ARM assembler in Raspberry Pi – Chapter 11

March 16, 2013 rferrer, 4

Several times, in earlier chapters, I stated that the ARM architecture was designed with the embedded world in mind. Although the cost of the memory is everyday lower, it still may account as an important part of the budget of an embedded system. The ARM instruction set has several features meant to reduce the impact of code size. One of the features which helps in such approach is **predication**.

Predication

We saw in chapters 6 and 7 how to use branches in our program in order to modify the execution flow of instructions and implement useful control structures. Branches can be unconditional, for instance when calling a function as we did in chapters 9 and 10, or conditional when we want to jump to some part of the code only when a previously tested condition is met.

Predication is related to conditional branches. What if, instead of branching to some part of code meant to be executed only when a condition C holds, we were able to *turn* some instructions *off* when that C condition does not hold?. Consider some case like this.

```
if (C)
  T();
else
  E();
```

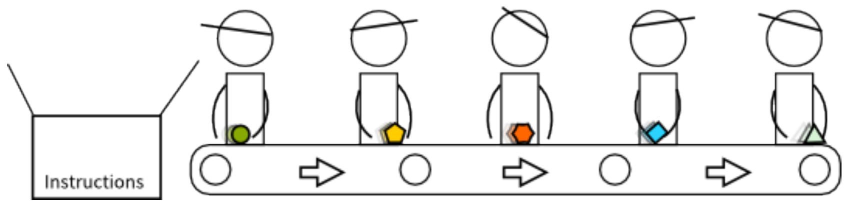
Using predication (and with some invented syntax to express it) we could write the above if as follows.

```
P = C;
[P]  T();
[!P] E();
```

This way we avoid branches. But, why would be want to avoid branches? Well, executing a conditional branch involves a bit of uncertainty. But this deserves a bit of explanation.

The assembly line of instructions

Imagine an assembly line. In that assembly line there are 5 workers, each one fully specialized in a single task. That assembly line *executes* instructions. Every instruction enters the assembly line from the left and leaves it at the right. Each worker does some task on the instruction and passes to the next worker to the right. Also, imagine all workers are more or less synchronized, each one ends the task in as much 6 seconds. This means that at every 6 seconds there is an instruction leaving the assembly line, an instruction fully executed. It also means that at any given time there may be up to 5 instructions being processed (although not fully executed, we only have one fully executed instruction at every 6 seconds).



The first worker *fetches* instructions and puts them in the assembly line. It fetches the instruction at the address specified by the register pc. By default, unless told, this worker *fetches* the

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
ท.ร.ม. ด้วยภาษา Assembly บน Raspberry Pi | Raspberry Pi Thailand on ARM assembler in Raspberry Pi – Chapter 9

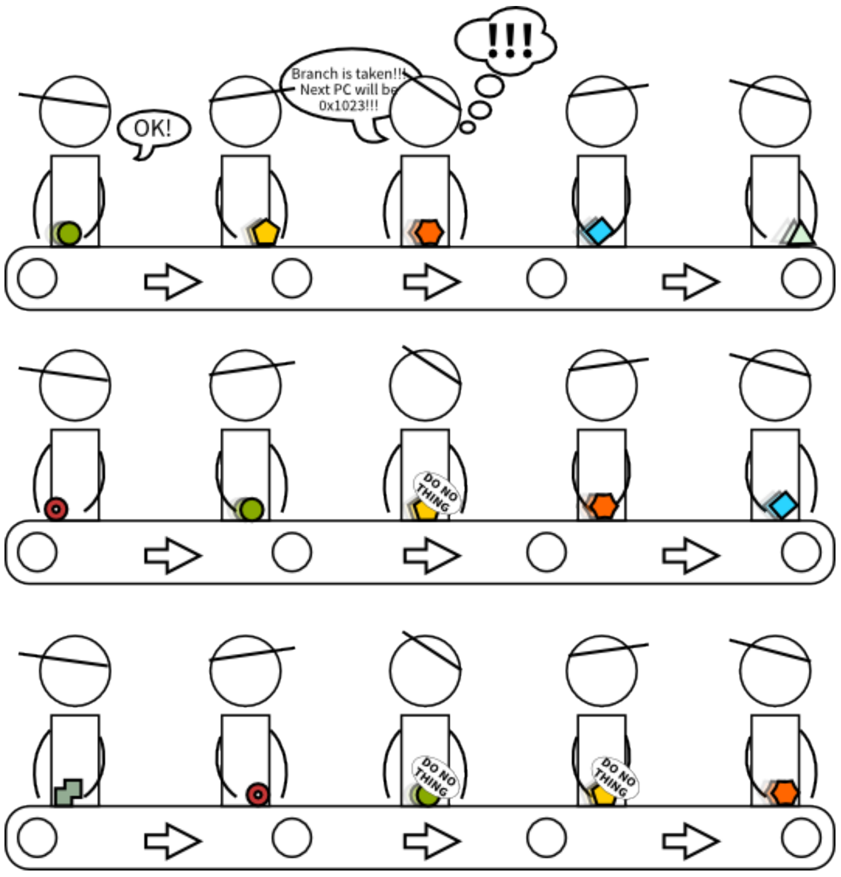
Tags

.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

instruction physically following the one he previously fetched (this is *implicit sequencing*).

In this assembly line, the worker that checks the condition of a conditional branch is not the first one but the third one. Now consider what happens when the first worker fetches a conditional branch and puts it in the assembly line. The second worker will process it and pass it to the third one. The third one will process it by checking the condition of the conditional branch. If it does not hold, nothing happens, the branch has no effect. But if the condition holds, the third worker must notify the first one that the next instruction fetched should be the instruction at the address of the branch.

But now there are two instructions in the assembly line that should not be fully executed (the ones that were physically after the conditional branch). There are several options here. The third worker may pick two stickers labeled as DO NOTHING, and stick them to the two next instructions. Another approach would be the third worker to tell the first and second workers «hey guys, stick a DO NOTHING to your current instruction». Later workers, when they see these DO NOTHING stickers will do, huh, nothing. This way each DO NOTHING instruction will never be fully executed.



But by doing this, that nice property of our assembly line is gone: now we do not have a fully executed instruction every 6 seconds. In fact, after the conditional branch there are two DO NOTHING instructions. A program that is constantly doing branches may well reduce the performance of our assembly line from one (useful) instruction each 6 seconds to one instruction each 18 seconds. This is three times slower!

Truth is that modern processors, including the one in the Raspberry Pi, have *branch predictors* which are able to mitigate these problems: they try to predict whether the condition will hold, so the branch is taken or not. Branch predictors, though, predict the future like stock brokers, using the past and, when there is no past information, using some sensible assumptions. So branch predictors may work very well with relatively predictable codes but may work not so well if the code has unpredictable behaviour. Such behaviour, for instance, is observed when running decompressors. A compressor reduces the size of your files removing the redundancy. Redundant stuff is predictable and can be omitted (for instance in “he is wearing his coat” you could ommit “he” or replace “his” by “its”, regardless of whether doing this is rude, because you know you are talking about a male). So a decompressor will have to decompress a file which has

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

very little redundancy, driving nuts the predictor.

Back to the assembly line example, it would be the first worker who attempts to predict where the branch will be taken or not. It is the third worker who verifies if the first worker did the right prediction. If the first worker mispredicted the branch, then we have to apply two stickers again and notify the first worker which is the right address of the next instruction. If the first worker predicted the branch right, nothing special has to be done, which is great.

If we avoid branches, we avoid the uncertainty of whether the branch is taken or not. So it looks like that predication is the way to go. Not so fast. Processing a bunch of instructions that are actually turned off is not an efficient usage of a processor.

Back to our assembly line, the third worker will check the predicate. If it does not hold, the current instruction will get a DO NOTHING sticker but in contrast to a branch, it does not notify the first worker.

So it ends, as usually, that no approach is perfect on its own.

Predication in ARM

In ARM, predication is very simple to use: almost all instructions can be predicated. The predicate is specified as a suffix to the instruction name. The suffix is exactly the same as those used in branches in the chapter 5: `eq`, `neq`, `le`, `lt`, `ge` and `gt`. Instructions that are not predicated are assumed to have a suffix `al` standing for ***al**ways*. That predicate always holds and we do not write it for economy (it is valid though). You can understand conditional branches as predicated branches if you feel like.

Collatz conjecture revisited

In chapter 6 we implementd an algorithm that computed the length of the sequence of Hailstone of a given number. Though not proved yet, no number has been found that has an infinite Hailstone sequence. Given our knowledge of functions we learnt in chapters 9 and 10, I encapsulated the code that computes the length of the sequence of Hailstone in a function.

```
/* -- collatz02.s */
.data

message: .asciz "Type a number: "
scan_format : .asciz "%d"
message2: .asciz "Length of the Hailstone sequence for %d is %d\n"

.text

collatz:
    /* r0 contains the first argument */
    /* Only r0, r1 and r2 are modified,
       so we do not need to keep anything
       in the stack */
    /* Since we do not do any call, we do
       not have to keep lr either */
    mov r1, r0                 /* r1 ← r0 */
    mov r0, #0                 /* r0 ← 0 */
  collatz_loop:
    cmp r1, #1                 /* compare r1 and 1 */
    beq collatz_end            /* if r1 == 1 branch to collatz_end */
    and r2, r1, #1             /* r2 ← r1 & 1 */
    cmp r2, #0                 /* compare r2 and 0 */
    bne collatz_odd            /* if r2 != 0 (this is r1 % 2 != 0) branch to collatz_o
  collatz_even:
    mov r1, r1, ASR #1         /* r1 ← r1 >> 1. This is r1 ← r1/2 */
    b collatz_end_loop         /* branch to collatz_end_loop */
  collatz_odd:
    add r1, r1, r1, LSL #1     /* r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    add r1, r1, #1             /* r1 ← r1 + 1. */
  collatz_end_loop:
    add r0, r0, #1             /* r0 ← r0 + 1 */
    b collatz_loop             /* branch back to collatz_loop */
```

```
34   collatz_end:
35     bx lr
36
37 .global main
38 main:
39     push {lr}                    /* keep lr */
40     sub sp, sp, #4               /* make room for 4 bytes in the stack */
41                                  /* The stack is already 8 byte aligned */
42
43     ldr r0, address_of_message      /* first parameter of printf: &message */
44     bl printf                    /* call printf */
45
46     ldr r0, address_of_scan_format  /* first parameter of scanf: &scan_format */
47     mov r1, sp                   /* second parameter of scanf:
48                                     address of the top of the stack */
49     bl scanf                     /* call scanf */
50
51     ldr r0, [sp]                 /* first parameter of collatz:
52                                     the value stored (by scanf) in the top of the s
53     bl collatz                   /* call collatz */
54
55     mov r2, r0                   /* third parameter of printf:
56                                     the result of collatz */
57     ldr r1, [sp]                 /* second parameter of printf:
58                                     the value stored (by scanf) in the top of the s
59     ldr r0, address_of_message2     /* first parameter of printf: &address_of_message2
60     bl printf
61
62     add sp, sp, #4
63     pop {lr}
64     bx lr
65
66
67 address_of_message: .word message
68 address_of_scan_format: .word scan_format
69 address_of_message2: .word message2
```

Adding predication

Ok, let’s add some predication. There is an *if-then-else* construct in lines 22 to 31. There we check if the number is even or odd. If even we divide it by 2, if even we multiply it by 3 and add 1.

```
22     and r2, r1, #1           /* r2 ← r1 & 1 */
23     cmp r2, #0               /* compare r2 and 0 */
24     bne collatz_odd          /* if r2 != 0 (this is r1 % 2 != 0) branch to collatz_c
25   collatz_even:
26     mov r1, r1, ASR #1       /* r1 ← r1 >> 1. This is r1 ← r1/2 */
27     b collatz_end_loop       /* branch to collatz_end_loop */
28   collatz_odd:
29     add r1, r1, r1, LSL #1   /* r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
30     add r1, r1, #1           /* r1 ← r1 + 1. */
31   collatz_end_loop:
```

Note in line 24 that there is a `bne` (**b**ranch if **n**ot **e**qual). We can use this condition (and its opposite `eq`) to predicate this *if-then-else* construct. Instructions in the *then* part will be predicated using `eq`, instructions in the *else* part will be predicated using `ne`. The resulting code is shown below.

```
    cmp r2, #0                 /* compare r2 and 0 */
    moveq r1, r1, ASR #1       /* if r2 == 0, r1 ← r1 >> 1. This is r1 ← r1/2 */
    addne r1, r1, r1, LSL #1   /* if r2 != 0, r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    addne r1, r1, #1           /* if r2 != 0, r1 ← r1 + 1. */
```

As you can se there are no labels in the predicated version. We do not branch now so they are not needed anymore. Note also that we actually removed two branches: the one that branches from the condition test code to the *else* part and the one that branches from the end of the *then* part to the instruction after the whole *if-then-else*. This leads to a more compact code.

Does it make any difference in performance?

Taken as is, this program is very small to be accountable for time, so I modified it to run the same calculation inside the collatz function 4194304 (this is 2^{22}) times. I chose the number after some tests, so the execution did not take too much time to be a tedium.

Sadly, while the Raspberry Pi processor provides some hardware performance counters I have not been able to use any of them. `perf` tool (from the package `linux-tools-3.2`) complains that the counter cannot be opened.

```
$ perf_3.2 stat -e cpu-cycles ./collatz02
  Error: open_counter returned with 19 (No such device). /bin/dmesg may provide additiona

  Fatal: Not all events could be opened
```

`dmesg` does not provide any additional information. We can see, though, that the performance counters was loaded by the kernel.

```
$ dmesg | grep perf
[    0.061722] hw perfevents: enabled with v6 PMU driver, 3 counters available
```

Supposedly I should be able to measure up to 3 hardware events at the same time. I think the Raspberry Pi processor, packaged in the BCM2835 SoC does not provide a PMU (Performance Monitoring Unit) which is required for performance counters. Nevertheless we can use `cpu-clock` to measure the time.

Below are the versions I used for this comparison. First is the branches version, second the predication version.

```
collatz:
    /* r0 contains the first argument */
    push {r4}
    sub sp, sp, #4  /* Make sure the stack is 8 byte aligned */
    mov r4, r0
    mov r3, #4194304
  collatz_repeat:
    mov r1, r4                 /* r1 ← r0 */
    mov r0, #0                 /* r0 ← 0 */
  collatz_loop:
    cmp r1, #1                 /* compare r1 and 1 */
    beq collatz_end            /* if r1 == 1 branch to collatz_end */
    and r2, r1, #1             /* r2 ← r1 & 1 */
    cmp r2, #0                 /* compare r2 and 0 */
    bne collatz_odd            /* if r2 != 0 (this is r1 % 2 != 0) branch to collatz_c
  collatz_even:
    mov r1, r1, ASR #1         /* r1 ← r1 >> 1. This is r1 ← r1/2 */
    b collatz_end_loop         /* branch to collatz_end_loop */
  collatz_odd:
    add r1, r1, r1, LSL #1     /* r1 ← r1 + (r1 << 1). This is r1 ← 3*r1 */
    add r1, r1, #1             /* r1 ← r1 + 1. */
  collatz_end_loop:
    add r0, r0, #1             /* r0 ← r0 + 1 */
    b collatz_loop             /* branch back to collatz_loop */
  collatz_end:
    sub r3, r3, #1
    cmp r3, #0
    bne collatz_repeat
    add sp, sp, #4  /* Make sure the stack is 8 byte aligned */
    pop {r4}
    bx lr
```

```
collatz2:
    /* r0 contains the first argument */
    push {r4}
    sub sp, sp, #4  /* Make sure the stack is 8 byte aligned */
    mov r4, r0
    mov r3, #4194304
  collatz_repeat:
    mov r1, r4                 /* r1 ← r0 */
    mov r0, #0                 /* r0 ← 0 */
  collatz2_loop:
    cmp r1, #1                 /* compare r1 and 1 */
    beq collatz2_end           /* if r1 == 1 branch to collatz2_end */
    and r2, r1, #1             /* r2 ← r1 & 1 */
    cmp r2, #0                 /* compare r2 and 0 */
    moveq r1, r1, ASR #1       /* if r2 == 0, r1 ← r1 >> 1. This is r1 ← r1/2 */
    addne r1, r1, r1, LSL #1   /* if r2 != 0, r1 ← r1 + (r1 << 1). This is r1 ← 3*r1
    addne r1, r1, #1           /* if r2 != 0, r1 ← r1 + 1. */
  collatz2_end_loop:
    add r0, r0, #1             /* r0 ← r0 + 1 */
    b collatz2_loop            /* branch back to collatz2_loop */
  collatz2_end:
    sub r3, r3, #1
    cmp r3, #0
    bne collatz_repeat
    add sp, sp, #4             /* Restore the stack */
    pop {r4}
    bx lr
```

The tool `perf` can be used to gather performance counters. We will run 5 times each version. We will use number 123. We redirect the output of `yes 123` to the standard input of our tested program. This way we do not have to type it (which may affect the timing of the comparison).

The version with branches gives the following results:

```
$ yes 123 | perf_3.2 stat --log-fd=3 --repeat=5 -e cpu-clock ./collatz_branches 3>&1
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46

 Performance counter stats for './collatz_branches' (5 runs):

       3359,953200 cpu-clock                  ( +-  0,01% )

       3,365263737 seconds time elapsed                                  ( +-  0,
```

(When redirecting the input of `perf` one must specify the file descriptor for the output of `perf stat` itself. In this case we have used the file descriptor number 3 and then told the shell to redirect the file descriptor number 3 to the standard output, which is the file descriptor number 1).

```
$ yes 123 | perf_3.2 stat --log-fd=3 --repeat=5 -e cpu-clock ./collatz_predication 3>&1
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46
Type a number: Length of the Hailstone sequence for 123 is 46

 Performance counter stats for './collatz_predication' (5 runs):

       2318,217200 cpu-clock                  ( +-  0,01% )

       2,322732232 seconds time elapsed                                  ( +-  0,
```

So the answer is, yes. In **this case** it does make a difference. The predicated version runs 1,44 times faster than the version using branches. It would be bold, though, to assume that in general predication outperforms branches. Always measure your time.

That's all for today.

Share / Save

arm, assembler, branches, function, function call, functions, pi, predication, raspberry

ARM assembler in Raspberry Pi – Chapter 10

ARM assembler in Raspberry Pi – Chapter 12

4 thoughts on "ARM assembler in Raspberry Pi – Chapter 11"