CHAPTER 11. INSTRUCTION DETAILS

@ assignment2.s
@ Assignment three ways.
@ 2017-09-29: Bob Plantz

@ Define my Raspberry Pi
.cpu  cortex-a53
.fpu  neon-fp-armv8
.syntax  unified  @ modern syntax

@ Useful source code constants
.equ  z,-20
.equ  local,8

@ Constant program data
.section  .rodata
.align  2
formatMsg:
.ascii "^I^%i + %i = %i^\n"

@ Program code
.text
.align  2
.global  main
.type  main, %function

main:
  sub  sp, sp, 16  @ space for saving regs
  str  r4, [sp, 0]  @ save r4
  str  r5, [sp, 4]  @ r5
  str  fp, [sp, 8]  @ fp
  str  lr, [sp, 12]  @ and lr
  add  fp, sp, local  @ our frame pointer
  sub  sp, sp, local  @ allocate memory for local var
  mov  r5, 123  @ x = 123;
  ldr  r4, yValue  @ y = 4567;
  add  r3, r5, r4  @ x + y
  str  r3, [fp, z]  @ z = x + y;
  ldr  r0, formatMsgAddr  @ printf( "^%i + %i = %i^\n"
  mov  r1, r5  @ x
  mov  r2, r4  @ y
  ldr  r3, [fp, z]  @ z;
  bl  printf
  mov  r0, 0  @ return 0;
  add  sp, sp, local  @ deallocate local var
  ldr  r4, [sp, 0]  @ restore r4
  ldr  r5, [sp, 4]  @ r5
  ldr  fp, [sp, 8]  @ fp
  ldr  lr, [sp, 12]  @ and lr
  add  sp, sp, 16  @ restore sp
  bx  lr  @ return

.align  2
yValue:
.word  4567
formatMsgAddr:
.word  formatMsg

Listing 11.2.3 Assignment to a register variable (prog asm).
First, notice that the values in the r4 and r5 registers must be saved on the stack in the prologue:

\[
\begin{align*}
\text{sub} & \quad \text{sp, sp, 16} \quad @ \text{space for saving regs} \\
\text{str} & \quad \text{r4, [sp, 0]} \quad @ \text{save r4} \\
\text{str} & \quad \text{r5, [sp, 4]} \quad @ \quad \text{r5} \\
\text{str} & \quad \text{fp, [sp, 8]} \quad @ \quad \text{fp} \\
\text{str} & \quad \text{lr, [sp, 12]} \quad @ \quad \text{and lr} \\
\end{align*}
\]

and restored in the epilogue:

\[
\begin{align*}
\text{ldr} & \quad \text{r4, [sp, 0]} \quad @ \text{restore r4} \\
\text{ldr} & \quad \text{r5, [sp, 4]} \quad @ \quad \text{r5} \\
\text{ldr} & \quad \text{fp, [sp, 8]} \quad @ \quad \text{fp} \\
\text{ldr} & \quad \text{lr, [sp, 12]} \quad @ \quad \text{and lr} \\
\text{add} & \quad \text{sp, sp, 16} \quad @ \text{restore sp} \\
\end{align*}
\]

as is specified in Table 10.1.1.

After setting up our frame pointer, we move the stack pointer to allocate space on the stack for the local variable:

\[
\begin{align*}
\text{add} & \quad \text{fp, sp, 12} \quad @ \text{our frame pointer} \\
\text{sub} & \quad \text{sp, sp, local} \quad @ \text{allocate memory for local var} \\
\end{align*}
\]

where the value of local was computed to (a) allow enough memory space for the int variable, and (b) make sure the stack pointer is always on an eight-byte addressing boundary, as required by the protocol when calling a public function (printf in this case).

You have already seen the first two assignment implementations:

\[
\begin{align*}
\text{mov} & \quad \text{r5, 123} \quad @ \text{\textit{x} = 123;} \\
\text{ldr} & \quad \text{r4, yValue} \quad @ \text{\textit{y} = 4567;} \\
\end{align*}
\]

in Listing 10.1.4. The integer value, 123, is within the range that can be moved directly into a register. However, 4567 cannot, so it is stored in memory and loaded into a register from memory.

The compiler honored our request to use registers for both the x and y variables. However, the z variable is allocated in the stack frame. So after the addition is performed, the sum is stored in memory at a location relative to the frame pointer:

\[
\text{str} \quad \text{r3, [fp, z]} \quad @ \text{z = x + y;}
\]

Recall from Section 9.2 that [fp, z] specifies the address obtained by adding the value of z to the value contained in the fp register. In this function z is an offset of -16 bytes from the address in fp.

In Section 11.3 we discuss the machine code for the instructions that implement these assignment statements. In particular, we will be looking at how the location of each variable is encoded in the machine language.

### 11.3 Machine Code, Assignment

Each assembly language instruction must be translated into its corresponding machine code, including the locations of any data it manipulates. It is the bit pattern of the machine code that directs the activities of the control unit.

The goal here is to show you that a computer performs its operations based on bit patterns. That is, on-off switches that are connected in ways that were introduced in Chapters 5–8.

As you read through this material, keep in mind that even though this material is quite tedious, the operations are very simple. Fortunately, instruction execution is very fast, so lots of meaningful work can be done by the computer.