ECE3375B: Answers to Practice Final

Prof. John McLeod

2022 04 08

I Microcontroller Concepts [20 marks]

1. What is the distinction between the **polled** and **interrupt-based** approaches to monitoring peripherals? [**3 marks**]

   Answer: A peripheral is polled by repeatedly checking a status register on that peripheral until the bits change reflecting some event. When that happens, code is executed to handle that event. If interrupts are enabled the peripheral can be ignored by the main program. The status register is not checked in software. However, when the status register bits change due to some event, the CPU will immediately execute an interrupt service routine that can handle the event.

2. After a byte is "popped" from the stack into a register, that data still exists in memory at `[sp, #-1]`. Can you use indexed addressing with a negative offset of the stack pointer to use this byte of data later in the program? Explain why this will (or will not) work. [**3 marks**]

   Answer: This won't work very well. This only works if the stack is not used for anything further. If more data is popped from the stack, the stack pointer changes so the negative offset needs to increase dynamically. Worse, if data is subsequently pushed to the stack it may overwrite the original data.

3. We need to use a 16-bit count-up timer with a $10\,\text{MHz}$ clock to time an interval of $4\,\text{ms}$. What value should we write to the appropriate data register on the timer so it will count for this interval? [**2 marks**]

   Answer: We need to count for $4\,\text{ms} \times 10\,\text{MHz} = 40\,000$ cycles. The timer is "count-up", and 16-bits limits to $2^{16} = 65\,536$, so we should write $65\,536 - 40\,000 = 25\,535$ to the timer's data register.

4. A 16-bit count-down timer with a $5\,\text{MHz}$ clock currently has the value `0xE081` in the counter's data register. How long will it take the timer to complete this interval? [**2 marks**]

   Answer: `0xE081` $= 57\,473$. At $5\,\text{MHz}$, this will take $11.4946\,\text{ms}$ to complete.

5. A 6-bit successive approximation register (SAR) analog-to-digital converter (ADC) with a voltage reference of $3.3\,\text{V}$ is converting a $1.25\,\text{V}$ signal. Complete the following table showing the binary search process. [**3 marks**]

Step	Bit Pattern	Digitized Voltage (V)
1	100000	1.65 V
2	010000	0.825 V
3	011000	1.2375 V
4	011100	1.443 75 V
5	011010	1.340 625 V
6	011001	1.289 062 5 V

The final result is `0b011000`.

6. A UART communications frame is defined as `9-1-2`. The following set of pulses is received: `0b0111101000011` (the first pulse received is on the left, the last is on the right). Is this data correctly formatted and uncorrupted? Explain your answer. [**2 marks**]

   Answer: Yes, this frame is correctly formatted and appears to be uncorrupted. Written as start-data-parity-stop the data frame is 0-111 101 000-0-11. This has the correct number of bits, the correct start pulse, and the correct stop pulses. An odd parity check is requested, and the data bits sum to an odd number (5), so the parity pulse is zero. However it is never possible to completely rule out corruption in the data, as if two, four, six, or eight errors in the data occur the parity will still be correct.

7. A peripheral on the DE1-SoC has IRQ #139. What is the appropriate set-enable bit and the appropriate the set-enable register (*ICDISERn*) for this peripheral in the GIC? [**3 marks**]

   Answer: Set bit 11 in *ICDISERn* register at `0xFFFED110`.

8. We wish to enable interrupts on the DE1-SoC for a parallel GPIO port. Assume the GIC is already configured, the exception vector table already written, and global interrupts are already enabled. How do you enable interrupts on the specific hardware? [**2 marks**]

   Answer: Set the bits to 1 in the *Interrupt* register offset by `0x08` from the GPIO base address for each pin on which you want interrupts enabled.

II Interfacing with Peripherals [20 marks]

A model 16-bit microcontroller and several peripherals is described at the end of this exam paper in the **Appendix**. Use this model microcontroller and peripherals to address the following questions.

You may answer the following by writing C or Assembly code. If you cannot write working code, pseudocode and flowcharts that explain the required implementation will be accepted for partial credit. Include as much specific detail as possible to maximize your grade.

You do not need to write an entire program, just the required lines of code. Providing comments or other annotations to your code is recommended. For example, if you are just writing a few lines of C code it would be helpful to add comments to identify which variables should be locally defined and which are global.

If you write C code, please assume that the compiler treats an `int` as a 16-bit number. If you write Assembly code, remember that the registers and word sizes are 16 bits.

1. Work with GPIO port A for the following.

   (a) Provide the code snippet to configure pins 2,3,4,5 to input and 10,11,12,13 to output. Make no assumption on the default state of the pins, and do not modify any other pins. [**2 marks**]

   Answer: There are a lot of ways of doing this using various bitmasks. Here is one method that adds the bit pattern to the control bits, after the bits for all I/O pins given above were wiped to zero. First, in Assembly.

```
@ address of GPIO A from Appendix
ldr r0, =0x6000
@ bit mask for output pins 13..10
ldr r1, =0x3C00
@ bit mask for unused pins 15,14; 9..6; and 0,1
ldr r2, =0xC3C3
@ get state of pins from control register
ldr r3, [r0, #2]
@ wipe out everything except unused pins
and r3, r2
@ add mask
add r3, r1
@ write to control register
str r3, [r0, #2]
```

   In C.

```
// address of GPIO A from Appendix
volatile int* const gpio_a = (int*)0x6000;
// clear everything except unused pins
*(gpio_a + 1) &= 0xC3C3;
// set input/output pins
*(gpio_a + 1) += 0x3C00
```

   (b) Provide the code snippet to read from pins 2,3,4,5 and store as a 4-bit number. If you write C code, store the data in `int input_data`. If you write Assembly code, store the data in `r0`. [**2 marks**]

   Answer: In Assembly.

```
@ address of GPIO A from Appendix
ldr r1, =0x6000
@ get data
```

```
ldr r0, [r1]
@ slide it over because we don't care about pins 0,1
lsr r0, #2
@ wipe out everything except bottom 4 bits
and r0, #0xF
```

In C.

```
// address of GPIO A from Appendix
volatile int* const gpio_a = (int*)0x6000;
// slide data over to bottom 4 bits and wipe everything else
input_data = (*(gpio_a) >> 2) & 0x0F;
```

(c) Provide the code snippet to write the lowest 4 bits of some given data to pins 10,11,12,13. If you write C code, the data is already provided in `int output_data`. If you write Assembly code, the data is already provided in `r0`. [**2 marks**]

Answer: Here I am careful to preserve any existing output on the port, as we don't know if any of pins 0,1 or 6,7,8,9 are used as output. First, in Assembly.

```
@ address of GPIO A from Appendix
ldr r1, =0x6000
@ mask data, just in case
@ (question is a bit ambiguous here)
and r0, #0xF
@ shift data over
lsl r0, #10
@ get original port data
ldr r2, [r1]
@ wipe out original data on pins 13..10
mvn r3, #0x3C00
and r2, r3
@ add new data
add r2, r0
@ write to port
ldr r2, [r1]
```

In C.

```
// address of GPIO A from Appendix
volatile int* const gpio_a = (int*)0x6000;
// first erase any old output from pins 13..10
*(gpio_a) &= ~(0x3C00);
// mask, shift, and output data
*(gpio_a) |= (output_data & 0x0F) << 10;
```

2. Write the code snippet to use interval timer A to idle for a 4 s interval. [**2 marks**]

Answer: The timer is at 1 MHz, so a 4 s interval is 4×10^6 cycles. That is bigger than a 16-bit number! This 16-bit timer can at most idle for 65.536 ms. So I'll have it idle for 40 ms, and do that 100× in software. Note also the timer as given cannot count down and repeat. First, in Assembly:

```
@ address of timer A from Appendix
ldr r0, =0x6400
@ 40 ms interval at 1 MHz
ldr r1, =40000
```

```
@ repeat 40 ms 100 times for 4 s.
mov r6, #100

_count_loop:
 @ start timer by writing count period to it
 str r1, [r0]
_timer_loop:
 @ check status
 ldr r2, [r0, #2]
 @ loop until timer done
 cmp r2, #1
 bne _timer_loop
 @ decrement counter
 subs r6, #1
 @ loop until done
 beq _count_loop
```

In C, with everything hard-coded and no comments, just for a change.

```
for (i=0;i<100;i++)
{
  *((int*)0x6400) = 40000;
  while(*((int*)0x6402) == 0)
        ;
}
```

3. Write the code snippet to use the input capture to record when pin 7 goes high. You may assume this will happen within the maximum interval on the timer. [**2 marks**]

Answer: Question did not state which register or variable the recorded time should be stored in, so I can use anything. In Assembly:

```
@ address of input capture from appendix
ldr r0, =0x6C20
@ control bits to select pin 7, check for value 1, and start is
@ 0b0000 0111 0100 0001 = 0x0741
@ see appendix for details
ldr r1, =0x0741

@ start input capture
str r1, [r0, #2]
_loop:
  @ check status
  ldr r1, [r0, #4]
  @ wait until input is captured
  cmp r1, #1
  bne _loop
@ get the time into r1
ldr r1, [r0]
```

In C. I'm being sloppy about where variables are defined — a compiler might complain but this is an exam and nobody else will care.

```
// define structure, why not
typedef struct _ic_unit
{
```

```
  int data;
  int control;
  int status;
} ic_unit;

volatile ic_unit* const ic_ptr = (ic_unit*)0x6C20;

// start input capture
ic_ptr->control = 0x0741;
// wait until done
while(ic_ptr->status == 0)
  ;
// get time
int capture_time = ic_ptr->data;
```

4. Work with the ADC for the following.

   (a) Write the code snippet to use this ADC to start sampling channel 0. Make sure you do not modify the interrupt enable bit. [**2 marks**]

   Answer in Assembly:

```
@ address of ADC, from appendix
ldr r0, =0x7000
@ get current ADC control so we don't accidentally modify interrupt bit
ldr r1, [r0]
@ set bits 1,0 to 00 for channel 0
mvn r2, #2
and r1, r2
@ write to ADC to start conversion
str r1, [r0]
```

   In C.

```
// one line of magic
*((int*)0x7000) = (*((int*)0x7000) & ~(0x3));
```

   (b) Write the code snippet to idle until conversion is complete. If you write C code, store the result in `int adc_input`. If you write Assembly code, store the result in `r0`. [**2 marks**]

   Answer: In a question like this, it is fair to assume that the code immediately follows from (a), so it is fine to continue using `r0` as the address of the ADC. But here I will explicitly define it again. In Assembly:

```
@ address of ADC, from appendix
ldr r0, =0x7000

_loop:
  @ get status
  ldr r1, [r0, #2]
  @ check if done, note status = 0 when done
  @ carefully read ADC description in appendix!
  cmp r1, #1
  beq _loop

@ get data
```

```
ldr r1, [r0, #4]
@move to r0 as requested
mov r0, r1
```

In C.

```
while(*((int*)0x7002) == 1)
  ;
int adc_input = *((int*)0x7004);
```

5. Work with the UART communications peripheral for the following.

   (a) Write the code snippet to configure the UART for 7-1-1 at 4800 baud. [**2 marks**]

   Answer: In Assembly.

```
@ address of UART, from appendix
@ put in r8 for a change
ldr r8, =0x7400
@ control bits for 7-1-1 ad 4800 are
@ 0000 0000 0011 0001 = 0x31
mov r0, #0x31
@ write to UART control
str r0, [r8, #2]
```

   In C.

```
// precompiler definitions, feelin' fancy today
#define UART_BASE            0x7400
#define UART_CONTROL_BITS    0x0031

// use a structure, because I am a gentleman-coder
typdef struct _mighty_fine_UART
{
  int status;
  int control;
  int rx_data;
  int tx_data;
} UART_type;

volatile UART_type* const uart = (UART_type*)UART_BASE;

// lots of preamble for 1 line of code, no?
uart->control = UART_CONTROL_BITS;
```

   (b) Write the code snippet to send one frame of data on the UART. If you write C code, assume the data to send is in `int uart_data`. If you write Assembly code, assume the data to send is in `r0`. The system should hang until the send operation is complete. [**2 marks**]

   Answer: In Assembly.

```
@ UART address still in r8 from above

@ send data
str r0, [r8, #6]
```

```
@ loop until transmitted
_loop:
  @ get status
  ldr r1, [r8]
  @ check tx bit
  tst r1, #1
  beq _loop
@ ok done
```

In C.

```
// reuse definitions and structure from above
while(uart->status & 1 == 1)
 ;
// that's it
```

6. Write the code snippet to enable interrupts from pin 11 of GPIO port A. Make no assumptions about the state of the system or that peripheral, and only change the bits required to enable interrupts from that peripheral. [**2 marks**]

Answer: In Assembly.

```
@ address of interrupt register in GPIO A
ldr r0, =0x6004
@ get existing state
ldr r1, [r0]
@ mask for pin 11
mov r2, #1
lsl r2, #11
@ set pin 11
orr r1, r2
@ write back to GPIO A
str r1, [r0]
```

In C.

```
// quick and dirty
// I am no longer a gentleman-coder
*((int*)0x6004) |= (1<<11);
```

III Embedded System Design [10 marks]

The following design problem should be implemented with the model 16-bit microcontroller described in the **Appendix**.

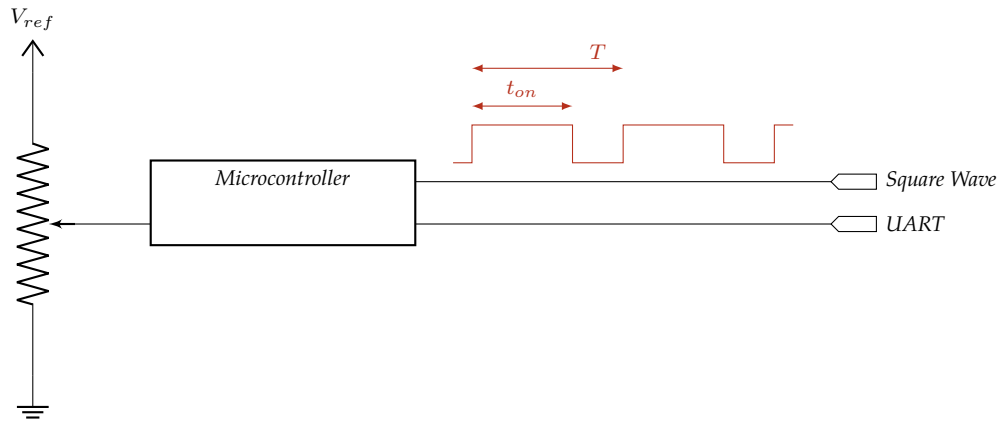
- You may use any of the peripherals listed in the **Appendix**.
- Your design **must use at least one interrupt-enabled peripheral**.
- Please implement the design problem as stated. **Make no assumption** regarding the initial state of the microprocessor and peripherals — it is necessary to appropriately configure all interrupts and peripheral control registers.
- If certain aspects are vague, or unspecified, make a design choice or an assumption and proceed.
- As always, make sure you clearly state any design choices and assumptions you make.

You may answer this question in two steps.

1. **Optional:** Describe your implementation in human-readable form using any combination of paragraphs, bullet points, comments, diagrams, flow charts, etc. you wish. You can describe the basic program flow, task scheduling, use of local and global variables, use of interrupts, and any design choices and assumptions.
2. **Mandatory:** Write the code in C or Assembly.

If you do not describe your implementation, we will attempt to infer your design from the code. This may result in you losing marks if we can't make heads or tails out of what you wrote: if your code is very vague, hard to follow, lacks descriptive variable or function names, and/or has no comments, such that we can't figure out how it is supposed to work, we will not give you the benefit of the doubt unless you also provide a sensible description of the implementation. You may also combine (1) and (2) by writing code and adding arrows linking that code to descriptive annotations, diagrams, flowcharts, etc.

Problem: A potentiometer is used to tune a voltage between $0\,\mathrm{V}$ and $V_{ref} = 5\,\mathrm{V}$. Program the microcontroller to use this voltage to set the duty cycle of a $1\,\mathrm{kHz}$ square wave. The duty cycle should also be logged (as a suitable binary number) on an external device by transmitting it over UART at $9600\,\mathrm{baud}$ with a `8-0-1` frame.



In case you forget your electronics lessons, the duty cycle η of a square wave is the ratio of the length of time a pulse is held high (t_{on}) and the period of a cycle (T), and ranges from 0 to 1.

$$\eta = \frac{t_{on}}{T}$$

The analog input signal and the square wave output signal signal need to be connected to the microcontroller somehow. You need to decide how to do this and configure everything appropriately.

Answer: This is a reasonably complicated problem. It is also worth half as much as the other two sections, so you should keep that in mind when answering it: if you are pressed for time, a quick pseudocode outline might be worth 3/10 and you'd get a better overall mark by focusing on maximizing your grade on part I or II.

Anyway, I recommend thinking about these three problems in three steps, so that even if you get stuck you can at least present a partial solution.

1. Decide which peripherals are needed.
2. Decide what functionality is needed from each peripheral. Make a list of the subroutines that do these basic tasks (or in C, write the function declarations), such as "set timer for n interval" or "read from ADC".
3. Determine the flow of the main program, and how the peripheral functionality is used.

I'd recommend actually *implementing* the peripheral functionality in step 2 last, and only if you have time. If you don't have time to implement everything, it is much better to have a well thought-out main program and just use pseudocode like "set timer for 100 ms and wait until done" than to have perfectly-written generic subroutines for the peripherals but no idea how they go together to solve the main problem.

So first, I'll think about what kind of peripherals are needed.

- The potentiometer must connect to an ADC. There is no other way to read in an analog voltage.
- The square wave output must be from a GPIO pin.
- The UART output has to be using the UART, obviously.
- The square wave duty cycle can be controlled by an interval timer, or we can use an output compare unit.

I'll use an interval timer and a GPIO pin to generate the output square wave.

Now what kind of code/subroutines do I need to write?

- I need to write code to configure the UART for 9600 baud and a `8-0-1`.
- I can write code or a subroutine to transmit a byte on the UART.
- I need to use a timer to control the output to achieve a square wave with a variable duty cycle, so a generic timer subroutine that sets the timer for an arbitrary time is useful.
- I was instructed to use interrupts on at least one peripheral, so I'll use it on the timer. This means the timer subroutine is an ISR.
- I also need to sample from the ADC. I'm feeling ambitious, so that will be interrupt-enabled too.

Finally, how does this program work?

- The ADC controls the duty cycle of the 1 kHz output. The ADC is 8-bit, so it will return a value between `0x00` and `0xFF`.
- The timer is 16-bit and 1 MHz, so a 1 kHz wave requires a count interval of 1000.
- The output frequency is not evenly divisible by the maximum ADC output, but it is close. I'll make sure to restrict the ADC value to a maximum of `0xFA` (or decimal 250), so I can multiply the ADC value by 4 to get a number between 0 and 1000.

Ok, so now the timer just needs to control the output.

- But! I need to change the duty cycle, so the timer should count for the high periods and low periods separately, and the GPIO output should be toggled at the end of each.
- Since I want the timer to be interrupt-driven, I can't just set the timer for 1 ms and poll for when it is finished the high duty cycle (or low duty cycle).

- Instead, I'll make two global variables: one for the high period t_{on}, and one to record whether the output is currently high or low (actually this last global variable isn't necessary — we could obtain this information by reading from the GPIO).

This whole page of text outlines my thought process. I don't expect someone to do this much writing for their solution.

Ok, so the program is structure as follows:

1. Set up interrupt vector table (in Assembly only), or define global variables and function declarations (in C only).
2. At the start of the main program, configure interrupts for a timer and the ADC.
3. Configure the UART as described.
4. Assume a 50-50 duty cycle and a rising edge at the start, and start the timer and GPIO output accordingly.
5. Loop and do nothing.
6. When the timer interrupt occurs, check if we finished high or low part of duty cycle.
   (a) If we finished the high part, set the output to low. Set the timer for $1\,\text{ms} - t_{on}$. Also, start sampling the ADC.
   (b) If we finished the low part, set the output to high. Write the duty cycle to UART. Set the timer for t_{on}.
7. When an ADC interrupt occurs, convert the ADC value to t_{on}.

These steps provide a pretty clear description of the program and is worth at least 5/10. Now I need to read the Appendix to understand how to write the code. Complete code implementing the program described above is provided in Assembly and C as separate documents.

Please note that the complete code provided is longer and has many more comments than I would expect a from a student to earn 10/10. Please also note that there are many other ways to solve this problem. Three other methods:

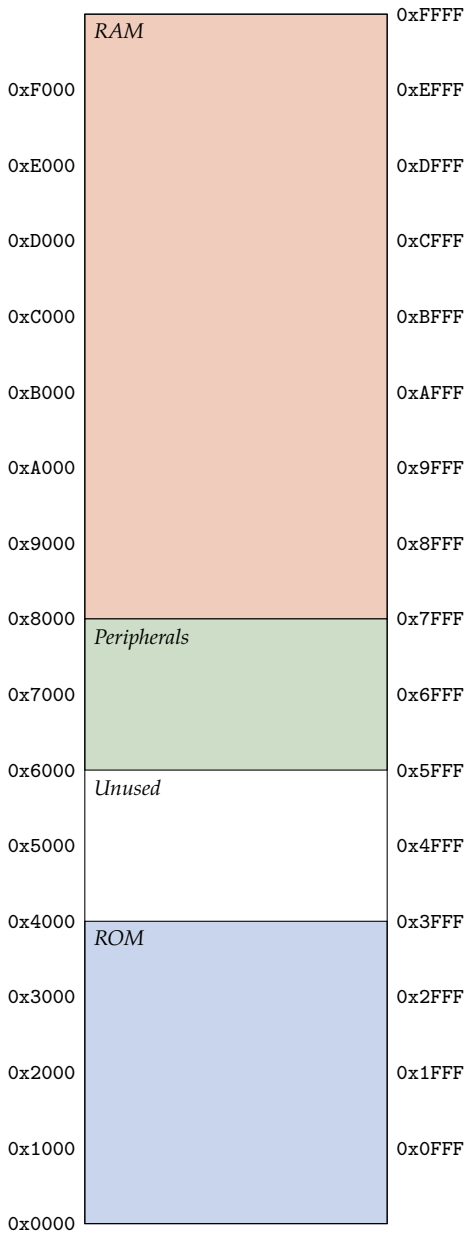
- "I don't like interrupts." The question specified that an interrupt must be used, but it didn't specify how it should be used. Enable interrupts on the ADC only, and have the ISR always restart conversion. Run the timer for 1 ms and poll the count, checking against the converted ADC value to see when the output should flip from 1 to 0. The ADC will update 1000s of times during a single clock cycle, but does that matter? Probably not.
- "I can read a manual, even for new peripherals." Use the real-time interrupt timer to count-down-and-repeat a 1 ms interval. At each timer interrupt, set GPIO output to 1, write to UART, start the output compare to set output to 0 after the elapsed high period, and also start ADC conversion. ADC is also interrupt enabled, when it interrupts store the value as the high period. The output compare does not need to be interrupt enabled, and it will automatically handle the low-period part of the output.
- "I don't understand interrupts." Forget about interrupts and write a program that just polls the timer and triggers ADC, output, and UART accordingly. Sure you won't get 10/10 but a good solution might still be 7 or 8/10.

Appendix

This exam uses a model 16-bit microcontroller with simplified structure and peripherals. This microcontroller **memory-mapped**, uses **von Neumann architecture**, and is **little-endian**. It has a 16-bit address bus and a 8-bit data bus.

When writing C code for this microcontroller, please assume that the compiler treats an `int` as a 16-bit number. If you wish to write Assembly code, you may assume that this microcontroller accepts code with similar mnemonics and format as ARMv7 Assembly, and has access to the same general purpose registers (except the registers, and the word size, is only 16-bits). You may also assume that all memory cells are also both **byte-addressable** and **word-addressable**, including the registers on peripherals.

The memory map for this microcontroller is shown below.



Interrupts

This model microcontroller has a simple configuration for hardware interrupts. This microcontroller has only one operating mode, and no banked registers.

- Interrupts are enabled in the CPU by setting bits in the special-purpose `irqen` register, each of the interrupt-enabled hardware peripherals maps to a single bit in this register. This can be done in C with the code:

```
asm("msr␣irqen,␣%[ps]" : : [ps]"r"(interrupt_mask));
```

  where `int interrupt_mask` is the appropriate 16-bit sequence of interrupts to enable or disable. This can also be done in Assembly with the code:

```
msr irqen, rX
```

  where `rX` is a general purpose register (`r0`, `r1`, `r2`, or whatever) that contains the appropriate 16-bit sequence of interrupts to enable or disable. For example, if you wanted to enable interrupts on IRQ 3 and IRQ 10, the appropriate bit sequence would be zero except bits 3 and 10 set to 1: `0b0000 0100 0000 1000` = `0x0408`.

- Interrupts are enabled in hardware by setting the appropriate bit(s) in a control register, as described below for each peripheral.

- The interrupt vector table (IVT) starts at `0x0000` has a row (a word) for each interrupt-enabled hardware peripheral, this row consists of the address for the interrupt service routine (ISR). In C, the ISR must be written with the definition:

```
void _isr_[iqr]( void )
```

  where `[irq]` should be replaced with the appropriate IRQ number. In C you only need to write ISRs for interrupts that are enabled, the compiler will figure out the rest. In Assembly, you must define the entire IVT at the start of the program:

```
1  .section .vectors, "a"
2    .word unused           @ IRQ 0
3    .word unused           @ IRQ 1
4    .word isr_irq_2        @ IRQ 2
5    .word unused           @ IRQ 3
6    .word unused           @ IRQ 4
7    .word mystrangelabel   @ IRQ 5
8    .word unused           @ IRQ 6
9    .word unused           @ IRQ 7
10   .word unused           @ IRQ 8
11   .word unused           @ IRQ 9
12   .word unused           @ IRQ 10
13   .word lollipop_isr     @ IRQ 11
14   .word unused           @ IRQ 12
15   .word unused           @ IRQ 13
16   .word unused           @ IRQ 14
17   .word unused           @ IRQ 15
```

  Then you can follow with the main program:

```
20 .text
21 .global _start
22 _start:
23   @ your main code
```

And later define the ISRs:

```
@ ISR code for IRQ 2
isr_irq_2:
  @ preserve context
  push {r0-r12, lr, sp}
  @@@@@
  @ main code for ISR
  @ does something useful
  @@@@@
  @ restore context
  pop {r0-r12, lr, sp}
  @ return
  subs pc, lr, #2

@ ISR code for IRQ 5
mystrangelabel:
  @ more code
  subs pc, lr, #2

@ ISR code for IRQ 11
lollipop_isr:
  @ more code
  subs pc, lr, #2

@ ISR for unused IRQs
@ this should never happen, so just hang
unused:
  b unused
```

As you can see, you are free to name the ISRs any silly label you like in Assembly, as long as that label is linked to the correct row of the IVT.

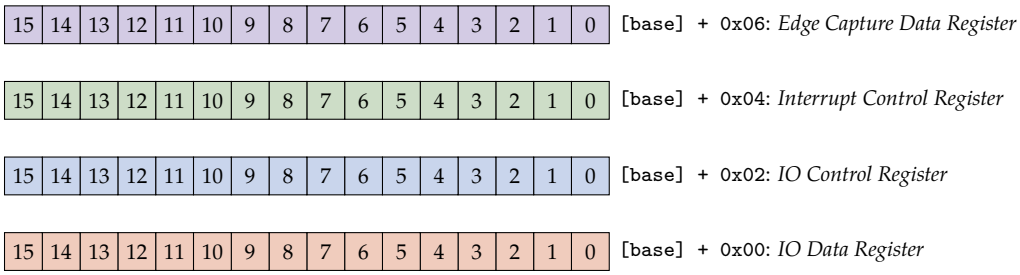
- For the purposes of this exam, if you are writing code in Assembly, it is sufficient to only explicitly provide the rows of the IVT that are used for interrupts — you may assume all other rows for IRQs that are not used already have the address of a suitable subroutine (like `unused`).

Once `irqen` is correctly configured, the IVT is set up with the appropriate ISRs, and the peripheral control bits are appropriately set for interrupts, the system is ready to go. Note that the only way to disable all interrupts globally is to set all bits in `irqen` to zero.

General Purpose Input/Output Ports

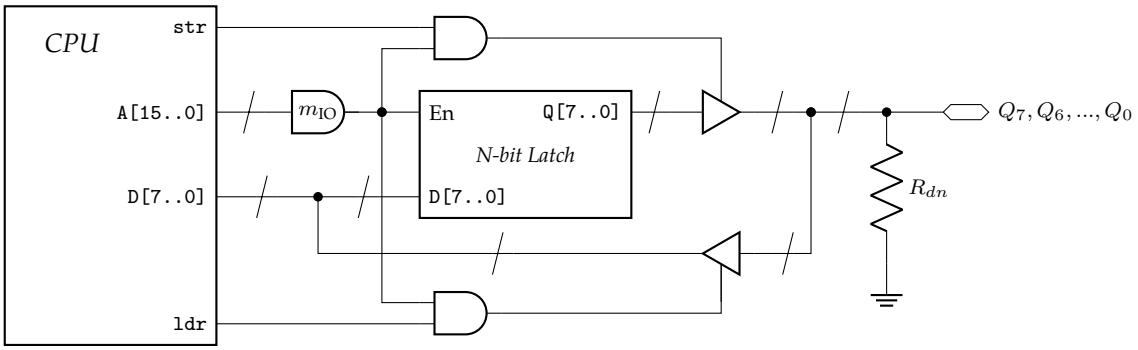
This model 16-bit microcontroller has six 16-pin general purpose input/output (GPIO) ports. These have the following structure.

- Data may be written to, or read from, the IO data register. Each bit in this register is independently configured for input or output, and maps to a corresponding pin (pin 0 to bit 0, pin 1 to bit 1, up to pin 15 to bit 15).
- Setting a bit to 0 or 1 in the IO control register assigns the corresponding bit in the IO data register to input or output, respectively.
- Setting a bit to 1 in the interrupt control register allows hardware interrupts to be generated when data on the corresponding pin changes. This is usually for pins set to input, but it will still work if pins are set to output and new data is written to that pin.
- A bit is set in the edge capture data register if the value on the corresponding pin changes from 0 to 1 since this register was last read from. Writing any value to this register clears all bits to zero.



The GPIO ports' addresses and IRQs are as follows.

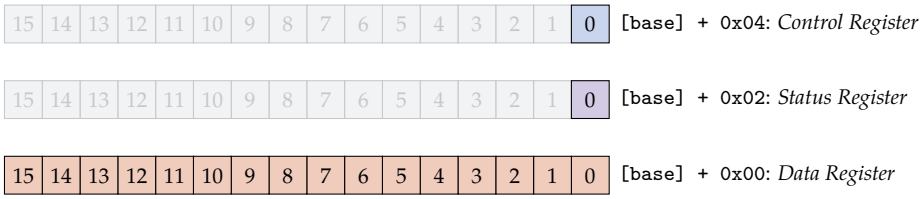
	GPIO A	GPIO B	GPIO C	GPIO D	GPIO E	GPIO F
Address	0x6000	0x6020	0x6040	0x6060	0x6080	0x60A0
IRQ	#2	#3	#4	#5	#6	#7



Interval Timers

This model 16-bit microcontroller has two interval timers. These have the following structure:

- The timer always counts down with an internal clock speed of 1 MHz.
- To set the countdown interval, write the appropriate value to the data register. This also starts the timer.
- When the timer is counting down, the current counting time can also be read from the data register.
- Writing to the data register while counting down will overwrite the current count.
- Bit 0 in the status register is set to 1 when the timer reaches zero.
- Writing any value to the status register clears bit 0.
- Bit 0 in the control register is set to enable interrupts.



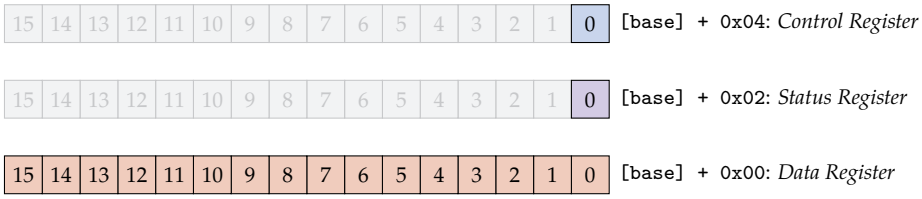
The timers' addresses and IRQs are as follows.

	Timer A	Timer B
Address	0x6400	0x6420
IRQ	#8	#9

Real-Time Interrupt Timer

This model 16-bit microcontroller has a special timer solely for periodically generating interrupts. This timer has the following structure:

- The timer always counts down with an internal clock speed of 1 MHz.
- To set the countdown interval, write the appropriate value to the data register.
- The current count on the timer is hidden and cannot be accessed in software.
- Bit 0 in the status register is set to 1 when the timer reaches zero.
- Writing any value to the status register clears bit 0.
- Bit 0 in the control register is set to enable interrupts, this also starts the timer on countdown-and-repeat mode. Clear bit 0 to stop the timer (and also disable interrupts).

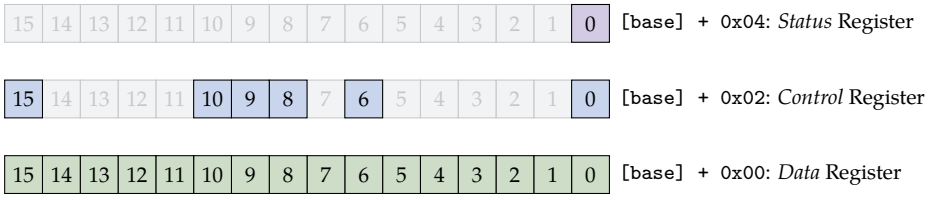


The real-time interrupt timer is at address `0x6800` with IRQ #10.

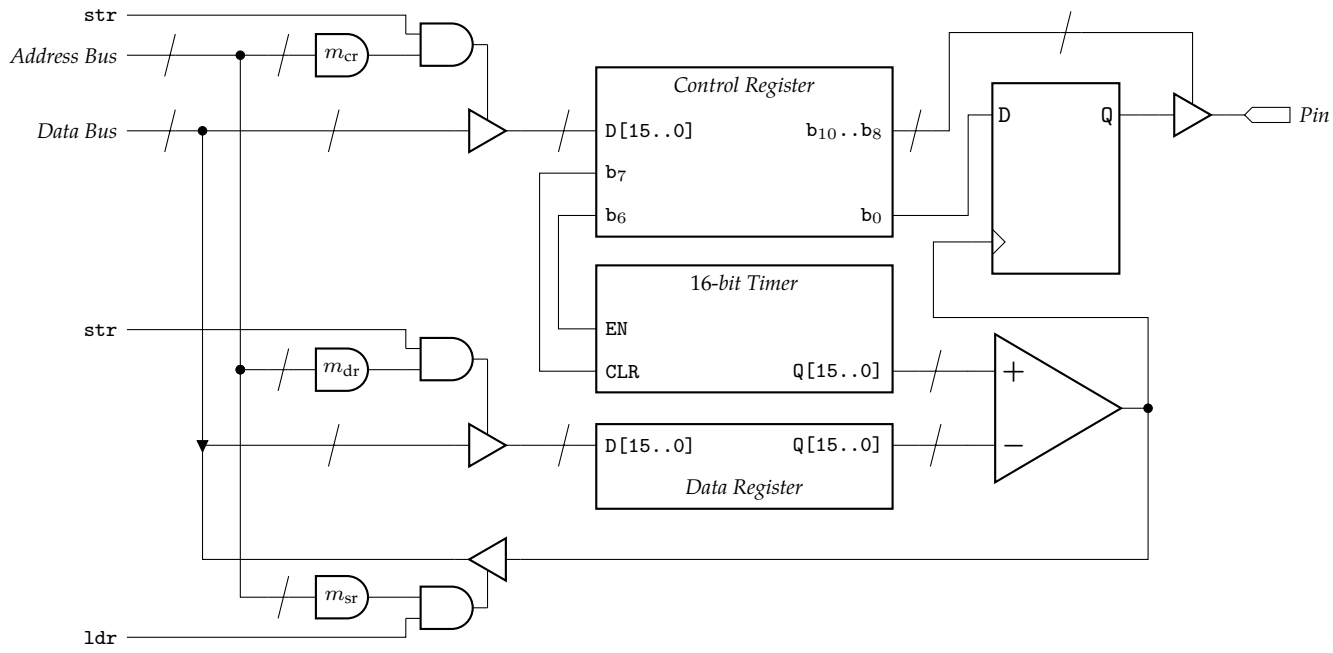
Output Compare

This model 16-bit microcontroller has one output compare unit. This has the following structure:

- The 16-bit timer counts up at 1 MHz.
- Bits 8, 9, and 10 in the control register are interpreted as as a 3-bit binary number to select one of 8 output pins.
- The output compare unit is linked to pins 0 to 7 of GPIO port B. The appropriate pin on GPIO port B must also be set as output, otherwise the output compare unit will not operate properly.
- Bit 15 of the control register is set to 1 to enable interrupts. Bit 6 is set to 1 to start the timer, it can be cleared to stop the timer. Bit 0 is the value to be checked against the input pin.
- The time interval is loaded into the data register.
- The time interval is compared to the current count, as soon as the count exceeds that interval, the value in control register bit 0 is written to the output pin and the status register bit 0 is set to one.
- Writing any value to the status register will reset (clear) the internal timer.



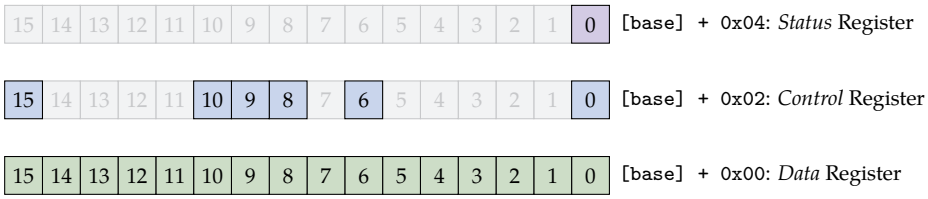
The output compare is at address `0x6C00`, with IRQ #11. If interrupts on the output compare are enabled, interrupts on the corresponding pin of GPIO port B should be disabled.



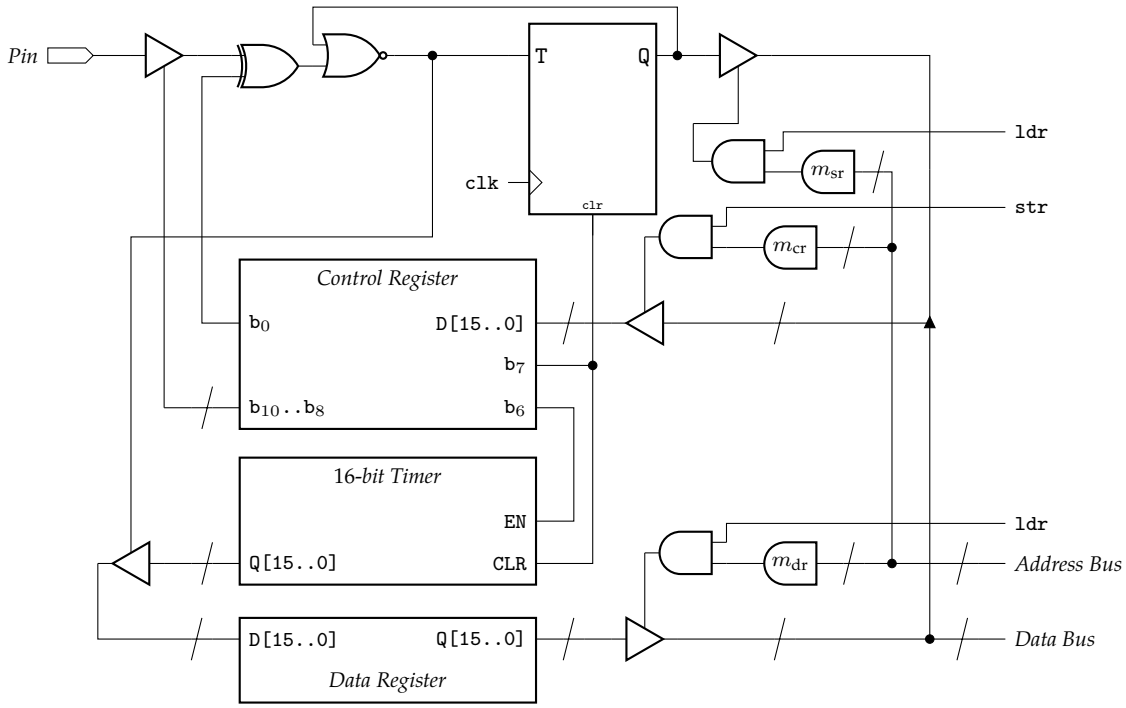
Input Capture

This model 16-bit microcontroller has one input capture unit. This has the following structure:

- The 16-bit timer counts up at 1 MHz.
- Bits 8, 9, and 10 in the control register are interpreted as as a 3-bit binary number to select one of 8 input pins.
- The input capture unit is linked to pins 0 to 7 of GPIO port B. The appropriate pin on GPIO port B must also be set as input, otherwise the output compare unit will not operate properly.
- Bit 15 of the control register is set to 1 to enable interrupts. Bit 6 is set to 1 to start the timer, it can be cleared to stop the timer. Bit 0 is the value to be checked against the input pin.
- The first time the input pin matches the specified value, the current count on the timer is stored in the data register and bit 0 in the status register is set to 1.
- Writing any value to the status register will reset (clear) the internal timer.



The input capture is at address `0x6C20`, with IRQ #12. If interrupts on the input capture are enabled, interrupts on the corresponding pin of GPIO port B should be disabled.

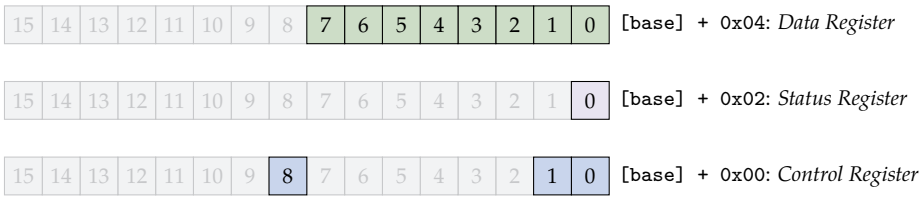


Analog-to-Digital Conversion

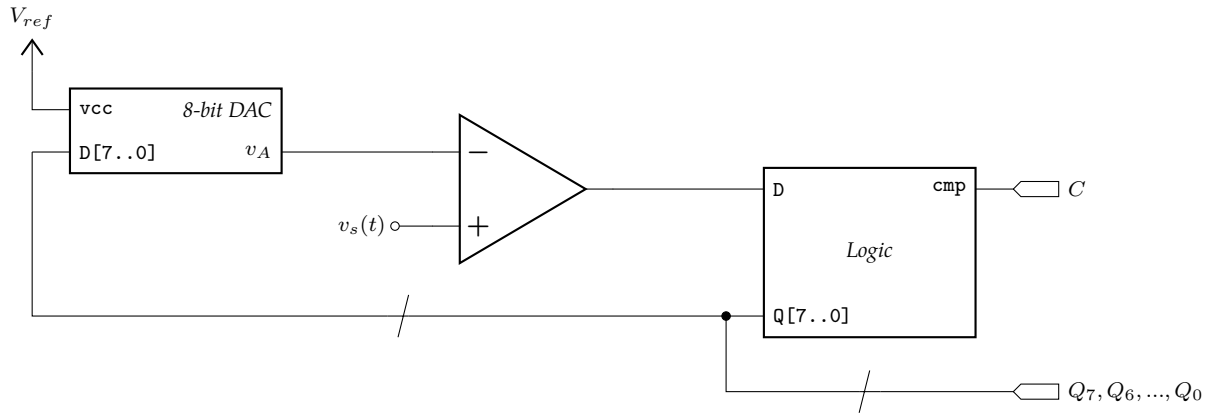
The model 16-bit microcontroller has one analog-to-digital converter (ADC). This has the following structure:

- The ADC does 8-bit conversions and has 4 channels (numbered 0 to 3). It uses the successive approximation register (SAR) method.
- Writing the channel number to bits 0 and 1 of the control register tells the ADC to start sampling the corresponding channel. Bit 8 of the control register is set to 1 to enable hardware interrupts.
- Bit 0 of the status register is normally 1, when it is cleared to 0 it means the conversion has finished. This bit is automatically reset to 1 when the ADC starts another conversion.
- The data register holds the 8-bit converted value from then channel selected by the control register.

Note that this ADC structure only allows one of the four channels to be sampled at a time, as there is only one data register to hold the result.



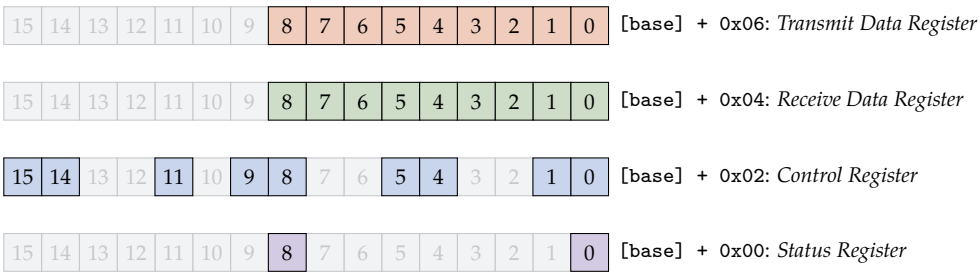
The ADC is at address `0x7000`, with IRQ #13. For this microcontroller, $V_{ref} = 5\,\mathrm{V}$.



Asynchronous Serial Communications

The model 16-bit microcontroller has one universal asynchronous receiver-transmitter (UART) peripheral. This has the following structure:

- The UART is full duplex with separate registers for receiving and transmitting data.
- Bit 0 of the status register is set when data is successfully transmitted. Bit 8 of the status register is set when data is successfully received.
- The UART hardware will automatically check the parity (if a parity check is enabled). Bit 8 will only be set when data is received and the parity check is valid.
- The UART communication frame is set by the control register, as described below.
- Writing data to the the transmit data register tells the UART to transmit that data. This also clears bit 0 in the status register. Only the appropriate number of data bits (7, 8, or 9, as determined by the frame) should be written to this register. All extra bits will be silently ignored. The UART will automatically add appropriate start, parity, and stop pulses to the frame when transmitting.
- Data received by the UART is stored in the receive data register. Only the data bits are stored here, the start, stop, and parity pulses are not. Reading from this register clears bit 8 in the status register, subsequently received data will overwrite this register. If data is received but not read (i.e. while bit 8 in the status register remains high), additional received data will be silently ignored.



The UART can be configured for 7, 8, or 9 bits of data; even, odd, or no parity; the choice of baud rates; and interrupts as follows.

Bits	Purpose	Value	Definition	Bits	Purpose	Value	Definition
1,0	Data frame	00	*Undefined*	9,8	Baud Rate	00	4800 baud
		01	*7 bits*			01	9600 baud
		10	*8 bits*			10	19 200 baud
		11	*9 bits*			11	28 800 baud
5,4	Parity check	00	*No check*	11	Stop Pulses	0	*1 stop pulse*
		01	*No check*			1	*2 stop pulses*
		10	*Even parity check*	14	Interrupt	0	*Recieve interrupt disabled*
		11	*Odd parity check*			1	*Recieve interrupt enabled*
				15	Interrupt	0	*Transmit interrupt disabled*
						1	*Transmit interrupt enabled*

The UART is at address `0x7400`, with IRQ #14.