

# COSE321 Computer Systems Design

## Final Exam, Spring 2022

Name: \_\_\_\_\_

**Note: No Explanations, No Credits!**

1. You want to implement system calls (`fork()`, `open()`, and `close()`) via `svc` instruction and decided to use `'svc #1'` for `fork()`, `'svc #2'` for `open()` and `'svc #3'` for `close()`. Write the code in `SVC_ISR` that calls the appropriate handler. **(20 points)**

```
csd_vector_table:
```

```
    b .
    b SVC_ISR
    b .
    b .
    b .
    b .
    b .
    b .
```

```
// system call handlers
```

```
fork_handler:
```

```
    ...
    movs pc, lr
```

```
open_handler:
```

```
    ...
    movs pc, lr
```

```
close_handler:
```

```
    ...
    movs pc, lr
```

SVC\_ISR:

### Encoding A1

ARMv4\*, ARMv5T\*, ARMv6\*, ARMv7

SVC<c> #<imm24>

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
cond				1	1	1	1	imm24																							

Table 1. LR adjustments

Exceptions	Adjustment	Returned Place
Software Interrupt (SVC)	0	Next Instruction
Undefined Instruction	0	Next Instruction
Prefetch Abort	-4	Aborting Instruction
Data Abort	-8	Aborting Instruction
FIQ	-4	Next Instruction
IRQ	-4	Next Instruction

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 $N != V$ )

2. Assume that there are two I/O devices (Private Timer and GPIO) generating normal interrupts (IRQ) to Arm CPU. Write the Arm assembly **1)** for setting up priority-related registers in GIC for the two I/Os and **2)** for jumping to an appropriate ISR (`Private_Timer_ISR` or `GPIO_ISR`) upon interrupt. The Private Timer's ID is 29, and the GPIO's ID is 61. Assume that the Private Timer has a higher priority than the GPIO. **Explain** your code. **(25 points)**

<pre>// Interrupt Vector Table csd_vector_table:     b .     b .     b .     b .     b .     b .     b csd_IRQ_ISR     b .  main:  //Assume that VBAR and stack pointers //are set up appropriately beforehand  // 1) priority-related register //    set up in GIC //    (Distributor &amp; CPU Interface)</pre>	<pre>// ----- // IRQ_ISR // -----  csd_IRQ_ISR:  // 2) Jump to an appropriate ISR   // ISR for Private Timer Private_Timer_ISR:     ...     subs pc, lr, #4  // ISR for GPIO GPIO_ISR:     ...     subs pc, lr, #4</pre>
---	--

## <GIC Distributor>

**Base address: 0xF8F0\_1000**

**Table 4-1 Distributor register map**

Offset	Name	Type	Reset <sup>a</sup>	Description
0x000	<a href="#">GICD_CTLR</a>	RW	0x00000000	Distributor Control Register
0x004	<a href="#">GICD_TYPER</a>	RO	IMPLEMENTATION DEFINED	Interrupt Controller Type Register
0x008	<a href="#">GICD_IIDR</a>	RO	IMPLEMENTATION DEFINED	Distributor Implementer Identification Register
0x00C-0x01C	-	-	-	Reserved
0x020-0x03C	-	-	-	IMPLEMENTATION DEFINED registers
0x040-0x07C	-	-	-	Reserved
0x080	<a href="#">GICD_IGROUPRn<sup>b</sup></a>	RW	IMPLEMENTATION DEFINED <sup>c</sup>	Interrupt Group Registers
0x084-0x0FC			0x00000000	
0x100-0x17C	<a href="#">GICD_ISENABLERn</a>	RW	IMPLEMENTATION DEFINED	Interrupt Set-Enable Registers
0x180-0x1FC	<a href="#">GICD_ICENABLERn</a>	RW	IMPLEMENTATION DEFINED	Interrupt Clear-Enable Registers
0x200-0x27C	<a href="#">GICD_ISPENDRn</a>	RW	0x00000000	Interrupt Set-Pending Registers
0x280-0x2FC	<a href="#">GICD_ICPENDRn</a>	RW	0x00000000	Interrupt Clear-Pending Registers
0x300-0x37C	<a href="#">GICD_ISACTIVERn<sup>d</sup></a>	RW	0x00000000	GICv2 Interrupt Set-Active Registers
0x380-0x3FC	<a href="#">GICD_ICACTIVERn<sup>e</sup></a>	RW	0x00000000	Interrupt Clear-Active Registers
0x400-0x7F8	<a href="#">GICD_IPRIORITYRn</a>	RW	0x00000000	Interrupt Priority Registers
0x7FC	-	-	-	Reserved
0x800-0x81C	<a href="#">GICD_ITARGETSRn</a>	RO <sup>f</sup>	IMPLEMENTATION DEFINED	Interrupt Processor Targets Registers
0x820-0xBF8		RW <sup>f</sup>	0x00000000	
0x8FC	-	-	-	Reserved
0xC00-0xCFC	<a href="#">GICD_ICFGRn</a>	RW	IMPLEMENTATION DEFINED	Interrupt Configuration Registers
0xD00-0xDFC	-	-	-	IMPLEMENTATION DEFINED registers
0xE00-0xEFC	<a href="#">GICD_NSACRn<sup>e</sup></a>	RW	0x00000000	Non-secure Access Control Registers, optional
0xF00	<a href="#">GICD_SGIR</a>	WO	-	Software Generated Interrupt Register
0xF04-0xF0C	-	-	-	Reserved
0xF10-0xF1C	<a href="#">GICD_CPENDSGIRn<sup>e</sup></a>	RW	0x00000000	SGI Clear-Pending Registers
0xF20-0xF2C	<a href="#">GICD_SPENDSGIRn<sup>e</sup></a>	RW	0x00000000	SGI Set-Pending Registers
0xF30-0xFCC	-	-	-	Reserved
0xFD0-0xFFC	-	RO	IMPLEMENTATION DEFINED	<i>Identification registers on page 4-119</i>

- a. For details of any restrictions that apply to the reset values of IMPLEMENTATION DEFINED cases see the appropriate register description.
- b. In a GICv1 implementation, present only if the GIC implements the GIC Security Extensions, otherwise RAZ/WI.
- c. For more information see [GICD\\_IGROUPR0 reset value on page 4-92](#).
- d. In GICv1, these are the Active Bit Registers, ICDABRn. These registers are RO.

## <GIC CPU Interface>

### Base address: 0xF8F0\_0100

Table 4-2 CPU interface register map

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

Figure 4-27 shows the IAR bit assignments.



Figure 4-27 GICC\_IAR bit assignments

Table 4-34 shows the IAR bit assignments.

Table 4-34 GICC\_IAR bit assignments

Bit	Name	Function
[31:13]	-	Reserved.
[12:10]	CPUID	For SGIs in a multiprocessor implementation, this field identifies the processor that requested the interrupt. It returns the number of the CPU interface that made the request, for example a value of 3 means the request was generated by a write to the <a href="#">GICD_SGIR</a> on CPU interface 3. For all other interrupts this field is RAZ.
[9:0]	Interrupt ID	The interrupt ID.

3. Refer to the following code. Assume that `VAR` is already set correctly for using our vector table, and all stack pointers are also set appropriately beforehand. **(30 points)**
- a. While CPU is executing `forever` loop, assume that a hardware interrupt is delivered to Arm CPU via the IRQ input. **Explain the execution flow** of the code from `main`. **(15 points)**

<pre>csd_vector_table:     b .     b csd_undefined     b .     b .     b .     b csd_IRQ_ISR     b .  main:     cpsIE aif, #0x10 // Change to User Mode</pre>	<pre>forever:     b forever  csd_undefined:     movs pc, lr  csd_IRQ_ISR:      .word 0xffffffff     subs pc, lr, #4</pre>
---	---

// Execution flow

- b. While CPU is executing `forever` loop, assume that a hardware interrupt is delivered to Arm CPU via the IRQ input. Then another hardware interrupt is delivered to Arm CPU via the IRQ input while executing `nops` in ISR. **Explain the execution flow** of the code from `main`. **(15 points)**

```
csd_vector_table:
```

```
    b .  
    b .  
    b .  
    b .  
    b .  
    b .  
    b .  
    b csd_IRQ_ISR  
    b .
```

```
main:
```

```
    cpsIE aif, #0x10 // Change to User Mode
```

```
forever:
```

```
    b forever
```

```
csd_IRQ_ISR:
```

```
    cpsIE i
```

```
    nop
```

```
    nop
```

```
    nop
```

```
    subs pc, lr, #4
```

```
// Execution flow
```

4. Figure out the address translation sizes and mappings (virtual address-to-physical address mappings), with the following page tables. Refer to the page table entry information in the next page. Assume TTBR0 is set correctly beforehand. **Explain** your answer in detail. **(20 points)**

```
// 1st level page table (csd_MMUTable)

csd_MMUTable:
.set SECT, 0xAAA00000
.word SECT + 0x15de6
.word csd_MMUTable_lv2_1 + 0x1e1
.word csd_MMUTable_lv2_2 + 0x1e1

// 2nd level page tables (csd_MMUTable_lv2_1, csd_MMUTable_lv2_2)

csd_MMUTable_lv2_1:
.word 0xAAA00002

csd_MMUTable_lv2_2:
.word 0xAAA00002
```

Translation size (4KB, 64KB, 1MB, or 16MB?)	Virtual address range		Physical address range
		→	



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

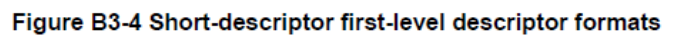
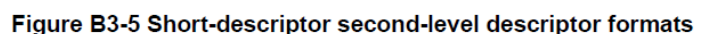


Figure B3-5 shows the possible formats of a second-level descriptor.



5. Assume that the monitor node was entered via `smc` instruction. **(25 points)**
- a. Explain what operations are performed by CPU (hardware) upon the execution of `smc` instruction. **(10 points)**
- b. You want to do both **world switching** and **context-switching** in the monitor mode. The context switching code is prepared for you below. Write code of performing the world-switching and calling an appropriate context-switching function. **(15 points)**

```
// Exception Vector Table in Monitor mode
csd_monitor_vector:
    b .
    b .
    b csd_SMC_handler
    b .
    b .
    b .
    b .
    b .

csd_SMC_Handler:

// Write the code of 1) performing the world-switching and
//                               2) calling an appropriate context-switching function


// context-switching code below

// push the normal world context to stack
// pop the secure world context from stack
push_Normal_pop_Secure:
    ...

// push the secure world context to stack
// pop the normal world context from stack
push_Secure_pop_Normal:
    ...
```

### Accessing the SCR

To access the SCR, software reads or writes the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c1, `<CRm>` set to c1, and `<opc2>` set to 0. For example:

```
MRC p15, 0, <Rt>, c1, c1, 0    ; Read SCR into Rt
MCR p15, 0, <Rt>, c1, c1, 0    ; Write Rt to SCR
```