Pipelining & Verilog

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

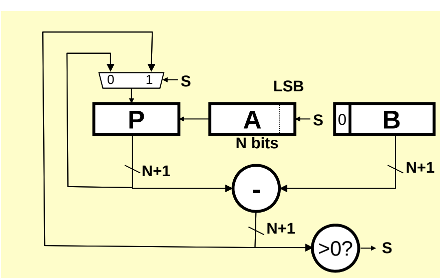
Table 7-9: **Supported Expressions**

Expression	Symbol	Status
Concatenation	{}	Supported
Replication	{{}}	Supported
Arithmetic	+, -, *,**	Supported
Division	/	Supported only if the second operand is a power of 2, or both operands are constant.
Modulus	%	Supported only if second operand is a power of 2.
Addition	+	Supported
Subtraction	-	Supported
Multiplication	*	Supported
Power	**	Supported: • Both operands are constants, with the second operand being non-negative. • If the first operand is a 2, then the second operand can be a variable. • Vivado synthesis does not support the real data type. Any combination of operands that results in a real type causes an error. • The values X (unknown) and Z (high impedance) are not allowed.
Relational	>, <, >=, <=	Supported
Logical Negation	!	Supported
Logical AND	&&	Supported

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug901-vivado-synthesis.pdf

Sequential Divider

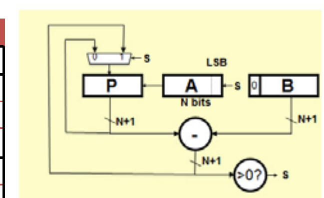
Assume the Dividend (A) and the divisor (B) have N bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single subtraction at a time and then cycle the circuit N times. This circuit works on unsigned operands; for signed operands one can remember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

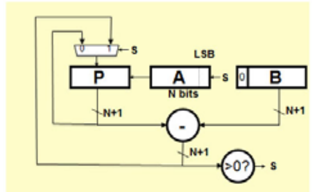
P	A	P-B	7/3 0111/11 B=0011
0000	0111		Initial value
0000	1110		Shift
0000		-3	Subtract
0000	1110		Restore, set A_{lsb} = 0
0001	1100		Shift
0001		-2	Subtract
0001	1100		Restore, set A_{lsb} = 0
0011	1000		Shift
0011		0	Subtract
0000	1001		Subtact, set A_{lsb} = 1
0001	0010		Shift
0001		-2	Subtract
0001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

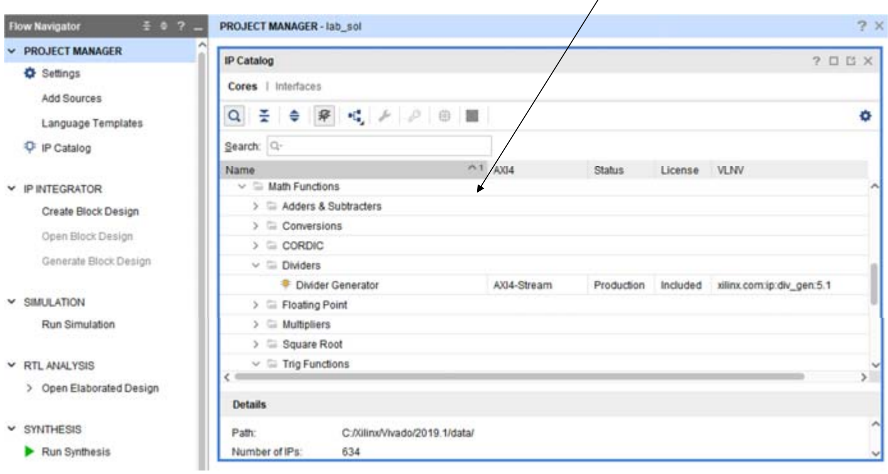

Verilog divider.v

```
// 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
```

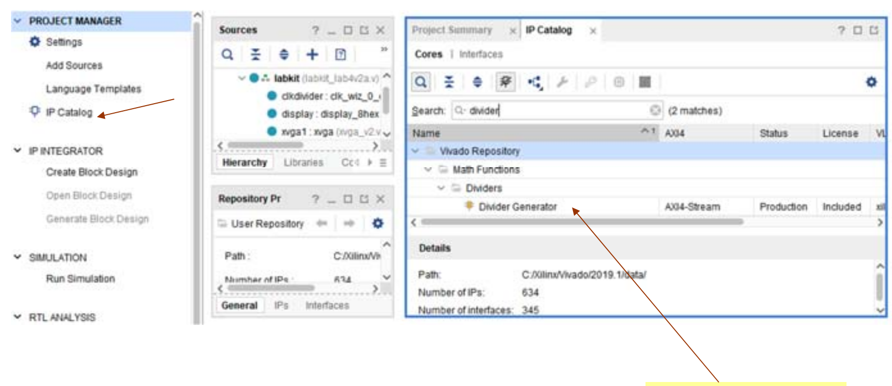
L. Williams MIT '13

Math & Other Functions in IP Catalog

Wide selection of math functions available



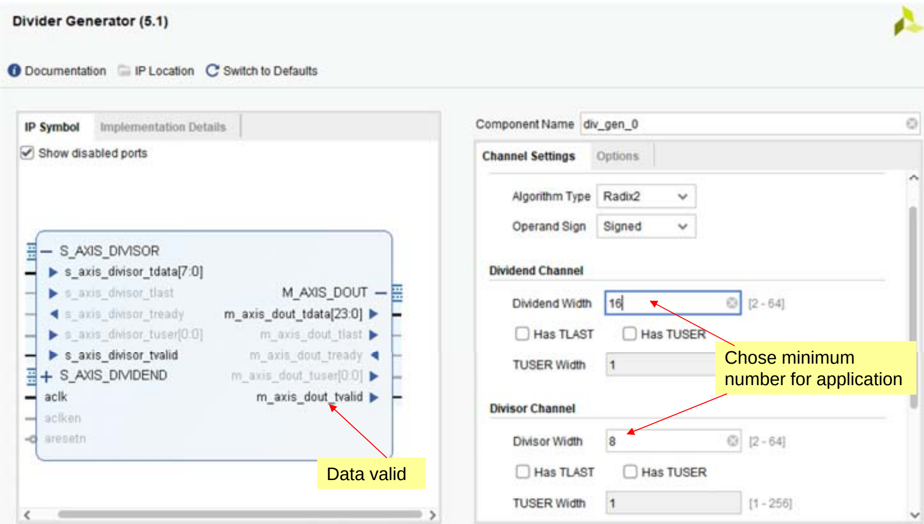
Divider Generator



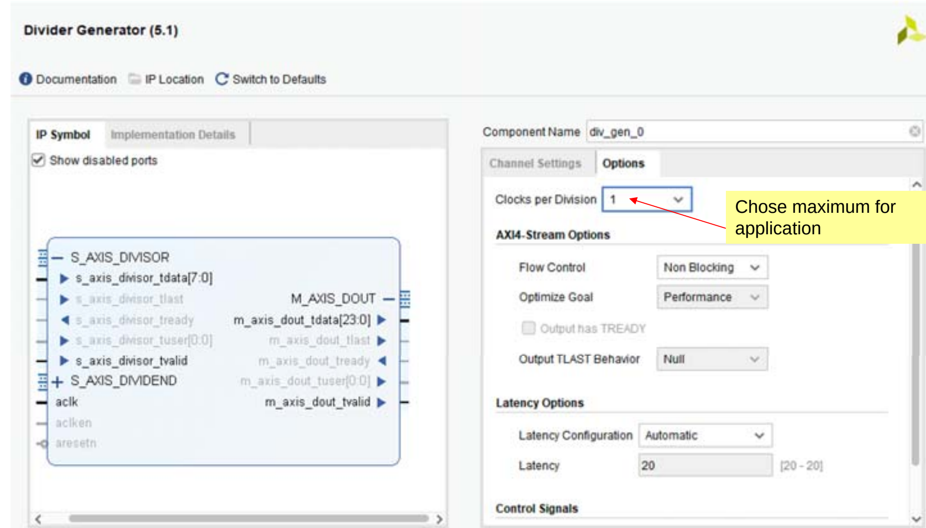
Select Divider

https://www.xilinx.com/support/documentation/ip_documentation/div_gen/v5_1/pg151-div-gen.pdf

IP Catalog Divider



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

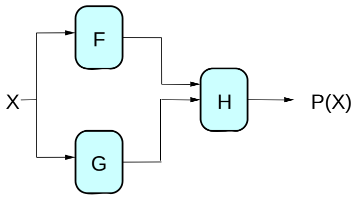
Latency dependent on dividend width + fractioanl reminder width

Table 2-1: **Latency of Radix-2 Solution Based on Divider Parameters**

Signed	Fractional	Clocks Per Division	Fully Pipelined Latency(1)
FALSE	FALSE	1	M+A+2
FALSE	FALSE	>1	M+A+3
FALSE	TRUE	1	M+F+A+2
FALSE	TRUE	>1	M+F+A+3
TRUE	FALSE	1	M+A+4
TRUE	FALSE	>1	M+A+5
TRUE	TRUE	1	M+F+A+4
TRUE	TRUE	>1	M+F+A+5

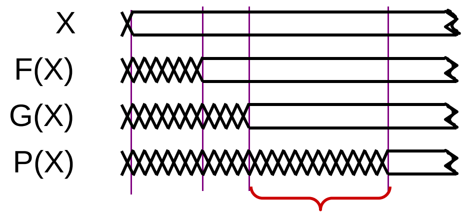
Notes:
1. M = Dividend and Quotient Width, F = Fractional Width, A = total Latency of AXI interfaces.

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?

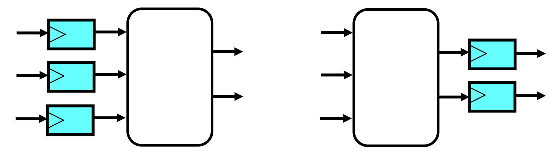


F & G are "idle", just holding their outputs stable while H performs its computation

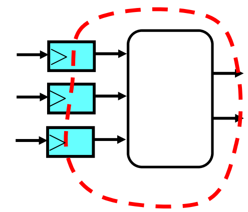
Retiming: A very useful transform

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

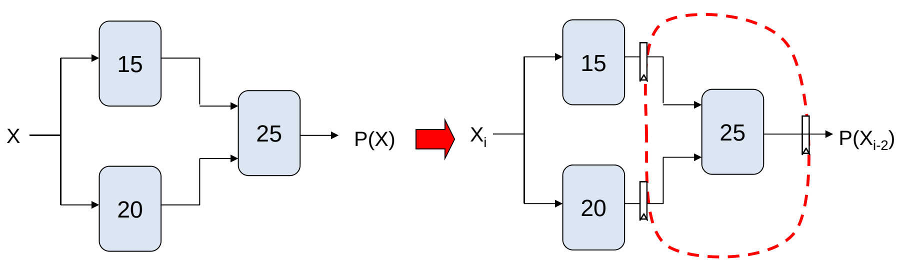


Cutset retiming: A cutset intersects the edges, such that this would result in two disjoint partitions of the edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.



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

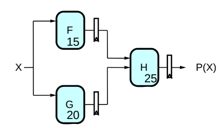
Retiming Combinational Circuits aka "Pipelining"



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

Assuming ideal registers: i.e., $t_{PD} = 0$, $t_{SETUP} = 0$ → $t_{CLK} = 25$
$L = 2 * t_{CLK} = 50$
$T = 1/t_{CLK} = 1/25$

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:

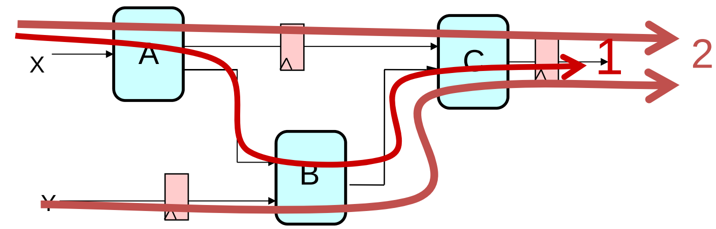
The CLOCK common to all registers must have a period sufficient to cover propagation over combinational paths PLUS (input) register t_{PD} PLUS (output) register t_{SETUP}.

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.

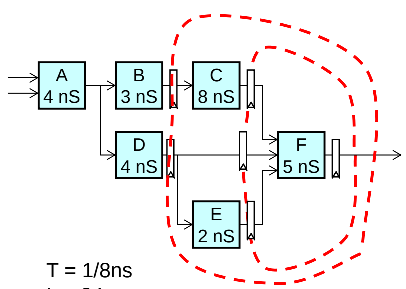
Step 2:

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.

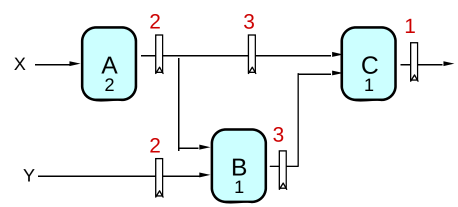
STRATEGY:

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



T = 1/8ns
L = 24ns

Pipeline Example

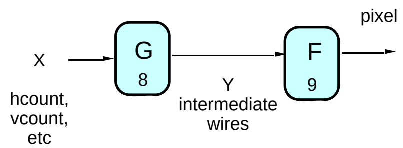


	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

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.

Pipeline Example - Verilog

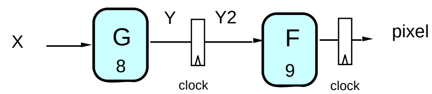


Lab 3 Pong

- G = game logic 8ns tpd
- C = draw fancy object puck, lots of multiplies 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
```



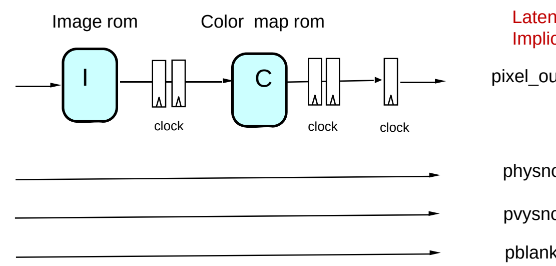

```
Pipeline
   always @(posedge clock)  begin
     ...
     y2 <= G(x);       // pipeline y
     pixel <= C(y2)    // pipeline pixel
end
```

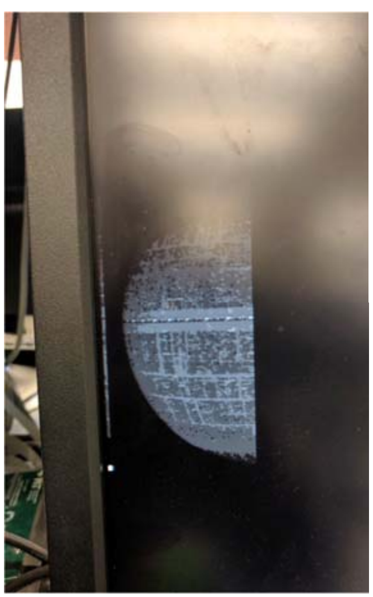
Latency = 2 clock cyles!
Implications?

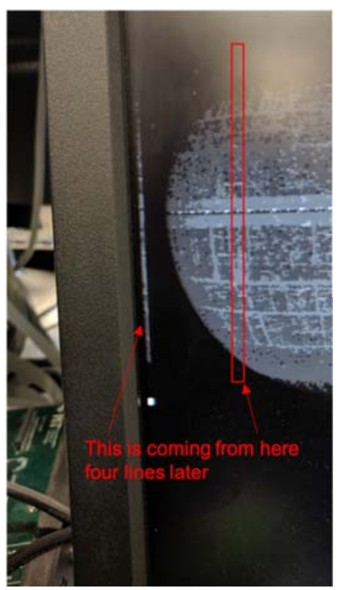
Pipeline Example – Lab 3

```
// calculate rom address and read the location
assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
image_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

red_coe rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));

always @ (posedge pixel_clk) begin
  if ((hcount_in >= x && hcount_in < (x_in+WIDTH)) &&. (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))

  pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale

   else pixel_out <= 0;
end
```

Latency = 5 clock cyles!
Implications?



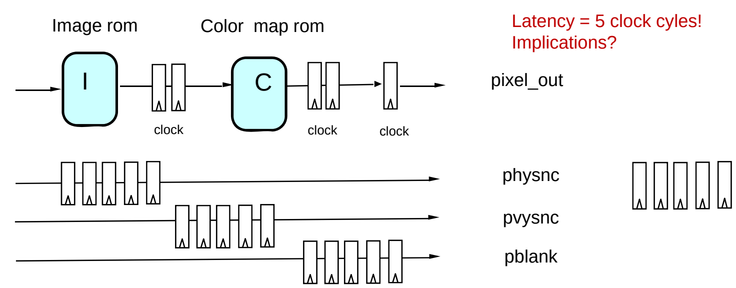




Pipeline Example – Lab 3

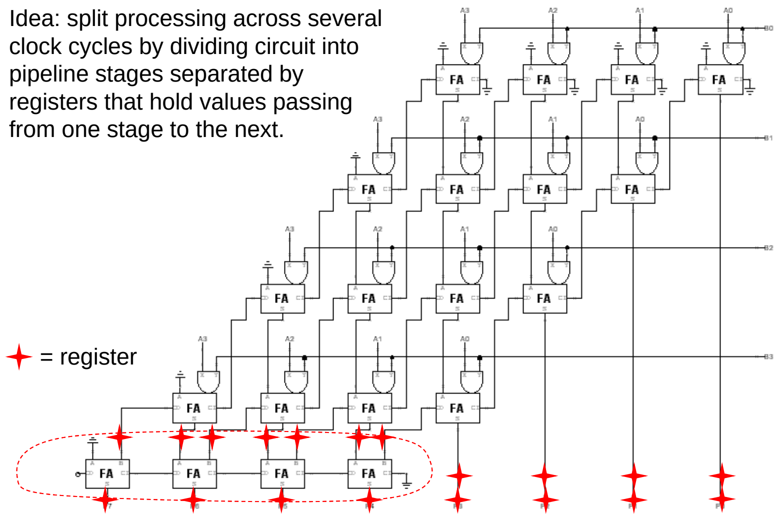
```
// calculate rom address and read the location
assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
image_rom  rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits));

red_coe rcm (.clka(pixel_clk_in), .addra(image_bits), .douta(red_mapped));

always @ (posedge pixel_clk) begin
  if ((hcount_in >= x && hcount_in < (x_in+WIDTH)) &&. (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))

  pixel_out <= {red_mapped[7:4], red_mapped[7:4], red_mapped[7:4]}; // greyscale

   else pixel_out <= 0;
end
```

Latency = 5 clock cyles!
Implications?



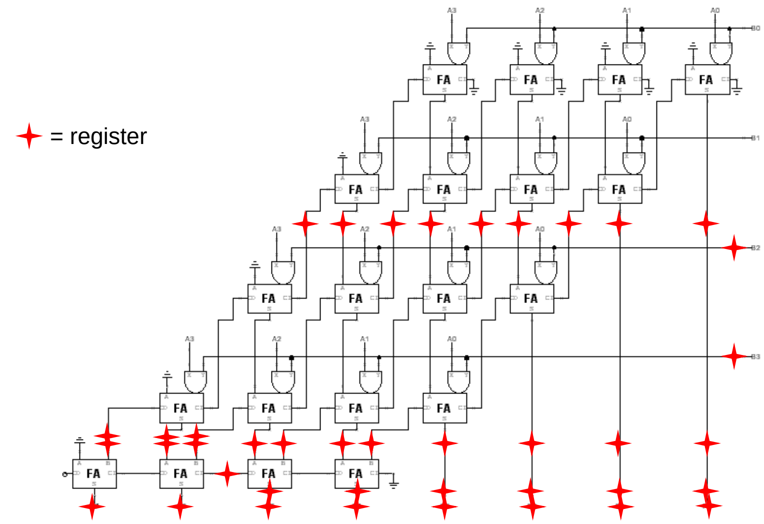
Increasing Throughput: Pipelining

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

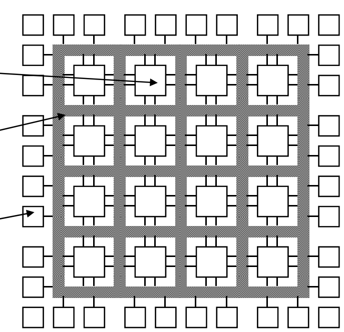


History of Computational Fabrics

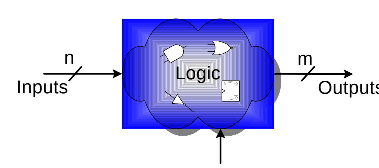
- Discrete devices: relays, transistors (1940s-50s)
- Discrete logic gates (1950s-60s)
- Integrated circuits (1960s-70s)
  - e.g. TTL packages: Data Book for 100's of different parts
- Gate Arrays (IBM 1970s)
  - Transistors are pre-placed on the chip & Place and Route software puts the chip together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

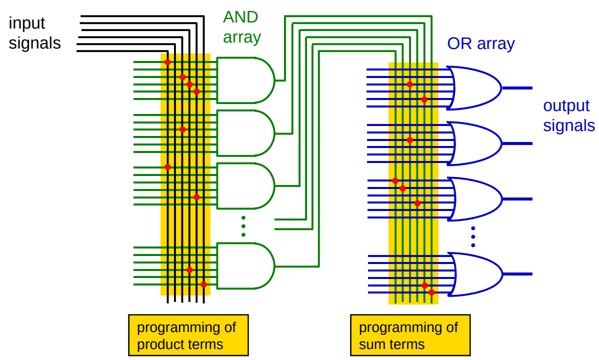


- Key questions:
  - How to make logic blocks programmable? (after chip has been fabbed!)
  - What should the logic granularity be?
  - How to make the wires programmable? (after chip has been fabbed!)
  - Specialized wiring structures for local vs. long distance routes?
  - How many wires per logic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

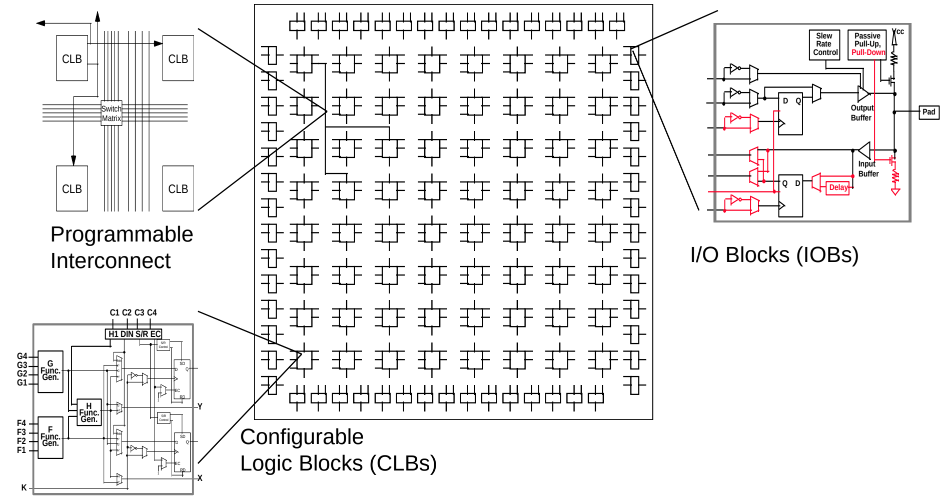


Programmable Array Logic (PAL)

- 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 - FPGA



FPGA RAM based Interconnect

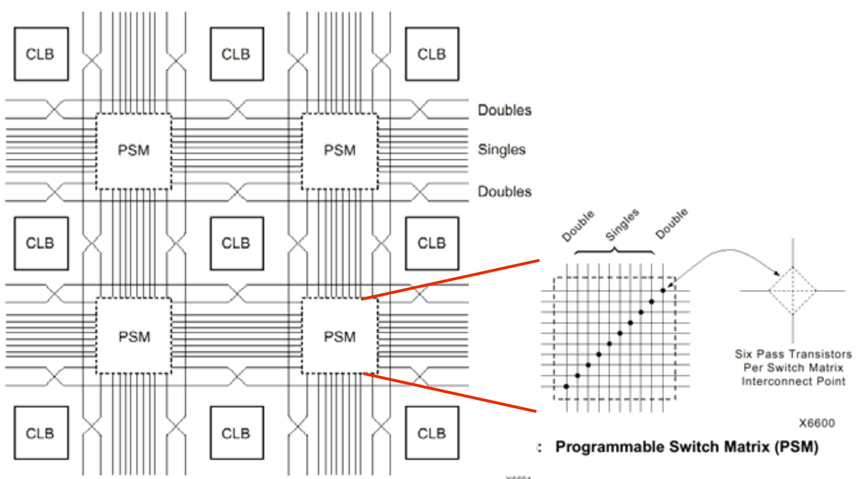
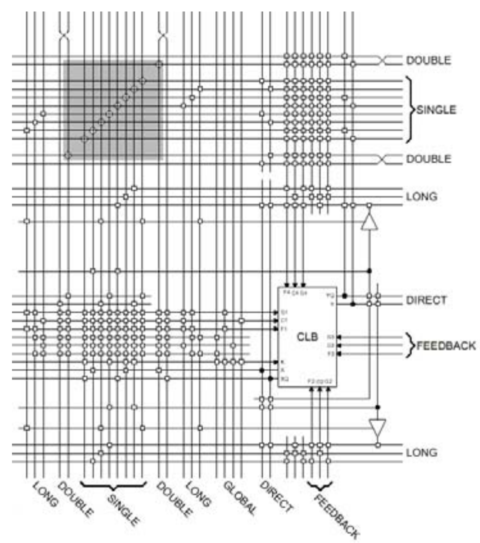
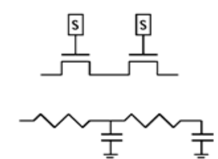


Figure 28: Single- and Double-Length Lines, with Programmable Switch Matrices (PSMs)

Xilinx Interconnect Details

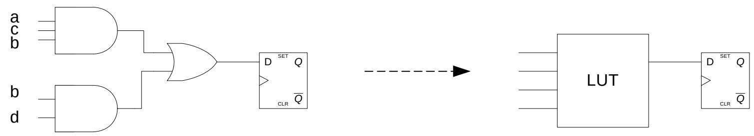




Wires are not ideal!

Design Flow - Mapping

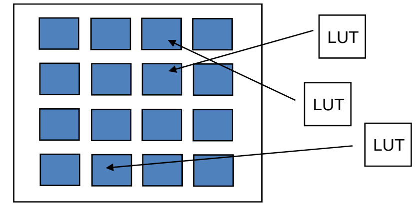
- Technology Mapping: Schematic/HDL to Physical Logic units
- Compile functions into basic LUT-based groups (function of target architecture)



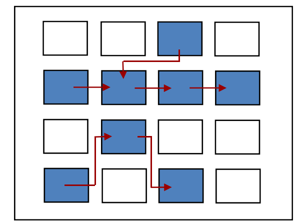

```
always @(posedge clock or negedge reset)
  begin
    if (! reset)
       q <= 0;
    else
       q <= (a&b&c)||(b&d);
  end
```


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*



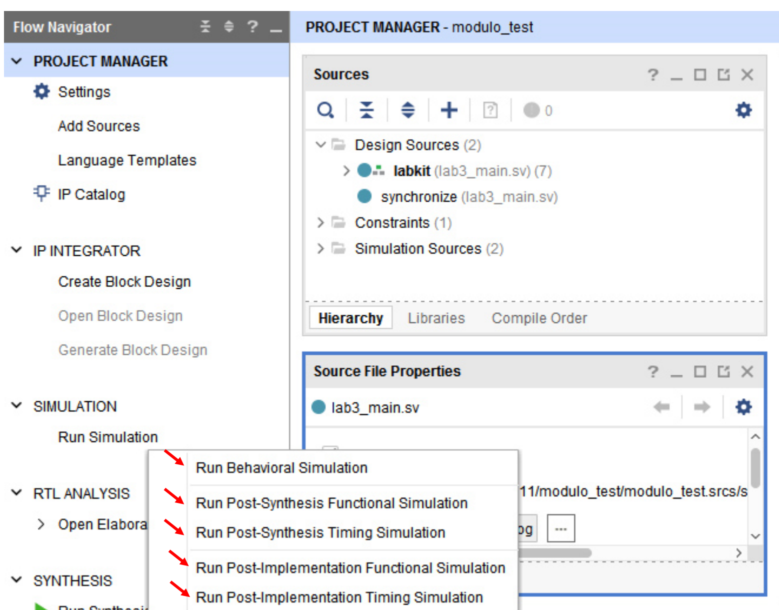
Iterate placement if timing not met

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.

Simulation – Five Options



Simulations

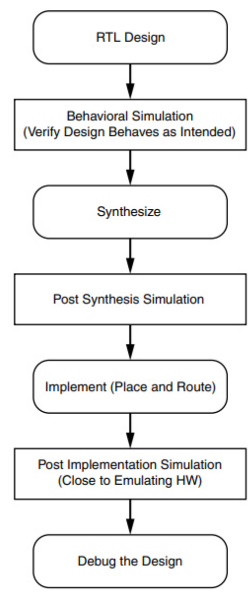


Figure 1-1: **Simulation Flow**

Uses the Verilog source code to model the behavior of the module under test. Neither gate delays nor interconnect delays are modeled. Behavioral model may not match that of the synthesized logic.

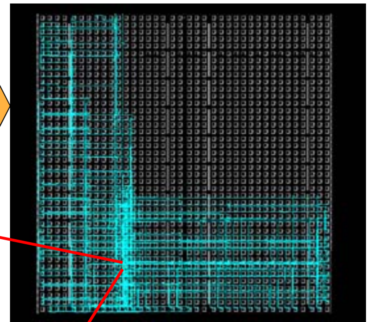
Synthesized netlist mapped to the FPGA device being targeted. Simulation includes some timing information. But no interconnect delay.

Can perform functional or timing simulation after implementation. Timing simulation is the closest emulation to actually downloading a design to a device. Can ensure that the implemented design meets functional and timing requirements

Example: Verilog to FPGA

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

  assign sum = a + b;
endmodule
```

- Synthesis
- Tech Map
- Place&Route

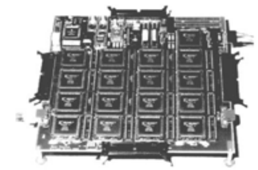


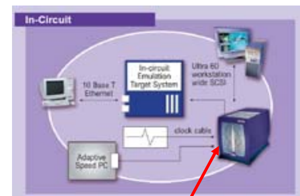
64-bit Adder Example

Virtex II – XC2V2000

How are FPGAs Used?

Logic Emulation





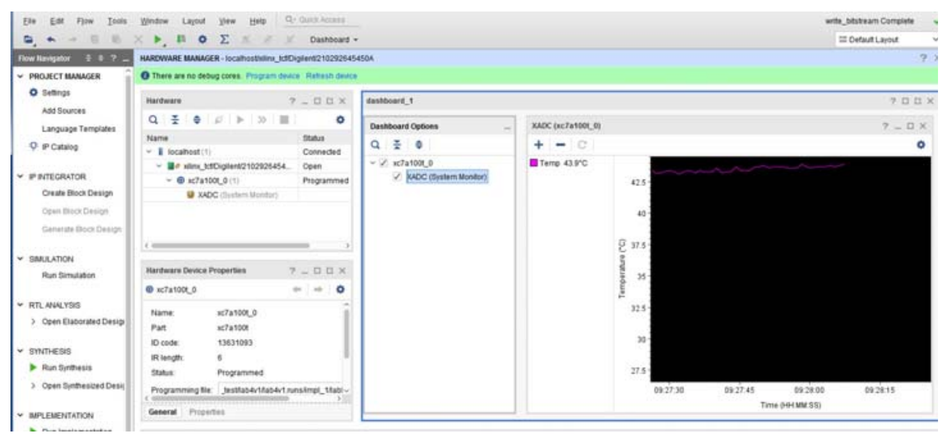
FPGA-based Emulator
(courtesy of IKOS)

- 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)

Dashboard



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

Test Bench

```
module sample_tb;

     // Inputs
     logic clk;
     logic data_in;

     // Outputs
     wire [7:0] data_out;

     // Instantiate the Unit Under Test (UUT)
     sample uut (
          .clk(clk),
          .data_in(data_in),
          .data_out(data_out)
     );

     always #5 clk = !clk;  // create a clock

     initial begin
          // Initialize Inputs
          clk = 0;
          data_in = 0;

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

          // Add stimulus here

     end
```

```
module sample(
   input clk,
   input data_in,
   output [7:0] data_out
   );

 // Verilog

endmodule
```

inputs must be initialized