COSE321 Computer Systems Design
Final Exam, Spring 2022

Name: _Solutions_

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 handers

fork_handler:
            …
            movs pc, lr

open_handler:
            …
            movs pc, lr

close_handler:
            …
            movs pc, lr
```

SVC_ISR:

```
   sub  r1, lr, #4
   ldr  r1, [r1]  // read svc instruction
   and  r1, r1, #0xFFFFFF
   cmp  r1, #1
   beq  fork_handler
   cmp  r1, #2
   beq  open_handler
   cmp  r1, #3
   beq  close_handler
```

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**)

```
// 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 & CPU Interface)

      ldr r0, =GICD_PRIOR7
      ldr r1, [r0]
// 16 for ID# 29 (Private Timer)
      mov r2, #0x10 << 8
      orr r1, r1, r2
      str r1, [r0]

      ldr r0, =GICD_PRIOR15
      ldr r1, [r0]
// 32 for ID# 61 (GPIO)
      mov r2, #0x20 << 8
      orr r1, r1, r2
      str r1, [r0]


      // CPU Interface PMR
      ldr r0, =GICC_PMR
      mov r2, #0xFF     // Lowest
      str r1, [r0]
```

```
// --------
// IRQ ISR
// ---------

csd_IRQ_ISR:

// 2) Jump to an appropriate ISR

      // Read Interrupt Ack register
      ldr r0, =GICC_IAR
      ldr r3, [r0]

      mov r4, #0x3FF
      and r4, r3, r4
      cmp r4, #29
      beq  Private_Timer_ISR
      cmp r4, #61
      beq  GPIO_ISR
       …




// ISR for Private Timer
Private_Timer_ISR:
      …
      subs pc, lr, #4



// ISR for GPIO
GPIO_ISR:
      …
      subs pc, lr, #4
```

<GIC Distributor>
Base address: 0xF8F0_1000

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
0x800-0x81C	GICD_ITARGETSRn	RO[f]	IMPLEMENTATION DEFINED	Interrupt Processor Targets Registers
0x820-0xBF8		RW[f]	0x00000000	
0xBFC	-	-	-	Reserved
0xC00-0xCFC	GICD_ICFGRn	RW	IMPLEMENTATION DEFINED	Interrupt Configuration Registers
0xD00-0xDFC	-	-	-	IMPLEMENTATION DEFINED registers
0xE00-0xEFC	GICD_NSACRn[e]	RW	0x00000000	Non-secure Access Control Registers, optional
0xF00	GICD_SGIR	WO	-	Software Generated Interrupt Register
0xF04-0xF0C	-	-	-	Reserved
0xF10-0xF1C	GICD_CPENDSGIRn[e]	RW	0x00000000	SGI Clear-Pending Registers
0xF20-0xF2C	GICD_SPENDSGIRn[e]	RW	0x00000000	SGI Set-Pending Registers
0xF30-0xFCC	-	-	-	Reserved
0xFD0-0xFFC	-	RO	IMPLEMENTATION DEFINED	*Identification registers* on page 4-119

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	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.

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 GICD_SGIR 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**)

```
csd_vector_table:
      b .
      b csd_undefined
      b .
      b .
      b .
      b .
      b csd_IRQ_ISR
      b .

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

```
forever:
      b forever

csd_undefined:
      movs  pc, lr

csd_IRQ_ISR:

      .word 0xffffffff
      subs  pc, lr, #4
```

Execution flow:

main → csd_IRQ_ISR → Undefined exception → return to csd_IRQ_ISR → return to main

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 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:

main → csd_IRQ_ISR (1st interrupt) → csd_IRQ_ISR (2nd interrupt) → return to IRQ hander → return to IRQ handler (not `forever` loop in main code)

4. Figure out the address translation sizes and mappings from virtual address to physical address, 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
1MB	0x0000_0000 ~ 0x000F_FFFF		0xAAA0_0000 ~ 0xAAAF_FFFF
4KB	0x0010_0000 ~ 0x0010_0FFF	→	0xAAA0_0000 ~ 0xAAA0_0FFF
4KB	0x0020_0000 ~ 0x0020_0FFF		0xAAA0_0000 ~ 0xAAA0_0FFF

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.

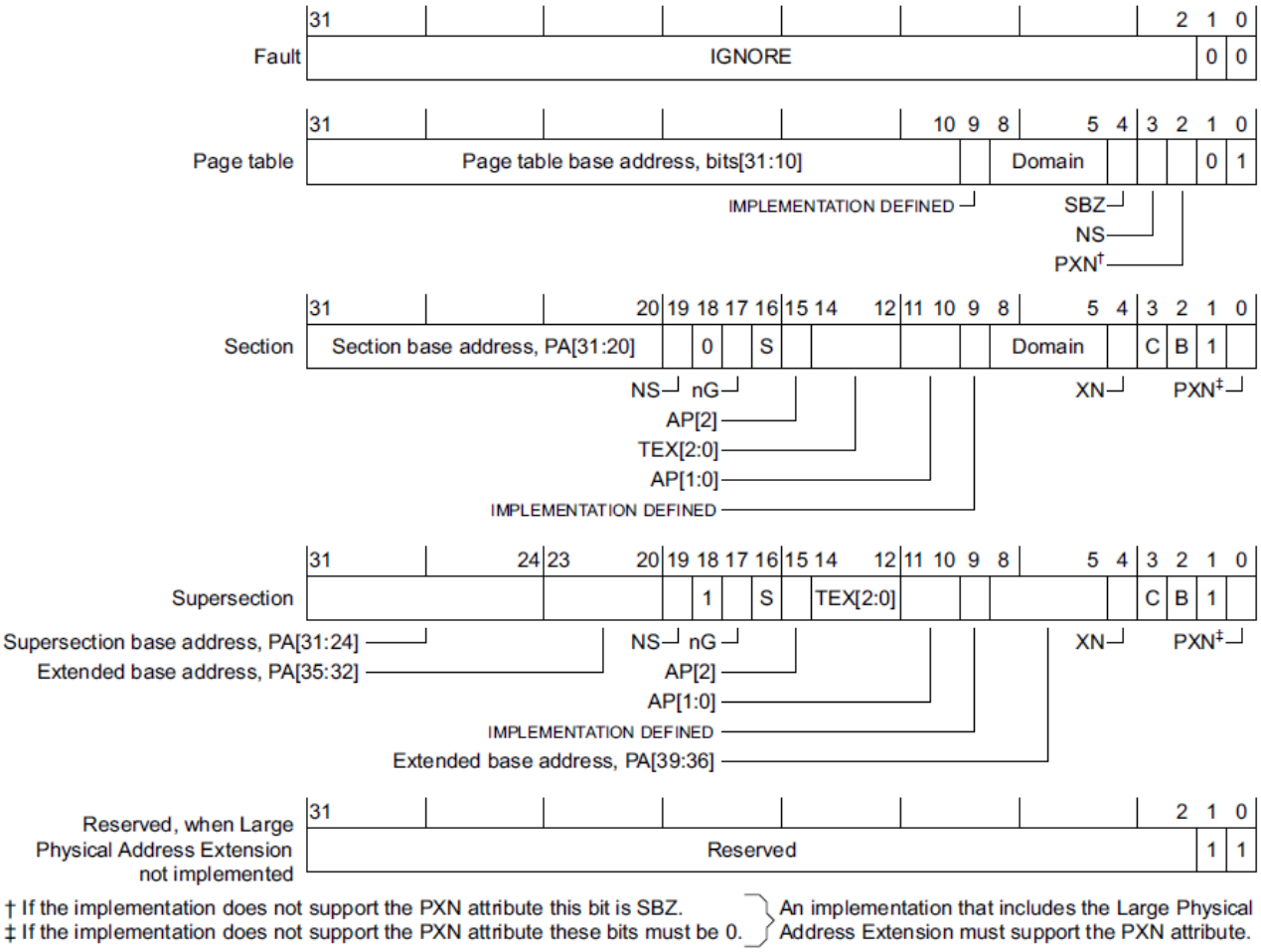

Figure B3-4 Short-descriptor first-level descriptor formats

Short-descriptor translation table second-level descriptor formats

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

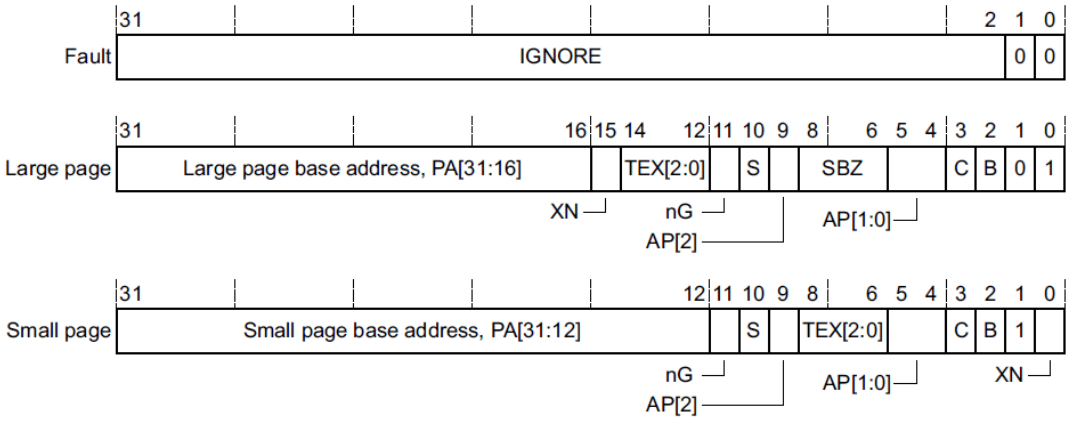

Figure B3-5 Short-descriptor second-level descriptor formats

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**)

- `SPSR_mon` ← `CPSR`
- `lr_mon (r14)` ← address of next inst.
- `CPSR'mode` ← monitor mode (10110)
- `CPSR'A,I,F = 111`
- `PC` ← `MVBAR + 0x8`

   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

mrc     p15, 0, r0, c1, c1, 0  ; Read SCR
tst     r0, #1                 ; Is the NS bit set?
eor     r0, r0, #b'1           ; Toggle NS bit
mcr     p15, 0, r0, c1, c1, 0  ; Write to SCR
beq     push_secure_pop_normal
bne     push_normal_pop_secure




// 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
```