

EEE3095/6S Class Test 2 Solutions

10 October 2023

Duration: 90 minutes

Total: 80 marks

Empl ID: _____

Instructions:

- This is a closed-book test.
- There are five questions to this test, each of which contains sub-questions.
- Use a **pen** to complete the test — pencil will only be marked for drawings. Values indicated on drawings must be written in pen. Please show all of your working and reasoning clearly — your method and approach matter.
- Keep all your answers **within** the allocated block(s) for each question; anything outside of these blocks may not be marked.

Question	Available Marks	Grade
1	10	
2	20	
3	16	
4	15	
5	19	
Total:	80	

Question 1: Multiple Choice [10 marks]

- a. If we wanted to sum (with carry) the values in registers r0 and r1 into r2 while setting the Arithmetic Logic Unit (ALU) flags, which ARM Assembly instruction would we use? (2)
- ☐ adds r2, r0, r1
 - ☒ **adcs r2, r0, r1**
 - ☐ adcs r0, r1, r2
 - ☐ adc r2, r0, r1
- b. If data is pushed to the stack in an ARM Assembly function, then... (2)
- ☐ the frame pointer will increment
 - ☒ **the stack pointer will decrement**
 - ☐ the stack pointer will increment
 - ☐ the frame pointer will decrement
- c. In a flash ADC, when $h(t) > V_{ref}$ is decoded... (2)
- ☐ the converter is reset
 - ☐ the output corresponds to a logic '1'
 - ☐ the output corresponds to a logic '0'
 - ☒ **there will be an electrical problem**
- d. For a R2R Ladder DAC, what will be the V_{out} for a digital in of 1011_2 – assuming the D values are either 0 V or 5 V? (2)
- ☒ **3.4375 V**
 - ☐ 2.8125 V
 - ☐ 0 V
 - ☐ 5 V
- e. Consider that a and b are variables and initially $a = 1$. If you ran the commands "a += 3;" and "b = a + 7;" sequentially, what would be the value of b? (2)
- ☐ 4
 - ☐ 7
 - ☒ **11**
 - ☐ 8

Question 2: ARM Assembly [20 marks]

- a. If the following ARM Assembly code was run on a 32-bit processor, what would be the base-16 value in register **r0** after each line of code? Briefly explain your working. (5)

```
mov r0, 0x16
mov r1, #10
sub r0, r0, r1
lsl r0, #1
add r0, r0, 0b11
```

Solution: See below:

First mov operation sets $r0 = 0x16$ (22 in base-10)

Second mov operation does not affect r0, so $r0 = 0x16$ (and now $r1 = 10 = 0x0A$)

sub operation sets $r0 = r0 - r1 = 22 - 10 = 12 = 0x0C$

lsl operation sets $r0 = r0 \times 2^1 = 24 = 0x18$

add operation sets $r0 = r0 + 0b11 = 24 + 3 = 27 = 0x1B$

- b. Given the C code below, you are asked to write an equivalent program in ARM Assembly and then connect that code to your overall C program. Answer the following questions.

```
int divcomp (int a, int b) {
    int res;
    res = (a + b)/4;
    if (a > res) res = 1;
    return res;
}
```

- (i) Provide two reasons why, in general, we may want to combine ARM Assembly with C code. (2)

Solution: Any of the following (or equivalently valid) reasons:

- Interfacing an Assembly program with C libraries (or vice-versa)
- Performance – writing speed-critical key parts of a program in Assembly, and leaving the rest in C, may boost overall performance
- Size – human-written Assembly can be smaller than compiler-generated code
- Doing something very fast and specialised (through Assembly) as a tiny portion of your (larger) C program



- (ii) Write an appropriate function prologue for the C function in ARM Assembly; you do not need to include any Assembly directives ("dot" keywords) or labels, nor reserve any stack space for local variables. (4)

Solution: "divcomp" label is included here but is not explicitly necessary. Similarly, students could include Assembly directives (such as ".global") and allocate space for local variables on the stack, but these are not required.

```
divcomp:
    mov ip, sp
    stmfd sp!, {fp, ip, lr, pc}
    sub fp, ip, #4
```

- (iii) Write an appropriate function body for the C function in ARM Assembly; as this is a very simple program, you do not need to perform any load/store operations with the stack. (7)

Solution: By default, *a* should be loaded into r0 and *b* into r1. Therefore, the function body should look something like:

```
(prologue here)
    add r2, r0, r1 @ res = r2 = a + b
    asr r2, r2, #2 @ res = res / 4; signed (due to int return type)
    cmp r0, r2 @ compare r0 to r2, i.e. a vs res
    bgt Greater @ branch to Greater if a > res
    mov r0, r2 @ set r0 = r2 = res for the return
    b Return @ branches to return; default option
Greater:
    mov r0, #1 @ set res = 1 for the return
Return:
(epilogue here)
```



- (iv) Write an appropriate function epilogue for the C function in ARM Assembly.

(2)

Solution: Since r0 already has the return value – either r2 or the value 1 (if the Greater branch was taken) – the epilogue only needs to do the "return" to the calling function:

```
ldmea fp, {fp, sp, pc}
```

Question 3: ADCs [16 marks]

- a. Suppose that we are working with a successive approximation-based 4-bit ADC. The input voltage range extends from $V_{\min} = 1 \text{ V}$ ($= "0000"$) to $V_{\max} = 5 \text{ V}$ ($= "1111"$). Which two steps are used to convert the input voltage to a corresponding digital output signal? You may use pseudocode to explain your answer.

(4)

Solution: Two steps:

1. Initialisation:

Start by setting the most significant bit (MSB) ADC to "1" (i.e., "1000") in the output code to test the impact of the bit on the ADC output.

2. Compare the output to the input:

After setting each test bit, compare the current output code to the input voltage. If the output voltage is less than or equal to the input voltage, keep the bit set to "1" and move to the next most significant bit. Otherwise, set the bit back to "0".

Using pseudocode:

```
set MSB="1" (with all others ="0")
{
  if too large:
    reset MSB, set MSB - 1 = "1";
    if too large:
      reset MSB-1, set MSB - 2 = "1";
      etc...;
  else if too small:
    set MSB - 1 = "1"...
}
```

- b. Is a pipeline Flash ADC likely to be quicker than an SA-ADC? Explain your answer.

(4)

Solution: Yes, it does the process in one go – about $\log_2(n)$ times faster than the SA-ADC. The voltage input to the FAT flash is tested by all $n - 1$ comparators in one go. Whereas for the SA-ADC, the voltage moves down the line being tested (for too high or too low) in each stage until it gets to the last stage and final bit decision.

c. With the aid of appropriate figures, depict the following ADC metrics:

(i) Gain error

(4)

Solution: See Figure 1 below.

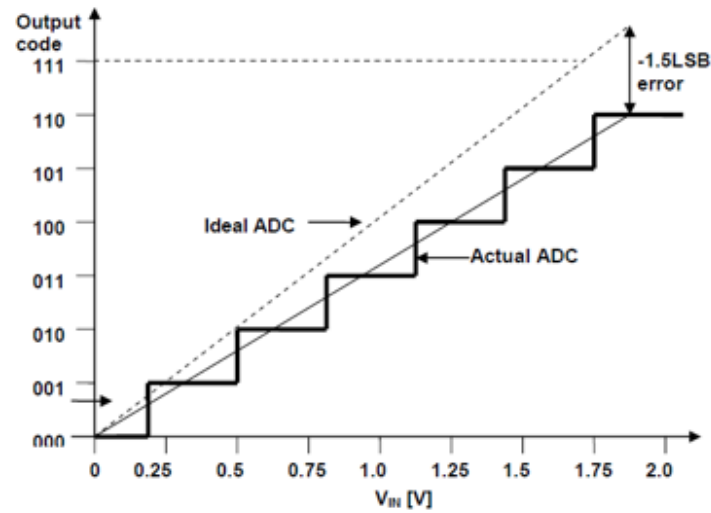


Figure 1: Gain error

(ii) Differential Non-Linearity (DNL)

(4)

Solution: See Figure 2 below.

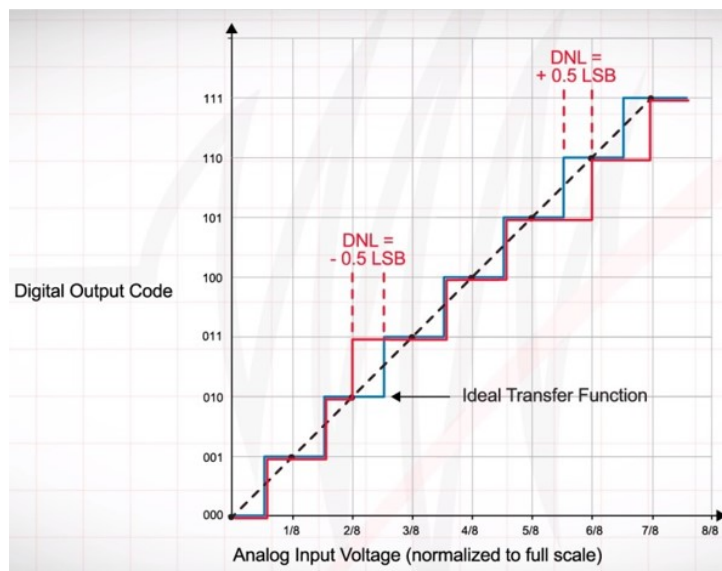


Figure 2: DNL

Question 4: Finite State Machines [15 marks]

- a. As part of embedded system design, engineers often use computational models to map out and visualise the syntax and rules of a system before the embedded code is written. List any 5 embedded models commonly used in embedded systems design. (5)

Solution: 1. Data Flow Graph (DFG) model
2. State Machine model
3. Concurrent Process model
4. Sequential Program model
5. Object Oriented model

- b. Consider an embedded system for a driver/passenger seat-belt warning system in an automotive. Given that the system requirements are captured as:
1. When the vehicle ignition is turned on and the seat belt is not fastened within 10 seconds of ignition ON, the system generates an alarm signal for 5 seconds.
 2. The alarm is turned off when the alarm time (5 seconds) expires, or when the driver/passenger fastens the belt or if the ignition is turned off – whichever happens first.
- (i) What are the states and events of the FSM? (4)

Solution: States: Alarm off, waiting, alarm on
Events: Ignition key on, ignition key off, timer expire, alarm time expire and seat belt on.



(ii) Draw the full FSM design diagram.

(6)

Solution: See Figure 3 below.

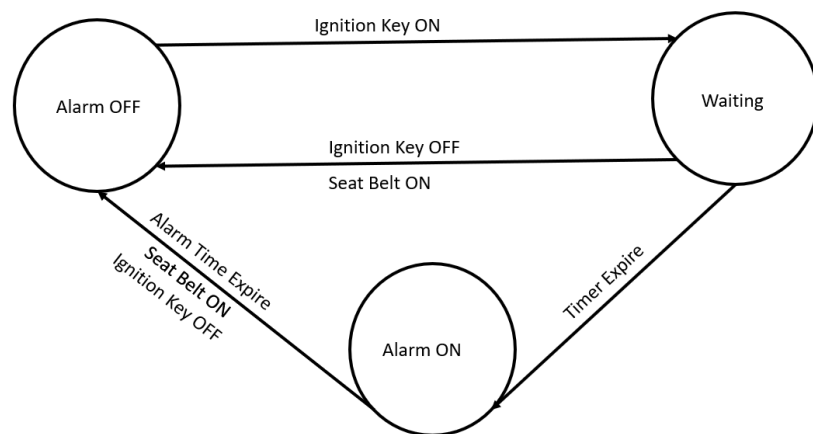


Figure 3: FSM

Question 5: Interrupts and Exceptions [19 marks]

- a. List the 7 modes of operation for an ARM processor.

(7)

Solution:

1. User Mode: applications run in this mode
2. Interrupt ReQuest (IRQ): mode switched to when low priority interrupt is signalled
3. Fast Interrupt reQuest (FIQ): used when a high priority interrupt is signalled
4. Supervisor: on software interrupt/reset
5. Abort: memory exceptions
6. Undef: handles undefined instructions
7. System Mode: privileged mode (for OS); similar to user mode with fewer restrictions

- b. Interrupts are usually set up at system start-up. List and describe the four operations for setting up an Interrupt Service Routine (ISR) on an ARM processor.

(8)

Solution:

- Set up ARM exception vector:
The ARM Exception Vector is a table that contains the addresses of various exception handlers, including interrupt service routines (ISRs). Setting up this involves configuring the base address of this table in the processor's control registers.
- Configure the interrupt controller:
Specifying which interrupts are enabled and their priorities.
- Set status flags as needed:
Specifying status flags to communicate information or conditions to the ISR so that it differentiates them accordingly (i.e., condition, control or error).
- On ARM processor: clear IRQ and FIQ bits of CPSR:
The Current Program Status Register (CPSR) contains several bits to enable the processor to respond to interrupt requests, so we need to clear the IRQ and FIQ bits in the CPSR by setting them to "0".



- c. Why is the use of the Advanced Interrupt Controller (AIC) more advantageous than relying on the built-in ARM interrupt lines when developing complex embedded systems? Provide four reasons.

(4)

Solution:

- Allows for 8-level priority control (individually maskable and vectored)
- Up to 32 interrupt sources
- Set status flags as needed
- Reduces software and real-time overhead in handling internal and external interrupts

END OF TEST

ARM Assembly Language Cheatsheet

Memory access instructions

```
LDR Rd, [Rn]           ; load 32-bit number at [Rn] to Rd
LDR Rd, [Rn,#off]      ; load 32-bit number at [Rn+off] to Rd
STR Rt, [Rn]           ; store 32-bit Rt to [Rn]
STR Rt, [Rn,#off]      ; store 32-bit Rt to [Rn+off]
PUSH {Rt}              ; push 32-bit Rt onto stack
POP {Rd}               ; pop 32-bit number from stack into Rd

MOV{S} Rd, <op2>       ; set Rd equal to op2
MOV Rd, #iml6           ; set Rd equal to iml6, iml6 is 0 to 65535
MVN{S} Rd, <op2>       ; set Rd equal to -op2
```

Branch instructions

```
B label                ; branch to label Always
BEQ label              ; branch if Z == 1 Equal
BNE label              ; branch if Z == 0 Not equal
BCS label              ; branch if C == 1 Higher or same, unsigned ≥
BHS label              ; branch if C == 1 Higher or same, unsigned ≥
BCC label              ; branch if C == 0 Lower, unsigned <
BLO label              ; branch if C == 0 Lower, unsigned <
BMI label              ; branch if N == 1 Negative
BPL label              ; branch if N == 0 Positive or zero
BVS label              ; branch if V == 1 Overflow
BVC label              ; branch if V == 0 No overflow
BHI label              ; branch if C==1 and Z==0 Higher, unsigned >
BLS label              ; branch if C==0 or Z==1 Lower or same, unsigned ≤
BGE label              ; branch if N == V Greater than or equal, signed ≥
BLT label              ; branch if N != V Less than, signed <
BGT label              ; branch if Z==0 and N==V Greater than, signed >
BLE label              ; branch if Z==1 or N!=V Less than or equal, signed ≤
BX Rm                  ; branch indirect to location specified by Rm
BL label               ; branch to subroutine at label
BLX Rm                 ; branch to subroutine indirect specified by Rm
```

Logical instructions

```
AND{S} {Rd}, Rn, <op2> ; Rd=Rn&op2 (op2 is 32 bits)
ORR{S} {Rd}, Rn, <op2> ; Rd=Rn|op2 (op2 is 32 bits)
EOR{S} {Rd}, Rn, <op2> ; Rd=Rn^op2 (op2 is 32 bits)
BIC{S} {Rd}, Rn, <op2> ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S} {Rd}, Rn, <op2> ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S} Rd, Rm, Rs      ; logical shift right Rd=Rm>>Rs (unsigned)
LSR{S} Rd, Rm, #n      ; logical shift right Rd=Rm>>n (unsigned)

ASR{S} Rd, Rm, Rs      ; arithmetic shift right Rd=Rm>>Rs (signed)
ASR{S} Rd, Rm, #n      ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S} Rd, Rm, Rs      ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n      ; shift left Rd=Rm<<n (signed, unsigned)
```

Arithmetic instructions

```
ADD{S} {Rd}, Rn, <op2> ; Rd = Rn + op2
ADD{S} {Rd}, Rn, #iml2 ; Rd = Rn + iml2, iml2 is 0 to 4095
SUB{S} {Rd}, Rn, <op2> ; Rd = Rn - op2
SUB{S} {Rd}, Rn, #iml2 ; Rd = Rn - iml2, iml2 is 0 to 4095
RSB{S} {Rd}, Rn, <op2> ; Rd = op2 - Rn
RSB{S} {Rd}, Rn, #iml2 ; Rd = iml2 - Rn
CMP Rn, <op2>          ; Rn - op2 sets the NZVC bits
MUL{S} {Rd}, Rn, Rm     ; Rd = Rn * Rm signed or unsigned
```

Notes

Ra Rd Rm Rn Rt represent 32-bit registers

value any 32-bit value: signed, unsigned, or address

{S} if S is present, instruction will set condition codes

#im12 any value from 0 to 4095

#im16 any value from 0 to 65535

{Rd,} if Rd is present Rd is destination, otherwise Rn

#n any value from 0 to 31

#off any value from -255 to 4095

label any address within the ROM of the microcontroller

op2 the value generated by <op2>