Exception Handling

The Vector Table

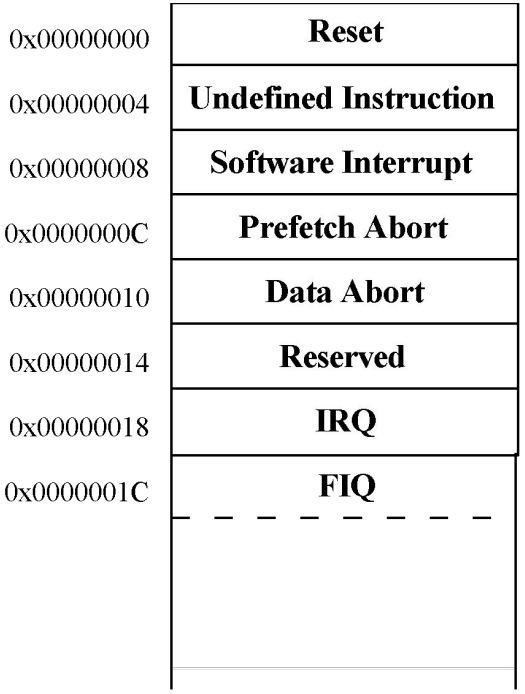
* **Reserved area at bottom of memory map.**

* **One word allocated to each exception type.**

* **Typically contains branch instruction to exception handler.**
  - This handler may itself be only a top-level handler function which then calls the appropriate routine to deal with the cause of exception

* **Provides flexible management of exceptions.**

Layout of the Vector Table



Exception Priorities

* **Several exceptions can occur at once.**
* **Thus exceptions are assigned priorities and are serviced in a fixed order:**
  - Reset
  - Data Abort
  - FIQ
  - IRQ
  - Prefetch Abort
  - SWI
  - Undefined instruction

When an exception occurs ...

* **The ARM:**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - mode field bits - exceptions accompanied by a mode change.
    - Interrupt disable flags if appropriate.
  - Maps in appropriate banked registers
    - No actual data movement so done in “zero time”.
  - Stores the “*return address*” (PC-4) in LR_<mode>
  - Sets PC to vector address
    - Forcing branch to exception handler

To return to previous mode (and main program) ...

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

“Return Address” SWI and Undefined Instruction

* **Exception handler is called, in effect, by the instruction itself. Thus PC not updated at point that LR value calculated.**

* **Storing (PC - 4) in LR therefore means that this actually points to next instruction to be executed.**

* **Thus to continue from that next instruction, to return from handler use:**
  - MOVS pc,lr

“Return Address” FIQ and IRQ

* **Exception handler called after instruction has finished executing. Thus PC updated before LR value calculated.**

* **Storing (PC - 4) in LR therefore means that this actually points two instructions beyond where exception occured.**

* **Thus to continue from next instruction, return from handler using:**
  - SUBS pc,lr,#4

“Return Address” Prefetch Abort

* **Exception taken when instruction reaches execute stage of pipeline. Thus PC not updated at this point.**

* **Storing (PC - 4) in LR therefore means that this actually points to next instruction after one that caused the abort.**

* **Thus to retry executing from aborted instruction, return from handler using:**
  - SUBS pc,lr,#4

“Return Address” Data Abort

* **Exception taken after instruction has updated PC. Thus value stored in LR actually points two instructions beyond where exception occured.**

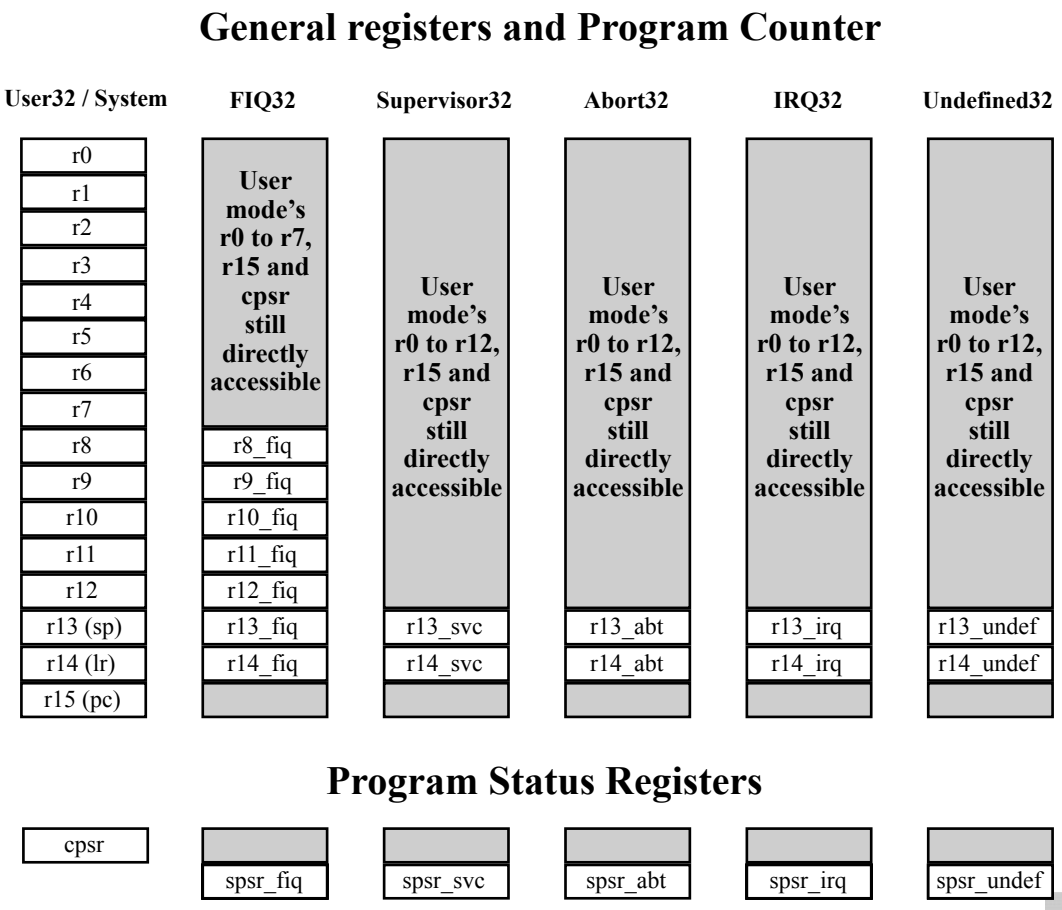
* **Storing (PC - 4) in LR therfore means that this actually points two instructions after one that caused the abort.**

* **Thus to retry execution of aborted instruction, the handler should return using:**
  - SUBS pc,lr,#8

Loading the Vector Table

* **The branch instruction required to reach the appropriate exception handler can be constructed as follows:**
  - Take the address of the exception handler.
  - Subtract the address of the corresponding vector.
  - Subtract 0x8 to allow for pipeline.
  - Shift result right by 2 to give word offset rather than byte offset.
  - Test that top 8 bits of this are clear, thus ensuring result is only 24 bits long (as offset for branch is limited to this).
  - Logically OR this with 0xea000000 (branch instruction) to produce the value to be placed in the vector.

Register Organisation



Register Usage in Exception Handlers

* **The mode change associated with an exception occuring means that as a minimum, the particular exception handler called will have access to:**
  - its own stack pointer (SP_<mode>).
  - its own link register (LR_<mode>).
  - its own saved program status register (SPSR_<mode>).
  - and, in the case of a FIQ handler, 5 other general purpose registers (r8_fiq to r12_fiq).
* **It is necessary for the exception handler to ensure that other registers are restored to their original state upon exit.**
* **This can be done by storing the contents of any registers it needs to use on its stack, and restoring them before returning.**

Setting up Stacks and Registers

```
MRS     r0, cpsr                ; get PSR
BIC r0, r0, #mode_flags     ; #0x1f for all flags
ORR     r1, r0, #mode_FIQ       ; set bits for FIQ mode
MSR     cpsr, r1                ; force new mode
LDR     sp, stack_top_FIQ       ; set sp to top of FIQ stack
LDR     r12, stack_bottom_FIQ   ; FIQ has its own r12 to be set
ADD     r12, r12, #stack_headroom       ; leave room for overflow
    check
;
; Initialise r8-r11 as required
;
ORR     r1, r0, #mode_super     ; reset flags
MSR     cpsr, r1                ; restore mode
```

Setting up Stacks and Registers (cont'd)

Where

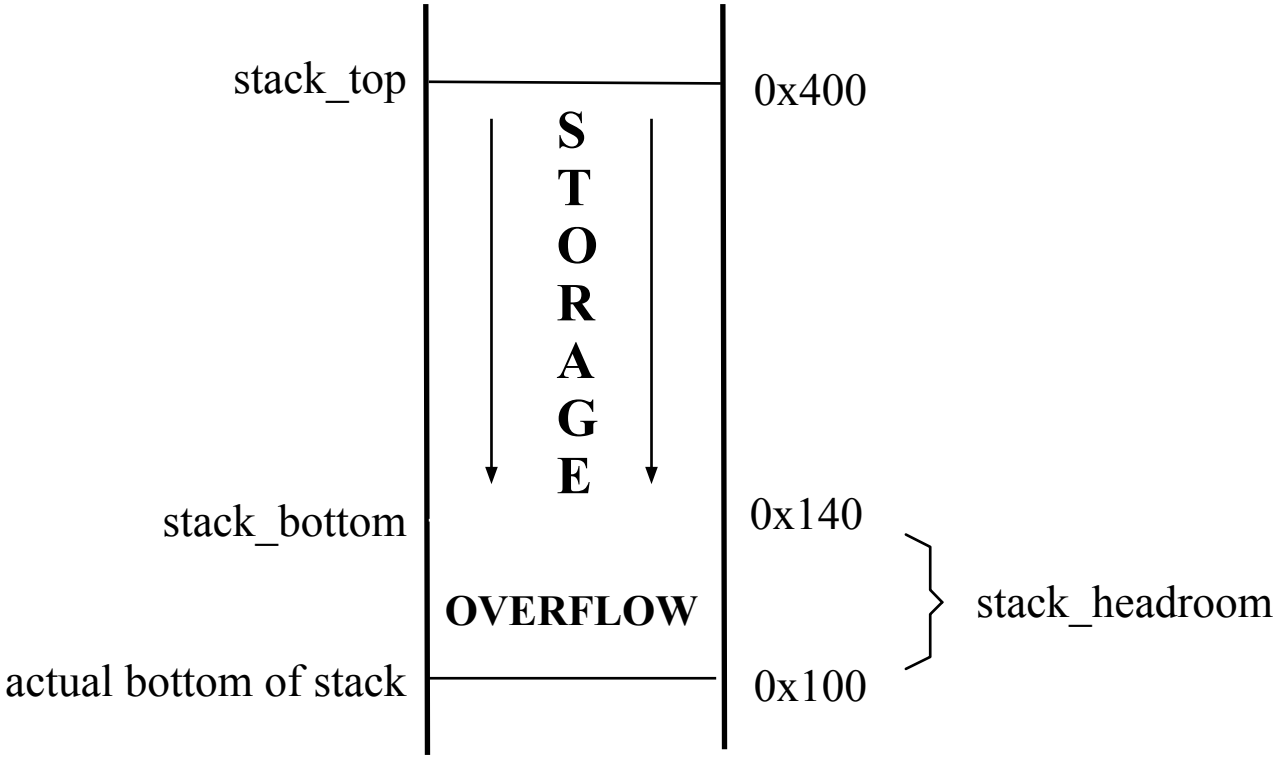
```
mode_flags          EQU     0x1f; for example
mode_FIQ            EQU     0x11
mode_super          EQU     0x13
stack_headroom      EQU     0x40      ; ie 16 words
```

and

```
stack_top_FIQ       DCD     0x400     ; for example
stack_bottom_FIQDCD     0x100     ;
```

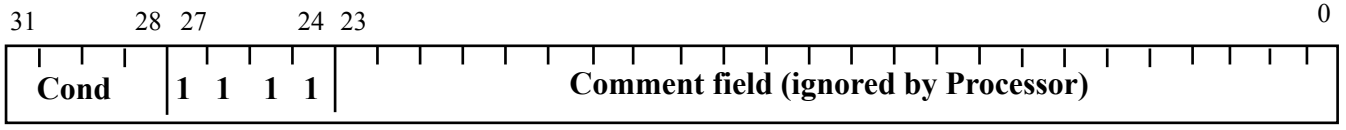
* **This should be done similarly for the other modes, allowing for the fact that they have fewer banked registers to deal with.**
* **The lack of a banked r12 in these can be dealt with by storing the stack bottom in a user-defined memory location.**

A Stack


stack_top
0x400
STORAGE
stack_bottom
0x140
OVERFLOW
stack_headroom
actual bottom of stack
0x100

SWI Handler

* Upon reaching the handler through the branch from the vector table, the handler needs to decide what SWI is being called.
* This information is stored in the SWI instruction itself.



* Handler thus needs to load and interpret instruction as appropriate.

* Note that SWI causes supervisor mode to be entered.
* This means care needed when calling a SWI when already in supervisor mode as LR_svc will be corrupted.

Examining the SWI Instruction

* **PC-4 was stored in LR_svc upon the SWI exception.**
  - Thus LR_svc actually points 1 instruction beyond SWI because of pipeline.
* **Therefore to load the SWI instruction into a register, use:**
  - LDR r0, [lr,#-4]
* **The comment field can then be extracted by:**
  - BIC r0, r0, #0xff000000
* **The resulting value can then be used in, say, a lookup table, to branch to the routine implementing that particular SWI.**
* **Note that because of the need to access the link register and then load in the actual SWI instruction within the handler, it is not possible to write the top-level SWI handler in C. Assembler will therefore be normally used, for the top-level at least.**

Example SWI Handler (1)

```
top_level_SWI
        SUB     sp, sp, #4          ; Leave room to store spsr
        STMFD   sp!, {r0-r12,lr}    ; Store registers
        MOV     r1, sp              ; Set up pointer to stack
        LDR     r0, [lr,#-4]        ; Load instruction
        BIC     r0, r0, #0xff000000 ; Extract comment field
        MRS     r2, spsr            ; Get spsr
        STR     r2, [sp,#14*4]; Store spsr onto stack
        BL      _SWI_handler        ; Call C handler
```

Example SWI Handler (2)

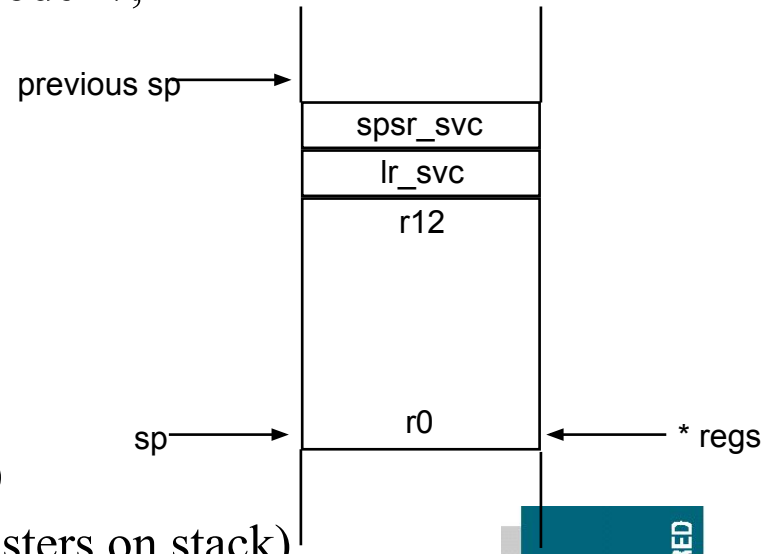
* **Once the C routine has handled the SWI, it returns back to the top level handler, which continues by unstacking the stored status:**

```
LDR     r2, [sp, #14*4]     ; restore SPSR
MSR     spsr, r2
LDMFD   sp!, {r0-r12,lr}    ; unstack register
ADD     sp, sp, #4          ; remove space used for SPSR
MOVS    pc, lr              ; return from handler
```

Example SWI Handler (3)

* **The previous assembler extracts the SWI number and then calls a C subroutine that deals with the actual call:**

```
void SWI_handler (int number, int *regs) {
        switch (number) {
        case 0: /*SWI number 0 code */;
                break;
            :
            :
        }
}
```



* **APCS means that**
  - number = r0 (ie actual SWI)
  - *regs = r1 (ie pointer to registers on stack)

Interrupt Handling

* **The ARM has two levels of external interrupt - FIQ and IRQ.**
* **FIQs have a higher priority than IRQs in two ways:**
  - Serviced first when multiple interrupts arise.
  - Servicing a FIQ in turn disables IRQs thus preventing any IRQs from being serviced until after the FIQ handler has re-enabled them.
* **It is possible to set up C functions as interrupt routines by using the special function declaration keyword '__*irq*'.**
* **This causes**
  - all registers (excluding the floating point ones) to be preserved (not just those normally preserved under the APCS).
  - the function to be exited by setting the pc to lr-4 and restoring the cpsr to its original value.

Example Interrupt Handler in C

* **The example routine below reads a byte from location 0xc0000000 and writes it to location 0xc0000004.**

```
void __irq IRQHandler (void)
{
        volatile char *base = (char *) 0xc0000000;
        *(base+4) = *base;
}
```

Installing this Handler

* **This IRQHandler function can be installed by the main program:**

```
int main()
{
        unsigned *irqvec = (unsigned *)0x18;
        *irqvec = Install_Handler ((unsigned)IRQHandler,
                                                (unsigned)irqvec);
        return 0;
}
```

Install_Handler Routine

* **This implements the method described for loading the vector table discussed earlier in this presentation.**
* **It returns the encoding that represents the branch instruction that should be placed in the appropriate vector.**

```
unsigned Install_Handler (unsigned routine, unsigned vector) {
/* To suceed‘routine’ must be within 32Mb of ‘vector’ */
        unsigned vec;
        vec =((routine - vector - 0x8) >>2)
        if (vec & 0xff000000) flag_error(“Out of range handler”);
        vec = 0xea000000 | vec
        return vec;
}
```

Reset Handlers

* **Upon reset:**
  - Self test (hardware)
  - Locate memory (see how much available)
  - Initialise stacks and registers (as seen earlier)
  - Initialise peripherial hardware
    - eg/ clear i/o ports.
  - Initialise MMU if used (ARM x00,x10)
  - Call main routine (__main)

Undefined Instructions (1)

* **Any instructions that are not recognised by an ARM are first of all offered to any coprocessors that are attached to the system.**
* **If, after this, the instruction is still unrecognised, then the undefined instruction trap is taken.**
* **It is still possible that the instruction is one for a coprocessor**
  - but relevant coprocessor is not attached to the system (eg FPA)

  and emulator for it might be in system software (eg FPE).
* **Such an emulator can attach itself to the undefined instruction trap:**
  - Intercept vector, storing old vector.
  - Examine instruction to see if it should emulate it.
  - If can do emulation then process it, then return to user program.
  - Else go to original handler
    - Report error and quit.

Undefined Instructions (2)

* **An emulator can examine the instruction in a similar way to that used by the SWI handler to find out the number, but rather than extracting the bottom 24 bits, instead it should extract bits 24 to 27 which determine if the instruction is a coprocessor operation.**
* **If bits 27-24 = 1110 or 110x then**
  - The instruction is a coprocessor instruction
  - So extract bits 8-11 which define which coprocessor should deal with instruction.
  - If these show that this emulator should handle instruction:
    - Process it.
    - Return using "MOVS pc,lr' to user program.
  - Else use stored vector to proceed to original handler.
    - This might be another emulator, forming a chain of handlers.

Undefined Instructions (3)

* **Once the chain of emulator handlers is exhausted, then no further processing can take place, so final handler should:**
  - Report error and quit.
    - Exact process is system dependent.

Prefetch and Data Abort Handlers

* **In simple case, where there is no MMU then:**
  - Report error and quit

Prefetch Handler with MMU

* **The instruction that caused the exception is at lr-4.**
  - On exception, lr = pc - 4.
  - ie LR points to instruction following one which caused abort, since PC not updated before exception taken.
* **Thus offending address is at lr-4.**
* **Thus can deal with virtual memory fault for lr-4.**
* **Then jump back to that address**
* **Note that when leaving exception handler, need to return to actual instruction where exception occured so that can fetch it again. Thus return using:**
  - SUBS pc, lr, #4

Data Abort with MMU (1)

* **Instruction which caused abort is at lr-8.**
  - On exception, lr = pc - 4.
  - ie LR points to instruction two after the one that caused abort, since PC was updated before exception taken.
* **Three possible cases of instruction:**
  - If instruction is a single register load or store
    - In early abort (ARM6 only)

      Do not need to worry about address register update.
    - In late abort

      With writeback, address register will have been updated.

      Undo change to address register

Data Abort with MMU (2)

- If instruction is a SWAP
  - Do not need to worry about address register update
- If instruction is multiple load or store:
  - If writeback enabled

    Base register is updated as if whole transfer took place, but NOT overwritten when register in list for a load.

    Therefore, count number of registers involved and add / subtract from base, to restore original.

* **In each case the address which caused the abort is stored in the MMU's Fault Address Register.**
* **Load Memory Page as required.**
* **Return back to instruction that caused abort, so can re-execute it:**
  - SUBS pc, lr, #8

Locating the FIQ Handler at 0x1c

* **FIQ vector placed last in vector table.**
* **Allows handler to be run sequentially from that address**
  - removes need for branch and its associated delays.
  - cache locality
* **Important because speed essential for FIQ.**
* **FIQ handler can be placed at 0x1c by copying it there.**
  - memcpy (0x1c, FIQ_Start, FIQ_End - FIQ_Start);
* **ARM code is inherently relocatable, but note that:**
  - Code should not use any absolute addresses.
  - PC relative addresses are allowable as long as the data is copied as well (so remains in same place relative to the relocated code).
  - 5 FIQ registers mean that status can be held between calls.

Example FIQ Handler: Single Channel DMA Transfer

```
LDR     r11, [r8, #IOData] ; load port data
STR     r11, [r9], #4      ; store to memory
CMP     r9, r10            ; reached the end?
SUBNES  pc, lr, #4         ; return
; Insert transfer complete code here
```

* **Locate this code at location 0x1c. The code executes in 9 clock cycles from zero wait state memory (LDR 3 cycles, STR 2 cycles, CMP 1 cycle, SUBS (return) 3 cycles)**
* **R8 points the IO device, IOData is the offset to a 32 bit wide data register (use LDRB for 8 bit transfers)**
* **R9 is the pointer to the memory buffer that the data is being transferred to**
* **R10 marks the end of the transfer buffer**
* **R11 is a temporary**

Example FIQ Handler: Dual Channel DMA Transfer

```
LDR     r13, [r8, #IOStat] ; load port data
TST     r13, #IOPort1Active; check which port
LDREQ   r13, [r8, #IOPort1]; load port 1 data
LDRNE   r13, [r8, #IOPort2]; load port 2 data
STREQ   r13, [r9], #4      ; store to buffer 1
STRNE   r13, [r10], #4     ; store to buffer 2
CMP     r9, r11          ; reached the end ?
CMPNE   r10, r12         ; on either port?
SUBNES  pc, lr, #4       ; return
; Insert transfer complete code here
```

* **Locate this code at location 0x1c. The code executes in 16 clock cycles from zero wait state memory (for either channel). Inserting a branch after the test can reduce this to 14 for one channel, 16 for the other.**

Example FIQ Handler: Nested, Prioritised Interrupts (1)

```
; first save the critical state
SUB     lr, lr, #4              ; adjust the return address...
                                ; ... before we save it.
STMFD   r13!, {lr}              ; stack return address
MRS     r14, SPSR               ; get the SPSR ...
STMFD   r13!, {r12,r14}         ; ... and stack that plus a...
                                ; ... working register too.

; now get the priority level of the highest priority active interrupt
MOV     r12, #IntBase           ; get the interrupt controller's ...
                                ; ... base address
LDR     r12, [r12, #IntLevel]   ; get the interrupt level (0 to 31)

; now read-modify-write the CPSR to re-enable interrupts
MRS     r14, CPSR               ; read the status register
BIC     r14, r14, #0x40         ; clear the F bit (use 0x80 for the I bit)
MSR     CPSR, r14               ; write it back to re-enable interrupts

; jump to the correct handler
LDR     PC, [PC, r12, LSL #2]   ; and jump to the correct handler...
                                ; ... PC base address points to this...
                                ; ... instruction + 8
NOP                             ; pad so the PC indexes this table

; table of handler start addresses
        DCD     Priority0Handler
        DCD     Priority1Handler
        ........
```

Example FIQ Handler: Nested, Prioritised Interrupts (2)

```
Priority0Handler
    STMFD   r13!, {r0 - r11}     ; save other working registers

    ;........
    ; insert handler code here
    ;........

    LDMFD   r13!, {r0 - r11}     ; restore working registers (not r12).

    ; now read-modify-write the CPSR to disable interrupts again
    MRS     r12, CPSR            ; read the status register
    ORR     r12, r12, #0x40      ; set the F bit (use 0x80 for the I bit)
    MSR     CPSR, r12            ; write it back to disable interrupts

    ; now that interrupt disabled, can safely restore SPSR then return
    LDM     r13!, {r12, r14}     ; restore r12 and get SPSR
    MSR     SPSR, r14            ; restore status register from r14
    LDMFD   r13!, {PC}^          ; return from handler

Priority1Handler
    ........
```

- Locate this code at location 0x1c.
- Code takes 14 clock cycles to re-enable interrupts. Interrupts then re-disabled for a further 8 cycles at the end of the handler.

Example FIQ Handler: Context Switch

```
STMIA    r13, {r0 - lr}^   ; dump user registers above r13
MSR r0, SPSR      ; pick up the user status
STMDB    r13, {r0, lr}     ; dump it with return address below r13
LDR r13, [r12], #4    ; load next process info pointer
CMP r13, #0       ; if it's zero, it's invalid
LDMNEDB  r13, {r0, lr}     ; pick up status and return address
MRSNE    SPSR, r0      ; restore the status
LDMNEIA  r13, {r0 - lr}^   ; get the rest of the registers
MOVNES   pc, lr        ; and return + restore CPSR
; insert "no next process code" here
```

* Locate this code at location 0x1c. The code executes in 49 clock cycles from zero wait state memory.
* R12 points to a list of pointers to Process Control Blocks. A zero pointer indicates no free process.

Dealing with long distance branches

* **In almost all circumstances the 32 Mb range of the branch instruction will be sufficient to reach the appropriate handler from the vector table.**
* **However this will occasionally not be the case, eg:**
  - A system might have its memory map set up with the ROM containing the code (and hence the handler) at the top end of memory. If the code is run directly from the ROM (rather than copied down into RAM first) then this distance from the vector table could be too great for the branch instruction to encode.
* **This can be got around by directly forcing the PC to the address of the appropriate handler**
  - Store the address of the handler in a suitable memory location.
  - Place in vector encoding of instruction to load PC with contents of the memory location.

Long Distance Example

* **Thus the initialisation of the IRQ vector for the previously mentioned IRQHandler might become:**

```
int main ()
{
        unsigned *irqvec = (unsigned *) 0x18;
        unsigned *irqaddr = (unsigned *) 0x38;
        *irqvec = 0xe59ff018;              /* LDR pc, [pc,#24] */
        *irqaddr = (unsigned)IRQHandler;
        return 0;
}
```