

Lab 08: Coprocessors and exceptions



R. Ferrero, A. C. Marceddu,
M. Russo

Politecnico di Torino

Dipartimento di Automatica e Informatica (DAUIN)

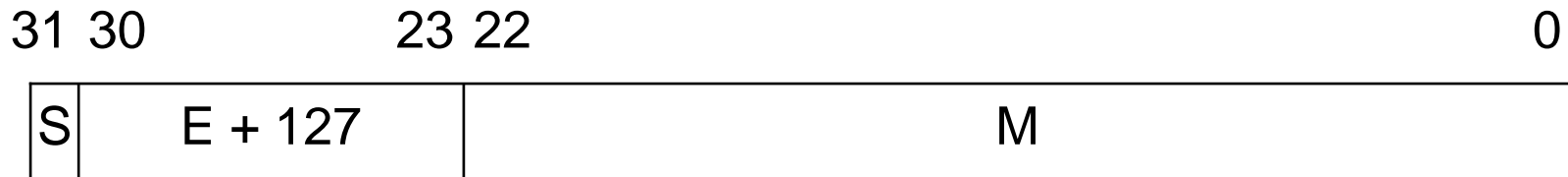
Torino - Italy

This work is licensed under the Creative Commons (CC BY-SA) License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>



Representation of floating point

- Normalized floating point representation:
 $\pm X * 2^E \quad (1 \leq X < 2)$
- IEEE-754 SP standard expresses floating point numbers in 32 bits:
 - Sign S: 0 if positive, 1 if negative.
 - Exponent E of 2, +127.
 - Mantissa M: fractional part of X.



Exercise introduction

- There is not any native implementation of floating point numbers in Cortex-M3: we want to implement it in software.
- With R_n = integer part, R_m = fractional part, we want to write in R_d the floating point number according to IEEE-754 SP standard.

Example

- $R_n = 1998$, $R_m = 142578125$
- 1998.142578125 expressed in normalized scientific notation is $1.9513111145 * 2^{10}$
- $E + 127 = 137 = 10001001_2$

31 30 23 22 0

0	1	0	0	0	1	0	0	1	1	1	1	1	0	0	1	1	1	0	0	0	1	0	0	1	0	0	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- You can check the result here: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

Computation of the exponent

- The exponent is equal to the position of the first bit set to 1 in the binary representation of the **integer** part.
- Example: $1998 = 11111001110_2$

31 10 0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- After adding the bias (127), $E = 137$

Computation of the mantissa (1)

- The first bits of the mantissa are taken from the binary representation of the **integer** part, after removing the initial '1'.
- Example: $1998 = 11111001110_2$
 - Therefore, 1111001110
- The remaining N bits of the mantissa are obtained by converting the **fractional** part.
- The fractional part is interpreted as an integer number.

Computation of the mantissa (2)

- Example: $X = 142578125$
- Let P be the lowest power of 10 which is higher than X . $P = 1000000000$
- This loop is repeated N times:
 - $X = 2X$
 - if $X \geq P$
 - the next bit of the mantissa is 1 and
 $X = X - P$
 - else the next bit of the mantissa is 0
 - repeat loop

Computation of the mantissa (3)

iteration	X	2 * X	bit
1	142578125	285156250	0
2	285156250	570312500	0
3	570312500	1140625000	1
4	140625000	281250000	0
5	281250000	562500000	0
6	562500000	1125000000	1
7	125000000	250000000	0
8	250000000	500000000	0
9	500000000	1000000000	1
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0

First steps

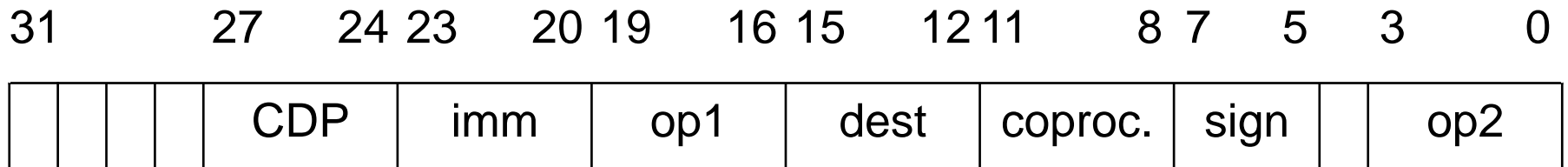
We call a coprocessor as follows:

`CDP proc, imm, dest, op1, op2, sign`

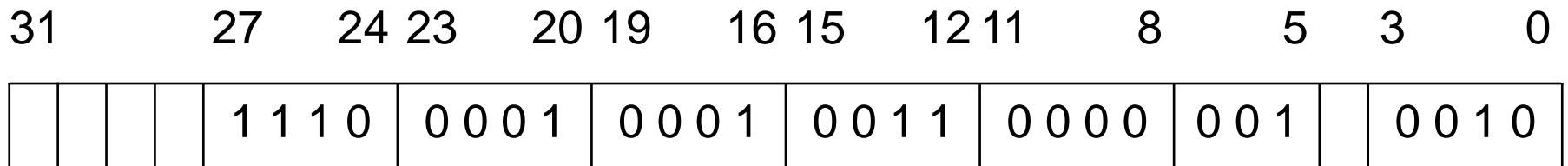
- `CDP`: instruction to call a coprocessor.
- `proc`: called coprocessor. We call `p0`.
- `imm`: operation executed by coprocessor.
 - `imm = 1` for conversion to the IEEE-754 SP.
- `dest, op1, op2`: registers containing result, integer part, fractional part, respectively.
- `sign`: 0 if positive, 1 if negative.

CDP encoding

- Only meaningful bits are shown:



- The registers used by a coprocessor are named $c0, c1, \dots, c15$.
- Example:** CDP $p0, \#1, c3, c1, c2, \#1$



Software implementation

- Cortex-M3 has not any coprocessor.
- CDP raises a usage fault.
 - usage fault must be enabled, otherwise a hard fault is raised.
- The conversion to the IEEE-754 SP format is done in the exception handler.
- The following slides list the steps to be implemented in the exception handler.

1) Recognizing coprocessor fault

- Check the proper bit in the Usage Fault Status Register.
- If the exception is due to a coprocessor instruction, branch to the corresponding piece of code.
- Otherwise, write a dummy implementation of other usage faults (e.g., `B`).

Reminder #1:

Usage Fault Status Register

Bit	Possible causes of usage fault
9	Division by zero and DIV_0_TRP is set.
8	Unaligned memory access attempted and UNALIGN_TRP is set
3	Attempt to execute a coprocessor instruction
2	Invalid EXC_RETURN during exception return. Invalid exception active status. Invalid value of stacked IPSR (stack corruption). Invalid ICI/IT bit for current instruction.
1	Branch target address to PC with LSB equals 0
0	Use of not supported (undefined) instruction.

2) Recognizing offending instruction

- Before entering the exception handler, PC is saved automatically in the stack (MSP or PSP) with offset 24.
 - use of MSP or PSP is determined by reading LR* .
- Load 4 bytes (1 word) from the address of PC
 - due to little endianness, the two halfwords must be switched, by rotating (ROR) 16 positions.
- If the instruction is CDP p0, #1, ...
execute code for IEEE-754 SP conversion.
 - bits must be 0xXE1XX0XX

Reminder #2:

- *By reading LR at the beginning of the handler, we can see what stack the caller was using:

EXC_RETURN

- Bit 0: processor state. 0 -> ARM, 1 -> Thumb
- Bit 1: reserved, it must be 0
- Bit 2: return stack. 0 -> MSP, 1 -> PSP
- Bit 3: return mode. 0 -> handler, 1 -> thread
- Bits 4-31: each bit must be 1
- Allowed values of EXC_RETURN are:
 - 0xFFFFFFFF1: return to handler mode
 - 0xFFFFFFFF9: return to thread mode with MSP
 - 0xFFFFFFFDD: return to thread mode with PSP

Reminder #2 (cont.):

- r0-r3, r12, LR, PC and PSR are saved in the currently used stack (MSP or PSP)
- Then, MSP is always used during exception
- PC and PSR are stacked first, so instruction fetch and PSR update can be started early.

Register	r0	r1	r2	r3	r12	LR	PC	PSR
Stacking order	3	4	5	6	7	8	1	2
- Offset SP_{old}	32	28	24	20	16	12	8	4
Offset SP_{new}	0	4	8	12	16	20	24	28

3) Changing return address

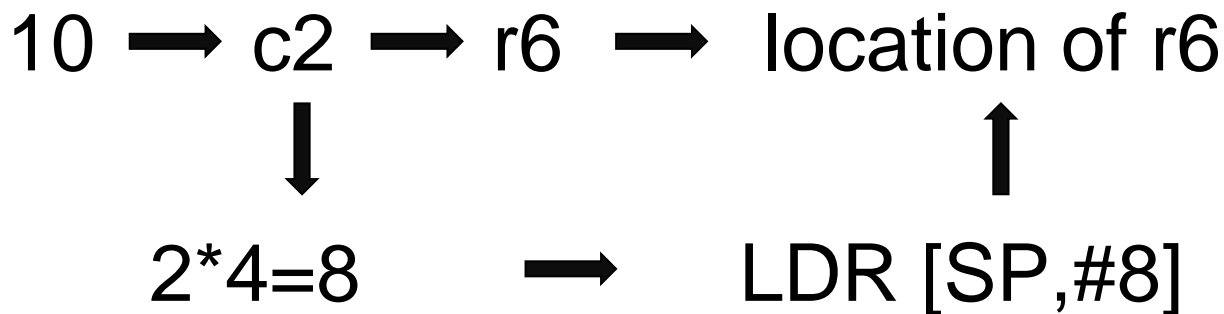
- Usage faults return to the same instructions that triggered the fault.
- Since we do not want to execute CDP again after the handler, the return address must be updated to the next instruction.
- Update the value of PC saved in the stack (with offset 24) with the new value $PC + 4$.

4) Accessing source registers

- Index of registers ranges between 0 and 7.
- We assume that $r4$ corresponds to $c0$, $r5 = c1$, $r6 = c2$, ..., $r11 = c7$.
- Save registers $r4$ - $r11$ in the handler stack (as required by AAPCS); instructions in steps 1, 2, 3 can not modify $r4$ - $r11$.
- After extracting the index of a source register from the encoded instruction, you can extract its content from the stack with offset $\text{index} * 4$ (see following page).

Reminder #3:

- Since it's a FD stack, PUSH {R4-R11} pushes R11 first, then R10, etc..., up to R4
- At the end, SP points to R4
- We said that, for us, c0=r4, c1=r5...
- Assume we read op1=10



Reminder #3 (cont.):

- The fact that $c0=r4$, etc. is just used by the caller when calling the coprocessor for indicating which registers are the two sources and the destination:
 - CDP $p0, \#1, c3, c1, c2, \#1$ just means that the destination is R7 and the sources are R5 and R6
- For the handler it doesn't make any difference if you put [op1], [op2], [dest] in R4, R5, ...

5) Computing the result

- Bit 5 in the encoded instruction (representing the sign) must be the most significant bit of the result.
- The other bits are set by computing exponent and mantissa.
- Finally, update (STR) the destination register saved in the stack with the new computed value.

6) Concluding the handler

- Restore values of register $r4-r11$ with a `POP`.
 - The destination register will contains the IEEE-754 SP representation, since the corresponding entry in the stack was updated in step 5.
- Return to the program with `BX LR`.
 - the next instruction will be the one after `CDP`, since the value of `PC` (automatically retrieved from the stack) has been updated in step 3 and will be automatically popped.