Shapes of functions - Practice in MIPS coding

1a ⟨* 1a⟩≡
⟨*sample code* 1b⟩

1 Just return

```
void silly(void)
{
   return;
}
```

Code

The caller did: `jal silly`, so the return address is in $ra.

1b ⟨*sample code* 1b⟩≡

```
silly:   jr   $ra
```

2 Call something, then return

```
void f( void )
{
   silly();
   return;
}
```

Layout of Stack Frame

The return address is kept in `0($sp)`.

Code

1c ⟨*sample code* 1b⟩+≡

```
f:
```

⟨*f prologue* 1e⟩
⟨*f body* 1d⟩
⟨*f epilogue* 2a⟩

The body clobbers $ra, so we need to save it. See the prologue.

1d ⟨*f body* 1d⟩≡

```
          jal  silly
```

1e ⟨*f prologue* 1e⟩≡

```
          add  $sp, $sp, -4
          sw   $ra, 0($sp)
```

2a ⟨*f epilogue* 2a⟩≡
```
          lw   $ra, 0($sp)
          addi $sp, $sp, 4
          jr   $ra
```

3 One parameter

```
int g( int x )
{
   return( x );
}
```

Code

The **a**rguments are in $a0, $a1, ..., and the return **v**alue goes in $v0.

2b ⟨*sample code* 1b⟩+≡
```
  g:      move $v0, $a0
          jr   $ra
```

4 Two parameters

```
int h( int x , int y )
{
   return( x + y );
}
```

Code

Just like function g in the last section.

2c ⟨*sample code* 1b⟩+≡
```
  h:       add   $v0, $a0, $a1  #calculate the return value
           jr    $ra
```

5 Combine...

```
int p( int x , int y )
{
   return( g( x + y ) );
}
```

Layout of stack frame

The return address goes in `0($sp)`.

Code

3a ⟨*sample code* 1b⟩+≡
```
p:
⟨p prologue 3c⟩
⟨p body 3b⟩
⟨p epilogue 3d⟩
```

3b ⟨*p body* 3b⟩≡
```
          add   $a0, $a0, $a1 # prepare the argument for g
          jal   g             # and call
            # and g's return value is our return value
            # so we don't have to do any more work
```

3c ⟨*p prologue* 3c⟩≡
```
          addi  $sp, $sp, -4
          sw    $ra, 0($sp)
```

3d ⟨*p epilogue* 3d⟩≡
```
          lw    $ra, 0($sp)
          addi  $sp, $sp, 4
          jr    $ra
```

6 Local variables

```
void q( int x )
{
   int i, j ,k;

   i = x;
   j = x + i;
   k = x + 2;
   x = k;
}
```

Layout of stack frame

Variables i, j, and k go in `0($sp)`, `4($sp)`, and `8($sp)`.

Code

3e ⟨*sample code* 1b⟩+≡
```
q:
⟨q prologue 4b⟩
⟨q body 4a⟩
⟨q epilogue 4c⟩
```

4a ⟨*q body* 4a⟩≡
```
          sw   $a0, 0($sp)  # i = x;

          lw   $t0, 0($sp)  # get i
          add  $t1, $a0, $t0  # calculate x + i
          sw   $t1, 4($sp)  # value goes in j

          addi $t3, $a0, 2  # x + 2
          sw   $t3, 8($sp) # goes in k

          lw   $a0, 8($sp) # x gets k
```

4b ⟨*q prologue* 4b⟩≡
```
          addi $sp, $sp, -12
```

4c ⟨*q epilogue* 4c⟩≡
```
          addi $sp, $sp, 12
          jr   $ra
```

7 Local variables, plus call

```
void r( int x );
{
   int i, j ,k;

   q( x );
   k = x;
   i = j;
   return;
}
```

Layout of stack frame

Variables i, j, and k go in `0($sp)`, `4($sp)`, and `8($sp)`.

The return address goes in `12($sp)`.

Code

4d ⟨*sample code* 1b⟩+≡
```
  r:
  ⟨r prologue 5b⟩
  ⟨r body 5a⟩
  ⟨r epilogue 5c⟩
```

5a ⟨*r body* 5a⟩≡
```
          jal  q   # the argument x is already in $a0

          sw   $a0, 8($sp)   # k = x;

          lw   $t0, 4($sp)   #
          sw   $t0, 0($sp)   # i = j;
```

5b ⟨*r prologue* 5b⟩≡
```
          addi $sp, $sp, -16
          sw   $ra, 12($sp)
```

5c ⟨*r epilogue* 5c⟩≡
```
          lw   $ra, 12($sp)
          addi $sp, $sp, 16
          jr   $ra
```

8 How to apply a function to an argument

We can take any function of the form `int foo(int)`, apply it to any integer argument, and get back the return value.

The new idea here is making the function a variable, rather than a constant.

Analogy: `j label` jumps to a constant label, while `jr` jumps to a variable address, which is in a register.

Present situation: `jal label` calls a function at a (constant) label.

The new instruction `jalr` calls a function whose (variable) address is in a register.

5d ⟨*sample code* 1b⟩+≡
⟨*function to call* (never defined)⟩
⟨*calling apply* 5f⟩
⟨*apply* 6⟩

We pick a very simple function:

5e ⟨*sample code* 1b⟩+≡
```
#       int square(int n)
#
square: mul     $v0,$a0,$a0
        jr      $ra
```

Show how to call apply:

5f ⟨*calling apply* 5f⟩≡
```
#
        la      $a0,square      # pointer to a function
        li      $a1,1024        # argument for that function
        jal     apply           # call apply
```

and the function apply:

6 ⟨*apply* 6⟩≡

```
#
#       int apply( int (*pf)(int), int arg)
#
apply:  addi    $sp,$sp,-4
        sw      $ra,0($sp)

        move    $t0,$a0 # address of function
        move    $a0,$a1 # prepare argument
        jalr    $t0     # call the function

        lw      $ra,0($sp)
        addi    $sp,$sp,4
        jr      $ra
```

9 Scaffolding

7 ⟨*sample code* 1b⟩+≡

```
main:
          jal   silly

          jal   f

          li    $a0, 691
          jal   g
          move  $a0, $v0 # print out the return value
          li    $v0, 1   # print_int
          syscall

          li     $a0, 131072
          li     $a1, -1
          jal    h
          move   $a0, $v0
          li     $v0, 1  # print_int
          syscall

          li     $a0, 131072
          li     $a1, -1
          jal    p
          move   $a0, $v0
          li     $v0, 1  # print_int
          syscall

          li     $a0, -262144
          jal    q

          li     $a0, 4369
          jal    r

          li     $v0, 10  # exit
          syscall
```