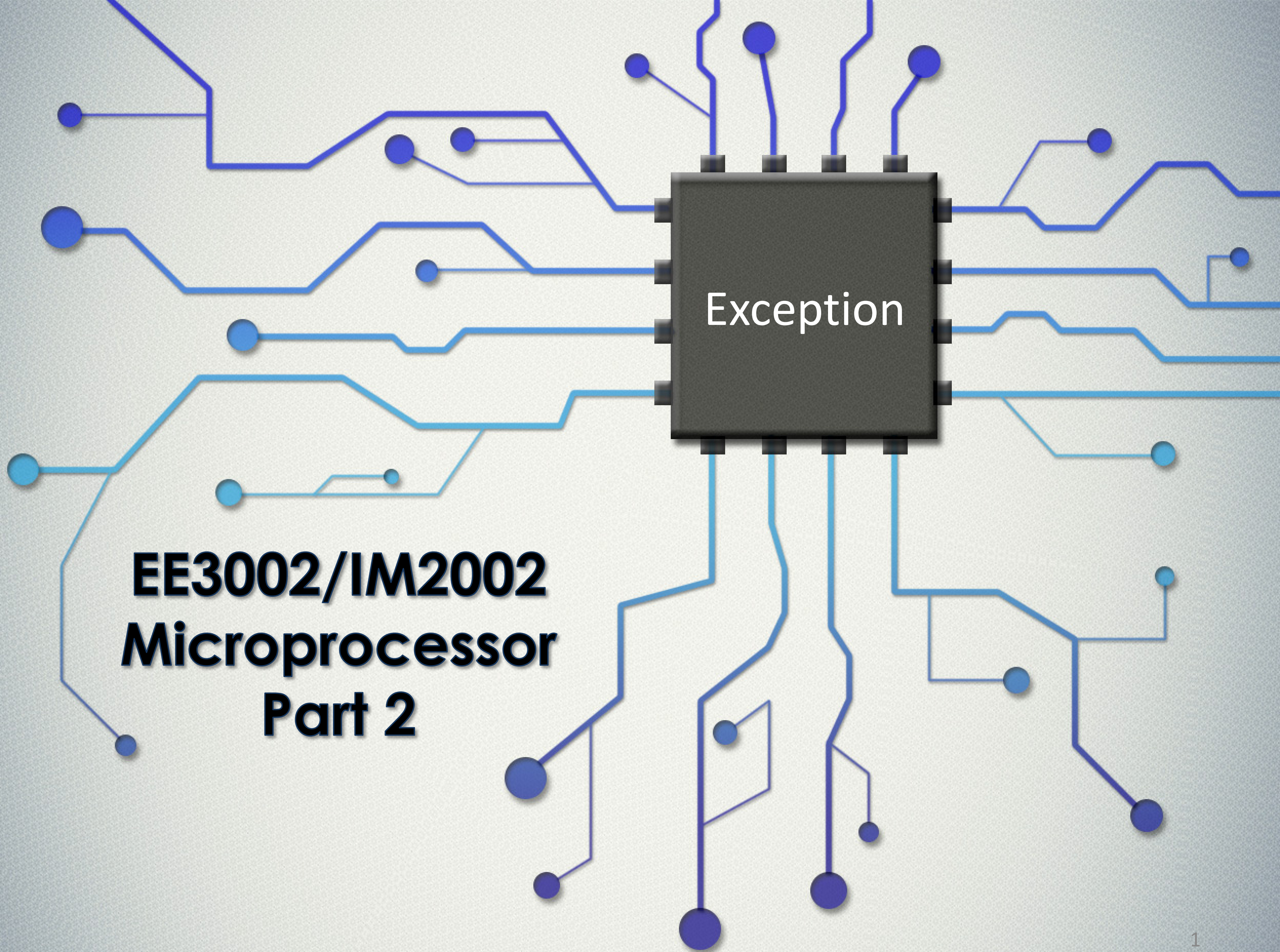


EE3002/IM2002 Microprocessor Part 2

Exception Definition From Computing Dictionary

- An error condition that changes the normal flow of control in a program.
- An exception may be generated ("raised") by hardware or software.
- Hardware exceptions include reset, interrupt or a signal from a memory management unit.
- Exceptions may be generated by the arithmetic logic unit or floating-point unit for numerical errors such as divide by zero, overflow or underflow or instruction decoding errors such as privileged, reserved, trap or undefined instructions.
- Software exceptions are even more varied and the term could be applied to any kind of error checking which alters the normal behaviour of the program.

Exception Handlers

- In general, an exception is *handled* (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an *exception handler*.

- They are really just subroutines, except:
  - Can occur between any two instructions
  - Are transparent to the running program
  - Are not usually requested by the running program
  - Call a subroutine at a predefined address (**exception vector)** determined by the type of exception, not the program

I/O Devices In Microcontrollers

- Very common for a device to request service from a microcontroller
- Example
  - Universal asynchronous receiver/transceiver (UART) attached to a wireless keyboard
  - Whenever a character shows up, it is stored in some memory location waiting for the processor to get it
- 3 ways it can be handled

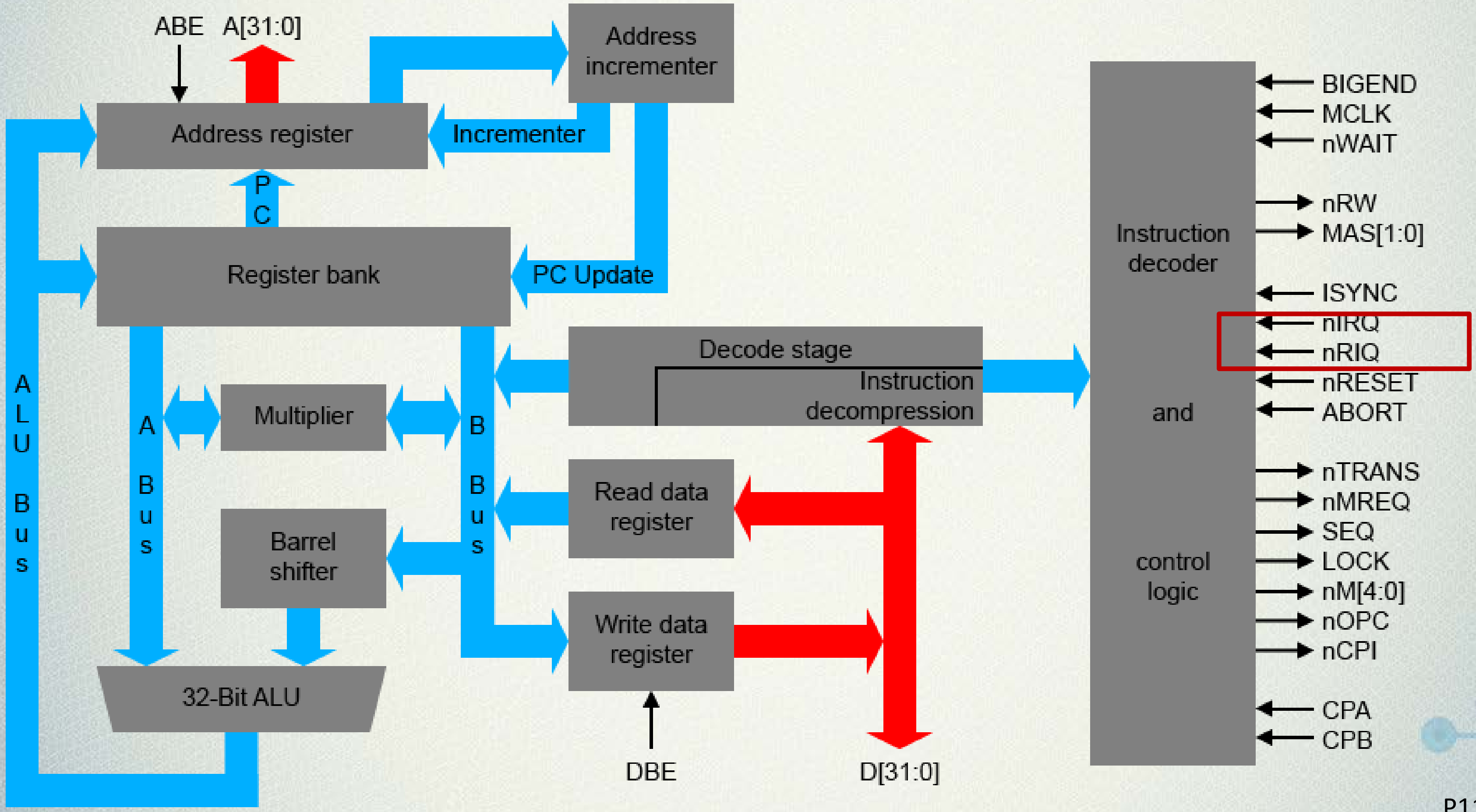
3 Ways To Response To A Device Request

1. Processor runs a loop endlessly doing nothing except waiting for character to show up
2. Polling – processor occasionally check through it - may also be doing some other things
3. Have the device interrupt the processor when there is new data available → the **best option**

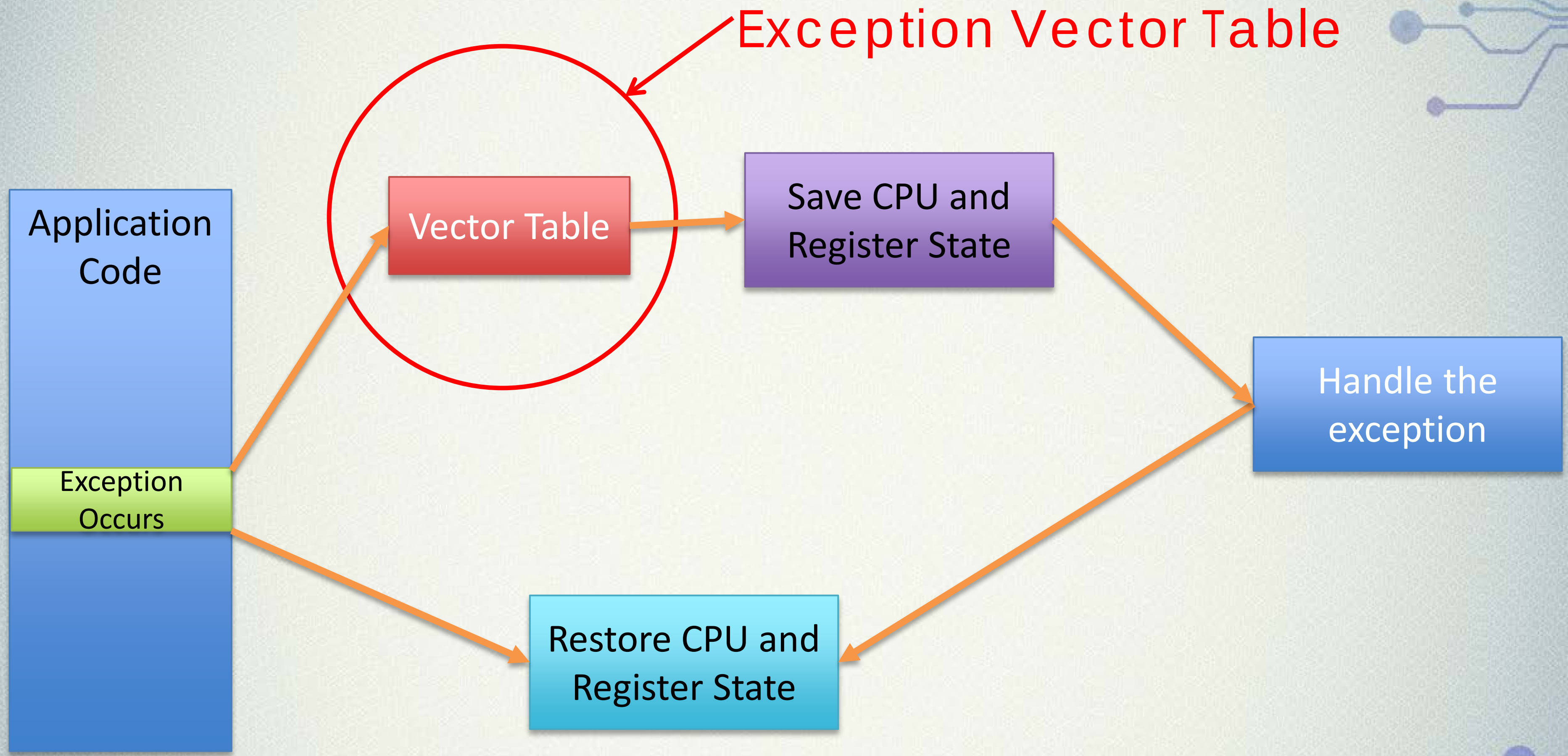
Conclusion: interrupt is the best approach and is popularly used

Interrupt Structure: 2 hw And 1 sw

- 2 external lines: nIRQ(low priority) and nFIQ – high priority
- Software Interrupt or SWI



Exception Handling Process



Exception Vector Table

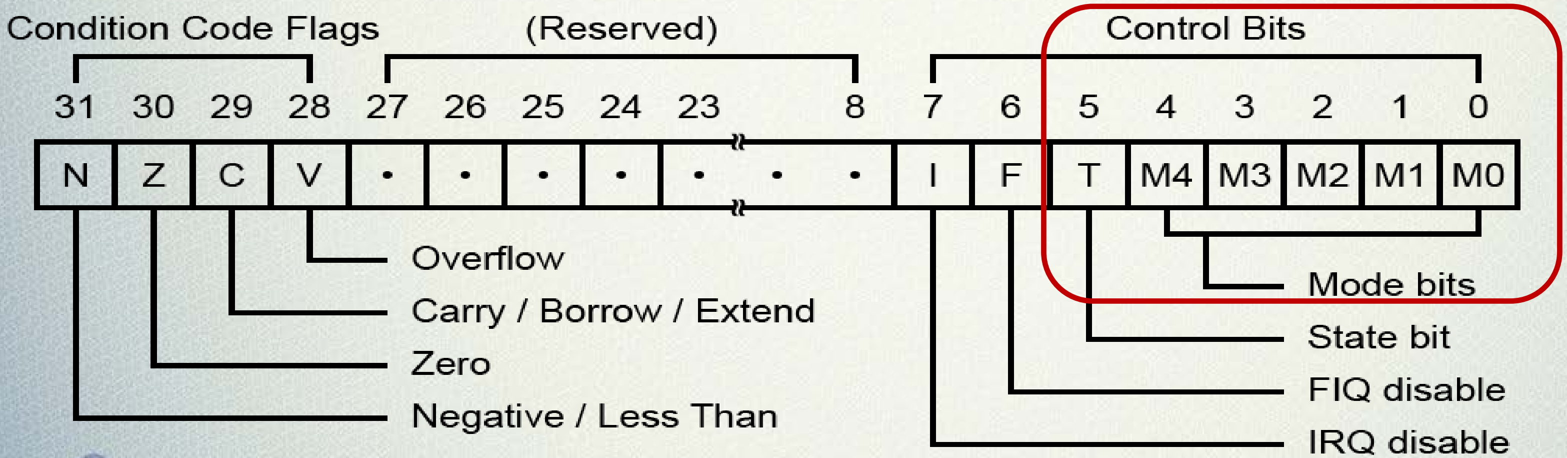
- A table that the processor jumps to when an exception occurs
- The location in the table where the processor jumps to when an exception happens is predetermined by the type of exception
- The entry in the table is an instruction which is a branch instruction to the address of the subroutine that handles this exception (address of exception handler)

Exception Vector Table

Exception	**Jump to Address …**	**Mode in Entry**
FIQ	0x0000001C	FIQ
IRQ	0x00000018	IRQ
Reserved	0x00000014	Reserved
Data abort	0x00000010	Abort
Prefetch abort	0x0000000C	Abort
Software interrupt	0x00000008	Supervisor
Undefined instruction	0x00000004	Undefined
Reset	0x00000000	Supervisor

Example: processor jumps to 0x00000018 and enters IRQ mode when IRQ occurs

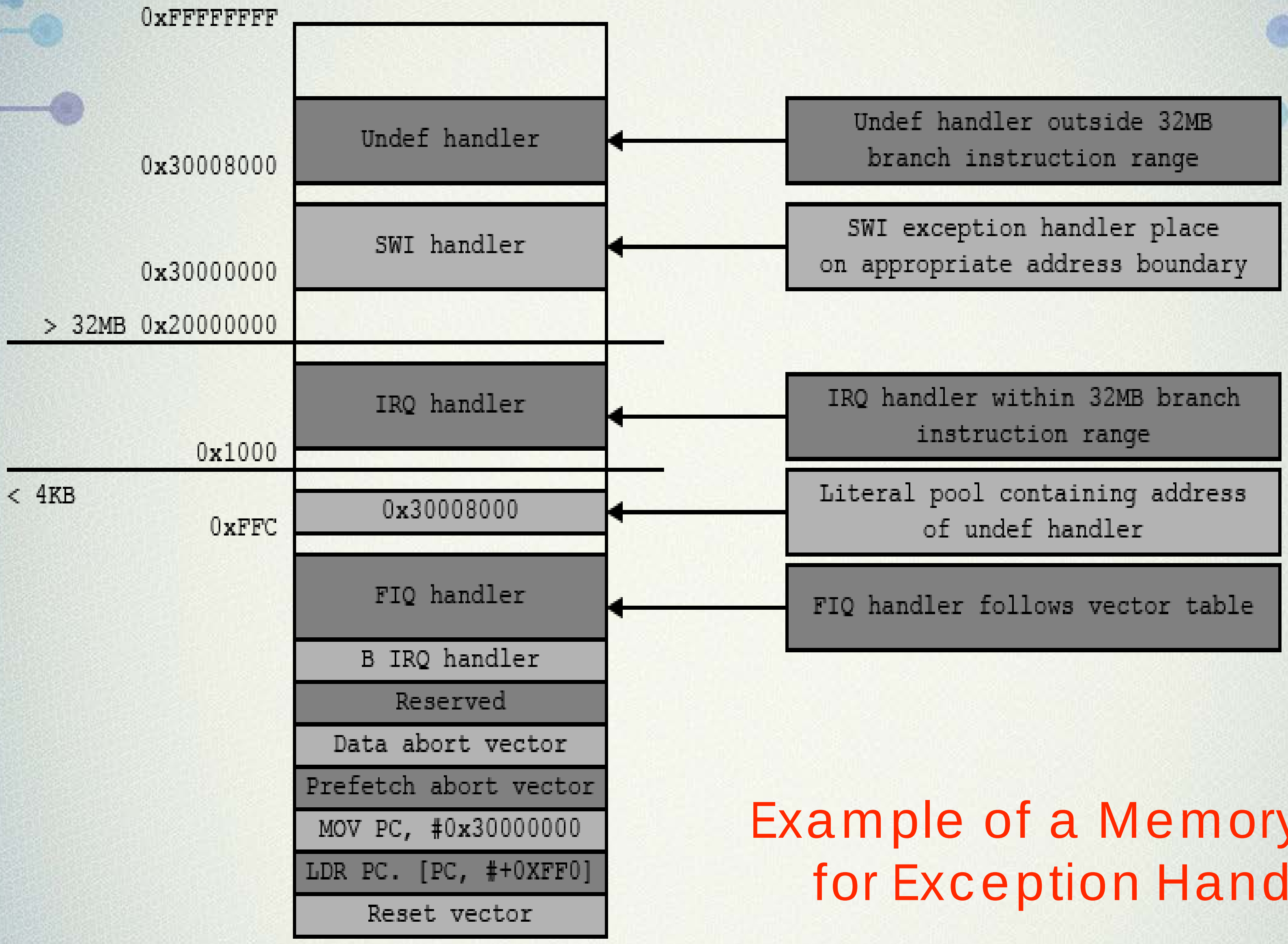
Mode Bits		Mode	Purpose
Binary	Hex		
10000	0x10	USR (user)	Normal operating mode
10001	0x11	FIQ (fast interrupt)	High priority interrupt
10010	0x12	IRQ (interrupt)	Low priority interrupt
10011	0x13	SVC (supervisor)	Upon startup/reset, software interrupt
10111	0x17	ABT (abort)	Memory access violation
11011	0x1B	UND (undefined)	Undefined/unimplemented instruction
11111	0x1F	SYS (system)	Operating system privileged mode



Register cpsr/spsr

Types of Vector Table Entry

- **The vector table entries commonly contain branch instructions of one of the following forms**
- ▪ MOV pc, #immediate
- This MOV copies an immediate value into the PC. It let you span the full address space but at limited alignment (the address must be an 8-bit immediate rotated *right* by an *even* number of bits).
- ▪ B <addr>
- This branch instruction provide a branch relative from PC. However, this offset from pc < +/- 32M
- ▪ LDR pc, [pc, #offset]
- This LDR instruction loads the handler address from memory to the pc. The address is an absolute 32-bit value stored close to the vector table. Loading this absolute literal value results in a slight delay in branching to a specific handler due to the extra memory access. However, you can branch to any address in memory.



0xFFFFFFFF
Undef handler
0x30008000
SWI handler
0x30000000
> 32MB 0x20000000
IRQ handler
0x1000
< 4KB
0x30008000
0xFFC
FIQ handler
B IRQ handler
Reserved
Data abort vector
Prefetch abort vector
MOV PC, #0x30000000
LDR PC. [PC, #+0XFF0]
Reset vector
Undef handler outside 32MB branch instruction range
SWI exception handler place on appropriate address boundary
IRQ handler within 32MB branch instruction range
Literal pool containing address of undef handler
FIQ handler follows vector table

Example of a Memory Map for Exception Handlers

Memory Map Of Exception Handlers – Notes 1

- Each block holds the corresponding exception handler
- After executing instructions in the handler, it may return to the point from where it left
- Reset exception can use a B <address> as long as <address - pc> < 32MB
- Example

–B resetHandler

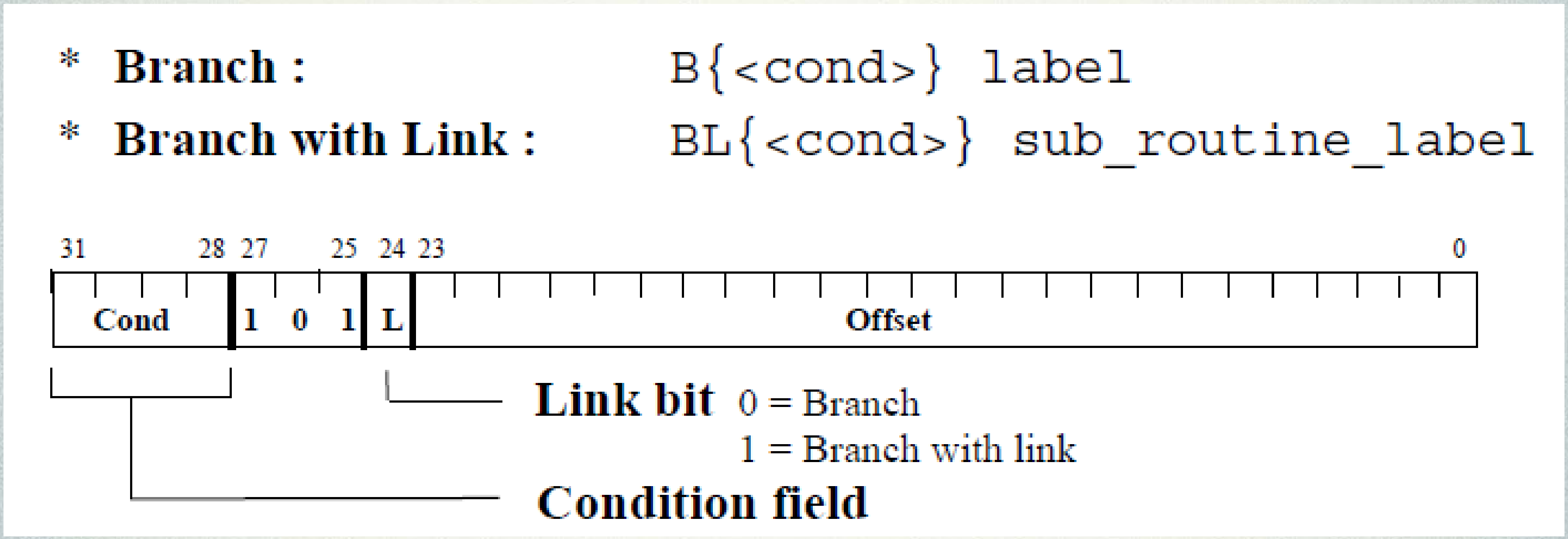
–assume resetHandler address = 0x4000

–and pc = 0x0 + 0x8 (2 instructions ahead of 0x0)

–(address of resetHandler – pc) = offset from pc

–= (0x4000 – 0x8) = 0x3FF8 < 32MB (0x1FFFFFF)

Review Of Branch Instruction



* **Branch :** `B{<cond>} label`

* **Branch with Link :** `BL{<cond>} sub_routine_label`

Offset Of Branch Instruction

- **The offset for branch instructions is calculated by the assembler**
  - Difference between the branch instruction and the target address minus 8 (to allow for the pipeline)
  - This gives a 26 bit offset which is right shifted 2 bits (as the bottom two bits are always zero as instructions are word-aligned) and stored into the instruction encoding
  - This gives a range of +/- 32Mbytes (2s complement offset) (+/- 0x1FFFFFF)
- **When executing the instruction, the processor:**
  - shifts the offset left two bits, sign extends it to 32 bits, and adds it to register pc.

Memory Map Of Exception Handlers – Notes 2

- **LDR pc, [pc, #0xFF0]**
- Undefined instruction handler address is 0x30008000 and is stored as literal at 0xFFC
- Offset of literal from pc = 0xFFC – 0xC = 0xFF0
- Why pc = 0xC?
  - ✓ 3 stages pipeline, pc is pointing to instruction being fetched (2 instructions away → 8 bytes away)
  - ✓ Exception address = 0x4, hence pc = 0x4+0x8 = 0xC
- Review of LDR instructions
  - ✓ 12 bits for offset, max offset = 2^12 = 4KB
  - ✓ Hence 0xFFF is last reachable address
  - ✓ 0xFFC is the furthest offset.
  - ✓ 0x1000 is too far away! WHY?

Review - Single Register Data Transfer

- **The basic load and store instructions are:**
- Syntax: <LDR|STR>{<cond>}{<size>} Rd, <address>
- Load and Store Word or Byte or halfword
- – LDR / STR / LDRB / STRB / LDRH / STRH
- For Signed Byte or Halfword - load value and sign extend it to 32 bits
  - – LDRSB / LDRSH
- All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR, e.g. LDREQB
- Examples:
  - Store content of r0 to location pointed to by content of r1
    - ✓ STR r0, [r1]
  - Load r2 with content of memory location pointed to by content of r1
    - ✓ LDR r2, [r1]

LDR Pseudo Instruction To Load Numeric Constant To Register

- Although the MOV/MVN instructions will load a large range of constants into a register, sometimes they will not generate the required constant
- The assembler also provides a method which will load *ANY 32* bit constant:
  - LDR rd, =numeric constant
- If the constant can be constructed using either a MOV or MVN then this will be the instruction actually generated
- Otherwise, the assembler will produce an LDR instruction with a pc-relative address to read the constant from a literal pool
  - LDR r0, =0x42 ;generates MOV r0, #0x42
  - LDR r0, =0x55555555
  - ; generate LDR r0, [pc, offset to lit pool]

LDR Pseudo Instruction

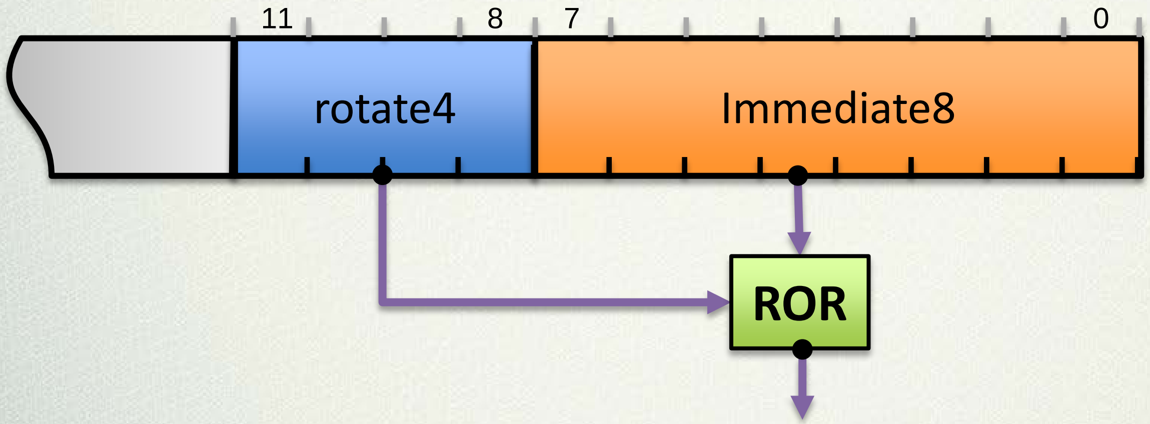
- As this mechanism will always generate the best instruction for a given case, it is the recommended way of loading constants
- The assembler uses literal pools to hold certain constant values that are to be loaded into registers
- It places a literal pool at the end of each section
- In large sections the default literal pool can be out of range of one or more LDR pseudo instructions
- The offset from the pc to the constant must be less than 4KB (+/- 0xFFF) but can be in either direction

Memory Map Of Exception Handlers – Notes 3

- SWI handler can be reached using the MOV instruction
- An address that is far away, like 0x30000000, may be generated by using the byte rotation scheme
- Hence MOV instruction can be used for such cases
- Equivalent to MOV pc, #0x03, 4

Review Of MOV Instruction

- You can't fit an arbitrary 32-bit value into a 32-bit instruction word
- MOV instructions have 12 bits of space for values in their instruction word
- This is arranged as a four-bit rotate value and an eight-bit immediate value



Memory Map Of Exception Handlers – Notes 4

- ❖ FIQ needs to be served quickly
- ❖ Usually critical which demands immediate attention
- ❖ Hence it is not branched!
  - ✓ Save execution of one branch instruction
  - ✓ The exception address 0x1C is the first instruction of the FIQ handler

Priority Of Exception And Flags

Exception	Priority	I bit	F bit
Reset	1	1	1
Data abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch abort	5	1	-
SWI	6	1	-
Undefined instruction	6	1	-

Decide which of the currently raised exceptions is more important

Decide if the exception handler can be interrupted or not

Exception Priority (1)

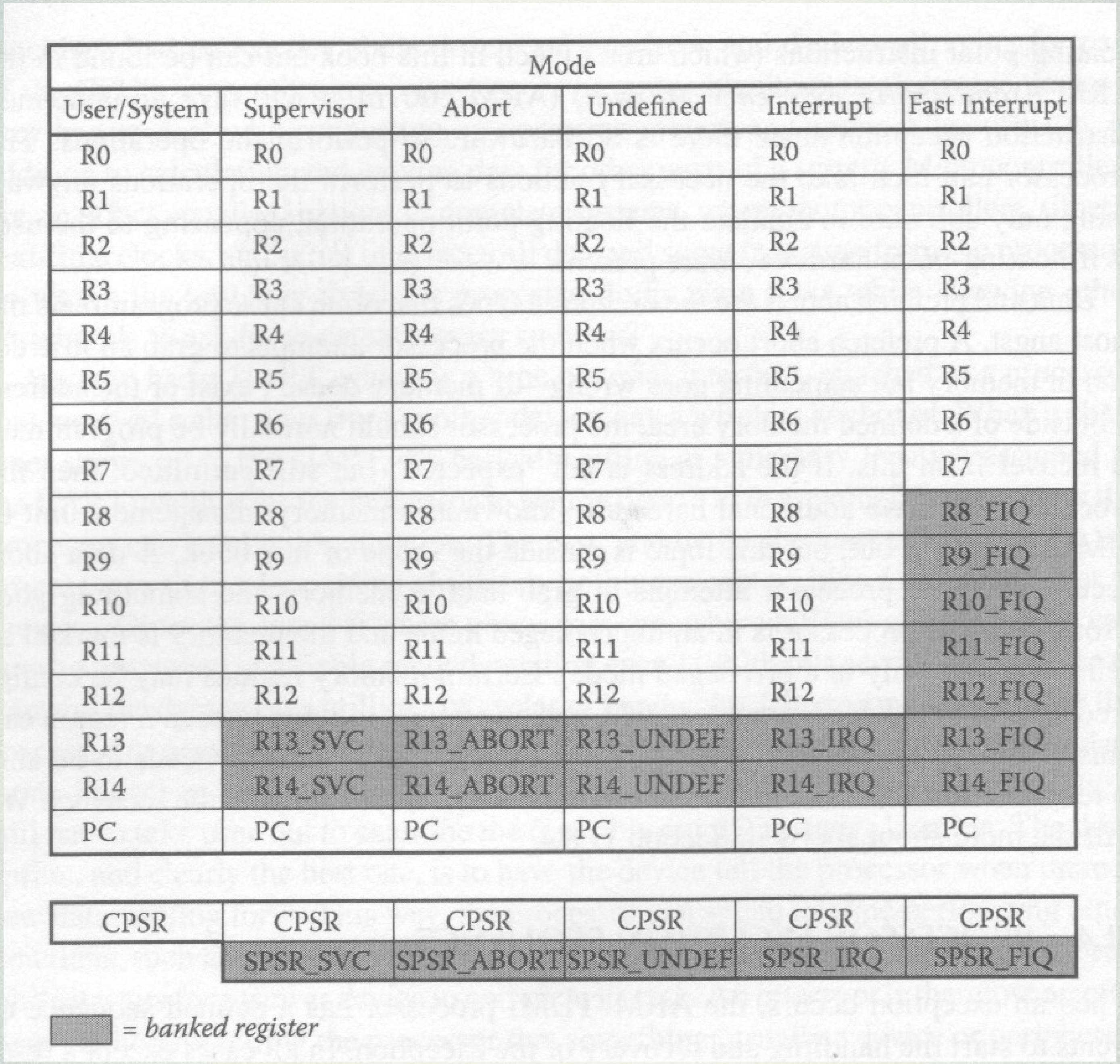
When multiple exceptions are valid at the same time (i.e. more than one exception occurs during execution of an instruction), they are handled by the core (after completing the execution of the current instruction) according to the following priority scheme.

1. Reset
2. Data Abort
3. FIQ
4. IRQ
5. Prefetch Abort
6. Undefined Instruction, SWI

Exception Priority (2)

The Undefined Instruction and SWI cannot occur at the same time because they are both caused by an instruction entering the execution stage of the ARM instruction pipeline, so are mutually exclusive and thus they have the same priority.

Registers and Banked 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*

Registers and Banked Registers (1)

- ARM has 37 registers in all.
- 20 registers are hidden from program at different times. These registers are called banked registers.
- Banked registers become available, when processor is in particular mode.
- The user registers R0-R7 are common to all operating modes.
- FIQ mode has its own R8-R14 that replace the user registers. Why?
- Each of the other modes have their own R13 and R14.

Registers and Banked Registers (2)

- There is only one CPSR common to all modes and there are 5 Saved program Status registers (SPSR), one for each mode
- If processor goes to some other modes from user mode, example user mode to FIQ mode, FIQ has bank registers from R8 to R14. So effectively a fresh copy of R8 to R14 is made available in FIQ mode.
- When mode change takes place, content of CPSR will be saved to the SPSR of the new mode. When returned back, content of the appropriate SPSR should be restored back to CPSR.

Processor Exception Sequence

- When handling an exception in ARM
  - Depends on the type of exception, it may either handle the exception immediately or wait until the completion of the current instruction
  - Copy the address of the next instruction to the appropriate link register
  - Copies the CPSR into the appropriate SPSR
  - Forces the CPSR mode bits to a value which depends on the exception

Processor Exception Sequence

- Forces the register pc to fetch the next instruction from the relevant exception vector
- It may also set the interrupt disable flags to prevent otherwise unmanageable nesting of exceptions.
- If the processor is in THUMB state when an exception occurs, it will automatically switch into ARM state when the register pc is loaded with the exception vector address.

Processor Exception Sequence

- On completion, the exception handler:
    - Copy the link register, minus an appropriate offset, to the register pc.
    - The offset depends on the type of exception.
    - Copies the SPSR back to the CPSR
    - Clears the interrupt disable flags, if they were set on entry

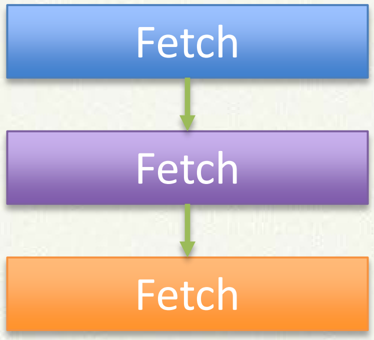
Link Register In Exception

- Link register is used to return the ***pc*** register to the appropriate place in the interrupted task
- Note: lr = pc-4 at the time of handling the exception
- However, pc may go beyond the current instruction, depending on the type of exception
- Hence the content of lr should be modified according to the type of exception

The Instruction Pipeline

- The ARM7TDMI uses a 3-stage pipeline in order to increase the speed of the flow of instructions to the processor
  - Allows several operations to be performed s

ARM	Thumb
PC	PC
PC – 4	PC – 2
PC – 8	PC - 4



FETCH Instruction fetched from memory

Decoding of registers used in instruction

Register(s) read from Register Bank
Shift and ALU operation
Write register(s) back to Register Bank

- The PC points to the instruction being fetched , not executed
  - Debug tools will hide th is from you
  - This is now part of the ARM Architecture and applies to all processorsimultaneously, rather than serially

Adjustment To Link Register Lr On Return From Exception

Exception	Return address
Reset	None
Data abort	lr-8
Prefetch abort	lr-4
FIQ, IRQ	lr-4
SWI, undefined instruction	lr

"Return Address" SWI and Undefined Instruction

- Exception is handled immediately
- Exception handler is called, in effect, by the instruction itself
- Thus pc is not updated at the point that the lr value is calculated.
- lr = (pc - 4), which actually points to the next instruction to be executed upon return from exception
- No adjustment to lr is needed on return from exception
- Therefore return address = lr

“Return Address” FIQ and IRQ

- Interrupt is only handled upon completing execution of the current executing instruction
- Interrupt handler called after execution is completed
- This implies that pc will be updated before the lr value is calculated
- lr = (pc - 4) actually points to two instructions beyond where exception has occurred
- Hence lr need to be adjusted by one instruction on return from interrupt
- Therefore return address = (lr – 4)

“Return Address” Prefetch Abort

- Exception taken when instruction reaches execute stage of pipeline
- Thus pc is not updated at this point
- lr = (pc - 4) actually points to next instruction after one that caused the abort
- Thus to retry executing aborted instruction, lr need to be adjusted by one instruction on return from exception
- Therefore return address = (lr – 4)

“Return Address”Data Abort

- Exception taken after instruction is executed
- pc will then be updated
- lr = (pc - 4) actually points to two instructions after the one that caused the abort
- Thus to retry executing aborted instruction, lr need to be adjusted by two instructions on return from exception
- Therefore return address = (lr – 8)

To Return To Previous Mode And Interrupted Program (1)

- **Need to do two things:**
  1. Restore CPSR from SPSR_<mode>
  2. Restore pc using "*return address*" stored in lr_<mode>
- **Can be done in one instruction, which depends upon exception**
  - For SWI and Undefined Instructions, use MOVS pc, lr
  - For FIQ, IRQ and Prefetch Abort, use SUBS pc, lr, #4
  - For Data Abort, use SUBS pc, lr, #8
- Adding the 'S' in privileged mode will copy SPSR into CPSR
- This will automatically restore the original mode, interrupt settings and condition codes

To return to previous mode and interrupted program (2)

- **Alternatively, we can use LDM instruction with the '^' option, example:**
  - For SWI and Undefined Instructions
    - LDMFD sp!, {… , pc}^ ;return
  - For FIQ, IRQ and Prefetch Abort
    - SUBS lr, lr, #4 ;adjust return address
    - STMFD sp!, {… , lr} ;save registers and adjusted lr
    - … ;body of handler
    - LDMFD sp!, {… , pc}^ ;return
  - For Data Abort
    - SUBS lr, lr, #8 ;adjust return address
    - STMFD sp!, {… , lr} ;save registers and adjusted lr
    - … ;body of handler
    - LDMFD sp!, {… , pc}^ ;return
- With '^' option, SPSR copy to CPSR, which automatically restore to the original mode, interrupt settings and condition codes

Exception Handlers

- Programmers must write their exception handlers
- Once the mode has been changed, the handler will have to access its own stack pointer, its link register and its SPSR
- Any other registers must first be saved in the stack
- However for FIQ, there are 5 additional registers (r8_FIQ – r12_FIQ) for faster processing
- Upon return, the handler is responsible to restore the state and content of all registers

Procedures For Handling Exceptions

- **Reset Exception**
  - On power up, the address bus receives 0x00000000 to access its first instruction, usually a branch instruction
  - This branch will take the processor to the reset handler
  - Common tasks of reset handler
    - ✓ Set up exception vectors
    - ✓ Initialize the memory system
    - ✓ Initialize all required processor mode stacks and registers
    - ✓ Initialize any critical I/O devices
    - ✓ Initialize any peripheral registers, control registers, or clocks
    - ✓ Enable interrupts
    - ✓ Change processor mode and/or state
  - Reset handler does not have a return sequence

Procedures For Handling Exceptions

- **Undefined Instruction Exception**
  - Unrecognized bit pattern as a valid instruction
  - However it can also be one of the 2 following cases
  - Intended for a coprocessor (eg., floating point operation)
  - A coprocessor not responding to an instruction either it is having its own exception (eg., divide by zero) or is not in a privileged mode

Procedures For Handling Exceptions

Interrupts

- 2 types
  - Fast interrupt (FIQ)
  - Low priority interrupt (IRQ)
- FIQ has highest priority and will disable IRQ when servicing a request
- 2 ways to have more interrupts
  - Wired OR several interrupts and let processor polls the device that trigger the interrupt
  - Use an external interrupt controller → Vectored Interrupt Controller (VIC)

Vectored Interrupt Controller (VIC)

- Specialized hardware
- Makes design more straightforward
- Provide enough information to the processor in determining the device that raise an interrupt, such as via a register
- Assign priority to different interrupts
- It has multiple inputs but only 2 outputs (FIQ and IRQ) as well as the address of the interrupt handler in a register
- Needs to
  - Configure registers
  - Enable interrupts
  - Defined an initialize memory locations

Procedures For Handling Exceptions

- **Prefetch abort**
  - Attempt to fetch an instruction with invalid address
  - The abort handler tries to fix the problem or die a graceful death
  - Alternatively it might just reset the system
- **Data abort**
  - Load and store instructions generate data abort if the address is not valid or the memory area is privileged
  - Usually fatal and has to be reported

Procedures For Handling Exceptions

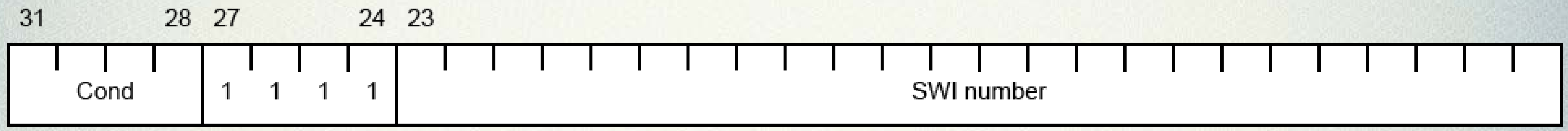
SWI

- Software interrupts, or SWIs, are generated by using the ARM instruction SWI
- This cause an exception and forces the processor into supervisor mode, which is privileged
- User program requests a service by encoding a number in the SWI instruction (24 bits for ARM and 8 bits for THUMB)
- The handler must determines the state (ARM or THUMB) to locates the SWI instruction to extract the number and hence perform the task accordingly

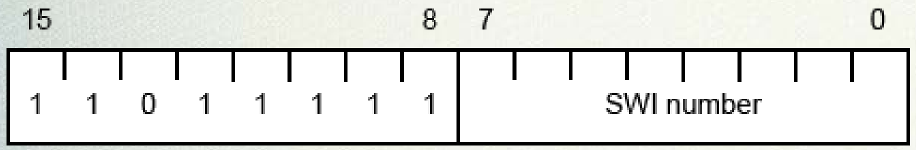
SWI Instruction Format

For ARM: SWI instruction is found at effective address <**lr-4**> after entering SWI handler

ARM format:



THUMB format:

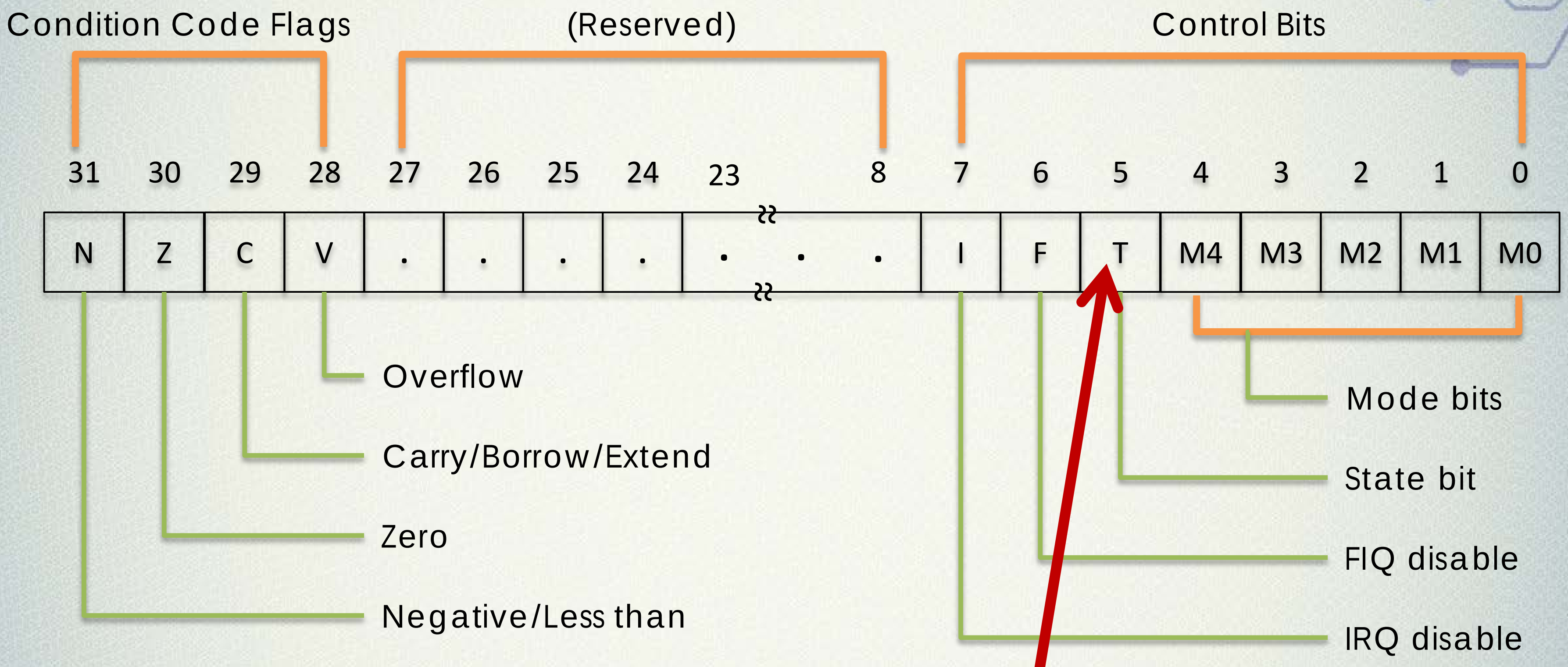


For THUMB: SWI instruction is found at effective address <**lr-2**> after entering SWI handler

SWI *Number*

- *number (24 bits for ARM; 8 bits for THUMB)*
- The SWI instruction causes a SWI exception
- This means that
  - the processor state changes to ARM, if necessary
  - the processor mode changes to Supervisor,
  - the CPSR is saved to the Supervisor Mode SPSR,
  - and execution branches to the SWI handler
- It is retrieved by the SWI handler to determine what service is being requested
- Eg., SWI 0x123456

Review Of CPSR/SPSR Registers



Note:

cpsr_c (I, F, T, Mode)

cpsr_f (flags)

Tbit

0: ARM

1: THUMB

Example - SWI (1)

```
ram_base    EQU 0x40000000
Mode_USR  EQU 0x10
    AREA SWIexample, CODE, READONLY
    ENTRY
    ;init exception vector table
    B       reset_handler
    B       undefined_instruction_handler
    B       SWI_handler
    B       prefetch_abort_handler
    B       data_abort_handler
    SPACE 4
    B       IRQ_handler

FIQ_handler
    SPACE 256
    ;handle FIQ here
```

Example - SWI (2)

```
    AREA resetHandler, CODE, READONLY
reset_handler           ;handle reset exception here
     ;start up in supervisor mode
    LDR   sp, =ram_base + 0x200  ;init sp in supervisor mode
    MSR cpsr_c, #Mode_USR;switch to user mode, normal operating mode

     ;now in user mode
    LDR  sp, =ram_base + 0x100   ;init sp in user mode
    SWI 1             ;trigger software interrupt 1
stop B stop         ;return from SWI

undefined_instruction_handler B undefined_instruction_handler
    ;handle undefined instruction exception here
```

Example - SWI (3)

```
    AREA swiHandler, CODE, READONLY
SWI_handler             ;handle software interrupt here
Tbit  EQU 0x20          ;Thumb bit of CPSR/SPSR, that is, bit 5
    STMFD   sp!, {r0-r2,lr}      ;store registers
    MRS      r0, spsr   ;copy SPSR into a general purpose register
    TST      r0, #Tbit       ;test for Thumb state?

    LDRNEH  r0,[lr,#-2]     ;Yes: load halfword and...
    BICNE     r0,r0,#0xFF00    ; ...extract SWI number

    LDREQ     r0,[lr,#-4]    ;No: load word and...
    BICEQ     r0,r0,#0xFF000000  ;...extract SWI number

    ;r0 now contains the SWI number
    LDR        r1, =switable     ;load start address of swi jump table
    LDR        pc, [r1, r0, LSL#2]    ;Jump to the appropriate routine
```

Example - SWI (4)

```
switable
    DCD      do_swi_0
    DCD      do_swi_1
    DCD      do_swi_2
do_swi_0         ;handle SWI 0
    ADD      r2, r0, #0x100
    LDR      r1, =ram_base
    STR      r2, [r1]   ;Store result in memory
    LDMFD sp!, {r0-r2, pc}^      ;Restore registers and return
do_swi_1         ;Handle SWI 1
    ADD      r2, r0, #0x500
    LDR      r1, =ram_base
    STR      r2, [r1]   ;Store result in memory
    LDMFD sp!, {r0-r2, pc}^      ;Restore registers and return.
```

Example - SWI (5)

```
do_swi_2          ;Handle SWI 2
    MOV   r2, #0
    LDR   r1, =ram_base
    STR   r2, [r1]        ;Store result in memory
    LDMFD sp!, {r0-r2, pc}^    ;Restore registers and return

prefetch_abort_handler   B  prefetch_abort_handler
    ;handle prefetch abort exception here

data_abort_handler  B  data_abort_handler
    ;handle data abort exception here

IRQ_handler  B  IRQ_handler
    ;handle interrupt here
    END
```

psr Transfer

- **MRS{*cond*} *Rd*, *psr***
  - Rd – destination register
  - *psr* – could be CPSR or SPSR
  - Move the content of a *psr* to a general-purpose register

- **MSR{cond}, psr,** *Rn or immed*
  - Move the content of a general purpose register or immediate value to *psr*
  - *psr* could also be cpsr_f (set flags) or cpsr_c (set I, F, Mode)

Review Of TST Instruction

TST{*cond*} *Rn*, *Operand2*

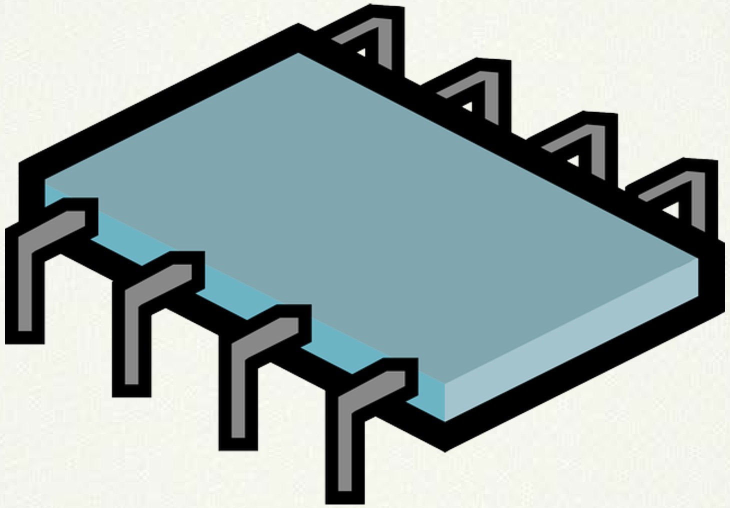
- *Rn* – register holding the 1st operand
- *Operand2* – flexible 2nd operand
- TST instruction performs a bitwise AND operation on the value in *Rn* and the value of *Operand2*
- Same as a ANDS instruction, except that the result is discarded.
- These instructions:
  - update the N and Z flags according to the result
  - can update the C flag during the calculation of *Operand2*
  - do not affect the V flag

Review Of BIC Instruction

BIC{*cond*}{S} *Rd*, *Rn*, *Operand2*

- *Rd* – register for the result
- *Rn* – register holding 1st operand
- *Operand2* – flexible second operand
- The BIC (BIt Clear) instruction performs an AND operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*

Demo - SWI



Startup Code For ARM (1)

The startup code consists of:

- the reset handler of your application
- initialization functions that set up the environment and peripherals before the main body of your application can run
- it also reserves memory for the stack and heap and initializes them
- These operations are specific to a particular core and target

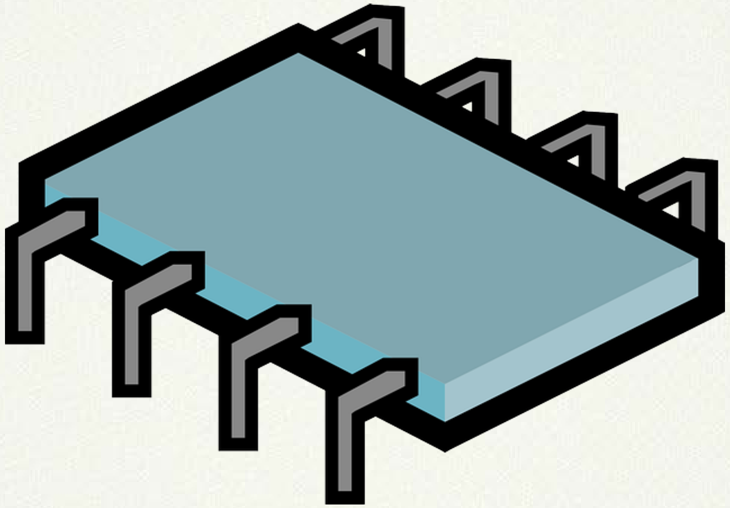
Startup Code For ARM (2)

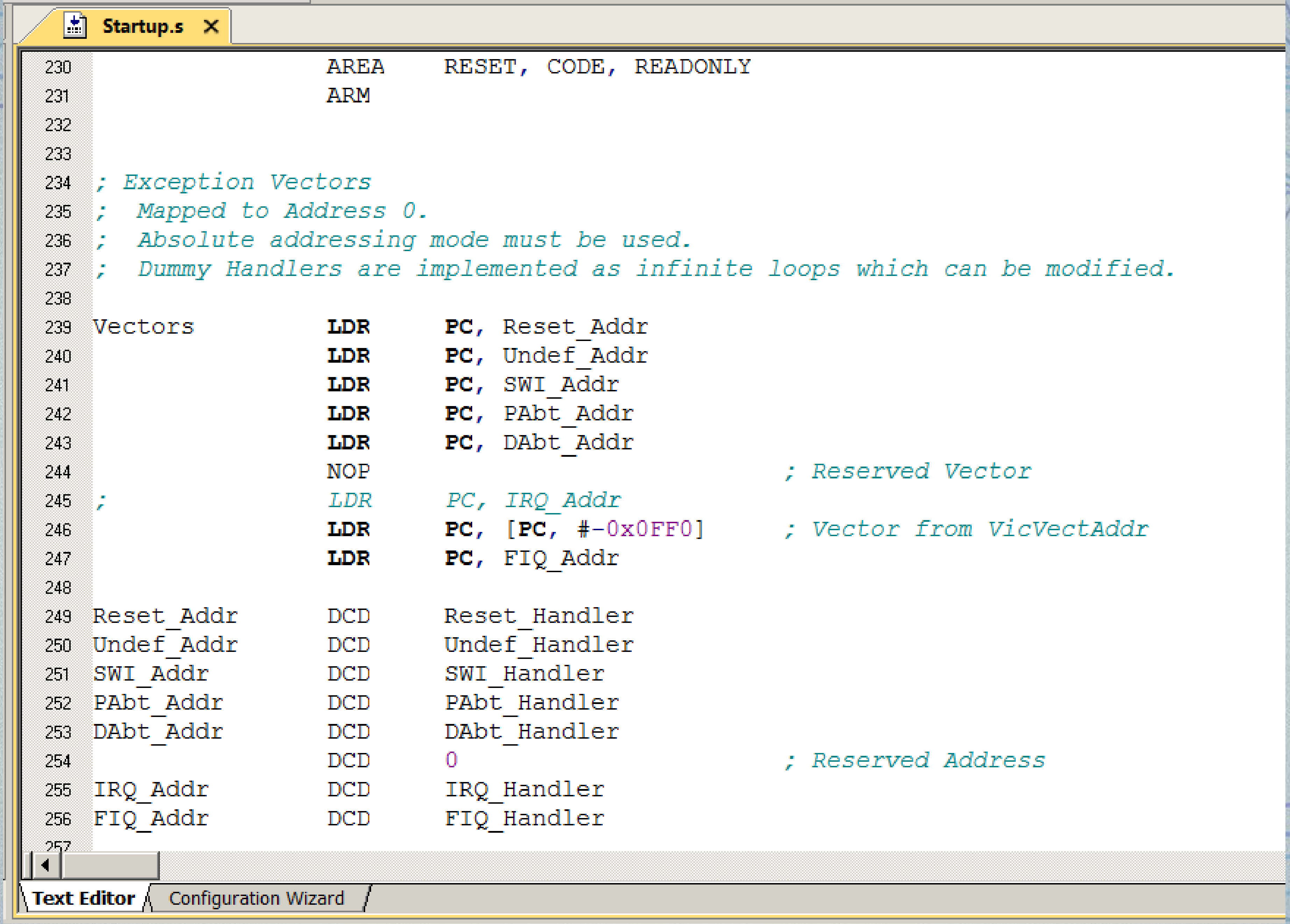
- If your system is simple, it might be sufficient to specify the C library entry point (the __main() function) as your reset handler in the vector table, and perform additional initialization from the main() function in your own code
- However, if there are peripherals that require critical initialization, you might need to write a short assembly code function to act as your initial reset handler before branching to __main()

Startup Code For Keil µvision Project

- The target program requires a startup-code to initialize the CPU to match the configuration of the hardware design
- When creating a project, µVision adds the device specific CPU startup-code for most devices to the project
- The startup-code executes immediately upon RESET of the target system and performs the following operations:
  - Defines interrupt and exception vectors.
  - Configures the CPU clock source (on some devices).
  - Initializes the external bus controller.
  - Copies the exception vectors from ROM to RAM for systems with memory remapping
  - Initializes other low level peripherals, if necessary
  - Reserves and initializes the stack for all modes
  - Reserves the heap
  - Transfers control to the main C function

Demo – Startup Code For NXP LPC2104





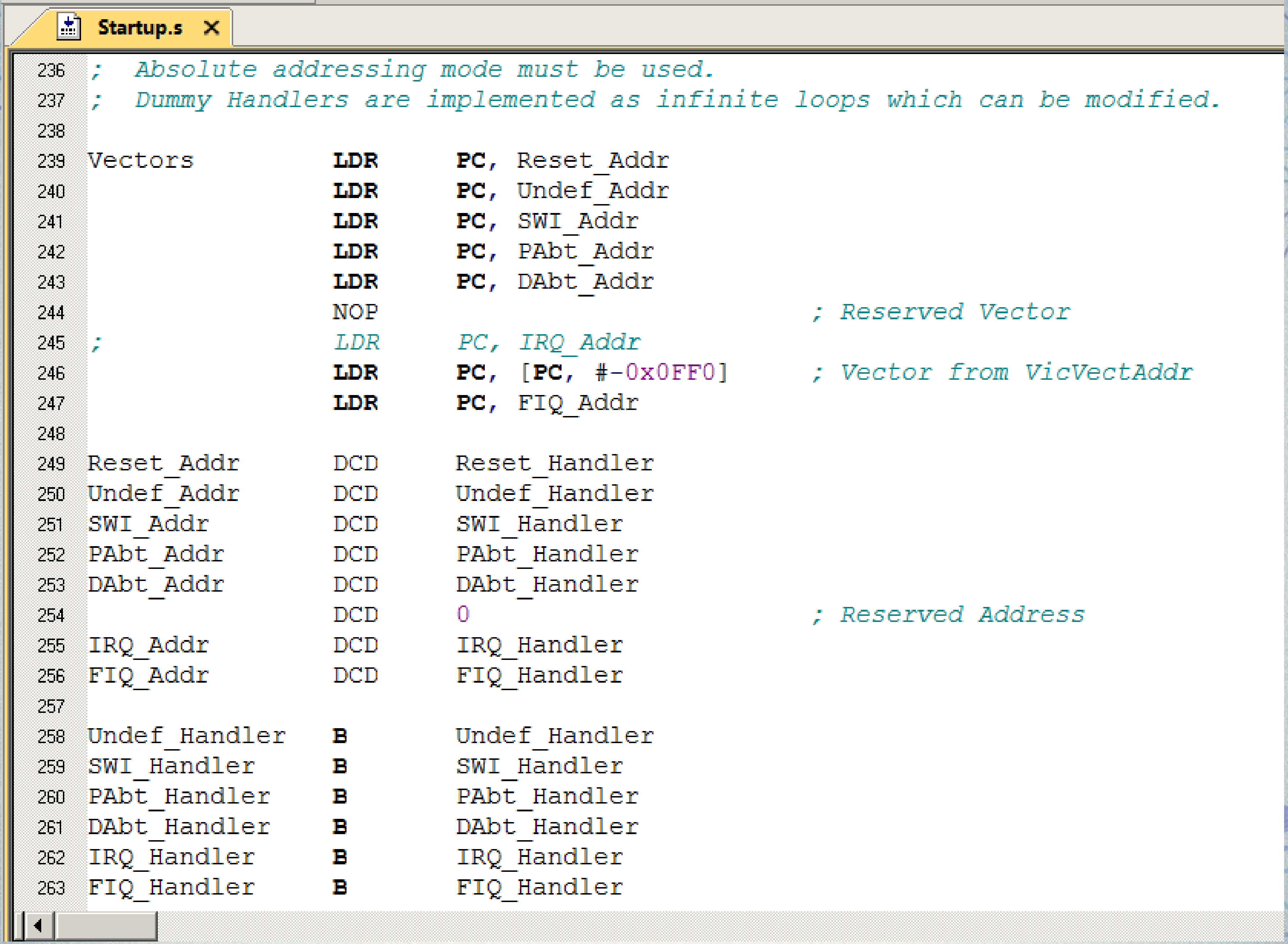
Startup.s

```
                AREA    RESET, CODE, READONLY
                ARM


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

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP                            ; Reserved Vector
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0FF0]     ; Vector from VicVectAddr
                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                      ; Reserved Address
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler
```


Startup.s

```
;   Absolute addressing mode must be used.
;   Dummy Handlers are implemented as infinite loops which can be modified.

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP                                 ; Reserved Vector
;               LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0FF0]          ; Vector from VicVectAddr
                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                           ; Reserved Address
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
```

Summary

- Exception type
- Vector Table
- Priority, mode and banked registers
- Exception handlers
- Vectored Interrupt Controllers
- SWI
- Startup codes