ARM assembler in Raspberry Pi – Chapter 5

January 19, 2013 rferrer, 3

Branching

Until now our small assembler programs execute one instruction after the other. If our ARM processor were only able to run this way it would be of limited use. It could not react to existing conditions which may require different sequences of instructions. This is the purpose of the *branch* instructions.

A special register

In chapter 2 we learnt that our Raspberry Pi ARM processor has 16 integer general purpose registers and we also said that some of them play special roles in our program. I deliberately ignored which registers were special as it was not relevant at that time.

But now it is relevant, at least for register `r15`. This register is very special, so special it has also another name: `pc`. It is unlikely that you see it used as `r15` since it is confusing (although correct from the point of view of the ARM architecture). From now we will only use `pc` to name it.

What does `pc` stand for? `pc` means *program counter*. This name, the origins of which are in the dawn of computing, means little to nothing nowadays. In general the `pc` register (also called `ip`, *instruction pointer*, in other architectures like 386 or x86_64) contains the address of the next instruction going to be executed.

When the ARM processor executes an instruction, two things may happen at the end of its execution. If the instruction does not modify `pc` (and most instructions do not), `pc` is just incremented by 4 (like if we did `add pc, pc, #4`). Why 4? Because in ARM, instructions are 32 bit wide, so there are 4 bytes between every instruction. If the instruction modifies `pc` then the new value for `pc` is used.

Once the processor has fully executed an instruction then it uses the value in the `pc` as the address for the next instruction to execute. This way, an instruction that does not modify the `pc` will be followed by the next contiguous instruction in memory (since it has been automatically increased by 4). This is called *implicit sequencing* of instructions: after one has run, usually the next one in memory runs. But if an instruction does modify the `pc`, for instance to a value other than `pc + 4`, then we can be running another instruction of the program. This process of changing the value of `pc` is called *branching*. In ARM this done using *branch instructions*.

Unconditional branches

You can tell the processor to branch unconditionally by using the instruction `b` (for *branch*) and a label. Consider the following program.

```
1 /* -- branch01.s */
2 .text
3 .global main
4 main:
5     mov r0, #2 /* r0 ← 2 */
6     b end      /* branch to 'end' */
7     mov r0, #3 /* r0 ← 3 */
8 end:
9     bx lr
```

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

If you execute this program you will see that it returns an error code of 2.

```
$ ./compare01 ; echo $?
2
```

What happened is that instruction `b end` *branched* (modifying the `pc`) to the instruction at the label `end`, which is `bx lr`, the instruction we run at the end of our program. This way the instruction `mov r0, #3` has not actually been run at all (the processor never reached that instruction).

At this point the unconditional branch instruction `b` may look a bit useless. It is not the case. In fact this instruction is essential in some contexts, in particular when linked with conditional branching. But before we can talk about conditional branching we need to talk about conditions.

Conditional branches

If our processor were only able to branch just because, it would not be very useful. It is much more useful to branch *when some condition is met*. So a processor should be able to evaluate some sort of conditions.

Before continuing, we need to unveil another register called `cpsr` (for Current Program Status Register). This register is a bit special and directly modifying it is out of the scope of this chapter. That said, it keeps some values that can be read and updated when executing an instruction. The values of that register include four *condition code flags* called N (**n**egative), Z (**z**ero), C (**c**arry) and V (o**v**erflow). These four condition code flags are usually read by branch instructions. Arithmetic instructions and special testing and comparison instruction can update these condition codes too if requested.

The semantics of these four condition codes in instructions updating the `cpsr` are roughly the following

- N will be enabled if the result of the instruction yields a negative number. Disabled otherwise.
- Z will be enabled if the result of the instruction yields a zero value. Disabled if nonzero.
- C will be enabled if the result of the instruction yields a value that requires a 33rd bit to be fully represented. For instance an addition that overflows the 32 bit range of integers.
- V will be enabled if the result of the instruction yields a value that cannot be represented in 32 bits two's complement.

So we have all the needed pieces to perform branches conditionally. But first, let's start comparing two values. We use the instruction `cmp` for this purpose.

```
cmp r1, r2 /* updates cpsr doing "r1 - r2", but r1 and r2 are not modified */
```

This instruction substracts to the value in the first register the value in the second register. Examples of what could happen in the snippet above?

- If `r2` had a value (strictly) greater than `r1` then N would be enabled because `r1-r2` would yield a negative result.
- If `r1` and `r2` had the same value, then Z would be enabled because `r1-r2` would be zero.
- If `r1` was 0 and `r2` was 1 then `r1-r2` would borrow, so in this case C would be enabled.
- If `r1` was 2147483648 (the largest positive integer in 32 bit two's complement) and `r1` was -1 then `r1-r2` would be 2147483649 but such number cannot be represented in 32 bit two's complement, so V would be enabled to signal this.

How can we use these flags to represent useful conditions for our programs?

- EQ (**eq**ual) When Z is enabled (Z is 1)

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

- NEQ (**n**ot **eq**ual). When Z is disabled. (Z is 0)
- GE (**g**reater or **e**qual than, in two's complement). When both Z and N are enabled or disabled (Z is N)
- LT (**l**ower **t**han, in two's complement). This is the opposite of GE, so when Z and N are not both enabled or disabled (Z is not N)
- GT (**g**reather **t**han, in two's complement). When Z is disabled and N and V are both enabled or disabled (Z is 0, N is V)
- LE (**l**ower or **e**qual than, in two's complement). When Z is enabled or if not that, N and V are both enabled or disabled (Z is 1. If Z is not 1 then N is V)
- MI (**mi**nus/negative) When N is enabled (N is 1)
- PL (**pl**us/positive or zero) When N is disabled (N is 0)
- OS (**o**verflow **s**et) When V is enabled (V is 1)
- OC (**o**verflow **c**lear) When V is disabled (V is 0)
- HI (**hi**gher) When C is enabled and Z is disabled (C is 1 and Z is 0)
- LS (**l**ower or **s**ame) When C is disabled or Z is enabled (C is 0 or Z is 1)
- CS/HS (**c**arry **s**et/**h**igher or **s**ame) When C is enabled (C is 1)
- CC/LO (**c**arry **c**lear/**lo**wer) When C is disabled (C is 0)

These conditions can be combined to our b instruction to generate new instructions. This way, `beq` will branch only if Z is 1. If the condition of a conditional branch is not met, then the branch is ignored and the next instruction will be run. It is the programmer task to make sure that the condition codes are properly set prior a conditional branch.

```
1  /* -- compare01.s */
2  .text
3  .global main
4  main:
5      mov r1, #2        /* r1 ← 2 */
6      mov r2, #2        /* r2 ← 2 */
7      cmp r1, r2        /* update cpsr condition codes with the value of r1-r2 */
8      beq case_equal    /* branch to case_equal only if Z = 1 */
9  case_different :
10     mov r0, #2        /* r0 ← 2 */
11     b end             /* branch to end */
12 case_equal:
13     mov r0, #1        /* r0 ← 1 */
14 end:
15     bx lr
```

If you run this program it will return an error code of 1 because both `r1` and `r2` have the same value. Now change `mov r1, #2` in line 5 to be `mov r1, #3` and the returned error code should be 2. Note that `case_different` we do not want to run the `case_equal` instructions, thus we have to branch to `end` (otherwise the error code would always be 1).

That's all for today.

arm, assembler, branches, pi, raspberry

ARM assembler in Raspberry Pi – Chapter 4 **ARM assembler in Raspberry Pi – Chapter 6**

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

Mikael Hartzell says:
February 7, 2013 at 11:25 pm

Hi

I though that you can manipulate the cpsr directly, isn't the commands MSR and MRS meant for that ? (Source: Arm v6 reference manual: