ARM assembler in Raspberry Pi – Chapter 12

March 28, 2013 rferrer, 0

We saw in chapter 6 some simple schemes to implement usual structured programming constructs like if-then-else and loops. In this chapter we will revisit these constructs and exploit a feature of the ARM instruction set that we have not learnt yet.

Playing with loops

The most generic form of loop is this one.

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

There are also two special forms, which are actually particular incarnations of the one shown above but are interesting as well.

```
for (i = lower; i <= upper; i += step)
  S;
```

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

Some languages, like Pascal, have constructs like this one.

```
repeat
  S
until E;
```

but this is like a `do S while (!E)`.

We can manipulate loops to get a form that may be more convenient. For instance.

```
   do
     S
   while (E);

/* Can be rewritten as */

   S;
   while (E)
      S;
```

```
   while (E)
     S;

/* Can be rewritten as */

   if (E)
   {
      S;
      do
        S
      while (E);
   }
```

The last manipulation is interesting, because we can avoid the `if-then` if we directly go to the `while` part.

```
/* This is not valid C */
goto check;
do
  S
```

Calendar

March 2013

M	T	W	T	F	S	S
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
« Feb				Apr »		

Recent Posts

Capybara, pop up windows and the new PayPal sandbox
ARM assembler in Raspberry Pi – Chapter 12
ARM assembler in Raspberry Pi – Chapter 11
ARM assembler in Raspberry Pi – Chapter 10
ARM assembler in Raspberry Pi – Chapter 9

Recent Comments

rferrer on ARM assembler in Raspberry Pi – Chapter 11
Einstieg in Pi-Assembler | ultramachine on ARM assembler in Raspberry Pi – Chapter 1
Fernando on ARM assembler in Raspberry Pi – Chapter 7
Loren Blaney on ARM assembler in Raspberry Pi – Chapter 11
ห.ร.ม. ด้วยภาษา Assembly บน Raspberry Pi | Raspberry Pi Thailand on ARM assembler in Raspberry Pi – Chapter 9

Tags

.net activerecord ajax apple archlinux arm assembler bind branches C# dhcp firebug firefox function function call functions gadgets html indexing modes ipod Java javascript jquery linux mac os mac os x MVC networking parallels pi programming tips rails raspberry ruby ruby on rails security software sports sql server subversion tips and tricks tools ubuntu visual studio xmonad

```
check: while (E);
```

In valid C, the above transformation would be written as follows.

```
goto check;
loop:
  S;
check:
  if (E) goto loop;
```

Which looks much uglier than abusing a bit C syntax.

The -s suffix

So far, when checking the condition of an `if` or `while`, we have evaluated the condition and then used the `cmp` intruction to update `cpsr`. The update of the `cpsr` is mandatory for our conditional codes, no matter if we use branching or predication. But `cmp` is not the only way to update `cpsr`. In fact many instructions can update it.

By default an instruction does not update `cpsr` unless we append the suffix `-s`. So instead of the instruction `add` or `sub` we write `adds` or `subs`. The result of the instruction (what would be stored in the destination register) is used to update `cpsr`.

How can we use this? Well, consider this simple loop counting backwards.

```
/* for (int i = 100 ; i >= 0; i--) */
mov r1, #100
loop:
  /* do something */
  sub r1, r1, #1      /* r1 ← r1 - 1 */
  cmp r1, #0          /* update cpsr with r1 - 0 */
  bge loop            /* branch if r1 >= 100 */
```

If we replace `sub` by `subs` then `cpsr` will be updated with the result of the substration. This means that the flags N, Z, C and V will be updated, so we can use a branch right after `subs`. In our case we want to jump back to loop only if `i >= 0`, this is when the result is non-negative. We can use `bpl` to achieve this.

```
/* for (int i = 100 ; i >= 0; i--) */
mov r1, #100
loop:
  /* do something */
  subs r1, r1, #1      /* r1 ← r1 - 1  and update cpsr with the final r1 */
  bpl loop             /* branch if the previous sub computed a positive number (N flag i
```

It is a bit tricky to get these things right (this is why we use compilers). For instance this similar, but not identical, loop would use `bne` instead of `bpl`. Here the condition is `ne` (not equal). It would be nice to have an alias like `nz` (not zero) but, unfortunately, this does not exist in ARM.

```
/* for (int i = 100 ; i > 0; i--). Note here i > 0, not i >= 0 as in the example above */
mov r1, #100
loop:
  /* do something */
  subs r1, r1, #1      /* r1 ← r1 - 1  and update cpsr with the final r1 */
  bne loop             /* branch if the previous sub computed a number that is not zero (
```

A rule of thumb where we may want to apply the use of the -s suffix is in codes in the following form.

```
s = ...
if (s @ 0)
```

where @ means any comparison respect 0 (equals, different, lower, etc.).

Archives

- April 2013
- March 2013
- February 2013
- January 2013
- December 2012
- November 2012
- August 2012
- July 2012
- June 2012
- February 2012
- January 2012
- December 2011
- November 2011
- October 2011
- July 2011
- June 2011
- May 2011
- April 2011
- March 2011
- February 2011
- December 2010
- November 2010
- October 2009
- July 2009
- June 2009
- March 2009
- November 2008
- July 2008
- September 2007
- July 2007
- June 2007

Operating 64-bit numbers

As an example of using the suffix -s we will implement three 64-bit integer operations in ARM: addition, substraction and multiplication. Remember that ARM is a 32-bit architecture, so everything is 32-bit minded. If we only use 32-bit numbers, this is not a problem, but if for some reason we need 64-bit numbers things get a bit more complicated. We will represent a 64-bit number as two 32-bit numbers, the lower and higher part. This way a 64-bit number n represented using two 32-bit parts, n_{lower} and n_{higher} will have the value $n = 2^{32} \times n_{higher} + n_{lower}$

We will, obviously, need to kep the 32-bit somewhere. When keeping them in registers, we will use two consecutive registers (e.g. r1 and r2, that we will write it as `{r1,r2}`) and we will keep the higher part in the higher numbered register. When keeping a 64-bit number in memory, we will store in two consecutive addresses the two parts, being the lower one in the lower address. The address will be 8-byte aligned.

Addition

Adding two 64-bit numbers using 32-bit operands means adding first the lower part and then adding the higher parts but taking into account a possible carry from the lower part. With our current knowledge we could write something like this (assume the first number is in `{r2,r3}`, the second in `{r4,r5}` and the result will be in `{r0,r1}`).

```
add r1, r3, r5       /* First we add the higher part */
                     /* r1 ← r3 + r5 */
adds r0, r2, r4      /* Now we add the lower part and we update cpsr */
                     /* r0 ← r2 + r4 */
addcs r1, r1, #1     /* If adding the lower part caused carry, add 1 to the higher part */
                     /* if C = 1 then r1 ← r1 + 1 */
                     /* Note that here the suffix -s is not applied, -cs means carry set *
```

This would work. Fortunately ARM provides an instructions `adc` which adds two numbers and the carry flag. So we could rewrite the above code with just two instructions.

```
adds r0, r2, r4      /* First add the lower part and update cpsr */
                     /* r0 ← r2 + r4 */
adc r1, r3, r5       /* Now add the higher part plus the carry from the lower one */
                     /* r1 ← r3 + r5 + C */
```

Substraction

Substracting two numbers is similar to adding them. In ARM when substracting two numbers using `subs`, if we need to borrow (because the second operand is larger than the first) then C will be disabled (C will be 0). If we do not need to borrow, C will be enabled (C will be 1). This is a bit surprising but consistent with the remainder of the architecture (check in chapter 5 conditions CS/HS and CC/LO). Similar to `adc` there is a `sbc` which performs a normal substraction if C is 1. Otherwise it substracts one more element. Again, this is consistent on how C works in the `subs` instruction.

```
subs r0, r2, r4      /* First add the lower part and update cpsr */
                     /* r0 ← r2 - r4 */
sbc r1, r3, r5       /* Now add the higher part plus the NOT of the carry from the lower c
                     /* r1 ← r3 - r5 - ~C */
```

Multiplication

Multiplying two 64-bit numbers is a tricky thing. When we multiply two N-bit numbers the result may need up to 2*N-bits. So when multiplying two 64-bit numbers we may need a 128-bit number. For the sake of simplicity we will assume that this does not happen and 64-bit will be enough. Our 64-bit numbers are two 32-bit integers, so a 64-bit x is actually $x = 2^{32} \times x_1 + x_0$, where x_1 and x_0 are two 32-bit numbers. Similarly another 64-bit number y would be $y = 2^{32} \times y_1 + y_0$.

Multiplying x and y yields z where $z = 2^{64} \times x_1 \times y_1 + 2^{32} \times (x_0 \times y_1 + x_1 \times y_0) + x_0 \times y_0$. Well, now our problem is multiplying each x_i by y_i, but again we may need 64-bit to represent the value.

ARM provides a bunch of different instructions for multiplication. Today we will see just three of them. If we are multiplying 32-bits and we do not care about the result not fitting in a 32-bit number we can use `mul Rd, Rsource1, Rsource2`. Unfortunately it does not set any flag in the `cpsr` useful for detecting an overflow of the multiplication (i.e. when the result does not fit in the 32-bit range). This instruction is the fastest one of the three. If we do want the 64-bit resulting from the multiplication, we have two other instructions `smull` and `umull`. The former is used when we multiply to numbers in two's complement, the latter when we represent unsigned values. Their syntax is `{s,u}mull RdestLower, RdestHigher, Rsource1, Rsource2`. The lower part of the 64-bit result is kept in the register `RdestLower` and the higher part in he register `RdestHigher`.

In this example we have to use `umull` otherwise the 32-bit lower parts might end being interpreted as negative numbers, giving negative intermediate values. That said, we can now multiply x_0 and y_0. Recall that we have the two 64-bit numbers in `r2,r3` and `r4,r5` pairs of registers. So first multiply `r2` and `r4`. Note the usage of `r0` since this will be its final value. In contrast, register `r6` will be used later.

```
umull r0, r6, r2, r4
```

Now let's multiply x_0 by y_1 and x_1 by y_0. This is `r3` by `r4` and `r2` by `r5`. Note how we overwrite `r4` and `r5` in the second multiplication. This is fine since we will not need them anymore.

```
umull r7, r8, r3, r4
umull r4, r5, r2, r5
```

There is no need to make the multiplication of x_1 by y_1 because if it gives a nonzero value, it will always overflow a 64-bit number. This means that if both `r3` and `r5` were nonzero, the multiplication will never fit a 64-bit. This is a suficient condition, but not a necessary one. The number might overflow when adding the intermediate values that will result in `r1`.

```
adds r2, r7, r4
adc r1, r2, r6
```

Let's package this code in a nice function in a program to see if it works. We will multiply numbers 12345678901 (this is 2×2^{32} + 3755744309) and 12345678 and print the result.

```
/* -- mult64.s */
.data

.align 4
message : .asciz "Multiplication of %lld by %lld is %lld\n"

.align 8
number_a_low: .word 3755744309
number_a_high: .word 2

.align 8
number_b_low: .word 12345678
number_b_high: .word 0

.text

/* Note: This is not the most efficient way to doa 64-bit multiplication.
   This is for illustration purposes */
mult64:
   /* The argument will be passed in r0, r1 and r2, r3 and returned in r0, r1 */
   /* Keep the registers that we are going to write */
   push {r4, r5, r6, r7, r8, lr}
   /* For covenience, move {r0,r1} into {r4,r5} */
   mov r4, r0    /* r0 ← r4 */
   mov r5, r1    /* r5 ← r1 */

   umull r0, r6, r2, r4    /* {r0,r6} ← r2 * r4 */
   umull r7, r8, r3, r4    /* {r7,r8} ← r3 * r4 */
   umull r4, r5, r2, r5    /* {r4,r5} ← r2 * r5 */
   adds r2, r7, r4         /* r2 ← r7 + r4 and update cpsr */
```

```
31    adc r1, r2, r6          /* r1 ← r2 + r6 + C */
32
33    /* Restore registers */
34    pop {r4, r5, r6, r7, r8, lr}
35    bx lr                   /* Leave mult64 */
36
37 .global main
38 main:
39     push {r4, r5, r6, r7, r8, lr}       /* Keep the registers we are going to modify
40                                         /* r8 is not actually used here, but this way
41                                            the stack is already 8-byte aligned */
42     /* Load the numbers from memory */
43     /* {r4,r5} ← a */
44     ldr r4, addr_number_a_low       /* r4 ← &a_low */
45     ldr r4, [r4]                    /* r4 ← *r4 */
46     ldr r5, addr_number_a_high      /* r5 ← &a_high  */
47     ldr r5, [r5]                    /* r5 ← *r5 */
48
49     /* {r6,r7} ← b */
50     ldr r6, addr_number_b_low       /* r6 ← &b_low  */
51     ldr r6, [r6]                    /* r6 ← *r6 */
52     ldr r7, addr_number_b_high      /* r7 ← &b_high  */
53     ldr r7, [r7]                    /* r7 ← *r7 */
54
55     /* Now prepare the call to mult64
56     /*
57        The first number is passed in
58        registers {r0,r1} and the second one in {r2,r3}
59     */
60     mov r0, r4                  /* r0 ← r4 */
61     mov r1, r5                  /* r1 ← r5 */
62
63     mov r2, r6                  /* r2 ← r6 */
64     mov r3, r7                  /* r3 ← r7 */
65
66     bl mult64                  /* call mult64 function */
67     /* The result of the multiplication is in r0,r1 */
68
69     /* Now prepare the call to printf */
70     /* We have to pass &message, {r4,r5}, {r6,r7} and {r0,r1} */
71     push {r1}                   /* Push r1 onto the stack. 4th (higher) parameter */
72     push {r0}                   /* Push r0 onto the stack. 4th (lower) parameter */
73     push {r7}                   /* Push r7 onto the stack. 3rd (higher) parameter */
74     push {r6}                   /* Push r6 onto the stack. 3rd (lower) parameter */
75     mov r3, r5                  /* r3 ← r5.                 2rd (higher) parameter */
76     mov r2, r4                  /* r2 ← r4.                 2nd (lower) parameter */
77     ldr r0, addr_of_message     /* r0 ← &message            1st parameter */
78     bl printf                   /* Call printf */
79     add sp, sp, #16             /* sp ← sp + 16 */
80                                 /* Pop the two registers we pushed above */
81
82     mov r0, #0                  /* r0 ← 0 */
83     pop {r4, r5, r6, r7, r8, lr}        /* Restore the registers we kept */
84     bx lr                       /* Leave main */
85
86 addr_of_message : .word message
87 addr_number_a_low: .word number_a_low
88 addr_number_a_high: .word number_a_high
89 addr_number_b_low: .word number_b_low
90 addr_number_b_high: .word number_b_high
```

Observe first that we have the addresses of the lower and upper part of each number. Instead of this we could load them by just using an offset, as we saw in chapter 8. So, in lines 41 to 44 we could have done the following.

```
40     /* {r4,r5} ← a */
41     ldr r4, addr_number_a_low       /* r4 ← &a_low */
42     ldr r5, [r4, +#4]               /* r5 ← *(r4 + 4) */
43     ldr r4, [r4]                    /* r4 ← *r4  */
```

In the function `mult64` we pass the first value (x) as `r0`, `r1` and the second value (y) as `r2`, `r3`. The result is stored in `r0`, `r1`. We move the values to the appropiate registers for parameter passing in lines 57 to 61.

Printing the result is a bit complicated. 64-bits must be passed as pairs of consecutive registers where the lower part is in an even numbered register. Since we pass the address of the message in `r0` we cannot pass the first 64-bit integer in `r1`. So we skip `r1` and we use `r2` and `r3` for the first argument. But now we have run out of registers for parameter passing. When this happens, we have to use the stack for parameter passing.

Two rules have to be taken into account when passing data in the stack.

1. You must ensure that the stack is aligned for the data you are going to pass (by adjusting the stack first). So, for 64-bit numbers, the stack must be 8-byte aligned. If you pass an 32-bit number and then a 64-bit number, you will have to skip 4 bytes before passing the 64-bit number. Do not forget to keep the stack always 8-byte aligned per the Procedure Call Standard for ARM Architecture (AAPCS) requirement.
2. An argument with a lower position number in the call must have a lower address in the stack. So we have to pass the arguments in opposite order.

The second rule is what explains why we push first `r1` and then `r0`, when they are the registers containing the last 64-bit number (the result of the multiplication) we want to pass to `printf`.

Note that in the example above, we cannot pass the parameters in the stack using `push {r0,r1,r6,r7}`, which is equivalent to `push {r0}`, `push {r1}`, `push {r6}` and `push {r7}`, but not equivalent to the required order when passing the arguments on the stack.

If we run the program we should see something like.

```
$ ./mult64_2
Multiplication of 12345678901 by 12345678 is 152415776403139878
```

That's all for today.

arm, assembler, pi, raspberry

ARM assembler in Raspberry Pi – Chapter 11

Capybara, pop up windows and the new PayPal sandbox