ZYNQ/ARM
Interrupt Controller

Events, Exceptions (faults), and Interrupts

Events are expected or unexpected signals that indicate a new situation may need immediate attention. Events include:

Fault conditions, like a processor trying to read or write non-existent memory, or trying to execute an illegal or unimplemented instruction.

Hardware-generated interrupt service requests that result from a peripheral needing urgent attention due to data needing to be transferred, or a fault condition needing to be addressed.

Software-generated interrupt service requests that result from user code requesting access to a protected resource.

Events, Exceptions (faults), and Interrupts

Some events do not require immediate attention. These events may set a status bit, or record some data, and remain “pending” (or awaiting service) until the processor gets around to checking on and servicing them.

Some events do require immediate attention, and these generate exceptions or interrupts.

- “Exceptions” generally refer to unexpected events that are generated as a result of program execution, or from hardware faults (like power-fail).

- “Interrupts” generally refer to signals asserted by peripherals that need urgent attention, or by user programs that need access to protected resources.

Interrupts

Interrupts generally arise from peripheral circuits that have produced new data that needs to be consumed in a timely fashion (i.e., before another new data point comes along), or that need new data to continue operations.

Consider a keyboard key press. If a new keypress arrives every 250ms, the ARM processor running at 600MHz can do roughly 150 million instructions between key presses.

Consider a sensor measuring motion, or light intensity, or temperature, or... Most sensors use a relatively low data rate, and so don’t need constant attention. Data typically gets to the CPU via a serial bus.

Bus controllers also commonly generate interrupts (why)?

SPI is a point-to-point serial bus that can run up to about 50MHz, for a peak data rate of around 5Mbytes/second. Most run slower, and most “burst” to get a few bytes, and then are dormant for some time.

Common uses include sensors, actuators, ADCs, DACs in embedded systems

I2C has one master and up to 1000 “slaves”, and can run up to 5MHz (most run < 100KHz). That’s a data rate of 500Kbytes to 10Kbytes per second.

Common uses include low-rate sensors, motor controllers, channel setup

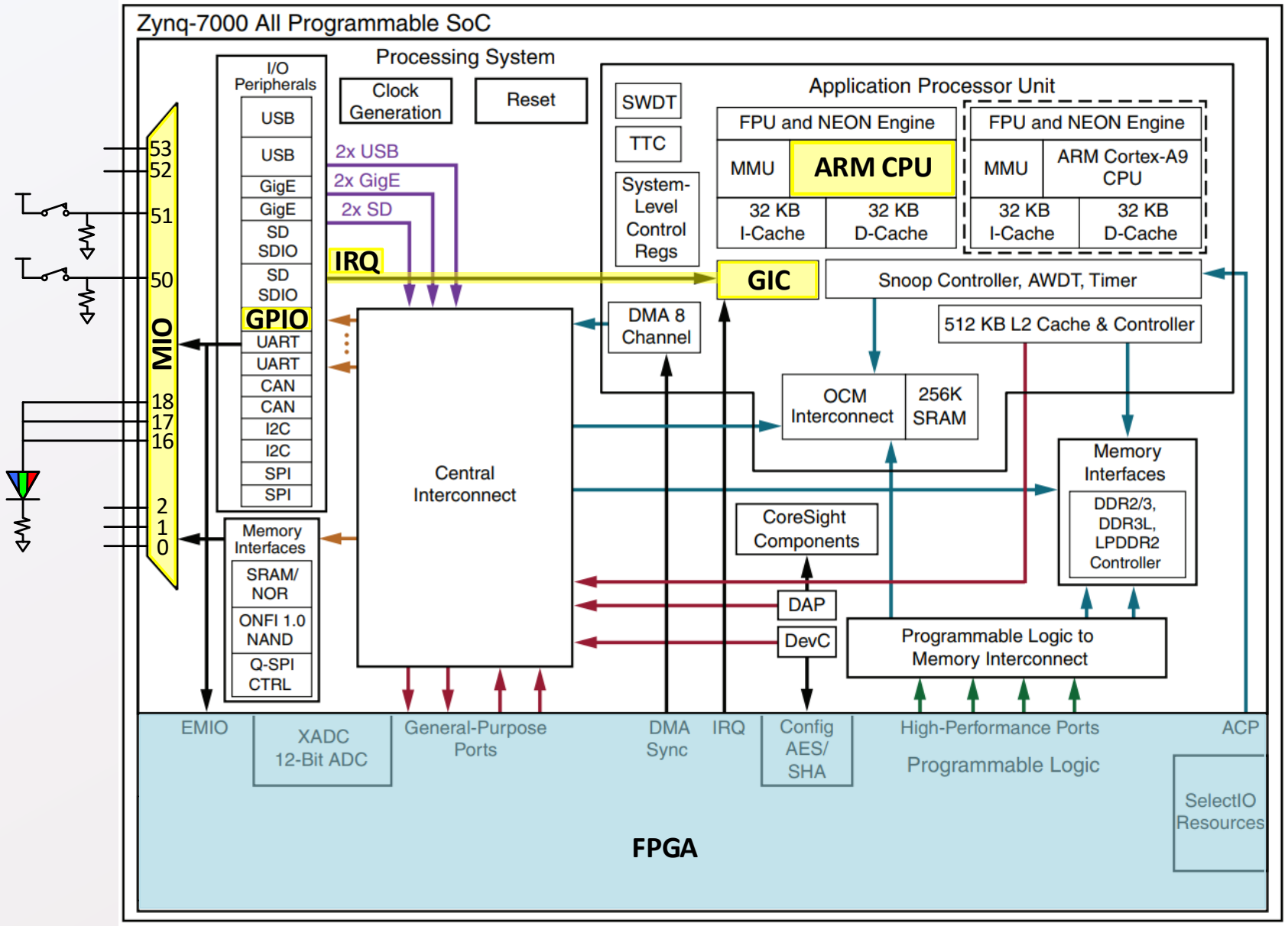
UART can run up to 200KHz, or 20KBytes/second.

These busses are discussed in an upcoming presentation...

With ZYNQ, GPIO's from the MIO pins or from the FPGA can generate interrupts.

FPGA-based IP blocks can also generate interrupts.

All on-board peripherals can generate interrupts



Fault conditions can also cause interrupts.

Faults are unexpected events that occur during instruction execution. For example, a program could attempt to read or write a non-existent memory location, or attempt to execute and unimplemented instruction.

In either case, processing cannot continue, and a context switch is required.

In many contexts, faults that occur during instruction stream execution are called exceptions or traps. There is no “standard” definition for these terms – you must refer to the devices data sheet.

Software can also “force” an interrupt by executing an SVC (service call), or sometimes by writing the same data that hardware would otherwise have produced.

Exceptions and Interrupts

ARM presents exceptions and interrupts in a unified way (and they are similar), and generally refers to both as exceptions.

> An exception is *generated* in one of the following ways:
>
> — Directly as a result of the execution or attempted execution of the instruction stream. For example, an exception is generated as a result of an undefined instruction.
>
> — Indirectly, as a result of something in the state of the system. For example, an exception is generated as a result of an interrupt signaled by a peripheral.

ARM DDI 0406C.c “ARM Architecture Reference Manual ARM v7-A/R” page B1-1137

Exception Handling

An exception causes the processor to suspend program execution to handle an event.

When an exception is taken, the processor state is preserved immediately so that execution can be resumed from the point where the exception was taken.

More than one exception might be generated at the same time, and a new exception can be generated while the processor is handling an exception.

When an exception is taken, processor execution is forced to an address that corresponds to the type of exception. This address is called the exception vector for that exception.

Exceptions Handling

The ARM processor has eight exception vectors stored in eight consecutive memory locations starting at an exception base address. These form a vector table.

Exception Vectors, vector address, and modes		
Offset	Vector	Mode
0x00	Reset	Supervisor
0x04	Undefined Instruction	Undefined
0x08	Supervisor Call	Supervisor
0x0C	Prefetch Abort	Abort
0x10	Data Abort	Abort
0x14	Not Used	NA
0x18	IRQ Interrupt	IRQ
0x1C	FIQ Interrupt	FIQ

When an Exception is taken…

The processor leaves user mode and enters a privileged mode

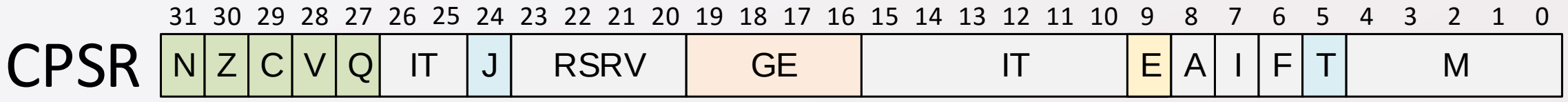
The LR and CPSR are saved in local shadow registers

Application View	User	System	Hypervisor	Supervisor	Abort	Undefined	Monitor	IRQ	FIQ
R0	R0_usr								
R1	R1_usr								
R2	R2_usr								
R3	R3_usr								
R4	R4_usr								
R5	R5_usr								
R6	R6_usr								
R7	R7_usr								
R8	R8_usr								R8_fiq
R9	R9_usr								R9_fiq
R10	R10_usr								R10_fiq
R11	R11_usr								R11_fiq
R12	R12_usr								R12_fiq
SP	SP_usr		SP_hyp	SP_svc	SP_abt	SP_und	SP_mon	SP_irq	SP_fiq
LR	LR_usr			LR_svc	LR_abt	LR_und	LR_mon	LR_irq	LR_fiq
PC	PC_usr								
APSR	CPSR		SPSR_hyp	SPSR_svc	SPSR_abt	SPSR_und	SPSR_mon	SPSR_irq	SPSR_fiq

When an Exception is taken…

The CPSR is updated with new context information

- Mode
- Mask bits
- Instruction set
- Endianess

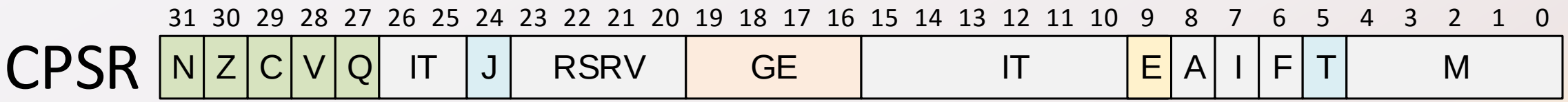


When an Exception is taken, a privileged mode is entered

Table B1-1 ARM processor modes

Processor mode		Encoding	Privilege level	Implemented	Security state
User	usr	10000	PL0	Always	Both
FIQ	fiq	10001	PL1	Always	Both
IRQ	irq	10010	PL1	Always	Both
Supervisor	svc	10011	PL1	Always	Both
Monitor	mon	10110	PL1	With Security Extensions	Secure only
Abort	abt	10111	PL1	Always	Both
Hyp	hyp	11010	PL2	With Virtualization Extensions	Non-secure only
Undefined	und	11011	PL1	Always	Both
System	sys	11111	PL1	Always	Both

Mode changes can be made under software control, or can be caused by an external or internal exception.



Privileged modes

User mode is for applications. It is the only unprivileged mode, and has restricted access to system resources. Typically, a processor spends more than 99% of its time in user mode.

Supervisor mode has unrestricted access to all resources. Entered on reset or power-up, or when software executes a Supervisor Call instruction (SVC). Typically used by OS's managing kernel or other protected files.

System mode is entered from another non-user mode by writing the CPSR. Same GPRs as user mode, but can access protected resources (recently added to aid in dealing with nested IRQs)

Abort mode is entered if a program attempts to access a non-existing memory location or execute an undefined instruction.

IRQ mode is entered in response to a normal interrupt request from an external device.

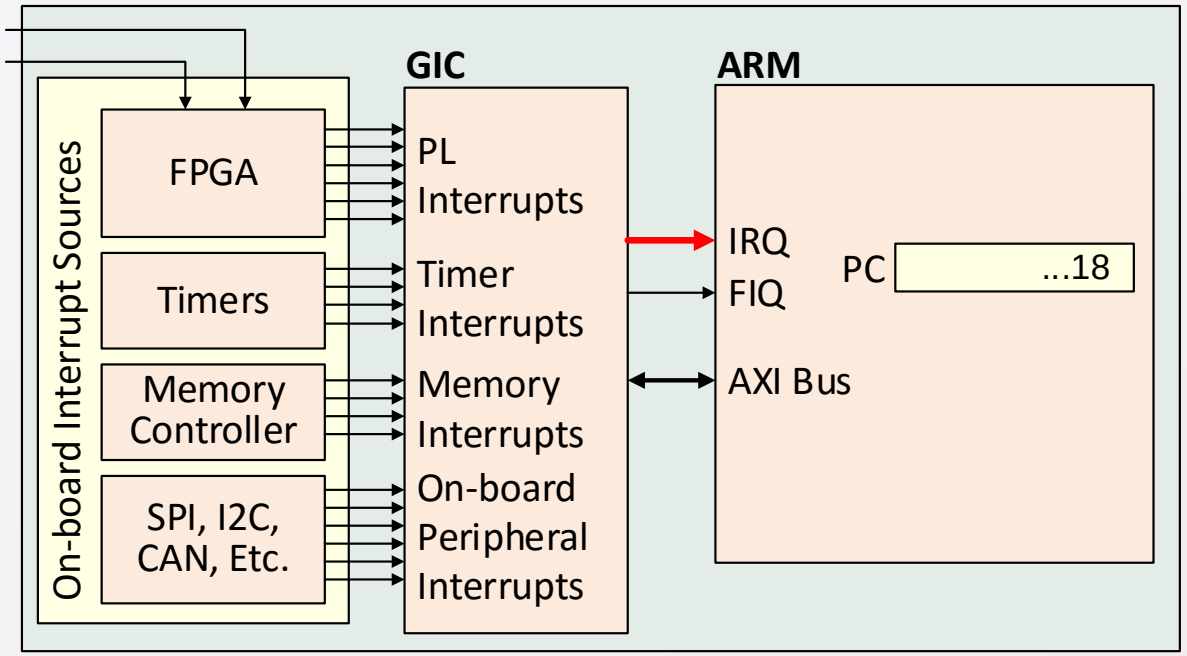
FIQ mode is entered in response to a fast interrupt request from an external device and offers faster service for more urgent requests. Dates from 8MHz/floppy disk days. FIQ automatically masks IRQ.

Every processor mode except user mode can change to a new mode by writing the CPSR

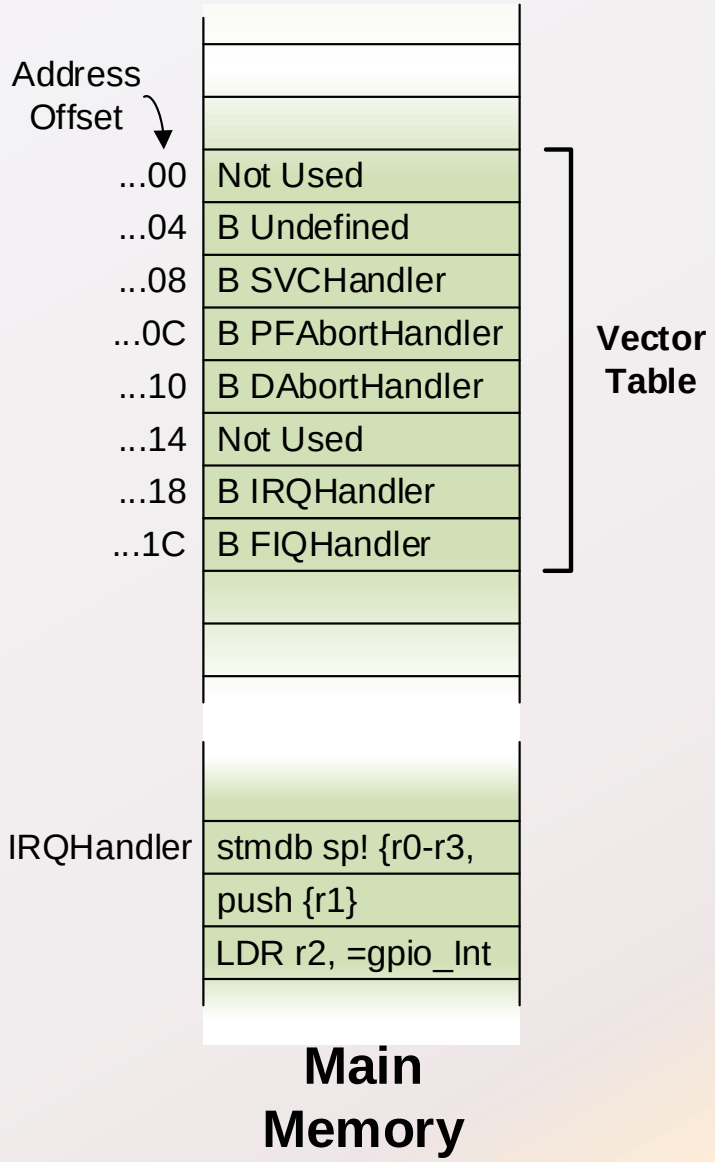
When an Exception is taken…

The appropriate exception vector is loaded

Asserting IRQ input causes ARM to change modes (to IRQ mode) and load PC with “IRQ vector” address. A branch instruction at that address will branch to IRQHandler



ZYNQ



Main Memory

When an Exception is taken…

The code shown is auto-generated by SDK and taken from an .elf file.

Note the vector table and the B 100020 instruction. The auto-generated IRQHandler can be modified to call your IRQ handler function.

SDK provides a simple way to branch to your IRQ handler

```
00100000 <_vector_table>:
   100000: ea000049 b 10012c <_boot>
   100004: ea000025 b 1000a0 <Undefined>
   100008: ea00002b b 1000bc <SVCHandler>
   10000c: ea00003b b 100100 <PrefetchAbortHandler>
   100010: ea000032 b 1000e0 <DataAbortHandler>
   100014: e320f000 nop {0}
   100018: ea000000 b 100020 <IRQHandler>
   10001c: ea00000f b 100060 <FIQHandler>

00100020 <IRQHandler>:
   100020: e92d500f push {r0, r1, r2, r3, ip, lr}
   100024: ed2d0b10 vpush {d0-d7}
   100028: ed6d0b20 vpush {d16-d31}
                .
                .
                .
```

When an Exception is taken…

You must include

include xil_exception.h

In your source file.

Then you can use the function: “Xil_ExceptionRegisterHandler” with parameters ‘5’ (for IRQ vector), the name of your C function, and NULL.

```
00100000 <_vector_table>:
   100000: ea000049 b 10012c <_boot>
   100004: ea000025 b 1000a0 <Undefined>
   100008: ea00002b b 1000bc <SVCHandler>
   10000c: ea00003b b 100100 <PrefetchAbortHandler>
   100010: ea000032 b 1000e0 <DataAbortHandler>
   100014: e320f000 nop {0}
   100018: ea000000 b 100020 <IRQHandler>
   10001c: ea00000f b 100060 <FIQHandler>

00100020 <IRQHandler>:
   100020: e92d500f push {r0, r1, r2, r3, ip, lr}
   100024: ed2d0b10 vpush {d0-d7}
   100028: ed6d0b20 vpush {d16-d31}
                .
                .
                .
```

Xil_ExceptionRegisterHandler(5, **My_IRQ_Handler**, NULL);

When an Exception is taken…

Note address 100024 – only some registers are pushed.

Why?

```
00100000 <_vector_table>:
   100000: ea000049 b 10012c <_boot>
   100004: ea000025 b 1000a0 <Undefined>
   100008: ea00002b b 1000bc <SVCHandler>
   10000c: ea00003b b 100100 <PrefetchAbortHandler>
   100010: ea000032 b 1000e0 <DataAbortHandler>
   100014: e320f000 nop {0}
   100018: ea000000 b 100020 <IRQHandler>
   10001c: ea00000f b 100060 <FIQHandler>

00100020 <IRQHandler>:
   100020: e92d500f push {r0, r1, r2, r3, ip, lr}
   100024: ed2d0b10 vpush {d0-d7}
   100028: ed6d0b20 vpush {d16-d31}
                .
                .
                .
```

Callee-saved Registers

Xilinx ARM GCC compiler defines R4-R11 as callee-save registers – it is up to user code to save and restore them.

For example, in XUartPs_ResetHw code (user code), R4-R6 are used in the function, so they are saved and restored.

xuartps_hw.c: Disassembly

```
/* void XUartPs_ResetHw(u32 BaseAddress); */
00101568 <XUartPs_ResetHw>:
   push {r4, r5, r6, lr}
   movw r1, #16383    ; 0x3fff
   mov r3, #0
   mov r2, #32
   mov r6, #40        ; 0x28
   mov r5, #3
   movw r4, #651      ; 0x28b
   mov lr, #15
   mov ip, #296       ; 0x128
   str r1, [r0, #12]
   str r6, [r0]
   str r5, [r0]
   str r1, [r0, #20]
   str r3, [r0, #4]
   str r2, [r0, #32]
   str r2, [r0, #68]   ; 0x44
   str r3, [r0, #28]
   str r4, [r0, #24]
   str lr, [r0, #52]   ; 0x34
   str ip, [r0]
   pop {r4, r5, r6, pc}
```


Reference:
1. IHI0042F ARM Procedure Call Standard for the ARM Architecture (AAPCS) Section 5.1
2. Wikipedia: Calling Convention – ARM(A32)

Programming Model for Using Interrupts

Initialization

- Vector table setup (done automatically by SDK, but can be changed)
- Configure GIC (default done automatically, but app must customize)
  - Establish priorities for selected interrupts; select sensitivity; enable source and CPU
- Configure source to produce interrupts (User app must do)
  - Select sensitivity; select polarity; enable source

Interrupt Handling: Write IRQ handler

- Save CPU States
- Get Interrupt ID from GIC
- Service valid interrupts
- Inform GIC that the interrupt has been serviced
- Restore CPU States

Initialize GIC

The GIC (Generic Interrupt Controller) is the centralized resource for managing interrupts sent to Cortex-A9 processor.

The GIC is a separate IP block from the ARM, and it is memory-mapped like any other IP block.

It has more than 100 inputs (interrupt signals), and two outputs (what are they)?

Every GIC input gets a unique ID and configurations for:

- enabling the particular input to generate an interrupt into a given CPU (in our case, there is only one CPU, but this still must be done);
- Setting the priority (0 is highest, 255 is lowest, but ZYNQ only supports 32 levels, so bits [2:0] are ignored);
- Setting the sensitivity (level or edge).

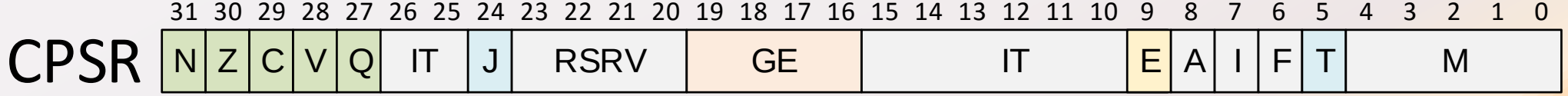
GIC Registers

Name	#	Bits	Base Addr	Function
ICCICR	1	5	0x0F8F0 0100	CPU interface control
ICCPMR	1	8	0x0F8F0 0104	CPU priority mask defines minimum priority interrupts must need to be taken
ICDDCR	1	2	0x0F8F0 1000	Distributor global enable
ICDISER	3	32	0x0F8F0 1100	Enable interrupt sources to be forwarded to CPU (1 bit per interrupt)
ICDICER	3	32	0x0F8F0 1180	Turn on or off interrupt sources (1 bit per interrupt)
ICDIPR	24	32	0x0F8F0 1400	Priority fields (8 bits per interrupt). Must be > ICCPMR for interrupt to be taken
ICDIPTR	24	8	0x0F8F0 1800	Processor targets (2 bits per interrupt). Must set to “01”
ICDICFR	6	32	0x0F8F0 1C00	Sensitivity (2-bits per interrupt). “01” for level sensitivity; “11” for edge

Enabling Interrupts

To enable interrupts, the I or F bit in the CSPR must be set (for IRQ or FIQ). The CPSR cannot be accessed from user mode, except by two special instructions. The “Move Special to Register” (MSR) instruction can move the CPSR to a GPR, and the “Move Register to Special” (MRS) can move from GPR to CPSR. These special instructions have no “C” correlate, so they must be “passed through” using a special C syntax:

```
void disable_ARM_A9_interrupts(){
        uint32_t mode = 0xDF;                    // System mode [4:0] and IRQ disabled [7]
        uint32_t read_cpsr=0;                    // used to read previous CPSR value
        uint32_t bit_mask = 0xFF;                // used to clear bottom 8 bits
        __asm__ __volatile__("mrs %0, cpsr\n" : "=r" (read_cpsr) );
        __asm__ __volatile__("msr cpsr,%0\n" : : "r" ((read_cpsr & (~bit_mask))| mode));
return; }
```



Enabling Interrupts

The GIC can be configured following the 9-step procedure in the Project 4 description.

After the GIC is configured, the interrupt source must be configured as well. Typically, every peripheral that can generate interrupts will have one to several registers that must be properly configured to enable interrupts.

Once the hardware is configured to produce interrupts, software can be written to handle them.

Interrupt Handler

Interrupt handlers run at unscheduled times, and disrupt unknown programs. They must be sure to save and restore all context.

When writing in assembly, all context should be stored on the stack. When writing in C, the compiler will do that for you. The LR and CPSR are automatically saved in local registers (and restored on exit) during mode changes.

Since the ARM has only one interrupt signal (or two if you count FIQ), the handler must determine the interrupt ID#, and then branch appropriatly.

The IRQ handler for each ID# can perform it’s task and then return.

Interrupt Handler

In general, you should spend as little time as possible in the handler.

You must decide if you want to allow other (or higher priority) interrupts to interrupt your handler, and enable or disable interrupts as appropriate.

You must choose priority levels for all interrupts.

Interrupts: General Concepts

Latency

Priority (NMI)

(Note if two interrupts share the same priority, the lowest ID# wins).

Shadow registers (in general)

Interrupt inputs signals (in general)

Interrupts: ZYNQ ID#'s

Source	Name	ID#
APU	CPU	33,32
	L2 Cache	34
	OCM	35
PMU	PMU [1:0]	
XADC	XADC	37, 38
DevC	DevC	40
SWDT	SWDT	41
Timer	TTC0	44:42
DMAC	DMAC[3:0], Abort	49:46
Memory	SMC	50
	Quad SPI	51
IOP	GPIO	52
	USB 0	53
	Ethernet 0	54, 55
	SDIO 0	56
	I2C 0	57
	SPI 0	58
	UART 0	59
	CAN 0	60

Source	Name	ID#
PL	PL [2:0]	63:61
	PL [7:3]	68:64
Timer	TTC1	71:69
DMAC	DMAC[7:4]	75:72
IOP	USB 1	76
	Ethernet 1	77
	SDIO 1	79
	I2C 1	80
	SPI 1	81
	UART 1	82
	CAN 1	83
PL	PL[15:8]	91:84
SCU	Parity	92