

Note to ECE 243 Students in 2022: These questions are from the exam in this course, because the course has been re-ordered this year. Also, these questions were done online at home, in a longer time frame with access to CPULATOR. So you should take from these questions our expectations of the 'level' of understanding we expect you to have for the midterm in 2022. You would not have the amount of time given on this exam to do all the questions in a midterm.

University of Toronto

Faculty of Applied Science and Engineering

Unsupervised Final Assessment
ECE243 – Computer Organization
April 27, 2021

Examiners – Stephen Brown and Jonathan Rose

Print and submit this page:

First Name _____ Last Name _____

Student Number _____

1. There are **6** graded questions in this final assessment and one honour pledge. You have **4 hours** to complete this assessment, plus an additional 30 minutes to complete the uploading of your answers to the Crowdmark platform. Your answer to **each question** is to be submitted, separately, to the Crowdmark platform where you received this test. In some cases the *file names* to be submitted are specified in each question in this document.
2. This is an *unsupervised final assessment*. You are allowed to use **only** the following aids: a calculator, spreadsheet, textbooks, your notes and the instructors' notes, your lab work in the course, any materials provided along with the labs, and the *CPULATOR* tool. Writing an unsupervised final assessment necessitates a high level of integrity and honesty. Before beginning to write the assessment please read and sign the following statement:

I pledge upon my honour that I will not violate our Faculty's Code of Behaviour on Academic Matters during this assessment by acting in any way that would constitute cheating, misrepresentation, or unfairness, including but not limited to, using unauthorized aids and assistance, impersonating another person, and committing plagiarism. I acknowledge that providing unauthorized assistance to someone else is also considered a serious academic offence.

_____ (sign your name)

If you have access to a printer: print this page, fill in your name and student number, sign the pledge, and then scan/photograph the page and submit it with the file name **pledge.jpg** or **pledge.pdf** to the Crowdmark Question 7 (at the end of the test).

If you do not have access to a printer: in a text file named **pledge.txt**, type your name and student number, type *the full pledge* into this file, and type your name in place of the signature. Take a picture or screen capture of this file and submit it to Question 7.

[12 marks] 1. Short answer questions; answer briefly (in a few sentences at most) in the space provided for each part of this question.

[2] (a) Why is the polling method of *input* synchronization inefficient?

[2] (b) Is it necessary to have a synchronization method (such as polling) for *output*? Explain your answer and use example(s) from this course's lab to do so.

Question 1 continued . . .

[2]

- (d) Consider the two types of timing mechanisms we have used in the ARM assembly language related to labs in this course:
- i. A software coded delay loop, which does an explicit loop-based computation to “wait.”
 - ii. Using a hardware timer (such as the A9 private timer used in Lab Assignment 5).

Which of these methods provides a more accurate way of implementing a delay? Explain your answer.

~~[2]~~

- ~~(e) What problem does the double buffering of a graphics display solve?~~

- [20 marks] 2. In this question you are **given** a program that is written in the C language, and are **required to write** an equivalent program using the ARM assembly language. You will submit your solution in a single file named **Q2.s**. To create this file you can use *any text editor*—for example, the editor that you normally use to make your lab exercise assembly code. When grading your exam the code from your submitted file **Q2.s** will be loaded into the *CPUlator*, assembled and executed. *To receive full marks you have to submit correctly-working code.* **Important:** your code for each part of the question will be tested independently of the other parts of the question. This means that you can obtain part-marks even if you have not correctly answered, or completed, all parts of the question.

Very Important: no marks will be given if your answer uses the assembly code that is produced by a C compiler, or if you do not **exactly follow the specification for how the ARM registers are to be used** in each part of the question.

Consider the C code shown below. It displays a kind of *analog timer* on the seven-segment displays **HEX3_0**. There are two “hands” in this timer, which are shown side by side. One hand is made up of the two displays **HEX3** and **HEX2**, and the other hand is made up of **HEX1** and **HEX0**.

```
void config_timer(int);
void wait_timer(void);

int hands[] = {0b0000001000100000, 0b0000000001000000,
               0b0000010000010000, 0b0100000000000000};

int main()
{
    unsigned int i = 0, j = 0, right, left;
    volatile int *HEX3_0_ptr = (int *) 0xFF200020;
    int rate = 0;

    config_timer (rate);
    while (1){
        right = hands[i % 4];           // get pattern
        left = hands[j % 4];            // get pattern
        *HEX3_0_ptr = (left << 16) | right; // show "hands"
        wait_timer ( );
        i = i + 1;                       // move right hand
        if ((i % 4) == 0)                // move left hand
            j = j + 1;
    }
}
```

To see how this analog timer operates, please view the video at the URL:

<https://web.microsoftstream.com/video/c3666284-9b7c-49bb-a59a-aefc0610a324>

As shown in the video, each hand moves through four positions in one full revolution. The right hand moves one “tick” each 0.25 second, making a full turn every second. Each time the right hand passes its top position, this causes the left hand to move one “tick”. Thus, the left hand moves once every second and makes a full turn in four seconds.

Question 2 continued ...

[5 marks]

- (a) The main program (on the previous page) calls a subroutine named `config_timer`, which is given below.

```
/* set up the Interval Timer */
void config_timer(int p)
{
    // timer base address
    volatile int * timer_ptr = (int *) 0xFF202000;

    // set the timer period:
    // 1/(100 MHz) x (1562500) x 2^(p+4) = .25 * 2^p sec
    int counter = 1562500 << (p + 4);
    // write to low 16-bit counter start reg
    *(timer_ptr + 2) = counter;
    // write to high 16-bit counter start reg
    *(timer_ptr + 3) = counter >> 16;

    // start timer: STOP = 0, START = 1, CONT = 1, ITO = 0
    *(timer_ptr + 1) = 0b0110;
}
```

The `config_timer` subroutine sets up the *Interval Timer*, which is implemented in the FPGA within the *DE1-SoC Computer*. The operation of this timer is described in Section 2.11 of the document *DE1-SoC Computer ARM.pdf*, which was provided along with Lab 3.

The `config_timer` subroutine receives the integer parameter `p`, which is used to set the timeout period of the timer. The value of `p` can be either positive or negative (in the range $-4 \leq p \leq 4$), and affects the timeout according to the expression given in the code for the counter variable, which is

```
counter = 1562500 << (p + 4);
```

Your task for this part of the question is to write equivalent code in the ARM assembly language for the `config_timer` subroutine. Start your subroutine with the label (all capitals):

CONFIG_TIMER:

You **must** follow the ARM *procedure call standard* (PCS) when writing your subroutine code. Specifically, in PCS a subroutine is allowed to modify only ARM registers R0–R3, but it is not permitted to have changed the contents of any other ARM registers after execution of the subroutine. Your `CONFIG_TIMER` subroutine must receive the parameter `p` in register R0.

Write the code in your file **Q2.s**. Directly above this code there should be a *comment* that appears exactly as follows:

```
// Question 2(a)
```

Question 2 continued ...

[3 marks]

- (b) The main program also calls a subroutine named `wait_timer`, which uses polled-IO to wait for the timeout period. This subroutine is shown below.

```
/* wait for the Interval Timer */
void wait_timer()
{
    // timer base address
    volatile int * timer_ptr = (int *) 0xFF202000;

    while ((*timer_ptr & 0b1) == 0)
        ;
    *timer_ptr = 1;
}
```

For this part of the question you are to write equivalent code in the ARM assembly language for the `wait_timer` subroutine. As mentioned in part (a) of this question, you have to follow the ARM procedure call standard when writing your subroutine code. Start your subroutine with the label (all capitals):

`WAIT_TIMER:`

Write the code in your file **Q2.s**. Directly above this code there should be a *comment* that appears exactly as follows:

```
// Question 2(b)
```

[12 marks]

- (c) For this part of the question you are to write equivalent code for the `main` program in the ARM assembly language.

Your `MAIN` program is required to follow the ARM PCS rules, in the same way as your subroutines from previous parts of the question. In particular your `MAIN` program should assume that the general-purpose registers `R0–R3` may be modified by any subroutine that is called, but general-purpose registers `R4–R12` will be preserved across subroutine calls. Try to write code that is not excessively lengthy; in general, higher marks will be given for programs that are not longer than needed. *Hint:* try to think of an efficient way of computing the modulus operation (`% 4`) that is needed in this program.

Write the code in your file **Q2.s**. Directly above this code there should be a *comment* that appears exactly as follows:

```
// Question 2(c)
```

Make sure that you have included useful comments in your code (for all parts of this question), so that a grader of your question will be able to award part marks in the case that your code does not work correctly.

Submit the file **Q2.s** using the Crowdmark platform.

- [20 marks] 3. Consider the main program shown below. It displays an *analog timer*, in the same manner as in Question 2, but with some additional features: the timer can be stopped/started, reversed in direction, sped up, and slowed down. Also, the rate at which the timer is operating is indicated on the displays **HEX5_4**. Your task is to write equivalent assembly-language code for this C program. You will submit your solution in a single file named **Q3.s**.

```
void config_timer(int);
void wait_timer(void);
int KEY_press(int *, int *);
void display(int);

int hands[] = {0b00000001000100000, 0b00000000001000000,
               0b00000010000010000, 0b01000000000000000};
char seg7[] = {0b00111111, 0b000000110, 0b01011011, 0b01001111,
               0b01100110};

int main()
{
    unsigned int i = 0, j = 0, right, left;
    volatile int *HEX3_0_ptr = (int *) 0xFF200020;
    int rate = 0, dir = 1;

    config_timer (rate);
    while (1){
        right = hands[i % 4];           // get pattern
        left = hands[j % 4];           // get pattern
        *HEX3_0_ptr = (left << 16) | right; // show "hands"
        wait_timer ( );
        i = i + dir;                     // move right hand
        if ((i % 4) == 0)                // move left hand
            j = j + dir;
        if (KEY_press (&rate, &dir) != 0) // check for updates
            config_timer (rate);
        display (rate);
    }
}
```

To see how this analog timer operates, please view the video at the URL:

<https://web.microsoftstream.com/video/33acc55a-64b5-446d-8f74-bd0e14f508d9>

As shown in the video, pressing KEY₀ causes the direction to be reversed (either clockwise or counter-clockwise), and pressing KEY₁ stops, or restarts, the timer. Each time KEY₂ is pressed the rate at which the timer rotates is doubled (until a maximum is reached). Similarly, pressing KEY₃ halves the rotation speed (until a minimum is reached). Finally, the rate of speed (−4 to 4) is indicated on **HEX5_4**.

Question 3 continued ...

[4 marks]

- (a) The main program (on the previous page) calls the subroutines `config_timer` and `wait_timer`, which were discussed in Question 2. The main program for this question also calls a subroutine named `display`, which is given below.

```
void display(int s){
    volatile int *HEX5_4_ptr = (int *) 0xFF200030;
    if (s <= 0) // show rate
        *HEX5_4_ptr = seg7[0 - s];
    else // show -rate
        *HEX5_4_ptr = 0b01000000 << 8 | seg7[s];
}
```

The `display` subroutine indicates on **HEX5.4** the rate at which the timer is rotating, which is in the range from **-4** to **4**. You are to write equivalent code in the ARM assembly language for the `display` subroutine. As described for Question 2, you **must** follow the ARM *procedure call standard* (PCS). Also, (**Very Important**) no marks will be given if your answer uses the assembly language code generated by a C compiler. Your `DISPLAY` subroutine must receive the parameter *rate* in register R0. Start your subroutine with the label (all capitals):

`DISPLAY:`

Write the code in your file **Q3.s**. Directly above this code there should be a *comment* that appears exactly as follows:

```
// Question 3(a)
```

[8 marks]

- (b) The main program for this question also calls a subroutine named `KEY_press`. It has two parameters: a *pointer* to the `rate` variable that indicates how quickly/slowly the timer is rotating, and a *pointer* to the `dir` variable, which indicates if the timer is rotating clockwise (`dir = 1`), or counter-clockwise (`dir = -1`). The `KEY_press` subroutine performs different actions, depending on which pushbutton KEY has been pressed (if any). If `KEY0` or `KEY1` is pressed, then the `dir` variable is modified (via its *pointer*). Similarly, the `rate` variable gets changed (via its *pointer*) if either `KEY2` or `KEY3` has been pressed. The `KEY_press` subroutine returns the value 1 if it modifies the `rate` variable, else it returns 0.

For this part of the question, write equivalent code in the ARM assembly language for the `KEY_press` subroutine. Use register R0 to pass the parameter `&rate`, and use R1 for `&dir`. Return *ret_val* in register R0. As stated previously, you have to follow the ARM PCS, and **no marks** will be given if your answer uses code generated by a C compiler. Start your subroutine with the label (all capitals):

`KEY_PRESS:`

Write the code in your file **Q3.s**. Directly above this code there should be a *comment* that appears exactly as follows:

```
// Question 3(b)
```

The C code for `KEY_press` is given on the next page.

Question 3 continued ...

```
/* returns 1 if rate changes, else returns 0 */
int KEY_press(int *rate, int *dir){
    volatile int *KEY_ptr = (int *) 0xFF200050;
    int press, ret_val;

    ret_val = 0;
    press = *(KEY_ptr + 3);           // read EdgeCapture
    if (press == 0b0001)              // KEY 0?
        *dir = -(*dir);              // reverse direction
    else if (press == 0b0010){        // KEY 1?
        if (*dir != 0)
            *dir = 0;                // stop
        else
            *dir = 1;                // start
    }
    else if (press == 0b0100){        // KEY 2?
        if (*rate > -4)
            *rate = *rate - 1;
        ret_val = 1;
    }
    else if (press == 0b1000){        // KEY 3?
        if (*rate < 4)
            *rate = *rate + 1;
        ret_val = 1;
    }
    *(KEY_ptr + 3) = 0xF;            // clear EdgeCapture
    return (ret_val);
}
```

[8 marks]

(c) For this part of the question you are to write assembly code for the main program.

As stated for Question 2, your MAIN program has to follow the ARM PCS rules. Write the code in your file **Q3.s**. Directly above this code there should be a *comment* that appears exactly as follows:

```
// Question 3(c)
```

Make sure that you have included useful comments in your code (for all parts of this question), so that a grader of your question will be able to award part marks in the case that your code does not work correctly.

Submit the file **Q3.s** using the Crowdmark platform.