Dariusz Makowski

Department of Microelectronics and Computer Science

tel. 631 2720

dmakow@dmcs.pl

http://neo.dmcs.pl/es

From Acorn Computers Ltd. ARM to ARM Ltd.

Acorn

- Small company founded in November 1990,
  - Spun out of Acorn Computers (BBC Micro computer),
- Design the ARM range of RISC processor cores,
- ***ARM company does not fabricate silicon itself***,
- Licenses ARM cores to partners: **I**ntellectual **P**roperty **C**ores of ARM processors and peripheral devices,
- Develop tools (compilers, debuggers), starter-kits for embedded system development and creates standards, etc...

List of ARM silicon partners

Agilent, AKM, Alcatel, Altera, Atmel, Broadcom, Chip Express, Cirrus Logic, Digital Semiconductor, eSilicon, Fujitsu, GEC Plessey, Global UniChip, HP, Hyundai, IBM, Intel, ITRI, LG Semicon, LSI Logic, Lucent, Matsushita, Micrel, Micronas, Mitsubishi, Freescale, NEC, OKI, Philips, Qualcomm, Rockwell, Rohm, Samsung, Samsung, Sanyo, Seagate, Seiko Epson, Sharp, Sony, STMicroelectronics, Symbios Logic, Texas Instruments, Xilinx, Yamaha, Zeevo, ZTEIC, ...

History of ARM Processors

1983 – Sophie Wilson and Steve Furber fabricate the first RISC processor in Acorn Computers Limited, Cambridge, ARM = **A**corn (**A**dvanced) **R**ISC **M**achine

1985 – The first processor ARM 1 (architecture version v1)

1986 – First ARM 2 processors left company (32-bits, 26-bits address, 16 registers 16-bits, 30.000 transistors, architecture version v2/v2a, 8 MHz)

1990 – Apple Computer and VLSI Technology start work on the next version of ARM core,

1990 – New company is created Advanced RISC Machines Ltd. Responsible for the development of ARM cores,

1991 – The cooperation of Apple and VLSI Tech. provides new ARM 6 processor (ARM 610 applied in Apple Newton PDA, architecture version v3, 33 MHz)

1995 – ARM company offers famous **ARM7TDMI core** (core architecture **ARMv4T**) and Intel offers StrongARM (233 MHz)

2001 – ARM company offers **ARM9TDMI core** (core architecture **ARMv5TEJ**, 220 MHz)

2004 – Cortex M3 processor (**ARMv7-M**, 100 MHz)

2008 – ARM Cortex A8 (core architecture ARMv7, 1 GHz)

2012 – ARM Cortex A9/A15/A17 (**ARMv7-A**, 32-bit, 1-2 GHz) – MPCore architecture

2021 - 2020 – ARM Cortex A53/A65/A78 (**ARMv8-A**, 64-bit, 2.0-2.6 GHz) – MPCore architecture, GPU

2021 – now – ARM Cortex A510/A710 (**ARMv9-A**, 64-bit, 2.0-3.1 GHz) – MPCore architecture, GPU

Processors with ARM Core

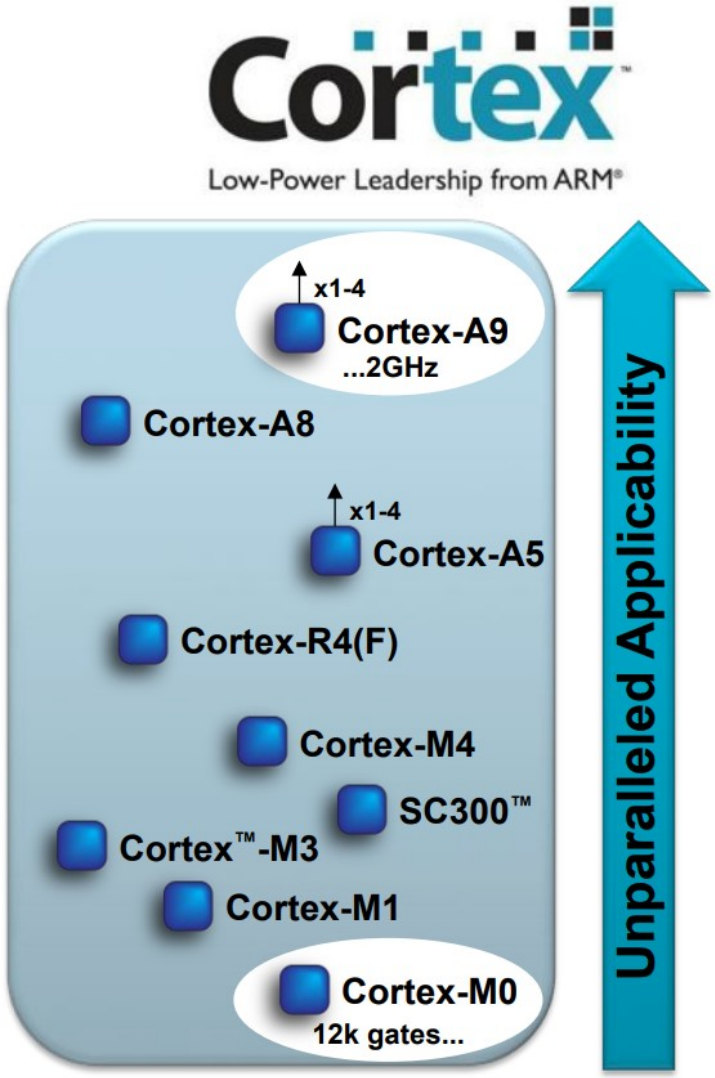
- ARM processors are widely used in embedded systems and mobile devices that require low power devices
- The ARM processor is the most commonly used device in the World. You can find the processor in hard discs, mobile phones, routers, calculators and toys,
- Currently, more than 75% of 32-bits embedded CPUs market belongs to ARM processors,
- The most famous and successful processor is ARM7TDMI, very often used in mobile phones,
- Processing power of ARM devices allows to install multitasking operating systems with TCP/IP software stack and filesystem (e.g. FAT32).
- The known operating systems for ARM processors: embedded Linux (Embedded Debian, Embedded Ubuntu), Windows CE, Symbian, NUTOS (Ethernut), RTEMS,...

ARM Powered Products

ARM Cortex Advanced Processors

Architectural innovation, compatibility across diverse application spectrum

- **ARM Cortex-A family**
  - Applications processors for feature-rich OS and 3rd party applications
- **ARM Cortex-R family**
  - Embedded processors for real-time signal processing, control applications
- **ARM Cortex-M family**
  - Microcontroller-oriented processors for MCU, ASSP, and SoC applications
- **Mali GPUs**
  - Graphics processors for a range of mobile devices from smartwatches to autonomous vehicles.



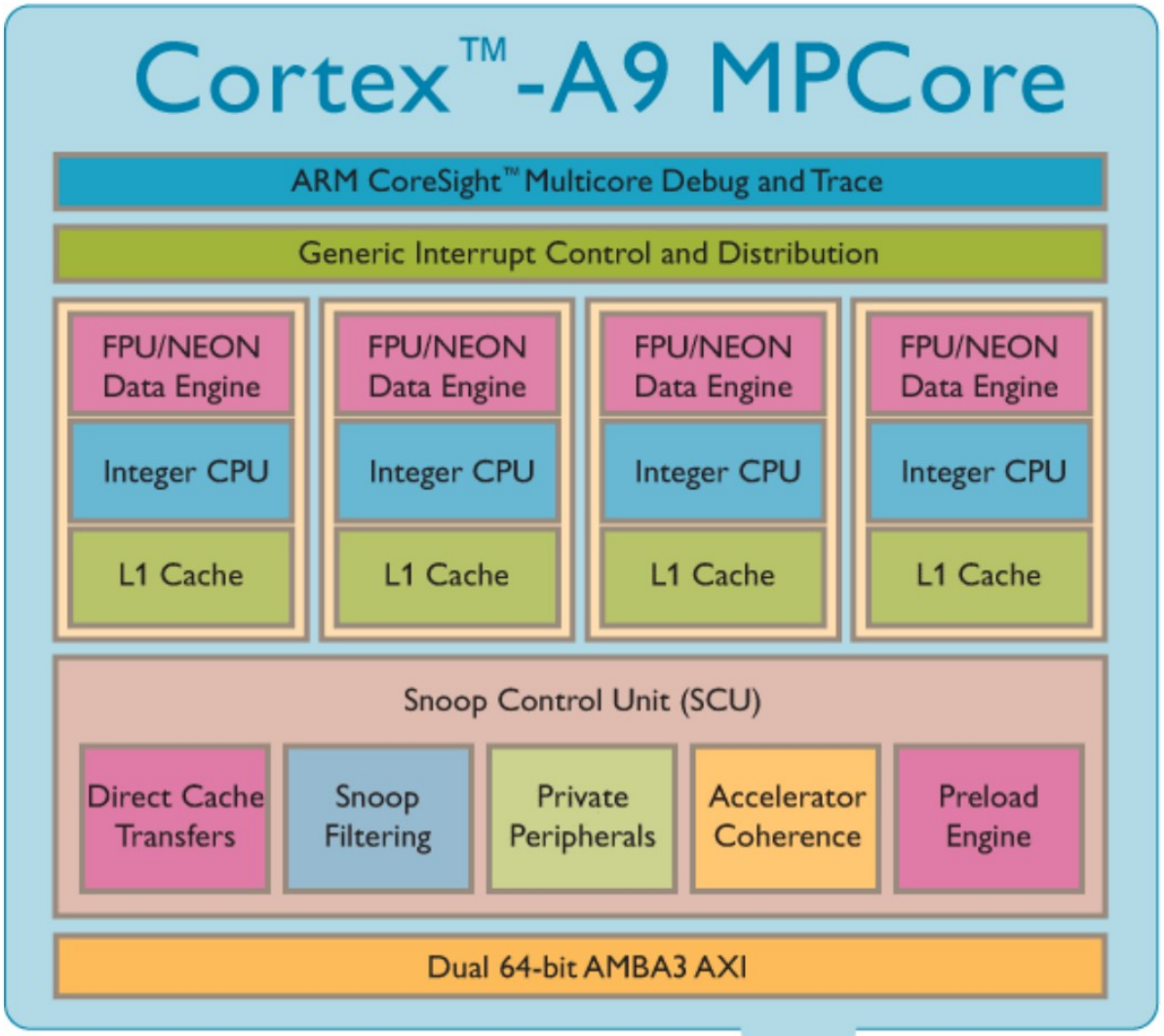
Comparison of Selected ARMs

Family	Architecture Version	Core	Feature	Cache (I/D)/MMU	Typical MIPS @ MHz
ARM6	ARMv3	ARM610	Cache, no coprocessor	4K unified	17 MIPS @ 20 MHz
ARM7	ARMv3	ARM7500FE	Integrated SoC. "FE" Added FPA and EDO memory controller.	4 KB unified	55 MIPS @ 56 MHz
ARM7TDMI	ARMv5TEJ	ARM7EJ-S	Jazelle DBX, Enhanced DSP instructions, 5-stage pipeline	8 KB	120 MIPS @ 133 MHz
StrongARM	ARMv4	SA-110	5-stage pipeline, MMU	16 KB/16 KB, MMU	235 MIPS @ 206 MHz
ARM8	ARMv4	ARM810[7]	5-stage pipeline, static branch prediction, double-bandwidth memory	8 KB unified, MMU	1.0 DMIPS/MHz
ARM9TDMI	ARMv4T	ARM920T	5-stage pipeline	16 KB/16 KB, MMU	245 MIPS @ 250 MHz
ARM9E	ARMv5TEJ	ARM926EJ-S	Jazelle DBX, Enhanced DSP instructions	variable, TCMs, MMU	220 MIPS @ 200 MHz
ARM10E	ARMv5TE	ARM1020E	VFP, 6-stage pipeline, Enhanced DSP instructions	32 KB/32 KB, MMU	300 MIPS @ 325 MHz
XScale	ARMv5TE	PXA27x	MMX and SSE instruction set, four MACs,	32 Kb/32 Kb, MMU	800 MIPS @ 624 MHz
ARM11	ARMv6	ARM1136J(F)-S	SIMD, Jazelle DBX, VFP, 8-stage pipeline	variable, MMU	740 @ 532-665 MHz
Cortex	ARMv7-A	Cortex-A8	Application profile, VFP, NEON, Jazelle RCT, Thumb-2, 13-stage superscalar pipeline	variable (L1+L2), MMU+TrustZone	>1000 MIPS@ 600 M-1 GHz

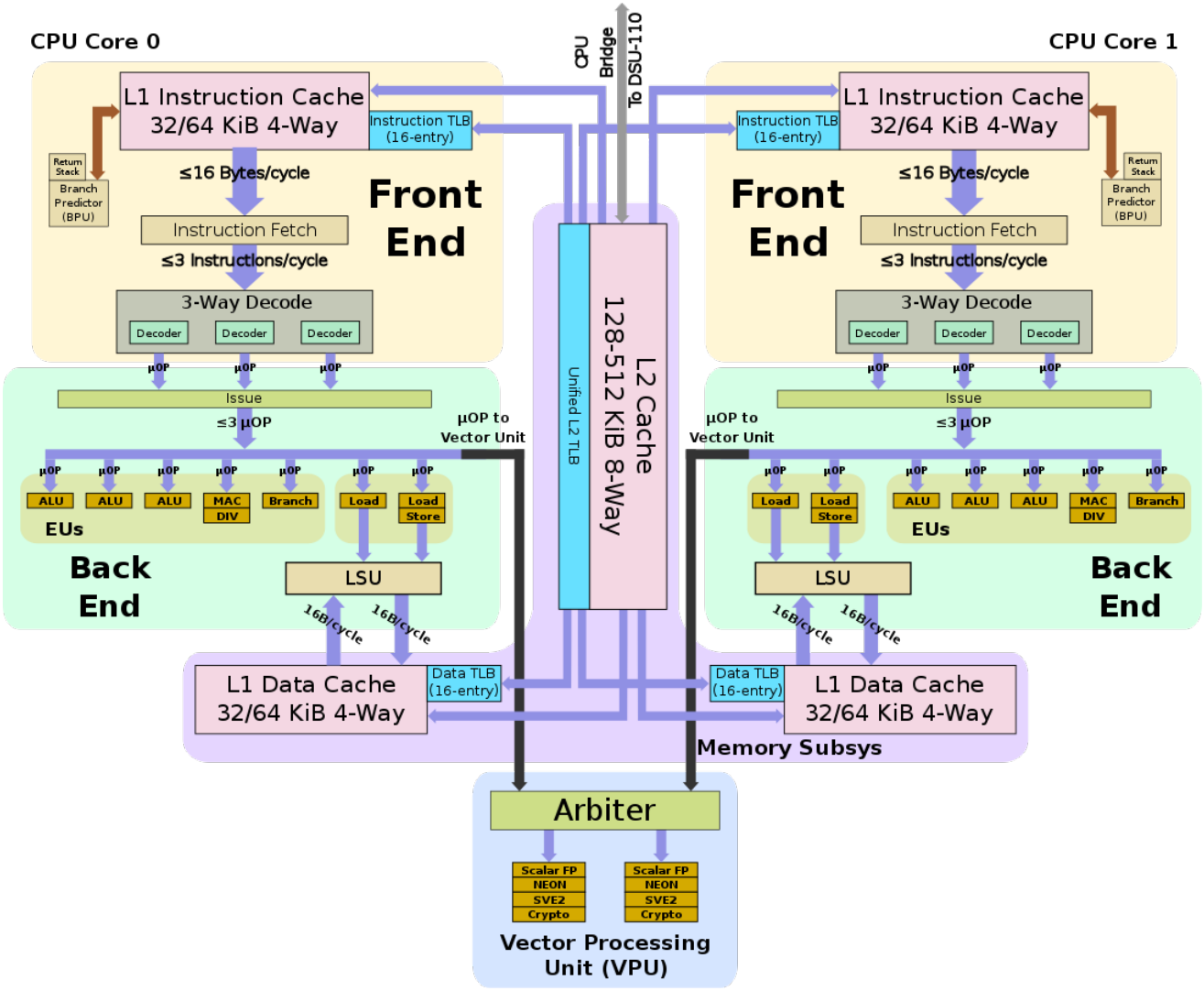
Cortex – A9 MPCore

Typical applications

- Smartphones and Tablets
  - Cortex A Series
  - Combines Power Efficiency and performance
- Primary Products
  - Cortex A5
  - Cortex A9
- Some manufactures like Qualcomm only use the ARMv7 ISA



ARM Cortex A510 in MPCore Configuration



Cortex M Microcontrollers' Family (1)

Feature	Cortex-M0	Cortex-M0+	Cortex-M1	Cortex-M23	Cortex-M3	Cortex-M4	Cortex-M33	Cortex-M35P	Cortex-M55	Cortex-M7
Instruction Set Architecture	Armv6-M	Armv6-M	Armv6-M	Armv8-M Baseline	Armv7-M	Armv7-M	Armv8-M Mainline	Armv8-M Mainline	Armv8.1-M Mainline	Armv7-M
TrustZone for Armv8-M	No	No	No	Yes (option)	No	No	Yes (option)	Yes (option)	Yes (option)	No
Digital Signal Processing (DSP) Extension	No	No	No	No	No	Yes	Yes	Yes	Yes	Yes
Hardware Divide	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Arm Custom Instructions	No	No	No	No	No	No	Yes	No	Yes	No
Coprocessor Interface	No	No	No	No	No	No	Yes	Yes	Yes	No
DMIPS/MHz*	0.87	0.95	0.8	0.98	1.25	1.25	1.5	1.5	1.6	2.14
CoreMark®/MHz*	2.33	2.46	1.85	2.64	3.34	3.42	4.02	4.02	4.2	5.01

Cortex M Microcontrollers' Family (1)

Arm Core	Cortex M0[2]	Cortex M0+[3]	Cortex M1[4]	Cortex M3[5]	Cortex M4[6]	Cortex M7[7]	Cortex M23[8]	Cortex M33[12]	Cortex M35P	Cortex M55
ARM architecture	ARMv6-M[9]	ARMv6-M[9]	ARMv6-M[9]	ARMv7-M[10]	ARMv7E-M[10]	ARMv7E-M[10]	ARMv8-M Baseline[15]	ARMv8-M Mainline[15]	ARMv8-M Mainline[15]	Armv8.1-M
Computer architecture	Von Neumann	Von Neumann	Von Neumann	Harvard	Harvard	Harvard	Von Neumann	Harvard	Harvard	Harvard
Instruction pipeline	3 stages	2 stages	3 stages	3 stages	3 stages	6 stages	2 stages	3 stages	3 stages	4 to 5 stages
Thumb-1 instructions	Most	Most	Most	Entire	Entire	Entire	Most	Entire	Entire	Entire
Thumb-2 instructions	Some	Some	Some	Entire	Entire	Entire	Some	Entire	Entire	Entire
Multiply instructions 32x32 = 32-bit result	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Multiply instructions 32x32 = 64-bit result	No	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes
Divide instructions 32/32 = 32-bit quotient	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Saturated instructions	No	No	No	Some	Yes	Yes	No	Yes	Yes	Yes
DSP instructions	No	No	No	No	Yes	Yes	No	Optional	Optional	Optional
Single-Precision (SP) Floating-point instructions	No	No	No	No	Optional	Optional	No	Optional	Optional	Optional
Double-Precision (DP) Floating-point instructions	No	No	No	No	No	Optional	No	No	No	Optional
Half-Precisions (HP)	No	No	No	No	No	No	No	No	No	Optional
TrustZone instructions	No	No	No	No	No	No	Optional	Optional	Optional	Optional
Co-processor instructions	No	No	No	No	No	No	No	Optional	Optional	Optional
Helium technology	No	No	No	No	No	No	No	No	No	Optional
Interrupt latency (if zero-wait state RAM)	16 cycles	15 cycles	23 for NMI 26 for IRQ	12 cycles	12 cycles	12 cycles 14 worst case	15 no security ext 27 security ext	12 no security ext ?? security ext	TBD	TBD

ARM Processor Core

ARM architecture (1)

ARM processor core – processor designed according to ARM processor architecture described in high level description language (VHDL lub Verilog) provided as macro-cell or Intellectual Property (IP).

Features of ARM processor cores:

- Supposed to be used for further development – microcontroller, SoC
- 32 or 64-bits RISC architecture
- Optimised for low power consumption
- Support three different modes of operation:
  - ARM instructions, 32 bits,
  - Thumb instructions, 16 bits,
  - Jazelle DBX - Direct java instructions.
- Supported Big or Little Endian
- Fast Interrupt Response mode for Real-time applications
- Virtual memory
- List of efficient and powerful instructions selected from both RISC and CISC architectures
- Hardware support for higher level software (Ada, C, C++)

ARM architecture (2)

Nomenclature:

ARM {x} {y} {z} {T} {D} {M} {I} {E} {J} {F} {S}

- **x** – core family
- **y** – implemented Memory Management Unit
- **z** – cache memory
- **T** – Thumb mode (16 bit command)
- **D** – Build in debugger, (usually via JTAG interface)
- **M** – Build in multiplier, hardware multiplier (32x32 => 64 bits)
- **I** – In-Circuit Emulator, another ICE debugger
- **E –** Enhanced DSP instructions, Digital Signal Processing
- **J** – Jazelle mode
- **F** – Floating-point unit
- **S** – Synthesizable version, available source code for further synthesis and EDA tools

Example of ARM cores:

ARM7TDMI **ARM9TDMI-EJ-S**

ARM architecture (3)

- **Core in version 1, v1**
  - Base arithmetic and logic operations,
  - Hardware interrupts,
  - 8 and 32 bits operations,
  - 26 bits address
- **Core in version 2, v2**
  - Implemented Multiply ACcumulate unit,
  - Available coprocessor,
  - Additional commands for threads synchronisation ,
  - 26 bits address
- **Core in version 3, v3**
  - New registers CPSR, SPSR, MRS, MSR,
  - Additional modes Abort and Undef,
  - 32 bits address

ARM architecture (4)

- **Core in version 4, v4**
  - First standardised architecture
  - Available 16 bits operations
  - THUMB - new mode of operation, 16 bits commands
  - Added privileged mode
  - PC can be incremented by 64 bits
- **Core in version 5, v5**
  - Improved cooperation between ARM and THUMB modes, mode of operation can be changed during program execution,
  - Added instruction CLZ
  - Software breakpoints
  - Support for multiprocessor operation
- **Core in version 6, v6**
  - Improved MMU (Management Memory Unit)
  - Hardware support for video and sound processing (FFT, MPEG4, SIMD etc...)
  - Improved exception handing (new flag in PSR)

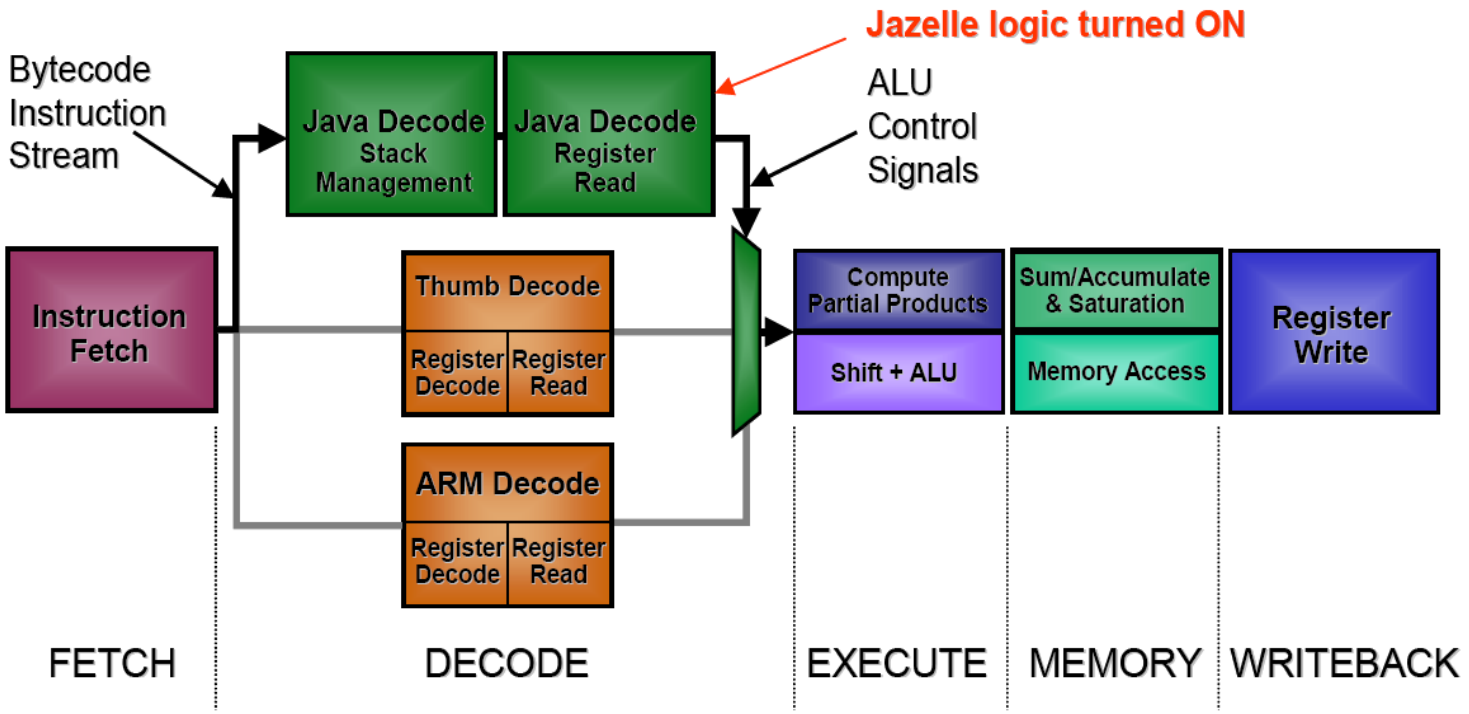
ARM instruction sets

Taking into consideration executed commands ARM processor can operate in one of the following modes:

- **ARM** – 32-bits instructions optimised for time execution (code must be aligned to 4 bytes),
- **Thumb**, **Thumb-2** – 16-bits instructions optimised for code size (code must be aligned to 2 bytes, processor registers are still 32 bits wide),
- **Jazelle v1** – mode used for direct execution of Java code (without virtual machine JVM) (1000 Caffeine Marks @ 200MHz)

Support for Java language

- ARM core marked with 'J'
- Dynamic exchange of registers and stack
- Hardware decoder of Java instructions

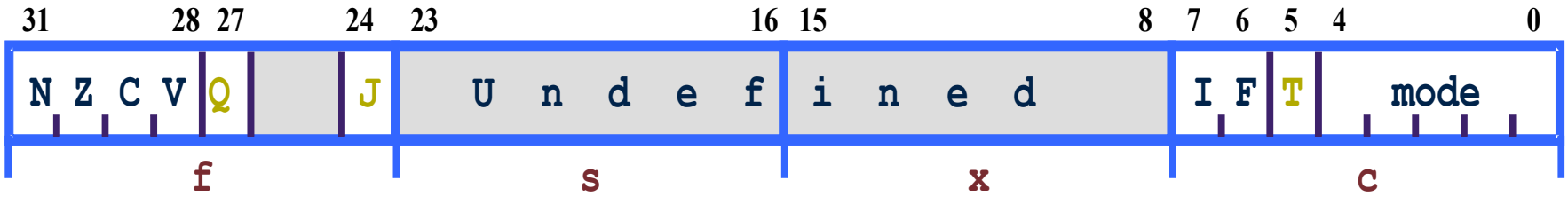


Programming Model – Registers

ARM Processor provides 37 registers (all are 32-bits wide). The registers are arranged into several banks (accessible bank being governed by the current processor mode):

- **PC (r15)** – Program Counter
- **CPSR** – Main status register, Current Program Status Register
- **SPSR** – Copy of status register, available in different modes of operation Saved Program Status Register
- **LR (r14)** – Link Register, used for stack frame during execution of subroutines or return address register
- **SP (r13)** – used as a Stack Pointer
- **r0 - r12** – General purpose registers (dependent of the mode of operation)

Program Status Register



Condition code flags

- V – ALU operation o**V**erflowed
- C – ALU operation **C**arried out
- Z – **Z**ero result from ALU operation
- N – **N**egative result from ALU operation

Flags for processor from family 5TE/J

- J – Processor in Jazelle mode
- Q – Sticky Overflow – saturation flag, set during ALU operations (QADD, QDADD, QSUB or QDSUB, or operation of SMLAxy, SMLAWx, result more than 32 bits)

Interrupt disable bits

- I=1 Disables the IRQ
- F=1 Disables the FIQ

Flags for xT architecture

- T=0 Processor in ARM mode
- T=1 Processor in Thumb mode

Mode bits

- Specify the processor operation mode (seven modes)

Read/Modify/Write strategy should be used to write data to PSR (to ensure further compatibility)

Programming Model – modes of processor operation

Operating mode – defined which resources of processor are available, e.g. registers, memory regions, peripheral devices, stack, etc...

ARM processor can operate in on of 7 modes:

- **User –** user mode (not privileged), dedicated for user programs execution
- **FIQ –** fast interrupts and high priority exceptions (used only when really necessary)
- **IRQ –** handling of low or normal priority interrupts
- **Supervisor** – supervisor mode gives access to all resource of the processor, used during debugging. Available after reset or during interrupt handling.
- **Abort –** used for handling of memory access exceptions (memory access violations)
- **Undef –** triggered when unknown or wrong commands is detected
- **System –** privileged mode, access to registers as in user mode, however various memory segments are available

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

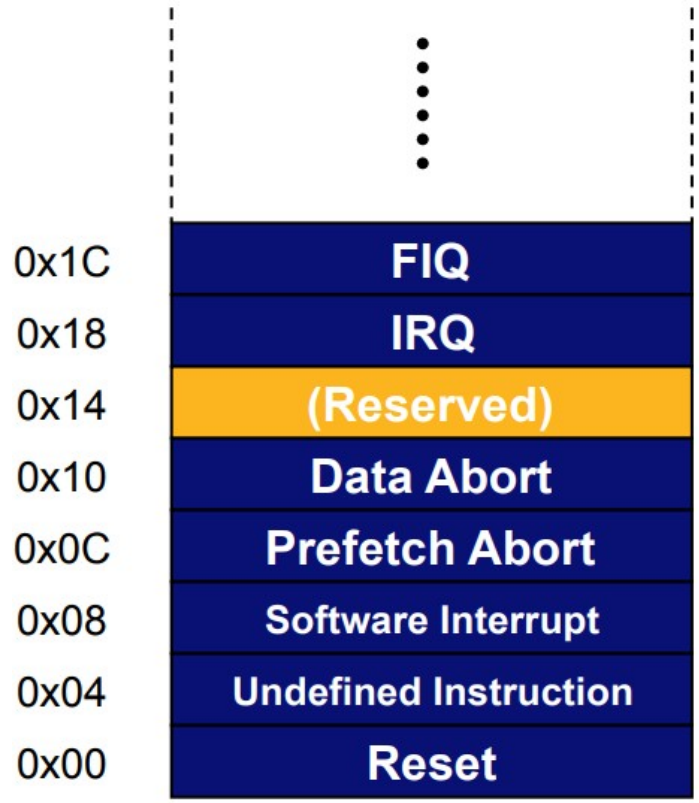
Exception Handling

When an exception occurs, the ARM:

- Copies CPSR into SPSR_<mode>
- Sets appropriate CPSR bits
  - Change to ARM or Thumb state
  - Change to exception mode
  - Disable interrupts (if appropriate)
- Stores the return address in LR_<mode>
- Sets PC to vector address

To return, exception handler needs to:

- Restore CPSR from SPSR_<mode>
- Restore PC from LR_<mode>



Vector Table

Vector table can be at
0xFFFF0000 on ARM720T
and on ARM9/10 family devices

Programming Model – registers available in User or System modes

Current Visible Registers

User Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

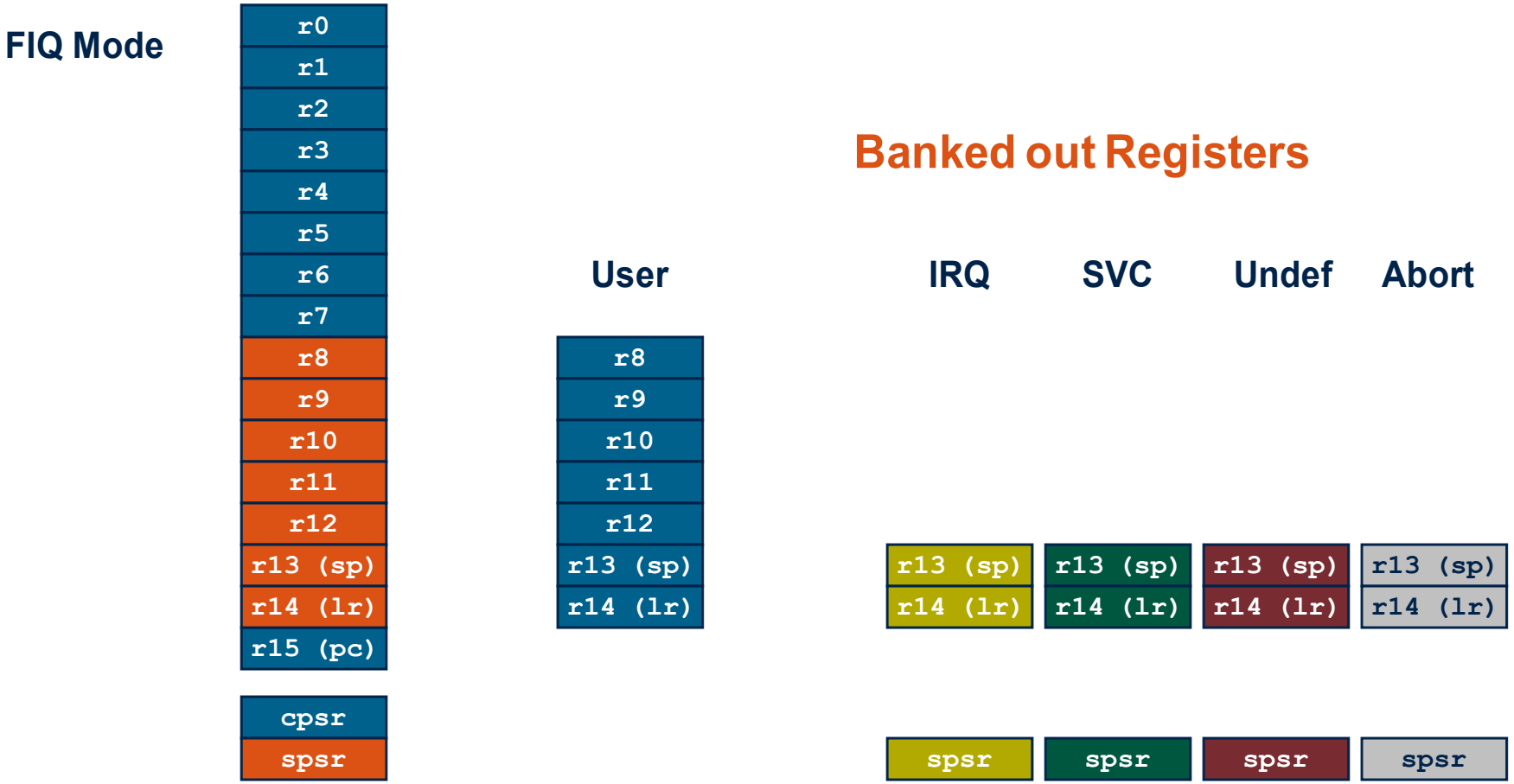
cpsr

Banked out Registers

FIQ	IRQ	SVC	Undef	Abort
r8				
r9				
r10				
r11				
r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
spsr	spsr	spsr	spsr	spsr

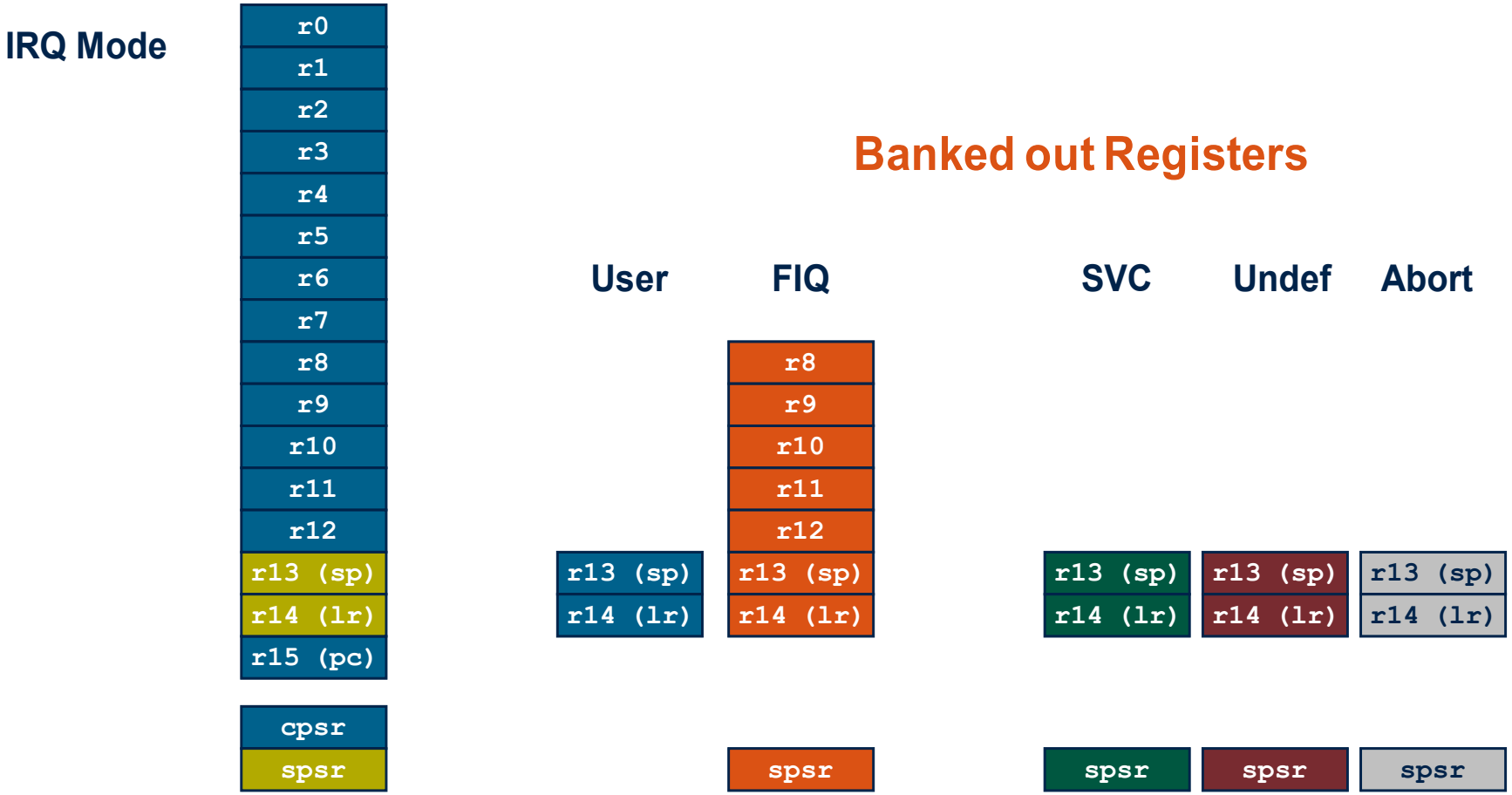
Programming Model – registers available in FIQ mode

Current Visible Registers



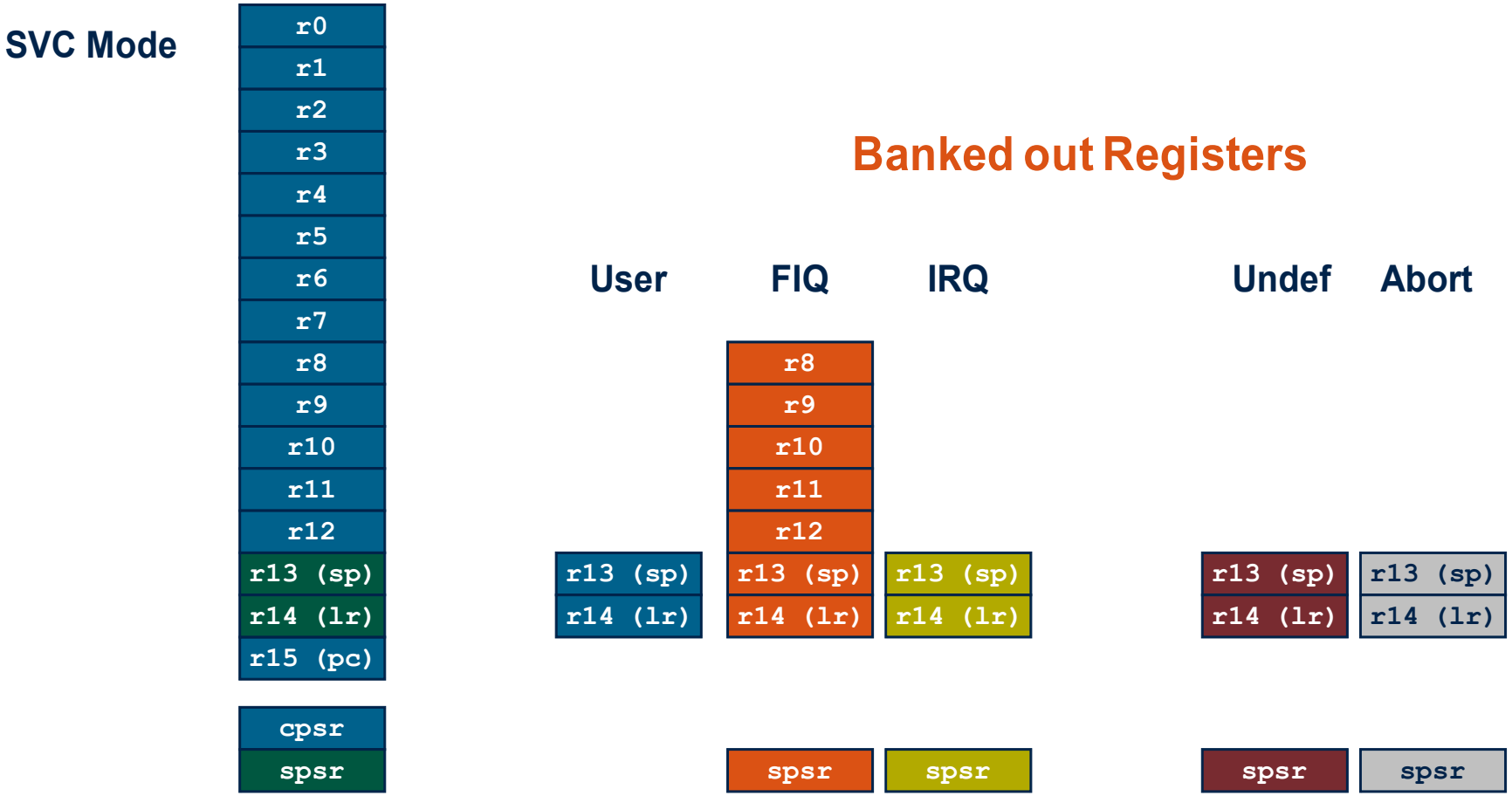
Programming Model – registers available in IRQ mode

Current Visible Registers



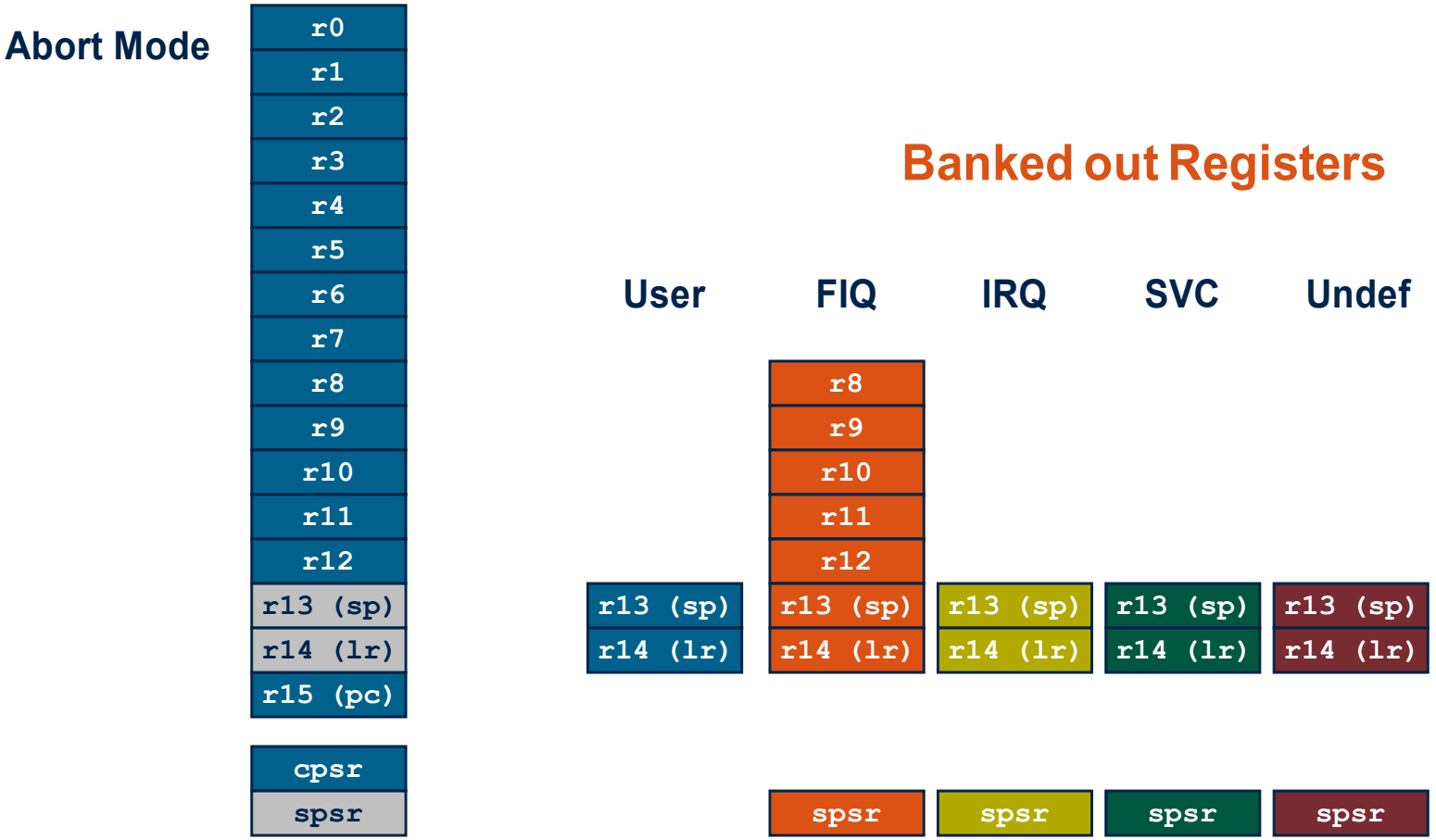
Programming Model – registers available in Supervisor mode

Current Visible Registers



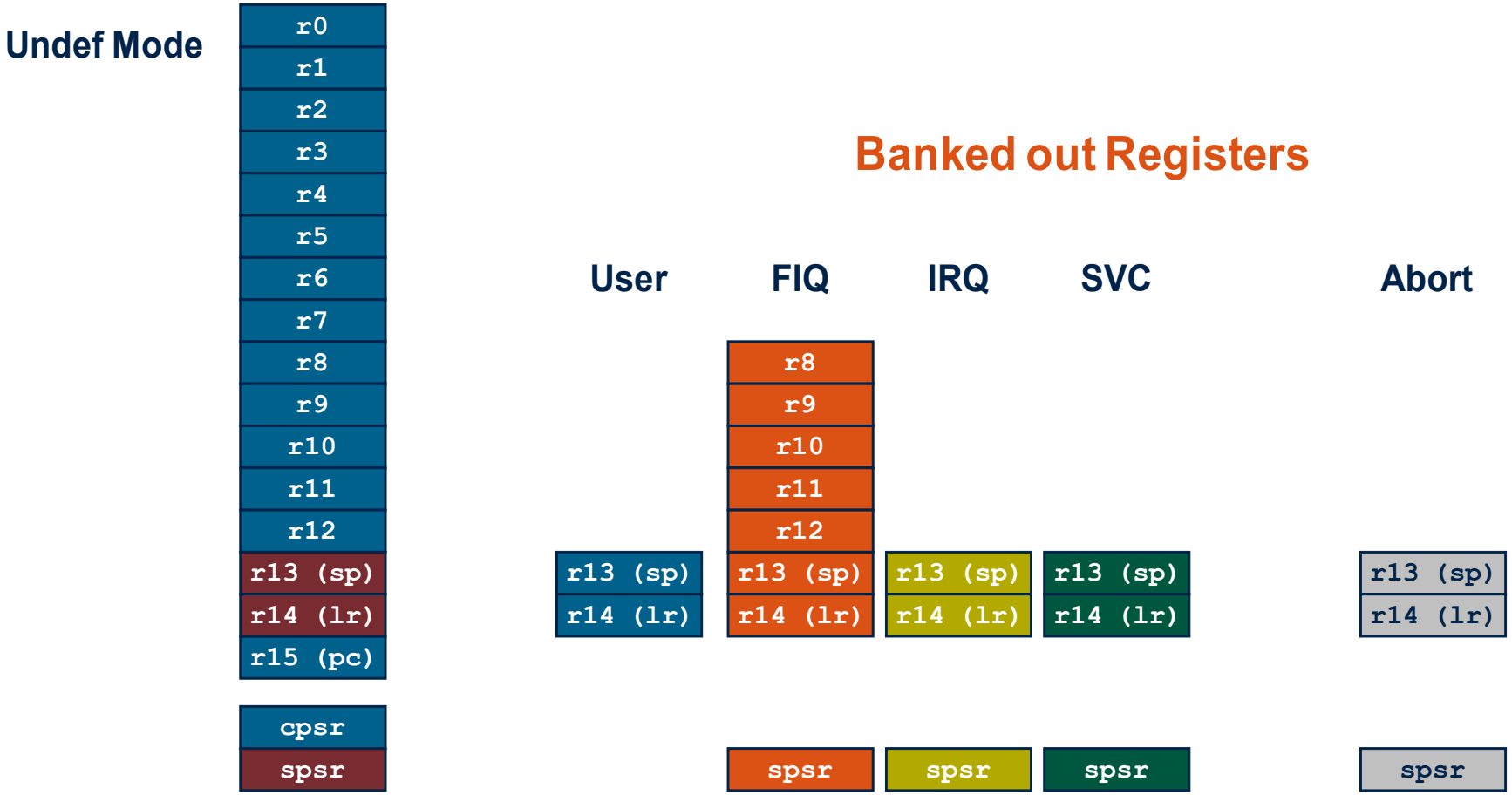
Programming Model – registers available in Abort mode

Current Visible Registers

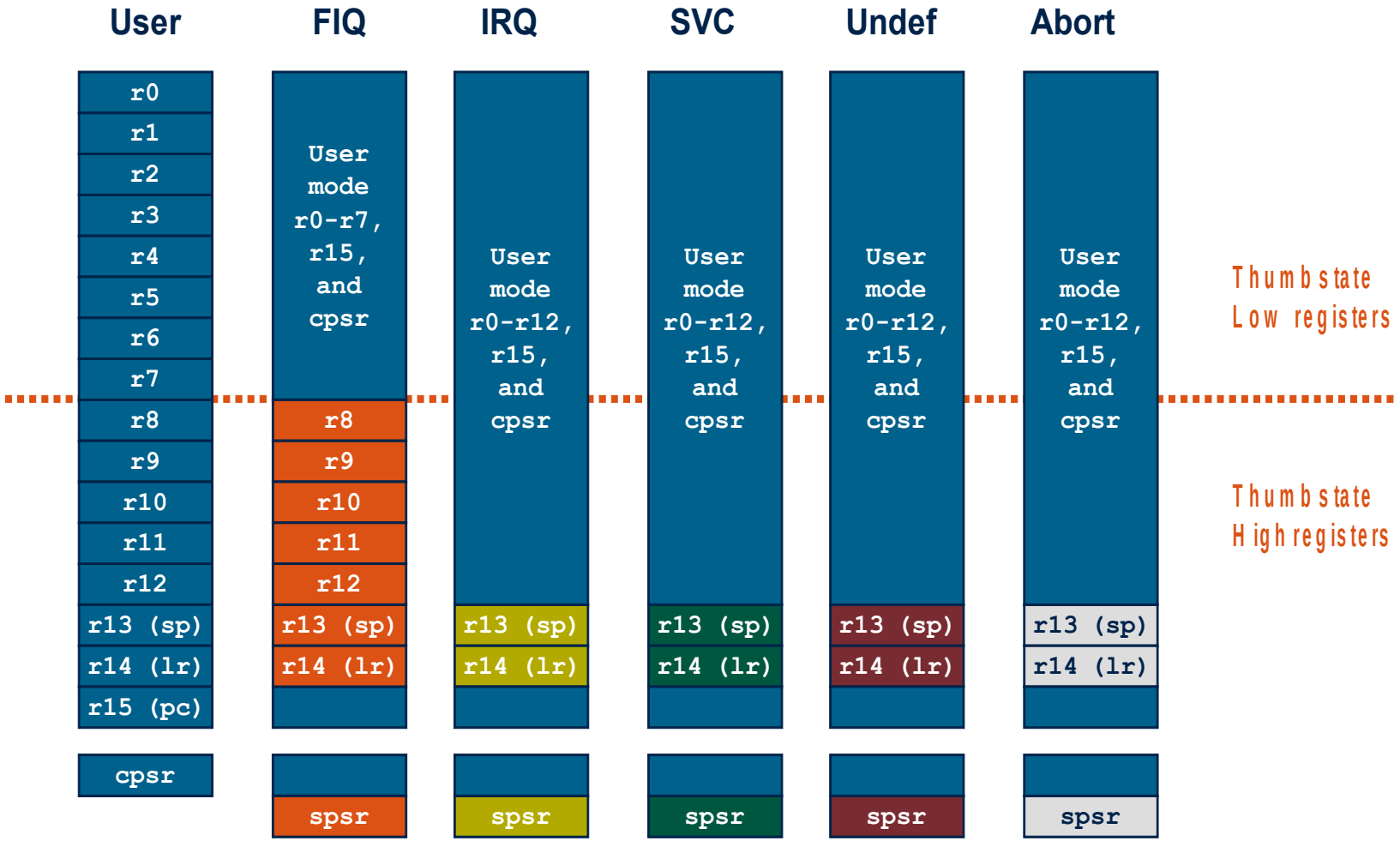


Programming Model – registers available in Undef mode

Current Visible Registers



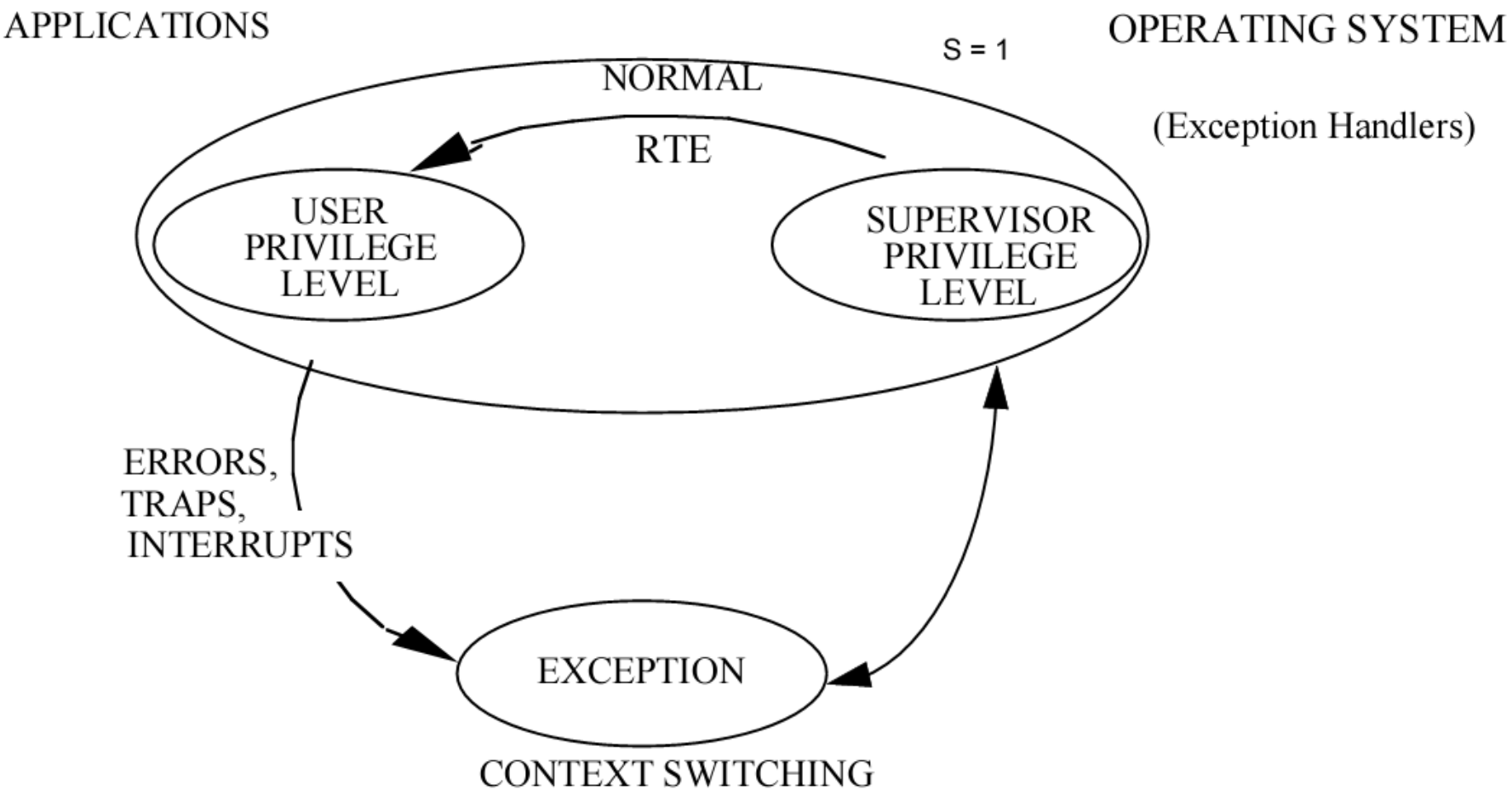
Programming Model – registers summary



Note: System mode uses the User mode register set

Interrupts and Exceptions

Handling of Exceptions



Exceptions

Exception – mechanism that control flow of data used in microprocessors-based systems and programming languages to handling asynchronous and unpredictable situations.

Exceptions can be divided into:

- Faults,
- Aborts,
- Traps.

In addition to exceptions processor supervises also interrupts.

ARM processors can handle two different modes of interrupts:

- **FIQ** - Fast interrupt (interrupt with low latency handling),
- **IRQ** - Normal Interrupt.

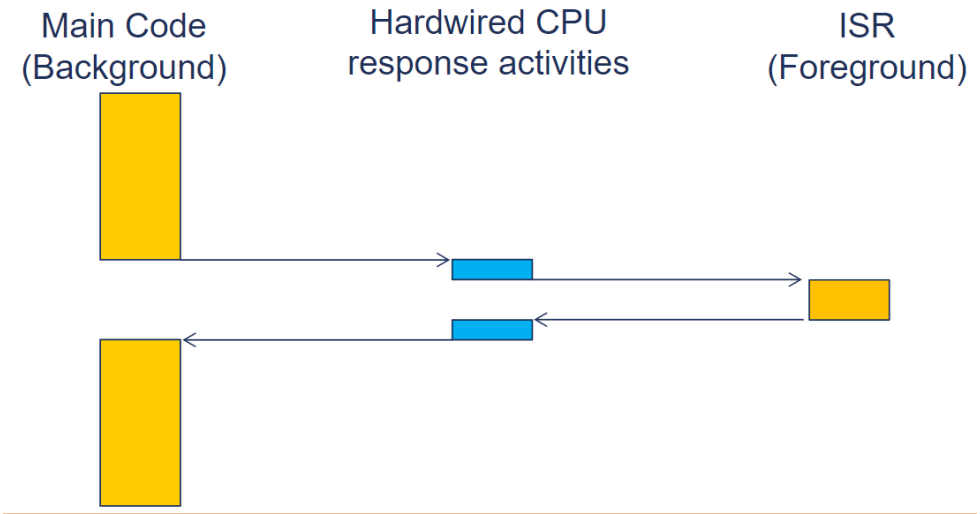
Interrupts

- **Hardware-triggered asynchronous software routine**
  - Triggered by hardware signal from peripheral or external device
  - Asynchronous - can happen anywhere in the program (unless interrupt is disabled)
  - Software routine - Interrupt Service Routine (ISR) runs in response to interrupt

- **Fundamental mechanism of microcontrollers**
  - Provides efficient event-based processing rather than polling
  - Provides quick response to events regardless* of program state, complexity, location
  - Allows many multithreaded embedded systems to be responsive without an operating system (specifically task scheduler)

Interrupts Processing Sequence

- Main code is running (background)
- Interrupt is triggered
- Processor executes context switching to ISR
- Processor executed ISR (foregroung), including return from ISR at the end
- Processor resumes execution of the main code



Interrupts

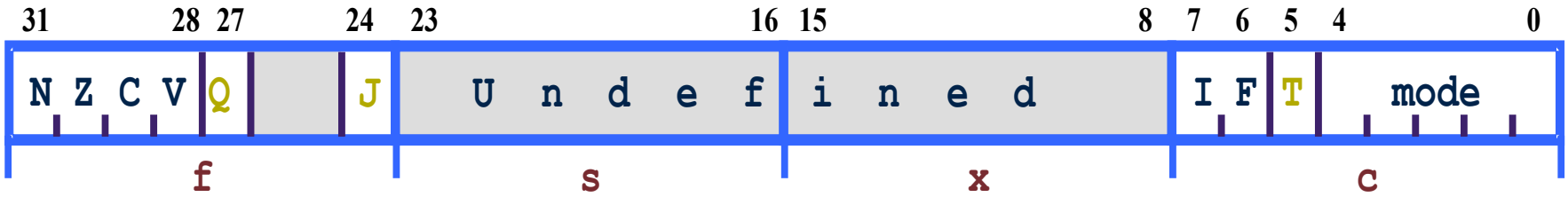
Interrupt or IRQ – **Interrupt ReQuest** – is an asynchronous signal indicating the need for attention or a synchronous event in software indicating the need for a change in execution. A hardware interrupt causes the processor to save its state of execution and begin execution of an interrupt handler. Software interrupts are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven.

Examples of interrupts:

- Receive or transmission of data via serial interface (e.g. EIA RS232),
- Change of state or detected slope on processor's pin.

Status of device can be checked using software commands, however it requires continuous reading and checking of status register of the device. This operation is called polling. Even simple polling usually requires a significant amount of processing power and unnecessary loads processor, e.g. transmission of single symbol lasts ~100 us (processor can execute hundreds of thousands of instructions during this time).

Program Status Register



Condition code flags

- V – ALU operation o**V**erflowed
- C – ALU operation **C**arried out
- Z – **Z**ero result from ALU operation
- N – **N**egative result from ALU operation

Flags for processor from family 5TE/J

- J – Processor in Jazelle mode
- Q – Sticky Overflow – saturation flag, set during ALU operations (QADD, QDADD, QSUB or QDSUB, or operation of SMLAxy, SMLAWx, result more than 32 bits)

Interrupt disable bits

- I=1 Disables the IRQ
- F=1 Disables the FIQ

Flags for xT architecture

- T=0 Processor in ARM mode
- T=1 Processor in Thumb mode

Mode bits

- Specify the processor operation mode (seven modes)

Handling of exceptions

Execution of not allowed operation in given processor mode can cause exception, e.g. access to protected memory segment.

Handling of exception covers all operations when the exception was detected until the first command of exception handler.

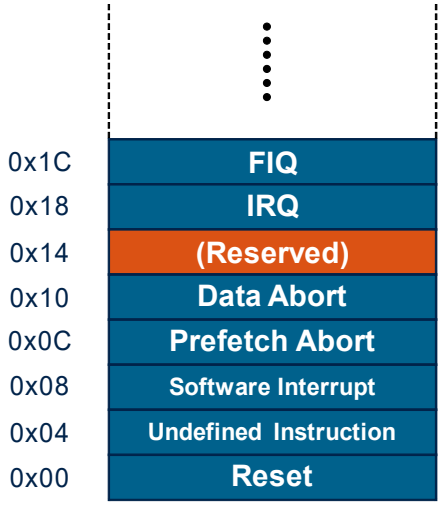
0. Finish current instruction
1. a) Change operating mode to ARM (from Thumb or Jazelle),
   b) Change to interrupt of exception mode (FIQ/IRQ),
   c) Set interrupt level mask on level equal to the handling interrupt (disable interrupts).
   d) Change registers bank:
      make a copy of CPSR → SPSR and PC (r15) → Link Register (r14),
   e) Make active SPSR register.
2. Calculate exception vector (interrupt).
3. Branch to the first instruction handling exception or interrupt.
4. Return from exception/interrupt:
   a) Recover CPSR (r15) register,
   b) Recover PC (Link Register r14),
   c) Return to the interrupted program.

Exceptions (1)

Exception handling by the ARM processor is controlled through the use of an area of memory called the vector table. This lives (normally) at the bottom of the memory map from 0x0 to 0x1c. Within this table one word is allocated to each of the various exception types. This word will contain some form of ARM instruction that should perform a branch. It does **not** contain an address.

When one of these exceptions is taken, the ARM goes through a low-overhead sequence of actions in order to invoke the appropriate exception handler. The current instruction is always allowed to complete (except in case of Reset).

IRQ is disabled on entry to all exceptions; FIQ is also disabled on entry to Reset and FIQ.



Memory image

Exceptions (2)

Reset - executed on power on

Undef - when an invalid instruction reaches the execute stage of the pipeline

SWI - when a software interrupt instruction is executed

Prefetch - when an instruction is fetched from memory that is invalid for some reason, if it reaches the execute stage then this exception is taken

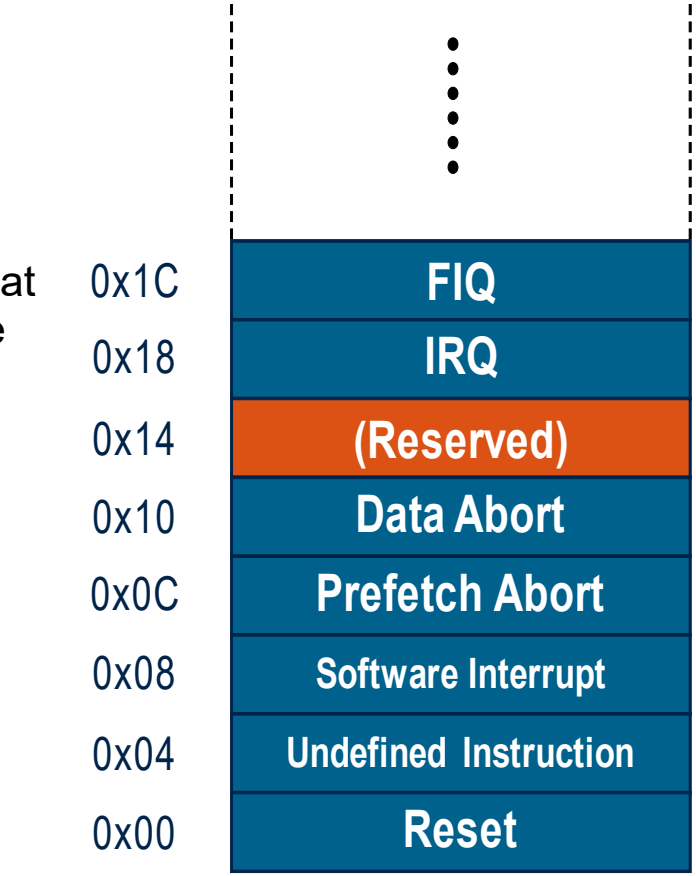
Data - if a load/store instruction tries to access an invalid memory location, then this exception is taken

IRQ - normal interrupt

FIQ - fast interrupt

Vector table is located in memory address 0x0.

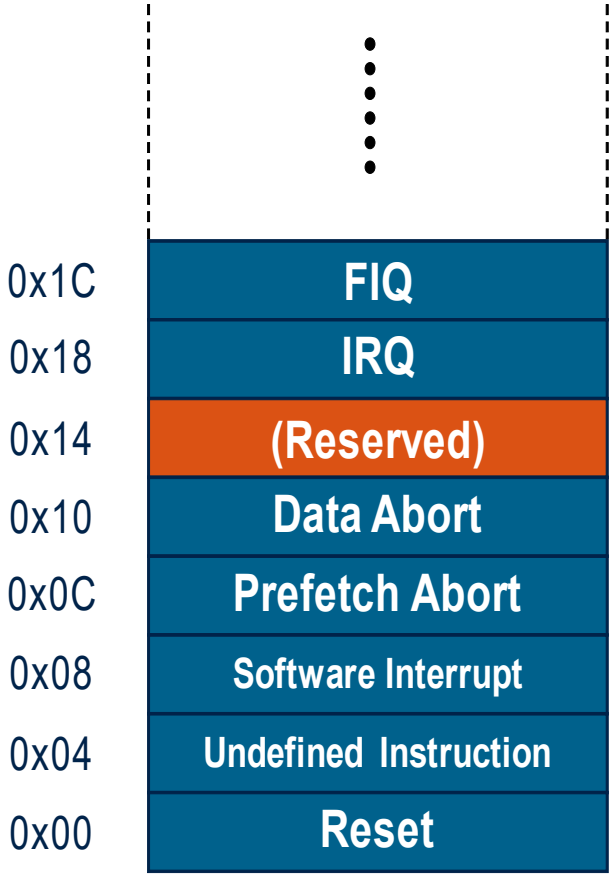
The base address of exception table can be modified: 0xFFFF.0000 (ARM 7/9/10).



Memory image

Exceptions Table

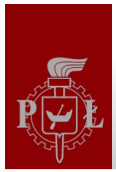
```
LDR     PC, =FIQ_Addr
LDR     PC, =IRQ_Addr
NOP ; Reserved vector
LDR     PC, =Abort_Addr
LDR     PC, =Prefetch_Addr
LDR     PC, =SWI_Addr
LDR     PC, =Undefined_Addr
LDR     PC, =Reset_Addr
```



Memory image

Exception Handlers (1)

```
IRQ_Addr:
/*- Manage Exception Entry */
/*- Adjust and save LR_irq in IRQ stack */
        sub      lr, lr, #4
        stmfd    sp!, {lr}
/*- Save r0 and SPSR in IRQ stack */
        mrs      r14, SPSR
        stmfd    sp!, {r0,r14}
/*- Write in the IVR to support Protect Mode */
/*- No effect in Normal Mode */
/*- De-assert the NIRQ and clear the source in Protect Mode */
        ldr      r14, =AT91C_BASE_AIC
        ldr      r0 , [r14, #AIC_IVR]
        str      r14, [r14, #AIC_IVR]
...
/*- Branch to the routine pointed by the AIC_IVR */
        mov      r14, pc
        bx       r0                          /* Branch to IRQ handler */
...
/*- Restore adjusted  LR_irq from IRQ stack directly in the PC */
        ldmia    sp!, {pc}^                  /* ^ - Recover CSPR */
```



Exception Handlers (2)

```
/* lowlevel.c */
/*-----------------------------------------------------------------------
 * Function Name       : default_spurious_handler
 * Object              : default handler for spurious interrupt
 *-----------------------------------------------------------------------*/
void default_spurious_handler(void)
{
    dbgu_print_ascii("-F- Spurious Interrupt\n\r ");
    while (1);
}

/*-----------------------------------------------------------------------
 * Function Name       : default_fiq_handler
 * Object              : default handler for fast interrupt
 *-----------------------------------------------------------------------*/
void default_fiq_handler(void)
{
    dbgu_print_ascii("-F- Unexpected FIQ Interrupt\n\r ");
    while (1);
}
```

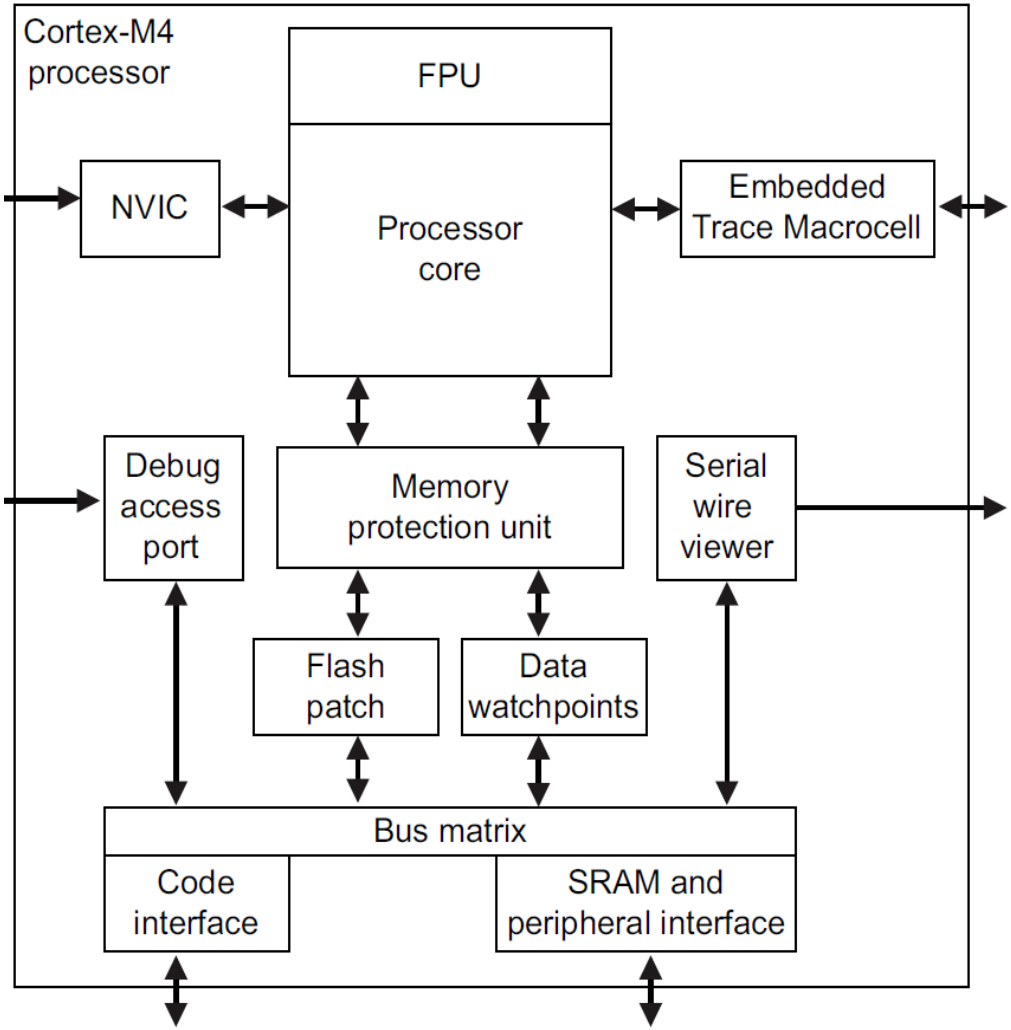
Nested Vectored Interrupt Controller (NVIC)

Chapter 13

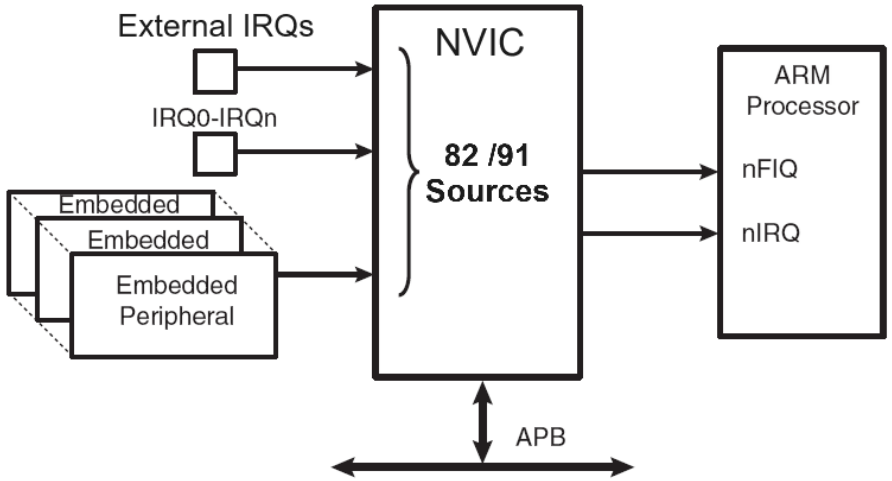
STM32 Cortex-M4 implementation

Cortex-M4 processor

- Built on a high-performance Cortex-M4 processor core
- 3-stage pipeline
- Harvard architecture,
- IEEE754- compliant single-precision floating-point computation
- Single-cycle and SIMD multiplication and multiply-with-accumulate capabilities, saturating arithmetic and dedicated hardware division
- Power control optimization of system components
- Integrated sleep modes for low power consumption



Block diagram of NVIC of ARM processor



- Manages vectorised interrupts (up to 240 IRQs)
- Can monitor up to 82 or 91 (for STM32L496xx) internal and external interrupts,
- Each interrupt can be disabled/enabled (masked),
- 16 priority levels (0 – the highest, 15 – the lowest, 4 bits), default 0
- Handles interrupts triggered with level or edge,
- Dynamic reprioritization of interrupts
- Low-latency exception and interrupt handling
- Power management control.

Nested Vectored Interrupt Controller of ARM processor

- NVIC uses system clock, however the clock signal cannot be disabled to save power.
- Interrupts can be used to wake up processor from sleep or hibernation mode.
- 11 exceptions (Priorities: -3..0..6)
- 81 or 91 interrupts assigned to peripheral devices (Priorities: 7 – 97)
- Exceptions marked with negative numbers
- Interrupts with positive numbers

Vectored Interrupts

- **Vectored interrupts** - type of interrupts that allows user to modify the Interrupt Handler (function) that is assigned to an **interrupt vector**.
- This is a number (ID) that identifies a particular interrupt handler. This **vector** may be fixed, configurable (using jumpers or switches), or programmable.
- When the **interrupt handler** is registered **its address** is saved as a **interrupt vector in a table**.
- When the device interrupts, the system enters the interrupt acknowledge cycle, asking the interrupting device to identify itself. The device responds with its interrupt vector. The system then uses this vector to find the responsible interrupt handler.
- Interrupts has assigned ID numbers and assigned interrupt vector (address of interrupt handler).

Exception Types (1)

Exception number[1]	IRQ number[1]	Exception type	Priority	Vector address or offset[2]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable[3]	0x00000010	Synchronous
5	-11	Bus fault	Configurable[3]	0x00000014	Synchronous when precise Asynchronous when imprecise
6	-10	Usage fault	Configurable[3]	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable[3]	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable[3]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[3]	0x0000003C	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable[4]	0x00000040 and above[5]	Asynchronous

Vectors Table

Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
.		.	.
.		.	.
.		.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

MS30018V1

Exception Types (2)

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-1	fixed	HardFault	All classes of fault	0x0000 000C
-	0	settable	MemManage	Memory management	0x0000 0010
-	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
-	-	-	-	Reserved	0x0000 001C - 0x0000 0028
-	3	settable	SVCall	System service call via SWI instruction	0x0000 002C
-	4	settable	Debug	Monitor	0x0000 0030
-	-	-	-	Reserved	0x0000 0034
-	5	settable	PendSV	Pendable request for system service	0x0000 0038
-	6	settable	SysTick	System tick timer	0x0000 003C

Exception Types (3)

Reset	Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode.
NMI	A *NonMaskable Interrupt* (NMI) can be signalled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2. NMIs cannot be: • Masked or prevented from activation by any other exception • Preempted by any exception other than Reset.
Hard fault	A hard fault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. Hard faults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority.
Memory management fault	A memory management fault is an exception that occurs because of a memory protection related fault. The MPU or the fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is used to abort instruction accesses to *Execute Never* (XN) memory regions.

Exception Types (4)

Bus fault

A bus fault is an exception that occurs because of a memory related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system.

Usage fault

A usage fault is an exception that occurs in case of an instruction execution fault. This includes:

- An undefined instruction
- An illegal unaligned access
- Invalid state on instruction execution
- An error on exception return.

The following can cause a usage fault when the core is configured to report it:

- An unaligned address on word and halfword memory access
- Division by zero

SVCall

A *supervisor call* (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.

PendSV

PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

SysTick

A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

Interrupt (IRQ)

An interrupt, or IRQ, is an exception signalled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

Peripheral Devices Interrupts

Position	Priority	Type of priority	Acronym	Description	Address
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	PVD_PVM	PVD/PVM1/PVM2/PVM3/PVM4 through EXTI lines 16/35/36/37/38 interrupts	0x0000 0044
2	9	settable	RTC_TAMP_STAMP /CSS_LSE	RTC Tamper or TimeStamp /CSS on LSE through EXTI line 19 interrupts	0x0000 0048
3	10	settable	RTC_WKUP	RTC Wakeup timer through EXTI line 20 interrupt	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable	DMA1_CH1	DMA1 channel 1 interrupt	0x0000 006C

Peripheral Devices Interrupts

Position	Priority	Type of priority	Acronym	Description	Address
63	70	settable	DFSDM1_FLT2	DFSDM1_FLT2 global interrupt	0x0000 013C
64	71	settable	COMP	COMP1/COMP2 through EXTI lines 21/22 interrupts	0x0000 0140
65	72	settable	LPTIM1	LPTIM1 global interrupt	0x0000 0144
66	73	settable	LPTIM2	LPTIM2 global interrupt	0x0000 0148
67	74	settable	OTG_FS	OTG_FS global interrupt	0x0000 014C
68	75	settable	DMA2_CH6	DMA2 channel 6 interrupt	0x0000 0150
69	76	settable	DMA2_CH7	DMA2 channel 7 interrupt	0x0000 0154
70	77	settable	LPUART1	LPUART1 global interrupt	0x0000 0158
71	78	settable	QUADSPI	QUADSPI global interrupt	0x0000 015C
72	79	settable	I2C3_EV	I2C3 event interrupt	0x0000 0160
73	80	settable	I2C3_ER	I2C3 error interrupt	0x0000 0164

NVIC Registers (1)

Complex 32-bit registers including:

- **Interrupt set-enable register (NVIC_ISERx) x=0-7**
- Interrupt clear-enable register (NVIC_ICERx) x=0-7
- Interrupt set-pending register (NVIC_ISPRx) x=0-7
- Interrupt clear-pending register (NVIC_ICPRx) x=0-7
- Interrupt active bit register (NVIC_IABRx) x=0-7
- **Interrupt priority register (NVIC_IPRx) x=0-59**
- Software trigger interrupt register (NVIC_STIR) x=1

NVIC Registers (2)

Offset	Register	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
0x100	NVIC_ISER0	SETENA[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x104	NVIC_ISER1	SETENA[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x11C	NVIC_ISER7	Reserved															SETENA [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x180	NVIC_ICER0	CLRENA[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x184	NVIC_ICER1	CLRENA[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x19C	NVIC_ICER7	Reserved															CLRENA [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x200	NVIC_ISPR0	SETPEND[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x204	NVIC_ISPR1	SETPEND[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x21C	NVIC_ISPR7	Reserved															SETPEND [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

NVIC Registers (3)

Offset	Register	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
0x21C	NVIC_ISPR7	Reserved															SETPEND [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x280	NVIC_ICPR0	CLRPEND[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x284	NVIC_ICPR1	CLRPEND[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x29C	NVIC_ICPR7	Reserved															CLRPEND [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x300	NVIC_IABR0	ACTIVE[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Interrupt Set-Enable Register

For LPUART we use:

- **NVIC_ISER[2] = 2^6 => 70 (2x 32 + 6) => 0-31, 32-63, 64-70 (70 is bit 6, 0x40)**

Address offset: 0x100 + 0x04 * x, (x = 0 to 7)

Reset value: 0x0000 0000

Required privilege: Privileged

NVIC_ISER0 bits 0 to 31 are for interrupt 0 to 31, respectively

NVIC_ISER1 bits 0 to 31 are for interrupt 32 to 63, respectively

....

NVIC_ISER6 bits 0 to 31 are for interrupt 192 to 223, respectively

NVIC_ISER7 bits 0 to 15 are for interrupt 224 to 239, respectively

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

Bits 31:0 **SETENA**: Interrupt set-enable bits.

Write:

0: No effect

1: Enable interrupt

Read:

0: Interrupt disabled

1: Interrupt enabled.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

Bits 16 to 31 of the NVIC_ISER7 register are reserved.

Interrupt Priority Register

For LPUART we use:

- **NVIC_IPR[70] = 0 (set priority to 0),**

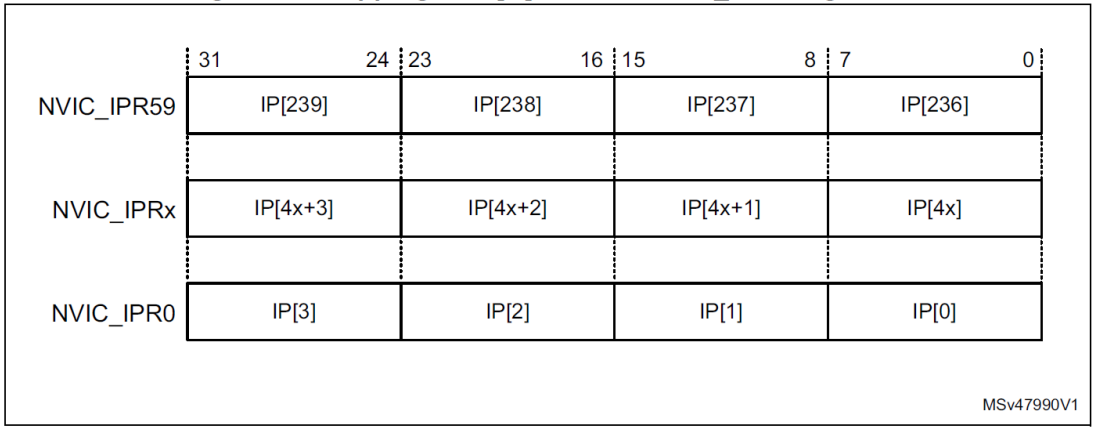
Address offset: 0x400 + 0x04 * x, (x = 0 to 59)

Reset value: 0x0000 0000

Required privilege: Privileged

The NVIC_IPRx (x = 0 to 59) byte-accessible registers provide 8-bit priority fields IP[N] (N = 0 to 239) for each of the 240 interrupts. Every register holds four IP[N] fields of the CMSIS interrupt priority array, as shown in *Figure 19*.

Figure 19. Mapping of IP[N] fields in NVIC_IPRx registers

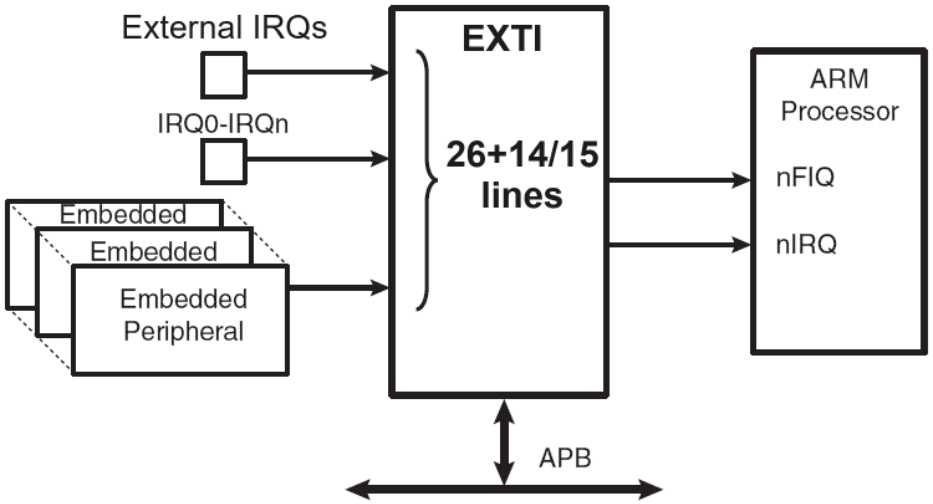


The following table shows the bit assignment of any NVIC_IPRx register. Each IP[N] field order can be expressed as N = 4 * x + byte offset.

Table 47. NVIC_IPRx bit assignment

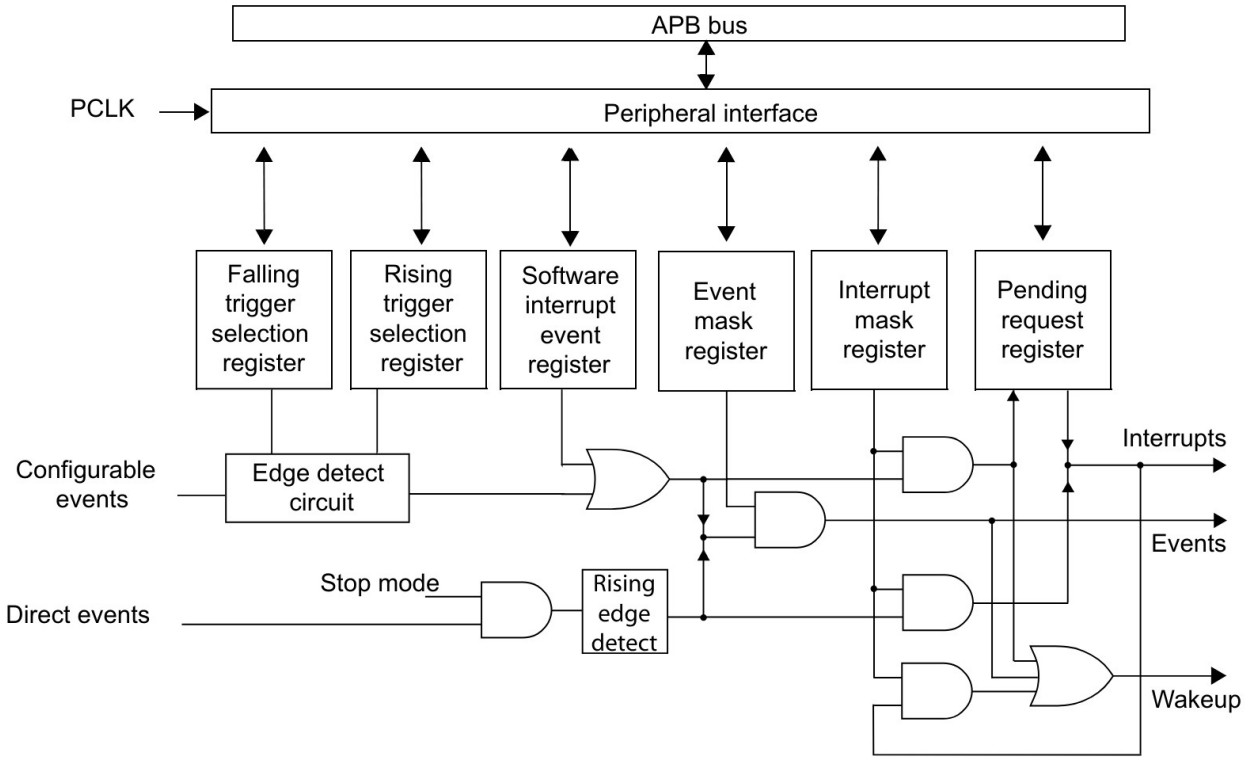
Bits	Name	Function
[31:24]	Priority, byte offset = 3	Each priority field holds a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. The processor implements only bits[7:4] of each field, bits[3:0] read as zero and ignore writes.
[23:16]	Priority, byte offset = 2	
[15:8]	Priority, byte offset = 1	
[7:0]	Priority, byte offset = 0	

Extended Interrupts and Events Controller (EXTI)



- Can monitor up to 40 or 41 (for STM32L496xx) ext. events or interrupt requests,
  - 26 configurable lines,
  - 14/15 lines with dedicated functionalities,
- Independent mask on each event/interrupt line
- Configurable rising or falling edge (configurable lines only)
- Dedicated status bit (configurable lines only)
- Emulation of event/interrupt requests (configurable lines only)
- Could wake up processor (from Stop 0 and Stop 1 modes, some from Stop 2)

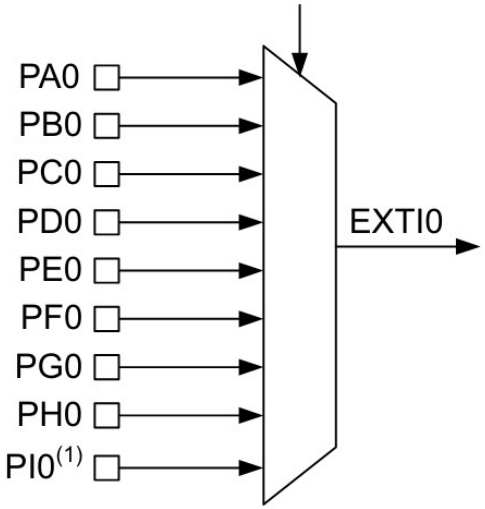
Extended Interrupts and Events Controller (2)



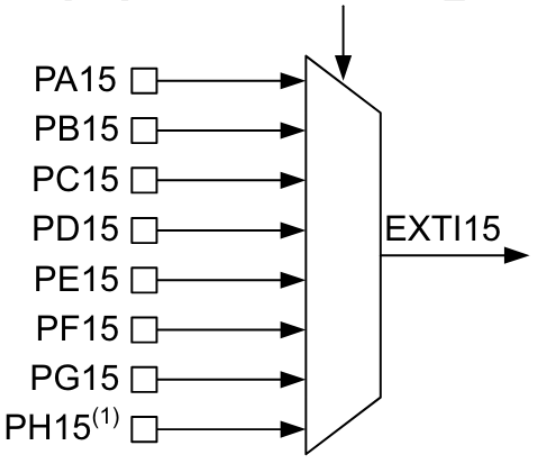
- Handles interrupts triggered with level or edge,
- Low-latency exception and interrupt handling
- Power management control.

Extended Interrupts and Events Controller (2)

EXTI0[3:0] bits in the SYSCFG_EXTICR1 register



EXTI15[3:0] bits in the SYSCFG_EXTICR4 register



- 40 or 41 interrupt/event lines are available.
- Connected to 16 configurable interrupt/event lines (EXTI0 .. EXTI15)

Extended Interrupts and Events Controller – Lines Mapping

EXTI line	Line source(1)	Line type
0-15	GPIO	configurable
16	PVD	configurable
17	OTG FS wakeup event(2) (OTG_FS_WKUP)	direct
18	RTC alarms	configurable
19	RTC tamper or timestamp or CSS_LSE	configurable
20	RTC wakeup timer	configurable
21	COMP1 output	configurable
22	COMP2 output	configurable
23	I2C1 wakeup(2)	direct
24	I2C2 wakeup(2)	direct
25	I2C3 wakeup	direct
26	USART1 wakeup(2)	direct
27	USART2 wakeup(2)	direct
28	USART3 wakeup(2)	direct

Extended Interrupts and Events Controller – Registers (1)

Offset	Register	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
0x00	EXTI_IMR1	IM31	IM30	IM29	IM28	IM27	IM26	IM25	IM24	IM23	IM22	IM21	IM20	IM19	IM18	IM17	IM16	IM15	IM14	IM13	IM12	IM11	IM10	IM9	IM8	IM7	IM6	IM5	IM4	IM3	IM2	IM1	IM0
	Reset value	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	EXTI_EMR1	EM31	EM30	EM29	EM28	EM27	EM26	EM25	EM24	EM23	EM22	EM21	EM20	EM19	EM18	EM17	EM16	EM15	EM14	EM13	EM12	EM11	EM10	EM9	EM8	EM7	EM6	EM5	EM4	EM3	EM2	EM1	EM0
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	EXTI_RTSR1	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RT22	RT21	RT20	RT19	RT18	Res.	RT16	RT15	RT14	RT13	RT12	RT11	RT10	RT9	RT8	RT7	RT6	RT5	RT4	RT3	RT2	RT1	RT0
	Reset value										0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	EXTI_FTSR1	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	FT22	FT21	FT20	FT19	FT18	Res.	FT16	FT15	FT14	FT13	FT12	FT11	FT10	FT9	FT8	FT7	FT6	FT5	FT4	FT3	FT2	FT1	FT0
	Reset value										0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	EXTI_SWIER1	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SWI22	SWI21	SWI20	SWI19	SWI18	Res.	SWI16	SWI15	SWI14	SWI13	SWI12	SWI11	SWI10	SWI9	SWI8	SWI7	SWI6	SWI5	SWI4	SWI3	SWI2	SWI1	SWI0
	Reset value										0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	EXTI_PR1	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PIF22	PIF21	PIF20	PIF19	PIF18	Res.	PIF16	PIF15	PIF14	PIF13	PIF12	PIF11	PIF10	PIF9	PIF8	PIF7	PIF6	PIF5	PIF4	PIF3	PIF2	PIF1	PIF0
	Reset value										0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Extended Interrupts and Events Controller – Registers (1)

Offset	Register	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
0x20	EXTI_IMR2	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	IM40	IM39	IM38	IM37	IM36	IM35	IM34	IM33	IM32
	Reset value																								1	1	0	0	0	0	1	1	1
0x24	EXTI_EMR2	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	EM40	EM39	EM38	EM37	EM36	EM35	EM34	EM33	EM32
	Reset value																								0	0	0	0	0	0	0	0	0
0x28	EXTI_RTSR2	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RT38	RT37	RT36	RT35	Res.	Res.	Res.
	Reset value																										0	0	0	0			
0x2C	EXTI_FTSR2	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	FT38	FT37	FT36	FT35	Res.	Res.	Res.
	Reset value																										0	0	0	0			
0x30	EXTI_SWIER2	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SWI38	SWI37	SWI36	SWI35	Res.	Res.	Res.
	Reset value																										0	0	0	0			
0x34	EXTI_PR2	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PIF38	PIF37	PIF36	PIF35	Res.	Res.	Res.
	Reset value																										0	0	0	0			

Shared Interrupts

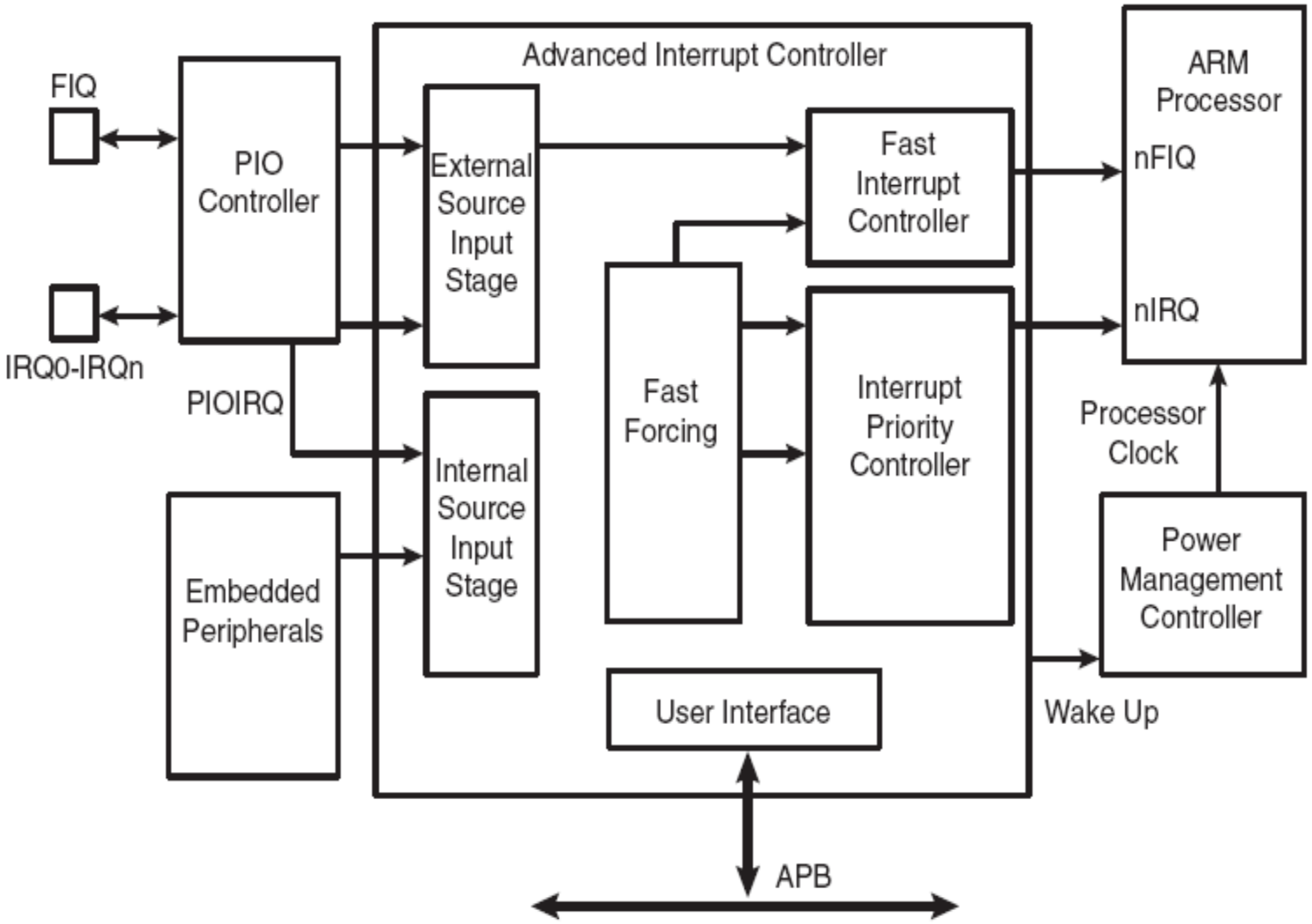
Internal peripheral devices use a single system shared interrupt SYS (number defined by constant AT91C_ID_SYS = 1).

Devices handled by system interrupt:

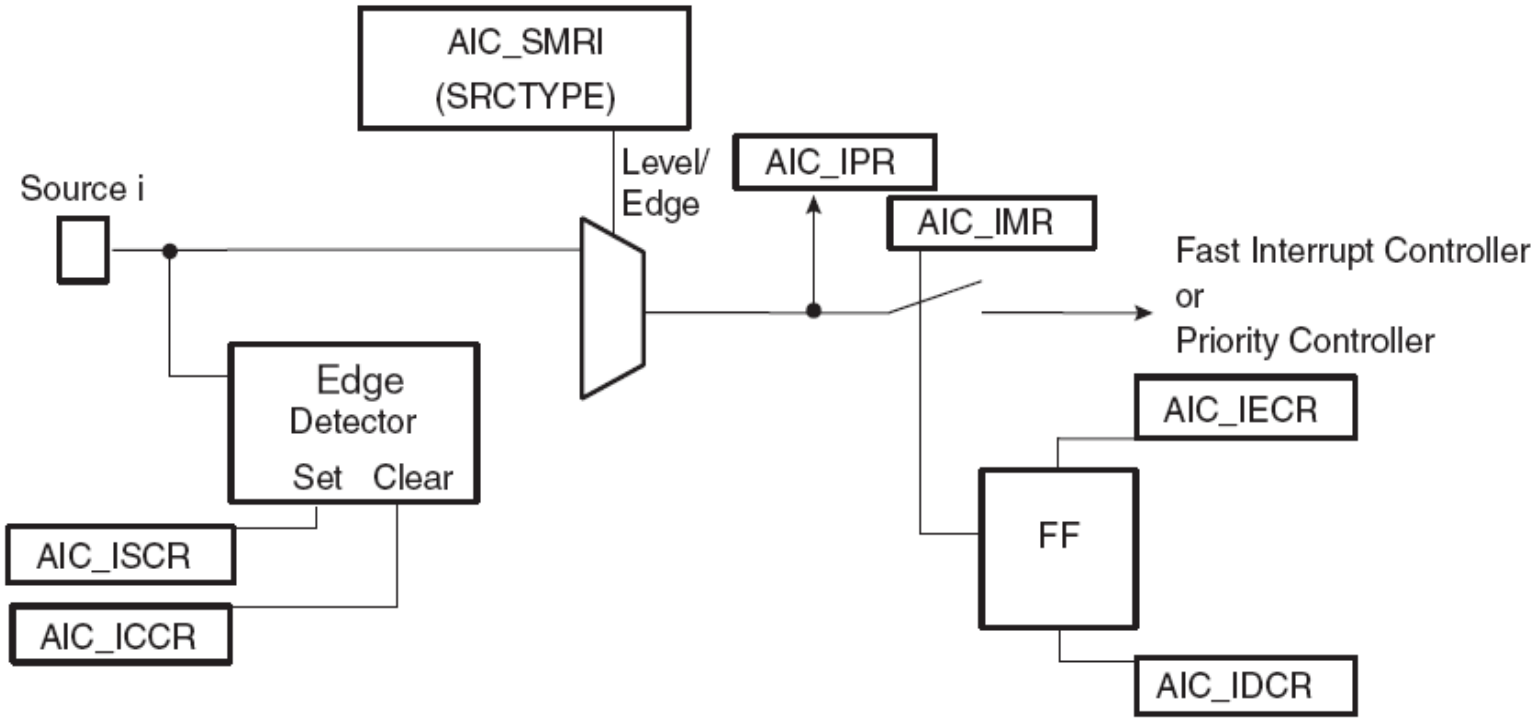
- Timers PIT, RTT, WDT,
- Diagnostic interface (DBGU),
- DMA controller (PMC),
- Reset circuit (RSTC),
- Memory Controller (MC).

Therefore, the SYS handler should check state of all interrupts and execute functions-handlers for the active interrupts (mask register AIC_MSK).

Block diagram of AIC

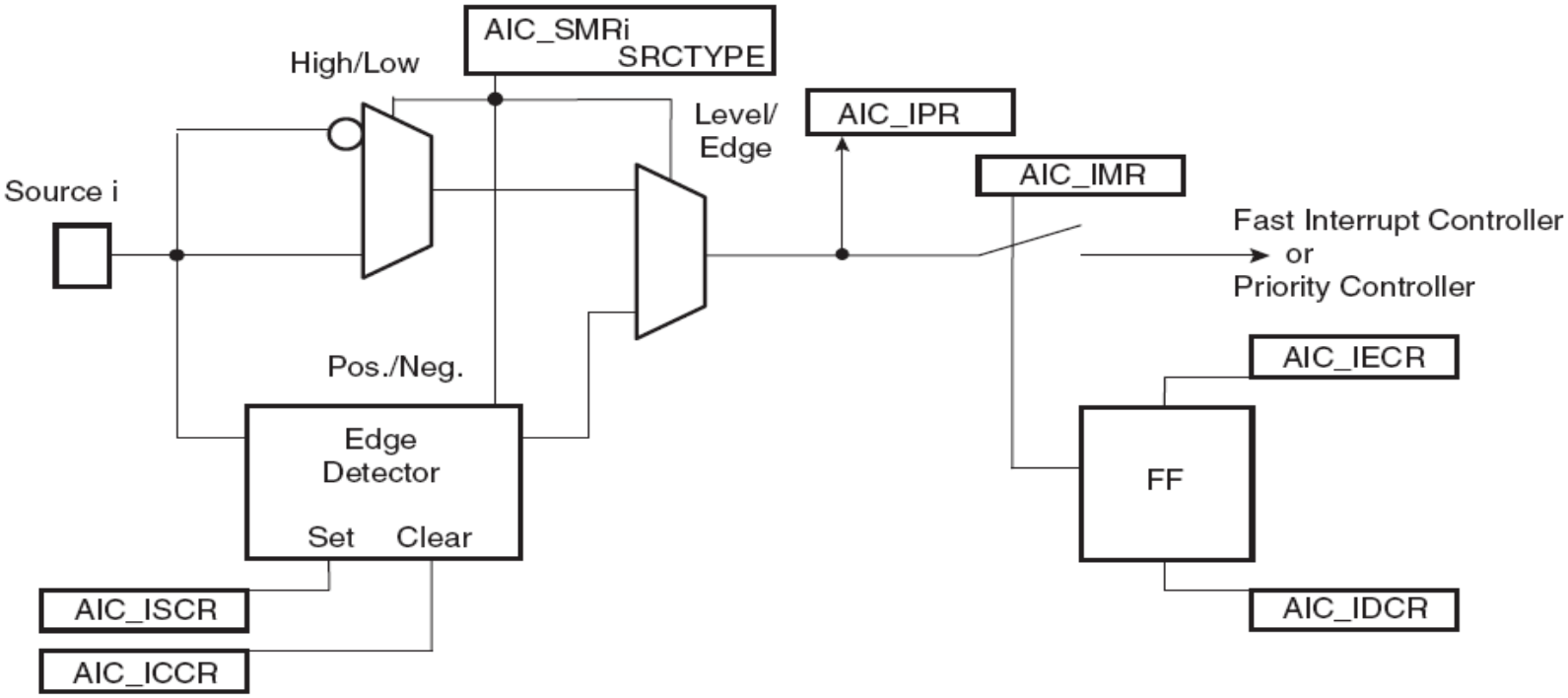


Internal Interrupts



- IRQ mask – AIC_IECR/IDCR (status → AIC_IMR),
- Clear interrupt flag when AIC_IVR register is read (for FIQ → AIC_FVR),
- Interrupt status available in AIC_IPR
- Interrupt can be triggered by high level or rising edge

External Interrupts



- User can select method of triggering: level (high, low) or edge (rising, falling)

ID Numbers for Peripheral Devices

```
// ********************************************************************************
//               PERIPHERAL ID DEFINITIONS FOR AT91SAM9263
// ********************************************************************************
#define AT91C_ID_FIQ    ( 0) // Advanced Interrupt Controller (FIQ)
#define AT91C_ID_SYS    ( 1) // System Controller
#define AT91C_ID_PIOA   ( 2) // Parallel IO Controller A
#define AT91C_ID_PIOB   ( 3) // Parallel IO Controller B
#define AT91C_ID_PIOCDE ( 4) // Parallel IO Controller C, Parallel IO Controller D, Parallel IO Controller E
#define AT91C_ID_US0    ( 7) // USART 0
#define AT91C_ID_US1    ( 8) // USART 1
#define AT91C_ID_US2    ( 9) // USART 2
#define AT91C_ID_MCI0   (10) // Multimedia Card Interface 0
#define AT91C_ID_MCI1   (11) // Multimedia Card Interface 1
#define AT91C_ID_CAN    (12) // CAN Controller
#define AT91C_ID_TWI    (13) // Two-Wire Interface
#define AT91C_ID_SPI0   (14) // Serial Peripheral Interface
```

ID=0, ID=30-31 external interrupts, others are internal

Registers of AIC (1)

Offset	Register	Name	Access	Reset
0x00	Source Mode Register 0	AIC_SMR0	Read-write	0x0
0x04	Source Mode Register 1	AIC_SMR1	Read-write	0x0
---	---	---	---	---
0x7C	Source Mode Register 31	AIC_SMR31	Read-write	0x0
0x80	Source Vector Register 0	AIC_SVR0	Read-write	0x0
0x84	Source Vector Register 1	AIC_SVR1	Read-write	0x0
---	---	---	---	---
0xFC	Source Vector Register 31	AIC_SVR31	Read-write	0x0
0x100	Interrupt Vector Register	AIC_IVR	Read-only	0x0
0x104	FIQ Interrupt Vector Register	AIC_FVR	Read-only	0x0
0x108	Interrupt Status Register	AIC_ISR	Read-only	0x0
0x10C	Interrupt Pending Register(3)	AIC_IPR	Read-only	0x0(1)
0x110	Interrupt Mask Register(3)	AIC_IMR	Read-only	0x0
0x114	Core Interrupt Status Register	AIC_CISR	Read-only	0x0
0x118 - 0x11C	Reserved	---	---	---
0x120	Interrupt Enable Command Register(3)	AIC_IECR	Write-only	---
0x124	Interrupt Disable Command Register(3)	AIC_IDCR	Write-only	---
0x128	Interrupt Clear Command Register(3)	AIC_ICCR	Write-only	---
0x12C	Interrupt Set Command Register(3)	AIC_ISCR	Write-only	---
0x130	End of Interrupt Command Register	AIC_EOICR	Write-only	---
0x134	Spurious Interrupt Vector Register	AIC_SPU	Read-write	0x0
0x138	Debug Control Register	AIC_DCR	Read-write	0x0
0x13C	Reserved	---	---	---
0x140	Fast Forcing Enable Register(3)	AIC_FFER	Write-only	---
0x144	Fast Forcing Disable Register(3)	AIC_FFDR	Write-only	---
0x148	Fast Forcing Status Register(3)	AIC_FFSR	Read-only	0x0
0x14C - 0x1E0	Reserved	---	---	---
0x1EC - 0x1FC	Reserved			

Registers of AIC – mapped as struct

```
typedef struct _AT91S_AIC {
    AT91_REG    AIC_SMR[32];    // Source Mode Register
    AT91_REG    AIC_SVR[32];    // Source Vector Register
    AT91_REG    AIC_IVR;        // IRQ Vector Register
    AT91_REG    AIC_FVR;        // FIQ Vector Register
    AT91_REG    AIC_ISR;        // Interrupt Status Register
    AT91_REG    AIC_IPR;        // Interrupt Pending Register
    AT91_REG    AIC_IMR;        // Interrupt Mask Register
    AT91_REG    AIC_CISR;       // Core Interrupt Status Register
    ...
} AT91S_AIC, *AT91PS_AIC;
#define AT91C_BASE_AIC      (AT91_CAST(AT91PS_AIC)   0xFFFFF000) // (AIC)
    Base Address
```

Registers of AIC (2)

AIC_SMR[32]; // Source Mode Register – configure method of int triggering, priority

AIC_SVR[32]; // Source Vector Register – 32-bit addresses for int handlers

AIC_IVR; // IRQ Vector Register – address of currently handled normal interrupt

AIC_FVR; // FIQ Vector Register – address of currently handled fast interrupt

AIC_ISR; // Interrupt Status Register – number of currently handled interrupt

AIC_IPR; // Interrupt Pending Register – register with pending interrupts, bits 0-31

AIC_IMR; // Interrupt Mask Register – register with masks for interrupts, bits 0-31

AIC_CISR; // Core Interrupt Status Register – status for IRQ/FIQ core interrupts

AIC_IECR; // Interrupt Enable Command Register – register for enabling interrupts

AIC_IDCR; // Interrupt Disable Command Register – register for disabling interrupts

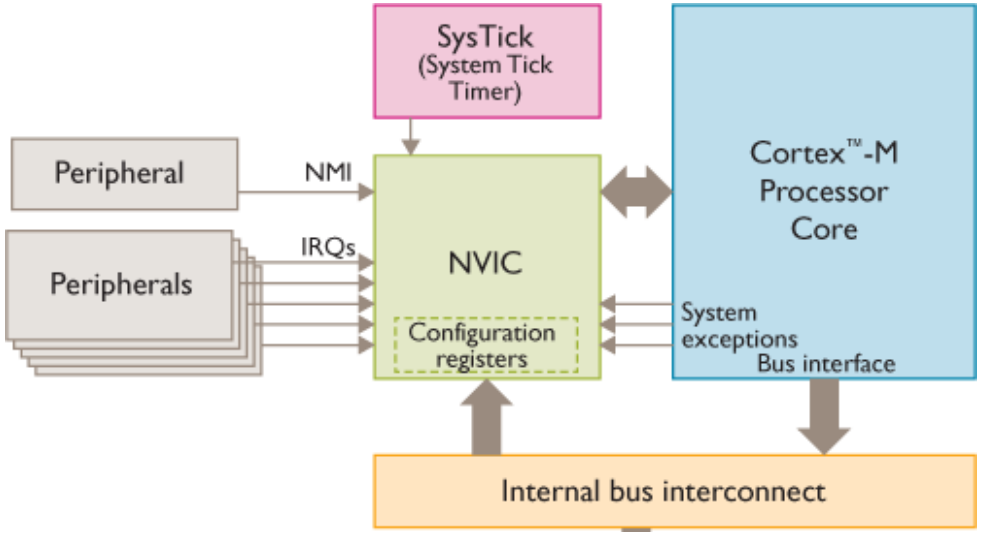
AIC_ICCR; // Interrupt Clear Command Register – register for deactivating interrupts

AIC_ISCR; // Interrupt Set Command Register – register for triggering interrupts

AIC_EOICR; // End of Interrupt Command Register – inform that INT treatment is finished

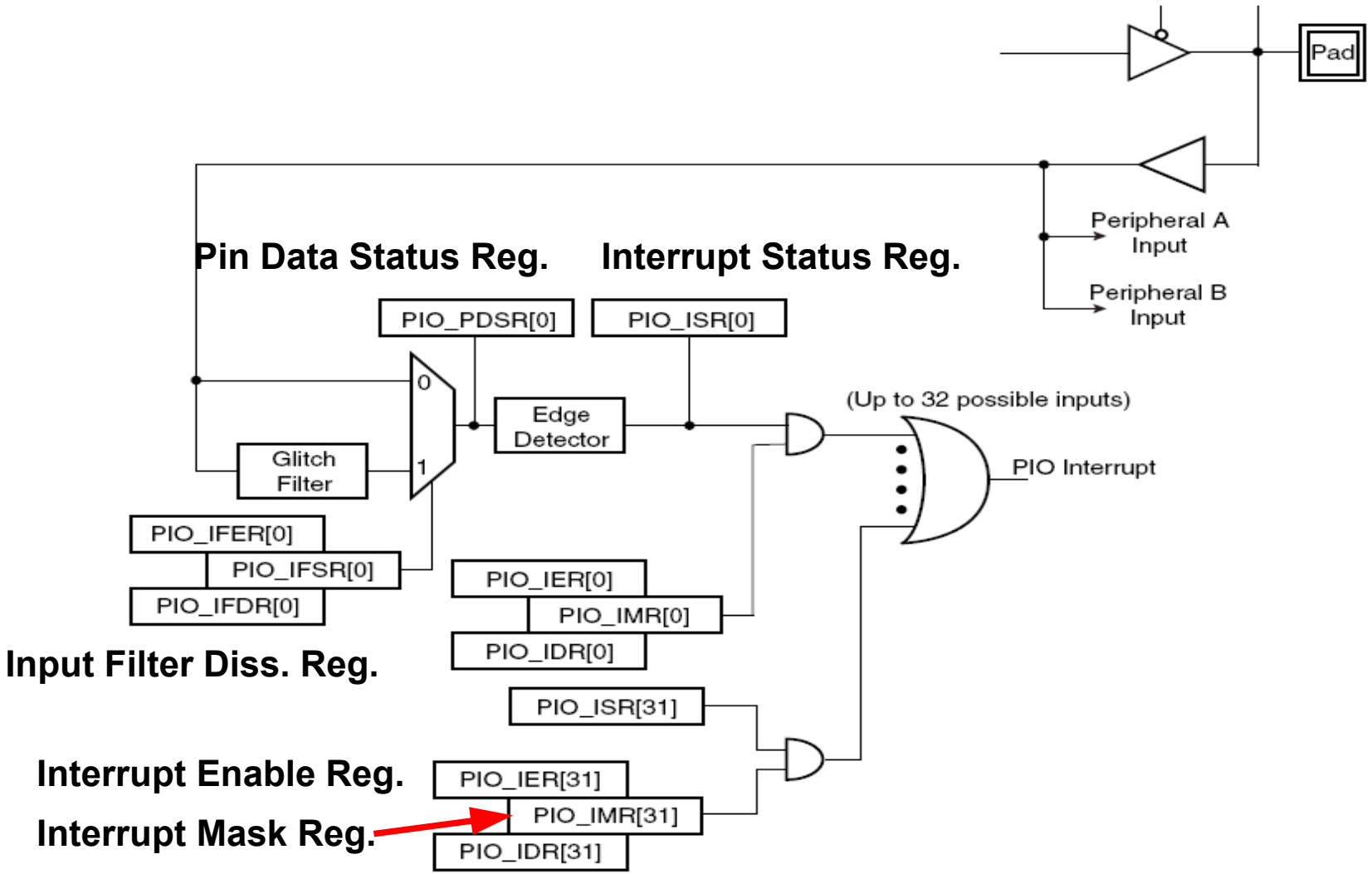
AIC_SPU; // Spurious Vector Register – handler for spurious interrupt

Nested Vectored Interrupt Controller



- NVIC manages and prioritizes interrupts,
- Can support up to 240 internal and external interrupts,
- Each interrupt can be disabled/enabled (masked),
- Handles normal nIRQ and fast nFIR interrupts,
- Handles interrupts triggered with level or edge.

I/O – Interrupts



Keyboard interrupts configuration

Buttons are connected to Port C – interrupt generated by input signals of ports C/D/E (use mask AT91C_ID_PIOCDE)

Configuration of interrupts for C/D/E port(s):

1. Configure both ports as inputs (left and right hand buttons), activate clock signal
2. **Turn off interrupts for port C/D/E (register AIC_IDCR, mask AT91C_ID_PIOCDE)**
3. Configure pointer for C/D/E port interrupt handler – use AIC_SVR table
   AIC_SVR[AT91C_ID_PIOCDE] = ...
4. Configure method of interrupt triggering: high level, (AIC_SMR register, triggered by AT91C_AIC_SRCTYPE_EXT_HIGH_LEVEL and priority, e.g. AT91C_AIC_PRIOR_HIGHEST)
5. **Clear interrupt flag for port C/D/E (register AIC_ICCR)**
6. Turn on interrupts for both input ports (register PIO_IER)
7. Turn on interrupts for C/D/E port (register AIC_IECR)

INT Handler for Keyboard

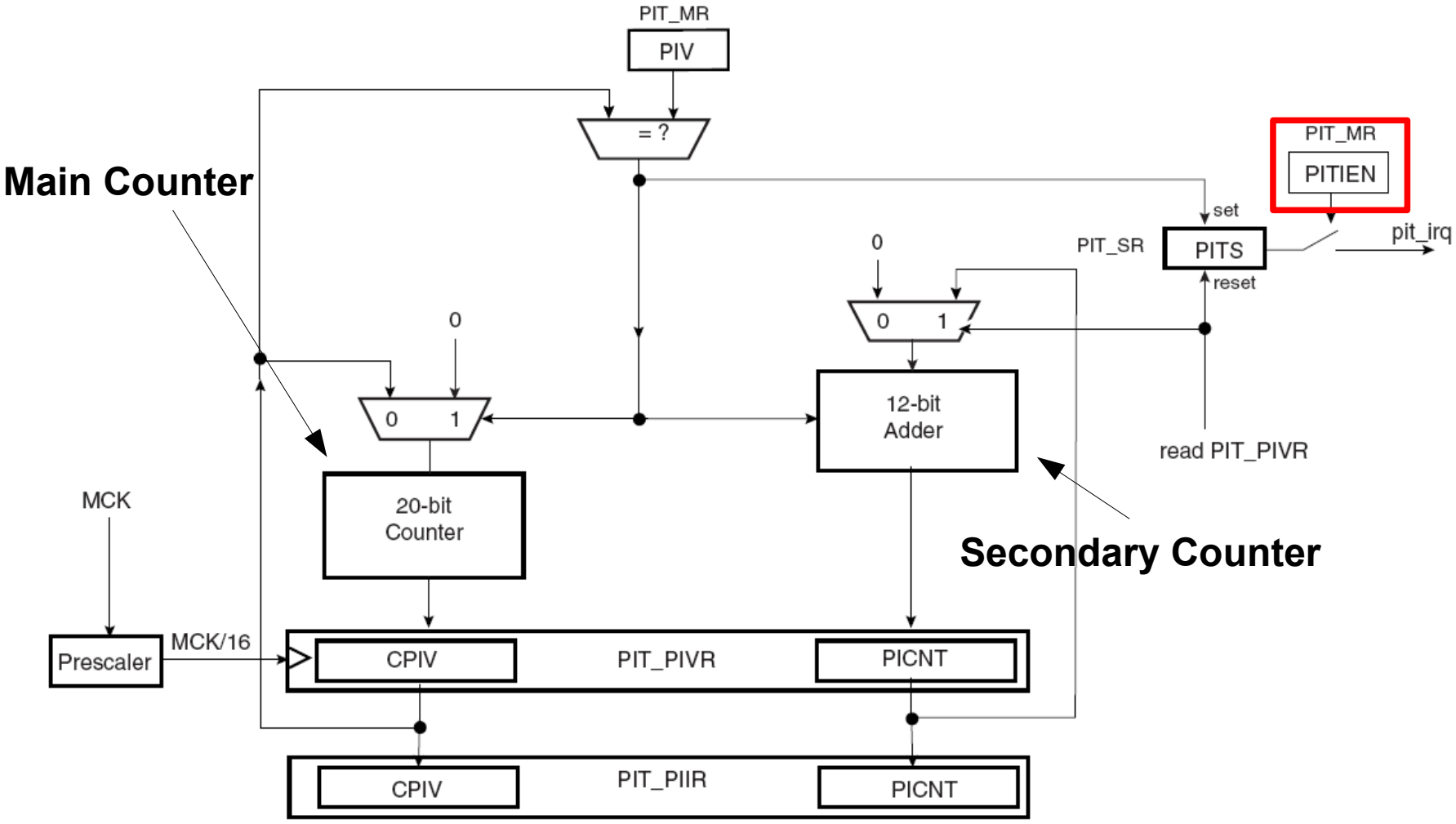
Set address for interrupt function (handler) for the interrupt (32-bits address)

AT91C_BASE_AIC->AIC_SVR[AT91C_ID_SYS] = (unsigned int) BUTTON_IRQ_handler;

Keyboard interrupt handler

```
void BUTTON_IRQ_handler (void) {
        If flag on the suitable bit-position is active the button is/was pressed (PIO_ISR)
                Read PIO_ISR status register to clear the flag
}
```

Interrupt from PIT



PIT Timer interrupts configuration

PIT Timer generates system interrupt (ID number 1) – interrupt from processor peripheral devices (System Controller, mask AT91C_ID_SYS)

Configuration of PIT Timer interrupts:

1. Calculate time counter value for defined period of time, e.g. 5 ms
2. Disable PIT Timer interrupts – only during configuration (AIC_IDCR, interrupt nr 1 – processor peripheral devices, used defined constant AT91C_ID_SYS)
3. Configure pointer for timer interrupt handler – handler for processor peripheral devices, see AIC_SVR table (AIC_SVR[AT91C_ID_SYS])
4. Configure method of interrupt triggering: level, edge, (AIC_SMR register, triggered by AT91C_AIC_SRCTYPE_INT_LEVEL_SENSITIVE, and priority, e.g. AT91C_AIC_PRIOR_LOWEST)
5. Clear interrupt flag of peripheral devices (AIC_ICCR register)
6. Turn on the interrupt AT91C_ID_SYS (AIC_IECR register)
7. Turn on PIT Timer interrupt (AT91C_PITC_PITIEN register)
8. Turn on PIT Timer (AT91C_PITC_PITEN)
9. Clear local counter (variable Local_Counter) to see if Timer triggers interrupts

INT Handler for Timer

Set address for interrupt function (handler) for the interrupt (32-bits address)

AT91C_BASE_AIC->AIC_SVR[AT91C_ID_SYS] = (unsigned int) TIMER_INT_handler;

Timer interrupt handler

```
void TIMER_INT_handler (void) {
        if flag PITIE for Timer interrupt is set (PIT_MR register)      /* interrupt enabled */
            if flag PITS in PIT_SR register is set                      /* timer requested int */
                read the PITC_PIVR register to clear PITS flag in PIT_SR
                /* delay ~100 ms */
                TimerCounter++;                                         /* LedToggle... */
        else another device requested interrupts
            check which device requested INT,
            process INT, clear INT flag,
            if unknown device, just increase counter of unknown interrupts
}
```

Interrupts from DBGU transceiver

DGBU generates system interrupt (ID number 1) – interrupt from processor peripheral devices (System Controller, mask AT91C_ID_SYS). We have distinguish which device triggered interrupt. A few interrupts can be triggered.

DGBU can generate the following interrupts:

- RXRDY: Enable RXRDY Interrupt
- TXRDY: Enable TXRDY Interrupt
- ENDRX: Enable End of Receive Transfer Interrupt
- ENDTX: Enable End of Transmit Interrupt
- OVRE: Enable Overrun Error Interrupt
- FRAME: Enable Framing Error Interrupt
- PARE: Enable Parity Error Interrupt
- TXEMPTY: Enable TXEMPTY Interrupt
- TXBUFE: Enable Buffer Empty Interrupt
- RXBUFF: Enable Buffer Full Interrupt
- COMMTX: Enable COMMTX (from ARM) Interrupt
- COMMRX: Enable COMMRX (from ARM) Interrupt

Interrupts from DBGU transceiver

DGBU interrupt handler

```
void DGBU_INT_handler (void) {
        int IntStatus;
        SysIRQCounter++;              /* to have a feeling how many system INTs are triggered */
        IntStatus = DGBU->SR;
        if (IntStatus & DBGU->IMR )   /* interrupt from DGBU */
              if INT from TxD         /* transmitter interrupt */
                    WriteNewData ();  /* be careful INTcan be also generated in case of error */
              else if INT from RxD
                    ReadDataToBuffer();/* INT can be also generated when error occur */
              else
                    other device triggered INT;
}
```

Interrupt Handlers in C (1)

Functions used as handlers require usage of preprocessor directive __attribute__ ((interrupt("IRQ")))

```
void INTButton_handler()__attribute__ ((interrupt("IRQ")));
void INTPIT_handler()__attribute__ ((interrupt("IRQ")));
void Soft_Interrupt_handler()__attribute__ ((interrupt("SWI")));
void Abort_Exception_handler()__attribute__ ((interrupt("ABORT")));
void Undef_Exception_handler()__attribute__ ((interrupt("UNDEF")));
void __irq IRQ_Handler(void)
```

Functions used as a handler is similar to normal function in C language

```
void INTButton_handler() {
      // standard C function
}
```

During laboratory we do not use __attribute__ ((interrupt("IRQ"))), we use functions provided by ATMEL, defined in startup.S file.