Xi'an Jiaotong-Liverpool University
西交利物浦大學

Handling Exceptions

Jianjun Chen

This Lecture

- Exceptions introduction
  - Interrupt
  - Error conditions
- ARM processor modes and register bank.
  - Other fields of CPSR and SPSR.
- The vector table and ARM startup.
- Exception handling example

Exceptions Introduction

Interrupt

Error conditions

Input and Output

- Large applications need to deal with I/O with various sources:
  - Keyboard, mice, USB ports ...
  - System events like battery level change, ambient light change.

- Input to and output from a microprocessor can be arranged in two ways:
  - Either as an additional subsystem with dedicated hardware.
  - Or, as part of the memory system.

Memory Mapped I/O

- ARM uses “memory mapped” input and output.
  - The ARM7 has a 32 bit address and can address 2^32 bytes or 4 gigabytes of memory.
  - Not all of this “addressable space” will be filled with actual memory so that there are many “empty” memory locations.

- Memory mapped input ports and output ports are assigned a memory address and
  - A register load from that address is equivalent to an input.
  - A register store to that address is equivalent to an output.

- Detailed mapping varies for different ARM boards.
  - Need to check documentation.

Interrupt and I/O

- How can a CPU respond to I/O events like key presses?

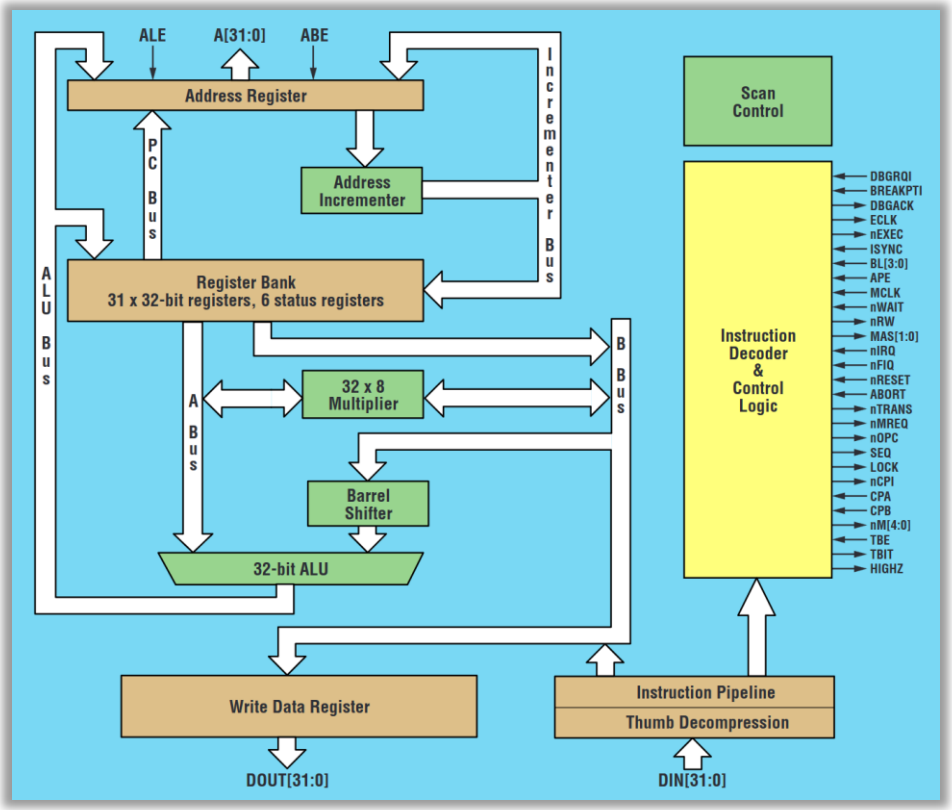
- Idea 1: CPU sits in a loop that only waits for the key press.
  - Cannot run any other program.
  - Most CPU cycles are wasted.

- Idea 2: CPU occasionally checks the memory to see if the key press happend or not (this method is called **polling**).
  - Still inefficient if a lot of devices are connected.
  - Some CPU cycles will be wasted.

- Idea 3: Let the device tell the processor when there's new data waiting to be processed.
  - Much more efficient.

Interrupt

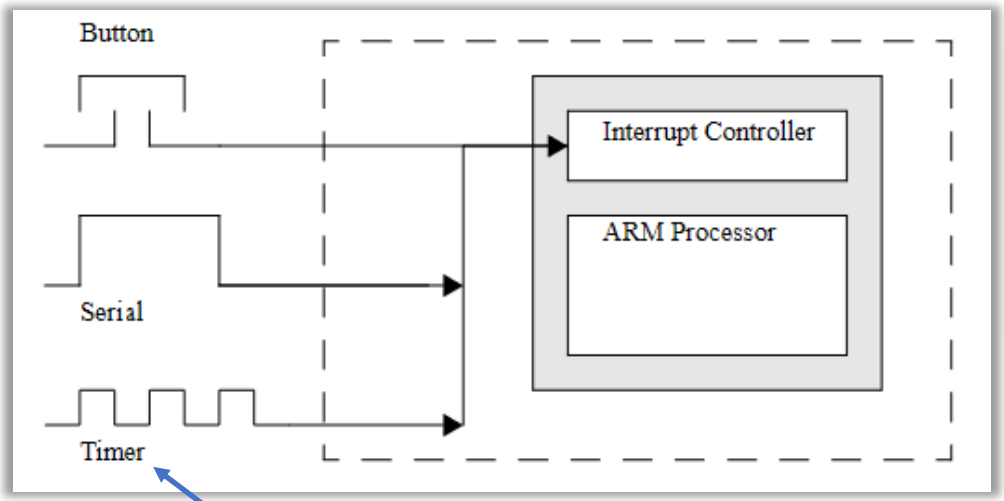
- The third method is called **interrupt**.
- Hardware interrupt is supported by 2 interupt lines in ARM7TDMI CPU.
  - nIRQ and nFIQ.
- Software interrupt is also available using an instruction called `SWI`.



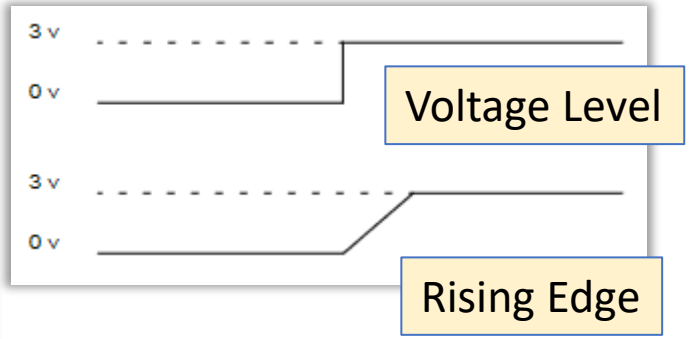
Hardware Interrupts

- Triggered by processor pin voltage change, which is managed by a piece of hardware called interrupt controller.

- Many devices are connected to the interrupt controller.
  - Example below: Key/Button, Serial, Timer



Used for handling multi-tasking



Error Conditions

- In addition to interrupts, CPU also needs to deal with error conditions.

- Example 1: If a program is run on a system without floating-point instruction support. It will take an **undefined instruction** exception.

- Example 2: When a program tries fetch an instruction from a non-existing memory area, **prefetch abort** exception may happen.

Exception Handling

- Interrupts and error conditions are also called **exceptions**.
- When exception occurs, the ARM7TDMI CPU will execute instructions located in special memory regions.
- As a result, users/operating systems can define their own handlers to handle exceptions.
- ARM exception handling relies on CPU **processor modes**.

Processor Modes

Banked registers

CPSR and SPSR registers

Processor Modes

- ARM CPU provides a mechanism to handle exceptions: **processor modes**.
  - There are 7 processor modes supported by ARMv7.
  - Each processor mode has its designed purpose/scenario.
- Mode changes can be made under software control or may be brought about by external interrupts or exception processing.

Exception modes

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Software Interrupt (SWI) instruction is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most applications/OS tasks run	Unprivileged mode

Processor Modes

- **User**: normal program execution state.
- **System**: a privileged mode for the operating system.
- **Supervisor**: entered on reset or when a software interrupt (SWI) instruction is executed.

- **Fast interrupt** : supports a data transfer or channel process.
- **Interrupt**: used for general-purpose interrupt handling.

- **Abort**: entered when a memory access violation occurs either on an instruction fetch or on a data read/write.
  - E.g. Writing to a read-only memory address.
- **Undefined**: entered when an instruction cannot be handled.

Privileged Modes

- Modes other than User mode are collectively known as **privileged modes**.
- Certain operations can only be done in privileged modes
  - Such as disabling IRQ or FIQ.

	Mode	Description	
Exception modes	Supervisor (SVC)	Entered on reset and when a Software Interrupt (SWI) instruction is executed	Privileged modes
	FIQ	Entered when a high priority (fast) interrupt is raised	
	IRQ	Entered when a low priority (normal) interrupt is raised	
	Abort	Used to handle memory access violations	
	Undef	Used to handle undefined instructions	
	System	Privileged mode using the same registers as User mode	
	User	Mode under which most applications/OS tasks run	Unprivileged mode

Processor Modes and Registers

- There are 37 registers in the <u>register file</u> of ARM7TDMI processor.
  - 30 general-purpose registers that can hold any value.
  - 6 status registers
  - A Program Counter register
- Grey registers are banked register
  - They are available only when the processor is in a particular mode.

30 general-purpose registers

Mode					
User/System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
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	R8	R8	R8	R8_FIQ
R9	R9	R9	R9	R9	R9_FIQ
R10	R10	R10	R10	R10	R10_FIQ
R11	R11	R11	R11	R11	R11_FIQ
R12	R12	R12	R12	R12	R12_FIQ
R13	R13_SVC	R13_ABORT	R13_UNDEF	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABORT	R14_UNDEF	R14_IRQ	R14_FIQ
PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_SVC	SPSR_ABORT	SPSR_UNDEF	SPSR_IRQ	SPSR_FIQ

= *banked register*

6 Status registers

Banked Registers

- When CPU switches the processor mode, banked registers will replace the corresponding registers.
  - They can be referenced in the same way, but are different registers.
  - E.g., by switching to IRQ mode, accessing R13 will lead to R13_IRQ. (Done at the CPU level)
- Internally, a banked register is just a separate register
  - R13_IRQ and R13 are two different registers.
  - While the R0 in user mode and the R0 in IRQ mode refers to the same register.
- A design like this allows the CPU to respond to real-time events faster

User
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
PC
CPSR

IRQ
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_IRQ
R14_IRQ
PC
CPSR
SPSR_IRQ

The CPSR Register

- The current processor mode and execution state is contained in the *Current Program Status Register* (CPSR).

Whether running ARM instructions (0) or Thumb (1) instructions

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
N	Z	C	V	Do not modify/Read as zero																				I	F	T	M4	M3	M2	M1	M0

condition flags

Whether IRQ is disabled
1: Disabled
0: Enabled

Whether FIQ is disabled
1: Disabled
0: Enabled

The Mode Bits

xPSR[4:0]	Mode
10000	User mode
10001	FIQ mode
10010	IRQ mode
10011	Supervisor mode
10111	Abort mode
11011	Undefined mode
11111	System mode

The SPSR Register

- Each privileged mode (except System mode) has a **Saved Program Status Register** (SPSR).
  - It has the exact same arrangement like CPSR.
- This allows the original processor state to be restored when the exception handler returns to normal program execution.
  - See the next slide.
- User mode and System mode are not entered on any exception.
  - They do not have an SPSR.
  - A register to preserve the CPSR is not required.

Processor Exception Sequence

When an exception occurs, the CPU will do:

- **STEP 1**: In all cases except a reset exception, the current instruction is allowed to complete.
- **STEP 2**: The `CPSR` is copied into `SPSR_<mode>`. ← Target mode
- **STEP 3**: The appropriate `CPSR` bits are set.
  - CPU will switch to **ARM state** if in Thumb state as certain instructions that can access status registers do not exist in Thumb state.
  - IRQ interrupts are also disabled automatically on entry to all exceptions.
  - FIQ interrupts are disabled on entry to Reset and FIQ exceptions.
- **STEP 4**: The return address is stored in `LR_<mode>`.
- **STEP 5**: The Program Counter changes to the appropriate **vector address** (explained later) in memory.

Processor Exception Sequence

- After an exception is handled, the processor should then return to the main code.
  - Whether or not it returns is a human decision based on the situations.

- Generally steps:
- **STEP 1**: The `CPSR` must be restored from `SPSR_<mode>`, where `<mode>` is the exception mode in which the processor currently operates.
- **STEP 2**: The PC must be restored from LR_<mode>.

- These actions can only be done in ARM state (32-bit long instructions).

Vector Table and ARM Startup

Interrupt handlers

The Vector Table

- Exception handling is supported by the **vector table**.
  - It is a list of addresses in memory that hold information necessary to handle an exception.
- The data value stored at each address is an instruction.
  - Usually an instruction that branches to another memory location that has more instructions to handle this exception.

Two exception types can lead to a same processor mode

ARM7TDMI Exception Vectors

Exception Type	Mode	Vector Address
Reset	SVC	0x00000000
Undefined instruction	UNDEF	0x00000004
Software Interrupt (SVC)	SVC	0x00000008
Prefetch abort (instruction fetch memory abort)	ABORT	0x0000000C
Data abort (data access memory abort)	ABORT	0x00000010
IRQ (interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

IRQ Exception Occurs

The CPU executes Processor Exception Sequence

STEP 5: The Program Counter (R15) changes to the appropriate **vector address**

Mode	Vector Address
SVC	0x00000000
UNDEF	0x00000004
SVC	0x00000008
ABORT	0x0000000C
ABORT	0x00000010
IRQ	0x00000018
FIQ	0x0000001C

The IRQ exception handler will do necessary operations

Will execute the instruction at 0x00000018, which is usually a branch to a function that can handle the exception (handler)

Vector Table: Example

Code is from LPC2400.s (startup file)

```
Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0120]
                LDR     PC, FIQ_Addr

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler

Undef_Handler   B       Undef_Handler
SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
IRQ_Handler     B       IRQ_Handler
FIQ_Handler     B       FIQ_Handler
```

Infinite loops

ARM7TDMI Exception Vectors

Exception Type	Mode	Vector Address
Reset	SVC	0x00000000
Undefined instruction	UNDEF	0x00000004
Software Interrupt (SVC)	SVC	0x00000008
Prefetch abort (instruction fetch memory abort)	ABORT	0x0000000C
Data abort (data access memory abort)	ABORT	0x00000010
IRQ (interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

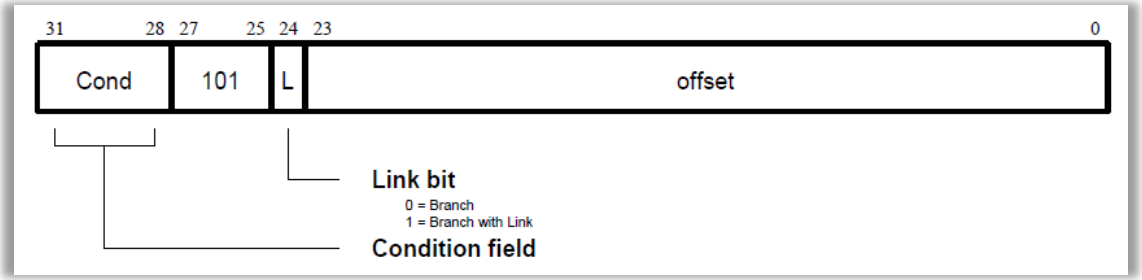
```
; Reset Handler

                EXPORT  Reset_Handler
Reset_Handler


; Clock Setup ----------------------------

                IF      (:LNOT:(:DEF:NO_CLO
                LDR     R0, =SCB_BASE
                MOV     R1, #0xAA
                MOV     R2, #0x55

;  Configure and Enable PLL
                LDR     R3, =SCS_Val
                STR     R3, [R0, #SCS_OFS]
```

Vector Table Change-of-flow Instructions

- In the previous example, LDR is used to change the value of `PC`. But there are other instructions that can do this:

- MOV instruction, example:
  - `MOV  PC, #0xF1`

- Branch instruction:
  - The offset (signed 24-bit integer) will be shifted left two bits, sign extended to 32 bits, and then added to the PC.
  - (2^24 << 2) bytes = 67,108,864 bytes = 64 MB
  - Since it is a signed integer, the resulting offset will be ±32MB.



RESET: the Starting Point

- The execution of ARM board programs starts from an area called "RESET".
- This is specified in a "Scatter-Loading Description File" <u>automatically generated</u> by uVision.
- If you are not using the startup file provided by Keil, you must define this area by yourself.

main.s | LPC2400.s | test.sct

```
; Area Definition and Entry Point
;  Startup Code must be linked first at Addres

                AREA    RESET, CODE, READONLY
                ARM


; Exception Vectors
;  Mapped to Address 0.
;  Absolute addressing mode must be used.
;  Dummy Handlers are implemented as infinite

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
```

```
; ***************************************
; *** Scatter-Loading Description File c
; ***************************************

LR_IROM1 0x00000000 0x00080000  {    ; L
  ER_IROM1 0x00000000 0x00080000  {  ; L
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
  }
  RW_IRAM1 0x40000000 0x00010000  {  ; R
   .ANY (+RW +ZI)
  }
}
```

Reset_Handler

- After powering on the ARM board, the first instruction to be fetched is located at memory address 0x0 (in `RESET` area):
  - `LDR  PC, Reset_Addr`
  - Note that it is different from "`LDR PC,=Reset_Addr`" in that the first instruction directly fetches the value located at "`Reset_Addr`".
  - This syntax is not supported in VisUAL.
- The value associated with "`Reset_Addr`" is the address of the first instruction in "`Reset_Handler`"

```
Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0120]
                LDR     PC, FIQ_Addr

Reset_Addr      DCD     Reset_Handler
```

```
; Reset Handler

                EXPORT  Reset_Handler
Reset_Handler


; Clock Setup ----------------------------

                IF      (:LNOT:(:DEF:NO_CLC
                LDR     R0, =SCB_BASE
                MOV     R1, #0xAA
                MOV     R2, #0x55
```

Reset_Handler

- In the startup file provided by Keil, the reset handler provide an initialization routine for turning on parts of the device and setting bits.
- You don't need these if you are only experimenting with the simulator.
  - But they are important for real ARM boards.

main.s | LPC2400.s | test.sct

```
                BNE     M_N_Lock

;  Setup CPU clock divider
                MOV     R3, #CCLKCFG_Val
                STR     R3, [R0, #CCLKCFG_OFS]

;  Setup USB clock divider
                LDR     R3, =USBCLKCFG_Val
                STR     R3, [R0, #USBCLKCFG_OFS]

;  Setup Peripheral Clock
                LDR     R3, =PCLKSEL0_Val
                STR     R3, [R0, #PCLKSEL0_OFS]
                LDR     R3, =PCLKSEL1_Val
                STR     R3, [R0, #PCLKSEL1_OFS]

;  Switch to PLL Clock
                MOV     R3, #(PLLCON_PLLE:OR:PLLCON_PLLC)
                STR     R3, [R0, #PLLCON_OFS]
                STR     R1, [R0, #PLLFEED_OFS]
                STR     R2, [R0, #PLLFEED_OFS]
                ENDIF   ; CLOCK_SETUP


; Setup Memory Accelerator Module ------------------------

                IF      MAM_SETUP != 0
                LDR     R0, =MAM_BASE
                MOV     R1, #MAMTIM_Val
                STR     R1, [R0, #MAMTIM_OFS]
                MOV     R1, #MAMCR_Val
                STR     R1, [R0, #MAMCR_OFS]
                ENDIF   ; MAM_SETUP
```

Reset_Handler

- Below is a program without defining the reset handler. Note that the `AREA` name must be `RESET`.

main.s | test.sct

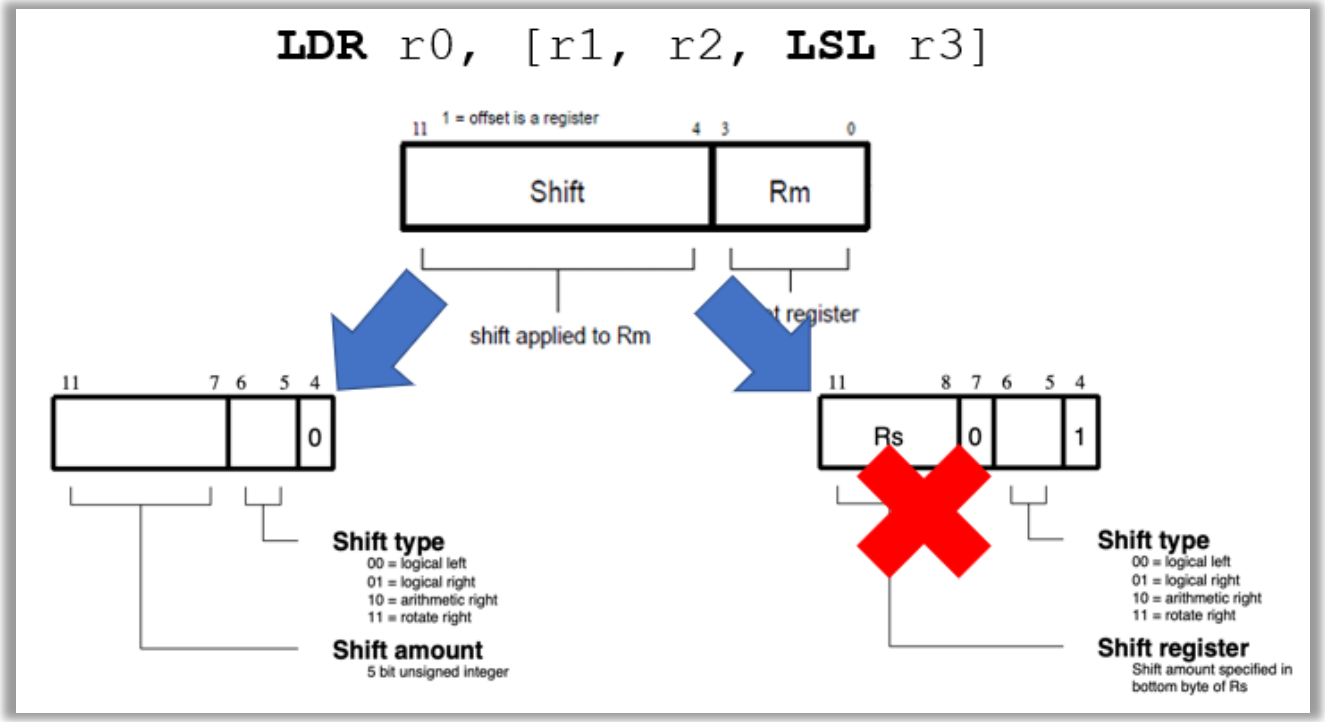
```
1    area RESET, code, readonly
2    entry
3    add r0, r0, #1
4    add r0, r0, #1
5    add r0, r0, #1
6    add r0, r0, #1
7    end
```

Exception Handling Example

Using undef exception to implement new “instructions”

Undefined Exception Example

- We previously learned that `LDR` does not support register-specified shift.
- Let's implement this instruction as a function by making use of the undefined exception



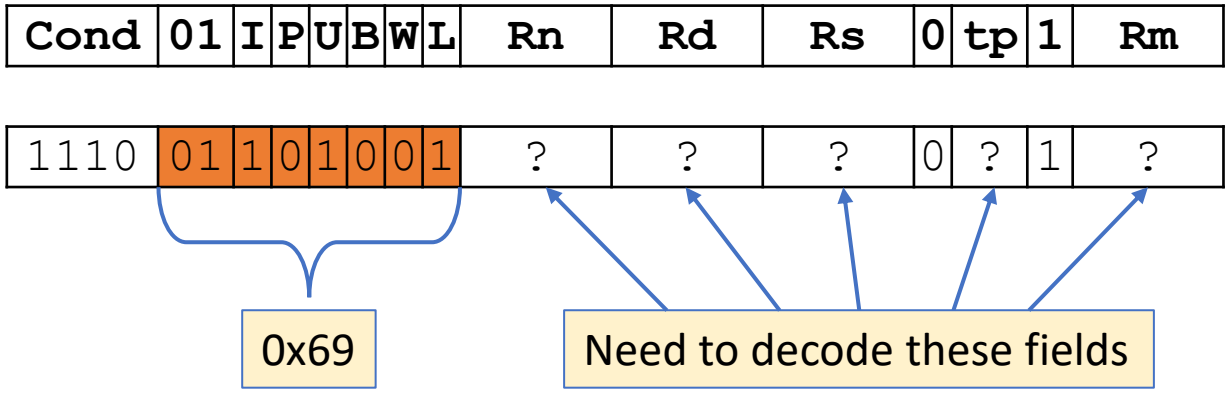
Undefined Exception Example

- To keep the example simple, we will only consider the syntax

  LDR `Rt, [Rn], Rm,` **shift_type** `Rs`

- Other forms like pre-indexed loading and the `STR` version will not be considered. Condition codes will also be ignored.

- As a result, the opcode of the instruction is always 0x69.



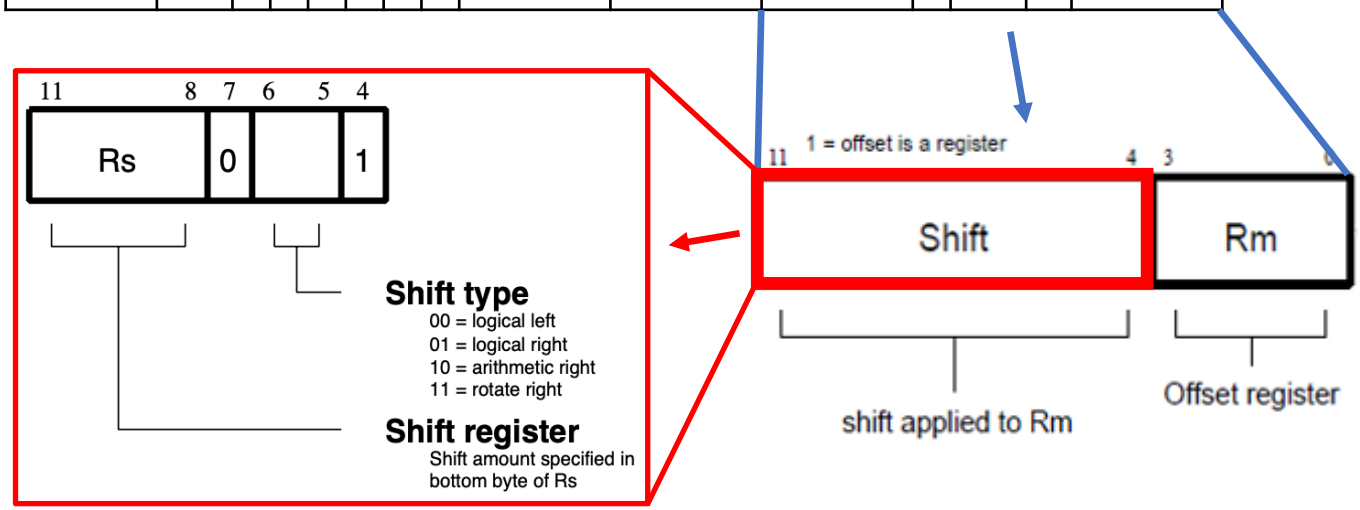
The Example

- The following instruction will be used as the example:

```
LDR R0, [R1], R2, LSR R3
```

- Its encoding is 0xE6910332 and has the following field values:

Cond	01	I	P	U	B	W	L	Rn	Rd	Rs	0	Shift type	1	Rm
1110	01	1	0	1	0	0	1	0001	0000	0011	0	01	1	0010



The ARM board starts and the CPU enters the RESET area

execute a non-existing instruction

Undef Exception Occurs

The CPU executes Processor Exception Sequence

Mode	Vector Address
SVC	0x00000000
UNDEF	0x00000004
SVC	0x00000008
ABORT	0x0000000C
ABORT	0x00000010
IRQ	0x00000018
FIQ	0x0000001C

Redirected to undef exception handler

The non-existing instruction will be manually decoded and executed

Continue to execute the next instruction

The Non-existing Instruction

- This non-existing instruction cannot be parsed by the assembler due to syntax error.

```
27                  LDR     R1, =numbers
28                  MOV     R2, #8
29                  MOV     R3, #1
30                  LDR R0, [R1], R2, LSR R3
```

main.s(30): error: A1110E: Expected constant expression

- Workaround: put a `DCD` directive in the place where you would put the instruction: 0xE6910332 for this case

```
28                  LDR     R1, =numbers
29                  MOV     R2, #8
30                  MOV     R3, #1
31  special_LDR     DCD     0xE6910332
32  deadend         B       deadend
```

```
    28:                     LDR     R1, =numbers
0x00000058  E59F10E8  LDR       R1,[PC,#0x00E8]
    29:                     MOV     R2, #8
0x0000005C  E3A02008  MOV       R2,#0x00000008
    30:                     MOV     R3, #1
    31: special_LDR         DCD     0xE6910332
0x00000060  E3A03001  MOV       R3,#0x00000001
0x00000064  E6910332  ???
```

Exception Vector Table

main.s | test.sct

```
; Example created by Jianjun Chen (Jianjun.Chen@xjtlu.edu.cn)
                AREA    RESET, CODE, READONLY
                ARM
                ENTRY

Vectors         LDR     PC, =Reset_Handler
                LDR     PC, =Undef_Handler
                LDR     PC, =SWI_Handler
                LDR     PC, =PAbt_Handler
                LDR     PC, =DAbt_Handler
                NOP                                ; Reserved Vector
                LDR     PC, =IRQ_Handler
                LDR     PC, =FIQ_Handler

SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
IRQ_Handler     B       IRQ_Handler
FIQ_Handler     B       FIQ_Handler
```

Reset and Undef are defined separately

The Reset Handler

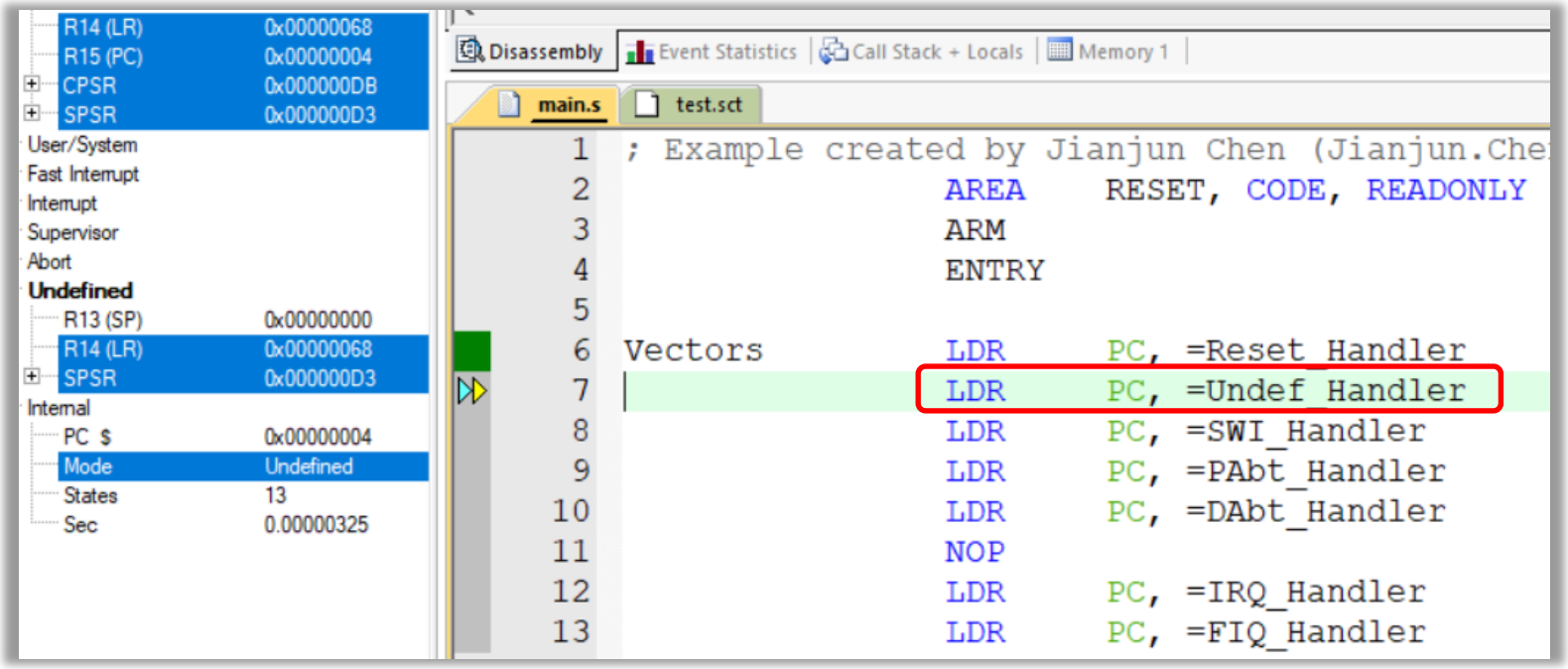
- The reset handler is the starting point of execution. It contains the “special_LDR” instruction we are trying to implement.

```
; Reset Handler
numbers          DCD     0x778899AA, 1, 2, 3, 4, 5, 6, 7, 8
                 EXPORT  Reset_Handler
Reset_Handler
                 LDR     R1, =numbers
                 MOV     R2, #8
                 MOV     R3, #1
special_LDR      DCD     0xE6910332
deadend          B       deadend
```

Will cause undef exception here.

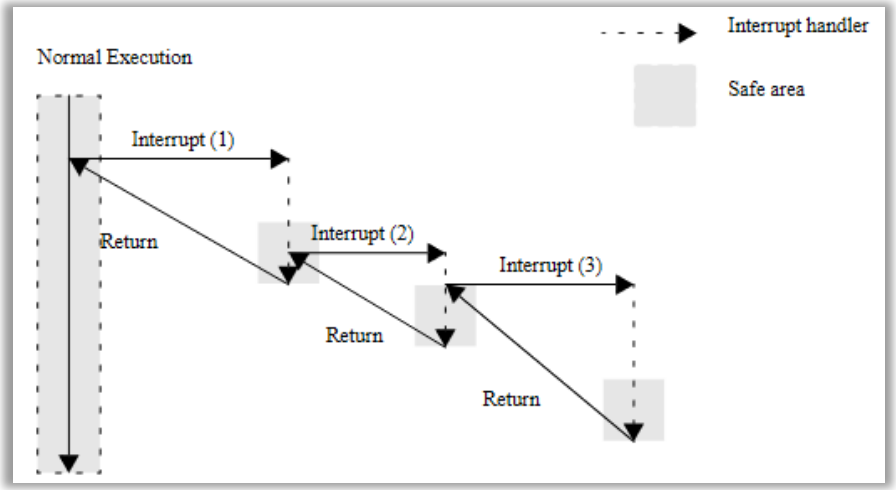
Switching Processor Mode

- Upon executing our special_LDR, the undef exception is triggered.
- The execution will branch to the Undef_Handler.



The Undef_Handler

- MRS copies *PSR to a general-purpose register.
- MSR does the opposite.



Need to back up them because exceptions can be nested (like functions)

```
35                  EXPORT  Undef_Handler
36  Undef_Handler
37                  STMDB   sp!, {r0-r12, LR}       ; Save Workspace & LR to Stack
                    MRS     r0, SPSR                ; Copy SPSR to r0
                    STR     r0, [sp, #-4]!          ; Save SPSR to Stack

                    LDR     r0, [lr,#-4]            ; r0 = undefined instruction
42
43                  ; start to check for the special LDR
44                  AND     r1, r0, #0x0FF00000     ; get the opcode
45                  TEQ     r1, #0x06900000         ; check the opcode (TEQ = EORS)
46                  BNE     not_special_ldr
47
48                  ANDS    r1, r0, #0x00000010     ; check whether the 4th bit is 1
49                  BEQ     not_special_ldr         ; If Z == 0, then it is not special LDR
50                  BL      do_special_ldr          ; if a valid special ldr, handle it
51  not_special_ldr
52                  ; insert tests for other undefined instructions here
53
54                  LDR     r1, [sp], #4            ; Restore SPSR to R1
55                  MSR     SPSR_cxsf, r1           ; Restore SPSR
56                  LDMIA   sp!, {r0-r12, PC}       ; Return to program after
57                                                  ; Undefined Instruction
```

The Undef_Handler

- `MRS` copies *PSR to a general-purpose register.
- `MSR` does the opposite.
  - The `cxsf` refers to the fields of the *PSR.

specifies the SPSR or CPSR fields to be moved. `fields` can be one or more of:

`c`
: control field mask byte, PSR[7:0] (privileged software execution)

`x`
: extension field mask byte, PSR[15:8] (privileged software execution)

`s`
: status field mask byte, PSR[23:16] (privileged software execution)

`f`
: flags field mask byte, PSR[31:24] (privileged software execution).

```
53
54          LDR     r1, [sp], #4          ; Restore SPSR to R1
55          MSR     SPSR cxsf, r1         ; Restore SPSR
56          LDMIA   sp!, {r0-r12, PC}     ; Return to program after
57                                        ; Undefined Instruction
```

Implementation of special_ldr

Please read the code "main.s" I uploaded

```
do_special_ldr
                ; This function implements:
                ; LDR  Rt, [Rn], Rm, shift_type Rs
                ;
                ; Example used:
                ; LDR R0, [R1], R2, LSR R3
                ; 1110 0110 1001 0001 0000 0011 0011 0010 -> 0xE6910332
                ; R1's address is "numbers": 0x778899AA, 1, 2, 3, 4, 5, 6, 7, 8
                ; R2's value is 8
                ; R3's value is 1
                ; After running this function:
                ;    R0 should store 0x778899AA
                ;    R1 should point to the next word of numbers

                ADD     r8, sp, #4              ; get the address of the old r0 valu
                                                ; Other registers' addresses are bas
                ; do "LDR  Rt, [Rn]" first
                AND     r1, r0, #0x000F0000
                LSR     r1, r1, #16             ; r1 = field value of Rn in the inst
                LDR     r7, [r8, r1, LSL #2]    ; r7 = the value stored in Rn
                LDR     r1, [r7]                ; r1 = the value pointed by [Rn]

                AND     r2, r0, #0x0000F000
                LSR     r2, r2, #12             ; r2 = field value of Rt in the inst
                STR     r1, [r8, r2, LSL #2]    ; Store the value pointed by [Rn] to

                ; do "Rm shift_type Rs"
                AND     r1, r0, #0x0000000F     ; r1 = field value of Rm in the inst
```

Result

File Edit View Project Flash Debug Peripherals Tools SVCS Window Help

Registers

Register	Value
Current	
R0	0x778899AA
R1	0x00000038
R2	0x00000008
R3	0x00000001
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000094
R15 (PC)	0x00000068
CPSR	0x400000DB
SPSR	0x000000D2

Disassembly

```
0x00000068  EAFFFFFE  B        0x00000068
    37:                STMDB   sp!, {r0-r12, LR}      ;
0x0000006C  E92D5FFF  STMDB   R13!,{R0-R12,R14}
    38:                MRS     r0, SPSR               ;
0x00000070  E14F0000  MRS     R0,SPSR
    39:                STR     r0, [sp, #-4]!         ;
    40:
```

Disassembly | Event Statistics | Call Stack + Locals | Memory 1

main.s | test.sct

```
25  Reset_Handler
26                  LDR     R1, =numbers
27                  MOV     R2, #8
28                  MOV     R3, #1
29  special_LDR     DCD     0xE6910332
30  deadend         B       deadend
31
```

Image from www.natureaustralia.org.au