Procedures 201:

Higher Level Calling Conventions

Slides revised 3/25/2014 by Patrick Kelley

Procedures

- Higher Level languages have adopted a standard
  - Referred to as C-style calling
  - Uses the stack to pass parameters and returns
  - Keeps local variables on the stack
  - Allows for recursive calling
- Every call to a procedure maintains a unique stack frame
- Registers are always preserved by the callee

Procedure Design

- Begins with the calling parameters and returns
- Avoid complexity
  - Use 32-bit data and pointers
  - Avoid bytes, half-words, strings, and arrays
- Add the data required; this is the stack-frame size
- Caller puts arguments onto the stack
  - In reverse order
  - Without moving stack pointer
  - Must ensure that there is no stack overflow

Ex. Factorial function

- Imagine an hll (say, C++) function for factorial
- The prototype is: int factorial(int input)
- Let’s use 32-bit unsigned ints
- So we need 2 words, one for input and one for the result
- Without knowing anything more, we can define the calling code for the fact function.

The Calling Stack

Special	Address	Data
	0x00377238	
	0x0037723C	
Input(sp-8)	0x00377240	0x0
return (sp-4)	0x00377244	0x0
Stack Pointer	0x00377248	0x000045F7
	0x0037724C	

- Remember, the top of the stack grows toward smaller addresses.
- Caller does not change the stack pointer
- Must check for stack overflow (grow past top of stack) if possible (not easily on our MIPS)

Example Call

```
.text
… # some code before this, then set up for the call
  li    $t0, 0xB                 # get the parameter
  sw    $t0, -8(sp)              # put it on the stack frame
                                 # return value is at -4(sp)
  jal   Factorial                # make the call
  lw    $t0, -4(sp)              # get the return value into $t0

…
```

- Note that we didn’t care about preserving $to
- If we had some way to know the top of the stack, we could compare it plus our framesize to $sp
- Caller did not change $sp or initialize the return value
- Multiple parameters are put on stack in reverse order

Push and Pop

```
.text
…
  # push a value (register) onto the stack
  sw     $t0, -4(sp)                # any register will do
  addiu  $sp, $sp, -4               # adjust stack pointer
…
  # pop a value (register) from the stack
  lw     $t0, 0(sp)                 # any register will do
  addiu  $sp, $sp, 4                # adjust stack pointer
…
```

- Pop values in the reverse order they were pushed
- Should check before pushing if there is room:
  - Platform dependent
  - On QTSPIM, put a label at end of data section:

```
       .data
…      # data declarations
enddata:
       .text
…
       la     $t0, enddata        # any register will do
       subu   $t0, $sp, $t0       # the register now holds the
                                  # available stack space
```

Pushes and Pops

```
.text
…
  # push multiple values (registers) onto the stack
  sw     $t0, -4(sp)                # any registers
  addiu  $sp, $sp, -4               # adjust stack pointer
  sw     $t1, -4(sp)
  addiu  $sp, $sp, -4               # adjust stack pointer
  sw     $s5, -4(sp)
  addiu  $sp, $sp, -4               # adjust stack pointer
…
  # pop multiple values (registers) from the stack
  # in reverse order
  lw     $s5, 0(sp)                 # any registers
  addiu  $sp, $sp, 4                # adjust stack pointer
  lw     $t1, 0(sp)
  addiu  $sp, $sp, 4                # adjust stack pointer
  lw     $t0, 0(sp)
  addiu  $sp, $sp, 4                # adjust stack pointer
…
```

Pushes and Pops(alternate)

```
.text
…
  # push multiple values (registers) onto the stack
  sw     $t0, -4(sp)                # any registers
  sw     $t1, -8(sp)
  sw     $s5, -12(sp)
  addiu  $sp, $sp, -12              # adjust stack pointer
…
  # pop multiple values (registers) from the stack
  lw     $t0, 8(sp)                 # any registers
  lw     $t1, 4(sp)
  lw     $s5, 0(sp)
  addiu  $sp, $sp, 12               # adjust stack pointer
…
-  OR  -
 # pop multiple values (registers) from the stack
  addiu  $sp, $sp, 12               # adjust stack pointer
  lw     $t0, -4(sp)                # any registers
  lw     $t1, -8(sp)
  lw     $s5, -12(sp)
…
```

Notice that pop order does not matter now

Procedure Design 2

- The called procedure:
  - Saves frame pointer
  - Copies the stack pointer to the frame pointer (so we can get it back later)
  - Moves the stack pointer to the top of the passed params
  - If it uses registers, it pushes them onto the stack
- Local variables are put on the stack as well
- Before return:
  - local variables are deallocated
  - Registers are popped off the stack
  - Copy frame pointer to stack pointer
  - Restore original frame pointer

Ex. Factorial function

- Let’s implement this:

  ```
  int Factorial (int input)
  {
      int dummy;  // to make it a little interesting
      dummy = 5;
      if (input == 0)
          return 1;
      else return input * Factorial(input – 1);
  }
  ```

- And we’ll use registers $s0, $s1, $s2, and $s3

The Factorial Stack Frame (at call)

	Special	Address	Data
		0x00377220	?
		0x00377224	?
		0x00377228	?
		0x0037722C	?
		0x00377230	?
		0x00377234	?
		0x00377238	?
		0x0037723C	?
	Input(sp-8)	0x00377240	0xB
	return (sp-4)	0x00377244	?
SP		0x00377248	0x000045F7
		0x0037724C	

Setting the Frame Pointer

```
.text
…
  # Factorial expects an input at $sp – 8 and computes the factorial
  # of that input.  It returns the value at $sp – 4.  Since the
  # factorial algorithm is recursive, a stack frame is used…
Factorial:
  # push current frame pointer onto stack ($fp or $30)
  sw     $fp, -12(sp)             # remember -8 and -4 currently in use
  move   $fp, $sp                 # copy stack pointer to frame pointer
  addiu  $sp, $sp, -12            # adjust stack pointer
…
```

The Factorial Stack Frame ($fp set)

	Special	Address	Data
		0x00377220	?
		0x00377224	?
		0x00377228	?
		0x0037722C	?
		0x00377230	?
		0x00377234	?
		0x00377238	?
SP	Old Frame Pointer	0x0037723C	$fp (old)
	Input(fp-8)	0x00377240	0xB
	return (fp-4)	0x00377244	?
FP		0x00377248	0x000045F7
		0x0037724C	

Saving Registers

```
.text
…
  # Factorial expects an input at $sp – 8 and computes the factorial
  # of that input.  It returns the value at $sp – 4.  Since the
  # factorial algorithm is recursive, a stack frame is used…
Factorial:
  # push current frame pointer onto stack ($fp or $30)
  sw     $fp, -12(sp)             # remember -8 and -4 currently in use
  move   $fp, $sp                 # copy stack pointer to frame pointer
  addiu  $sp, $sp, -12            # adjust stack pointer

  # save $ra and any other registers we need
  sw     $ra, -4(sp)              # return address and our other regs
  sw     $s0, -8(sp)
  sw     $s1, -12(sp)
  sw     $s2, -16(sp)
  sw     $s3, -20(sp)
  addiu  $sp, $sp, -20            # adjust stack pointer
…
```

The Factorial Stack Frame (save regs)

	Special	Address	Data
		0x00377220	?
		0x00377224	?
SP	$s3	0x00377228	$s3 (old)
	$s2	0x0037722C	$s2 (old)
	$s1	0x00377230	$s1 (old)
	$s0	0x00377234	$s0 (old)
	$ra	0x00377238	$ra (old)
	Old Frame Pointer	0x0037723C	$fp (old)
	Input(fp-8)	0x00377240	0xB
	return (fp-4)	0x00377244	?
FP		0x00377248	0x000045F7
		0x0037724C	

Space for Local Variables

```
.text
…
  # Factorial expects an input at $sp – 8 and computes the factorial
  # of that input.  It returns the value at $sp – 4.  Since the
  # factorial algorithm is recursive, a stack frame is used…
Factorial:
  # push current frame pointer onto stack ($fp or $30)
  sw     $fp, -12(sp)             # remember -8 and -4 currently in use
  move   $fp, $sp                 # copy stack pointer to frame pointer
  addiu  $sp, $sp, -12            # adjust stack pointer

  # save $ra and any other registers we need
  sw     $ra, -4(sp)              # any registers
  sw     $s0, -8(sp)
  sw     $s1, -12(sp)
  sw     $s2, -16(sp)
  sw     $s3, -20(sp)
  addiu  $sp, $sp, -20            # adjust stack pointer

  # reserve space for the local variable ‘dummy’ at 0($sp)
  addiu  $sp, $sp, -4
  li     $s0, 5                   # for storing into ‘dummy’
  sw     $s0, 0($sp)              # store the local value
…
```

The Factorial Stack Frame (locals)

	Special	Address	Data
		0x00377220	?
SP	‘dummy’	0x00377224	5
	$s3	0x00377228	$s3 (old)
	$s2	0x0037722C	$s2 (old)
	$s1	0x00377230	$s1 (old)
	$s0	0x00377234	$s0 (old)
	$ra	0x00377238	$ra (old)
	Old Frame Pointer	0x0037723C	$fp (old)
	Input(fp-8)	0x00377240	0xB
	return (fp-4)	0x00377244	?
FP		0x00377248	0x000045F7
		0x0037724C	

The Factorial Algorithm

```
.text
…
# reserve space for the local variable ‘dummy’ at 0($sp)
  addiu  $sp, $sp, -4
  li     $s0, 5                    # for storing into ‘dummy’
  sw     $s0, 0($sp)               # store the local value

# load the input parameter into a register
  lw     $s1, -8($fp)              # remember, $fp points where $sp was
                                   # on the procedure call

# see if the input is 0 or not
  bnez   $s1, callFact             # if not 0, do recursive call
  li     $s2, 1                    # otherwise set the return ($s2) to 1
  j      doneFact                  # jump to return code

callFact:
  addiu  $s3, $s1, -1              # $s3 is parameter for recursive call
  sw     $s3, -8(sp)               # put it on the stack frame
  jal  Factorial                   # make the call
  lw     $s3, -4(sp)               # get the return value into $s3
  multu  $s3, $s1                  # multiply the return * input
  mflo   $s2                       # assume not bigger than LO and put
                                   # in $s2 for return
…
```

Preparing to Return

```
.text
…
# see if the input is 0 or not
  bnez   $s1, callFact            # if not 0, do recursive call
  li     $s2, 1                   # otherwise set the return ($s2) to 1
  j      doneFact                 # jump to return code

callFact:
  addiu  $s3, $s1, -1             # $s3 is parameter for recursive call
  sw     $s3, -8(sp)              # put it on the stack frame
  jal  Factorial                  # make the call
  lw     $s3, -4(sp)              # get the return value into $s3
  multu  $s3, $s1                 # multiply the return * input
  mflo   $s2                      # assume not bigger than LO and put
                                  # in $s2 for return
doneFact:
  sw     $s2, -4(fp)              # put our return value relative to $fp

  # now we can begin cleanup prior to return
  addiu  $sp, $sp, 4              # done with local variables so adjust $sp
…
```

The Factorial Stack Frame (free vars)

	Special	Address	Data
		0x00377220	?
		0x00377224	5
SP	$s3	0x00377228	$s3 (old)
	$s2	0x0037722C	$s2 (old)
	$s1	0x00377230	$s1 (old)
	$s0	0x00377234	$s0 (old)
	$ra	0x00377238	$ra (old)
	Old Frame Pointer	0x0037723C	$fp (old)
	Input(fp-8)	0x00377240	0xB
	return (fp-4)	0x00377244	0x2611500
FP		0x00377248	0x000045F7
		0x0037724C	

Restoring Registers

```
.text
…
doneFact:
  sw     $s2, -4(fp)               # put our return value relative to $fp

  # now we can begin cleanup prior to return
  addiu  $sp, $sp, 4               # done with local variables so adjust $sp

  # pop saved registers
  addiu  $sp, $sp, 20              # adjust stack pointer
  sw     $ra, -4(sp)               # restore return address
  sw     $s0, -8(sp)               # restore the other registers we used
  sw     $s1, -12(sp)
  sw     $s2, -16(sp)
  sw     $s3, -20(sp)
…
```

The Factorial Stack Frame (pop regs)

	Special	Address	Data
		0x00377220	?
		0x00377224	5
		0x00377228	$s3 (old)
		0x0037722C	$s2 (old)
		0x00377230	$s1 (old)
		0x00377234	$s0 (old)
		0x00377238	$ra (old)
SP	Old Frame Pointer	0x0037723C	$fp (old)
	Input(fp-8)	0x00377240	0xD
	return (fp-4)	0x00377244	0x2611500
FP		0x00377248	0x000045F7
		0x0037724C	

Collapse Stack Frame

```
.text
…
doneFact:
  sw     $s2, -4(fp)               # put our return value relative to $fp

  # now we can begin cleanup prior to return
  addiu  $sp, $sp, 4               # done with local variables so adjust $sp

  # pop saved registers
  addiu  $sp, $sp, -20             # adjust stack pointer
  sw     $ra, -4(sp)               # restore return address
  sw     $s0, -8(sp)               # restore the other registers we used
  sw     $s1, -12(sp)
  sw     $s2, -16(sp)
  sw     $s3, -20(sp)

  # restore stack pointer and frame pointer to collapse stack frame
  move   $sp, $fp                  # stack pointer is back where it was
  lw     $fp, -12(sp)              # get old $fp from where we stored it
  jr     $ra                       # everything back like it was, so return
# END OF Factorial Procedure
…
```

The Factorial Stack Frame (collapsed)

	Special	Address	Data
		0x00377220	?
		0x00377224	5
		0x00377228	$s3 (old)
		0x0037722C	$s2 (old)
		0x00377230	$s1 (old)
		0x00377234	$s0 (old)
		0x00377238	$ra (old)
		0x0037723C	$fp (old)
	Input(sp-8)	0x00377240	0xD
	return (sp-4)	0x00377244	0x2611500
SP		0x00377248	0x000045F7
		0x0037724C	

Stack Frame Summary

Caller:

- Pushes $ra if not already on stack before anything else
- Puts parameters on stack above stack pointer location
- Leaves room for returns below params
- Does NOT adjust the stack pointer after storing params
- Calls the procedure with the ‘jal’ instruction

Stack Frame Summary (cont.)

Callee:

- Saves $fp on stack, copies $sp to $fp, and then adjusts $sp to point to saved $fp
- Pushes $ra and other registers on stack
- Adjusts stack pointer to allow for local variables

When ready to return:

- Adjusts stack pointer back to before local variables
- Pops $ra and other registers from stack
- Copies $fp to $sp and then restores $fp from stack
- Returns by 'jr $ra' to the Caller