

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

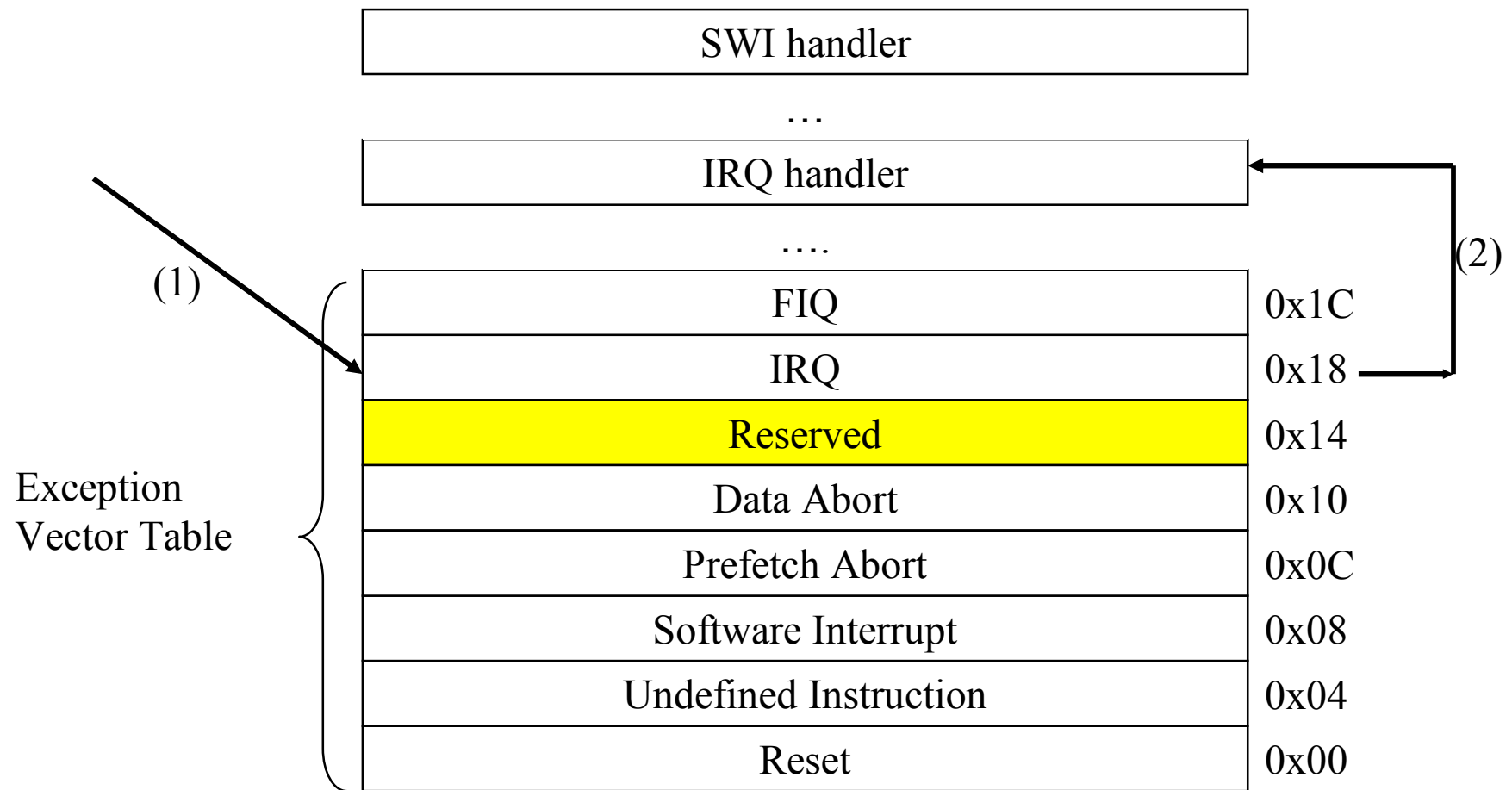
- Each entry has only 32 bit
 - Not enough to contain the full code for a handler
 - Thus, usually is a *branch instruction* or *load pc instruction* to the actual handler

- Example: `armc_startup.s`

ARM Exception

Exception type	Mode	Normal address	High vector address
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

ARM Exception Vector Table



ARM Exception Events

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

ARM Exception Priorities

Vector address	Exception type	Exception mode	Priority (1=high, 6=low)
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor (SVC)	6
0xC	Prefetch Abort	Abort	5
0x10	Data Abort	Abort	2
0x14	<i>Reserved</i>	<i>Not applicable</i>	<i>Not applicable</i>
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

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

System & User	FIQ	Supervisor	Abort	IRQ	Undefined
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

ARM State Program Status Register

CPSR	CPSR SPSR_fiq	CPSR SPSR_svc	CPSR SPSR_abt	CPSR SPSR_irq	CPSR SPSR_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.)

Reset	R14_svc = unexpected SPSR_svc = unexpected CPSR[4:0] = 0b10011 //Supervisor Mode CPSR[5] = 0 // ARM state CPSR[6] = 1 // Disable FIQ CPSR[7] = 1 // Disable IRQ PC = 0x00000000
Undefined Instructions	R14_und = PC+4 SPSR_und = CPSR CPSR[4:0] = 0b11011 //Undefined Mode CPSR[5] = 0 // ARM state CPSR[6] unchanged CPSR[7] = 1 // Disable IRQ PC = 0x00000004

The Process Response to an Exception (Cont.)

Software Interrupt	<p>R14_svc = PC + 4 SPSR_svc = CPSR CPSR[4:0] = 0b10011 //Supervisor Mode CPSR[5] = 0 // ARM state CPSR[6] unchanged CPSR[7] = 1 // Disable IRQ PC = 0x00000008</p>
Prefetch Abort	<p>R14_abt = PC+4 SPSR_abt = CPSR CPSR[4:0] = 0b10111 //Abort Mode CPSR[5] = 0 CPSR[6] unchanged CPSR[7] = 1 // Disable IRQ PC = 0x0000000C</p>

The Process Response to an Exception (Cont.)

Data Abort	<p>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</p>
Interrupt Request	<p>R14_abt = PC+4 SPSR_abt = CPSR CPSR[4:0] = 0b10010 //Abort Mode CPSR[5] = 0 CPSR[6] unchanged CPSR[7] = 1 // Disable IRQ PC = 0x00000018</p>

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*

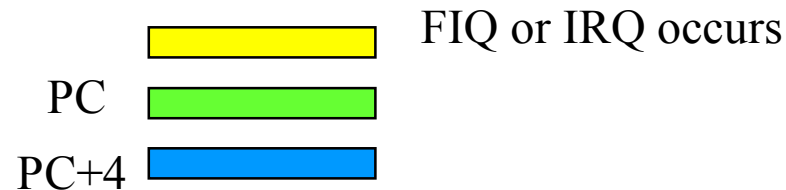
Returning from SWI and Undefined Instruction Handlers

- 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: `MOVS 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: `SUBS pc, lr, #4` //pc = lr-4

- If using stack to store the return address

`SUB lr, lr, #4` //when entering the handler

`STMFD sp!, {reglist, lr}`

...

`LDMFD sp!, {reglist, pc}^` //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
SUBS pc,lr,#4
- In contrast, with complex return
SUB lr,lr,#4 ;handler entry code
STMFD sp!,{reglist,lr}
;...
LDMFD sp!,{reglist,pc}^ ; handler exit code



Return from Data Abort

- `lr_ABRT` 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 *$lr_ABRT - 8$*

- So the address to be restored is at *$lr_ABRT - 8$*

Return from Data Abort (Cont.)

- So the address to be restored is at $lr_ABT - 8$
- 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

	Return Instruction	Previous State		Notes
		ARM R14_x	THUMB R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	1
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	1
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	1
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	2
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	2
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	1
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	3
RESET	NA	–	–	4

□ 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 1

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 2

Vector_Init_Block

```
LDR    PC, Reset_Addr
LDR    PC, Undefined_Addr
LDR    PC, SWI_Addr
LDR    PC, Prefetch_Addr
LDR    PC, Abort_Addr
NOP                                ;Reserved vector
LDR    PC, IRQ_Addr
LDR    PC, FIQ_Addr
```

```
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

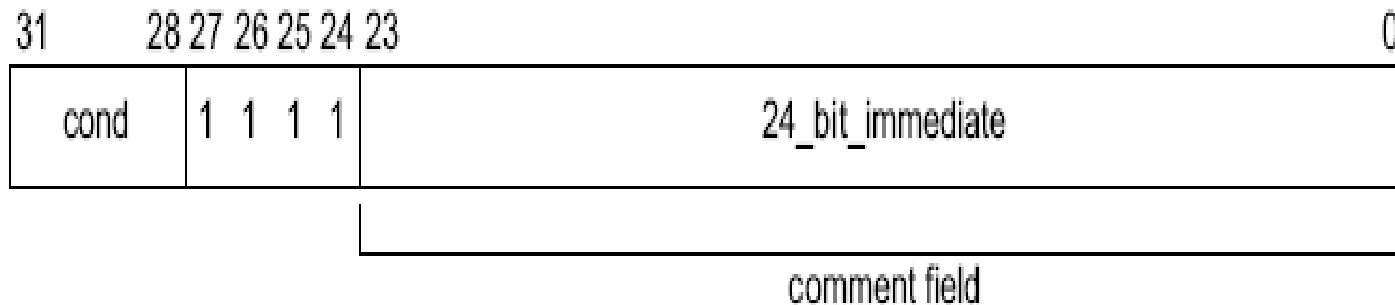


SWI Handlers

- 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
 - LDR *r0*, [*lr*, #-4] ; derive the SWI instruction's address
 - The SWI number is extracted by clearing the top eight bits of the opcode:
 - BIC *r0*, *r0*, #0xFF000000

Top-Level SWI Handlers (Cont.)

```
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 are 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      SWIOutOfRange

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
; Rest of SWI handling code

EndofSWI
; 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 are 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 *first* to *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 $r0-r3$.

SWI Routine in C (Cont.)

□ Thus, the C handler is like the following

```
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 */
    }
}
```



SWI Routine in C (Cont.)

- 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`

SWI Routine in C (Cont.)

- Then, the C_SWI_Handler can access it

```
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 $r0\sim r12$ registers



How to Pass Values in and out of a SWI Routine (Cont.)

- Thus, the application code and SWI routine can communicate by $r0\sim r12$ 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



Example

```
#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**

```
__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      ISR_IRQ

    LDMFD sp!, {r0-r12,pc} ; Restore registers and return
    SUBS    pc, lr, #4
    END                ; Mark end of this file.
```

Samsung S3C4510B Interrupt Controller

- The ISR_IRQ depends on which interrupt controller used
- For example, in Samsung S3C4510B
 - The interrupt controller has a total 21 interrupt sources
 - Each interrupt can be categorized as either IRQ or FIQ
 - Each interrupt has an *interrupt pending bit*

S3C4510B Interrupt Sources

Index Values	Interrupt Sources
[20]	I ² C-bus interrupt
[19]	Ethernet controller MAC Rx interrupt
[18]	Ethernet controller MAC Tx interrupt
[17]	Ethernet controller BDMA Rx interrupt
[16]	Ethernet controller BDMA Tx interrupt
[15]	HDLC channel B Rx interrupt
[14]	HDLC channel B Tx interrupt
[13]	HDLC channel A Rx interrupt
[12]	HDLC channel A Tx interrupt
[11]	Timer 1 interrupt
[10]	Timer 0 interrupt
[9]	GDMA channel 1 interrupt
[8]	GDMA channel 0 interrupt
[7]	UART 1 receive and error interrupt
[6]	UART 1 transmit interrupt
[5]	UART 0 receive and error interrupt
[4]	UART 0 transmit interrupt
[3]	External interrupt 3
[2]	External interrupt 2
[1]	External interrupt 1
[0]	External interrupt 0

Samsung S3C4510B Interrupt Controller (Cont.)

- 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 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 Priority Registers (*INTPRI0*–*INTPRI5*)

- Contain information about which interrupt source is assigned to the pre-defined interrupt priority
- Each *INTPRI_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 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* $\ll 2$ "
 - If all interrupt pending bits are "0" when you read this register, the return value is "0x00000054"



Interrupt Offset Register (Cont.)

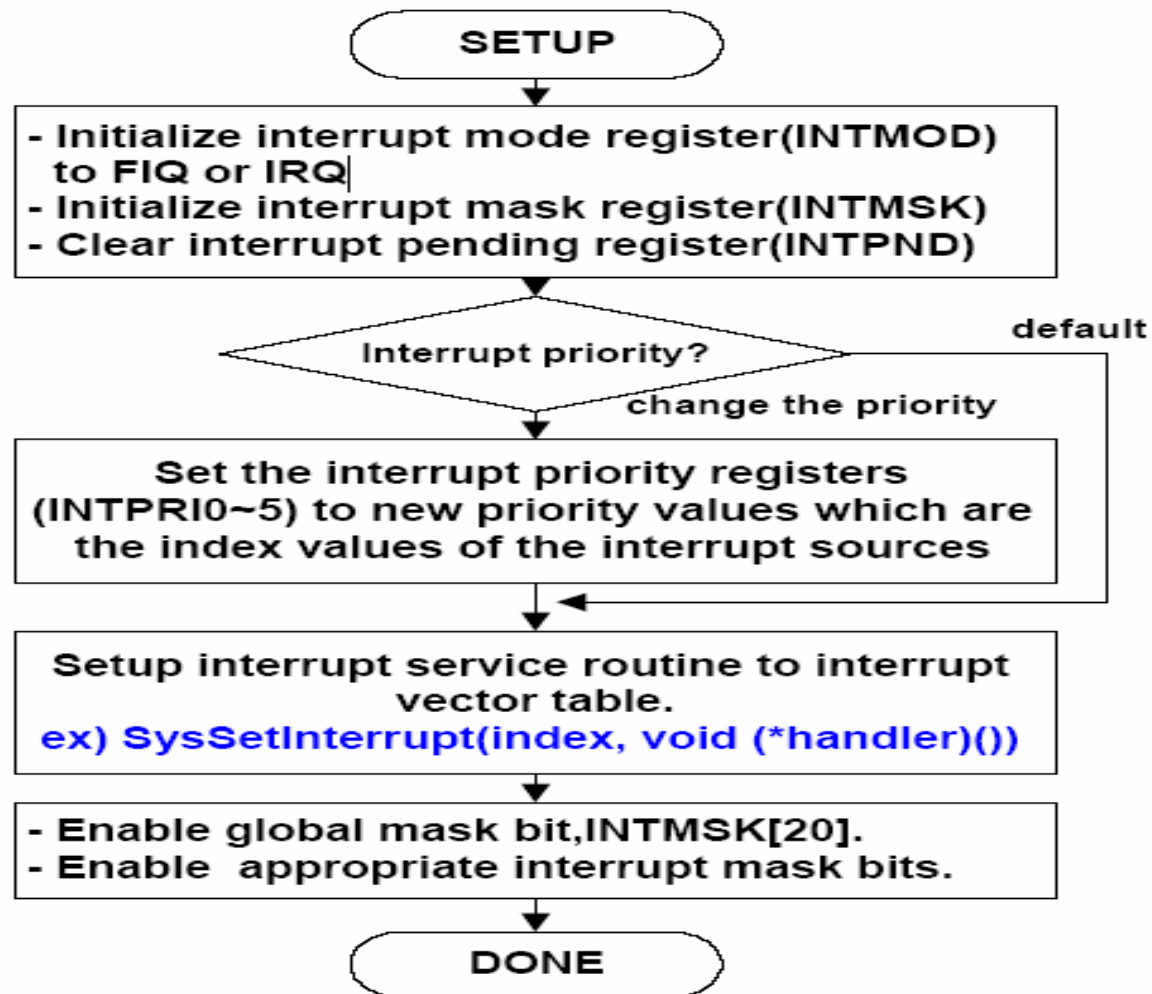
- This register is valid only under the *IRQ mode* or *FIQ mode* in the ARM7TDMI.
 - In the interrupt service routine, you may change CPU mode to perform other works
 - Thus, read this register before you changing the CPU mode from *IRQ or FIQ* to other modes.
 - $IRQNumber = INTOFFSET \gg 2$

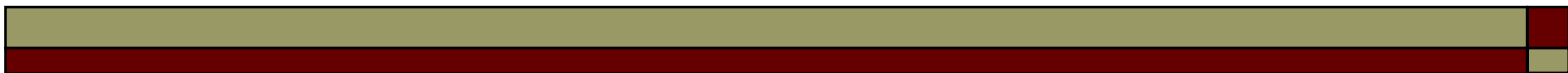


Interrupt Offset Register (Cont.)

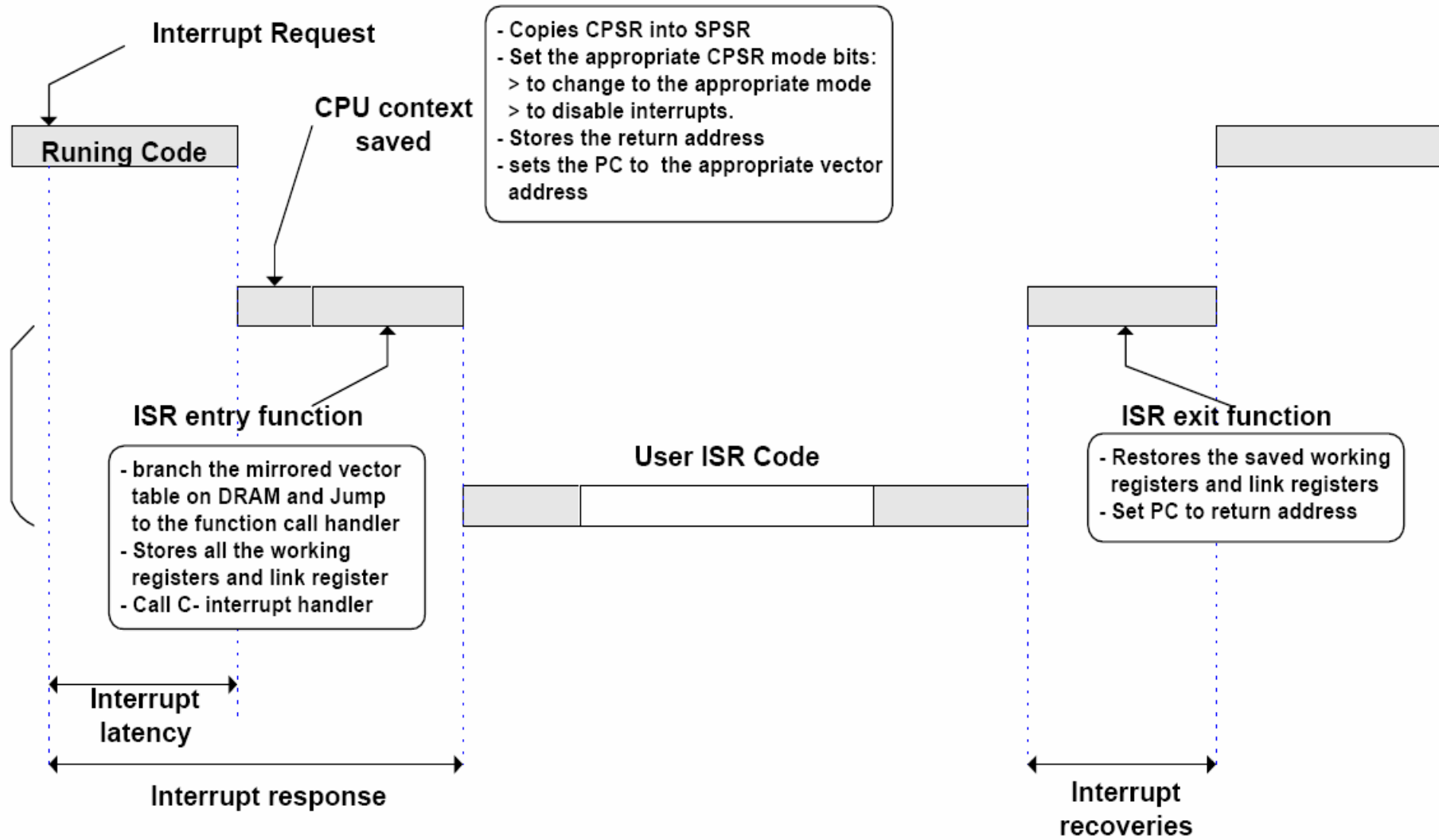
- INTOSET_FIQ/INTOSET_IRQ register can also be used to get the highest priority interrupt
 - INTOSET_FIQ: FIQ interrupt offset register
 - INTOSET_IRQ: IRQ interrupt offset register

IRQ Handler (Cont.)





TIME →





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

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

Cond	0	0	1	Opcode				S	Rn	Rd	Operand2							Data processing/ PSR Transfer							
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	Multiply								
Cond	0	0	0	0	1	U	A	S	RdHi	RnLo	Rn				1	0	0	1	Rm	Multiply Long					
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0	0	0	0	1	0	0	1	Rm	Single data swap					
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	Branch and exchange
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0	0	0	0	1	S	H	1	Rm	Halfword data transfer: register offset					
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset				1	S	H	1	Offset	Halfword data transfer: immediate offset					
Cond	0	1	1	P	U	B	W	L	Rn	Rd	Offset							Single data transfer							
Cond	0	1	1													1			Undefined						
Cond	1	0	0	P	U	S	W	L	Rn	Register List								Block data transfer							
Cond	1	0	1	L	Offset														Branch						
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset					Coprocessor data transfer								
Cond	1	1	1	0	CP Opc				CRn	CRd	CP#	CP#	0	CRm	Coprocessor data Operation										
Cond	1	1	1	0	CP Opc			L	CRn	Rd	CP#	CP#	1	CRm	Coprocessor register Transfer										
Cond	1	1	1	1	Ignored by processor												Software Interrupt								

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



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

- Samsung S3C4510B User's Manual
 - *Chapter 13 Interrupt Controller*
 - <http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/S3C4510B.htm>
- Samsung S3C4510B application notes
 - <http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/S3C4510B/S3C4510B.htm>
- ARM® Developer Suite: Developer Guide
 - *Chapter 5: Handling Processor Exceptions*