Interrupts and Traps in Oberon-ARM

Niklaus Wirth

22.2.2008

1. Interrupts, and the ARM-Architecture

An interrupt is by definition a break in the sequential stream of instruction execution caused by an external signal. It is useful to consider an interrupt as causing the execution of a procedure. The programmer's problem is that this procedure may be inserted anywhere in the normal program flow. An interrupt should be considered as a procedure call, which may occur at any time, anywhere, asynchronously. It is the programmer's duty to ensure that such procedures, called *interrupt handlers*, cooperate harmoniously with the remaining program.

In the ARM-processor, there are 6 sources of interrupt signals. They are: *Fast Interrupt* (FIQ), *Interrupt* (IRQ), *Software Interrupt* (SWI), *Undefined Instruction*, and *Abort*. The first two are caused by external signals, the third by the SWI instruction, and the last by internal conditions. When an interrupt signal is sensed, the ARM processor picks the next instruction from a fixed location, which is called the *interrupt vector*. Every interrupt source has its own associated interrupt location. The locations are therefore called a vector.

The processor is said to be in a *mode*. This mode is represented by the *Processor Status Register* (8 bits). The mode is set whenever an interrupt happens. The PSR also contains two bits which, when set to 1, disable interrupts from the external sources.

In detail, the following actions are taken when an interrupt condition is sensed:

1. The address of the current instruction + 8 is saved in register R14 (LNK), just as in the case of a Branch and Link instruction, and the PSR is saved in the SPSR register: PC -> LNK, PSR -> SPSR.
2. The interrupt location's address is assigned to the PC, and the mode associated with the interrupt source is assigned to the PSR: intadr -> PC, intmode -> PSR.
3. In case of an external interrupt, the corresponding disable bit in the PSR is set in order to prevent a recursive interrupt.

The following table specifies the vector locations and the offsets required in the return instruction for the various interrupt sources.

Interrupt	vect. location	offset
FIQ	28	-4
IRQ	24	-4
Abort data	16	-8
Abort instr.	12	-4
SWI	8	0
Und. Instr.	4	0

The ARM's structure is in fact somewhat more complex. It features for each mode its own link and saved mode registers (LNK, SPSR). The processor automatically uses the pair belonging to the current mode. For the following, however, we may ignore this, as if there were only one pair for all modes.

2. Interrupts in Oberon-ARM

In Oberon, interrupts are handled by procedures. Because their end must not consist of a simple branch and link instruction (BL), which restores the PC, but by one that also restores the PSR, an interrupt procedure must be specially marked. This is done by an offset specification enclosed in brackets in place of a regular parameter list.

In addition, the procedure must be “installed” in the corresponding vector location. This is done by placing a branch instruction (BR) in that location using the SYSTEM.PUT operator. The address field of the jump must contain the offset of the interrupt procedure from the interrupt location.

The following example shows a handler of the IRQ signal, which is assumed to be driven by a timer. It causes an LED to blink once per second. LED and TIM are the addresses of a light emitting diode and of the timer respectively. *Count* and *sense* are variables used by the particular handler.

```
MODULE TestIRQ;  (*NW 25.8.98*)
   IMPORT SYSTEM;  (*Pulse LED, 1ms timer interrup*)
   CONST IRQvec = 18H; LED = 3000024H; TIM = 3000000H; SVstk = 800h;
   VAR count, sense: INTEGER;

   PROCEDURE Handle [4];
   BEGIN INC(count); SYSTEM.PUT(TIM, 0);  (*reset int req*)
      IF count = 500 THEN SYSTEM.PUT(LED, sense); count := 0; sense := 1 - sense END
   END Handle;

BEGIN count := 0; sense := 1; (*system initialization; install handler*)
   SYSTEM.PUT(IRQvec, (SYSTEM.ADR(Handle) - IRQvec - 8) DIV 4 + 0EA000000H);
   SYSTEM.LDPSR(0, 0D2H); SYSTEM.SP := SVstk;  (*set IRQ-stack pointer SP*)
   SYSTEM.LDPSR(0, 53H);  (*enable IRQ interrupt and assume supervisor mode*)
   REPEAT UNTIL FALSE  (*main program, idle*)
END TestIRQ.
```

The compiled code for the handler is shown below.

```
 1   E92D5FFF    STM   SP              save registers R0-R11, FP, LNK
 2   E1A0C00D    MOV   FP R0 SP        FP := SP
 3   E51FB018    LDR   R11 PC -24      INC(count)
 4   E28BB001    ADD   R11 R11 1
 5   E50FB020    STR   R11 PC -32
 .....
20   E1A0D00C    MOV   SP R0 FP        SP := FP
21   E8BD5FFF    LDM   SP              restore registers R0-R11, FP, LNK
22   E25EF004    SUB   PC LNK 4        restore PSR, PC := LNK
```

In the case of the fast interrupt FIQ, the processor switches registers. It uses a fresh set of registers R8 – R14. This makes it unnecessary to save those registers to memory and to restore them on exit. In order to make use of this property in Oberon, an interrupt handler may be marked by an asterisk like a leaf procedure. Local variables are then allocated in R11, R10, R9, R8. As in leaf procedures, only variables of types INTEGER and SET are admitted.

In order to make this interrupt as fast as possible, no registers are saved at all. Hence, no FP is valid, and therefore no local variables in memory must be declared. These restrictions are not checked by the compiler!

3. Traps in Oberon-ARM

Traps are handled by the processor through the same mechanism as interrupts, although their meaning is quite different. Whereas interrupts (caused by external signals, here IRQ and FIQ) call for an action, typically a short action) and a subsequent return to the interrupted program, traps (caused by internal signals reporting a failure) require a transfer to a specific program point, possibly after execution of a recovery routine. Therefore, whereas genuine interrupts correspond to a procedure call, traps correspond to a jump, a GOTO statement, an exception.

How is a jump without return achieved in Oberon? There exists no explicit language construct for this purpose. Evidently, one must resort to the means of accessing system registers through use of PUT and GET operators.

The following example shows how this is done for the case of the Software Interrupt (SWI). Typically, there is exactly one trap handler in a system. The example must therefore be considered as the core of a system with a single, fixed return point from traps. In this example, the handler sends out the low byte of the instruction that caused the trap. We recognize that the

handler is essentially like the interrupt handler of the preceding section. However, it terminates by a jump to the *resumption point* in the body of the system.

This is achieved by assigning the resumption address to the PC register and the desired mode to the PSR. The latter is done by the LDPSR intrinsic procedure. Note that this procedure cannot be executed in user mode. It is feasible here, because a software trap puts the processor into supervisor mode.

Three global variables called *FPsv, SPsv, PCsv* serve to hold the values which the three registers *FP, SP, and PC* must have at the resumption point. The three values are assigned to the respective registers at the end of the trap handler.

```
MODULE System;
   IMPORT SYSTEM, IO;
   CONST trapvec = 8; mode = 0D0H; StkOrg = 300000H;
   VAR FPsv, SPsv, PCsv, n: INTEGER;

   PROCEDURE Trap [0];
      VAR adr, instr: INTEGER;
   BEGIN adr := SYSTEM.LNK;  (*adr of trap instruction + 4*)
      SYSTEM.GET(adr-4, instr); IO.SendStr("trap"); IO.SendInt(instr MOD 100H); IO.End;
      SYSTEM.LDPSR(0, mode);
      SYSTEM.FP := FPsv; SYSTEM.SP := SPsv; SYSTEM.PC := PCsv  (*transfer*)
   END Trap;

   PROCEDURE Q(n: INTEGER);
   BEGIN ASSERT(n < 10, 34); IO.SendStr(" ok"); IO.End
   END Q;

BEGIN (*system initialization; install trap vector*)
   SYSTEM.PUT(trapvec, (SYSTEM.ADR(Trap) - trapvec - 8) DIV 4 + 0EA000000H);
   SYSTEM.LDPSR(0, mode); SYSTEM.SP := StkOrg; SYSTEM.FP := StkOrg;
   FPsv := SYSTEM.FP; SPsv := SYSTEM.SP;
   PCsv := SYSTEM.PC + 0; (*return point*)
   REPEAT IO.RecInt(n); Q(n) UNTIL FALSE (*main program*)
END System.
```

The compiled code corresponding to the trap handler is shown below.

```
 1   E92D5FFF     STM    SP                save registers R0-R11, FP, LNK
 2   E1A0C00D     MOV    FP R0 SP          FP := SP
 3   E24DD008     SUB    SP SP 8
 4   E50CE004     STR    LNK FP -4         adr := FP
 …..
16   E3A0BE0D     MOV    R11 R0 208
17   E129F00B     CMN    PC R9 R11         PSR := mode
18   E51FC054     LDR    FP PC -84         FP := FPsv
19   E51FD05C     LDR    SP PC -92         SP := SPsv
20   E51FF064     LDR    PC PC -100        PC := PCsv,  transfer
21   E1A0D00C     MOV    SP R0 FP          not executed
…..
```

The Oberon compiler generates *SWI n* instructions for the assert statement "ASSERT *cond, n"*, and for various exceptions with the following identification numbers *n*:

n	cause
1	index out of bounds
2	type test failure
3	destination array shorter than source array
4	invalid value in case statement
5	
6	string too long or destination array too short
7	integer division by zero or negative divisor

In order that traps will not be generated recursively, the compiler suppresses index checks in interrupt handlers. It does so too in leaf procedures.