ARM assembler in Raspberry Pi – Chapter 6

January 20, 2013 rferrer, 5

Control structures

In the previous chapter we learnt branch instructions. They are really powerful tools because they allow us to express control structures. *Structured programming* is an important milestone in better computing engineering (a foundational one, but nonetheless an important one). So being able to map usual structured programming constructs in assembler, in our processor, is a Good Thing™.

If, then, else

Well, this one is a basic one, and in fact we already used this structure in the previous chapter. Consider the following structure, where E is an expression and S1 and S2 are statements (they may be compound statements like `{ SA; SB; SC; }`)

```
if (E) then
   S1
else
   S2
```

A possible way to express this in ARM assembler could be the following

```
if_eval:
    /* Assembler that evaluates E and updates the cpsr accordingly */
bXX else /* Here XX is the appropiate condition */
then_part:
   /* assembler for S1, the "then" part */
   b end_of_if
else:
   /* assembler for S2, the "else" part */
end_of_if:
```

If there is no else part, we can replace `bXX else` with `bXX end_of_if`.

Loops

This is another usual one in structured programming. While there are several types of loops, actually all reduce to the following structure.

```
while (E)
  S
```

Supposedly S makes something so E eventually becomes false and the loop is left. Otherwise we would stay in the loop forever (sometimes this is what you want but not in our examples). A way to implement these loops is as follows.

```
while_condition : /* assembler to evaluate E and update cpsr */
  bXX end_of_loop  /* If E is false, then leave the loop right now */
  /* assembler of S */
  b while_condition /* Unconditional branch to the beginning */
end_of_loop:
```

A common loop involves iterating from a single range of integers, like in

```
for (i = L; i < N; i += K)
```


Calendar

January 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			
« Dec				Feb »		

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

```
S
```

But this is nothing but

```
i = L;
while (i < N)
{
   S;
   i += K;
}
```

So we do not have to learn a new way to implement the loop itself.

1 + 2 + 3 + 4 + … + 22

As a first example lets sum all the numbers from 1 to 22 (I'll tell you later why I chose 22). The result of the sum is `253` (check it with a calculator). I know it makes little sense to compute something the result of which we know already, but this is just an example.

```
1  /* -- loop01.s */
2  .text
3  .global main
4  main:
5      mov r1, #0        /* r1 ← 0 */
6      mov r2, #1        /* r2 ← 1 */
7  loop:
8      cmp r2, #22       /* compare r2 and 22 */
9      bgt end           /* branch if r2 > 22 to end */
10     add r1, r1, r2    /* r1 ← r1 + r1 */
11     add r2, r2, #1    /* r2 ← r2 + 1 */
12     b loop
13 end:
14     mov r0, r1        /* r0 ← r1 */
15     bx lr
```

Here we are counting from 1 to 22. We will use the register `r2` as the counter. As you can see in line 6 we initialize it to 1. The sum will be accumulated in the register `r1`, at the end of the program we move the contents of `r1` into `r0` to return the result of the sum as the error code of the program (we could have used `r0` in all the code and avoid this final `mov` but I think it is clearer this way).

In line 8 we compare `r2` (remember, the counter that will go from 1 to 22) to 22. This will update the `cpsr` thus in line 9 we can check if the comparison was such that r2 was greater than 22. If this is the case, we end the loop by branching to `end`. Otherwise we add the current value of `r2` to the current value of `r1` (remember, in `r1` we accumulate the sum from 1 to 22).

Line 11 is an important one. We increase the value of `r2`, because we are counting from 1 to 22 and we already added the current counter value in `r2` to the result of the sum in `r1`. Then at line 12 we branch back at the beginning of the loop. Note that if line 11 was not there we would hang as the comparison in line 8 would always be false and we would never leave the loop in line 9!

```
$ ./loop01; echo $?
253
```

Well, now you could change the line 8 and try with let's say, #100. The result should be 5050.

```
$ ./loop01; echo $?
186
```

What happened? Well, it happens that in Linux the error code of a program is a number from 0 to 255 (8 bits). If the result is 5050, only the lower 8 bits of the number are used. 5050 in binary is `1001110111010`, its lower 8 bits are `10111010` which is exactly 186. How can we check the computed `r1` is 5050 before ending the program? Let's use GDB.

```
$ gdb loop
...
```


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

```
(gdb) start
Temporary breakpoint 1 at 0x8390
Starting program: /home/roger/asm/chapter06/loop01

Temporary breakpoint 1, 0x00008390 in main ()
(gdb) disas main,+(9*4)
Dump of assembler code from 0x8390 to 0x83b4:
   0x00008390 <main+0>: mov     r1, #0
   0x00008394 <main+4>: mov     r2, #1
   0x00008398 <loop+0>: cmp     r2, #100        ; 0x64
   0x0000839c <loop+4>: bgt     0x83ac <end>
   0x000083a0 <loop+8>: add     r1, r1, r2
   0x000083a4 <loop+12>:        add     r2, r2, #1
   0x000083a8 <loop+16>:        b       0x8398 <loop>
   0x000083ac <end+0>:  mov     r0, r1
   0x000083b0 <end+4>:  bx      lr
End of assembler dump.
```

Let's tell gdb to stop at `0x000083ac`, right before executing `mov r0, r1`.

```
(gdb) break *0x000083ac
(gdb) cont
Continuing.

Breakpoint 2, 0x000083ac in end ()
(gdb) disas
Dump of assembler code for function end:
=> 0x000083ac <+0>:     mov     r0, r1
   0x000083b0 <+4>:     bx      lr
End of assembler dump.
(gdb) info register r1
r1             0x13ba   5050
```

Great, this is what we expected but we could not see due to limits in the error code.

Maybe you have noticed that something odd happens with our labels being identified as functions. We will address this issue in a future chapter, this is mostly harmless though.

3n + 1

Let's make another example a bit more complicated. This is the famous *3n + 1* problem also known as the Collatz conjecture. Given a number n we will divide it by 2 if it is even and multiply it by 3 and add one if it is odd.

```
if (n % 2 == 0)
  n = n / 2;
else
  n = 3*n + 1;
```

Before continuing, our ARM processor is able to multiply two numbers but we should learn a new instruction `mul` which would detour us a bit. Instead we will use the following identity `3 * n = 2*n + n`. We do not really know how to multiply or divide by two yet, we will study this in a future chapter, so for now just assume it works as shown in the assembler below.

Collatz conjecture states that, for any number n, repeatedly applying this procedure will eventually give us the number 1. Theoretically it could happen that this is not the case. So far, no such number has been found, but it has not been proved otherwise. If we want to repeatedly apply the previous procedure, our program is doing something like this.

```
n = ...;
while (n != 1)
{
  if (n % 2 == 0)
     n = n / 2;
  else
     n = 3*n + 1;
}
```

If the Collatz conjecture were false, there would exist some n for which the code above would hang, never reaching 1. But as I said, no such number has been found.

```
1  /* -- collatz.s */
2  .text
3  .global main
4  main:
5      mov r1, #123            /* r1 ← 123 */
6      mov r2, #0              /* r2 ← 0 */
7  loop:
8      cmp r1, #1              /* compare r1 and 1 */
9      beq end                 /* branch to end if r1 == 1 */
10
11     and r3, r1, #1          /* r3 ← r1 & 1 */
12     cmp r3, #0              /* compare r3 and 0 */
13     bne odd                 /* branch to odd if r3 != 0 */
14 even:
15     mov r1, r1, ASR #1      /* r1 ← (r1 >> 1) */
16     b end_loop
17 odd:
18     add r1, r1, r1, LSL #1 /* r1 ← r1 + (r1 << 1) */
19     add r1, r1, #1          /* r1 ← r1 + 1 */
20
21 end_loop:
22     add r2, r2, #1          /* r2 ← r2 + 1 */
23     b loop                  /* branch to loop */
24
25 end:
26     mov r0, r2
27     bx lr
```

In `r1` we will keep the number n. In this case we will use the number 123. 123 reaches 1 in 46 steps: [123, 370, 185, 556, 278, 139, 418, 209, 628, 314, 157, 472, 236, 118, 59, 178, 89, 268, 134, 67, 202, 101, 304, 152, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]. We will count the number of steps in register `r2`. So we initialize `r1` with 123 and `r2` with 0 (no step has been performed yet).

At the beginning of the loop, in lines 8 and 9, we check if `r1` is 1. So we compare it with 1 and if it is equal we leave the loop branching to `end`.

Now we know that `r1` is not 1, so we proceed to check if it is even or odd. To do this we use a new instruction `and` which performs a *bitwise and operation*. An even number will have the least significant bit (LSB) to 0, while an odd number will have the LSB to 1. So a bitwise and using 1 will return 0 or 1 on even or odd numbers, respectively. In line 11 we keep the result of the bitwise and in `r3` register and then, in line 12, we compare it against 0. If it is not zero then we branch to `odd`, otherwise we continue on the `even` case.

Now some magic happens in line 15. This is a combined operation that ARM allows us to do. This is a `mov` but we do not move the value of `r1` directly to `r1` (which would be doing nothing) but first we do an *arithmetic shift right* (ASR) to the value of `r1` (to the value, no the register itself). Then this shifted value is moved to the register `r1`. An *arithmetic shift right* shifts all the bits of a register to the right: the rightmost bit is effectively discarded and the leftmost is set to the same value as the leftmost bit prior the shift. Shifting right one bit to a number is the same as dividing that number by 2. So this `mov r1, r1, ASR #1` is actually doing `r1 ← r1 / 2`.

Some similar magic happens for the even case in line 18. In this case we are doing an `add`. The first and second operands must be registers (destination operand and the first source operand). The third is combined with a *logical shift left* (LSL). The value of the operand is shifted left 1 bit: the leftmost bit is discarded and the rightmost bit is set to 0. This is effectively multiplying the value by 2. So we are adding `r1` (which keeps the value of n) to `2*r1`. This is `3*r1`, so `3*n`. We keep this value in `r1` again. In line 19 we add 1 to that value, so `r1` ends having the value `3*n+1` that we wanted.

Do not worry very much now about these LSL and ASR. Just take them for granted now. In a future chapter we will see them in more detail.

Finally, at the end of the loop, in line 22 we update `r2` (remember it keeps the counter of our steps) and then we branch back to the beginning of the loop. Before ending the program we move the counter to `r0` so we return the number of steps we did to reach 1.

```
$ ./collatz; echo $?
46
```

Great.

That's all for today.

Postscript

Kevin Millikin rightly pointed (in a comment below) that usually a loop is not implemented in the way shown above. In fact Kevin says that a better way to do the loop of `loop01.s` is as follows.

```
 1 /* -- loop02.s */
 2 .text
 3 .global main
 4 main:
 5     mov r1, #0        /* r1 ← 0 */
 6     mov r2, #1        /* r2 ← 1 */
 7     b check_loop      /* unconditionally jump at the end of the loop */
 8 loop:
 9     add r1, r1, r2    /* r1 ← r1 + r1 */
10     add r2, r2, #1    /* r2 ← r2 + 1 */
11 check_loop:
12     cmp r2, #22       /* compare r2 and 22 */
13     ble loop          /* branch if r2 <= 22 to the beginning of the loop */
14 end:
15     mov r0, r1        /* r0 ← r1 */
16     bx lr
```

If you count the number of instruction in the two codes, there are 9 instructions in both. But if you look carefully in Kevin's proposal you will see that by unconditionally branching to the end of the loop, and reversing the condition check, we can skip one branch thus reducing the number of instructions of the loop itself from 5 to 4.

There is another advantage in this second version, though: there is only one branch in the loop itself as we resort to *implicit sequencing* to reach again the two instructions performing the check. For reasons beyond the scope of this post, the execution of a branch instruction may negatively affect the performance of our programs. Processors have mechanisms to mitigate the performance loss due to branches (and in fact the processor in the Raspberry Pi does have them). But avoiding a branch instruction entirely avoids the potential performance penalization of executing a branch instruction.

While we do not care very much now about the performance of our assembler. However, I thought it was worth developing a bit more Kevin's comment.

arm, assembler, control structures, pi, raspberry

ARM assembler in Raspberry Pi – Chapter 5 **ARM assembler in Raspberry Pi – Chapter 7**

5 thoughts on "ARM assembler in Raspberry Pi – Chapter 6"

Kevin Millikin says:
January 21, 2013 at 3:38 am

Thanks for writing this. It's been fun reading so far.

In practice one would normally test at the bottom of a while loop. In your loop01.s there are two tests in the loop on each iteration — a forward conditional branch (not taken) and a backward unconditional branch.

Instead compile the test at the bottom, invert the branch condition, and enter the loop with an unconditional branch to the test. Each iteration of the loop has a backward conditional