sp)    #   /   restore registers
        lw      $ra, 36($sp)    #  / load return address
        addu    $sp, 40         # / Pop the stack
        jr      $ra             #/    exit(0) ;
        .end    main            #  end of program

#               Data Segment

        .data                   # Start of data segment
list:   .word   35, 16, 42, 19, 55, 91, 24, 61, 53
msg:    .asciiz "The sum is "
nln:    .asciiz "\n"
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

Some Details/Quirks of the MIPS ISA

- Register zero always has the value zero
  - Even if you try to write it!
- jal puts the return address PC+4 into the link register ($ra)
- All instructions change all 32 bits of destination register (lui, lb, lh) and read all 32 bits of sources (add, sub, and, or, …)

- Immediate arithmetic and logical instructions are extended:
  - logical immediates are zero-extended to 32 bits
  - arithmetic immediates are sign-extended to 32 bits
- lb and lh extend data as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended

© 2012 Daniel J. Sorin
from Roth and Lebeck

Outline

- Instruction Sets in General
- MIPS Assembly Programming
- Other Instruction Sets
  - Goals of ISA Design
  - RISC vs. CISC
  - Intel x86 (IA-32)

© 2012 Daniel J. Sorin
from Roth and Lebeck

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Implementability**
  - Easy to design high-performance implementations (i.e., microarchitectures)?
- **Compatibility**
  - Easy to maintain programmability as languages, programs evolve?
  - Easy to maintain implementability as technology evolves?

© 2012 Daniel J. Sorin
from Roth and Lebeck

Programmability

- Easy to express programs efficiently?
  - For whom?
- **Human**
  - Want high-level coarse-grain instructions
    - As similar to HLL as possible
  - This is the way ISAs were pre-1985
    - Compilers were terrible, most code was hand-assembled
- **Compiler**
  - Want low-level fine-grain instructions
    - Compiler can't tell if two high-level idioms match exactly or not
  - This is the way most post-1985 ISAs are
    - Optimizing compilers generate much better code than humans
    - ICQ: Why are compilers better than humans?

© 2012 Daniel J. Sorin
from Roth and Lebeck

Implementability

- Every ISA can be implemented
  - But not every ISA can be implemented well
  - Bad ISA → bad microarchitecture (slow, power-hungry, etc.)
- We'd like to use some of these high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution
  - We'll discuss these later in the semester
- Certain ISA features make these difficult
  - Variable length instructions
  - Implicit state (e.g., condition codes)
  - Wide variety of instruction formats

© 2012 Daniel J. Sorin
from Roth and Lebeck

Compatibility

- Few people buy new hardware if it means they have to buy new software too
  - Intel was the first company to realize this
  - ISA must stay stable, no matter what (microarch. can change)
    - x86 is one of the ugliest ISAs EVER, but survives
  - Intel then forgot this lesson: IA-64 (Itanium) is new ISA
- **Backward compatibility**: very important
  - New processors must support old programs (can't drop features)
- **Forward (upward) compatibility**: less important
  - Old processors must support new programs
  - Old processors emulate new instructions in low-level software

© 2012 Daniel J. Sorin
from Roth and Lebeck

Compatibility in the Age of VMs

- Virtual machine (VM): piece of software that emulates behavior of hardware platform
  - Examples: VMWare, Xen, Simics
- VM emulates target system while running on host system
  - Key: host and target ISAs do not have to be the same!
  - Ex: On x86 desktop, I can run VM that emulates MIPS processor
  - Upshot: you can run code of target ISA on host with different ISA → don't need to buy x86 box to run legacy x86 code
  - Very cool technology that's commonly used
- ICQ: given a VM, does ISA compatibility really matter?

© 2012 Daniel J. Sorin
from Roth and Lebeck

RISC vs. CISC

- **RISC**: reduced-instruction set computer
  - Coined by P+H in early 80's (ideas originated earlier)
- **CISC**: complex-instruction set computer
  - Not coined by anyone, term didn't exist before "RISC"

- Religious war (one of several) started in mid 1980's
  - RISC (MIPS, Alpha, Power) "won" the technology battles
  - CISC (IA32 = x86) "won" the commercial war
    - Compatibility a stronger force than anyone (but Intel) thought
    - Intel beat RISC at its own game … more on this soon

© 2012 Daniel J. Sorin
from Roth and Lebeck

The Setup

- Pre-1980
  - Bad compilers
  - Complex, high-level ISAs
  - Slow, complicated, multi-chip microarchitectures
- Around 1982
  - Advances in VLSI made single-chip microprocessor possible…
    - Speed by integration, on-chip wires much faster than off-chip
  - …but only for very small, very simple ISAs
  - Compilers had to get involved in a big way
- **RISC manifesto**: create ISAs that…
  - Simplify single-chip implementation
  - Facilitate optimizing compilation

© 2012 Daniel J. Sorin
from Roth and Lebeck

The RISC Tenets

- **Single-cycle execution (simple operations)**
  - CISC: many multi-cycle operations
- **Load/store architecture**
  - CISC: register-memory and memory-memory instructions
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance

Summary

(1) Make it easy to implement in hardware

(2) Make it easy for compiler to generate code

© 2012 Daniel J. Sorin
from Roth and Lebeck

PowerPC ISA → POWER ISA

- RISC-y, very similar to MIPS
- Some differences:
  - Indexed addressing mode (register+register)
    - `lw $t1,$a0,$s3   # $t1 = mem[$a0+$s3]`
  - Update addressing mode
    - `lw $t1,4($a0)    # $t1 = mem[$a0+4]; $a0 += 4;`
  - Dedicated counter register
    - `bc loop # ctr--; branch to loop if ctr != 0`
- In general, though, similar to MIPS

© 2012 Daniel J. Sorin
from Roth and Lebeck

Intel 80x86 ISA (aka x86 or IA-32)

- Binary compatibility across generations
- 1978: 8086, 16-bit, registers have dedicated uses
- 1980: 8087, added floating point (stack)
- 1982: 80286, 24-bit
- 1985: 80386, 32-bit, new instrs → GPR almost
- 1989-95: 80486, Pentium, Pentium II
- 1997: Added MMX instructions (for graphics)
- 1999: Pentium III
- 2002: Pentium 4
- 2004: “Nocona” 64-bit extension (to keep up with AMD)
- 2006: Core2
- 2007: Core2 Quad

© 2012 Daniel J. Sorin
from Roth and Lebeck

Intel x86: The Penultimate CISC

- DEC VAX was ultimate CISC, but x86 (IA-32) is close
  - Variable length instructions: 1-16 bytes
  - Few registers: 8 and each one has a special purpose
  - Multiple register sizes: 8,16,32 bit (for backward compatibility)
  - Accumulators for integer instrs, and stack for FP instrs
  - Multiple addressing modes: indirect, scaled, displacement
  - Register-register, memory-register, and memory-register insns
  - Condition codes
  - Instructions for memory stack management (push, pop)
  - Instructions for manipulating strings (entire loop in one instruction)
- Summary: yuck!

© 2012 Daniel J. Sorin
from Roth and Lebeck

80x86 Registers and Addressing Modes

- Eight 32-bit registers (not truly general purpose)
  - EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- Six 16-bit registers for code, stack, & data
- 2-address ISA
  - One operand is both source and destination
- NOT a Load/Store ISA
  - One operand can be in memory

© 2012 Daniel J. Sorin
from Roth and Lebeck

80x86 Addressing Modes

- Register Indirect
  - mem[reg]
  - not ESP or EBP register
- Base + displacement (8 or 32 bit)
  - mem[reg + const]
  - not ESP or EBP
- Base + scaled index
  - mem[reg + (2^{scale} x index)]
  - scale = 0,1,2,3
  - base any GPR, index not ESP
- Base + scaled index + displacement
  - mem[reg + (2^{scale} x index) + displacement]
  - scale = 0,1,2,3
  - base any GPR, index not ESP

© 2012 Daniel J. Sorin
from Roth and Lebeck

Condition Codes

- Both Power ISA and x86 ISA have condition codes
- Special HW register that has values set as side effect of instruction execution
- Example conditions
  - Zero
  - Negative
- Example use

```
subi $t0, $t0, 1
bz  loop  // branch to loop if result of previous instruction is zero
```

© 2012 Daniel J. Sorin
from Roth and Lebeck

80x86 Instruction Encoding

- Variable size 1-byte to 17-bytes
- Examples
  - Jump (JE) 2-bytes
  - Push 1-byte
  - Add Immediate 5-bytes
- W bit says 32-bits or 8-bits
- D bit indicates direction
  - memory → reg or reg → memory
  - movw EBX, [EDI + 45]
  - movw [EDI + 45], EBX

© 2012 Daniel J. Sorin
from Roth and Lebeck

Decoding x86 Instructions

- Is a &$%!# nightmare!
- Instruction length is variable from 1 to 17 bytes
- Crazy "formats" → register specifiers move around
- But key instructions not terrible
- Yet, everything must work correctly

© 2012 Daniel J. Sorin
from Roth and Lebeck

How Intel Won Anyway

- x86 won because it was the first 16-bit chip by 2 years
  - IBM put it into its PCs because there was no competing choice
  - Rest is historical inertia and "financial feedback"
    - x86 is most difficult ISA to implement and do it fast but…
    - Because Intel (and AMD) sells the most processors…
    - It has the most money…
    - Which it uses to hire more and better engineers…
    - Which it uses to maintain competitive performance …
    - And given equal performance compatibility wins…
    - So Intel (and AMD) sells the most processors…

- Moore's law has helped Intel in a big way
  - Most engineering problems can be solved with more transistors

© 2012 Daniel J. Sorin
from Roth and Lebeck

Current Approach: Pentium Pro and beyond

- Instruction decode logic translates into μops
- Fixed-size instructions moving down execution path
- Execution units see only μops

+ Faster instruction processing with backward compatibility
+ Execution unit as fast as RISC machines like MIPS
– Complex decoding
– We work with MIPS to keep decoding simple/clean
– Learn x86 on the job!

© 2012 Daniel J. Sorin
from Roth and Lebeck

Aside: Complex Instructions

- More powerful instructions → not necessarily faster execution
- E.g., string copy or polynomial evaluation

- Option 1: use “repeat” prefix on memory-memory move
  - Custom string copy
- Option 2: use a loop of loads and stores through registers
  - General purpose move through simple instructions

- Option 2 is often faster on same machine

© 2012 Daniel J. Sorin
from Roth and Lebeck

Concluding Remarks

1. Keep it simple and regular
    - Uniform length instructions
    - Fields always in same places
2. Keep it simple and fast
    - Small number of registers
3. Make sure design can be pipelined (will learn soon)
4. Make the common case fast

- Compromises inevitable → there is no perfect ISA

© 2012 Daniel J. Sorin
from Roth and Lebeck

Outline

- Instruction Sets in General
- MIPS Assembly Programming
- Other Instruction Sets

© 2012 Daniel J. Sorin
from Roth and Lebeck