Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar, Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood
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; 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, 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:
```

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

```
A calls B
```
```
B calls C
```
```
C returns
```
```
B returns
```
```
A
```
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
```
MIPS Procedure Call: Factorial (Naïve version)

```mips
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)
      addi $sp,$sp,128 // return value below caller’s frame
      jr $ra // return
```

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