COSE321 Computer Systems Design
Final Exam, Spring 2018

Name: _Solutions_

Note: No Explanations, No Credits!

1. The C code in (a) calls the function in (b). Convert the code in (a) to Thumb2 assembly, and the code in (b) to ARM assembly. Then, answer to the following questions. **(25 points)**

```
According to the ARM calling convention:

1. Arguments in function are passed via r0, r1, r2, and r3
2. Return value are passed via r0 and r1
```

(a) (10 points)

```
unsigned a, b;

 if (x ≥ y)
        a = x - y;
 else
        a = x + y;

 b = arm_C_test(a);

 a = b + 1;
```

(b) (8 points)

```
#pragma GCC target ("arm")

int arm_C_test(int a)
{
      int c;

      c = a + 0x11223344;

      return c;
}
```

↓

<Thumb2 assembly equivalent>

// you **must** use the conditional instruction (IT)
// Assume that r4 = x, r5 = y

```
0x68a:  cmp  r4, r5;
0x68c:  ITE  hs;
0x68e:  subhs  r4, r5
0x690:  addlo  r4, r5;
0x692:  mov   r0, r4
0x694:  blx  arm_C_test
0x698:  add   r0, 1
```

↓

<ARM assembly equivalent>

```
ldr  r1, =0x11223344
add  r0, r0, r1;
bx   lr
```

(c) (7 points = 3 + 4)

- What instruction did you use to call the function (`arm_C_test()`) in (a) and **why**?, and what instruction did you use to return to the caller in (b) and **why**?

  blx arm_C_test // to change to ARM mode.
  bx lr // to change to Thumb2 mode

- What will be stored in the link register (r14) after the function call to `arm_C_test()` in (a) and **why**? Assume the memory locations of the instructions in your answer in (a).

  lr (r14) = 0x699

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. Answer to the following questions (**30 points**)

a. Refer to the following **nested** exception case. The exception handlers are very simple as you can see. In fact, they don't do anything useful. **Explain the execution flow** of the code from **`main`**. Is there any problem (or issue) in the exception handlers, judging from the execution flow of your answer? If so, how would you fix the problem? (**10 points**)

```
csd_vector_table:
      b .
      b csd_undefined
      b csd_software_interrupt
      b .
      b .
      b .
      b .
      b .

main:

   cps   #0x10    // Change to User Mode
```

```
forever:
    svc #100
    nop
    b forever

csd_undefined:
      movs   pc, lr

csd_software_interrupt:

      .word 0xffffffff
      movs   pc, lr
```

```
// Assume that
//     (a) VBAR is set correctly to use our vector table (csd_vector_table).
//     (b) all stack pointers are already set appropriately.
```

Execution flow: main → sw interrupt → Undefined exception within sw interrupt handler → return to sw interrupt handler → return to return to main

No problem

b. Zynq's GIC can take 1020 different interrupts. For simplicity, let's assume that all 1020 interrupts come from I/O devices. The actual interrupt handler for each interrupt is given to you in the handler table below. Write the ARM assembly code in `csd_ISR` to jump to the appropriate handler, depending on the interrupt sources. Use **a small number of instructions as possible**. Refer to the next page for GIC registers if needed. **(10 points)**

```
csd_vector_table:
      b .
      b .
      b .
      b .
      b .
      b .
      b csd_IRQ
      b .


// Handler table for 1020 interrupt sources

src0: b intr_src0 // ISR for src#0
src1: b intr_src1 // ISR for src#1
src2: b intr_src2 // ISR for src#2
                         …

src1019: b intr_src1019 // ISR for src#1019
```

// For simplicity,
// let's assume that nested interrupt is **NOT** allowed.

csd_IRQ:

The GICC_IAR (Interrupt Acknowledge Register) contains the interrupt source number with the highest priority,

```
   ldr r1, = GICC_IAR
   ldr r1, [r1]
   ldr r2, =src0
   add r2, r2, r1, lsl #2
   blx  r2
```

c. What is the **difference** between software interrupt and software-generated interrupt (SGI) in Zynq? What are the **typical usage cases** of those interrupts? **How** would those interrupts be generated? **Answer with example codes.** Do not worry about the detailed bit fields in the SGI-related register. **(10 points)**

- Sotware interrupt is geneated with `svc` instruction. Used to implement system calls in OS

```
svc #100
```

- SGI is generated with `str` instruction. Use for IPI (inter-processor interrupt)

```
ldr r0, = GICD_SGIR
str  r1, [r0]
```

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.

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

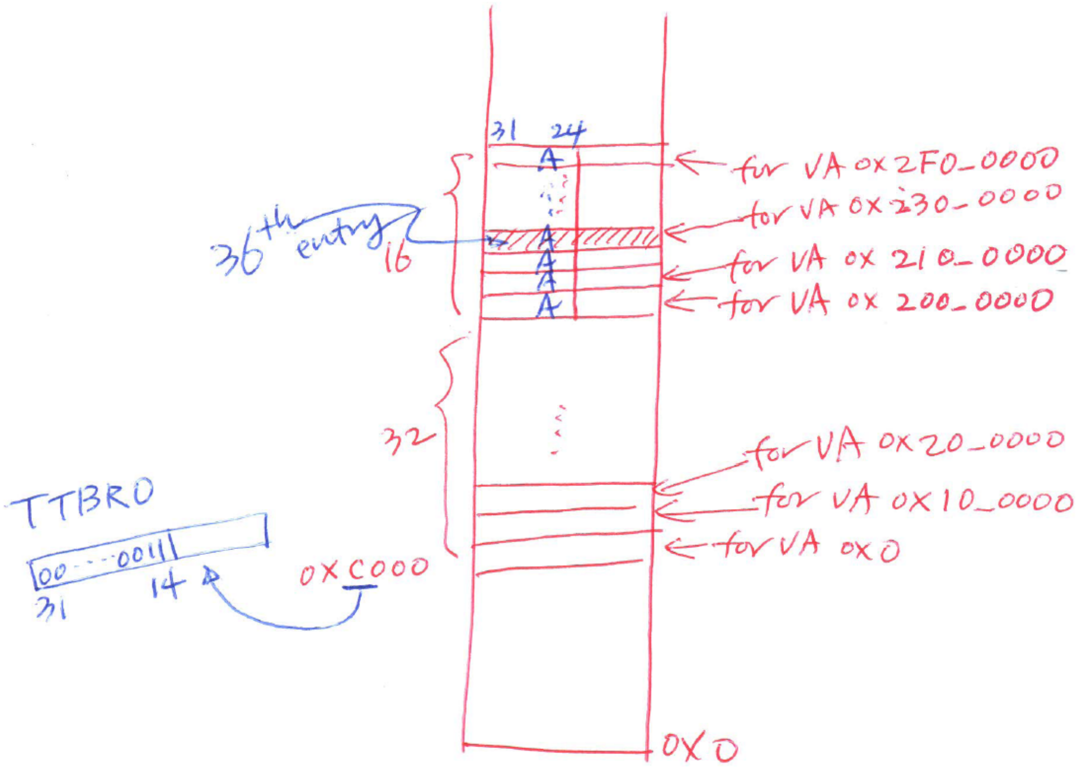
a. See the register description for more information.
b. Present in GICv1 if the GIC implements the GIC Security Extensions. Always present in GICv2.
c. GICv2 only.

3. Page Table & TLB (**30 points**)

   a. Cortex-A9 supports 4 different page sizes for virtual memory, and you want to use **only 16MB supersections**. How big is the page table size with a 32-bit virtual address and **why**? (**5 points**)

4K entries x 4 Bytes = 16KB

   b. You want to map **a virtual 16MB supersection from 0x0200_0000** to **a physical 16MB from 0x0A00_0000**. Draw a page table in memory as detailed as possible. Focus only on memory addresses when you draw the page table entries (that is, ignore the other bit fields for attributes). Specify the base location of the page table in the figure as well. Elaborate why you chose the base location. What value would you program to TTBR? Refer to the page table entry information in the next page. **(15 points)**



   c. Which entry in the page table is accessed and allocated into TLB if CPU executes the code below, and **why**? We assume that a TLB miss occurs when CPU executes the `ldr` instruction below. **(10 points)**

```
// assume that r0 = 0x02345678
      ldr r1, [r0]
```

0x**023**4_5678

36th entry

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.

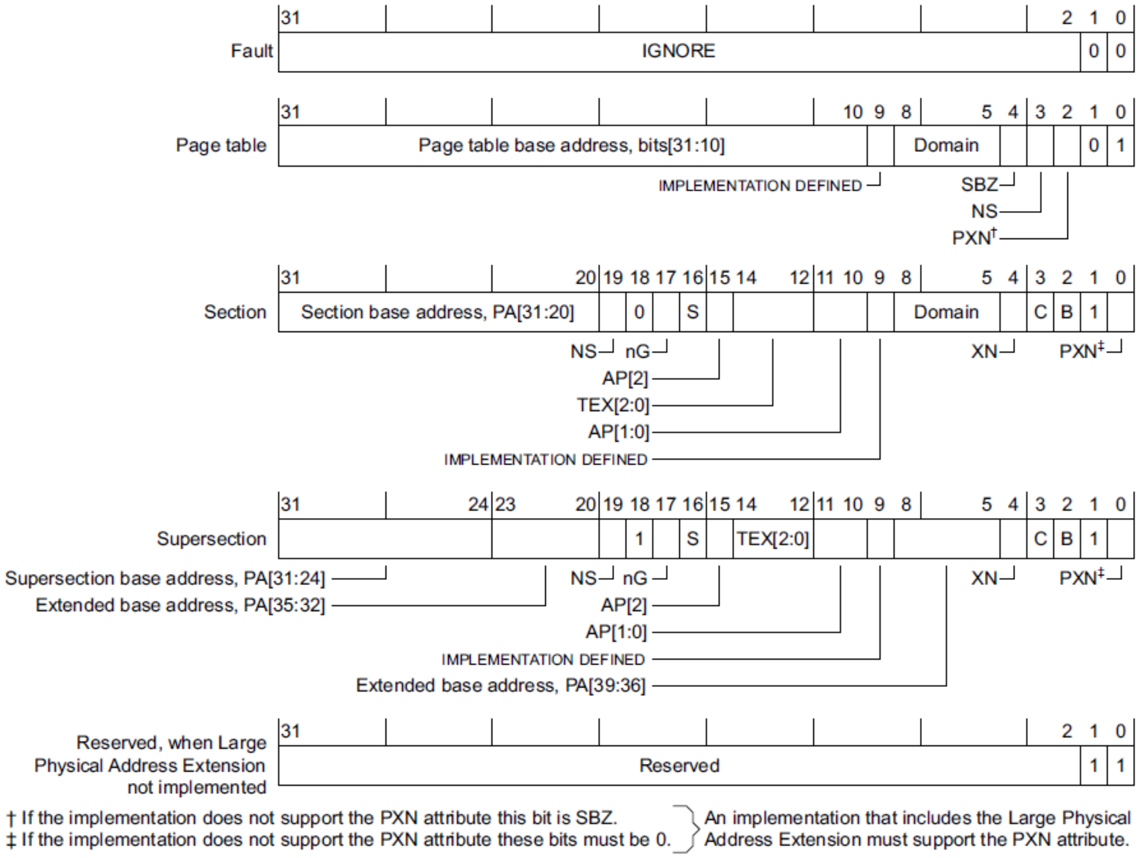


Figure B3-4 Short-descriptor first-level descriptor formats

Short-descriptor translation table second-level descriptor formats

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

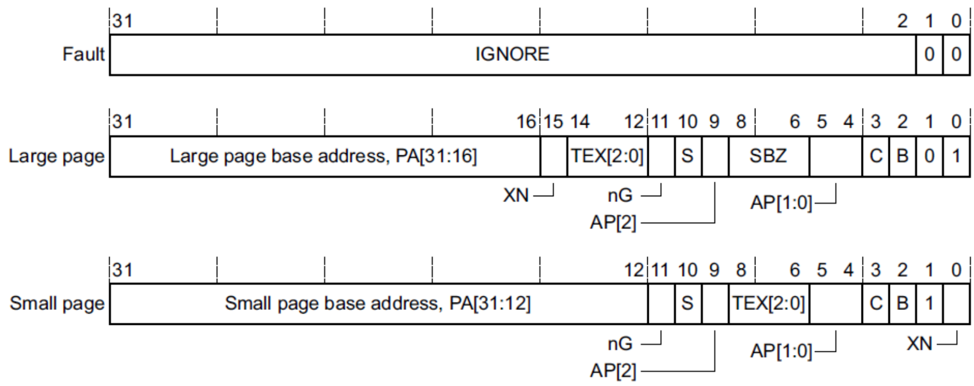
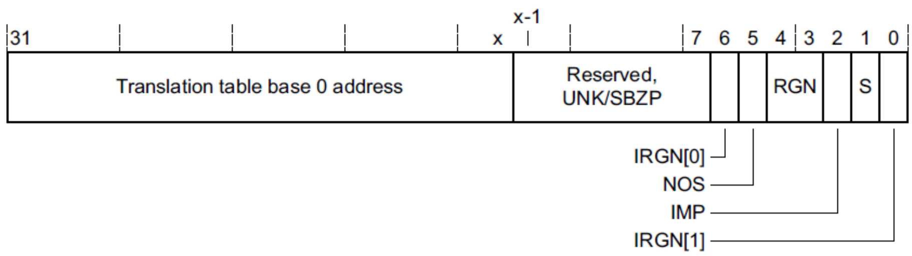


Figure B3-5 Short-descriptor second-level descriptor formats

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



4. The table below shows the worst-case scenario when executing the 3 instructions in Cortex-A9. The cache line size is 8 words (=32 bytes). **Determine the memory location** of each instruction and **explain** why you chose those locations. **What value would be in stack pointer** (sp) in the `ldmfd` instruction below to incur the worst case scenario? and **explain** your answer. **(15 points)**

Address	Instruction sequence	Worst case scenario in L1 Caches (I$ and D$)
0x01C	`sub r0, r1, r2`	• I$ miss (8-word line fill)
0x020	`mov r10, #0xA004`	• I$ miss (8-word line fill)
0x024	`ldmfd sp!, {r0-r12}`	• 3 D$ misses • For each miss, dirty line replacement and 8-word line fill

`sp` in `ldmfd` = 0x101C

For, the full-descending stack, sp in `ldm` = 0x101C
It is loading from the last data in a memory block and increment the address for the next data.
Note that it is a pop operation.