DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 391 – Computer Systems Design Studio
Fall 2015/2016 Term 2

Exercise 1.3
Adding a Serial Port

So far we have used hardware from a selection of devices that came with the Altera University Program files that we installed at the start of the course. In the real world, we may be designing our own microprocessor/CPU hardware devices (*you'll learn more about this in CPEN 412 - Microcomputer System Design if you take that elective*) or we may search the internet looking for designs, either free or paid for.

www.opencores.org is a great site to visit for VHDL/Verilog computer hardware - they have everything from simple parallel IO devices to full microprocessor cores that you can download and use for free in your FPGAs or ASICs.

In this exercise, we are going to use some of that free open cores IP and integrate it into our NIOS based computer via the Avalon-to-External Bridge circuit that you added to in Tutorial 1.9. **Marking**: When you have completed this exercise, show it to the TA for marking.

IMPORTANT: Because we want everyone to succeed on this course and it's possible that some of you may have got a *little* lost or may not have got previous tutorials and exercises fully working, you should download the ***predesigned NIOS II*** system associated with this exercise from Connect. It has the same sort of things that we have been doing so far, but it's a common starting point to move forward with this and the remaining exercises now that you have completed the tutorials.

Serial Ports

You may have encountered serial ports in other courses, for example an RS232 port. Serial ports are extremely useful as they permit simple 3 wire communication between a computer and a slave or peripheral device. All you need is a transmit data signal, a receive data signal and a common ground. Data is communicated 1 bit at a time at a fixed speed.

At one point it seemed as though the whole world was moving to very high speed USB, Fire wire, Serial ATA etc. for serial communication, but embedded microcomputer systems designers quickly realized that these standards were complicated from an Electrical, protocol and software perspective. For simple systems where data rates were low, a simpler serial interface actually makes more sense and is quicker/cheaper to develop. Below is a picture of some common slave/peripheral devices that can communication via a simple 3 wire serial port.

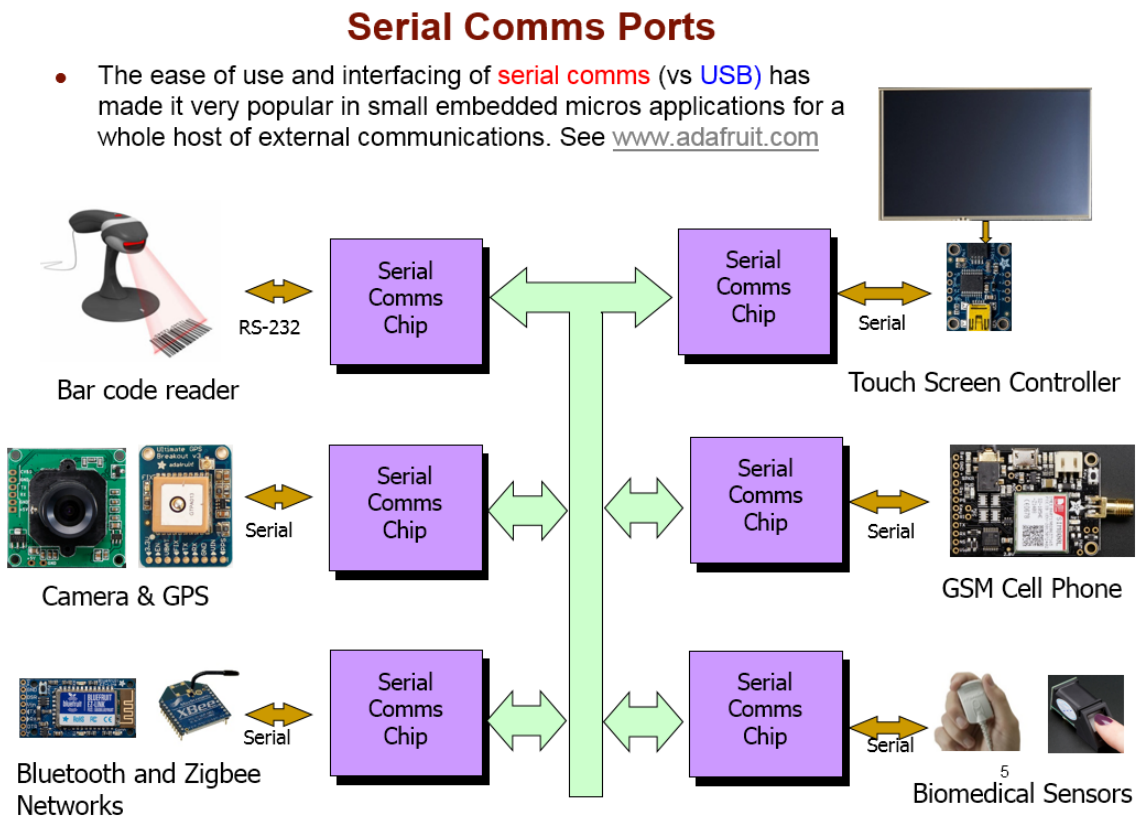


Figure 1

The end goal of the next few exercises is to interface a Touch Screen, GPS chip and Bluetooth communications to our computer system by adding 3 serial ports to our design. Each serial port will be based around a Motorola 6850 chip (a VHDL design we obtained from opencores.org) and a programmable baud rate generator (again from opencores.org plus some tweaks). The CPU can control these devices by writing/reading to them as if they were memory locations using 'C' style pointers.

If you are unfamiliar with serial communications and/or RS232 then please take a look at the overview of serial communication and the RS232 standard on Connect. Note that we will be using simple *digital* serial comms, not full RS232 which uses different voltage levels.

Step 1:

The University program files installed for Quartus already have provision for a serial port and you can use Qsys to add a serial port to your computer. In fact there is one already in the version of the computer that you will (optionally) *download* for this exercise (see the image below)

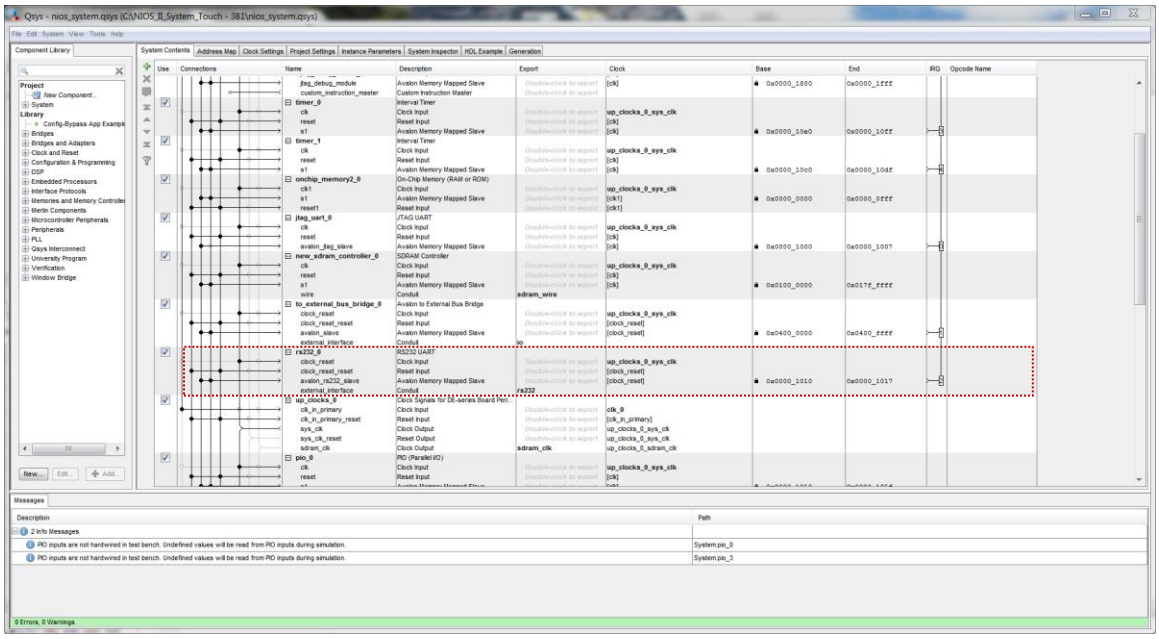


Figure 2

You can see that this has been brought out on the top level schematic for connection to the real world (see below). The Rx input from the RS232 port is labelled STDIN, while the Tx output is labelled STDOUT. You can read about the University Program serial port on Connect (*see document Altera IP – RS232.pdf*)

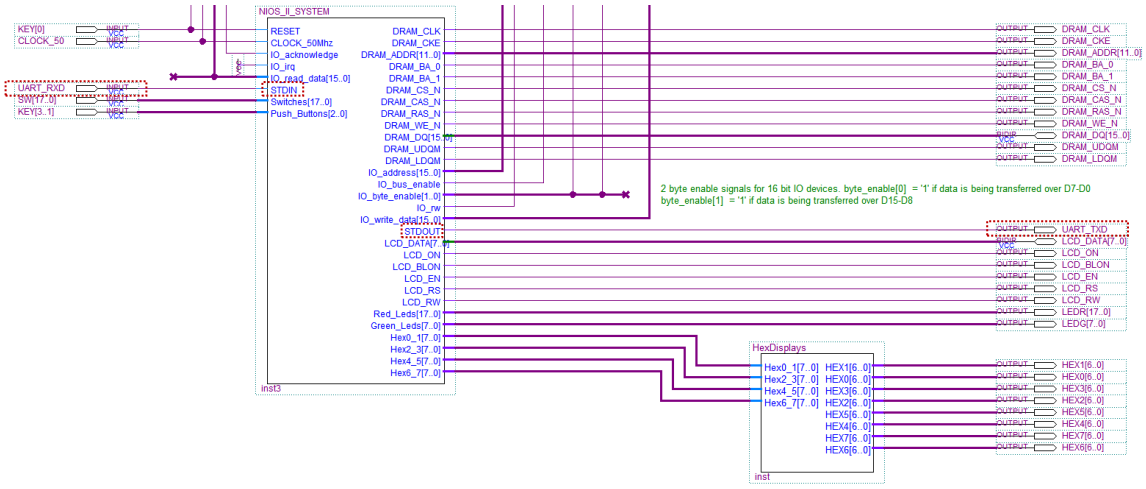


Figure 3

At present these are connected to pins UART_RXD and UART_TXD and are hard wired in the Quartus Pin Planner to the RS232 serial port connector on the DE2

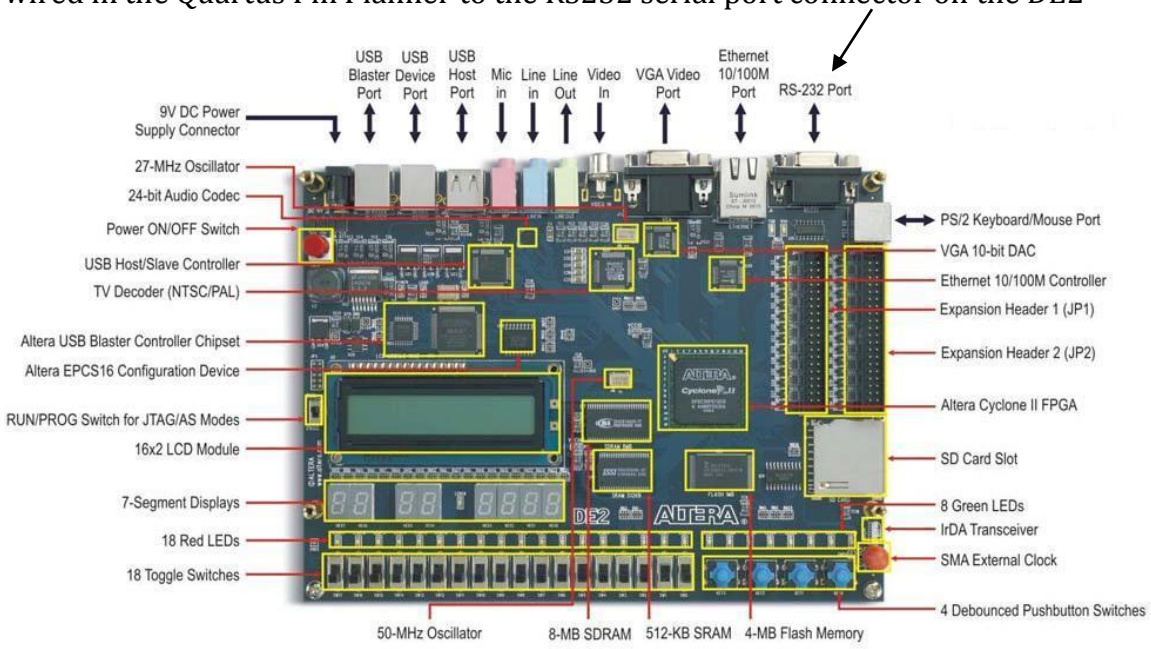


Figure 4

Making our own Serial Ports

Rather than use the built in Qsys files that are part of the University Program files from Altera, we'll make our own "raw" serial ports from "scratch" using VHDL files downloaded from OpenCores.org and write our own low level 'C' functions to drive them.

On Connect you will find 3 VHDL files related to this exercise that have been downloaded from opencores.org. Three of them ACIA_6850.vhd, ACIA_RX.vhd and ACIA_TX.vhd form part of the 6850 serial comms chip.

Two other files ACIA_Clock.vhd and Register3bit.vhd are part of the clock or programmable baud rate generator used to drive the 6850 chips. Add these files to your Quartus project (*the working one you have downloaded from Connect*) - to add the files, copy them to the project folder with the other VHDL files, then open them one at a time (remembering to tick the "**add files to current project**" button).

Create symbols for the ACIA_6850.vhd, ACIA_Clock.vhd and Register3bit.vhd. To do this open the file and select menu File->Create/Update->Create Symbol Files for Current File. This will give us a symbol we can paste onto a schematic diagram.

Now open the Block diagram or schematic (.bdf) file ACIA_BaudRate_Generator.bdf in your project. It should contain just input and output pins like this.

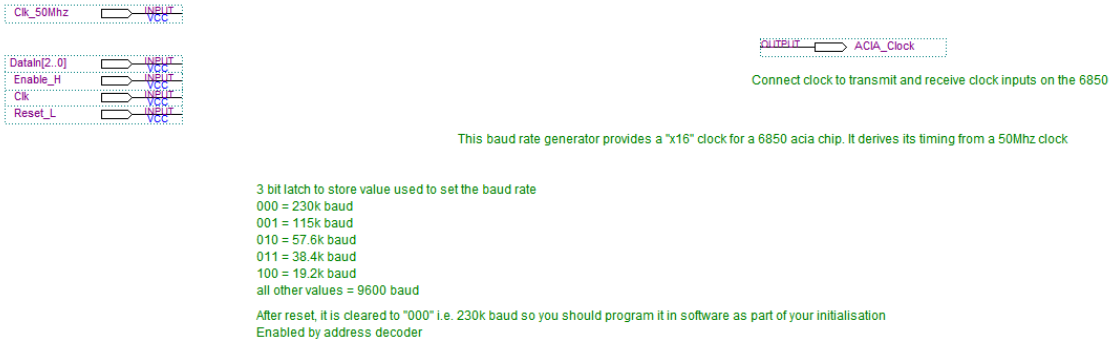


Figure 5

Paste the new symbols and wire up the circuits as shown below. Save the file. This circuit is a programmable baud rate (clock) generator and provides a circuit to create a "**x16**" clock for the 6850 ACIA chip to set the baud rate or communication speed. The 3 bit register on the LHS can be written to by the CPU to set the baud rate for the 6850, i.e. define the speed of communication.

Later on you will write C code to program this register as part of your initalisation routines. Different chips communicate at different baud rates (e.g. the Touch Screen and GPS chip runs at 9600 baud - the Bluetooth at 115k baud) so you have to look up the speed in the datasheets and program the baud rate generator accordingly.

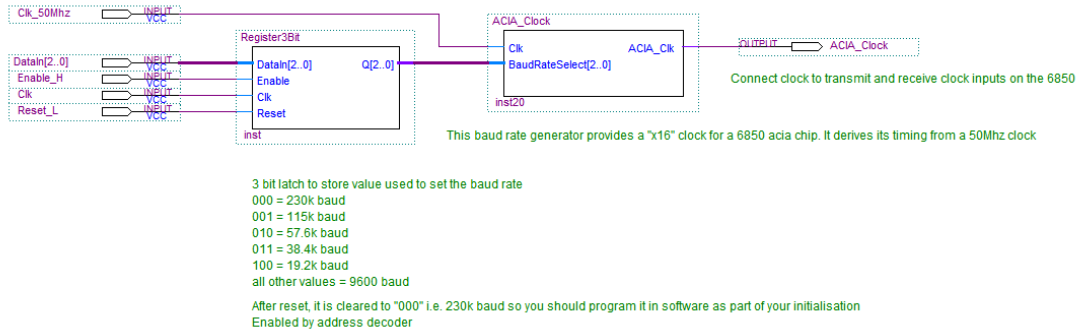


Figure 6

Now **create** a symbol for the above schematic, like we did above for the other VHDL files.

Step 2:

Open the file OnChipM68xxIO.bdf and paste symbols for your baud rate generator and ACIA 6850 and wire up as shown below. Note carefully the Signal bottom left labelled Address[1]. This 1 wire signal comes from the 16 bit address bus and 1 wire from that but (Address[1]_ connects to the RS or register select signal on the ACIA_6850 chip. To set the name, right mouse click on the wire and select properties and type in Address[1]. Likewise the DataIn bus that connects to the Baud rate generator is connected to the lower 3 bits of the data bus (right click to set wires properties and name is DataIn[2..0] as shown below.)

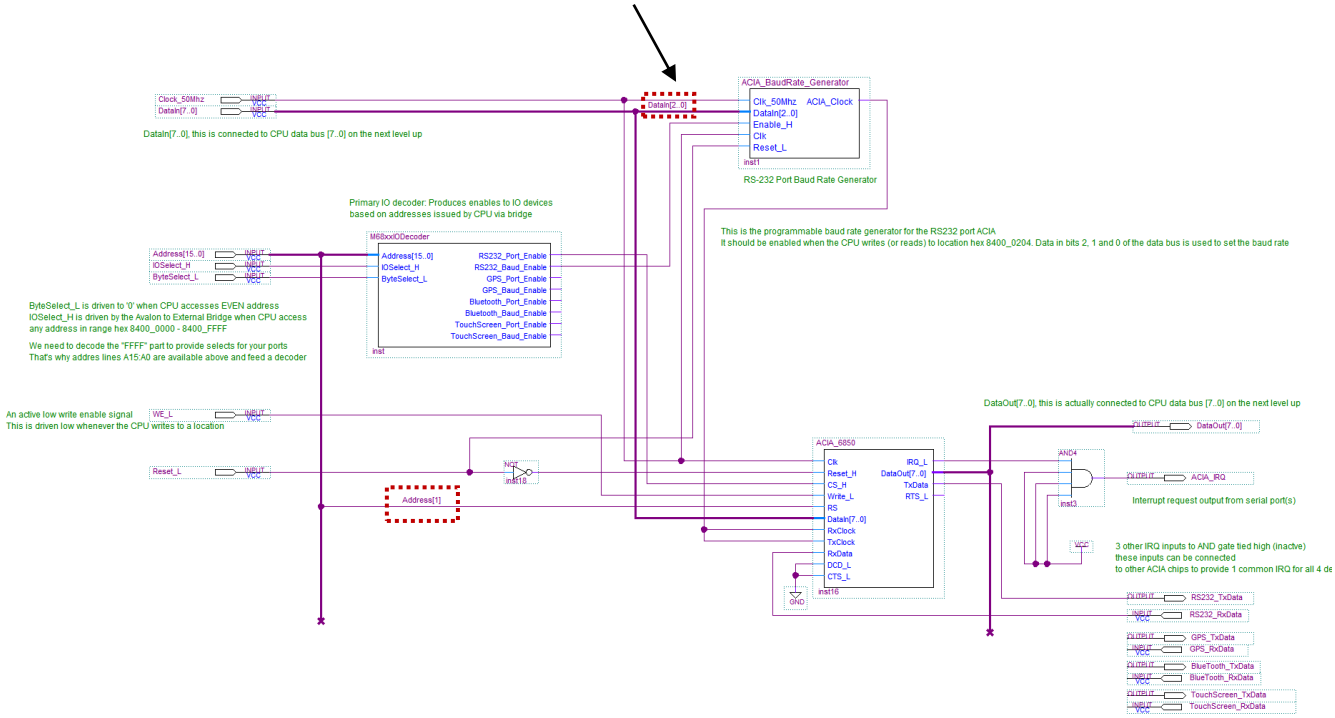


Figure 7

The above circuit contains a complete ACIA 6850 and a programmable baud rate generator for a single serial port. Later in this exercise we will connect this to the actual RS232 port connector on the DE2 in place of the University program one already connected (*see figures 3 and 4*). Note the pins bottom right, for connection to the 3 serial port peripherals: - GPS chip, Bluetooth dongle and Touch screen.

Note also the M68xxIODecoder circuit shown above. This generates various "enable" signals to activate the 3 serial ports (each has a 6850 + baud rate generator) when the CPU reads or writes to various addresses associated with those ports. The signals on the LHS, and those that go to the above M68xxIODecoder come from an instance of "Avalon to External IO Bridge" that we created in **Tutorial 1.8**. We'll come back to that later.

If you double click on the M68xxIODecoder block above you will open up the VHDL file shown below:-

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity M68xxIODecoder is
	Port (
		Address			: in std_logic_vector(15 downto 0) ;
		IOSelect_H		: in std_logic ;
		ByteSelect_L		: in std_logic ;

		RS232_Port_Enable	: out std_logic;
		RS232_Baud_Enable	: out std_logic;
		GPS_Port_Enable		: out std_logic;
		GPS_Baud_Enable		: out std_logic;
		Bluetooth_Port_Enable	: out std_logic;
		Bluetooth_Baud_Enable	: out std_logic;
		TouchScreen_Port_Enable	: out std_logic;
		TouchScreen_Baud_Enable	: out std_logic
	);
end ;


architecture bhvr of M68xxIODecoder is
Begin
	process(Address, IOSelect_H, ByteSelect_L)
	Begin

-- these are the default values that will be output to 6850 ACIA and Baud rate generator chip enables
-- the default value is low i.e. disabled. Default values are necessary to avoid inferring latches in VHDL
-- remember in VHDL unless an output is assigned a value through each and every path through a process, a
-- latch will be created to remember a previous value - we don't want memory in our address decoder,
-- we only want combinational logic
-- Therefore the easiest way to do this is to make sure each output is assigned a value at the START of a process,
-- then, it doesn't matter what VHDL if-else logic comes next, the output will always have ben assigned a value
-- so no latches will be inferred
-- Of course we can override the default output ant any time we like within our if-else VHDL statements
-- we must override them below when the CPU outputs an address for the 6850 or baud rate generator

		RS232_Port_Enable	<= '0' ;
		RS232_Baud_Enable	 <= '0' ;

		GPS_Port_Enable		<= '0' ;
		GPS_Baud_Enable		<= '0' ;

		Bluetooth_Port_Enable	<= '0' ;
		Bluetooth_Baud_Enable	 <= '0' ;

		TouchScreen_Port_Enable	<= '0' ;
		TouchScreen_Baud_Enable	<= '0' ;


-- IOSelect_H comes from the Avalon to External Bridge (see tutorial 1.8a) and is driven to logic 1 whenever the
-- CPU outputs an  address in the range A31:A0 = hex [8400_0000] to [8400_FFFF]. -- that is, IOSelect_H is active
-- high for all addresses in the range [8400_XXXX]. All we have to do for is decode the XXXX i.e. the lowest 16
-- address lines (A15:A0) to provide various chip enables.
-- Both the 6850 ACIA and the baud rate generates are connected to the lower half of the data bus (D7:D0) and
-- thus occupy EVEN valued addresses in the CPU address space e.g. 8400_0000, 8400_0002 etc.
-- ByteSelect_L is asserted for an even byte transfer of D7-D0

-- example decoder for the 1st 6850 chip (RS232 Port)
-- 2 internal registers at locations 0x8400 0200 and 0x8400 0202 so that they occupy same half of data bus
-- on D7-D0. Enabled when ByteSelect_L = 0 (i.e. activated when the data is being transferred over data bus
-- decoder for the Baud Rate generator at 0x8400_0204 on D7-D0 and ByteSelect_L = 0
```

```
                    if(IOSelect_H = '1') then                    -- if Avalon to External Bridge being accessed by CPU

-- if address in range hex 8400_020X i.e. 8400_0200 - 8400_020F  i.e. 16 bytes in total even though not all used

                                if((Address(15 downto 4) = X"020") and ByteSelect_L = '0') then

-- if address is hex 8400_0200 or 8400_0202

                                        if((Address(3 downto 0) = X"0") OR (Address(3 downto 0) = X"2")) then
                                                RS232_Port_Enable <= '1' ;              -- enable the ACIA device
                                        end if ;

-- -- if address is hex 8400_0204 enable baud rate generator

                                        if(Address(3 downto 0) = X"4") then
                                                RS232_Baud_Enable <= '1' ;
                                        end if ;
                                end if ;
                        end if ;

-------------------------------------------------------------------------------------------------------------------------------
-- add other decoders for 3 more 6850 ACIA and 3 more baud rate generators,
-- one for the GPS, Bluetooth and Touch screen devices
-- Make sure IOSelect_H is logic 1 and ByteSelect_L = logic 0, then decode the address and produce
-- and produce the relevant enable output
-- to override the default output set above
-------------------------------------------------------------------------------------------------------------------------------


-- decoder for the 2nd 6850 chip (GPS)- 2 internal registers at addresses 0x8400_0210 and 0x8400_0212 so that
-- they occupy same half of data bus on D7-D0 and ByteSelect_L = 0
-- decoder for the Baud Rate generator. 1 internal register at addresses 0x84000214 on D7-D0
-- and ByteSelect_L = 0

-- your VHDL code goes here

-- decoder for the 3rd 6850 chip (Bluetooth)- 2 internal registers at addresses 0x8400_0220 and 0x8400_0222
-- so that they occupy same half of data bus on D7-D0 and ByteSelect_L = 0
-- decoder for the Baud Rate generator. 1 internal register at address 0x8400_0224 on D7-D0
-- and ByteSelect_L = 0

-- your VHDL code goes here

-- decoder for the 4th 6850 chip (Touch Screen)- 2 internal registers at addresses 0x8400_0230 and x8400_0232
-- so that they occupy same half of data bus on D7-D0 and ByteSelect_L = 0
-- decoder for the Baud Rate generator. 1 internal register at address 0x8400_0234 data on D7-D0
-- and ByteSelect_L = 0

-- your VHDL code goes here

            end process;
END ;
```

This simple circuit looks at the signals produced by the "NIOS to Avalon bridge" and decides which 6850 or baud rate generator circuit to activate, based on the address the CPU is issuing - remember we designed the bridge to produce a BusEnable signal (which is connected to IO_Select_H above) whenever the CPU accesses an address in the range **0x8400_0000 to 0x8400_FFFF** i.e. a block of 64k bytes in reserved for external IO devices.

Step 3

- Add 3 more instances of a 6850 and a baud rate generator circuit to the above schematic and wire up in a similar manner to the 1st serial port. That is, each 6850 should have its own associated baud rate generator so that its speed can be programmed independently of the other serial ports.

- Make sure you connect the various "enable" signals for these chips to the correct output on the decoder.

- Remember to connect the TX and RX signals from each of your new 6850 based serial ports to the associated 3 sets of pins shown in the schematic e.g. GPS_TxData, GPS_RxData etc.

- Connect the IRQ outputs from the 3 new 6850 ACIA chips to the 4 input AND gate in the schematic, replacing one of the inputs connected to Vcc. You can eventually delete the Vcc symbol when all 3 ports have been wired up.

- Finally add your own VHDL code to the above (i.e. the file M68xxIODecoder.vhd) to produce the various enable output signals to activate the 3 remaining serial ports (6850 + associated baud rate generators) so that the enables correspond to the addresses given in the comments.

Step 4

Once you have completed the above schematic and written your own VHDL for the decoder, create a new symbol for the above schematic.

Now open the top level schematic for the project and paste a copy of your new IO and connect to the Avalon to External IO bridge signals coming out of the NIOS II processor as shown below and wire accordingly. Make sure you label signals properly (by setting their properties), in particular, IO_read_data[7..0] and IO_write_data[7..0] and the NOT gate connected to IO_byte_enable[0] as shown below – double check your connections.

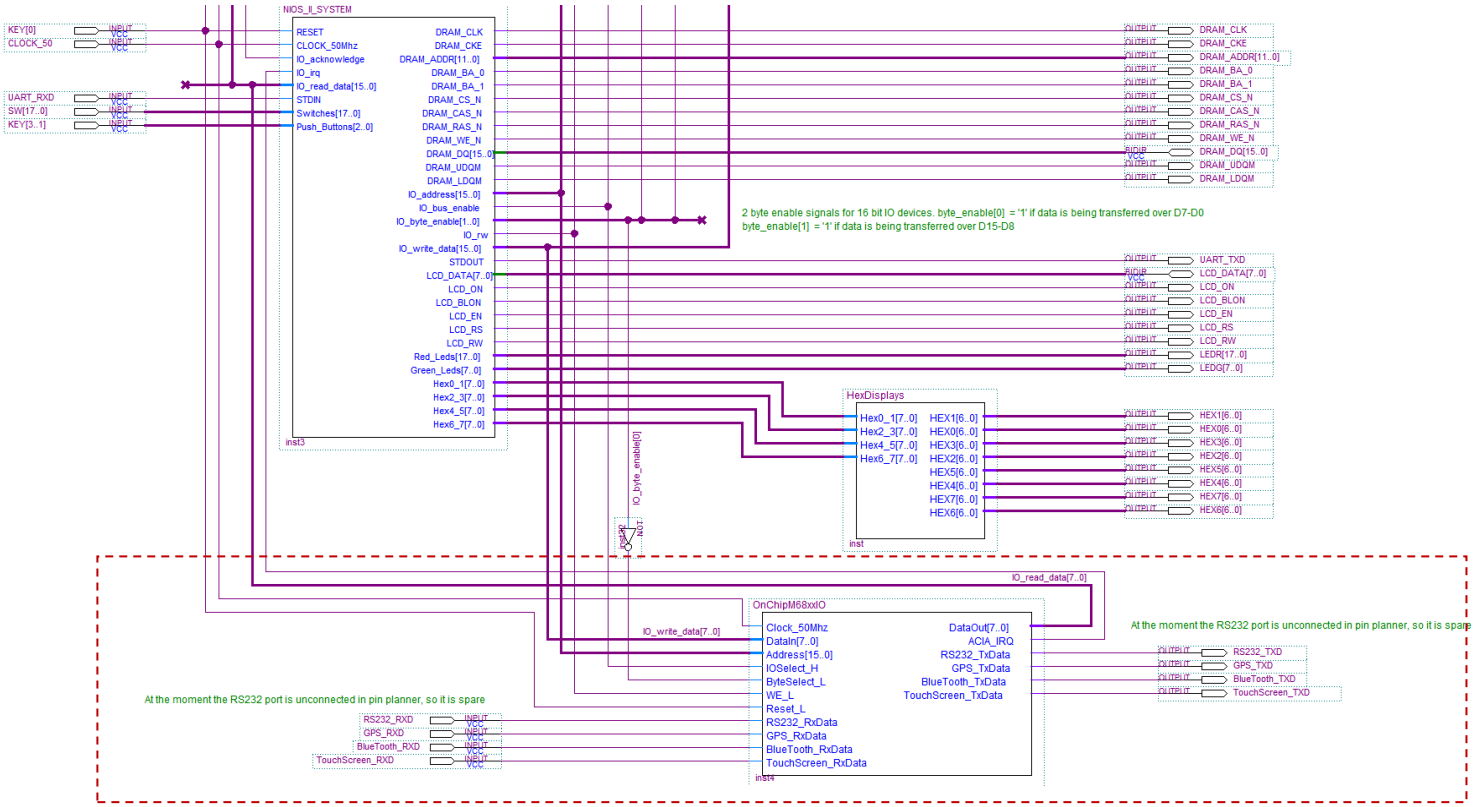


Figure 8

Note the pins associated with the GPS, Bluetooth and Touchscreen serial ports above have already been set up in the Quartus II Pin planner. Just check to make sure. They should be set to use GPIO connector pins - you can change them if you wish. However the RS232 ports signals labeled RS232_TXD and RS232_RXD above have not.

Let's wire up the RS232_TXD and RS232_RXD to the real serial port connector on the DE2 (*see figure 4*). To do this, delete the RS232_TXD and RS232_RXD pins in Figure 8 above. Detach the pins UART_TXD and UART_RXD also from figure 8 above (*and delete the wires that connected them to the NIOS II symbol.)*

Now drag the UART_TXD and UART_RXD pins and connect them to the RS232_TxData and RS232_RxData signals coming from OnChip68xxIO symbol as shown below in figure 9. This should be wired up in the Quartus II Pin planner (*maybe a good idea to check*).

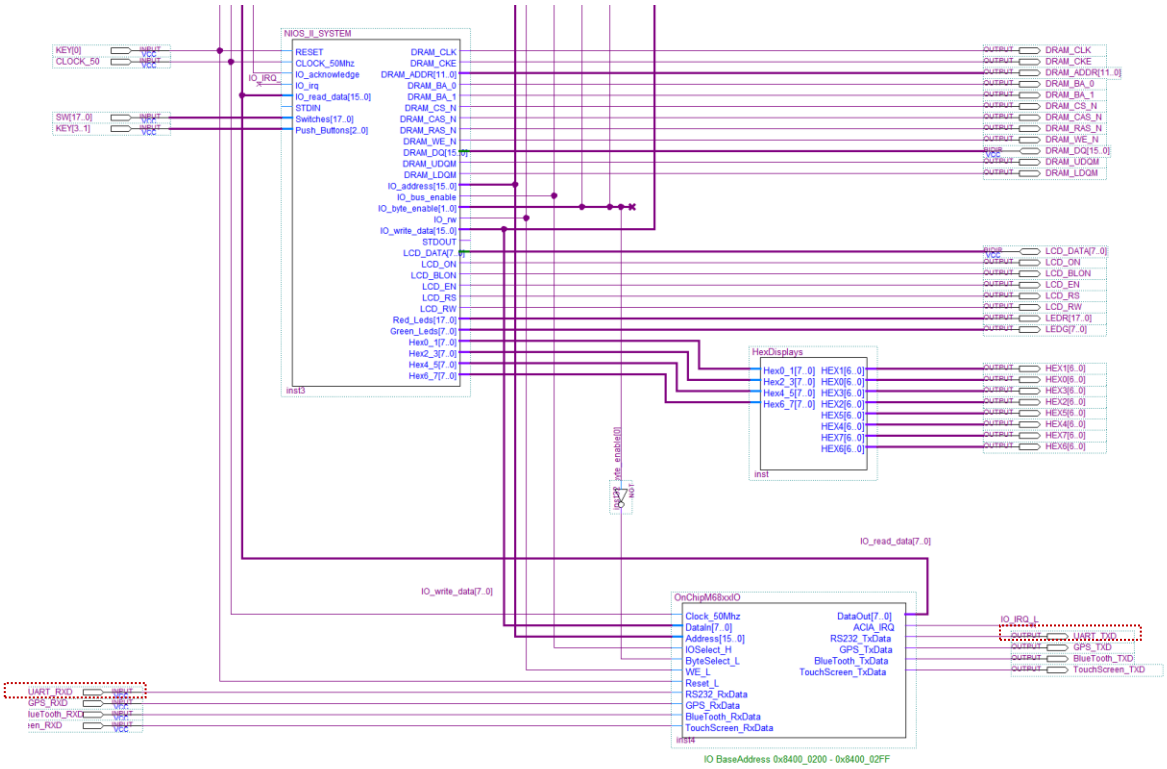


Figure 9

Save the files and make sure everything compiles in Quartus without errors - download it to the DE2 and make sure the system still comes to life.

Driving a 6850 ACIA and Baud Rate Generator in 'C'

The Avalon to External IO Bridge in the computer above has been designed to occupy a 64k byte space in the NIOS II memory map starting at location 0x0400_0000 and ending at address 0x0400_FFFF. The bridge was also designed to have a 16 bit data bus.

If NIOS performs read or write operations to any addresses within these two bounds, then the Avalon to External Bridge will be activated and data will be transferred to a Slave device, i.e. our 6850 serial comms chips and baud rate generator.

The 6850 and baud rate generator are in fact byte wide peripherals (having an 8 bit data bus), but even so, we can still connect them to the 16 bit wide data bus coming from the bridge by connecting them to one half of the bridge data bus, either D15-D8, or D7-D0. And then enable them via one of the ByteEnable[1] or [0] signals respectively.

This has been done on the hardware above where we connected the 6850 and baud rate generator to Data bus lines D7-D0 and enabled them via ByteEnable[0] (go review the design above to make sure you understand this). In order to make sure ByteEnable[0] will be driven when the CPU access the bridge address space, we must ensure that our programs transfer a byte of data to an *even* valued address (if our program wrote a byte to an *odd* value address, the bridge would activate ByteEnable[1] and transfer the data over D15-D8 which would mean that the data would never reach our chips and they would never be activated).

We wrote the VHDL for the M68XXIODecoder so that each of the 6850/baud rate generator chips would respond to a subset of *even* valued addresses reserved within the bridges 64k byte address space. For example the RS232 baud rate generator is activated when the CPU accesses memory location **hex 0400_0204**

Note however, that we want to ensure that the *data cache* is bypassed when using the *NIOS/f* variant of the CPU, so we have to make sure that any address issued by code in our programs sets address line **a31** to '1'. Thus, the address our program should issue to access the RS232 baud rate generator is actually
hex 8400_0204

Using Pointers to Program the 6850 and Baud Rate Generator chips

In C, we can set up pointers to byte wide registers using **#define** statements like this

```
#define RS232_Control     (*(volatile unsigned char *)(0x84000200))
#define RS232_Status      (*(volatile unsigned char *)(0x84000200))
#define RS232_TxData      (*(volatile unsigned char *)(0x84000202))
#define RS232_RxData      (*(volatile unsigned char *)(0x84000202))
#define RS232_Baud        (*(volatile unsigned char *)(0x84000204))
```

The "`unsigned char *`" bit of the declaration ensures a pointer to byte wide data is created, so that when we use it bytes of data will be transferred. The `volatile` bit stops the compiler making any clever assumptions about whether to access the real address or simply to generate assembly language code to cache the last value inside a CPU register (*nothing to do with the data cache on NIOS*). To write a byte to the Control Register of the 6850, we could write something like this

```
RS232_Control = 0x55 ;
```

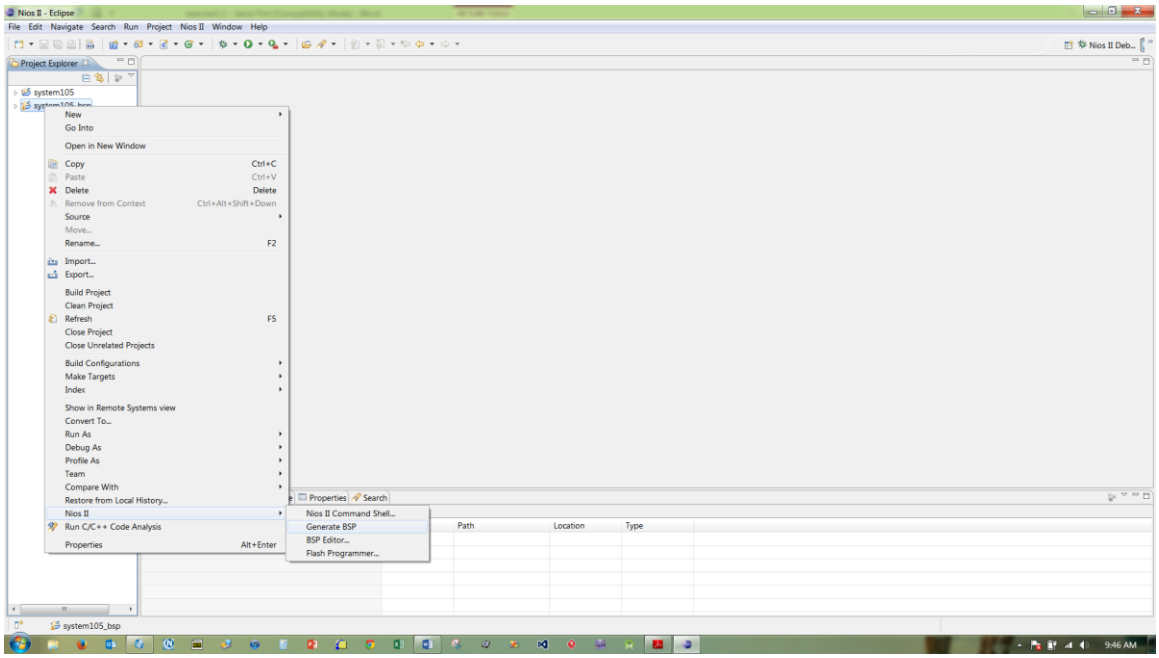
To program the Baud Rate Generator, we could write something like this

```
RS232_Baud = 0x01 ;        // program for 115k baud
```

Note we didn't need to prefix any of the above defined constants with a '*' i.e. we didn't need to say ***RS232_Baud = 0x01** to store a value, since the '*' was already included in the define - it's a personal choice to include it in the define, or each time you use it in your code.

Step 5

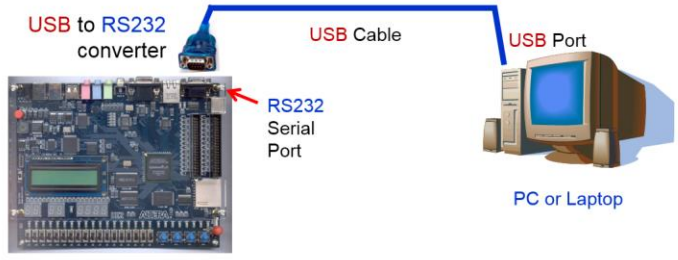
Write the following functions to Initialise and use the 6850 and Baud Rate Generator. Use the Eclipse development environment as before to write this code. Note every time you recompile a NIOS II system in Quartus you have to have to "Generate a new .BSP" file for Eclipse - see below




```
/*****************************************************************************
```

```
/*  Subroutine to initialise the RS232 Port by writing some data
**  to the internal registers.
**  Call this function at the start of the program before you attempt
**  to read or write to data via the RS232 port
**
**  Refer to 6850 data sheet for details of registers and
*****************************************************************************/

void Init_RS232(void)
{
    // set up 6850 Control Register to utilise a divide by 16 clock,
    // set RTS low, use 8 bits of data, no parity, 1 stop bit,
    // transmitter interrupt disabled
    // program baud rate generator to use 115k baud
}

int putcharRS232(int c)
{
        // poll Tx bit in 6850 status register. Wait for it to become '1'

        // write 'c' to the 6850 TxData register to output the character

      return c ;    // return c
}

int getcharRS232( void )
{
    // poll Rx bit in 6850 status register. Wait for it to become '1'

    // read received character from 6850 RxData register.
}

// the following function polls the 6850 to determine if any character
// has been received. It doesn't wait for one, or read it, it simply tests
// to see if one is available to read

int RS232TestForReceivedData(void)
{
    // Test Rx bit in 6850 serial comms chip status register
    // if RX bit is set, return TRUE, otherwise return FALSE
}
```

It's not necessary for this exercise, but you might like to test this code by connecting an "RS232 to USB convert " between the serial port and your laptop and using a program such as Hyper terminal to send/receive data to the DE2 (see below).



Later on, when we connect the serial ports to the touch screen, GPS and Bluetooth devices we can use similar code. When you have done this exercise, present it to the TAs for marking.