SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Exceptions

Ensc254

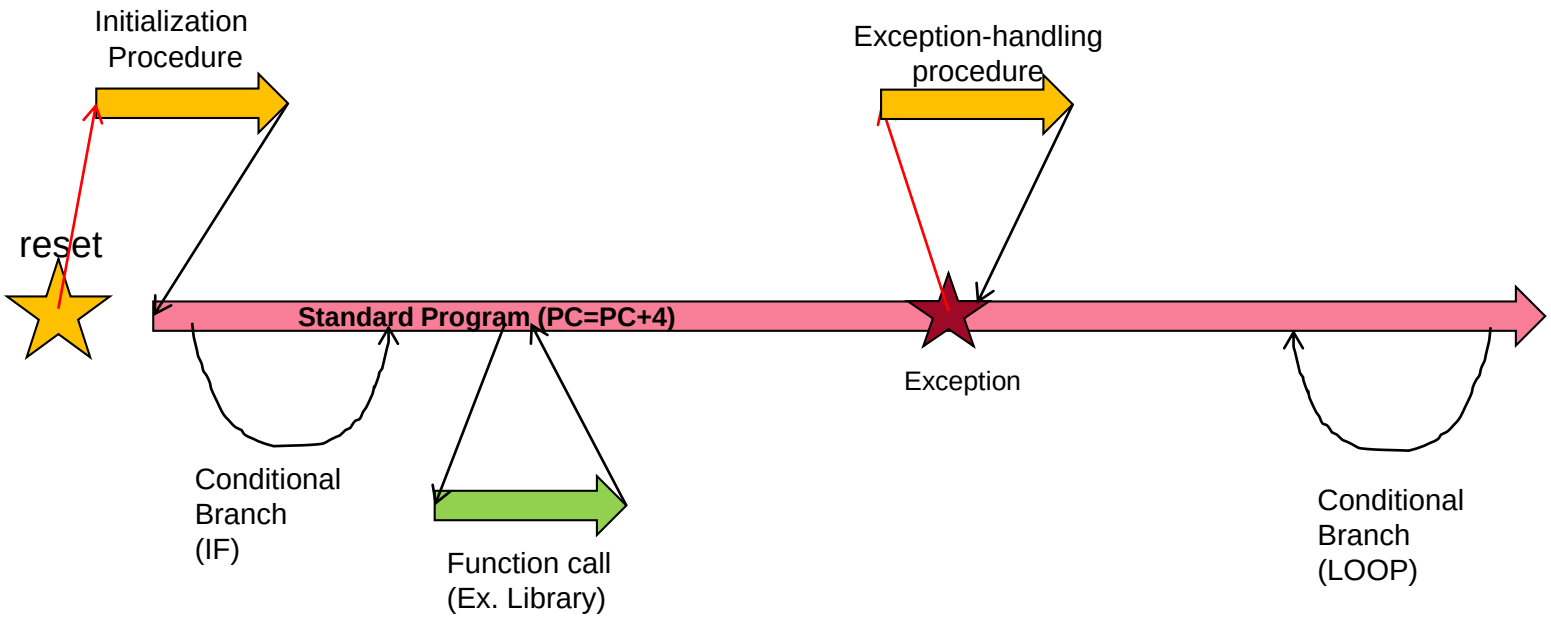
© 2021 -- Fabio Campi and Craig Scratchley
School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Notes for Lab (1)

- We are using the ZedBoard, which is a PWB-based (PCB-based) computer system (mainly used for educational purposes) based on the Xilinx Zynq Chip
- Zynq is a System-on-a-Chip (SoC) that includes an FPGA and a high-performance Dual-Core Cortex A9 ARM processing module (@667MHZ max)

FPGA = Field Programmable Gate Array, is a hardware device that can be programmed (better said CONFIGURED) to provide many different digital hardware functionalities (You could even configure another ARM or your own processor on the FPGA, and that is often done, but we will use one of the “Real”, HARDWIRED processing elements – not one configured on the FPGA.)

Exception



- The normal flows of a program are somewhat predictable, and much of its evolution is encoded in the instruction memory before it starts. However, exceptions can allow for behaviour unpredictable to the program.

Many EXCEPTIONS are ASYNCHRONOUS, unpredicted alterations to the program flow

Interrupts

- We have often stressed how

THE ONLY WAY FOR A PROCESSOR TO COMMUNICATE WITH THE EXTERNAL WORLD IS BY MEANS OF LOAD / STORE OPERATIONS

A peripheral is a hardware block in a system that performs a given task and needs to communicate with the processor. We can have

- COMPUTATION peripherals (i.e. HW Divider, Graphic Accelerator, ….)
- COMMUNICATION peripherals (ADC, UART, I2C “Inter-Integrated Circuit”, USB possibly for Mouse and keyboard, …)

ALL PERIPHERALS TO ARM processors are memory-mapped

Interrupts (2) : Asynchronous communication

- Very often peripherals have timescales that are much longer than that of the processor, and the processor should not be stalled waiting for such peripherals.
- An ordinary program should not necessarily be expected to need to service a given peripheral
    - e.g. When using a mouse or a keyboard, the processor does not know WHEN we will click/type (Humans are very slow peripherals…)

- There are two strategies for handling communication with a peripheral
    1. POLLING: A program regularly checks the peripheral i/o registers to see if there is need for servicing the peripheral.

    2. INTERRUPT: A “normal” program gets INTERRUPTED by the peripheral requiring service. In this case, the processor executes an interrupt SERVICE routine and then returns to resume what was interrupted.

Interrupts

A hardware interrupt is essentially a SUBROUTINE that is NOT called as part of the code, but is *triggered* by an external event

A hardware interrupt is an efficient way of telling the processor that something (usually a peripheral) needs attention

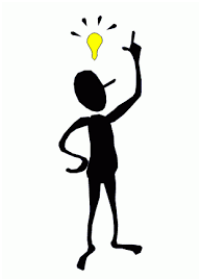
Exception

- In the course of the program execution there may be events IN THE PROCESSOR that may require special servicing.
- In these cases, an *event in the processor itself is interrupting the program flow*. These are other forms of EXCEPTIONS

- Typical Exceptions:
  1. Hardware Interrupts
  2. Division by zero (if we had a divider on chip, we usually don't)
  3. Undefined opcode
  4. Illegal memory access (Data or Instruction Memory in Harvard processors)
  5. SuperVisor Calls / SoftWare Interrupts (SVC/SWI)

Silly Example

- Programming a CPU without an OS is a bit like dating the CPU:
  - It's just YOU TWO, nothing else in between
  - It's complicated, confusing, at time frustrating
  - You can make BIG mistakes, and one small insignificant detail can break down the whole relationship

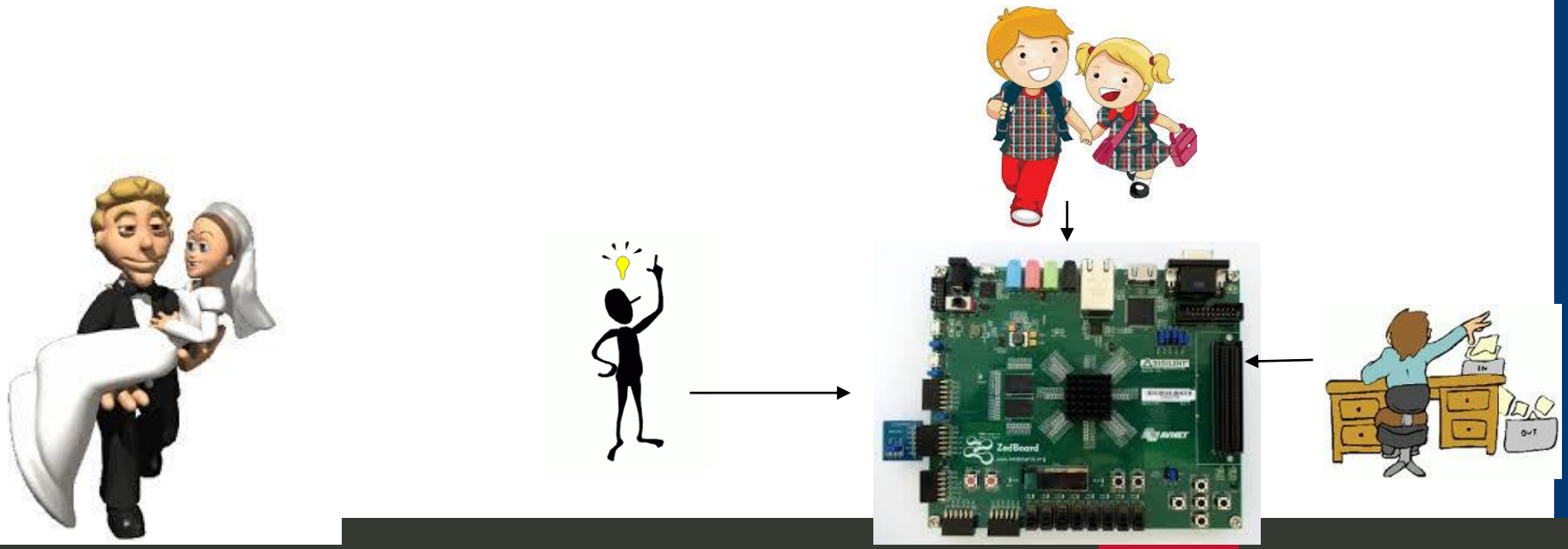


Silly Example

Using an OS is a bit like being married to your CPU:

- Yes, you are together, but now you barely recognize your CPU any more
- Multiple concurrent other applications (Kids, Work, your in-laws are now accessing your CPU, and you find your time with your CPU strictly scheduled)

- When you need some specific services you have to issue a (software) interrupt request and hope for a timely acknowledgment that never seems to happen

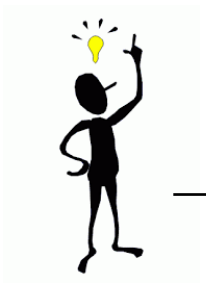


How to interface a program to the operating system

- You can enjoy specific SERVICES offered by an OS
- A process is a running application controlled by an Operating System. How can a process communicate with the OS?
  - The primary way for a process to get out of its own program flow and ask for “SERVICE” is to make a Supervisor Call (SVC/SWI).
  - Creating such an exception forces the processor to step out of the process and branch to the OS code

How to interface a program to the operating systems

- This exception is called a SUPERVISOR CALL or SOFTWARE INTERRUPT, and represents a request for services from the program to the OS
- Another case when the OS may get the processor from an application is when the time has come to offer the processor to a different application (Concurrent computing). In this case, a TIMER will signal that a timeslice on the processor is over and the processor will be given to a different application.



Processor Exception Sequence (IMPORTANT!)

- When an exception occurs, ARM has a definite sequence of events to handle the issue:

1. The current instruction is usually allowed to complete

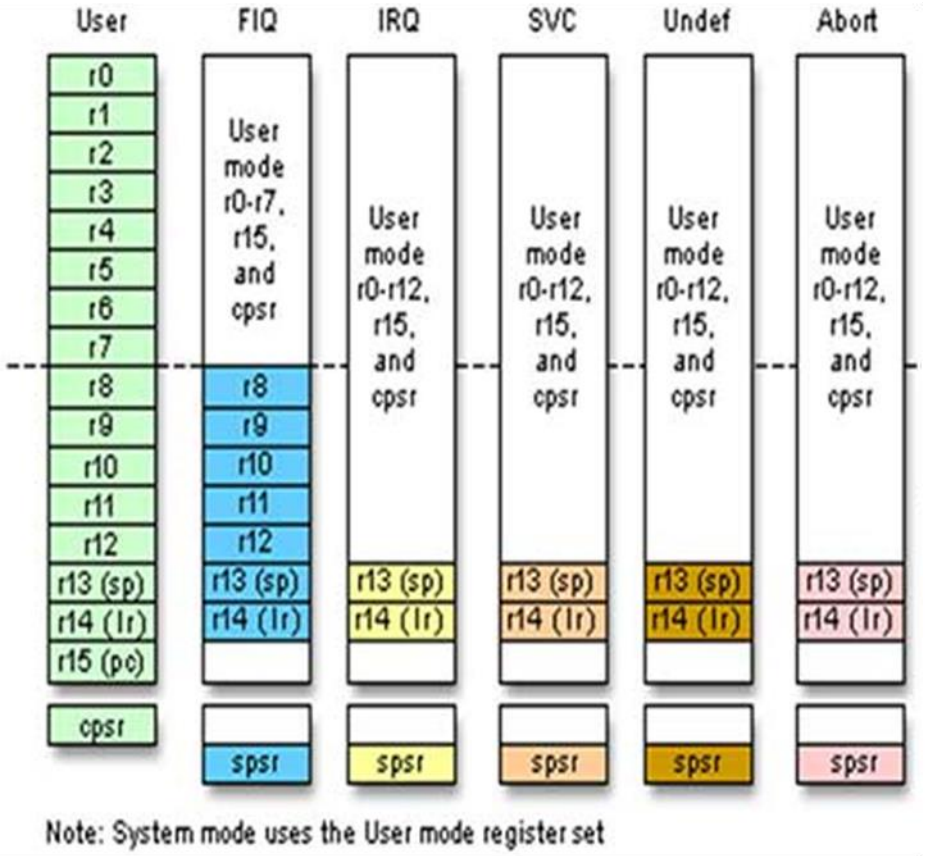
2. Current Program Status Register (CPSR) is copied into the **SAVED** program status register (SPSR) of the <mode> the processor is moving into (see ARM Register File following)

3. CPSR flags are set:
   a. ARM mode will be restored if we are in 16-bit THUMB mode
   b. The appropriate Processing mode will be set
      - ✓ FIQ (Fast Interrupt) mode OR
      - ✓ IRQ (Interrupt) mode OR
      - ✓ Other exception-specific mode (Undefined, Data Abort, Supervisor)

4. Exception Return address is stored into LR_<mode> (See ahead)

5. The PC is moved to the specific address of an element in the Vector Table (See ahead)

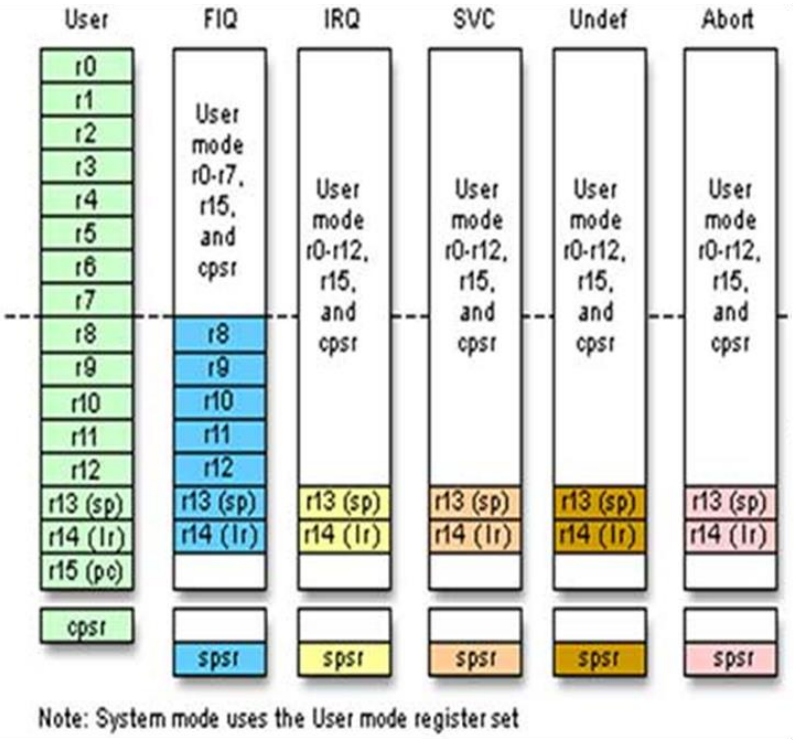
The ARM Register File



- Registers r0-r7, PC are shared between all modes
- FIQ mode has a dedicated r8-r14 set
- Most modes have dedicated r13,r14 (SP and LR)
- User and System modes share all registers

- There is only one CPSR, but each mode except User/System has a dedicated SPSR

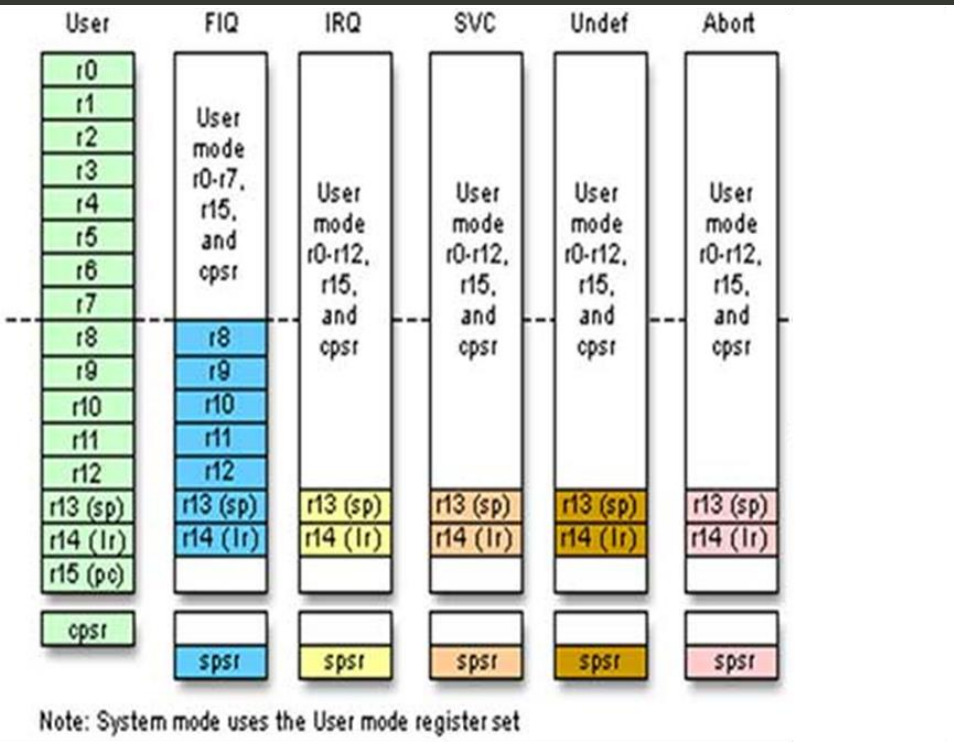
Quiz: How many registers are physically inside ARM



- Registers r0-r7, PC are shared between all modes
- FIQ mode has a dedicated r8-r14 set
- Most modes have dedicated r13,r14 (SP and LR)
- User and System modes share all registers
- There is only one CPSR, but each mode except User/System has a dedicated SPSR

a) 16
b) 17
c) 31
d) 37
e) 42

Quiz: How many registers are physically inside ARM



- Registers r0-r7, PC are shared between all modes
- FIQ mode has a dedicated r8-r14 set
- All modes have dedicated r13,r14 (SP and LR)
- There is only one CPSR, but each mode except User has a dedicated SPSR

a) 16
b) 17
c) 31
d) 37
e) 42

#Registers = 16 + 7 + 2x4 + 1 + 5

Exception Handler (IMPORTANT!)

- After the status info has been set the processor will execute the ***Exception handler***, that will have the following structure

1. In general, shared registers that will be used in the handler should be saved on the stack. Note that, in addition to its own link register, each exception mode has its own stack pointer. In addition, the FIQ mode has it's own instances of r8-r12. NOTE: unlike subroutine calls, we may need to save r0-r3 and, except for FIQ mode, r12 also, because here the interrupted code will NOT KNOW that it is being interrupted and may well be using those registers (the interrupt will be known for SVC/SWI, but it is still convenient to allow those registers to be used like normal)! If we have full control of the handler, then we can avoid those registers.

2. <The interrupt/exception is serviced according to the System Programmers' instructions>

3. Saved registers are restored, probably from a mode-specific stack.

4. The PC is restored to the value it had prior to the exception, and the SPSR is copied back on the CPSR (this will restore the processor to the mode at the time of the exception). It is not uncommon for this all to be part of step 3.

Quiz

When an Interrupt/Exception Handler gets invoked, something similar to a subroutine call is made.

Remember, that according to the ARM APCS

- r0-r3 contain input arguments and can be changed
- r4-r11 needs to be saved on the stack before changing them
- r12 is a scratch register and is typically not saved on the stack

- Do you think this is applicable to Interrupts?

Quiz

Remember, that according to the ARM APCS

- r0-r3 contain input arguments and can be changed
- r4-r11 needs to be saved on the stack before changing them
- r12 is a scratch register and is typically not saved on the stack

- Do you think this is applicable to Interrupts?

It is not generally applicable: in the case of Hardware Interrupts the processor is not following a program, the compiler did not "plan" the interrupt. SO SOME "NORMAL" REGISTERS MUST BE SAVED (A SMALLER SET FOR FIQ)!

Moreover, a program does not generally pass arguments to hardware interrupt handlers.

Interrupt Handler (Important)

When an Interrupt/Exception Handler gets invoked, something similar to a subroutine call is made. There are 2 fundamental differences:

1. The processor mode gets set (from the current mode to the specific exception mode)

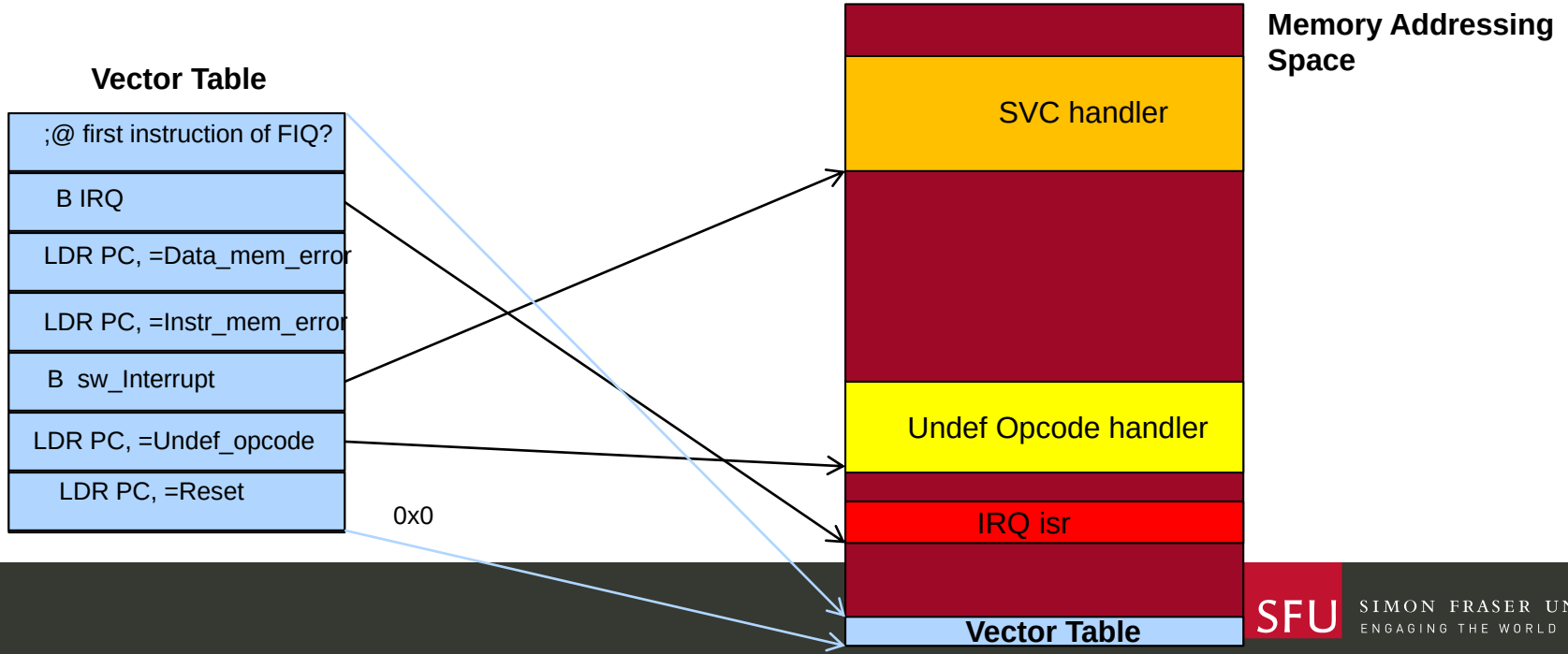
2. We save registers differently: in case of Interrupts/Exceptions the processor is not always following a program, the compiler did not “plan” a function call and can not make sure that r0-r3 (and maybe r12) are not needed after the interrupt. SO in the case of Interrupt/Exception Handlers AT LEAST SOME USED “NORMAL” REGISTERS MUST BE SAVED in some way!

The Vector Table (IMPORTANT!)

- In the *event* of an exception, the processor must get to an ***Exception Handler***

- Instructions to get to most exception handlers are stored in a TABLE, called the ***VECTOR TABLE,*** located in the memory. Sometimes FIQ starts at the table entry.
- *The vector table will contain an instruction for each type of interrupt/exception.*
- Note that while most processors have ADDRESSES in the Vector Table, ARM actually has instructions.

  Note the below figure is “upside down” from our typical convention for memory.



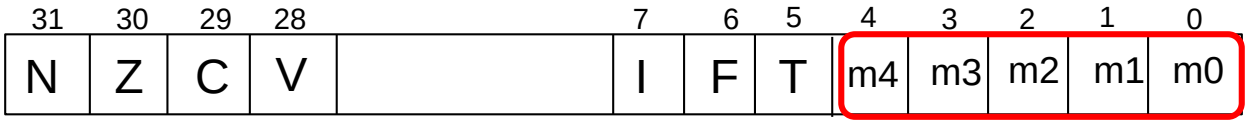
Vector Table assembly implementation

```
        ;@ one possibility …

Vector_Table
        LDR     PC, =MyOwnReset
        B       MyOwnUndefined
        LDR     PC, =MyOwnSVC
        LDR     PC, =MyOwnInstrAbort
        LDR     PC, =MyOwnDataAbort
        NOP     ;@ reserved
        B       MyOwnIRQ
        ;@ starting instruction of FIQ

        ;@ …
        .LTORG
```

Processor Modes



- We discussed earlier that ARM has 7 processor modes, specified in the Program Status Register
  - The processor at RESET is in Supervisor mode. During Interrupts/Exceptions it will be moved to the specific mode of the Interrupt/Exception.
  - After servicing an Interrupt or an Exception the processor should go back to the previous mode, often USER mode with limited privileges. This is a NO RETURN operation.
    - Supervisor mode can change the mode flags (m4-m0) in the CPSR, *user mode can only read them*

1) **SVC (Supervisor)**
2) **FIQ (High Priority Interrupt Raised)**
3) **IRQ (Low priority Interrupt Raised)**
4) **Abort (Memory Access violation)**
5) **Undef (Undefined Instruction)**
6) **System**
7) **User (Normal mode for a program under an OS)**

How Can We change processor mode?

1) From Supervisor mode to User Mode
   a) With a MSR operation
   b) With a MRS operation
   c) With a SVC operation
   d) With a MOV operation
   e) With an interrupt

How Can We change processor mode?

1) One allowable way to change from Supervisor mode to User Mode
   a) *With a MSR operation*
   b) With a MRS operation
   c) With a SVC operation
   d) With a MOV operation
   e) With an interrupt

Supervisor mode has write access to all bits in the CPSR so Mode can be changed with the MSR operation

How Can We change processor mode?

1) From Supervisor mode to User Mode
   a) *With a MSR operation*
   b) With a MRS operation
   c) With a SVC operation
   d) With a MOV operation
   e) With an interrupt

2) From User Mode to Supervisor mode
   a) With a MSR operation
   b) With a MRS operation
   c) With a SVC operation
   d) With a MOV operation
   e) With an interrupt

How Can We change processor mode?

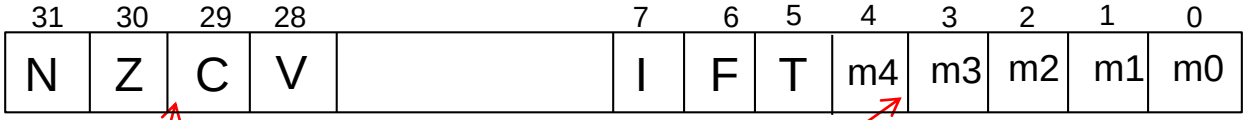
1) From Supervisor mode to User Mode

Supervisor mode has write access to all bits in the CPSR so Mode can be changed with the MSR operation

2) From User Mode to Supervisor mode
   a) With a MSR operation
   b) With a MRS operation
   c) With a SVC operation
   d) With a MOV operation
   e) With an interrupt

User mode has no ability to change mode in CPSR. The only way to switch mode is, then, to issue a Supervisor Call, so that before exception servicing mode will be switched to Supervisor mode

Changing processor mode

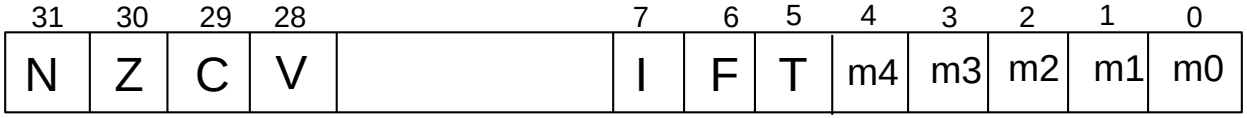


- Processor mode ***is read*** using the MRS operation
  - MRS Rd,CPSR moves the CPSR into the general purpose register Rd
- Processor mode ***can be changed*** using the MSR operation.
  - Please note that to avoid disruptive errors, full MSR is not available in User Mode, and there MSR CAN ONLY BE USED ON A DEFINED SUBSET OF THE CPSR:
    - MSR CPSR_f, operand (Immediate or register) only changes flags
    - MSR CPSR_c, operand only changes control bits (including processor mode)

I = IRQ Mask
F=FIQ Mask
T= Thumb

10000 User Mode
10001 FIQ Mode
10010 IRQ Mode
10011 Supervisor Mode
10111 Abort Mode
11011 Undefined Mode
11111 System Mode

Changing processor mode



If we want to independently change to User mode, we need to perform

```
MRS R0, CPSR
BIC  R0,0x1f
ORR R0,0x10
MSR CPSR_c, R0
```

CLEAR and OR

If we instead need to mask FIQ and IRQ

```
MRS R0, CPSR
ORR R0,0xc0
MSR  CPSR_c,  R0
```

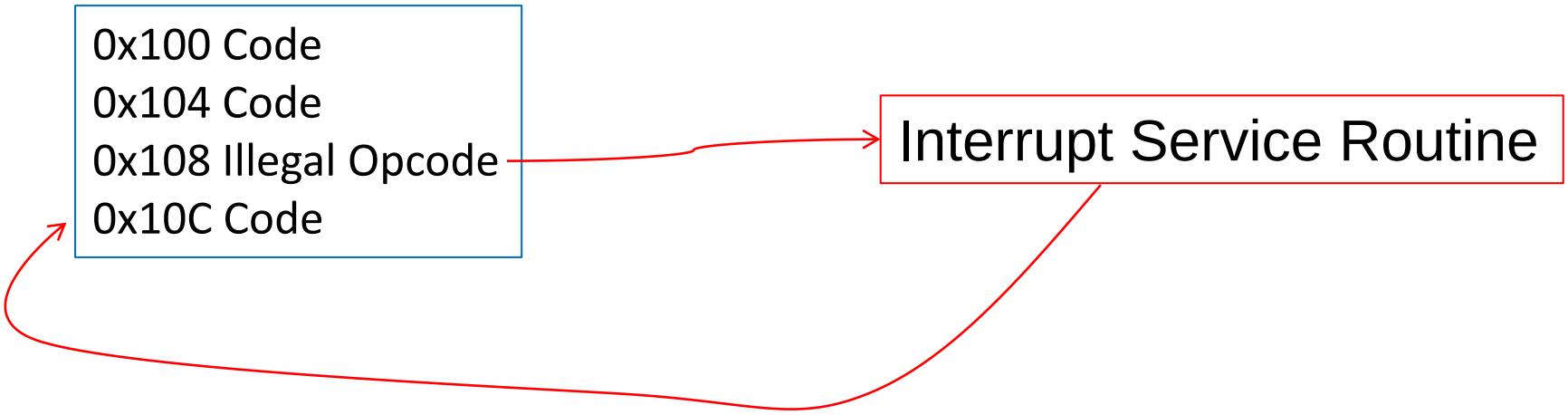
Here we want to set the bits to 1,
there is no point in clearing them first

I = IRQ Mask
F=FIQ Mask
T= Thumb

10000 User Mode
10001 FIQ Mode
10010 IRQ Mode
10011 Supervisor Mode
10111 Abort Mode
11011 Undefined Mode
11111 System Mode

Exception Return Address

- When an exception or an interrupt occurs, prior to jumping to the Vector table the processor will save the «INTERRUPT RETURN ADDRESS» to the link register of the appropriate mode
- ***Often the LINK REGISTER WILL CONTAIN THE ADDRESS OF THE EXCEPTING INSTRUCTION + 4***
- ***It is up to the programmer of the Interrupt Handler to decide where to Return:*** for example,
  - in the following case we may want to return to LR (0x10C) to avoid detecting a new illegal opcode exception and keep on iterating forever on a loop.
  - In different cases (e.g. prefetch abort) we may want to repeat the excepting instruction (LR - 4) after having serviced the exception



Code Template for an Interrupt / Exception Handler (Important!)

```
;@ adjust LR if desired
STMIA SP!,{r0-r12,LR}        ;@ Saving Workspace
MRS   r0,SPSR                ;@ Saving SPSR to stack …
STMIA SP!,{r0}               ;@  … in case of further (nested) interruptions
;@ enable nested interrupt
```

< Specific Servicing Code Here>

```
LDMDB sp!,{r0}               ;@ Retrieving SPSR from stack
MSR   SPSR_cxsf, r0          ;@ Restoring SPSR
LDMDB SP!,{r0-r12,PC}^       ;@ Retrieving Workspace, PC,
                             ;@ and loading SPSR into CPSR
```

Notes: 1) STMIA Stores LR, that is by default the address following the interrupting instruction, but LDMDB loads PC. In this way, the routine returns to the code where it was interrupted. (PC is R15, LR is R14)

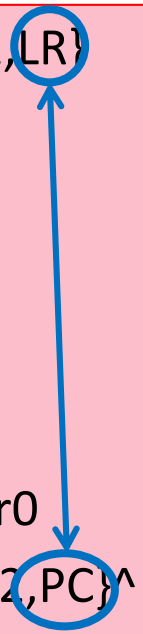
2) The ^ Sign in the LDMDB forces ARM to copy SPSR into CPSR, thus restoring previous mode at the end of the service routine

3) R13 is not saved/restored: R13,R14 are mode specific so are not overwritten by the actual exception. FIQ has other mode-specific registers.

Quiz: IMPORTANT

```
STMIA SP!,{r0-r12,LR}      ; Saving Workspace
MRS   r0,SPSR              ; Saving SPSR to stack …
STMIA SP!,{r0}             ;  … in case of further (nested) interruptions

LDMDB sp!,{r0}             ; Retrieving SPSR from stack
MSR   SPSR_cxsf, r0        ; Restoring SPSR
LDMDB SP!,{r0-r12,PC}^     ; Retrieving Workspace, PC,
                           ; and loading SPSR into CPSR
```



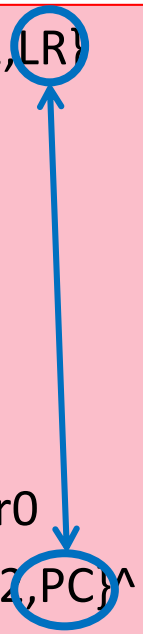
Why those two are different ?

a) LR and PC are two names for the same register
b) That is equivalent to LDMDB SP!, {….,LR}; MOV**S** PC,LR only faster
c) Interrupt servicing instruction, differently from other instructions, always return to PC instead of LR

Quiz: IMPORTANT

```
STMIA SP!,{r0-r12,LR}      ; Saving Workspace
MRS   r0,SPSR              ; Saving SPSR to stack …
STMIA SP!,{r0}             ;  … in case of further (nested) interruptions

LDMDB sp!,{r0}             ; Retrieving SPSR from stack
MSR   SPSR_cxsf, r0        ; Restoring SPSR
LDMDB SP!,{r0-r12,PC}^     ; Retrieving Workspace, PC,
                           ; and loading SPSR into CPSR
```



Why those two are different ?

a) LR and PC are two names for the same register

b) That is equivalent to LDMDB SP!, {….,LR}; ***MOVS LR,PC only faster***

c) Interrupt servicing instruction, differently from other instructions, always return to PC instead of LR

Quiz: IMPORTANT

```
STMIA SP!,{r0-r12,LR}        ; Saving Workspace
MRS   r0,SPSR                ; Saving SPSR to stack …
STMIA SP!,{r0}               ;  … in case of further (nested) interruptions



LDMDB sp!,{r0}               ; Retrieving SPSR from stack
MSR   SPSR_cxsf, r0          ; Restoring SPSR
LDMDB SP!,{r0-r12,PC}^
```

What does the ^ sign signify?

a) It means that the PC has to be incremented before being saved
b) It forces the writeback of PC
c) It forces the copy of the CPSR on to the SPSR
d) It forces the copy of the SPSR on to the CPSR

Quiz: IMPORTANT

```
STMIA SP!,{r0-r12,LR}        ; Saving Workspace
MRS   r0,SPSR                ; Saving SPSR to stack …
STMIA SP!,{r0}               ;  … in case of further (nested) interruptions



LDMDB sp!,{r0}               ; Retrieving SPSR from stack
MSR   SPSR_cxsf, r0          ; Restoring SPSR
LDMDB SP!,{r0-r12,PC}^
```

What does the ^ sign signify?

a) It means that the PC has to be incremented before being saved
b) It forces the writeback of PC
c) It forces the copy of the CPSR on to the SPSR
d) It forces the copy ot the SPSR on to the CPSR, so that the processor goes back to previous mode at the end of the exception procedure

Priorities

- Interrupts and exceptions may of course happen concurrently in a complex system. For this reason, ***they must be PRIORITIZED***

1. Data Aborts (Illegal Dmem access) have higher priority
2. FIQ have second priority, so that if an IRQ is being served it WILL be interrupted by FIQ (Unless we disable FIQ)
3. SVC and Undefined opcode come last. There is no priority between them, as we can't have both concurrently

Exceptions	Code	Priority
RESET	RES	1
DMem Access Abort	ABT	2
Fast Interrupt	FIQ	3
Normal Interrupt	IRQ	4
Imem Access Abort	ABT	5
Undefined Opcode	UND	6
Software Interrupt	SVC	6

Priorities Example

- While an ARM processor is happily doing its job, suddenly we have a number of things happen simultaneously

1. The ADC and the UART have received data from outside and they are asking the processor to read such data from their Rx Registers (these are two IRQs)
2. A high-priority hardware event has occurred (this is a FIQ)
3. The processor was trying to call a function but it run out of stack and it is then performing an illegal access in Data memory
4. It is raining outside!

Priorities Example (2)

- While an ARM processor is happily doing its job, suddenly we have a number of things happen simultaneously

1. The ADC and the UART have received data from outside and they are asking the processor to read such data from their Rx Registers (these are two IRQs)
2. A high-priority hardware event has occurred (this is a FIQ)
3. The processor was trying to call a function but it run out of stack and it is then performing an illegal access in Data memory
4. It is raining outside!

The processor will handle the memory abort first, signaling stack overflow and possibly initiating the termination of the program that caused it and reporting an error. Then the high-priority hardware will be serviced, then the A/D and the UART will be serviced.

RESET

When using the Keil uVision simulator we have been working-around with dirty tricks the RESET configuration of our processor systems

- In fact, ***the RESET is the highest priority form of Exception***
  - Upon reset, the processor goes to the vector table and branches to the servicing routine that is dedicated to the reset, the RESET HANDLER
  - The RESET HANDLER needs to set up the state of the processor before handing off the control of the processor to the next portion of the program:
    - Set System registers (ex. Setting up the Stack Pointers)
    - Configure any necessary peripherals (Including clock)
    - Enable / Disable interrupts
    - Change processor State to System (or possibly User)
    - Branch to the next portion of the program

Undefined Instruction (Example)

The undefined Instruction Interrupt handler can be particularly useful to emulate assembly instructions that are available in a version of the instruction set but are not implemented in the particular processor version being used:

Note: Example below is for your reference/enjoyment, does not need to be reviewed for written exams!

Example 1: We would like to use a MLS r1,r2,r3,r4 (Multiply Subtract) instruction. We don't have that in our version of ISA, but we want to use the machine code of a program that was written for a version of the ISA that does.

- When we get to MLS, we get an undefined instruction exception
- In the Exception Handler routine, we look at the instruction that caused the exception and if it happens to be a MLS, we just compute using MUL and SUB instructions as appropriate, returning the result on the appropriate register r1

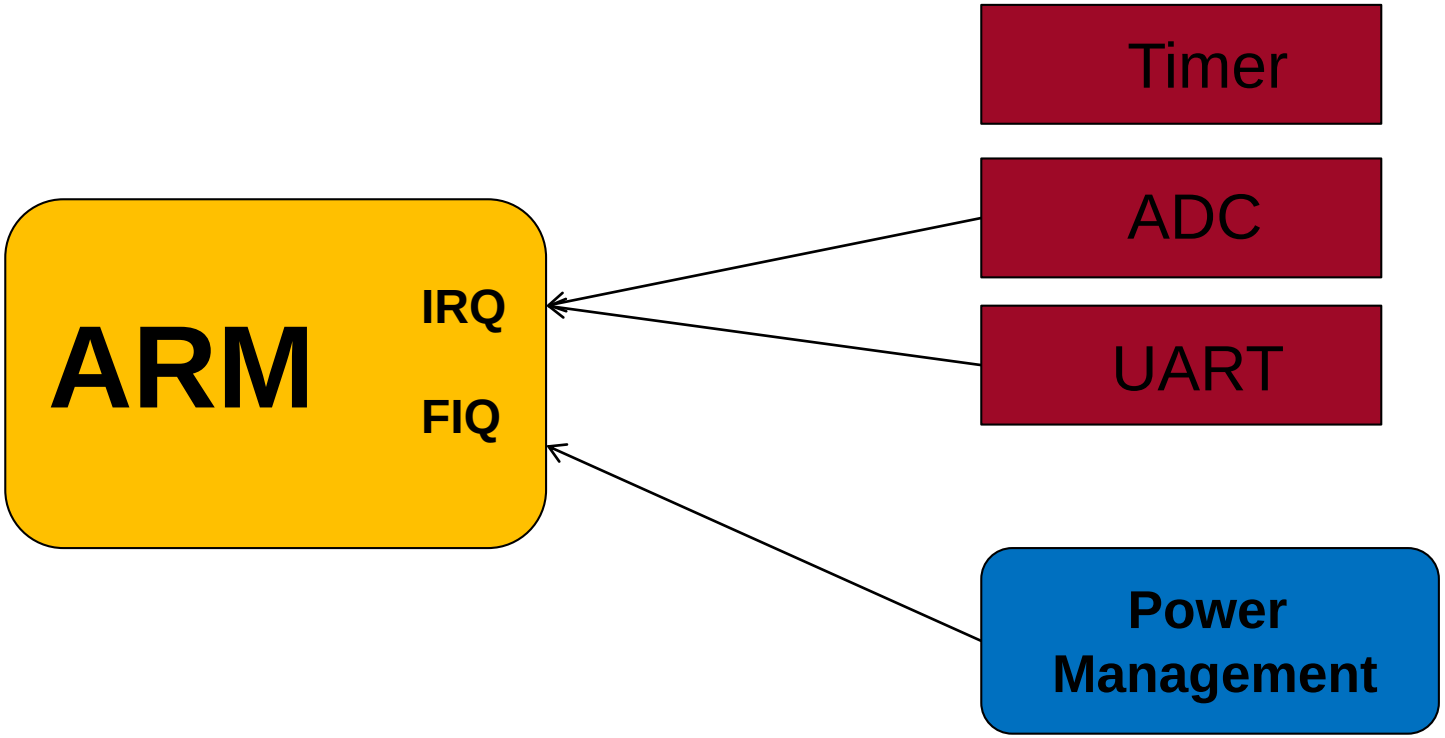
Software Interrupt

- A software interrupt is an exception willingly inserted by the programmer in the program flow.

- This can be done to enjoy special privileges that exceptions enable: we have discussed how the change to USER mode during the reset handler is a no-return operation: the only way for a USER MODE program to alter the status of the processor (For example, enable or disable an iterrupt or access to reserved address areas) is to move back into SUPERVISOR MODE.
    - operating systems are based on “System Calls” being executed in supervisor mode to access restricted resources (File system, peripherals..)
- This can be done issuing a software interrupt: The instruction that causes a software interrupt in ARM is ***SWI <immed_24>*** or (***SVC <immed_24>***)
- ***The user can specify an immediate argument to SWI/SVC to signal to the handler which kind of service he wants. BUT THE entry in the Vector table is the same for all SVC. The Exception Handler will need to go into instruction memory, load the instruction, and detect the argument from the instruction!***

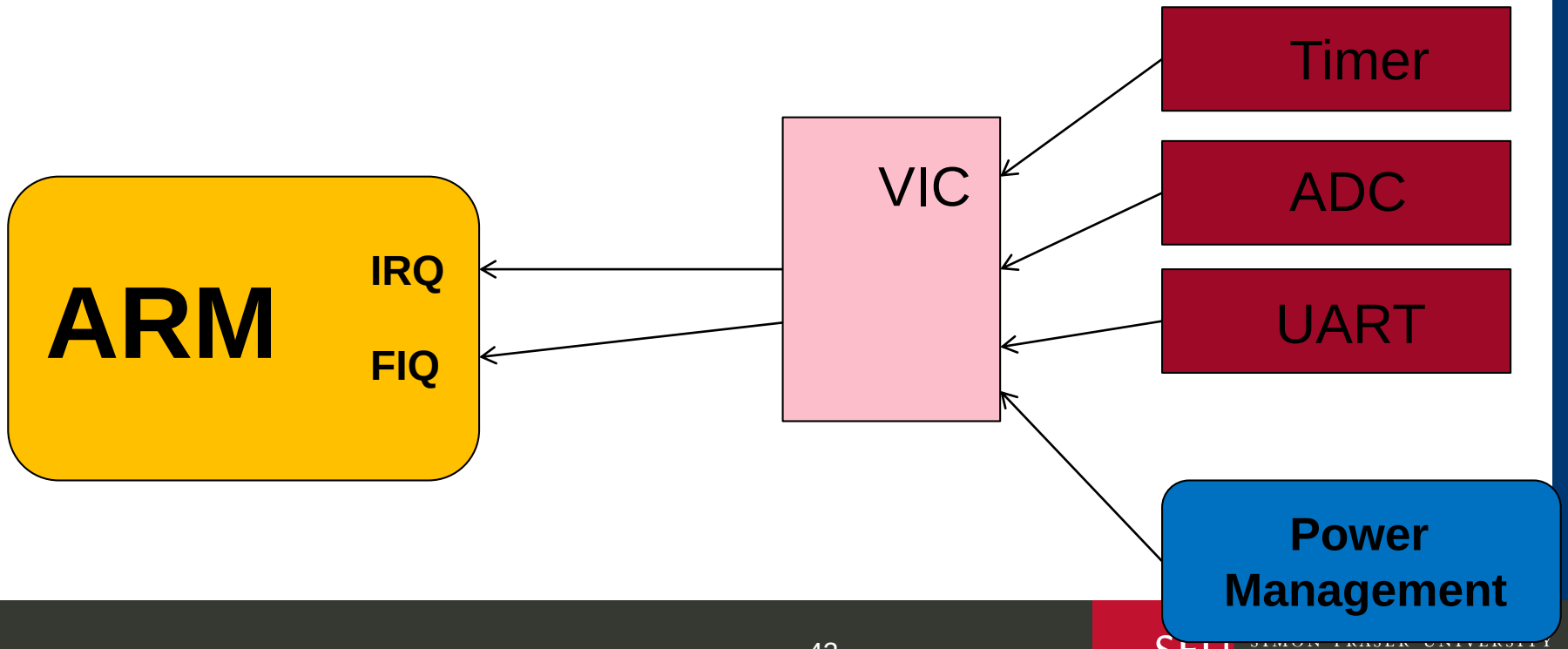
INTERRUPTS

- As introduced previously, Interrupts are signals coming from OUTSIDE the processor, and asking for the processor attention in a way that is totally asynchronous to the flow of the program



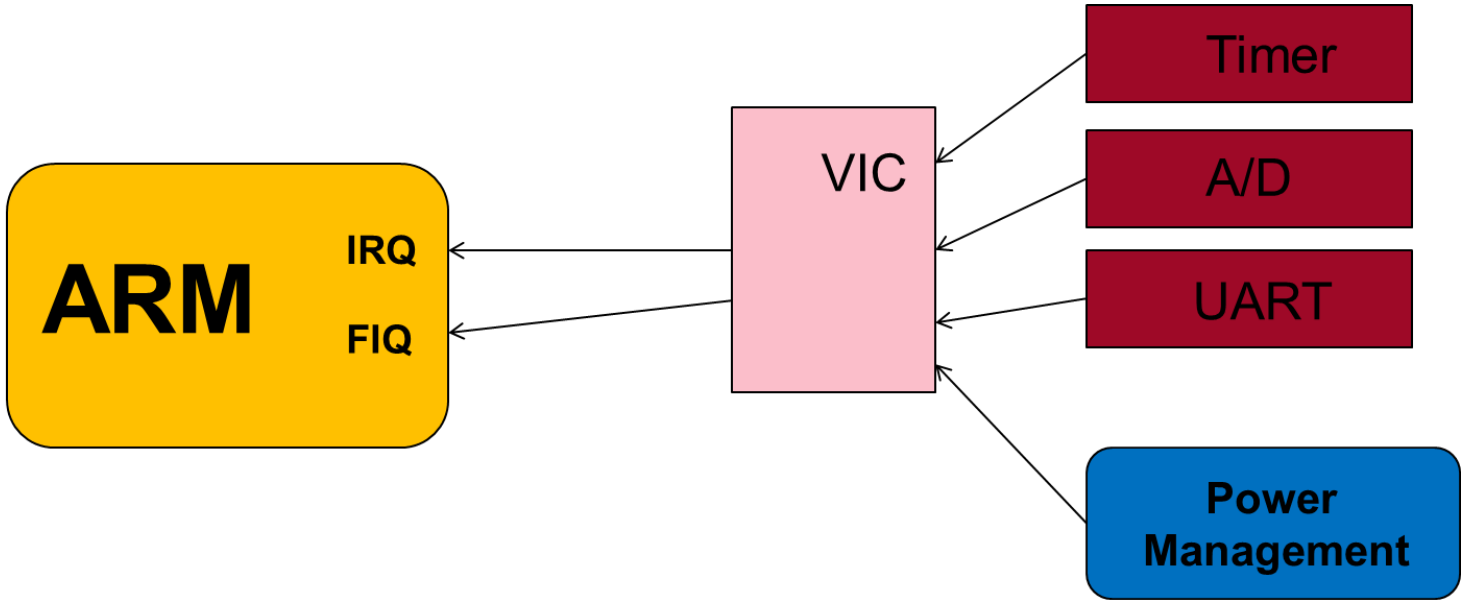
INTERRUPTS (2)

- Some processors feature 8 to 15 different interrupt pins in order to handle various independent sources of interruption.
- ARM features only 2 pins. Since most microcontrollers will need to handle Multiple independent sources of interrupts, these are normally handled making use of a peripheral known as the VIC (Vectored Interrupt Controller)



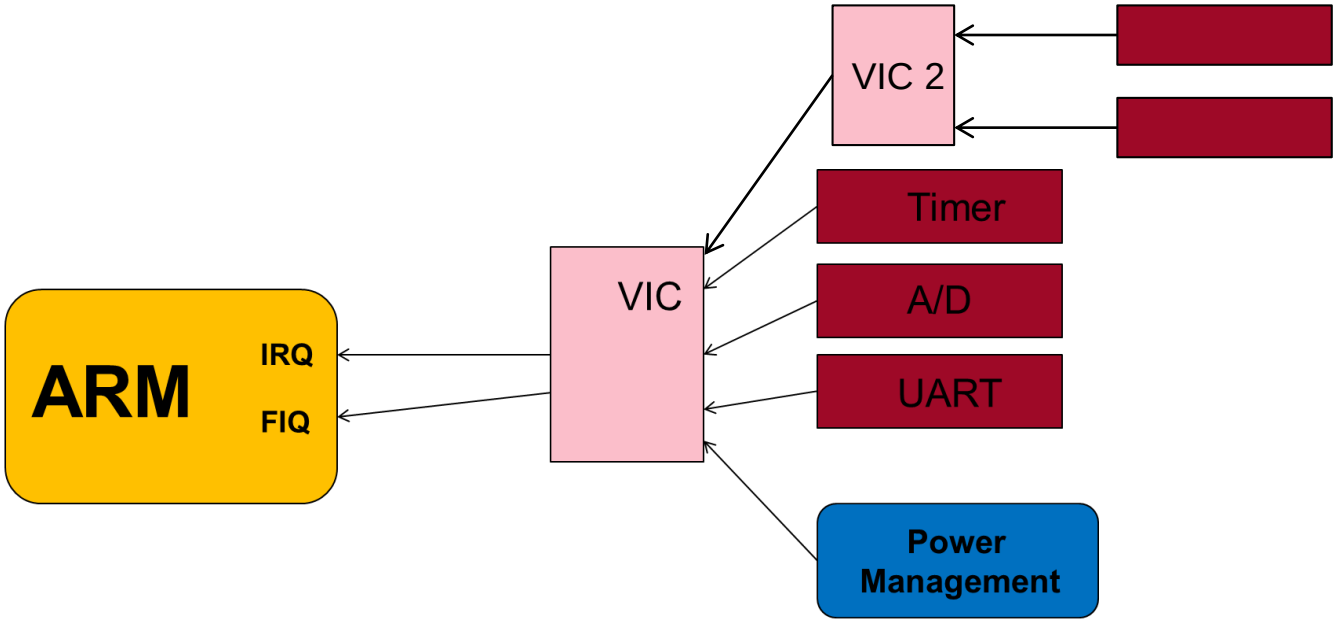
INTERRUPTS (3)

- The VIC will collect all non-urgent sources of interrupts, organize them according to their priority and send a single «general» Interrupt signal to ARM
- Upon receiving the IRQ, ARM will read the VIC status through the bus, decoding the information about what peripheral needs service next, and hence proceeding to servicing in the appropriate order all peripherals requiring attention



Pros and Cons of the VIC Approach

- The advantage of this approach is that ARM has a smaller number of independent pins to handle, and that allows for more compact circuits in the processor
- Another advantage is that this solution is scalable: if we need more interrupts than are available in our VIC, we can use a second VIC as peripheral of the first and so on ……



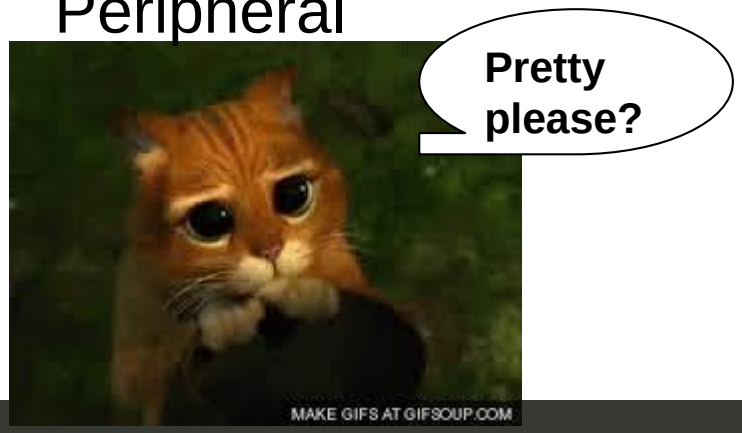
Interrupt Masking

- At some point in the operation of an embedded system, we may decide that the processor is performing a task that is too critical to be interrupted.
- Although this is dangerous in general, it could be useful for short periods and specific situations to ***MASK*** the interrupts

We define INTERRUPT MASKING THE ACT OF DISABLING FOR A CERTAIN PERIOD THE RECEPTION OF INTERRUPTS FROM THE PROCESSOR

- ***Masking is obtained by means of specific control flags. Of course, in case of vectorized interrupt control, it can be done at different levels***

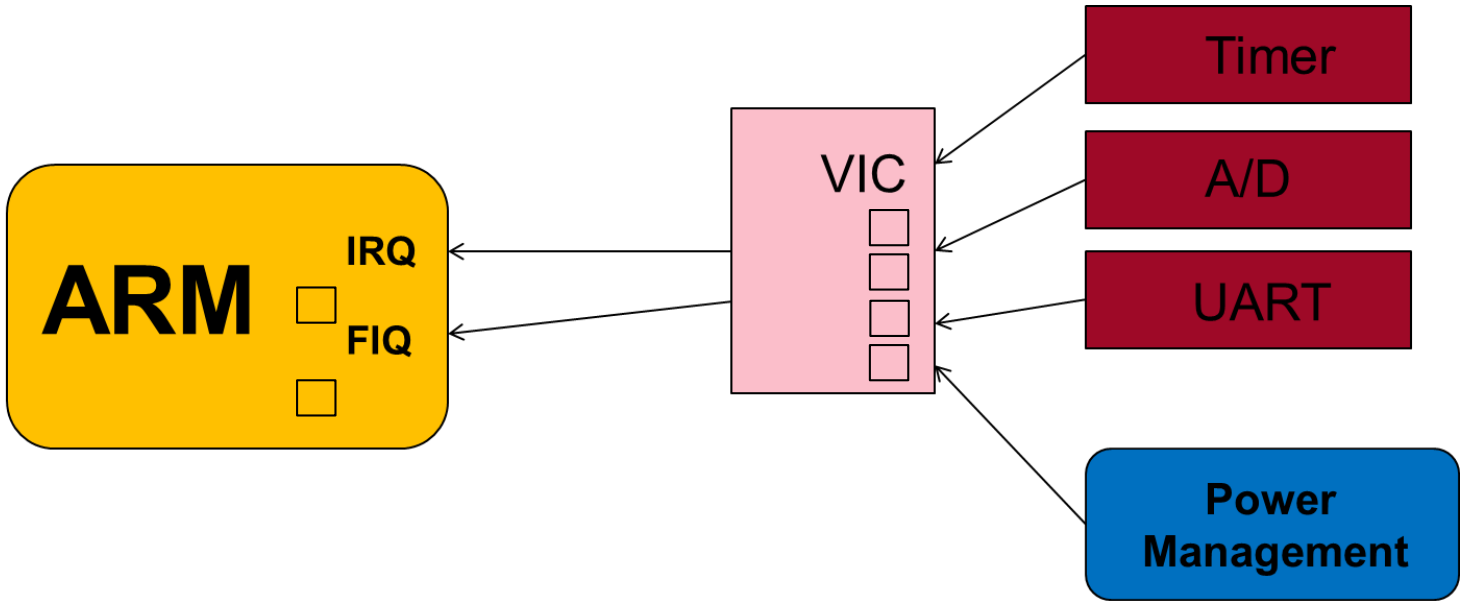
Peripheral



ARM

Interrupt Masking in ARM-based systems

- Two Bits in the ARM Program Status registers can be used to mask IRQ and FIQ interrupts. FIQ and IRQ typically come from the VIC controller (if it is present in the system)
- Those flags can be changed only in certain modes (such as supervisor (SVC) mode), with the instructions MSR or CPS. Cannot be done in user mode.
- If we want to control every single interrupt source, we will have to use the VIC-specific masking registers



Quizzes

Name 3 ways in which FIQ interrupts are faster than IRQ interrupts?

How can you change from Supervisor mode to User mode?

How can you change from User mode to Supervisor mode?

How many Interrupt lines are available with ARM?

What if we have more Interrupt sources ?

Quizzes

Name 3 ways in which FIQ interrupts are faster than IRQ interrupts?

1. Higher priority
2. No need to branch to FIQ exception handler
3. Smaller number of registers to save on stack

How can you change from Supervisor mode to User mode?

With the MSR or CPS instructions, or with LDM…^, or with MOVS PC, …

How can you change from User mode to Supervisor mode?

Only by means of a system Call (SWI/SVC instruction), causing an exception

How many Interrupt lines are available with ARM?

Two: FIQ and IRQ

What if we have more Interrupt sources ?

The interrupts need to share the two lines or we need to use a VIC or chained VICs