Topic 3-a

Calling Convention
Calling convention is a standardized method for a program to pass parameters to a procedure and receive result values back from it.
Consideration of Call Convention

• Where to place parameters and return values (in registers, on the activation stack, or a mix of both)
• The order in which parameters are passed
• Responsibility for setting up and cleaning up a function call - distributed between the calling and the called code.
• Different platforms use different call conventions, and so can different programming languages, even on the same platform.
Issues in Calling Convention

• Register usage convention
• Calling sequence
• Parameter passing
• Local data layout
Register Usage Convention

• Special registers
  -- stack register, frame register, return address register, return value registers)

• Reserved registers

• Temporary registers (not preserved across a procedure call)

• Saved Registers (preserved across a procedure call)
Context Switching – Register Saving

Register contents are saved before context switching into another procedure.

• Caller-save versus Callee-save disciplines
• Contents saved in frame of saver
• Often choose to keep values in registers for efficiency
Register Saving (Con’t)

• **Caller save**

  The calling procedure (caller) is responsible for saving and restoring any registers that must be preserved across the call. The called procedure (callee) can then modify any register without constraint.

• **Callee save**

  The callee is responsible for saving and restoring any registers that it might use. The caller uses registers without worrying about restoring them after a call.
Without Register Saving

Assume register set: saved registers + temporary registers

```java
main (){
   ...; subA;
}
can use saved registers, temporary registers

subA(){
   ...; subB;...;
}  
can’t use saved registers but can use temporary registers

subB()
    can’t use saved registers but can use temporary registers
```

*can’t save any value in saved registers!*
Caller Save/Callee Save Trade-Off

• If all caller save?
  ▪ Even callee doesn’t kill any of the saved registers
    – waste of cycles and memory resource

• If all callee save?
  ▪ Callee has to save all the register (which will be used by callee), even caller doesn’t use them
Caller Save/Callee Save Trade-Off

- caller save
  temporary registers
- callee save
  saved registers
Caller save example

Function A

Add $10, $11,$12
Save $10, $12, $13
Jal B
Restore $10, $12, $13
Sub $11, $2, $12
Mul $12, $10, $13

Function B

Add $2, $4, $5
Br $31

How to save ?
Callee Save example

Function A
- Add $10, $11, $12
- Jal B
- Add $11, $2, $12
- Mul $12, $10, $13

Function B
- Save $10, $11, $12 (if they are used in B)
- Lw $10, 4(sp)
- Sub $11, $8, $9
- Sub $12, $11, $10
- Sub $2, $12, $11
- Restore $10, $11, $12
- Br $31
Callee Save example

Function A
- Add $10, $11, $12
- Jal B
- Sub $11, $2, $12
- Mul $12, $10, $13

Function B
- Add $2, $4, $5
- Br $31

No save!
Calling Sequences

A calling sequence is code statements to create activation records on the stack and enter data in them. It consists of call sequence and return sequence.
Calling Sequences (Con’t)

• A call sequence allocates an activation record and enters information into its fields.
  -- parameters, return address, old stack top, saving registers, local data, etc.

• A return sequence restores the state of the machine
  -- return value, restore registers, restore old stack top, branch to return address

• The code in calling sequence is often divided between the caller and the callee.
More responsibilities are in caller side.

Call sequence:

- Caller evaluates actual parameters
- Caller stores a return address and the old stack top into callee’s AR.
- Caller pushes the AR (i.e. increment top_sp)
- Callee saves register values and other status information
- Callee initializes local data and begin execution
Return sequence:

- Callee places a return value next to the AR of the caller.
- Callee restores top_sp and other registers
- Callee branches to the return address
- Caller can copy return value into its own AR
Calling Sequence Implementation: Approach I (Con’t)

- Actual parameters
- Return values
- Optional control link
- Optional access link
- Save machine status
- Local Data
- Temporaries

Caller's AR

- Caller put actual parameters
- Control link
- access link
- Return address
- top_sp

Caller’s responsibility

2/29/2008
Calling Sequence Implementation: Approach I (Con’t)

- Actual parameters
- Return values
- Optional control link
- Optional access link
- Save machine status
- Local Data
- Temporaries

Caller’s AR

Caller put actual parameters

- Control link
- access link
- Return address
- top_sp
- Local Data
- Temporaries

Callee’s AR

Caller’s responsibility

Callee’s responsibility

2/29/2008
Calling Sequence Implementation: Approach I (Con’t)

- Actual parameters
- Return values
- Optional control link
- Optional access link
- Save machine status
- Local Data
- Temporaries

Caller’s AR

Callee’s AR

2/29/2008
Calling Sequence Implementation: Approach II (MIPS)

More responsibilities are in callee side.

Call sequence:

1. Push onto the register save field for any temporary registers ($t0-$t9) that contain values that must be saved. The callee procedure might change these registers.

2. Put first 4 argument values into $a0-$a3 and others into incoming argument field.

3. Call the subroutine
Calling Sequence Implementation: Approach II (MIPS)

**In callee:**

**Prolog (done by the callee at its beginning):**
1. If this procedure might call other procedures, save return address onto the stack.
2. Push onto the stack any saved registers ($s0$-$s7$) that this procedure might alter.

**Callee Body:**
1. The procedure may alter any register that it saved in the prolog.
2. If the procedure calls another procedure, then it does so by following above rules.
Calling Sequence Implementation: Approach II (MIPS)

Return sequence:

Epilog (done by the procedure just before it returns to the caller):

1. Put returned values in registers($v0-$v1)
2. Pop from the stack (in reverse order) any registers ($s0-$s7) that were pushed in the prolog.
3. If it was pushed in the prolog, pop the return address from the stack into register $ra.
4. Return to the caller using jr $ra.
Calling Sequence Implementation: Approach II (MIPS)

A frame of Implementation code sequence:

- Procedure Call
  - Regain control from procedure

- Prolog of procedure

- Procedure body

- Epilog of procedure
Parameter Passing

• Parameters
  - Names that appear in the declaration of a procedure are formal parameters.
  - Variables and expressions that are passed to a procedure are actual parameters (or arguments)

• Parameter passing modes
  - Call by value
  - Call by reference
  - Copy-restore
  - Call by name
Call-by-Value

An evaluation strategy where arguments are evaluated before the procedure is entered. Only the values of the arguments are passed and changes to the arguments within the called procedure have no effect on the actual arguments as seen by the caller.
Features of Call-by-Value

1. The actual parameters are evaluated and their \textit{r-values} are passed to the called procedure.

2. A procedure called by value can affect its caller either through nonlocal names or through pointers.

3. Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

4. Pascal uses pass by value by default, but \textit{var} parameters are passed by reference.
Call-by-Reference

An argument passing convention where the address of an argument variable is passed to a procedure. Execution of the procedure may have side-effect on the actual argument as seen by the caller. The C language's "&" (address of) and "*" (dereference) operators allow the programmer to code explicit call-by-reference. Other languages provide special syntax to declare reference arguments.
Features of Call-by-Reference

1. Also known as call-by-address or call-by-location. The caller passes to the called procedure the \textit{l-value} of the parameter.

2. If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

3. Parameters in Fortran are passed by reference.
Parameter Passing: An Example

main()
{
    int i = 10;
    fun( i );
    printf(“i=%d\n”, i);
}
fun( x )

int x
{
    x = 20;
}

Call-by–value:

print: i = 10;

Call-by–reference:

Print: i = 10 or 20 ?
    Print: i = 10
How can i = 20?

Why?
Copy-Restore

1. A hybrid between call-by-value and call-by reference.

2. The actual parameters are evaluated and their r-values are passed as in call-by-value. In addition, l-values are determined before the call.

3. When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual parameters.
Why Copy-Restore

Copy-Restore avoids the problem when a procedure call has more than one way to access a variable.

program A
int a;

program f(var x)
  { x:=2; a:=0; }
  { a:=1; f(a); print(a) }

In this example, call-by-reference will print 0, copy-restore will print 2.
Call-by-Name

1. The actual parameters literally substituted for the formals. This is like a macro-expansion or in-line expansion.

2. Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used.

3. In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.