MCC150 - Implementation of Digital Signal Processing Systems

Lab 1: BPSK Transmitter

Sining An, Erik Börjeson
Zhongxia Simon He, Per Larsson-Edefors

Version 1.3 - April 14, 2021

Contents

1 Introduction

During the lab sessions in this course you will work with MathWorks' Simulink and Intel's DSP Builder software to design DSP systems than can be run on an FPGA platform, in this case the Cyclone V FPGA mounted on the Terasic DE10-Standard development board. Simulink is a graphical programming language used for modelling a system and simulating its behavior. DSP Builder can be described as a plug-in to Simulink, used to generate hardware descriptions of DSP algorithms for Intel FPGAs. These hardware descriptions will then be imported into Intel Quartus Prime, synthesized and uploaded to the FPGA. The lab instructions are based on MATLAB 2017b and version 18.0 of Intel's FPGA Software, Standard edition, which can be found at `http://fpgasoftware.intel.com`. Instructions on how to install the software on your own computer can be found on Canvas.

This document starts with a preparation section, which should be read before starting the lab. It briefly describes the tools and hardware that you will use during the lab along with pointers to reference manuals and data sheets. Section 3 contains the tasks that are to be performed during the lab session, and Section 4 contains home assignments to be handed in on Canvas before 23:59 on Friday, April 17.

2 Pre-Lab Reading

The exercises in this lab series are based around Intel's DSP Builder for FPGAs. DSP Builder is a number of libraries for Simulink that can be used to implement the desired DSP architecture, simulate it, and generate a hardware description in either VHDL or Verilog. This hardware description can be imported into Intel Quartus Prime, compiled and uploaded to an FPGA. There are two types of blocksets (library groups) available for DSP Builder, named Standard and Advanced. In this lab series we will only use the latter one.

2.1 DE10-Standard and ARRadio Daughter Card

The main hardware used in this lab series is the DE10-Standard development board designed by Terasic, which contains an Intel Cyclone V FPGA, featuring a dual-core ARM processor. The DE10-Standard board also contains support circuitry to program and run the FPGA, and a wide range of peripherals such as memories, displays, buttons and other interfaces. Reference documents, such as the user manual [1] and board schematics [2] can be found on Canvas.

The Terasic ARRadio card contains an Analog Devices AD9361 RF tranceiver chip with surrounding support circuitry, and is connected to the DE10-Standard board using a high speed mezzanine card (HSMC) connector. The user manual [3] and schematics [4] for the daughter board, and the reference manual for the RF chip [5] can be found on Canvas, should they be needed.

A simplified block diagram describing the ARRadio daughter card is shown in Fig. 1. There are three main component on this card: The analog-to-digital converter (ADC) and digital-to-analog converter (DAC), up-convert and down-convert mixers, and a local oscillator (LO). The HSMC connector is used to connect the daughter card to the DE10 board, and the digital baseband signal generated from the FPGA is sent to the DAC through the HSMC connector. The analog baseband signal at the output of the DAC is mixed with the LO through the up-convert mixer and becomes a radio frequency (RF) signal. The RF signals are transmitted and received through SMA connectors. At the receiver side, the received RF signal is converted to a baseband signal by mixing with LO signal through the down-convert mixer. The baseband signal is then digitalized by the ADC and the resulting digital signal is transmitted to the FPGA on the DE10 board for further processing.

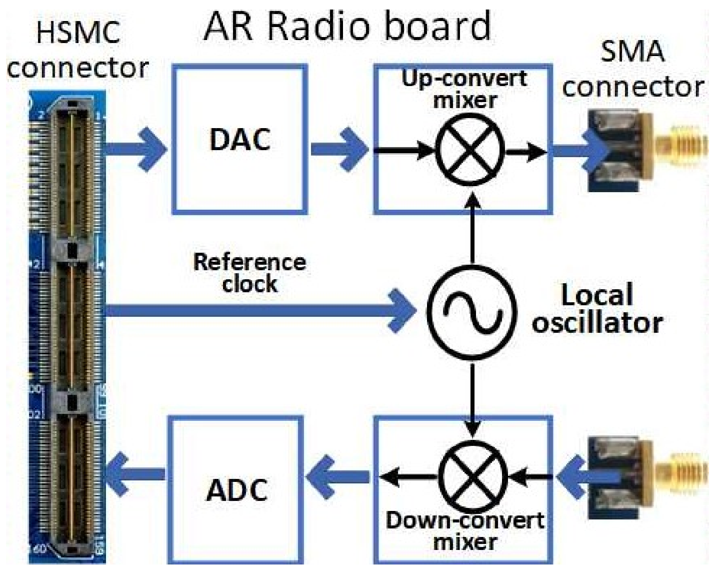


Figure 1: *Simplified block diagram of the ARRadio daughter card.*

2.2 DSP Builder

There are three common types of blocks in the DSP Builder Advanced blockset: Control blocks, Primitive blocks, and IP blocks. The control blocks are used configure how DSP Builder generates the hardware description and to set parameters such as what FPGA hardware device the design should be optimized for. They are also used to describe boundaries between different blocks. Primitive blocks are low-level functions, such as addition, sample delay or multiplication, which are used to construct more complex functional blocks.

The last block type describes IP blocks, provided by Intel. These blocks are dedicated to performing more complex tasks, such as different types of filtering, mixers, and oscillators. A typical IP block is shown in Fig. 2 and in this case it has three inputs, placed on the left side of the block, and three outputs, placed on the right. The `a` signal is the input data and `v` is a Boolean signal indicating that the data present at `a` is valid. The channel signal, `c`, indicates which channel the data at `a` currently represents. The `c` signal is an 8-bit unsigned integer (`uint(8)`) and has a function in multi-channel systems. In systems using only one channel, the `c` signal can be set to a constant `0`. An example of how these signals can switch are shown in Fig. 3, for a system where the data rate is half of the clock rate. The `q` output is the data output, while the `v` and `c` outputs are the output valid and channel signals, respectively. These signals are used as inputs to the next block. More information about the different blocks in DSP Builder and how they can be connected is available in [6].

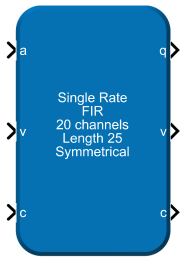


Figure 2: *A DSP Builder FIR filter IP block from the advanced blockset.*

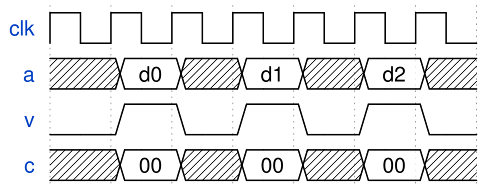


Figure 3: *Timing diagram of the input to an IP block with one input channel and a sample rate half of the clock rate.*

2.3 Quartus Prime

Quartus Prime is a suite of design tools, or an integrated development environment (IDE), used to create hardware descriptions, synthesize, simulate and upload bitstreams to Intel FPGAs. Quartus allows you to connect the system you design in Simulink/DSP Builder to other functional blocks, either IP or user-designed. During the lab session, you will use Quartus to connect your transmitter design to a block controlling the ARRadio board and synthesize the design. You will also use the Signal Tap logic analyzer to evaluate your design. The process of moving your design, either constructed in a tool like DSP Builder or written by hand in your favorite hardware description language, to a version that can be run on an FPGA, can be divided into three main steps: logic synthesis, place and route, and assembly. The process is sometimes called compilation, and the different steps are common to most FPGA vendors, even if the naming can differ slightly.

Logic Synthesis

The input to the first step of the compilation is your hardware description. During synthesis, this description is converted to a gate-level representation, describing components like logic gates, muxes, registers etc. and the connections between them. The synthesis process also contains an optimization step where the gate-level representation is minimized as much as possible, after which the design is customized for the target hardware by mapping the gate-level netlist to the LUTs, RAMs and other specialized hardware found in the FPGA fabric.

Place and Route

During the place-and-route stage, the functional blocks in the synthesized netlist are mapped to physical blocks in the FPGA: the placement step. The signals between the blocks are then routed, followed by an optimization stage, where the placement and routing are refined until, hopefully, the specified timing is met.

Assembly

The placed and routed design is used to generate a bitstream that can be used to program the FPGA. This step is called assembly by Intel, but other vendors call it Bitstream generation.

3 Lab

The following sections describe the exercises that are to be performed during this lab session.

3.1 Exercise 1 - DSP Builder Model

In this first exercise you will design a simple RF transmitter using a binary phase shift keying (BPSK) modulation format.

- Start DSP Builder using **DSP Builder - Start in MATLAB 2017b** under **Intel FPGA 18.0.0.614 Standard Edition** in the Windows start menu. Navigate to a folder where you would like your lab files to be placed, preferably on the `Z:` drive (if you are using the computers in the lab), so that you can access your files later. Create a folder entitled `lab1` and make sure it is the current folder. Start Simulink by executing the `simulink` command in the MATLAB **Command Window** or by clicking the Simulink button in the **Home** tab. Create a new model by selecting the **Fixed-step** template (you might have to press **Show more** under **Simulink**), as shown in Fig. 4.

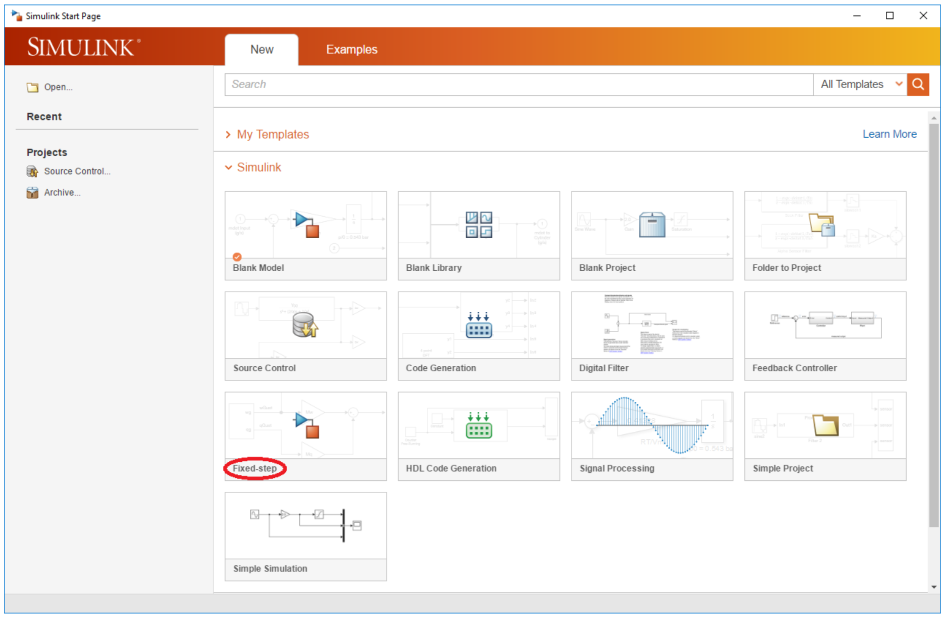


Figure 4: *Simulink Start Page.*

- An empty Simulink model should now be shown on the screen; save this file as `MCC150.slx`. You will soon fill this model with a basic Simulink description of a BPSK transmitter, but to be able to parameterize the Simulink model you will first need to create a MATLAB script in the original MATLAB window. Click **New Script** under the **Home** tab and save the new file as `MCC150_setup.m`. This file will later be filled with variables used to setup different model parameters. As a start, define the following variables:

```
clockRate  = 40e6;          % Clock rate of the DSP block
dataRate   = 1e6;           % Data rate of the DSP system
sampleRate = clockRate      % Sample rate of the DSP system
stepSize   = 1/sampleRate;  % Step size of the simulation
stopTime   = 1000*stepSize  % Stop time of the simulation
wordLength = 12;            % Word length of the signals

tx.N    = 1024;                      % Number of symbols to transmit
tx.data = randsrc(tx.N, 1, [0 1]);  % Generate PRBS
```

- Save the file and run it by pressing **F5** on the keyboard or the **Run** button the the MATLAB toolbar. This will store the variables and values in the MATLAB Workspace, which makes it possible for you to use them in Simulink.

- The setup script that you just created should be executed before each new simulation run in Simulink, to automatically update the workspace variables if you change the values. This can be accomplished by going back to the Simulink window and selecting **File** → **Model Properties** → **Model Properties**. Go to the **Callbacks** tab, enter `MCC150_setup` under **InitFcn** and click **OK**. To make the model use our variables for simulation control, go to **Simulation** → **Model Configuration Parameters** and set **Stop time** to `stopTime`, click **Solver details**, and set **Fixed-step size (fundamental sample time)** to `stepSize`. Click **OK** to save your simulation settings.

- You will now build a BPSK transmitter model, which can be used to generate a hardware description that can be synthesized and uploaded to an FPGA. Start by opening the library browser, shown in Fig. 5, (**View** → **Library Browser**) and navigate to **DSP Builder for Intel FPGAs - Advanced Blockset/Design Configurations**. Place a *Control* block in your design by dragging it from the **Library Manager** to the Simulink canvas. Open the settings for the new block by double clicking on it, change **Hardware destination directory** to `./rtl` and tick the **Generate hardware** box. Click the **Clock** tab; note that the clock frequency is given in MHz, change it to `clockRate/1e6`. You will use one of the push buttons on the DE10 development board as a reset, and these are driven low when pressed [1], thus you need to change the **Reset Active** option to Low and rename the reset signal to `areset_n` for clarity. Confirm that the settings match Fig. 6 and click **OK**.

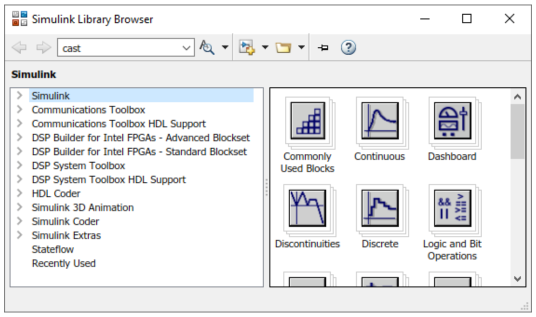
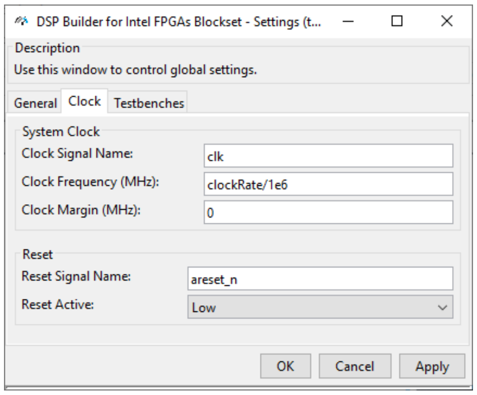


Figure 5: *Simulink Library Browser.*



DSP Builder for Intel FPGAs Blockset - Settings (t...

Description
Use this window to control global settings.

General | Clock | Testbenches

System Clock
Clock Signal Name: clk
Clock Frequency (MHz): clockRate/1e6
Clock Margin (MHz): 0

Reset
Reset Signal Name: areset_n
Reset Active: Low

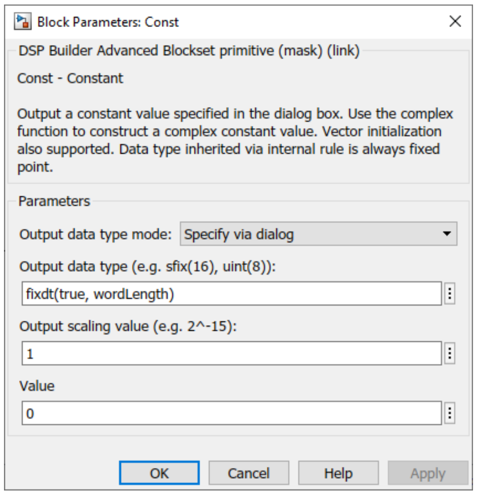
OK | Cancel | Apply

Figure 6: *Parameter settings for the Control block.*

- Add the *DisplayResources* block from **DSP Builder for Intel FPGAs - Advanced Blockset/Utilities/Analyze And Test/DisplayResources** in the **Simulink Library Browser**.

This block can later be used to check the amount of FPGA resources occupied by the current design.

- Add a *Subsystem* block from **Simulink/Ports & Subsystems** and name it `TransmitterBPSK` by double clicking on the name under the block. This subsystem is the portion of the Simulink model that will be placed on the FPGA.
- Double click on the *TransmitterBPSK* subsystem to edit it and add a *Device* block from **DSP Builder for Intel FPGAs - Advanced Blockset/Design Configurations**. This block indicates to DSP Builder that the subsystem, which it is placed inside of, is the top level design module in our FPGA design. In the settings for the *Device* block, change **Device Family** to `Cyclone V` and **Family member** to `5CSXFC6D6F31C6`, since this is the FPGA used on the DE10 development board. Save the settings by clicking **OK**.
- Our transmitter design will need two outputs and no inputs, thus delete the input (the left circled 1) by right clicking it and selecting **Delete**. The wire connecting the input to the output should now have turned red and you will have to delete it the same way. Add a second output from the **Simulink/Ports & Subsystems** and name the two outputs `I_out` and `Q_out`.
- In this simple transmitter you will only use the I signal to transmit data, and keep the Q signal at a constant 0. Add a *Const* block from the **DSP Builder for Intel FPGAs - Advanced Blockset/Primitives/Primitive Basic Blocks** library and connect it to the *Q_out* port by clicking close to the visible terminal of the *Const* block and dragging the wire to the terminal of the *Q_out* port. You will also need to specify the data type and value of the *Const* block, thus open the settings for this block and set **Output data type mode** to **Specify via dialog**. The **Output data type** should be set to `fixdt(true, wordLength)`, where the first argument to the `fixdt` command indicates that the data type is signed and the second set the number of bits used to store the value, i.e. the word length (that you previously stored in the `wordLength` variable in the MATLAB workspace). The **Output scaling value** should be set to 1 and the **Value** to 0. Verify that the settings match Fig. 7 and click **OK**.


Block Parameters: Const

DSP Builder Advanced Blockset primitive (mask) (link)

Const - Constant

Output a constant value specified in the dialog box. Use the complex function to construct a complex constant value. Vector initialization also supported. Data type inherited via internal rule is always fixed point.

Parameters

Output data type mode: Specify via dialog

Output data type (e.g. sfix(16), uint(8)): fixdt(true, wordLength)

Output scaling value (e.g. 2^-15): 1

Value: 0

Figure 7: *Parameter settings for the Const block.*

- You will now create a memory to store the pseudorandom bit sequence (PRBS) used as data for your transmission. Start by adding a new subsystem inside the *TransmitterBPSK* subsystem and name it `PRBSGenerator`. Enter the subsystem by double clicking it and remove the input port and the wire. Add two new output ports and name the first one `PRBS`, the second `qv` and the last one `qc`. Next, add a *ChannelOut* block from the **DSP Builder for Intel FPGAs - Advanced Blockset/Primitives/Primitive Configuration** library and connect its `q0` output to the `PRBS` port. Connect the other ouputs from the *ChannelOut* block to the ports with matching names.

- Since you will be using a single channel in your system, add a *Const* block from the **DSP Builder for Intel FPGAs - Advanced Blockset/Primitives/Primitive Basic Blocks** library, set its **Output data type** to `uint(8)`, **Output scaling value** to 1 and **Value** to 0 and connect the output of the *Const* block to the `dc` input of the *ChannelOut* block.

- You will now generate the valid signal for your output, which will be high for one clock cycle every `clockRate/dataRate` clock cycle and low for all other clock cycles. Add a *Sequence* block from the **DSP Builder for Intel FPGAs - Advanced Blockset/Primitives/Primitive Basic Blocks** library and connect a Boolean constant with the value `true` to its input. This input works as an enable for the *Sequence* block. Open the settings for the *Sequence* block, set the **Sequence setup** to `[0 clockRate/dataRate-1 clockRate/dataRate]` and click **OK**. Connect the output of the *Sequence* block to the `dv` input of the *ChannelOut* block.

- Add a *DualMem* from the **Primitive Basic Blocks** library to the design and open its settings dialog. This memory will contain the PRBS sequence that you generate in the MATLAB setup script (`tx.data`). The data is just one bit per address, so set the **Output data type** to `fixdt(false, 1)`, for a one bit, unsigned number representation. Set the **Output scaling value** to 1 and **Initial contents** to `tx.data` and click **OK**. The memory is a dual port type, but you will only use the first port in this lab and the content will not be updated after synthesis. Therefore, connect a Boolean constant with the value `false` to the write enable input (`w1`) and two zero constants to the data input (`d1`) and the second address bus (`a2`). The **Output data type mode** of the last two constants can be set to **Inherit via internal rule**, to let DSP Builder detect a suitable data type.

- The memory can be addressed using a counter that counts the number of high pulses of the valid signal generated by the *Sequence*. Add a *Counter* from the **Primitive Basic Blocks** library, connect its input to the output of the *Sequence*, and change **Counter setup** in the *Counter* settings to `[0 tx.N 1]`. Add a *SampleDelay* from the **Primitive Basic Blocks** library and connect its input to the output of the *Counter* and its output to the `a1` port of the *DualMem*. The *SampleDelay* is necessary to introduce a one cycle delay in the addressing of the memory, otherwise the first value of the symbol sequence would be skipped when the system starts.

- Finally you need to add a *SynthesisInfo* block from the **Primitive Configuration** library to your *DataGeneration* subsystem to make it synthesizable. The final subsystem should look similar to Fig. 8.

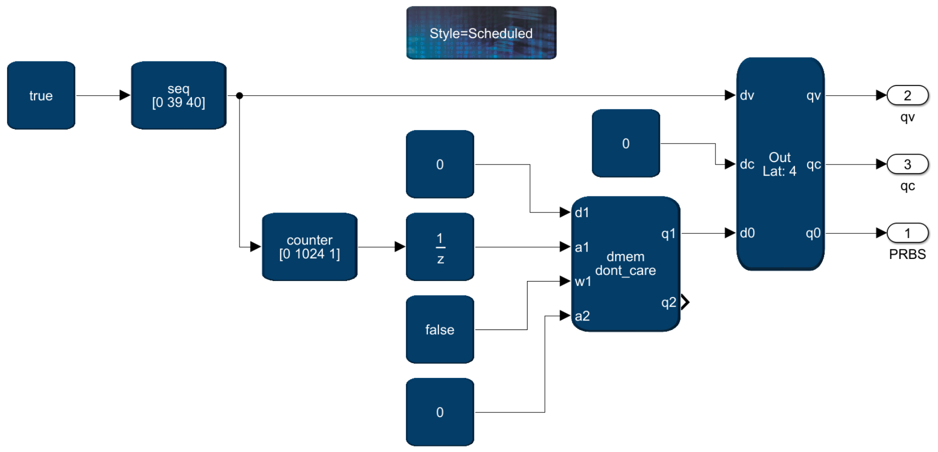


Figure 8: *View of the PRBSGeneration subsystem.*

- Go back up to the top-level *TransmitterBPSK* subsystem by pressing the **Up to Parent** button (light-blue up-arrow) in the toolbar. The output of of the PRBS is a one bit binary value, and you will now create a subsystem used to modulate that bit using a BPSK format. Add a new subsystem inside *TransmitterBPSK* and name it `Modulator`. Enter the subsystem an add a *SynthesisInfo*, a *ChannelIn* and a *ChannelOut* block. Add two additional input ports and name them `Data`, `dv` and `dc`. Add two new output ports and the outputs `I`, `qv` and `qc`. Connect the ports to the corresponding ports on the *ChannelIn* and *ChannelOut* block.

- Add a *Select* block from the **DSP Builder for Intel FPGAs - Advanced Blockset/ Primitives/Primitive Basic Blocks** library, open its **Block Parameters** and set **Number of cases** to `1`. Take the opportunity to read the block description in the dialog box and make sure that you understand how the block works. Press **OK** when finished.

- Connect the `q0` output of the *ChannelIn* block to the `0` input of the *Select* block to use the input data as a select signal. Add two *Const* blocks and connect them to the `d` and `a` input of the *Select* block. In a typical BPSK modulation format, a data input of 0 should result in `I` being positive, and a `1` in `I` being negative. To use the complete range of our available 12 bits, set the constant connected to `d` to `2^(wordLength-1)-1`, which is the largest positive value we can fit in our word length when using a signed representation. Correspondingly, set the constant connected to `a` to `-(2^(wordLength-1)-1)`. For both *Const* blocks, the **Output data type** should be `fixdt(true, wordLength)` and **Output scaling value** should be `1`. When completed, the *Modulator* subsystem should look similar to Fig. 9

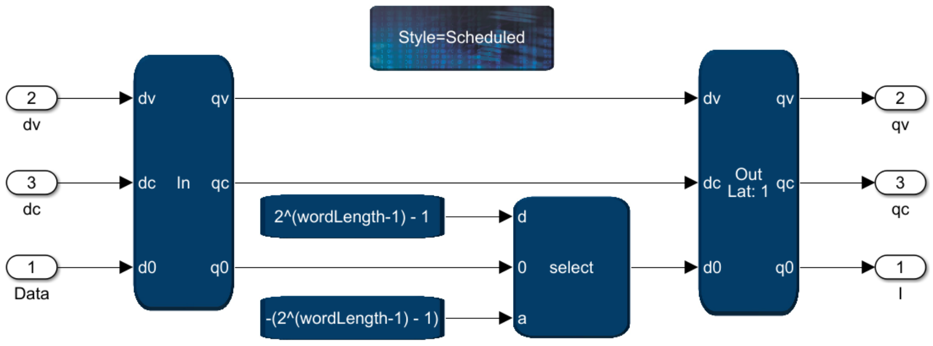


Figure 9: *View of the Modulator subsystem.*

- Go back up to the top-level *TransmitterBPSK* subsystem and add two *Scopes* from the **Simulink/ Sinks** library to the design and change the number of input ports of both of them to three, by right-clicking on each *Scope* and selecting **Signals & Ports** → **Number of Input Ports** → **3**. Open the *Scopes* and change the layout of the waveform window by selecting **View** → **Layout...** and choosing a suitable version, e.g. 3 × 1. Finally, connect all blocks and ports together as shown in Fig. 10.

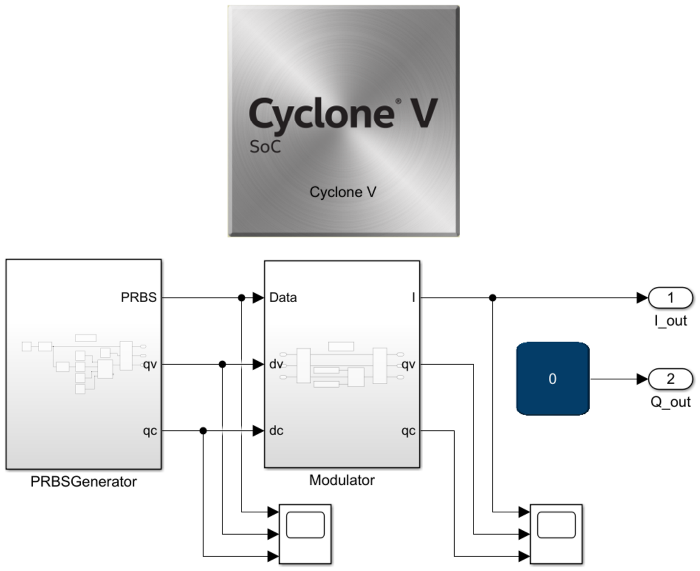


Figure 10: *View of the TransmitterBPSK subsystem.*

- You are now ready to start a simulation of the system by using **Simulation** → **Run** or the green **Run** button in the toolbar. If you encounter any errors at this stage, check the error messages and use them to figure out where to start debugging. Study the waveforms in the *Scopes* and make sure that they match what you expect by comparing them to the values of `tx.data` in MATLAB. You can see at first 20 values of `tx.data` but executing the command `tx.data(1:20)` in the MATLAB **Command Window**. A hardware description, in VHDL, of your design is generated when you run the simulation, and placed in the `rtl` directory in your project folder.

- Verify with your TA that the output from your modulator is correct.

Reflection Questions

- Why do we want to use the full range of `wordLength` as the output of the modulator?
- What would happen to the resource utilization in the *Resource Overview* if we set the design to use a wordlength of 4 bits instead?

3.2 Exercise 2 - Channel Model

You will now create a Simulink model of a simplified transmission system, emulating a phase offset and additive white Gaussian noise (AWGN). Note that the blocks that you will use in this exercise are Simulink blocks and not DSP Builder blocks, thus they are used only for simulation and not for hardware generation.

- Go back up to the top of the *MCC150* Simulink project and add a *Real-Imag to Complex* block from the **Simulink/Math Operations** library. Connect the output signals from the *TransmitterBPSK* to the inputs of this block. The *Real-Imag to Complex* block will convert our I/Q representation to a complex representation as $Z = I + iQ$, which makes the following steps easier.

- Add a *Data Type Conversion* block from the **Simulink/Signal Attributes** library and set its **Output Data Type** to double. Since you want to emulate a wireless analog signal, the fixed point representation is abandoned for the simulation part.

- You will now build the phase offset emulation, by multiplying each transmitted symbol with a complex number of unit length and a randomly generated phase offset. Add the following line to your `MCC150_setup.m` file:

  ```
  chan.phase = pi/2*rand() - pi/4;
  ```

  This operation will create a random phase between $-\pi/4$ and $\pi/4$. Run the script after adding the new line by pressing **F5**. In your *MCC150* Simulink model, add a *Constant* block from the **Simulink/Sources** library and set its **Constant Value** to `exp(1i*chan.phase)`. Add a *Product* block from the **Simulink/Math Operations** library and connect the *Constant* to one of the inputs and the *Data Type Conversion* block to the other.

- To emulate the AWGN, you will use an *AWGN Channel* from the **Communications Toolbox/Channels** library. Add the following line to your `MCC150_setup.m` file to define a variable to store the signal-to-noise ratio (SNR):

  ```
  chan.snr = 20;
  ```

  Add a *AWGN Channel* block to your *MCC150* Simulink model and set its **Mode** to **Signal to noise ratio (SNR)**. Set the **SNR (dB)** to `chan.snr` and the **Input Signal Power** to `(2^(wordLength-1)-1)^2`. If you view the signal amplitud of $2^{\text{wordLength}-1} - 1$ as a voltage, the expression comes from the power this voltage would dissipate in a 1 Ω load. Connect the *AWGN Channel* to the output of the *Product* block.

- A real communication system also has an inherent delay of the transmitted signals. Store the delay setting in a variable in your `MCC150_setup.m` script as `chan.dly = 26;` and run the script. Add a *Delay* block from the **Simulink/Discrete** library and set its **Delay length** to `chan.dly`. Connect the *Delay* block to the output of the *AWGN Channel*.

- To be able to view the channel output in a *Scope*, add a *Complex to Real-Imag* block from the **Simulink/Math Operations** library and connect it to the output of the *Delay* block. Connect a 2-channel *Scope* to the output of the *Complex to Real-Imag* block, and second 2-channel *Scope* to the the outputs of the *TransmitterBPSK* block.

- You will need to save both the input and the output of your channel model for later processing when you work through your home assignment. This can be achieved by adding two *To Workspace* blocks from the **Simulink/Sinks** library to your design. Connect one of them to the output of the *Data Type Conversion* and the other to output of the *AWGN Channel*. Set the **Variable Name** of the first one to `txSave` and the second to `rxSave`. When finished your design should look similar to Fig. 11. This figure also shows the data types of all signals, an option that you can enable by selecting **Display** → **Signals & Ports** → **Port Data Type**.

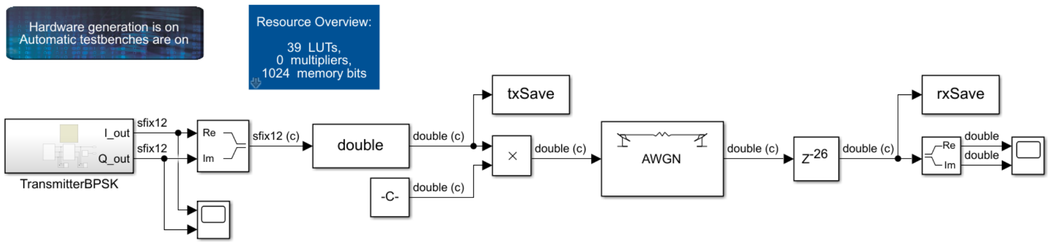


Figure 11: *View of the MCC150 Simulink model.*

- Run the simulation and compare the signals shown in the two *Scopes*. Save your output data by executing `save('results1')` in the MATLAB **Command Window**. This creates a file storing all your MATLAB variables, including `txSave` and `rxSave`, and these can be loaded later by executing `load('results1')`. Repeat the simulation one more time and save the variables using `save('results2')`.

- Verify with your TA that the output from the channel model is correct.

Reflection Questions

- In the channel model, you added a phase offset to the signal. What can be the source of this impairment in a real communication system?

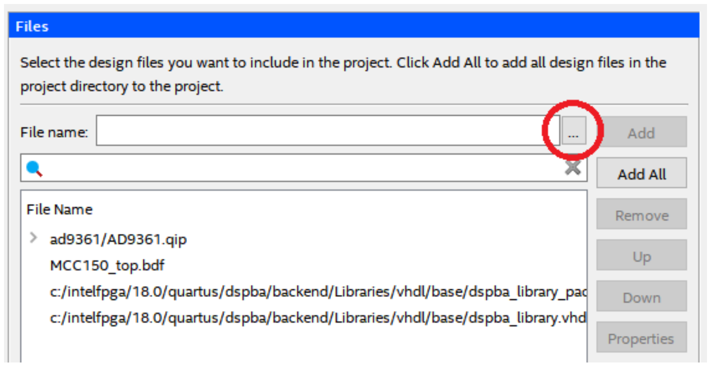
- Decrease the SNR to 10 dB and study the output. What happened? In a real communication system, what factors can affect the SNR?

3.3 Exercise 3 - Synthesis in Quartus Prime

In this exercise, you will insert the hardware description of your transmitter, generated using DSP Builder, into a Quartus project to enable hardware synthesis. A project template is provided on Canvas, containing a module controlling the ARRadio daughter card, a set of timing constraints suitable for the FPGA hardware and pre-made pin planning.

- Download the compressed Quartus Prime project from Canvas and extract the content in your `lab1` folder, so that the `MCC150_top.qpf` file is located in the same directory as your Simulink project file (`MCC150.slx`). Open the project by double clicking the main project file, `MCC150_top.qpf`. You will now import the hardware descriptions generated in the previous assignment into the project.

- Select **Project** → **Add/Remove Files in Project** and click the button marked **...**, circled in red in Fig. 12. Navigate to the `rtl/MCC150` folder and select the the `MCC150_TransmitterBPSK.qip` file, which contains pointers to all your generated VHDL-files. Press the **Open** button to import the files and press **OK** in the next dialog.



Files

Select the design files you want to include in the project. Click Add All to add all design files in the project directory to the project.

File name: [...] Add

Add All

File Name

- ad9361/AD9361.qip
- MCC150_top.bdf
- c:/intelfpga/18.0/quartus/dspba/backend/Libraries/vhdl/base/dspba_library_pac
- c:/intelfpga/18.0/quartus/dspba/backend/Libraries/vhdl/base/dspba_library.vhd

Remove · Up · Down · Properties

Figure 12: *Add/Remove Files in Project dialog.*

- You will use the schematic editor to connect your DSP Builder project to the AD9361 controller module provided in the Quartus project, but in order to do that you must create a schematic symbol for your design. Start by changing the view in the top-left corner of the **Project Navigator**, from **Hierarchy** to **Files**. You should now see a list of all the files used in the Quartus project. Expand `rtl/MCC150/MCC150_TransmitterBPSK.qip` file, right click on the `MCC150_TransmitterBPSK.vhd` and select **Create Symbol Files for Current File**.

- Switch back to the **Hierarchy** view in the **Project Navigator** and double click on the top level design file, named `MCC150_top`. This should open a schematic view of the top level where you can see the input port on the left side and the output ports on the right side of the AD9361 controller module. Add your transmitter design to the schematic by clicking the **Symbol Tool** button, marked with a red circle in Fig. 13, select the `MCC150_TransmitterBPSK` symbol in the `Project` directory and click **OK**. Place the symbol in a suitable place in the schematic by left-clicking and stop placing using the **Esc** key. Connect the AD9361 controller and the transmitter by wires as shown in Fig. 13. The wires are placed by click-and-drag, in a similar way as in Simulink. Finalize the schematic by renaming the **MCC150_TransmitterBPSK** instance by double-clicking **inst** in the symbol. Use the name `MCC150_TransmitterBPSK_inst`. Save the schematic and close the schematic editor.

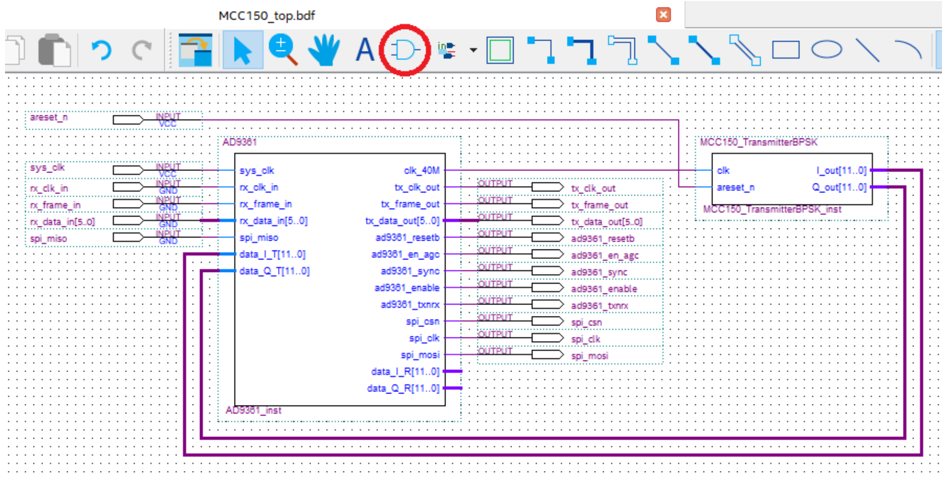


Figure 13: *The schematic editor with the Symbol Tool marked with a red circle*

- You are now ready to compile your design by doing **Processing** → **Start Compilation**. The compilation step can take some time to finish and if everything is working, no errors should be reported. When the compilation is finished, study the **Compilation Report**, which should now have appeared on your screen. In this view you can see how much of the FPGA resources your design utilizes, the timing results, and other design statistics. Look especially under **Analysis & Synthesis/Resource Utlization by Entity** to see how much resources your transmitter uses.

Reflection Questions

- Which parts of your design has the largest utilization?

3.4 Exercise 4 - Signal Tap

Signal Tap is a tool that can be very helpful when debugging your design once its uploaded to the FPGA. In this exercise, you will configure a Signal Tap instance, use it to study the output waveform and save the data for later analysis in MATLAB. If you are performing the lab at your home computer or using the Heffalump server, you will not be able to complete the items marked with a red bullet (●), since you don't have access to the DE10-Standard board. For now, just read through these steps so that you know how to perform them when you get remote access to a computer with the hardware connected.

- Start **Signal Tap Logic Analyzer** from the **Tools** menu and make sure that the **Setup** pane is activated in the bottom left. On the right side of the screen you should see the **Signal Configuration** pane, where you can define how to capture the FPGA signals. First we will define a clock signal but pressing the **...** button to the right of the **Clock** field, marked with a red circle in Fig. 14. We want to see the signal names before synthesis, since the synthesis process might have changed them, so select **Signal Tap: pre-synthesis** in the drop down menu and press the **List** button. Double-click on the `rx_clk_in`, which is a 80 MHz clock signal coming from the ARRadio board, to select it and press **OK**.

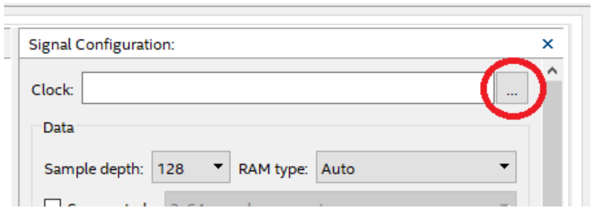


Figure 14: *Button used to define the clock signal in Signal Tap.*

- In the **Signal Configuration** pane, locate **Sample depth** and set it to 2K. This defines how many samples we will capture on the rising edge of the selected clock. The rest of the settings can be kept as they are.

- Next, we want to define what signals to capture. Double-click in the central pane to open the **Node Finder** and once again select **Signal Tap: pre-synthesis** in the drop-down menu. Press **List** and select the signals that you want to capture. Expand **AD9361:AD9361_inst** and add the transmitted IQ signals, `data_I_T` and `data_Q_T`, and the received IQ signals, `data_I_R` and `data_Q_R`. Also, add the output from the *PRBSGenerator* subsystem that you created in DSP Builder. These signals are located further down in the tree structure and prefixed with `out_`. Close the **Node Finder** window by first pressing **Insert** and then **Close**.

- You will now need to recompile the system to include the Signal Tap module on board the FPGA. Start the compilation by running **Processing→Compile**. Save your Signal Tap settings and click **Yes** if asked if you want to enable Signal Tap for your project.
- When the compilation process is finished, connect the DE10-Standard board, fitted with the ARRadio daugther card and two antennas, to the computer using an USB cable and turn it on. Restart Signal Tap, if the window was closed during compilation, and locate the **JTAG Chain Configuration** pane in the top-right corner. You will now upload the generated programming files (or bitstream) to the FPGA. Select **DE-SoC [USB-1]** as the **Hardware** and `5CSEBA6` as the **Device**. The DE-SoC is the communication cable used and the 5CSEBA6 is the name of the FPGA located in the IC. The chip also contains a dual-core ARM processor, which is the other available device.
- Press the **...** button circled in red in Fig. 15, select the `.sof` file and press **Open**. This is the file containing the design to be programmed onto the FPGA. Upload the file to FPGA by pressing the **Program Device** button circled in green in Fig. 15. Once the upload process is finished, your design should be up and running on the FPGA.

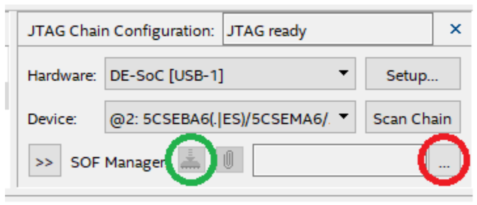
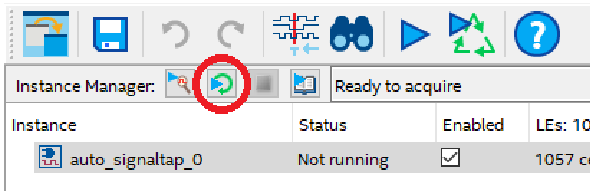

JTAG Chain Configuration: JTAG ready
Hardware: DE-SoC [USB-1]
Device: @2: 5CSEBA6(.|ES)/5CSEMA6/

Figure 15: *Buttons used to select programming file (red) and start programming (green).*

- When the **Instance Manager**, shown in Fig. 16, shows **Ready to acquire**, press the **Autorun Analysis** button, circled in Fig. 16, to begin acquiring data. To make sure that the system was initialized properly, press the reset button (**KEY0**) on the FPGA board to reset the transmitter.


Instance Manager: Ready to acquire

Instance	Status	Enabled	LEs: 10
auto_signaltap_0	Not running	✓	1057 c

Figure 16: *Signal Tap when ready to acquire data from the FPGA*

- To view a waveform representation of the data sent to and from the ARRadio board, right-click on these signals and select **Bus Display Format→Signed Line**. Verify that the IQ signals match what you expect by comparing to the output of the memory. If they don't, you will need to go back and debug your design. The acquisition can be stopped by pressing the **Stop Analysis** button, or by pressing **Esc**.
- The data that is currently shown in the **Signal Tap Logic Analyzer** window can be saved in a file as comma-separated values (.csv), which can be processed later in e.g. MATLAB. Perform measurement run and save the results in a .csv-file.

4 Post-Lab Home Assignment

The following tasks should be performed before the next lab session, and a lab report should be handed in on Canvas before 23:59 on Sunday, April 18. There is a strict 4-page limit on the report, and it should contain descriptions of the blocks designed in DSP Builder and their functionality, answers to the reflection questions at the end of Sections 3.1–3.3 in the lab instructions, and solutions to the home-assignment questions below, including the resulting figures. You are encouraged to use figures and screenshots to support your explanations and answers. When handing in your report, you will need submit both a `.pdf` version of your report and a `.zip` archive containing your DSP Builder and Quartus projects.

Home Assignment Tasks

- Load the results from Section 3.2 into Matlab and plot the I and Q signals of the transmitted (`txSave`) and received signals (`rxSave`) as a function of time. Describe and explain the difference between two different simulation runs. Note that the actual data points are available in `txSave.Data` and `rxSave.Data`.
- Estimate the phase offset for the received signals for both simulation runs. Compare your results with the setting in the `chan.phase` variable.
- Plot an eye diagram of the received data for one of the simulation runs, see the MATLAB documentation for `eyediagram`. Explain how you select the parameters used when plotting the diagram.
- Downsample the received I and Q data to one sample per symbol.
- Plot the constellation diagram of the transmitted, and the received and downsampled I and Q data, see MATLAB documentation for `scatterplot`.

References

[1] Terasic Inc., *DE10-Standard User Manual*, 2018.

[2] ——, *DE10-Standard Schematics*, 2017.

[3] ——, *ARRadio User Manual*. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/arradio

[4] ——, *ARRadio Schematics*, 2013.

[5] Analog Devices, *AD9361 Reference Manual*, 2015.

[6] Intel Corporation, *DSP Builder for Intel FPGAs (Advanced Blockset) Handbook*, 2018.