Gentle Intro

Yehowshua Immanuel

May 11, 2020

This is your first assignment. It is not too hard and its worth a few points.

Instead of copying and pasting text included in this lab, you can download all the files below here.

Background Information

Submission Format

You are encouraged to write your documents in Markdown and render them into PDFs using LaTeX. You can submit your assignment PDFs via Pandoc. See more on how to do that here.

Questions that must be answered in your submission are of the formate QX.X

What is RTL or HDL?

A Hardware Description Language(HDL) is purely that. It merely describes hardware. There are a couple types of HDLs, digital and mixed(digital+analog). This class deals with digital HDLs.

The digital subset of HDL is usually referred to as Register Transfer Logic(RTL).

RTLs aren't very useful by themselves. Typically, they might be passed to a synthesizer which reduces the RTL into a gate list that describes all the connections between various gates. A gate list is more commonly known as a netlist.

Synthesizers

There are different kinds of synthesizers. Some synthesizer frameworks target FPGAs while others target physical fabrication or VLSI. In VLSI, RTL is just the first step in a long laborious sequence of tasks that usually results in a finished physical chip.

All synthesizers typically support both VHDL and Verilog. Writing Verilog or VHDL is very primitive however, so it is not uncommon to use a higher level language to generate Verilog (I heard somewhere that Perl is sometimes used in industry to generate Verilog).

Higher Level Languages and Simulation

Using a higher level language can make complex tasks such as generating an out-of-order, multi-cache, multi-issue, pipelined processor easier.

Today, languages such as Scala's Chisel or Python's nMigen can be used to reliably generate and internally simulate RTL.

After writing some RTL, you may wish to know whether or not it does what you want. This can be accomplished using an RTL simulator. Currently, there are three Free and Open Source RTL simulators.

Language	Simulator
VHDL	GHDL
Verilog	Icarus Verilog Simulator
Verilog	Verilator

We use Icarus in this class, because although it is slower than Verilator, it has advanced debug capabilities, which become particularly useful when combined with the Python frontend Cocotb.

For example, we can access the content of memory arrays in Verilog submodules with Cocotb+Icarus.

Questions

Q1

What is the first thing that happens when you turn on your processor?

Please refer to page 25, section 5.2.4 of the MIPS32 Manual Volume III [1].

Q2

Computer programs can have many different sub-routines spread amongst different files. RTLs like verilog are no different. But instead of sub-routines, textual representations of physical hardware can be separated into distinct components called modules.

Below we have a synchronous adder module.

```
module adder(sum, a, b, clk, reset);

input wire clk, reset;
input wire [7:0] a;
input wire [7:0] b;
output reg [7:0] sum;

always @(posedge clk)
    if (reset == 1)
        sum <= 0;
    else
        sum <= a+b;

endmodule
```

Q2.1

What is the difference between `wire` and `reg`?

Q2.2

How do `wire` and `reg` compare to System Verilog's `logic`?

[1] https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol3.pdf

Q2.3

What do you think happens when we assert `reset`?

3

Now lets simulate the Verilog adder from above. First make sure you follow the instructions to install Cocotb and the icarus verilog simulator backend as detailed here.

create the files

Next modify the end of your Verilog from above so that it can write the simulation results to a waveform VCD file as shown below. Name the file `sync_adder.v`.

sync_adder.v

```
module adder(sum, a, b, clk, reset);

input wire clk, reset;
input  wire [7:0] a;
input  wire [7:0] b;
output reg [7:0] sum;

always @(posedge clk)
    if (reset == 1)
        sum <= 0;
    else
        sum <= a+b;

// the "macro" to dump signals
`ifdef COCOTB_SIM
integer num_regs;
initial begin
    $dumpfile ("sim.vcd");
    $dumpvars (0, adder);
end
`endif

endmodule
```

Next, create the makefile. Simply name it `makefile` without an extension. See What are Makefiles? for more information on makefiles.

makefile

```
VERILOG_SOURCES = sync_adder.v
SIM=icarus
TOPLEVEL_LANG=verilog
COCOTB_REDUCED_LOG_FMT=1
SIM_BUILD = out
#COMPILE_ARGS =

# TOPLEVEL is the name of the toplevel module in your VHDL file:
TOPLEVEL=adder
```

```
# MODULE is the name of the Python test file:
MODULE=tb

#included for cocotb
include $(shell cocotb-config --makefiles)/Makefile.sim
```

Lastly, create the testbench file named `tb.py`.

tb.py

```python
import cocotb
from cocotb.triggers import Timer, ClockCycles, FallingEdge, RisingEdge
from cocotb.clock import Clock

@cocotb.test()
def test_adder(dut):
    #cocotb is lazy and event driven. We can give cocotb
    #a list of events to do, but it won't do them
    #until we request to see Cocotb's state at
    #some point in time.

    #below, we tell cocotb there's an event
    #that repeats every nanosecond, namely
    #the ticking of clock
    cocotb.fork(Clock(dut.clk,1,'ns').start())

    #we set reset high
    dut.reset = 1;

    #simulation is currently at time zero
    #we force cocotb to advance simulation until
    #clock has ticked exactly once
    yield cocotb.triggers.ClockCycles(dut.clk, 1, 'ns')

    dut.reset = 0;
    dut.a = 1
    dut.b = 2

    #Timer() processes any queued events until the
    #specified time expires
    #Since we forked a tick process above with a clock
    #period of 1ns, advancing the sim clock 1ns below
    #with Timer has the same effect as waiting for clock
    #to tick once
    yield Timer(1,'ns')

    #at this point, dut.sum should be 3
    #we can have python test this for us
    #as well as write it to our log
    assert(dut.sum == 3)
    dut._log.info("SUCCESS!!")
```

Place all these files into a folder. You file tree should now look like this:

```
.
|-- makefile
|-- sync_adder.v
|-- tb.py
```

Q3.1 modify the files

Go ahead and type `make` to run the simulation.

Read through the comments in `tb.py` to make sure that you understand what is going on.

Now modify the simulation above to do `4+5=9`. Use an `assert` statement to show its true.

waveform

Go ahead and open up sim.vcd using either `scansion` or `GTKWave`.

Here is a guide to install Scansion or GTKWAVE.

You should see something similar to the screenshot below.

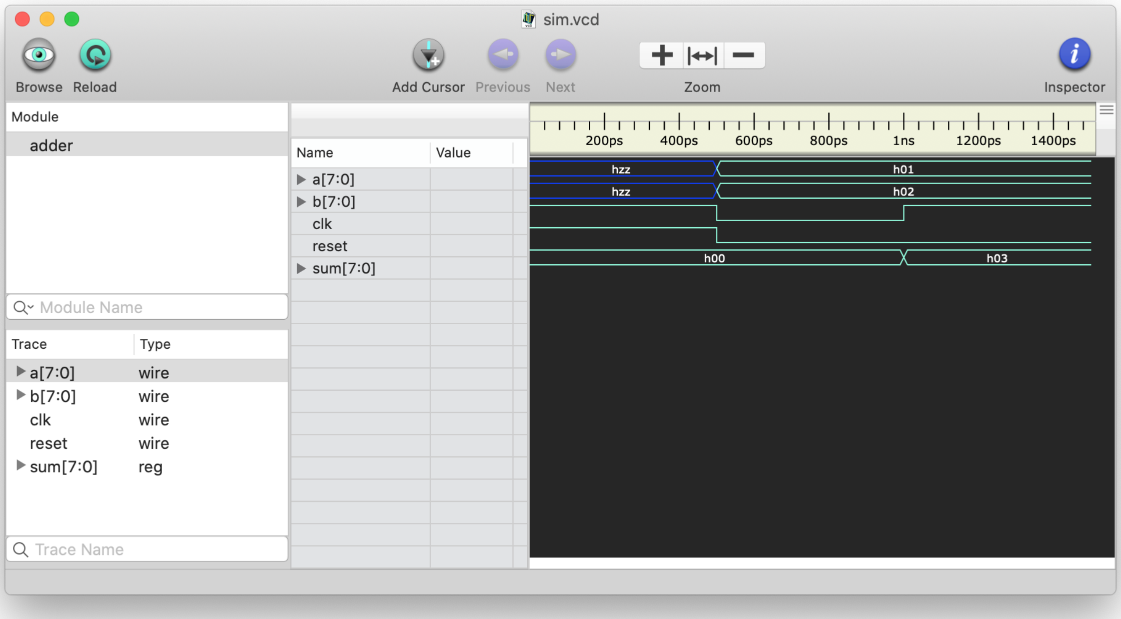


Figure 1: A waveform for a simple synchronous adder

notes

Congratulations!! You just wrote your first testbench. A testbench allows you to test your RTL before you deploy it to a chip or FGPA, and before you mix it with other RTL.

You can also write testbenchs directly using Verilog, but Verilog is not very easy, fast, or pleasant to work with on larger projects.

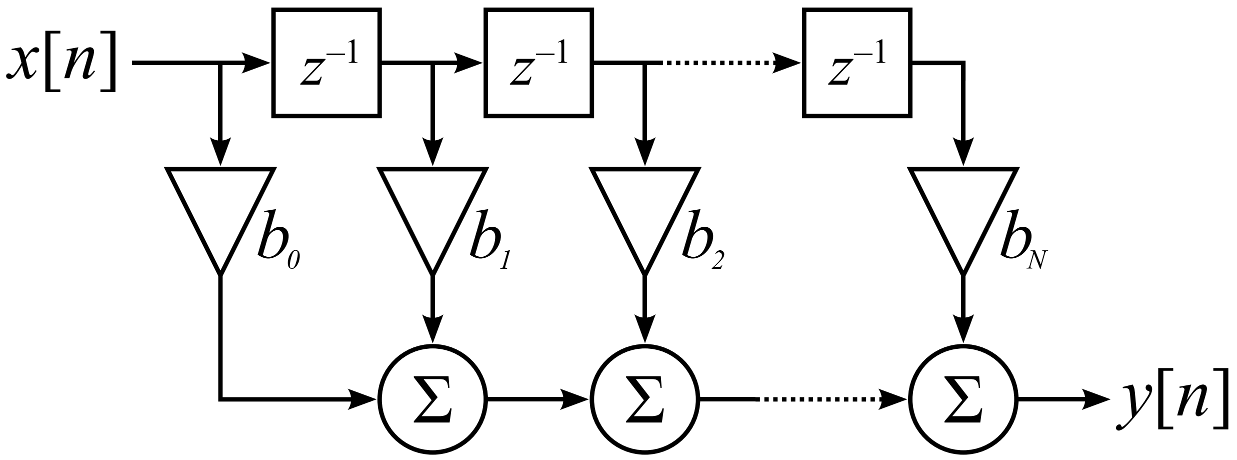
4 nMigen

nMigen is a popular high-level Python based RTL that makes it easier to quickly build and test complex hardware designs.

You can use nMigen Python to simulate RTL and generate verilog.

Please follow this guide to install nMigen.

You may remember FIR filters from ECE 2026, Digital Signal Processing.



$$y[t] = b_0 x[t] + b_1 x[t-1] + \ldots + b_N x[t-N]$$

$$= \sum_{i=0}^{N} b_i \cdot x[t-i]$$

In nMigen Python, this becomes:

```
from nmigen import *
from nmigen.back import verilog

class DSP_chain(Elaboratable):
    def __init__ (self,
                  weights,
                  #set the bit-width of our chain
                  WIDTH
                  ):

        self.params = {"weights":weights, "WIDTH":WIDTH}

        self.x_t = Signal(signed(WIDTH))
        self.y_t = Signal(signed(WIDTH))

    def ports(self):
        return [self.x_t, self.y_t]

    def elaborate(self, platform):
        m = Module()

        reg_array = []
        for el in range(len(self.params["weights"])):
            reg_array += [Signal(
                            signed(self.params["WIDTH"]),
                            name=f"x_t{el}",
                            reset=0)
                            ]

            if el > 0:
                m.d.sync += reg_array[el - 1].eq(reg_array[el])

        m.d.comb += reg_array[-1].eq(self.x_t)

        weighted_sum = sum([el*weight for el,weight in zip(reg_array, self.params["weights"])])
        m.d.comb += self.y_t.eq(weighted_sum)
        return m

f = open("dsp.v", "w")
top = DSP_chain(WIDTH=8, weights=[1,1,1,1])
f.write(verilog.convert(top,
        name='DSP_chain',
        strip_internal_attrs=True,
        ports=top.ports()
        ))
```

It may seem overwhelming at first, but we'll go through it step by step.

step by step

Here is the same file with comments. Try and understand what's happening and ask on Piazza if you need extra help.

```
from nmigen import *
from nmigen.back import verilog

class DSP_chain(Elaboratable):
    def __init__ (self,
                    weights,
                    #set the bit-width of our chain
                    WIDTH
                    ):

        #build a parameters dictionary
        self.params = {"weights":weights, "WIDTH":WIDTH}

        #We allow our input and output to be signed
        #incase we were to have an input containing
        #negative weights
        self.x_t = Signal(signed(WIDTH))
        self.y_t = Signal(signed(WIDTH))

    def ports(self):
        return [self.x_t, self.y_t]

    #platform is used to specify FPGA specifics
    #we don't need to use it here beyond making sure
    #the elaborate function signature is correct for
    #nMigen's AST builder when it is called
    def elaborate(self, platform):
        #a module componenent we can return
        m = Module()

        reg_array = []
        for el in range(len(self.params["weights"])):
            #make an array containing registers
            #x[t], x[t - 1], ... x[t - N]
            reg_array += [Signal(
                            signed(self.params["WIDTH"]),
                            name=f"x_t{el}",
                            reset=0)
                            ]

            #when we advance by a clock cycle,
            #we want to pump values as shown below:
            #x[t] = x[t - 1]
            #x[t - 1] = x[t - 2]
            #x[t - 1] = x[t - 2]
            #etc...
            if el > 0:
                #since this is to occur every clock cycle
                #it is a synchronous event. We add the statement
                #to the sync domain

                #in addition, reg_array is laid out in memory
                #such that reg_array[0] = x[t],
                #thus the statement below does implement
                #x[t] = x[t - 1], though it may not seem that way
                m.d.sync += reg_array[el - 1].eq(reg_array[el])

        #grab the last register in the array and set
        #it equal to the input
        #this occurs instantaneously and is never delayed,
        #so we place it under combinational logic
        m.d.comb += reg_array[-1].eq(self.x_t)

        #we use a list comprhrension to mutiply the weights
        #by the values in the respective registers
        #it expands to wieghted_sum = w[0]*x[t] + w[1]+x[t-1] ...
        weighted_sum = sum([el*weight for el,weight in zip(reg_array, self.params["weights"])])
        m.d.comb += self.y_t.eq(weighted_sum)
        return m

#write the verilog
f = open("dsp.v", "w")
top = DSP_chain(WIDTH=8, weights=[1,1,1,1])
f.write(verilog.convert(top,
        name='DSP_chain',
        strip_internal_attrs=True,
        ports=top.ports()
        ))
```

Q4.1

Go ahead and generate the Verilog.

```
python3 dsp.py
```

Put the generated Verilog in your submission.

Q4.2

Can you tell what type of FIR filter this is?

Hint, the weights are set to $[1, 1, 1]$

nMigen Simulation

We can simulate our DSP chain by adding the following to the bottom of our file.

```
from nmigen.back.pysim import Simulator, Delay, Settle

def testbench():
    signal_in = [1,1,1,1,1,0]
    signal_out = []

    #pump in all the values from the signal_in
    #list, one value per cycle
    for value in signal_in:
        yield dut.x_t.eq(value)
        #collect the values coming out
        #every cycle
        signal_out += [(yield dut.y_t)]

        #tick the clock by one cycle
        yield

    #run the simulation for 5 more cycles
    #after we finish pumping in values
    for cycle in range(5):
        #collect the values coming out
        #every cycle
        signal_out += [(yield dut.y_t)]

        #tick the clock by one cycle
        yield

    print(signal_out)

#make a Module() the root of our simulation to
#enable top signal to show up in VCDs
m = Module()
dut = top
#we use setattr so that the module is
#named in the VCD
#we could also do m.submodules += dut
#but then the module in the VCD wouldn't be named
setattr(m.submodules,f"dsp_chain",dut)
```

```
#instantiate a simulator with 1ns clock
sim = Simulator(m)
sim.add_sync_process(testbench)
sim.add_clock(1/(1e9), domain="sync")

with sim.write_vcd(f"test.vcd", traces=dut.ports()):
    sim.run()
```

run it

```
$python3 dsp.py
[0, 1, 2, 3, 4, 4, 3, 2, 1, 0, 0]
```

Q4.3

What are the values of the input signal in the simulation above?

Q4.5

Please include a screenshot of the VCD Waveform in your submission.

MIPS Refresher

Q5

Encode the following MIPS instructions into 32 bit values.

You can use this reference to help you.

addi \$1, \$2, 16

add \$1, \$2, \$3

beq \$1, \$2, 4