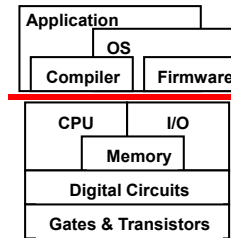# ECE/CS 250
Introduction to Computer Architecture

Instruction Set Architecture (ISA) and Assembly Language
Copyright Daniel J. Sorin
Duke University

Slides are derived from work by
Amir Roth (Penn) and Alvy Lebeck (Duke)

1

Instruction Set Architecture (ISA)

Application		
	OS	
Compiler		Firmware

CPU	I/O
	Memory

Digital Circuits

Gates & Transistors

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

2

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

3

Outline

- What is an Instruction Set Architecture (ISA)?
- Assembly programming (in the MIPS ISA)
- Other ISAs

4

Reminder: What Is a Computer?

- Machine that has storage (to hold instructions and data) and that executes instructions
- Storage (as seen by each running program)
  - Memory:
    - 2^{32} bytes (=4GBytes) 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
  - Etc.

5

What Is An ISA?

- **Functional & precise** specification of computer
  - What storage does it have? How many registers? How much memory?
  - What instructions does it have?
  - How do we specify operands for instructions?

  And how do we specify all of this in bits?

- ISA = **"contract"** between software and hardware
  - Sort of like "hardware API"
  - Specifies what hardware will do when executing each instruction

6

Architecture vs. Microarchitecture

- **ISA specifies WHAT hardware does, not HOW it does it**
  - No guarantees regarding these issues:
    - How operations are implemented (could involve hamsters!)
    - Which operations are fast and which are slow
    - Which operations take more power and which take less
  - These issues are determined by **microarchitecture**
    - Microarchitecture = how hardware implements architecture
    - Can be any number of microarchitectures that implement same architecture (Pentium and Core i7 are almost same architecture, but very different microarchitectures)
- Strictly speaking, ISA <u>is</u> 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 architecture is implemented

7

Von Neumann Model of a Computer

Fetch PC
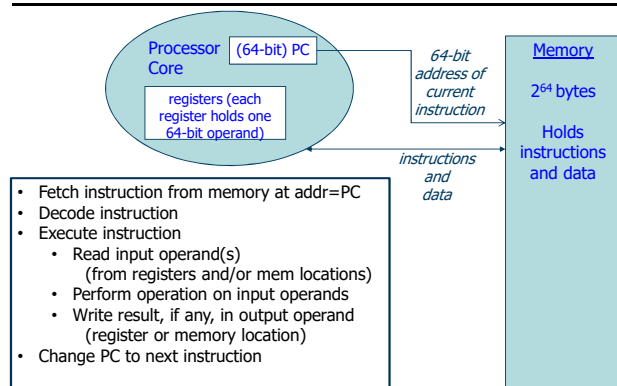Decode
Read Inputs
Execute
Write Output
Next PC
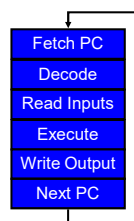
- Implicit model of all modern ISAs
  - Often called Von Neumann, but in ENIAC before
    - "von NOY-man" (German name)
  - Everything is in memory (and perhaps elsewhere)
    - instructions and data
- Key feature: **program counter (PC)**
  - PC is memory address of currently executing instruction (abbrev: instr or insn)
  - Next PC is PC + length_of_instruction unless instruction specifies otherwise
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instr X finishes before instr X+1 starts

8

An Abstract 64-bit Von Neumann Architecture

Processor Core (64-bit) PC

registers (each register holds one 64-bit operand)

64-bit address of current instruction

instructions and data

Memory
2^{64} bytes
Holds instructions and data

- Fetch instruction from memory at addr=PC
- Decode instruction
- Execute instruction
  - Read input operand(s) (from registers and/or mem locations)
  - Perform operation on input operands
  - Write result, if any, in output operand (register or memory location)
- Change PC to next instruction

9

Outline

- What is an ISA?
- Assembly programming (in the MIPS ISA)
- Other ISAs

10

MIPS

- MIPS is ISA used in textbook and in class
  - Note: somewhat simplified from actual MIPS
- MIPS not currently used in high-performance processors
  - Still exists in embedded processors
- Why teach MIPS?
  - It's easiest to learn → you can learn x86 or ARM later
- 32-bit architecture
  - Every address is 32 bits
  - 2^{32} bytes of memory
  - A "word" of data is 32 bits
- Processor has 32 registers, each of which is 32-bits (4B)
  - Named register 0 ($0) through register 31 ($31)

11

Simple, Running Example

```
// silly C code

int temp;
int sum=0;
int x=2;
int y=3;
while (true) {
    temp = x + y;
    sum = sum + temp;
}
```

```
// equivalent MIPS assembly code

loop:   lw $8, Memory[1004]
        lw $9, Memory[1008]
        add $10, $8, $9
        add $11, $11, $10
        j loop
```

OK, so what does this assembly code mean?
Let's dig into each line ...

12

Simple, Running Example

```
loop:   lw $8, Memory[1004]      # read x from memory
        lw $9, Memory[1008]
        add $10, $8, $9
        add $11, $11, $10
        j loop
```

NOTES

"loop:" = line label (in case we need to refer to this instruction's PC)
lw = "load word" → load (read) a word (32 bits =4B) from memory
$8 = "register 8" → put result read from memory into register 8
Memory[1004] = address in memory to read from (where x lives, i.e., where compiler decided to put x)

So now the value of x is in $8

- Note: almost all MIPS instructions put destination (where result gets written) first
  - in this case, destination is $8

© Daniel J. Sorin
from Hilton, Lebeck, Roth

13

Simple, Running Example

```
loop:   lw $8, Memory[1004]
        lw $9, Memory[1008]    # read y from memory
        add $10, $8, $9
        add $11, $11, $10
        j loop
```

NOTES

lw = "load word" → read a word (32 bits) from memory
$9 = "register 9" → put result read from memory into register 9
Memory[1008] = address in memory to read from (where y lives)

So now the value of y is in $9

© Daniel J. Sorin
from Hilton, Lebeck, Roth

14

Simple, Running Example

```
loop:   lw $8, Memory[1004]
        lw $9, Memory[1008]
        add $10, $8, $9        # temp = x + y
        add $11, $11, $10
        j loop
```

NOTES

add $10, $8, $9= add what's in $8 to what's in $9 and put result in $10
We're using $10 to hold temp

Note: remember that destination ($10) is listed first

© Daniel J. Sorin
from Hilton, Lebeck, Roth

15

Simple, Running Example

```
loop:   lw $8, Memory[1004]
        lw $9, Memory[1008]
        add $10, $8, $9
        add $11, $11, $10                # sum = sum + temp
        j loop
```

NOTES

add $11, $11, $10= add what's in $11 to what's in $10 and put result in $11
We're using $11 to hold sum (note: never set it equal to zero, as in C code)

Note: this instruction overwrites previous value in $11

© Daniel J. Sorin
from Hilton, Lebeck, Roth

16

Simple, Running Example

```
loop:   lw $8, Memory[1004]
        lw $9, Memory[1008]
        add $10, $8, $9
        add $11, $11, $10
        j loop
```

NOTES

j = "jump"
loop = PC of instruction at label "loop" (the first lw instruction above)
This instruction sets next_PC to address labeled by "loop"

Note: all other instructions in this code set next_PC = PC++

© Daniel J. Sorin
from Hilton, Lebeck, Roth

17

Assembly Code Format

- Every line of program has:
        label (optional) – followed by ":"
        instruction
        comment (optional) – follows "#"

```
loop:   lw $8, Memory[1004]    # read from address 1004
        lw $9, Memory[1008]
        add $10, $8, $9
        add $11, $11, $10
        j loop                 # jump back to instr at label loop
```

Note: label is just convenient way to represent address so programmers don't have to worry about numerical addresses

© Daniel J. Sorin
from Hilton, Lebeck, Roth

18

Assembly ←→ Machine Code

- Every MIPS assembly instruction has a unique 32-bit representation
  - add $3, $2, $7   ←→ 0000001010 0100
  - lw $8, Mem[1004] ←→ 1000111101 0111

  don't trust these!

- Computer hardware deals with bits
- We find it easier to look at assembly
  - But they're equivalent! No magical transformation.

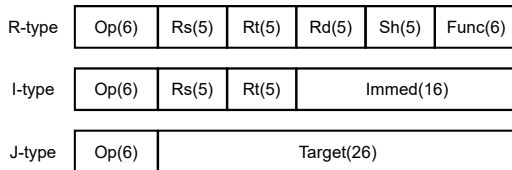- So how do we represent each MIPS assembly instruction with string of 32 bits?

MIPS Instruction Format

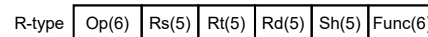| opcode (6 bits)	operands (26 bits)
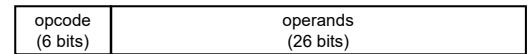
- **opcode** = what type of operation to perform
  - add, subtract, load, store, jump, etc.
  - 6 bits → how many types of operations can we specify?
- operands specify: inputs, output (optional), and next PC (optional)
- operands can be specified with:
  - register numbers
  - memory addresses
  - **immediates** (values wedged into last 26 bits of instruction)

MIPS Instruction Formats

- 3 variations on theme from previous slide
  - All MIPS instructions are either R, I, or J type
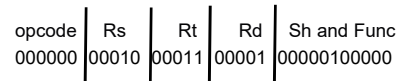 - Note: all instructions have opcode as first 6 bits

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)	

J-type	Op(6)	Target(26)	

MIPS Format – R-Type Example

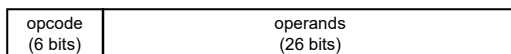| R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)

- add $1, $2, $3   # $1 = $2 + $3
  - add Rd, Rs, Rt   # d=dest, s=source, t=??
  - Op = 6-bit code for "add" = 000000
  - Rs = 00010 = 2 ($2)
  - Rt = 00011 = 3 ($3)
  - Rd = 00001 = 1 ($1)
  - don't worry about Sh and Func fields for now

Note: the MIPS architecture has 32 registers. Therefore, it takes $\log_2 32 = 5$ bits to specify any one of them.

opcode	Rs	Rt	Rd	Sh and Func
000000	00010	00011	00001	00000100000

Uh-Oh

| opcode (6 bits)	operands (26 bits)

- Let's try a lw (load word) instruction
- lw $1, Memory[1004]
  - 6 bits for opcode
  - That leaves 26 bits for address in memory
- But an address is 32 bits long!
  - What gives?

Memory Operand Addressing (for loads/stores)

- We have to use indirection to specify memory operands
- **Addressing mode**: way of specifying address
  - **(Register) Indirect:** `lw $1,($2)` # $1=memory[$2]
    - $2 specifies address = pointer!
  - **Displacement:** `lw $1,8($2)`   # $1=memory[$2+8]
  - **Index-base:** `lw $1,($2,$3)`   # $1=memory[$2+$3]
  - **Memory-indirect:** `lw $1,@($2)` # $1=memory[memory[$2]]
  - **Auto-increment:** `lw $1,($2)+` # $1=memory[$2++]

  - ICQ: What HLL program idioms are these used for?

MIPS Addressing Modes

- MIPS implements only displacement addressing mode
  - 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

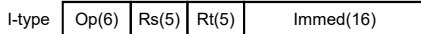| I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)
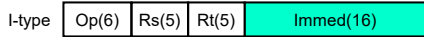
Back to the Simple, Running Example

```
# assume $16=1004=address of variable x in C code example
# and recall that 1008=address of variable y in C code example

loop:   lw $8, Memory[1004] → lw $8, 0($16)   # put val of x in $8
        lw $9, Memory[1008] → lw $9, 4($16)   # put val of y in $9
        add $10, $8, $9
        add $11, $11, $10
        j loop
```

MIPS Format – I-Type Example

| I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)

- lw $1, 0($16) // $1 = Memory [$16 + 0]
  - lw Rt, immed(Rs)
  - Opcode = 6-bit code for "load word" = 100011
  - Rs = 16 = 10000
  - Rt = 1 = 00001
  - Immed = 0000 0000 0000 0000 = 0_{10}

opcode	Rs	Rt	immed
100011	10000	00001	0000000000000000

Memory Addressing Issue: Endian-ness

Byte Order

- **Big Endian:** byte 0 is 8 **most** significant bits: MIPS, IBM 360/370, Motorola 68k, SPARC, HP PA-RISC
- **Little Endian:** byte 0 is 8 **least** significant bits Intel 80x86, DEC Vax, DEC/Compaq Alpha
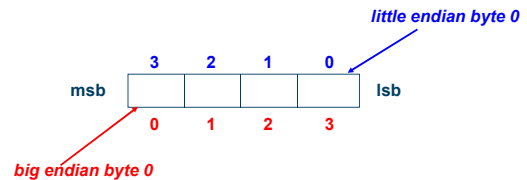
Memory Addressing Issue: Alignment

- **Alignment:** require that objects fall on address that is multiple of their size
- 32-bit integer (int in C)
  - Aligned if address % 4 = 0 [% is symbol for "mod"]
  - Aligned: lw @xxxx00
  - Not: lw @XXXX10
- 64-bit integer (long int in C)?
  - Aligned if ?
- Question: what to do with unaligned accesses (uncommon case)?
  - Support in hardware? Makes all accesses slower
  - Trap to software routine? Possibility
  - **MIPS? ISA support**: unaligned access using two instructions:
    lw @XXXX10 = lwl @XXXX10; lwr @XXXX10
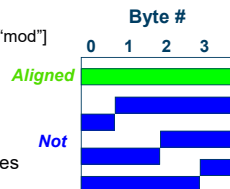
Declaring Space in Memory for Data

- Add two numbers x and y:

```
    .text          # declare text segment
main:              # label for main
  la  $8, x        # la = "load address" of x into $8
  lw  $9, 0($8)    # load value of x into $9
  la  $8, y        # load address of y into $8
  lw  $10, 0($8)   # load value of y into $10
  add $11,$9,$10   # compute x+y, put result in $6
  ...


    .data          # declare data segment
x:.word 10         # initialize x to 10
y:.word 3          # initialize y to 3
```

MIPS Operand Model

- MIPS is a "load-store" architecture
  - All computations done on values in registers
    - Can only access memory with load/store instructions
  - 32 32-bit integer registers
    - Actually 31: $0 is hardwired to value 0 → ICQ: why?
    - Also, certain registers conventionally used for special purposes
      - We'll talk more about these conventions later
  - 32 32-bit FP registers
    - Can also be treated as 16 64-bit FP registers
  - HI,LO: destination registers for multiply/divide

How Many Registers? Why 32 for MIPS?

- Registers faster than memory → have 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 using registers
  - More registers means **more saving/restoring** them
    - At procedure calls and context switches
  - Upshot: trend to more registers: 8(IA-32)→32(MIPS) →128(IA-64)

Control Instructions – Changing the PC

- Most instructions set Next PC = PC+ "1"
  - I put the "1" in quotes, because depends on size of instr
- But what about handling control flow?
- Conditional control flow: if condition is satisfied, then change control flow
  - if/then/else
  - while() loops
  - for() loops
  - switch
- Unconditional control flow: always change control flow
  - procedure calls

- How do we implement control flow in assembly?

Control Instructions

- Three issues:
  1. Testing for condition: Does Next PC != PC + "1"?
  2. Computing target: If Next PC != PC+"1", then what is Next PC?
  3. Dealing with procedure calls (later)

- Types of control instructions
  - conditional branch: beq, beqz, bgt, etc.
    - if condition met, "branch" to new PC; else Next PC=PC+"1"
    - many flavors of branch based on condition (<, >0, <=, etc.)
  - unconditional jump: j, jr, jal, jalr
    - change Next PC to some new address
    - several flavors of jump based on how new address is specified

Quick Aside on Instruction Names

- Most assembly instructions look like gibberish, but there's usually a pattern
- For example, branch instructions:
  - beq = Branch EQual
  - bne = Branch Not Equal
  - beqz = Branch EQual Zero
  - blt = Bacon Lettuce Tomato or Branch Less Than
  - bltz = Branch Less Than Zero
  - bgt = Branch Greater Than
  - bn = Branch Negative
- What do you notice? Mix and match letter combos
  - EQ = equal, Z=zero, N=not or negative, LT=less than, etc.

Control Instructions I: Condition Testing

- Three options for **testing conditions**
  - Option I: **compare and branch instructions** (not used by MIPS)
    ```
    blti $1,10,target // if $1<10, goto target
    # blti = "branch-less-than immediate"
    ```
  - Option II: **implicit condition codes (CCs)**
    ```
    subi $2,$1,10   // sets "negative" CC
    bn target  // if negative CC set, goto target
    # bn = "branch if negative"
    ```
  - Option III: **condition registers, separate branch insns**
    ```
    slti $2,$1,10  // set $2 if $1<10
    # slti = "set less-than immediate"
    bnez $2,target  // if $2 != 0, goto target
    # bnez = "branch not-equal to zero"
    ```

MIPS Conditional Branches

- MIPS uses combination of options II (with twist) and III
  - Compare 2 registers and branch: `beq`, `bne`
    - Equality and inequality only
    - + Don't need adder for comparison
  - Compare 1 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

© Daniel J. Sorin
from Hilton, Lebeck, Roth

Control Instructions II: Computing Target

- Three options for **computing targets (target = next PC)**
  - Option I: **PC-relative** (next PC = current PC +/- some value)
    - Position-independent within procedure
    - Used for branches and jumps within procedure
  - Option II: **Absolute** (next PC = some value)
    - Position independent outside procedure
    - Used for procedure calls
  - Option III: **Indirect** (next PC = contents of a 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 procedure (they don't get very big)
  - Further from one procedure to another

© Daniel J. Sorin
from Hilton, Lebeck, Roth

MIPS: Computing Targets
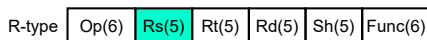
- MIPS uses all 3 ways to specify target of control insn
  - PC-relative → conditional branches: `bne`, `beq`, `blez`, etc.
    - 16-bit relative offset, <0.1% branches need more
    - PC = **PC + 4 + immediate** if condition is true (else PC=PC+4)

I-type	Op(6)	Rs(5)	Rt(5)	Immed(16)
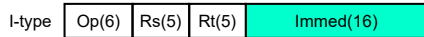
  - Absolute → unconditional jumps: `j target`
    - 26-bit offset (can address 2^{28} words < 2^{32} → what gives?)

J-type	Op(6)	Target(26)

  - Indirect → Indirect jumps: `jr $31`

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)

© Daniel J. Sorin
from Hilton, Lebeck, Roth

Control Idiom: If-Then-Else

- First control idiom: **if-then-else**

```
if (A < B) A++;      // assume A in register $1
else B++;            // assume B in $2
```

```
        slt   $3,$1,$2      // if $1<$2, then $3=1
        beqz  $3,else       // branch to else if !condition
        addi  $1,$1,1       // A=A+1
        j     join          // jump to join
  else: addi  $2,$2,1       // B=B+1
  join:
```

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

© Daniel J. Sorin
from Hilton, Lebeck, Roth

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 $1, N in $2
   sum += A[i];             // &A[i] in $3, sum in $4
}
```

```
        sub  $1,$1,$1     # initialize i to 0
  loop: slt  $8,$1,$2     # if i<N, then $8=1; else $8=0
        beqz $8,exit      # test for exit at loop header
        lw   $9,0($3)     # $9 = A[i]  (not &A[i])
        add  $4,$4,$9     # sum = sum + A[i]
        addi $3,$3,4      # increment &A[i] by sizeof(int)
        addi $1,$1,1      # i++
        j loop            # backward jump
  exit:
```

© Daniel J. Sorin
from Hilton, Lebeck, Roth

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 $1, head in $2
   sum += p->val;                 // sum in $3
```

```
        addi $1,$2,0      # p = head
  loop: beq  $1,$0,exit   # if p==0 (NULL), goto exit
        lw   $5,0($1)     # $5 = *p = p→val
        add  $3,$3,$5     # sum = sum + p→val
        lw   $1,4($1)     # p = *(p+1) = p→next
        j loop            # go back to top of loop
  exit:
```

© Daniel J. Sorin
from Hilton, Lebeck, Roth

Some of the Most Important Instructions

- Math/logic
  - add, sub, mul, div

<div style="border:1px solid blue">Note: sw is unusual in that the destination of instruction isn't first operand!</div>

- Access memory
  - lw = load (read) word: lw $3, 4($5)   # $3 = memory[$5+4]
  - sw = store (write) word: sw $3, 4($5)   # memory[$5+4] = $3
- Change PC, perhaps conditionally
  - Branches: blt, bgt, beqz, etc.
  - Jumps: j, jr, jal (will see last two later)
- Handy miscellaneous instructions
  - la = load address
  - move: move $1, $5 # copies (doesn't move!) $5 into $1
  - li = load immediate: li $5, 42   # writes value 42 into $5
    - terrible name for instr!! not a load – no memory access!

Many Other Operations

- Many types of operations
  - Integer arithmetic: add, sub, mul, div, mod/rem (signed/unsigned)
  - FP arithmetic: add, sub, mul, div, sqrt
  - Integer logical: and, or, xor, not, sll, srl, sra
  - Packed integer: padd, pmul, pand, por… (saturating/wraparound)
- What other operations might be useful?
- 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 …

Flavors of Math Instructions

- We already know about add
  - `add $3, $4, $5`
- Also have addi = "add immediate" [Note: I-type instr]
  - `addi $3, $4, 42  #  $3 = $4 + 42`
- And addu = "add unsigned"
  - `addu $3, $4, $5 # same as add, but treat vals as unsigned ints`
- And even addiu = "add immediate unsigned"
  - `addiu $3, $4, 42`

- Same variants for sub, etc.

Flavors of Load/Store Instructions

- We already know about lw and sw
  - `lw $3, 12($5)`
  - `sw $4, −5($6)`
- Also have load/store instructions that operate at non-word-size granularity
  - lb = load byte, lh = load halfword
  - sb = store byte, sh = store halfword
- Loads can access smaller size but always write all 32 bits of destination register
  - By default, sign-extend to fill register
  - Unless specified as unsigned with instrs: lbu, lhu

Datatypes

- Datatypes
  - Software view: property of data
  - Hardware view: data is just bits, property of operations
    - Same 32 bits could be interpreted as int or as instruction, etc.
- Hardware datatypes
  - Bit strings: 8 bits (byte), 16b (half), 32b (word), 64b (double word)
  - Integers: 32b (int), 64b (long int)
  - IEEE754 FP: 32b (single-precision), 64b (double-precision)
  - Packed integer: treat 64b int as 8 8b int's or 4 16b int's
  - Packed FP

Procedure Calls: A Simple, Running Example

```
main:   li $1, 1              # $1 = 1
        li $2, 2              # $2 = 2
        $3 = call foo($1, $2)   # this is NOT actual MIPS code
        add $4, $3, $3
        {rest of main}
        {end program}

foo:    add $5, $1, $2
        return ($5)           # this is also NOT actual MIPS code
```

----
main is the caller
foo is the callee

Procedure Calls: Jump-and-Link and Return

```
main:   li $1, 1
        li $2, 2
        $3 = call foo($1, $2) → jal foo      # jal = jump and link
        add $4, $3, $3
        {rest of main}

foo:    sub $5, $1, $2
        return ($5) → jr $ra
```

----
jal does two things:
        1) sets PC = foo (just like a regular jump instruction)
        2) "links" to PC after the jal → saves that PC in register $31
MIPS designates $31 for a special purpose: it's the return address ($ra)
        $ra is handy mnemonic for $31 (assembler understands both)
jr sets PC to value in $ra → computer executes add instr after jal

Procedure Calls: What if We Didn't Link?

```
main:   li $1, 1
        li $2, 2
        $3 = call foo($1, $2) → j foo        # j = jump (instead of jal)
r1:     add $4, $3, $3
        add $1, $1, $4
        j foo
r2:     sub $2, $1, $3
        {rest of main}

foo:    sub $5, $1, $2
        return ($5) → OK, now what??  Jump to r1?  Jump to r2?
```

Since function can be called from multiple places, must explicitly remember (link!) where called from.

Procedure Calls: Passing Args & Return Values

```
main:   li $1, 1
        li $2, 2
        move $a0, $1    # pass first arg in $a0
        move $a1, $2    # pass second arg in $a1
        jal foo
        add $4, $3, $3 → add $4, $v0, $v0   # return value in $v0 now
        {rest of main}

foo:    sub $5, $a0, $a1
        move $v0, $5    # pass return value in $v0
        jr $ra
```

Must use specific registers for passing arguments and return values.
MIPS denotes $a0-$a3 (mnemonics for $4-$7) as argument registers.
MIPS denotes $v0-$v1 ($2-$3) as return value registers.

Passing Arguments by Value or by Reference

- Passing arguments
  - **By value**: pass contents [$3+4] in $a0
    ```
    int n;                    // n in 4($3)
    foo(n);
            lw $a0,4($3) # reading from Mem[$3 + 4]
            jal foo
    ```

  - **By reference**: pass address $3+4 in $a0
    ```
    int n;                    // n in 4($3)
    bar(&n);
            addi $a0,$3,4  # computing $3 + 4
            jal bar
    ```

Procedures Must Play Nicely Together

```
main:   li $1, 1
        li $2, 2
        move $a0, $1
        move $a1, $2
        jal foo
        add $4, $v0, $v0
        add $6, $4, $1  # $1 should still be 1
        {rest of main}

foo:    sub $5, $a0, $a1
        li $1, 3          # $1 now equals 3
        add $5, $5, $1
        move $v0, $5
        jr $ra
```

> What would happen if main uses $1 after calling foo but foo also uses $1?
>
> Not good, right? Let's see why …

Brief Detour to HLL Programming

```
int main (){
        int x=1;
        int y=2;
        int z = foo(x,y);
        z = z + x;
}

int foo(int a1, int a2){
        // code written by other person
        return a1+a2;
}
```

> Programmer of main() assumes that x will still equal 1 after call to foo(). But that won't happen if foo() messes with registers that x was using.

Procedures Must Play Nicely Together

```
main:   li $1, 1
        li $2, 2
        move $a0, $1
        move $a1, $2
        jal foo
        add $4, $v0, $v0
        add $6, $4, $1   # $1 should still be 1
        {rest of main}

foo:    sub $5, $a0, $a1
        li $1, 3          # $1 now equals 3
        add $5, $5, $1
        move $v0, $5
        jr $ra
```

This seems contrived. Why can't the programmer of foo just not use $1? Problem solved, right?

Nope! In real-world, one person doesn't write all of the software. My code must play well with your code.

55

Procedures Use the Stack

- In general, procedure calls obey **stack discipline**
  - Local procedure state contained in **stack frame**
  - Where we can save registers to avoid problem in last slide
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses
- Procedure stack is **in memory**
  - Starts at "top" of memory and grows down
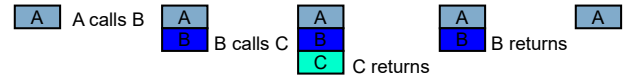
A   A calls B   A / B   B calls C   A / B / C   C returns   A / B   B returns   A

56

Preserving Registers Across Procedures

```
main:   li $1, 1
        li $2, 2
        move $a0, $1
        move $a1, $2
        jal foo
        add $4, $v0, $v0
        add $6, $4, $1
        {rest of main}

foo:    sub $5, $a0, $a1
        li $1, 3
        add $5, $5, $1
        move $v0, $5
        jr $ra
```

stack pointer (during main)

memory: main's frame

Stack pointer is address of bottom of current stack frame. Always held in register $sp.

57

Preserving Registers Across Procedures

```
main:   li $1, 1
        li $2, 2
        move $a0, $1
        move $a1, $2
        jal foo
        add $4, $v0, $v0
        add $6, $4, $1
        {rest of main}

foo:    make frame (move stack ptr)
        save $1 in stack frame
        sub $5, $a0, $a1
        li $1, 3
        add $5, $5, $1
        move $v0, $5
        restore $1 from stack frame
        destroy frame
        jr $ra
```

stack pointer (during foo)

memory: main's frame / foo's frame

58

Preserving Registers Across Procedures

```
main:   li $1, 1
        li $2, 2
        move $a0, $1
        move $a1, $2
        jal foo
        add $4, $v0, $v0
        add $6, $4, $1
        {rest of main}

foo:    make frame → subi $sp, $sp, 4
        save $1 on stack frame → sw $1, 0($sp)
        sub $5, $a0, $a1
        li $1, 3
        add $5, $5, $1
        move $v0, $5
        restore $1 from stack frame → lw $1, 0($sp)
        destroy frame → addi $sp, $sp, 4
        jr $ra
```

$sp (during foo)

memory: main's frame / foo's frame

59

Who Saves/Restores Registers?

```
main:   li $1, 1
        li $2, 2
        move $a0, $1
        move $a1, $2
        jal foo
        add $4, $v0, $v0
        add $6, $4, $1
        {rest of main}

foo:    subi $sp, $sp, 4
        sw $1, 0($sp)
        sub $5, $a0, $a1
        li $1, 3
        add $5, $5, $1
        move $v0, $5
        lw $1, 0($sp)
        addi $sp, $sp, 4
        jr $ra
```

$sp (during foo)

memory: main's frame / foo's frame

In this example, the callee (foo) saved/restored registers. But why didn't the caller (main) do that instead?
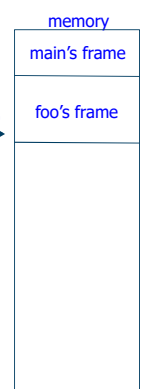
60

MIPS Register Usage/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

© Daniel J. Sorin
from Hilton, Lebeck, Roth

MIPS Register Usage/Naming Conventions

0	zero	constant
1	at	reserved for assembler

16	s0	callee saves
. . .		

Important: The only general purpose registers are the $s and $t registers. (Note: prior slides didn't always do this!)

Everything else has a specific usage:
$a = arguments
$v = return values
$ra = return address
etc.

15	t7	

31	ra	return address

© Daniel J. Sorin
from Hilton, Lebeck, Roth

MIPS/GCC Procedure Calling Conventions

Calling Procedure ("Caller"):

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

- Step-2: Saves caller-saved registers on stack
  - First make room on stack if necessary
  - Then save registers $t0-$t9 if they contain live values at call site

- Step-3: Executes a jal instruction

And now we're at the Callee procedure …

© Daniel J. Sorin
from Hilton, Lebeck, Roth

MIPS/GCC Procedure Calling Conventions (cont.)

Called Routine ("Callee")

- Step-1: Establishes stack frame for callee
  - Subtract the frame size from the stack pointer
    subiu $sp, $sp, <frame-size>
- Step-2: Saves callee-saved registers in the frame
  - Register $ra is saved if callee will make a call
  - Registers $s0-$s7 are saved if they are used

  Now the callee does its work until it's ready to return …

© Daniel J. Sorin
from Hilton, Lebeck, Roth

MIPS/GCC Procedure Calling Conventions (cont.)

Callee Does Following to Return to Caller:

- Step-1: Puts returned values in registers $v0 and $v1
  (if values are returned)
- Step-2: Restores callee-saved registers
  - $ra, $s0 - $s7
- Step-3: Pops the stack
  - Add the frame size to $sp
    addiu $sp, $sp, <frame-size>
- Step-4: Returns to caller
  - Jump to address in $ra
    jr $ra

© Daniel J. Sorin
from Hilton, Lebeck, Roth

Generic Template for Any Procedure Foo

- Set up foo's stack frame by moving $sp down
- Save callee-saved registers that will be written by foo (including $ra)
- Do some work
  - While doing work, if foo makes a call to some procedure bar:
    - Save caller-saved registers
    - Pass argument(s) in $a registers
    - Do jal to procedure bar
    - Restore caller-saved registers
    - Use value returned by procedure bar in $v0
- Put foo's return value, if any, in $v0
- Restore callee-saved registers that were previously saved
- Destroy stack frame by moving $sp up
- Return to whoever called foo with jr $ra

Note: If foo is recursive, then foo is making a call to foo. Does not change anything in this template, though! Just change bar to foo.

© Daniel J. Sorin
from Hilton, Lebeck, Roth

System Call Instruction

- System call is used to communicate with the operating system and request services (memory allocation, I/O)
  - syscall instruction in MIPS
- Sort of like a procedure call, but call to ask OS for help
- 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

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		

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 …
```

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
nln:   .asciiz "\n"
```

Factorial (skimming base case of recursion!)

```
fact: addi $sp,$sp,-8    // open frame (2 words)
      sw $ra,4($sp)      // save return address
      sw $s0,0($sp)      // save $s0

      # handle base case (not real code here)
      # if $a0=1, set $v0=1 and jump to clean

      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
      …
clean:lw $s0,0($sp)      // restore $s0
      lw $ra,4($sp)      // restore $ra
      addi $sp,$sp,8     // collapse frame
      jr $ra             // return, value in $v0
```

Outline

- What is an ISA?
- Assembly programming (in the MIPS ISA)
- Other ISAs

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 and programs evolve?
  - Easy to maintain implementability as technology evolves?

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?

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

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
    - New processors only re-define opcodes that trapped in old ones
    - Old processors emulate new instructions in low-level software

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

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

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

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

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
- 2013: Haswell – added transactional mem features

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!

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

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

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

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

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!

Learn exactly how this all works in ECE 552 / CS 550

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 inst
  - 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

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 the common case fast

- Compromises inevitable → there is no perfect ISA

Outline

- What is an ISA?
- Assembly programming (in the MIPS ISA)
- Other ISAs