SPIM Procedure Calls

22C:60 Jonathan Hall

March 29, 2008

1 Motivation

We would like to create procedures that are easy to use and easy to read. To this end we will discuss standard conventions as it relates to the stack and registers. We will compare JAVA procedure calls and see how they relate to a SPIM procedure and how to enforce similarities.

Most of this is gleaned from the February 7 lecture notes and *Patterson* 79-90.

2 Conventions

We would like to establish certain conventions so that we need not know anything about the procedure that we are calling. This is equivalent to asking how we use a procedure, and we would like every procedure to be used the same – for them to have the same interface. Of course this is not possible in practice, but we can come close.

The conventions are that \$a0-\$a3 are the arguments of a procedure and \$v0-\$v1 are the return values of a procedure. So this could look something like this in JAVA:

```
word[2] procedure(word $a0, word $a1, word $a2, word $a3)
{ word v[2] = { $v0, $v1}; return v; }
```

So to call a procedure in SPIM, we put the arguments in \$a0-\$a3 and jump to it. The procedure puts the return values in \$v0 and/or \$v1 where the caller can then read the values. In the case that the procedure has more than four arguments or more then two return values, we can use the stack to transfer data.

3 Preserving calling state

We would like to ensure that whenever a procedure is called that the caller has certain guaranties about what to expect – what has changed and what has not after a procedure call. This is equivalent to making certain registers 'local' to the procedure. These guaranties are as follows:

What should be preserved after a call:

- $s0-$s7, the save registers
- $ra, the return address register
- $sp, the stack pointer register
- $fp, the frame pointer register
- Every thing that was on the stack previous to the procedure call

What may change after a call:

- $t0-$t9, the temporary registers may be different
- $a0-$a3, the argument registers may be different
- $v0-$v1, the return value registers may be different
- Anything 'above' the stack may be different

These requirements may be met by the called procedure saving the necessary information on the stack.

4 Managing the stack

As discussed before, a procedure has certain obligations to its caller and to the whole program. To meet these requirements we use the stack. We would like, as programmers, management of the stack to be as unobtrusive and readable as possible. There are two cases we need to consider: a static stack frame and a dynamic one.

4.1 Static stack frame

This is the case where the maximum size of the stack frame is known by the procedure. For the most part this will be the common case (and easier case). In this case we only need to store a subset of $s0-$s7, $ra, and any local information (including $a0-$a3). Of these, we store the ones that will be written over by our procedure, and any local information that we would like to keep before calling another procedure. We do not need to save $sp since we know the maximum size that it will be incremented, and we don't need to save $fp since we do not use it.

We would like to get all the stack stuff done as soon as possible so we can focus on our code and not managing the stack. In general, we store everything that we will need to save right upon entering the procedure, making space for anything we do not know yet. This would look like this:

```
procedure:
```

```
addi $sp, $sp, -MaxStackFrameSize #push the frame
(save everything that needs to be saved using sw)
(your procedure goes here)
(using lw, load everything that was saved)
addi $sp, $sp, MaxStackFrameSize #pop the frame
jr $ra
```

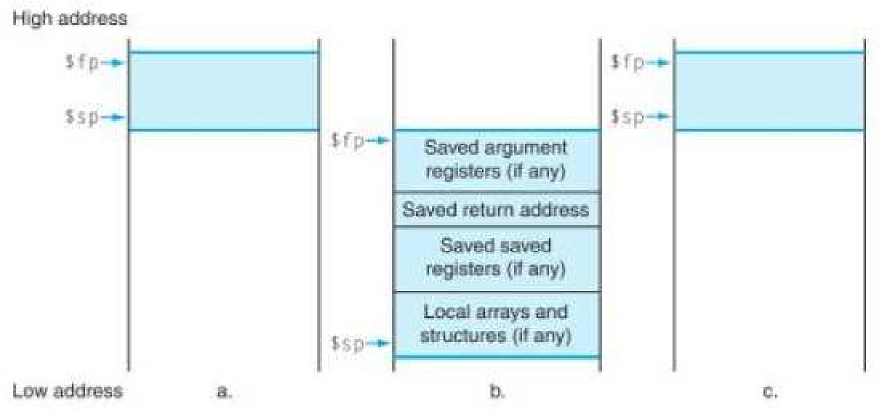
Here we have got all the pushing and popping of the stack out of the way. We may still need to read and write from the stack, but no managing the stack. The only thing we need to remember is where we put everything. Even this could be taken care of by declaring some symbols (i.e. PROC1VAR1 = Offset of variable 1).

4.2 Dynamic stack frame

Our maximum stack frame size may not be known when we enter a procedure (e.g. putting an array on the stack). In this case we need to do more stack management and introduce another register to handle it. We still need to save everything as in the previous case with the addition of two more things: the frame pointer register $fp and the stack pointer register $sp. We store $fp on the stack and $sp in $fp. Now it looks something like this:

```
procedure:
    addi $sp, $sp, -NewStackFrameSize #push the frame
    (save $fp and everything that needs to be saved using sw)
    add $fp, $sp,NewStackFrameSize #store old $sp in $fp
    (your procedure goes here)
    (using lw, load everything except $fp that was saved)
    move $sp, $fp #pop the frame
    lw $fp, FPOFFSET($sp) #restore old frame pointer
    jr $ra
```

We may still need to push new things to the stack in our procedure, but with the addition of the frame pointer we do not need to worry too much about indexing into the stack. This is because instead of using the changing $sp as a pointer into the stack , we can use $fp that stays the same during the procedure. So when we read or write to the stack we do so by offsetting from the frame pointer (i.e. lw $s0, 8($fp)).



High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

FIGURE 2.16 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call.

5 JAVA to SPIM examples

5.1 Static stack frame:

JAVA:

```
void main()
{
    foo(777);
}
int foo(int n)
{
    return n*bar();
}
int bar(){ return 1337; }
```

SPIM:

```
###### void main() #######
main:
    # foo(777)
    li $a0, 777
    jal foo
```

```
###### end main ######
###### int foo(int n) #####
# $a0 = n
# $v0 = return value
# stack frame = $ra, maxSize = 4
foo:
    #push stack frame
    addi $sp, $sp, -4
    sw $ra, 0($sp)
    #$v0 = bar()
    jal bar
    # return n*bar()
    mult $v0 , $a0, $v0
    #pop stack frame
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra
###### end foo ######
###### int bar() ######
# $v0 = 1337
bar:
    # no use of $s0-$s7 and no procedure calls so
    # no stack frame
    li $v0, 1337
    jr $ra
###### end bar ######
```

5.2 Too many arguments

Here we will use five arguments, too many to store in $a0-$a3, so we put the fifth on the stack. It may seem redundant to not use another register (i.e. $t0) but what if we had 33 arguments?

JAVA:

```
void main()
{
    foo(1,2,3,4,5);
}
int foo(int a, int b, int c, int d, int e)
{
    return a+b+c+d+e;
}
```

SPIM:

```
###### void main() ######
main:
    # foo(1,2,3,4,5)
    li $a0, 1
    li $a1, 2
    li $a2, 3
    li $a3, 4
    # the fifth argument goes on top of the stack
    li $t0, 5
    sw $t0, -4($sp)
    jal foo
###### end main ######
###### int foo(int a, int b, int c, int d, int e) ######
# $a0 = a, $a1 = b, $a2 = c, $a3 = d
# arg[4] = -4($sp) = e
# $v0 = a+b+c+d+e
foo:
    # push stack frame including the fifth arg
    addi $sp, $sp, -8
    sw $ra, 0($sp)
    # Note: now 4($sp) = d.
    # In general arg[i] = frameSize - 4*(i-3) for i > 3,2,1,0
    # $t0 = 4($sp) = arg[4]
    lw $t0, 4($sp)
    # put $a0+$a1+$a2+$a3+$t0 in $v0
    # return $v0
    #pop stack frame
    lw $ra, 0($sp)
    addi $sp, $sp, 8
    jr $ra
###### end foo ######
```