ECE3375B: Final Examination

Profs. Ken McIsaac & John McLeod

Exam Starts: 14:00, 2021 04 27

Instructions:

- The final examination is **take-home** and **open-book**. You may use any printed or electronic resources to complete this exam.
- You should **not** collaborate with other students in this class or ask for help from other people (including online tutorial websites). **Doing so constitutes an academic offence!**
- In the event that two or more students submit suspiciously similar answers, the course instructors **reserve the right to request personal interviews** with students to check their understanding of their submissions.
- You should **not** distribute you exam paper after the exam, or upload questions to an online tutorial site.
- You should submit your **final answers** to the "Final Examination" *Assignments* module on OWL.
- You must be prepared to submit a digital copy of you answers and any rough work, diagrams, comments, etc. you would like us to see for when marking your exam. Your submission should include all answers as a **single PDF**. You do not have to include this exam question paper in your submission.
- Your uploaded answers must be **within the upload size limit of** 20 MB. This is large enough for most text/-graphics, but if you photograph written work with a high-resolution camera, you may need to reduce the size of the images.
- It is each student's responsibility to ensure they can submit **legible, digitized answers** to the "Final Examination" *Assignments* module within the time limit provided.
- Late submission is accepted, with a penalty of **50% per hour** after the deadline.
    - You have **four (4) hours** to complete the examination. This includes the time required to digitize and upload your answers. Please manage your time wisely.
    - If you experience serious technical difficulties, you need to let Prof. McLeod (jmcleod7@uwo.ca) know as soon as possible, but there is no guarantee accommodations can, or will, be made.
- The entire exam is worth **50 marks**.
- University policy states that cheating, including plagiarism, on an examination is a scholastic offence. The commission of a scholastic offence is attended by academic penalties which might include expulsion from the program. If you are caught cheating, there will be no second warning.

Submit your completed exam paper by 18:00 on Tuesday, April 27th 2021

I Microcontroller Concepts [20 marks]

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

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**]

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**]

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**]

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		
2		
3		
4		
5		
6		

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**]

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**]

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**]

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

III Embedded System Design [10 marks]

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

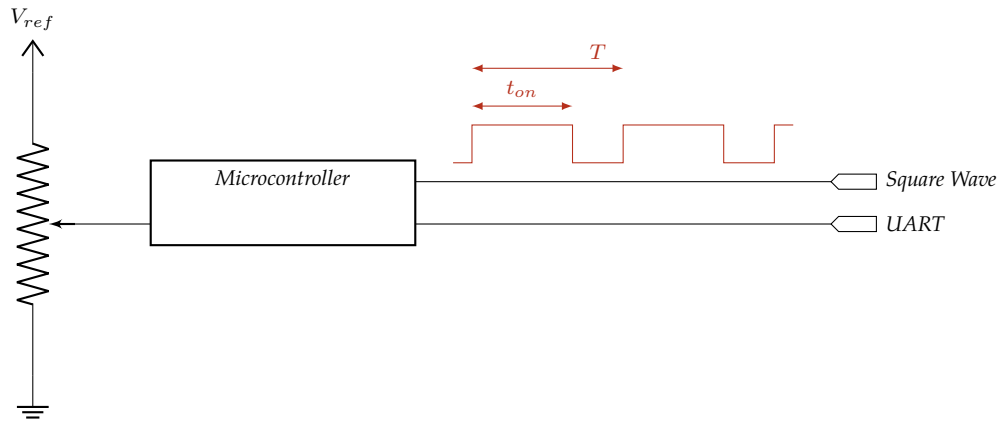
- You may use any of the peripherals listed in the **Appendix**.
- Your design **must use at least one interrupt-enabled peripheral**.
- Please implement the design problem as stated. **Make no assumption** regarding the initial state of the microprocessor and peripherals — it is necessary to appropriately 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. **[5 or 0 marks]**
2. **Mandatory:** Write the code in C or Assembly. **[5 or 10 marks]**.

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.

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

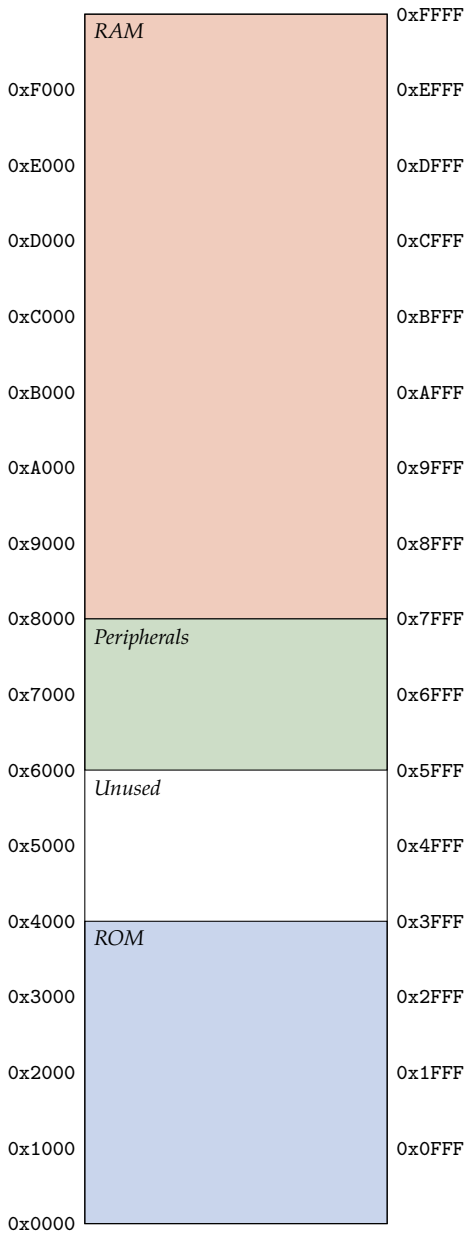
The analog input signal and the square wave output signal signal need to be connected to the microcontroller somehow. You need to decide how to do this and configure everything appropriately.

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.

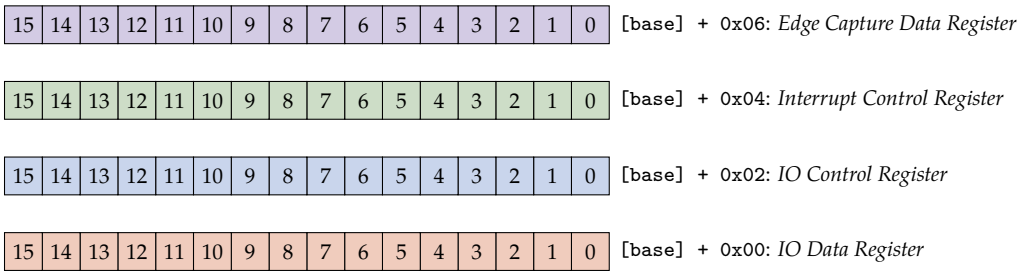
- For the purposes of this exam, if you are writing code in Assembly, it is sufficient to only explicitly provide the rows of the IVT that are used for interrupts — you may assume all other rows for IRQs that are not used already have the address of a suitable subroutine (like `unused`).

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

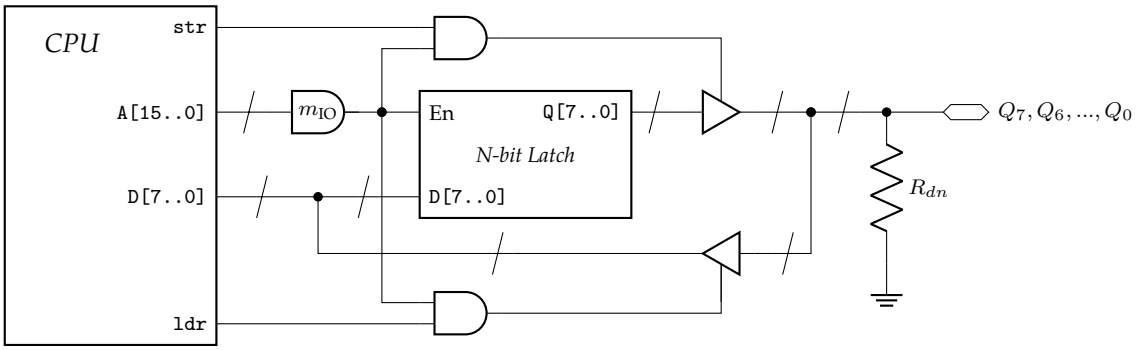
This model 16-bit microcontroller has six 16-pin general purpose input/output (GPIO) ports. These have the following structure.

- 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.

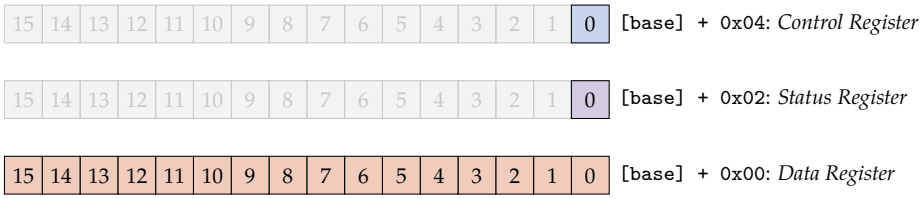
	GPIO A	GPIO B	GPIO C	GPIO D	GPIO E	GPIO F
Address	0x6000	0x6020	0x6040	0x6060	0x6080	0x60A0
IRQ	#2	#3	#4	#5	#6	#7



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.



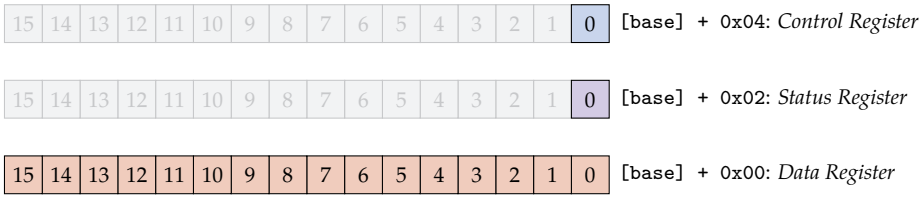
The timers' addresses and IRQs are as follows.

	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).

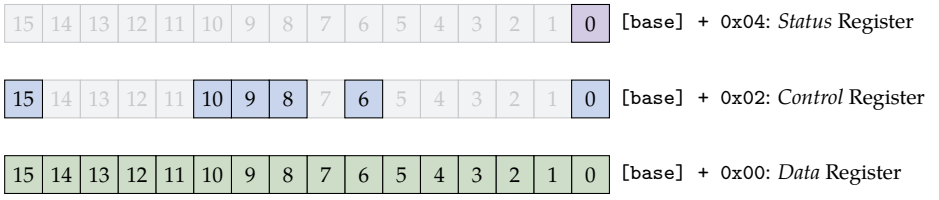


The real-time interrupt timer is at address `0x6800` with IRQ #10.

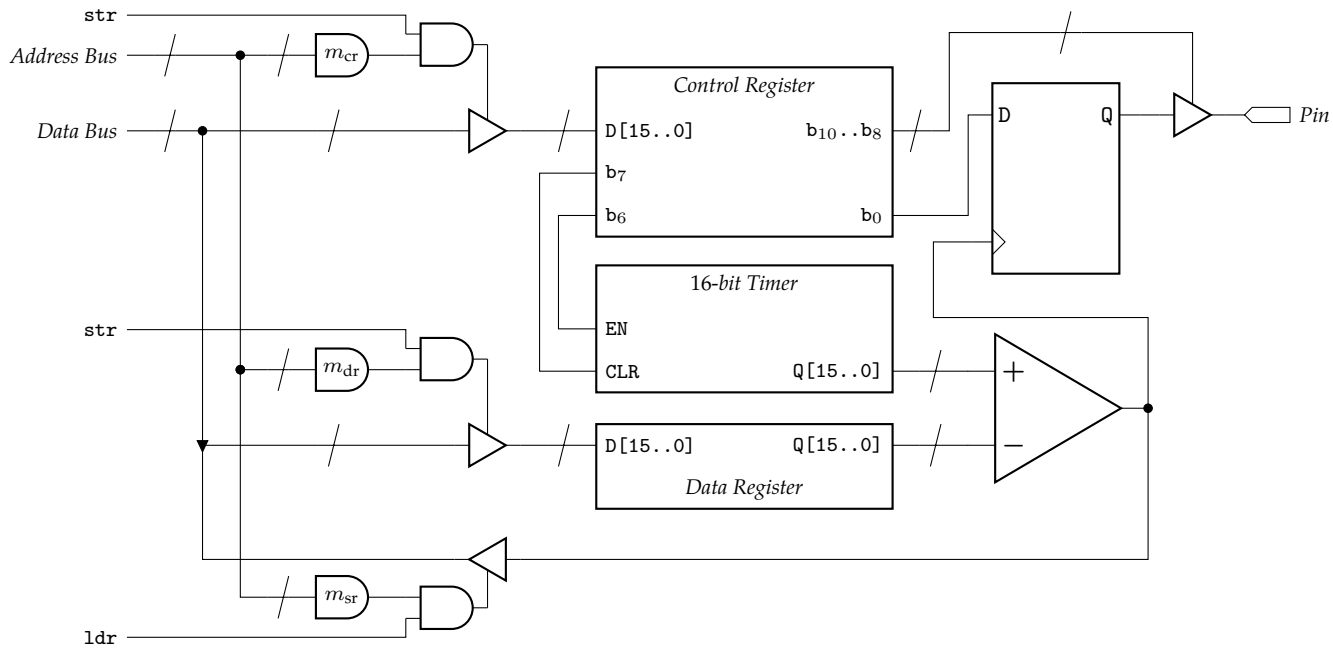
Output Compare

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.



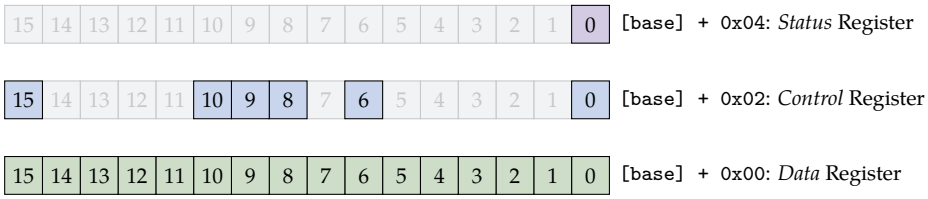
The output compare is at address `0x6C00`, with IRQ #11. If interrupts on the output compare are enabled, interrupts on the corresponding pin of GPIO port B should be disabled.



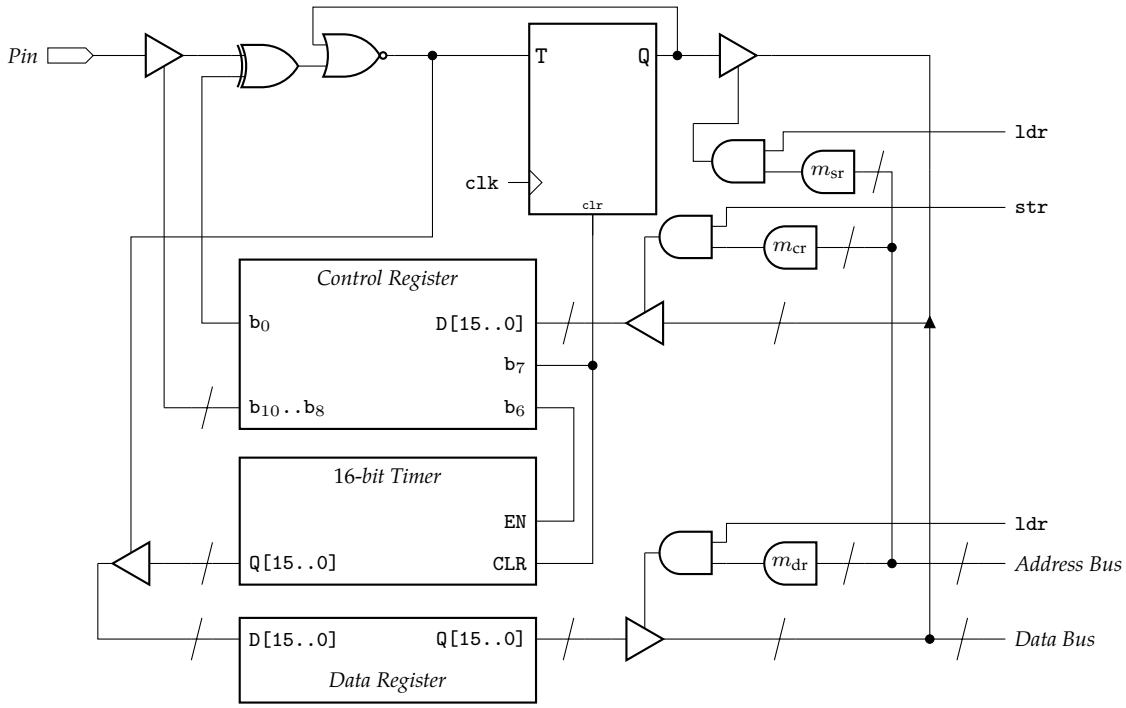
Input Capture

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.

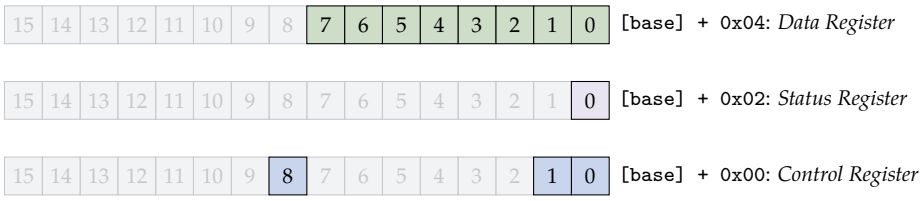


Analog-to-Digital Conversion

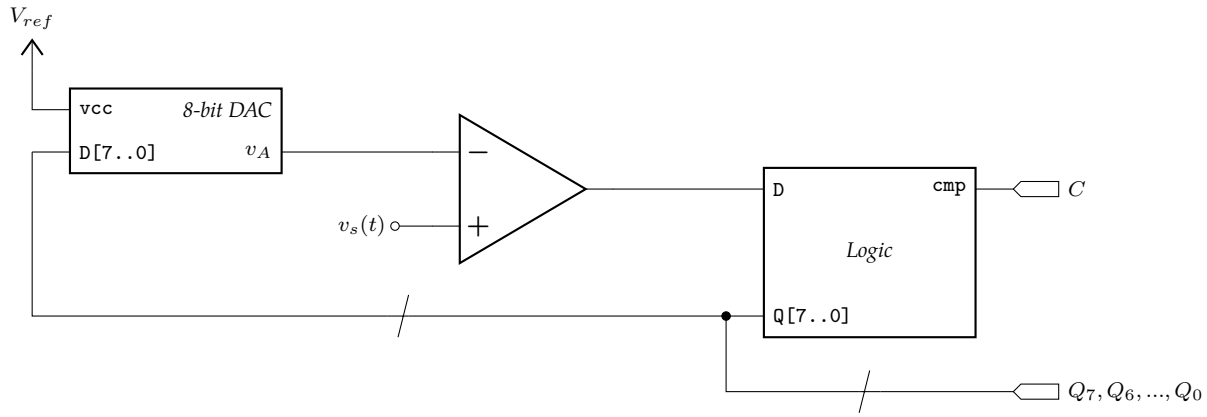
The model 16-bit microcontroller has one analog-to-digital converter (ADC). This has the following structure:

- 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.



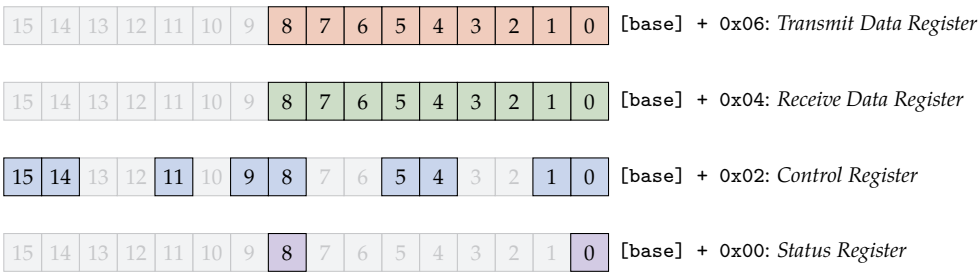
The ADC is at address `0x7000`, with IRQ #13. For this microcontroller, $V_{ref} = 5\,\mathrm{V}$.



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.