(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
    ```
  - Option II: implicit condition codes (CCs)
    
    ```
    subi $2,$1,10   // sets “negative” CC
    bn target  // if negative CC set, goto target
    ```
  - Option III: condition registers, separate branch insns
    
    ```
    slti $2,$1,10  // set $2 if $1<10
    bnez $2,target  // if $2 != 0, goto target
    ```

- 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)!

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`
    - Greaterless than comparisons
    - Don’t need adder for comparison
  - Set explicit condition registers: `slt, sltu, slti, sltiu, etc.`

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 types of control instructions:
  - PC-relative conditional branches: `bne, beq, blez`, etc.
  - 16-bit relative offset, <0.1% branches need more PC-relative
  - PC = PC + 4 + immediate if condition is true (else PC=PC+4)
  - Absolute unconditional jumps: `j target`
  - 26-bit offset (can address $2^{28}$ words $< 2^{32}$)
  - Indirect conditional jumps: `jr $rs`

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-reg: `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 …

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 $s1, $s1, $s2` // if $s1<$s2, then $s3=1
    - `beqz $s3, else` // branch to else if !condition
    - `addi $s1, $s1, 1` // i++
    - `j join` // jump to join
    - `else: addi $s2, $s2, 1` // icq: assembler converts "else" target of beqz into immediate → what is the immediate?

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]; // sum += A[i];
    - `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
    - `jo join` // jump to join
    - `else: addi $s2, $s2, 1` // icq: assembler converts "else" target of beqz into immediate → what is the immediate?
  - `exit:` // return from loop
Control Idiom: Pointer For Loop

• Third idiom: for loop with pointer induction
  struct node_t { int val; struct node_t *next; };  
  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
      add $s3,$s3,$t1  // sum = sum + p->val
      lw $s1,4($s1)  // p = *(p+1) = p->next
    j loop
  exit:

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 ($sp$): points to bottom of current frame
      • Why? For dynamically variable sized frames
  • Frame layout
    • Contents accessed using $sp$
      sw $ra,24($sp)  // Passed arguments
      Saved arguments
      Saved $ra,$fp
      Saved registers
      Local variables

• 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

MIPS Procedure Call: Factorial (Naïve version)

  fact:  addi $sp,$sp,-128  // open frame (32 words of storage)
         sw $ra,124($sp)
         sw $1,120($sp)
         sw $2,116($sp)
    --
    lw $s0,128($sp)
    subi $s1,$s0,1
    sw $s1,0($sp)
    jal fact  // store (argument-1) to frame
    lw $s1,-4($sp)
    mul $s1,$s1,$s0  // recursive call
    --
    lw $2,116($sp)
    lw $1,120($sp)
    lw $ra,124($sp)
    sw $s1,124($sp)
    addi $sp,$sp,128  // collapse frame
    jr $ra  // return
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 values
  - $4-7($a0-$a3): function arguments
  - $8-$15, $24, $25($t0-$t9): caller saved temporaries
- A saves before calling B only if needed after B returns
- $16-$23($s0-$s7): callee saved
- A needs after B returns, B saves if it uses also
- We’ll discuss complete set of MIPS registers and conventions soon

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 $v0,$s0,0,0        // value returned via $v0
...  
lw $s0,0($sp)           // restore $s0
lw $t2a,4($sp)          // restore $t2a
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 ($a0, $ra) rather than 31 (excl. $0)

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(4n);
    add $a0,$sp,4
    jal bar

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` // put result in $s0
  - `mflo $s1`         // move it from $s0 to $s1`