Introducing ARM

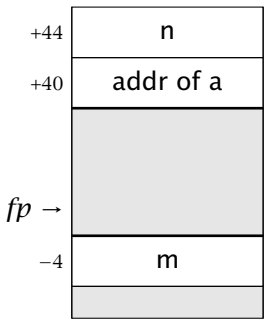
Mike Spivey, November 2017

We will translate the following procedure to ARM code.

```
type vec = array 10 of integer;

proc max(var a: vec; n: integer): integer;
  var i, j, k: integer; (* Register variables *)
    m: integer; (* Variable in stack frame *)
begin
  i := 0; m := 0;
  while i < n do
    if a[i] > m then m := a[i] end;
    i := i+1
  end;
  return m
end;
```

I've added unused variables `j` and `k` that will be allocated registers in order to force the variable `m` to be kept in the stack frame for the sake of variety. Shown at the right is the frame layout used on the ARM.



The code begins with some boilerplate that stores the incoming arguments from registers `r0` and `r1` into the stack frame, saves other registers that the procedure must preserve, and allocates space for local variables.

```
_max:
   mov ip, sp
   stmfd sp!, {r0-r1}
   stmfd sp!, {r4-r10, fp, ip, lr}
   mov fp, sp
   sub sp, sp, #8
```

Next comes code for the assignments `i := 0; m := 0` that reflects the fact that `i` lives in the register `r4` but `m` lives in the stack frame.

```
@ i := 0;
   mov r4, #0
@ m := 0;
   mov r0, #0
   str r0, [fp, #-4]
```


The test for the **while** loop uses **i** from the register, but must load the parameter **n** from the stack frame.

```
@ while i < n do
.L2:
   ldr r0, [fp, #44]
   cmp r4, r0
   bge .L4
```

Next comes code the **if** statement, where both the condition and the body refer to **a**[**i**], a *common subexpression*. We can save fetching the value twice by saving it in a register (r7) between the two uses. First comes code to put **a**[**i**] in the register, computing the offset of the element by shifting the value of **i** left by 2. We exploit an addressing mode that forms an address by adding two registers.

```
   ldr r0, [fp, #40]
   lsl r1, r4, #2
   ldr r7, [r0, r1]
```

Now we can compile the **if** test easily.

```
@ if a[i] > m then
   ldr r0, [fp, #-4]
   cmp r7, r0
   ble .L7
```

Also the body is easily translated into a single store instruction.

```
@ m := a[i]
   str r7, [fp, #-4]
```

The loop body ends with code to increment **i** and branch back the the start.

```
.L7:
@ i := i+1
   add r4, r4, #1
   b .L2
```

Finally, the value of **m** is returned by putting it in r0 and restoring registers, including the program counter.

```
.L4:
@ return m
   ldr r0, [fp, #-4]
   ldmfd fp, {r4-r10, fp, sp, pc}
```