Pipelining & Verilog

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

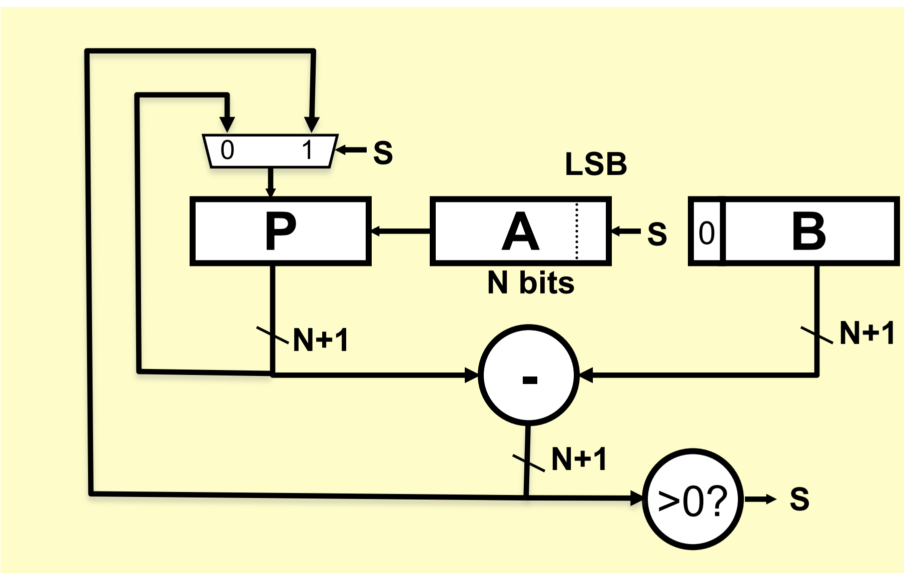
Verilog Grading

- Grading criteria
  - Proper use of blocking and non-blocking assignments
  - Readable Code with comments and consistent indenting
  - Use of default in case statement
  - Use of parameter statements for symbolic name and constants
    (state==5 vs state==DATA_READY)
  - Parameterized modules when appropriate
  - Properly formatted – see tools
  - No long nested if statements

- Labs 2 and 3 Verilog may be resubmitted for regrade. Email grader.

Sequential Divider

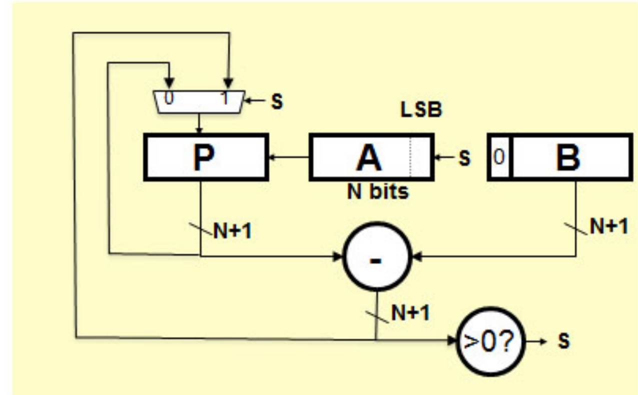
Assume the Dividend (A) and the divisor (B) have N bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single subtraction at a time and then cycle the circuit N times. This circuit works on unsigned operands; for signed operands one can remember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

Sequential Divider

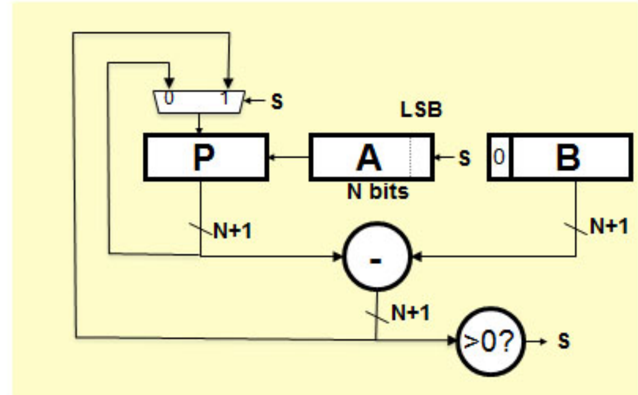
P	A	P-B	7/3 0111/11 B=0011
0000	0111		Initial value
0000	1110		Shift
0000		-3	Subtract
0000	1110		Restore, set $A_{lsb} = 0$
0001	1100		Shift
0001		-2	Subtract
0001	1100		Restore, set $A_{lsb} = 0$
0011	1000		Shift
0011		0	Subtract
0000	1001		Subtact, set $A_{lsb} = 1$
0001	0010		Shift
0001		-2	Subtract
0001	0010		Restore, set $A_{lsb} = 0$
R	Q		



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

Sequential Divider

P	A	P-B	0001/0000
0000	0001		Initial value
0000	0010		Shift
0000		0	Subtract
0000	0011		Subtact, set $A_{lsb} = 1$
0000	0110		Shift
0000		0	Subtract
0000	0111		Subtact, set $A_{lsb} = 1$
0000	1110		Shift
0000		0	Subtract
0000	1111		Subtact, set $A_{lsb} = 1$
0000	1110		Shift
0000		0	Subtract
0000	1111		Subtact, set $A_{lsb} = 1$
R	Q		



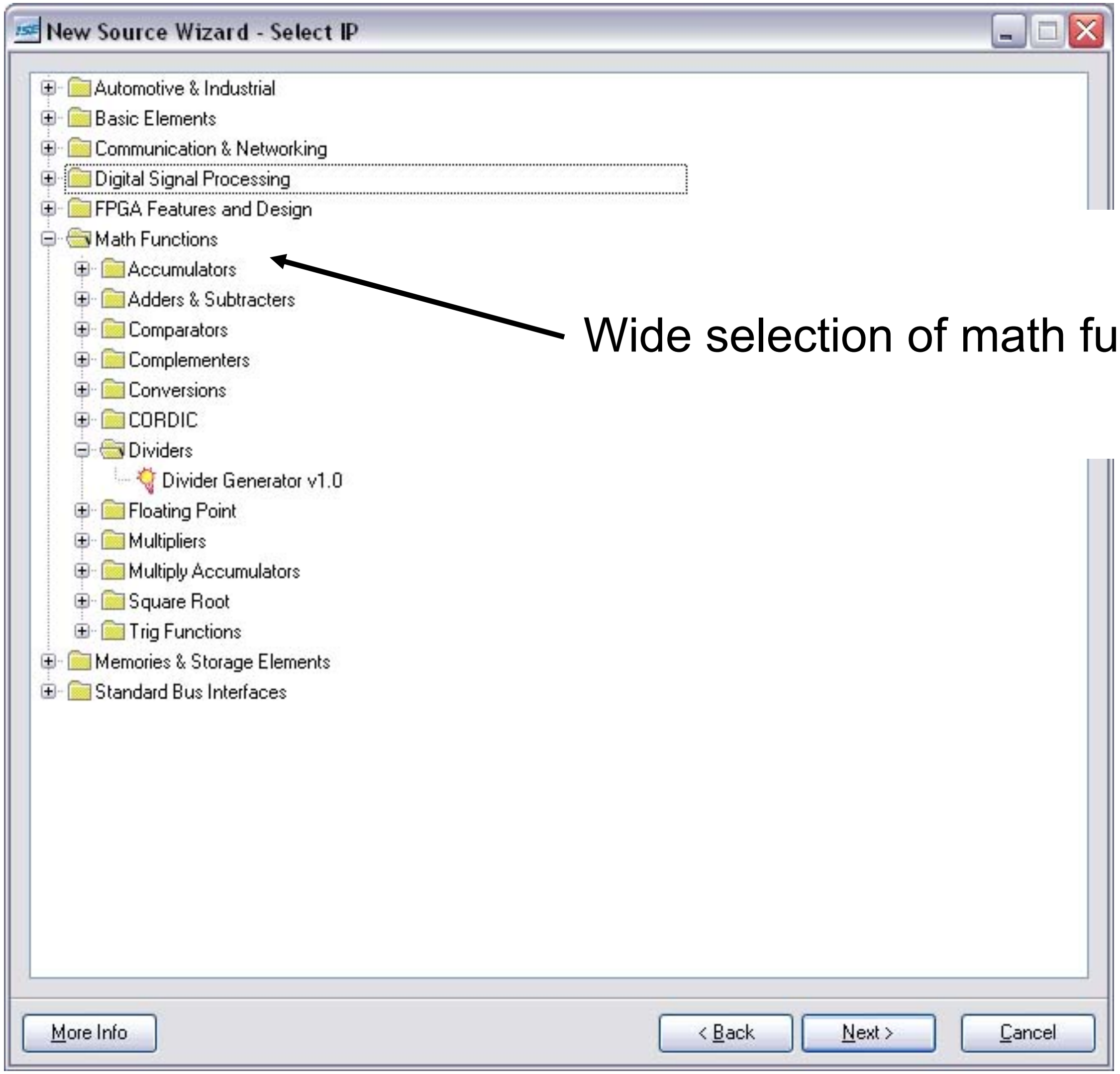

```
Init: P←0, load A and B
Repeat N times {
   shift P/A left one bit
   temp = P-B
   if (temp >= 0)
     {P←temp, A_LSB←1}
   else A_LSB←0
}
Done: Q in A, R in P
```

Verilog divider.v

```verilog
// 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)
  (input clk, sign, start,
   input [WIDTH-1:0] dividend,
   input [WIDTH-1:0] divider,
   output reg [WIDTH-1:0] quotient,
   output [WIDTH-1:0] remainder;
   output ready);

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

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

   reg [5:0] 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
```

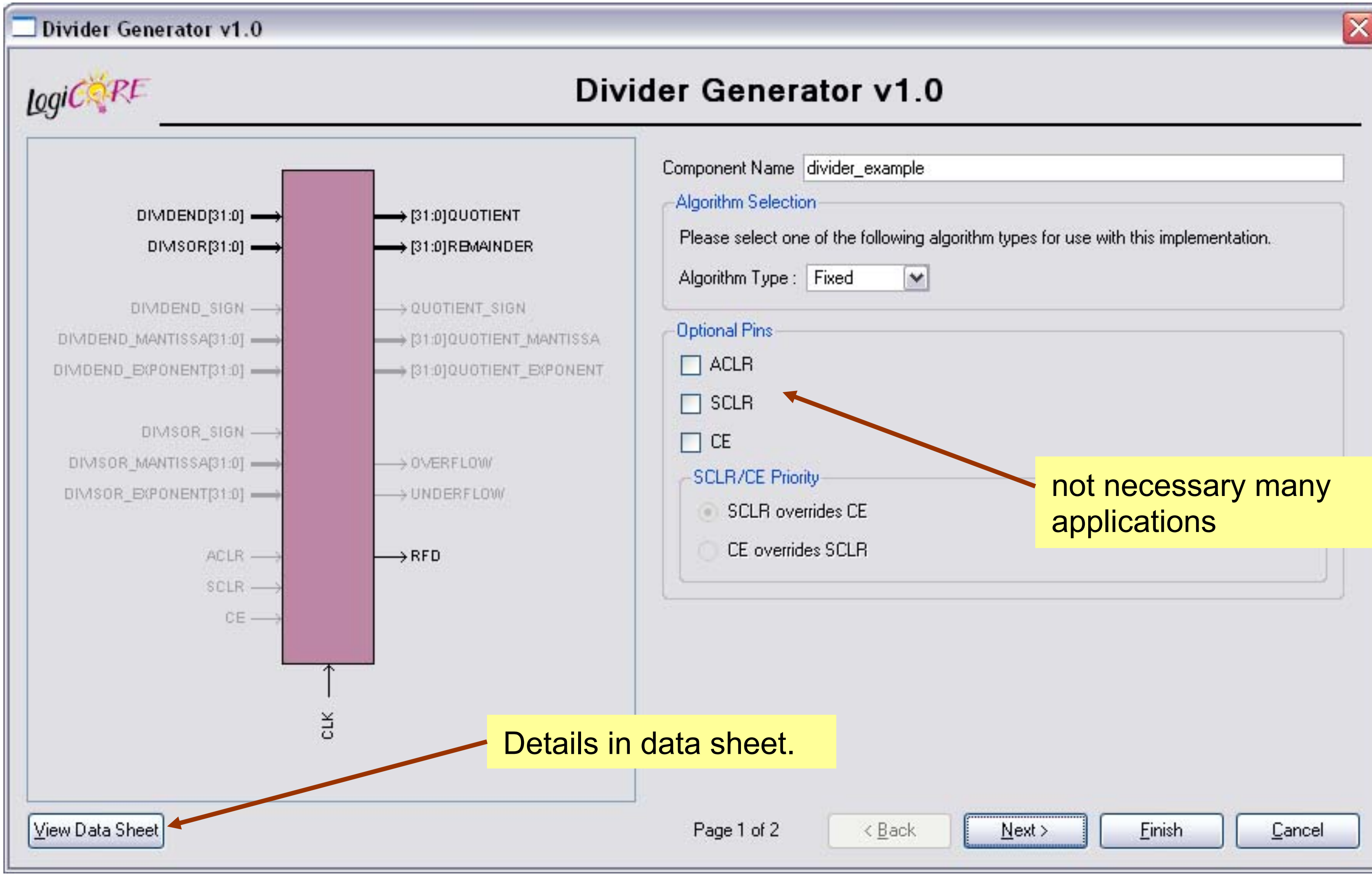
L. Williams MIT '13

Math Functions in Coregen



Wide selection of math functions available

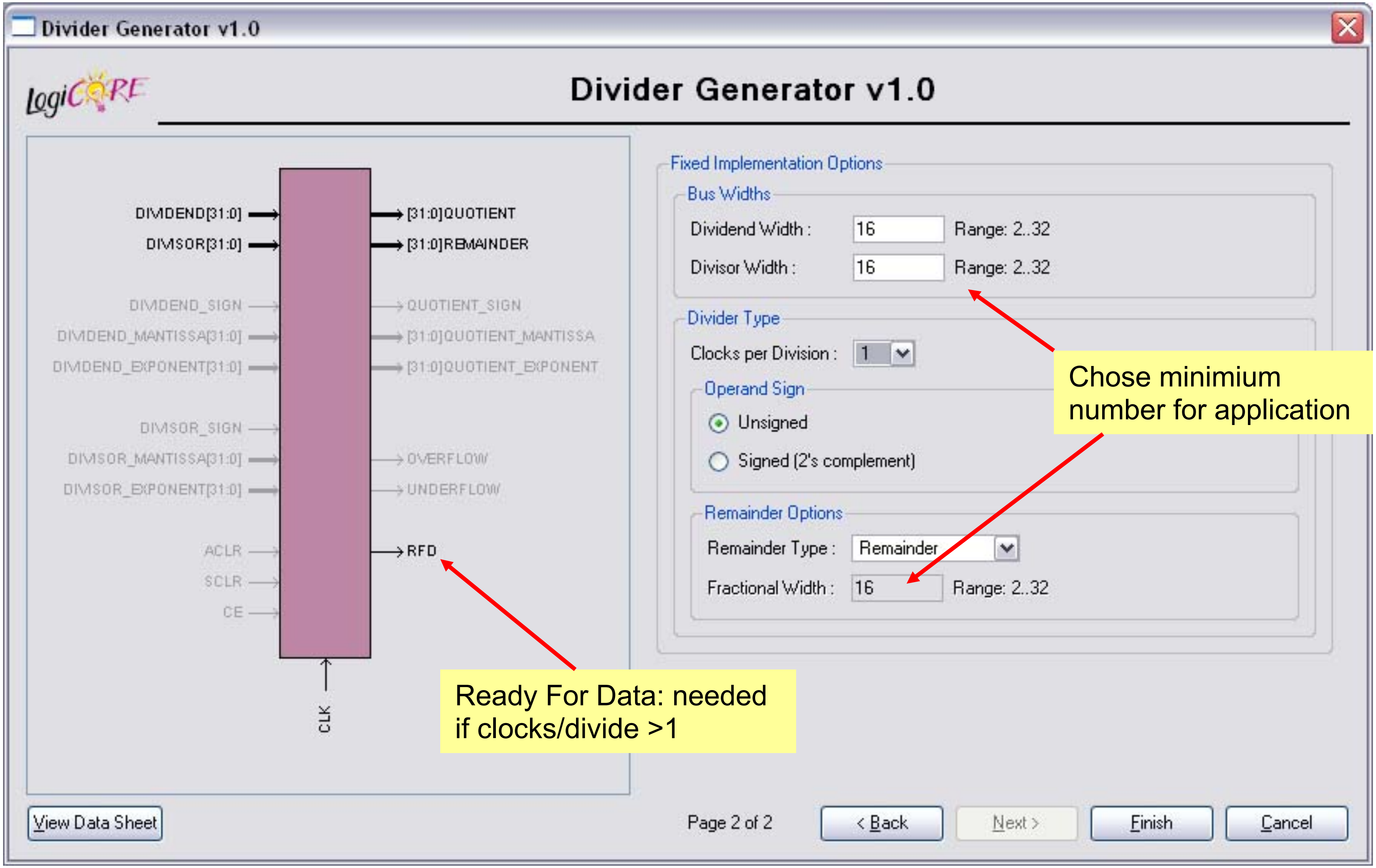
Coregen Divider



not necessary many applications

Details in data sheet.

Coregen Divider



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

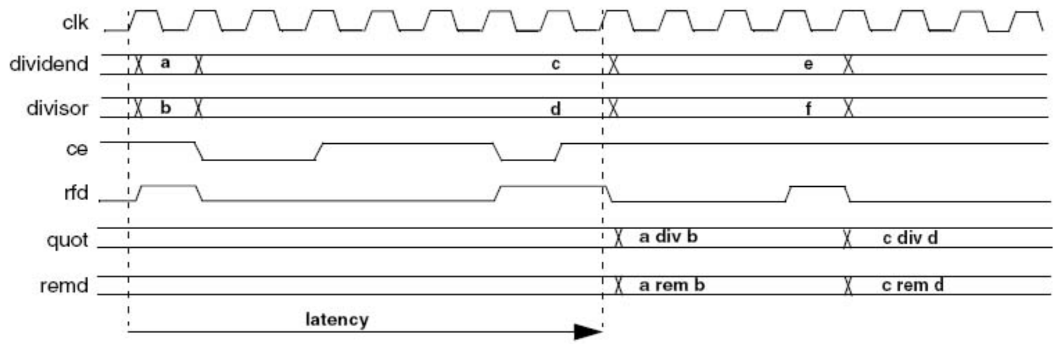


Figure 2: **Latency Example (Clocks per Division = 4)**

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

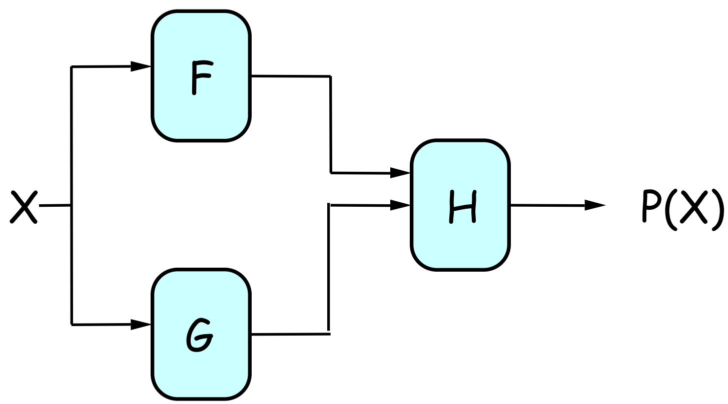
Signed	Fractional	Clks/Div	Latency
False	False	1	M+2
False	False	>1	M+3
False	True	1	M+F+2
False	True	>1	M+F+3
True	False	1	M+4
True	False	>1	M+5
True	True	1	M+F+4
True	True	>1	M+F+5

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

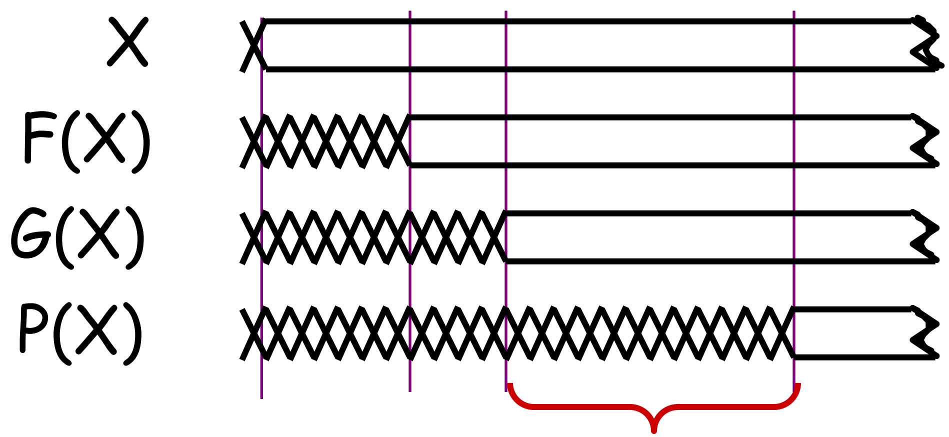


For combinational logic:

$L = t_{PD}$,

$T = 1/t_{PD}$.

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

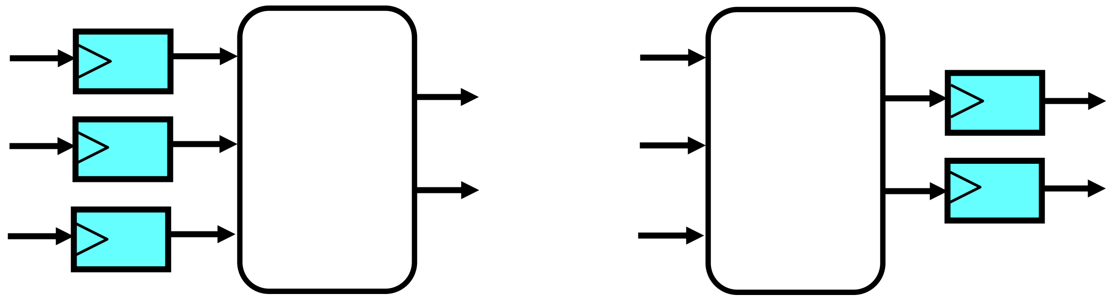


F & G are "idle", just holding their outputs stable while H performs its computation

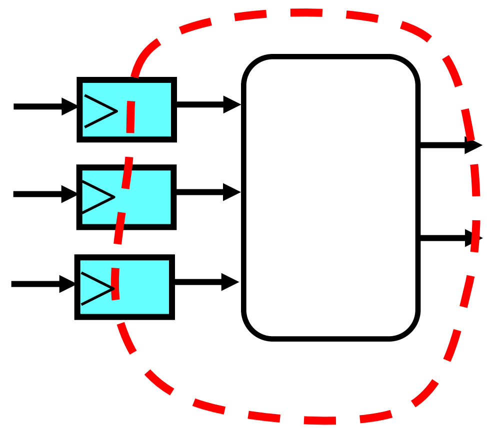
Retiming: A very useful transform

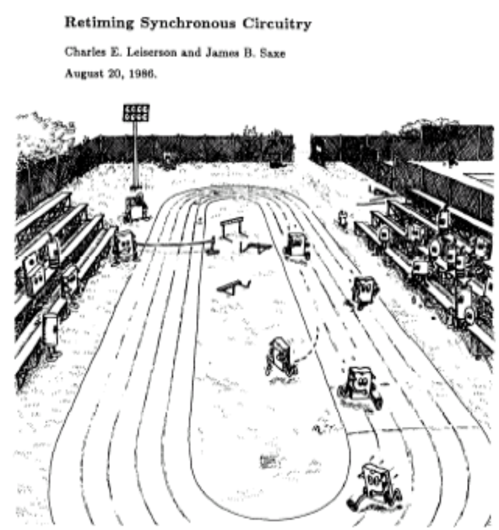
Retiming is the action of moving registers around in the system

- 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.

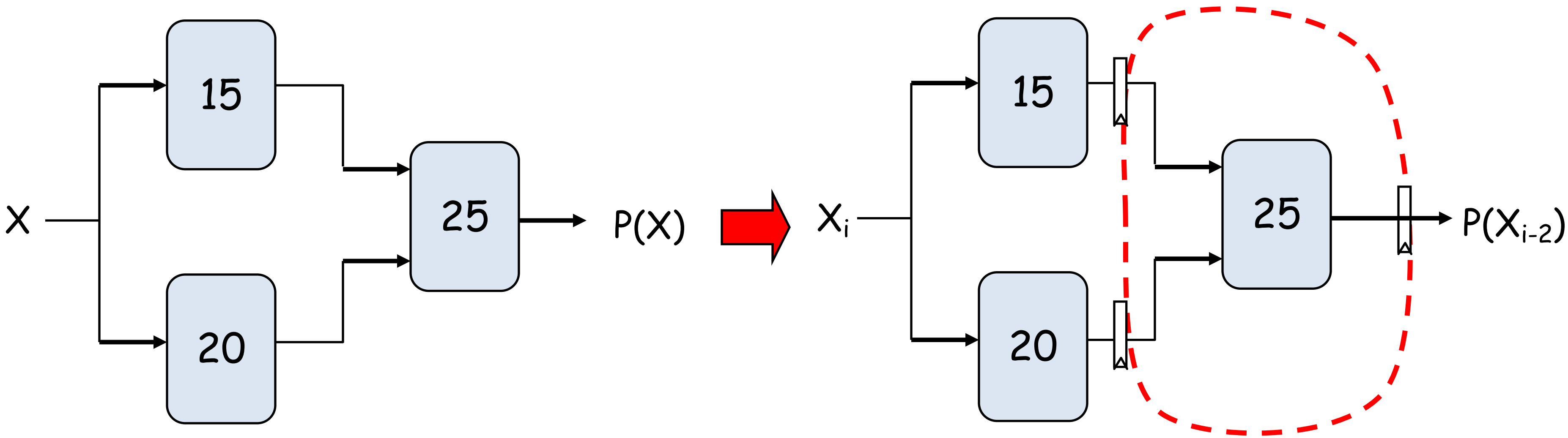




Benefits of retiming:

- Modify critical path delay
- Reduce total number of registers

Retiming Combinational Circuits aka "Pipelining"

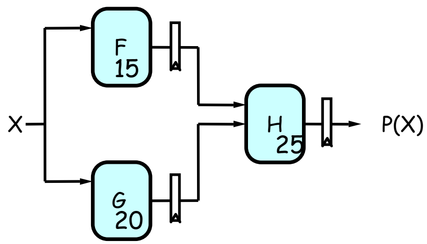


Assuming ideal registers:
i.e., $t_{PD} = 0$, $t_{SETUP} = 0$ →

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

$L = 45$
$T = 1/45$

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		$F(X_i)$	$F(X_{i+1})$	$F(X_{i+2})$	…
G Reg		$G(X_i)$	$G(X_{i+1})$	$G(X_{i+2})$	
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:

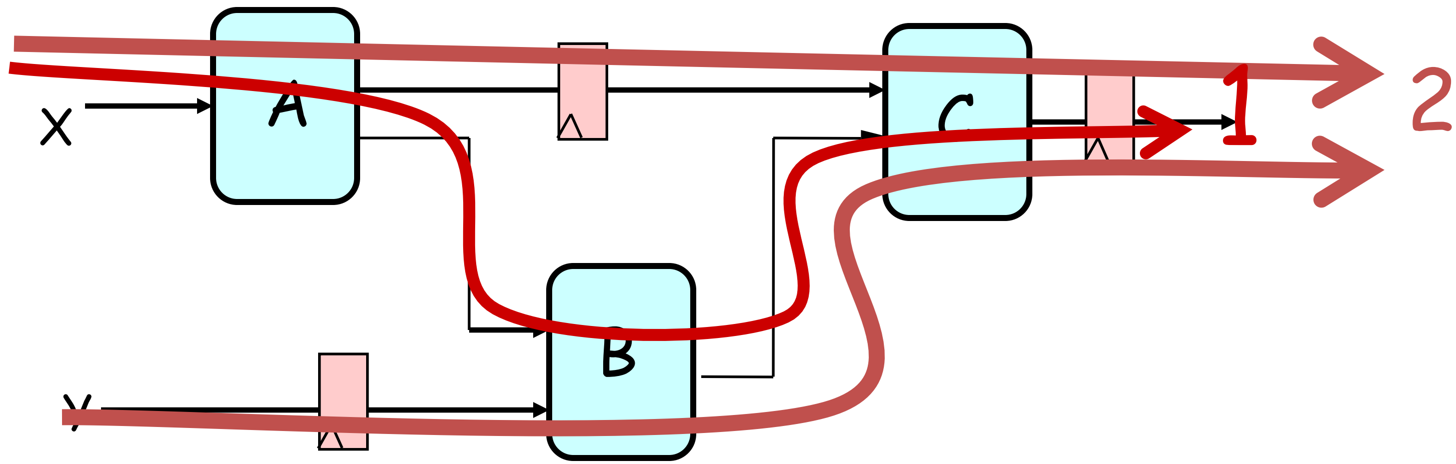
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

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.

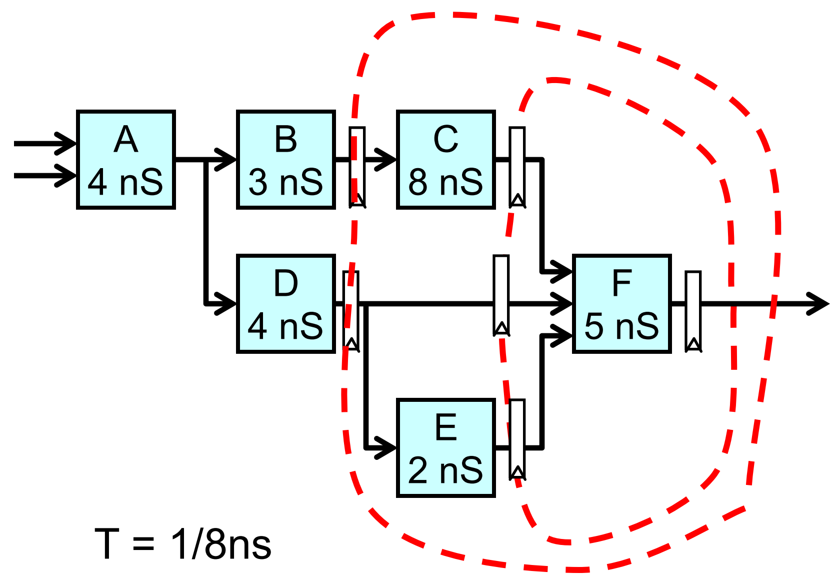
Step 2:

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.

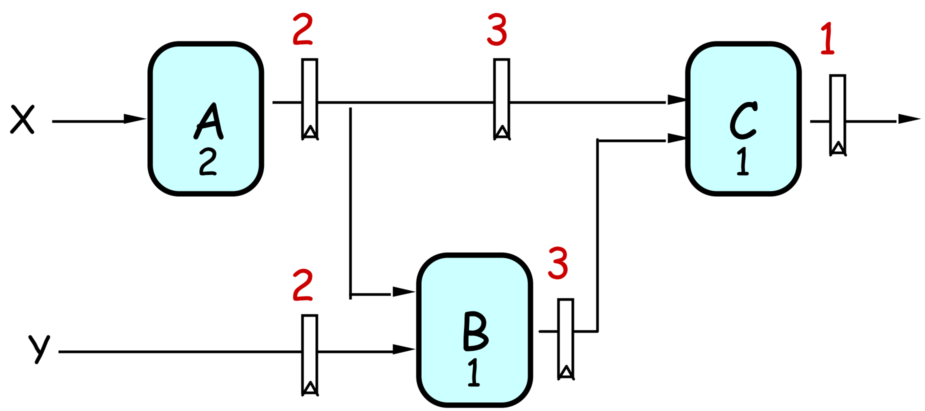
STRATEGY:

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



T = 1/8ns
L = 24ns

Pipeline Example

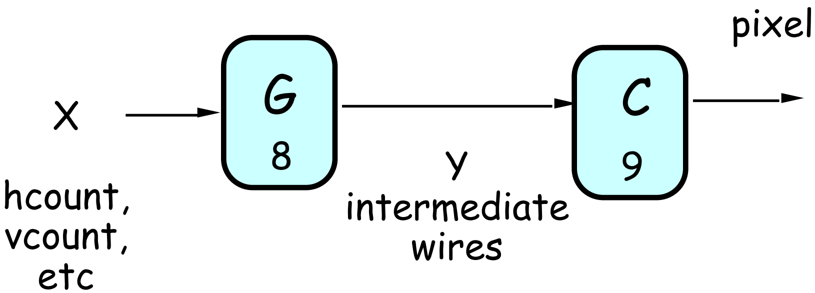


OBSERVATIONS:

- 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.

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

Pipeline Example - Verilog



Lab 3 Pong

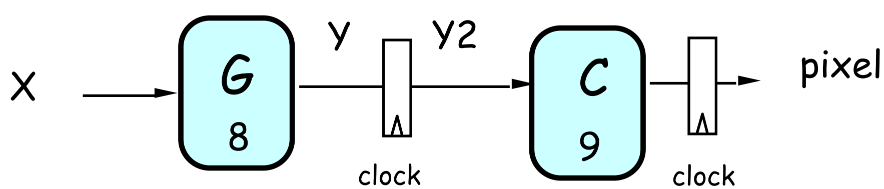
- G = game logic 8ns tpd
- C = draw round puck, use multiply with 9ns tpd
- System clock 65mhz = 15ns period – opps

No pipeline

```
assign y = G(x);       // logic for y
assign pixel = C(y)   // logic for pixel
```

⟺

```
reg [N:0] x,y;
reg [23:0] pixel
always @ *  begin
   y=G(x);
   pixel = C(y);
end
```



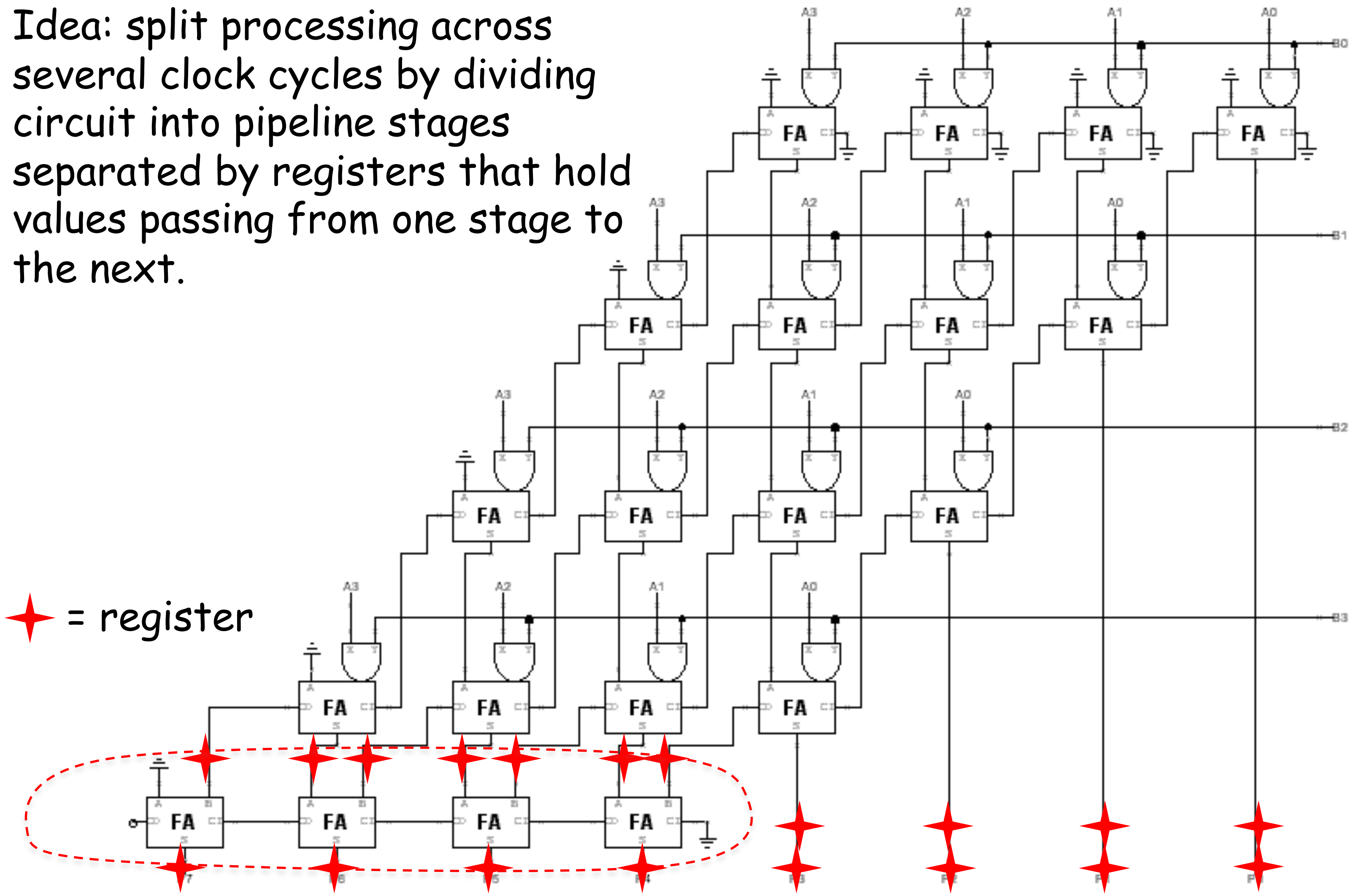
Pipeline

```
always @(posedge clock)  begin
   ...
   y2 <= G(x);       // pipeline y
   pixel <= C(y2)    // pipeline pixel
end
```

Latency = 2 clock cyles!
Implications?

Increasing Throughput: Pipelining

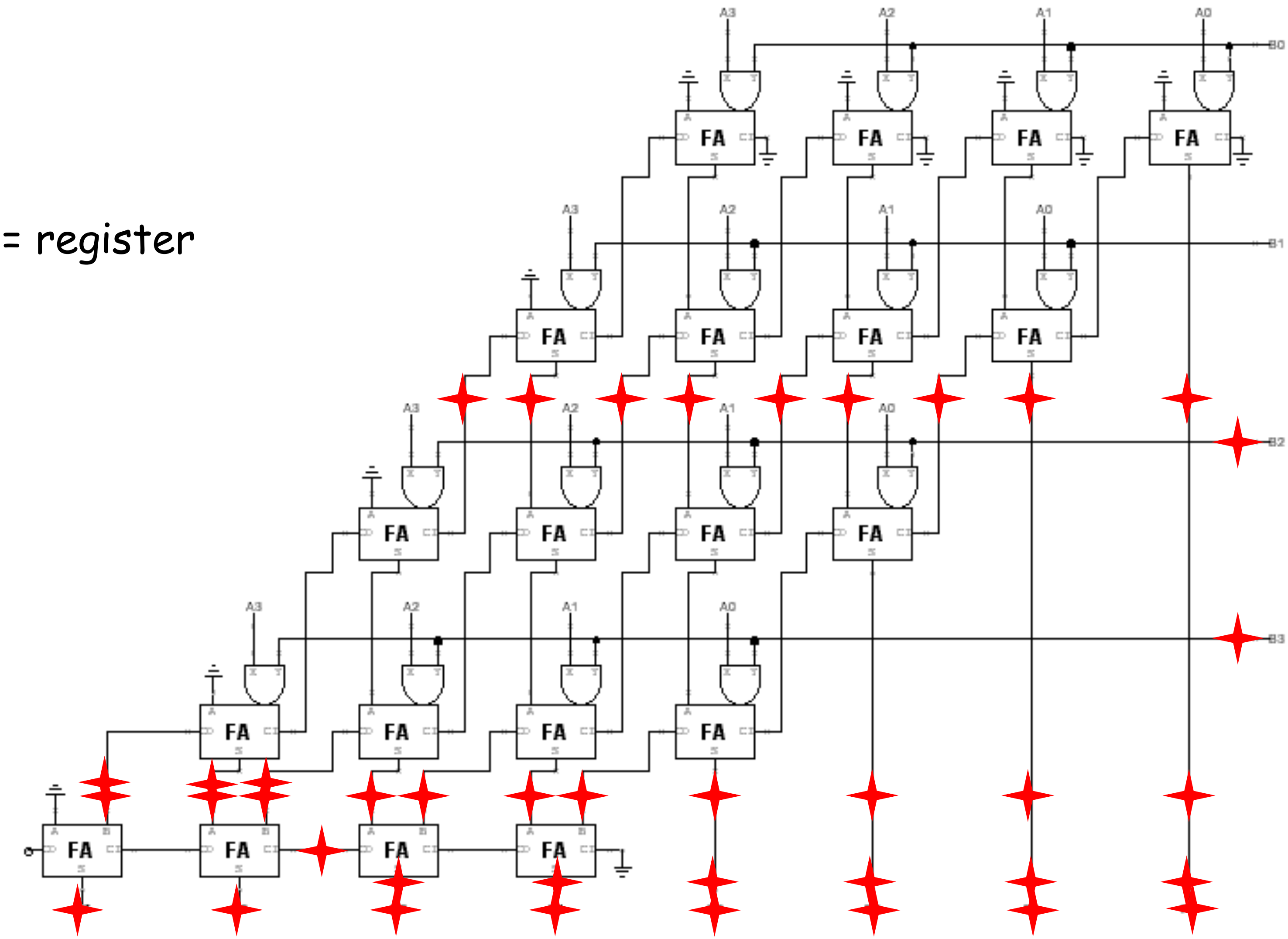
Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.



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

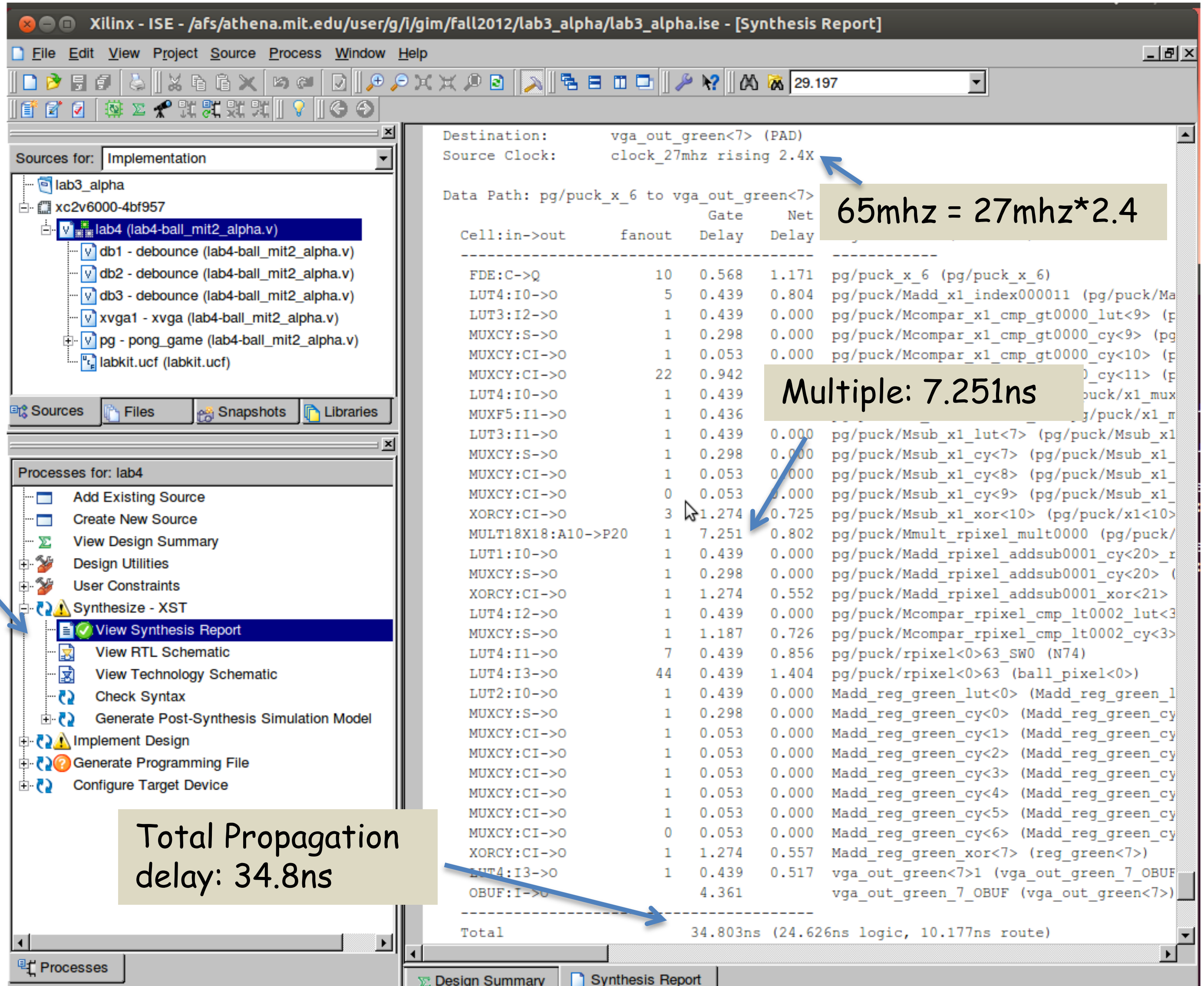
How about $t_{PD} = 1/2t_{PD,FA}$?

✦ = register



Timing Reports

Synthesis report



65mhz = 27mhz*2.4

Multiple: 7.251ns

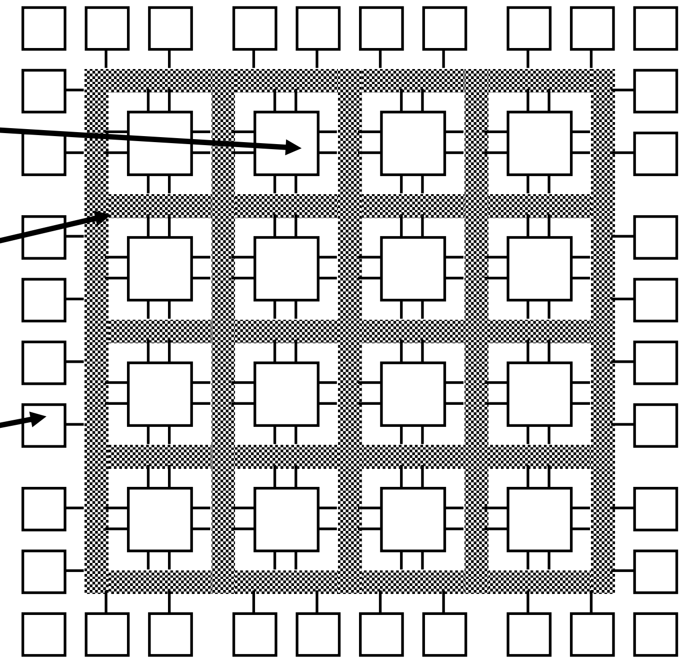
Total Propagation delay: 34.8ns

History of Computational Fabrics

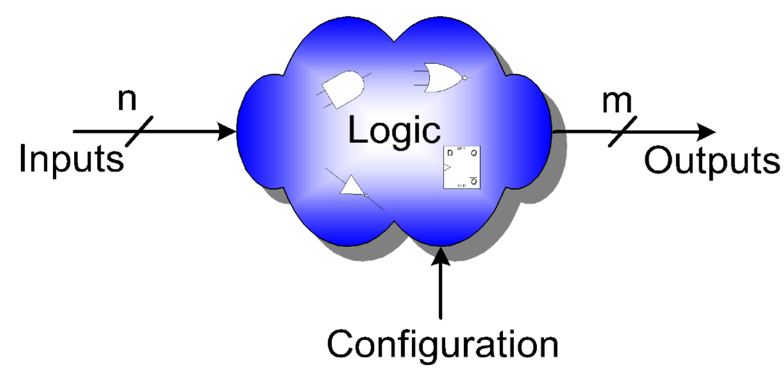
- Discrete devices: relays, transistors (1940s-50s)
- Discrete logic gates (1950s-60s)
- Integrated circuits (1960s-70s)
  - e.g. TTL packages: Data Book for 100's of different parts
- Gate Arrays (IBM 1970s)
  - Transistors are pre-placed on the chip & Place and Route software puts the chip together 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

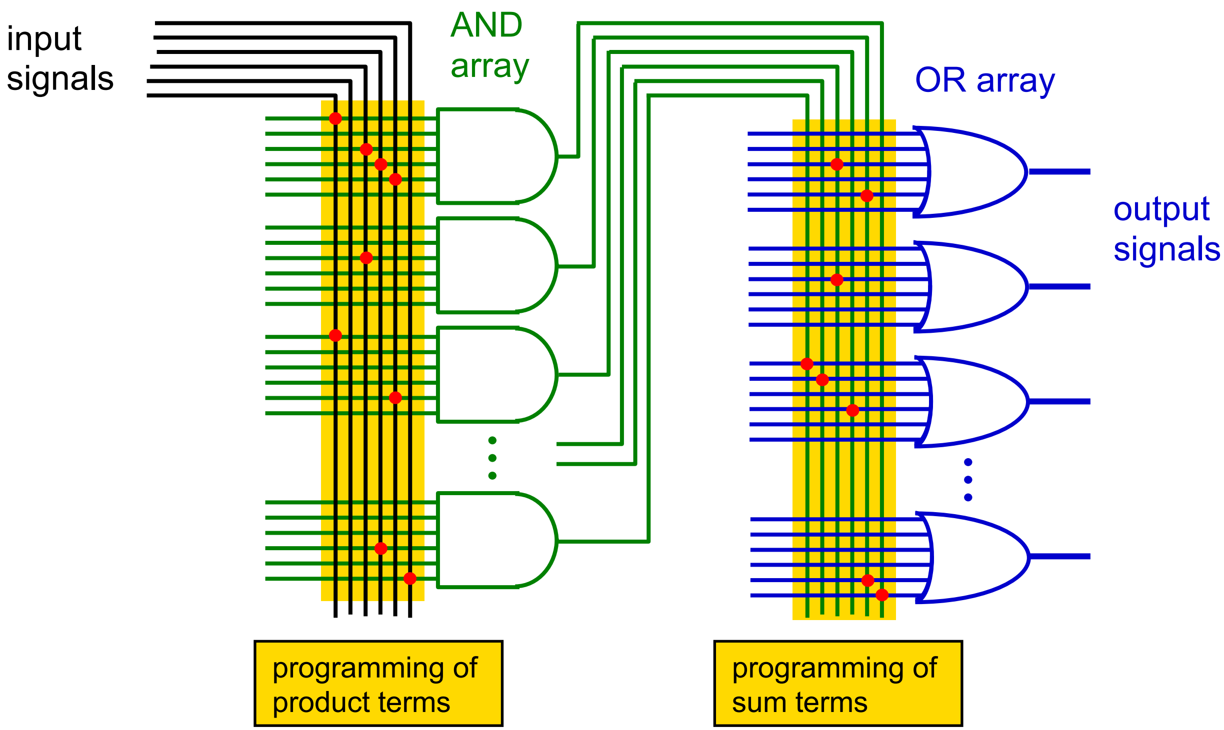


- Key questions:
  - How to make logic blocks programmable? (after chip has been fabbed!)
  - What should the logic granularity be?
  - How to make the wires programmable? (after chip has been fabbed!)
  - Specialized wiring structures for local vs. long distance routes?
  - How many wires per logic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

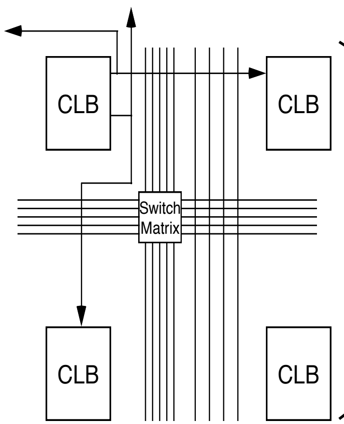


Programmable Array Logic (PAL)

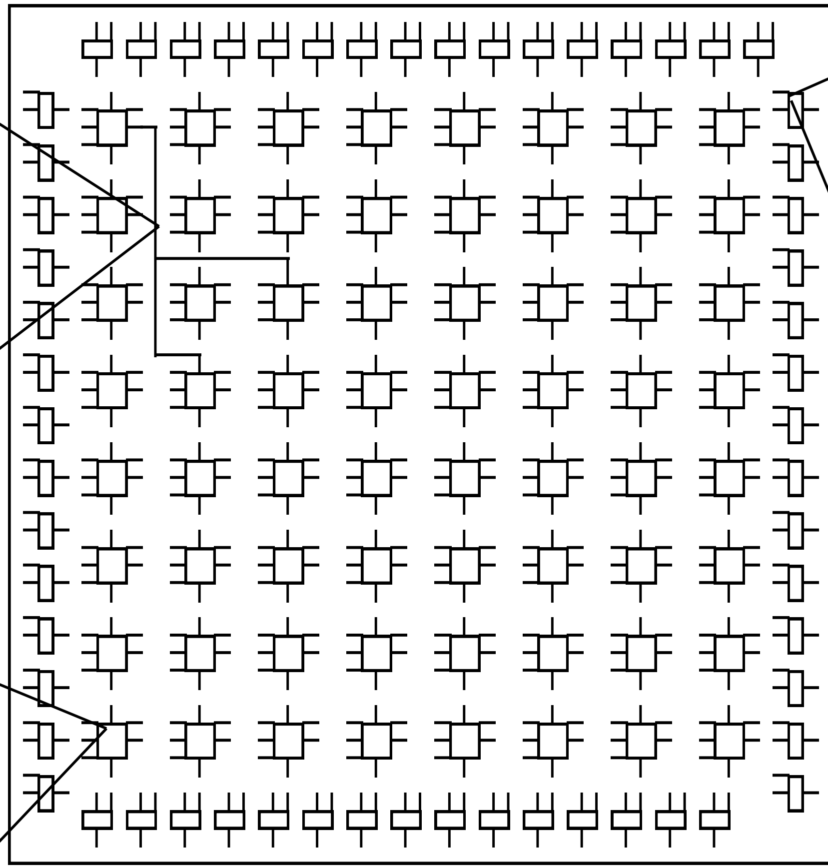
- Based on the fact that any combinational logic can be realized as a sum-of-products
- PALs feature an array of AND-OR gates with programmable interconnect

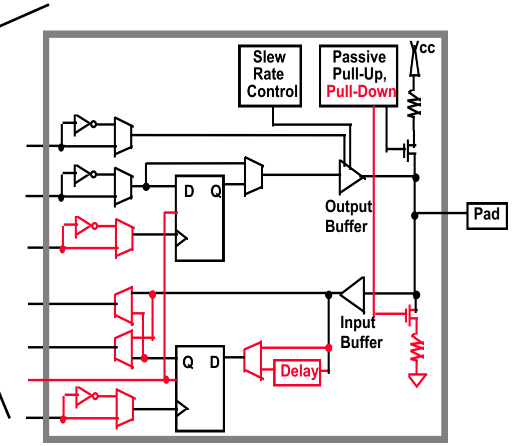


RAM Based Field Programmable Logic - Xilinx

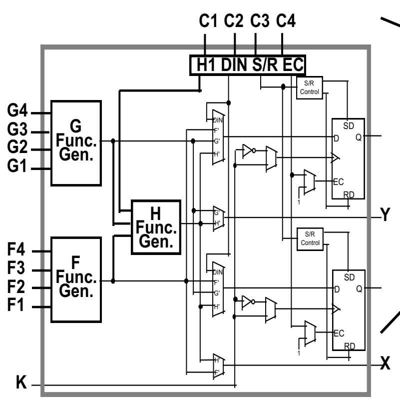


Programmable
Interconnect





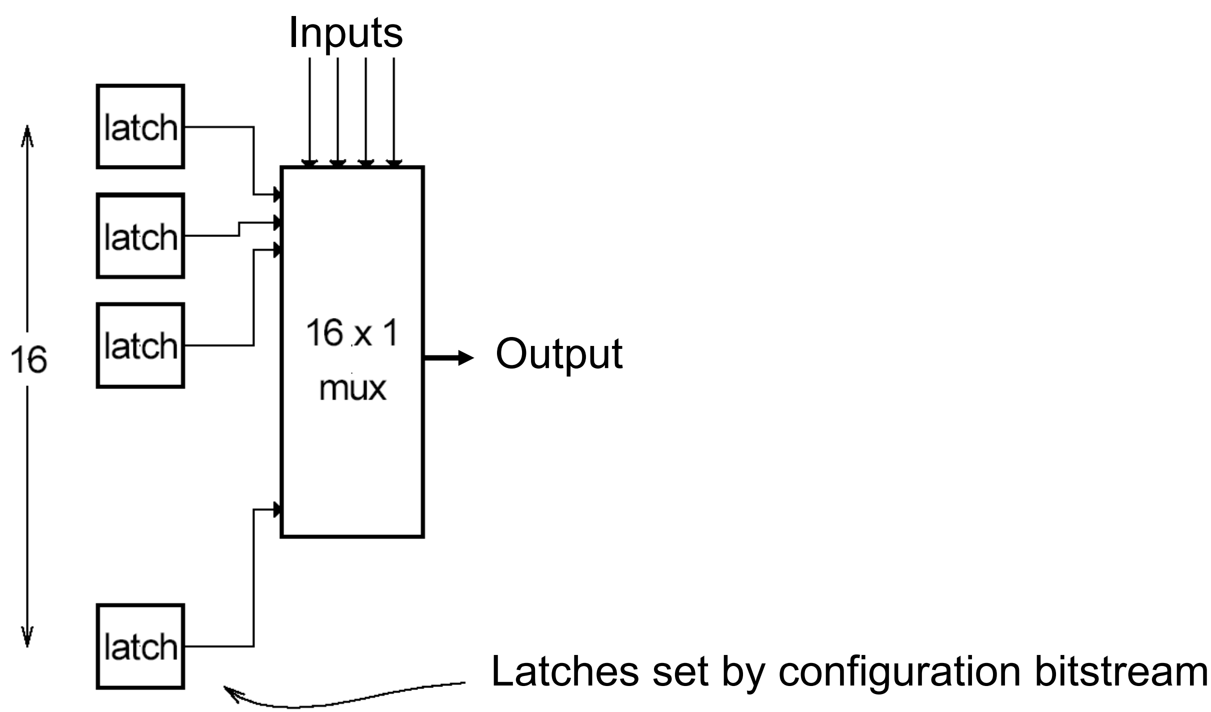
I/O Blocks (IOBs)



Configurable
Logic Blocks (CLBs)

LUT Mapping

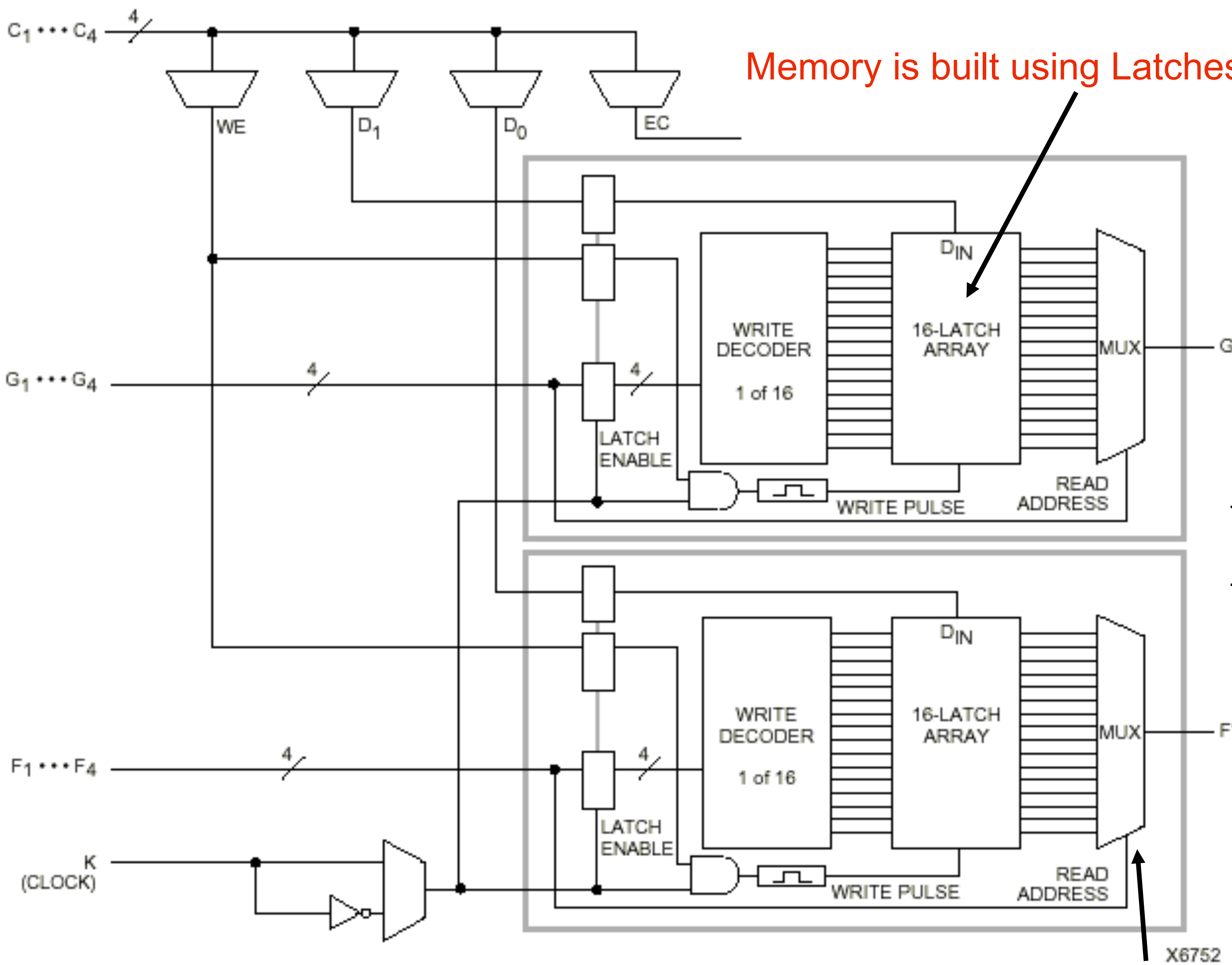
- N-LUT direct implementation of a truth table: any function of n-inputs.
- N-LUT requires 2^N storage elements (latches)
- N-inputs select one latch location (like a memory)



4LUT example

Configuring the CLB as a RAM

Memory is built using Latches not FFs



16x2

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

Read is same a LUT Function!

Xilinx 4000 Interconnect

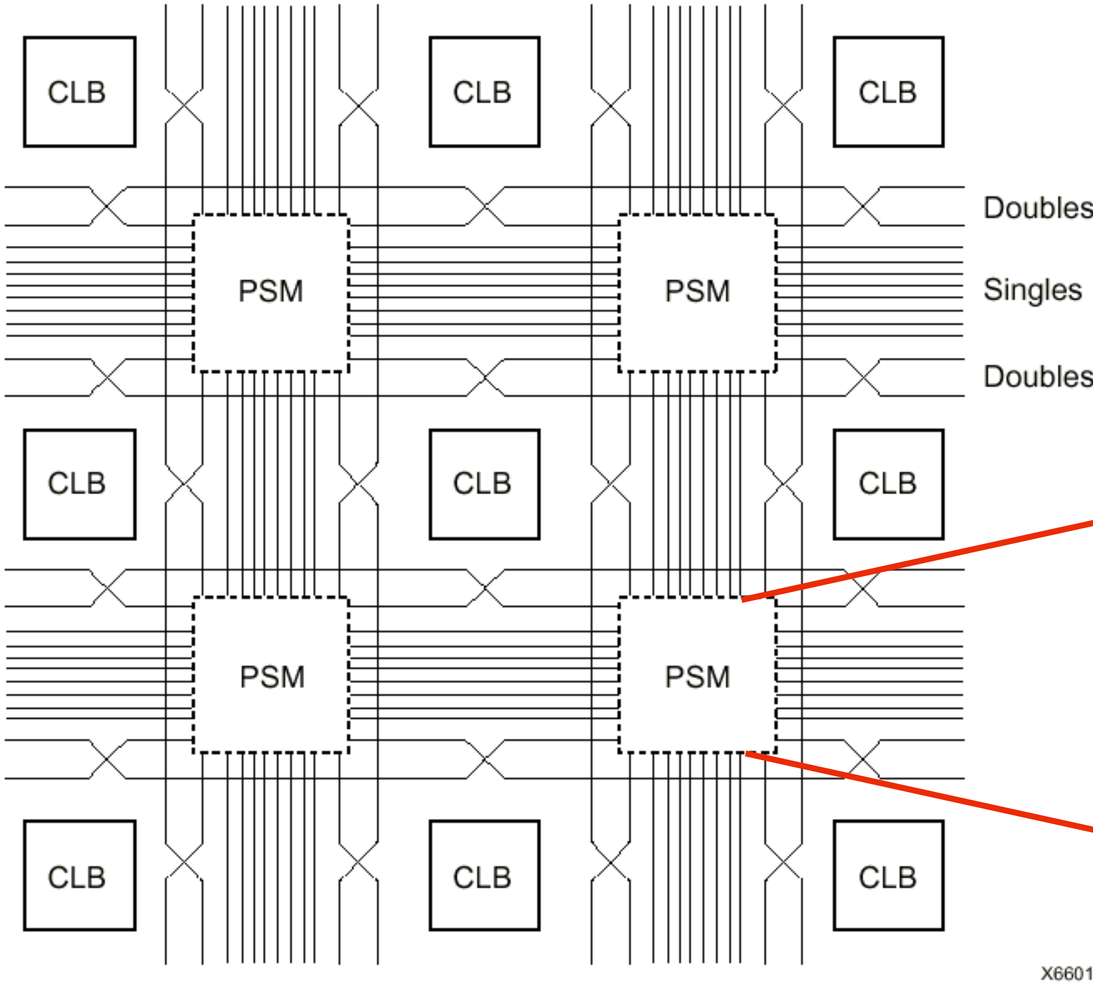
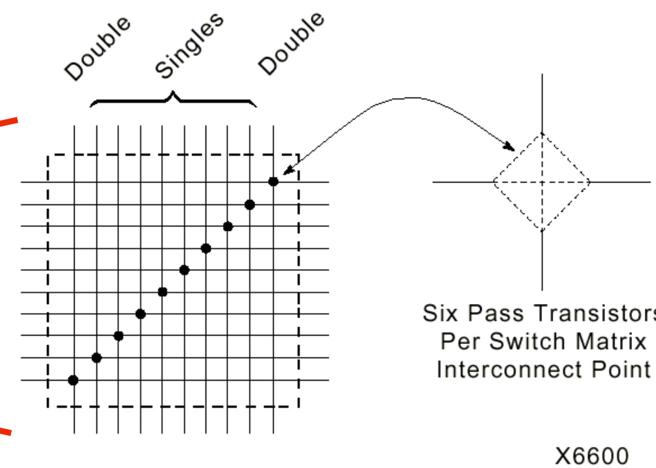
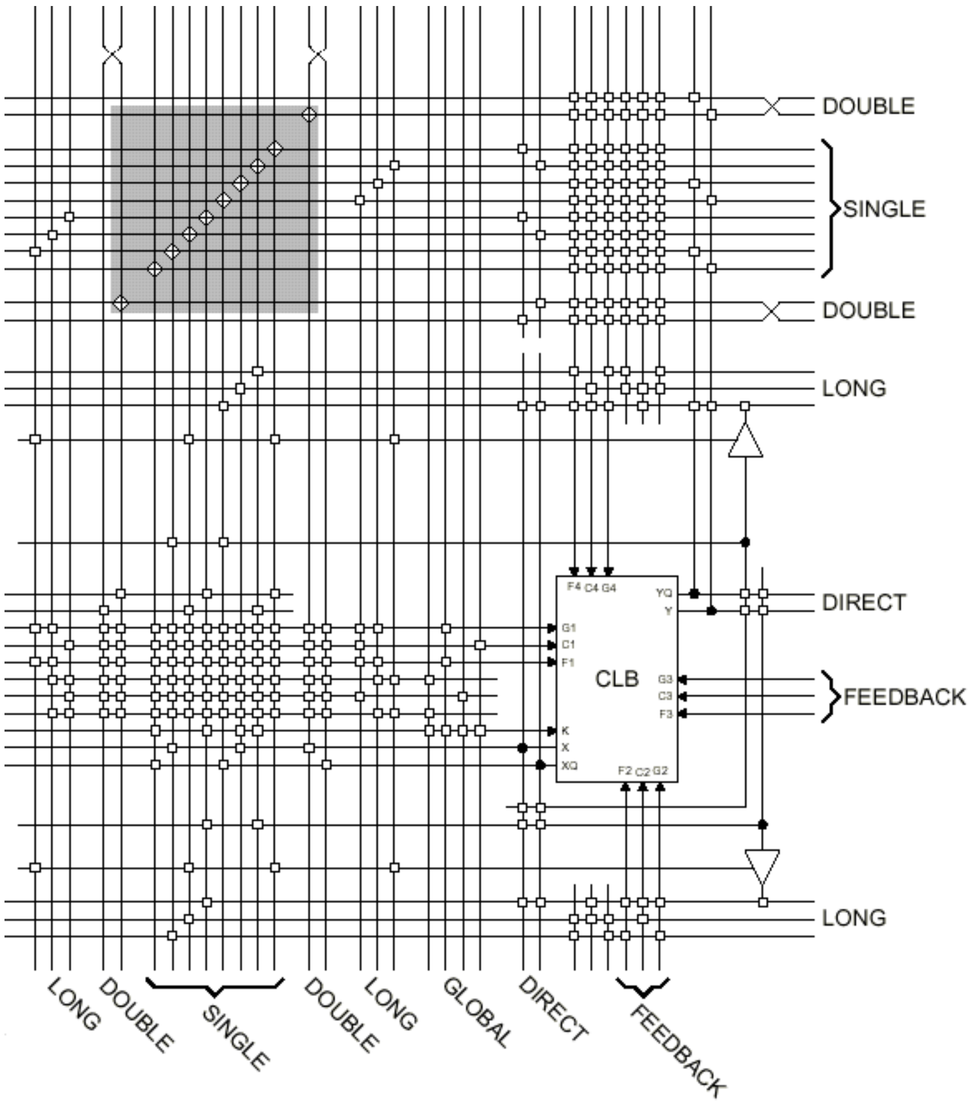


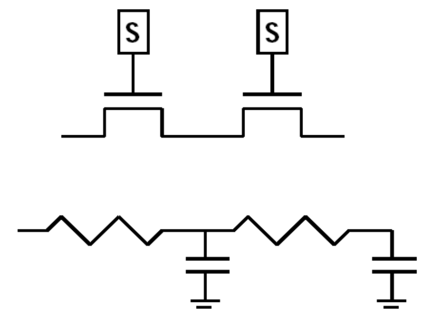
Figure 28: Single- and Double-Length Lines, with Programmable Switch Matrices (PSMs)



: Programmable Switch Matrix (PSM)

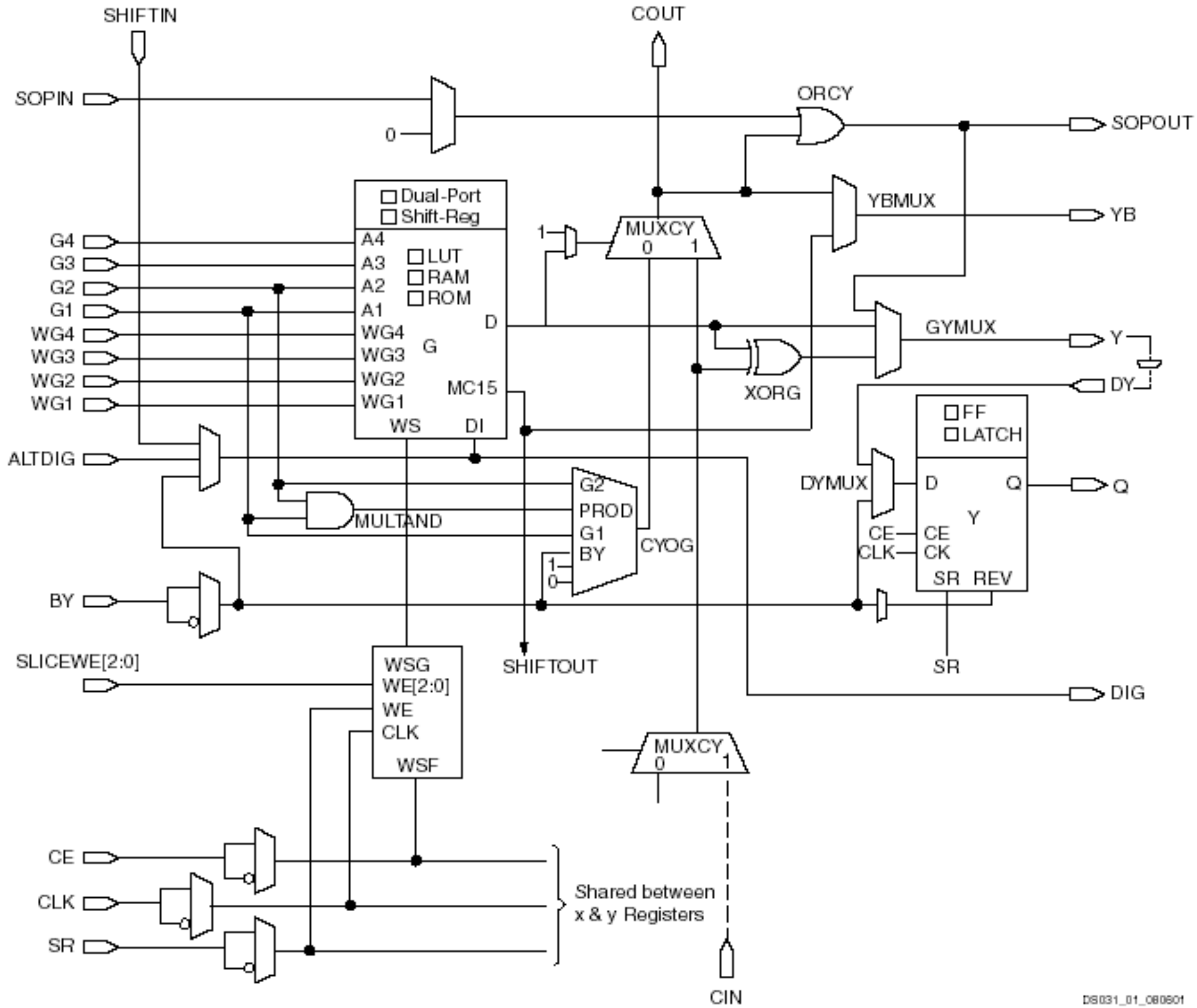
Xilinx Interconnect Details



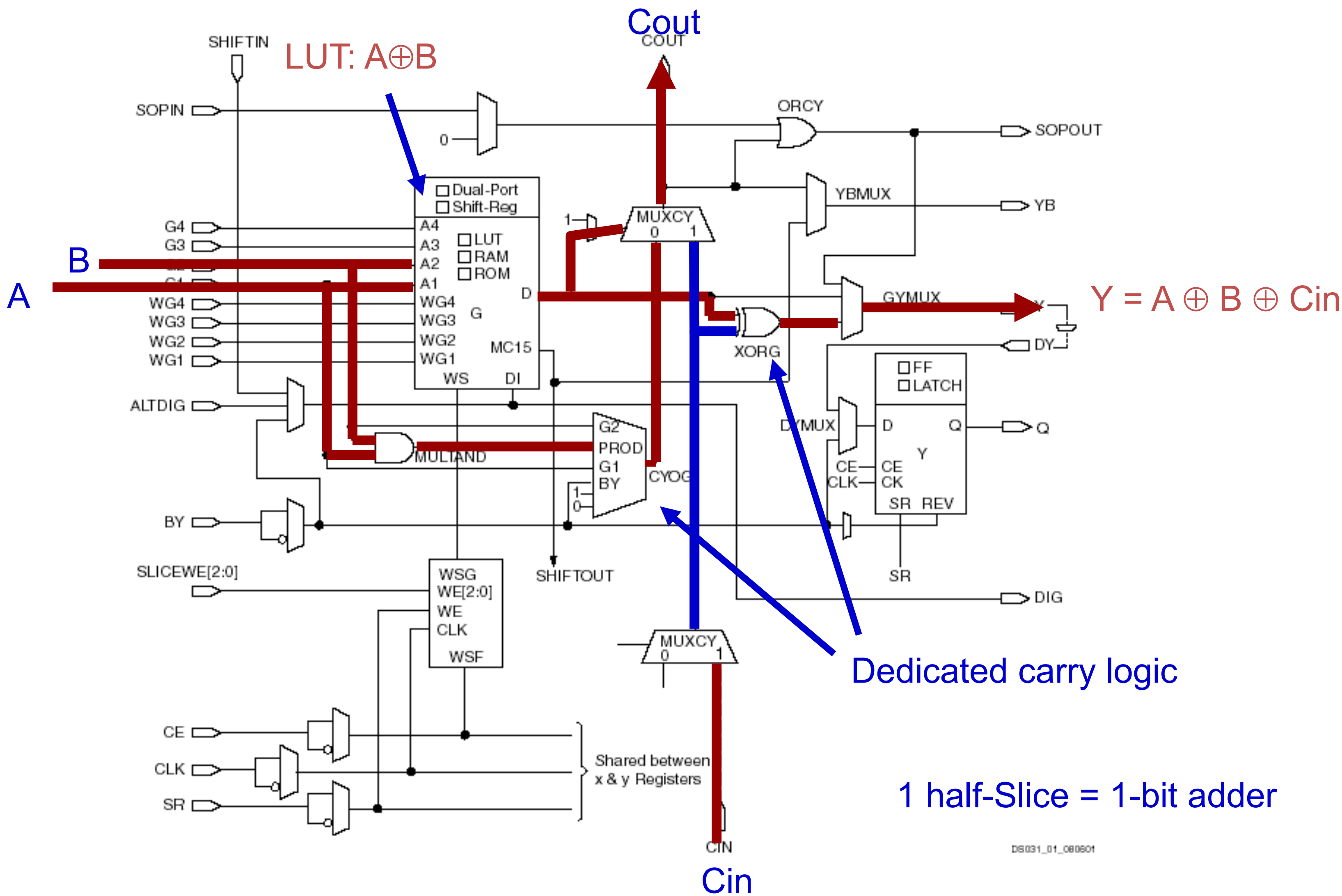


Wires are not ideal!

The Virtex II CLB (Half Slice Shown)



Adder Implementation



1 half-Slice = 1-bit adder

FPGA's

DSP with 25x18 multiplier

Gigabit ethernet support

Part Number	LX75T	LX130T	LX195T	LX240T	LX365T	LX550T	LX760	SX315T	SX475T
Logic Cells	74.5K	128K	200K	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

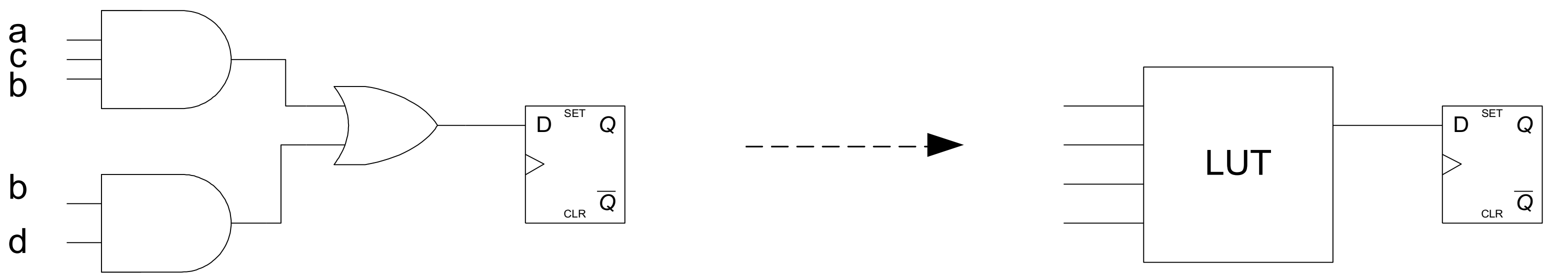
Package	Area (Pitch)	Maximum User I/O: Select IO™ Interface Pins (GTX Transceivers)								
FF484	23 x 23 mm (1.0 mm)	240 (8)	240 (8)							
FF784	29 x 29 mm (1.0 mm)	360 (12)	400 (12)	400 (12)	400 (12)					
FF1156	35 x 35 mm (1.0 mm)		600 (20)	600 (20)	600 (20)	600 (20)				
FF1759	42.5 x 42.5mm (1.0 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.

	CLB	Dist RAM	Block RAM	Multipliers
Virtex 2	8,448	1,056 kbit	2,592 kbit	144 (18x18)
Virtex 6	667,000	6,200 kbit	22,752 kbit	1,344 (25x18)
Spartan 3E	240	15 kbit	72 kbit	4 (18x18)
Artix-7 A100	7,925	1,188 kbit	4,860 kbit	240 (25x18)

Design Flow - Mapping

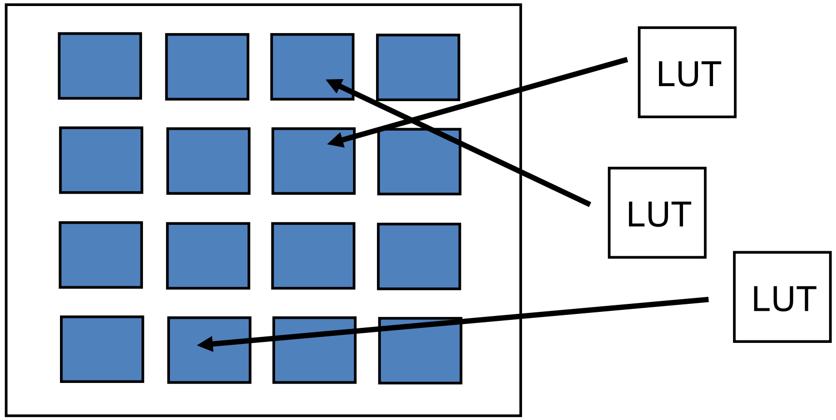
- Technology Mapping: Schematic/HDL to Physical Logic units
- Compile functions into basic LUT-based groups (function of target architecture)



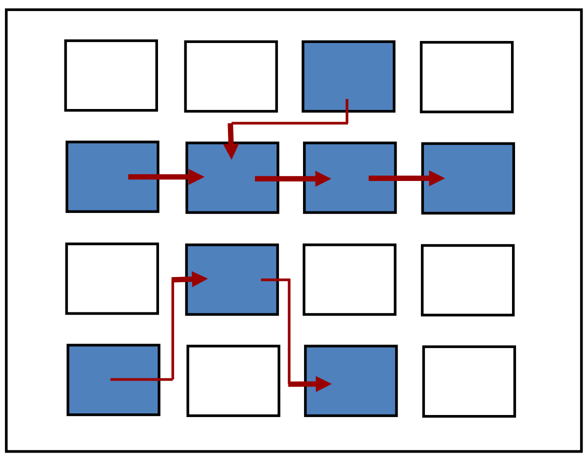

```
always @(posedge clock or negedge reset)
  begin
    if (! reset)
       q <= 0;
    else
       q <= (a&b&c)||(b&d);
  end
```

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

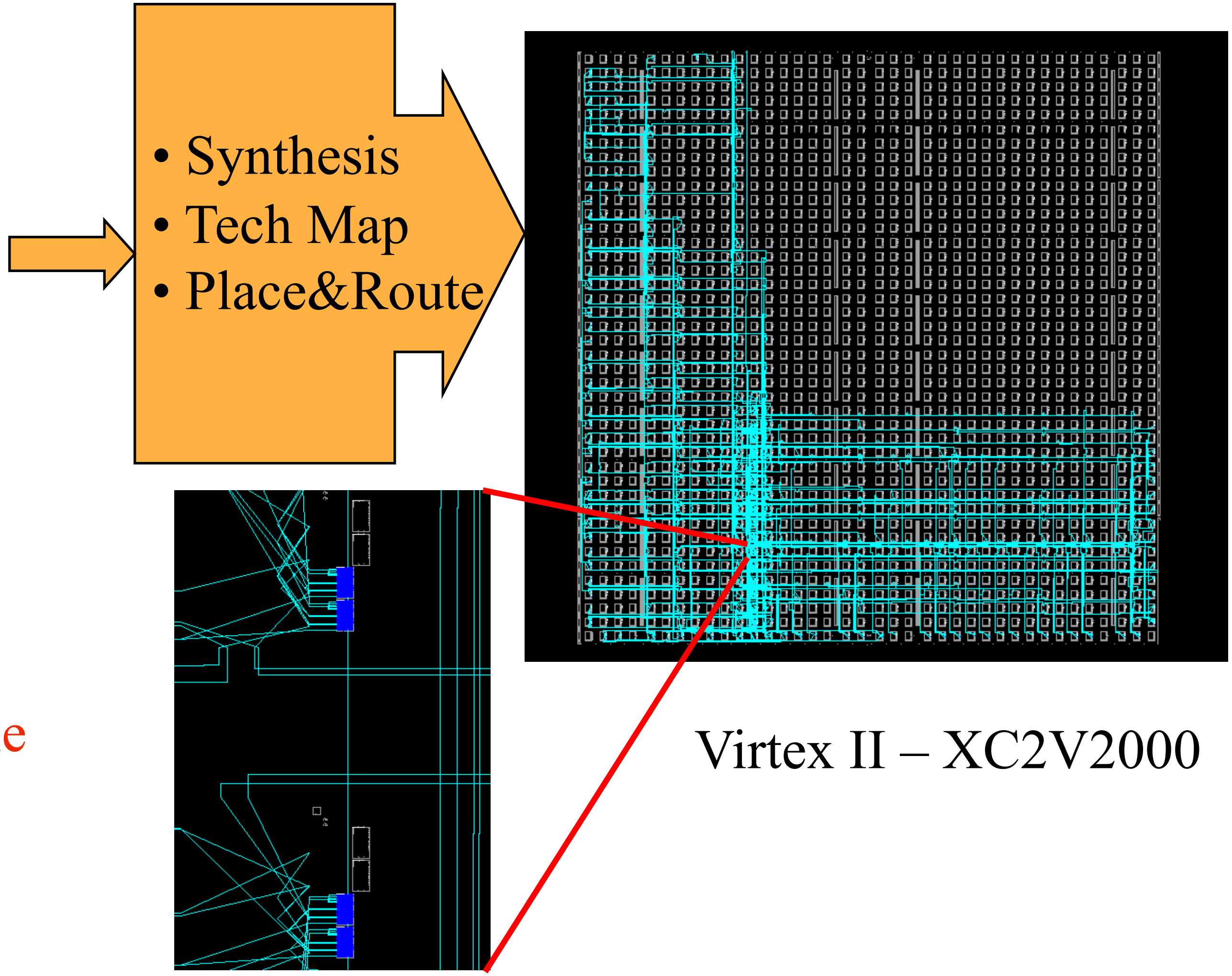
Satisfy timing? → Generate Bitstream to config device

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

Typically no more than 50% utilization.

Example: Verilog to FPGA

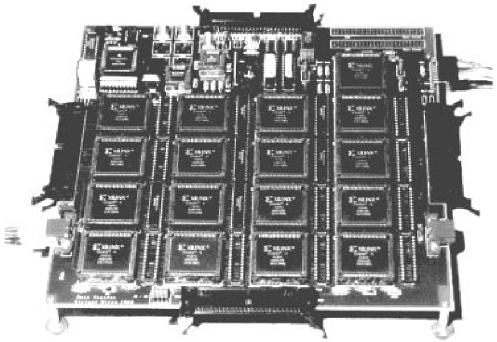
```
module adder64 (
  input  [63:0] a, b;
  output [63:0] sum);

  assign sum = a + b;
endmodule
```

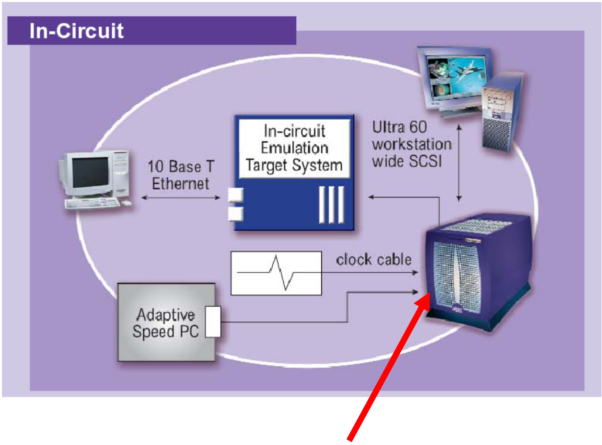


64-bit Adder Example

How are FPGAs Used?

Logic Emulation





FPGA-based Emulator

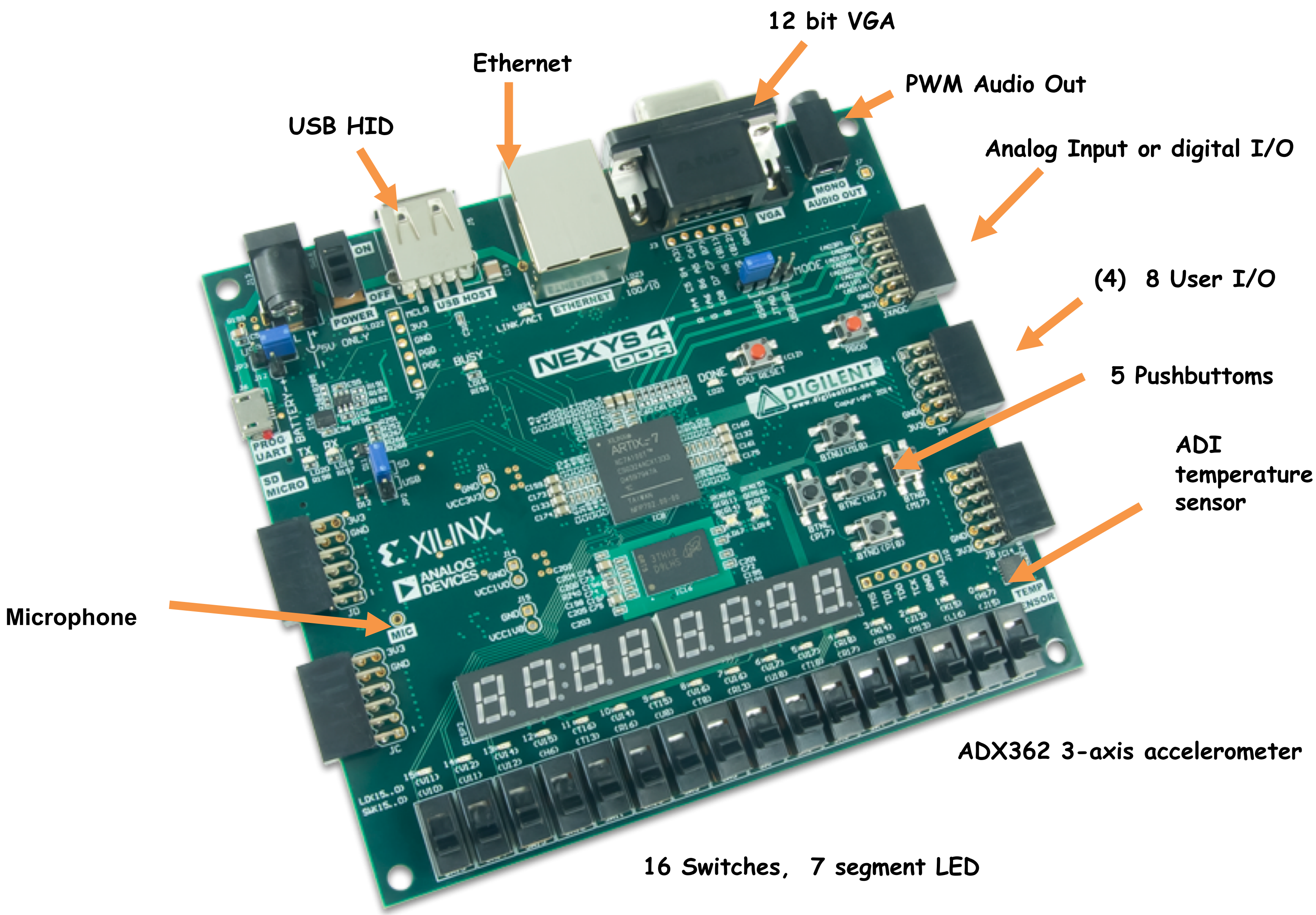
(courtesy of IKOS)

- 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)

Nexys 4 - DDR


12 bit VGA
Ethernet
PWM Audio Out
USB HID
Analog Input or digital I/O
(4) 8 User I/O
5 Pushbuttoms
ADI temperature sensor
Microphone
ADX362 3-axis accelerometer
16 Switches, 7 segment LED

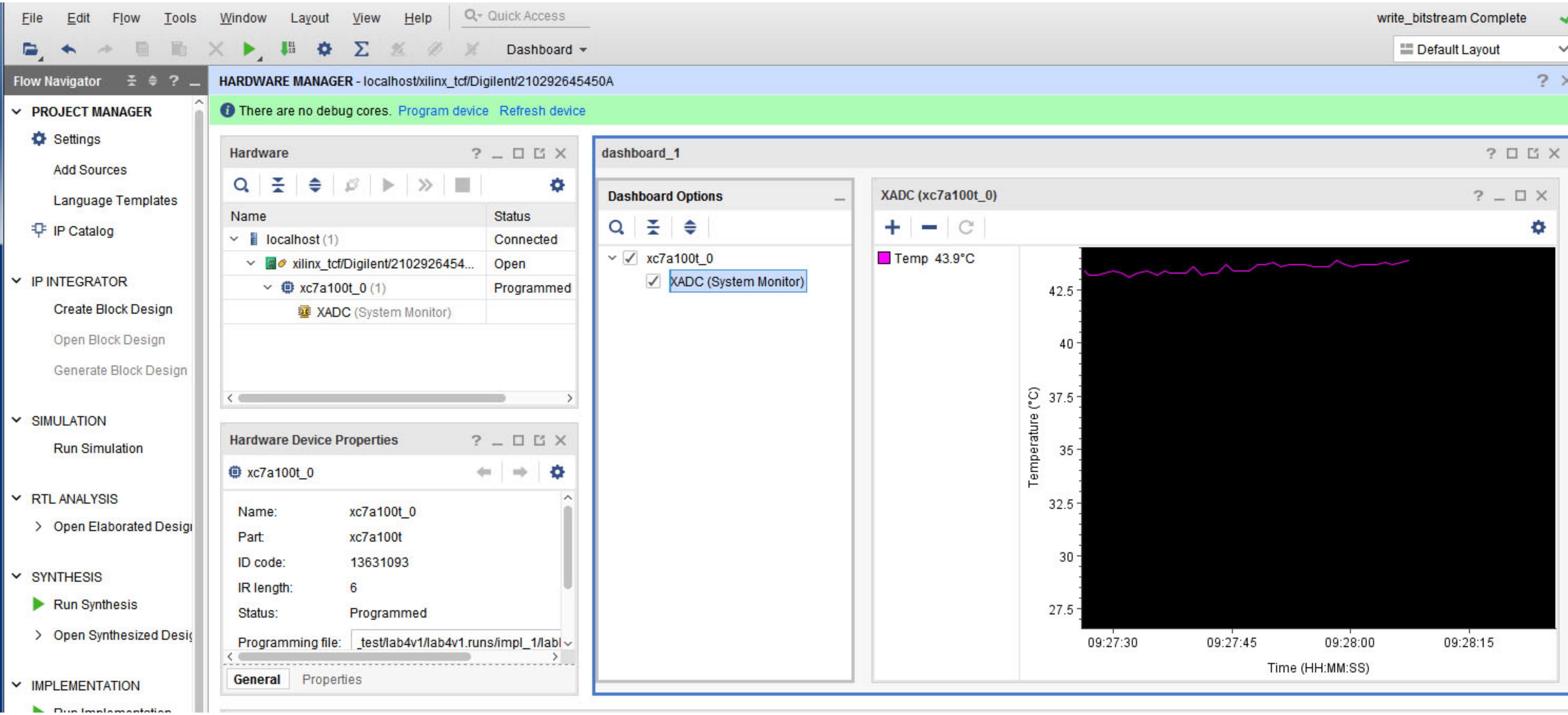
Nexy4 Input Output

```
module labkit(
    input CLK100MHZ,
    input[15:0] SW,
    input BTNC, BTNU, BTNL, BTNR, BTND,
    output[3:0] VGA_R,
    output[3:0] VGA_B,
    output[3:0] VGA_G,
    output[7:0] JA,
    output VGA_HS,
    output VGA_VS,
    output LED16_B, LED16_G, LED16_R,
    output LED17_B, LED17_G, LED17_R,
    output[15:0] LED,
    output[7:0] SEG,  // segments A-G (0-6), DP (7)
    output[7:0] AN    // Display 0-7
    );

    assign data = {28'h0123456, SW[3:0]}; // display 0123456 + SW
```

XDC File

```
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports { BTNC }];
      #IO_L9P_T1_DQS_14 Sch=btnc

set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports { BTNU }];
      #IO_L4N_T0_D05_14 Sch=btnu

set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports { BTNL }];
      #IO_L12P_T1_MRCC_14 Sch=btnl

set_property -dict { PACKAGE_PIN M17   IOSTANDARD LVCMOS33 } [get_ports { BTNR }];
      #IO_L10N_T1_D15_14 Sch=btnr

set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports { BTND }];
      #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

Dashboard



File Edit Flow Tools Window Layout View Help
Quick Access
write_bitstream Complete
Dashboard
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Desig
SYNTHESIS
Run Synthesis
Open Synthesized Desi
IMPLEMENTATION
HARDWARE MANAGER - localhost/xilinx_tcf/Digilent/210292645450A
There are no debug cores. Program device Refresh device
Hardware
Name
Status
localhost (1)
Connected
xilinx_tcf/Digilent/2102926454...
Open
xc7a100t_0 (1)
Programmed
XADC (System Monitor)
Hardware Device Properties
xc7a100t_0
Name: xc7a100t_0
Part: xc7a100t
ID code: 13631093
IR length: 6
Status: Programmed
Programming file: _test/lab4v1/lab4v1.runs/impl_1/labl
General Properties
dashboard_1
Dashboard Options
xc7a100t_0
XADC (System Monitor)
XADC (xc7a100t_0)
Temp 43.9°C
42.5
40
37.5
35
32.5
30
27.5
Temperature (°C)
09:27:30
09:27:45
09:28:00
09:28:15
Time (HH:MM:SS)

Loading Nexys4 Flash

1. Format a flash drive to have 1 fat32 partition

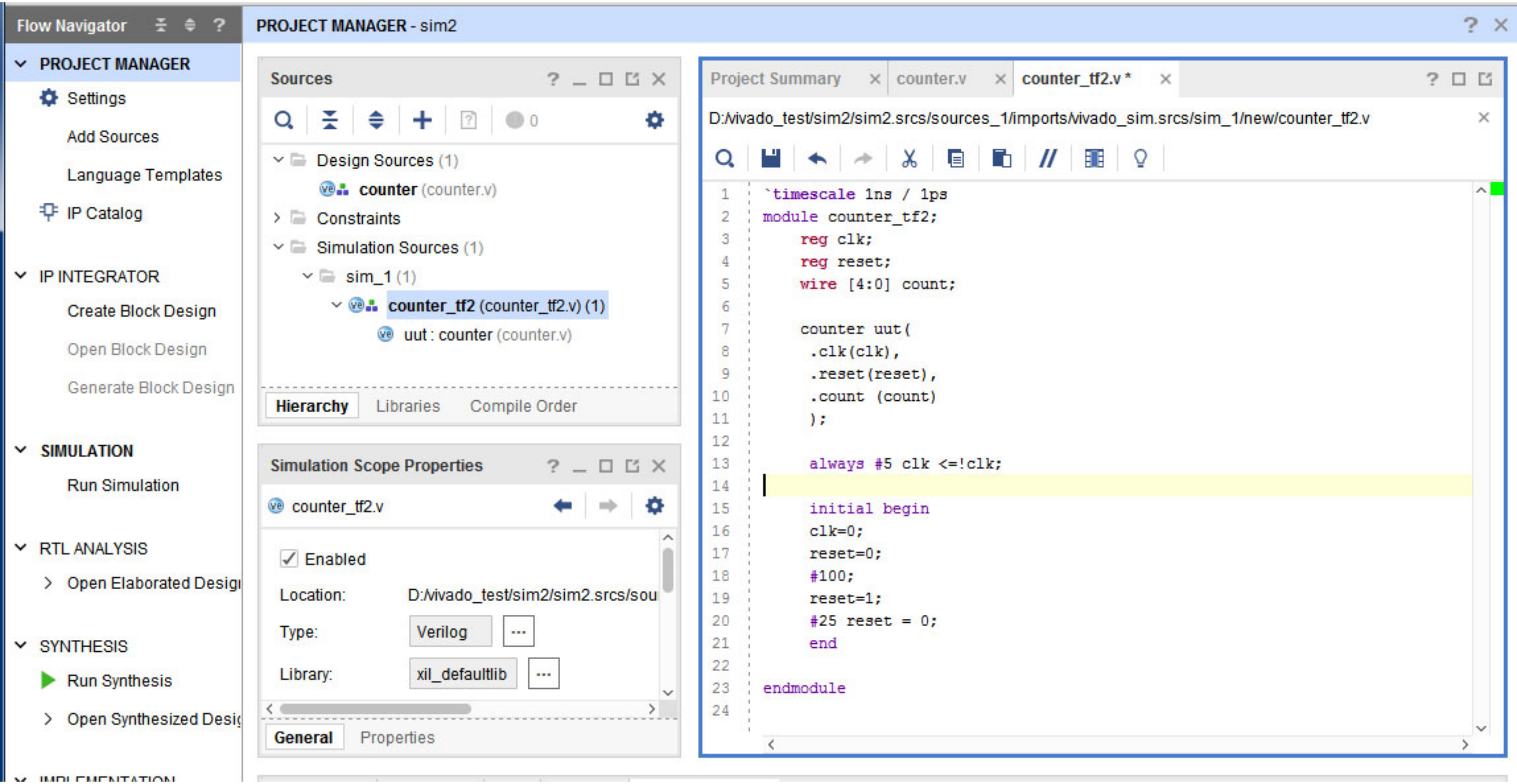
2. In vivado, click generate bitstream and afterwards do file->Export->Export_Bitstream_File to flash top-level directory

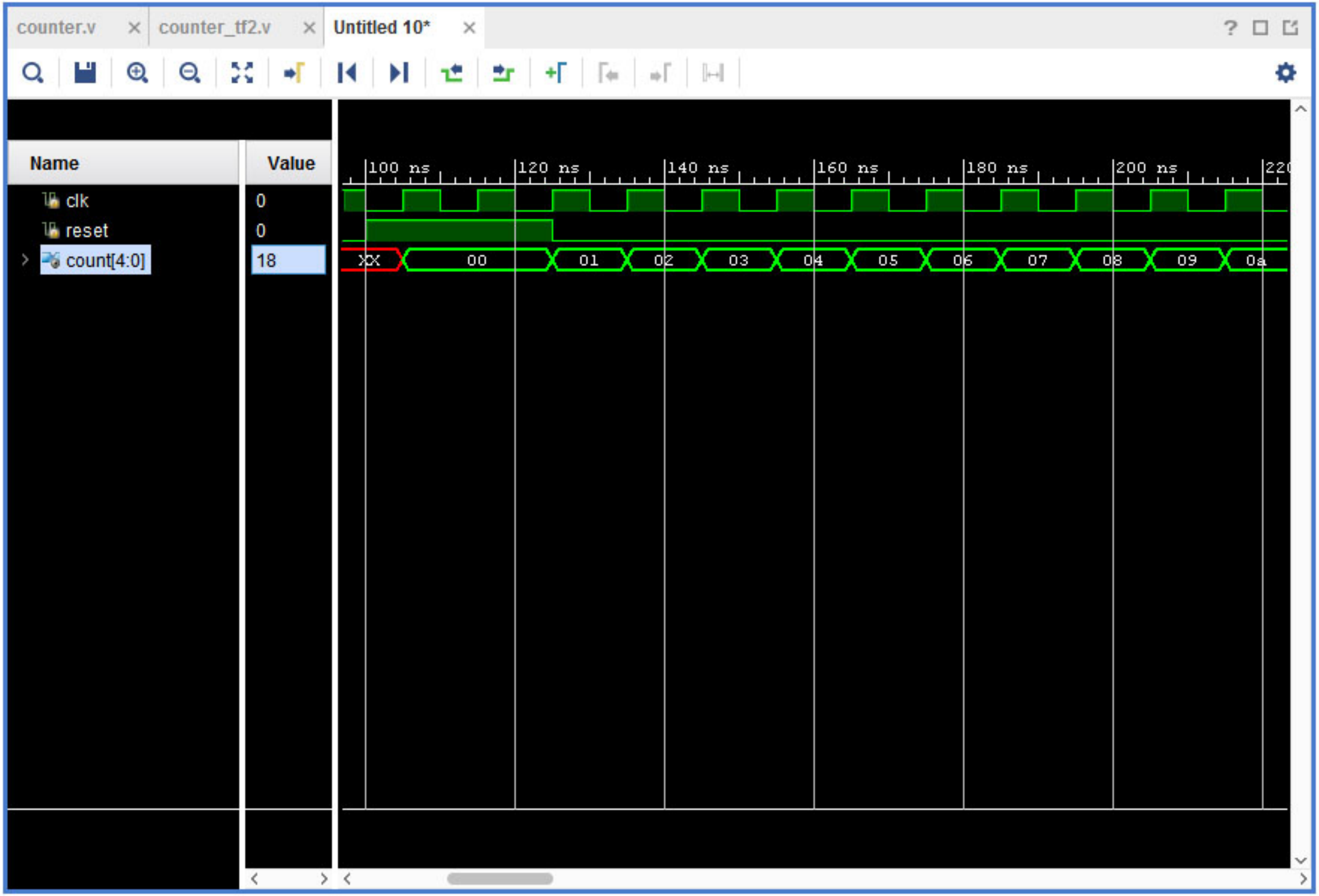
3. On the nexys 4, switch jumper JP1 to be on the USB/SD mode

4. Plug the usb stick into the nexys 4 while it's off and then power on. A yellow LED will flash while the bitstream is being loaded. When it's done, the green DONE led will turn on

5. You can remove the usb drive after your code is running

Vivado Simulation




counter.v
counter_tf2.v
Untitled 10*
Name
Value
clk
0
reset
0
count[4:0]
18
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns
XX
00
01
02
03
04
05
06
07
08
09
0a

Test Bench

```
module sample_tf;
   // Inputs
   reg bit_in;
   reg [3:0] bus_in;

   // Outputs
   wire out_bit;
   wire [7:0] out_bus;

   // Instantiate the Unit Under Test (UUT)
   sample uut (
      .bit_in(bit_in),
      .bus_in(bus_in),
      .out_bit(out_bit),
      .out_bus(out_bus)
   );

   initial begin
      // Initialize Inputs
      bit_in = 0;
      bus_in = 0;

      // Wait 100 ns for global reset to finish
      #100;

      // Add stimulus here

    end

endmodule
```

```
module sample(
   input bit_in,
   input [3:0] bus_in,

   output out_bit,
   output [7:0] out_bus
   );
 . . . Verilog . . .

endmodule
```

inputs must be initialized