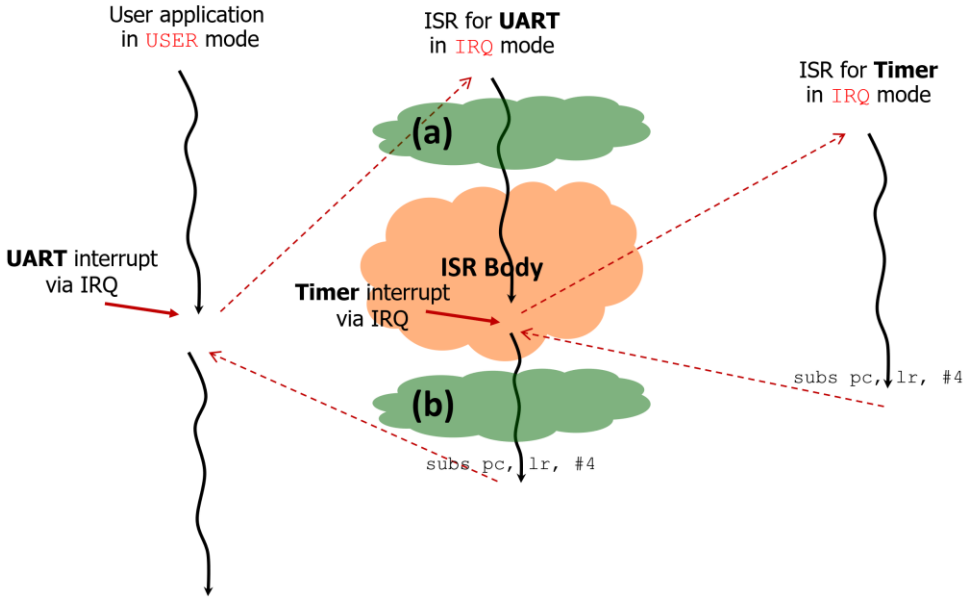
COSE321 Computer Systems Design
Final Exam, Spring 2023

Name: _Solutions_

Note: No Explanations, No Credits!

1. You want to handle the nested-interrupt case shown in the Figure below. **What kind of operations** in (a) and (b) should be done from the CPU side for the nested interrupt? Write Arm assembly code for **those operations**. Assume that stack pointers were set properly beforehand. **(20 points)**

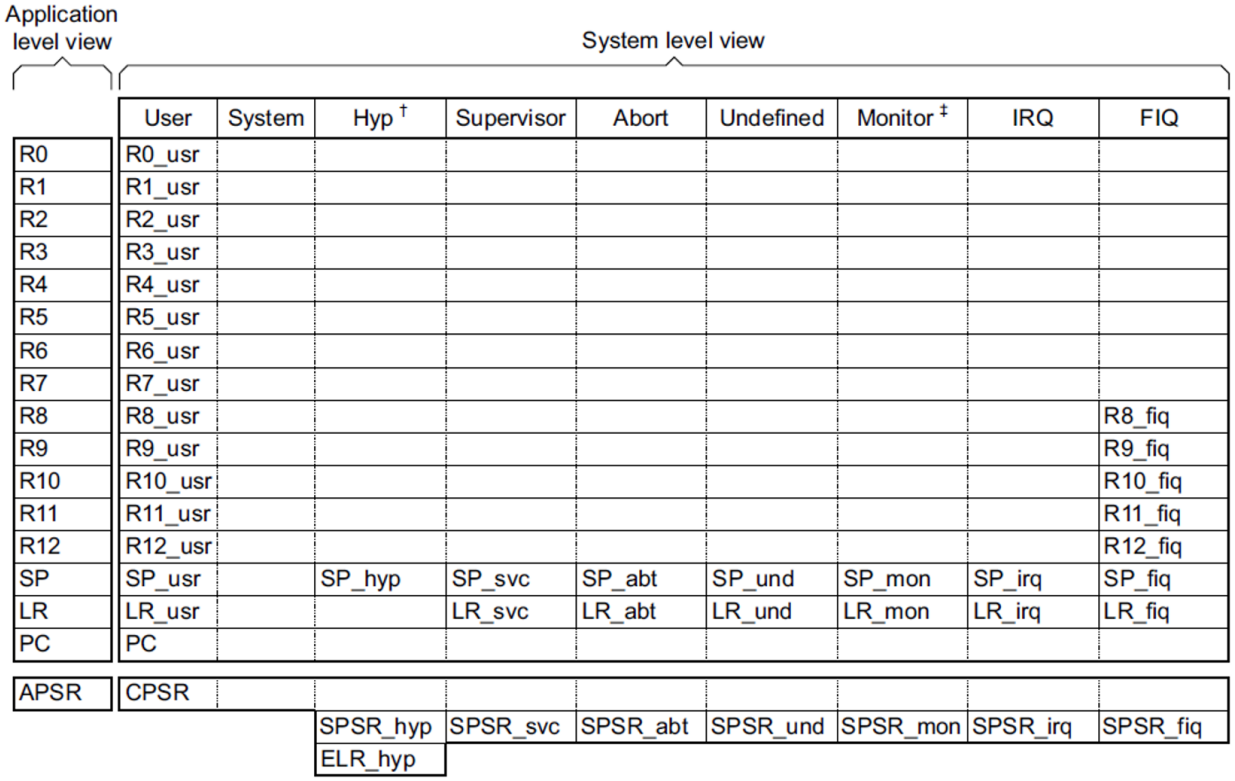


Assembly code for the operations in (a)	Assembly code for the operations in (b)

`* Push CPU context to stack`
`* IRQ enable in CPSR'I`
 	
`stmfd sp!, {r0~r12}`
`srsfd sp!, #0x12`
`cpsIE i  // IRQ enable`
 	
OR
 	
`stmfd sp!, {r0~r12}`
`mrs  r0, spsr`
`stmfd sp!, {r0, lr}`
`cpsIE i  // IRQ enable`
	`* Pop CPU context from stack`
	
	`ldmfd sp!, {r0, lr}`
	`msr  spsr, r0`
	`ldmfd sp!, {r0~r12}`

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



2. Assume a multicore system with 8 Arm CPUs. You want to use IPI (Inter-processor Interrupt) for 2 purposes: 1) task dispatch and 2) TLB shootdown. For the task dispatch, you decided to use SGI (Software-generated Interrupt) ID #3. For the TLB shootdown, you decided to use SGI ID #4. (**30 points**)

   a. Write the Arm assembly **1)** for setting up priority-related registers in GIC for the two SGIs and **2)** for jumping to an appropriate ISR (`Task_dispatch` or `TLB_shootdown`) upon interrupt. Assume that the TLB shootdown has a higher priority than the Task dispatch. **Explain** your code. (**20 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_PRIOR0
      ldr r1, [r0]
// 32 for ID# 3 (Task Dispatch)
      mov r2, #0x20 << 24
      orr r1, r1, r2
      str r1, [r0]

      ldr r0, =GICD_PRIOR1
      ldr r1, [r0]
// 16 for ID# 4 (TLB Shootdown)
      mov r2, #0x10 << 0
      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, #3
      beq  Task_dispatch
      cmp  r4, #4
      beq  TLB_shootdown
       …


// ISR for Task Dispatch
Task_dispatch:
      …
      subs pc, lr, #4


// ISR for TLB Shootdown
TLB_shootdown:
      …
      subs pc, lr, #4
```

b. Write the Arm assembly of generating the IPIs **1)** for the task-dispatch and **2)** for the TLB shootdown. You want to send the IPIs to 4 CPUs (#0, #1, #6 and #7). (**10 points**)

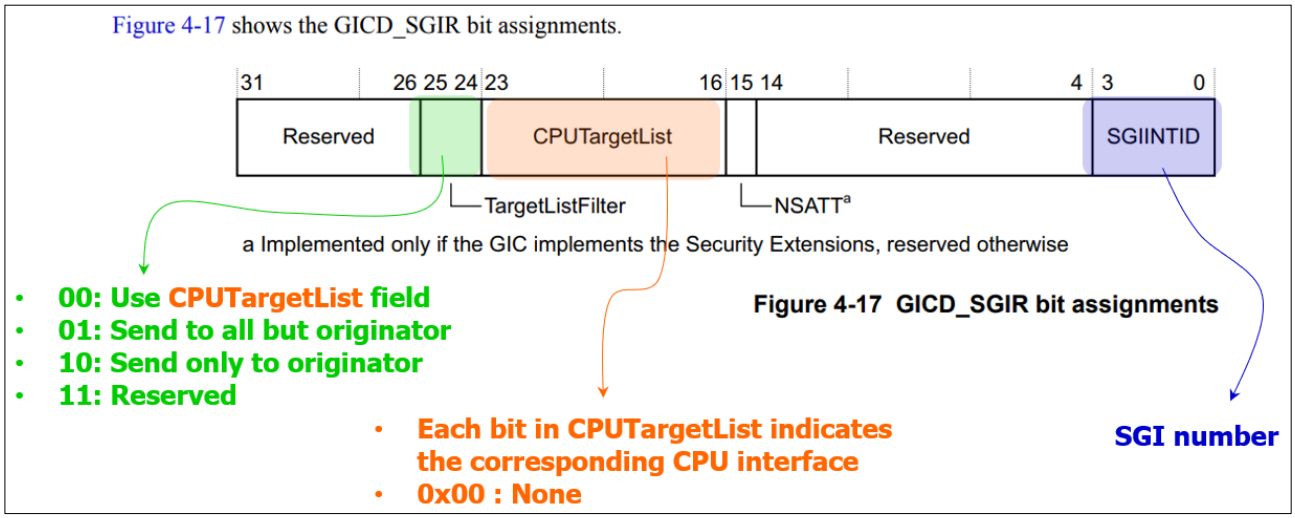
```
// 1) IPI generation for the task dispatch

      ldr r0, =GICD_SGIR
      mov r2, #0b00 << 24  // Use CPUTargetList
      mov r3, #0xC3 << 16  // 0b'1100_0011
      mov r4, #0x3  << 0   // ID #3
       orr r1, r2, r3
       orr r1, r1, r4
      str r1, [r0]         // SGI #3 to the target CPUs




// 2) IPI generation for the TLB shootdown


      ldr r0, =GICD_SGIR
      mov r2, #0b00 << 24  // Use CPUTargetList
      mov r3, #0xC3 << 16  // 0b'1100_0011
      mov r4, #0x4  << 0   // ID #4
       orr r1, r2, r3
       orr r1, r1, r4
      str r1, [r0]         // SGI #4 to the target CPUs
```



<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. Answer to the following questions regarding the address translation. **(30 points)**

   a. Assume that the page tables were already set up correctly for Cortex-A9. If CPU executes the code below, **how many page tables** should be accessed to find out the translation information? **Which entry in each page table** should be accessed, and **why**? **(15 points = 5 + 10)**

Translation granule	Arm Instruction	How many page tables? Which entry in each table?
1MB	// assume r1 = 0x0023_4560 `ldr r0, [r1]`	L1 only, 3rd (=0x2) entry in L1
4KB	// assume r1 = 0x0034_5670 `str r0, [r1]`	L1, L2. 4th (=0x3) entry in L1, 70th (=0x45) entry in L2

   b. With the following page tables, figure out the address translation granules and mappings from virtual space to physical space. Refer to the page table entry information in the next page. Assume TTBR0 already is set to `csd_MMUTable` beforehand. **Explain** your answer in detail. **(15 points)**

```
csd_MMUTable:
.set  SECT,  0xBBB00000
.word  csd_MMUTable_L2_1 + 0x1e1
.word  SECT + 0x15de6
.word  csd_MMUTable_L2_2 + 0x1e1

csd_MMUTable_L2_1:
.word 0xAAA00002

csd_MMUTable_L2_2:
.word 0xCCC00002
```

Translation granule (4KB, 64KB, 1MB, or 16MB?)	Range of virtual space		Range of physical space
4KB	0x0000_0000 ~ 0x000_0FFF	→	0xAAA0_0000 ~ 0xAAA0_0FFF
1MB	0x0010_0000 ~ 0x001F_FFFF		0xBBB0_0000 ~ 0xBBBF_FFFF
4KB	0x0020_0000 ~ 0x0020_0FFF		0xCCC0_0000 ~ 0xCCC0_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.

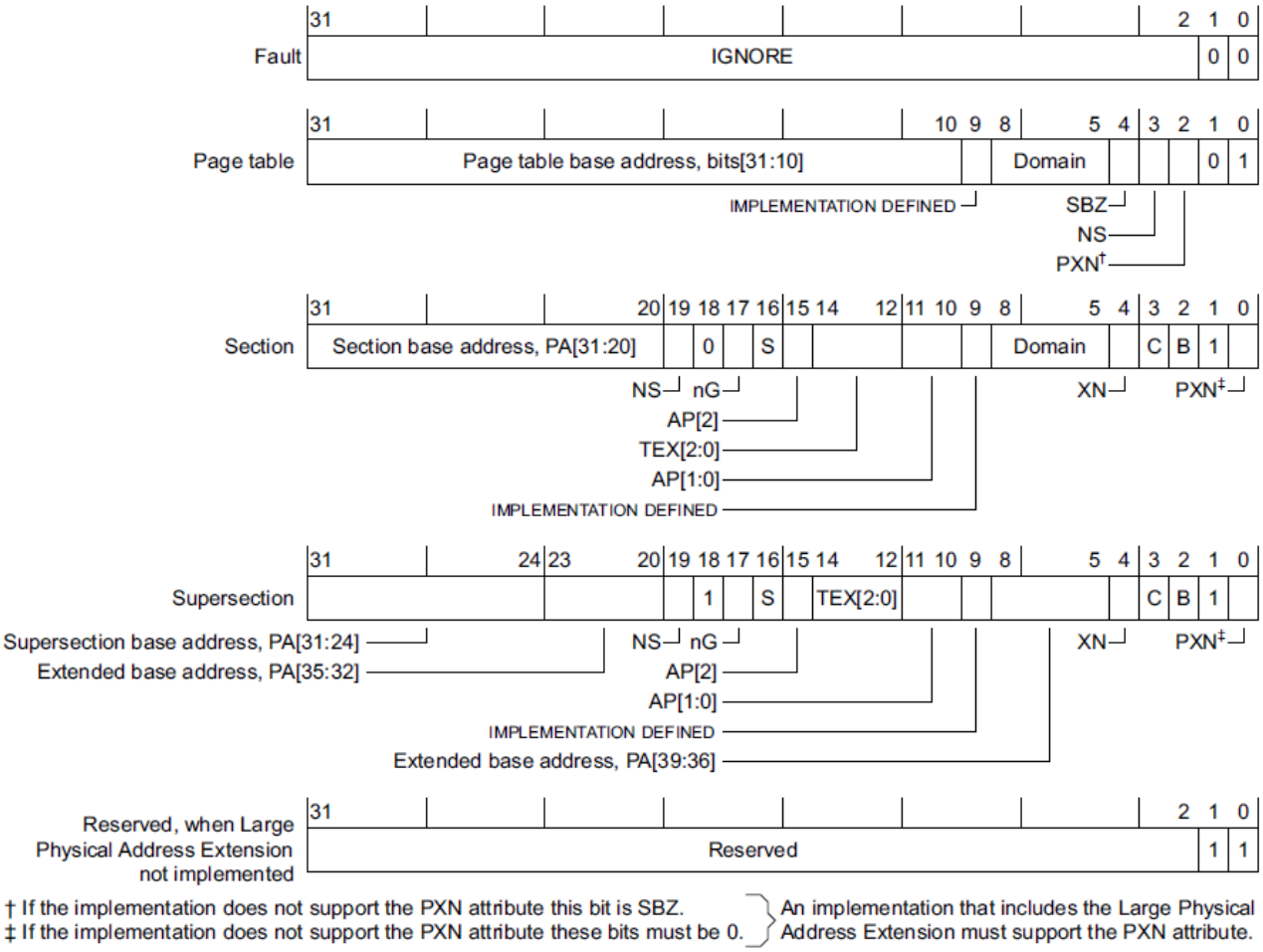


Figure B3-4 Short-descriptor first-level descriptor formats

Short-descriptor translation table second-level descriptor formats

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

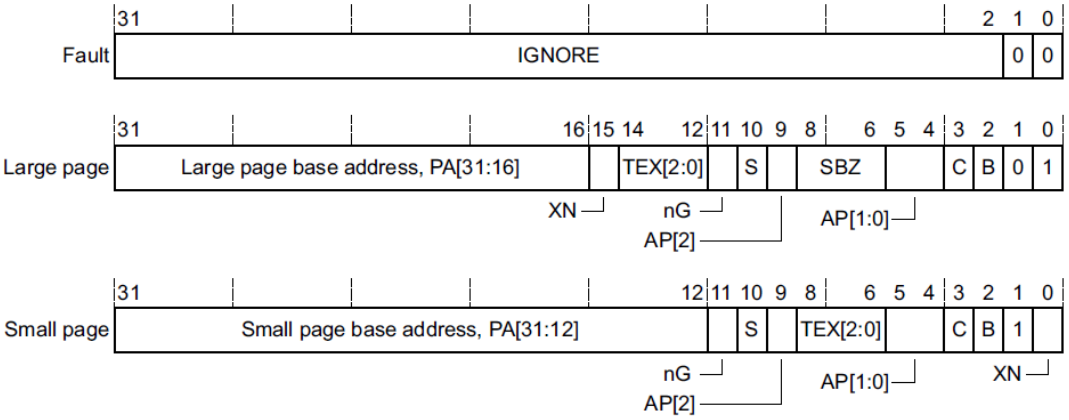


Figure B3-5 Short-descriptor second-level descriptor formats

4. The `money_transfer()` is in Secure World. The following code is solely for the purpose of calling `money_transfer()` from Normal world via `smc` instruction. Write Arm assembly code of **1)** performing the world-switching and **2)** calling `money_transfer()`. The `money_transfer()` should be executed under **Supervisor** mode in Secure world. You don't have to write code for the context-switching because it is only for calling `money_transfer()`. (**20 points**)

```
// Normal World

  smc #1  // call money_transfer()
  add  r0, r1, r0
  …
```

```
// Secure World

money_transfer:

  ldr r0, [r1]
  ldr r2, [r3]
  …
```

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

csd_SMC_Handler:

   // 1) World-switching to Secure World

   mrc   p15, 0, r0, c1, c1, 0  ; Read SCR
   bic   r0, r0, #b'1           ; Clear NS bit
   mcr   p15, 0, r0, c1, c1, 0  ; Write to SCR




   // 2)
   // 1. Call 'bank_transfer' function in Secure World
   // 2. 'bank_transfer' function should be executed
   //     under Supervisor mode in Secure world

   ldr   lr, =bank_transfer
   mov   r0, #0b10011
   msr   spsr, r0
   movs  pc, lr
```

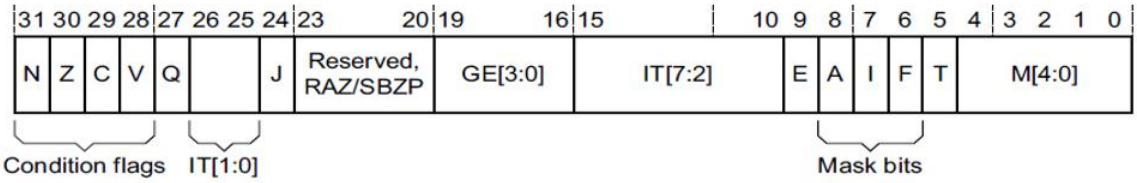
Accessing the SCR

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
```

Format of the CPSR and SPSRs

The CPSR and SPSR bit assignments are:



Processor mode		Encoding
User	usr	10000
FIQ	fiq	10001
IRQ	irq	10010
Supervisor	svc	10011
Monitor	mon	10110
Abort	abt	10111
Hyp	hyp	11010
Undefined	und	11011
System	sys	11111