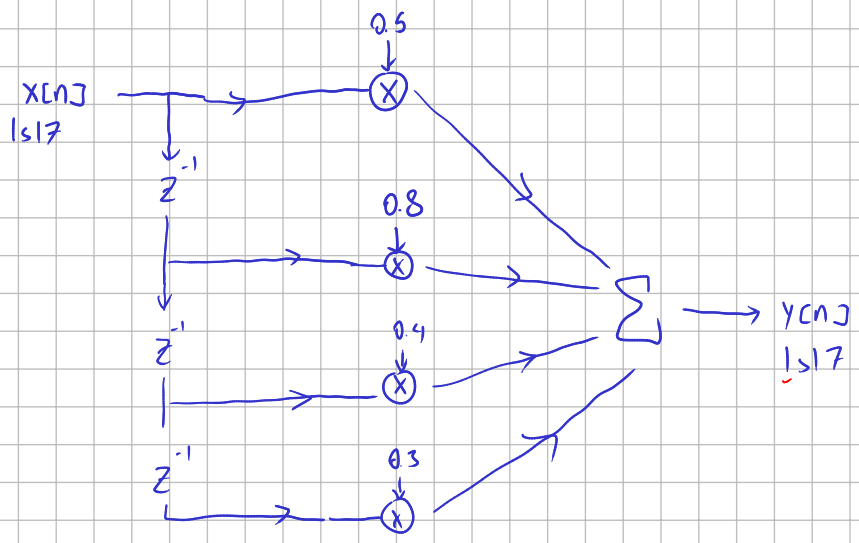
Example: 4-tap filter: h[n] = [0.5 0.8 0.4 0.3]

Draw Direct form of the above filter



Max positive #: $1 - 2^{-17}$

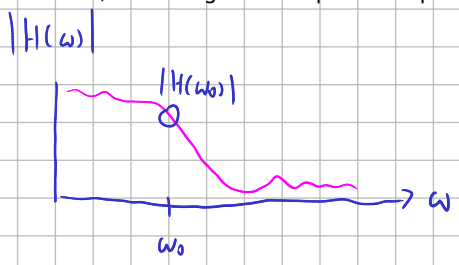
Max negative #: $(-2)^{(1-1)}$

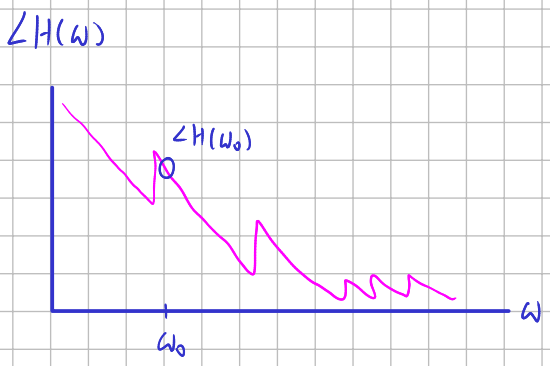
Frequency response:

Note that FIR filters are linear system, meaning that if the input is a sinusoid, the output will be a sinusoid at the same frequency BUT with a different magnitude and possibly a phase offset

Freq resp: $H(\omega) = \sum_{n=0}^{3} h[n]\, e^{-j\omega n}$

Example of typical LPF below, NOT magnitude & phase response of above:





How can we interpret the above graphs to help us design our filter?

If we have a filter and the input is a sinusoid @ a frequency of w0, our output will be a sinusoid with a frequency @ w0 BUT it will have a gain of |H(w0)| and a phase shift of <H(w0)

Sine, A = 1 → h[n] → Sine, $A = |H(\omega_0)|$

$f = \omega_0$ → $f = \omega_0 + \angle H(\omega_0)$

Q: What happens if we scale our coefficents? From pg.101, we need to scale the coefficients so that the max magnitude response is exactly 1

(Since the max mag resp is 1, that means our gain is typically <= 1; since the gain is less than 1, than the output can be a 1s17 format.)

A: By scaling our coefficients by a factor of k, we also scale the frequency response by the same constant k

(Magnitude response is 1 aspect of the frequency response)

If we scale the coefficients of our system, does it not affect the gain of the system?

Ex: a = b + c

1) Determine bit width of LHS

2) Look at RHS to determine signed vs unsigned operation
   -Both operands must be signed in order to perform a signed operation
   -If only one operand is signed, than the overal operation is unsigned

3) Extend RHS operands to width found in step 1
   -Zero or sign extension (which depends on outcome of step 2)
   -zero extension for unsigned
   -sign extension for signed

4) Add operands

5) Assign to LHS variable

Determine signal format of b & c

Comparators (p.36)

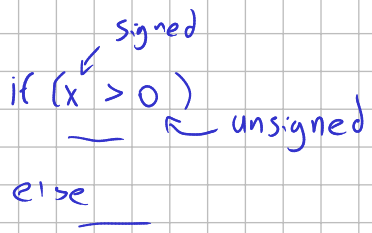
(*comparators are common to give errors)

The rules for how comparators are built are the same as the adders

For a signed comparison, both operands must be signed

if one operand is unsigned, then it is a unsigned comparison

Make sure to check if your operands are signed or unsigned in a comparator
The key thing to watch for is that constants are unsigned by default



If widths of your operands are different, then the extension is performed based on whether your operands are signed or not (just like above)

Multipliers (p.37)

If one or both inputs are unsigned, then an unsigned multiplier is constructed

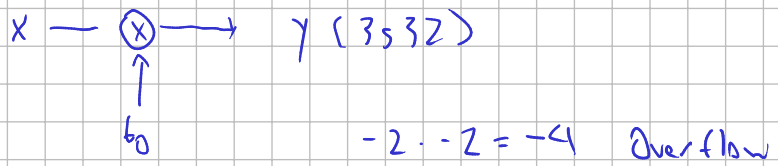
ex: 2s16 * 2s16 = 4s32

4s32 can store numbers from -8 to 8-(2^-32)

In most cases in this class, we have some input and we multiple it by some constant b0

we should know what value b0 is and typically b0 is not equal to the maximum full scale negative value

if the above is true, then no matter the input, we will never have to take the product of 2 full scale negative values and get overflow:



since this case is rare, we can safely discard the MSB of the output

however with unsigned products we NEED all the bits

verilog is always biased to unsigned numbers (p.37)

Debugging (Bonus lecture 3 part 1)

After you created your verilog design, you'll need to debug it

There are 3 main methods to debug an FGPA design

Functional simulation
Timing simulation
JTAG debugging/signal tap

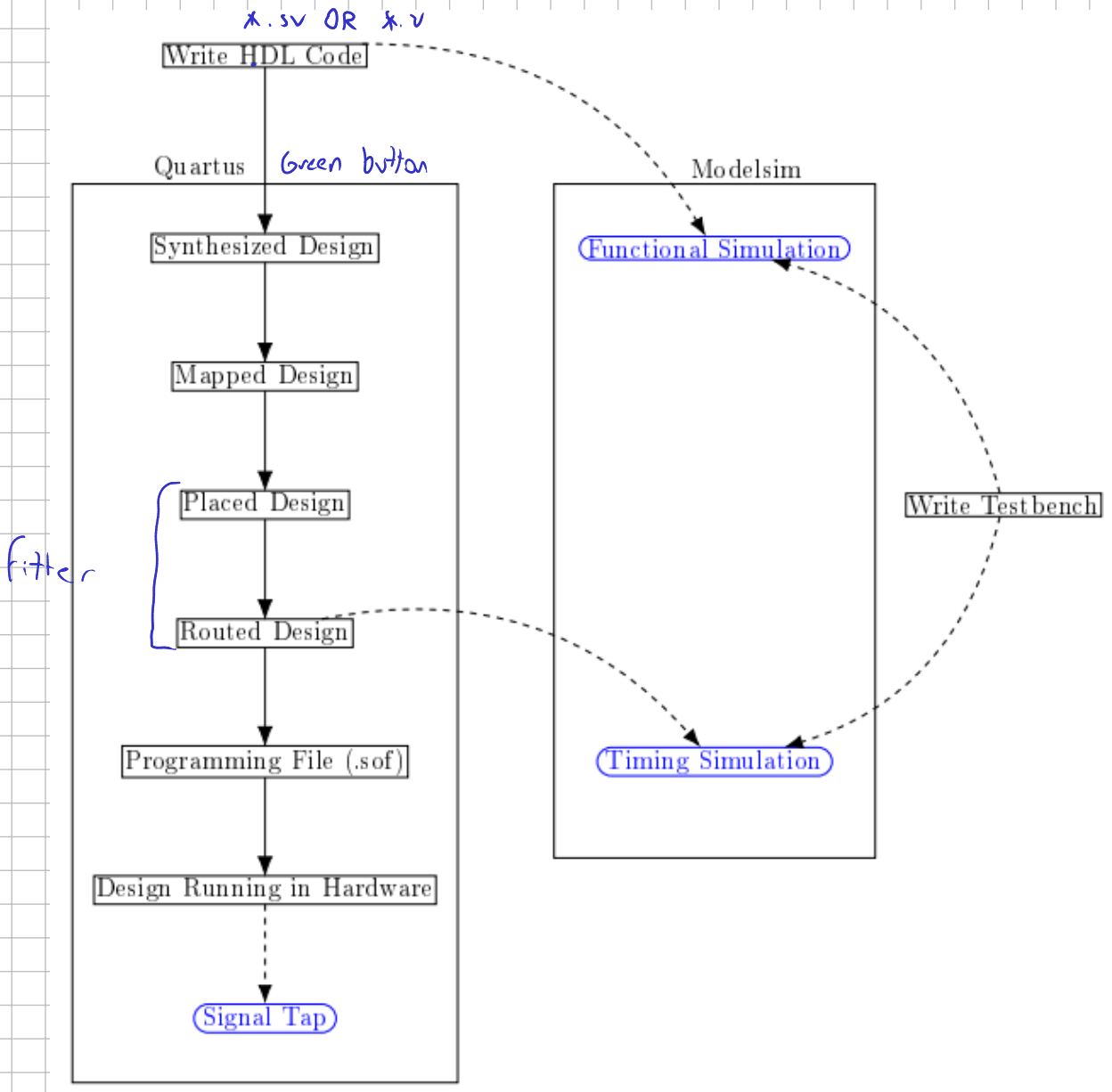
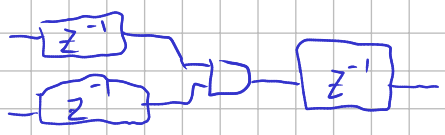


Figure 1.15: Methods of debugging an FPGA design.

In "synthesized design" takes your verilog code and sketches an optimized block diagram
ex:



Quartus will also optimized your design where it will remove logic that is not used.

Debugging FPGA cont'd

Next stage is Mapped design, where the mapper takes synthesized design and translate it into specific hardware elements that are in your FPGA

It tries to translate the synthesis design to a low level block diagram

In this example, the FPGA builds a series of flip-flops and it will map each of the registers from the synthesis diagram into flip-flops

our FPGA does not have specific AND gates within but rather Look up tables, which can be configued in various ways to implement any logic circuity we would like to implement. It may decide it needs 1 LUT to implement an AND gate and it will determined the opcodes for the LUT to simulate an AND gate

depending on the FPGA, we might have different operations for LUT or sizes, specialized circuits like multipliers, block memory etc.

Mapper takes all the available hardware to build this "Mapping" block diagram

Once we have this Mapping block diagram, the complexity begins in the placer and router

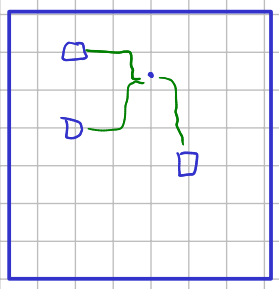
Quartus uses the term "fitter" in the compliation report, where its a placement and router operation

To implement this design from our maper into an FPGA, we need to figure out which of the elements in the FGPA should be used to represent these individual flip-flops from our design

Remember that an FGPA can be viewed as a big grid of logic elements, which can programaticaly connected in run-time

Each of these lines represents a row or column of flip-flop elements. Our DE2 board has about 100k LUT and 100k FLip flops

This quartus software has to figure which flip flop and LUT should be used to implement the mapped design



In this case, we need 3 flip flops for our design so the number of possibilities for the flip flops to be place is roughly 100k^3. On the left shows just one possible selection

The next stage is routing, which involves connecting all these elements together. We need to use the programmable routing tracks to connect the various signals togther as shown in green.

Quartus can create any path from each hardware element to the next. This is important as there are propagation delays in these tracks and elements in the FPGA.

In order for our design to work correctly, we need to match the propagation dely to the sped of operation of our circuit.

Some logic might need to work quickly thus it needs a more direct route.

To direct the operation of the fitter (placing and routing steps), we need to create an SDC file. This file tells the objectives and goals needed when it comes to routing these elements

the output of the routed design is passed into a programming file (.sof file) to do the implementation on FPGA

we can use our jtag programmer to load the programming file into FPGA, now we have a design running in FPGA hardware

we can then use signal tap to capture the signals in real-time for debugging.

Another method for debugging is related to simulation, where we use modelsim.\

Modelsim allows us to generate any stimulus we want and see how our design responds. These simulations are more flexible than signal tap.

2 ways of simulating:
-Timing simulation: after Quartus compiles takes place, the output from routing stage (contains all info aobut how the FPGA implements), generates a .vo file, which is passed into modelsim for the timing simulation

the timing simulation can duplicate all the delays in the FGPA when it is running, it knows the how much propagation delay there will be when each signal is toggling

In functional simulation, we take the verilog file and directly pass it into Modelsim and complie it

for either functional or timing simulation, we need to write the testbench

With a functional simulation, we dont need to compile the design thru quartus at all for simulation or debugging. However we won't knkow what the propagation delay will be in our design

Functional simulation assumes signals will transition instaneously on clock edges, but we don't need to go thru a quartus compliation and determine its behaviour. This will help speed up things for u since the fitter operation is expensive and time consuming as your design gets larger

As this class progress, the compliations will take quite a long time, compiles may take up to 5 mins just to be routed.

Everytime a change is made, it takes time to compile and process in quartus but in modelsim we can just run in instantly.

This can help us debug issues.

In industry, designs are more complicated and time is important. Some compiles could take an hour to get the .sof file

The dominant method of debuggin in industry is done thru functional simulation.

Introduction to Testbenches:

Recommended to use functional simulation. Should get comfortable with doing cfunctional simulation ASAP.

The testbench is a standard verilog module but it has no inputs or outputs.

the 3 components in this testbench are:

*Most important: The instantiation of the verilog model to test AKA top level module

stimulus generation: generates the signals for the inputs of the DUT

The outputs from the DUT are connected to the monitor within the testbench which is displayed as a wave in modelsim or as a textfile.

In industry, the outputs are compared with expected outputs and the testbench will specify if it failed or not but we wont get here

-Module is generally named based on the design that is being tested ex: to test a module, "aaa", our testbench would be name aaa_tb

-Testbenches have NO inputs or outputs

-TB just have internal signals that connect to the ports of the DUT

-Since testbenches don't run inside the FPGA, they don't need to rely on synthesizable code constructs, we can use powerful tools like fork join

The most difficult part of writing a tb is figuring out what are the critical test scenarios that must be exercised to verify the DUT

Can't exhaustively test all possible inputs/signal timings
-just aim for worst-case/most stressful combinations of input signals
-proper testbench design requires a solid understanding of the design requirements

Try to set your DUT code aside and treat it as a black-box test

some ways to test the SRRC filter:

1) Test Impulse Response; input is an impulse and see if the impulse response of filter matches that IR in matlab

2) Sinusoid test: use matlab and FR analysis, you shouod be able to predict the output sinsuoid's output and phase

3) Test worse case scenario: Need to make sure no overflow occurs, given the worst case input sequence

4) Test the reset: reset the filter during its simulation

Modelsim debugging tips (p.31)