Fall ’02 Project

- Fall ’00 project – designed a standard cell library.
- Fall ’01 project – students picked own project, had to combine custom layout with automated layout, fit within a TinyChip padframe
  - Students were allowed to use Computer Arithmetic project as VLSI project
- Problems with Fall ’01 project is that the projects were too ill-defined, or too large to fit within a TinyChip
- This semester will provide a ‘default’ project that students can do

Requirements for Fall ‘02

- Fall ’02 project requirements
  - Must combine custom layout with automated layout, but the custom layout can be a new standard cell that you add to the library. If new standard cell, then must be fully installed in library and be place/routed with Cadence
  - Design must have two or more APL (automated placed/routed) standard cell blocks, with Cadence routing tool used to connect the standard cell blocks together
  - Any memory arrays must be placed in a reasonable manner (do not just use the standard cell APR to place these cells).
  - Padframe must be placed around the design, with signal/power routing between core and padframe (can be done manually)
  - Must use Spectre to simulate design through pads.
  - Must have a gate level Verilog simulation of design, and show that Verilog and Spectre simulations match.
  - If create own project, must have a high level VHDL or Verilog model that specifies what the design does.
  - Can be 2-person teams or individual effort

Default Project: FIFO

- FIFO: 256 x 4 synchronous FIFO.
- Interface
  - reset – asynchronous reset
  - wclk, rclk – write, read clocks
  - empty_b, full_b - empty, full flags, low true
  - oe - output enable
  - din, dout – data in, data out
  - wen, ren – write enable, read enable
  - test_full - test mode input, used to set FIFO to full state without having to write all locations.
- If you do not use the default project, you must provide me with a project specification via a VHDL or Verilog model

VHDL Model

- Zip Archive contains modelsim/src/fifo library
  - fifo.vhd, fifo_a.vhd - entity, architecture of fifo
  - tb_fifo.vhd - testbench
  - fifo_package.vhd – misc support functions
  - stim.vhd - stimulus entity for testing
  - stim_emptytest.vhd – stimulus architecture for testing empty flags. Just does a few writes followed by reads to see if empty flag works
  - stim_fulltest.vhd - stimulus architecture for testing full conditions. Writes all locations, checks full flag, sees if contents are read back during read
  - stim_rw.vhd - misc testing, uses the ‘test_full’ input to check full condition without writing all locations, checks tri-state condition.

VHDL Model (cont)

- To compile
  - gmake –f fifo/Makefile
- To run the test cases:
  - qhsim –lib fifo cfg_emptytest –c –do “run 30 us;quit”;
  - qhsim –lib fifo cfg_fulltest –c –do “run 30 us;quit”;
  - qhsim –lib fifo cfg_rw –c –do “run 30 us;quit”;
- Use the waveform viewer to see exact clock cycle relationship (use the fifo.do command file for waveform display)
  - The current tests all have the read clock and write clock as the same frequency, I may add more tests later on and/or may change the test bench.
- The FIFO functionality is FIXED. You can start working on your implementation.

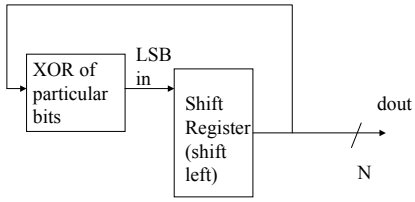
Example FIFO: Cypress Synchronous FIFO

- I have posted a data sheet for a Cypress Synchronous FIFO
  - The timing for our FIFO matches the Cypress FIFO in most respects, but for a definitive clock cycle relationship you must look at the waveforms produced our FIFO VHDL model.
- Use the data sheet as an a general reference on how FIFOs work
  - Our FIFO does not have as much capability as the Cypress FIFO
  - Does not support almost full or almost empty programmable registers.

How Do You Use the provided VHDL Model?

- Use it only as a specification for how it works
  - Do not attempt to simply synthesize it "as is".
- I am going to require that you modify the model so that adders are NOT used to produce the FIFO read/write addresses
  - Currently, the read/write address registers in the FIFO are simply incremented to get to the next address
  - This is a slow and expensive way to produce the next address, especially for large FIFOs. We don't care what sequence of addresses we go through, as long as all addresses are used
  - Use an XOR sequence counter to produce the next read/write addresses. This is much faster and less area expensive than an incrementer.

XOR Sequence Generator



For a 16 bit register,

LSB = dout(1) xor dout(2) xor dout(4) xor dout(15)

This is actually a Primitive Polynomial.

Primitive Polynomial

- Look on the WWW for a primitive polynomial sequence that is 8 bits long
  - One problem is that it will only generate 255 codes, not 256 . One code will be left out
  - You will have to detect the missing code and supply this in your sequence generator.
- If initial value starts at zero, then some valid XOR combinations are (missing code = 255 for all cases)
  - Lsb = 1 xor b(1) xor b(2) xor (3) xor (7)
  - Lsb = 1 xor b(3) xor b(4) xor (5) xor (7)
  - Lsb = 1 xor b(2) xor b(4) xor (6) xor (7)
- Look at psrn_8bit.pl script on web link to see what the sequence is for the above combinations.

Due Date

- Project will due on Monday, December 2nd
- This is the week after the Thanksgiving holidays.