The DSP Primer

Presented by:

Bob Stewart
University of Strathclyde, Scotland, UK
r.stewart@eee.strath.ac.uk

Steve Alexander
University of Strathclyde, Scotland, UK
salexander@eee.strath.ac.uk

Jeff Weintraub
Xilinx Inc, San Jose, USA
jeff.weintraub@xilinx.com

Presented as part of Xilinx University Program (XUP)

August 2005, Version 0.97

Xilinx DSP Primer WorkBook Contents

1 Introduction

In this DSP for FPGAs Primer workbook the aim is:

- To allow all users experience of using the entire toolchain from DSP algorithm concept to FPGA implementation.

After completing this workbook you will be able to:

- Understand fundamental DSP algorithms for FPGA implementation;
- Be aware of the FPGA hardware for implementation of DSP algorithms;
- Know how to use Simulink and SystemGenerator for the simulation of DSP algorithms, architectures and systems
- Know how to correctly design a DSP system with knowledge of issues relating to wordlengths, overflow, saturation, wraparound and so on.
- Be able to take a design from Simulink System Generator implementation to Xilinx ISE tools.
- Know how to use Xilinx ISE tools to synthesize and place and route the design.
- Know how to use the Xilinx FPGA editor to inspect the actual hardware implementation with respect to on-chip hardware.
- Know how to perform hardware-in-the-loop simulation.
- Know how to run DSP algorithms on the XUP Virtex 2 Pro board.

1.1 Software Required

The following software is required to complete the various examples in this workbook:

1. MatLab Release 14, Simulink 6
2. Xilinx System Generator v7.1
3. Xilinx ISE Tools v7.1 + ServicePack 3 + IP update;
4. Xilinx Chipscope v7.1

If any of these are missing please contact the appropriate vendor for install files and a licence.

We will be using Xilinx ISE tools for synthesis and HDL simulation. It is of course possible to use other synthesis tools (Leonardo, Synplicity) to do these stages.

1.2 Example Files and Directories

The examples for this workbook should be copied to the location:

```
c:\Xilinx_DSP_Primer
```

As a short cut notation we will use the Xilinx symbol to specify the directory `c:\Xilinx_DSP_Primer`

You will note that all top level Simulink models (with the.mdl suffix) are usually located in a directory with the same name. For example the Simulink model `shortfilter.mdl` would be found in directory:

```
\shortfilter\shortfilter.mdl
```

or

```
c:\Xilinx_DSP_Primer\shortfilter\shortfilter.mdl
```

The reason for placing the examples in their own named directory is to ensure that if you decide to set up a Xilinx ISE project file, then there is no interaction or mixing of different ISE project files which could happen if the ISE tools were used on two or more different `.mdl` files in the same directory.

1.3 Shorthand for Mouse Clicks

Shorthand symbols for mouse clicks will be used in this workbook according to the table below:

Figure 1.1: Mouse and keyboard click notation used in this document.

2 Top-Level View of the Design Flow

In this section we take some very simple designs through the entire design flow from simulation to bitstream form ready for implementation on an FPGA.

2.1 Virtex2 XC2V40 Target FPGA

For these first exercises we will use a small device, the Xilinx Virtex2 XC2V40 device and go from simple DSP implementation to actual floorplanning on the device.

We have chosen this device in order to be able to easily inspect the DSP implementation on the FPGA. (Later in the workbook we will be working with the larger XCVP30 device which is available on the actual board use in this course.) Some of the key features of this device are given in the table below, next to a comparison of some of the other parts from the Xilinx family.

Device	Family Name	CLB Array	Number of Slices	Logic Cells	CLB FlipFlops	Block RAM (bits)	Max I/O	18 x 18 Multipliers	No of DCMs	DCM Freq (min\max) MHz
XC3S200	Spartan	24 x 20	1920	4320	3840	216k	173	12	4	25/326
XC3S5000	Spartan	104 x 80	33280	74480	66560	1872k	784	104	4	25/326
XC2V40	Virtex II	8 x 8	256	576	512	72	88	4	4	24/420
XC2V8000	Virtex II	112 x 104	46592	104832	93184	3024	1108	168	12	24/420
XCVP2	Virtex II Pro	16 x 22	1408	3168	2816	216	204	12	4	24/420
XCVP30	Virtex II Pro	80 x 46	13696	30816	27392	2448	644	136	8	24/420
XCVP100	Virtex II Pro	120 x 94	44096	99216	88192	7792	1164	444	12	24/420

2.2 Running a SystemGenerator Simulation

In this section we will open MatLab and Simulink. For those unfamiliar with Simulink a short on-screen review will also be progressed.

This section has a series of **Actions**. On completion of each action the DSP to FPGA design has proceeded one simple step. At the end of the section you should have an understanding of the main points of the design flow. If not - go back and repeat!

Exercise 2.1 Running a DSP Simulation

Action 1: ***OPENING MATLAB .*** *Open MatLab by choosing:*

MatLab will now open and you will see the general workspace (or a slight

variation thereof):

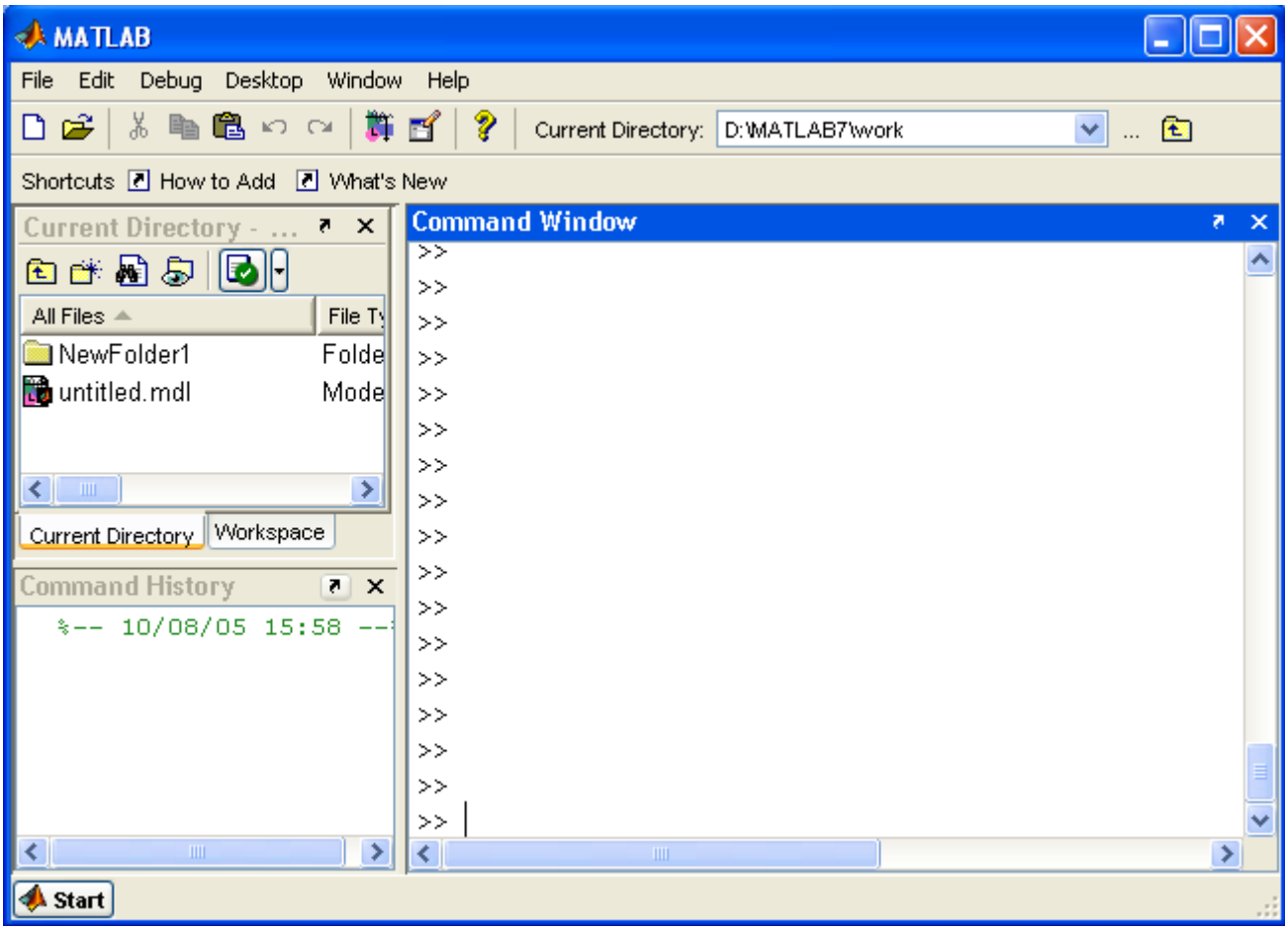
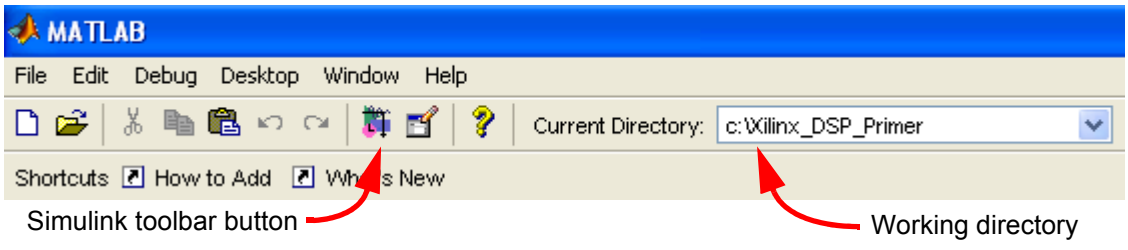


Figure 2.1: MatLab Window

Clearly being skilled in MatLab will be of benefit to this course, however in order to make progress on the key topics of DSP to FPGAs, we will only introduce essential MatLab skills.

Action 2: ***SETTING THE WORKING DIRECTORY.*** *The pre-prepared examples in this workbook have been installed to the directory* **`c:\Xilinx_DSP_Primer`**. *Therefore in order to make opening files easier, this should be set to the working directory. Therefore set by typing in the folder name to the MatLab console as below:*



Action 3: ***STARING SIMULINK.*** *Start Simulink by choosing the Simulink toolbar button*  *highlighed in above.*

You should now see the Simulink Library Browser Window open up:

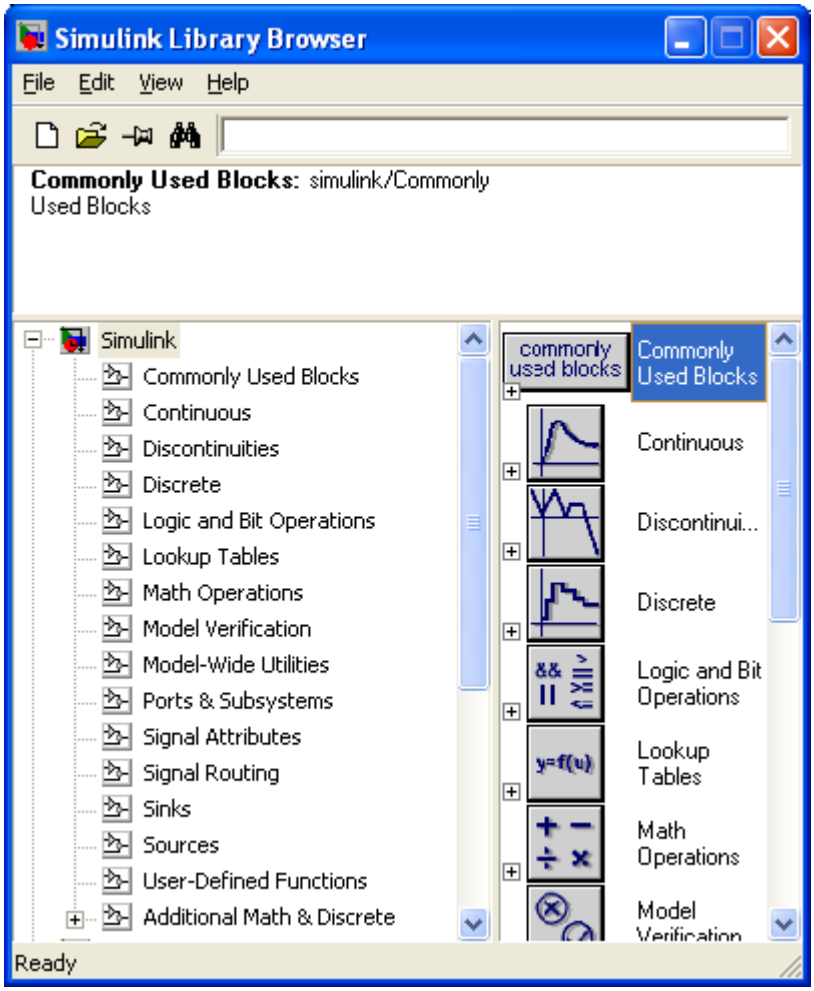


Figure 2.2: Simulink Library Browser

Action 4: ***OPENING A SIMULINK SYSTEM.*** *Now for the first system we will open a pre-exisiting design.*

In the Simulink Library Browser open the system (using the menu option **`File > Open`***)*

`\intro\delay\delay.mdl`

and on the screen you should see the first simple system:

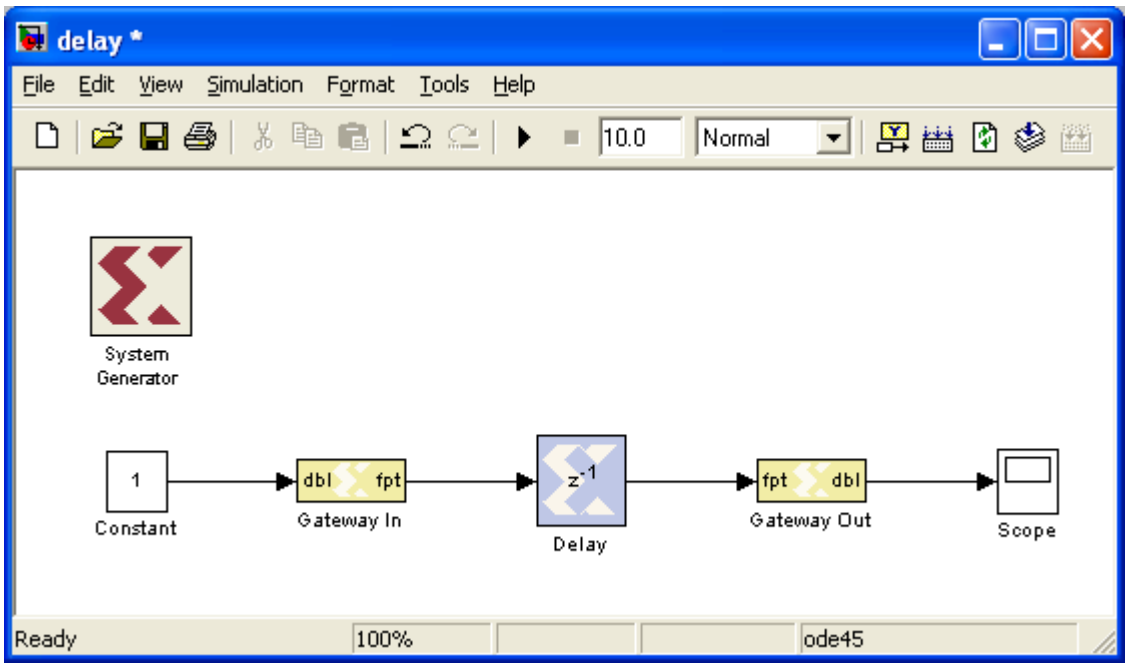


Figure 2.3: First Simulink example of a delay!

This system takes a unit step input and converts from floating point to 8 bit fixed point, then delays for one sample. The output is then converted back from fixed point to floating point and displayed on a scope display.

Action 5: ***THE GATEWAY PARAMETERS.*** *Using the mouse on the gateway-in block* dbl fpt *and view the parameters. In the table below fill in the parameters for this block:*

Parameter	Value
Output Data Type	
No. of bits	
Binary Point	
Quantisation	
Overflow	
Sample Period	

Action 6: ***RUNNING A SIMULATION.*** *We will now run the system by choosing the Simulink run toolbar option* ▶ *or choose* **`Simulation > Start`** *from the menu.*

After the system has been run, on the Scope and this will give a visual of the

output of the simulation:

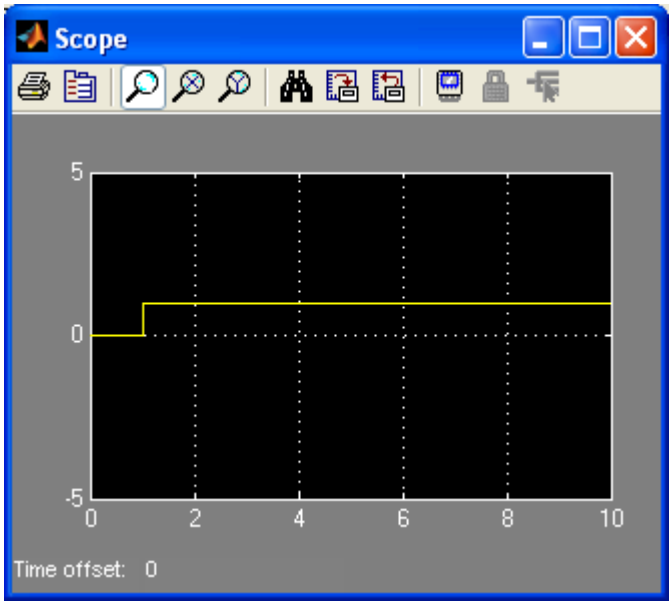


Figure 2.4: Simulink scope output.

Action 7: ***ZOOMING IN AND OUT OF THE SCOPE.*** *You can zoom in and out of the scope window to see a signal in more detail.*

on the zoom toolbar button on the scope and then in the time window to select an area; on release the window will zoom to that area.

on the to autoscale (rescale) the window

Now select the zoom x-axis toolbar button. Then in the scope window and draw a straight horizontal line; on release the window will zoom to this x-axis.

Now select the zoom y-axis toolbar button. Then in the scope window and draw a straight vertical line; on release the window will zoom to that y-axis.

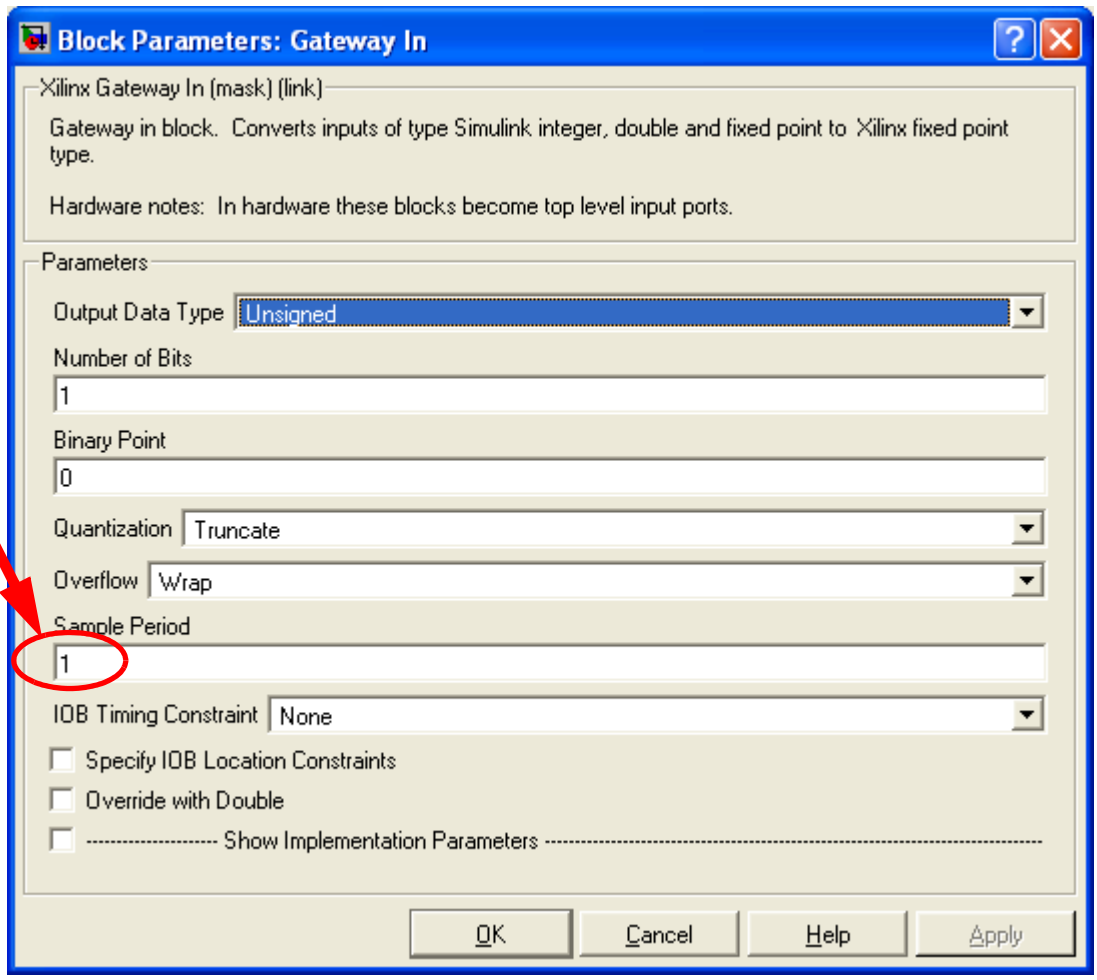
on the to autoscale (rescale) the window and move to the next action.

Action 8: ***SETTING THE SAMPLING RATE.*** *To interpret what is happening in this simple simulation we need to understand a few of the Simulink parameters.*

First note the visible parameters below the Simulink toolbar buttons:

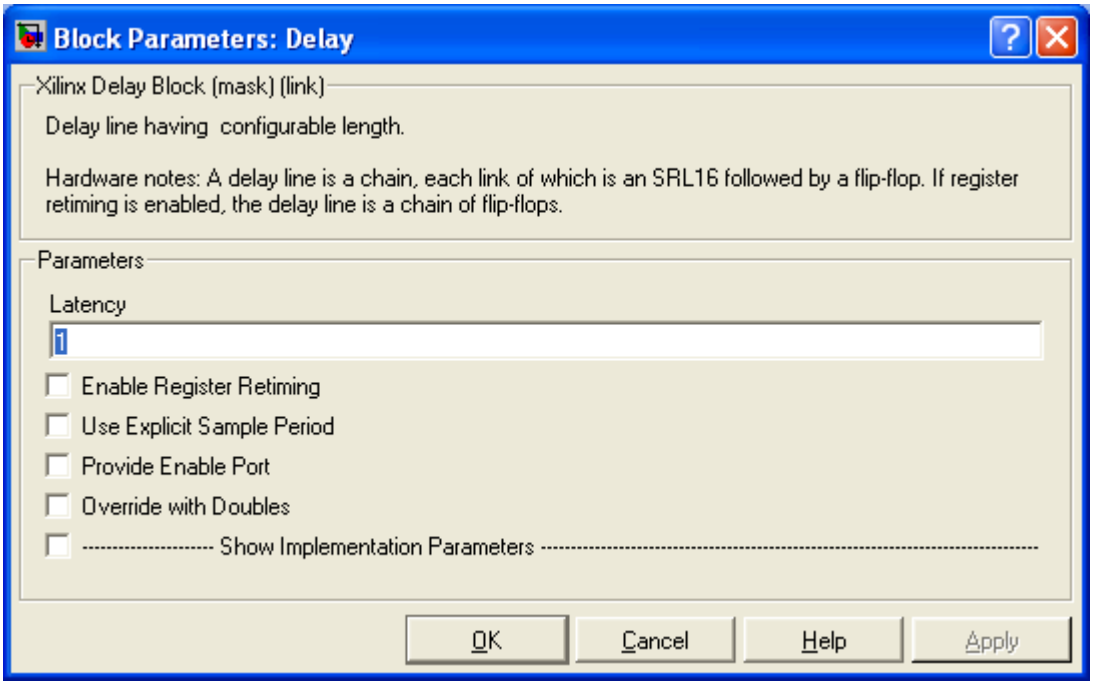
The "10" indicates that the system will run for 10 seconds.

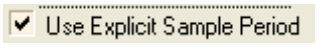
The sample period of this simulation was set to $f_s = 1$. *This can be seen by* 2 *on the gateway in block* dbl → fpt *and noting the sample period is set to 1 (you can also see the sampling rate in the System Generator block - more on this later):*



Subsequent blocks such as the delay z^{-1} *block have the sampling period set to "-1". In the world of Simulink this means that the block will inherit the sampling rate from the previous block.*

Action 9: ***CHANGING THE PARAMETERS OF A BLOCK.*** *View the delay block parameters by* 2 *on the delay block* z^{-1} *:*

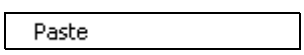


The only parameter is the latency or delay of the block set to 1. Tick the checkbox for  *and observe the information that the sample period is set to 1. If this is set to "-1" then in Simulink language this means that it inherits the sampling rate from previous blocks.*

Note there is NO parameter for the number of bits in this delay. This is because the number of bits in this delay block is a function of the input to the block and therefore is infered from the previous block (the gateway in). In our example the gateway has 8 bits resolution and therefore this delay block has 8 bits.

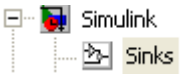
Action 10: ***ADDING ANOTHER BLOCK.*** *add in another Scope*  *to the delay example in order that you can view input. This can be done in two ways.*

First by placing the mouse on top of the Scope  *and then selecting the option to "copy":*

then in a empty space in the model window choose  *option and a new Scope*  *will appear.*

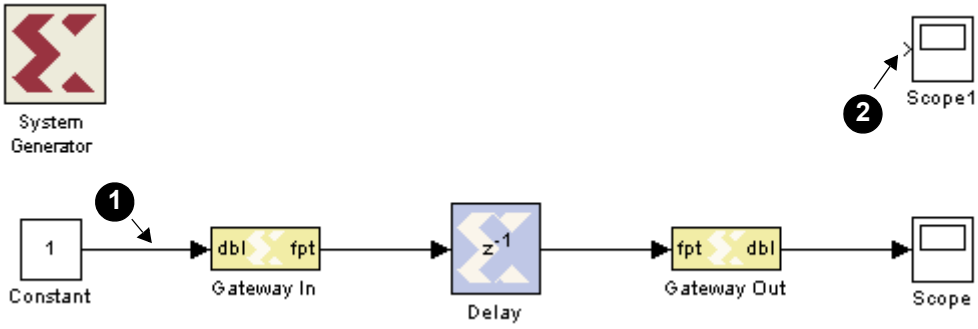
Alternatively you can get a new Scope block from the Simulink library browser

(see Figure 2.2 on page 9) and select from the option:



and select Scope  *and*  *the scope into the Simulink workspace and drop where you want to place the scope.*

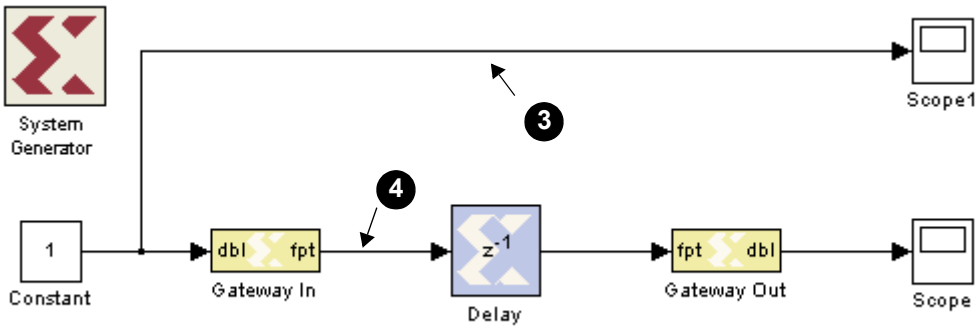
Action 11: ***CONNECTING BLOCKS TOGETHER.*** *You should now have a system with an additional scope similar to below:*



If you want to move any block then  *on the block and drag and drop to where you want to put it.*

To connect the new scope block place the mouse on top of the actual "wire" connection and  *at position* ❶ *and observe the mouse pointer change to crosshairs* + *; then (still holding the right mouse button down) drag to connect to the > at position* ❷ *- when you are at the connection point observe that the cross-hairs become double* ++ *now let go of the mouse button.*

You should now have successfully connected the constant to the scope as below!



Action 12: ***DISCONNECTING BLOCKS.*** *To disconnect the connection just made, highlight the connecting line of interest at position* ❸ *by*  *and note that the line is now selected. You can then either hit BACKSPACE on the keyboard, or*  *and select* Cut.

Reconnecting the line using the procedure from the previous Action.

Action 13: ***RECONNECTING BLOCKS.*** *Disconnect the two blocks at position* ❹ *by deleting the wire. Note that to connect two blocks (i.e. not tapping off an exisiting connection as in Action 11) you simply select the "from" block ">" (on the right side) and then drag and hold* H *to connect to the ">" on the "destination" block.*

Before proceeding ensure all disconnections are remade and continue.

Action 14: ***RE-RUNNING THE SYSTEM.*** *Run the system with the added scope. Open both scopes by* 2 *on each scope. Observe that the output is in fact a delayed version of the input with a single sample delay, and a sampling rate of 1Hz.*

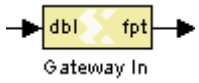
Exercise 2.2 Modifying Time Parameters of a Simulink Simulation

Open the system:

\intro\delay2\delay2.mdl

(This is the same system that would be generated by completing all of the *Action 1* to *Action 14* on the delay.mdl file.)

(a) Run the system and confirm a sampling rate of 1Hz, and a single 8 bit sample delay.

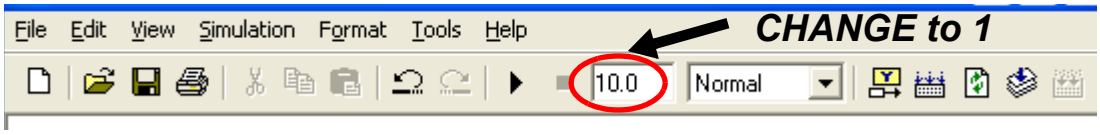
(b) Change the sample rate of the system to 100Hz by modifying the sample period of the input gateway block to 0.01. Do this by 2 on the 

and changing the Sample Period to 0.01 seconds (or type 1/100).

Because this is a System Generator system we must ***also*** update the System Generator block with this sample rate. Therefore 2 on the System Generator block:

and set the Simulink System Period (sec): also to 0.01 (or 1/100). (Note if you fail to do this Simulink will detect a discrepancy and then offer to do this for you when you run the system.)

(c) Now change the simulation time to 1 second (i.e. 100 samples) by changing the 10 parameter in the toolbar to 1



(d) Run the system and then observe the two scopes. You should see 100 samples, and still a delay of 1 sample. The time axis on the scopes should now reflect that the simulation was performed at a sampling rate of $f_s = 100$.

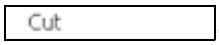
Exercise 2.3 Modifying Block Functionality in a Simulink Simulation

Open the system:

\intro\delay3\delay3.mdl

This system has sampling rate set to $f_s = 10000Hz$ and will run for 0.02 seconds or 200 samples. In this example we will change the input signal to a sine wave and then change the parameters of the delay block.

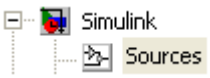
(a) First delete the constant source 

by  on the block and selecting 

. Note that the now unconnected wires remain but are highlighted red and are dotted indicating they are free-floating and not connected.

(b) Goto the 

browser window and find the sources and view the sources set of blocks.



Scroll down to find the sine wave  and then  into the example workspace and place carefully just at the the end of the red line where the previous source was. This should then make the connections as were previously set. If the connections are not made for you, simply place the sine wave and then manually make the connections according to the procedures described earlier in *Action 11* (page 14) and *Action 13* (page 15).

(c) On the sine wave  now  on the parameter dialog, change the

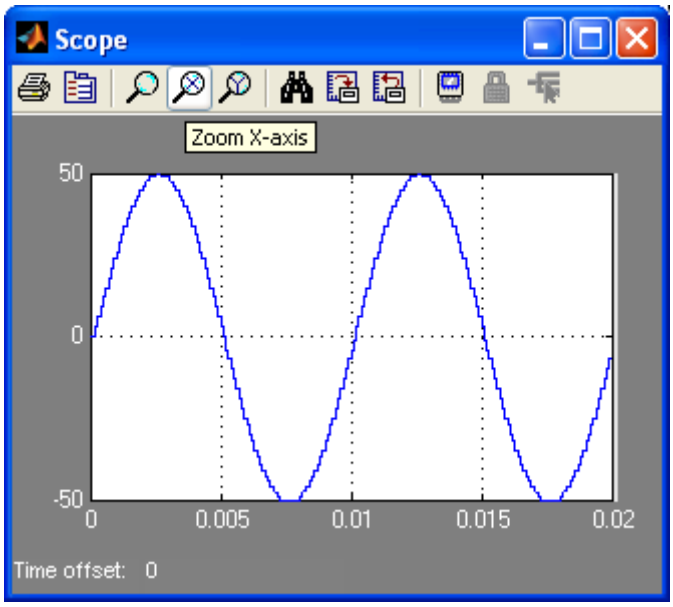
to `50`

and 

to `2*pi*100`

Note that the frequency parameter is in radians/sec hence to convert to Hz we simply multiply by 2π. In Simulink just type “pi” for π and “`*`” for multiply.

(d) Run the simulation and confirm the generation of a sine wave of amplitude 50 and frequency 100 Hz for 200 samples at a sampling rate of 1000Hz. The output should look like as shown below.



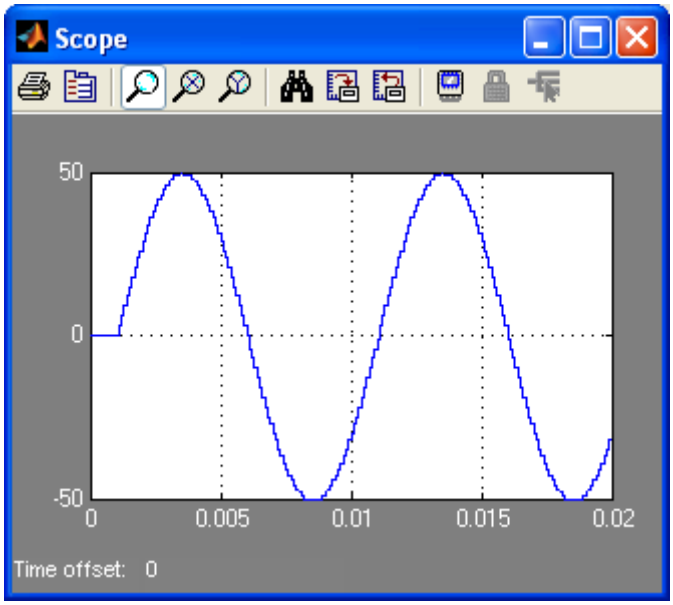
Note that in order to appropriately see the full sine wave you may require to appropriately zoom in or out of the window using the various scope toolbar buttons marked above.

(e) Next we will modify the parameters of the delay block. Open the delay block by on it and change the Latency to 10.

(f) Rerun the simulation and view the scopes to observe the sine wave output is delayed by 10.

Note that for user information the delay icon in Simulink shows the internal parameter by labelling with z^{-10} to indicate 10 delays or a latency of 10.

The scope output is likely to look as shown below (remember to scale/ autoscale if necessary to see the full sine wave in the window.)



Exercise 2.4 Working with Fixed Point Input - Wrap and Saturate

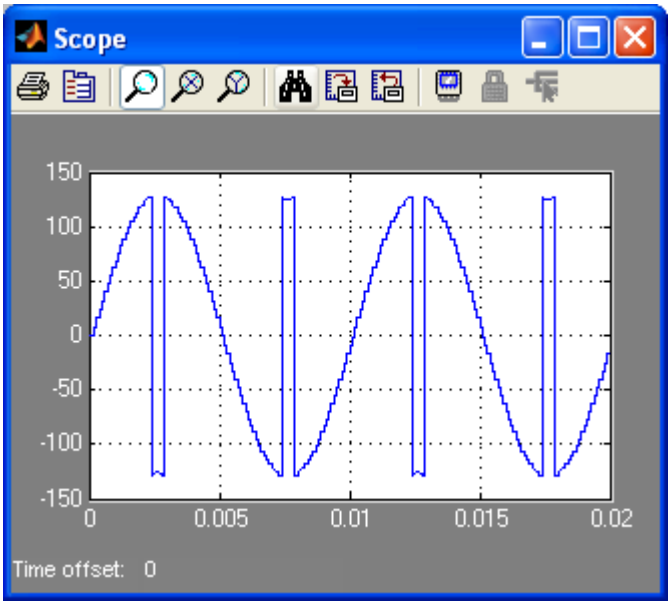
Open the system:

\fixedpoint\delay\delay.mdl

This system is simply a delay. The gateway in is set to a resolution of 8 bits, and the gateway out is set to a resolution also of 8 bits.

(a) Run the simulation and note that the sine wave in, gives a sine wave out.

(b) Change the input amplitude of the sine wave input from 50 to 130 and rerun the system. You should note that the output now has the form:



(c) Explain what has happened here?

Answer:

(d) Now  on the gateway in block  and change Overflow from *wrap* to *saturate* (use the pulldown). This will change the arithmetic mode of the gateway in from wrap around to saturate.

Run the system and observe that the peak values ***saturate*** at 127 and -128 (use the zoom in the scope to confirm this is precisely the case. This block is now operating in saturate mode.

(e) Change the input amplitude of the sine wave input from 130 to 270 and rerun the system again. You should now observe quite significant clipping.

Exercise 2.5 Flagging Overflow as Error

Open the system:

`\fixedpoint\delay_overflow\delay_overflow.mdl`

This example is the system you produced after making the modifications in the previous Exercise 2.4.

(a) Run the simulation and note again in the output scope that significant overflow occurs on the gateway-in block  which has been set on saturate mode.
Of course in a real simulation overflow is a serious problem and therefore it is important that this is flagged to the designer. System Generator will allow this.

(b)  on the block  and change Overflow from "*saturate"* to "*flag as error*". Rerun the simulation and observe what happens.

System Generator has essentially trapped the overflow and flagged this as an error.

(c) We will now "fix" the error. First note that the input amplitude of the sine wave is 270. To represent this range would require 10 bits, i.e:

-2^9 to $2^9 - 1$ or -512 to 511

2 on the block 

and change Number of Bits from 8 to 10.

Run the simulation and note that the sine wave in, gives a sine wave out.

(d) This simulation should now run with no error and if you view the output ampltude of the sine wave it should look OK.

---

3 Simple Arithmetic

This section of the workbook reviews some fundamental issues of arithmetic for DSPs.

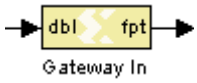
Exercise 3.1 Gateway Block Underflow and Overflow

Recall that overflow is when a number is too large to be represented in a fixed number of bits, and underflow is when a number is too small to be represented (and the result is zero).

Open the system:

\fixedpoint\gateway_overflow\gateway_overflow.mdl

This system simply uses the gateway-in block to quantise data to 4 bits or 5 bits with various modes of overflow set on and off: .



Run the system and complete the table below for the various input values. Ensure you interpret what is happening with respect to the binary numbers.

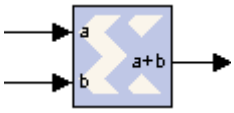
Decimal Input	5 bits Binary	4 bits Wraparound		4 bit Saturate	
		Binary	Decimal	Binary	Decimal
6	00110	00110	6	0110	6
13	01101	1101	-3	0111	7
7					
-4					
-9					
-16					

Exercise 3.2 Arithmetic Overflow

Open the system:

\arithmetic\add_overflow\add_overflow.mdl

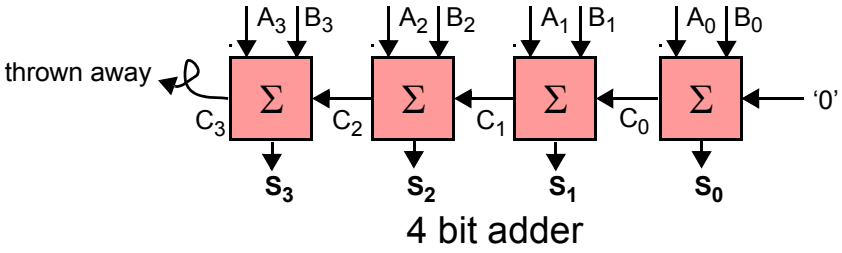
(a) Note that there are three systems. Each differs in the setting of the adder:



Top System: 4 bit adder has Overflow set to wraparound
Middle System: 4 bit adder has Overflow set to saturate
Bottom System: 5 bit adder has Overflow set to wrap

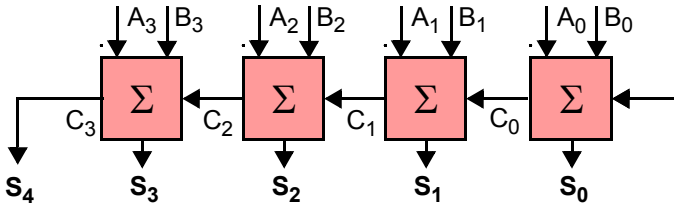
In these systems 4 bit inputs are being added together and in the case of the 4 bit adder the 5th bit is discarded and for the 5 bit adder, the fifth bit is available from the final carry out:

$$\begin{array}{r} A_3\ A_2\ A_1\ A_0 \\ +\ B_3\ B_2\ B_1\ B_0 \\ \hline S_3\ S_2\ S_1\ S_0 \end{array}$$



4 bit adder

$$\begin{array}{r} A_3\ A_2\ A_1\ A_0 \\ +\ B_3\ B_2\ B_1\ B_0 \\ \hline S_4\ S_3\ S_2\ S_1\ S_0 \end{array}$$



5 bit adder

(b) Run the system and observe the outputs for adding 6+3 with first 4 bits and then 5 bits. Write the binary answer in the table below and hence interpret the decimal answer shown by Simulink.

Adder	A + B	Binary	Result	Binary Result
4 bits in / 4 bits out/ **Wraparound**	6 + 3	0110 +0011	-7	
4 bits in / 4 bits out/ **Saturate**	6 + 3	0110 +0011	7	
4 bits in / 5 bits out/ **Saturate**	6 + 3	0110 +0011	9	

(c) Run the system and observe the outputs for calculation -4 + (-8). Write the binary answer in the table below and again interpret the decimal answer shown by Simulink.

Adder	A + B	Binary	Result	Binary Result
4 bits in / 4 bits out/ **Wraparound**	- 4 - 8	1100 −1000		
4 bits in / 4 bits out/ **Saturate**	- 4 - 8	1100 −1000		
4 bits in / 5 bits out/ **Saturate**	- 4 - 8	1100 −1000		

(d) Change the middle system to stop on error by changing the Overflow option to *error*.

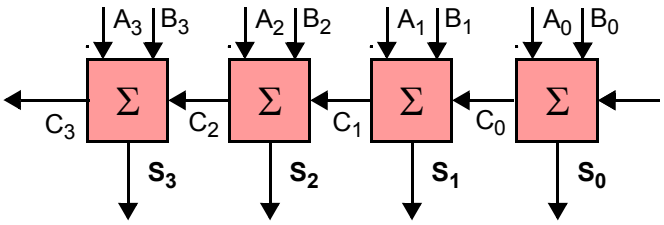
Generally we would like to design DSP FPGA systems such that they "never" overflow, however this is not always possible. Hence it is important to fully understand how to manage overflows by allow wraparound (no action, or setting up saturate) when they occur.

3.1 What Hardware Cost is Saturate?

In most DSP situations saturate on overflow is the favoured option as the sign of the number will at least be correct. In some applications where feedback is required this is very important as sign changes in a feedback loop cause serious problems.

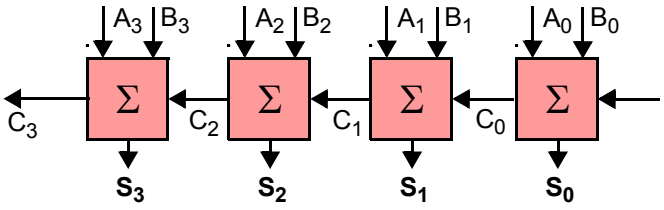
Saturation does not come for free however!

QUESTION 1:
For the 4 bit adder (using 2's complement arithmetic) below, design some additional circuitry that will detect when the result has overflowed.



Once overflow has been detected, based on an overflow output we could use a multiplexor to then switch in the most positive or most negative 4 bit value depending on whether the overflow was negative or positive.

QUESTION 2:
Using your answer to the previous question design a circuit that will allow saturate (positive or negative) to be implemented on detection of overflow.



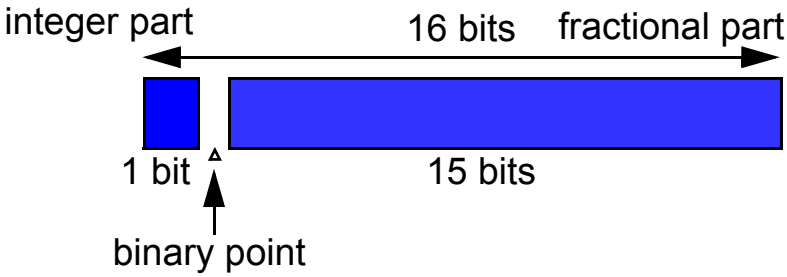
5 bit adder

Exercise 3.3 Adding with rounding

Open the system:

\arithmetic\add_rounding\add_rounding.mdl

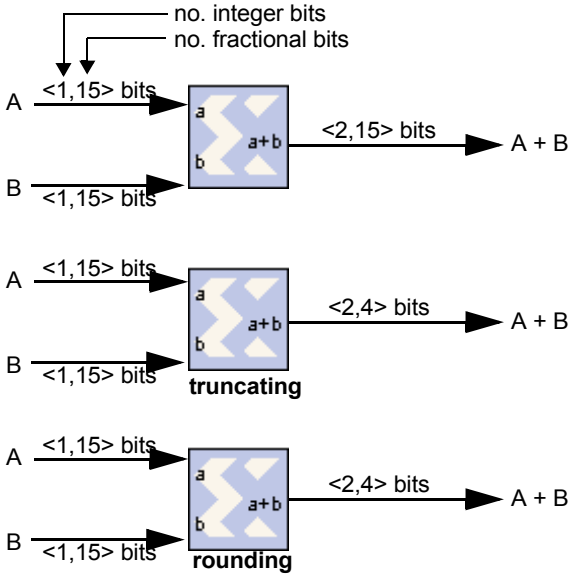
In this example two sequences of random numbers are generated. These take values in the range [-1,1[, i.e. -1 is included but 1 is not. 16 bits are used to represent this sequence, 1 bit to represent the integer part and 15 to represent the fractional part.



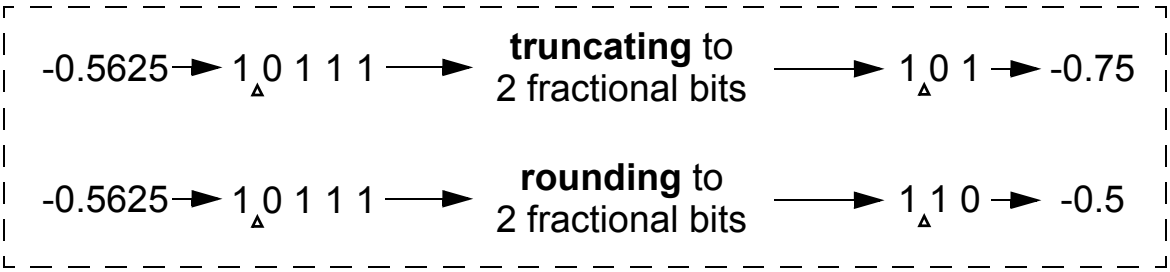
examples:

decimal	signed two's complement
0.15625	0.001010000000000
0.157379150390625	0.001010000100101

Three adders are presented in this system as shown below. The first one will produce a result with 2 integer and 15 fractional bits. The second produces an addition with 2 integer and 4 fractional bits. The reduction in wordlength is achieved by truncating the result. The third adder produces the same size of result but uses rounding instead of truncating.



Remember that truncating "throws away" the bits that are not needed, while rounding finds the closest representable value as shown below:



(a) Note that the top adder in the simulation have all parameters set to ☑ Override with Double in order that the result is computed with floating point double precision. The Simulink subtractor blocks therefore calculate the difference between the floating point and the quantised values.

(b) Noting the magnitude of the errors in each case (you might need to use the Scope scaling buttons), confirm truncation/rounding errors lie in the range that is theoretically expected.

The procedure of rounding does not come free. In fact the cost is one full extra adder.

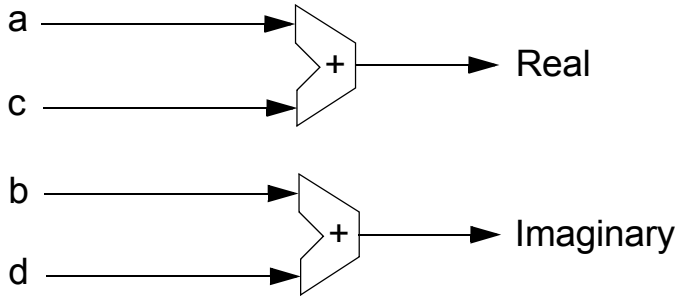
3.2 Complex Arithmetic

In a number of DSP implementations the use of complex arithmetic is required. In particular for quadrature modulated systems and related equalisers the use of complex filters and complex arithmetic is very useful. In this section we will use use System Generator blocks to implement complex arithmetic.

The addition (or subtraction) of two complex numbers is very simply accomplished using two real adders.

$$(a + jb) + (c + jd) = (a + c) + j(b + d)$$

Thus 2 additions/subtractions are required:



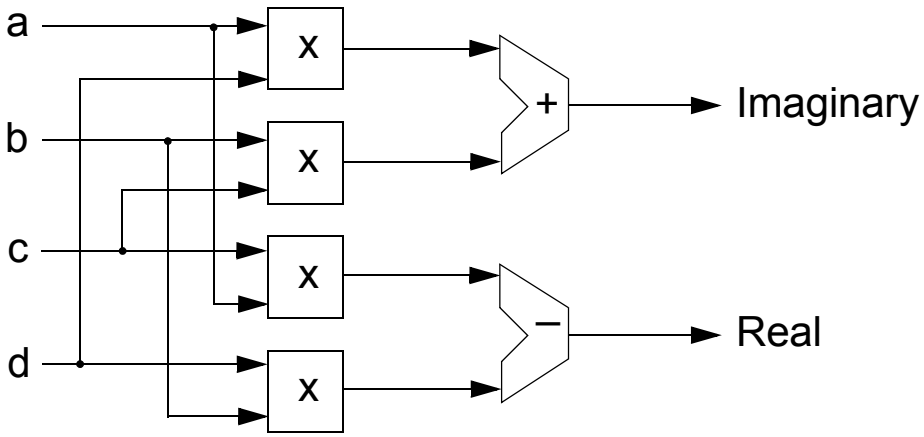
Exercise 3.4 Complex Addition

Confirm the correct operation of the 8 bit adder.

\arithmetic\complex_add\complex_add.mdl

Complex multiply is more complex and requires the use of 4 real multipliers and 2 real adders:

$$(a+jb)\times(c+jd) = (ac-bd)+j(bc+ad)$$



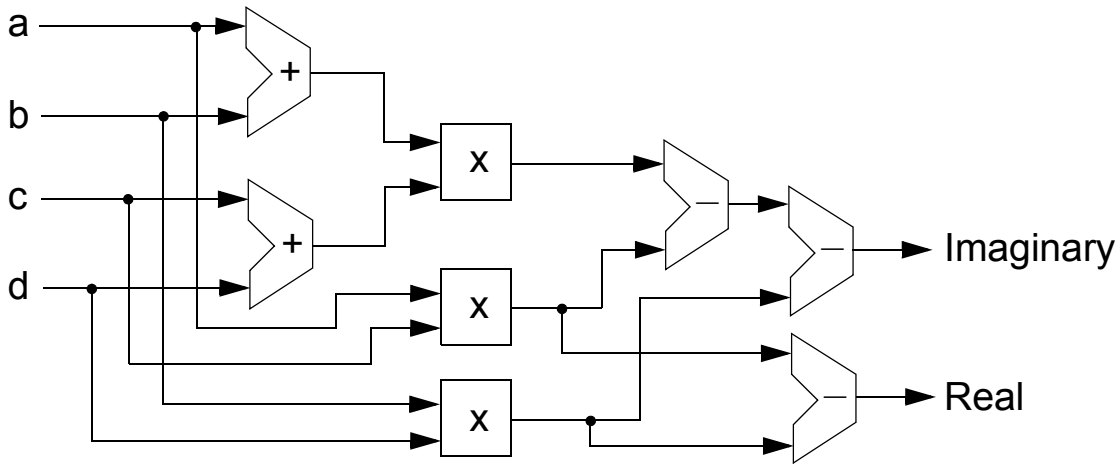
Exercise 3.5 Complex Multiply

Confirm the correct operation of the 8 bit multiplier:

\arithmetic\complex_mult\complex_mult.mdl

Using some simple algebra, the 4 real multiply, 2 real add version of the complex multiply can actually be implemented with 3 real multiplies and 5 real adds:

$$(a+jb)\times(c+jd) = (ac-bd)+j[(a+b)\times(c+d)-ac-bd]$$

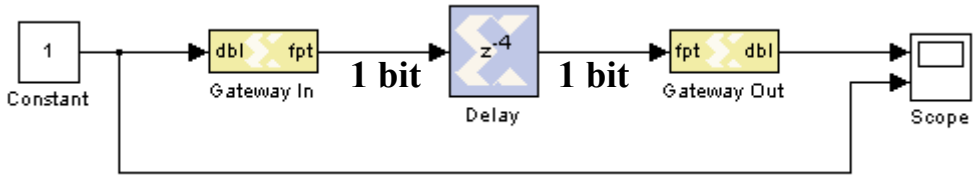


4 Designing for Xilinx ISE Tools

In this section we will take some simple designs and from first principles synthesise them to Xilinx FPGAs. We will start with some very simple examples where we will describe each step and facility in detail.

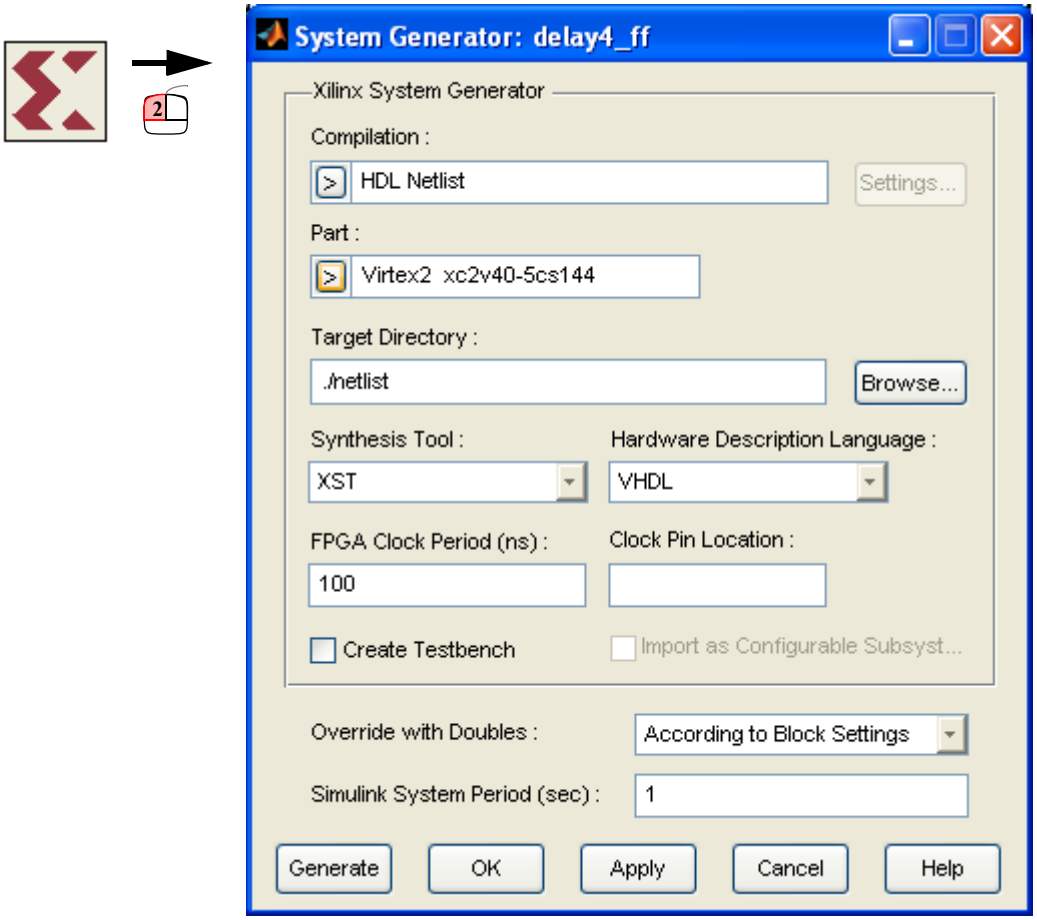
Action 15: ***SYSTEM GENERATOR SYSTEM DESIGN.*** *Open the Simulink*

`\isetools\delay4_ff\delay4_ff.mdl`



This system simply implements four single bit delays. Run the system and confirm on the scope output shows a delay of 4 samples.

Action 16: ***CONFIRMING THE SYSTEM GENERATOR PARAMETERS.*** *View the parameters in the System Generator block by* 2 *on the Xilinx block*



View the various parameters in this window and where necessary check in the Help *on the meanings of the various parameters.*

Note that the target device is set to the Virtex2 XC2v40 device.

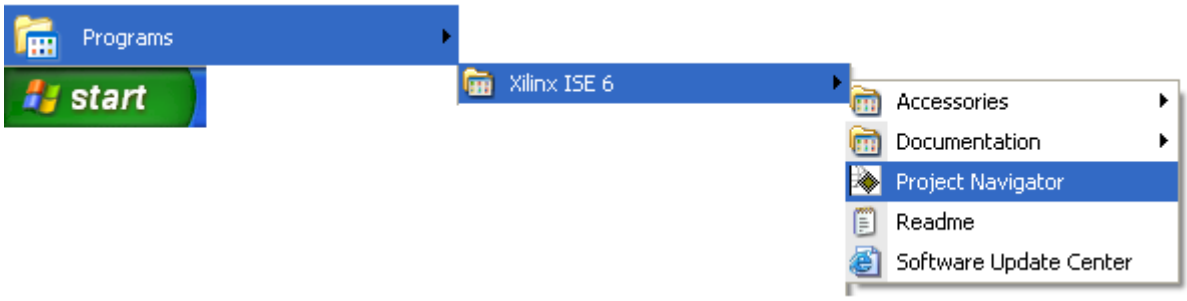
Action 17: ***GENERATING TO ISE TOOLS.*** *In the System Generator dialog,* *on* Generate *until you receive the message:*

Note that in the example directory `\isetools\delay4_ff` *a new directory called* **`\netlist`** *(observe in the above System Generator dialog this was set as the target directory) has been produced and inside this directory there are a number of files that have been produced:*

Name	Size
temp	
clkwrapperinterface	23 KB
globals	1 KB
hdlFiles	1 KB
sysgeninterface	11 KB
delay4_ff_clk_wrapper.xcf	1 KB
delay4_ff_clk_wrapper.ise	5 KB
delay4_ff_config.m	2 KB
conv_pkg.vhd	56 KB
delay4_ff_clk_wrapper.vhd	24 KB
delay4_ff_dcm_wrapper.vhd	4 KB
delay4_ff_files.vhd	33 KB
xst_delay4_ff.prj	1 KB
xst_delay4_ff.scr	1 KB
clkwrapperinterface.txt	12 KB
delay4_ff_clk_wrapper_import.txt	3 KB
postnetlist.log	2 KB
sysgen.log	1 KB
sysgeninterface.txt	7 KB

These files can now be taken into the Xilinx ISE (Integrated System Environment) in order to begin the stages of taking the design to FPGA.

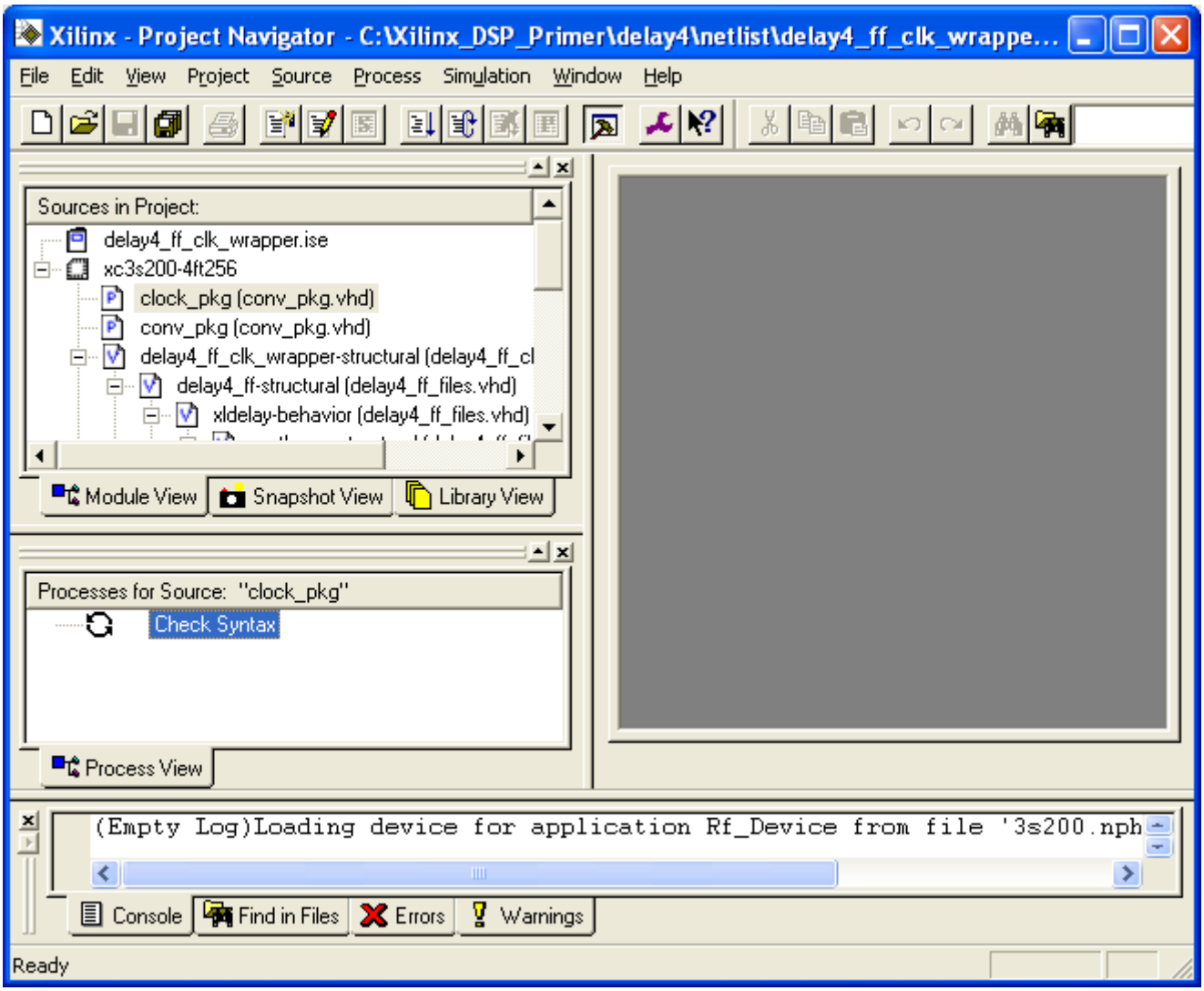
Action 18: ***OPENING THE XILINX ISE TOOLS.*** *Open the Xilinx ISE tools from:*



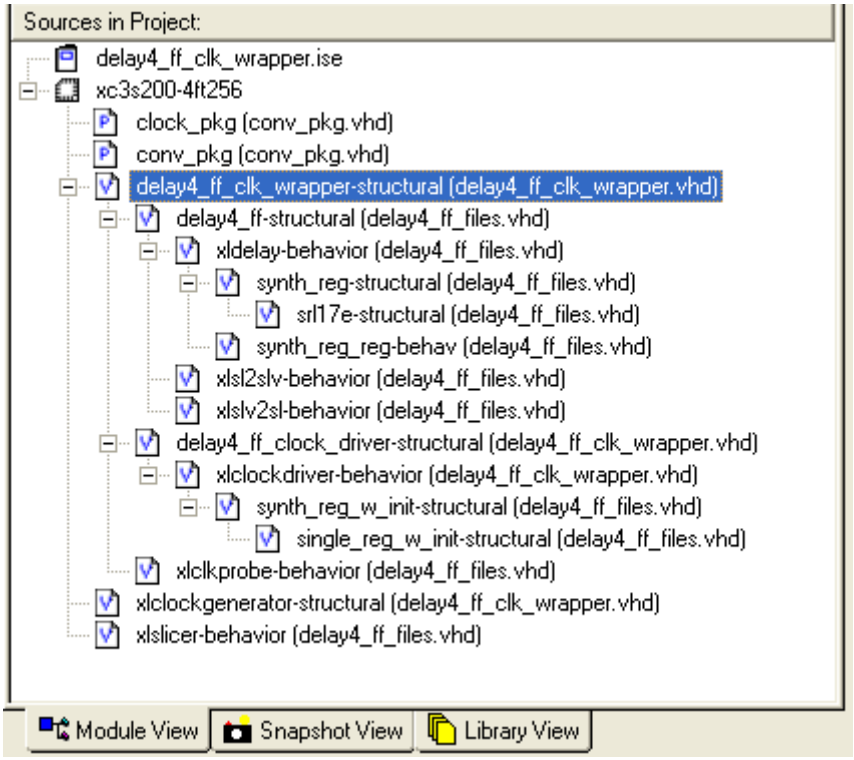
Select **FILE > OPEN** *and choose the file:*

`\isetools\delay4_ff\netlist\delay4_ff_clk_wrapper.ise`

This will open the main project file that was just created by System Generator and you should see a window similar to:

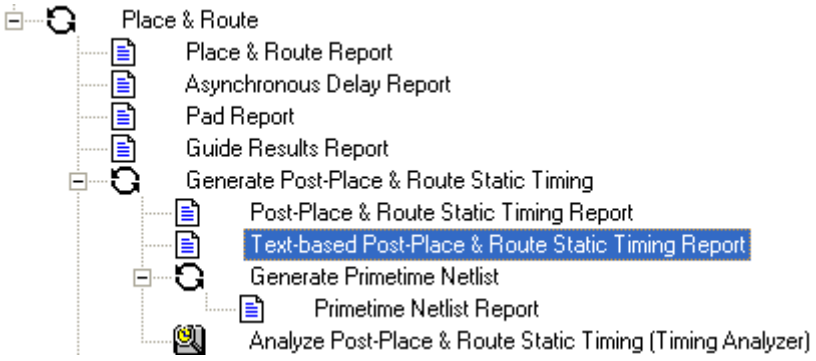


Action 19: ***SOURCES IN PROJECT.*** *Observe the various sources that are listed in the project. Using the mouse select the main VHDL file by* 1 *on the file highlighted below:*



Notice that in the Processes for Source window there are a large number of options available to be applied to the source (See Figure 4.1). We will use few of these options in this course, however there will not be time to work with or explain all available facilities.

Action 20: ***IMPLEMENTING THE DESIGN.*** *Select the option in "Place and Route" shown below by* 1 *on "Text-based Post Map Static Timing Report": by*



Choosing this option will run a series of process on the selected source with a timing report finally being implemented that we can view. This can be done by either 2 *on* Text-based Post-Place & Route Static Timing Report *or choose to* 1 *and select* ***run***

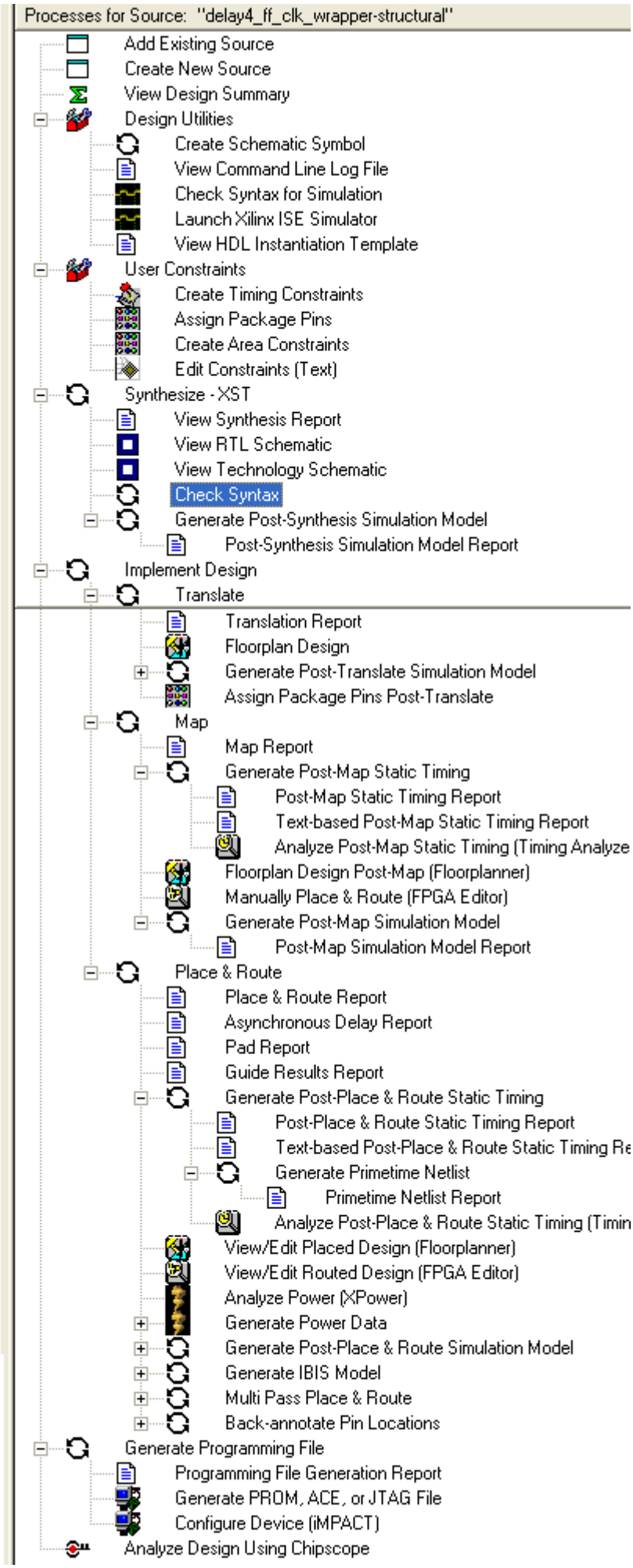
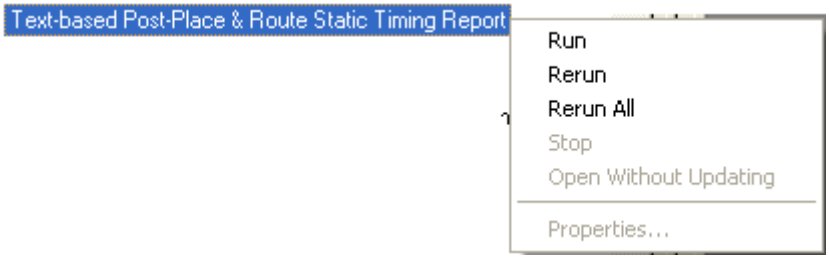


Figure 4.1: Xilinx ISE available processes for source

as shown below:



Observe the activity that is now being reported in the Console *window.*

When complete, the last message you should see in the console is:

```
Generating Report ...

Number of warnings: 0
Total time: 5 secs
```

You will also note that there are green ticks ✔ *against the processes that were successfully completed.*

Finally if you look again in the **\isetools\delay4_ff\netlist** *directory you will notice that many new directories and files have been created (recall viewing this directory in Action 17)*

Action 21: ***VIEWING THE TIMING REPORT.*** *The timing report should now be visible in the top right pan of the ISE window. You can scroll through this data to view more closely the reported information. Note that by* 1 *on the ">>" icon in the top right*  *you undock the window from the main ISE window. The name of this file is* **delay4_clk_ff_wrapper.twr**.

To redock the window, simply select << from the top right of the undocked window.  .

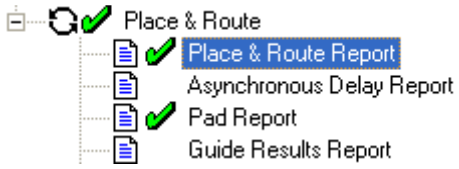
Scroll down the file until you come to Timing Summary.

What is the minimum period? *ANSWER:* _____

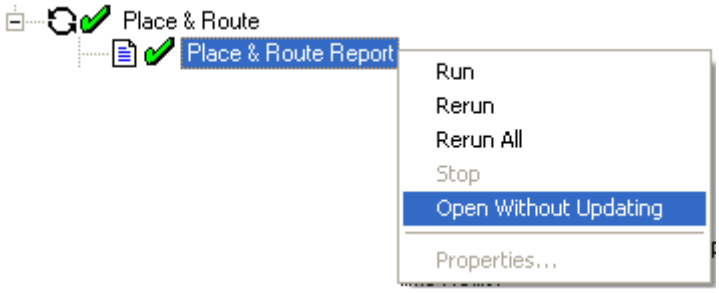
What is the maximum frequency? *ANSWER:* _____

Action 22: ***PLACE AND ROUTE REPORT OF THE DESIGN.*** *Next we will view the Place and Route report which will tell us about the size of the design.*

Using the mouse, 2 *on* ✔ Place & Route Report *in the Proceses for Source:*



or alternatively 1 *on* Place & Route Report *and select*



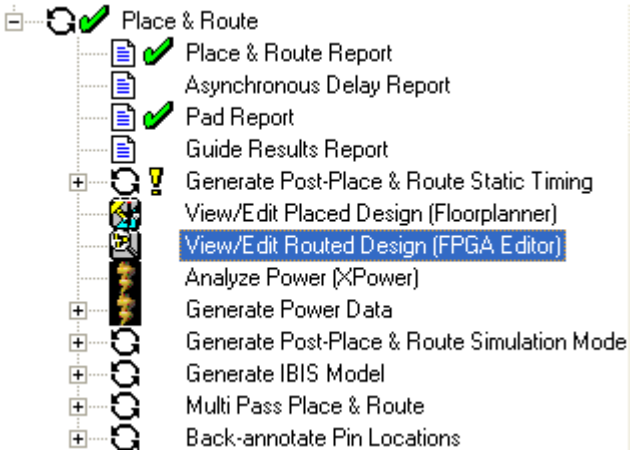
This will open the place and route report **`delay4_clk_ff_wrapper.par`**

Scroll down a few lines until you see the device utilisation summary:

```
32  Device Utilization Summary:
33
34     Number of BUFGMUXs                  1 out of 8
35     Number of External IOBs             3 out of 173
36        Number of LOCed IOBs             0 out of 3
37
38     Number of Slices                    1 out of 1920
39        Number of SLICEMs                0 out of 960
```

Action 23: ***VIEWING THE FLOORPLAN.*** *We will now look at the floorplan of the device with the intention of viewing the actual hardware location.*

In the processes for source window, 2 *on* View/Edit Routed Design (FPGA Editor) *:*



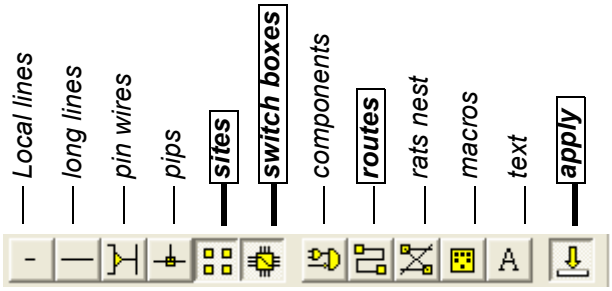
This will start up the FPGA editor which will allow use to view the actual

implementation and inspect various slices and interconnections.

Action 24: ***VIEWING THE DEVICE.*** *Recall the size of the target device beng used in this example:*

Device	Family Name	CLB Array	Number of Slices	Logic Cells	CLB FlipFlops	Block RAM (bits)	Max I/O	18 x 18 Multipliers	No of DCMs	DCM Freq (min\max) MHz
XC2V40	Virtex II	8 x 8	256	576	512	72	88	4	4	24/420

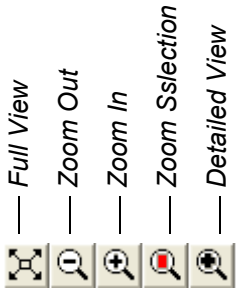
With 8 CLBs and 4 slices per CLB this gives a total of 256 slices. To get a clear view of the FPGA, select the toolbar options shown below. i.e select on ***sites****,* ***switch boxes****,* ***routes****, and enure* ***apply*** *is selected (in order that the actual selections are applied):*



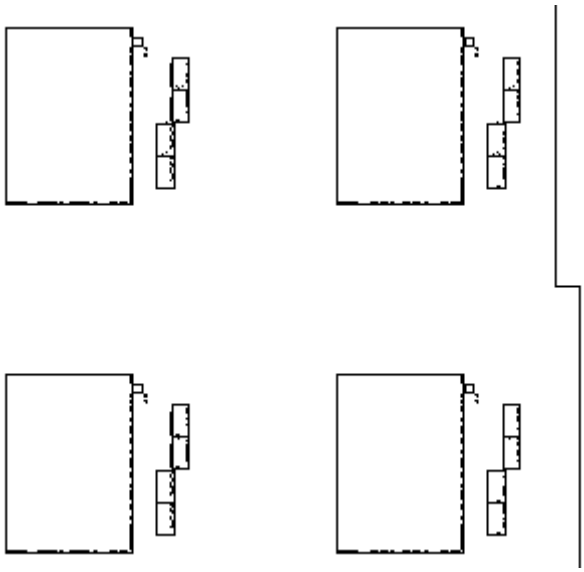
The Array1 window should now show the FPGA layout and highlight the require connections for our design. Your result should be similar to (but may

not be identical to) the result below.:

Action 25: ***CONFIRMING THE FPGA COMPONENTS.*** *We will now zoom into the design in order to view the various components. FPGA editor provides a number of useful zoom facilities. Note the various zoom buttons available:*



Select zoom in  *twice using*  *in the Array1 window and you should now begin to see a more detailed view of this particular device. Zoom in far enough and use the scroll bars to see a view of a few slices as shown below:*



Note in the World1 *window on the lower right of the FPGA editor you can see where you are viewing on the device.*

In the slices and switch boxes you have zoomed to, on the slices 1 *until one is highlighted red and select the zoom selection toolbar button* *to zoom into that device.*

After zooming you should then be able to see the actual slice number (ensure that A *is selected to show text). Using the zoom in* *button (and zoom out* *if necessary) enlarge a slice until it can be clearly viewed in the window and you can see that actual slice number.*

Note if you just place the mouse pointer on top of a component (such as a slice) a hint box will detail the actual contents.

Action 26: ***VIEWING A SLICE IN DETAIL.*** *From the previous Action or otherwise, find a slice, then select* 1 *it (until it turns red) and* 2 *on the slice which will then show the actual slice in detail as shown in Figure 4.2 (next page).*

After viewing close the slice window.

Action 27: ***FINDING THE DESIGN.*** *Recall that the actual logic design was rather*

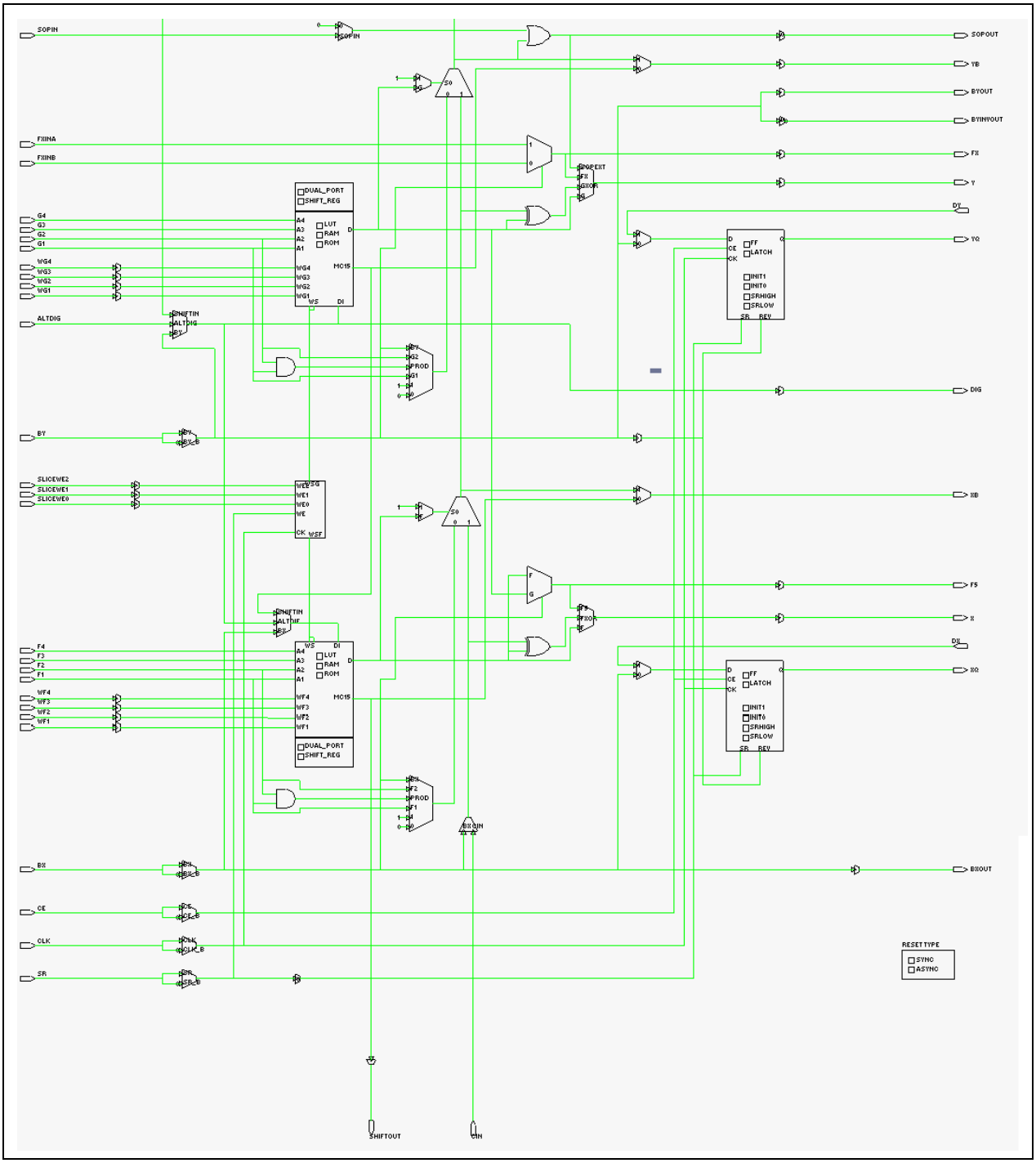
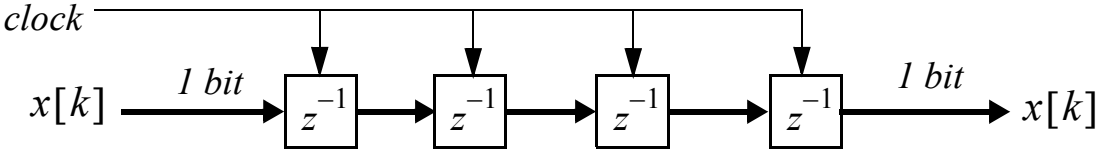


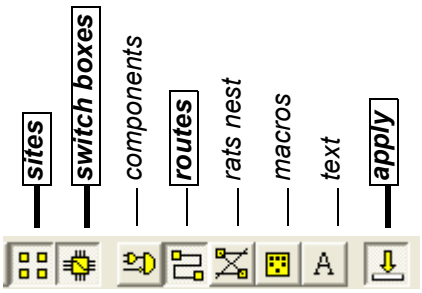
Figure 4.2: Viewing a single slice with FPGA editor

simple and was composed of simple single bit delays.



So in order to confirm our design we will use the FPGA editor to "search" for our implementation.

Choose to zoom out to a full screen view by 1 *on* . *Ensure that the following toolbar options are set:*



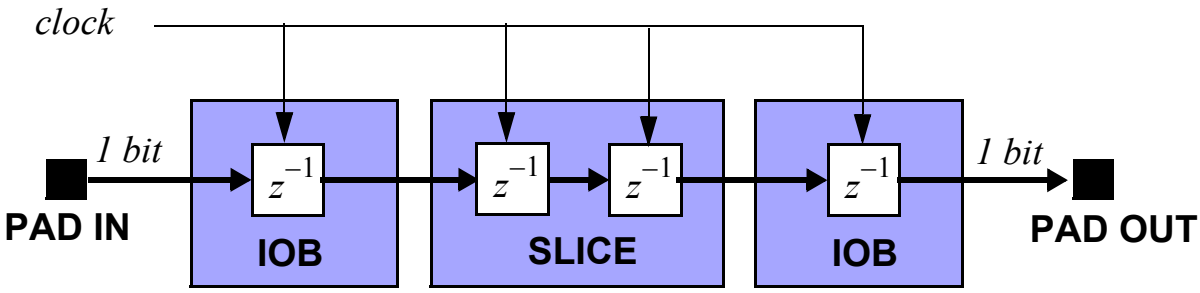
You should now be able to see the entire FPGA with light blue lines showing the various connections between IOBs, slices and other components. Our interest is to find the three flip-flops, the inputs and outputs and the clock signals.

Find the IOBs (which will be at the edges of the FPGA). Zoom in as necessary and place the mouse on top of either of the IOBs and you should see a message smilar to the following hint box (note the actual IOB number may be different):

comp "gateway_out", site "G12", bonded type = IOB, pad name = PAD37, pin name = G12 (RPM grid X27Y19)
comp "gateway_in", site "H12", bonded type = IOB, pad name = PAD38, pin name = H12 (RPM grid X27Y18)

You can see that the FPGA editor maintains the integrity of the names from the original SystemGenerator simulation.

2 *on both IOBs and on the used slice and you should be able to track the actual three delays, the input output pads, and the actual clock (if you* 1 *on the clock wire it should highlight the entire clock path in red). Therefore the three delays are implemented as two flip-flops in one slice and one flip flop in each of the IOBs:*



Action 28: ***INCREASING THE COMPLEXITY OF THE DESIGN (..a little!).*** *Open the Simulink model:*

`\isetools\delay10_ff\delay10_ff.mdl`

Run the system and confirm a 10 sample delay in the scope.

in the System Generator block and then in the System Generator dialog, on Generate *. The new ISE project will now be produced.*

Go to the Xilinx ISE tools again and select **FILE > OPEN** *and choose the file:*

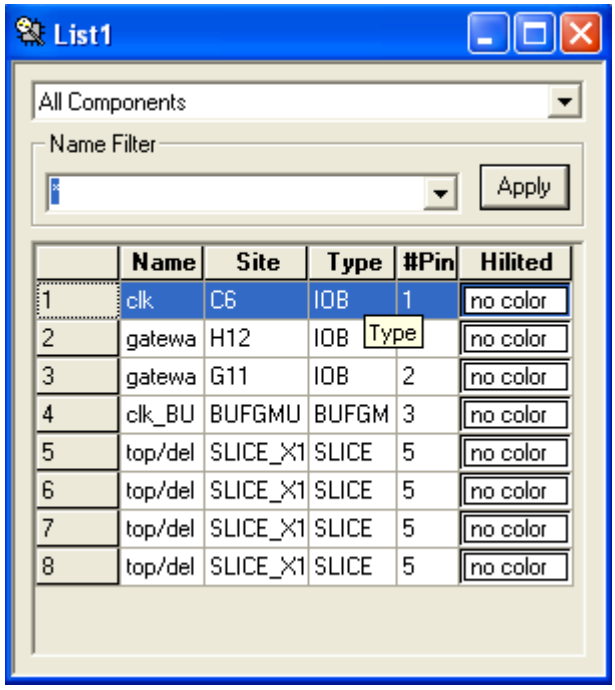
`\isetools\delay10_ff\netlist\delay10_ff_clk_wrapper.ise`

Repeat Action 19 (page 32) to Action 22 in order to produce the Timing Report and the Place and Route report and complete the table below:

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

You should notice that the speed has dropped slighlty, and the number of slices has increased by a factor 4. Can you justify this compared to the previous design with only 3 delays?

Action 29: ***USING THE COMPONENT LIST TO FIND SAMPLES.*** *For the system in the previous action, open the FPGA editor for the synthesised design and note the* List1 *window which displays information on the various components:*



Try on various Names, and note that the component is highlighted in both the World1 *window and the main (floorplan)* Array1 *window. Note the other details given, and also the "hilited" option to allow different colours to be*

assigned to different components.

Search through the floorplan to find the 8 single delays.

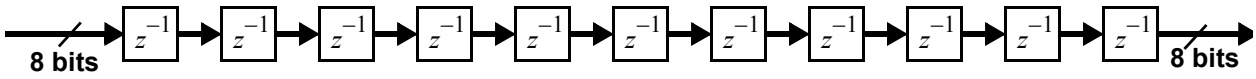
4.1 Building Delays Lines

In this section (now that we have used the various Simulink, System Generator and Xilinx ISE tools) we will produce delays lines and select a few options from in doing this.

Exercise 4.1 Length 10 delay line with 8 bit data using flip flops (FF)

Open the system:

\delaylines\delay_8bits_ff\delay_8bit_ff.mdl



(a) Run the system and confirm a 10 sample delay for the 8 bit input.

(b) Run the system and confirm a 10 sample delay in the scope.

(c) 2 in the System Generator block and then in the System Generator dialog, 1 on Generate to produce the new project file for the ISE tools.

(d) Goto the ISE tools and open the new ISE file:

\delaylines\delay_8bits_ff\netlist\delay_8bits_ff.ise

In the table below fill in the various fields:

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(e) Recall there are 2 FFs per slice, confirm that the the number of slices you have is consistent with the 10 delays in an 8 bit data path (remember that the IOBs provide input/output FFs also.

Exercise 4.2 Length 10 delay line with 8 bit data using SRL16s

Open the system:

\delaylines\delay_8bit_srl16\delay_8bit_srl16.mdl

In this system the FPGA implementation will actually use the LUTs to implement 16 bit shift registers (SRL16s) rather than the flip-flops (FF). Note

that this is something that can be explicitly setup in the actual delay block.

(a)  on the delay block  and note the text at the top:

"Hardware notes: A delay line is a chain, each link of which is an SRL16 followed by a flip-flop. If register timing is enabled, the delay line is a chain of flip-flops."

In the previous Exercise 4.1, register timing was enabled and hence the final implementation used a chain of flip-flops.

(b) Run the Simulink system and then following the same procedures as previously, take the system all the way to the ISE tools and complete the table below:

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(c) Compare this to the previous result from Exercise 4.1.

(d) View the actual FPGA implementation and observe the use of the LUTs as shift registers (SRL16).

(e) Given the total number of slices used, confirm this is about the number you would expect for this 8 bit, 10 sample delay line.

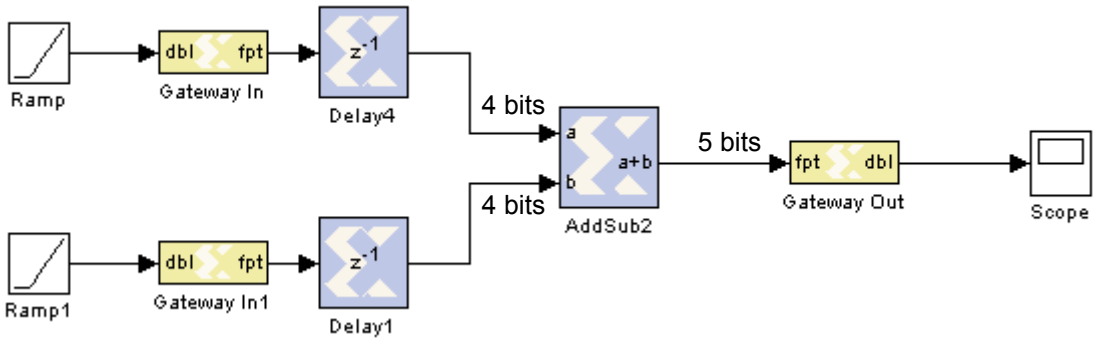
4.2 Arithmetic Components

In this section we will produce a few simple implementations of the arithmetic operations of addition and multiplication in order to see how this is implemented in slice logic.

Exercise 4.3 Implementation of a 4 bit adder producing a 5 bit result

Open the system:

\arithmetic\AdderChain\AdderChain.mdl



In this system we have set up a simple adder chain with the objective of viewing this on the FPGA.

(a) Run the system and confirm that the two 4 bit numbers are indeed added together to produce a 5 bit result.

(b) Generate a suitable ISE project and complete the table below after timing and place and route reports have been completed.

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of SLICEs	

(c)

Exercise 4.4 Implementation of a 16 bit adder producing a 17 bit result

Open the system:

\arithmetic\AdderChain_16\AdderChain_16.mdl

In this system we have set up a 16 bit adder producing a 17 bit result.

(a) Run the system and confirm that it runs correctly.

(b) Generate a suitable ISE project and complete the table below after timing and place and route reports have been completed.

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of SLICEs	

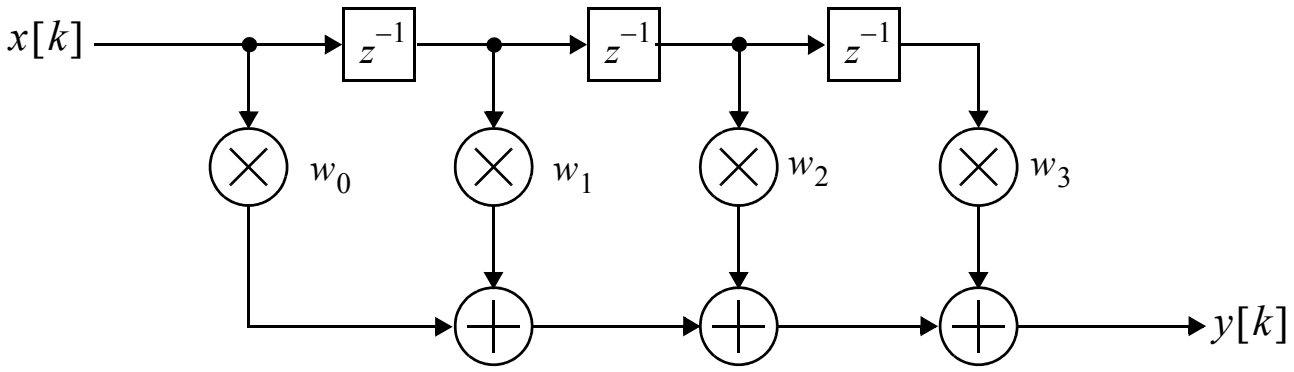
(c) How does this compare to the previous exercise?

---

5 FIR filtering

This section of the workbook will consider the implementation of FIR filters via a number of different strategies.

To illustrate these issues, we focus on just a simple four-tap FIR filter as follows:



The filter weights are chosen to be:

$$w_0 = -10, w_1 = 20, w_2 = 50, w_3 = 80 \quad [5.1]$$

These weights are not chosen to give a specific frequency response for the FIR filter - just to show something observable.

5.1 Wordlength growth

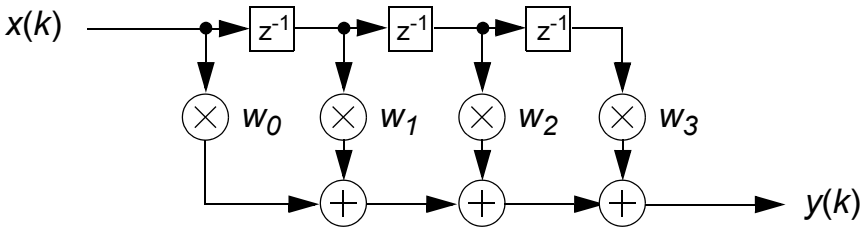
To illustrate the issue of wordlength we must of course consider finite precision signals. For this example, we make the choice that the input signal $x[k]$ is an 2-bit integer. Consequently the signal $x[k]$ has values lying in the range -2 to 1.

In these examples we shall consider the filter weights $\boldsymbol{w}$ to be 8 bits, and therefore the weights could take any value between -128 and 127.

Exercise 5.1 Simple FIR Filter

Open the system:

\filter\FIR1\FIR1.mdl



(a) Sketch the critical path in the above FIR filter. How many multiplier and adder delays in this critical path?

ANSWER:

(a) Run the system and view the impulse response in the scope.

(b) View the wordlength in the adder line and confirm and note the word growth from 8 bits to 10 bits.

(c) Using system generator produce an ISE file and in the Xilinx ISE tools view the timing and place and route reports to complete the table below:

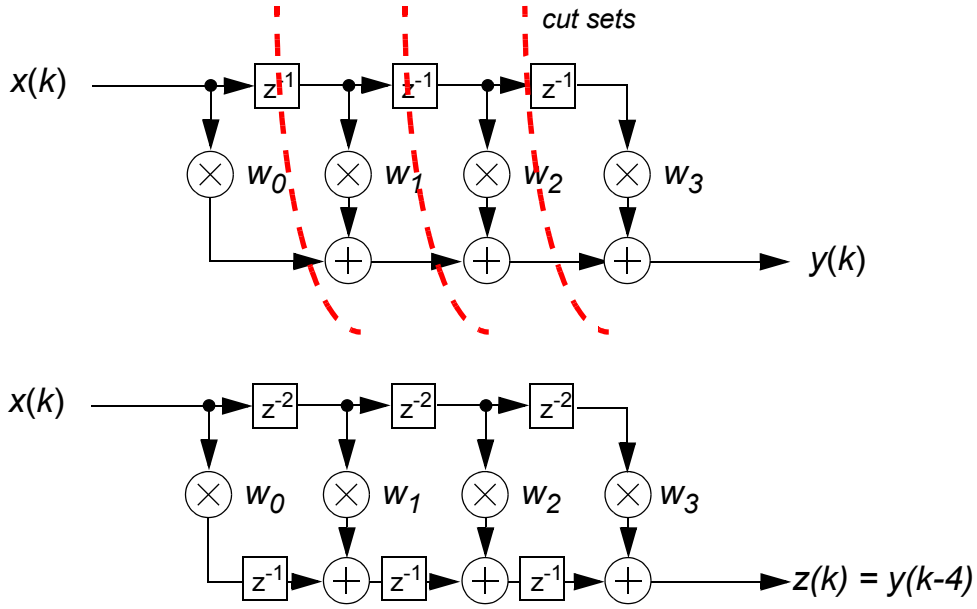
Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(Note that there will be no multipliers used in this example as we have currently set the implementation not to use embedded multipliers.)

Exercise 5.2 Retiming FIR Filter

Open the system:

\filter\FIR2\FIR2.mdl



This system has been produced by applying the cut sets in the top figure to produce the signal flow graph in the lower figure in order to reduce the critical path latency.

(a) Run the system and view the impulse response in the scope.

Note that compared to the previous example the critical path latency is now much shorter. What is the new critical path latency?

ANSWER:

(b) Using system generator produce an ISE file and in the Xilinx ISE tools view the timing and place and route reports to complete the table below: :

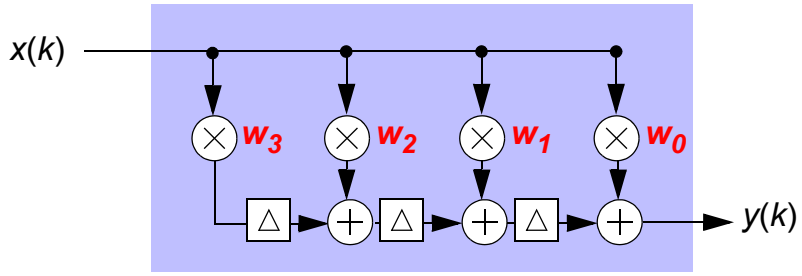
Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

This should run faster than the previous example due to the shorter critical path, but will require some more hardware.

Exercise 5.3 The Transpose FIR Filter

Open the system:

`\filter\fir_transpose\fir_transpose.mdl`



(a) Run the system and view the impulse response in the scope. What is the critical path latency of the transpose FIR?

ANSWER:

(b) Using system generator produce an ISE file and in the Xilinx ISE tools view the timing and place and route reports to complete the table below:

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Exercise 5.4 Reducing the Critical Path of the Transpose FIR Filter

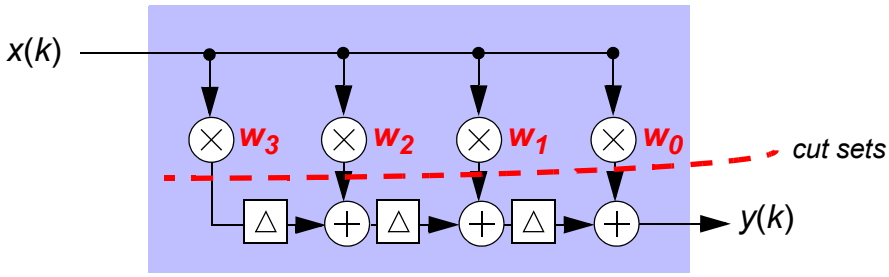
The critical path in the transpose filter can be further reduced by introducing further pipelining delays after the multiplier. By performing the cut sets below, modify the system:

\filter\fir_transpose\fir_transpose.mdl

to include delays/registers ***after*** the multiplier.

Save the implementation to a new **`fir_transpose2`**:

\filter\fir_transpose2\fir_transpose2.mdl



(a) Run the system and view the impulse response in the scope. What is the critical path latency of the transpose FIR?

ANSWER:

(b) Using system generator produce an ISE file and in the Xilinx ISE tools view the timing and place and route reports to complete the table below:

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Is it faster than the standard transpose without pipelining the multipliers?

Exercise 5.5 Different FIR Filter Costs

In this exercise we will open a number of different FIR filters implemented using various combinations of LUTs, slice logic, block multipliers, and different delay line implementations.

In all cases the number of weights is 4, the input wordlength is 8 bits, the filter weights wordlength is 8 bits and there is rounding on the final adder stage only.

Run them in Simulink then compare the timings and hardware costs from the place and route reports and the timing reports.

(i) FIR Filter Using SRL16s Delay Lines

\filter\fir_srl16\fir_srl16.mdl :

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(ii) FIR Filter Using FF Delay Lines

\filter\fir_ff\fir_ff.mdl :

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(iii) FIR Filter Using Distributed Arithmetic Core

\filter\fir_ff_da\fir_ff_da.mdl :

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(iv) FIR Filter Using Block Multipliers

\filter\fir_block\fir_block.mdl :

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(v) FIR Filter Serial MAC and One Block Multiplier

\filter\fir_MAC\fir_MAC.mdl :

Report	Result	Values
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

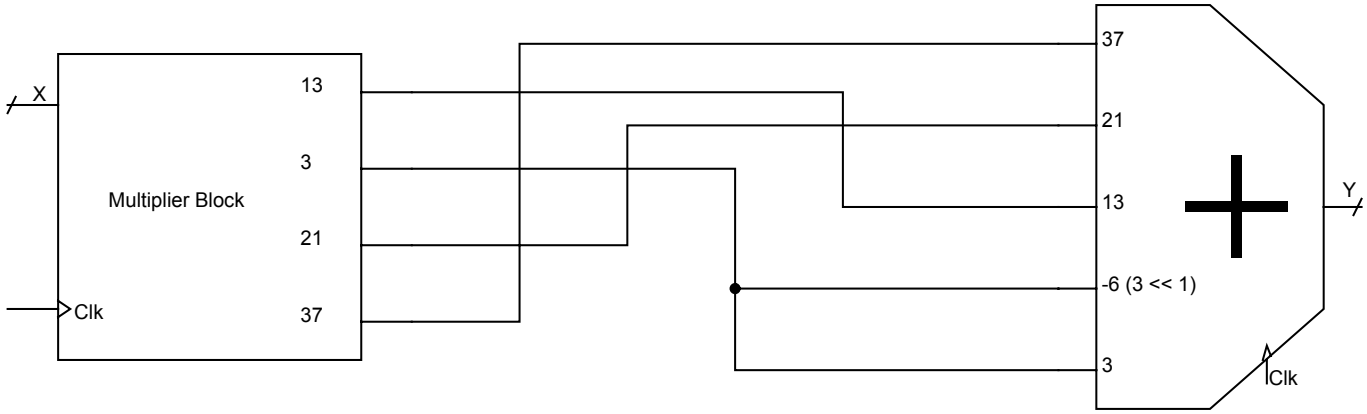
For the systems, view the floorplan to see the 4 block multipliers actually being used.

---

6 FIR Digital Filter by Multiplier Block Synthesis

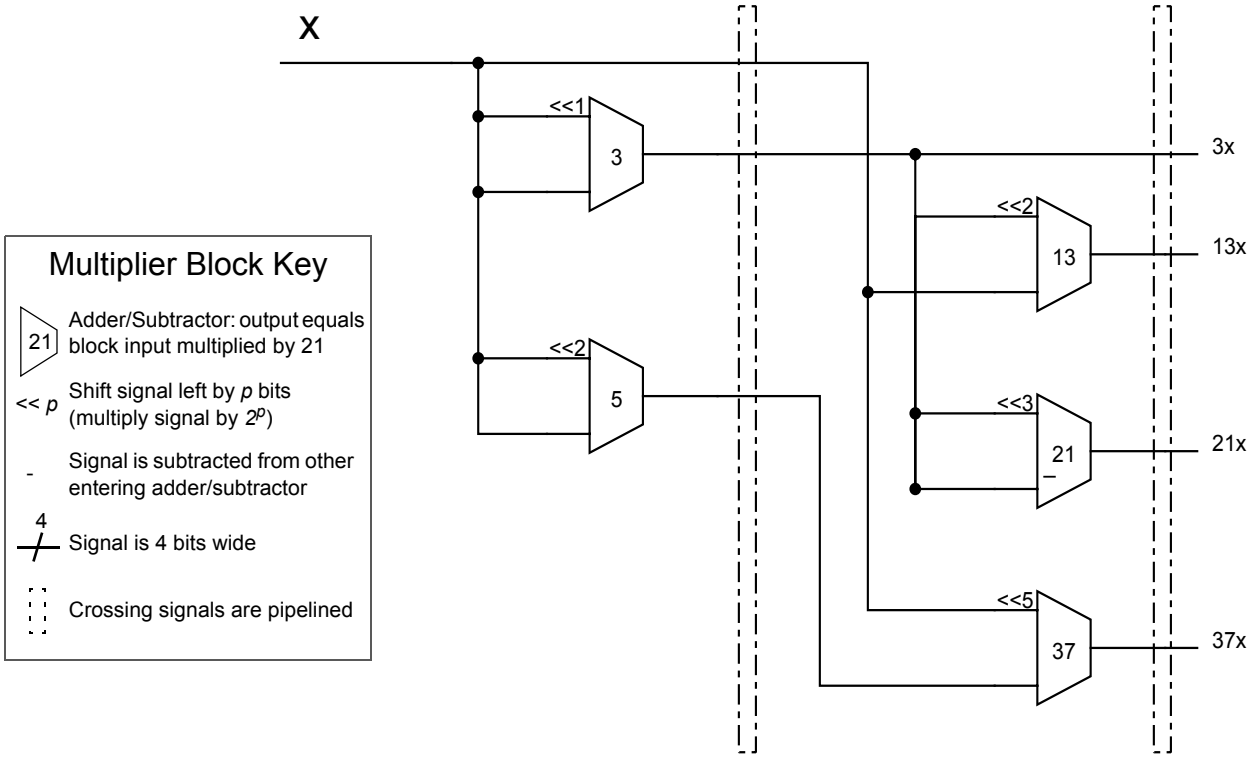
In this exercise we will look at how multiplication is performed for the full-parallel, fixed coefficient Transposed FIR filter with multiplier block. Recall from the class notes that the block generates multiplication products of the input

samples that are fed to the summation chain as shown below.

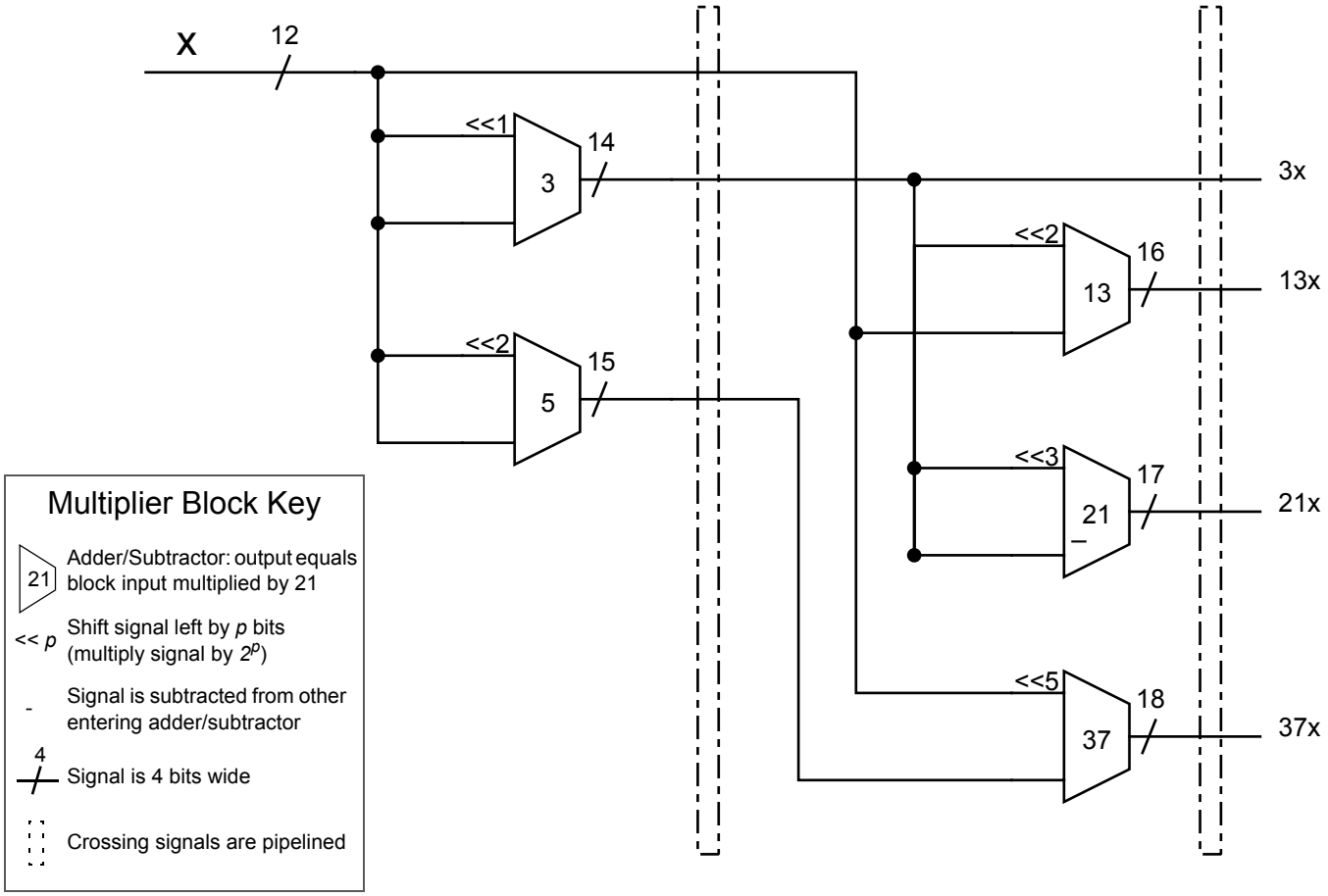


6.1 Inside the Multiplier Block

The Multiplier Block performs multiplication using adds/subtracts and shifts. The example block below multiplies the input x by 3, 13, 21 and 37 in two clock cycles. For example, the output of the "3" adder is the input left shifted once added to the input which is $2x + x = 3x$ as required. Similarly, the "21" adder output is $8(3x) - 3x = 24x - 3x = 21x$. Note that in this case the bottom input is subtracted from the top input. The term "adder" is used interchangeably to describe elements that add and elements that subtract one input from the other. This is because both element types have the same FPGA hardware cost and are almost identical in their implementation.



If we consider a 12-bit input signal by way of an example, the following diagram shows the same Multiplier Block with all bit-widths included:



Due to the filter coefficients being fixed, signal bit-widths can be calculated down to the last bit to be as small as possible to minimise hardware consumption while maintaining full arithmetic precision. The following bit-width analysis uses the "3" adder as an example.

If the input is 12-bit, the worst case inputs are -2048 (-2^{11}) and 2047 $(2^{11}-1)$. Applying -2048 to the multiplier block input gives an output value from the "3" adder of $2(-2048)+(-2048) = -6144$ (i.e. $3(-2048)$). This is represented as 10100000000000 in two's complement which is 14-bits.

Applying 2047 gives $3(2047) = 6141$ from the "3" adder. This is 01011111111101in two's complement which is again 14-bits.

Using logarithms, the required output bit-width (o) of any adder in the multiplier block can be calculated using the multiplication value the adder represents (m) and the multiplier block input width (i):

$$o = ceil(\log_2(m)) + i$$

For example, the "3" adder would be $ceil(\log_2(3)) + 12 = 2 + 12 = 14$ bits

For the "21" adder, $ceil(\log_2(21)) + 12 = 5 + 12 = 17$ bits

---

Exercise 6.1 Multiplier Block Synthesis

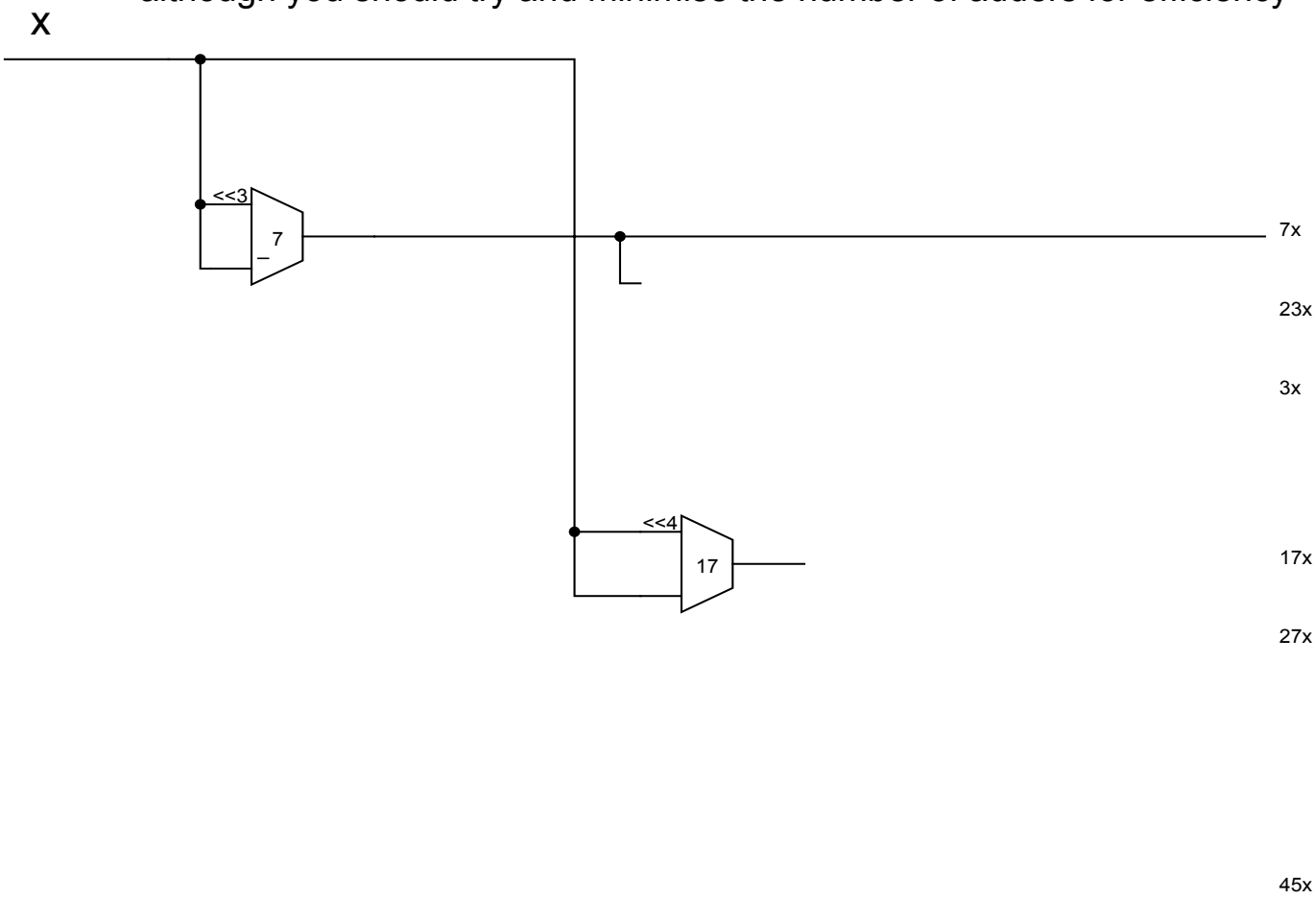
Complete the skeleton Multiplier Block below to multiply the input by 7, 23, 3, 17, 27 and 45. If you have time, calculate all signal bit-widths. Products 7 and 17 have been completed for you. Bear the following points in mind:

(a) Draw in additional 2-input adder blocks that either sum both inputs or subtract one from the other

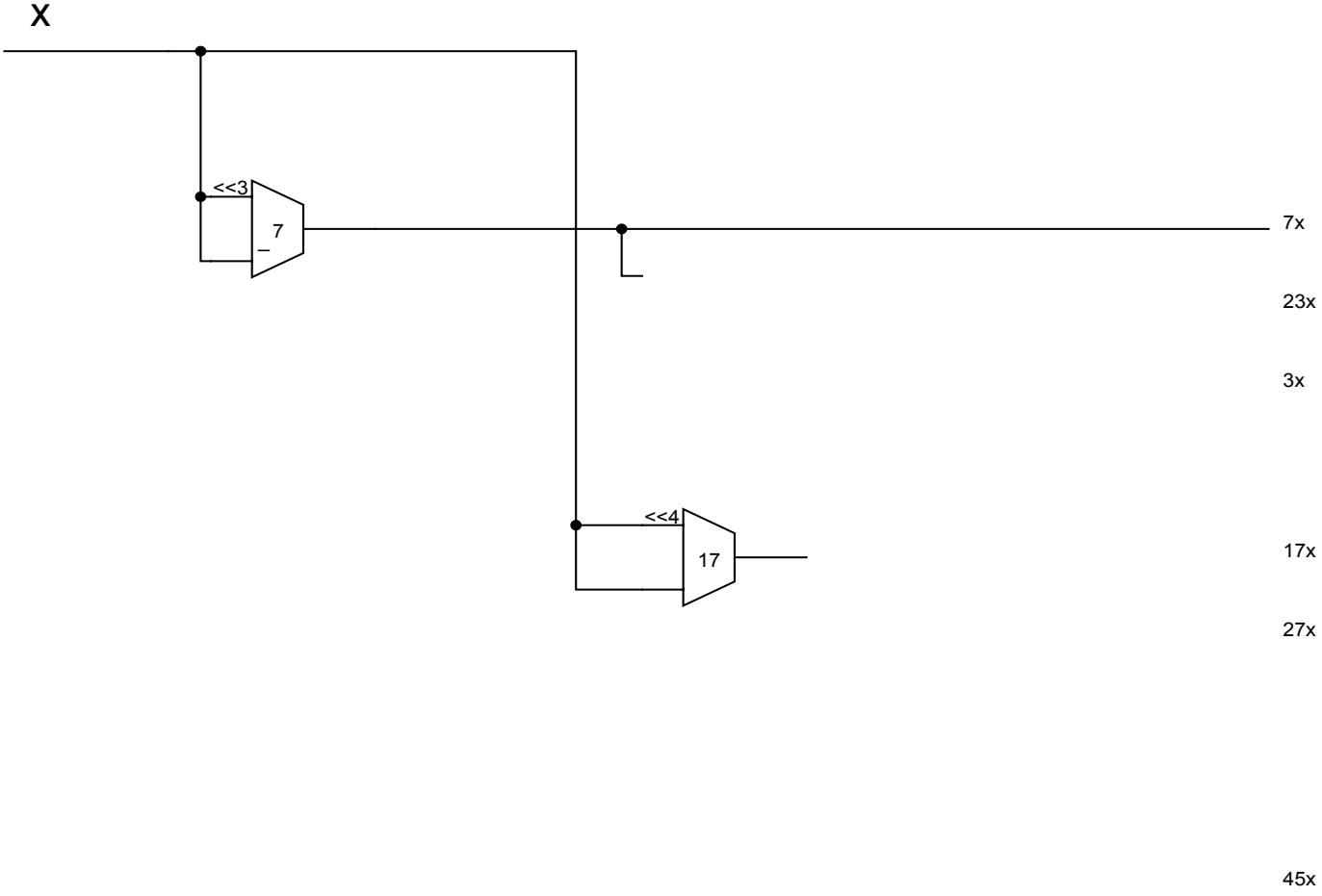
(b) One of the adder inputs can be left-shifted to multiply the signal by a power of two. Note that there is no point left-shifting both adder inputs as this will give an even number! (recall from the notes that multiplier blocks need only generate +ve, odd numbers)

(c) Use the multiplier block input signal and adder outputs to feed other adders (there is a clue hanging from the output of the "7" adder)

(d) You can create adders that are only required internally to feed others although you should try and minimise the number of adders for efficiency



An additional (identical) skeleton is given below for restarts/entering the answer.



7 Adaptive Filtering

In this section we will implement some simple on adaptive signal processing. In particular, issues related to implementation in FPGAs are considered. In the first exercises we will design a fully parallel adaptive FIR filter. In particular we will take note of the feedback that is present and how this means that there is a large critical path.

Exercise 7.1 Standard Parallel Adaptive LMS Filter

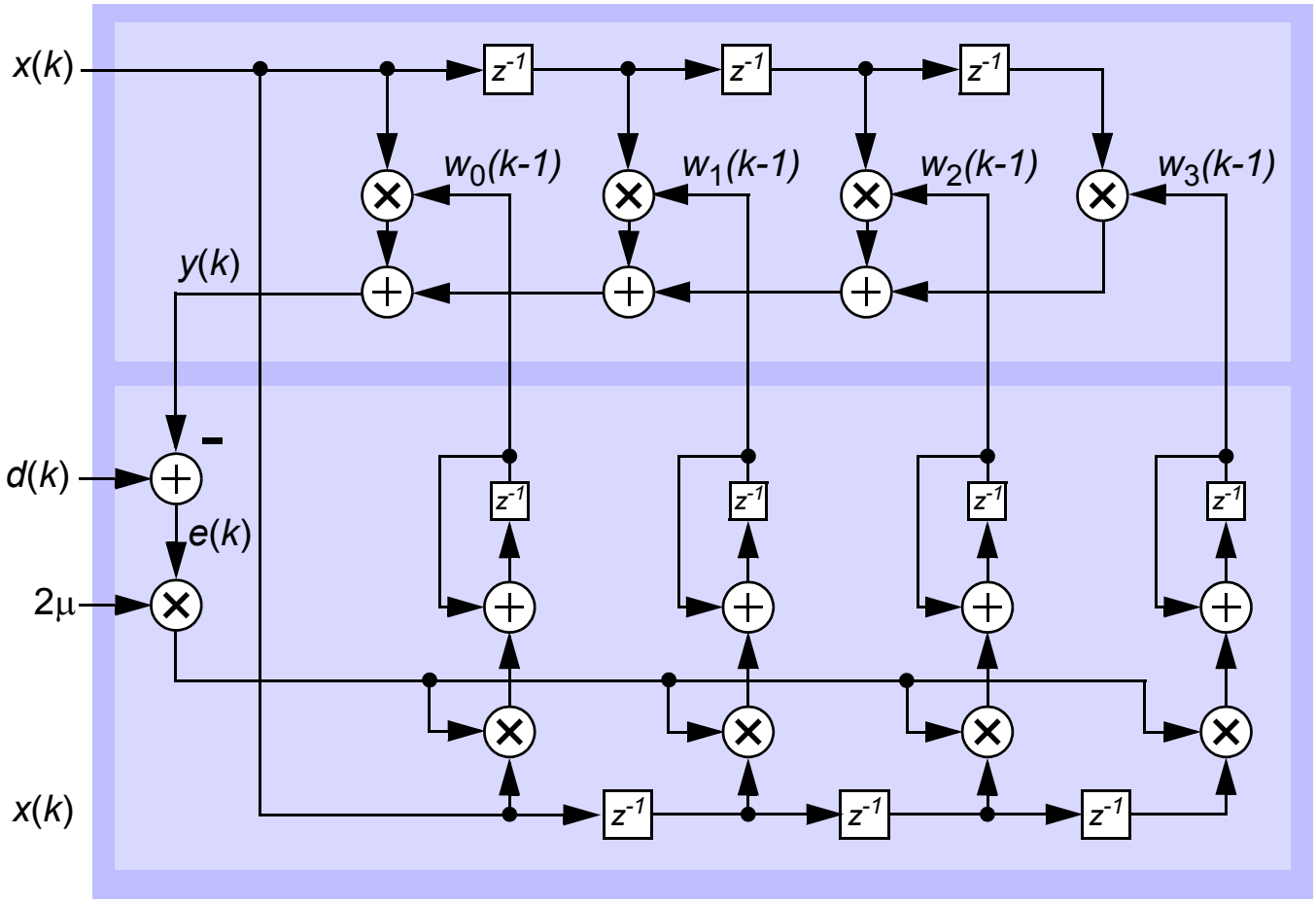
Open the system:

\adaptive\lms1\lms1.mdl

This system implements the update equation for the complete adaptive structure

$$\begin{bmatrix} w_0(k) \\ w_1(k) \\ w_2(k) \\ w_3(k) \end{bmatrix} = \begin{bmatrix} w_0(k-1) \\ w_1(k-1) \\ w_2(k-1) \\ w_3(k-1) \end{bmatrix} + 2\mu e(k) \begin{bmatrix} x(k) \\ x(k-1) \\ x(k-2) \\ x(k-3) \end{bmatrix}$$

The implementation maps the structure shown below



(a) Run the simulation and confirm the filter weights converge to the desired solution for this system identification.

(b) Change the set of weights in the unknown system and run the simulation. Does the filter still converge?

(c) Reduce the step size by a factor of 10 and run the simulation again, what can you observe? Do the weights converge? You may want to increase the number of samples in this simulation.

(d) In the above LMS signal flow graph what is the critical path?

(e) Using System Generator and again targetting the XC2V40 with its 4 multipliers, synthesise this to an FPGA design using the usual procedure to the ISE tools.

(f) If you synthesized successfully, complete the table below.

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(g) Save the Simulink file to a new directory **`lms2`**, and filename of **`lms2.mdl`**. Use the System Generator tools again, but this time target the larger device, the XCVP30 from the VIrtex II Pro family (and available on the board used in this course). Again, complete the table below:

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Exercise 7.2 Non canonical LMS implementation

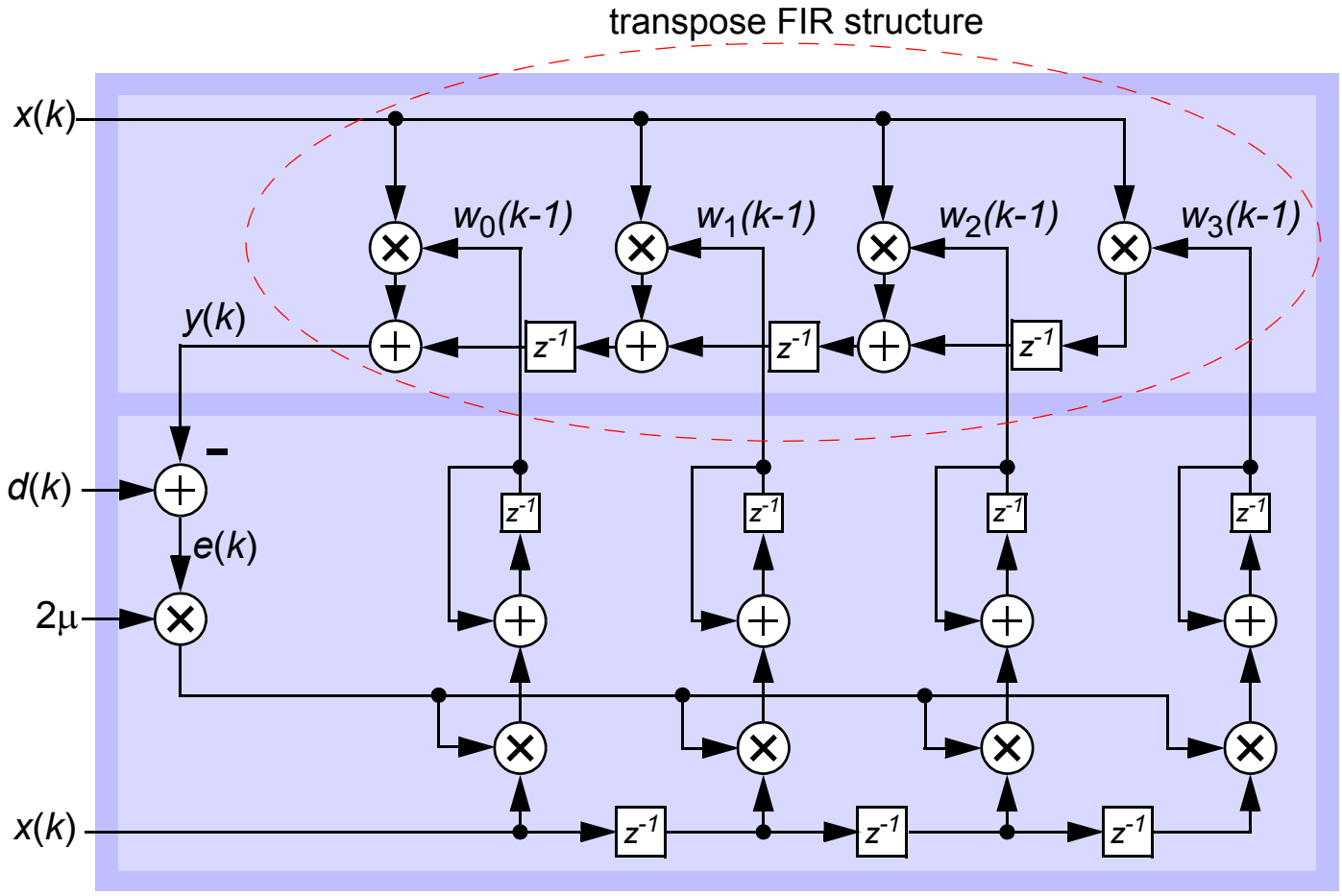
Open the system:

`\adaptive\LMS_transpose\LMS_transpose.mdl`

This system implements the non canonical LMS structure. Remember that the reason for using the transpose FIR is because it presented some advantages when implemented in FPGAs in terms of critical path. However most importantly note that the integrity of the algorithm has been slightly changed and is ***NOT*** identical to the standard LMS (in fact it is sub-optimal).

In this example we present an LMS implementation in which we have introduced the transpose FIR structure instead of the canonical. The resulting

implementation of the LMS is called the non-canonical LMS and is shown in the figure below.



(a) Check the configuration in the provided System Generator file and confirm it implements the structure shown above.

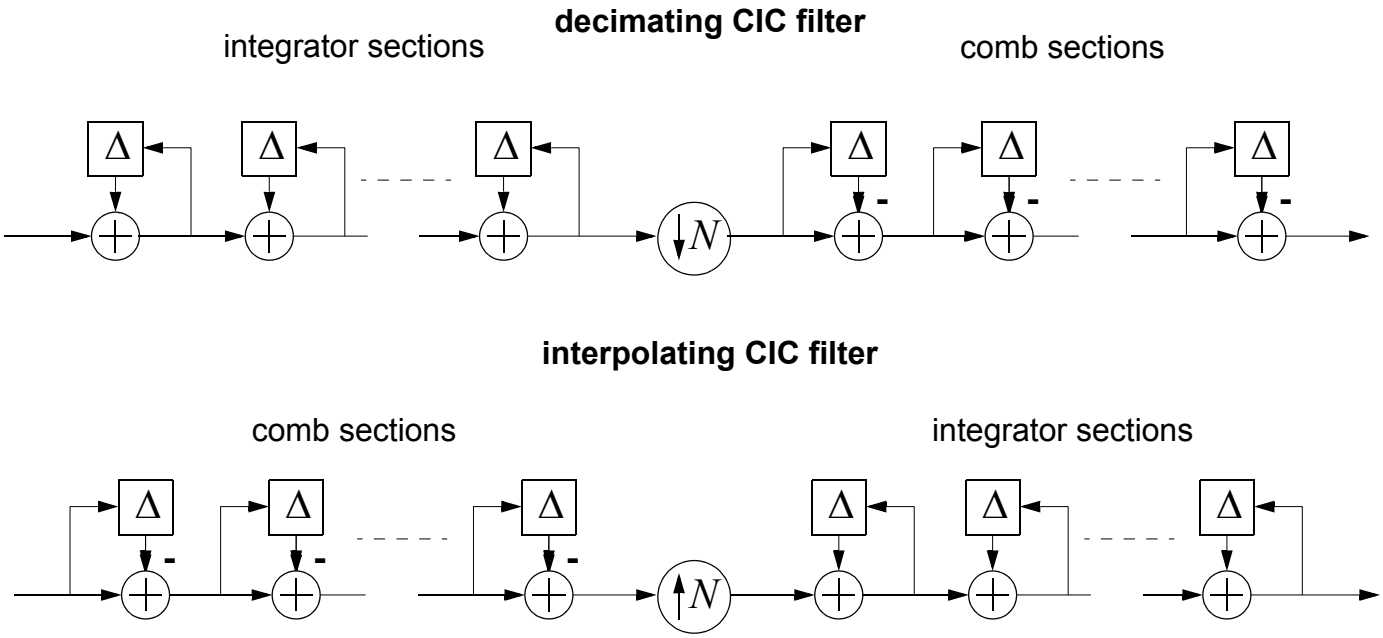
(b) Run the simulation and confirm the filter weights converge to the desired solution.

---

8 Low Pass Cascaded Integrator-Comb (CIC) Filters

Cascaded integrator-comb filters provide an efficient way to perform resampling in a multi-rate DSP system. As suggested by the name, these filters are constructed using integrators and comb filters. The diagram below shows the two most common CIC filter configurations. In the first configuration the integrator sections preceed the comb sections and the filter performs decimation. In the second configuration the comb sections preceed the integrator sections

and the filter performs interpolation.



Δ = 1 sample delay

In the sequence of examples which follows a number of Cascaded Integrator-Comb (CIC) filters are constructed using fixedpoint arithmetic.

8.1 CIC filters

Exercise 8.1 Moving Average Filter

Before considering a fixed point implementation let us explain the basics of a moving average filter.

Open the system:

\CIC\CIC_01\CIC_01.mdl

This system applies an impulse input to an 8-tap moving averaging filter.

(a) Inspect the parameters of the filter.

(b) Run the system and view the impulse response.

(c) Increase the number of samples to 1024 and analyse the FFT of this impulse response.

Exercise 8.2 Cascaded Integrator Comb vs Moving Average

Open the system:

\CIC\CIC_02\CIC_02.mdl

This example shows how a Cascaded Integrator-Comb filter is equivalent to a non-recursive moving average filter.

(a) Inspect the block parameters, run the system and, observe that the CIC frequency response is equivalent to that of the moving average filter.
(b) Swap the order of the integrator and comb sections. Does the system still have the same response?

Exercise 8.3 Multi-Stage CIC Filter

Open the system:

\CIC\CIC_03\CIC_03.mdl

This system consists of three CIC "stages" where each stage consists of an integrator-comb pair. The impulse response at the output of each stage is analysed. Note that each CIC stage has a dc gain of 8. For this reason compensation gains have been used to normalise the impulse response measurements to have a dc gain of 1 (0dB).

(a) Run the system and view the frequency response.
(b) Observe the difference between the impulse response and the frequency at each stage. How does the anti-alias performance compare with an increase in the number of sections?

Exercise 8.4 Fixedpoint CIC Filter

Open the system:

\CIC\CIC_04\CIC_04.mdl

This example includes a parallel fixed point implementation of the CIC filter. The 3-stage CIC system from the previous example is shown along with a parallel fixed point version which uses 19 bit words. The input signal has been swapped for a noisy sine wave which we intend to "clean up" using the CIC filter. Note how the integrator and comb sections are implemented explicitly using adds, subtracts and delays.

(a) Inspect the parameters of the fixed point blocks.
(b) Run the system and compare the fixed point and floating point outputs.

Exercise 8.5 Interpolation using a CIC Filter

Open the system:

\CIC\CIC_05\CIC_05.mdl

This example shows a CIC being used to implement an upsampling / interpolation filter. A sinusoidal input signal is expanded by a factor of 8 and then filtered using the CIC to attenuate the spectral images.

(a) Run the system and observe the signals at the output of each stage.
(b) Compare the spectrum of the output at each stage

Exercise 8.6 Cascaded Integrator Comb Filter - Fixed Point Upsampler

Open the system:

\CIC\CIC_06\CIC_06.mdl

In this example the interpolating CIC filter is implemented with the resampler in two different positions. In the upper CIC filter the upsampler appears prior to filtering. In the lower CIC filter the three comb sections have been grouped together and the three integrators have been grouped together and the resampler appears in between the two groups.

(a) Run the system and verify that the two filters produce the same output signal.
(b) What do you notice about the delays used in the comb sections of the lower filter? Can you explain this?

(c) What do you think the advantages of the lower implementation are?

Exercise 8.7 Cascaded Integrator Comb Filter - Fixed Point Decimator

Open the system:

\CIC\CIC_07\CIC_07.mdl

In this exercise, we now build a CIC which decimates by a factor of 16 and uses 4 sections. This filter is designed to decimate from a 10MHz sampling rate down to a sampling rate of 625kHz for a pass band of 78.125kHz

(a) Observe the truncate and shift block before the first filter stage. This compensates for the non zero dB gain of the CIC filter

(b) Run the system and observe the frequency response. In the passband there is clearly a “droop” present.

(c) Use System Generator and the ISE tools, take this implementation to an FPGA design using the XCVP30 from the VIrtex II Pro family. Note that for obvious reasons there are no block multipliers to be used in the implementation.

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Exercise 8.8 Decimating CIC Filter - Maximum Bit widths

In this example the maximum bit width is demonstrated and the filter is shown to overflow at certain stages.

Open and the system:

\CIC\CIC_08\CIC_08.mdl

The maximum bit width at each integrator and comb stage for a decimating CIC filter with decimation factor *R*, delays per comb *M* and number of combs *N* is given by:

$$B_{OUT} = \lceil N \log_2 R + B_{IN} \rceil$$

where $\lceil \ \rceil$ is the ceiling operator which rounds up to the nearest integer. In this case, for *R*=16, *N*=4, B_{IN}= 16, B_{OUT} = 32.

(a) Run the system and observe the output.

(b) In the duplicate system, try increasing the number of bits in a component, run the system and view the results. Try also decreasing the wordlength in a component and again view the difference between the two signals. Observe the results for different wordlengths at different points in the filter cascade.

(c) Run the system again and view the results for the components in system 1 (this system should not have been edited). Note that there are overflows in some of the filter stages.

9 Direct Digital Downconversion

In this section we shall consider some of the hardware components required to implement a digital down converter. A downconverter brings an RF (radio frequency) signal down to an intermediate or baseband frequency. Ultimately downconversion consists of shifting the signal of interest to a lower frequency, removing unwanted signal components and sampling the desired information signal at a "reasonable" sampling rate.

This section is split into a number of exercises. Firstly we shall build a downconverter using standard Simulink blocks and using double precision arithmetic. We shall then consider how to optimise each component for implementation on an FPGA.

9.1 Downconversion using DSP

Figure 9.1 shows two common downconverter architectures. The first converts to an intermediate frequency (IF) and requires a further stage of demodulation to bring down to baseband. An additional stage is then required to demodulate the signal to baseband. The second technique is known as direct downconversion. In this case the signal is demodulated directly from RF to baseband.

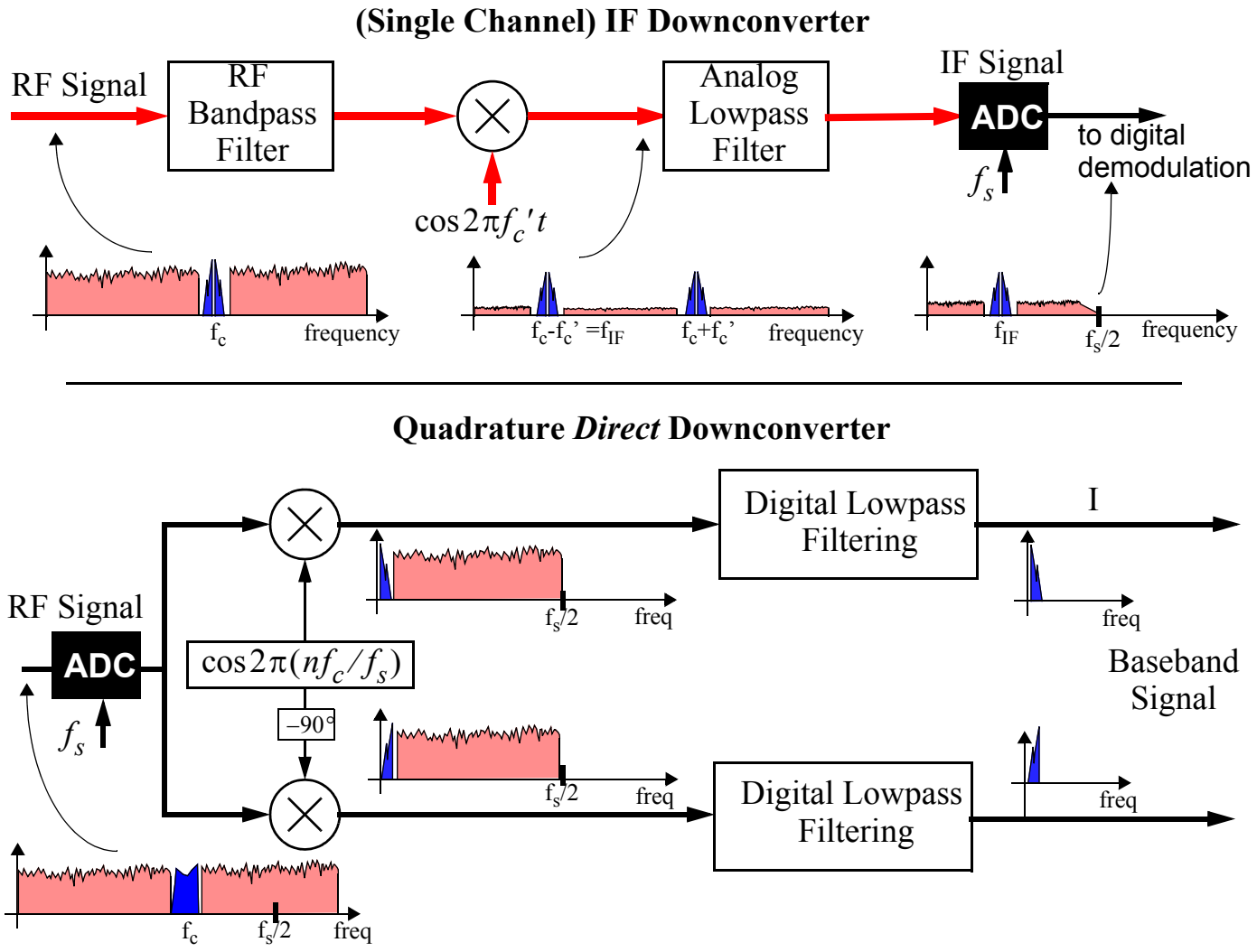


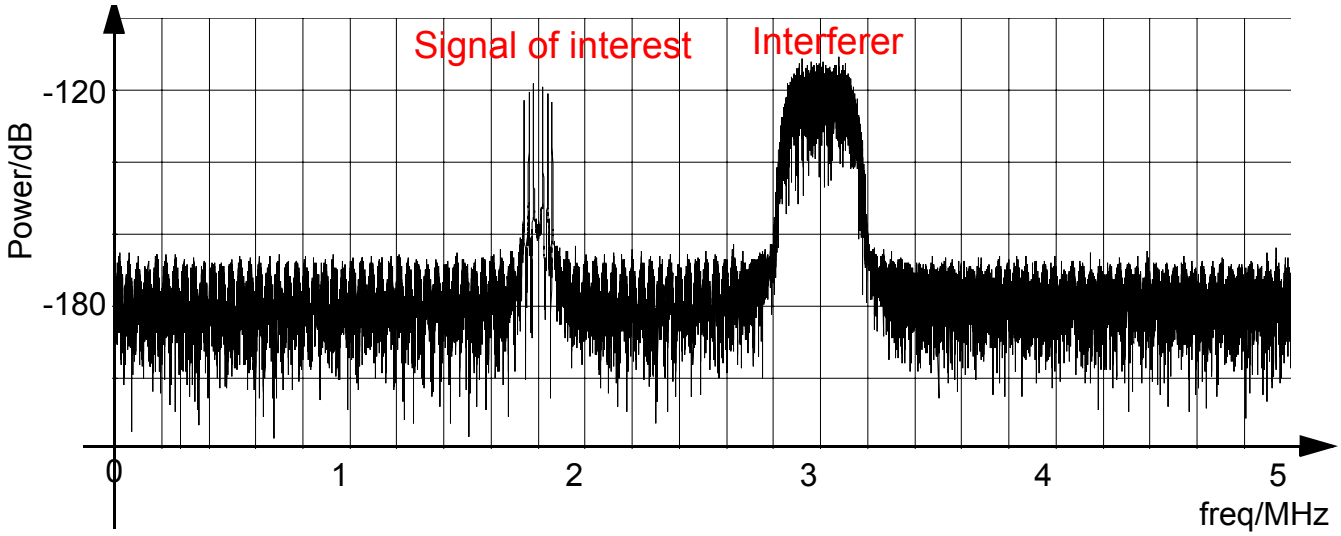
Figure 9.1: Demodulating or downconverting a signal to baseband. For diagram simplicity the necessary anti-alias filters are not shown.

Exercise 9.1 Building a Downconverter

Open the System

\Downconversion\mixer\RF_signal.mdl

The source signal in this example is stored on file and has been sampled at $f_s = 10\text{MHz}$. The signal of interest spectrum was three sine waves modulated onto a cosine carrier at 1.8MHz. In addition to noise across the entire 5MHz spectrum, there is a specific interference signal centered around 3MHz. The spectrum is shown below:



Run the system and view the received signal. As shown above you should see two distinct components; the first centred on 1.8MHz comprises a number of tones. The second is an interfering signal centred on 3 MHz.

In order to gain an understanding of how a downconverter works we shall build a system using Xilinx System Generator blocks which will demodulate the 1.8MHz component to baseband. The following steps should help you to achieve this.

(a) We can use a multiplier to mix down the signal to baseband using an 1.8MHz cosine. Implement this or open the system in:

\Downconversion\mixer\mixer.mdl

Can you account (by some *pseudo*-trigonometry) for the frequency components which appear centered at 1.2MHz, 3.6MHz and 4.8MHz?

What would happen if you mixed down with a sine-wave - Why would this be?

(b) Design a lowpass filter which retains the 1.8MHz component however removes the 3MHz signal by at least 60dB. Filters can be designed using the Simulink **`Xilinx Blockset>Tools>FDATool`**. A designed filter can be found in:

\Downconversion\mixer\lpfilter.mdl

(c) Assuming the original signal can be considered to have a bandwidth of 1MHz, place a suitable downsampling at the output of the filter to decimate by 50. A designed filter can be found in:

\Downconversion\mixer\decimate.mdl

(d) Confirm that the downconvert is functioning correctly, using System Generator the take this final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family - does you

design actually synthesise OK?

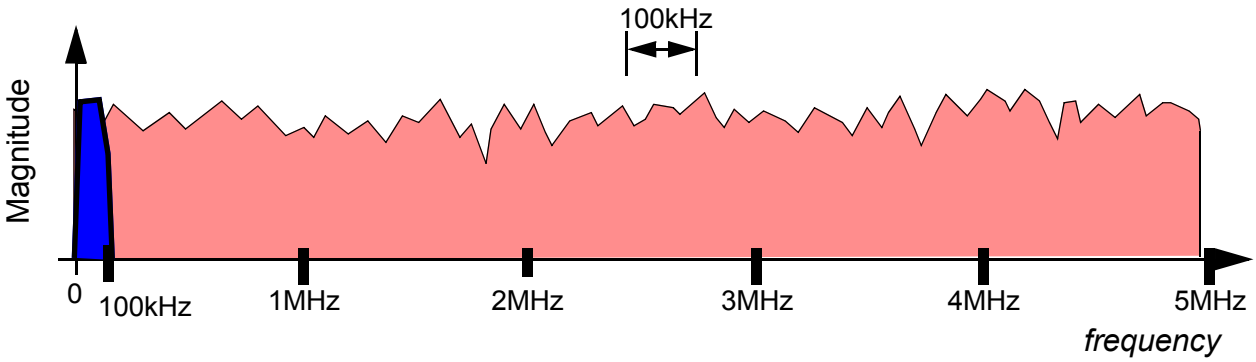
Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Any comments on the final design?

9.2 CICs for Downconversion

Exercise 9.2 Designing a Downconverter using a CIC filter

In this example we will build another downconverter this time using a CIC filter. The target signal spectrum is in the range 0-100kHz as shown below; the sampling rate is 10MHz:

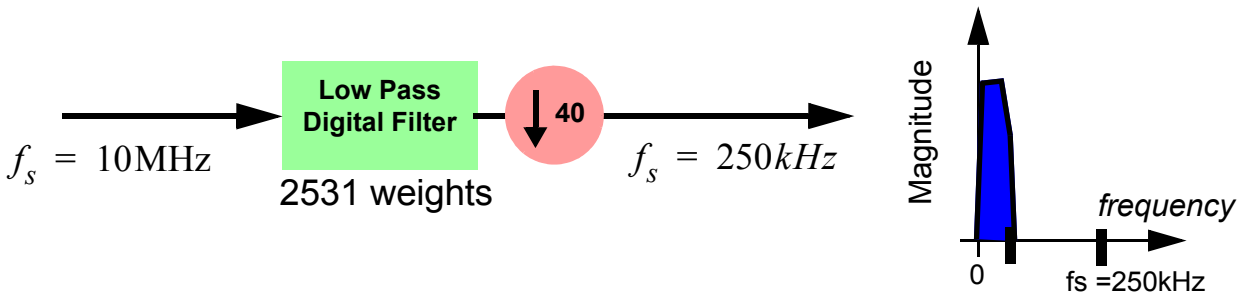


The final sampling rate required is 250kHz and therefore at the output of the filter we can downsample by 40.

Open the system:

\downconversion\filter_10MHz\filter_10MHz.mdl

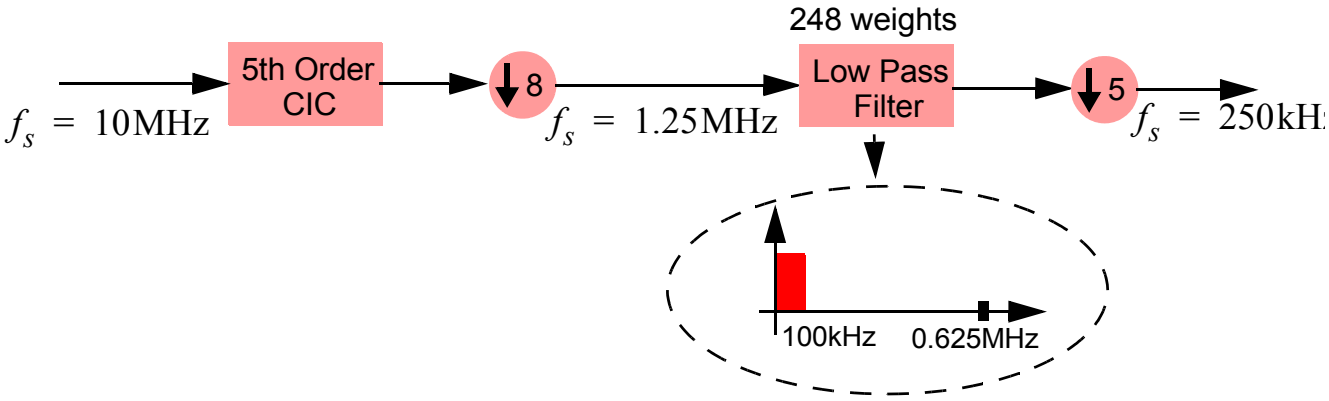
(a) View the parameters of the low pass filter used in the system.



(b) Open the system:

\downconversion\cic_10MHz\cic_10MHz.mdl

View the various filters in the lower line which uses a cascade of CIC filters and a final stage FIR low pass filter:



(c) Run both systems and confirm that the outputs are "very similar" (which they shuld be).

The outputs are not identical - why should this be? (refer to the class notes to recall the "droop" of the CIC.

(d) Using System Generator take the CIC design to FPGA implementation and complete the table below. (Note that for the fullband filter it will be very expensive and likely will not synthesise - try and see).....

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

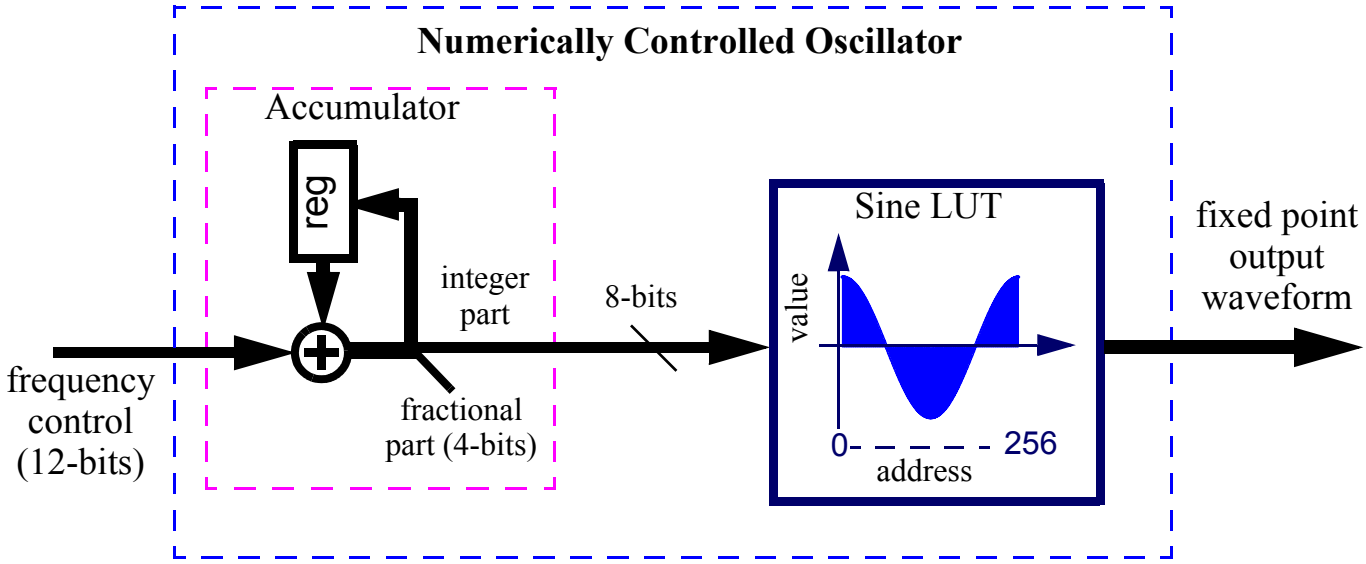
The systems just considered have very briefly introduced the concept of downconversion. In any practical scenario setting the parameters of the oscillators and filters is only the first step in realising a downconverter. For example how do we generate a cosine waveform of arbitrary frequency and phase on an FPGA (the answer is coming up in the next section)? What about the filters? These have large numbers of weights and consequently lots of multiplications and additions are required. There are many optimisation techniques which can be used to minimise the cost in terms of device usage. We shall later consider some examples which explain how the typical components of a downconverter can be designed and optimised to run on an FPGA device.

10 Numerically Controlled Oscillators

For the downconverters discussed previously, an oscillator is required for mixing to basedband. These oscillators can be implemented in a number of ways. In this section we will implement the most comment technique which uses a look-up table. We will also look at a simple technique to generate using marginally stable IIR filter.

10.1 Look-up Table Technique

A technique which is commonly used to generate a numerically controlled oscillator is to use a sine lookup table (LUT). A sine waveform can be generated by stepping through the entries of the LUT. When the end of the table is reached we simply wrap around to the first entry thereby creating a periodic waveform. This scheme is depicted in the diagram below. The ramp function is generated by a fixed point accumulator. The input to the accumulator is added to its value on each step. The magnitude of this input value therefore controls the frequency of the oscillator.



Exercise 10.1 Generating a Ramp Function using an Accumulator

Open the system

\oscillator\NCO_01\NCO_01.mdl

This system generates a ramp function using an accumulator as described above. The system sampling rate is f_s = 100kHz.

(a) Examine the parameters of the various blocks and determine how the system works.

(b) Run the system and view the resulting waveform at the output.

(c) Derive an equation which determines the frequency of the resulting sawtooth waveform for a given step size. You may ignore numerical inaccuracy at this stage.

(d) Set the step size such that the sawtooth waveform has a frequency of 1000Hz.

Exercise 10.2 Effect of Accumulator Precision

Open the system

\oscillator\NCO_02\NCO_02.mdl

This system contains two fixed point ramp (sawtooth) generators. They are both currently configured so that the accumulator has a precision of 8 integer bits and 2 fractional bits.

(a) Run the system and view the outputs.

(b) What do you think would be the effect of changing the precision (i.e. number of fractional bits) of one of the ramp generators?

(c) Set one of the function generators to have 6 fractional bits rather than 2. This only requires you to change the gateway block. The adder is configured to use the same precision as its input.

(d) Run the system and view the results. Is this what you expected?

Exercise 10.3 Generating a Sine Wave using a Lookup Table

Open the System

\oscillator\NCO_03\NCO_03.mdl

This system builds on the previous ones by introducing a sine lookup table. The fractional part of the ramp signal is disregarded and the integer part is used to address the look-up table stored in memory.

(a) Study the parameters of the converter block and the memory with the sine wave.

(b) Run the system and view the output. Note that the sawtooth wave and sine wave have the same frequency.

(c) Observe the frequency response of the sinusoid which is generated by the NCO. Measure the approximate difference in power between the dominant tone and the harmonic tones.

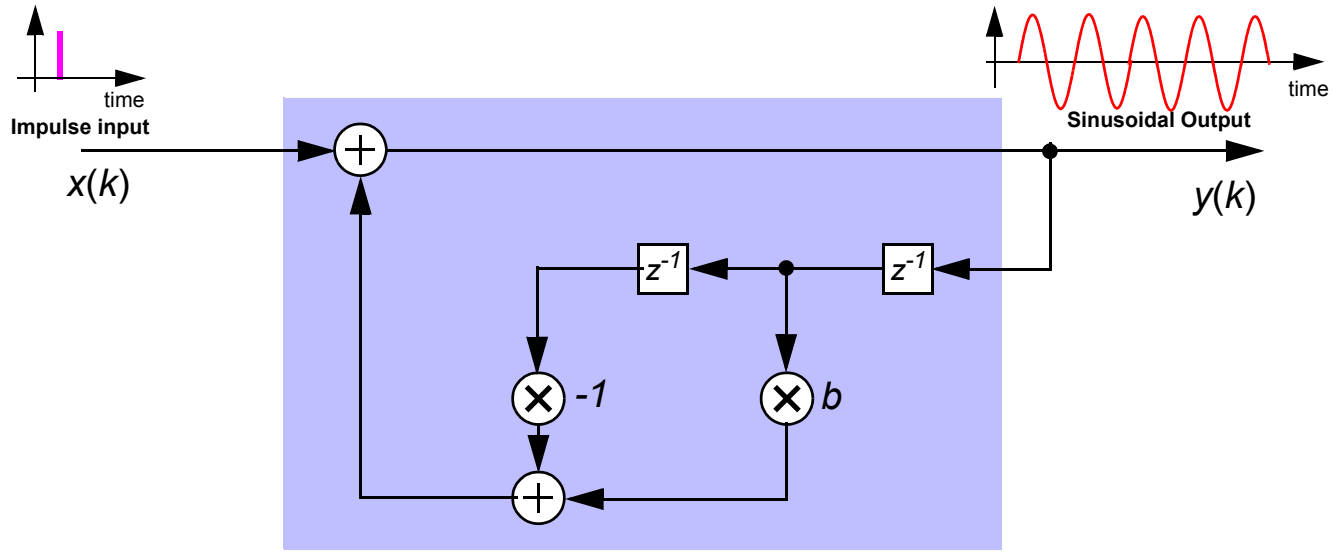
(d) In the design, change the precision of the lookup table entries to 16 bits (i.e. 16 bit register with 15 fractional bits). Run the system and view the new results. Now measure the difference between the dominant tone and the harmonics. How does this compare to the measurement in (c)?

(e) Why do you think the quantisation noise appears as harmonics in this example?

(f) Leave the lookup table precision at 16 bits and change the frequency of the NCO to 1kHz by setting the step size to 2.56. Run the system again. What has happened to the power of the harmonics. Can you explain why you obtain this result?

10.2 Sine Wave Generation Using IIR Filters

We can also generate sine waves using marginally stable sine waves. Consider the simple two weight IIR (all-pole) filter:



The output of this filter is:

$$y(k) = x(k) + by(k-1) - y(k-2) \tag{1}$$

Writing this in the z-domain gives:

$$Y(z) = X(z) + bz^{-1}Y(z) - z^{-2}Y(z) \tag{2}$$

The transfer function, $H(z)$, is therefore:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - bz^{-1} + z^{-2}} = \frac{1}{(1 - p_1 z^{-1})(1 - p_2 z^{-1})} = \frac{1}{1 - (p_1 + p_2)z^{-1} + p_1 p_2 z^{-2}} \tag{3}$$

where, p_1 and p_2 are the poles of the filter, and $b = p_1 + p_2$ and $p_1 p_2 = 1$. The poles of the filter, $p_{1,2}$ (where the notation $p_{1,2}$ means p_1 and p_2) can be

calculated from the quadratic formula as:

$$p_{1,2} = \frac{b \pm \sqrt{b^2 - 4}}{2} = \frac{b \pm j\sqrt{4 - b^2}}{2} \quad (4)$$

Given that b is a real value, then p_1 and p_2 are complex conjugates. Rewriting Eq. 4 in polar form gives:

$$p_{1,2} = e^{\pm j \tan^{-1} \frac{\sqrt{4 - b^2}}{b}} \quad (5)$$

Considering the denominator polynomial of Eq. 3, the magnitude of the complex conjugate values p_1 and p_2 are necessarily both 1, and the poles will lie on the unit circle. In terms of the frequency placement of the poles, noting that this is given by:

$$\left|p_{1,2}\right| = 1 = e^{\frac{\pm j 2\pi f}{f_s}} \quad (6)$$

(where $\left|e^{j\omega}\right| = 1$ for any ω) for a sampling frequency f_s, from Eq. 6 and Eq. 5 it follows that:

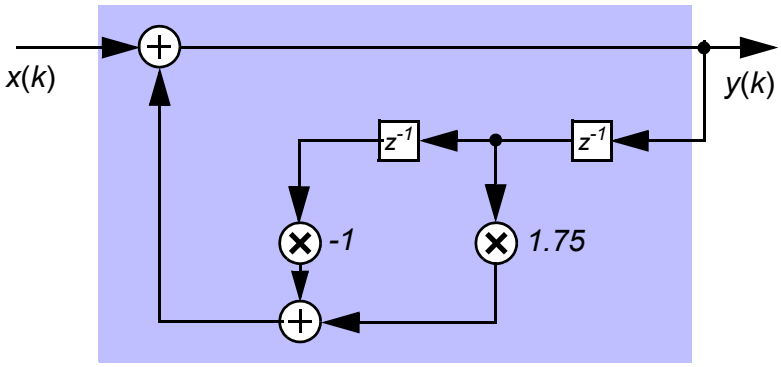
$$\frac{2\pi f}{f_s} = \tan^{-1} \frac{\sqrt{4 - b^2}}{b} \quad (7)$$

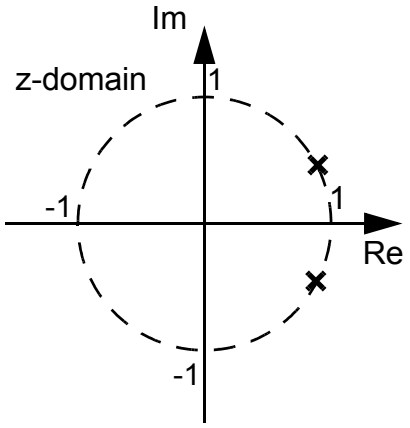
Exercise 10.4 Sine Wave Oscillators using IIR filters

Open the system

\oscillator\sine_wave_iir.mdl

In this example we have implemented an IIR filter sampling at f_s = 100MHz with *b = 1.75*





$$H(z) = \frac{1}{1 - 1.75z^{-1} + z^{-2}}$$

(a) Using the root-locus of facility of the Linear System block, confirm that the poles of the filter are as shown above.

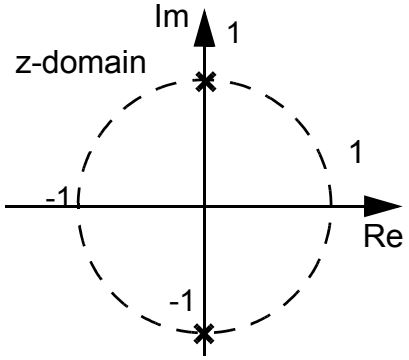
(b) Run the system observe the sine wave and the frequency spectrum of the IIR filters (implemented via a linear system block and also from discrete blocks) and confirm the frequency of the sine wave is 8.04MHz.

(c) From above the equation $(2\pi f)/f_s = \tan^{-1}(\sqrt{4 - b^2})/b$ confirm that this is correct for *b = 1.75*.

(d) Change the value to *b = 1.95* (***note*** that in the Linear System block you type in the coefficients of the z-polynomial therefore you will enter -1.95). Calculate the frequency of the output using the equation above. Confirm this is correct by running the simulation.

(e) If we set b = 0, then:

$$H(z) = \frac{1}{1 + z^{-2}}$$



what is the frequency of the output now? Run the simulation and observe the frequency of the signal. Observe the time domain signal (zoom in to see individual periods.)

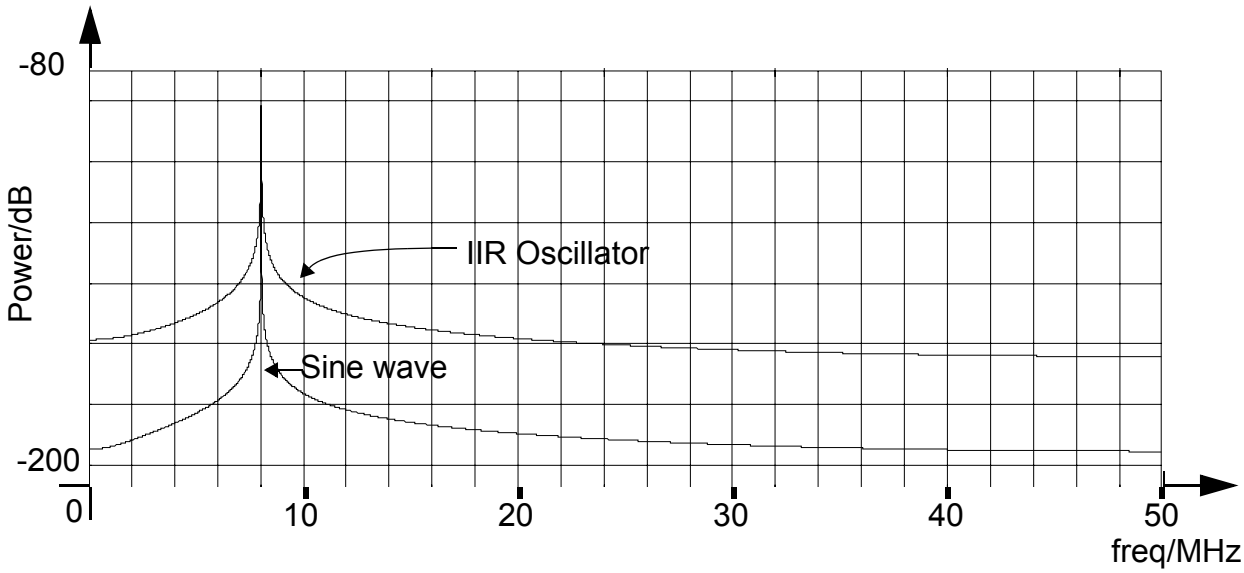
Exercise 10.5 **Sine Wave Oscillators using IIR filters**

Open the system

\oscillator\sine_wave_iir_spectrum.mdl

In this example we will observe the spectral purity of the generated sine waves. This system contains three components, (i) an IIR oscillator, (ii) a sine wave generator, and (iii) an IIR oscillator made from discrete blocks.

(a) Run the system and note that the overlay of the IIR oscillator and the sine wave indicate that the sine wave is “purer”:



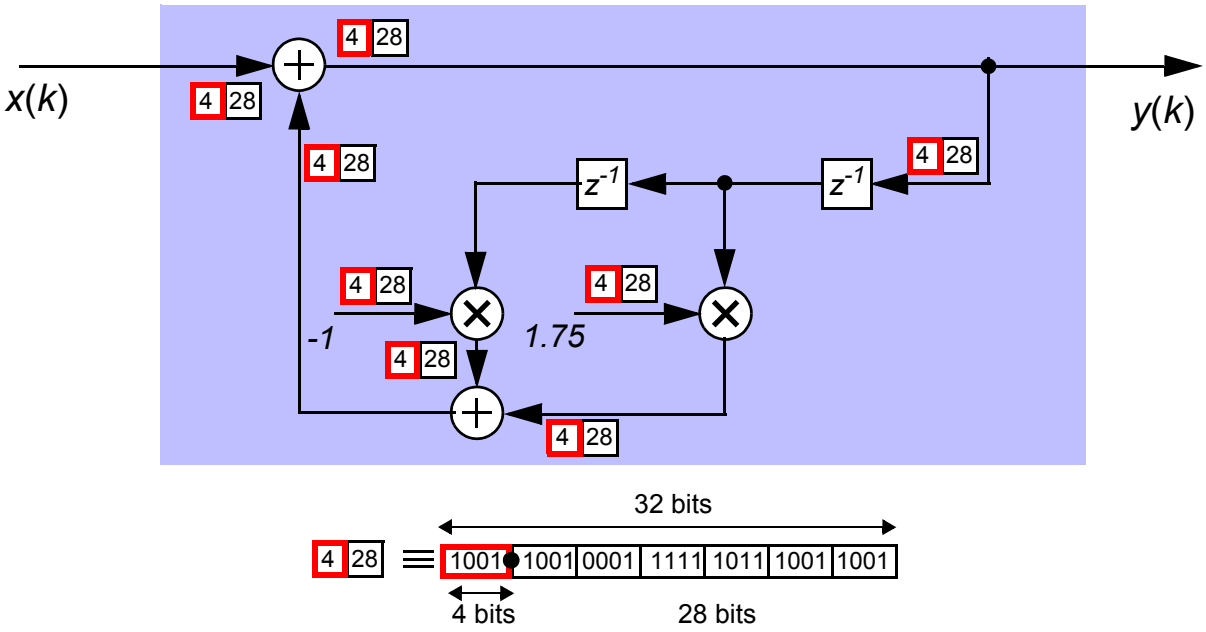
(b) Note the output of IIR oscillator has a spectrum that is “identical” to that of the linear system block IIR.

Exercise 10.6 Sine Wave Oscillators using fixed point IIR filters

Open the system

\oscillator\sine_wave_iir_32bit.mdl

In this example we have added a fixed point IIR filtering implementation where all adders and multipliers have been set to a wordlength of 4 integer bits and 28 fractional bits.

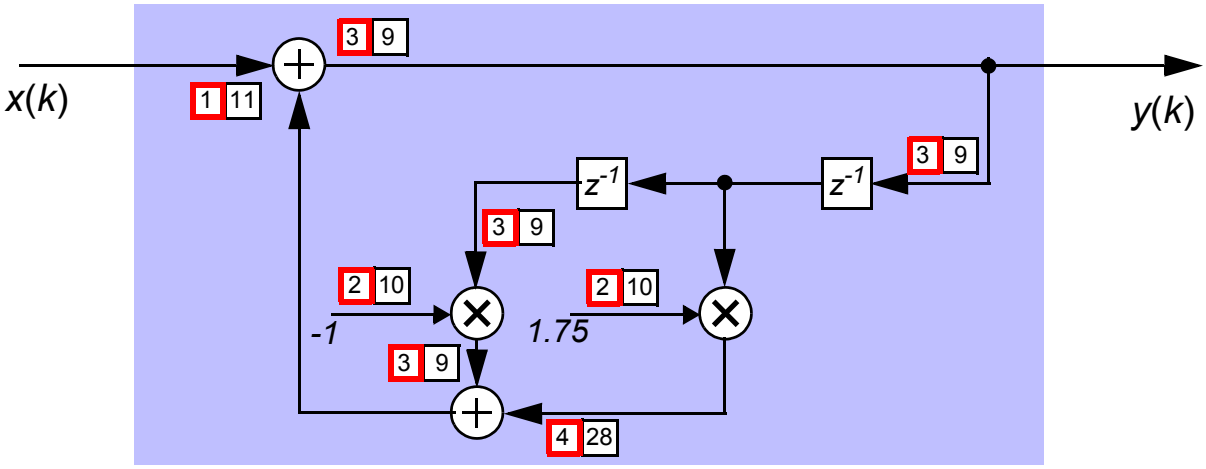


Exercise 10.7 Sine Wave Oscillators using 12 bit fixed point IIR filters

Open the system

\oscillator\sine_wave_iir_12bit\sine_wave_iir_12bit.mdl

In this example we have set the wordlengths of the various adders and multipliers to be only 12 bits and therefore more amenable to a lower cost FPGA implementation.



(a) Run the simulation and observe the frequency spectra of the floating point

and 12 bit implementations.

(b) If you view the frequency spectra you should notice that the two spectra are very similar but not identical. The 12 bit representation has slightly more noise components visible (i.e. a less pure sine wave).

(c) Note in the above figure and the Simulink system we have set the 12 bit resolution to formats of 1 11 or 3 9 at different places in the structure. Can you work out why this should be?

(d) Using System Generator the take the final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family.

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

(e) Revist the Simulink system and change the multipliers to use slice based multiplication rather then block multipliers and complete the cost and timing table below:Using System Generator the take the final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family.

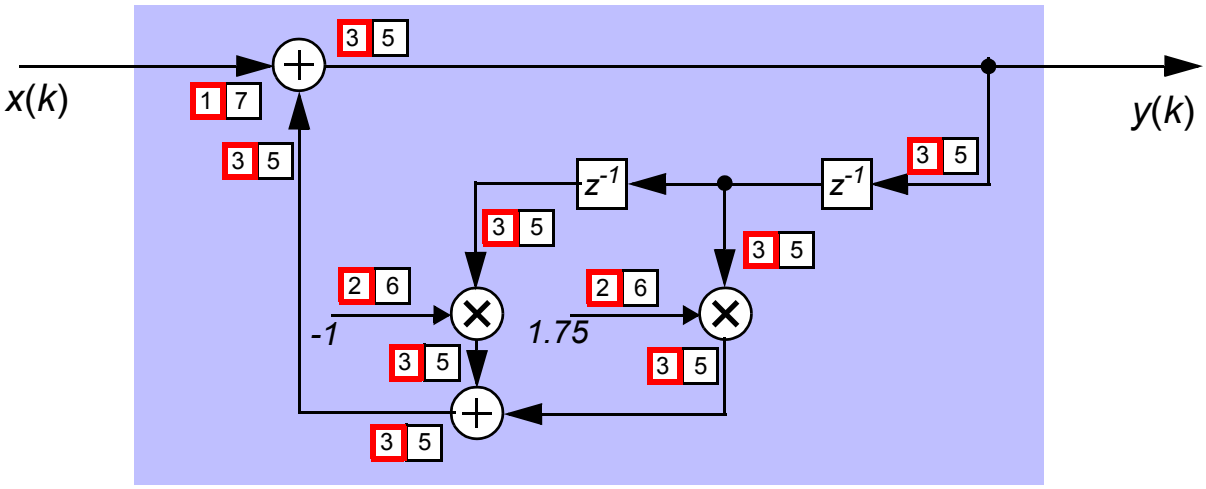
Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Exercise 10.8 Sine Wave Oscillators using 8 bit fixed point IIR filters

Open the system

`\oscillator\sine_wave_iir_8bit.mdl`

In this example we have further reduced the wordlength to be only 8 bits which, of course, means the system is even less FPGA cost than the 12 bit version! However we will now see that the quality of the sine wave is reduced.



(a) Run the simulation and observe the frequency spectra of the floating point and 8 bit implementations. The “sine wave” clearly has quite a high level of noise.

(b) What frequency does the oscillator actually produce?

Why would this be different from the expected value?

---

11 CORDIC - Vector Magnitude Calculations.

In this section we will design and verify a fixed point CORDIC system to compute the magnitude of a vector to a desired accuracy.

11.1 The Golden Reference Design

To allow the fixed point CORDIC design that we are going to build to be accurately tested, we need a reference system with which to compare it to. Hence, the first step is to build a system that we are sure generates a correct output. Once the two systems are built the output from the CORDIC system will be subtracted from the reference system to gauge how accurate it is.

Exercise 11.1 Golden Reference Design

Open and run the following system:

\CORDIC_Mag\GoldenReference.mdl

In this example we will see that floating point blocks have been used to implement a Direct approach. Using floating point blocks we are sure that the magnitude that is computed will have a high degree of accuracy.

(a) Change the input values of the Constant blocks and verify that the system is actually computing the magnitude correctly.

11.2 The Fixed Point CORDIC Design

The fixed point CORDIC system that we are going to build will compute the magnitude of a vector to an accuracy of 5 effective fractional bits. The first step is to consult the table below which predicts values of d_{eff} (the number of effective fractional bits) for systems with n iterations and b fractional bits in the data path.

	Predicted		
n / b	**7**	**8**	**9**
1	2.21	2.24	2.26
2	3.51	3.62	3.68
3	4.74	5.09	5.31
4	5.31	6.03	6.59
5	5.36	6.28	7.13
6	5.23	6.21	7.17

From the table it is clear that we could choose either $n = 4$, $b = 7$ or $n = 3$, $b = 8$ to obtain $d_{eff} = 5$. When faced with trading n for b, always choose the smallest n as this offers the most efficient design.

One final parameter for our design that must be discussed is the maximum magnitude $|v(0)|$ of the vector that will be rotated. The above table was generated with the assumption that the x and y inputs would be constrained to ± 0.5. This means that the maximum value that the magnitude could be is $\sqrt{0.5}$ and hence this is the value that was used to generate the table. Thus, the inputs to our design must be in the range ± 0.5. Now that all the parameters required to build our system are known, we can begin.

Exercise 11.2 **Set Up The Input Signals**

Starting with the system in the previous exercise we shall now begin to build the fixed point CORDIC system. The first step is to put down the input signals.

Reopen the system (same as previous exercise):

\CORDIC_Mag\GoldenReference.mdl

and save it with a new filename to a desired location. This new file will be used to build the CORDIC system.

At this stage the reference design is driven by Constant blocks which provide a source with a constant amplitude. However, we are now going to use Uniform Random Number blocks to drive both the reference design and the fixed point CORDIC design that we will shortly build.

(a) Drag in a Uniform Random Number block from the Simulink>Sources library. Open up the block parameter window and set Minimum = -0.5 and Maximum = 0.5. Close the block by clicking the OK button.

(b) We need two of these blocks so to duplicate the original, right click on it and select Copy and then paste the block into the workspace. A second Uniform Random Number block with the same parameters will appear. Do not connect the new input blocks to anything yet.

(c) Rename both blocks to 'x input' and 'y input' and then save the file and check it against:

\CORDIC_Mag\CORDIC_mag1.mdl

Exercise 11.3 Set The Fixed Point Format Of The Input

The Uniform Random Number blocks created will produce floating point data. Hence, we must now convert these signals into a desired fixed point format.

(a) Building on the previous system, drag in a Gateway In block from the Xilinx Blockset>Basic Elements library. Open the block parameters window and set Number of Bits = 9 and Binary Point = 8. Make sure the Output Data Type is set to Signed and close the block. Hence, the fixed point format will be $\pm$ <1,8>.

(b) Again, we need two Gateway In blocks so duplicate the original.

(c) Connect the Uniform Random Number blocks to the Gateway In blocks.

(d) Save the file and check it against:

\CORDIC_Mag\CORDIC_mag2.mdl

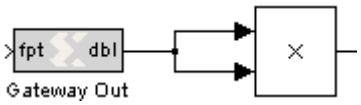
Exercise 11.4 Connect The Fixed Point Inputs To The Reference Design

The fixed point inputs are used to drive both the reference design and the CORDIC design. Hence, the Constant blocks driving the reference design at present can be replaced by the fixed point inputs just created. However, the floating point blocks of the reference design must be driven with floating point data. This means that the fixed point inputs must be converted back into a floating point format.

(a) Delete the Constant blocks driving the reference design.

(b) Drag in a Gateway Out block from the Xilinx Blockset>Basic Elements library. Open the block parameters and uncheck the "Translate into Output Port" option. Close the parameters and notice that the block has turned gray.

(c) Then connect the Gateway Out block to one of the multipliers like so:



(d) Duplicate the Gateway Out block and connect the new block to the other multiplier in the same fashion as above.

(e) Now connect the Gateway In blocks to the Gateway Out blocks.

(f) Finally drag in a System Generator block from the Xilinx Blockset>Basic Elements library. Save the design and check against:

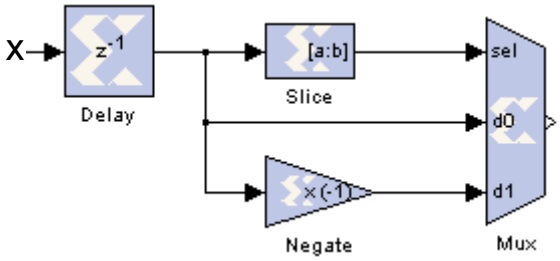
\CORDIC_Mag\CORDIC_mag3.mdl

Note that the reference design has simply been wrapped up into a subsystem which can be acheived easily by dragging a box around the blocks that are to go into the subsystem and then right clicking on the selected blocks and selecting Create Subsystem.

Exercise 11.5 *x* Input Converter

If the *x* input data is not controlled so that only positive values enter the CORDIC system, the magnitude will not be generated correctly. Hence, a small circuit must be designed to convert negative *x* input data to positive data.

(a) Design the small circuit shown below using blocks from the Xilinx Blockset library:



(b) Set the output of the Negate block to $\pm$ <1,8> (Number of Bits = 9, Binary Point = 8).

(c) Make sure and check the "Boolean Output" option for the Slice block.

(d) Set the output of the Mux block to $\pm$ <1,8>.

(e) Save the file and check against:

\CORDIC_Mag\CORDIC_mag4.mdl

Exercise 11.6 Import The First Cell

The cell that will compute the first iteration of the CORDIC system has already been developed and can be imported into your design and connected to the appropriate inputs.

Open the file:

\CORDIC_Mag\Cell1.mdl

Open the cell and note that the circuit represents the following equations:

$$x^{(1)} = (x^{(0)} - d_i(2^{-0}y^{(0)}))$$

$$y^{(1)} = (y^{(0)} + d_i(2^{-0}x^{(0)}))$$

By opening the Shift blocks it can be seen that a right shift of 0 bits is set, as is required for the first iteration. The width of the data paths can all be seen to be set to $\pm$ <2,8> thus satisfying the 8 fractional bits that were specified in the d_{eff} table. Note that 2 integer bits are required to accommodate the growth due to the scaling factor *K*.

(a) Copy the cell into your design.

(b) Now connect the output of the Mux block to the *x* input of the cell.

(c) Before connecting the y input to the Cell, insert a Delay block between the y Gateway In block and the Cell. Then connect the y Gateway In block to the Delay and the output from the Delay to the y input of the cell.

(d) Once again save the file and check against:

\CORDIC_Mag\CORDIC_mag5.mdl

Note that the circuit built to convert the x input data into positive values has been captured in a subsystem.

Exercise 11.7 Create The Second Cell

The initial cell can be used to generate any existing cells although it will have to be altered appropriately.

(a) Make a copy of the first cell and place it next to the original.

(b) Connect the *x* output of cell 1 to the *x* input of cell 2 and do the same with the *y* input/output of both cells.

(c) Now edit **both** Shift blocks in the second cell so that the number of bits shifted = 1.

(d) Save the file and check against:

\CORDIC_Mag\CORDIC_mag6.mdl

Exercise 11.8 Create The Third Cell

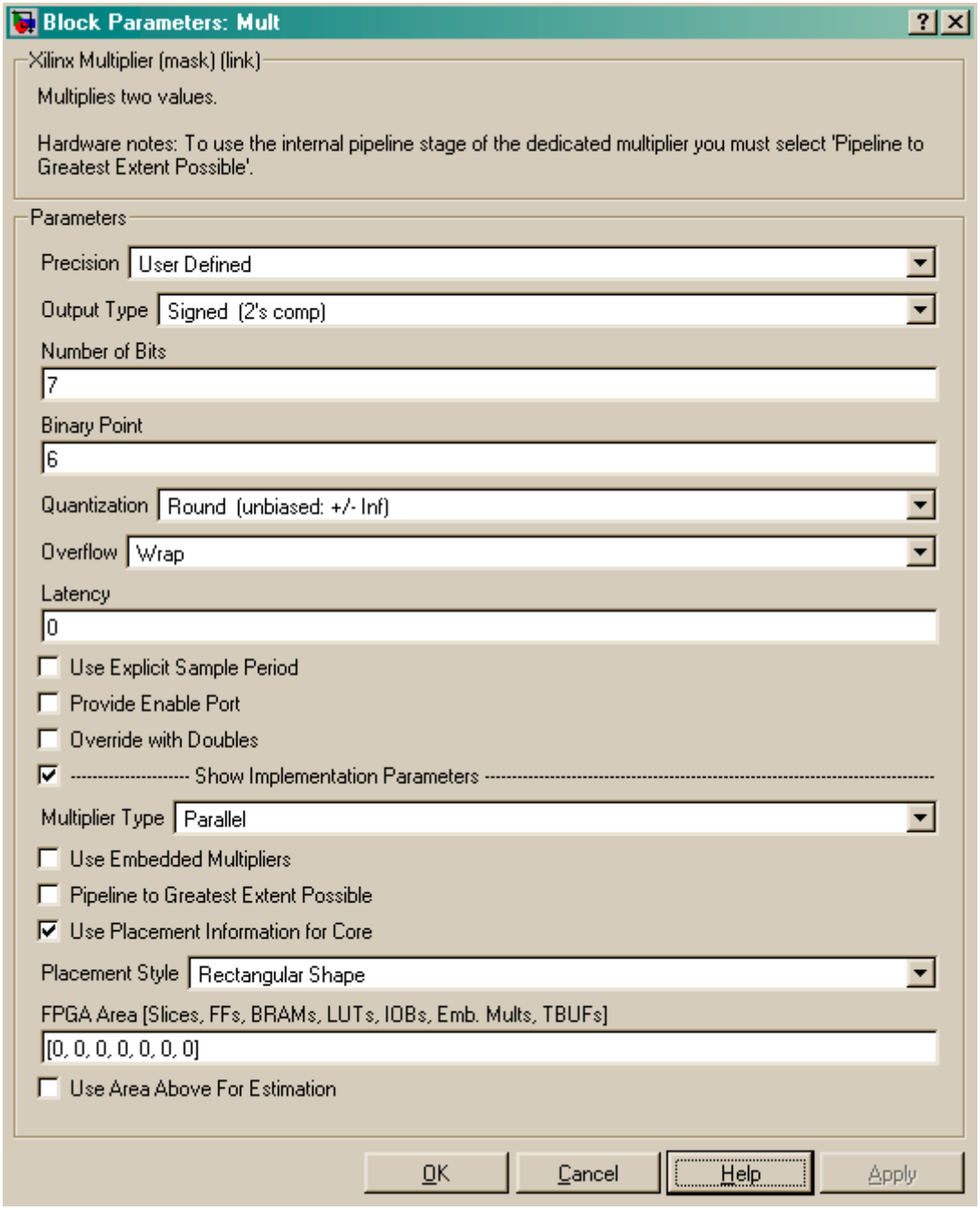
The second cell can now be used to create the third cell. This will be the final cell that is required (*n* = 3 was given by the d_{eff} table).

(a) Make a copy of the second cell and then connect the *x* output/input and *y* output/input of both cells as done previously.

(b) Open the third cell and edit both Shift blocks so that the number of bits shifted = 2.

(c) Save the file and check against:

\CORDIC_Mag\CORDIC_mag7.mdl



Exercise 11.9 Add The Post Scaling Multiplier

Now that all the cells have been implemented, all that remains of the CORDIC hardware is the post scaling multiplier and the constant source generating *1/K*.

(a) Drag in a Multiplier block and a Constant block from the Xilinx Blockset library.

(b) Connect the Constant block to one of the Multiplier inputs.

(c) Connect the *x* output from cell 3 to the other Multiplier input.

(d) Set the parameters of the Multiplier block according to:

(e) Set the output of the Constant block to $\pm<1,8>$ and enter the desired value to 1/1.62980060130066, which is equal to *1/K(3)*. Also check the “Sampled Constant” option.

(f) Drag in a Delay block and connect it to the output from the Multiplier.

(g) Finally select all the fixed point CORDIC blocks (by dragging a box around them) and put them into a subsystem.

(h) Save the file and check it against:

\CORDIC_Mag\CORDIC_mag8.mdl

Exercise 11.10 Subtract CORDIC Output From Reference Signal

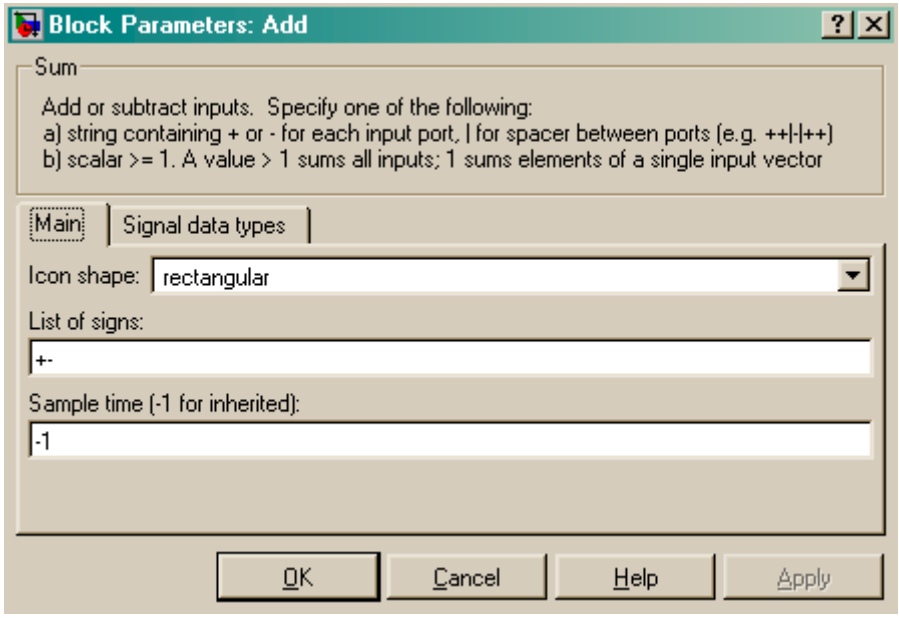
To obtain the error of the CORDIC system, its output must be subtracted from the output of the reference system. However, the reference system outputs a floating point signal so the CORDIC output must also be converted to floating point format before subtracting it. Also, we need to compensate for any delays so that both the reference output and the CORDIC output are synchronised.

(a) Drag in a Gateway Out block and connect it to the output of the CORDIC subsystem.

(b) Drag in an Integer Delay block from the Simulink>Discrete library. Open up the parameters of the block and set Number of Delays = 1. Then, connect the input of this block to the output of the Gateway Out block.

(c) The delay through the entire CORDIC data path is now 3 clock periods. Hence, we must delay the reference signal by the same amount. Copy the Integer Delay block and change its parameters to Number of Delays = 3. Connect the output of the reference subsystem to the input of this delay.

(d) Now drag in an Add block from the Simulink>Math Operations library. Open up the parameters of this block and set them to:



(e) Connect the delayed output from the reference system to the + port of the Add block and connect the delayed output from the CORDIC system to the - port of the Add block.

(f) Now drag in an Abs block from the Simulink>Math Operations library and connect it to the output of the Add block.

(g) Next, drag in a MinMax Running Resettable block from the Simulink>Math Operations library and set its parameters to Function = max. Connect the output of the Abs block to the input of this block. You do **not** need to connect the R port to anything.

(h) Finally, to view the output, drag in a Scope block from the Simulink>Sinks library and then connect the output of the Add block to the input of the Scope block. This will show all the errors. To view the max absolute error, drag in a Display block from the Simulink>Sinks library and connect it to the the MinMax block. Set the parameters of the this sink to Format = long_e. This will show the maximum absolute error for the simulation.

(i) Save the system and check it against:

\CORDIC_Mag\CORDIC_mag9.mdl

Exercise 11.11 **Run The System**

Now the system is ready to be run.

(a) Run the system by clicking on the ▶ button on the Simulink toolbar.

(b) Wait for the simulation to finish and then view the absolute error by double clicking on the Scope block. The maximum absolute error will be shown in the display block.

(c) We now need to compute $-(\log_2 \text{Max Absolute Error})$. This will give us the number of effective fractional bits.

(d) Perform this calculation using the value given in the Display. The following equation may be useful:

$$\log_2 x = \frac{\log_{10} x}{\log_{10} 2}$$

(e) Confirm that the number of effective fractional bits is > 5 as was specified at the start by the d_{eff} table.

(f) Change the length of the simulation and the seeds for the Uniform Random Number generators and reconfirm that 5 effective bits are still acheived.

11.3 Build A Fixed Point CORDIC System.

Exercise 11.12 **Build A CORDIC System To Compute d_{eff} = 10.**

(a) Using the system that was built to compute d_{eff} = 5 as a template, build another system to compute d_{eff} = 10. Use the table below to help find the parameters that you will need. As before, this table was generated for $|v(0)| = \sqrt{0.5}$. Verify the accuracy of your design using a floating point

reference design.

n / b	11	12	13	14	15	16	17
3	5.50	5.53	5.55	5.56	5.56	5.56	5.56
4	7.22	7.36	7.44	7.48	7.50	7.51	7.51
5	8.47	8.90	9.17	9.33	9.41	9.46	9.48
6	8.97	9.74	10.37	10.83	11.12	11.30	11.40
7	9.01	9.94	10.83	11.62	12.27	12.76	13.08
8	8.90	9.89	10.86	11.80	12.69	13.50	14.18
9	8.77	9.77	10.76	11.75	12.72	13.67	14.57
10	8.65	9.65	10.65	11.64	12.64	13.63	14.60

The following values may be useful:
K(5) = 1.64568891575726, K(6) = 1.64649227871248,
K(7) = 1.64669325427364, K(8) = 1.64674350659690
Once your system is complete, save it and check it against:

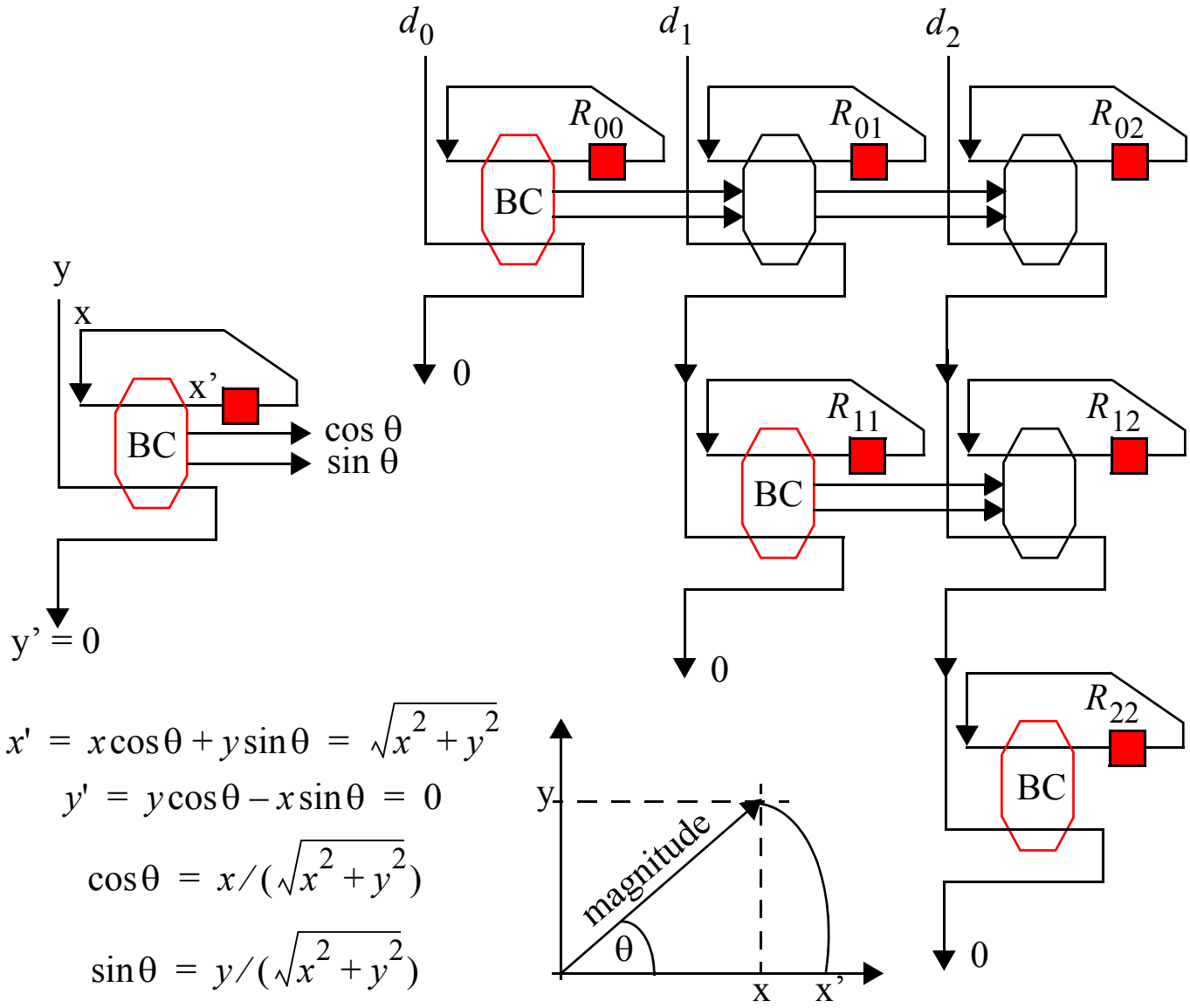

\CORDIC_Mag\CORDIC_deff10.mdl

How does your system compare? Have you managed to obtain d_{eff} = 10 with less iterations? Also, have you noticed that the final cell can be reduced to half the logic of the other cells because the *y* output is not required?

11.4 Using CORDIC In A QR-Array

In this section we are going to see how the CORDIC systems that have just been

built can be used in a QR-array.



Using the diagram above we can see that the function of each boundary cell (BC) is to rotate an input vector (x,y) onto the x-axis. Hence, the new x value (x') is equal to the magnitude of the initial vector and the new y value (y') is 0. However, each boundary cell must pass on the angle θ to the other cells in its row. The function of each remaining cell in the row is to rotate an input vector by the same angle θ as the boundary cell. One way to do this is to have each boundary cell compute cosθ and sinθ and pass these values on instead of θ itself. This means that each of the remaining cells in a row must only perform 4 multiplications and 2 additions/subtractions to perform the rotation. The added benefit of this approach is that each boundary cell must compute the magnitude of the initial vector anyway to obtain x'. To obtain cosθ and sinθ then involves inverting the magnitude and multiplying it by x and y respectively.

Exercise 11.13 QR Boundary Cell1

In this exercise we are simply going to look at one implementation of a QR boundary cell. Note that the vector magnitude is computed with 9 effective fractional bits. Open the file:

\CORDIC_Mag\QR_BoundCell1\QR_BoundCell1.mdl

(a) Open the CORDIC QR Boundary Cell subsystem and explore its contents. Notice that the same architecture has been used to generate the magnitude as was used in the previous exercises.

(b) Run the system and take a note of the max errors for both Cos and Sin.

Exercise 11.14 QR Boundary Cell2

The system used in this exercise is exactly the same as the previous one except that the vector magnitude is computed with 16 effective fractional bits as opposed to 9. Open the file:

\CORDIC_Mag\QR_BoundCell2\QR_BoundCell2.mdl

(a) Open the subsystems and confirm that the system is the same as the previous exercise with the exception of the CORDIC magnitude unit.

(b) Run the system and take a note of the max errors for both Cos and Sin again. What do you notice?

The fact that the CORDIC vector magnitude is computed to 16 effective bits rather than 9 has had a dramatic effect on the accuracy of the values of Cos and Sin. In the previous exercise the Cos and Sin were generated with approximately 6 effective bits. However, now that the accurcay of the magnitude has been improved, Cos and Sin are now generated with 11 effective bits. The QR algorithm is sensitive to quantisation so it is important to know where we can improve the accuracy of the algorithm.

Exercise 11.15 The Cost Of A QR Boundary Cell

Using System Generator and ISE the hardware for the previous system has been generated, synthesised and implemented. This was done to save time as the process can be quite lengthy. However, feel free to go through each stage on your own. To view the ISE project that was generated open:

\CORDIC_Mag\QR_BoundCell2\netlist\qr_boundcell2_clk_wrapper.ise

(a) Explore the project and take a look at the cost of the hardware by viewing the Place & Route Report. You should see the following:

```
Device Utilization Summary:

   Number of BUFGMUXs                    1 out of 16      6%
   Number of External IOBs              85 out of 556     15%
      Number of LOCed IOBs               0 out of 85      0%

   Number of MULT18X18s                 11 out of 136     8%
   Number of SLICEs                   1334 out of 13696   9%
```

(b) The speed at which the design can be clocked at can be found in the Text-based Post-Place & Route Static Timing Report. The timing summary is given below:

```
Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 11598636304080013000000O paths, 0 nets, and 8355 connections

Design statistics:
   Minimum period:  49.638ns (Maximum frequency:  20.146MHz)
```

12 Fixed Point Sigma-Delta

This section demonstrates on sigma delta techniques using fixed point arithmetic. It is recommended that the section on sigma delta principles of the DSPedia is covered before going through the examples below

The sequence of examples presented next implement a fixed point arithmetic sigma delta converter for an audio application. It is important to note that this example considers feedback issues. Under these conditions the designer has to take into account certain points which are highlighted in the examples below.

Exercise 12.1 10 bit to 1bit oversampling Sigma Delta

Open the system.

\sigmadelta\SigmaDelta01\SigmaDelta01.mdl

The one bit oversampled sigma delta signal is low pass filtered and downsampled in order to produce the original signal.

(a) Run the simulation view the 1 bit output of the sigma delta oversampler.

(b) Using System Generator the take the final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family.

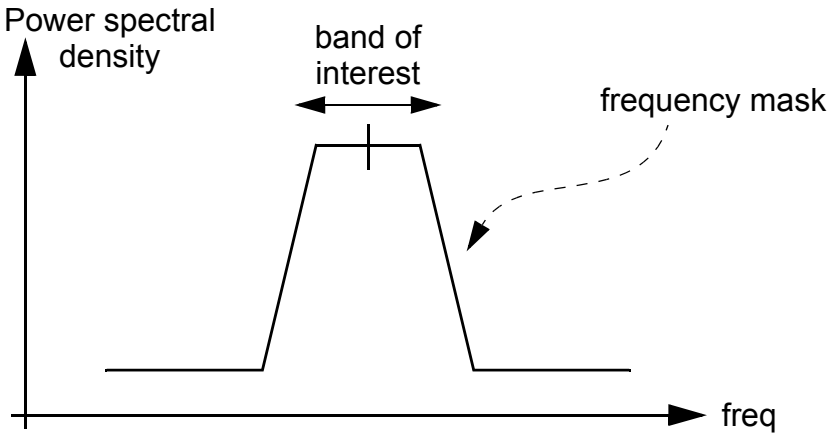
Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

---

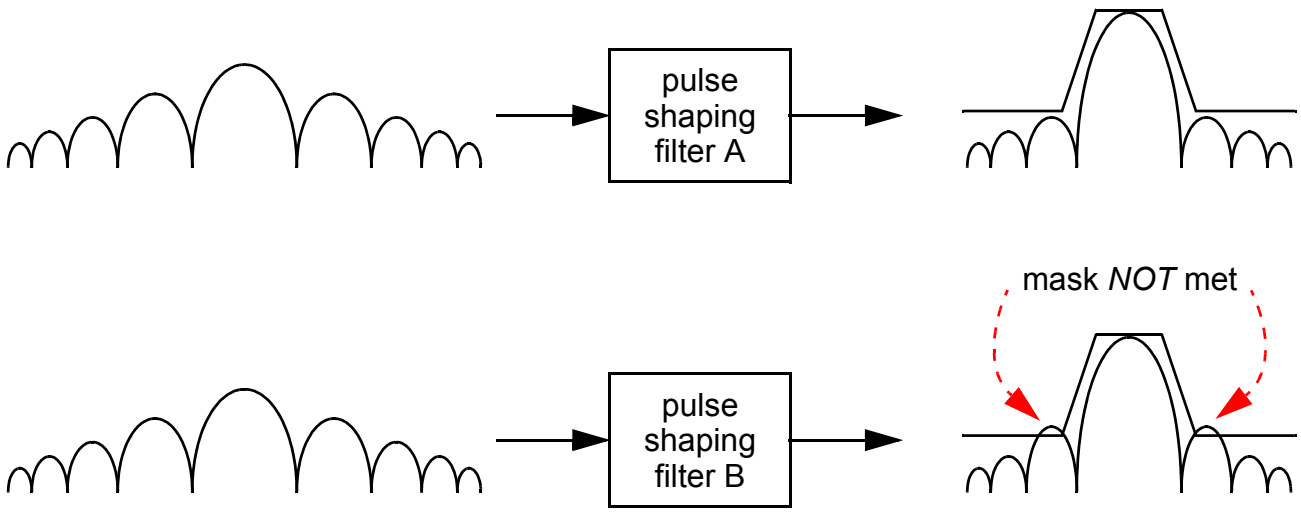
13 Fixed Point Bandlimiting: RRC Filtering

It is very important to make sure a transmitter only uses its assigned bandwidth. It is normally the role of the pulse shaping filter to limit the bandwidth of the transmitted signal so it does not leak into adjacent frequency bands. In order to ensure this the pulse shaping filter has to be designed properly.

Very often the amount of power that can be leaked into adjacent bands is given as a frequency mask as shown below. This mask specifies the maximum power levels which are allowed in the frequency band of interest and adjacent channels.



If the spectrum at the output of a pulse shaping filter does not meet the frequency mask requirements the amount of power leaked into adjacent bands larger than allowed and the filter does not meet the specifications. An example of this is given below, where filter B does not meet the mask but filter A does.



A filter designed with floating point precision will change its characteristic when its coefficients are quantised. Thus, a filter operating correctly and meeting the mask in floating point may be unsuitable once its coefficients have been quantised.

The next sequence of examples shows how the frequency response of a pulse shaping filter changes as its weights are quantised with different number of bits.

Exercise 13.1 64QAM Sequence Generation

Open the system.

\PulseShape\64QAM_01.mdl

This system shows the mapping of a sequence of symbols onto a 64QAM constellation. The original sequence of symbols has 64 levels and is generated at a rate of 28 Msymbols/sec. Since 64 levels can be represented with 6 bits the total bit rate is $28 \times 6 = 168$ Mbits/sec.

(a) Run the simulation and i check the produced constellation.

Exercise 13.2 Quantisation of 64QAM Constellation

Open the system.

\PulseShape\64QAM_02.mdl

In this example the generated 64QAM constellation is quantised and represented in fixed point precision using 8 integer bits signed format (<8,0> signed).

(a) Run the simulation and check the produced constellation.

Exercise 13.3 RRC Filter: Floating Point Weights

Open the system.

\PulseShape\64QAM_03.mdl

A certain RRC filter design has been chosen and used in this example. The In-phase and quadrature components generated (I & Q) are independently filtered with two real RRC filters. The coefficients of these filters are represented with double precision.

(a) Open the filters parameter box and check that they both have the same impulse response, i.e. they are the same filter. Observe the filters frequency response.

(b) Run the simulation, and observe the spectrum of the generated signal. Is it similar to the frequency response of the filters used?

Exercise 13.4 RRC Filter: 12-bit Weights, 20-bit Output

Open the system.

\PulseShape\64QAM_04.mdl

In this example fixed point filters are used. The coefficients of the RRC filter have been quantised using 12 bits while the output of the filter is represented using 20 bits.

(a) Open the fixed point filter parameter box and check that the wordlengths used are the ones indicated above.

(b) The gain blocks after the fixed point filters compensate for differences in amplitude between the two systems and have been used only to simplify the visual analysis of the results.

(c) Run the simulation and observe the results. The automated sequence will calculate the spectrum of the output of both implementations (floating point and 20-bit coefficients), and will overlay them. An averaging is performed so the differences can be more easily observed. Observe the differences between the two produced spectra plots.

Exercise 13.5 RRC Filter: 8-bit Weights, 16-bit Output

Open the system.

\PulseShape\64QAM_05.mdl

In this example the number of bits used to represent the weights and the filter output has been reduced. The coefficients are now represented with 8 bits while the output of the filter is coded using 16 bits. Note that gain blocks have been used again to normalise the output of the filters.

(a) Open the fixed point filter parameter box and check that the wordlengths used are the ones indicated above.

(b) Run the simulation and view the spectrum at the output of the different filters.

(c) Observe the differences in the spectrum.

(d) Using System Generator the take the final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family.

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

Exercise 13.6 RRC Filter: 5-bit Weights, 13-bit Output

Open the system.

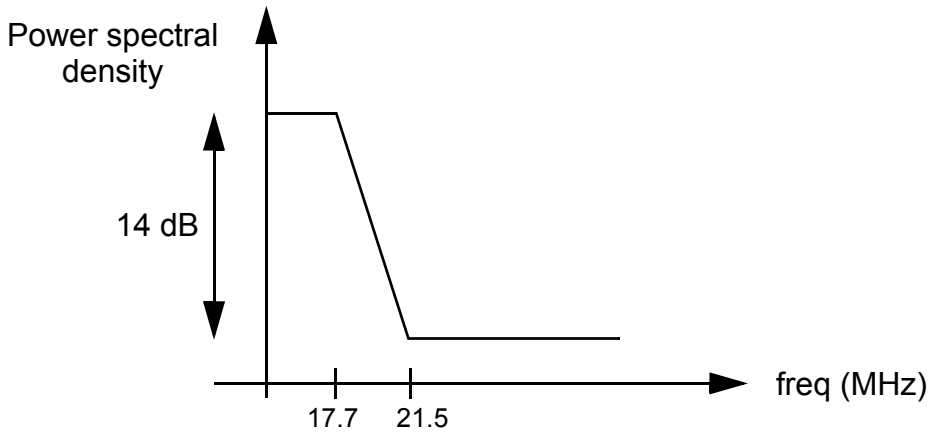
\PulseShape\64QAM_06.mdl

The number of bits used to represent the weights and the filter output has been reduced even further. The coefficients are now represented with 5 bits while the output of the filter is coded using 13 bits.

(a) Run the simulation and view the results. Observe the overlay of the spectrum at the output of the different filters after the automated sequence has been executed.

(b) What is the main effect in the spectrum when the wordlength used to represent weights and outputs is reduced?

(c) Observe the differences in the spectrum. Which implementations meet the mask specified below?



(d) Using System Generator the take the final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family.

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	

How does this compare to the cost from the previous exercise with the smaller wordlength?

---

14 FPGA as an ASIC - Digital Downconverter

In this sequence of examples a digital downconverter is constructed in Simulink.

As a quick view the general DSP structure of the system is shown below:

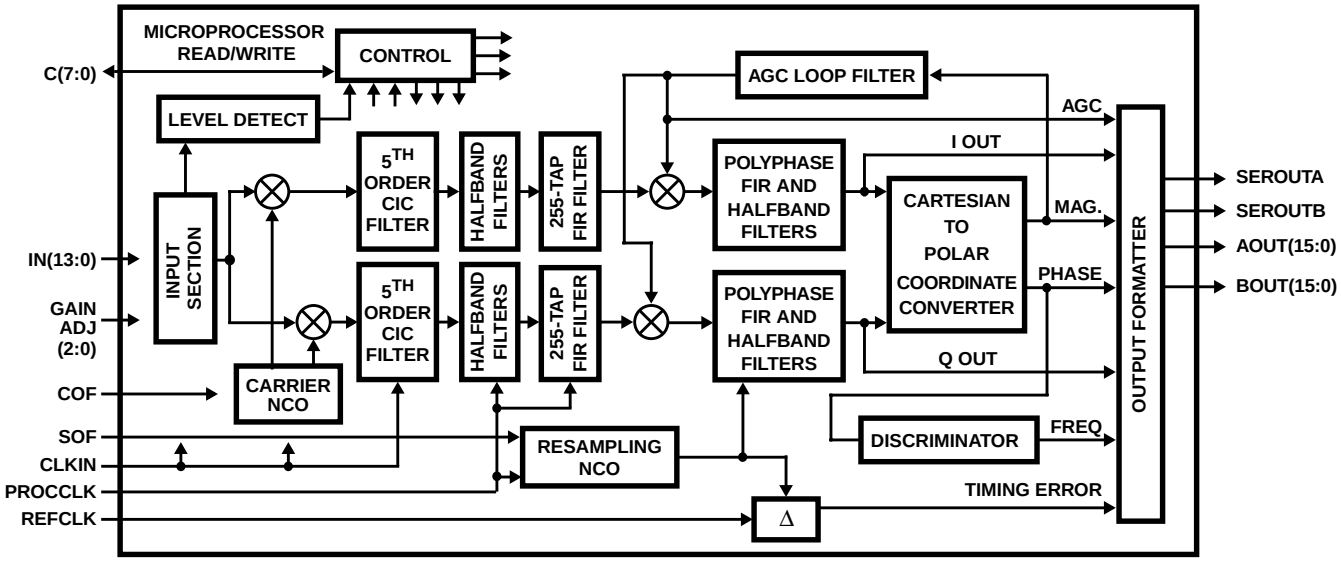
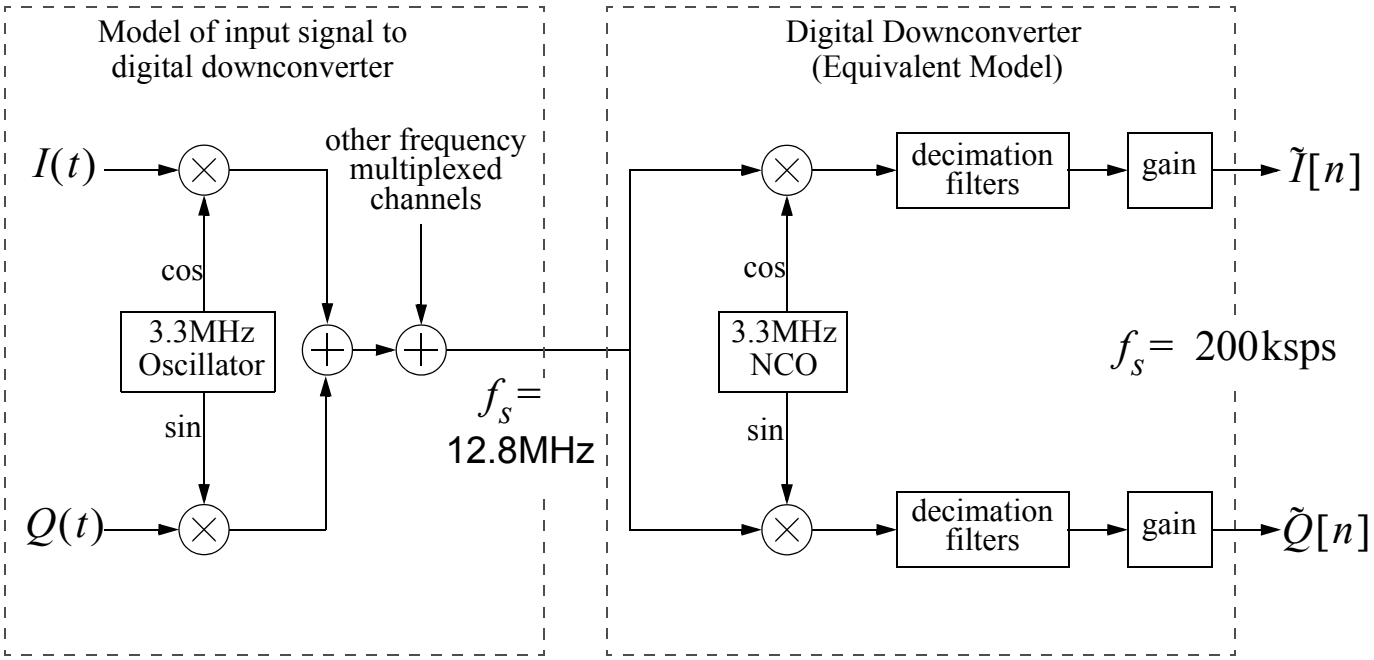


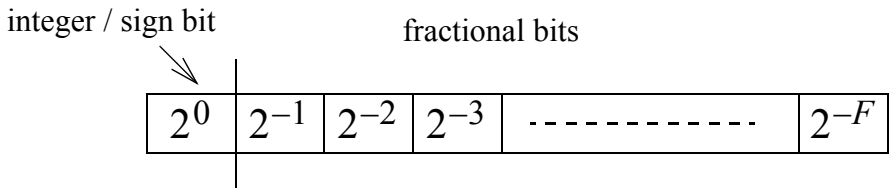
Figure 14.1: Digital Downconverter DSP functional block diagram.

The examples demonstrate the digital downconversion of two modulated information signals, $I(t)$ and $Q(t)$. These signals arrive at the input to the digital downconverter modulated onto the in-phase and quadrature components of a 3.3MHz intermediate frequency (IF). The diagram below shows a simplified model of the system which will be constructed. The sampling rate at the input to the downconverter is 12.8Msps however at the output this rate is reduced to 200ksps and the unwanted channels are removed from the signal.



In the simulations almost all of the signals are represented using N bit signed

fixedpoint values interpreted as having 1 integer bit and $F = N - 1$ fractional bits (denoted as <1,F> format). This format allows values in the range $-1 \le x \le 1 - 2^{-F}$ to be represented. The bit weightings of this format are shown in the diagram below.



Bit weighting of the <1,F> signed fixedpoint format

Exercise 14.1 Bandlimited Information Signals

Open the system

\qam-ddc\Downconverter_01.mdl

This system reads a pair of pre-generated bandlimited information signals into Simulink from external files.

(a) Run the system and estimate the bandwidth occupied by these signals.

Exercise 14.2 Modulation of Information Signals onto IF Carrier

Open the system

\qam-ddc\Downconverter_02.mdl

In this system the I and Q waveforms are modulated onto a 3.2MHz intermediate frequency (IF) carrier signal. A model of the adjacent channels has also been added.

(a) Inspect the new blocks and their parameters.

(b) Run the system.

(c) Now view the resulting signal at the output of the generator. Notice how it looks like random noise.

(d) View the spectrum and you should be able to see the signal of interest centred on the carrier frequency and surrounded by noise representing the other channels.

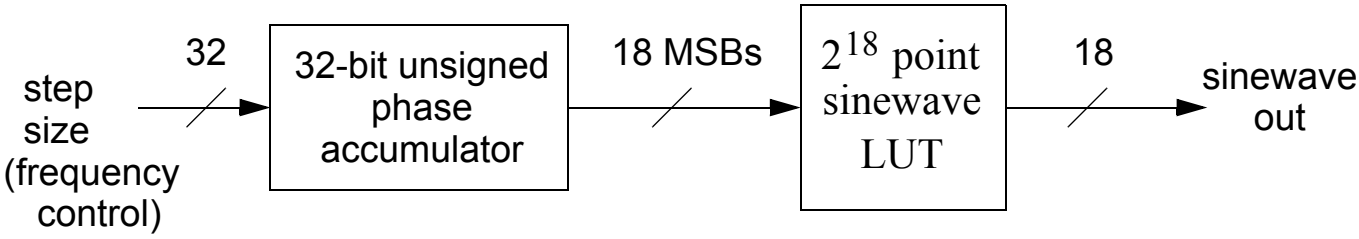
Exercise 14.3 Numerically Controlled Oscillator

Open the system

\qam-ddc\Downconverter_03.mdl

In this system the received signal is mixed down to baseband using a numerically controlled oscillator (NCO) and two multipliers. The NCO generates sinusoidal signals by using an accumulator to generate a variable frequency ramp function. The ramp function is then used to address a sine Look-Up Table (LUT). The result is a variable frequency digital oscillator. A block diagram of the NCO is shown below.

Numerically controlled Oscillator



(a) Inspect the blocks which form the NCO and the mixer. Pay particular attention to the number of bits used at each stage.

(b) Run the system and view the various signals produced.

Note that the signal has now been mixed down to baseband however the noise components are still there.

Exercise 14.4 CIC Filter

Open the system

\qam-ddc\Downconverter_04.mdl

In this example a 5-stage CIC filter has been added to begin the downsampling process. This reduces the sampling rate by a factor of 16 and removes a significant proportion of the adjacent channels.

(a) Run the system.

(b) Compare the in-phase transmitted signal to the in-phase signal at the output of the CIC filter. Does there appear to be any correlation between the two?

Exercise 14.5 Halfband Filters

Open the system

\qam-ddc\Downconverter_05.mdl

Although CIC filters are an efficient to perform decimation in a downconverter they may only be used to reduce the rate to a certain degree due to the problem known as “CIC droop”. CIC droop can modify the frequency content of our channel of interest and lead to inter-symbol interference.

In our downconverter system the CIC brings the sampling rate down to 8x the symbol rate. The remaining downconversion is performed using an FIR “halfband” filter and a programmable FIR filter.

(a) View the parameters of the two new filter blocks which have been added. In particular view the “weights” property page and view the impulse and frequency responses of the filters. Why are these called “halfband” filters? Note that every second weight is zero (except for the middle weight) and hence these should lead to cheaper implementation.

(b) Now compare the transmitted and received information signals. What do you notice about the signal levels. Remember that the input to the Downconverter is scaled to utilize the full dynamic range.

Exercise 14.6 Active Gain Control

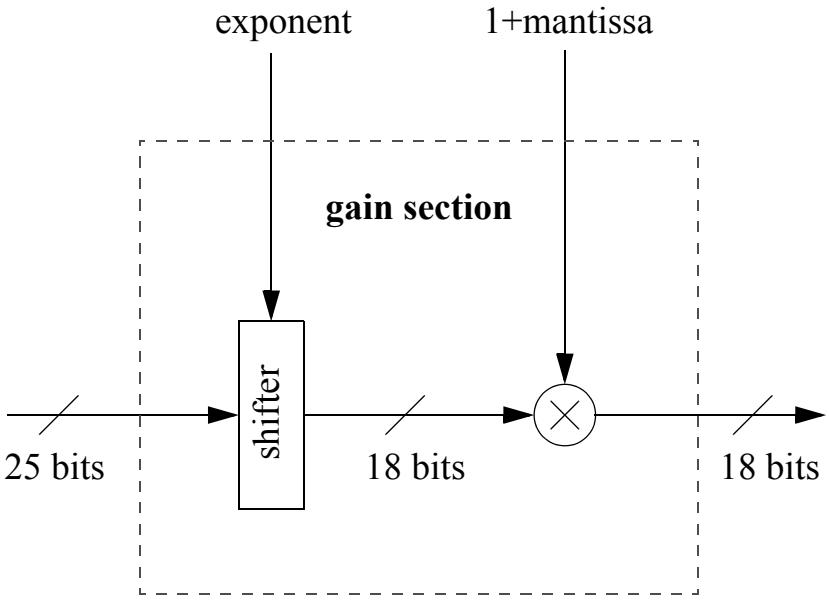
As was seen in the previous example, the output of all of the filtering stages can be a signal with relatively low amplitude. This is because all of the adjacent channels (which form the majority of the downconverter input) have been rejected. The Active Gain Control (AGC) stage is used to amplify the signal so that it utilizes the dynamic range of the output word.

The gain part of the AGC loop has been included in this final simulation. It consists of a shift and a multiplier as shown in the diagram below. The *exponent* controls the shift and therefore provides gains which are powers of 2. The *mantissa* may be used to implement non-power of 2 gains. The overall gain is

$$\text{gain} = (1 + \text{mantissa})(2^{\text{exponent}})$$

$$\text{gain[dB]} = (6.02)\text{exponent} + 20\log_{10}(1 + \text{mantissa})$$

The exponent is a number between 0 and 15 and the mantissa is a number between 0 and $1 - 2^{-8}$.



Open the system

\qam-ddc\Downconverter_06.mdl

(a) Inspect the parameters of the gain section in the system.
(b) Calculate the total gain in dB which is introduced.
(c) Now run the system and view the results.
(d) Using System Generator the take the final design through the ISE tools to an FPGA implementation using the XCVP30 from the VIrtex II Pro family.

Report	Result	Value
Place and Route Report	Number of BUFGXMUXs	
	Number of External IOBs	
	Number of 18 x 18 multipliers	
	Number of SLICEs	
Post place & route static timing report	Minimum Period	
	Maximum Frequency	