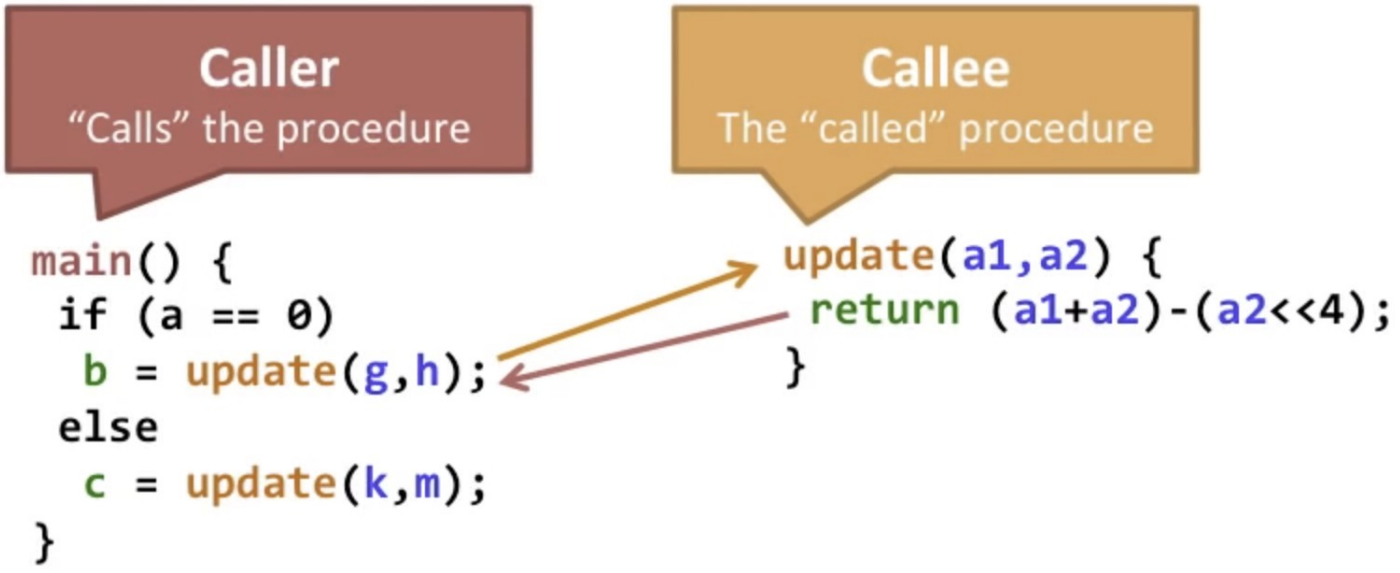
CENG311 Computer Architecture

Instructions: Language of the Computer

IZTECH, Fall 2023

26 October 2023

Procedure Call



Procedure Call Instructions

Caller procedure call: jump and link

```
jal ProcedureLabel
```

Address of return address (PC+4) put in $ra

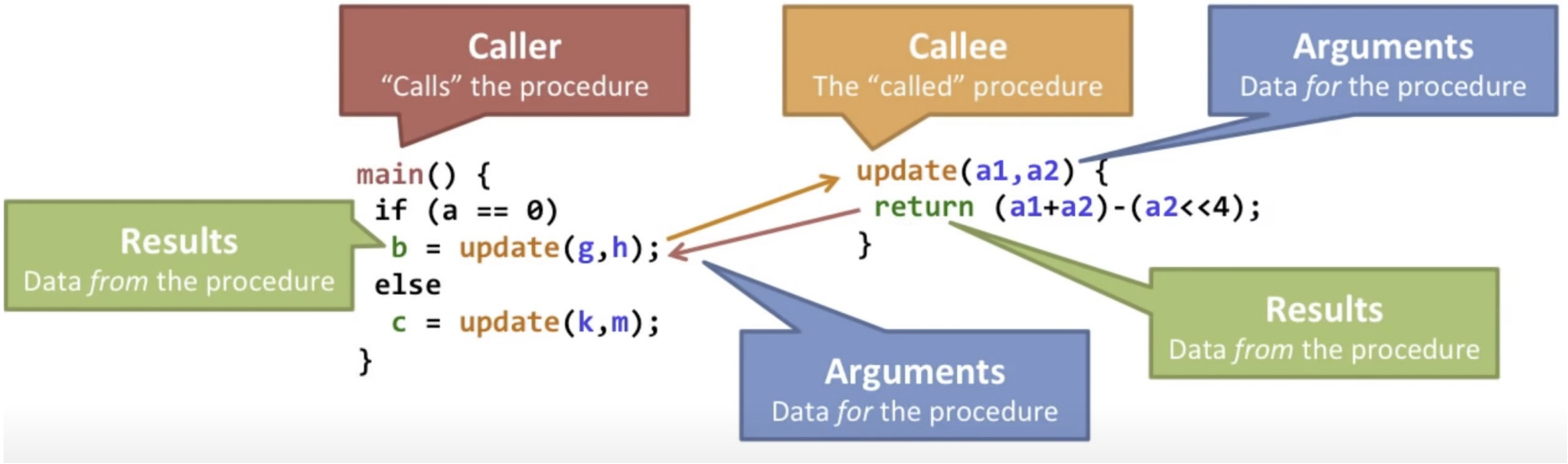
Jumps to target address

Callee procedure return: jump register

```
jr $ra
```

Copies $ra to program counter

Procedure Call



Registers

$a0 – $a3: arguments (reg's 4 – 7)

To pass parameters

$v0, $v1: result values (reg's 2 and 3)

To store return values

$ra: return address (reg 31)

To return to the point of origin

$sp: stack pointer (reg 29)

To point the call stack

Stack

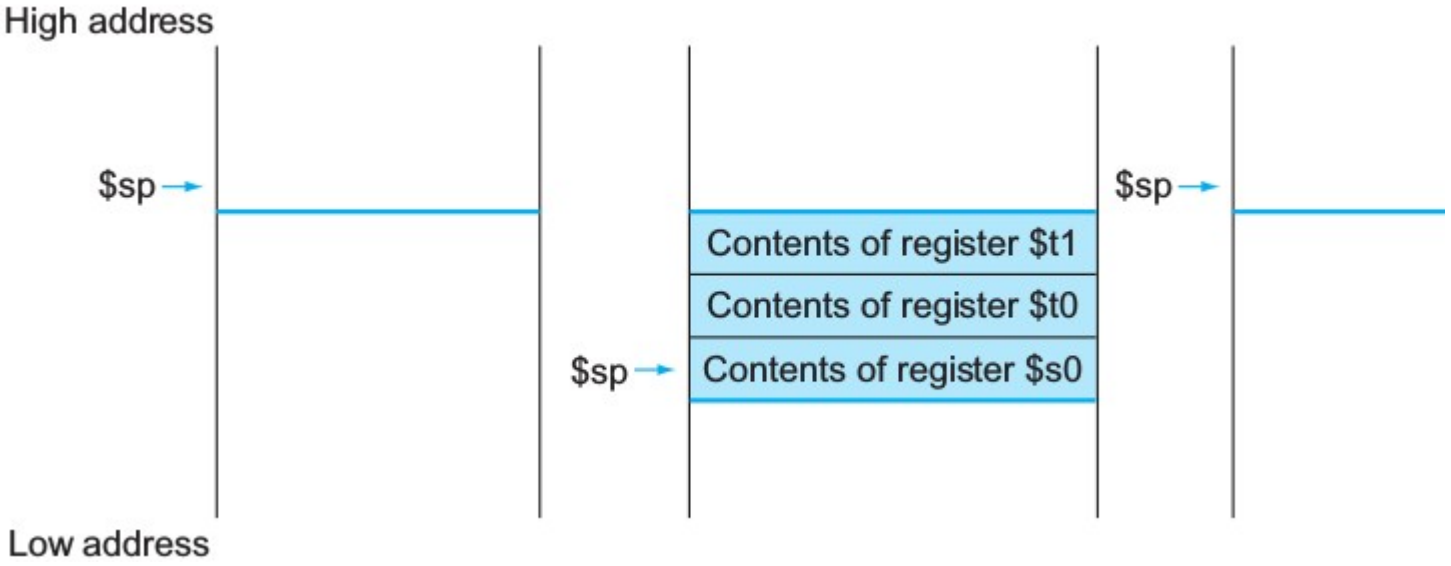
Any registers needed by the caller must be restored to the values that they contained before the procedure was invoked

The ideal data structure for spilling the registers is a stack

$sp: stack pointer (reg 29)

A value denoting the most recently allocated address in a stack that shows where registers should be spilled or where old register values can be found

Stack Pointer for a Procedure Call



Procedure Call

The caller

- Puts the parameter values in $a0-$a3
- Uses `jal X` to jump to procedure X (callee), `jal` stores PC+4 in $ra

The callee

- Performs the calculations
- Places the results in $v0 and $v1
- Returns the control to the caller using `jr $ra`

PC (program counter): the register containing the address of the instruction in the program being executed

Procedure Example

C code:

```
int leaf_example (int g, int h, int i, int j){
        int f;
        f = (g + h) - (i + j);
        return f;
}
```

Arguments g, …, j in $a0, …, $a3

f in $s0

Result in $v0

Procedure Example

```
;put argument values in $a0, $a1, $a2, $a3
jal leaf_example
;get return value from $v0
leaf_example:
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
jr   $ra
```

Procedure Example

```
...
jal leaf_example
...
leaf_example:
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
jr   $ra
```

Registers Across a Procedure Call

Preserved, guaranteeing that the caller will get the same data back

Preserved	Not preserved
Saved registers: `$s0–$s7`	Temporary registers: `$t0–$t9`
Stack pointer register: `$sp`	Argument registers: `$a0–$a3`
Return address register: `$ra`	Return value registers: `$v0–$v1`
Stack above the stack pointer	Stack below the stack pointer

Leaf Procedure Example

```
leaf_example:
  addi $sp, $sp, -4
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  addi $sp, $sp, 4
  jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Leaf Procedure Example

```
leaf_example:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

Nested Procedures

Procedures that call other procedures

For nested call, caller needs to save on the stack:

- Its return address
- Any arguments and temporaries needed after the call

Restore from the stack after the call

C Sort Example

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1)
  {
    for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j -= 1)
    {
      swap(v,j);
    }
  }
}
```

Swap Function

Swaps two locations in memory

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

v in $a0, k in $a1, temp in $t0

Swap Function

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

Loop 1

```
for (i = 0; i < n; i += 1)
```

```
        move $s0, $zero          # i = 0
for1tst: slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1   # go to exit1 if $s0 ≥ $s3 (i ≥ n)

        [ bgt $s0, $s3, exit1 ]
```

Loop 2

```
for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j -= 1)
```

```
        addi $s1, $s0, –1       # j = i – 1

for2tst: slt  $t0, $s1, $zero   # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2        # $t1 = j * 4
        add  $t2, $s2, $t1      # $t2 = v + (j * 4)
        lw   $t3, 0($t2)        # $t3 = v[j]
        lw   $t4, 4($t2)        # $t4 = v[j + 1]
        slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3 (v[j+1] ≥ v[j])
        beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3 (v[j+1] ≥ v[j])
```

Sort Body

```
          move $s2, $a0           # save $a0 into $s2
          move $s3, $a1           # save $a1 into $s3
          move $s0, $zero         # i = 0
for1tst:  slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, –1       # j = i – 1
for2tst:  slt  $t0, $s1, $zero    # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2        # $t1 = j * 4
          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
          lw   $t3, 0($t2)        # $t3 = v[j]
          lw   $t4, 4($t2)        # $t4 = v[j + 1]
          slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
          move $a0, $s2           # 1st param of swap is v (old $a0)
          move $a1, $s1           # 2nd param of swap is j
          jal  swap               # call swap procedure
          addi $s1, $s1, –1       # j –= 1
          j    for2tst            # jump to test of inner loop
exit2:    addi $s0, $s0, 1        # i += 1
          j    for1tst            # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

Full Sort

```
sort:    addi $sp,$sp, –20      # make room on stack for 5 registers
         sw $ra, 16($sp)        # save $ra on stack
         sw $s3, 12($sp)        # save $s3 on stack
         sw $s2, 8($sp)         # save $s2 on stack
         sw $s1, 4($sp)         # save $s1 on stack
         sw $s0, 0($sp)         # save $s0 on stack
         …                      # procedure body
         …
         exit1:
          lw $s0, 0($sp)  # restore $s0 from stack
         lw $s1, 4($sp)         # restore $s1 from stack
         lw $s2, 8($sp)         # restore $s2 from stack
         lw $s3, 12($sp)        # restore $s3 from stack
         lw $ra, 16($sp)        # restore $ra from stack
         addi $sp,$sp, 20       # restore stack pointer
         jr $ra                 # return to calling routine
```

Recursive Procedure Example

C code:

```
int fact (int n){
        if (n < 1) return 1;
        else return n * fact(n - 1);
 }
```

Argument n in $a0

Result in $v0

Recursive Procedure Example

```
fact:
    addi $sp, $sp, -8      # adjust stack for 2 items
    sw   $ra, 4($sp)       # save return address
    sw   $a0, 0($sp)       # save argument
    slti $t0, $a0, 1       # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1     # if so, result is 1
    addi $sp, $sp, 8       #   pop 2 items from stack
    jr   $ra               #   and return
L1: addi $a0, $a0, -1      # else decrement n
    jal  fact              # recursive call
    lw   $a0, 0($sp)       # restore original n
    lw   $ra, 4($sp)       #   and return address
    addi $sp, $sp, 8       # pop 2 items from stack
    mul  $v0, $a0, $v0     # multiply to get result
    jr   $ra               # and return
```

References

Chapter 2.8

Chapter 2.13

(Computer Organization and Design: The Hardware/Software Interface by Hennessy/Patterson, 5th edition)