# EE367 Lab 6
# Creating a FIR filter in VHDL

The goal of this lab will be to create a FIR filter of order M=20 in both Matlab and VHDL. The goal is to get familiar with the tool chain and create the necessary components using Xilinx's Core Generator. We will need to test our VHDL code and this is easily done using Matlab via the Matlab co-simulation capabilities of Aldec's Active-HDL.

The equation to implement a FIR filter is given as:

$$y[n] = \sum_{k=0}^{M} b_k x[n-k]$$

This means that we need a multiplier, an adder (accumulator), a memory to store the filter coefficients, a memory to store past sample values, a control block to coordinate the multiplier, accumulator, and memory, and an address generator to get the appropriate coefficiencts and delayed sample values.

The following tasks will be performed:

1. Create a filter in Matlab and examine the frequency response of the filter.
2. Write the VHDL code that does this filtering.
3. Test the VHDL code by comparing its output with the filter we created in Matlab.

**Note: There will be references to Aldec's Active-HDL in the lab. Ignore any references to Active-HDL and do the same tasks using Xilinx's ISE (for the next several pages).**

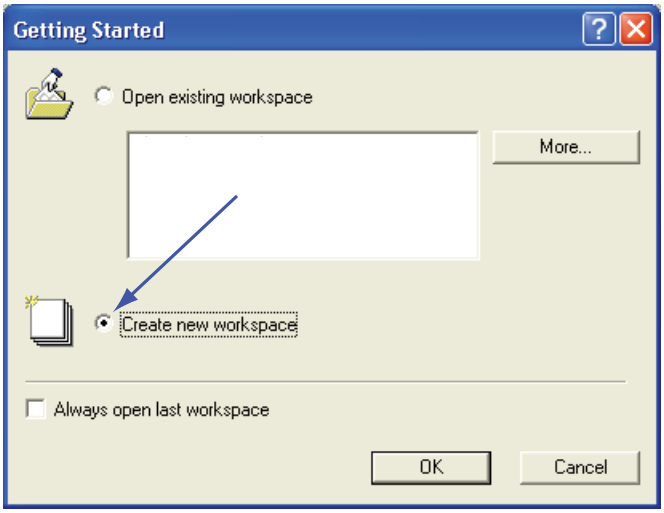
# Writing your FIR VHDL code

**Setting up the Workspace**

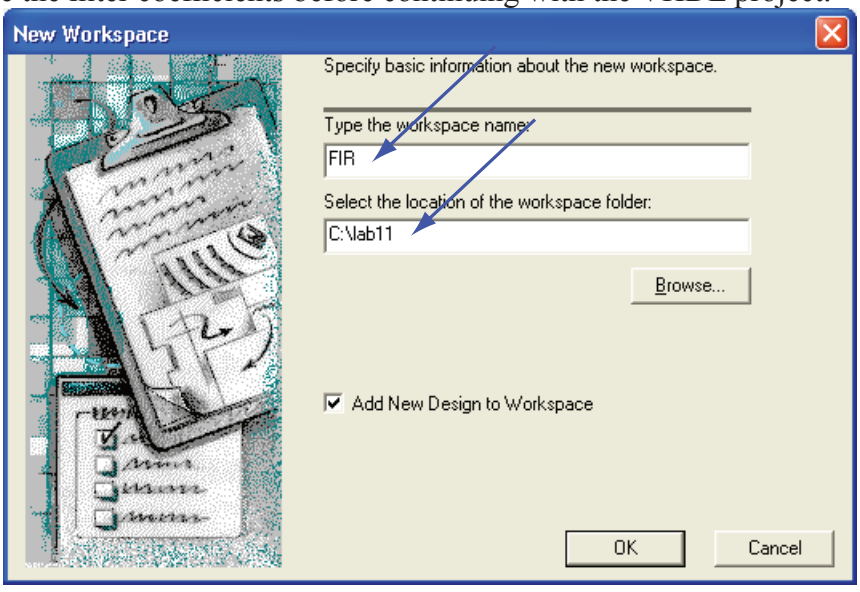
To get started, open ALDEC's Active-HDL (7.3) which is a FPGA Design and Verification Suite. Do this by selecting:

Start→All Programs→Active-HDL 7.3

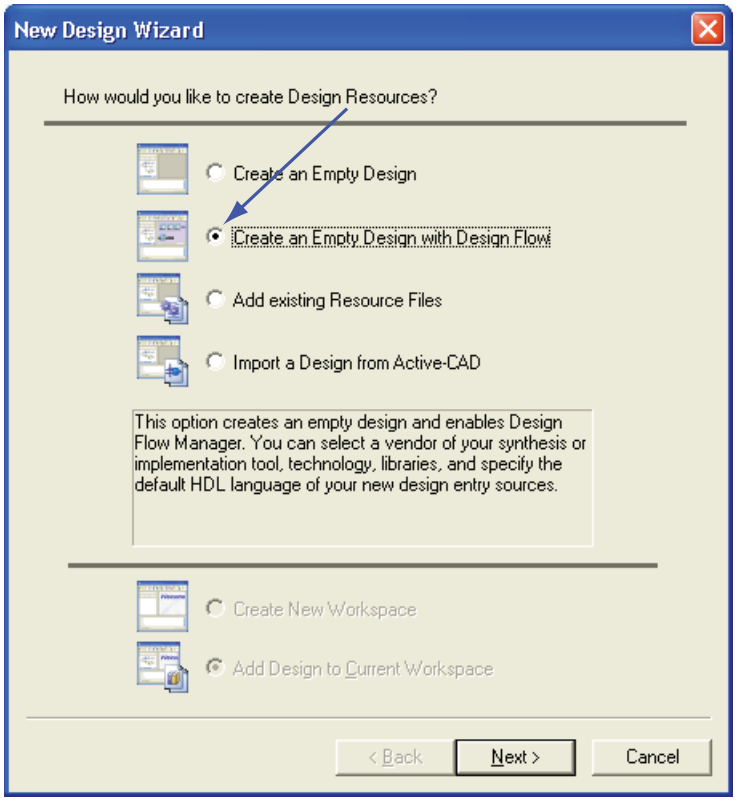
Select the *Create new workspace* option and click OK



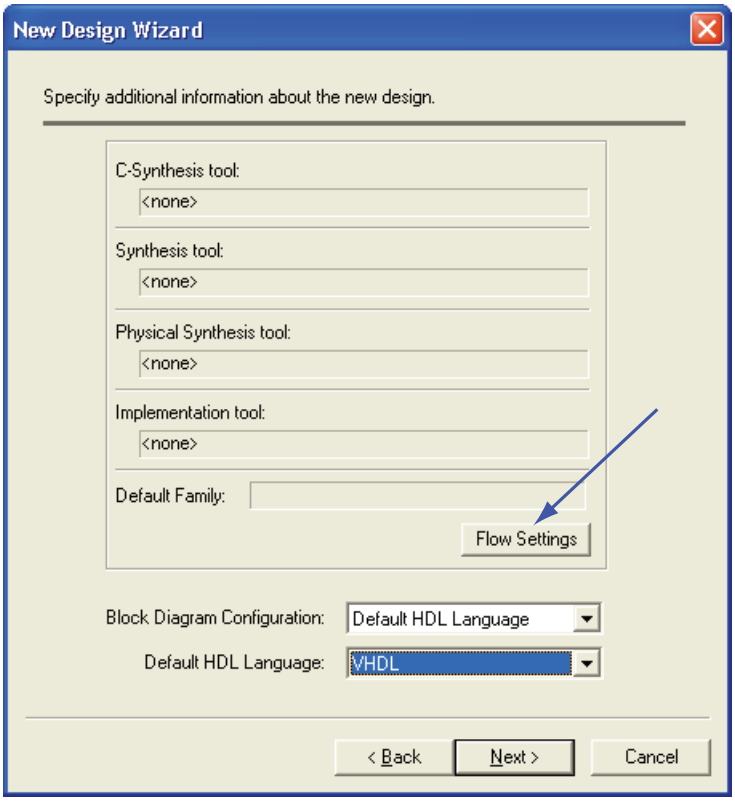

Give your workspace the name *FIR* and point the location to your lab directory. Note: the source directory "src" under this project location is where you will be storing all your VHDL, Matlab, and CoreGen files. As a result, we will first create the workspace and then jump back to Matlab to create the filter coefficients before continuing with the VHDL project.



In the New Design Wizard, select *Create an Empty Design with Design Flow* since we are targeting the ML403 board and we want Xilinx's ISE tools for synthesis and place & route.
Click Next.



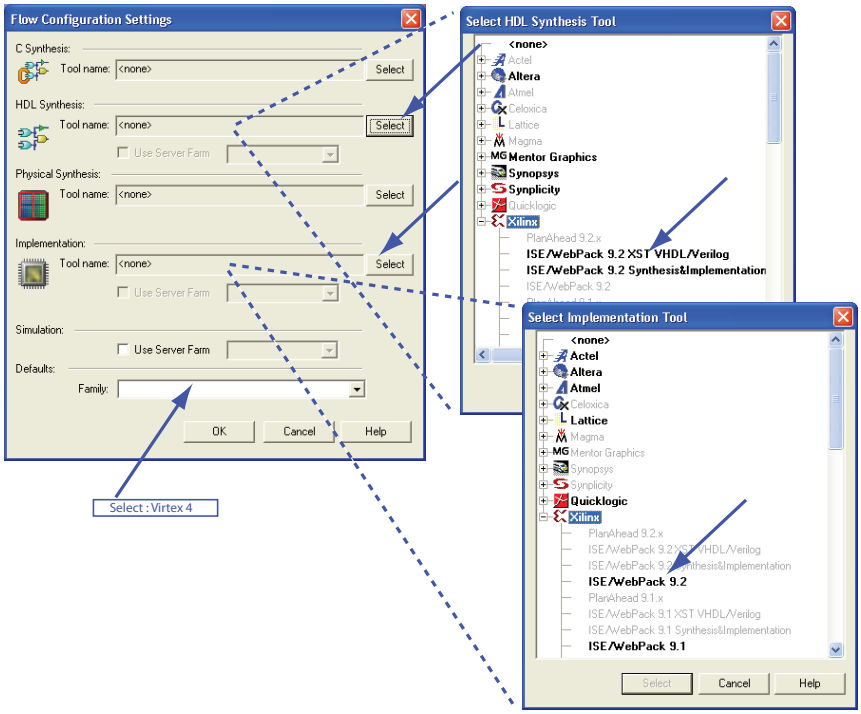

Click on Flow Settings to specify the Xilinx tools.



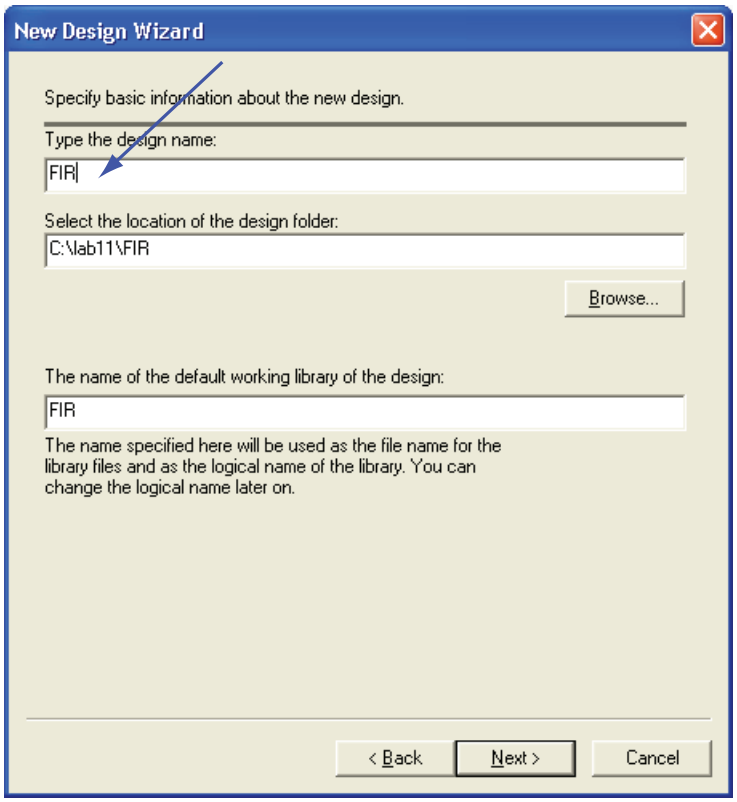
Select the following for the Flow Control Settings:

1. For *HDL Synthesis* select *Xilinx ISE/WebPack 9.2 XST VHDL/Verilog*
2. For *Implementation* select *Xilinx ISE/WebPack 9.2*
3. For *Family* select *Xilinx9x VIRTEX4*

Leave C Synthesis & Physical Synthesis as <none>.
Click OK, make sure the default language is VHDL, and then
Click Next




Enter FIR as the design name and click Next, then Finish.

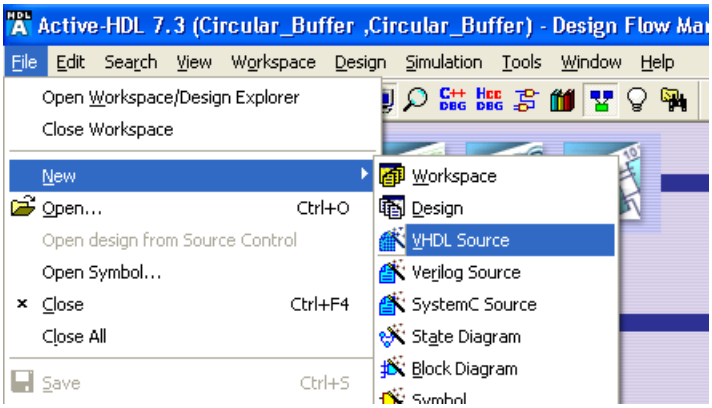


## Creating your FIR VHDL block

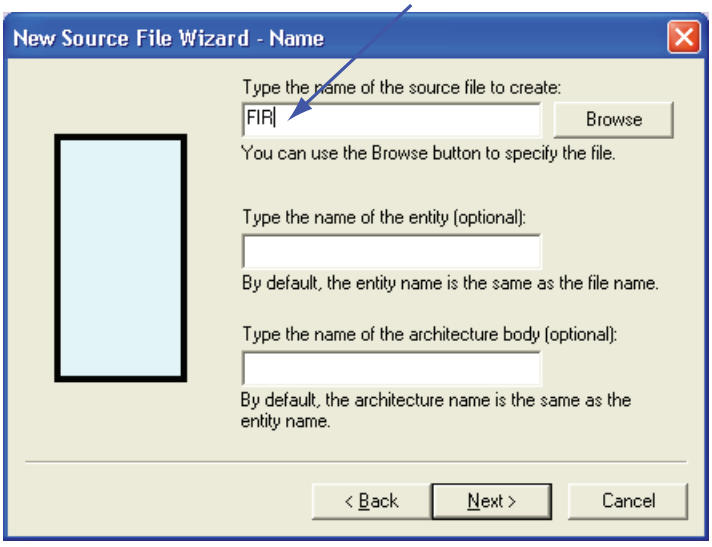
Our fist task will be to create the VHDL code of the FIR block. We will create a block with the following entity (interface) signals:

| | | |
|---|---|---|
| Input: | data_in | (x[n]) |
| | data_in_ready | |
| | clk | (data will be written on the rising edge of the clock signal) |
| | reset | |
| Output: | data_out | (y[n]) |
| | data_out_ready | |

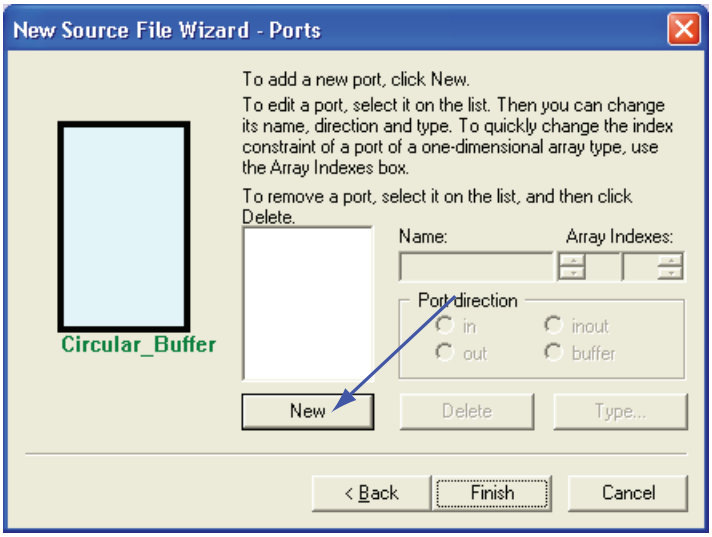
To do this:
1.) Select File→New→VHDL Source
2.) Click Next on the first popup window of the New Source File Wizard to add the file to your design.



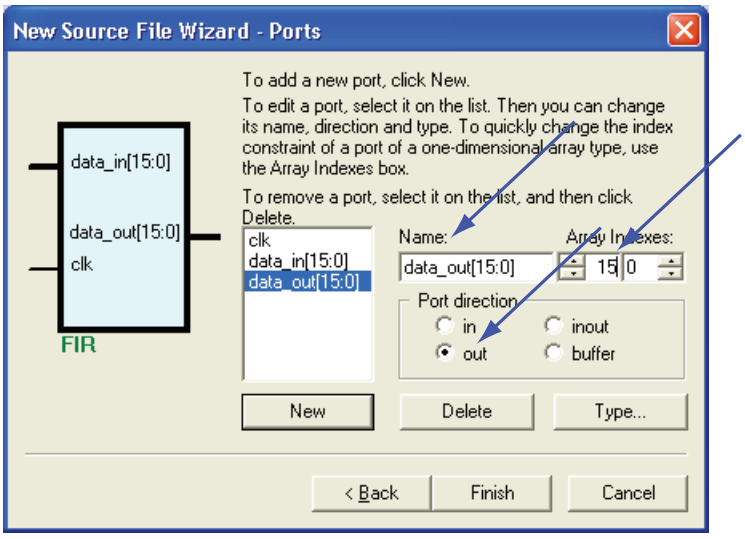

Enter FIR as the name of the source file and click Next.



Click New to add a new interface signal




Enter data_in in the Name: field (the numbers will show up automatically).
Enter 15 in the first Array Index block (zero will show up automatically). For 16 bits you have bits 15 downto bit 0. The in port direction is already set.



Repeat this for all your signals and then click Finish.

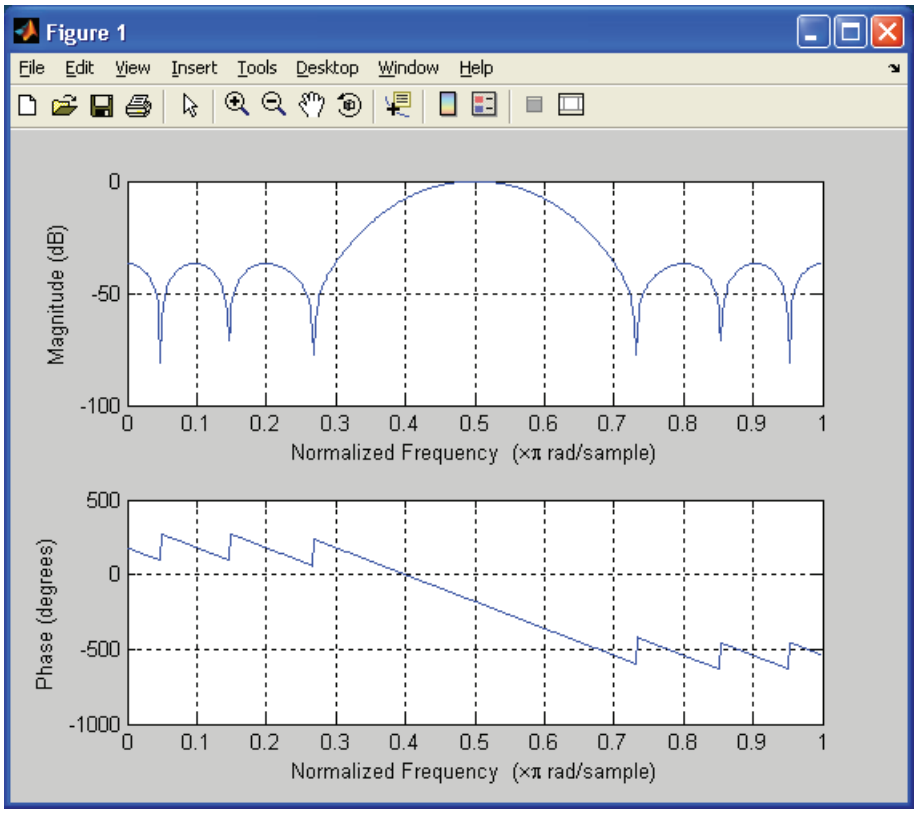
| Name | Direction | Size | Comment |
|---|---|---|---|
| data_in | in | [15:0] | 16 bits |
| data_in_ready | in | - | single bit |
| clk | in | - | single bit |
| reset | in | - | single bit |
| data_out | out | [15:0] | 16 bits |
| data_out_ready | out | - | single bit |

This generates the VHDL code with the following entity built for you:

```
entity FIR is
        port(
                clk                 : in STD_LOGIC;
                reset               : in STD_LOGIC;
                data_in             : in STD_LOGIC_VECTOR(15 downto 0);
                data_in_ready       : in std_logic;
                data_out            : out STD_LOGIC_VECTOR(15 downto 0);
                data_out_ready      : out std_logic
            );
end FIR;
```

# Creating the FIR Filter in Matlab

In Matlab, use the fir1() function to create a 20th order FIR bandpass filter with a bandpass of w1<w<w2 where w1 = 0.49, w2=0.51, and 1 represents Fs/2. Return the coefficients in b, i.e. b=fir1() and use freqz() to plot the frequency response. You should get the following figure:




Now plot the b coefficients and notice their values. Since we will be loading these coefficients into blockRAM that will function as a ROM, we need to scale the values appropriately so that they will fit into a 16-bit signed integer word type.

First, create a file called fir_init.m that will create these scaled b coefficients that will be converted to a fixed-point data type.

In Matlab, fixed-point data types are created using the fi() function. Typing *help fi* at the Matlab prompt will show various ways of creating a fixed-point object:

>> help fi

We will use the case with four inputs fi(v,s,w,f) that returns a fixed-point object with value v, signedness s, word length w, and fraction length f, where:

| | |
|---|---|
| v=0; | We don't care what the value is at this point - just create the object as zero. |
| s=1; | signed |
| w=16; | word length of 16 bits |
| f=0; | no fractional part (note, the binary point is between w and f, i.e. w.f) |

>>x=fi(0,1,16,0)

Which returns:

x =0

| | |
|---|---|
| DataTypeMode: | Fixed-point: binary point scaling |
| Signed: | true |
| WordLength: | 16 |
| FractionLength: | 0 |

| | |
|---|---|
| RoundMode: | nearest |
| OverflowMode: | saturate |
| ProductMode: | FullPrecision |
| MaxProductWordLength: | 128 |
| SumMode: | FullPrecision |
| MaxSumWordLength: | 128 |
| CastBeforeSum: | true |

Notice that there are additional modes that can be set. We can use these modes when simulating the hardware. Following is a brief description of the modes, however, we will not deal with these for our "simple" project, but you should know that they exist.

**RoundMode** - We won't worry about the RoundMode since this is used by the quantizer to convert floating point numbers in Matlab to fixed-point. We will accept the default of nearest that rounds to the nearest fixed-point number.

**OverflowMode** - We have two options: *saturate* or *wrap*. Wrap is the easiest to implement in hardware (do nothing), but it will do bad things to the signal if it occurs, i.e. one can get "pops" in the signal where a large signal just became small and vice versa. Saturate is the preferred method that "clips" the output to a maximum or minimum value. The default is saturate, which is the mode we would normally use.

**ProductMode** - If we multiply two 16-bit numbers together, we will get a 32-bit result. However, we can only keep 16 bits (if the output is not fed to an adder). So, which 16 bits do we keep?

> *KeepLSB* - keep the least significant bits. In this mode full precision is kept, but overflow is possible. This is similar to multiplying two 16 bit unsigned integers in C.
>
> *KeepMSB* - keep the most significant bits. In this mode, overflow is prevented, but precision is lost.
>
> *SpecifyPrecision* - Here you specify the word and fraction length. This is the most flexible case and can simulate the case where you are grabbing a specific bit sequence from your hardware multiplier when you know what range will occur.
>
> *FullPrecision* - Here Matlab keeps track of what is occuring in a dynamic fashion. This is the mode you would initially run first and then use the quantizer to start shrinking the word size until bad things started happening (overflow, loss of precision, etc.) to your algorithm. This is the process that you would generally take if you were targeting a FPGA, followed by *SpecifyPrecision*. If at some point this becomes greater than *MaxProductWordLength*, Matlab will generate an error.

The default is *FullPrecision*.

**ProductWordLength** - The word size that results when two fi objects are multiplied together.

We would set this to 32 for our hardware as your multiplier will generate a 32-bit result. The default is 32. The maximum is 128 (*MaxProductWordLength*).

In practice, when performing a multiple accumulate (MAC) you would create a 32-bit product and then your accumulator would be even larger to avoid overflow when summing terms. The MAC block in the Xilinx FPGA is called a DSP slice which can multiply two 18-bit numbers together and add the 36-bit result into a 48-bit accumulator. The ML403 board has a FX12 Virtex-4 that has 32 of these DSP slices that can run in parallel at several hundred MHz.

**SumMode** - Essentially identical to *ProductMode* except this configures the adder. The maximum length is 128 (*MaxSumWordLength*).

**CastBeforeSum** - This determines if both operands are cast to the sum data type before addition (true=1) or not (false=0). The default is true.

**Create the file fir_init.m with the following code:**

```
b=fir1(20,[.49 .51]);
freqz(b,1,521);
```

```
b=b-min(b);
b=b/max(b);
b=b*2^16;
b=b-2^15;
max(b)
min(b)
bf = fi(b, 1, 16, 0);
max(bf)
min(bf)
```

This will scale the b coefficients to a 16-bit range and create the 16-bit signed integer data type. Now add the following code:

```
v=[]
fid = fopen('ROM.coe','w');
fprintf(fid,'MEMORY_INITIALIZATION_RADIX=16;\n');
fprintf(fid,'MEMORY_INITIALIZATION_VECTOR=');
for k=1:length(b)-1
   bf = fi(b(k), 1, 16, 0);
   v=[v bf.int];
   fprintf(fid,'%s,',bf.hex);
end
bf = fi(b(length(bf)), 1, 16, 0);
v=[v bf.int];
fprintf(fid,'%s;\n',bf.hex);
v
```

This code creates a file called ROM.coe which is a file we need to initialize the memory when we create a ROM memory block using CoreGen. The file needs the key word MEMORY_INITIALIZATION_RADIX to tell CoreGen if the VECTOR numbers are hexidecimal, decimal, or binary. The bf.hex gives the value has a hex string. If you needed a binary string, you would use bf.bin. Run the file to create ROM.coe and put this file in the “src” directory in your workspace.

# Creating VHDL Blocks via the Xilinx Core Generator

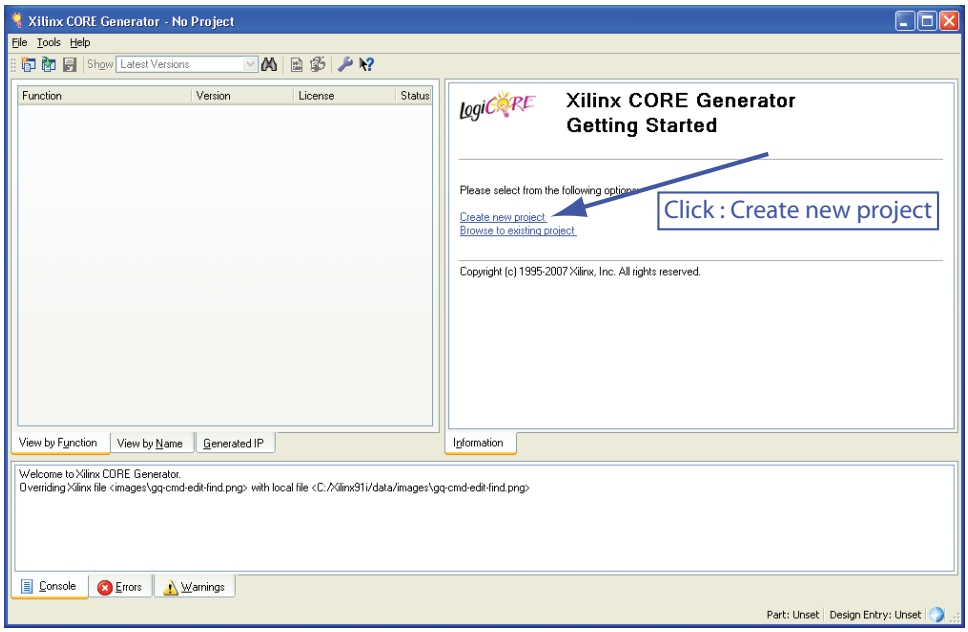
For our FIR filter we need two memories (one for the coefficients and one for the delays that will be used as a circular buffer) and a Multiplier-Accumulator (MAC). Thus the blocks we will create are:

1. Coefficient Memory - ROM (we will preload the coefficients)
2. Delay Memory - Dualport RAM (circular buffer to implement $z^{-n}$)
3. Multiplier with Adder (MAC)

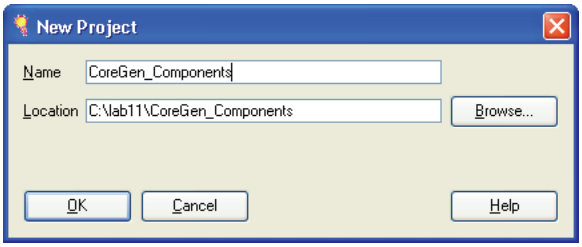
To start off we need to create a new project within CoreGen.

Step 1: Create a new projet.

Start→All Programs→Xilinx ISE 9.1i→Accesories→Core Generator

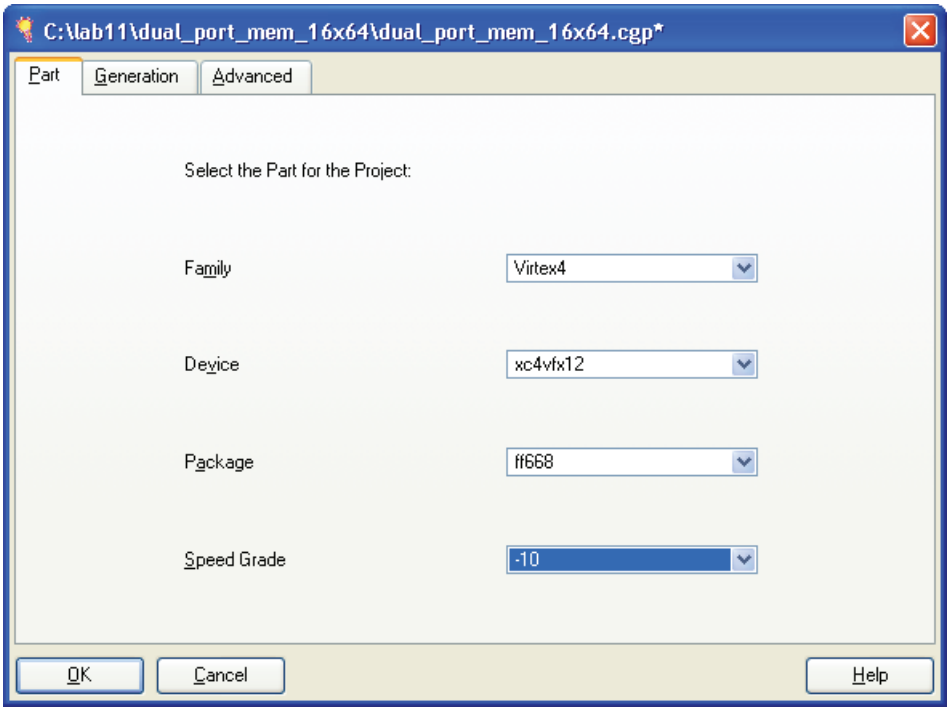



Step 2: Name the project "CoreGen_Components", point the location to the "src" directory that was created under your Active-HDL workspace, and then click OK.



Step 3: Since we are targeting the ML403 board, we need to specify the FPGA that the board contains. You can read these numbers off the chip. Enter the following information and click OK.

| | | |
|---|---|---|
| Family | - Virtex 4 | |
| Device | - xc4vfx12 | (Notice XC4VFX12 on the chip) |
| Package | - ff668 | (Notice the FF668AGQ0513 on the chip) |
| Speed Grade | - -10 | (Notice the 10C-ES on the chip) |



# Creating the Coefficient ROM

The first step in creating the coefficient ROM is to create the .coe initialization file that CoreGen will read when it creates the ROM. **Note: you already did this with your Matlab code.**
Open up a text editor and type in the first line as:

MEMORY_INITIALIZATION_RADIX = 16;

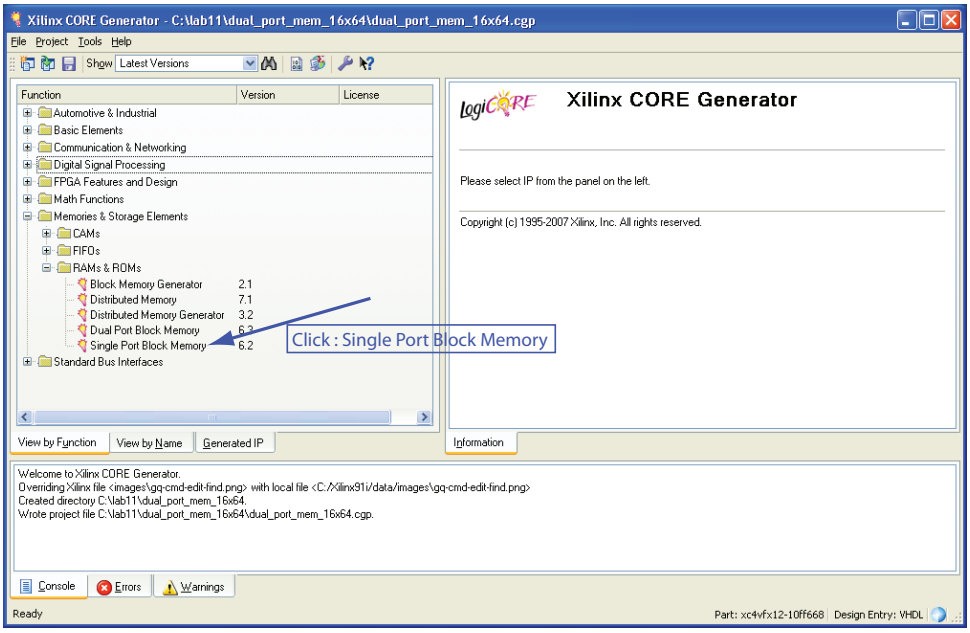
This specifies the radix of the numbers you will be entering, which will be in base 10. The other options are 16 (hex) and 2 (binary).

The next line will be the actual data (comma separated and ending with a semicolon) that starts as:
MEMORY_INITIALIZATION_VECTOR =1, 0, 3, 0, 0, 2;

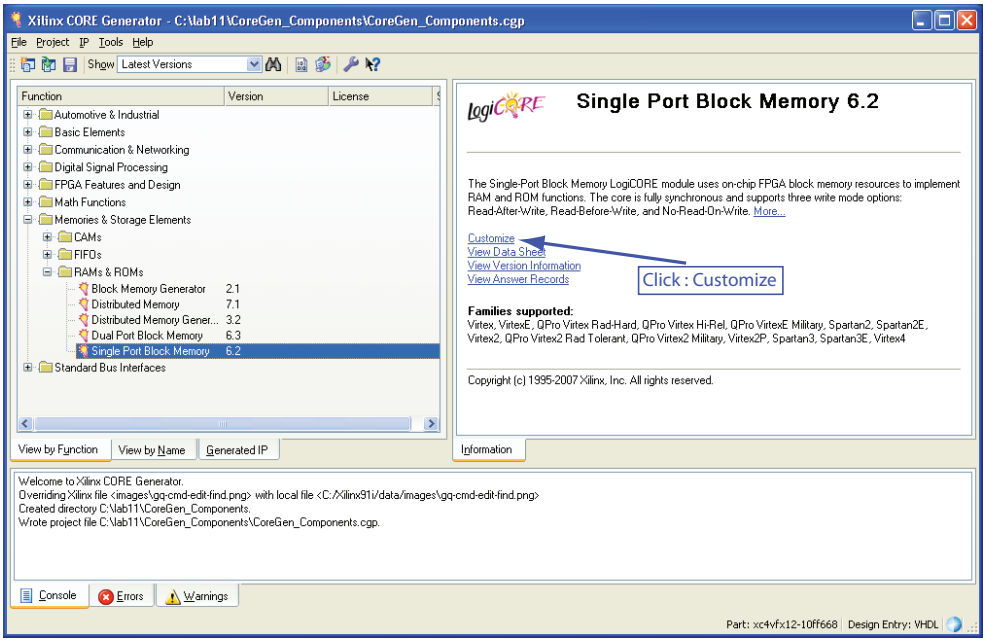
Thus your file which we will name as ROM.coe will have the following two lines:

MEMORY_INITIALIZATION_RADIX = 10;
MEMORY_INITIALIZATION_VECTOR =1, 0, 3, 0, 0, 2;

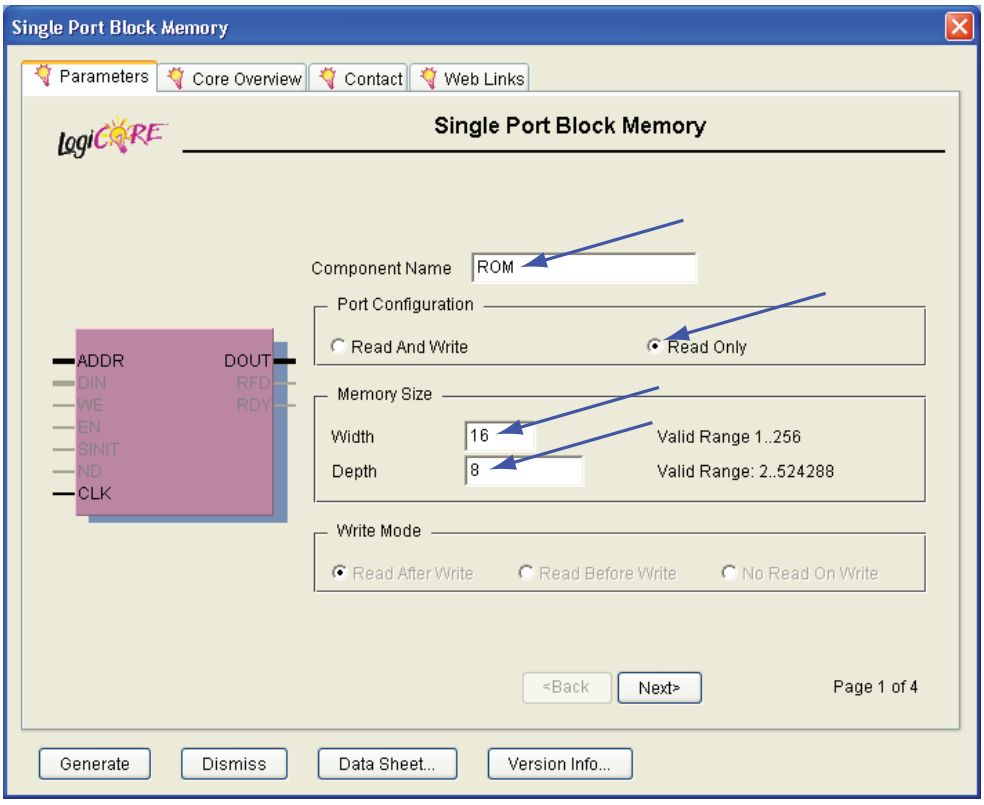
Step 1: Click on the Single Port Block Memory under Memories & Storage Elements / RAMs & ROMs



Step 2: Click Customize




Step 3: Name the component ROM, Select Read Only, Set the Width as 16, and the Depth as 32 (note 8 as in picture) and click Next.

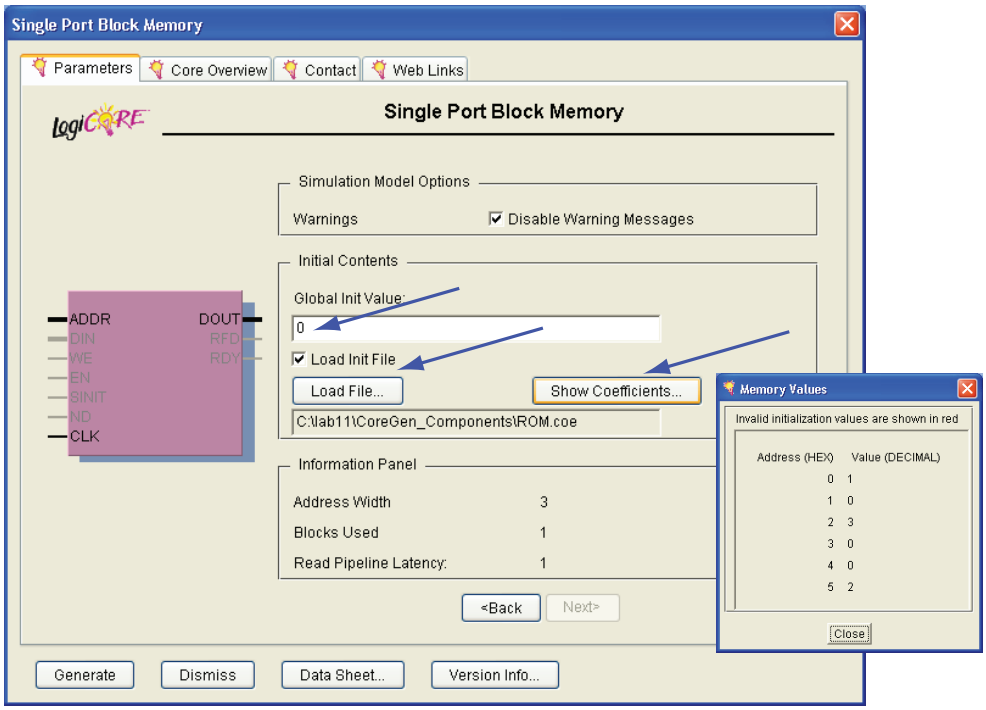


Step 4: Click Next again since we are not going to register the inputs or outputs. You would register the I/O if you were maximizing clock speed and this would shorten the critital path by inserting registers. The tradeoff would be increased latency (additional clock cycles) to get the data where it needed to go.

Step 5: Click Next again as we will accept the default of triggering on a rising clock edge and being active High.

Step 6: Set the Global Init Value as zero (if you don't specify values, they will be zero), Click Load Init File and press the Load File button and browse to ROM.coe and load it in. Press the Show Coefficients button and you should see a window with the values as shown below.

Step 7: Click the Generate button to create the ROM VHDL block.



# Creating the Circular Buffer

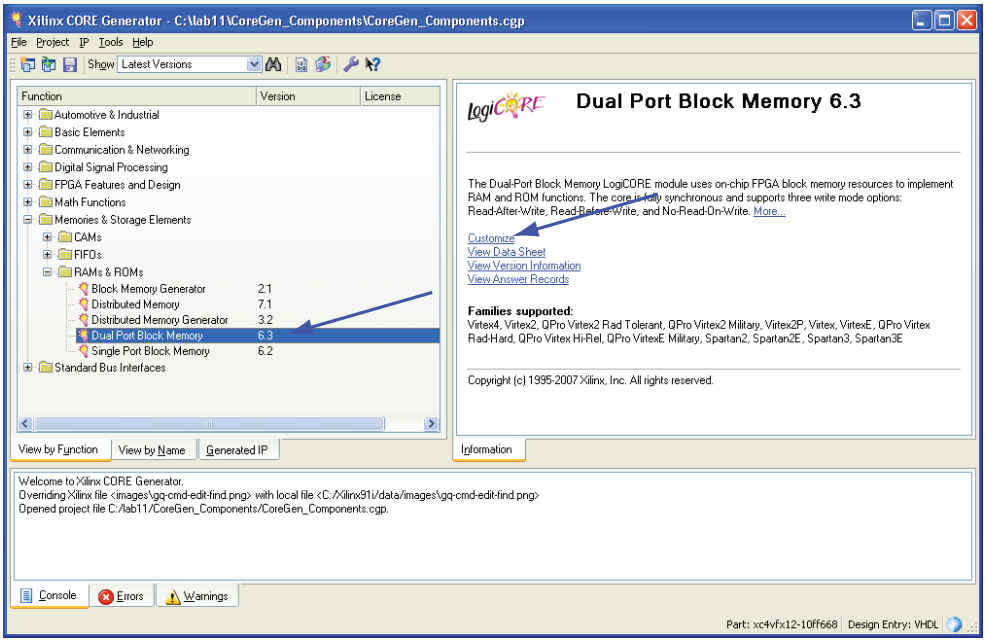
The next step is to create the Circular Buffer by creating a dual-port memory where we can read and write simultaneously.

Open up the CoreGen_Components project in CoreGen that you created previously.

Start→All Programs→Xilinx ISE 9.1i→Accesories→Core Generator

and under “Open existing project” select your CoreGen_Components project.

Step 1: Click on the Dual Port Block Memory under Memories & Storage Elements / RAMs & ROMs  and Click Customize.



Step 2: i.) Name the Component “Circular_Buffer”.

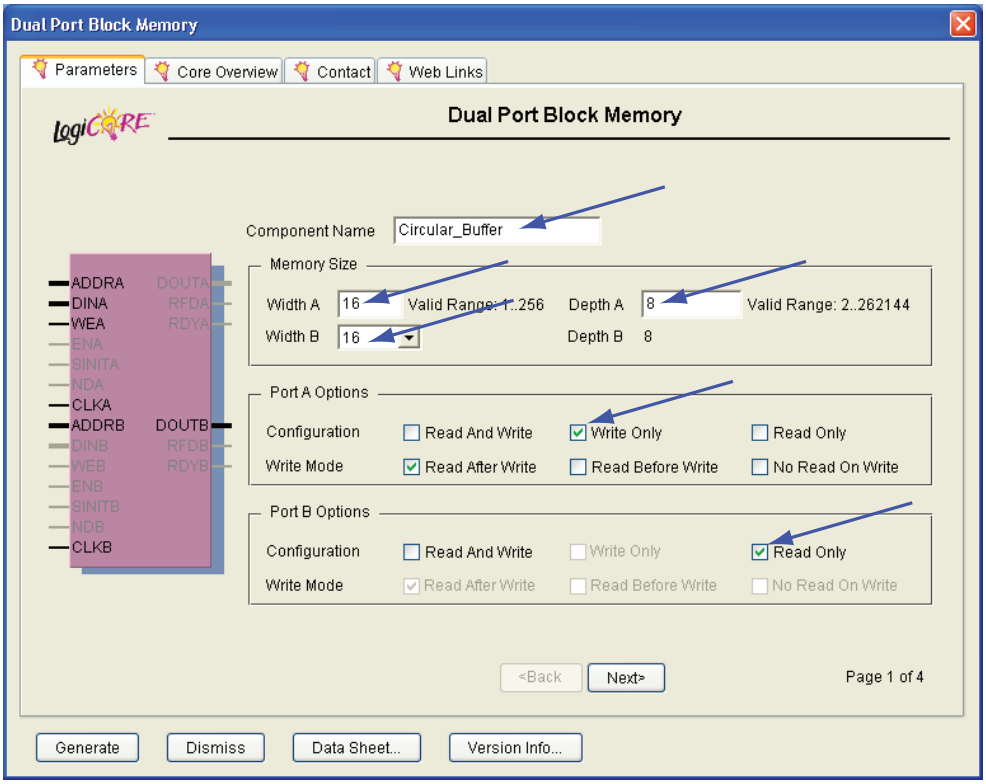
ii.) Set Width A&B to 16 since our data size is 16 bits.

iii.) Set Depth A&B to 32 (not 8) since we only need to have 21 locations and its best to pick a value that is a power of 2. The depth will depend on the order of the filter and the maximum delay that needs to be implemented.

iv.) Make Port A to be “Write Only”. Select Write Mode as Read After Write. The write mode says on collision to read the data before or after the data is written to the memory, i.e. read the original data or the data being written.

v.) Make Port B to be “Read Only”

Click Next.




Step 3: Click Next again since we won’t register I/O on Port A and we will accept the port being rising edge triggered.

Step 4: Click Next again since we won’t register I/O on Port B and we will accept the port being rising edge triggered.

Step 5: Click Generate since we are not going to initialize the memory as we did for the ROM.

## Creating the Multiplier

Step 1: Click on Multipliers under Math Functions, then Multipliers and Click Customize.
Step 2: i.) Name the Component "multiplier".
ii.) Select Parallel Multiplier
iii.) Set the inputs to be 16-bit, signed.
Click Next.
Step 3: i.) Under Multiplier Construction select Use Mults (this will use any hardware multipli ers available).
ii.) Select Area Optimized .
Click Next.
Step 3: i.) Use the default output width of (31..0)
ii.) Select Pipeline Stages as 3.
Click Finish

## ~~Creating the Adder~~ (see below)

Step 1: Click on Adders & Subtractors under Math Functions and Click Customize.
Step 2: i.) Name the Component "Adder".
ii.) Select Operation as just Add
iii.) Set the Port A input to be 32-bit, signed.
iv.) Set the Port B input to be 40-bit, signed.
Click Next.
Step 3: i.) Set the output to be Registered, a Latency of 1, and output width of 40.
Click Generate

**Note: ISE 10.1 no longer has the Adder core** (presumably because the synthesis tools now can create optimal adders from VHDL code). You have two options here:

1. Write the VHDL code to create the adder.
2. You will be allowed to create a MAC core to implement the multiplier-accumulator. You can do this in the Xilinx CORE Generator by selecting Math Functions -> Multiply Accumulators (which is also found under Digital Signal Processing).

We now need to add the components (ROM, memory, multiplier, adder) that we created with the Xilinx core generator to your workspace. To do this, Under the design browser, (left column with Files tab select at bottom) right click "Add New File" and then "Add Files to Design" and select the .vhd files of the Rom, circular buffer, multipler and adder.

This will allow you to easily add the Declarations and Instantiations into your code by expanding the *.vhd block and right clicking on the E/A block and selecting either "Copy Declaration" or "Copy VHDL Instantiation" and pasting it into your code in the appropriate places.

This should be enough to get you started. Your job now is to write the FIR VHDL architecture code that will implement the FIR equation given on the first page. You will need to instantiate the blocks, hook them together, and create the appropriate control signals to coordinate the MAC operations.

**Note: You will need to implement the following processes in your design:**

**1.** Three processes for your control state machine. You will need three states:
**state_wait** - to wait for the next data to arrive
**state_MAC_count** - in this state you will be incrementing the address generators to read your coefficients and data (these counters will need an enable signal provided in this date), and performing the MAC operations
**state_MAC_done** - here you will signal when the output is ready (you will need to account for any latency of the MAC unit due to pipeling) and increment the write pointer for your circular buffer to the next input location.
**2.** A process (counter) to implement the write pointer (address) to the circular buffer.
**3.** A process to implement the address to read the coefficient and data memories.
**4.** A process to signal when the MAC is done.
**5.** A process that determines when the accumulator is cleared
**6.** A process to register the output data y[n] when the MAC is done.

**For verification purposes**, you will need to do the following:

1. Show a simulation where your testbench will input a value 1 and then 30 zeros at the input signal.

A. What values to you see coming out of the FIR filter? They should be the coefficient values. Compare your Matlab coefficient values in hex with your simulation values in hex.

2. How fast can you clock your FIR entity? Provide a print-out of the timing report and explicitly state how fast your design can run. From this clock speed, determine that fastest data rate that this filter can accept.

# Instructor Verification Sheet
# (Lab 6)

Name:_____ Date:_____

1. Plot the frequency response of the FIR filter using freqz().

   Verified:_____ Date:_____

2. Show the simulation of the FIR filter responding to an inpulse. Show that the filter output is the coefficient values.

   Verified:_____ Date:_____

3. Show the timing report.

   Verified:_____ Date:_____

**Turn in this verification sheet with the following:**

1. Plot of the frequency response
2. Printout of the timing report
3. Answer the question as to the maximum data rate the filter can accept.
4. A printout of your VHDL code.