Microprocessors with Security
Final Exam, Fall 2015

Name: _Solutions_

Note: No Explanations, No Credits!

1. In computer systems, the interrupt controller is an essential hardware component. The GIC discussed in the class is shown in Figure 1. Another simple way to connect interrupt sources to CPU is through an OR gate as shown in Figure 2. (**20 points**)

   a. Assume that interrupts were generated from several I/O devices. So, CPU jumped to ISR (Interrupt Service Routine). Write the pseudo codes in C or ARM assembly to check the highest-priority interrupt source for Figure 1 and Figure 2. Refer to the next page for GIC registers. **(15 points)**

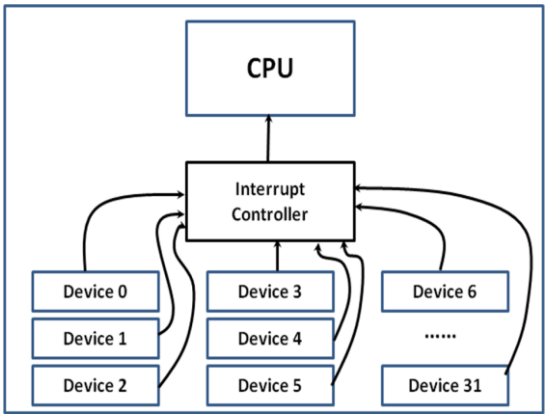


Figure 1. GIC

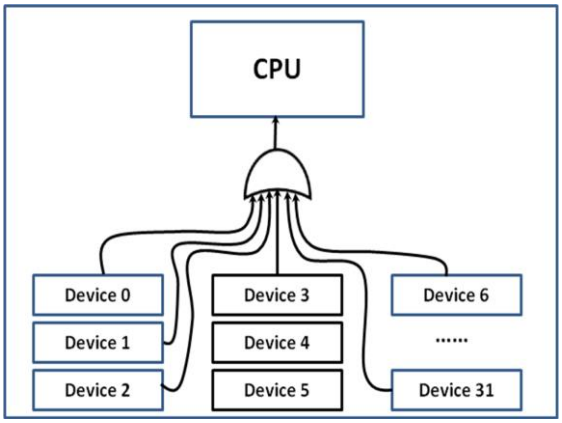


Figure 2. Simple interrupt controller

// Answer for the configuration of Figure 1	// Answer for the configuration of Figure 2

The GICC_IAR (Interrupt Acknowledge Register) contains the interrupt source number with the highest priority,
 	
ldr r1, = GICC_IAR
ldr r1, [r1]
jump to ISR based on r1
// r1 contains the source number
// with highest priority
	for (i=0; i<#devices;i++) {
	check status register of device(i);
	if (interrupt_requested)
	jump to its ISR
	}
	
	The status register should be checked in the order of the priority of devices

   b. Based on the answer in (a), what is the benefit of using Figure 1, compared to Figure 2 **(5 points)**

Interrupt latency (time to jump to an appripriate ISR) is small, compared to Figure 2

Table 4-1 Distributor register map

Offset	Name	Type	Reset[a]	Description
0x000	GICD_CTLR	RW	0x00000000	Distributor Control Register
0x004	GICD_TYPER	RO	IMPLEMENTATION DEFINED	Interrupt Controller Type Register
0x008	GICD_IIDR	RO	IMPLEMENTATION DEFINED	Distributor Implementer Identification Register
0x00C-0x01C	-	-	-	Reserved
0x020-0x03C	-	-	-	IMPLEMENTATION DEFINED registers
0x040-0x07C	-	-	-	Reserved
0x080	GICD_IGROUPRn[b]	RW	IMPLEMENTATION DEFINED[c]	Interrupt Group Registers
0x084-0x0FC			0x00000000	
0x100-0x17C	GICD_ISENABLERn	RW	IMPLEMENTATION DEFINED	Interrupt Set-Enable Registers
0x180-0x1FC	GICD_ICENABLERn	RW	IMPLEMENTATION DEFINED	Interrupt Clear-Enable Registers
0x200-0x27C	GICD_ISPENDRn	RW	0x00000000	Interrupt Set-Pending Registers
0x280-0x2FC	GICD_ICPENDRn	RW	0x00000000	Interrupt Clear-Pending Registers
0x300-0x37C	GICD_ISACTIVERn[d]	RW	0x00000000	GICv2 Interrupt Set-Active Registers
0x380-0x3FC	GICD_ICACTIVERn[e]	RW	0x00000000	Interrupt Clear-Active Registers
0x400-0x7F8	GICD_IPRIORITYRn	RW	0x00000000	Interrupt Priority Registers
0x7FC	-	-	-	Reserved

Table 4-2 CPU interface register map

Offset	Name	Type	Reset	Description
0x0000	GICC_CTLR	RW	0x00000000	CPU Interface Control Register
0x0004	GICC_PMR	RW	0x00000000	Interrupt Priority Mask Register
0x0008	GICC_BPR	RW	0x0000000x[a]	Binary Point Register
0x000C	GICC_IAR	RO	0x000003FF	Interrupt Acknowledge Register
0x0010	GICC_EOIR	WO	-	End of Interrupt Register
0x0014	GICC_RPR	RO	0x000000FF	Running Priority Register
0x0018	GICC_HPPIR	RO	0x000003FF	Highest Priority Pending Interrupt Register
0x001C	GICC_ABPR[b]	RW	0x0000000x[a]	Aliased Binary Point Register
0x0020	GICC_AIAR[c]	RO	0x000003FF	Aliased Interrupt Acknowledge Register
0x0024	GICC_AEOIR[c]	WO	-	Aliased End of Interrupt Register
0x0028	GICC_AHPPIR[c]	RO	0x000003FF	Aliased Highest Priority Pending Interrupt Register
0x002C-0x003C	-	-	-	Reserved
0x0040-0x00CF	-	-	-	IMPLEMENTATION DEFINED registers
0x00D0-0x00DC	GICC_APRn[c]	RW	0x00000000	Active Priorities Registers
0x00E0-0x00EC	GICC_NSAPRn[c]	RW	0x00000000	Non-secure Active Priorities Registers
0x00ED-0x00F8	-	-	-	Reserved
0x00FC	GICC_IIDR	RO	IMPLEMENTATION DEFINED	CPU Interface Identification Register
0x1000	GICC_DIR[c]	WO	-	Deactivate Interrupt Register

a. See the register description for more information.
b. Present in GICv1 if the GIC implements the GIC Security Extensions. Always present in GICv2.
c. GICv2 only.

2. You want to call a Thumb code from ARM mode, and return to the ARM code from Thumb mode. Fill up the gray box with instruction(s). **(10 points)**

<ARM code>

```
      mov r9,  #0x99
      mov r10, #0xaa
      mov r11, #0xbb
      mov r12, #0xcc
      mov r13, #0xdd

      1 // call ping (Thumb code)

pong: add  r1, r1, r2
      nop
      nop
```

<Thumb code>

```
.thumb
.thumb_func

ping:

mov r0, #0x77
mov r1, #0x66
mov r2, #0x55
mov r3, #0x44

1 // return to pong (ARM code)
```

Use **BLX instruction** for calling Thumb code and **BX** for returning to ARM code

```
// call ping

blx ping
```

```
// return to pong

bx  lr
```

Use **BX instructions ONLY** for calling Thumb code and returning to ARM code

```
// call ping

ldr r0,=ping
bx r0
```

```
// return to pong

ldr r0,=pong
bx r0
```

3. Convert the following C code to **Thumb** assembly code **using the `IT` instruction**. Show how the `CPSR'IT` field changes as your answer code gets executed. Assume that the variables `f`, `g`, `h`, `i`, and `j` are assigned to registers R0, R1, R2, R3, and R4, respectively. Refer to the ARM condition table **(20 points = 10 + 10)**

```
int  f, g, h, i, j;

if (i > j) {
    f = g + h;
    f++;
} else {
    f = g - h;
}
```

// R0 = f, R1 = g, R2 = h, R3 = i, R4 = j

```
        cmp  R3, R4;
        ITTE   GT;
        addgt  R0, R1, R2
        addgt  R0, R0, #1
        suble   R0, R1, R2;
```

// Show how the `CPSR'IT` field changes as your answer code gets executed.

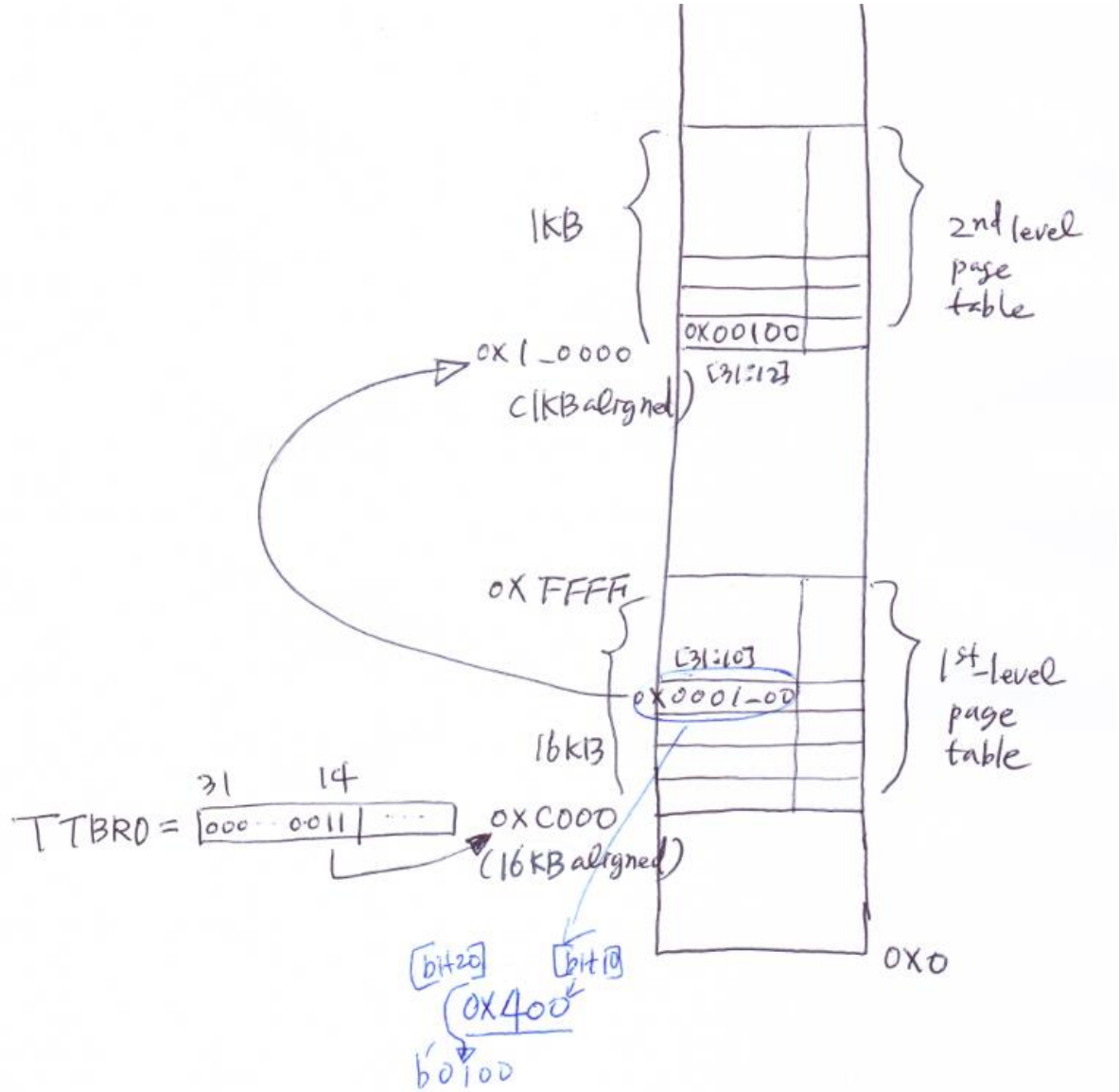
```
        cmp  R3, R4;
        ITTE   GT;              //  IT = 8'b1100_0110
        addgt  R0, R1, R2       //  IT = 8'b1100_1100
        addgt  R0, R0, #1       //  IT = 8'b1101_1000
        suble   R0, R1, R2;     //  IT = 8'b0000_0000
```

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)

4. Cortex A9 has separate L1 caches for instructions and data. Especially, L1 I$ is VIPT-based. In VIVT and VIPT caches, there could be homonym and synonym problems. **Explain what homonym and synonym are and why it happens**? (**10 points**)

Check out lecture slides

5. You want to map **a 4KB virtual page from 0x0030_0000** to **a physical page from 0x0010_0000**. Draw page tables in memory below as detailed as possible. Focus only on memory addresses when you draw page tables (that is, ignore the other bit fields). Specify the base locations of page tables in the figure as well. Elaborate why you chose the base locations. What value would you program to the TTBR register? (Refer to the page table entry information in the next page) **(25 points)**



Short-descriptor translation table first-level descriptor formats

Each entry in the first-level table describes the mapping of the associated 1MB MVA range.

Figure B3-4 shows the possible first-level descriptor formats.

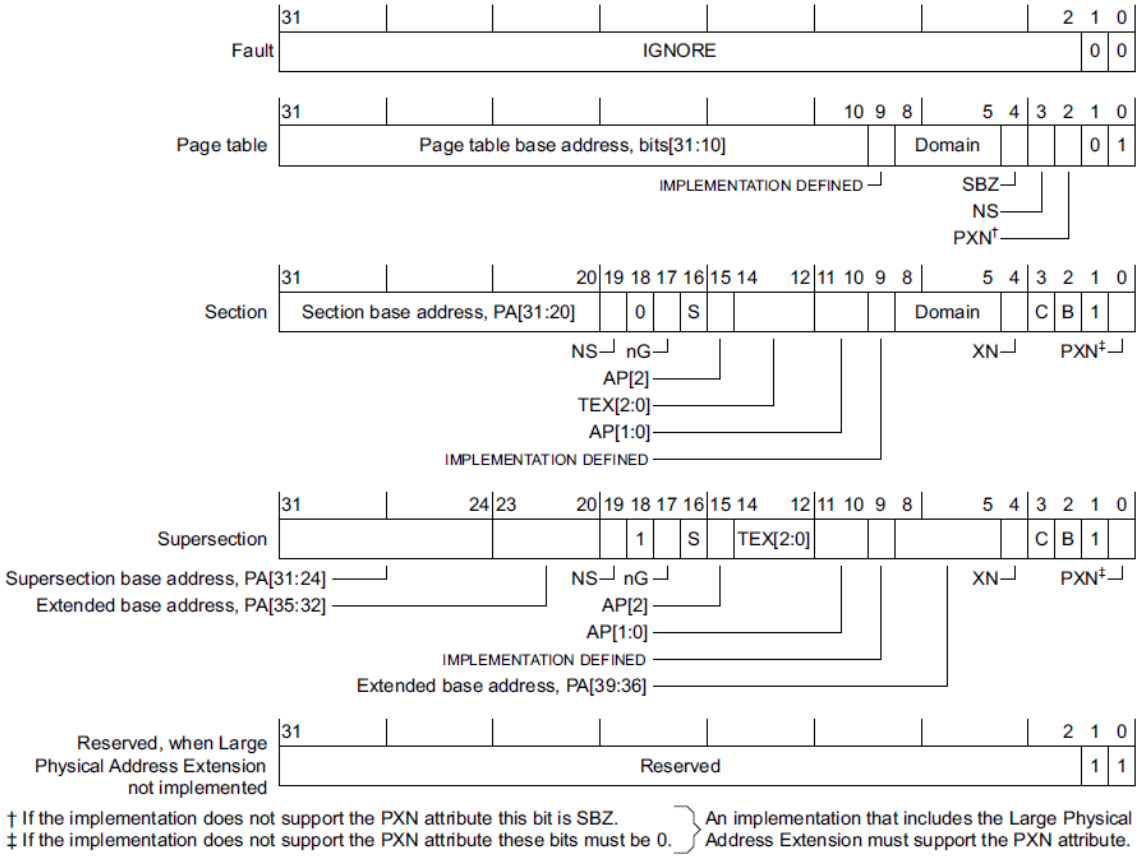


Figure B3-4 Short-descriptor first-level descriptor formats

Short-descriptor translation table second-level descriptor formats

Figure B3-5 shows the possible formats of a second-level descriptor.

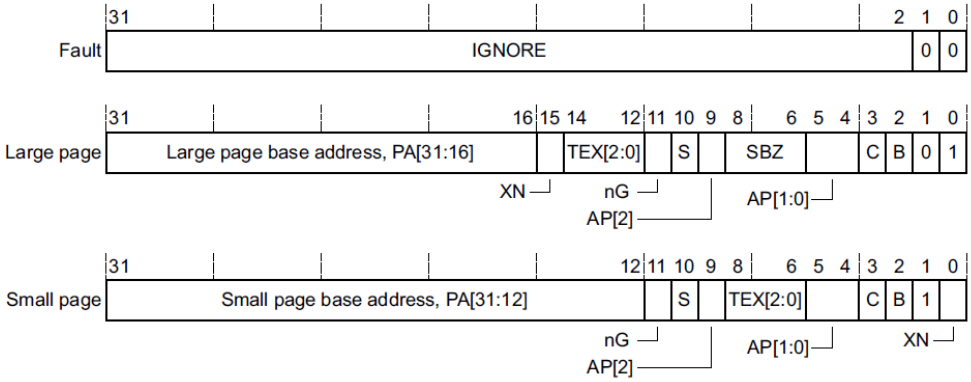
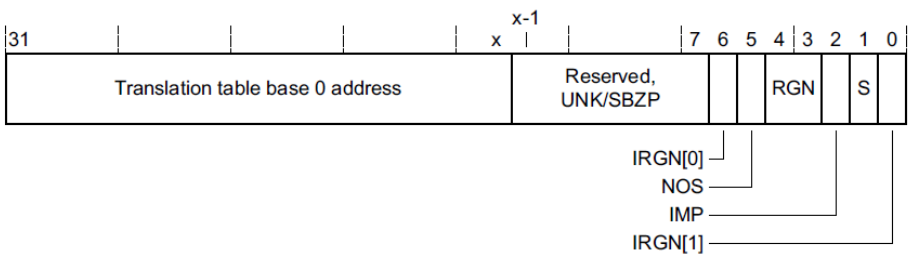
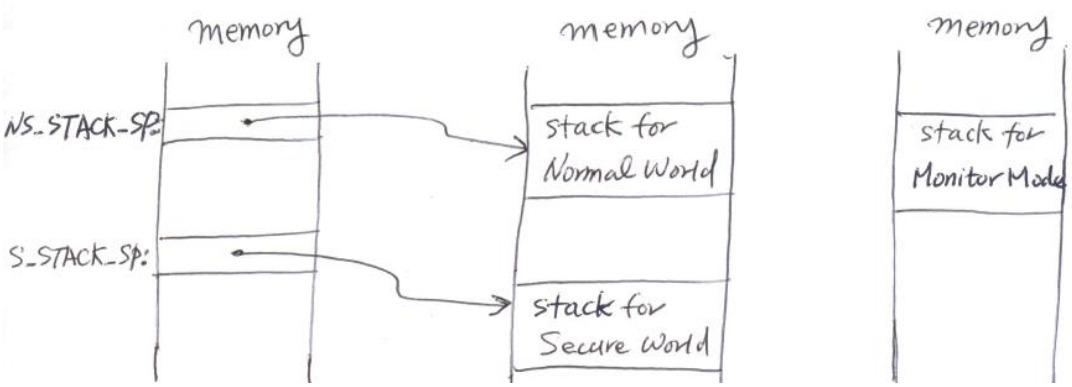


Figure B3-5 Short-descriptor second-level descriptor formats

In an implementation that includes the Multiprocessing Extensions, the 32-bit TTBR0 bit assignments are:



6. You want to do the context-switching between normal world and secure world. The following is an FIQ handler in the monitor node for the context-switching. Fill up the blanks below. The stack pointers for normal world and secure world were saved in NS_STACK_SP and S_STACK_SP, respectively, as shown below. r0 ~ r3 are used as arguments passing between worlds. **(15 points).**



FIQ handler Code for context-switching in Monitor Mode

FIQ_Handler:

// r0-r3 contain args to be passed between worlds
// Temporarily stack r0~r3, so can be used as scratch registers

PUSH {r0-r3}

// Figure out which world we have come from

MRC p15, 0, r0, c1, c1, 0 ; Read SCR
TST r0, #NS_BIT ; Is the NS bit set?
EOR r0, r0, #NS_BIT ; Toggle NS bit
MCR p15, 0, r0, c1, c1, 0 ; Write to SCR

// Load saved stack pointer to **r2** for **push** operation (for pushing current world context)

LDREQ r0, =S_STACK_SP ; If NS bit set, it was in Normal world. So restore Secure state
LDRNE r0, =NS_STACK_SP
LDR r2, [r0]

// Load saved stack pointer to **r3** for the **pop** operation (for popping the other world context)

LDREQ r1, =NS_STACK_SP
LDRNE r1, =S_STACK_SP
LDR r3, [r1]

// Save (push) current world's registers (r4~r12), SPSR and LR

STMFD r2!, {r4-r12} ; Save r4 to r12
MRS r4, spsr ; Also get a copy of the SPSR
STMFD r2!, {r4, lr} ; Save original SPSR and LR
STR r2, [r0] ; Save updated pointer bac
; r0 and r2 now free

// Restore (pop) other world's registers (r4~r12), SPSR and LR

LDMFD r3!, {r0, lr} ; Get SPSR and LR from
MSR spsr_cxsf, r0 ; Restore SPSR
LDMFD r3!, {r4-r12} ; Restore registers r4 to r12
STR r3, [r1] ; Save updated pointer back, r1 and r3 now free

// Now restore args (r0-r3)

POP {r0-r3}

// Perform exception return

MOVS pc, lr