Part 14: Generating ARM Assembly Code

In this part of our compiler writing journey, I've ported the compiler over to the ARM CPU on the Raspberry Pi 4.

I should preface this section by saying that, while I know MIPS assembly language quite well, I only knew a bit of x86-32 assembly language when I started this journey, and nothing about x86-64 nor ARM assembly language.

What I've been doing along the way is compiling example C programs down to an assembler with various C compilers to see what sort of assembly language they produce. That's what I've done here to write the ARM output for this compiler.

The Major Differences

Firstly, ARM is a RISC CPU and x86-64 is a CISC CPU. There are fewer addressing modes on the ARM when compared to the x86-64. There are also other interesting constraints that occur when generating ARM assembly code. So I will start with the major differences, and leave the main similarities to later.

ARM Registers

ARM has heaps more registers than x86-64. That said, I'm sticking with four registers to allocate: `r4`, `r5`, `r6` and `r7`. We will see that `r0` and `r3` get used for other things below.

Addressing Global Variables

On x86-64, we only have to declare a global variable with a line like:

```
        .comm   i,4,4        # int variable
        .comm   j,1,1        # char variable
```

and, later, we can load and store to these variables easily:

```
        movb    %r8b, j(%rip)    # Store to j
        movl    %r8d, i(%rip)    # Store to i
        movzbl  i(%rip), %r8     # Load from i
        movzbq  j(%rip), %r8     # Load from j
```

With ARM, we have to manually allocate space for all global variables in our program postamble:

```
        .comm   i,4,4
        .comm   j,1,1
...
.L2:
        .word i
        .word j
```

To access these, we need to load a register with the address of each variable, and load a second register from that address:

```
        ldr     r3, .L2+0
        ldr     r4, [r3]        # Load i
        ldr     r3, .L2+4
        ldr     r4, [r3]        # Load j
```

Stores to variables are similar:

```
        mov     r4, #20
        ldr     r3, .L2+4
        strb    r4, [r3]        # i= 20
        mov     r4, #10
        ldr     r3, .L2+0
        str     r4, [r3]        # j= 10
```

There is now this code in `cgpostamble()` to generate the table of .words:

```
  // Print out the global variables
  fprintf(Outfile, ".L2:\n");
  for (int i = 0; i < Globs; i++) {
    if (Gsym[i].stype == S_VARIABLE)
      fprintf(Outfile, "\t.word %s\n", Gsym[i].name);
  }
```

This also means that we need to determine the offset from `.L2` for each global variable. Following the KISS principle, I manually calculate the offset each time I want to load `r3` with the address of a variable. Yes, I should calculate each offset once and store it somewhere; later!

```
// Determine the offset of a variable from the .L2
// label. Yes, this is inefficient code.
static void set_var_offset(int id) {
  int offset = 0;
  // Walk the symbol table up to id.
  // Find S_VARIABLEs and add on 4 until
  // we get to our variable

  for (int i = 0; i < id; i++) {
    if (Gsym[i].stype == S_VARIABLE)
      offset += 4;
  }
  // Load r3 with this offset
  fprintf(Outfile, "\tldr\tr3, .L2+%d\n", offset);
}
```

Loading Int Literals

The size of an integer literal in a load instruction is limited to 11 bits and I think this is a signed value. Thus, we can't put large integer literals into a single instruction. That answer is to store the literal values in memory, like variables. So I keep a list of previously-used literal values. In the postamble, I output them following the `.L3` label. And, like variables, I walk this list to determine the offset of any literal from the `.L3` label:

```
// We have to store large integer literal values in memory.
// Keep a list of them which will be output in the postamble
#define MAXINTS 1024
int Intlist[MAXINTS];
static int Intslot = 0;
```

```
// Determine the offset of a large integer
// literal from the .L3 label. If the integer
// isn't in the list, add it.
static void set_int_offset(int val) {
  int offset = -1;

  // See if it is already there
  for (int i = 0; i < Intslot; i++) {
    if (Intlist[i] == val) {
      offset = 4 * i;
      break;
    }
  }

  // Not in the list, so add it
  if (offset == -1) {
    offset = 4 * Intslot;
    if (Intslot == MAXINTS)
      fatal("Out of int slots in set_int_offset()");
    Intlist[Intslot++] = val;
  }
  // Load r3 with this offset
  fprintf(Outfile, "\tldr\tr3, .L3+%d\n", offset);
}
```

The Function Preamble

I'm going to give you the function preamble, but I am not completely sure what each instruction does. Here it is for `int main(int x)`:

```
.text
.globl        main
.type         main, %function
main:         push  {fp, lr}          # Save the frame and stack pointers
              add   fp, sp, #4        # Add sp+4 to the stack pointer
              sub   sp, sp, #8        # Lower the stack pointer by 8
              str   r0, [fp, #-8]     # Save the argument as a local var?
```

and here's the function postamble to return a single value:

```
              sub   sp, fp, #4        # ???
              pop   {fp, pc}          # Pop the frame and stack pointers
```

Comparisons Returning 0 or 1

With the x86-64 there's an instruction to set a register to 0 or 1 based on the comparison being true, e.g. `sete`, but then we have to zero-fill the rest of the register with `movzbq`. With the ARM, we run two separate instructions which set a register to a value if the condition we want is true or false, e.g.

```
                moveq r4, #1            # Set r4 to 1 if values were equal
                movne r4, #0            # Set r4 to 0 if values were not
equal
```

A Comparison of Similar x86-64 and ARM Assembly Output

I think that's all the major differences out of the road. So below is a comparison of the `cgXXX()` operation, any specific type for that operation, and an example x86-64 and ARM instruction sequence to perform it.

Operation(type)	x86-64 Version	ARM Version
cgloadint()	movq $12, %r8	mov r4, #13
cgloadglob(char)	movzbq foo(%rip), %r8	ldr r3, .L2+#4
		ldr r4, [r3]
cgloadglob(int)	movzbl foo(%rip), %r8	ldr r3, .L2+#4
		ldr r4, [r3]
cgloadglob(long)	movq foo(%rip), %r8	ldr r3, .L2+#4
		ldr r4, [r3]
int cgadd()	addq %r8, %r9	add r4, r4, r5
int cgsub()	subq %r8, %r9	sub r4, r4, r5
int cgmul()	imulq %r8, %r9	mul r4, r4, r5
int cgdiv()	movq %r8,%rax	mov r0, r4
	cqo	mov r1, r5
	idivq %r8	bl __aeabi_idiv
	movq %rax,%r8	mov r4, r0

Operation(type)	x86-64 Version	ARM Version
cgprintint()	movq %r8, %rdi	mov r0, r4
	call printint	bl printint
		nop
cgcall()	movq %r8, %rdi	mov r0, r4
	call foo	bl foo
	movq %rax, %r8	mov r4, r0
cgstorglob(char)	movb %r8, foo(%rip)	ldr r3, .L2+#4
		strb r4, [r3]
cgstorglob(int)	movl %r8, foo(%rip)	ldr r3, .L2+#4
		str r4, [r3]
cgstorglob(long)	movq %r8, foo(%rip)	ldr r3, .L2+#4
		str r4, [r3]
cgcompare_and_set()	cmpq %r8, %r9	cmp r4, r5
	sete %r8	moveq r4, #1
	movzbq %r8, %r8	movne r4, #1
cgcompare_and_jump()	cmpq %r8, %r9	cmp r4, r5
	je L2	beq L2
cgreturn(char)	movzbl %r8, %eax	mov r0, r4
	jmp L2	b L2
cgreturn(int)	movl %r8, %eax	mov r0, r4
	jmp L2	b L2
cgreturn(long)	movq %r8, %rax	mov r0, r4
	jmp L2	b L2

Testing the ARM Code Generator

If you copy the compiler from this part of the journey to a Raspberry Pi 3 or 4, you should be able to do:

```
$ make armtest
cc -o comp1arm -g -Wall cg_arm.c decl.c expr.c gen.c main.c misc.c
      scan.c stmt.c sym.c tree.c types.c
cp comp1arm comp1
(cd tests; chmod +x runtests; ./runtests)
input01: OK
input02: OK
input03: OK
input04: OK
input05: OK
input06: OK
input07: OK
input08: OK
input09: OK
input10: OK
input11: OK
input12: OK
input13: OK
input14: OK

$ make armtest14
./comp1 tests/input14
cc -o out out.s lib/printint.c
./out
10
20
30
```

Conclusion and What's Next

It did take me a bit of head scratching to get the ARM version of the code generator `cg_arm.c` to correctly compile all of the test inputs. It was mostly straight-forward, I just wasn't familiar with the architecture and instruction set.

It should be relatively easy to port the compiler to a platform with 3 or 4 registers, 2 or so data sizes and a stack (and stack frames). As we go forward, I'll try to keep both `cg.c` and `cg_arm.c` functionally in sync.

In the next part of our compiler writing journey, we will add the `char` pointer to the language, as well as the '*' and '&' unary operators. Next step