University of Toronto
Faculty of Applied Science and Engineering

Midterm Test
March 9, 2023

ECE243 – Computer Organization

Examiners – Prof. Stephen Brown and Prof. Jonathan Rose

SOLUTIONS

1. There are **5** questions and **22** pages. Do **all** questions. The test duration is 105 minutes (1:45 hours).
2. **ALL WORK IS TO BE DONE ON THESE SHEETS.** You can use the blank pages included on Pages 16 – 18 if you need more space for any question. Be sure to indicate clearly if your work continues elsewhere.
3. Closed book. **Aid Sheets** are included for your reference starting on Page 20. We suggest that you detach the last 2 physical pages (pages 19-22). Page 19 is for use in question 5, but does not need to be handed in.
4. No calculators are permitted.

[11 marks] 1. Short answers:

[8 marks] (a) Consider the sequence of ARM instructions shown below. Fill in the comment field to the right of each instruction to show the contents of the destination register **after** the instruction on that line of code has been executed. Show your answers in hexadecimal.

```
        .global     _start
_start: LDR         R0, =DATA
        MOV         R10, R0

        LDR         R1, [R0]            // R1 = 0x11110000

        LDR         R2, [R0, #8]        // R2 = 0x33332222

        LDR         R3, [R0, #4]!       // R3 = 0x22221111

        LDR         R4, [R0], #4        // R4 = 0x22221111

        LDRB        R5, [R10, #3]       // R5 = 0x00000011

        LDRB        R6, [R10, #0xC]     // R6 = 0x00000033

        MOV         R7, #0xFF           // R7 = 0x000000FF
        STRB        R7, [R10]
        LDR         R8, [R10]           // R8 = 0x111100FF

END:    B           END

DATA:   .word       0x11110000
        .word       0x22221111
        .word       0x33332222
        .word       0x44443333
```

[3 marks] (b) In Lab Exercise 4, the ARM instruction that you used to return from an interrupt is

```
SUBS    PC, LR, #4
```

Answer the following question. Instead of using the above `SUBS` instruction, could you instead return from an interrupt using the following sequence of instructions? That is, would these instructions have exactly the same effect? Explain your answer.

```
SUBS    LR, #4
MOV     PC, LR
```

Answer:

These two instructions do not have the same exact effect. The second group does not restore the saved CPSR from the IRQ mode's SPSR, which restores flags, interrupt mask bits, and mode bits, allowing the interrupted code to proceed correctly.

[8 marks] 2. Short coding:

[4 marks] (a) The ARM code below loads into `R0` the number at address `X` and then calls the subroutine named `LOG2`. The number `X` could be any positive integer > 0 that is a power of 2. The `LOG2` subroutine is supposed to return in register `R0` the base 2 logarithm, $\log_2 X$. Write the `LOG2` subroutine in the space provided below.

Answer:

```
        .global     _start
_start: LDR         SP, =0x20000
        LDR         R0, =X
        LDR         R0, [R0]

        BL          LOG2

END:    B           END

LOG2:   MOV         R2, #0      // R2 will be log2(R0)
CONT:   LSR         R0, #1
        CMP         R0, #0
        BEQ         DONE
        ADD         R2, #1      // count # shifts
        B           CONT

DONE:   MOV         R0, R2      // return log2 in R0
        MOV         PC, LR

X:      .word       64          // example data
```

[4 marks] (b) The ARM code below loads into `R0` and `R1` the two numbers at addresses `X` and `Y`, respectively. The code then calls the subroutine named `MASK`. The number `X` could be any positive integer > 0 and `Y` can be any integer from 1 to 32. The `MASK` subroutine is supposed to return in register `R0` the `Y` least-significant bits of `X`. Write the `MASK` subroutine in the space provided below. For the example given in the code, where $X = 0\text{x}63$ and $Y = 4$, the `MASK` subroutine should return the result 3.

Answer:

```
            .global     _start
_start:     LDR         SP, =0x20000
            LDR         R0, =X
            LDR         R1, [R0, #4]     // load Y
            LDR         R0, [R0]         // load X

            BL          MASK

END:        B           END

MASK:       MOV         R2, #1        // mask
            LSL         R2, R1        // mask << Y
            SUB         R2, #1        // mask = Y 1 bits
            AND         R0, R2        // return Y lsb of R0
            MOV         PC, LR

X:          .word       0x63
Y:          .word       4

// Other soln:

MASK:       MOV         R2, #1        // init mask
LOOP:       SUBS        R1, #1        // count mask bits
            BEQ         DONE
            LSL         R2, #1        // extend mask
            ORR         R2, #1
            B           LOOP

DONE:       AND         R0,R2
            MOV         PC, LR
```

[15 marks] 3. Consider the C code shown below.

```
int rand (int, int);

volatile int *Timer_ptr = (int *) 0xFFFEC600;
volatile int *LEDR_ptr = (int *) 0xFF200000;
volatile int *KEY_ptr = (int *) 0xFF200050;

int main()
{
    int press, value;
    *Timer_ptr = 200000000;
    *(Timer_ptr + 2) = 3;

    while (1) {
        press = *(KEY_ptr + 3);
        if (press) {
            value = rand (press, 100);
            *LEDR_ptr = value;
            *(KEY_ptr + 3) = press;
        }
    }
}

int rand(int even, int range){
    int local;
    local = *(Timer_ptr + 1);
    local = local % range;

    if (even == 1) local = local & 0xFE;
    else local = local | 1;
    return local;
}
```

[1 mark] (a) The ARM A9 Private Timer is used in this program. How long (in seconds) does it take the timer to count down to zero?

Answer

1 second is correct for the 200MHz clock going into the timer;
other values were accepted if different frequency of clock given

[4 marks] (b) Explain, briefly, what this program "does." That is, if you were to execute this program, using the *CPUlator* or on a *DE1-SoC* board, what would the program display on the LEDR port?

Answer

When a pushbutton KEY is pressed, the program displays a number between 0 and 99, generated from the timer, on the LEDR port. If KEY 0 is pressed, then the displayed number will be even, else for any other KEY the displayed number will be odd.

[5 marks] (c) In this part you are to translate only the **main** function from the C program into ARM assembly language code. You are given part of the solution on the following page. Fill in the rest of the code. Make sure to follow the ARM Procedure Call Standard (PCS) in your code. For calling the `rand()` subroutine (which you will be translating in part (d) of this question), pass the `press` argument in register `R0`, and pass the constant 100 argument in register `R1`. Make your assembly code as simple as possible, and provide comments that help to illustrate how your assembly code corresponds to the orginal C code.

Put your answer on the next page.

Answer:

```
        .global _start
_start:
// set up the Timer
MAIN:   LDR     SP, =0x20000        // stack
        LDR     R12, =0xFFFEC600    // ARM A9 Private Timer address
        LDR     R0, =200000000      // 1/(200 MHz) x 200 M = 1 sec
        STR     R0, [R12]           // write to timer load register
        MOV     R0, #0b011          // mode = 1 (auto), enable = 1
        STR     R0, [R12, #0x8]     // start timer

        LDR     R4, =0xFF200000     // I/O Base Address

WHILE:  LDR     R0, [R4, #0x5C]     // R0 = press (EdgeCapture)
        CMP     R0, #0              // if (press)
        BEQ     WHILE
        MOV     R1, #100
        MOV     R5, R0              // save R0
        BL      RAND                // RAND (R0, R1)
        STR     R0, [R4]            // *LEDR_ptr = value
        MOV     R0, R5              // restore R0
        STR     R0, [R4, #0x5C]     // *(KEY_ptr + 3) = press
        B       WHILE
```

[5 marks] (d) In the space below, write assembly code for the `rand()` subroutine. Its `even` parameter is passed in `R0`, and its `range` parameter in `R1`. To implement the C *modulus* operator `%` the RAND subroutine should call the MOD subroutine that is provided at the bottom of this page.

Answer:

```
// parameters are in R0, R1
RAND:   PUSH    {R0, R1, LR}        // save parameters, LR
        LDR     R0, [R12, #4]       // local = *(Timer_ptr + 1)
        BL      MOD                 // local = local % range
        MOV     R2, R0              // R2 = local
        POP     {R0, R1, LR}        // restore parameters, LR
        CMP     R0, #1              // if (even)
        ANDEQ   R2, #0xFE
        ORRNE   R2, #1
        MOV     R0, R2              // return local

        MOV     PC, LR

// returns the modulus R0 = R0 % R1
MOD:    CMP     R0, R1              // n - i < 0?
        BLT     ENDM
        SUB     R0, R1              // n -= i
        B       MOD
ENDM:   MOV     PC, LR              // modulus is in R0
```

[12 marks] 4. As part of Lab Exercise 2 in this course the you were asked to write a program to find the largest sequence of 1's in a list of data *words*. An attempted solution to this problem is given below. In this solution (although not done in the Lab 2 version) the final answer is displayed on the LEDR lights.

```
             .global _start
_start:
             LDR     R4, =TEST_NUM
             LDR     R6, =0xFF200000
             MOV     R5, #0       // R5 will hold the result
MAIN_LOOP:   LDR     R0, [R4]
             CMP     R0, #0       // done ?
             BEQ     END_ONES
             BL      ONES
             CMP     R5, R1
             MOVLT   R5, R1
             ADD     R4, #4
             STR     R5, [R6, #0x20]
             B       MAIN_LOOP
END:         B       END

ONES:        MOV     R1, R0
             MOV     R0, #0
LOOP:        CMP     R1, #0
             BEQ     END_ONES
             LSR     R2, R0, #1
             AND     R1, R1, R2
             ADD     R0, #1
             B       ONES
END_ONES:    MOV     PC, LR

TEST_NUM:    .word   0x103fe00f    // the data
             .word   0x3fabedef
             .word   0x00000001
             .word   0x75a5a5a5
             .word   0x01ffC000
             .word   0x03ffC000
             .word   0x11111111
             .word   0              // end of data

             .end
```

The above program contains a number of logical errors. In the space on the following page, provide a corrected version of the code. You can either show all of the code, or else show only the lines of code that you corrected. Either way, indicate clearly where you have made changes to the code, for example by using the line numbers shown in the code, or encircling/underlining your corrections. Do *not add any additional lines of code* to fix the errors; just correct the errors in the code that is there.

There are no errors in lines 1 to 5, or 27 to 36.

PROVIDE YOUR CORRECTED CODE IN THE SPACE BELOW:

```
1                .global _start
2   _start:
3                LDR      R4, =TEST_NUM
4                LDR      R6, =0xFF200000
5                MOV      R5, #0       // R5 will hold the result
6   MAIN_LOOP:   LDR      R0, [R4]
7                CMP      R0, #0       // done ?
8                BEQ      END
9                BL       ONES
10               CMP      R5, R0
11               MOVLT    R5, R0
12               ADD      R4, #4
13               STR      R5, [R6]
14               B        MAIN_LOOP
15  END:         B        END
16
17  ONES:        MOV      R1, R0
18               MOV      R0, #0
19  LOOP:        CMP      R1, #0
20               BEQ      END_ONES
21               LSR      R2, R1, #1
22               AND      R1, R1, R2
23               ADD      R0, #1
24               B        LOOP
25  END_ONES:    MOV      PC, LR
26
27  TEST_NUM:    ...
```

[11 marks] 5. As part of Lab Exercise 5 in this course the you were asked to write a program that draws an animation on the VGA screen. In this question you are asked to write a similar program, making an animation with a number of square boxes that "move" vertically up and down on the screen. In the same way that you did for Lab 5, you are to use double-buffering for your animation. Some parts of the required C code are provided for you, starting on the next page and on **Page 19**. You are to fill in the missing lines of code.

Your code should use the subroutines in the code provided on **Page 19**, just before the **Aid Sheets**. The provided subroutines are called `init_boxes()`, `clear_screen()`, `plot_pixel()` and `wait_for_vsync()`. You are encouraged to **detach** Pages 19 to 22 of the test, for ease of reference. Keep these pages after the test (you should not hand them in).

Your animation involves 12 square boxes. The main program first finds random locations for each of these boxes, using the provided subroutine `init_boxes()` on Page 19. This subroutine also sets a variable `dy_box` for each box to either -1 or 1, which causes each box in the animation to move up or down on the screen. Also, a random color from the set red, green, or blue, is set for each box.

Next, the main program has to set up the `DMA` controller so that it uses two pixel buffers. Part of this code is provided on the next page, but you need to write additional code (indicated in the partial solution with the comment `// finish DMA setup in the space below ...`) to complete the setup of the DMA controller. Note that the code for the `clear_screen()` subroutine is provided for you, as is the code for the `plot_pixel()` subroutine on Page 19.

The main part of the animation is in the `while` loop. The first few lines of code in this loop are provided for you. This code calls a function `draw_box()`, to draw each box on the pixel buffer. You will write the code for `draw_box()` in part (*c*) of this question.

Write the rest of the required code for the animation in the while loop that makes the boxes appear to move vertically up and down on the VGA screen. Be sure to check for edge conditions, so that boxes appear to "bounce" off the bottom and top of the screen (like you did for your animations in Lab 5). Also, be sure to synchronous each frame of your animation with the DMA controller using `wait_for_vsync()`. The code for `wait_for_vsync()` is provided for you (on Page 19).

The C code for the required solution starts on the next page.

[3 marks] (a) Fill in your code for setting up the DMA in the space a t the bottom of this page.

```
#include <stdlib.h>          // needed for rand()
/* subroutine prototypes */
void init_boxes(void);
void clear_screen(void);
void draw_box(int, int, short int);
void plot_pixel(int, int, short int);
void wait_for_vsync(void);

#define NUM_BOXES 12     // number of boxes in the animation
#define SIZE_BOX 8       // width & height of each box in pixels

int x_box[NUM_BOXES], y_box[NUM_BOXES];    // box (x, y)
int dy_box[NUM_BOXES];                     // box delta-y
int color_box[NUM_BOXES];                  // box color
unsigned int color[] = {0xF800, 0x07E0, 0x001F};   // colors

int pixel_buffer_start; // specifies which memory is currently
                        // being used as the back buffer.
int main(void)
{
    int i;
    volatile int * pixel_ctrl_ptr = (int *) 0xFF203020; // DMA

    init_boxes();

    *(pixel_ctrl_ptr + 1) = 0xC8000000;
    pixel_buffer_start = *(pixel_ctrl_ptr + 1);
    clear_screen();
    // finish DMA setup in the space below ...
```

Answer:

```
    /* now, swap the BackBuffer and Buffer, which initializes
       the Buffer */
    wait_for_vsync();

    *(pixel_ctrl_ptr + 1) = 0xC0000000; // re-initialize
       BackBuffer
    pixel_buffer_start = *(pixel_ctrl_ptr + 1); // we draw on
       the back buffer
```

[5 marks] (b) The C code for main program continues below. Fill in the missing code.

Answer:

```
    while (1) {
        clear_screen();      // erase previous frame

        for (i = 0; i < NUM_BOXES; i++) {
            draw_box(x_box[i], y_box[i], color_box[i]);
        }
        for (i = 0; i < NUM_BOXES; i++) {
            y_box[i] += dy_box[i];       // move up or down

            if (y_box[i] < 0) {
                y_box[i] = 0;
                dy_box[i] = -dy_box[i];
            }
            else if (y_box[i] + SIZE_BOX >= 239) {
                y_box[i] = 239 - SIZE_BOX;
                dy_box[i] = -dy_box[i];
            }
        }
        wait_for_vsync();    // synchronize, and swap buffers
        pixel_buffer_start = *(pixel_ctrl_ptr + 1); // update back
           buffer pointer
    }    // end of while loop
} // end of main
```

[3 marks] (c) Put your code for the `draw_boxes()` subroutine in the space below. Draw each box as a *square* that is *filled* with the box's color. Each box is `SIZE_BOX` pixels in width and `SIZE_BOX` pixels in height.

Answer:

```
void draw_box(int x0, int y0, short int color) {
    int x, y;

    for (x = x0; x <= x0 + SIZE_BOX; x++)
        for (y = y0; y <= y0 + SIZE_BOX; y++)
            plot_pixel (x, y, color);
}
```

Extra answer space for any question on the test, if needed:

Extra answer space for any question on the test, if needed:

Extra answer space for any question on the test, if needed:

These subroutines are provided for you as part of **Question 5**.

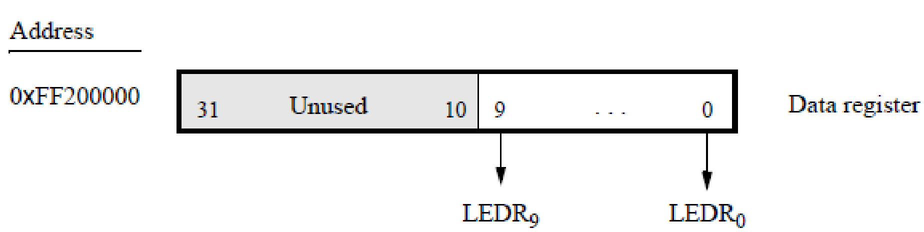
```
void init_boxes() {
    int i;
    for (i = 0; i < NUM_BOXES; i++) {
        x_box[i]  = (rand() % (320 - SIZE_BOX));   // random x
        y_box[i]  = (rand() % (240 - SIZE_BOX));   // random y
        dy_box[i] = ((rand() % 2) * 2) - 1;        // 1 or -1
        color_box[i] = color[(rand() % 3)];        // random color
    }
}

void clear_screen() {
    int y, x;

    for (x = 0; x < 320; x++)
        for (y = 0; y < 240; y++)
            plot_pixel (x, y, 0);
}

void plot_pixel(int x, int y, short int color) {
    *(short int *)(pixel_buffer_start + (y << 10) + (x << 1)) =
       color;
}

void wait_for_vsync() {
    volatile int * pixel_ctrl_ptr = (int *) 0xFF203020; // DMA
    int status;

    *pixel_ctrl_ptr = 1; // start the synchronization process

    status = *(pixel_ctrl_ptr + 3);
    while ((status & 0x01) != 0)
        status = *(pixel_ctrl_ptr + 3);
}
```

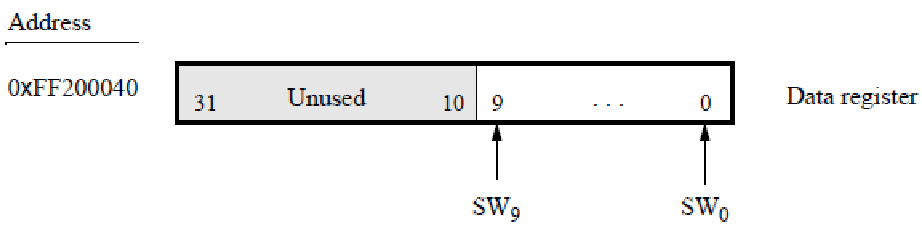
Aid Sheet

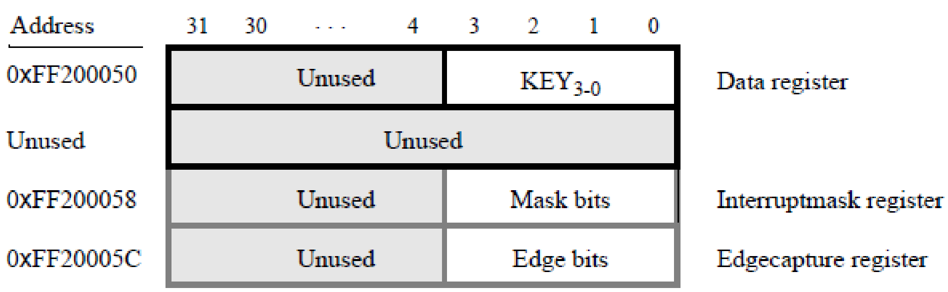
ARM Addressing Modes

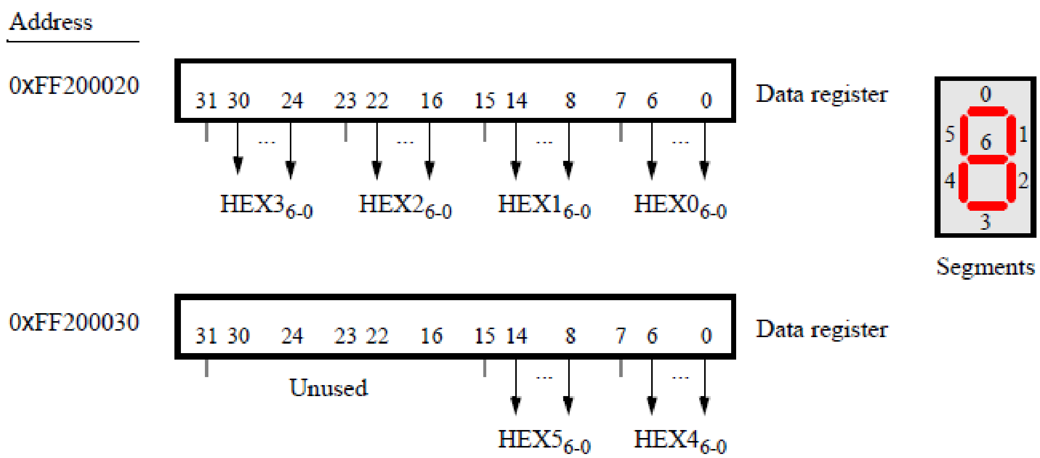
Name	Assembler syntax	Address generation
Offset:		
immediate offset	[Rn, #offset]	Address = Rn + offset
offset in Rm	[Rn, ±Rm, shift]	Address = Rn ± Rm shifted
Pre-indexed:		
immediate offset	[Rn, #offset]!	Address = Rn + offset; Rn ← address
offset in Rm	[Rn, ±Rm, shift]!	Address = Rn ± Rm shifted; Rn ← address
Post-indexed:		
immediate offset	[Rn], #offset	Address = Rn; Rn ← Rn + offset
offset in Rm	[Rn], ±Rm, shift	Address = Rn; Rn ← Rn ± Rm shifted

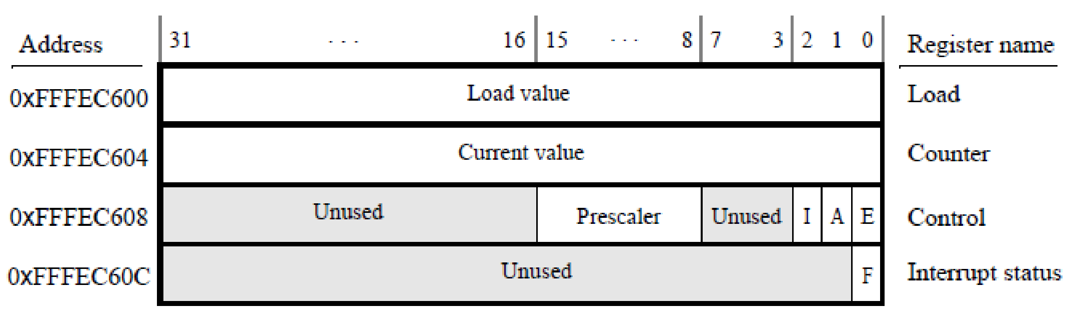
I/O Ports in the DE1-SoC Computer




Address
0xFF200040
31
Unused
10
9
. . .
0
Data register
SW9
SW0


Address
31
30
. . .
4
3
2
1
0
0xFF200050
Unused
KEY3-0
Data register
Unused
Unused
0xFF200058
Unused
Mask bits
Interruptmask register
0xFF20005C
Unused
Edge bits
Edgecapture register


Address
0xFF200020
31 30 24 23 22 16 15 14 8 7 6 0
Data register
HEX36-0
HEX26-0
HEX16-0
HEX06-0
0
5
6
1
4
2
3
Segments
0xFF200030
31 30 24 23 22 16 15 14 8 7 6 0
Data register
Unused
HEX56-0
HEX46-0


Address
31
. . .
16
15
. . .
8
7
3
2
1
0
Register name
0xFFFEC600
Load value
Load
0xFFFEC604
Current value
Counter
0xFFFEC608
Unused
Prescaler
Unused
I
A
E
Control
0xFFFEC60C
Unused
F
Interrupt status

15 … 11	10 … 5	4 … 0
red	green	blue

(a) Pixel values

31 … 18	17 … 10	9 … 1	0
00001000000000	y	x	0

(b) Pixel address

Address	31 … 24	23 … 16	15 … 8	7 … 4	3	2	1	0	
0xFF203020	front buffer address								Buffer register
0xFF203024	back buffer address								Backbuffer register
0xFF203028	Y		X						Resolution register
0xFF20302C	m	n	Unused	B	Unused		A	S	Status register

Note: OnChip memory starts at address is `0xC8000000`, and SDRAM starts at `0xC0000000`.