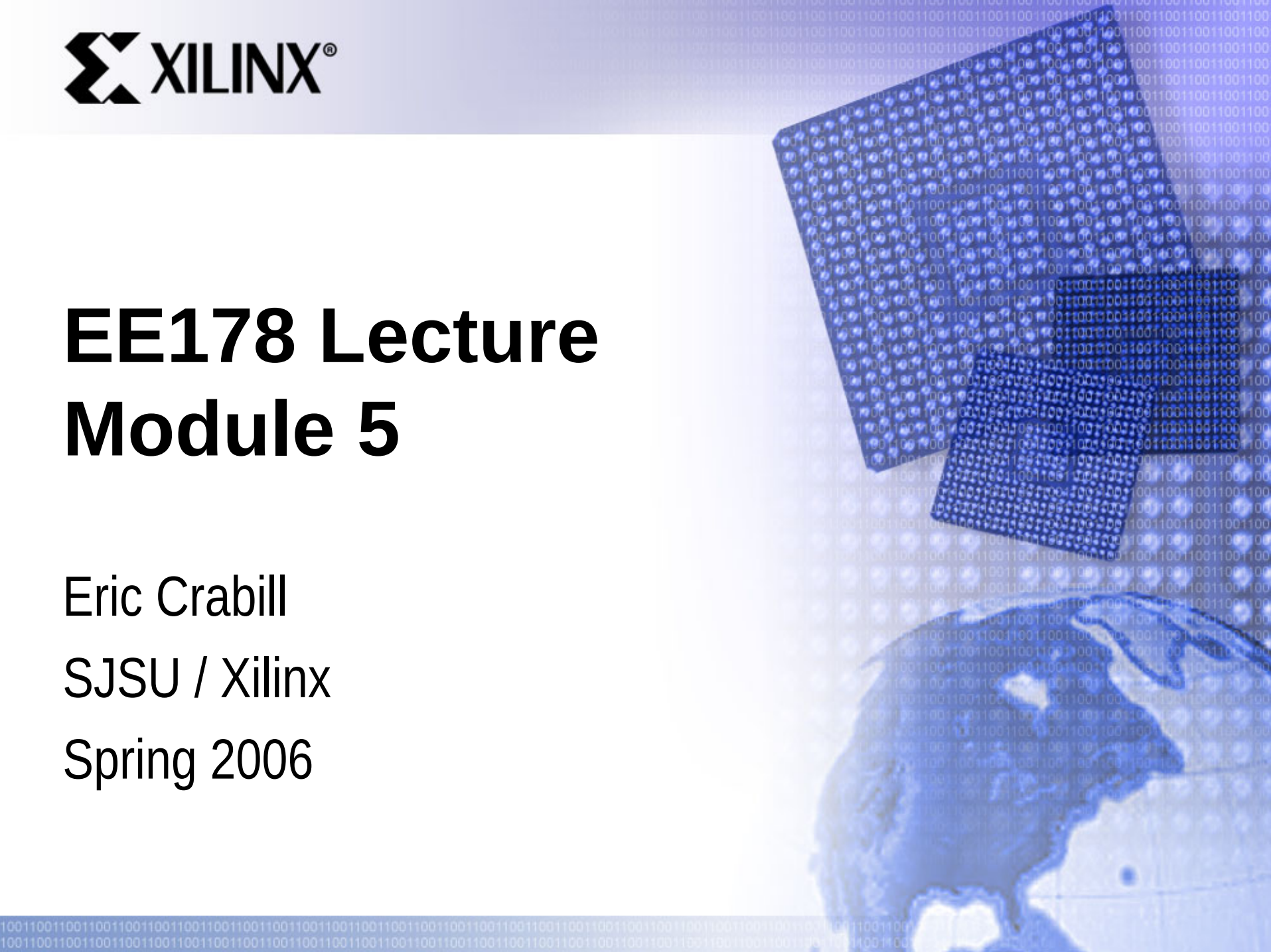


EE178 Lecture Module 5

Eric Crabill

SJSU / Xilinx

Spring 2006

Lecture #10 Agenda

- Implementation tradeoffs.
  - Design variables: throughput, latency, area.
- Pipelining for throughput.
- Retiming for throughput and latency.
- Interleaving for throughput and latency.
- Resource sharing for area.

Implementation Tradeoffs

- By now, you should realize there is more than one way to achieve a desired result.
- Your job is to implement a “near-optimal” hardware solution for your assigned task.
- It is important to understand the constraints.
  - Constraints determine what is optimal.
  - Throughput, Latency, Area?
  - Others, like Cost, Features, Time to Market?

Implementation Tradeoffs

- It is important to understand the assigned task.
  - The algorithm alone is only half the solution.
  - How you implement the algorithm is the other.
- Identify the “measure of optimal” and trade other dimensions in the design space for it.
  - Easier to do this on paper, at the design stage!
  - Harder, but possible, to optimize existing designs.
  - The “law of diminishing returns” applies.

Implementation Tradeoffs

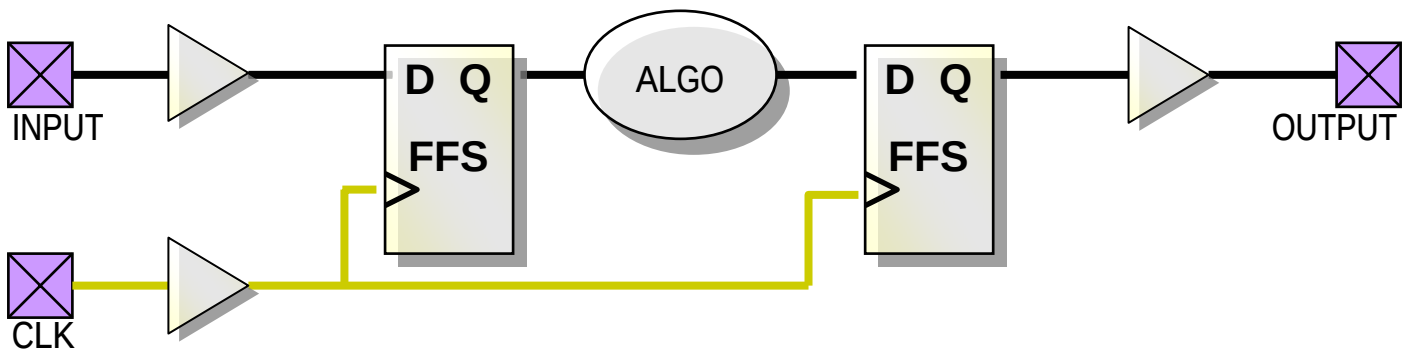
- Some definitions:
  - Throughput: The rate at which inputs are processed to create outputs; e.g. operations/second.
  - Latency: The delay from when an input is applied until the output associated with that input becomes available; e.g. seconds, or clock cycles.
  - Area: The resource usage required to implement a design; e.g. mm^2, or LUTs and FFs.

Pipelining

- Consider a circuit that does N operations per clock cycle at a frequency F. You could say the design has a throughput of N*F ops/sec.
- Pipelining is a technique to trade latency and area to improve throughput by increasing F.
- What sets the maximum frequency?
  - Think back to static timing analysis…
  - The maximum delay between flip flops!

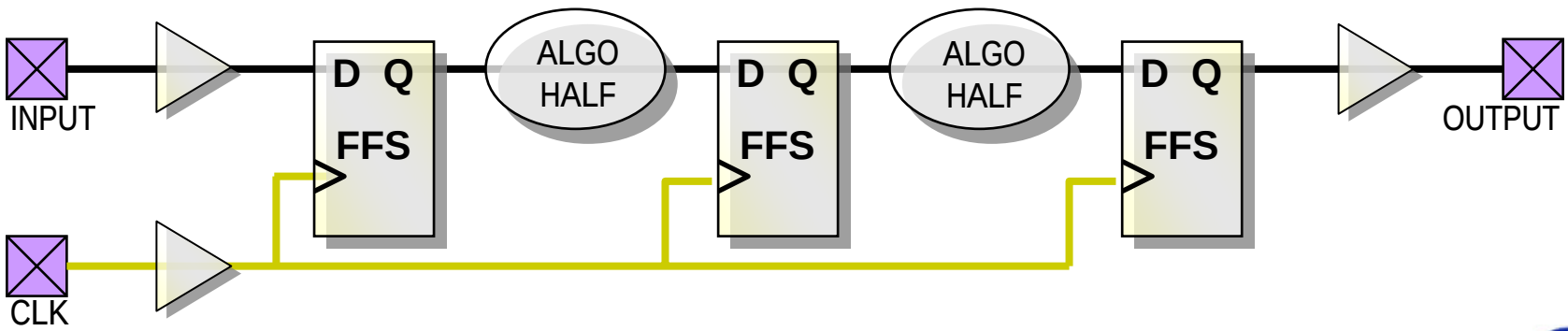
Pipelining

- What is causing the delay between flip flops?
  - Combinational logic to implement the algorithm.
  - Wires for distributing signals.
  - Inherent delays of the flip flops themselves.



Pipelining

- What if…
  - We partition the algorithm into two pieces, with each piece contributing 1/2 of the total delay?
  - We insert flip flops on all signals crossing the boundary between the two pieces?



Pipelining

- Results…
  - Frequency increases; the critical paths are halved.
  - Latency in cycles increases; not obvious yet but the latency in real (elapsed time) also increases.
  - Area increases; more flip flops required.
- What if you partitioned into M equal stages?
- Ideally, your new throughput is N*F*M ops/sec…

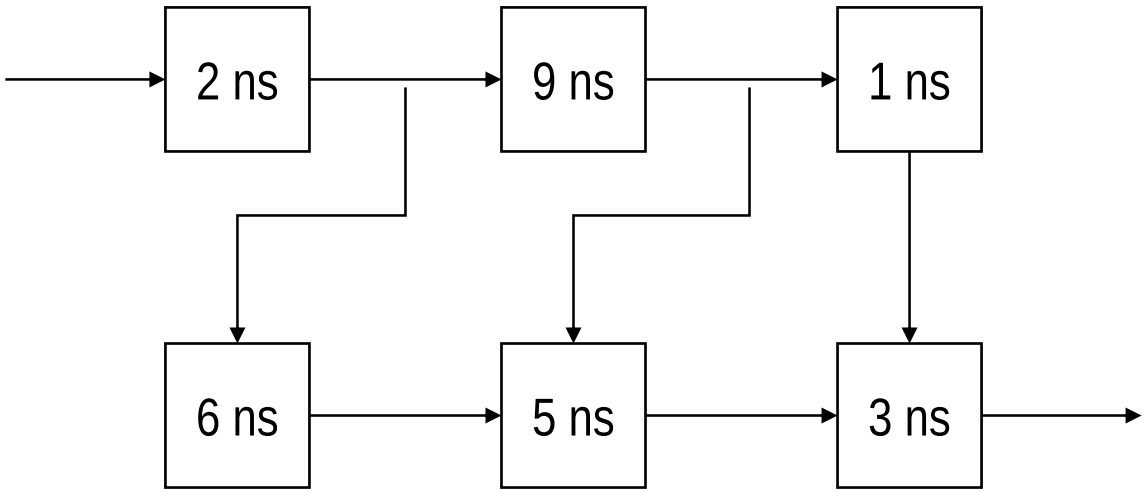
Pipelining Issues

- As M $\rightarrow \infty$, so does your circuit area and latency.
- Even if that were not true, you will reach a point where you can no longer sub-divide the design.
  - Can you partition a 2-input function in CMOS?
  - Can you partition a LUT in a Xilinx FPGA?
- Even if that were not true, the flip flop clock-to-out and input setup requirements do not change and set a limit on F, as M $\rightarrow \infty$ (diminishing returns).

Pipelining Issues

- Your algorithm may not easily partition into M equal stages -- in which case, the increase in F is set by the partition with the longest delay.
- Strategy: Focus your attention on placing the pipelining flip flops to isolate the slowest piece of the design and then maintain balance.

Simple Example



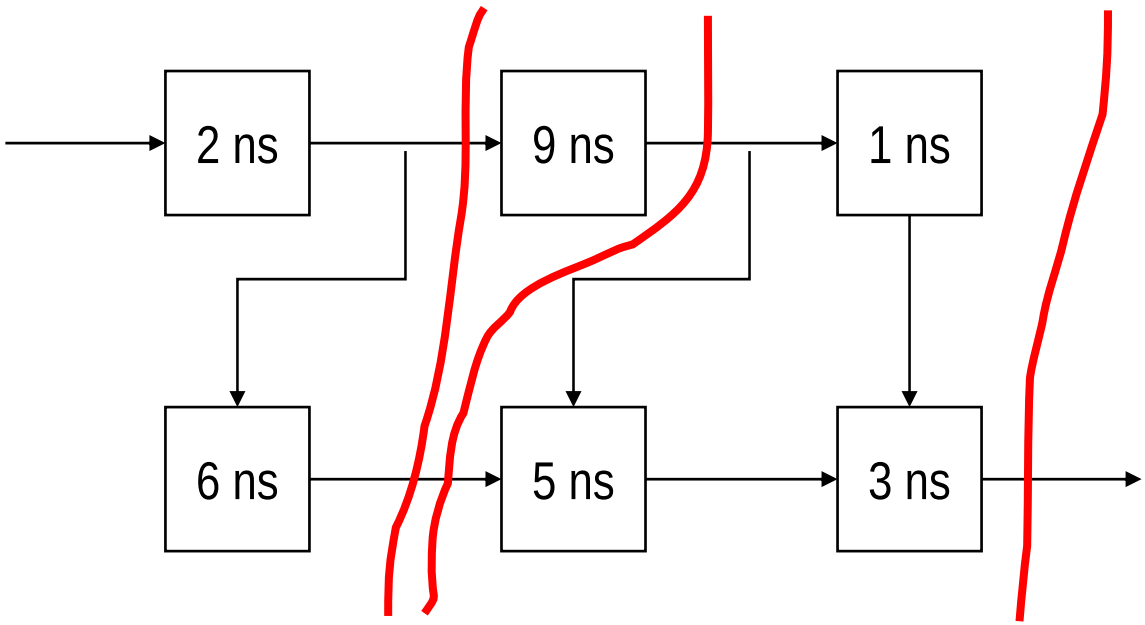
Individual functions are marked with their delay and cannot be further divided.

Draw lines to indicate where you would insert pipelining flip flops.

Optimize for throughput. Compare the old latency and the old throughput with your results.

Assume the pipelining flip flops are ideal (Tsu = Tout = Th = 0).

Simple Solution



Notice that signals (information) only pass one direction through the partitions -- forward!

Notice some signal paths need two flip flops to keep information flow synchronized.

Old Latency = 19 ns
Old Throughput = 1 op / 19 ns

New Latency = 27 ns (3 cycles)
New Throughput = 1 op / 9 ns

Pipelining Issues

- More problems can arise:
  - Pipelining data hazards occur when a computation depends on the result of a previous computation still in the pipeline.
  - Pipelining control hazards occur when a computation in the pipeline changes the selection of inputs to the design.
- If this excites you, read a computer architecture book for insight on how these can be handled.

Elaborate Example

```
// My manager told me I need to write a module that
// implements a sort.  The sort takes five, 16-bit
// numbers as input and outputs the same five numbers,
// but sorted.  He told me that the inputs are to be
// registered, and the outputs are to be registered.
// I remember from CS101 that there's something called
// a bubble sort, so I am going to implement this.
// Oh yeah, my manager also said something about the
// design needing to perform 50 megasorts per second,
// and that I'll have a 50 megahertz clock, and new
// data is provided at the inputs every clock cycle.

`timescale 1 ns / 1 ps

module bubble (clk, in1, in2, in3, in4, in5,
               out1, out2, out3, out4, out5);

  input clk;
  input [15:0] in1, in2, in3, in4, in5;
  output [15:0] out1, out2, out3, out4, out5;

  // First, I will register all the input data.
  // Only after it's registered can I use it.

  reg [15:0] dat1, dat2, dat3, dat4, dat5;

  always @(posedge clk)
  begin
    dat1 <= in1;
    dat2 <= in2;
    dat3 <= in3;
    dat4 <= in4;
    dat5 <= in5;
  end
```

Bubble.zip

Elaborate Example

```
// Here is the actual bubble sort.  I looked this
// up in my CS101 textbook.  I sure hope it works.

integer i, j;
reg [15:0] temp;
reg [15:0] array [1:5];

always @(dat1 or dat2 or dat3 or dat4 or dat5)
begin
  // put all the stuff into an array for
  // easy access using loop variables
  array[1] = dat1;
  array[2] = dat2;
  array[3] = dat3;
  array[4] = dat4;
  array[5] = dat5;
  // now perform the actual bubble sort
  for (i = 5; i > 0; i = i - 1)
  begin
    for (j = 1 ; j < i; j = j + 1)
    begin
      if (array[j] < array[j + 1])
      begin
        temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      end
    end
  end
end
```

Elaborate Example

```
  // Now I will register all the output data.

  reg [15:0] out1, out2, out3, out4, out5;

  always @(posedge clk)
  begin
    out1 <= array[1];
    out2 <= array[2];
    out3 <= array[3];
    out4 <= array[4];
    out5 <= array[5];
  end

endmodule
```

Elaborate Example

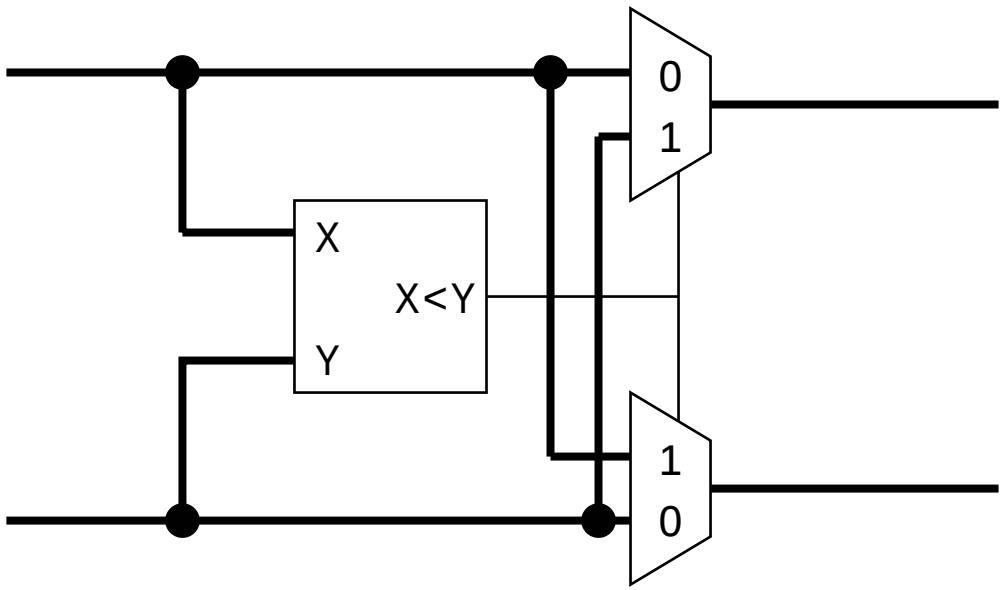
```
--------------------------------------------------------------------------------
  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
* NET "clk_BUFGP/IBUFG" PERIOD =  20 nS     | 20.000ns   | 49.378ns   | 52
  HIGH 50.000000 %                          |            |            |
--------------------------------------------------------------------------------

1 constraint not met.

Timing summary:
---------------

Timing errors: 216  Score: 3254922

Constraints cover 596759609344 paths, 0 nets, and 1987 connections (96.1% coverage)

Design statistics:
   Minimum period:  49.378ns (Maximum frequency:  20.252MHz)
```

Elaborate Example

- The results do not meet the requirements.
- Would you like to try and pipeline it? Not really!
  - I already shot myself in the foot by writing a Verilog hardware description as if it were a sequential C program.
    - Maybe not an issue had it worked…
    - Only rely on luck if you have no talent.
  - Unroll the loops to understand the algorithm.
  - Partition and then pipeline it at each loop iteration.

Elaborate Example

- What operation is taking place in the inner loop?



Elaborate Example

- The pipelined result has a very high latency.
  - Sequential nature of the algorithm.
  - Not an explicit design constraint.
- Hardware is inherently parallel; is there a better algorithm to solve this problem?
  - Yes, it is called the odd-even transposition sort.
    - I researched it on the internet using Google.
    - There is a large field called parallel computing and I can steal algorithms from it for hardware.

Elaborate Example

```
// My manager told me I need to write a module that
// implements a sort.  The sort takes five, 16-bit
// numbers as input and outputs the same five numbers,
// but sorted.  He told me that the inputs are to be
// registered, and the outputs are to be registered.
// I remember from CS101 that there's something called
// a bubble sort.  I was going to implement this, but
// then remembered from EE178 that Verilog isn't C.
// Since I'm being paid to be a logic designer, I
// did some research on sorting algorithms and found
// one called the odd-even transposition sort, which
// is basically a parallel version of the bubble sort.
// I unrolled the loop and pipelined it to make it
// reasonably fast, because my manager said that the
// design needs to perform 50 megasorts per second,
// and that I'll have a 50 megahertz clock, and new
// data is provided at the inputs every clock cycle.

`timescale 1 ns / 1 ps

module oetsort (clk, in1, in2, in3, in4, in5,
                out1, out2, out3, out4, out5);

  input clk;
  input [15:0] in1, in2, in3, in4, in5;
  output [15:0] out1, out2, out3, out4, out5;
  reg [15:0] dat1s0, dat2s0, dat3s0, dat4s0, dat5s0;
  reg [15:0] dat1s1, dat2s1, dat3s1, dat4s1, dat5s1;
  reg [15:0] dat1s2, dat2s2, dat3s2, dat4s2, dat5s2;
  reg [15:0] dat1s3, dat2s3, dat3s3, dat4s3, dat5s3;
  reg [15:0] dat1s4, dat2s4, dat3s4, dat4s4, dat5s4;
  reg [15:0] dat1s5, dat2s5, dat3s5, dat4s5, dat5s5;
```

Oetsort.zip

Elaborate Example

```
always @(posedge clk)
begin
  // This implements the input registers.
  dat1s0 <= in1;
  dat2s0 <= in2;
  dat3s0 <= in3;
  dat4s0 <= in4;
  dat5s0 <= in5;
  // This implements the first pipeline stage.
  dat1s1 <= (dat1s0 < dat2s0) ? dat2s0 : dat1s0;
  dat2s1 <= (dat1s0 < dat2s0) ? dat1s0 : dat2s0;
  dat3s1 <= (dat3s0 < dat4s0) ? dat4s0 : dat3s0;
  dat4s1 <= (dat3s0 < dat4s0) ? dat3s0 : dat4s0;
  dat5s1 <= dat5s0;
  // This implements the second pipeline stage.
  dat1s2 <= dat1s1;
  dat2s2 <= (dat2s1 < dat3s1) ? dat3s1 : dat2s1;
  dat3s2 <= (dat2s1 < dat3s1) ? dat2s1 : dat3s1;
  dat4s2 <= (dat4s1 < dat5s1) ? dat5s1 : dat4s1;
  dat5s2 <= (dat4s1 < dat5s1) ? dat4s1 : dat5s1;
  // This implements the third pipeline stage.
  dat1s3 <= (dat1s2 < dat2s2) ? dat2s2 : dat1s2;
  dat2s3 <= (dat1s2 < dat2s2) ? dat1s2 : dat2s2;
  dat3s3 <= (dat3s2 < dat4s2) ? dat4s2 : dat3s2;
  dat4s3 <= (dat3s2 < dat4s2) ? dat3s2 : dat4s2;
  dat5s3 <= dat5s2;
  // This implements the fourth pipeline stage.
  dat1s4 <= dat1s3;
  dat2s4 <= (dat2s3 < dat3s3) ? dat3s3 : dat2s3;
  dat3s4 <= (dat2s3 < dat3s3) ? dat2s3 : dat3s3;
  dat4s4 <= (dat4s3 < dat5s3) ? dat5s3 : dat4s3;
  dat5s4 <= (dat4s3 < dat5s3) ? dat4s3 : dat5s3;
  // This implements the fifth pipeline stage.
  dat1s5 <= (dat1s4 < dat2s4) ? dat2s4 : dat1s4;
  dat2s5 <= (dat1s4 < dat2s4) ? dat1s4 : dat2s4;
  dat3s5 <= (dat3s4 < dat4s4) ? dat4s4 : dat3s4;
  dat4s5 <= (dat3s4 < dat4s4) ? dat3s4 : dat4s4;
  dat5s5 <= dat5s4;
end
```

Elaborate Example

```
  // The output data is already registered, so all
  // I have to do is drive it out of the module.

  assign out1 = dat1s5;
  assign out2 = dat2s5;
  assign out3 = dat3s5;
  assign out4 = dat4s5;
  assign out5 = dat5s5;

endmodule
```

Elaborate Example

```
--------------------------------------------------------------------------------
  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  NET "clk_BUFGP/IBUFG" PERIOD =  20 nS     | 20.000ns   | 9.531ns    | 9
  HIGH 50.000000 %                          |            |            |
--------------------------------------------------------------------------------

All constraints were met.

Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 10320 paths, 0 nets, and 1747 connections (95.6% coverage)

Design statistics:
   Minimum period:   9.531ns (Maximum frequency: 104.921MHz)
```

Elaborate Example

- The results exceed the requirements by far.
- If desired, I could save area and latency by removing every other set of pipeline registers… and still meet the requirements!
- If you know you are going to have to optimize your hardware implementation, you can save some pain by doing research up front.

Pipelining Conclusion

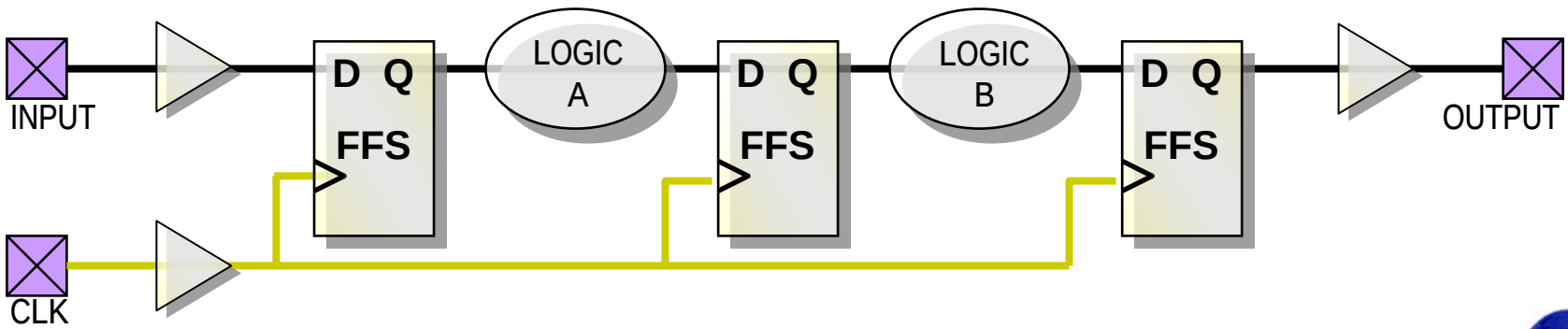
- Pipelining is not a panacea.
- Pipelining is a tool you can use to trade area and latency for throughput.
- In complex designs, you need to be careful!
- The decision to use this tool to optimize your design should be based on your design constraints.

Retiming

- Consider a circuit that does N operations per clock cycle at a frequency F. You could say the design has a throughput of N*F ops/sec.
- Retiming is a technique to improve throughput and latency by increasing F, with the possibility of an area change.

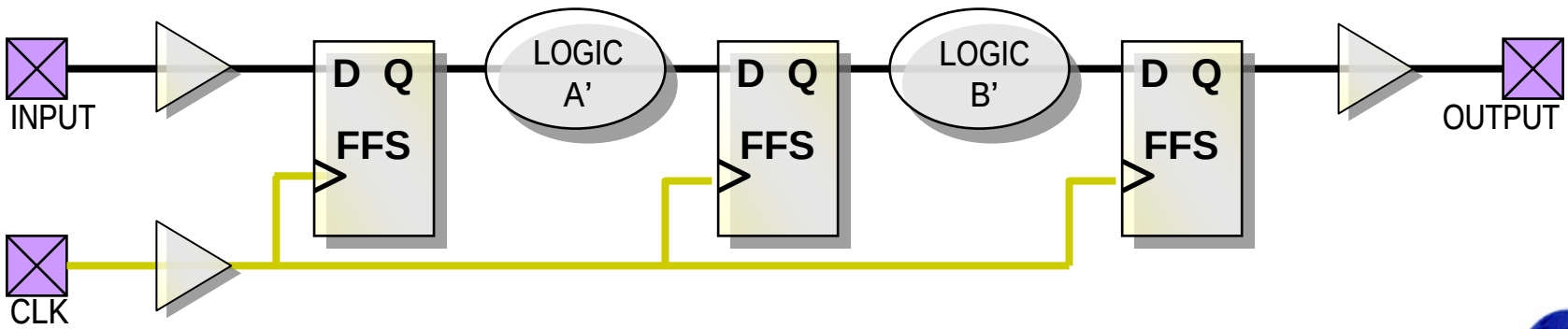
Retiming

- What sets the maximum frequency?
  - Combinational logic to implement the algorithm.
  - Wires for distributing signals.
  - Inherent delays of the flip flops themselves.



Retiming

- We push pieces of the logic forward and/or backward through existing registers in an attempt to balance delay between registers.
- Alternately, pick up existing registers and move them backward and/or forward through logic...



Retiming

- Frequency increases; unless it is already perfectly balanced, critical paths are reduced.
- Latency in cycles is constant, but latency in real (elapsed time) decreases because the cycle time decreases.
- Area may increase or decrease depending on the paths that are retimed.

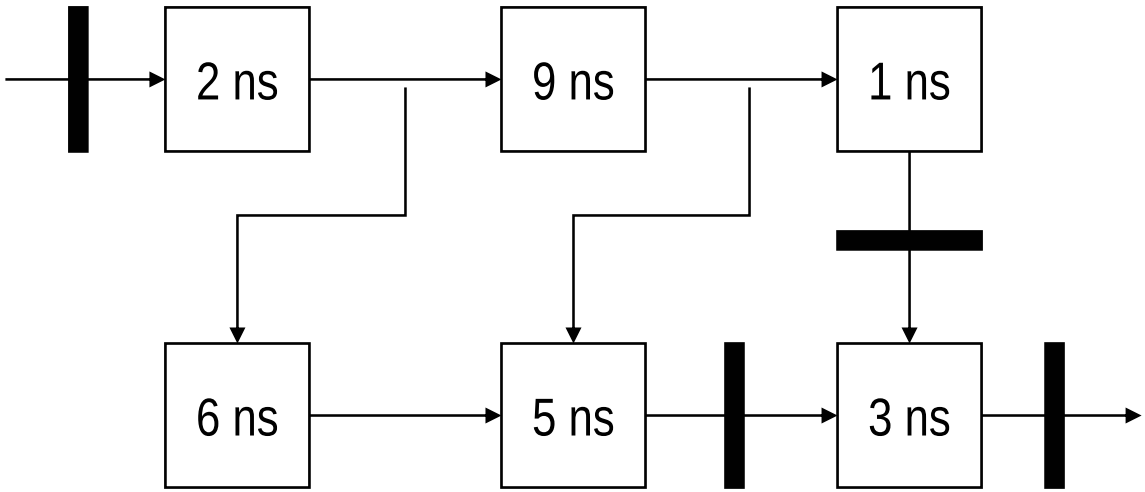
Retiming Issues

- Logic granularity limits what you can move.
  - Can you move half of a 2-input function in CMOS?
  - Can you move half of a LUT in a Xilinx FPGA?
- The resulting “state elements” of your design change, complicating debugging.

Retiming Issues

- Without changing the cycle latency, the extent of the changes you can make are limited.
  - Applied to a blobular/random design, frequency improvement may not be impressive.
  - Applied to a pipelined design with unbalanced stages may yield significant improvement.
- Difficult to do manually, best left to the synthesis tool as an after-the-fact optimization.

Simple Example 1



Individual functions are marked with their delay and cannot be further divided.

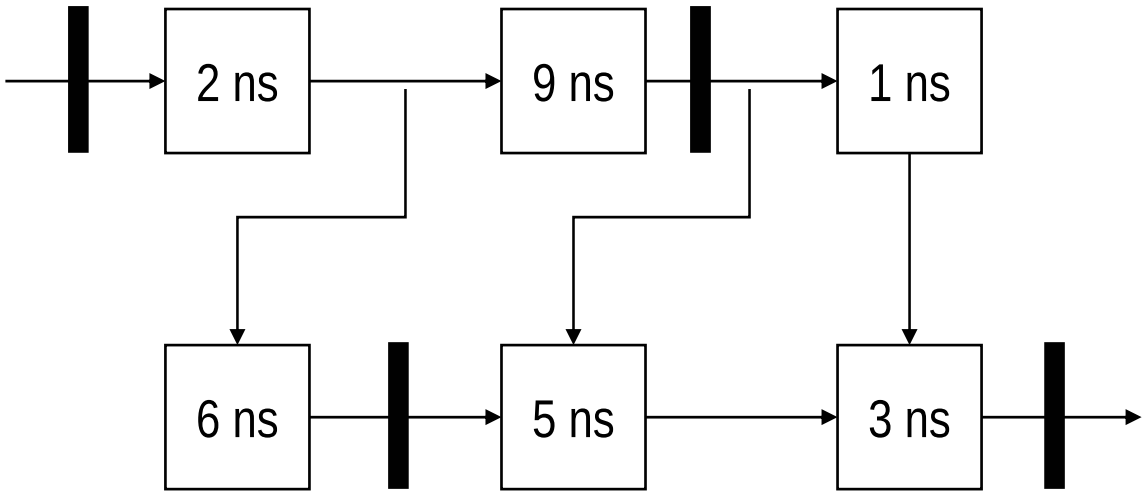
Solid bars represent flip flops. Assume the design flip flops are ideal (Tsu = Tout = Th = 0).

This is an unbalanced, pipelined design.

Retime the circuit without moving the input or output flip flops.

Compare the old latency and the old throughput with your results.

Simple Solution 1

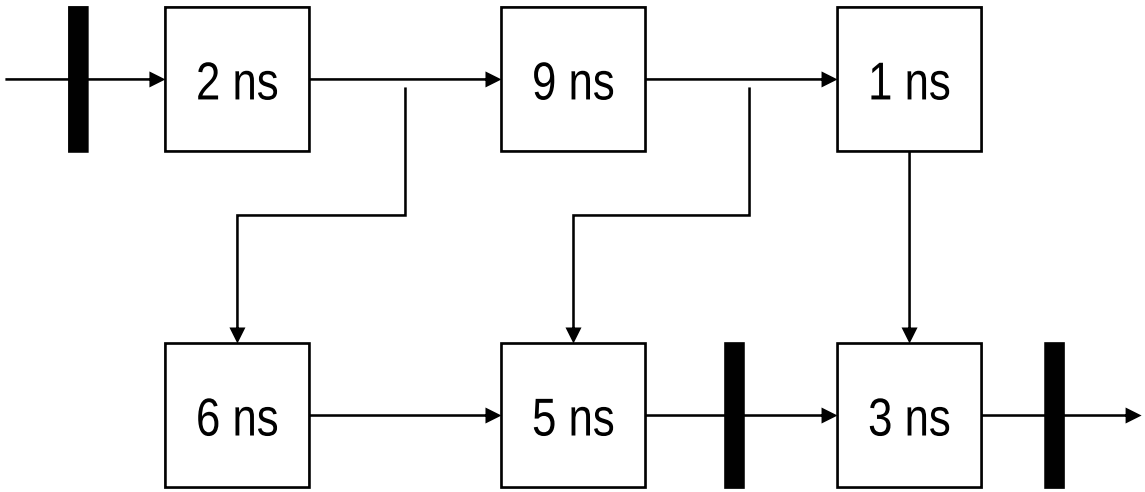


Old Latency = 32 ns (2 cycles)
Old Throughput = 1 op / 16 ns

New Latency = 22 ns (2 cycles)
New Throughput = 1 op / 11 ns

How about area cost?

Simple Example 2



Individual functions are marked with their delay and cannot be further divided.

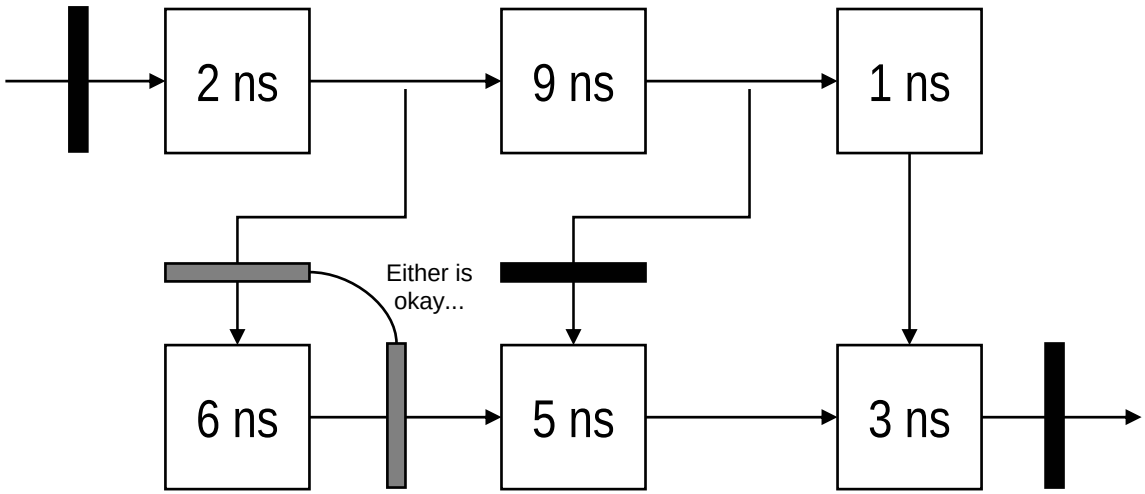
Solid bars represent flip flops. Assume the design flip flops are ideal (Tsu = Tout = Th = 0).

This is a blobular/random design.

Retime the circuit without moving the input or output flip flops.

Compare the old latency and the old throughput with your results.

Simple Solution 2



Old Latency = 32 ns (2 cycles)
Old Throughput = 1 op / 16 ns

New Latency = 30 ns (2 cycles)
New Throughput = 1 op / 15 ns

How about area cost?
Which gray flops are better?

Retiming Conclusion

- Retiming is not a magic wand.
- Retiming is a tool you can use to trade area for latency and throughput.
- May gain you very little, or nothing!
- The decision to use this tool to optimize your design should be based on your design constraints.

Interleaving

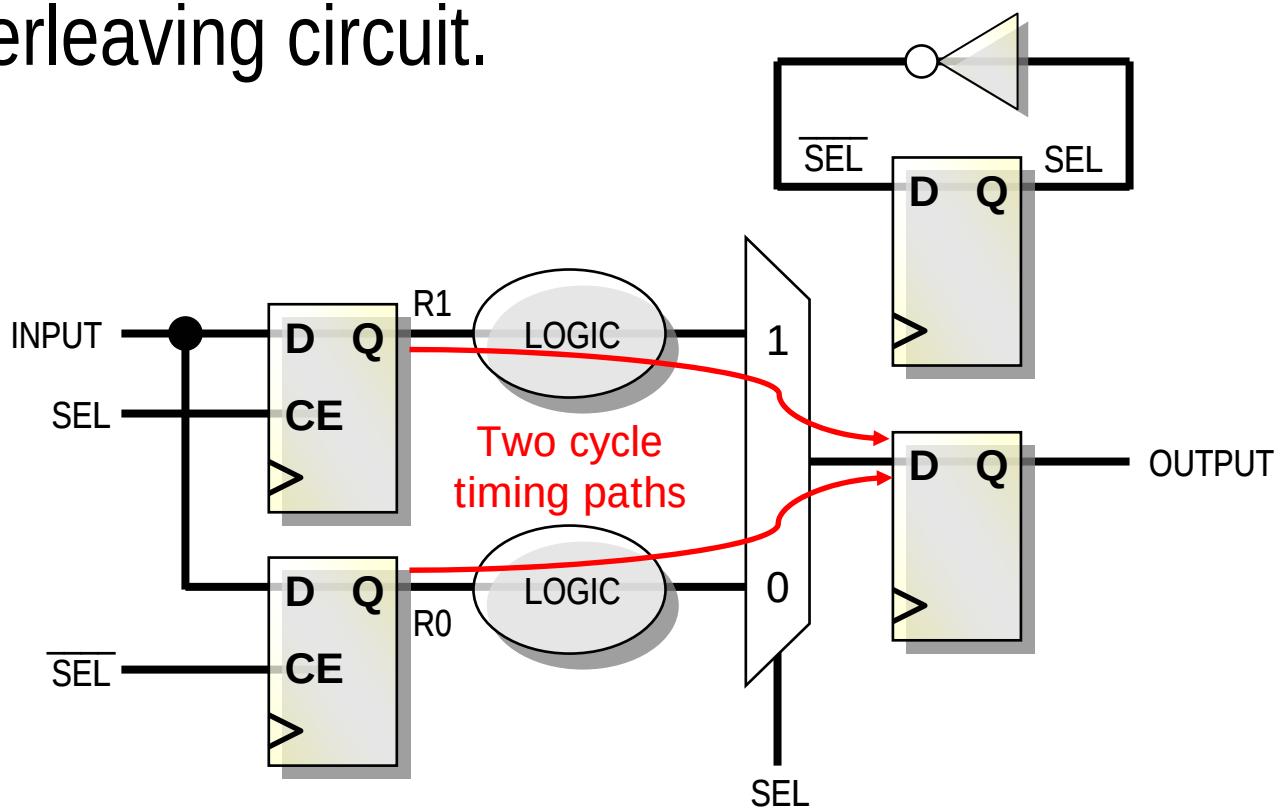
- Consider a circuit that does N operations per clock cycle at a frequency F. You could say the design has a throughput of N*F ops/sec.
- Interleaving is a technique to trade latency and area to improve throughput by increasing F.
- Interleaving is especially useful to break pipeline bottlenecks in a pipelined design.

Interleaving

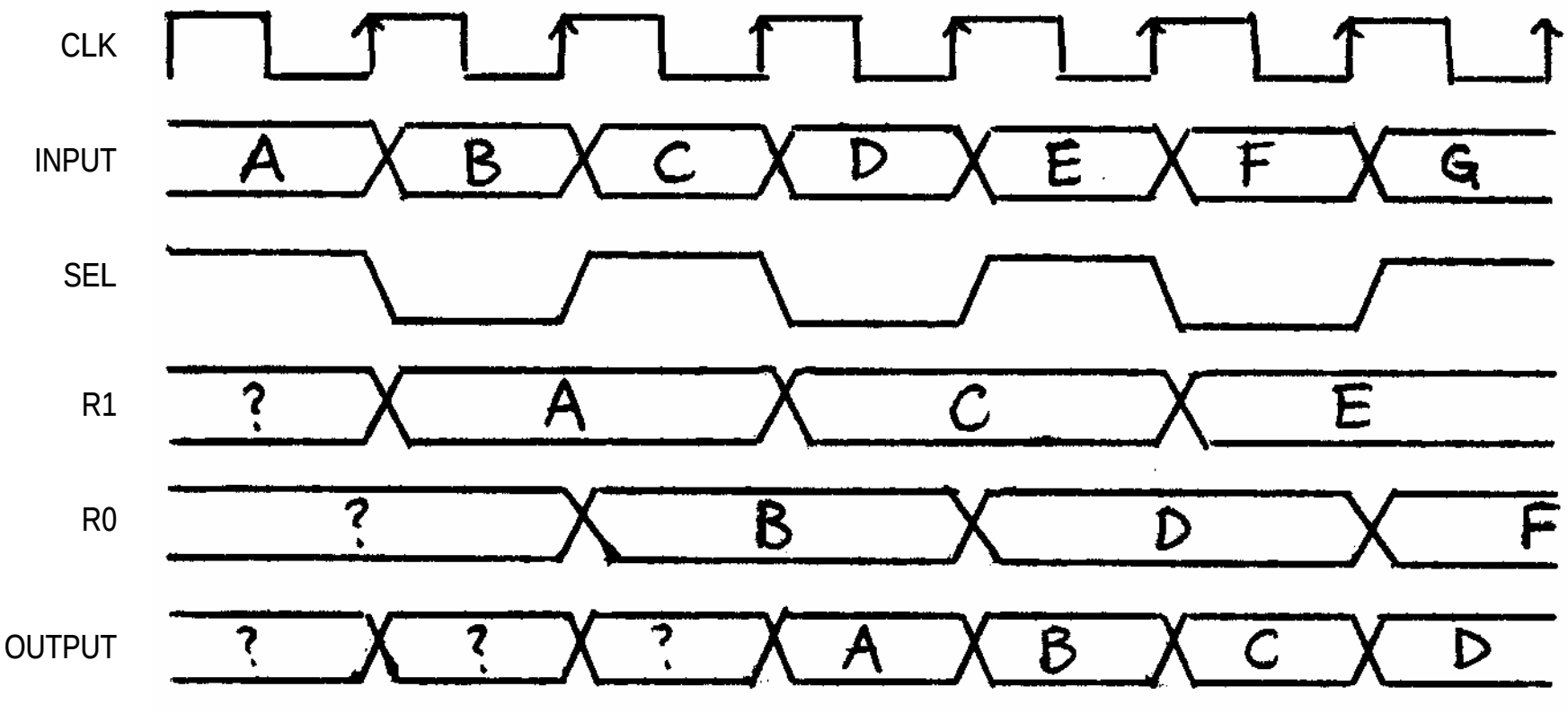
- What sets the maximum frequency?
  - Combinational logic to implement the algorithm.
  - Wires for distributing signals.
  - Inherent delays of the flip flops themselves.
- What if you could replicate the logic N times, and provide N times as many clock cycles for each instance to perform its function?
- Arriving data is interleaved between blocks...

Interleaving

- Example of two-way interleaving circuit.

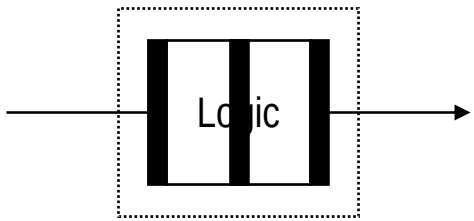


Interleaving

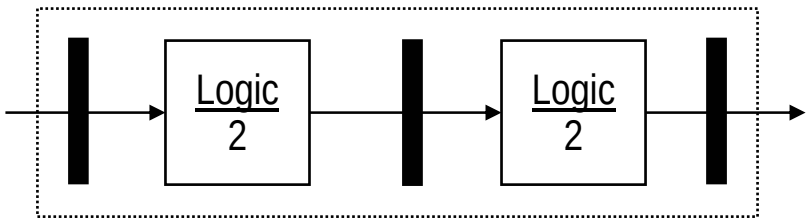


Interleaving

- The two-way interleaving circuit shown on the previous page may be represented by:



- This is functionally equivalent to the following:



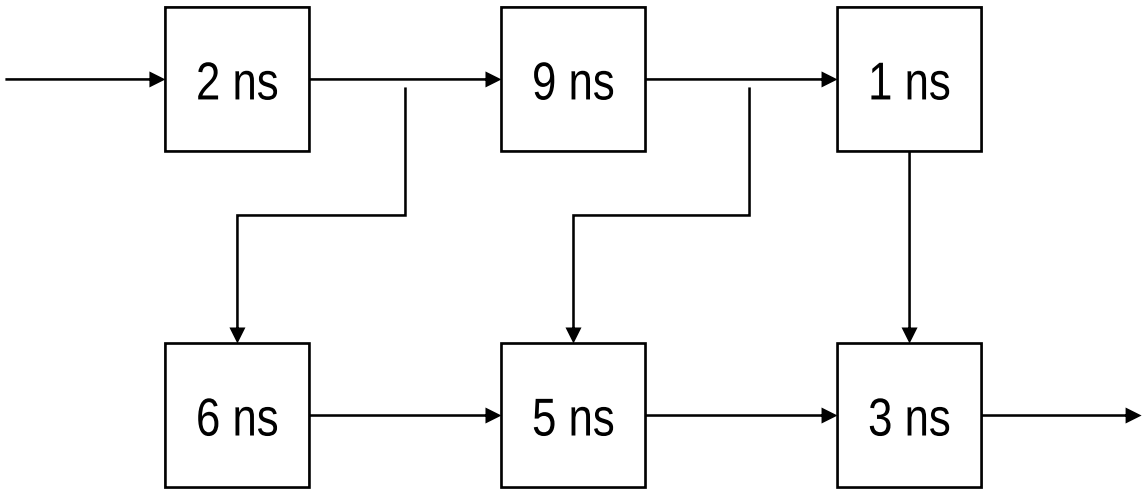
Interleaving

- Frequency increases; critical paths are reduced.
- Latency in cycles and in real time increases.
- Area increases; more flip flops and duplicate functional units are required.

Interleaving Issues

- Multi-cycle signal paths through replicated logic complicate static timing analysis.
  - A simple “period” constraint is no longer sufficient.
  - You need to pay careful attention to constraints.
- M-way interleaving has similar limits to those for an M-stage pipeline as $M \rightarrow \infty$.

Simple Example



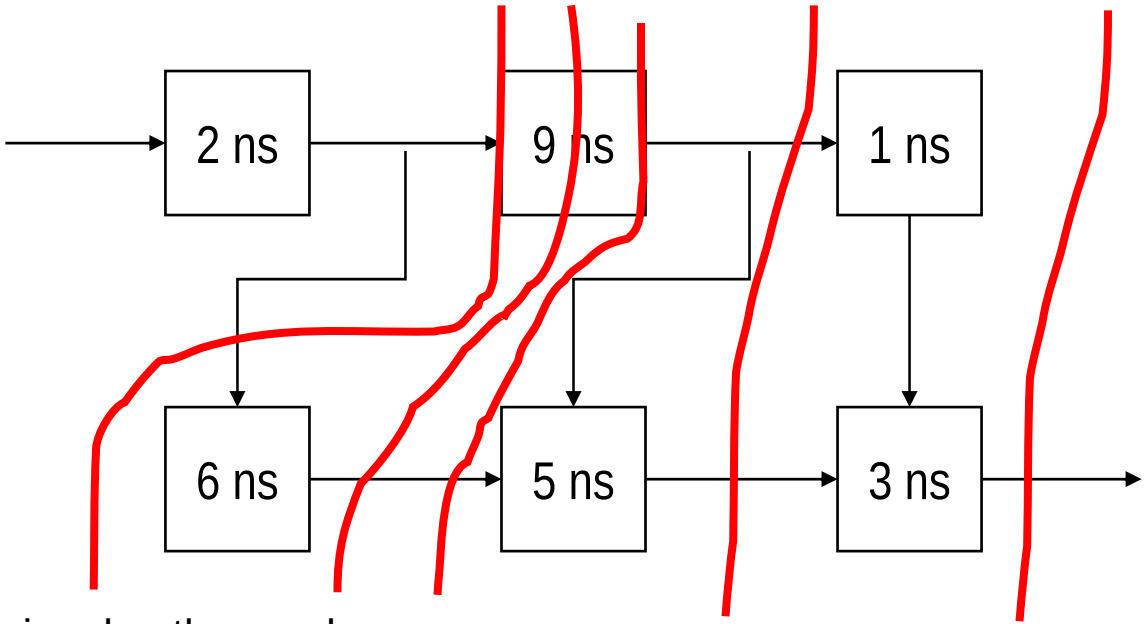
Individual functions are marked with their delay. You may apply two-way interleaving on a single component. Other components may not be further divided.

Draw lines to indicate where you would insert pipelining flip flops.

Optimize for throughput. Compare the old latency and the old throughput with your results.

Assume the pipelining flip flops are ideal (Tsu = Tout = Th = 0).

Simple Solution



Notice some signal paths need two flip flops to keep information flow synchronized.

What kind of results would you achieve if you could perform an additional two-way interleave?

Old Latency = 19 ns
Old Throughput = 1 op / 19 ns

New Latency = 30 ns (5 cycles)
New Throughput = 1 op / 6 ns

Compare this to the original pipelined results...

Interleaving Conclusion

- Interleaving is not a silver bullet.
- Interleaving is a tool you can use to trade area for latency and throughput.
- Requires attention to timing constraints.
- The decision to use this tool to optimize your design should be based on your design constraints.

Resource Sharing

- Resource sharing is a technique to trade frequency and latency for area.
- Resource sharing is when a single hardware resource (typically area-expensive) is used to implement several operations in the design.
- What kind of operations can share hardware?
  - Addition, Subtraction, Comparisons.
  - Multiplication, Division.

Resource Sharing Example

- Assume a 16x16 multiplier uses 100 area units.
- Assume a 4:1 multiplexer uses 1 area unit.
- Consider the following block of code:

```
wire [15:0] a, b, c, d, e, f, g, h, i, j;
wire [31:0] output1, output2;
wire  [1:0] op;

always @(op or a or b or c or d or e or f or g or h or i or j)
begin
  output1 = a * b;
  case (op)
    2'b00: output2 = c * d;
    2'b01: output2 = e * f;
    2'b10: output2 = g * h;
    2'b11: output2 = i * j;
  endcase
end
```

Resource Sharing Example

- A naïve implementation uses:
  - A multiplier to generate output1.
  - Four multipliers, followed by a 32-bit wide 4:1 multiplexer to generate output2.
  - Area cost is 532 area units.
- Draw a schematic of the circuit.

Resource Sharing Example

- A better implementation uses:
  - A multiplier to generate output 1.
  - Two 16-bit wide 4:1 multiplexers followed by a multiplier to generate output2.
  - Area cost is 232 area units.
- Draw a schematic of the circuit.

Resource Sharing

- Frequency and latency may increase if additional delay is added to the critical path.
- Area decrease as a result of sharing the area-expensive hardware resources.
- In order for resources to be shared, they must not need to operate simultaneously in the original design.

Resource Sharing Conclusion

- Resource sharing is not a silver bullet.
- Resource sharing is a tool you can use to trade latency and throughput for area.
- The decision to use this tool to optimize your design should be based on your design constraints.