ECE342 Computer Hardware
Department of Electrical and Computer Engineering
University of Toronto

Lab 2: Reaction Timer

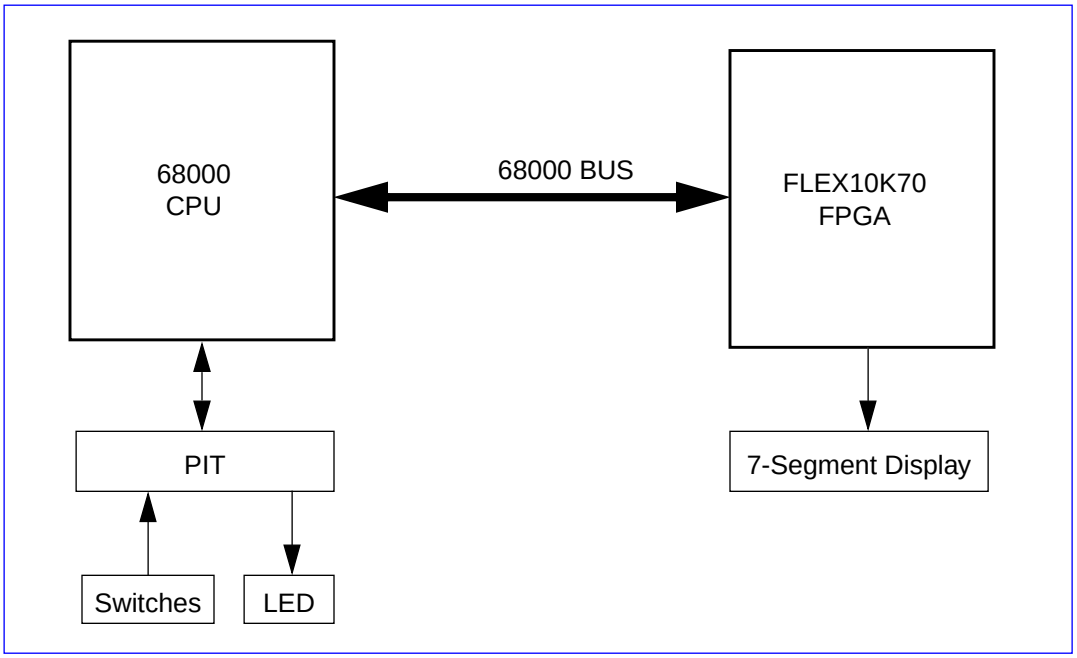
In this lab, you will implement a system to measure a persons reaction time to a visual stimulus. The system, called a reaction timer, consists of two inputs and a number of outputs. The two inputs are switches on the protoboard called *GO* and *STOP*. The outputs are: an LED on the protoboard and the 7-segment displays on the Ultragizmo board.

The reaction time is measured in the following way. The user toggles the GO switch to begin the test. After a short period of time, the reaction timer turns on the LED. As soon as the user sees the LED light up, he/she must toggle the STOP switch as fast as possible. The reaction timer measures the time from when the LED is turned on to when the user toggles the STOP switch. The time is to be displayed on the 7-segment displays in hundredths of a second. The reaction timer should then be ready for the next test and begin when the GO switch is turned on again.

You are to implement the reaction timer in three different ways. The first two ways (assembly and C) should be completed in the first week of the lab and the third way (full hardware) should be completed in the second week of the lab.

1. Assembly program with polling

The first way you will implement the reaction time is with an assembly program. To keep track of time you will use the DUART timer/counter which is integrated within the 68306 processor. Your program should read the values of the switches and drive the LED using the parallel interface/timer circuit (PIT) on the Ultragizmo board. The seven-segment displays should be accessed through a register implemented in the FPGA. A block diagram of the reaction timer and more information on each of the components is given below.



a) Timer/Counter

The DUART timer/counter is located in the serial module on the processor. The timer/counter is 16-bits wide and its clock can be programmed to come from one of a number of sources. It has two modes: timer and counter. We will only be using the counter mode in this lab. Once the counter is started, it counts down to zero from a preloaded value. When it reaches zero, `$0000`, it rolls over to `$FFFF` and continues counting.

Before the counter can be used it needs to be configured by writing values to a number of registers. The registers can be written just like any memory location. In the description below, only the names of the registers are given. The addresses are available in the file *rt_poll.s* available on the course webpage. You should use this file as a starting point for your program.

- **Selecting Counter mode and Setting the Clock Source:**

  To select counter mode a value of `$90` must be written to the *Auxiliary Control Register* (ACR). We will be using a clock of 19.2 kHz for Parts 1 and 2 of the lab. To set the counter clock to this value, write a `$CC` to the *Clock-Select Register* (CSR).

- **Preloading a Value and Reading the Count**

  The counter can be preloaded with a value by writing to the *Counter Upper* and *Lower Preload Registers* (CUR and CLR). The value of current count can be read by reading the CUR and CLR one at a time. The count value should only be read after the counter has been stopped.

- **Starting/Stopping the Counter**

  A read of the *Start Counter Command Register* (STC), starts the counter. A read of the *Stop Counter Command Register* (SPC), stops the counter.

- **Detecting a Rollover**

  Upon reaching `$0000`, the counter sets the *counter/timer ready* bit in the *Interrupt Status Register* (ISR[3]). By polling on bit 3 of the ISR, the reaction timer can detect when the counter has rolled over. Note that this bit is reset when the counter is stopped.

b) PIT

You are required to use the Parallel-Interface/Timer (PIT) port on the Ultragizmo board to access the switches and the LED on the protoboard. Since using the PIT has been covered in other courses, it will not be discussed here. For information, please refer to the Ultragizmo manual on pages 113-116. If you choose the I/O ports on the PIT wisely, you can connect the protoboard to the PIT with just a ribbon cable.

c) Seven-Segment Displays

You should write to the seven-segment displays the same way you did in Lab 1. Since you are displaying on all four displays, you will need to implement a 16-bit register in the FPGA. The values displayed should be a **decimal representation** of the reaction time. For example, if the reaction time is 0.95 seconds then you should display "`0095`" rather than "`005F`". Do not add

any special hardware to implement this functionality. The conversion is to be done in your assembly program.

d) Other Useful Info

For debugging purposes you may want to display intermediate results to the terminal. To refresh your memory on how to do this, read pages 96-99 of the manual.

2. C program with interrupts

This implementation of the reaction timer will be similar to Part 1 with two exceptions. The first is that this implementation will be written in the C programming language. The second is the way in which the counter rollover is detected. In Part 1 the rollover was detected by polling on bit ISR[3]. In this part, an interrupt service routine will be used to handle the rollover. The counter will be configured to generate an interrupt when `$0000` is reached.

a) Generating interrupts with the counter

Since the counter is an on-chip peripheral, it generates an internal interrupt. Some configuration needs to be done even for internal interrupts. Again, the registers are only given by name here and a file with the relevant register addresses is given on the webpage.

First, the level at which the counter will interrupt the processor needs to be specified. To interrupt at level 4, the value `$04` must be written to the *System Register* (SYSR). Next, a vector number needs to be chosen from Table 8 on page 101 of the manual. We will choose `64` by writing a `64` into the *Interrupt Vector Register* (IVR). Finally the processor needs to know the starting address of the interrupt service routine. This is done by writing the starting address of the interrupt service routine to the location corresponding to vector number 64 in the Interrupt Vector Table, location `$100`.

After you have set the registers above, you need to change the interrupt priority in the processors status register. You should set it to <4 to enable interrupts above this level (the counter is set to interrupt at level 4).

b) C programming Language

- **Accessing registers**

  The easiest way to access a registers is through a pointer to a character (`char`). We use chars because they are one byte long just like registers. For example:

  ```
  char *sysr;
  ```

  is a declaration of a pointer that will point to the system register. To assign the address of the register to the pointer:

  ```
  sysr = (char *)0xFFFFFFFE;
  ```

  To write a value to the register, you must dereference the pointer and assign it a value:

  ```
  *sysr = 0x4;
  ```

- **The interrupt service routine**

  The code you will write for the interrupt service routine is to be placed in a C function called *interrupt()*. To make it work, you are provided with the wrapping that allows a C function to be used as the interrupt service routine. The wrapping is located in a file called *cint.S* and is called *intrstub()*. It saves all the registers, calls the Interrupt function defined in C and uses the *rte* instruction instead of *rts* when it completes.

• Specifying the starting address of the interrupt service routine

Since addresses are long words (4 bytes), you should use a pointer to an `int` to access the location in the interrupt vector table where you will store the address of the interrupt service routine. Note that the address of *intrstub()*, and not *interrupt()*, will be stored in this location:

```
int *int_addr;
int_addr = (int *)(0x100);
*int_addr = (long)(intrstub);
```

• Writing to the hex displays

To access the 16-bit register implemented in the FPGA, declare a pointer to a `short int` (defined as 2 bytes long):

```
short int *hex;
```

• Files that need to be included and Makefiles

A stripped-down version of the C code, rt_int.c, is provided on the webpage. Use this as a starting point for your implementation. The necessary makefiles and other files you will need (*asm.h and cint.S, crt0.S*) are also there. You should place them all in the same directory.

To generate an srec file, type make rt_int. The makefile first generates the necessary object files (*crt0.o, rt_int.o, cint.o*). It then links them together to generate the executable *rt_int*. Finally, *rt_int* is converted to an srec file which can be downloaded to the Ultragizmo board.

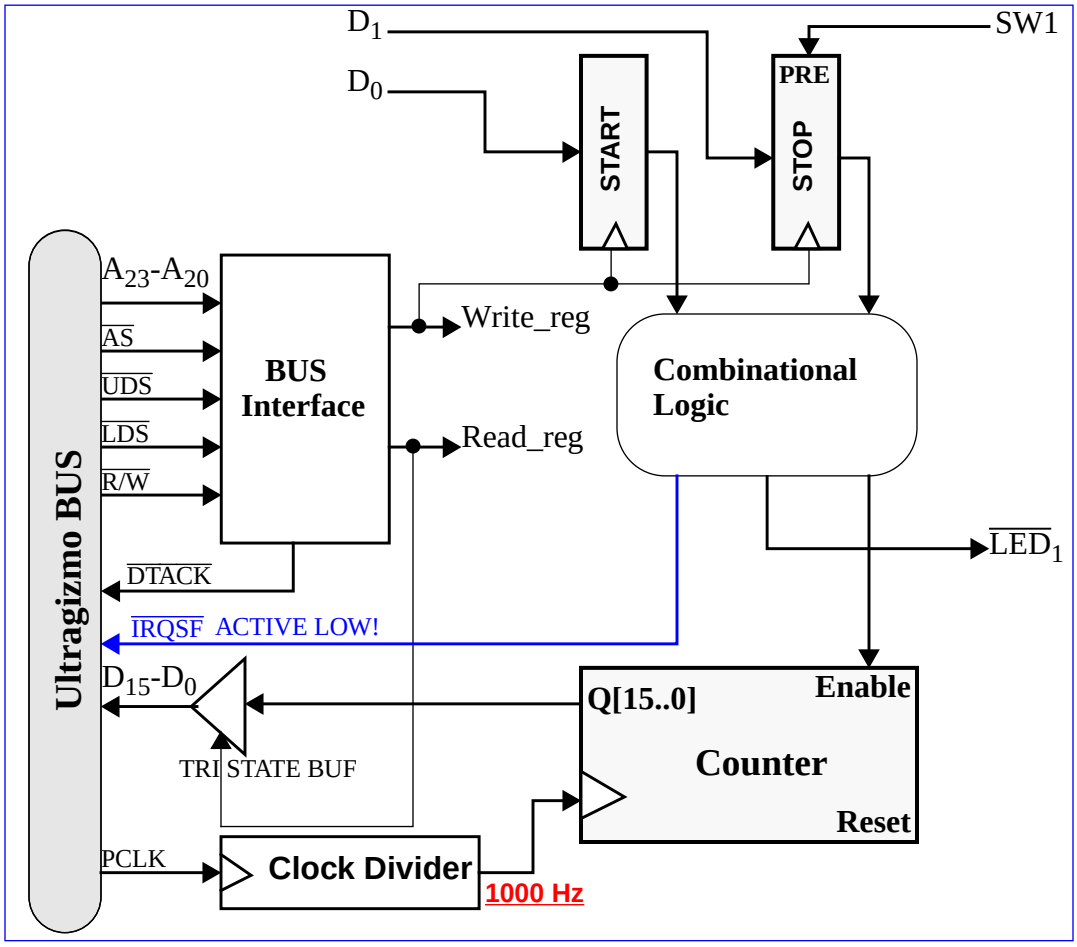
It is interesting to look at the code generated by the C compiler. To disassemble the executable use: `m68k-coff-objdump -d rt_int`. Can you see how the disassembled code relates to your C program? Even though they are not the same program compare your assembly code from part 1 to the code generated by the C compiler.

3. Full hardware implementation

In this portion of the lab, you are required to implement a hardware version of the reaction timer using the FPGA. In addition you will create interface circuitry to allow the reaction timer to communicate with the 68000 microprocessor. The 68000 should be able to initiate the reaction test while your reaction timer should be able to interrupt the 68000 when it has calculated the reaction time, and send the time over the bus. In this way, your hardware does all of the work and only interrupts processing once the reaction time is known. Thus no polling is required by the processor at all. It is free to do other useful computations. Each of the previous two versions has used polling to check when the user toggles the switch.

The Figure below shows the basic structure of the circuit that you are to implement. For simplicity, the GO switch will not be used in this implementation. You will write a short 68000 program that will communicate with the reaction timer, and initiate the test a short time after the program starts. **Note** that aside from the bus interface, this circuit is slightly different than the version given in **Section 7.14.3** of the course text, since it needs additional logic for proper interrupt handling.

.



Implement the circuit shown in the block diagram entirely in Verilog. The following is a more detailed description of the operation and the main parts of the design:

- Design a **Bus-Interface** circuit. This module should assert a signal named *Write_reg* when the 68000 requests a write to location $B00000 in memory. Similarly it should assert *Read_reg* when a read from $B00000 is requested. You may look at the *communicate.v* file given in Lab1 as a guideline for this circuit. Basically it is a simple combinational circuit that looks for when a "B" is on the upper nibble of the address lines, and all three of $\overline{AS}$, $\overline{LDS}$, $\overline{UDS}$ are low. Reads and writes are differentiated by the $R/\overline{W}$ signal. You must also generate an active-low acknowledge signal $\overline{DTACK}$, as soon as either *Read_reg* or *Write_reg* is asserted. This signal must be connected in an **open-drain** configuration.

- The **START** and **STOP** flip-flops are triggered by the *Write_reg* signal and latch the value on D_0,D_1 respectively. The 68000 can control the state of these flip-flops by writing to bit-0 and bit-1 of address $B00000. The circuit is initialized by writing a logic-1 to the **START** flip-flop and a logic-0 to the **STOP** flip-flop. Your combinational logic should respond to this condition by lighting up a LED on the Ultragizmo board and enabling the counter, so it keeps track of the reaction time. Notice that the **STOP** flip-flop has a preset input hooked to a switch on the digital board. When this switch is changed to a logic-1 the **STOP** flip-flop will immediately go high. Your combinational logic circuitry should now disable the counter enable, turn off the LED, and request an interrupt by bringing the $\overline{IRQSF}$ line low. Derive logic equations for combinational logic to produce the counter enable, interrupt request and LED drive signals.

- The Ultragizmo board is equipped with a programmable clock that outputs a signal named *PCLK*. This clock can be programmed to any frequency between 392kHz and 90MHz. Read **Section 8.7** of the Ultragizmo board manual to learn how to configure the clock frequency. Your reaction timer will use a counter clocked by a 1000Hz signal (we will measure milliseconds). Use an appropriate **clock-divider** circuit along with the programmable clock to generate the 1000Hz signal. What frequency would you set the programmable clock to? How many bits should the clock-divider use?

- The **16-bit counter** that will keep track of the reaction time. Since it is clocked by a 1000Hz signal, it will be able to keep track of reaction times up to 65.535 seconds. The counter should an *enable* input as shown in the block diagram so it does not start counting until the appropriate time.

- Notice that there is no **reset** signal connected on the counter. You will need to reset the counter every time after the first reaction test. Devise a method to allow the 68000 to reset the counter (the most simple solution just adds one extra wire to the circuit with no additional logic).

Implement each of the components discussed above in Verilog. In the top-level file *wrapper.v*, "wire" these components together so that they model the block diagram for the reaction timer. **Compile** and **simulate** each component, as well as the entire design. **Note** that the signals in *wrapper.v* may be labelled as lds,uds, etc., but they still correspond to $\overline{LDS}$, $\overline{UDS}$. You may use the signal *sfpga_digital[0]* to provide the **SW1** signal from the digital board. However it will be necessary to connect a 40 pin ribbon cable from the Ultragizmo board to the digital board. This will be demonstrated by a TA.

In most cases this circuit calculates the reaction time with a small error. What are the sources of this error, and what is its maximum value (in milliseconds)?

In order to use the circuit that you've created, it is necessary to write a 68000 program to initiate the test, and handle the interrupts. Write the software driver in 'C'. A skeleton program, rt_hw.c, is given on the webpage.

Examine the program carefully. You should notice that this program uses macros of the form ***pokeb****(address, data)*, ***pokew****(address, data)*, ***pokel****(address, data)*. These macros write the value of *data* into the memory location specified by the *address*. The ***pokeb*** macro writes byte sized data, ***pokew*** writes word (2-bytes) sized data while ***pokel*** write long (4-bytes) sized data. Similarly the program also uses the ***peekw****(address)* macro. It returns word sized data contained at the specified *address*. Code each of these macros in 'C' and place them in *poke.h*.

Create another makefile to compile this file. Add the following code to the template given:

- Add code to main that will initiate the reaction test. You need to write the appropriate value into the **START** and **STOP** flip-flops as discussed previously. Use the ***pokew*** macro.
- Add code to the *interrupt* routine so that after it reads the reaction time, it will de-assert the interrupt request. If you don't de-assert the request then the processor will continuously get interrupted. Again write the appropriate value into the flip-flops so that the combinational logic de-asserts the request.
- Add code to the interrupt routine to print out the reaction time to the terminal. Do this in two ways. The first method is to simply use the *printf* function. The second method is to write your own function to convert the time to a numeric string and print the string using the *puts* function. Compare the sizes of the srec files in the two different versions.