ECE232: Hardware Organization and Design

Part 7: MIPS Instructions III

http://www.ecs.umass.edu/ece/ece232/

Example: Array Access

- Access the ***i***-th element of an array **A** (each element is 32-bit long)

```
                      # $t0 = address of start of A
                      # $t1 = i
sll $t1,$t1,2         # $t1 = 4*i
add $t2,$t0,$t1       # add offset to the address of A[0]
                      # now $t2 = address of A[i]
lw $t3,0($t2)         # $t3 = whatever is in A[i]
```

if statement

```
if ( condition ) {
   statements
}

               # MIPS code for the condition expression
               #(if condition satisfied set $t0=1)
               beq $t0, $zero, if_end_label

               # MIPS code for the statements

if_end_label:
```


if else statement

```
if ( condition ) {
   if-statements
} else {
   else-statements
}
               # MIPS code for the condition expression
               #(if condition satisfied set $t0=1)
               beq $t0, $zero, else_label
               # MIPS code for the if-statements
               j if_end_label
else_label:
               # MIPS code for the else-statements
if_end_label:
```

while statement

```
while ( condition ) {
   statements
}

while_start_label:
            # MIPS code for the condition expression
            #(if condition satisfied set $t0=1)
            beq $t0, $zero, while_end_label
            # MIPS code for the statements
            j while_start_label
while_end_label:
```


do-while statement

```
do {
   statements
} while ( condition );

do_start_label:
            # MIPS code for the statements
do_cond_label:
            # MIPS code for the condition expression
            #(if condition satisfied set $t0=1)
            beq $t0, $zero, do_end_label
            j do_start_label
do_end_label:
```

for loop

```
for ( init ; condition ; incr ) {
  statements
}
             # MIPS code for the init expression
for_start_label:
             # MIPS code for the condition expression
             #(if condition satisfied set $t0=1)
             beq $t0, $zero, for_end_label
             # MIPS code for the statements
             # MIPS code for the incr expression
             j for_start_label
for_end_label:
```


switch statement

```
switch ( expr ) {
  case const1: statement1
  case const2: statement2
  ...
  case constN: statementN
  default: default-statement
}
```

MIPS code for *switch* statement

```
      # MIPS code for $t0=expr
      beq $t0, const1, switch_label_1
      beq $t0, const2, switch_label_2
      ...
      beq $t0, constN, switch_label_N
      j switch_default
switch_label_1:
            # MIPS code to compute statement1
switch_label_2:
            # MIPS code to compute statement2
      ...
switch_default:
            # MIPS code to compute default-statement
switch_end_label:
```


Logical AND in expression

```
if (cond1 && cond2){
  statements
}
      # MIPS code to compute cond1
      # Assume that this leaves the value in $t0
      # If cond1=false $t0=0
      beq $t0, $zero, and_end
      # MIPS code to compute cond2
      # Assume that this leaves the value in $t0
      # If cond2=false $t0=0
      beq $t0, $zero, and_end
      # MIPS code for the statements
 and_end:
```

Switch Example

```
switch (i) {                          //Assume i is in $s1 and j is in $s2;
    case 0: j = 3; break;
    case 1: j = 5; break;
    case 2: ;
    case 3: j = 11; break;
    case 4: j = 13; break;
    default: j = 17;
}
    main:
        add     $t0, $zero, $zero     # $t0 = 0, temp. variable
        beq     $t0, $s1, case0       # go to case0
        addi    $t0, $t0, 1           # $t0 = 1
        beq     $t0, $s1, case1       # go to case1
        addi    $t0, $t0, 1           # $t0 = 2
        beq     $t0, $s1, case2       # go to case2
        addi    $t0, $t0, 1           # $t0 = 3
        beq     $t0, $s1, case3       # go to case3
        addi    $t0, $t0, 1           # $t0 = 4
        beq     $t0, $s1, case4       # go to case4
        j       default               # go to default case
    case0:
        addi    $s2, $zero, 3         # j = 3
        j       finish                # exit switch block
```


Example: Conditional and unconditional branches

- Conditional branch: Jump to instruction L1 if register1 equals register2: `beq $s1, $s2, L1`
  Similarly, `bne`

- Unconditional branch:
  `j L1`
  `jr $s5` (useful for large case statements and big jumps)

- Convert to assembly:

```
if  (i == j)
      f = g+i;
 else
      f = g-i;
```

→

```
        bne $s0, $s1, ELSE
        add $s3, $s2, $s0
        j EXIT
ELSE:
        sub $s3, $s2, $s0
EXIT:
```

Example 2

- Convert to assembly:
- while (save[i] == k)
  i += 1;
- i and k are in $s3 and $s5 and
- base of array save[] is in $s6

```
Loop:  sll   $t1, $s3, 2
       add   $t1, $t1, $s6
       lw    $t0, 0($t1)
       bne   $t0, $s5, Exit
       addi  $s3, $s3, 1
       j     Loop
Exit:
```


Procedures

- Each procedure (function, subroutine) maintains a scratchpad of register values – when another procedure is called (the callee), the new procedure takes over the scratchpad – values may have to be saved so we can safely return to the caller
  - parameters (arguments) are placed where the callee can see them
  - control is transferred to the callee
  - acquire storage resources for callee
  - execute the procedure
  - place result value where caller can access it
  - return control to caller

Registers

- The 32 MIPS registers are partitioned as follows:

• Register 0 :	$zero	always stores the constant 0
• Regs 2-3 :	$v0, $v1	return values of a procedure
• Regs 4-7 :	$a0-$a3	input arguments to a procedure
• Regs 8-15 :	$t0-$t7	temporaries
• Regs 16-23:	$s0-$s7	variables
• Regs 24-25:	$t8-$t9	more temporaries
• Reg 28 :	$gp	global pointer
• Reg 29 :	$sp	stack pointer
• Reg 30 :	$fp	frame pointer
• Reg 31 :	$ra	return address

Jump-and-Link

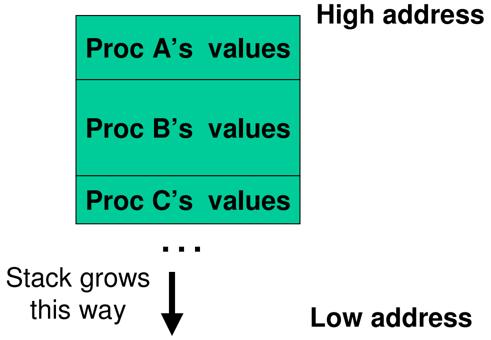
- A special register (not part of the register file) maintains the address of the instruction currently being executed – this is the **program counter (PC)**
- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and
- jump to the procedure's address (the PC is accordingly set to this address)

```
jal    NewProcedureAddress
```

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction
- How do we return control back to the caller after completing the callee procedure?

The Stack

- The register scratchpad for a procedure seems volatile
- It may be modified every time we switch procedures
- A procedure's values are therefore backed up in memory on a stack

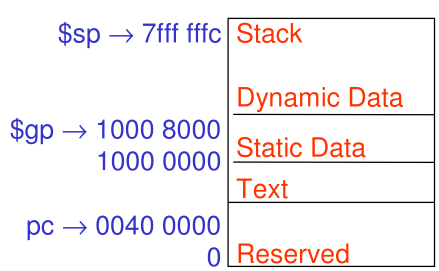



```
Proc  A

      call  Proc B
            ...
            call Proc C
              ...
            return
      return
return
```


What values are saved?

Preserved	Not Preserved
Saved registers: \$s0-\$s7	Temporary registers: \$t0-\$t9
Stack Pointer: \$sp	Argument registers: \$a0-\$a3
Return Address Register: \$ra	Return registers: \$v0-\$v1
Stack above the stack pointer	Stack below the pointer



Storage Management on a Call/Return

- Arguments are copied into \$a0-\$a3; the jal is executed
- The new procedure (callee) must create space for all its variables on the stack
- After the callee creates stack space, it updates the value of \$sp
- Once the callee finishes, it copies the return value into \$v0, frees up stack space, and \$sp is incremented
- On return, the caller may bring in its stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

Leaf Procedure Example

- Procedures that don't call other procedures
- C code:

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

  - Arguments g, …, j in \$a0, …, \$a3
  - f in \$s0 (hence, need to save \$s0 on stack)
  - Result in \$v0

Leaf Procedure Example

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

- MIPS code:

```
leaf_example:
  addi $sp, $sp, -4        Save $s0 on stack
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1       Procedure body
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero     Result
  lw   $s0, 0($sp)         Restore $s0
  addi $sp, $sp, 4
  jr   $ra                 Return
```


Non-Leaf 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

  - Example - **Recursion** (C code):

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

    - Argument n in $a0
    - Result in $v0

Non-Leaf Procedure Example

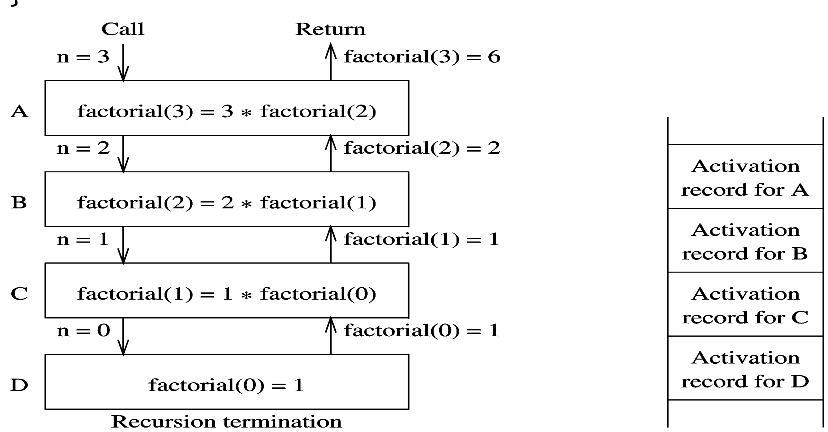
- **MIPS code:**

```
factorial:
    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 i < 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 i
    jal  factorial         # recursive call
    lw   $a0, 0($sp)       # restore previous i
    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
```

**Notes: The callee saves $a0 and $ra in its stack space.
Temps are never saved.**

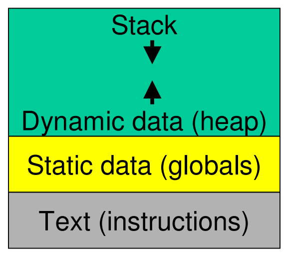
Recursion: Factorial

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



Memory Organization

- The space allocated on stack by a procedure is termed the activation record (includes saved values and data local to the procedure)
- Frame pointer points to the start of the record and stack pointer points to the end
- Variable addresses are specified relative to \$fp as \$sp may change during the execution of the procedure
- \$gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap



Summary

- The jal instruction is used to jump to the procedure and save the current PC (+4) into the return address register
- Arguments are passed in \$a0-\$a3; return values in \$v0-\$v1
- Since the callee may over-write the caller's registers, relevant values may have to be copied into memory
- Each procedure may also require memory space for local variables
- A stack is used to organize the memory needs for each procedure