MIPS Procedure and Function Calls

- When making a *procedure* or function *call*, it is necessary to:
  - Place parameters you wish to pass where they can be accessed by the procedure;
  - Transfer control to procedure;
  - Perform the desired task;
  - Place the result value(s) where the calling program can access them;
  - Return control to the point of origin.
- The procedure call should not clobber any registers other than those agreed upon by protocol.
- The MIPS architecture
  - Provides instructions to assist in procedure calls (jal) and returns (jr);
  - Uses software conventions to
    - Place procedure input and output values.
    - Control which registers are saved/restored by caller and callee.
  - Uses a software stack to save/restore values.
Review of the Machine Cycle

• When a program is executing, its instructions are located in main memory. The address of an instruction is the address of the first (the lowest addressed) byte of the four-byte instruction.

• Each machine cycle executes one machine instruction. At the top of the machine cycle, the PC (program counter) contains the address of an instruction to fetch from memory. The instruction is fetched into the processor and is prepared for execution. In the middle of the machine cycle the PC is incremented by four so that it points to the instruction that follows the one just fetched. Then the fetched instruction is executed and the cycle repeats. The machine cycle automatically executes instructions in sequence.

• When a jump instruction is executed it puts a new address into the PC. Now the fetch at the top of the next machine cycle fetches the instruction at the new address. Instead of executing the instruction that follows the jump instruction in memory, the processor “jumps” to an instruction somewhere else in memory.

• When a jump instruction is executed it puts a new address into the PC. Now the fetch at the top of the next machine cycle fetches the instruction at the new address. Instead of executing the instruction that follows the jump instruction in memory, the processor “jumps” to an instruction somewhere else in memory.

Review of the Machine Cycle

• However, it takes an extra machine cycle for the change in the PC to take effect. Before the PC changes, the next sequential instruction is fetched and is executed. The instruction that follows a jump instruction in memory is said to be in the branch delay slot.

• The reason for this delay is that MIPS is pipelined. Normally instructions are executed in sequence. In order to gain speed, the processor fetches several instructions in sequence and starts working on them. When the machine cycle calls for one of these instructions to be executed, most of the work is already done. These instructions are in an instruction pipe. The instruction after a jump instruction is already in the pipe when the jump is being executed.

• The instruction that follows a jump instruction in memory (in the branch delay slot) is always executed. The instruction that executes after the one in the branch delay slot is the instruction at the new address. Often the branch delay slot is filled with a no-op instruction.

• The SPIM simulator allows you to turn the pipeline feature off, but this is not an option with actual R2000 hardware.
Alterning the PC

• Here is a sequence of instructions. The "load" and "add" are there to represent any arbitrary instructions. Once started, the four instructions execute in an unending loop. The last instruction, a no-op, fills the branch delay slot to give the PC time to change.

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>PC just after this instruction has executed (at the bottom of the cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400000</td>
<td>load</td>
<td>00400004</td>
</tr>
<tr>
<td>00400004</td>
<td>add</td>
<td>00400008</td>
</tr>
<tr>
<td>00400008</td>
<td>jump 0x400000</td>
<td>0040000C</td>
</tr>
<tr>
<td>0040000C</td>
<td>no-op</td>
<td>00400000 -- effect of the jump</td>
</tr>
</tbody>
</table>

• A loop structure is created with the jump instruction. The intent of the jump instruction is to put the address 0x00400000 into the PC. However, this effect is not seen until after the instruction in the branch delay slot.

Jump Instruction - Review

• How does a 32-bit instruction specify a 32-bit address? Some of the instruction's bits must be used for the op-code. Here is the assembly language version of the jump instruction.

j target  # after a delay of one machine cycle
         # PC ← address of target

• There is room in the instruction for a 26-bit address. The 26-bit target address field is transformed into a 32-bit address. Here is how this is done:

  – Instructions always start on an address that is a multiple of four (they are word-aligned). So the low order two bits of a 32-bit instruction address are always "00". Shifting the 26-bit target left two places results in a 28-bit word-aligned address. Now four bits are concatenated to the high-order end of the 28-bit address to form a 32-bit address. These four bits are copied from the top four bits of the PC.
The Jump Instruction

<table>
<thead>
<tr>
<th>Target field of jump instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump instruction 00001010110001010001010001100010</td>
</tr>
</tbody>
</table>

PC 010101100111011011011111001010010111
Copy high order four bits from PC

32-Bit Jump Address 01011011000101000101000110001000
Shift left two positions

Simple Subroutine Linkage

- All high level languages have the concept of a **subroutine** (sometimes called procedure, function, or method). A subroutine is a logical division of the code that may be regarded as a self-contained operation. A subroutine might be executed several times with different data as the program executes.
- The next few slides look at a simple implementation in assembly language of this idea. The simple implementation is not adequate for the full power of subroutines (as implemented in high level languages), but is a good starting point.
Callers and Callees

- At the right is a sketch of what you can do with j (jump) and b (branch) instructions. If the main routine needs to start up ("call") a subroutine sub, it can jump to it with a j instruction. At the end of the subroutine, control can be returned with another j instruction.
- There is a label in main (ret) on the statement the subroutine should return to. The subroutine is written with the expectation that it is called at just one point in main and that it returns to an address a few instructions after that point.
- The subroutine cannot be called from several locations in the main program because it must always return to the same location.
- A subroutine call is when a main routine (or other routine) passes control to a subroutine. The main routine is said to be the CALLER and the subroutine is said to be the CALLEE. A return from a subroutine is when a subroutine passes control back to its CALLER.

Many Calls - One Return

- The problem is illustrated at right. The main routine is written to call a useful subroutine (sub) at several locations in the code. But sub is written to return to only one location. This will not work.
- What is needed is a method that sends the return address to the subroutine. When the subroutine finishes, it passes control to that return address.
- Of course, "passing control to a return address" means to load the PC (program counter) with the return address. The next instruction fetch of the machine cycle will get the instruction from that address.
jal Instruction

- The register that is used for linkage is register $31, which is called $ra by the extended assembler. It holds the return address for a subroutine. The instruction that puts the return address into $ra is the jal instruction.
- The jal instruction does the following (during the execute phase of the machine cycle):
  - Jal sub: $ra ← PC+4  # $ra ← address 8 bytes away from the jal
  - PC ← sub  # load the PC with the subroutine entry point
- Tricky: the middle step of the machine cycle has already incremented the PC by four (so that it points at the instruction after the jal instruction). Executing the jal instruction loads this address plus four so that the $ra register points at the second instruction after the jal instruction.
- The correct return address is "address of the jal plus eight". This is because: (i) returning from the subroutine to the jal instruction would be a disaster (since it would execute again, sending control back to the subroutine), and (ii) the instruction following the jal is in the branch delay slot.

Example jal Instruction

- It would not be a disaster to return control to an instruction that does nothing. But sometimes clever programmers or clever compilers put something unexpected in the branch delay slot, so it is best not to pass control to it.
- The diagram shows the execution of a jal instruction. The jal is at address 0x00400014. The return address is 0x0040001C which is the address of the jal plus eight. (The addu instruction there is just used as an example of what might be at the return address).
The jr Instruction

- The jr instruction returns control to the caller. It copies the contents of $ra into the PC:
  \[
  \text{jr } \$ra \quad \# \ PC \leftarrow \$ra
  \]
- Usually you think of this as "jumping to the address in $ra."
- To make the instruction more general, it can be used with any register, not just $ra. Like all jump and branch instructions, the jr instruction is followed by a branch delay.
- The diagram shows the subroutine returning to the return address that was loaded into $ra by the jal instruction in the caller.

Calling Convention

- A calling convention is an agreement about how subroutines are called and how control is returned to the caller. Mostly (as we will later see) this is an agreement about how software will work. In processors like MIPS that support subroutines, the convention says how those support features are used.
- By agreement between programmers, registers have been assigned different roles with subroutine linkage:
  - $t0 - $t9 — The subroutine is free to change these registers.
  - $s0 - $s7 — The subroutine must not change these registers.
  - $a0 - $a3 — These registers contain arguments for the subroutine. The subroutine can change them.
  - $v0 - $v1 — These registers contain values returned from the subroutine.
Pictorial Summary

- The picture shows `main` calling `mysub`. Two arguments are passed, in `$a0` and `$a1`. The subroutine reads the arguments from those registers.
- In the picture, the arguments are set up with `move` and `li` instructions, but any means of loading the argument registers can be used.
- The **Simple Linkage Convention** is limited in some obvious ways. A more advanced calling convention will be discussed when we talk about **stack** usage.

Example Program

- Let us write a program that uses the *simple linkage convention*. The program is to read three integers from the user and compute the sum. The outline of the program is:
  
  ```
  # read first integer
  # read second integer
  # read third integer
  # compute the sum
  # write the result
  ```

- The user will enter integers as characters from the keyboard. The program uses the exception handler service number five (syscall) to read the characters and convert them to a full word.
Prompt for and Read Integer Subroutine

- The subroutine prompts the user for an integer and reads it in. The integer is returned in $v0. Here is a start on the subroutine:

```assembly
# pread -- prompt for and read an integer
# on entry: $ra -- return address
# on exit: $v0 -- the integer
pread:
    la $a0,prompt # print string
    li $v0,4   # service 4
    syscall
    li $v0,5   # read int into $v0
    syscall   # service 5
    jr $ra    # return
    nop       # branch delay slot
.data
prompt: .asciiz "Enter an integer"
```

Main Program

```assembly
.text
.globl main
main: jal pread    # read first integer
    nop
    move $s0,$v0 # save it in $s0
    jal pread   # read second integer
    nop
    move $s1,$v0 # save it in $s1
    jal pread   # read third integer
    nop
    move $s2,$v0 # save it in $s2
    addu $s0,$s0,$s1 # compute the sum
    addu $a0,$s0,$s2
    li $v0,1    # print the sum
    syscall
    li $v0,10   # exit
    syscall
```
Global Symbols

- Subroutines should not know about each other's symbolic addresses. It would violate the idea of modularity for `main` to do something to `pread's` prompt, for example.
- But some symbolic addresses need to be used between modules. For example, `pread` is a symbolic address, and `main` must know about it and use it in the `jal` instruction.
- A symbol that a subroutine makes visible to other subroutines is a global symbol. Global symbols often label entry points. Symbols that are not global are called local symbols. In MIPS assembly and in QTSpim, a symbol is made global by placing it in a list of symbols following the .globl directive:
  
  .globl main

Complete Program

```
# addthree.asm — print sum of three integers
#
# This program uses simple subroutine linkage.
#
# Settings: Load delays ON; Branch delays ON
# Trap file ON; Pseudoinstructions ON
.text
.globl main
main:
  jal pread        # read first integer
  nop
  move $s0,$v0    # save it in $s0
  jal pread
  nop
  move $s1,$v0    # save it in $s1
  jal pread
  nop
  move $s2,$v0    # save it in $s2
  addu $s0,$s0,$s1 # compute the sum
  addu $a0,$s0,$s2
  li $v0,1
  syscall        # print the sum
  li $v0,10
  syscall        # exit

# pread -- prompt for and read an integer
#
# on entry:
#    $ra -- return address
#
# on exit:
#    $v0 -- the integer
.text
.globl pread
pread:
  la $a0,prompt  # print string
  li $v0,4
  syscall
  li $v0,5
  syscall
  jr $ra

.data
prompt:.asciiz "Enter an integer"
```

```
Summary

- Reviewed the jump (j) instruction;
- Introduced a simple linkage convention;
- Discussed the limitations of this convention;
- Illustrated with a complete program.