(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

© 2008 Daniel J. Sorin
from Roth and Lebeck

(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

© 2008 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 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)!

© 2008 Daniel J. Sorin
from Roth and Lebeck

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

© 2008 Daniel J. Sorin
from Roth and Lebeck

MIPS Control Instructions

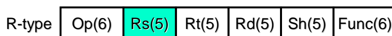
- MIPS uses all three
  - PC-relative → conditional branches: **bne**, **beq**, **blez**, etc.
    - 16-bit relative offset, <0.1% branches need more
    - PC = PC + 4 + immediate if condition is true (else PC=PC+4)

I-type

  - Absolute → unconditional jumps: `j target`
    - 26-bit offset (can address 2^{28} words < 2^{32} → what gives?)

J-type

  - Indirect → Indirect jumps: `jr $rs`

R-type 

© 2008 Daniel J. Sorin
from Roth and Lebeck

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`
    → $ra = PC+4; PC = address
  - Can then return from call with: `jr $ra`
  - Or can call with indirect jump-and-link: `jalr $rd, $rs`
    → $rd = PC+4; PC = $rs // explicit return address register
  - Then return with: `jr $rd`
- We'll see how procedure calls work in a few slides …

© 2008 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++;      // A in register $s1
else B++;            // 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?

© 2008 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:
```

© 2008 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: Pointer For Loop

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

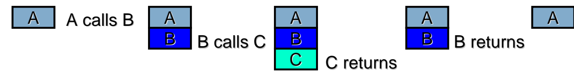
```
struct node_t { int val; struct node_t *next; };
struct node_t *p, *head;
int sum;
for (p=head; p!=NULL; p=p->next)   // p in $s1, head in $s2
   sum += p->val                   // sum in $s3

        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:
```

© 2008 Daniel J. Sorin
from Roth and Lebeck

Control Idiom: Procedure Call

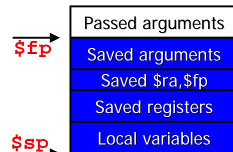
- In general, procedure calls obey **stack discipline**
  - Local procedure state contained in **stack frame**
  - When a procedure is called, a new frame opens
  - When a procedure returns, the frame collapses
- Procedure stack is **in memory**
  - Distinct from operand stack which is not addressable
- 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



© 2008 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 (popular 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
      - Why? For dynamically variable sized frames
- Frame layout
  - Contents accessed using $sp
    `sw $ra,24($sp)`
  - Displacement addressing



Passed arguments
Saved arguments
Saved $ra,$fp
Saved registers
Local variables

© 2008 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 all 32 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)

© 2008 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

© 2008 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)

© 2008 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
```

© 2008 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   // div puts result in $lo

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

© 2008 Daniel J. Sorin
from Roth and Lebeck