2.14 A C sort example to put it all together

This section has been set as optional by your instructor.

One danger of showing assembly language code in snippets is that you will have no idea what a full assembly language program looks like. In this section, we derive the LEGv8 code from two procedures written in C: one to swap array elements and one to sort them.

Figure 2.14.1: A C procedure that swaps two locations in memory (COD Figure 2.25).

This subsection uses this procedure in a sorting example.

```
void swap(long long int v[], size_t k)
{
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

The procedure `swap`

Let's start with the code for the procedure `swap` in the above figure. This procedure simply swaps two locations in memory. When translating from C to assembly language by hand, we follow these general steps:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

This section describes the `swap` procedure in these three pieces, concluding by putting all the pieces together.

Register allocation for `swap`

The LEGv8 convention on parameter passing is to use registers `X0` to `X7`. Since swap has just two parameters, `v` and `k`, they will be found in registers `X0` and `X1`. The only other variable is `temp`, which we associate with register `X9` since swap is a leaf procedure. This register allocation corresponds to the variable declarations in the first part of the swap procedure in the above figure

Code for the body of the procedure `swap`

The remaining lines of C code in swap are

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

Recall that the memory address for LEGv8 refers to the *byte* address, and so doublewords are really 8 bytes apart. Hence, we need to multiply the index `k` by 8 before adding it to the address. *Forgetting that sequential doubleword addresses differ by 8 instead of by 1 is a common mistake in assembly language programming.* Hence, the first step is to get the address of `v[k]` by multiplying `k` by 8 via a shift left by 3:

```
LSL   X10, X1, #3    // reg X10 = k * 8
ADD   X10, X0, X10   // reg X10 = v + (k * 8)
                     // reg X10 has the address of v[k]
```

Now we load `v[k]` using `X10`, and then `v[k+1]` by adding 8 to `X10`:

```
LDUR   X9, [X10,#0]  // reg X9 (temp) = v[k]
LDUR   X11,[X10,#8]  // reg X11 = v[k + 1]
                     // refers to next element of v
```

Next we store `X9` and `X11` to the swapped addresses:

```
STUR   X11,[X10,#0]  // v[k] = reg X11
STUR   X9, [X10,#8]  // v[k+1] = reg X9 (temp)
```

Now we have allocated registers and written the code to perform the operations of the procedure. What is missing is the code for preserving the saved registers used within `swap`. Since we are not using saved registers in this leaf procedure, there is nothing to preserve.

The full `swap` procedure

We are now ready for the whole routine, which includes the procedure label and the return branch. To make it easier to follow, we identify in the figure below each block of code with its purpose in the procedure.

Figure 2.14.2: LEGv8 assembly code of the procedure swap (COD Figure 2.26).

Procedure body

```
swap:  LSL    X10, X1,#3          // reg X10 = k * 8
       ADD    X10, X0,X10         // reg X10 = v + (k * 8)
                                  // reg X10 has the address of v[k]
       LDUR   X9, [X10,#0]        // reg X9 (temp) = v[k]
       LDUR   X11,[X10,#8]        // reg X11 = v[k + 1]
                                  // refers to next element of v
       STUR   X11,[X10,#0]        // v[k] = reg X11
       STUR   X9, [X10,#8]        // v[k+1] = reg X9 (temp)
```

Procedure return

```
       BR     LR                  // return to calling routine
```

PARTICIPATION ACTIVITY 2.14.1: Swap procedure in C and assembly.

Consider the above code for swap(). Assume:

- function header is void swap(int v[], int k)
- array v is {9, 47, 6, 25}
- k is 1
- variable temp is associated with register X9

1) What is the value of v[k] after the swap function executes?

Check Show answer

2) Which register holds v[]?

Check Show answer

3) What value is in X10 after: `LSL X10, X1, #3`

Check Show answer

4) Suppose v is located at address 4008. After executing `ADD X10, X0, X10`, what is in X10?

Check Show answer

5) What is the value of X9 after: `LDUR X9, [X10,#0]`?

Check Show answer

6) What is the value of X11 after: `LDUR X11, [X10,#8]`?

Check Show answer

7) What is the value of v[1] after this instruction: `STUR X11, [X10,#0]`?

Check Show answer

8) What is the value of v[k+1] after this instruction: `STUR X9, [X10,#8]`?

Check Show answer

The procedure `sort`

To ensure that you appreciate the rigor of programming in assembly language, we'll try a second, longer example. In this case, we'll build a routine that calls the swap procedure. This program sorts an array of integers, using bubble or exchange sort, which is one of the simplest if not the fastest sorts. The figure below shows the C version of the program. Once again, we present this procedure in several steps, concluding with the full procedure.

Figure 2.14.3: A C procedure that performs a sort on the array v (COD Figure 2.27).

```
void sort (long long int v[], size_t int n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```

Register allocation for `sort`

The two parameters of the procedure `sort`, `v` and `n`, are in the parameter registers `X0` and `X1`, and we assign register `X19` to `i` and register `X20` to `j`.

Code for the body of the procedure `sort`

The procedure body consists of two nested *for* loops and a call to `swap` that includes parameters. Let's unwrap the code from the outside to the middle.

The first translation step is the first *for* loop:

```
for (i = 0; i < n; i += 1) {
```

Recall that the C *for* statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize `i` to 0, the first part of the *for* statement:

```
        MOV X19, XZR        // i = 0
```

(Remember that `MOV` is a pseudoinstruction provided by the assembler for the convenience of the assembly language programmer.) It also takes just one instruction to increment `i`, the last part of the *for* statement:

```
        ADDI X19, X19, #1   // i += 1
```

The loop should be exited if `i < n` is *not* true or, said another way, should be exited if `i ≥ n`. This test takes two instructions:

```
for1tst: CMP X19, X1        // compare X19 to X1(i to n)
         B.GE exit1         // go to exit1 if X19 >= X1 (i >= n)
```

The bottom of the loop just branches back to the loop test:

```
         B  for1tst         // branch to test of outer loop
exit1:
```

The skeleton code of the first `for` loop is then

```
         MOV  X19, XZR      // i = 0
for1tst: CMP  X19, X1       // compare X19 to X1 (i to n)
         B.GE exit1         // go to exit1 if X19 ≥ X1 (i ≥ n)
            ...
            (body of first for loop)
            ...
         ADDI X19, X19, #1  // i+=1
         B for1tst          // branch to test of outer loop
exit1:
```

Voila! (The exercises explore writing faster code for similar loops.)

The second *for* loop looks like this in C:

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

The initialization portion of this loop is again one instruction:

```
        SUBI X20, X19, #1   // j = i - 1
```

The decrement of `j` at the end of the loop is also one instruction:

```
        SUBI X20, X20, #1   // j - = 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails (`j` < 0):

```
for2tst: CMP X20, XZR       // compare X20 to 0 (j to 0)
         B.LT exit2         // go to exit2 if X20 < 0 (j < 0)
```

This branch will skip over the second condition test. If it doesn't skip, then `j` ≥ 0.

The second test exits if `v[j] > v[j + 1]` is *not* true, or exits if `v[j] ≤ v[j + 1]`. First we create the address by multiplying `j` by 8 (since we need a byte address) and add it to the base address of `v`:

```
        LSL X10, X20, #3    // reg X10 = j * 8
        ADD X11, X0, X10    // reg X11 = v + (j * 8)
```

Now we load `v[j]`:

```
        LDUR X12, [X11,#0]  // reg X12 = v[j]
```

Since we know that the second element is just the following doubleword, we add 8 to the address in register `X11` to get `v[j + 1]`:

```
        LDUR X13, [X11,#8]  // reg X13 = v[j + 1]
```

We test of `v[j] ≤ v[j + 1]` to exit the loop

```
        CMP  X12, X13       // compare X12 to X13
        B.LE exit2          // go to exit2 if X12 <= X13
```

The bottom of the loop branches back to the inner loop test:

```
        B for2tst           // branch to test of inner loop
```

Combining the pieces, the skeleton of the second *for* loop looks like this:

```
        SUBI  X20, X19, #1        // j = i - 1
for2tst: CMP   X20,XZR            // compare X20 to 0 (j to 0)
        B.LT  exit2               // go to exit2 if X20 < 0 (j < 0)
        LSL   X10, X20, #3        // reg X10 = j * 8
        ADD   X11, X0, X10        // reg X11 = v + (j * 8)
        LDUR  X12, [X11,#0]       // reg X12 = v[j]
        LDUR  X13, [X11,#8]       // reg X13 = v[j + 1]
        CMP   X12,X13             // compare X12 to X13
        B.LE  exit2               // go to exit2 if X12 <= X13
           ...
           (body of second for loop)
           ...
        SUBI  X20, X20, #1        // j -= 1
        B     for2tst             // jump to test of inner loop
exit2:
```

The procedure call in `sort`

The next step is the body of the second *for* loop:

```
swap(v, j);
```

Calling `swap` is easy enough:

```
BL swap
```

Passing parameters in `sort`

The problem comes when we want to pass parameters because the `sort` procedure needs the values in registers `X0` and `X1`, yet the swap procedure needs to have its parameters placed in those same registers. One solution is to copy the parameters for `sort` into other registers earlier in the procedure, making registers `X0` and `X1` available for the call of `swap`. (This copy is faster than saving and restoring on the stack.) We first copy `X0` and `X1` into `X21` and `X22` during the procedure:

```
MOV X21, X0   // copy parameter X0 into X21
MOV X22, X1   // copy parameter X1 into X22
```

Then we pass the parameters to `swap` with these two instructions:

```
MOV X0, X21   // first swap parameter is v
MOV X1, X20   // second swap parameter is j
```

Preserving registers in `sort`

The only remaining code is the saving and restoring of registers. Clearly, we must save the return address in register `LR`, since `sort` is a procedure and is called itself. The `sort` procedure also uses the callee-saved registers `X19, X20, X21,` and `X22`, so they must be saved. The prologue of the `sort` procedure is then

```
SUBI  SP, SP, #40   // make room on stack for 5 regs
STUR  LR, [SP,#32]  // save LR on stack
STUR  X22,[SP,#24]  // save X22 on stack
STUR  X21,[SP,#16]  // save X21 on stack
STUR  X20,[SP,#8]   // save X20 on stack
STUR  X19,[SP,#0]   // save X19 on stack
```

The tail of the procedure simply reverses all these instructions, then adds a `BR` to return.

The full procedure `sort`

Now we put all the pieces together in the figure below, being careful to replace references to registers `X0` and `X1` in the *for* loops with references to registers `X21` and `X22`. Once again, to make the code easier to follow, we identify each block of code with its purpose in the procedure. In this example, nine lines of the `sort` procedure in C became 34 lines in the LEGv8 assembly language.

Figure 2.14.4: LEGv8 assembly version of procedure sort (COD Figure 2.28).

Saving registers

```
sort:   SUBI    SP,SP,#40       // make room on stack for 5 registers
        STUR    X30,[SP,#32]    // save LR on stack
        STUR    X22,[SP,#24]    // save X22 on stack
        STUR    X21,[SP,#16]    // save X21 on stack
        STUR    X20, [SP,#8]    // save X20 on stack
        STUR    X19, [SP,#0]    // save X19 on stack
```

Procedure body

```
Move parameters
        MOV     X21, X0         // copy parameter X0 into X21
        MOV     X22, X1         // copy parameter X1 into X22
Outer loop
        MOV     X19, XZR        // i = 0
for1tst:CMP     X19, X1         // compare X19 to X1 (i to n)
        B.GE    exit1           // go to exit1 if X19 ≥ X1 (i≥n)
Inner loop
        SUBI    X20, X19, #1    // j = i - 1
for2tst:CMP     X20,XZR         // compare X20 to 0 (j to 0)
        B.LT    exit2           // go to exit2 if X20 < 0 (j < 0)
        LSL     X10, X20, #3    // reg X10 = j * 8
        ADD     X11, X0, X10    // reg X11 = v + (j * 8)
        LDUR    X12, [X11,#0]   // reg X12 = v[j]
        LDUR    X13, [X11,#8]   // reg X13 = v[j + 1]
        CMP     X12, X13        // compare X12 to X13
        B.LE    exit2           // go to exit2 if  X12 ≤ X13
Pass parameters and call
        MOV     X0, X21         // first swap parameter is v
        MOV     X1, X20         // second swap parameter is j
        BL      swap
Inner loop
        SUBI    X20, X20, #1    // j -= 1
        B       for2tst         // branch to test of inner loop
Outer loop
exit2:  ADDI    X19, X19, #1    // i += 1
        B       for1tst         // branch to test of outer loop
```

Restoring registers

```
exit1:  STUR    X19, [SP,#0]    // restore X19 from stack
        STUR    X20, [SP,#8]    // restore X20 from stack
        STUR    X21,[SP,#16]    // restore X21 from stack
        STUR    X22,[SP,#24]    // restore X22 from stack
        STUR    X30,[SP,#32]    // restore LR from stack
        SUBI    SP,SP,#40       // restore stack pointer
```

Procedure return

```
        BR      LR              // return to calling routine
```

Elaboration

One optimization that works with this example is procedure inlining. Instead of passing arguments in parameters and invoking the code with a ***`BL`*** *instruction, the compiler would copy the code from the body of the* ***`swap`*** *procedure where the call to* ***`swap`*** *appears in the code. Inlining would avoid four instructions in this example. The downside of the inlining optimization is that the compiled code would be bigger if the inlined procedure is called from several locations. Such a code expansion might turn into lower performance if it increased the cache miss rate; see COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy).*

PARTICIPATION ACTIVITY 2.14.2: For loop in assembly.

1) What is the value of X19 after: `MOV X19, XZR`?

Check Show answer

2) How much is X19 increased by: `ADDI X19, X19, #1`?

Check Show answer

3) If the value of X19 is 3 and the value of X1 is 3, will the loop exit after the following instructions? (Answer yes or no.)

```
CMP X19, X1
B.GE exit1
```

Check Show answer

PARTICIPATION ACTIVITY 2.14.3: Building an assembly program from a C program.

1) By convention, LEGv8 uses 8 registers for parameter passing allocation.
   - True
   - False

2) Sequential doubleword addresses differ by 1 byte.
   - True
   - False

3) `MOV` is a pseudoinstruction as a convenience for the assembly language programmer.

- ○ True
- ○ False

Understanding program performance

COD Figure 2.29 (Comparing performance, instruction count, and CPI …) shows the impact of compiler optimization on sort program performance, compile time, clock cycles, instruction count, and CPI. Note that unoptimized code has the best CPI, and O1 optimization has the lowest instruction count, but O3 is the fastest, reminding us that time is the only accurate measure of program performance.

COD Figure 2.30 (Performance of two sort algorithms in C and Java …) compares the impact of programming languages, compilation versus interpretation, and algorithms on performance of sorts. The fourth column shows that the unoptimized C program is 8.3 times faster than the interpreted Java code for Bubble Sort. Using the JIT compiler makes Java 2.1 times faster than the unoptimized C and within a factor of 1.13 of the highest optimized C code. (COD Section 2.15 (Advanced Material: Compiling C and Interpreting Java) gives more details on interpretation versus compilation of Java and the Java and LEGv8 code for Bubble Sort.) The ratios aren't as close for Quicksort in Column 5, presumably because it is harder to amortize the cost of runtime compilation over the shorter execution time. The last column demonstrates the impact of a better algorithm, offering three orders of magnitude a performance increase when sorting 100,000 items. Even comparing interpreted Java in Column 5 to the C compiler at highest optimization in Column 4, Quicksort beats Bubble Sort by a factor of 50 (0.05 × 2468, or 123 times faster than the unoptimized C code versus 2.41 times faster).

Figure 2.14.5: Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort (COD Figure 2.29).

The programs sorted 100,000 32-bit words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (procedure integration)	2.41	65,747	44,993	1.46

Figure 2.14.6: Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version (COD Figure 2.30).

The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as in the figure above. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	O1	2.37	1.50	1562
	Compiler	O2	2.38	1.50	1555
	Compiler	O3	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

Elaboration

The ARMv8 compilers always save room on the stack for the arguments in case they need to be stored, so in reality they always decrement `SP` *by 64 to make room for all eight argument registers (64 bytes). One reason is that C provides a* `vararg` *option that allows a pointer to pick, say, the third argument to a procedure. When the compiler encounters the rare* `vararg`*, it copies the eight argument registers onto the stack into the eight reserved locations.*