ARM assembler in Raspberry Pi – Chapter 8

January 27, 2013 rferrer, 0

In the previous chapter we saw that the second operand of most arithmetic instructions can use a *shift operator* which allows us to shift and rotate bits. In this chapter we will continue learning the available *indexing modes* of ARM instructions. This time we will focus on load and store instructions.

Arrays and structures

So far we have been able to move 32 bits from memory to registers (load) and back to memory (store). But working on single items of 32 bits (usually called scalars) is a bit limiting. Soon we would find ourselves working on arrays and structures, even if we did not know.

An array is a sequence of items of the same kind in memory. Arrays are a foundational data structure in almost every low level language. Every array has a base address, usually denoted by the name of the array, and contains N items. Each of these items has associated a growing index, ranging from 0 to N-1 or 1 to N. Using the base address and the index we can access an item of the array. We mentioned in chapter 3 that memory could be viewed as an array of bytes. An array in memory is the same, but an item may take more than one single byte.

A structure (or record or tuple) is a sequence of items of possibly diferent kind. Each item of a structure is usually called a field. Fields do not have an associated index but an offset respect to the beginning of the structure. Structures are laid out in memory to ensure that the proper alignment is used in every field. The base address of a structure is the address of its first field. If the base address is aligned, the structure should be laid out in a way that all the field are properly aligned as well.

What do arrays and structure have to do with *indexing modes* of load and store? Well, these indexing modes are designed to make easier accessing arrays and structs.

Defining arrays and structs

To illustrate how to work with arrays and references we will use the following C declarations and implement them in assembler.

```
int a[100];
struct my_struct
{
  char f0;
  int f1;
} b;
```

Let's first define in our assembler the array 'a'. It is just 100 integers. An integer in ARM is 32-bit wide so in our assembler code we have to make room for 400 bytes (4 * 100).

```
1 /* -- array01.s */
2 .data
3
4 .balign 4
5 a: .skip 400
```

In line 5 we define the symbol a and then we make room for 400 bytes. The directive .skip tells the assembler to advance a given number of bytes before emitting the next datum. Here we are skipping 400 bytes because our array of integers takes 400 bytes (4 bytes per each of the 100

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

integers). Declaring a structure is not much different.

```
 7 .balign 4
 8 b: .skip 8
```

Right now you should wonder why we skipped 8 bytes when the structure itself takes just 5 bytes. Well, it does need 5 bytes to store useful information. The first field `f0` is a `char`. A `char` takes 1 byte of storage. The next field `f1` is an integer. An integer takes 4 bytes and it must be aligned at 4 bytes as well, so we have to leave 3 unused bytes between the field `f0` and the field `f1`. This unused storage put just to fulfill alignment is called *padding*. Padding should never be used by your program.

Naive approach without indexing modes

Ok, let's write some code to initialize every item of the array `a[i]`. We will do something equivalent to the following C code.

```
for (i = 0; i < 100; i++)
  a[i] = i;
```

```
10 .text
11
12 .global main
13 main:
14     ldr r1, addr_of_a       /* r1 ← &a */
15     mov r2, #0              /* r2 ← 0 */
16 loop:
17     cmp r2, #100            /* Have we reached 100 yet? */
18     beq end                 /* If so, leave the loop, otherwise continue */
19     add r3, r1, r2, LSL #2  /* r3 ← r1 + (r2*4) */
20     str r2, [r3]            /* *r3 ← r2 */
21     add r2, r2, #1          /* r2 ← r2 + 1 */
22     b loop                  /* Go to the beginning of the loop */
23 end:
24     bx lr
25 addr_of_a: .word a
```

Whew! We are using lots of things we have learnt from earlier chapters. In line 14 we load the base address of the array into `r1`. The address of the array will not change so we load it once. In register `r2` we will keep the index that will range from 0 to 99. In line 17 we compare it to 100 to see if we have reached the end of the loop.

Line 19 is an important one. Here we compute the address of the item. We have in `r1` the base address and we know each item is 4 bytes wide. We know also that `r2` keeps the index of the loop which we will use to access the array element. Given an item with index `i` its address must be `&a + 4*i`, since there are 4 bytes between every element of this array. So `r3` has the address of the current element in this step of the loop. In line 20 we store `r2`, this is `i`, into the memory pointed by `r3`, the `i`-th array item, this is `a[i]`.

Then we proceed to increase `r2` and jump back for the next step of the loop.

As you can see, accessing an array involves calculating the address of the accessed item. Does the ARM instruction set provide a more compact way to do this? The answer is yes. In fact it provides several *indexing modes*.

Indexing modes

In the previous chapter the concept *indexing mode* was a bit off because we were not indexing anything. Now it makes much more sense since we are indexing an array item. ARM provides **nine** of these indexing modes. I will distinguish two kinds of indexing modes: non updating and updating depending on whether they feature a side-effect that we will discuss later, when dealing

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

with updating indexing modes.

Non updating indexing modes

1. `[Rsource1, +#immediate]` or `[Rsource1, -#immediate]`

   It justs adds (or substracts) the immediate value to form the address. This is very useful to array items the index of which is a constant in the code or fields of a structure, since their offset is always constant. In `Rsource1` we put the base address and in `immediate` the offset we want in bytes. The immediate cannot be larger than 12 bits (0..4096). When the immediate is `#0` it is like the usual we have been using `[Rsource1]`.

   For example, we can set a `[4]` to 3 this way(we assume that `r1` already contans the base address of a). Note that the offset is in bytes thus we need an offset of 12 (4 bytes * 3 items skipped).

```
mov r2, #3          /* r2 ← 3 */
str r2, [r1, +#12]  /* *(r1 + 12) ← r2 */
```

2. `[Rsource1, +Rsource2]` or `[Rsource1, -Rsource2]`

   This is like the previous one, but the added (or substracted) offset is the value in a register. This is useful when the offset is too big for the immediate. Note that for the `+Rsource2` case, the two registers can be swapped (as this would not affect the address computed).

   Example. The same as above but using a register this time.

```
mov r2, #3          /* r2 ← 3 */
mov r3, #12         /* r3 ← 12 */
str r2, [r1,+r3]    /* *(r1 + r3) ← r2 */
```

3. `[Rsource1, +Rsource2, shift_operation #immediate]` or `[Rsource1, -Rsource2, shift_operation #immediate]`.

   This one is similar to the usual shift operation we can do with other instructions. A shift operation (remember: LSL, LSR, ASR or ROR) is applied to `Rsource2`, `Rsource1` is then added (or substracted) to the result of the shift operation applied to `Rsource2`. This is useful when we need to multiply the address by some fixed amount. When accessing the items of the intege array a we had to multiply the result by 4 to get a meaningful address.

   For this example, let's first recall how we computed above the address of a single item in the array.

```
19 add r3, r1, r2, LSL #2  /* r3 ← r1 + r2*4 */
20 str r2, [r3]            /* *r3 ← r2 */
```

   We can express this in a much more compact way (without the need of the register `r3`).

```
str r2, [r1, +r2, LSL #2]  /* *(r1 + r2*4) ← r2 */
```

Updating indexing modes

In these indexing modes the `Rsource1` register is updated with the address synthesized by the load or store instruction. You may be wondering why one would want to do this. A bit of detour first. Recheck the code of the array load. Why do we have to keep around the base address of the array if we are always effectively moving 4 bytes away from it? Would not it make much more sense to keep the address of the current entity? So instead of

```
19 add r3, r1, r2, LSL #2  /* r3 ← r1 + r2*4 */
20 str r2, [r3]            /* *r3 ← r2 */
```

we might want to do something like

```
str r2, [r1]        /* *r1 ← r2 */
add r1, r1, #4      /* r1 ← r1 + 4 */
```

because there is no need to compute everytime from the beginning the address of the next item (as we are accessing them sequentially). Even if this looks slightly better, it still can be improved a bit more. What if our instruction were able to upate `r1` for us? Something like this (obviously the exact syntax is not as shown)

```
/* Wrong syntax */
str r2, [r1] "and then" add r1, r1, #4
```

Such indexing modes exist. There are two kinds of updating indexing modes depending on at which time `Rsource1` is updated. If `Rsource1` is updated after the load or store itself (meaning that as the address to load or store is the initial `Rsource1` value) this is a *post-indexing* accessing mode. If `Rsource1` is updated before the actual load or store (meaning that the address to load or store is the final value of `Rsource1`) this is a *pre-indexing* accessing mode. In all cases, at the end of the instruction `Rsource1` will have the value of the computation of the indexing mode. Now this sounds a bit convoluted, just look in the example above: we first load using `r1` and then we do `r1 ← r1 + 4`. This is post-indexing: we first use the value of `r1` as the address where we store the value of `r2`. Then `r1` is updated with `r1 + 4`. Now consider another hypothetic syntax.

```
/* Wrong syntax */
str r2, [add r1, r1, #4]
```

This is pre-indexing: we first compute `r1 + 4` and use it as the address where we store the value of `r2`. At the end of the instruction `r1` has effectively been updated too, but the updated value has already been used as the address of the load or store.

Post-indexing modes

4. `[Rsource1], #+immediate` or `[Rsource1], #-immediate`

   The value of `Rsource1` is used as the address for the load or store. Then `Rsource1` is updated with the value of `immediate` after adding (or substracting) it to `Rsource1`. Using this indexing mode we can rewrite the loop of our first example as follows:

```
16 loop:
17     cmp r2, #100             /* Have we reached 100 yet? */
18     beq end                  /* If so, leave the loop, otherwise continue */
19     str r2, [r1], +4         /* *r1 ← r2 then r1 ← r1 + 4 */
20     add r2, r2, #1           /* r2 ← r2 + 1 */
21     b loop                   /* Go to the beginning of the loop */
22 end:
```

5. `[Rsource1], +Rsource2` or `[Rsource1], -Rsource2`

   Like the previous one but instead of an immediate, the value of `Rsource2` is used. As usual this can be used as a workaround when the offset is too big for the immediate value.

6. `[Rsource1], +Rsource2, shift_operation #immediate` or
   `[Rsource1], -Rsource2, shift_operation #immediate`

   The value of `Rsource1` is used as the address for the load or store. Then `Rsource2` is applied a shift operation (`LSL`, `LSR`, `ASR` or `ROL`). The resulting value of that shift is added (or substracted) to `Rsource1`. `Rsource1` is finally updated with this last value.

Pre-indexing modes

Pre-indexing modes may look a bit weird at first but they are useful when the computed address is going to be reused soon. Instead of recomputing it we can reuse the updated `Rsource1`. Mind the `!` symbol in these indexing modes which distinguishes them from the non updating indexing modes.

7. `[Rsource1, #+immediate]!` or `[Rsource1, #-immediate]!`

   It behaves like the similar non-updating indexing mode but `Rsource1` gets updated with the computed address. Imagine we want to compute `a[4] = a[4] + a[4]`. We could do this (we assume that `r1` already has the base address of the array).

```
ldr r2, [r1, #+12]!  /* r1 ← r1 + 12 then r2 ← *r1 */
add r2, r2, r2       /* r2 ← r2 + r2 */
str r2, [r1]         /* *r1 ← r2 */
```

8. `[Rsource1, +Rsource2]!` or `[Rsource1, +Rsource2]!`

   Similar to the previous one but using a register `Rsource2` instead of an immediate.

9. `[Rsource1, +Rsource2, shift_operation #immediate]!` or `[Rsource1, -Rsource2, shift_operation #immediate]!`

   Like to the non-indexing equivalent but Rsource1 will be updated with the address used for the load or store instruction.

Back to structures

All the examples in this chapter have used an array. Structures are a bit simpler: the offset to the fields is always constant: once we have the base address of the structure (the address of the first field) accessing a field is just an indexing mode with an offset (usually an immediate). Our current structure features, on purpose, a `char` as its first field `f0`. Currently we cannot work on scalars in memory of different size than 4 bytes. So we will postpone working on that first field for a future chapter.

For instance imagine we wanted to increment the field f1 like this.

```
b.f1 = b.f1 + 7;
```

If `r1` contains the base address of our structure, accessing the field `f1` is pretty easy now that we know all the available indexing modes.

```
1 ldr r2, [r1, #+4]!  /* r1 ← r1 + 4 then r2 ← *r1 */
2 add r2, r2, #7      /* r2 ← r2 + 7 */
3 str r2, [r1]        /* *r1 ← r2 */
```

Note that we use a pre-indexing mode to keep in `r1` the address of the field `f1`. This way the second store does not need to compute that address again.

That's all for today.

addresses, arm, assembler, indexing modes, pi, postindex, preindex, raspberry

ARM assembler in Raspberry Pi – Chapter 7 **ARM assembler in Raspberry Pi – Chapter 9**

Leave a Reply