ARM Exceptions

Hsung-Pin Chang
Department of Computer Science
National Chung Hsing University
Outline

- ARM Exceptions
- Entering and Leaving an Exception
- Installing an Exception Handler
- SWI Handlers
- Interrupt Handlers
- Reset Handlers
- Undefined Instruction Handlers
- Prefetch Abort Handler
- Data Abort Handler
ARM Exceptions

- ARM Exception Types
- ARM Exception Vector Table
- ARM Exception Priorities
- Use of Modes and Registers by Exceptions
ARM Exception Types

- Reset
- Undefined instruction
- Software Interrupt (SWI)
- Prefetch Abort
- Data Abort
- IRQ
- FIQ
ARM Exceptions Types (Cont.)

- **Reset**
  - Occurs when the processor reset pin is asserted
    - For signaling Power-up
    - For resetting as if the processor has just powered up
  - *Software reset*
    - Can be done by branching to the reset vector (0x0000)

- **Undefined instruction**
  - Occurs when the processor or coprocessors cannot recognize the currently execution instruction
ARM Exceptions Types (Cont.)

- **Software Interrupt (SWI)**
  - User-defined interrupt instruction
  - Allow a program running in User mode to request privileged operations that are in Supervisor mode
    - For example, RTOS functions

- **Prefetch Abort**
  - Fetch an instruction from an illegal address, the instruction is flagged as invalid
  - However, instructions already in the pipeline continue to execute until the invalid instruction is reached and then a Prefetch Abort is generated.
ARM Exceptions Types (Cont.)

- **Data Abort**
  - A data transfer instruction attempts to load or store data at an illegal address

- **IRQ**
  - The processor external interrupt request pin is asserted (LOW) and the I bit in the CPSR is clear (enable)

- **FIQ**
  - The processor external fast interrupt request pin is asserted (LOW) and the F bit in the CPSR is clear (enable)
Vector Table

- At the bottom of the memory map

- Each entry has only 32 bit
  - Not enough to contain the full code for a handler
  - Thus, usually is a *branch instruction* or *load pc instruction* to the actual handler

- Example: armc_startup.s
## ARM Exception

<table>
<thead>
<tr>
<th>Exception type</th>
<th>Mode</th>
<th>Normal address</th>
<th>High vector address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>0x00000000</td>
<td>0xFFFFF0000</td>
</tr>
<tr>
<td>Undefined instructions</td>
<td>Undefined</td>
<td>0x00000004</td>
<td>0xFFFFF0004</td>
</tr>
<tr>
<td>Software interrupt (SWI)</td>
<td>Supervisor</td>
<td>0x00000008</td>
<td>0xFFFFF0008</td>
</tr>
<tr>
<td>Prefetch Abort (instruction fetch memory abort)</td>
<td>Abort</td>
<td>0x0000000C</td>
<td>0xFFFFF000C</td>
</tr>
<tr>
<td>Data Abort (data access memory abort)</td>
<td>Abort</td>
<td>0x00000010</td>
<td>0xFFFFF0010</td>
</tr>
<tr>
<td>IRQ (interrupt)</td>
<td>IRQ</td>
<td>0x00000018</td>
<td>0xFFFFF0018</td>
</tr>
<tr>
<td>FIQ (fast interrupt)</td>
<td>FIQ</td>
<td>0x0000001C</td>
<td>0xFFFFF001C</td>
</tr>
</tbody>
</table>
### ARM Exception Vector Table

<table>
<thead>
<tr>
<th>Exception</th>
<th>Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWI handler</td>
<td>...</td>
</tr>
<tr>
<td>IRQ handler</td>
<td>....</td>
</tr>
<tr>
<td>FIQ</td>
<td>0x1C</td>
</tr>
<tr>
<td>IRQ</td>
<td>0x18</td>
</tr>
<tr>
<td>Reserved</td>
<td>0x14</td>
</tr>
<tr>
<td>Data Abort</td>
<td>0x10</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>0x0C</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>0x08</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>0x04</td>
</tr>
<tr>
<td>Reset</td>
<td>0x00</td>
</tr>
</tbody>
</table>

(1) Exception Vector Table

(2)
## ARM Exception Events

<table>
<thead>
<tr>
<th>Address</th>
<th>Exception</th>
<th>Mode in Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Reset</td>
<td>Supervisor</td>
</tr>
<tr>
<td>0x00000004</td>
<td>Undefined instruction</td>
<td>Undefined</td>
</tr>
<tr>
<td>0x00000008</td>
<td>Software Interrupt</td>
<td>Supervisor</td>
</tr>
<tr>
<td>0x0000000C</td>
<td>Abort (prefetch)</td>
<td>Abort</td>
</tr>
<tr>
<td>0x00000010</td>
<td>Abort (data)</td>
<td>Abort</td>
</tr>
<tr>
<td>0x00000014</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x00000018</td>
<td>IRQ</td>
<td>IRQ</td>
</tr>
<tr>
<td>0x0000001C</td>
<td>FIQ</td>
<td>FIQ</td>
</tr>
</tbody>
</table>
# ARM Exception Priorities

<table>
<thead>
<tr>
<th>Vector address</th>
<th>Exception type</th>
<th>Exception mode</th>
<th>Priority (1=high, 6=low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Reset</td>
<td>Supervisor (SVC)</td>
<td>1</td>
</tr>
<tr>
<td>0x4</td>
<td>Undefined Instruction</td>
<td>Undef</td>
<td>6</td>
</tr>
<tr>
<td>0x8</td>
<td>Software Interrupt (SWI)</td>
<td>Supervisor (SVC)</td>
<td>6</td>
</tr>
<tr>
<td>0xC</td>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>5</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort</td>
<td>Abort</td>
<td>2</td>
</tr>
<tr>
<td>0x14</td>
<td>Reserved</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>0x18</td>
<td>Interrupt (IRQ)</td>
<td>Interrupt (IRQ)</td>
<td>4</td>
</tr>
<tr>
<td>0x1C</td>
<td>Fast Interrupt (FIQ)</td>
<td>Fast Interrupt (FIQ)</td>
<td>3</td>
</tr>
</tbody>
</table>
Use of Modes and Registers by Exceptions

- An exception changes the processor mode
- Thus, each exception handler has access to a certain subset of banked registers
  - Its own r13 or Stack Pointer (r13_mode or sp_mode)
  - Its own r14 or Link Register (r14_mode or lr_mode)
  - Its own Saved Program Status Register (SPSR_mode).
Register Organization in ARM States

**ARM State General Registers and Program Counter**

<table>
<thead>
<tr>
<th>System &amp; User</th>
<th>FIQ</th>
<th>Supervisor</th>
<th>About</th>
<th>IRG</th>
<th>Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R8_fiq</td>
<td>R8_fiq</td>
<td>R8</td>
<td>R8_fiq</td>
<td>R8_fiq</td>
</tr>
<tr>
<td>R9</td>
<td>R9_fiq</td>
<td>R9_fiq</td>
<td>R9</td>
<td>R9_fiq</td>
<td>R9_fiq</td>
</tr>
<tr>
<td>R10</td>
<td>R10_fiq</td>
<td>R10_fiq</td>
<td>R10</td>
<td>R10_fiq</td>
<td>R10_fiq</td>
</tr>
<tr>
<td>R11</td>
<td>R11_fiq</td>
<td>R11_fiq</td>
<td>R11</td>
<td>R11_fiq</td>
<td>R11_fiq</td>
</tr>
<tr>
<td>R12</td>
<td>R12_fiq</td>
<td>R12_fiq</td>
<td>R12</td>
<td>R12_fiq</td>
<td>R12_fiq</td>
</tr>
<tr>
<td>R13</td>
<td>R13_fiq</td>
<td>R13_fiq</td>
<td>R13</td>
<td>R13_fiq</td>
<td>R13_fiq</td>
</tr>
<tr>
<td>R14</td>
<td>R14_fiq</td>
<td>R14_fiq</td>
<td>R14</td>
<td>R14_fiq</td>
<td>R14_fiq</td>
</tr>
</tbody>
</table>

**ARM State Program Status Register**

- CPSR
- CPSR_fiq
- CPSR_svc
- CPSR_abt
- CPSR_irq
- CPSR_und

= banked register
Entering and Leaving an Exception

- The Process Response to an Exception
- Returning from an Exception Handler
- The Return Address and Return Instruction
The Process Response to an Exception

- Copies the CPSR into the SPSR for the mode in which the exception is to be handled.
  - Saves the *current mode*, *interrupt mask*, and *condition flags*.
- Changes the appropriate CPSR mode bits
  - Change to the appropriate mode
- Map in the appropriate banked registers for that mode
- Disable interrupts
  - IRQs are disabled when any exception occurs.
  - FIQs are disabled when a *FIQ occurs*, and *on reset*
- Set *lr_mode* to the return address
  - Discuss in the next few slides
- Set the program counter to the *vector address* for the exception
The Process Response to an Exception (Cont.)

- For example, when reset, ARM
  - Overwrites $R14_{svc}$ and $SPSR_{svc}$ by copying the current values of the PC and CPSR into them
  - Forces $M[4:0]$ to 10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR's T bit
  - Forces the PC to fetch the next instruction from address 0x00.
  - Execution resumes in ARM state.
The Process Response to an Exception (Cont.)

<table>
<thead>
<tr>
<th></th>
<th>Reset</th>
<th>Undefined Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R14_svc = unexpected</td>
<td>R14_und = PC+4</td>
</tr>
<tr>
<td></td>
<td>SPSR_svc = unexpected</td>
<td>SPSR_und = CPSR</td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10011 //Supervisor Mode</td>
<td>CPSR[4:0] = 0b11011 //Undefined Mode</td>
</tr>
<tr>
<td></td>
<td>PC = 0x00000000</td>
<td>PC = 0x00000004</td>
</tr>
</tbody>
</table>
## The Process Response to an Exception (Cont.)

| Exception Type          | R14
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Interrupt</strong></td>
<td>R14_svc = PC + 4</td>
</tr>
<tr>
<td></td>
<td>SPSR_svc = CPSR</td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10011 //Supervisor Mode</td>
</tr>
<tr>
<td></td>
<td>CPSR[5] = 0 // ARM state</td>
</tr>
<tr>
<td></td>
<td>CPSR[6] unchanged</td>
</tr>
<tr>
<td></td>
<td>CPSR[7] = 1 // Disable IRQ</td>
</tr>
<tr>
<td></td>
<td>PC = 0x00000008</td>
</tr>
<tr>
<td><strong>Prefetch Abort</strong></td>
<td>R14_abt = PC+4</td>
</tr>
<tr>
<td></td>
<td>SPSR_abt = CPSR</td>
</tr>
<tr>
<td></td>
<td>CPSR[4:0] = 0b10111 //Abort Mode</td>
</tr>
<tr>
<td></td>
<td>CPSR[5] = 0</td>
</tr>
<tr>
<td></td>
<td>CPSR[6] unchanged</td>
</tr>
<tr>
<td></td>
<td>CPSR[7] = 1 // Disable IRQ</td>
</tr>
<tr>
<td></td>
<td>PC = 0x000000C</td>
</tr>
</tbody>
</table>
### The Process Response to an Exception (Cont.)

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Data Abort**       | R14\_abt = PC + 8  
SPSR\_abt = CPSR  
CPSR[4:0] = 0b10111 //Abort Mode  
CPSR[5] = 0 // ARM state  
CPSR[6] unchanged  
CPSR[7] = 1 // Disable IRQ  
PC = 0x00000010 |
| **Interrupt Request**| R14\_abt = PC+4  
SPSR\_abt = CPSR  
CPSR[4:0] = 0b10011 //Abort Mode  
CPSR[5] = 0  
CPSR[6] unchanged  
CPSR[7] = 1 // Disable IRQ  
PC = 0x00000018 |
The Process Response to an Exception (Cont.)

| Fast Interrupt Request | R14_abt = PC + 4  
| SPSR_abt = CPSR  
| CPSR[4:0] = 0b10010 //IRQ Mode  
| CPSR[5] = 0 // ARM state  
| CPSR[6] = 1 //Disable FIQ  
| CPSR[7] = 1 // Disable IRQ  
| PC = 0x0000001C |
Returning From an Exception Handler

- Returning from an exception handler
  - Depend on whether the exception handler uses the stack operations or not

- Generally, to return execution to the *original execution place*
  - Restore the CPSR from spsr_mode
  - Restore the program counter using the return address stored in lr_mode
Returning From an Exception Handler:
Simple Return

- If not require the destination mode registers to be restored from the stack
  - Above two operations can be carried out by a data processing instruction with
    - The S flag (bit 20) set
      - Update the CPSR flags when executing the data processing instruction
      - SUBS, MOVS
    - The program counter as the destination register
  - Example: MOVS pc, lr  //pc = lr
Returning From an Exception Handler: Complex Return

- If an exception handler entry code uses the stack to store registers
  - Must be preserved while handling the exception

- To return from such an exception handler, the stored register must be restored from the stack
  - Return by a *load multiple instruction* with ^ qualifier
  - For example: LDMFD sp!, {r0-r12,pc}^
Returning From an Exception Handler

- Note, do not need to return from the reset handler
  - The reset handler executes your main code directly

- The actual location when an exception is taken depends on the exception type
  - The return address may not necessarily be the next instruction pointed to by the pc
Returning from SWI and Undefined Instruction Handlers

- SWI and undefined instruction exceptions are generated by the instruction itself
  - `lr_mode = pc + 4 //next instruction`

- Restoring the program counter
  - If not using stack: `MOV pc, lr //pc = lr`
  - If using stack to store the return address
    - `STMFD sp!, {reglist, lr} //when entering the handler`
    - `LDMFD sp!, {reglist, pc}^ //when leaving the handler`
Returning from FIQ and IRQ

- FIQ and IRQ are generated only after the execution of an instruction
  - The program counter has been updated
    - $lr_{mode} = PC + 4$
      - Point to one instruction beyond the end of the instruction in which the exception occurred
Returning from FIQ and IRQ (Cont.)

- Restoring the program counter
  - If not using stack: \texttt{SUBS pc, lr, #4} \hfill //\texttt{pc = lr-4}
  
  - If using stack to store the return address
    \texttt{SUB lr, lr, #4} \hfill //when entering the handler
    \texttt{STMFD sp!, \{reglist, lr\}}
    ...
    \texttt{LDMFD sp!, \{reglist, pc\}^} \hfill //when leaving the handler
Return from Prefetch Abort

- If the processor supports MMU (Memory Management Unit)
  - The exception handler loads the unmapped instruction into physical memory
  - Then, uses the MMU to map the virtual memory location into the physical one.

- After that, the handler must return to *retry the instruction that caused the exception.*

- However, the $lr\_ABT$ points to the instruction at the address following the one that caused the abort exception
Return from Prefetch Abort (Cont.)

- So the address to be restored is at $lr_{ABT} - 4$
- Thus, with simple return
  \[
  \text{SUBS } pc, lr, \#4
  \]
- In contrast, with complex return
  \[
  \text{SUB } lr, lr, \#4 \quad ; \text{handler entry code}
  \]
  \[
  \text{STMFD sp!, \{reglist, lr\}}
  \]
  \[
  \ldots
  \]
  \[
  \text{LDMFD sp!, \{reglist, pc\}^+} \quad ; \text{handler exit code}
  \]
Return from Data Abort

- lr_ABTT points *two instructions beyond* the instruction that caused the abort
  - Since when a load or store instruction tries to access memory, the program counter has been updated.
  - Thus, the instruction caused the data abort exception is at $lr_{ABT} - 8$

- So the address to be restored is at $lr_{ABT} - 8$
Return from Data Abort (Cont.)

- So the address to be restored is at $lr_{\text{ABT}} - 8$
- Thus, with simple return
  
  ```
  SUBS pc,lr,#8
  ```
- In contrast, with complex return
  
  ```
  SUB lr,lr,#8 ; handler entry code
  STMFD sp!,{reglist,lr}
  ; ...
  LDMFD sp!,{reglist,pc}^ ; handler exit code
  ```
Summary

<table>
<thead>
<tr>
<th>Return Instruction</th>
<th>Previous State</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ARM R14_x</td>
<td>THUMB R14_x</td>
</tr>
<tr>
<td>BL</td>
<td>PC + 4</td>
<td>PC + 2</td>
</tr>
<tr>
<td>SWI</td>
<td>PC + 4</td>
<td>PC + 2</td>
</tr>
<tr>
<td>UDEF</td>
<td>PC + 4</td>
<td>PC + 2</td>
</tr>
<tr>
<td>FIQ</td>
<td>PC + 4</td>
<td>PC + 4</td>
</tr>
<tr>
<td>IRQ</td>
<td>PC + 4</td>
<td>PC + 4</td>
</tr>
<tr>
<td>PABT</td>
<td>PC + 4</td>
<td>PC + 4</td>
</tr>
<tr>
<td>DABT</td>
<td>PC + 8</td>
<td>PC + 8</td>
</tr>
<tr>
<td>RESET</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

NOTES

1. PC is the address of the BL/SWI/Undefined Instruction fetch which had the prefetch abort.
2. PC is the address of the instruction which did not get executed since the FIQ or IRQ took priority.
3. PC is the address of the Load or Store instruction which generated the data abort.
4. The value saved in R14_svc upon reset is unpredictable.
Install an Exception Handler

- Any new exception handler must be installed in the vector table

- Exception handlers can be installed in two ways
  - *Branch instruction*: simple but have one limitation
    - Branch instruction only has a range of 32 MB relative to the pc
  - *Load pc instruction*: set pc by
    - Load instruction to load the handler address into the program counter
Install an Exception Handler: Method

Vector_Init_Block

b Reset_Addr
b Undefined_Addr
b SWI_Addr
b Prefetch_Addr
b Abort_Addr
NOP ;Reserved vector
b IRQ_Addr
b FIQ_Addr

Reset_Addr  ...
Undefined_Addr  ...
SWI_Addr  ...
Prefetch_Addr  ...
Abort_Addr  ...
IRQ_Addr  ...
FIQ_Addr  ...
Install an Exception Handler: Method

Vector_Init_Block

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR PC, Reset_Addr</td>
<td></td>
</tr>
<tr>
<td>LDR PC, Undefined.Addr</td>
<td></td>
</tr>
<tr>
<td>LDR PC, SWI.Addr</td>
<td></td>
</tr>
<tr>
<td>LDR PC, Prefetch.Addr</td>
<td></td>
</tr>
<tr>
<td>LDR PC, Abort.Addr</td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>;Reserved vector</td>
</tr>
<tr>
<td>LDR PC, IRQ.Addr</td>
<td></td>
</tr>
<tr>
<td>LDR PC, FIQ.Addr</td>
<td></td>
</tr>
</tbody>
</table>

Reset_Addr DCD Start_Boot
Undefined_Addr DCD Undefined_Handler
SWI_Addr DCD SWI_Handler
Prefetch_Addr DCD Prefetch_Handler
Abort_Addr DCD Abort_Handler

DCD 0 ;Reserved vector

IRQ_Addr DCD IRQ_Handler
FIQ_Addr DCD FIQ_Handler
DCD

- Allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory

- Examples
  
  data1 DCD 1,5,20 ; defines 3 words containing
                  ; decimal values 1, 5, and 20
  data2 DCD mem06 + 4 ; defines 1 word containing 4 +
                  ; the address of the label mem06
SWI Handlers

- Top-Level SWI Handlers
- SWI Routine in Assembly Language
- SWI Routine in C
- How to Pass Values in and out of a SWI Routine
- Calling SWIs from an Application
SWI Handlers

- When the SWI handler is entered, it must know which SWI is being called
  - The SWI number is stored in bits 0-23 of the instruction
  - Or passed in an integer register, usually one of $r0-r3$
Top-Level SWI Handlers

- Because SVC only has its own \( LR_{svc} \) and \( SP_{svc} \)
  - Save all other \( r0~r12 \) to the stack

- To calculate the SWI number
  - Calculate the instruction address causing the SWI
    - Since \( lr_{SVC} \) holds the address of the instruction that follows the SWI instruction, thus
    - \( \text{LDR} \quad r0, [lr, #-4] \); derive the SWI instruction’s address
  - The SWI number is extracted by clearing the top eight bits of the opcode:
    - \( \text{BIC} \quad r0, r0, \#0xFF000000 \)
Top-Level SWI Handlers (Cont.)

```assembly
SWI_Handler ; top-level handler
    STMFD sp!,{r0-r12,lr} ; Store registers.
    LDR r0,[lr,#-4] ; Calculate address of SWI instruction
                     ; and load it into r0.
    BIC r0,r0,#0xff000000 ; Mask off top 8 bits of instruction
                         ; to give SWI number.

    ; Use value in r0 to determine which SWI routine to execute.
    ;
    LDMFD sp!, {r0-r12,pc}^ ; Restore registers and return.
    END ; Mark end of this file.
```
Top-Level SWI Handlers (Cont.)

- Above program is called **top-level handler**
  - Must always be written in ARM assembly language

- However, the routines to handle each SWI can be written in either assembly language or in C
SWI Routine in Assembly Language

- If the routines to handle each SWI in written in Assembly Language
  - The easiest way is using a *jump table*

- In the top-level handler, the $r0$ contains the SWI number

- Thus, the following code can be inserted into the top-level handler, i.e., SWI_Handler
  - Following on from the BIC instruction
SWI Routine in Assembly Language (Cont.)

```
CMP     r0, #MaxSWI ; Range check
LDRLS   pc, [pc,r0,LSL #2] ; LDRLS: LS (cond. exec.) lower or the same
    ; PC = PC + r0 * 4, (LSL: logical shift left)
B       SWIOOutOfRange
SWIJumpTable
    DCD SWInum0 ; stores the address of a routine
    DCD SWInum1 ; stores the address of a routine
    ... ; DCD for each of other SWI routines
SWInum0 ; SWI number 0 code
    ..... 
    B    EndofSWI
SWInum1 ; SWI number 1 code
    ..... 
    B    EndofSWI
EndofSWI ; Rest of SWI handling code
    ; Return execution to top level SWI handler so as to restore
    ; registers and return to program.
```
SWI Routine in C

- If the routines to handle each SWI in written in C
- The top-level handler uses a BL (branch and link) instruction to jump to the appropriate C function
  - BL C_SWI_Handler ; call C routine to handle

- Then, we must invoke the C routine that handles respective SWI
  - But, how to pass the SWI number, which is now stored in r0, to the C function?
ARM Procedure Call Convention

- Use registers $r0-r3$ to pass parameter values into routines
  - Correspond to the \textit{first} to \textit{fourth} arguments in the C routines
- Remaining parameters are allocated to the stack in order
- A function can return
  - A one-word integer value in $r0$
  - A two to four-word integer value in $r0-r1$, $r0-r2$ or $r0-r3$. 
Thus, the C handler is like the following:

```c
void C_SWI_handler (unsigned number)
{
    switch (number)
    {
        case 0 : /* SWI number 0 code */
            break;
        case 1 : /* SWI number 1 code */
            break;
        :
        :
        :
        default : /* Unknown SWI - report error */
    }
}
```
However, how to pass more parameters?
- Make use of the stack (supervisor stack)
- The top-level SWI handler can pass the stack pointer value (i.e. r13) to the SWI C routine as the, for example, second parameter, i.e., r1
  - sp is pointing to the supervisor stack,
  - MOV r1, sp
  - BL C_SWI_Handler
Then, the C_SWI_Handler can access it

```c
void C_SWI_handler (unsigned number, unsigned *reg)
{
    value_in_reg_0 = reg[0]; // can read from them:
    value_in_reg_1 = reg[1];
    value_in_reg_2 = reg[2];
    value_in_reg_3 = reg[3];

    reg[0] = updated_value_0;  // write back to them
    reg[1] = updated_value_1;
    reg[2] = updated_value_2;
    reg[3] = updated_value_3;
}
```
How to Pass Values in and out of a SWI Routine

- How the main program code passes values in and out of a SWI routine?

- Note that
  - The main program code is executing in the User mode
  - The SWI handler and their routines are in the Supervisor mode
  - However, both mode has the same r0~r12 registers
Thus, the application code and SWI routine can communicate by $r0$~$r12$ registers.
Calling SWIs from an Application

- The application code can call a SWI from assembly language or C/C++

- In assembly language
  - Set up any required register value
  - Then issue the relevant SWI
  - For example:
    - MOV r0, #65 ; load r0 with the value 65
    - SWI 0x0 ; Call SWI 0x0 with parameter value in r0
Calling SWIs from an Application (Cont.)

- From C/C++, declare the SWI as an __SWI function, and call it.

- Example:
  ```
  __swi(0) void my_swi(int);
  .
  .
  .
  my_swi(65);
  ```
Calling SWIs from an Application (Cont.)

- __SWI function allow a SWI to compiled inline
  - Without additional overhead

- However, it must have the restrictions that
  - Any arguments are passed in r0-r3 only
  - Any results are returned in r0-r3 only
```c
#include <stdio.h>
#include "swi.h"

int main( void )
{
    int result1, result2;
    struct four_results res_3;

    Install_Handler( (unsigned) SWI_Handler, swi_vec );
    printf("result1 = multiply_two(2,4) = %d\n", result1 =
multiply_two(2,4));
    printf("add_two( result1, result2 ) = %d\n", add_two( result1,
result2 ));
    return 0;
}
```
Calling SWIs from an Application
(Cont.)

- swi.h

```c
__swi(0) int multiply_two(int, int);
__swi(1) int add_two(int, int);
```
Interrupt Handlers

- The ARM processor has two levels of external interrupt
  - FIQ and IRQ
- FIQs have higher priority than IRQs because
  - FIQs are serviced first when multiple interrupts occur.
  - Servicing a FIQ causes IRQs to be disabled until after the FIQ handler has re-enabled them
- By restoring the CPSR from the SPSR at the end of the handler
Interrupt Handlers (Cont.)

- How the FIQ performs faster than IRQ
  - FIQ vector is the last entry in the vector table
    - FIQ handler can be placed directly at the vector location and run sequentially from that address
      - Removes the need for a branch and its associated delays
      - If the system has a cache, the vector table and FIQ handler may all be locked down in one block.
  - FIQ has more banked registers than IRQ
    - r8_FIQ~r12_FIQ registers
    - Have less time in the register save/restore
IRQ Handler

IRQ_Handler: ; top-level handler
  STMFD sp!,{r0-r12,lr} ; Store registers.
  BL ISRR_IRQ

  LDMFD sp!, {r0-r12,pc} ; Restore registers and return
  SUBS pc, lr, #4
  END ; Mark end of this file.
The ISR_IRQ depends on which interrupt controller used.

For example, in Samsung S3C4510B:
- The interrupt controller has a total of 21 interrupt sources.
- Each interrupt can be categorized as either IRQ or FIQ.
- Each interrupt has an interrupt pending bit.
## S3C4510B Interrupt Sources

<table>
<thead>
<tr>
<th>Index Values</th>
<th>Interrupt Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>[20]</td>
<td>I²C-bus interrupt</td>
</tr>
<tr>
<td>[19]</td>
<td>Ethernet controller MAC Rx interrupt</td>
</tr>
<tr>
<td>[18]</td>
<td>Ethernet controller MAC Tx interrupt</td>
</tr>
<tr>
<td>[17]</td>
<td>Ethernet controller BDMA Rx interrupt</td>
</tr>
<tr>
<td>[16]</td>
<td>Ethernet controller BDMA Tx interrupt</td>
</tr>
<tr>
<td>[15]</td>
<td>HDLC channel B Rx interrupt</td>
</tr>
<tr>
<td>[14]</td>
<td>HDLC channel B Tx interrupt</td>
</tr>
<tr>
<td>[13]</td>
<td>HDLC channel A Rx interrupt</td>
</tr>
<tr>
<td>[12]</td>
<td>HDLC channel A Tx interrupt</td>
</tr>
<tr>
<td>[10]</td>
<td>Timer 0 interrupt</td>
</tr>
<tr>
<td>[9]</td>
<td>GDMA channel 1 interrupt</td>
</tr>
<tr>
<td>[8]</td>
<td>GDMA channel 0 interrupt</td>
</tr>
<tr>
<td>[7]</td>
<td>UART 1 receive and error interrupt</td>
</tr>
<tr>
<td>[6]</td>
<td>UART 1 transmit interrupt</td>
</tr>
<tr>
<td>[5]</td>
<td>UART 0 receive and error interrupt</td>
</tr>
<tr>
<td>[4]</td>
<td>UART 0 transmit interrupt</td>
</tr>
<tr>
<td>[2]</td>
<td>External interrupt 2</td>
</tr>
<tr>
<td>[1]</td>
<td>External interrupt 1</td>
</tr>
<tr>
<td>[0]</td>
<td>External interrupt 0</td>
</tr>
</tbody>
</table>
Five special registers used to control the interrupt generation and handling

- **Interrupt mode register**
  - Defines the interrupt mode, IRQ or FIQ, for each interrupt source.

- **Interrupt pending register**
  - Indicates that an interrupt request is pending

- **Interrupt mask register**
  - The current interrupt is disabled if the corresponding mask bit is "1"
  - If the global mask bit (bit 21) is set to "1", no interrupts are serviced.

- **Interrupt priority registers**
  - Determine the interrupt priority

- **Interrupt offset register**
  - Determine the highest priority among the pending interrupts.
Interrupt Mode Register (INTMOD)

- Bit settings in the INTMOD specify if an interrupt is to be serviced as a FIQ or IRQ
- Each of the 21 bits corresponds to an interrupt source
  - 1: FIQ
  - 0: IRQ
Interrupt Pending Register (INTPND)

- Contains interrupt pending bits for each interrupt source
- Each of the 21 bits corresponds to an interrupt source
  - When an interrupt request is generated, its pending bit is set to 1
  - The service routine must then clear the pending condition by writing a 1 to the appropriate pending bit at start.
Interrupt Mask Register (INTMSK)

- Contains interrupt mask bits for each interrupt source
- Each of the 21 bits in the interrupt mask register corresponds to an interrupt source.
  - If bit is 1, the interrupt is not serviced by the CPU when the corresponding interrupt is generated
  - If the mask bit is 0, the interrupt is serviced upon request
Interrupt Mask Register (INTMSK) (Cont.)

- If global mask bit (bit 21) is 1, no interrupts are serviced
  - However, the source pending bit is set whenever the interrupt is generated
  - After the global mask bit is cleared, the interrupt is serviced.
Interrupt Priority Registers (INTPRI0–INTPRI5)

- Contain information about which interrupt source is assigned to the pre-defined interrupt priority
- Each INTPRI\textsubscript{n} register value determines the priority of the corresponding interrupt source
  - The lowest priority value is priority 0, and the highest priority value is priority 20
  - The index value of each interrupt source is written to one of the above 21 positions
- See the next slide
Interrupt Priority Registers (INTPRI0–INTPRI5) (Cont.)

| INTPRI0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| INTPRI1 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| INTPRI2 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| INTPRI3 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| INTPRI4 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
| INTPRI5 | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  |
Interrupt Offset Register (INTOFFSET)

- Contains the *interrupt offset address* of the interrupt
  - Hold the highest priority among the pending interrupts
  - The content of the interrupt offset address is "bit position value of the interrupt source << 2"
  - If all interrupt pending bits are "0" when you read this register, the return value is "0x00000054"
This register is valid only under the *IRQ mode* or *FIQ mode* in the ARM7TDMI.

In the interrupt service routine, you may change CPU mode to perform other works.

Thus, read this register before you changing the CPU mode from *IRQ* or *FIQ* to other modes.

- `IRQNumber = INTOFFSET >> 2`
Interrupt Offset Register (Cont.)

- INTOSET_FIQ/INTOSET_IRQ register can also be used to get the highest priority interrupt
  - INTOSET_FIQ: FIQ interrupt offset register
  - INTOSET_IRQ: IRQ interrupt offset register
IRQ Handler (Cont.)

- Initialize interrupt mode register (INTMOD) to FIQ or IRQ
- Initialize interrupt mask register (INTMSK)
- Clear interrupt pending register (INTPND)

Interrupt priority?

Set the interrupt priority registers (INTPRI0~5) to new priority values which are the index values of the interrupt sources

Setup interrupt service routine to interrupt vector table.
   ex) SysSetInterrupt(index, void (*handler)())

- Enable global mask bit, INTMSK[20].
- Enable appropriate interrupt mask bits.

DONE
Interrupt Request

- Copies CPSR into SPSR
- Set the appropriate CPSR mode bits:
  > to change to the appropriate mode
  > to disable interrupts.
- Stores the return address
- sets the PC to the appropriate vector address

CPU context saved

Running Code

ISR entry function
- branch the mirrored vector table on DRAM and Jump to the function call handler
- Stores all the working registers and link register
- Call C- interrupt handler

ISR exit function
- Restores the saved working registers and link registers
- Set PC to return address

User ISR Code

Interrupt latency

Interrupt response

Interrupt recoveries
FIQ Handler

FIQ_Handler

STMFD sp!, {r0-r7, lr}
BL ISR_FiqHandler
LDMFD sp!, {r0-r7, lr}
SUBS pc, lr, #4

- The same as IRQ Handler
  - Except the number of saved unbanked registers
Reset Handler

- The operation depend on the system for which the software is being developed
- For example
  - Initialize stacks and registers.
  - Initialize the memory system, if using an MMU.
  - Initialize any critical I/O devices.
  - Enable interrupts.
  - Change processor mode and/or state.
  - Initialize variables required by C and call the main application
Undefined Instruction Handlers

- Instructions that are not recognized by the processor are offered to any coprocessors
  - If the instruction remains unrecognized, an Undefined Instruction exception is generated
    - The instruction is intended for a coprocessor, but that the relevant coprocessor is not attached to the system.
  - However, a software emulator for such a coprocessor might be available.
Software Emulator

- Attach itself to the Undefined Instruction vector and store the old contents.
- Examine the undefined instruction to see if it should be emulated.
  - If bits 27 to 24 = b1110 or b110x,
    - The instruction is a coprocessor instruction
  - Bits 8-11: CP# (Co-Process Number)
    - Specify which coprocessor is being called upon
- If not, the emulator passes the exception onto the original handler or the next emulator in the chain
### ARM Instruction Set Format

<table>
<thead>
<tr>
<th>Cond</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Operand2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>000</td>
<td>A</td>
<td>S</td>
<td>Rd</td>
<td>Rn</td>
</tr>
<tr>
<td>0001</td>
<td>0000</td>
<td>U</td>
<td>A</td>
<td>RdHi</td>
<td>RnLo</td>
</tr>
<tr>
<td>0001</td>
<td>0100</td>
<td>B</td>
<td>0</td>
<td>Rn</td>
<td>Rd</td>
</tr>
<tr>
<td>0001</td>
<td>0000</td>
<td>1</td>
<td>111111111111111</td>
<td>0001</td>
<td>Rn</td>
</tr>
<tr>
<td>0001</td>
<td>0000</td>
<td>P</td>
<td>U</td>
<td>0</td>
<td>W L Rn</td>
</tr>
<tr>
<td>0001</td>
<td>0000</td>
<td>P</td>
<td>U</td>
<td>1</td>
<td>W L Rn</td>
</tr>
<tr>
<td>0111</td>
<td>P</td>
<td>B</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
</tr>
<tr>
<td>1000</td>
<td>P</td>
<td>U</td>
<td>S</td>
<td>W</td>
<td>L Rn</td>
</tr>
<tr>
<td>101</td>
<td>U</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>P</td>
<td>N</td>
<td>W</td>
<td>L</td>
<td>Rn</td>
</tr>
<tr>
<td>1110</td>
<td>CP Opc</td>
<td>CRn</td>
<td>CRd</td>
<td>CP#</td>
<td>Offset</td>
</tr>
<tr>
<td>1110</td>
<td>CP Opc</td>
<td>L</td>
<td>CRn</td>
<td>CP#</td>
<td>Offset</td>
</tr>
</tbody>
</table>

**Data processing/FSR Transfer**
- Multiply
- Multiply Long
- Single data swap
- Branch and exchange
- Halfword data transfer: register offset
- Halfword data transfer: immediate offset
- Single data transfer
- Undefined
- Block data transfer
- Branch
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt
Prefetch Abort

- If the system has no MMU
  - The Prefetch Abort handler can simply report the error and quit

- Otherwise
  - The address that caused the abort must be restored into physical memory

- In both cases, the handler must return to the instruction causing the prefetch abort exception
  - SUBS pc, lr, #4
Data Abort Handler

- If the system has no MMU
  - The Data Abort handler can simply report the error and quit

- Otherwise
  - The handler should deal with the virtual memory fault

- In both cases, the handler must return to the instruction causing the prefetch abort exception
  - `SUBS pc, lr, #8`
Data Abort Handler

- Three types of instruction can cause this abort
  - Single Register Load or Store (LDR or STR)
  - Swap (SWP)
  - Load Multiple or Store Multiple (LDM or STM)
Reference

- **Sansung S3C4510B User’s Manual**
  - *Chapter 13 Interrupt Controller*
  - [http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/S3C4510B.htm](http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/S3C4510B.htm)

- **Sansung S3C4510B application notes**
  - [http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/S3C4510B.htm](http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/S3C4510B.htm)

- **ARM® Developer Suite: Developer Guide**
  - *Chapter 5: Handling Processor Exceptions*