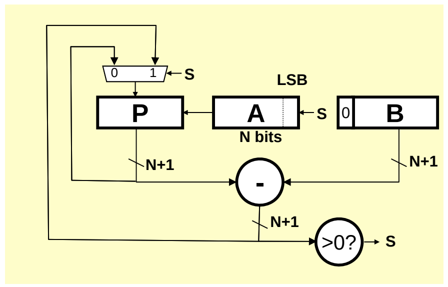
Lab #3 due tonight,
LPSet 8 Thurs 10/11

Pipelining & Verilog

- Latency & Throughput
- Pipelining to increase throughput
- Retiming
- Verilog Math Functions
- Debugging Hints

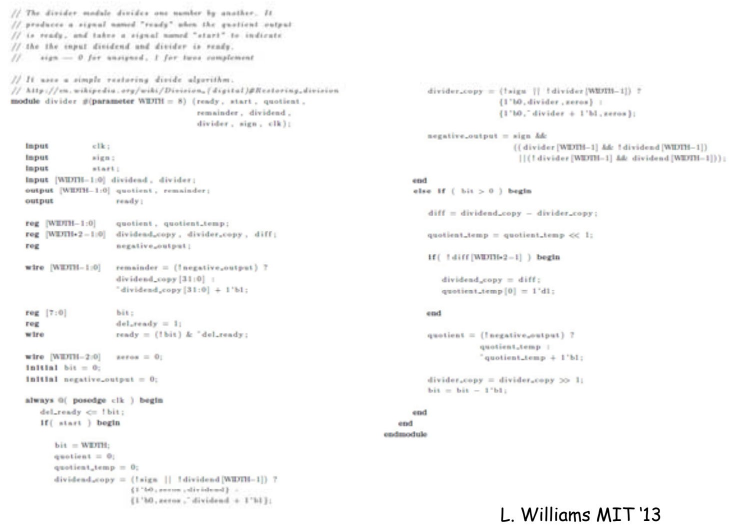
Sequential Divider

Assume the Dividend (A) and the divisor (B) have N bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single subtraction at a time and then cycle the circuit N times. This circuit works on unsigned operands; for signed operands one can remember the signs, make operands positive, then correct sign of result.



```
Init: P←0, load A and B
Repeat N times {
   shift P/A left one bit
   temp = P-B
   if (temp > 0)
     {P←temp, $A_{LSB}$←1}
   else $A_{LSB}$←0
}
Done: Q in A, R in P
```

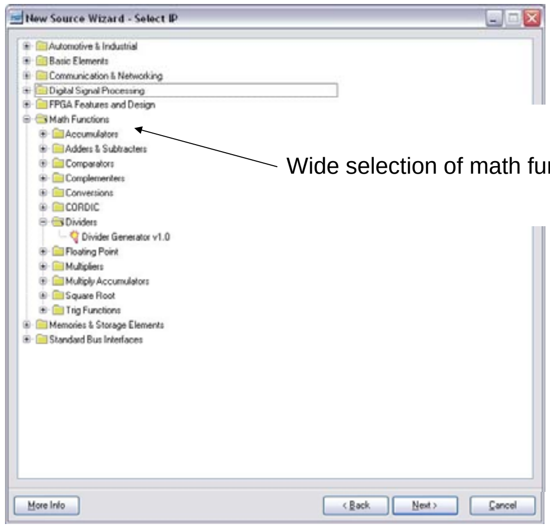
Verilog divider.v



```
// The divider module divides one number by another. It
// produces a signal named "ready" when the quotient output
// is ready, and takes a signal named "start" to indicate
// the the input dividend and divider is ready.
//    sign — 0 for unsigned, 1 for twos complement

// It uses a simple restoring divide algorithm.
// http://en.wikipedia.org/wiki/Division_(digital)#Restoring_division
module divider #(parameter WIDTH = 8) (ready, start, quotient,
                                       remainder, dividend,
                                       divider, sign, clk);

   input            clk;
   input            sign;
   input            start;
   input  [WIDTH-1:0] dividend, divider;
   output [WIDTH-1:0] quotient, remainder;
   output             ready;

   reg  [WIDTH-1:0]    quotient, quotient_temp;
   reg  [WIDTH*2-1:0]  dividend_copy, divider_copy, diff;
   reg                 negative_output;

   wire [WIDTH-1:0]    remainder = (!negative_output) ?
                       dividend_copy[31:0] :
                       ~dividend_copy[31:0] + 1'b1;

   reg  [7:0]          bit;
   reg                 del_ready = 1;
   wire                ready = (!bit) & ~del_ready;

   wire [WIDTH-2:0]    zeros = 0;
   initial bit = 0;
   initial negative_output = 0;

   always @( posedge clk ) begin
      del_ready <= !bit;
      if( start ) begin

         bit = WIDTH;
         quotient = 0;
         quotient_temp = 0;
         dividend_copy = (!sign || !dividend[WIDTH-1]) ?
                         {1'b0,zeros,dividend} :
                         {1'b0,zeros,~dividend + 1'b1};
         divider_copy = (!sign || !divider[WIDTH-1]) ?
                        {1'b0,divider,zeros} :
                        {1'b0,~divider + 1'b1,zeros};

         negative_output = sign &&
                           ((divider[WIDTH-1] && !dividend[WIDTH-1])
                            ||(!divider[WIDTH-1] && dividend[WIDTH-1]));

      end
      else if ( bit > 0 ) begin

         diff = dividend_copy - divider_copy;

         quotient_temp = quotient_temp << 1;

         if( !diff[WIDTH*2-1] ) begin

            dividend_copy = diff;
            quotient_temp[0] = 1'd1;

         end

         quotient = (!negative_output) ?
                    quotient_temp :
                    ~quotient_temp + 1'b1;

         divider_copy = divider_copy >> 1;
         bit = bit - 1'b1;

      end
   end
endmodule
```

L. Williams MIT '13

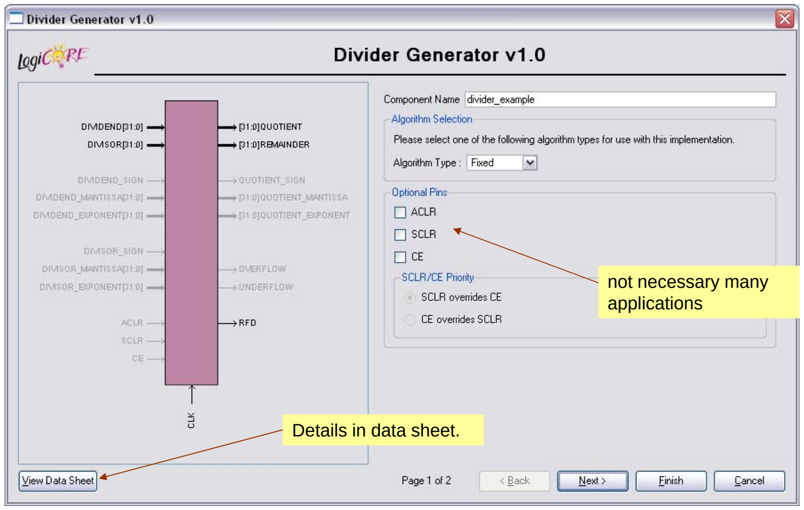
Math Functions in Coregen



```
Automotive & Industrial
Basic Elements
Communication & Networking
Digital Signal Processing
FPGA Features and Design
Math Functions
   Accumulators
   Adders & Subtracters
   Comparators
   Complementers
   Conversions
   CORDIC
   Dividers
      Divider Generator v1.0
   Floating Point
   Multipliers
   Multiply Accumulators
   Square Root
   Trig Functions
Memories & Storage Elements
Standard Bus Interfaces
```

Wide selection of math functions available

Coregen Divider


Divider Generator v1.0

Divider Generator v1.0

Component Name: divider_example

Algorithm Selection

Please select one of the following algorithm types for use with this implementation.

Algorithm Type: Fixed

Optional Pins

- [ ] ACLR
- [ ] SCLR
- [ ] CE

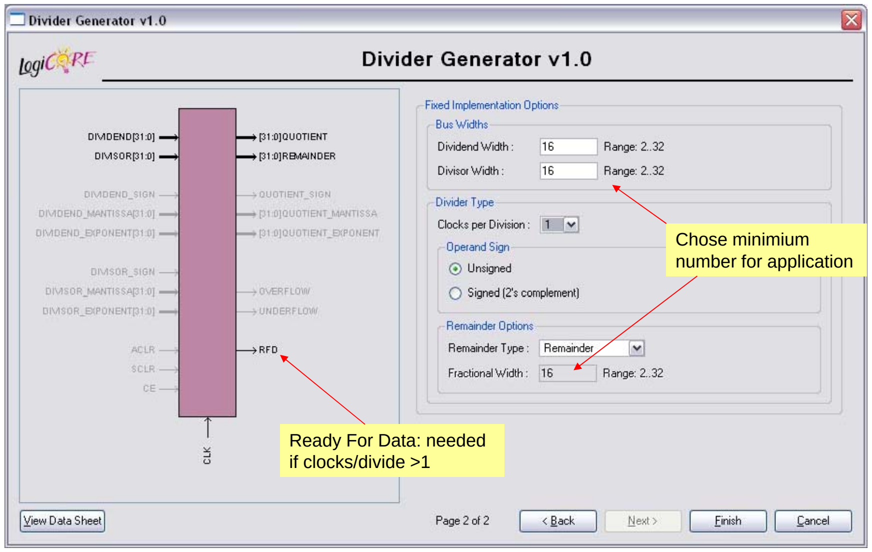
SCLR/CE Priority

- SCLR overrides CE
- CE overrides SCLR

not necessary many applications

Details in data sheet.

Coregen Divider


Divider Generator v1.0

Divider Generator v1.0

Fixed Implementation Options

Bus Widths

Dividend Width: 16 Range: 2..32

Divisor Width: 16 Range: 2..32

Divider Type

Clocks per Division: 1

Operand Sign

[✓ Unsigned]

Signed (2's complement)

Remainder Options

Remainder Type: Remainder

Fractional Width: 16 Range: 2..32

Chose minimium number for application

Ready For Data: needed if clocks/divide >1

Performance Metrics for Circuits

Circuit Latency (L): time between arrival of new input and generation of corresponding output.

For combinational circuits this is just t_{PD}.

Circuit Throughput (T): Rate at which new outputs appear.

For combinational circuits this is just $1/t_{PD}$ or $1/L$.

Coregen Divider Latency

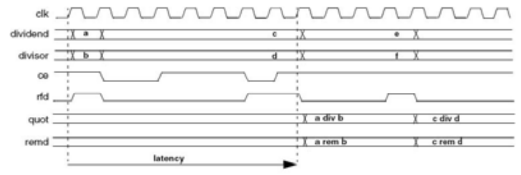


Figure 2: **Latency Example (Clocks per Division = 4)**

Table 4: **Latency of Fixed-point Solution Based on Divider Parameters**

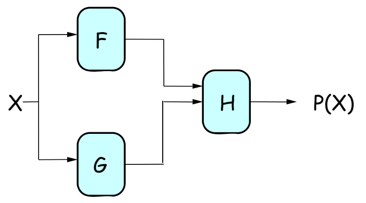
Signed	Fractional	Clks/Div	Latency
False	False	1	M+2
False	False	>1	M+3
False	True	1	M+F+2
False	True	>1	M+F+3
True	False	1	M+4
True	False	>1	M+5
True	True	1	M+F+4
True	True	>1	M+F+5

Note: M=dividend width, F=fractional remainder width.

Latency dependent on dividend width + fractioanl reminder width

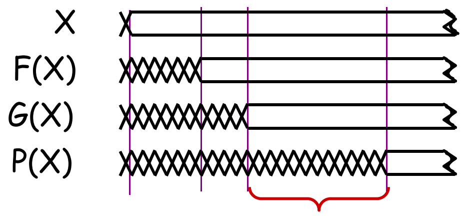
The divclk_sel parameter allows a range of choices of throughput versus area. With divclk_sel = 1, the core is fully pipelined, so it will have maximal throughput of one division per clock cycle, but will occupy the most area. The divclk_sel selections of 2, 4 and 8 reduce the throughput by those respective factors for smaller core sizes.

Performance of Combinational Circuits



For combinational logic:
$L = t_{PD}$,
$T = 1/t_{PD}$.

We can't get the answer faster, but are we making effective use of our hardware at all times?

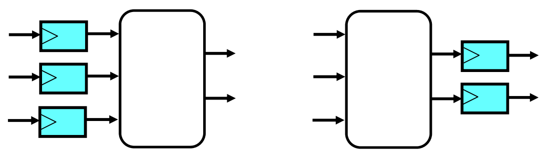


F & G are "idle", just holding their outputs stable while H performs its computation

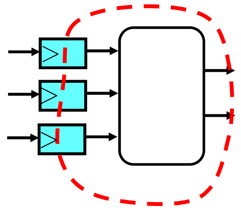
Retiming: A very useful transform

Retiming is the action of moving registers around in the system

- Registers have to be moved from ALL inputs to ALL outputs or vice versa

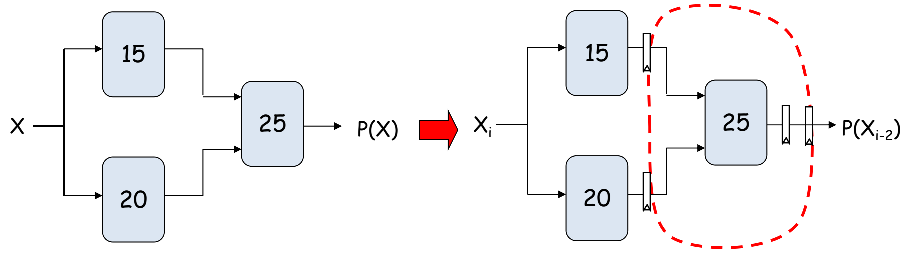


Cutset retiming: A cutset intersects the edges, such that this would result in two disjoint partitions of the edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.



Benefits of retiming:
- Modify critical path delay
- Reduce total number of registers

Retiming Combinational Circuits aka "Pipelining"

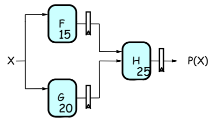


Assuming ideal registers: i.e., $t_{PD} = 0$, $t_{SETUP} = 0$

$t_{CLK} = 25$
$L = 2^*t_{CLK} = 50$
$T = 1/t_{CLK} = 1/25$

L = 45
T = 1/45

Pipeline diagrams



Clock cycle →

Pipeline stages ↓

	i	i+1	i+2	i+3	
Input	X_i	X_{i+1}	X_{i+2}	X_{i+3}	…
F Reg G Reg		$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$	$F(X_{i+2})$ $G(X_{i+2})$	…
H Reg			$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Conventions

DEFINITION:

a *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

a COMBINATIONAL CIRCUIT is thus an 0-stage pipeline.

CONVENTION:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *OUTPUT* (not on its input).

ALWAYS:

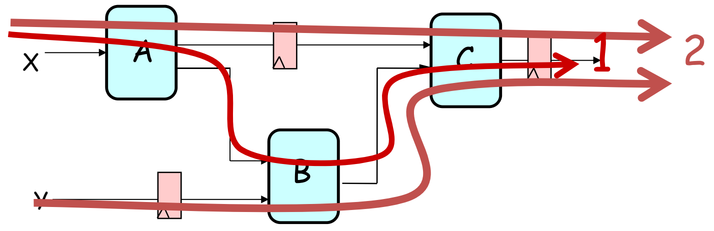
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

The LATENCY of a K-pipeline is K times the period of the clock common to all registers.

The THROUGHPUT of a K-pipeline is the frequency of the clock.

Ill-formed pipelines

Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline? ________ none

Problem:

Successive inputs get mixed: e.g., B(A(X_{i+1}), Y_i). This happened because some paths from inputs to outputs have 2 registers, and some have only 1!

This CAN'T HAPPEN on a well-formed K pipeline!

A pipelining methodology

Step 1:

Add a register on each output.

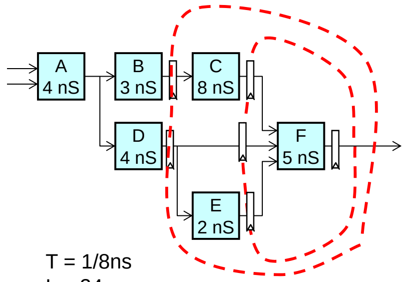
Step 2:

Add another register on each output. Draw a cut-set contour that includes all the new registers and some part of the circuit. Retime by moving regs from all outputs to all inputs of cut-set.

Repeat until satisfied with T.

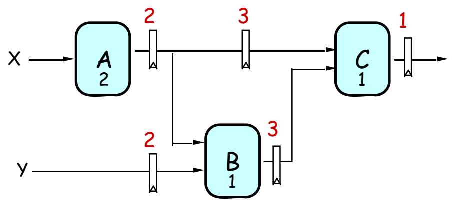
STRATEGY:

Focus your attention on placing pipelining registers around the slowest circuit elements (BOTTLENECKS).



T = 1/8ns
L = 24ns

Pipeline Example



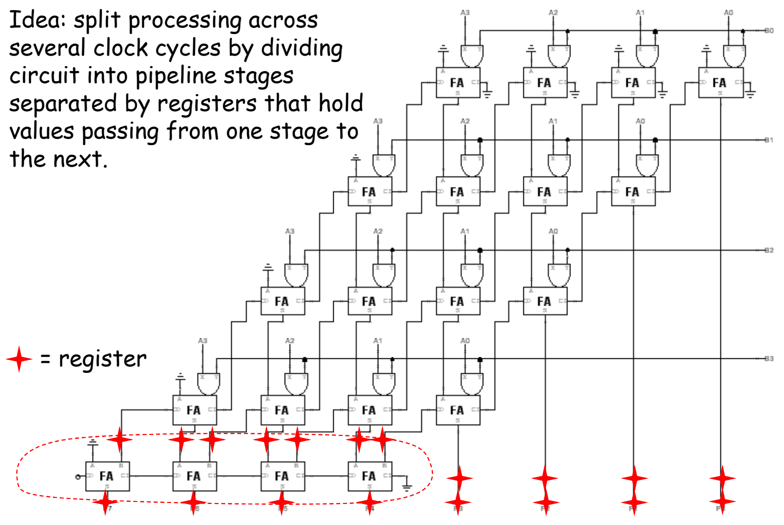
	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

OBSERVATIONS:

- 1-pipeline improves neither L or T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are often required to keep pipeline well-formed.

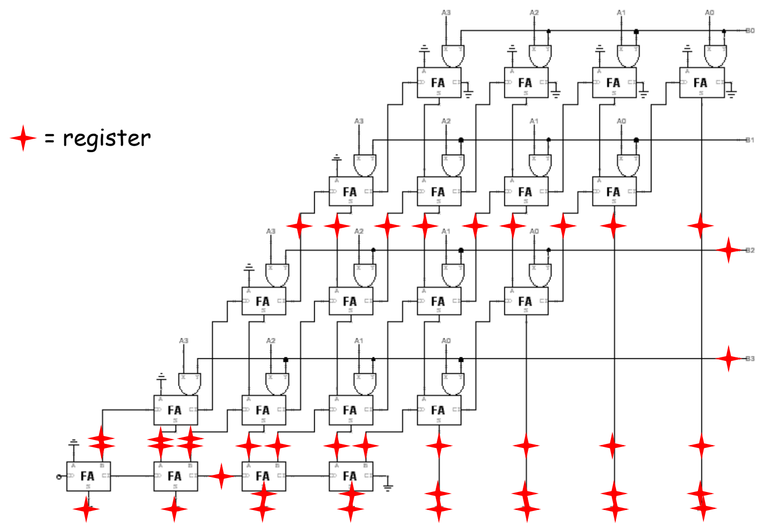
Increasing Throughput: Pipelining

Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.

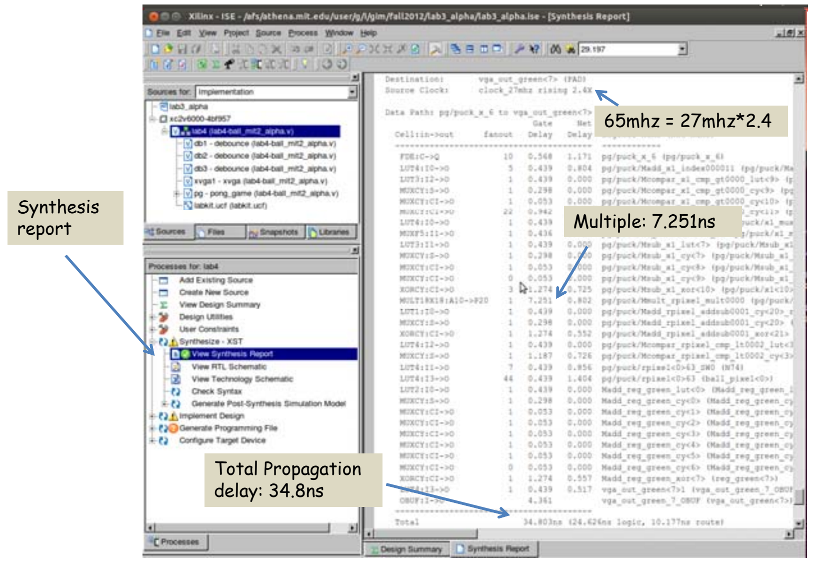


Throughput = $1/4t_{PD,FA}$ instead of $1/8t_{PD,FA}$)

How about $t_{PD} = 1/2t_{PD,FA}$?



Timing Reports



Xilinx - ISE - /afs/athena.mit.edu/user/g/i/gim/fall2012/lab3_alpha/lab3_alpha.ise - [Synthesis Report]
Synthesis report
65mhz = 27mhz*2.4
Multiple: 7.251ns
Total Propagation delay: 34.8ns
Design Summary
Synthesis Report

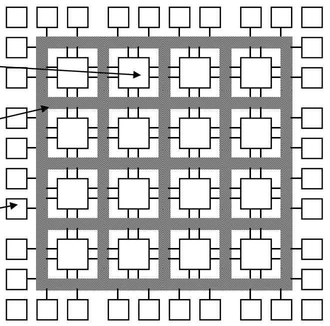
```
Destination:       vga_out_green<7> (PAD)
Source Clock:      clock_27mhz rising 2.4X

Data Path: pg/puck_x_6 to vga_out_green<7>
                                Gate     Net
  Cell:in->out       fanout   Delay   Delay
  ----------------------------------------  ------------
   FDE:C->Q              10   0.568   1.171  pg/puck_x_6 (pg/puck_x_6)
   LUT4:I0->O             5   0.439   0.804  pg/puck/Madd_x1_index000011 (pg/puck/Ma
   LUT3:I2->O             1   0.439   0.000  pg/puck/Mcompar_x1_cmp_gt0000_lut<9> (p
   MUXCY:S->O             1   0.298   0.000  pg/puck/Mcompar_x1_cmp_gt0000_cy<9> (pg
   MUXCY:CI->O            1   0.053   0.000  pg/puck/Mcompar_x1_cmp_gt0000_cy<10> (p
   MUXCY:CI->O           22   0.942                                      _cy<11> (p
   LUT4:I0->O             1   0.439                                   puck/x1_mux
   MUXF5:I1->O            1   0.436                                   g/puck/x1_m
   LUT3:I1->O             1   0.439   0.000  pg/puck/Msub_x1_lut<7> (pg/puck/Msub_x1
   MUXCY:S->O             1   0.298   0.000  pg/puck/Msub_x1_cy<7> (pg/puck/Msub_x1_
   MUXCY:CI->O            1   0.053   0.000  pg/puck/Msub_x1_cy<8> (pg/puck/Msub_x1_
   MUXCY:CI->O            0   0.053   0.000  pg/puck/Msub_x1_cy<9> (pg/puck/Msub_x1_
   XORCY:CI->O            3   1.274   0.725  pg/puck/Msub_x1_xor<10> (pg/puck/x1<10>
   MULT18X18:A10->P20     1   7.251   0.802  pg/puck/Mmult_rpixel_mult0000 (pg/puck/
   LUT1:I0->O             1   0.439   0.000  pg/puck/Madd_rpixel_addsub0001_cy<20>_r
   MUXCY:S->O             1   0.298   0.000  pg/puck/Madd_rpixel_addsub0001_cy<20> (
   XORCY:CI->O            1   1.274   0.552  pg/puck/Madd_rpixel_addsub0001_xor<21>
   LUT4:I2->O             1   0.439   0.000  pg/puck/Mcompar_rpixel_cmp_lt0002_lut<1
   MUXCY:S->O             1   1.187   0.726  pg/puck/Mcompar_rpixel_cmp_lt0002_cy<3>
   LUT4:I1->O             7   0.439   0.856  pg/puck/rpixel<0>63_SW0 (N74)
   LUT4:I3->O            44   0.439   1.404  pg/puck/rpixel<0>63 (ball_pixel<0>)
   LUT2:I0->O             1   0.439   0.000  Madd_reg_green_lut<0> (Madd_reg_green_l
   MUXCY:S->O             1   0.298   0.000  Madd_reg_green_cy<0> (Madd_reg_green_cy
   MUXCY:CI->O            1   0.053   0.000  Madd_reg_green_cy<1> (Madd_reg_green_cy
   MUXCY:CI->O            1   0.053   0.000  Madd_reg_green_cy<2> (Madd_reg_green_cy
   MUXCY:CI->O            1   0.053   0.000  Madd_reg_green_cy<3> (Madd_reg_green_cy
   MUXCY:CI->O            1   0.053   0.000  Madd_reg_green_cy<4> (Madd_reg_green_cy
   MUXCY:CI->O            1   0.053   0.000  Madd_reg_green_cy<5> (Madd_reg_green_cy
   MUXCY:CI->O            0   0.053   0.000  Madd_reg_green_cy<6> (Madd_reg_green_cy
   XORCY:CI->O            1   1.274   0.557  Madd_reg_green_xor<7> (reg_green<7>)
   LUT4:I3->O             1   0.439   0.517  vga_out_green<7>1 (vga_out_green_7_OBUF
   OBUF:I->O                  4.361          vga_out_green_7_OBUF (vga_out_green<7>)
  ----------------------------------------
  Total                      34.803ns (24.626ns logic, 10.177ns route)
```

History of Computational Fabrics

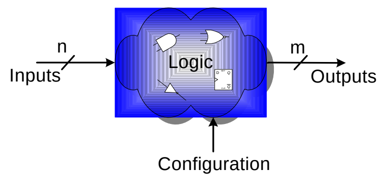
- Discrete devices: relays, transistors (1940s-50s)
- Discrete logic gates (1950s-60s)
- Integrated circuits (1960s-70s)
  - e.g. TTL packages: Data Book for 100's of different parts
- Gate Arrays (IBM 1970s)
  - Transistors are pre-placed on the chip & Place and Route software puts the chip together automatically – only program the interconnect (mask programming)
- Software Based Schemes (1970's- present)
  - Run instructions on a general purpose core
- Programmable Logic (1980's to present)
  - A chip that be reprogrammed after it has been fabricated
  - Examples: PALs, EPROM, EEPROM, PLDs, FPGAs
  - Excellent support for mapping from Verilog
- ASIC Design (1980's to present)
  - Turn Verilog directly into layout using a library of standard cells
  - Effective for high-volume and efficient use of silicon area

Reconfigurable Logic

- Logic blocks
  - To implement combinational and sequential logic
- Interconnect
  - Wires to connect inputs and outputs to logic blocks
- I/O blocks
  - Special logic blocks at periphery of device for external connections

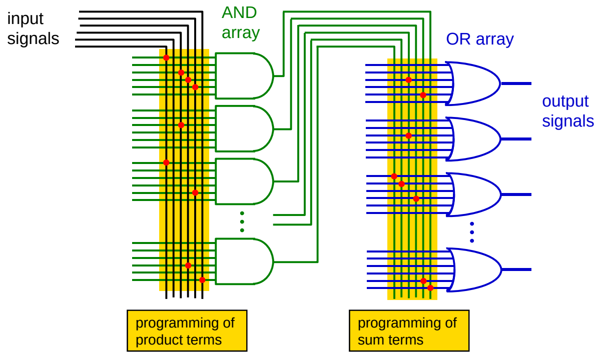


- Key questions:
  - How to make logic blocks programmable? (after chip has been fabbed!)
  - What should the logic granularity be?
  - How to make the wires programmable? (after chip has been fabbed!)
  - Specialized wiring structures for local vs. long distance routes?
  - How many wires per logic block?

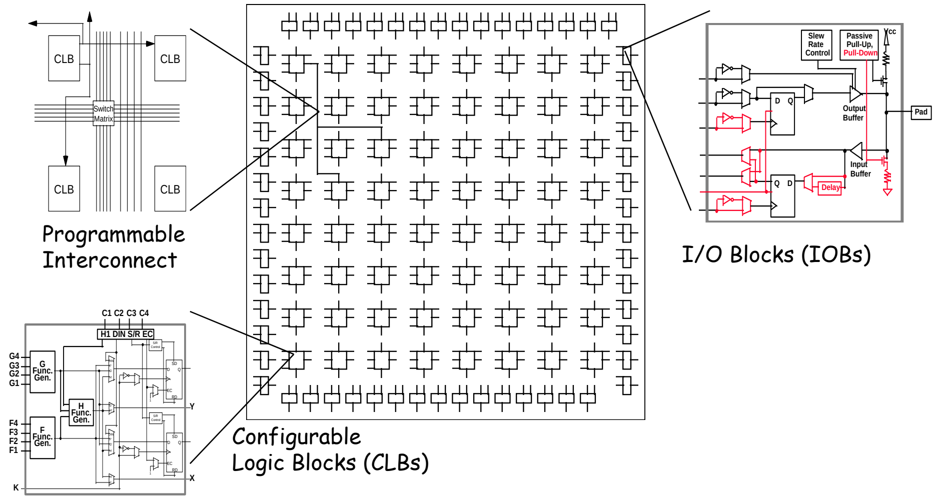


Programmable Array Logic (PAL)

- Based on the fact that any combinational logic can be realized as a sum-of-products
- PALs feature an array of AND-OR gates with programmable interconnect

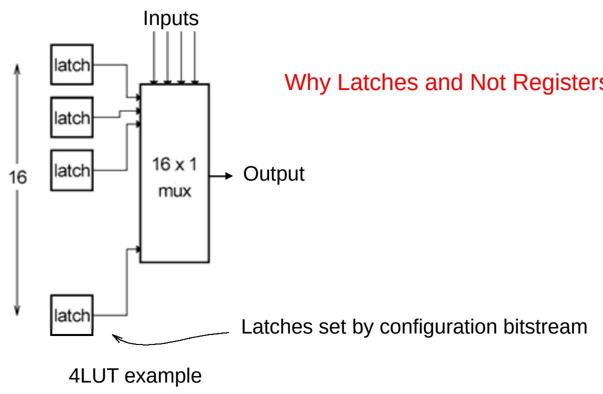


RAM Based Field Programmable Logic - Xilinx



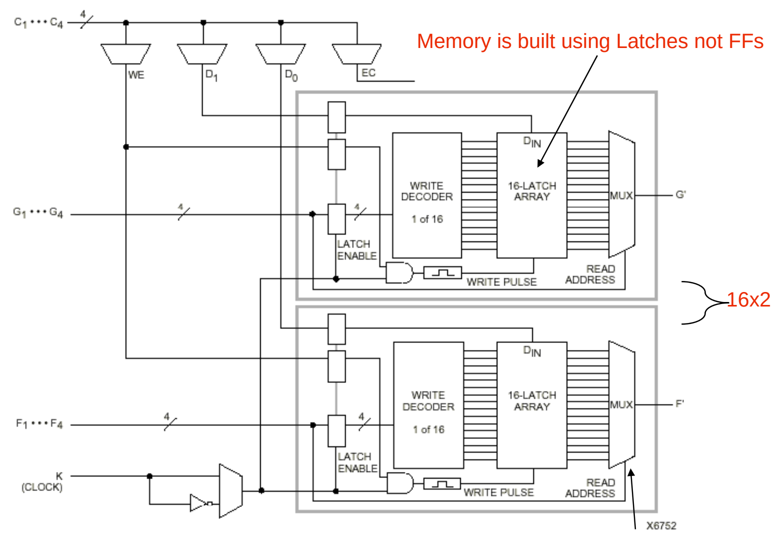
LUT Mapping

- N-LUT direct implementation of a truth table: any function of n-inputs.
- N-LUT requires 2^N storage elements (latches)
- N-inputs select one latch location (like a memory)



Why Latches and Not Registers?

Configuring the CLB as a RAM

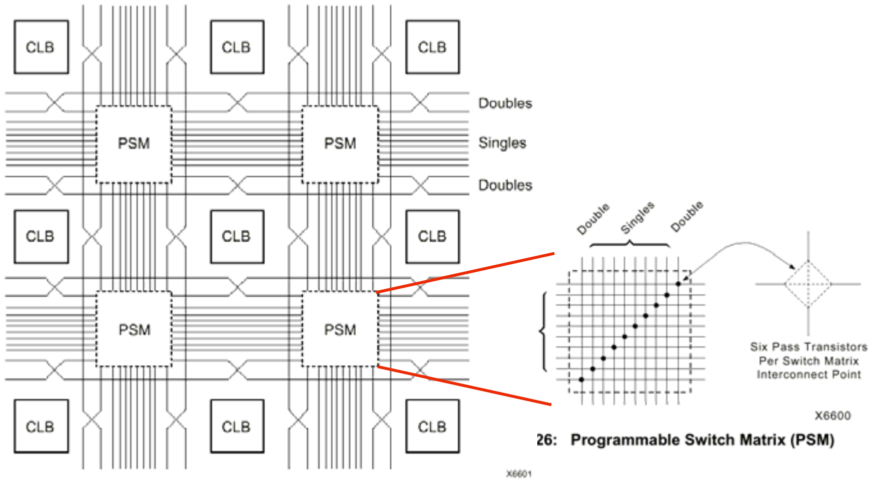


Memory is built using Latches not FFs

gure 4: 16x2 (or 16x1) Edge-Triggered Single-Port RAM

Read is same a LUT Function!

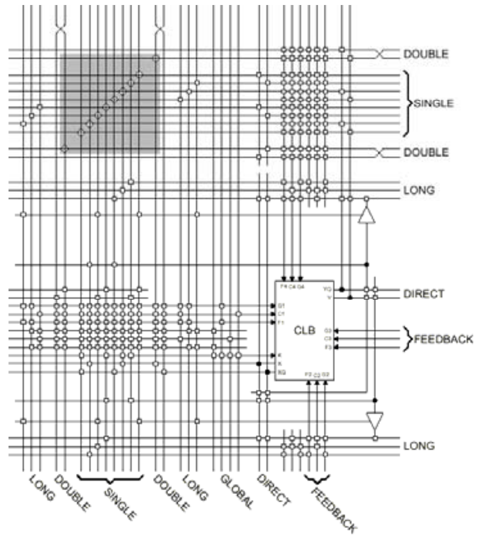
Xilinx 4000 Interconnect

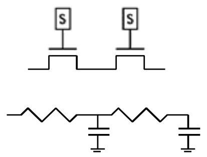


CLB
PSM
Doubles
Singles
Doubles
Double
Singles
Double
Six Pass Transistors Per Switch Matrix Interconnect Point
X6600
26: Programmable Switch Matrix (PSM)
X6601

Figure 28: Single- and Double-Length Lines, with Programmable Switch Matrices (PSMs)

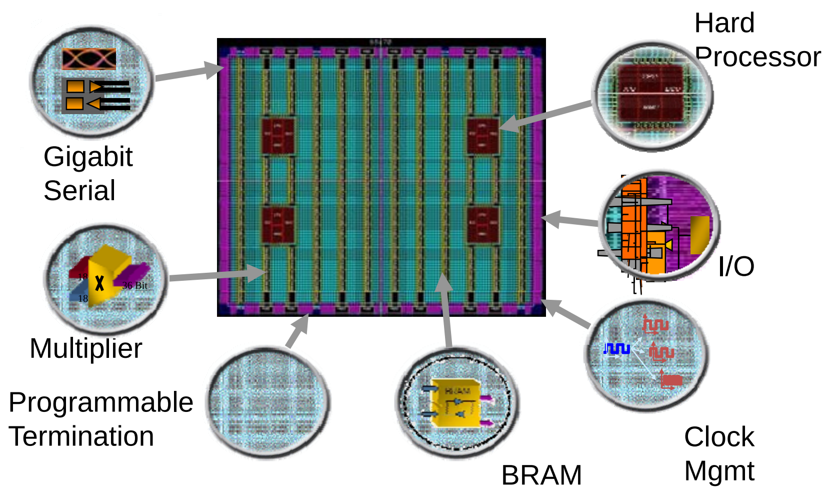
Xilinx 4000 Interconnect Details





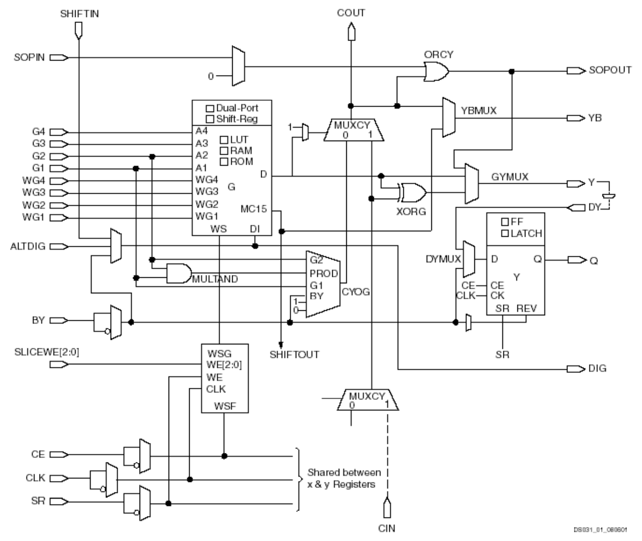
Wires are not ideal!

Add Bells & Whistles

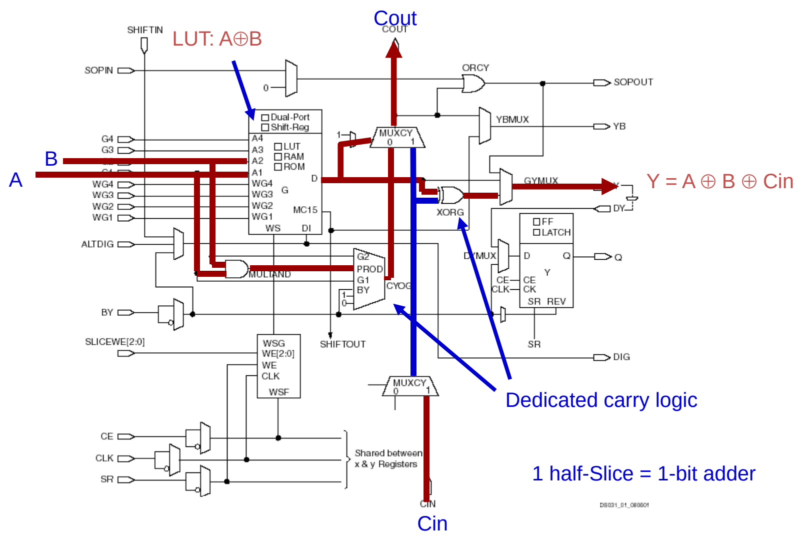


Courtesy of David B. Parlour, ISSCC 2004 Tutorial,
“The Reality and Promise of Reconfigurable Computing in Digital Signal Processing”

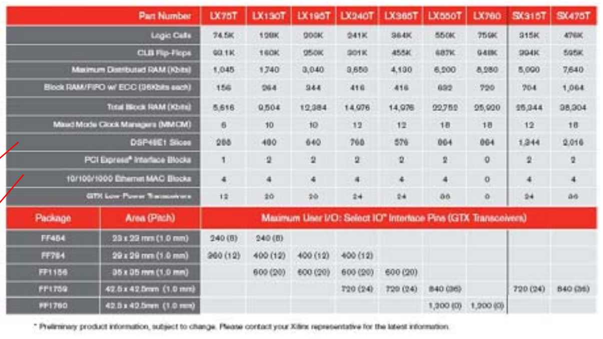
The Virtex II CLB (Half Slice Shown)



Adder Implementation



Virtex-6



Part Number	LX75T	LX130T	LX195T	LX240T	LX365T	LX550T	LX760	SX315T	SX475T
Logic Cells	74.5K	128K	199K	241K	364K	550K	759K	315K	476K
CLB Flip-Flops	93.1K	160K	250K	301K	455K	687K	948K	394K	595K
Maximum Distributed RAM (Kbits)	1,045	1,740	3,040	3,650	4,130	6,200	8,280	5,090	7,640
Block RAM/FIFO w/ ECC (36Kbits each)	156	264	344	416	416	632	720	704	1,064
Total Block RAM (Kbits)	5,616	9,504	12,384	14,976	14,976	22,752	25,920	25,344	38,304
Mixed Mode Clock Managers (MMCM)	6	10	10	12	12	18	18	12	18
DSP48E1 Slices	288	480	640	768	576	864	864	1,344	2,016
PCI Express® Interface Blocks	1	2	2	2	2	2	0	2	2
10/100/1000 Ethernet MAC Blocks	4	4	4	4	4	4	0	4	4
GTX Low-Power Transceivers	12	20	20	24	24	36	0	24	36

Package	Area (Pitch)	Maximum User I/O: SelectIO™ Interface Pins (GTX Transceivers)								
FF484	23 x 23 mm (1.0 mm)	240 (8)	240 (8)							
FF784	29 x 29 mm (1.0 mm)	360 (12)	400 (12)	400 (12)	400 (12)					
FF1156	35 x 35 mm (1.0 mm)		600 (20)	600 (20)	600 (20)	600 (20)				
FF1759	42.5 x 42.5mm (1.0 mm)				720 (24)	720 (24)	840 (36)		720 (24)	840 (36)
FF1760	42.5 x 42.5mm (1.0 mm)						1,200 (0)	1,200 (0)		

* Preliminary product information, subject to change. Please contact your Xilinx representative for the latest information.

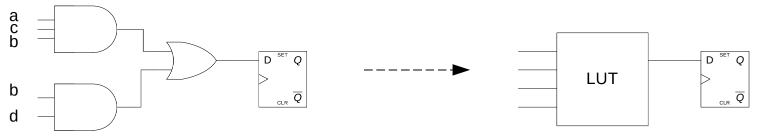
DSP with 25x18 multiplier

Gigabit ethernet support

	CLB	Dist RAM	Block RAM	Multipliers
Virtex 2	8,448	1,056kbit	2,592kbit	144 (18x18)
Virtex 6	667,000	6,200kbit	22,752kbit	1,344 (25x18)
Spartan 3E	240	15kbit	72kbit	4 (18x18)

Design Flow - Mapping

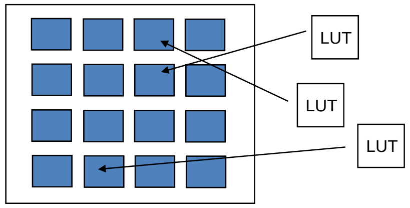
- Technology Mapping: Schematic/HDL to Physical Logic units
- Compile functions into basic LUT-based groups (function of target architecture)



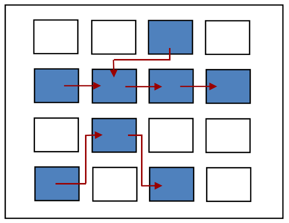
```
always @(posedge clock or negedge reset)
  begin
    if (! reset)
       q <= 0;
    else
       q <= (a & b & c) | (b & d);
  end
```

Design Flow – Placement & Route

- Placement – assign logic location on a particular device



- Routing – iterative process to connect CLB inputs/outputs and IOBs. Optimizes critical path delay – can take hours or days for large, dense designs



Iterate placement if timing not met

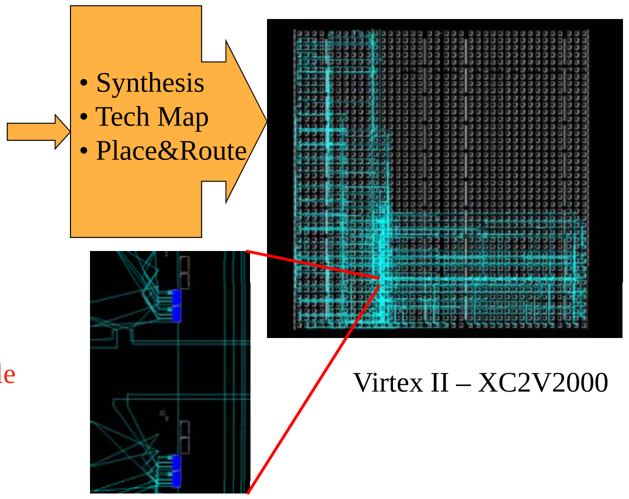
Satisfy timing? → Generate Bitstream to config device

Challenge! Cannot use full chip for reasonable speeds (wires are not ideal).

Typically no more than 50% utilization.

Example: Verilog to FPGA

```
module adder64 (
  input  [63:0] a, b;
  output [63:0] sum);

  assign sum = a + b;
endmodule
```

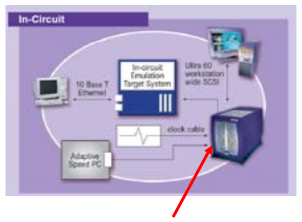


64-bit Adder Example

Virtex II – XC2V2000

How are FPGAs Used?

Logic Emulation


In-Circuit
In-circuit Emulation Target System
10 Base T Ethernet
Ultra 60 workstation wide SCSI
clock cable
Adaptive Speed PC

FPGA-based Emulator

(courtesy of IKOS)

- Prototyping
  - Ensemble of gate arrays used to emulate a circuit to be manufactured
  - Get more/better/faster debugging done than with simulation
- Reconfigurable hardware
  - One hardware block used to implement more than one function
- Special-purpose computation engines
  - Hardware dedicated to solving one problem (or class of problems)
  - Accelerators attached to general-purpose computers (e.g., in a cell phone!)

Summary

- FPGA provide a flexible platform for implementing digital computing
- A rich set of macros and I/Os supported (multipliers, block RAMS, ROMS, high-speed I/O)
- A wide range of applications from prototyping (to validate a design before ASIC mapping) to high-performance spatial computing
- Interconnects are a major bottleneck (physical design and locality are important considerations)

Lab 4 Car Alarm Design Approach

- Read lab/specifications carefully, use reasonable interpretation
- Use modular design – don't put everything into labkit.v
- Design the FSM!
  - Define the inputs
  - Define the outputs
  - Transition rules
- Logical modules:
  - fsm.v
  - timer.v
  - siren.v
  - fuel_pump.v
- Run simulation on each module!
- Use hex display: show state and time
- Use logic analyzer

Car Alarm – Inputs & Outputs

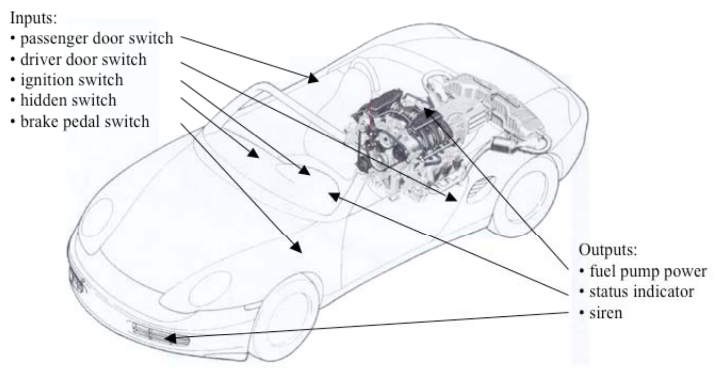
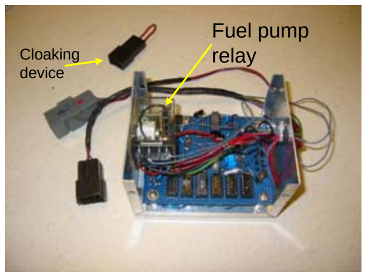


Figure 1: System diagram showing sensors (inputs) and actuators (outputs)

Car Alarm – CMOS Implementation



- Design Specs
  - Operating voltage 8-18VDC
  - Operating temp: -40C +65C
  - Attitude: sea level
  - Shock/Vibration
- Notes
  - Protected against 24V power surges
  - CMOS implementation
  - CMOS inputs protected against 200V noise spikes
  - On state DC current <10ma
  - Include T_PASSENGER_DELAY and Fuel Pump Disable
  - First car was stolen in Cambridge
  - System disabled (cloaked) when being serviced.

Debugging Hints – Lab 4

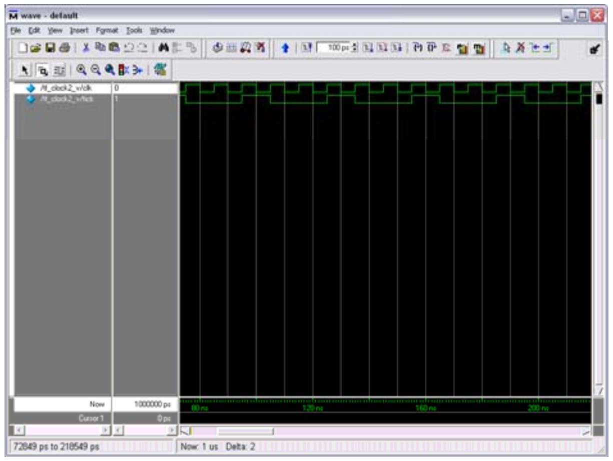
- Implement a warp speed debug mode for the one_hz clock. This will allow for viewing signals on the logic analyzer or Modelsim without waiting for 27 million clock cycles. Avoids recomplilations.

```
assign debug_on = switch[6];  // switch[6] is not used
always @ (posedge clk) begin
     if (count == (debug_on ? 3 : 26_999_999)) count <= 0;
     else count <= count +1;
  end

assign one_hz = (count == (debug_on ? 3 : 26_999_999)) ;
```

One Hz Ticks in Modelsim

To create a one hz tick, use the following in the Verilog test fixture:

```
always #5 clk=!clk;
always  begin
    #5 tick = 1;
    #10 tick = 0;
    #15;
 end

 initial begin
    // Initialize Inputs
    clk = 0;
    tick = 0;  . . .
```

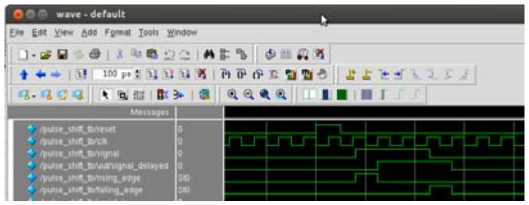


For Loops, Repeat Loops in Simulation

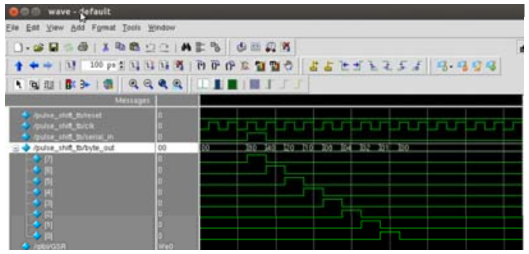
```
integer i;    // index must be declared as integer
integer irepeat;

// this will just wait 10ns, repeated 32x.
// simulation only! Can implement #10 in hardware!
        irepeat =0;
        repeat(32)  begin
        #10;
        irepeat = irepeat + 1;
        end


// this will wait #10ns before incrementing the for loop
          for (i=0; i<16; i=i+1)  begin
                #10;     // wait #10 before increment.
                         // @(posedge clk);
                         // add to index on posedge
        end
```

Edge Detection

```
reg signal_delayed;

always @(posedge clk)
       signal_delayed <= signal;

assign rising_edge = signal && !signal_delayed;
assign falling_edge = !signal && signal_delayed;
```


wave - default
/pulse_shift_tb/reset
/pulse_shift_tb/clk
/pulse_shift_tb/signal
/pulse_shift_tb/uut/signal_delayed
/pulse_shift_tb/rising_edge
/pulse_shift_tb/falling_edge

Shift Register



```
always @(posedge clk) begin
   if (reset) byte_out <=0;
   else    byte_out <= {serial_in, byte_out[7:1]};
end
```