Procedure Call

CSE 410
Lecture 06
Lecture Outline

❖ Procedure call requirements
❖ What the caller does
❖ What the called method does
❖ Optimizations
Procedure Call

❖ Suppose we have a function

   void printString(const char*pString);

   ▪ That’s C

   ▪ It means “printString” is a function that:
     • takes a single argument, a pointer to a character in memory
     • returns nothing
     • Aside: In C, strings are just consecutive bytes representing ASCII characters. The end of the string is marked by a byte that has 0 in it.

❖ We might call it like this:

   ▪ char str[] = “Example string”;
   ▪ printString(str);
Procedure call issues

- In general (so, considering both our example and in other examples as well) we need ways to:
  - get the argument values from the caller to the callee
  - get the return value from the callee back to the caller
  - create new instances of local variables in the callee for this call
  - transfer control to/from the caller to the callee (i.e., branch)
  - return from the callee to the place it was called from

- There might be 8 calls to printString in the application. When printString wants to return, where should it go?
The Stack

The space needed for the invocations of procedures forms a stack

- if A calls B and B calls C, then
  - We allocate space for A, then
  - allocate space for the invocation of B, then
  - allocate space for the invocation of C, then
  - deallocate space for C, then
  - deallocate space for B, then
  - deallocate space for A
Support for the stack

- The stack is so important that some hardware architectures provide direct support for it
- On the 410 machine, it’s supported by convention
  - R7 is the stack pointer
- Allocate: subtract from R7
- Free: add to R7s
Stack Frames

- Each procedure invocation results in a stack frame being constructed on the stack
- A calls B calls C...
410 Stack Frame Contents

- The exact composition of a stack frame is a software decision
  - For code generated by different compilers to interoperate, they must agree on frame format!

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Established by caller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Argument N</td>
<td></td>
</tr>
<tr>
<td>Return address</td>
<td></td>
</tr>
<tr>
<td>Local variable 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Local variable n</td>
<td></td>
</tr>
</tbody>
</table>
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if ( max < y ) max = y;
    return max;
}
```

```c
max(5, 10);
```
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if ( max < y ) max = y;
    return max;
}
```

... max(5, 10);
...
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if ( max < y ) max = y;
    return max;
}
```

```c
max(5, 10);
```
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if (max < y) max = y;
    return max;
}
```

```plaintext
... max(5, 10);
...
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

R7
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if (max < y) max = y;
    return max;
}
```

```
max(5, 10);
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

max(5, 10);
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if (max < y) max = y;
    return max;
}
```

```plaintext
max(5, 10);
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
```

```
R7...
```
Example: int max(int x, int y)

```c
int max(int x, int y) {
    int max = x;
    if ( max < y ) max = y;
    return max;
}
```

max(5, 10);

```
10
5
10
?
```
The Return Address

- The 410 ISA includes an instruction CALL
  - CALL R1  max
- Call does two things:
  - Puts the incremented PC in RA
    - That’s the address of the instruction that follows the CALL instruction
  - Sets the PC to IMMED_15 value in the instruction, shifted left one bit
    - We have only 15 bits, but addresses are 16 bits
      - We don’t store the lowest order bit of the address
      - We assume that bit is 0
    - This means functions must be “aligned” to an even address
Implementation: max

```assembly
# int max(int x, int y)

.align
max:    addi  r7  r7  $-4  # allocate stack space
addi  r2  r0  $4       # index from sp to return address field
sw    r1  r7  r2       # save return address

addi  r2  r0  $8       # index from sp to x
lw    r3  r7  r2       # r3 = x
addi  r4  r0  $2       # index from sp to max
sw    r3  r7  r4       # max = x

addi  r2  r0  $6       # index from sp to y
lw    r5  r7  r2       # r5 = y

cmp    r0  r3  r5      # x cmp y
bgt    :done           # skip ahead if x > y
be     :done           # skip ahead if x == y
sw    r5  r7  r4       # max = y
done:    lw    r5  r7  r4       # r5 = max
addi  r3  r0  $10      # index from sp to return value
sw    r5  r7  r3       # set return value

addi  r2  r0  $4       # index from sp to return address
lw    r1  r7  r2       # fetch return address
addi  r7  r7  $4       # restore stack pointer
jr     r0  r0  r1       # return
```
Implementation: main

```asm
# main()
  # r1 is return address
  # r7 is stack pointer

.align
main:  sw    r1  0  r7       # push return address
       addi  r7  r7  $-2  # update stack pointer

  addi  r7  r7  $-6  # allocate space for return val and 2 args
  addi  r2  r0  $4   # index from sp to 1st arg
  addi  r3  r0  $5   # first arg value
  sw    r3  r7  r2   # store in stack frame
  addi  r2  r2  $-2  # index from sp to 2nd arg
  addi  r3  r0  $10  # second arg value
  sw    r3  r7  r2   # store in stack frame
  call  r1  max      # call max
  # note: at this point we have no idea what's in any register
  #       (except r0 and r7)
  addi  r2  r0  $6   # index from sp to return value
  lw    r3  r7  r2   # fetch return value into r3
  addi  r7  r7  $6   # restore stack pointer

  printr r3           # print the max

  addi  r1  r7  $2    # restore stack pointer
  lw    r1  r0  r1   # fetch return address
  jr    r0  r0  r1   # return
```

Implementation: prologue

Execution actually starts here. (The prologue is loaded at address 0x0010.)

```
prologue:   addi   r1   r0   0x0f   # set up stack pointer register
            shftl  r7   r1   $12
            # we could pass argc/argv here, if we had them
            call   r1   main
            stop
```
Optimizations

- It’s common to leave the return value in a register, if it fits, rather than writing it to memory.
- It’s common to place the first few arguments in registers, rather than on the stack, if they fit.
- It’s common for the caller to save some registers on the stack before making the call.
  - The registers might hold values it will need after the call returns.
- It’s common for the callee to save some registers, if it uses them.
- There is often a frame pointer register pointing at the current frame. It is saved in a new frame before being reset, so that the frames are “chained.”