ARM Assembly Programming

April 19, 2020

1 Pipelining and Program Counter

From the architecture ARMv7, arm uses pipeline to make the operation fast. A three stage pipeline is used.

- Fetch
- Decode
- Execute

When the processor executes the current instruction, it will be decoding the previous instruction and fetching the fetching the instruction previous to that. So the program counter (PC) will have the address of the instruction which is being fetched.

- In ARM mode, value of PC will be addr_of_curr_ins + 8.
- In THUMB mode, value of PC will be addr_of_curr_ins + 4.

Example:

```
            .section .text
            .global _start
            _start:
0x00010054      mov r0, pc
0x00010058      mov r1, #2
0x0001005C      add r2, r1, r1
```

The value of r0 after executing instruction "1" will be 0x0001005C. While executing instruction "1", the value of PC will be 0x0001005C and after execution it will bee 0x00010058.
The Program Counter (PC) points to the instruction being fetched rather than to the instruction being executed.

2 Program Status Register

2.1 Negative Zero Carry

Let's go through some examples

```
    .section .text

    .global _start

    _start:

        mov r0, #2

        mov r1, #4

        cmp r0, r1
```

Use gdb to run the above example. After executing the last instruction, the registers will look like (use *info registers* to print register information)

```
 r0     0x2           2
 r1     0x4           4
 r2     0x0           0
 r3     0x0           0
 r4     0x0           0
 r5     0x0           0
 r6     0x0           0
 r7     0x0           0
 r8     0x0           0
 r9     0x0           0
 r10    0x0           0
 r11    0x0           0
 r12    0x0           0
 sp     0xbefff380    0xbefff380
 lr     0x0           0
 pc     0x10060       0x10060
 cpsr   0x80000010    -2147483632
```

Bit 4 is always 1. The bit 31 is set after the cmp operation. *cmp r0, r1* gives *r0-r1*, which results in a negative number thus sets the negative flag.
Now lets do the opposite, *cmp r1, r0*
cpsr will become 0x20000010
Now the carry flag got set. Why the operation (4-2) sets the carry flag, as though the operation does not introduce a carry?
To understand this lets do two more operations.
subs r2,r0,r1 where r0=2 and r1 = 4
which gives
r2 = 0xfffffffe (-2)
cpsr = 0x80000010
and

subs r2,r1,r0
which gives
r2 = 2
and
cpsr=0x20000010
Let's see the actual subtraction in hardware and find out why?
1. 4-2 = 4+(-2)

```
 11011   (carry)
  0100   (4)
  1101   (not 2)
  0010   (result)
```

Here carry out = 1

2. 2 - 4 = 2 + (-4)

```
 00111   (carry)
  0010   (2)
  1011   (not 4)
  1110   (result)
```

Here carry out = 0

Let's see one more operation.

```
    .section .text

    .global _start

    _start:

        mov r0, #2

        ld r1,[pc]

        adds r2, r0, r1

        .word 0xFFFFFFFE
```

CPSR = 0x60000010
The zero and carry flags are set.
So a carry occurs
1. When the result of a signed addition is greater than 2^{32}
2. If the result of subtraction is positive or zero.
3. The result of an inline barrel shifter operation in a move or logical instruction.
Most instructions update the status flags only if the S suffix is specified

Consider point 3.

```
    .section .text
    .global _start
    _start:
        mov r0, [pc]
        lsls r0,r0,#1
        .word 0x80000000
```

The above code will produce, cpsr=0x60000010.
Carry flag and zero flag are set.
The following operation will also sets the carry flag
lsrs r0,r0,#1 where r0 = 0xFFFFCDEF
r0 become 0x7FFFE6F7
What if r0 is simply 1
This also sets the carry flag.
So carry in ARM means both carry in and carry out.

Now consider the rotation operation.
rors r0,r0,#1 where r0 = 1
This results in r0 = 0x80000000 and sets sign and carry flag. If the rotation is by 2 then both carry and sign won't be set. This is because when the suffix 's' is used the old bit b[0] is placed in the carry. For the roration by two the old bit b[0] is '0' thus the carry flag won't be set.

Till now we have covered negative [1], zero and carry flags. Now lets cover other flags in CPSR.

2.2 Overflow and Cumulative saturation bit

Overflow flag is set when the result of an arithmetic operation doesn't fit in 32 bit 2's compliment.

```
    .section .text
    .global _start
    _start:
        mov r0, [pc]
        add r1,r0,#2
```

[1] Negative is referred as sign flag

```
    .word 0x7FFFFFFF
```

The above code will result in setting of overflow bit. The result in r1 will be 0x80000001 so the sign bit will also set.

Bit[27] of the CPSR is a sticky overflow flag, also known as the Q flag. This flag is set to 1 if any of the following occurs:

- Saturation of the addition result in a QADD or QDADD instruction
- Saturation of the subtraction result in a QSUB or QDSUB instruction
- Saturation of the doubling intermediate result in a QDADD or QDSUB instruction
- Signed overflow during an SMLA¡x¿¡y¿ or SMLAW¡y¿ instruction

```
.section .text

.global _start

_start:

    mov r0, [pc]

    qadd r1,r0,r0

    .word 0x70000000
```

When the above code executed, the resukt should be 0xE0000000, but since there is a saturation occurs, the resukt will be saturated to the maximum possible positive integer. i.e r1 = 0x7FFFFFFF and the cumulative overflow flag will be set.
The Q flag is sticky in that once it has been set to 1, it is not affected by whether subsequent calculations saturate and/or overflow.
Use "msr cpsr_f, #0" to clear the CPSR flags.

3 ARM Registers

- R0 - R6, R8 - R10 : General purpose registers
- R7 : Register to hold syscall number. General purpose.
- R11 : Frame pointer.
- R12 : Intra process call (ip).
- R13 : Stack pointer (sp).
- R14 : Link register (lr).
- R15 : Program counter (pc).
- CPSR

3.1 Use of registers in function call

Arm procedure call standards specifies that the registers r4-r11 must be preserved between function calls and that the called function is responsible for that preservation (otherwise it will be an overhead to push and pop it on every function call). The first four 32 bit values are passed through registers r0-r3. If there is a 64 bit value, either r0-r1 or r2-r3 combination is used to pass that value. All other values are passed by pushing them into the stack. The result is returned in either r0 or r0-r1 for 64 bit value.

Example of calling abs function is shown below.

```
.text
.global _start
abs:
    push {r4, lr}
    adds r0, #0
    poppl {pc}        ; pl is for a contional execution.
    mov r4, #0
    subs r0, r4, r0
    pop {r4, pc}
_start:
    mov r0, #-4        ; Input r0
    bl abs
    mov r0, r0        ; Result r0
```

Details of contional execution will be covered in later sections.

Calling 'c' function from assembly.

```
// Print abs(z) + x + y
.global main
    .extern printf
.text
out_str:
    .ascii ”The answer is %d \n\0”
.align 4
opr:
    push {lr, r4}         ; This will give a warning [2]
    bl abs
    add r4, r0, r1
    add r0, r4, r2
    pop {r4, pc}
abs:
    push {r4, lr}
    adds r0, #0
    poppl pc
    mov r4, #0
    subs r0, r4, r0
    pop {pc}
main:
    push {ip, lr}
    mov r0, #-4
    mov r1, #3
    mov r2, #6
    bl opr
    mov r1, r0
    ldr r0, =out_str
    bl printf
    mov r0, #0
    pop {pc, ip}          ; This will give a warning[2]
```

Compile the above code using gcc, instead of native 'as'. Because gcc wil look for printf in the library path.

[2]The registers must be in ascending order

3.2 Stack Pointer - Push and Pop

Lets see an example of printf with more than 4 inputs.

```
.global main
    .extern printf
.text
text:
    .ascii "This contains %s, %s, %s, %s \n\0"
text_1:
    .ascii "Fisrt text\0"
text_2:
    .ascii "Second text\0"
text_3:
    .ascii "Third text\0
text_4:
    .ascii "Fourth text\0"
.align 4
main:
    push {ip, lr}
    ldr r0, =text
    ldr r1, =text_1
    ldr r2, =text_2
    ldr r3, =text_3
    ldr r4, =text_4
    sub sp, sp, #4
    str r4, [sp]
    bl printf
    add sp, sp, #4
    mov r0, #0 // return 0
    pop {ip, pc}
```