Exploring AArch64 assembler – Chapter 9

Nov 5, 2017 • Roger Ferrer Ibáñez • AArch64

In chapter 6 we saw conditional branches and we ended commenting that they can be used to implement higher control constructs. In this chapter we will see a few of them.

If-statement

It may be at this point rather obvious how to implement an if-statement construct. Given a structure of the form

```
IF (E) THEN
  S1
ELSE
  S2
END IF
```

our code may look like

```
/* Evaluate condition E and update flags */
b<inverse-condition> else_label
  S1
  b end_if
else_label:
  S2
end_if:
```

It can be done the other way around.

```
/* Evaluate condition E and update flags */
b<condition> then_label
  S2
  b end_if
then_label:
  S1
end_if:
```

How to evaluate conditions?

Conditions in most programming languages are explicitly given a boolean type (e.g. `bool` in C++, `LOGICAL` in Fortran) or an integer type with a restricted set of values (e.g. 0 or 1 in C). Several programming languages implicitly define a conversion from regular integers to condition expressions. For instance in C, an expression `e` of type integer or pointer when used in a condition is the same as evaluating the condition `e != 0` (which will give a `0` or `1` value). C++ defines the same conversion but the resulting type is `bool` so it can be `true` or `false`. Fortran does not have any of these conversions so the operand will always be required.

For expressions of the form `e1` $\oplus$ `e2` where $\oplus$ is a relational operator like <, >, ≤, ≥, etc. We can use the instruction `cmp` followed by a `b.cond` with the relevant condition.

Evaluation of conditional expressions may be impacted by the language specification. Some programming languages implement conditionally evaluated logical operators (also called short-circuit evaluation). This makes these operators non-strict and closer to if-statements than the mathematical definition of the logical operators. Some other programming languages are strict in the definition of the logical operators, which means they evaluate all the operands and then apply the logical operation. C and C++ use short-circuit evaluation which means an expression like `e1 && e2` will only evaluate `e2` if `e1` is true. Similarly in an expression like `e1 || e2`, `e2` is evaluated only if `e1` is false (having to use "if" in the definition of these operators is what makes them "conditionally evaluated"). This means that in practice a C/C++ code like the following

```
if (e1 && e2)
  S1;
else
  S2
```

is implemented like

```
if (e1)
{
  if (e2)
    S1
  else
    S2
}
else
  S2
```

of course doing exactly the above would be ridiculous, so in practice looks more like this (in C/C++ again, though not very nice).

```
if (e1)
{
  if (e2)
     S1
  else
     goto else_block:
```

```
}
else
{
   else_block:
      S2;
}
```

We can simplify this further to make it closer to assembly (I'm adding a few redundant C/C++ labels below for clarity)

```
if_start:
  check_e1: if (e1) goto check_e2
  goto else_block;
  check_e2: if (e2) goto then_block;
  goto else_block;
then_block:
  S1;
  goto if_end;
else_block:
  S2;
if_end:
```

Do we always have to implement C/C++ logical operations like this? Not really, only if we're unsure that unconditionally evaluating the second operand may be different than conditionally evaluating it. For instance a case like

```
int a;
...
if ((5 < a) && (a < 25))
{
}
```

it won't harm to evaluate both operands so we can avoid a few branches and actually evaluate it as if the code had been (note a single `&` which means bitwise operation that always evaluates the two operands).

```
if ((5 < a) & (a < 25))
{
}
```

but in a case like

```
if (p != 0 && ((10 / p) > 5))
{
}
```

We don't really want to evaluate `((10 / p) > 5)` when `p == 0`. The same strategy is used for `||` in which sometimes we can evaluate it as `|`.

The following table shows some examples. Take these as suggestions, cleverer sequences can be used in some circumstances.

</tbody> </table>

Conditional assignment

A common idiom, in C/C++ is the following

```
if (a)
  x = e;
```

In Fortran there is a shorthand syntax for this case

```
IF (A) X = E;
```

Or more generally, in C/C++

```
if (a)
  x = e1;
else
  x = e2;
```

Which in C/C++ can be written more compactly as

```
x = a ? e1 : e2;
```

In many circumstances `e1` and `e2` are such that it is correct to rewrite the general case as

```
x = e2;
if (a)
  x = e1;
```

or the opposite form

```
x = e1;
if (!a)
  x = e2;
```

If we can write the code in one of the two forms above, and given that this case is relatively common, AArch64 has specific support for it.

The instruction `csel rdest, rsource1, rsource2, cond` sets the value of the destination register `rdest` to `rsource1` if the condition `cond` is true, otherwise it is set to `rsource2`. `csel` stands for conditional selection/

For instance, a typical `max` operation looks like this in C.

```
int m, a, b;
...
```

```
m = a > b ? a : b;
```

with csel we can implement the code above as

```
cmp x0, x1           // compare x0 and x1
csel x2, x0, x1, gt  // x2 ← if x0 > x1
                     //      then       x0
                     //      otherwise  x1
```

which is handy because it avoids a branch.

Loop constructs

To implement a loop construct of the form

```
while (E)
  S;
```

we can do

```
loop_check: if (!E) goto loop_end;
  S;
  goto loop_check;
loop_end:
```

Usually inverting the flow is a bit better

```
loop_start: goto loop_check;
loop_body:
  S;
loop_check: if (E) goto loop_body;
```

This has the added benefit that if we remove the initial branch to `loop_check`, this is the equivalent to the C `do-while` loop or the Pascal `repeat-until`.

While we can use regular branches and comparison instructions to implement loops, AArch64 provides a couple of specialized instructions for some common circumstances: `cbnz` and `cbz`. They conditionally branch to some label if a given register is non-zero or zero respectively. This is useful for loops like this.

```
i = ...;
while (i > 0)
{
 S;
 i--;
}
```

because we can now implement them as

```
                      // Assume "i" is in x0
    b loop_check
    loop_body:
      S
      sub x0, x0, #1     // x0 ← x0 - 1
    loop_check:
      cbnz x0, loop_body // branch to loop_body if x0 != 0
```

Similarly for `cbz`. That said, for this particular example it is possible to use subs instruction that acts like `sub` but also updates the condition flags. So we can write

```
// Assume "i" is in x0
add x0, x0, #1     // We need to compensate the subs below
b loop_check
loop_body:
  S
loop_check:
  subs x0, x0, #1  // x0 ← x0 - 1 and update condition flags
  b.ne loop_body   // branch to loop_body if x0 != 0
```

So `cbnz` and `cbz` can be more useful when a register being zero is a condition and its computation cannot update the flags (or doing that would require adding an extra `cmp`). We have not seen pointers yet, but a common scenario is traversing a linked list ended by a null pointer.

```
struct singly_linked_list {
  struct singly_linked_list *next;
  int some_data;
} *list;
...
it = list;
while (it != NULL)
{
  it->some_data++;
  it = it->next;
}
```

Because a null pointer is represented usually with all zeros after `it = it->next` we can use `cbnzr` without having to do `cmp` + `b.ne`.

```
// Assume x0 is "list" and x1 is "it"
mov x1, x0
b loop_check:
loop_body:
  ldr w2, [x1, #8]    // w2 ← *(x1 + 8) [32-bit load]
                      // this loads it->some_data into w2
  add w2, w2, #1      // w2 ← w2 + 1
  str w2, [x1, #8]    // *(x1 + 8) ← w2 [32-bit store]
                      // this updates it->some_data
  ldr x1, [x1]        // x1 ← *x1
                      // this loads it->next into x1
loop_check:
```

```
  cbnz x1, loop_body  // branch to loop_body if x1 != NULL
loop_end:
```

Fibonacci (using a loop)

In chapter 8 we made a function that recursively computed the Fibonacci numbers. Today we are going to see how to compute them using a loop. We will reuse the same main function and only change the `fibonacci` function itself.

As a reminder of the fibonacci function is actually a sequence of numbers F_i defined by the following recurrence:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$, where $n > 1$

To compute the current element we only need the previous two elements, we don't really need to strictly follow the mathematical definition doing a recursive function call, like we did in chapter 8. Results for `fibonacci(0)` and `fibonacci(1)` will be immediately solved, no need to loop. Any other fibonacci(n) can be computed by repeatedly computing `n - 1` previous Fibonacci numbers.

Condition to evaluate (C/C++)	**Instruction sequence**

Condition to evaluate (C/C++)	**Instruction sequence**

`e1 < e2`	Direct translation
	
	`cmp e1, e2`
	`b.lt true_case`
	
	If we want to evaluate `!(e1 < e2)`. This is the same as `e1 >= e2` when `false_case` is the `true_case`.
	
	`cmp e1, e2`
	`b.ge false_case`
	
	If we can reverse the operands `e2 > e1`
	
	`cmp e2, e1`
	`b.gt true_case`
	
	If we can reverse the operands `!(e2 > e1)`. This is the same as `e1 <= e2` when `false_case` is the `true_case`.
	
	`cmp e2, e1`
	`b.le false_case`
	
	</td> </tr>

Condition to evaluate (C/C++)	Instruction sequence

`e1 && e2`	Direct translation
	
	`cmp e1, #0`
	`b.ne check_e2`
	`check_e2:`
	`  cmp e2, #0`
	`  b.ne false_case`
	`true_case:`
	`  ...`
	
	Without short-cut evaluation (i.e. `e1 & e2`)
	
	`tst e1, e2  // update flags with the`
	`  // result of bitwise-and(e1, e2)`
	`b.eq false_case`
	`true_case:`
	`  ...`
	
	The `tst` instruction computes the bitwise and of the two register operands, updates the condition flags based on the result of that operation. The result itself, though, is discarded.

`e1 || e2`	`cmp e1, #0`
	`b.eq check_e2`
	`check_e2:`
	`  cmp e2, #0`
	`  b.eq false_case`
	`true_case:`
	`  ...`
	
	Without short-cut evaluation (i.e. `e1 | e2`)
	
	`orr x1, e1, e2 // x1 = bitwise-or(e1, e2)`
	`cmp x1, #0`
	`b.eq false_case:`
	`true_case:`
	`  ...`
	
	</td> </tr>

Condition to evaluate (C/C++)	Instruction sequence

`!e`	For the less common case when `e` is already a boolean, just compare it with 0.
	
	`cmp e1, #0`
	`b.ne false_case`
	`true_case:`
	`  ...`
	
	But usually `!` can be implemented by just evaluating the opposite condition. For instance `!(e1 < e2)` can be evaluated as `e1 >= e2`.
	
	In some other cases like !(e1 && e2) we can use De Morgan rules to evaluate it like `!e1 || !e2`. This is mostly useful for cases like `!((e11 < e12) && (e21 < e22))` where propagating the ! inside the expression is beneficial as this just becomes `(e11 >= e12) || (e21 >= e22)`

```
61 fibonacci:
62     // fibonacci(n) -> result
63     //   n is 32-bit and will be passed in w0
64     //   result is 64-bit and will be returned in x0
65     mov w3, w0          // w3 ← w0
66     cmp w3, #1          // compare w3 and 1 and update the flags
67     b.le simple_case    // jump to simple_case if w3 <= 1
68     sub w3, w3, #1      // w3 ← w3 - 1
69     mov x1, #0          // x1 ← 0
70     mov x2, #1          // x1 ← 1
71     b loop_check        // branch to loop_check
72 loop_body:
73     add x0, x1, x2      // x0 ← x1 + x2
74     mov x1, x2          // x1 ← x2
75     mov x2, x0          // x2 ← x0
76     sub w3, w3, #1      // w3 ← w3 - 1
77 loop_check:
78     cbnz w3, loop_body  // branch to loop_body if w3 != 0
79
80     b fibonacci_end     // branch to fibonacci_end
81 simple_case:
82     sxtw x0, w0         // x0 ← ExtendSigned32To64(w0)
```

```
83
84 fibonacci_end:
85     ret
```

As in the recursive version of this function, cases 0 and 1 are handled specially. We first keep w0 into w3, line 65, and then we compare w3 with 1, line 66. If the flags are such that `w3 ≤ 1` then we branch to the `simple_case` which the only thing it does is extending `w0` into `x0`, line 82. If the fibonacci number to compute is not 0 or 1, then we have to compute the `n - 1` earlier ones. So we have to decrease `w3` by 1, line 68. In register `x1` we will have `fibonacci(n-2)` and in x2 we will have fibonacci(n-1). The two initially known ones are 0 and 1 respectively (this is, for `fibonacci(2)`), lines 69-70.

Now everything is set for the loop, so we jump to the check, line 71. The loop check uses `cbnz` with `w3`, line 78. Once `w3` becomes zero means that we do not have to compute any other fibonacci number, so cbnz will branch back to `loop_body` only when `w3` is not 0.

The loop body simply computes the current fibonacci by adding `x1` and `x2` (recall, x1 is `fibonacci(n-2)` and x2 is `fibonacci(n-1)`) and leaving the result in `x0`, line 73. Now we need to prepare the next iteration of the loop: `x2` (which is `fibonacci(n-1)`) will be `fibonacci(n-2)` in the next iteration, so we put it in `x1`, line 74. Similarly, `x0` (which is `fibonacci(n)`) will be `fibonacci(n-1)` in the next iteration, so we put it `x2`, line 75.

At this point one fibonacci number has been computed, so one less to compute. So we decrease `w3` by one, line 76. Now the loop_check is executed which will iterate if needed. Once we leave the loop, we still need to skip the `simple_case`, line 80.

To test this, just take the code in chapter 8 and replace the recursive function with the iterative function.

```
$ ./fibo
Please type a number: 12↵
Fibonacci number 12 is 144
$ ./fibo
Please type a number: 25↵
Fibonacci number 25 is 75025
$ ./fibo
Please type a number: 40↵
Fibonacci number 40 is 102334155
```

Yay! :)

This is all for today.

« Walk-through flang – Part 5 A very simple memory pool in C++11 »

Powered by Jekyll. Theme based on whiteglass
Subscribe via RSS