How Many Registers?

- 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 putting more things in
  - 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)
MIPS Operand Model

• MIPS is load-store
  • 32 32-bit integer registers
    • Actually 31: r0 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

• Integer register conventions
  • Allows separate function-level compilation and fast function calls
    • Note: “function”, “method”, and “procedure” are equivalent terms in this course
  • We’ll discuss this more when we get to procedure calls
Memory Operand Addressing

• ISAs assume “virtual” address size
  • Either 32-bit or 64-bit
  • Program can name $2^{32}$ bytes (4GB) or $2^{64}$ bytes (16PB)
  • ISA impact? no room for even one address in a 32-bit instruction

• **Addressing mode:** way of specifying address
  • (Register) Indirect: \texttt{ld R1, (R2)} \hspace{1cm} R1=\text{mem}[R2]
  • Displacement: \texttt{ld R1, 8 (R2)} \hspace{1cm} R1=\text{mem}[R2+8]
  • Index-base: \texttt{ld R1, (R2,R3)} \hspace{1cm} R1=\text{mem}[R2+R3]
  • Memory-indirect: \texttt{ld R1, @(R2)} \hspace{1cm} R1=\text{mem}[\text{mem}[R2]]
  • Auto-increment: \texttt{ld R1, (R2)+} \hspace{1cm} R1=\text{mem}[R2++]
  • Scaled: \texttt{ld R1, (R2,R3,32,8)} \hspace{1cm} R1=\text{mem}[R2+R3*32+8]

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

© 2009 Daniel J. Sorin
from Roth and Lebeck
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
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
Another 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?
  - Aligned if ?

- Question: what to do with unaligned accesses (uncommon case)?
  - Support in hardware? Makes all accesses slow
  - Trap to software routine? Possibility

- **MIPS ISA support**: unaligned access using two instructions:
  \( lw \ @xxxx10 = lwl \ @xxxx10; lwr \ @xxxx10 \)
(5) Datatypes

• Datatypes
  • Software view: property of data
  • Hardware view: data is just bits, property of operations

• 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
MIPS Datatypes (and Operations)

• Datatypes: all the basic ones (byte, half, word, FP)
  • All integer operations read/write 32-bits
    • No partial dependences on registers
  • Only byte/half variants are load-store
    \texttt{lb, lbu, lh, lhu, sb, sh}
  • Loads sign-extend (or not) byte/half into 32-bits

• Operations: all the basic ones
  • Signed/unsigned variants for integer arithmetic
  • Immediate variants for all instructions
    \texttt{add, addu, addi, addiu}

• Regularity/orthogonality: all variants available for all operations
  • Makes compiler’s “life” easier
(6) Control Instructions

• Three issues:
  1. Testing for condition: Does PC \(!=\) PC++?
  2. Computing target: If PC \(!=\) PC++, then what is it?
  3. Dealing with procedure calls
(6) 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
    + Simple, – two ALUs: one for condition, one for target address
  • **Option II**: 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
  • **Option III**: 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
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 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
    - Make the common case fast (MCCF)!
Control Instructions II: Computing Target

• Three options for computing targets
  • Option I: PC-relative
    • Position-independent within procedure
    • Used for branches and jumps within a procedure
  • Option II: Absolute
    • Position independent outside procedure
    • 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 (they don’t get very big)
  • Further from one procedure to another
MIPS Control Instructions

- MIPS uses all three
  - PC-relative ➞ conditional branches: `bne`, `beq`, `blez`, etc.
    - 16-bit relative offset, <0.1% branches need more
    - \( \text{PC} = \text{PC} + 4 + \text{immediate if condition is true} \) (else \( \text{PC} = \text{PC} + 4 \))
  - Absolute ➞ unconditional jumps: `j target`
    - 26-bit offset (can address \( 2^{28} \) words < \( 2^{32} \) ➞ what gives?)
  - Indirect ➞ Indirect jumps: `jr $rs`

```
I-type
<table>
<thead>
<tr>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Immed(16)</th>
</tr>
</thead>
</table>

J-type
<table>
<thead>
<tr>
<th>Op(6)</th>
<th>Target(26)</th>
</tr>
</thead>
</table>

R-type
<table>
<thead>
<tr>
<th>Op(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Rd(5)</th>
<th>Sh(5)</th>
<th>Func(6)</th>
</tr>
</thead>
</table>
Control Instructions III: Procedure Calls

• Another issue: support for procedure calls?
  • We “link” (remember) address of the calling instruction + 4 (current PC + 4) so we can return to it after procedure

• MIPS
  • Implicit return address register is $ra (= $31)
  • Direct jump-and-link: jal address
    \[ \rightarrow \] $ra = PC + 4; PC = address
  • Can then return from call with: jr $ra

  • Or can call with indirect jump-and-link: jalr $rd, $rs
    \[ \rightarrow \] $rd = PC + 4; PC = $rs \quad \text{// explicit return address register}
  • Then return with: jr $rd

• We’ll see how procedure calls work in a few slides …
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

```assembly
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:
```

Note: assembler converts “else” target of beqz into immediate → what is the immediate?
Control Idiom: Arithmetic For Loop

- Second idiom: "for loop" with arithmetic induction

```c
int A[100], sum, i, N;
for (i=0; i<N; i++)
{
    sum += A[i];       // assume: i in $s1, N in $s2
    // &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:
```

© 2009 Daniel J. Sorin
from Roth and Lebeck
Control Idiom: Pointer For Loop

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

```c
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
```

```assembly
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:
```

© 2009 Daniel J. Sorin  
from Roth and Lebeck