Διακοπές Λογισμικού - SoftWare Interrupts (SWIs)

Γ.Κορνάρος – Σ.Μπλιώνας

Περίγραμμα

- Τι είναι ένα **S**oft**W**are **I**nterrupt (SWI)?
  - ✓What are SWIs useful for?
  - ✓What happens on an SWI?
  - ✓What happens on SWI completion?
  - ✓How to install a SWI handler
- Lab 1 Handout on BBoard

Εξαιρέσεις και Διακοπές

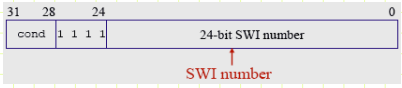
- Οι όροι *exception* and *interrupt συχνά συγχέονται*
- *Ένα Exception* συνήθως αναφέρεται σε ένα εσωτερικό γεγονός στη CPU όπως :
  - ✓ floating point overflow
  - ✓ MMU fault (e.g., page fault)
  - ✓ trap (SWI)
- *Ένα Interrupt* συνήθως αναφέρεται σε ένα γεγονός I/O όπως:
  - ✓ I/O device request
- Στα εγχειρίδια αρχιτεκτονικής του ARM, οι δύο όροι αναφέρονται χωρίς διάκριση
  - ✓ Και για αυτό το μάθημα exceptions and interrupts θα σημαίνουν το ίδιο πράγμα

Τι είναι τα SWI ?

- **S**oft**W**are **I**nterrupt (SWI) είναι μία εντολή που παρέχεται από το ARM Instruction Set
- SWI instruction causes an exception
  - ✓ Executes in the privileged mode
- Similar to a sub-routine call because
  - ✓ Parameters and return values can be passed through registers
- Differs from a sub-routine call because the ARM processor
  - ✓ Stashes away user mode CPSR
  - ✓ Switches to supervisor mode (if not already in supervisor mode)
  - ✓ Starts executing from a specific location (always 0x08)
- Τα SWI παρέχουν ένα τρόπο για ένα πρόγραμμα που τρέχει σε User mode να ζητήσει ένα service που μπορεί μόνο να προσπελαστεί σε privileged mode
  - ✓ Typically, used by applications to call operating system routines

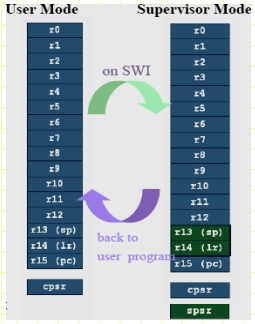
Μορφή εντολής SWI

- Example: SWI 0x18
  - ✓The number identifies the handler to be called
  - ✓Example:
    - o 0x18 could refer to a SWI handler that reads key input whereas
    - o 0x19 could refer to a SWI handler that outputs a char etc. …
- The last 24-bits of the SWI instruction contain the SWI number

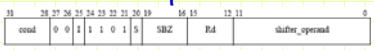


Χειρισμός SWI

- Έστω αρχικά σε User mode
- Όταν συμβεί μία εξαίρεση, ο **ARM processor**
  - ✓ Copies cpsr into spsr_svc
  - ✓ Sets appropriate cpsr mode bits to 10011 (svc mode)
- Disable IRQs
  - ✓ Stores return address (pc – 4) in lr_svc
  - ✓ Sets pc to vector address 0x08
- To return, **exception handler** needs to:
  - ✓ Restore cpsr from spsr_svc (cpsr now has mode bits 10000 = usr mode)
  - ✓ Restore pc from lr_svc
- In privileged modes, a single instruction can be used to cause the SPSR for the current mode to be copied into CPSR while copying lr into pc
  - ✓ MOVS pc, lr



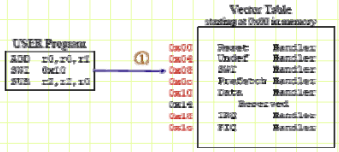
Εντολή MOV



- The MOV (Move) instruction moves the value of <shifter_operand> to the destination register <Rd>. The condition code flags are optionally updated, based on the result.
- **Syntax:** MOV{<cond>}{S} <Rd>, <shifter_operand>
- where:
  - ✓ <cond> Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-5. If <cond> is omitted, the AL (always) condition is used.
  - ✓ S Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If S is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction.
  - ✓ Two types of CPSR update can occur when S is specified:
    - o If <Rd> is not R15, the N and Z flags are set according to the value moved (post-shift if a shift is specified), and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
    - o If <Rd> is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.
  - ✓ <Rd> Specifies the destination register of the instruction.

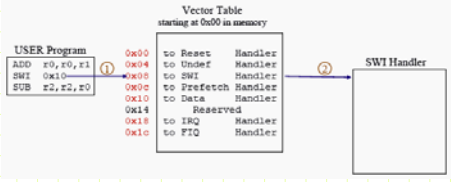
Τι συμβαίνει σε ένα SWI

- The ARM architecture defines a **Vector Table** indexed by exception type
- On SWI
- – lr_svc ⇦ pc-4
- – spsr_svc ⇦ cpsr
- – cpsr(mode bits) ⇦ 10011 (supervisor mode, no IRQ)
- – pc ⇦ 0x08 (address of SWI vector)



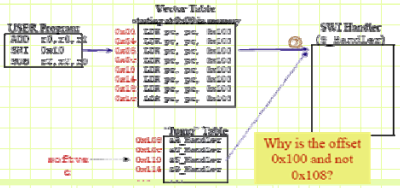
Τι συμβαίνει σε ένα SWI (2)

- Δεν υπάρχει αρκετός χώρος στο table (**only one instruction per entry**) για να χωρέσει όλος ο κώδικας για την συνάρτηση του SWI handler
- This *one* instruction must transfer control to appropriate SWI Handler



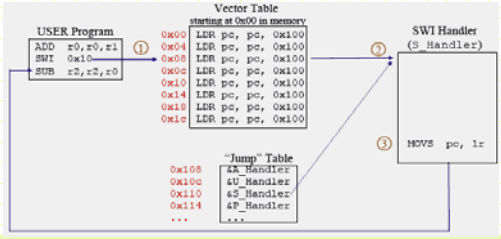
"Vectoring" Exceptions to Handlers

- **One option**: Use a branch instruction (limited range because branch offset must be within 32 MB of branch instruction)
- **Option of choice**: Load the address of the actual handler into the pc using LDR instruction
  - ✓ A memory location (called softvec) stores the address of the SWI handler (S_Handler)
  - ✓ Use LDR instruction to load pc with **contents** of softvec
- • Compute the offset (#offset) of softvec from SWI vector and use the LDR pc, [pc, #offset] to load the address of S_Handler into pc



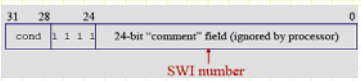
Τι συμβαίνει μετά το SWI ?

- When the handler is done, it returns to the USER program -- at the instruction following the SWI
- MOVS restores the original CPSR as well as changing pc



Φορμάτ εντολής SWI

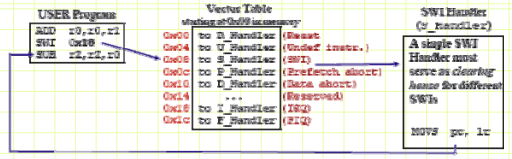
- Ο ARM processor αγνοεί τα τελευταία 24-bits της εντολής SWI instruction (τα οποία όμως περιέχουν τον SWI number)



- Exercise – how would we extract the SWI number, if we knew that the 32-bit representation of the instruction (SWI 0x18) was stored in a register (say r0)?
  - ✓ We will discuss how to get the instruction encoding into r0 later
- If r0 stored the 32-bit representation of the instruction (SWI 0x18) how would you extract the SWI number
  - ✓ You want the last 24 bits of the instruction
  - ✓ Use bit-clearing (BIC) to zero out the 8 MSBs
  - ✓ BIC r0, r0, #0xff000000; r0 ← r0 AND NOT(0xff000000)
  - ✓ This effectively sets r0 to the result of (r0 AND ~0xff000000)

Determining the SWI number – I

- **Problem**: All SWIs go to 0x08 regardless of the number that you put after the SWI instruction
  - ✓ What's the point of that number, then?
  - ✓ How do you actually find out which SWI to execute if they all end up at the same address?
- When you are at S_Handler in svc mode
  - ✓ Is it possible for you to find out where you came from in usr mode?
  - ✓ Can you exploit this information to find out which SWI you should execute?

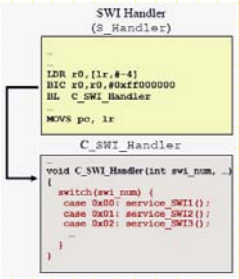


Determining the SWI number – II

- Take a look at lr_svc
  - ✓ Holds information about the instruction in user space that immediately follows the SWI (done by the processor at the time that it switches into svc mode)
  - ✓ If you step back 4 bytes from lr_svc, what do you get?
    - o LDR r0,[lr,#-4]
  - ✓ r0 stores the 32-bit encoding of the SWI instruction, in this case, SWI 0x18
    - o Last 24 bits ("comment field) contains the SWI number
  - ✓ Extract the SWI number now by clearing out everything but the last 24 bits
  - ✓ BIC r0,r0,#0xff000000
- Bingo! We now have the SWI number in r0

Use The SWI # to Jump to "Service Routine"

- Key Insight – need access to general purpose register (lr in this case)
  - ✓ Initial part of the SWI Handler has to be written in assembly
  - ✓ Exit also in assembly
- Μετά που ανακαλύψουμε τον SWI number και τον αποθηκεύσουμε στον r0
  - ✓ Call a C function from the assembly code to service the specific SWI
  - ✓ Easier to write SWI handler in C rather than assembly
- Remember ATPCS conventions
  - ✓ r0 contains the first argument of a function call
- **Προσοχή**:
  - ✓ In S_Handler the address of the return instruction (to user program) is stored in lr_svc
  - ✓ BL call will overwrite the value of lr_svc
  - ✓ Need to save lr_svc in S_Handler before using BL



SWI handler

```
S_Handler
   STMFD   sp!, {r0-r12, lr} ;store user's gp registers and lr_svc
   MOV     r1, sp  ;r1 now contains pointer to parameters on stack
   LDR     r0, [lr, #-4]     ; extract the SWI number
   BIC     r0,r0,#0xff000000 ; get SWI # in r0 by bit-masking
   BL      C_SWI_handler     ; go to handler (see next slide)
   LDMFD   sp!, {r0-r12, lr} ; unstack user's registers and lr_svc
   MOVS    pc, lr            ; return from handler
```


C_SWI_Handler

```
void C_SWI_handler(unsigned swi_num,
   unsigned *regs)
{
    switch (swi_num) {
    case 0: /* SWI number 0 code */
       break;
    case 1: /* SWI number 1 code */
       break;
    ...
    case XXX: /* SWI number XXX code */
       break;
    default:
    } /* end switch */
} /* end C_SWI_handler() */
```

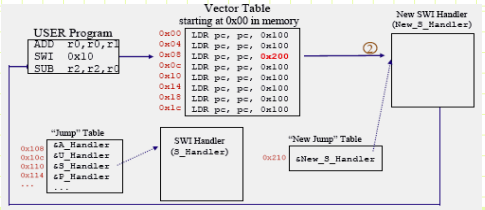
Ο ARM έχει έναν Angel…

- Ok, άρα τώρα γνωρίζουμε τι κάνει ο ARM όταν συναντήσει μία SWI instruction
  - ✓ But, who installs S_Handler and initializes the contents of softvec?
- ARM has developed a debug monitor called Angel which provides minimal OS-like functionality
  - ✓ Angel goes through the system initialization (bootstrapping) process
  - ✓ Setup hardware on startup
  - ✓ Load and run programs
  - ✓ Initialize stack pointers

- Angel installs S_Handler to support some SWIs
- What if needed to install a custom SWI not handled by Angel?
  - ✓ Need to install our own SWI handler

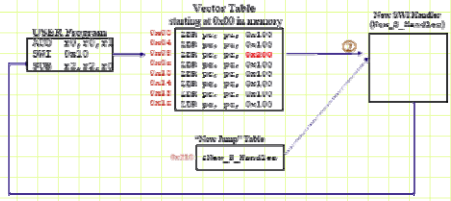
Εγκατάσταση του δικού σου SWI handler

- Δύο τρόποι για να εγκαταστήσεις τον δικό σου SWI handler:
- Method 1 (to be done in lab)
  - ✓ Change the address of S_Handler (stored in softvec) to point to the new SWI handler
- Method 2 (done in lecture)
  - ✓ Define a new location for softvec
  - ✓ Store the address of the new SWI handler in softvec
  - ✓ Change the instruction stored in vector table location 0x08 to point to the new softvec



Εγκατάσταση του SWI handler

- Load pc method
  - ✓ Compute the offset to be used in the LDR instruction (at 0x08)
    - o Take the address of the word containing the address of the new SWI handler
    - o Subtract the address of the SWI exception vector in the vector table
    - o Subtract 8 bytes
    - o Check that the result can be represented in 12 bits
  - ✓ Logically OR this with 0xe59FF000 (the opcode for LDR pc, [pc, #0])
  - ✓ Store this new LDR instruction in the SWI exception vector



Φόρτωμα του Πίνακα Διανυσμάτων

```
unsigned Install_Handler(unsigned location, unsigned int *vector)
/*Updates the contents of 'vector' stored at 0x08 to contain LDR pc, [pc, #offset]
   instruction to cause long branch to address in 'location' */
/*Function returns the original contents of 'vector' */
{
   unsigned offset;
   unsigned vec, oldvec;
   offset = ((unsigned) location - (unsigned) vector - 0x8)
   if(offset & 0xFFFFF000)  /* check if the offset can be represented using 12 bits */
   {     printf("Installation of handler failed");
         exit(0); }
   vec =(offset|0xe59FF000);/* vec now contains LDR pc, [pc, #offset] */
   oldvec = *vector;        /* save the old instruction at 0x08 */
   *vector = vec;           /* replace the contents of 0x08 with the new LDR instruction */
   return(oldvec);          /* return the old instruction at 0x08 for chaining */
}
```

The following code calls function this to install an SWI handler

```
unsigned *swivec = (unsigned *) 0x08;
unsigned *swiaddr= (unsigned *) 0x210; /*The address of the new softvec */

*swiaddr= (unsigned) New_S_Handler;//Store the address of your New_S_Handler in swiaddr
Install_Handler((unsigned) swiaddr, swivec);
```


Κλήση SWI από C κώδικα

User-Level C Source Code

```
char __swi(4) SWI_ReadC(void);
void readline (char *buffer)
{
   char ch;
   do {
     *buffer++ = ch =
     SWI_ReadC();
   while (ch != 13);
   }
   *buffer = 0;
} /* end readline() */
```

Assembly code produced by compiler

```
readline
        STMDF   sp!,{lr}
        MOV     lr, a1
readagain
        SWI     0x04
        STRB    a1,[lr],#1
        CMP     a1,#&d
        BNE     readagain
        MOV     a1,#0
        STRB    a1, [lr, #0]
        LDMIA   sp!, {pc}
```


Δια-συνδέοντας χειριστές Εξαιρέσεων

- Some SWIs are already implemented by Angel running on the ARM processor (Angel to be covered soon)
- Applications may want to support additional SWIs but may not want to rewrite the default SWI implementations provided by Angel
- The application's SWI handler should call the Angel SWI handler if it cannot handle a particular SWI
- This approach is called *chaining* exception handlers
  - ✓ Can be extended to many levels to build a chain of handlers
- The S_Handler and the C_SWI_handler presented in the previous lecture do not exhibit chaining
  - ✓ The SWI handler that you are required to write for Lab 1 has to support chaining
  - ✓ Extract the address of the old handler returned by Install_Handler

Reentrant Exception Handlers

- Reentrant programming – χρησιμοποιείται για να περιγράψει κώδικα ο οποίος μπορεί να έχει nested κλήσεις (invocations) οι οποίες ιδανικά δεν θα πρέπει να εμπλέκονται μεταξύ τους
- Τι συμβαίνει αν θες να καλέσεις ένα άλλο SWI μέσα από τον SWI handler σου ?
  - ✓ When an SWI occurs, the ARM
    - o Sets appropriate CPSR mode bits to 10011 (svc mode)
    - o Copies CPSR into spsr_svc
    - o Stores return address (pc – 4) in lr_svc
    - o Sets pc to vector address 0x08
- Πρέπει να αποθηκεύσεις συγκεκριμένους καταχωρητές (that do not need to be saved otherwise)
- Do you see any issues here?
- The SWI handler you will write for Lab 1 has to be reentrant

A Reentrant SWI Handler for Lab1

```
import C_SWI_Handler
export S_Handler
S_Handler
    SUB     sp, sp, #4          ; leave room on stack for SPSR
    STMFD   sp!, {r0-r12, lr}   ; store user's gp registers
    MRS     r2, spsr            ; get SPSR into gp registers
    STR     r2, [sp, #14*4]     ; store SPSR above gp registers
    MOV     r1, sp              ; pointer to parameters on stack
    LDR     r0, [lr, #-4]       ; extract the SWI number
    BIC     r0,r0,#0xff000000   ; get SWI # by bit-masking
    BL      C_SWI_Handler       ; goto handler (see prev lecture)
    LDR     r2, [sp, #14*4]     ; restore SPSR (NOT "sp!")
    MSR     spsr, r2            ; restore SPSR from r2
    LDMFD   sp!, {r0-r12, lr}   ; unstack user's registers
    ADD     sp, sp, #4          ; remove space used to store SPSR
    MOVS    pc, lr              ; return from handler
```

- *This SWI handler does not exhibit chaining*