ARM assembler in Raspberry Pi – Chapter 7

January 26, 2013 rferrer, 3

ARM architecture has been for long targeted at embedded systems. Embedded systems usually end being used in massively manufactured products (dishwashers, mobile phones, TV sets, etc). In this context margins are very tight so a designer will always try to spare as much components as possible (a cent saved in hundreds of thousands or even millions of appliances may pay off). One relatively expensive component is memory although every day memory is less and less expensive. Anyway, in constrained memory environments being able to save memory is good and ARM instruction set was designed with this goal in mind. It will take us several chapters to learn all of these techniques, today we will start with one feature usually named *shifted operand*.

Indexing modes

We have seen that, except for load (`ldr`), store (`str`) and branches (b and `bXX`), ARM instructions take as operands either registers or immediate values. We have also seen that the first operand is usually the destination register (being `str` a notable exception as there it plays the role of source because the destination is now the memory). Instruction `mov` has another operand, a register or an immediate value. Arithmetic instructions like `add` and `and` (and many others) have two more source registers, the first of which is always a register and the second can be a register or an immediate value.

These sets of allowed operands in instructions are collectively called *indexing modes*. Today this concept will look a bit off since we will not index anything. The name *indexing* makes sense in memory operands but ARM instructions, except load and store, do not have memory operands. This is the nomenclature you will find in ARM documentation so it seems sensible to use theirs.

We can summarize the syntax of most of the ARM instructions in the following pattern

```
instruction Rdest, Rsource1, source2
```

There are some exceptions, mainly move (`mov`), branches, load and stores. In fact move is not so different actually.

```
mov Rdest, source2
```

Both `Rdest` and `Rsource` must be registers. In the next section we will talk about `source2`.

We will discuss the indexing modes of load and store instructions in a future chapter. Branches, on the other hand, are surprisingly simple and their single operand is just a label of our program, so there is little to discuss on indexing modes for branches.

Shifted operand

What is this misterious `source2` in the instruction patterns above? If you recall the previous chapters we have used registers or immediate values. So at least that `source2` is this: register or immediate value. You can use an immediate or a register where a `source2` is expected. Some examples follow, but we have already used them in the examples of previous chapters.

```
mov r0, #1
mov r1, r0
add r2, r1, r0
add r2, r3, #4
```

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

But `source2` can be much more than just a simple register or an immediate. In fact, when it is a register we can combine it with a *shift operation*. We already saw one of these shift operations in chapter 6. Not it is time to unveil all of them.

- `LSL #n`
  Logical **S**hift **L**eft. Shifts bits n times left. The n leftmost bits are lost and the n rightmost are set to zero.
- `LSL Rsource3`
  Like the previous one but instead of an immediate the lower byte of a register specifies the amount of shifting.
- `LSR #n`
  Logical **S**hift **R**ight. Shifts bits n times right. The n rightmost bits are lost and the n leftmost bits are set to zero,
- `LSR Rsource3`
  Like the previous one but instead of an immediate the lower byte of a register specifies the amount of shifting.
- `ASR #n`
  Arithmetic **S**hift **R**ight. Like LSR but the leftmost bit before shifting is used instead of zero in the n leftmost ones.
- `ASR Rsource3`
  Like the previous one but using a the lower byte of a register instead of an immediate.
- `ROR #n`
  Rotate **R**ight. Like LSR but the n rightmost bits are not lost bot pushed onto the n leftmost bits
- `ROR Rsource3`
  Like the previous one but using a the lower byte of a register instead of an immediate.

In the listing above, n is an immediate from 1 to 31. These extra operations may be applied to the value in the second source register (to the value, not to the register itself) so we can perform some more operations in a single instruction. For instance, ARM does not have any shift right or left instruction. You just use the `mov` instruction.

```
mov r1, r2, LSL #1
```

You may be wondering why one would want to shift left or right the value of a register. If you recall chapter 6 we saw that shifting left (`LSL`) a value gives a value that the same as multiplying it by 2. Conversely, shifting it right (`ASR` if we use two's complement, `LSR` otherwise) is the same as dividing by 2. Since a shift of n is the same as doing n shifts of 1, shifts actually multiply or divide a value by 2^n.

```
mov r1, r2, LSL #1      /* r1 ← (r2*2) */
mov r1, r2, LSL #2      /* r1 ← (r2*4) */
mov r1, r3, ASR #3      /* r1 ← (r3/8) */
mov r3, 4
mov r1, r2, LSL r3      /* r1 ← (r2*16) */
```

We can combine it with `add` to get some useful cases.

```
add r1, r2, r2, LSL #1   /* r1 ← r2 + (r2*2) equivalent to r1 ← r1*3 */
add r1, r2, r2, LSL #2   /* r1 ← r2 + (r2*4) equivalent to r1 ← r1*5 */
```

You can do something similar with `sub`.

```
sub r1, r2, r2, LSL #3  /* r1 ← r2 - (r2*8) equivalent to r1 ← r2*(-7)
```

ARM comes with a handy `rsb` (**R**everse **Sub**stract) instruction which computes `Rdest` ←

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

source2 - Rsource1 (compare it to sub which computes Rdest ← Rsource1 - source2).

```
rsb r1, r2, r2, LSL #3      /* r1 ← (r2*8) - r2 equivalent to r1 ← r2*7 */
```

Another example, a bit more contrived.

```
/* Complicated way to multiply the initial value of r1 by 42 = 7*3*2 */
rsb r1, r1, r1, LSL #3  /* r1 ← (r1*8) - r1 equivalent to r1 ← 7*r1 */
add r1, r1, r1, LSL #1  /* r1 ← r1 + (2*r1) equivalent to r1 ← 3*r1 */
add r1, r1, r1          /* r1 ← r1 + r1     equivalent to r1 ← 2*r1 */
```

You are probably wondering why would we want to use shifts to perform multiplications. Well, the generic multiplication instruction always work but it is usually much harder to compute by our ARM processor so it may take more time. There are times where there is no other option but for many small constant values a single instruction may be more efficient.

Rotations are less useful than shifts in everyday use. They are usually used in cryptography, to reorder bits and "scramble" them. ARM does not provide a way to rotate left but we can do a n rotate left doing a 32−n rotate right.

```
/* Assume r1 is 0x12345678 */
mov r1, r1, ROR #1   /* r1 ← r1 ror 1. This is r1 ← 0x91a2b3c */
mov r1, r1, ROR #31  /* r1 ← r1 ror 31. This is r1 ← 0x12345678 */
```

That's all for today.

Share / Save

arm, assembler, indexing modes, pi, raspberry

ARM assembler in Raspberry Pi – Chapter 6 **ARM assembler in Raspberry Pi – Chapter 8**

3 thoughts on "ARM assembler in Raspberry Pi – Chapter 7"

Fernando says:
March 21, 2013 at 11:41 pm

Any insight of why is multiplication slower than addition?

Reply

rferrer says:
March 27, 2013 at 9:17 pm

The general multiplication algorithm is more expensive than addition because it involves more steps.

When you add two numbers by hand, you start by vertically aligning their digits. Then you add vertically each column starting from the right and moving to the left, taking care of the carry values you may need during the addition. Well, the circuitry of a processor does something similar. But a processor is able to advance the different carries found along the computation in a way that adding two numbers of N bits does not take a "k*N" amount of time but "k*log(N)". Compare how you calculate a sum: when you add two numbers of 10 digits you need twice more time than when you add two numbers of 5 digits. The processor would just need one extra step to add 10 digits compared to adding 5 digits. Cool, isn't it? 😃

Now consider multiplication. When you multiply two numbers, you start as well by vertically aligning them. But now, you pick each digit of one of the rows (I use the lower one) and then you multiply it with every number of the other row (in my case it would be the upper one). This gives you an intermediate row. Then you