AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

PETER HØGH MIKKELSEN
3. MARCH, 2015

CLOCK DOMAINS AND TIMING

AGENDA

- Designing for optimum speed / area
- Optimizing for speed:
  - Throughput, Latency, Timing
- Optimizing for area:
  - Memory Blocks, Reset

OPTIMIZING FPGA DESIGNS

```
// C-Example for Pwr3

XPower = 1;
For(i=0; i < 3 ; i++)
  XPower = X * Xpower;
```

- How do we optimize this in terms of:
  - Speed (Throughput/Latency)?
  - Used silicium (area)?

OPTIMIZING SPEED

- High Throughput
  - Data transmission and storage
  - DSP calculations
  - Bandwidth is important, delay less.

- Low Latency
  - 2-way streaming, skype, cellular phone, video conferencing
  - Delay is crucial, bandwidth less important
  - Latency = Propagation Delay

WHEN/HOW TO OPTIMIZE

- When must we optimize for speed?
  - If throughput is too low
  - If the system Fmax is too low
  - If latency is too high

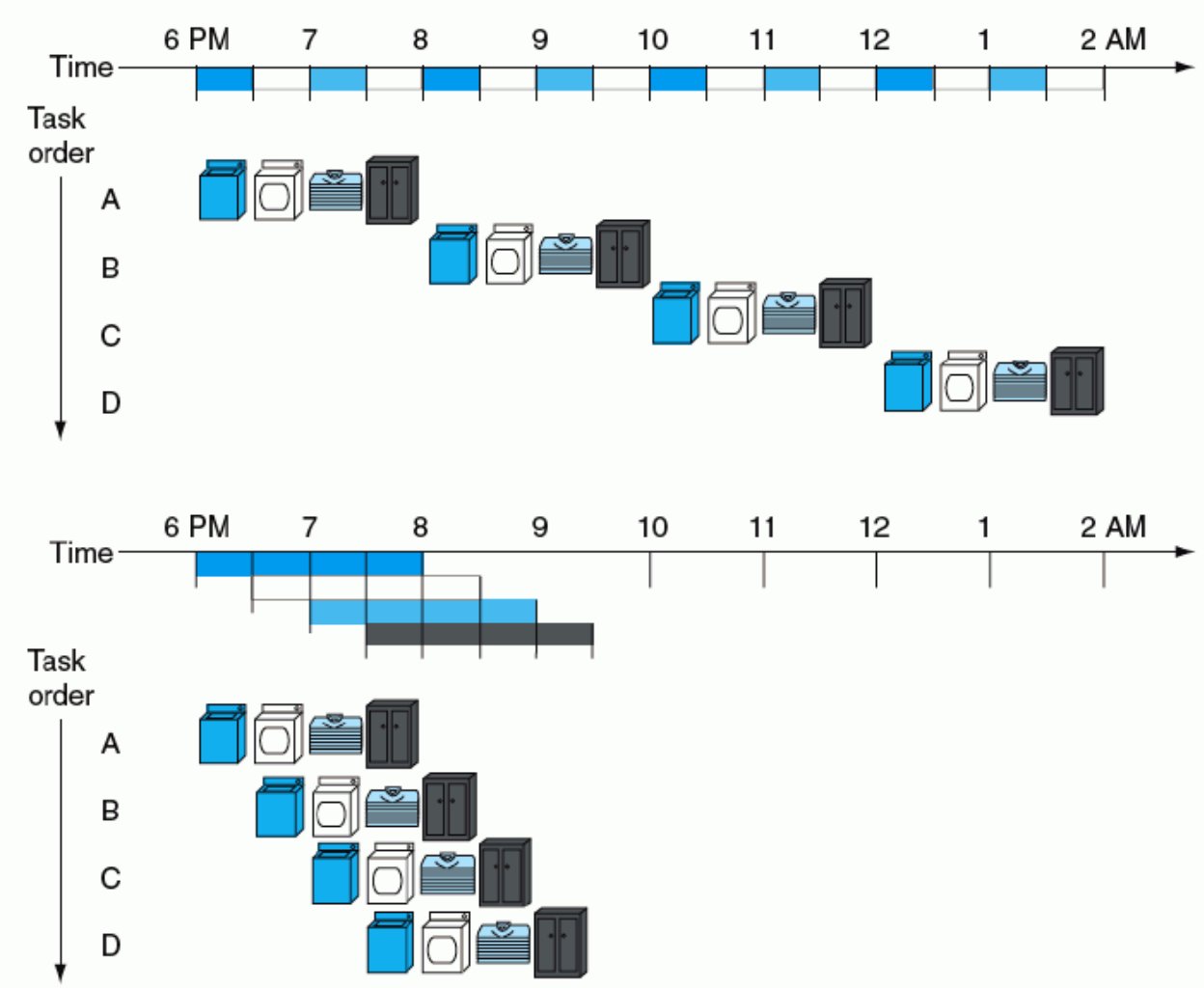
- What determines the solution?
  - Increase in area allowed?
  - Increase in latency allowed?

ADD REGISTER LAYERS

- Problem: Too Low Throughput
  - Increased Latency is OK
  - Increased Area is OK

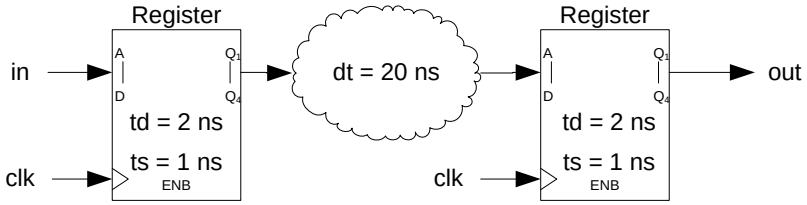
- Break the critical timing path by introducing an extra register in the path (Introduce pipelining)

PIPELINING


6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D
6 PM
7
8
9
10
11
12
1
2 AM
Time
Task order
A
B
C
D

ADD REGISTERS - EXAMPLE

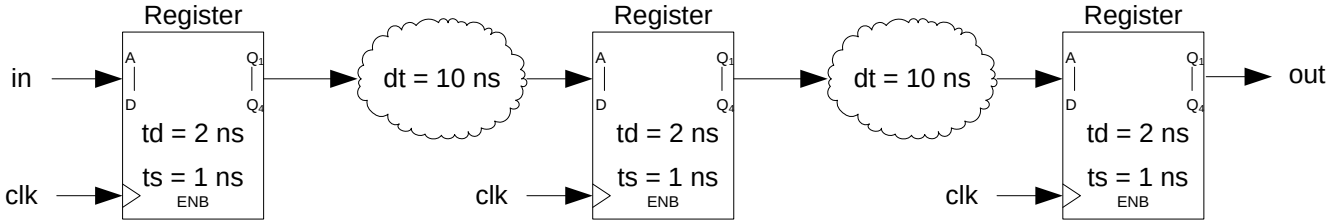
Original Pipeline



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(2+20+1 ns) = 43MHz

Latency = 2 * 1/fclk + 2 ns = 2 * 1/43MHz + 2 ns = 48 ns

Additional Pipeline Stage Added



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(2+10+1 ns) = 77MHz

Latency = 3 * 1/fclk + 2 ns = 3 * 1/77MHz + 2 ns = 41 ns

PIPELINING – EXAMPLE (1)

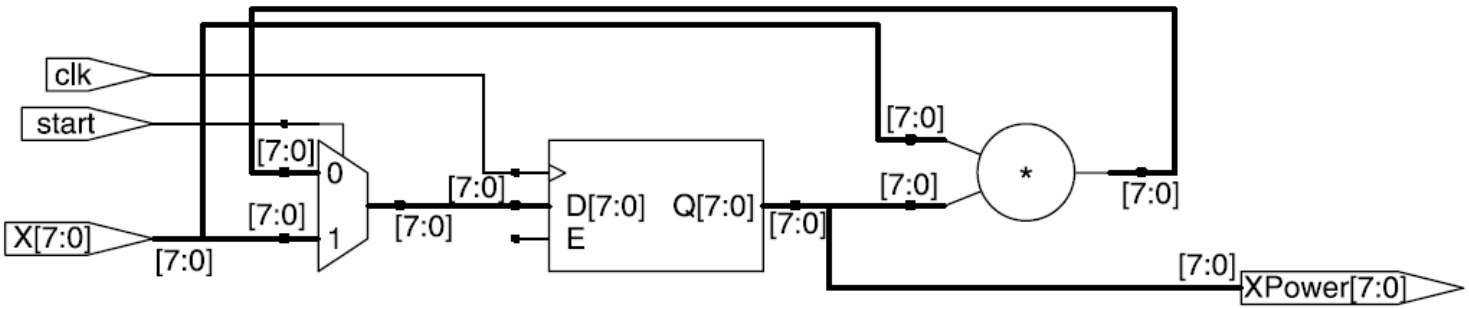


Figure 1.1 Iterative implementation.

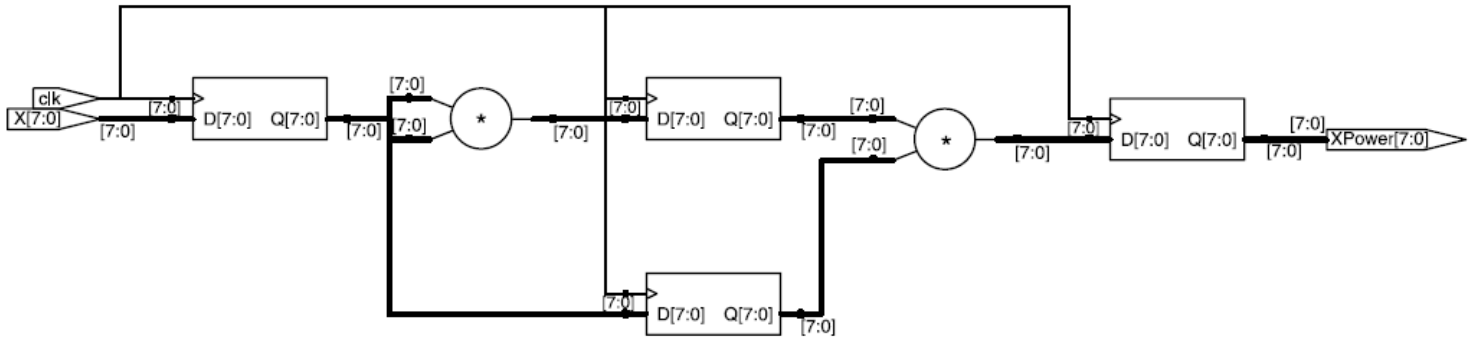


Figure 1.2 Pipelined implementation.

PIPELINE – EXAMPLE (2)

```
entity power3 is
  port(
    Xpower : in std_logic_vector(7 downto 0),
    clk    : in std_logic,
    X      : in std_logic_vector(7 downto 0));
end;
architecture rtl of power3 is
  signal XPower1, XPower2 : std_logic_vector(7 downto 0);
  signal X1, X2 : std_logic_vector(7 downto 0);

  proces(clk)
    if rising_edge(clk) then
      X1 <= X;
      XPower1 <= X;
      X2 <= X1;
      XPower2 <= XPower1 * X1;
      XPower <= XPower2 * X2;
  end process;
end rtl;
```

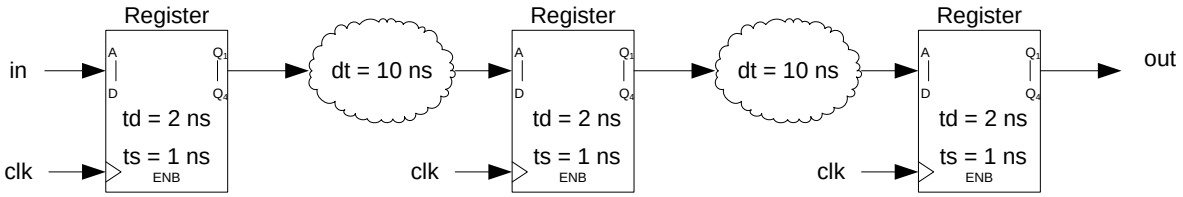
PARALLELIZE STRUCTURE

- Problem: Too Low Throughput/High Latency
  - Increased Latency is NOT OK.
  - Increased Area is OK

- Split the design into several parallel tasks (more processes)

PARALLELIZE - EXAMPLE

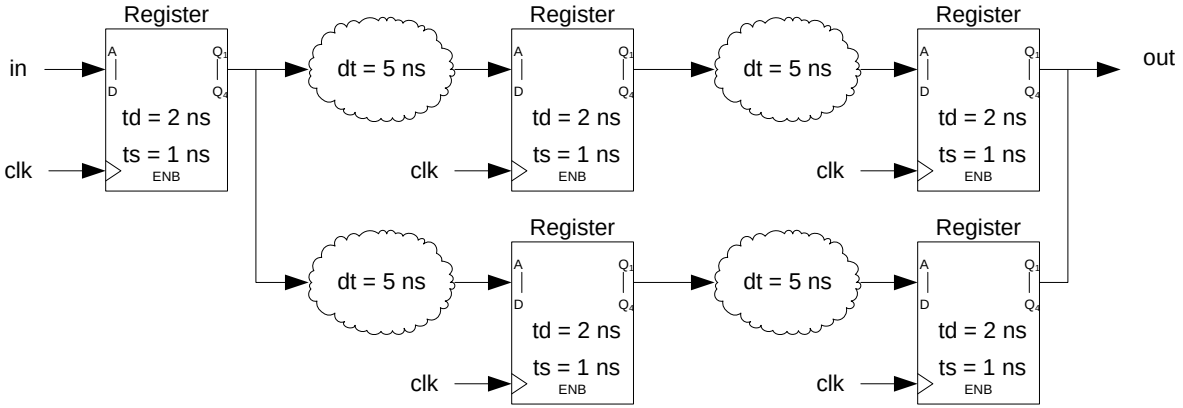
Original Pipeline



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(2+10+1 ns) = 77 MHz

Latency = 3 * 1/fclk + 2ns = 3 * 1/77MHz + 2 ns = 41 ns

Parallelized Structure



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(2+5+1 ns) = 125 MHz

Latency = 3 * 1/fclk + 2ns = 3 * 1/125MHz + 2 ns = 26 ns

FLATTEN STRUCTURE

- Problem: Too Low Throughput/High Latency
  - Increased Latency is NOT OK
  - Increased Area is NOT OK

- Remove un-intentional priorities in the design (typically ”if-else” statements, replace these by ”case” statements)

FLATTEN STRUCTURE – EXAMPLE (1)

Prioritized

```
if input = "0001" then
    OutA <= '1';
elsif input = "0010" then
    OutB <= '1';
elsif input = "0100" then
    OutC <= '1';
else
    OutF <= '1';
end if;
```

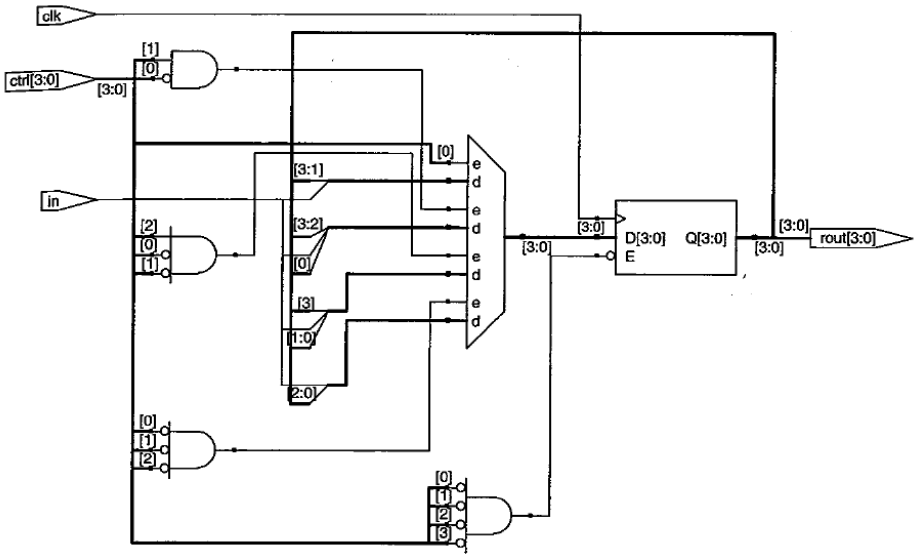
Not Prioritized

```
case input is
    when "0001" =>
      OutA <= '1';
    when "0010" =>
      OutB <= '1';
    when "0100" =>
      OutC <= '1';
    when others =>
      OutF <= '1';
end case;
```

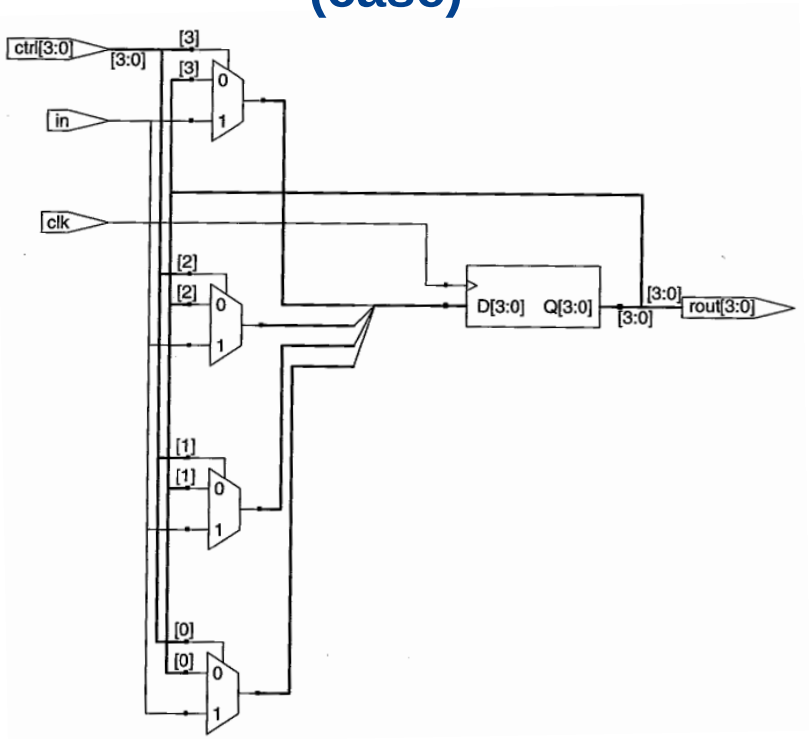
› Priority adds combinatorial logic to the signal delay path
› ”Case” sentense is implemented as a simple mux

FLATTEN STRUCTURE – EXAMPLE (2)

Prioritized (if)



Not Prioritized (case)

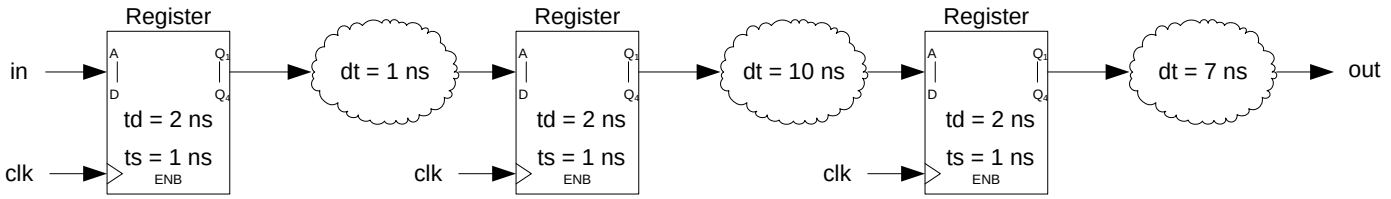


REGISTER BALANCING

- Problem: Too Low Throughput/High Latency
  - Increased Latency is NOT OK
  - Increased Area is NOT OK
- The Fmax of a pipelined design is constrained by the longest combinatorial delay in the pipeline.
- By distributing the the combinatorial logic evenly into each step of the pipeline, Fmax is maximized

REGISTER BALANCING - EXAMPLE

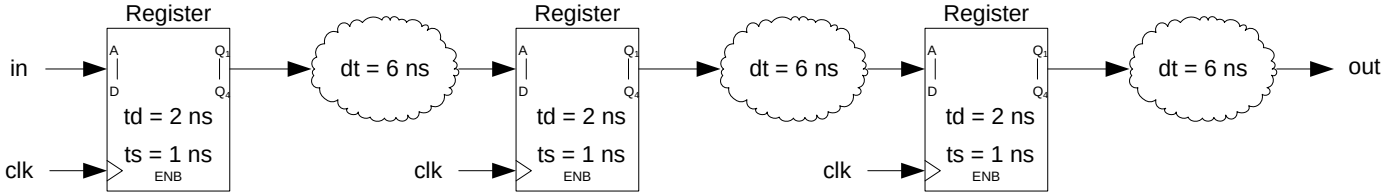
Original Pipeline



Fmax = 1/(td,reg + ts,reg + td,comb,max) = 1/(2+1+10 ns) = 77 MHz

Latency = 3 * 1/fclk + (2+7)ns = 3 * 1/77MHz + 12 ns = 48 ns

Balanced Pipeline



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(2+6+1 ns) = 111 MHz

Latency = 3 * 1/fclk + (2+6)ns = 3 * 1/111MHz + 8 ns = 35 ns

PATH RE-ORDERING

- Problem: Too Low Throughput/High Latency
  - Increased Latency is NOT OK
  - Increased Area is NOT OK

- The combinatorial delay path may be shortened, by decreasing the number of series elements in the combinatorial chain from source to drain.

PATH RE-ORDERING - EXAMPLE

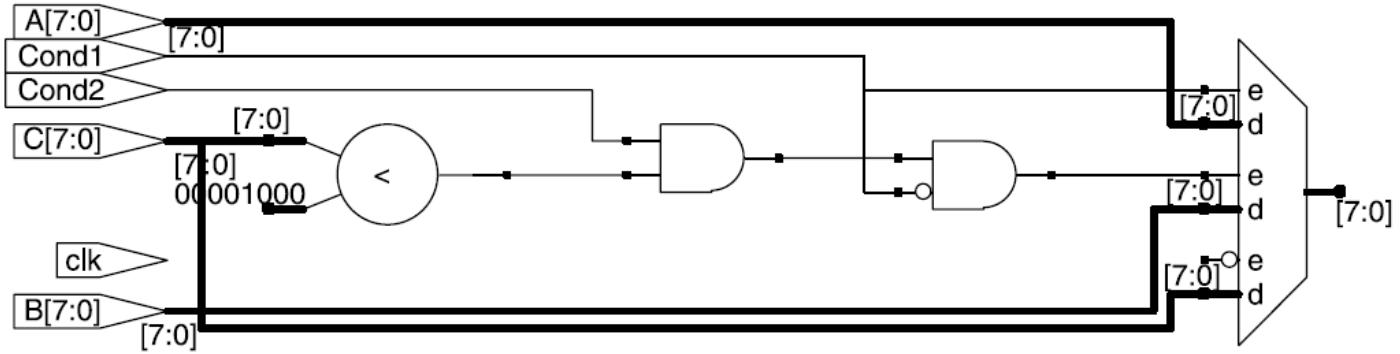


Figure 1.11 Long critical path.

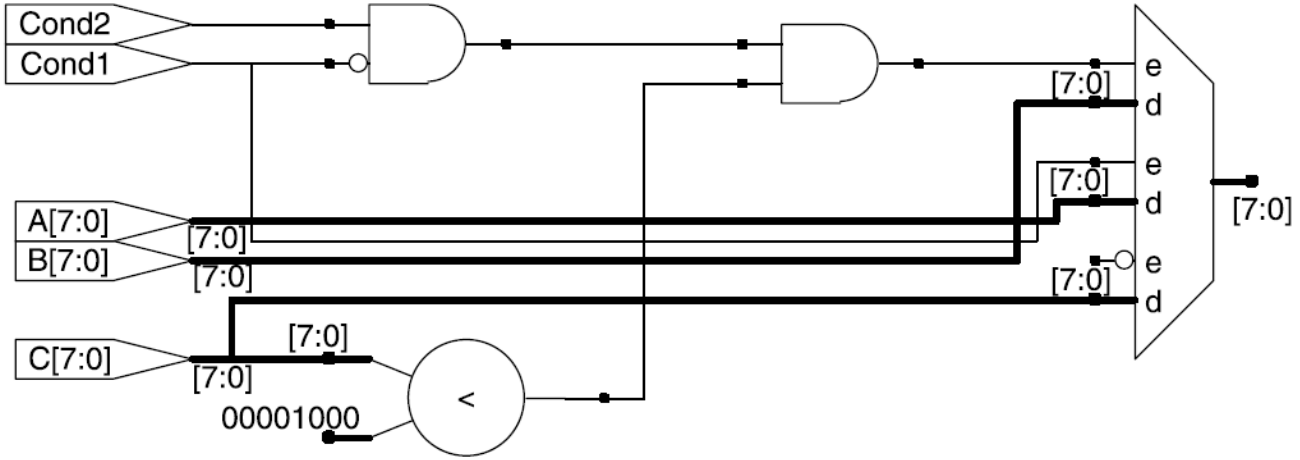


Figure 1.12 Logic reordered to reduce critical path.

PATH RE-ORDERING EXAMPLE

Original

```
module randomlogic(
  output reg [7:0] Out,
  input      [7:0] A, B, C,
  input            clk,
  input            Cond1, Cond2);
  always @(posedge clk)
    if(Cond1)
    Out <= A;
  else if(Cond2 && (C < 8))
    Out <= B;
  else
    Out <= C;
endmodule
```

Re-Ordered

```
module randomlogic(
  output reg [7:0] Out,
  input      [7:0] A, B, C,
  input            clk,
  input            Cond1, Cond2);
  wire CondB = (Cond2 & !Cond1);
  always @(posedge clk)
    if(CondB && (C < 8))
      Out <= B;
    else if(Cond1)
      Out <= A;
    else
      Out <= C;
endmodule
```

Translate to VHDL…

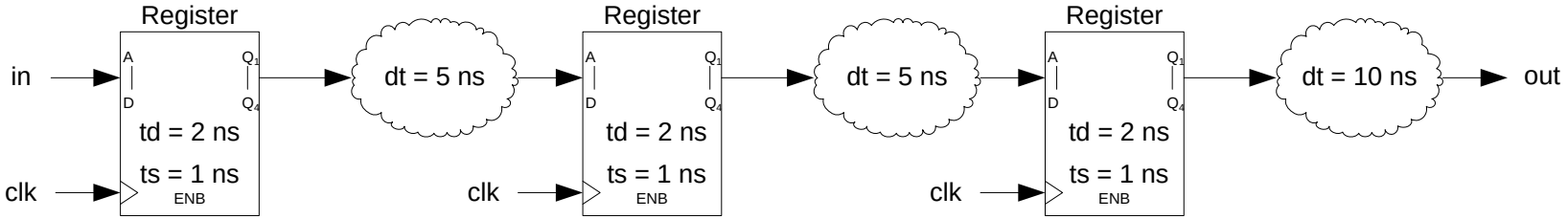
REDUCE LATENCY

- Problem: Too High Latency
  - Increased Latency is NOT OK
  - Increased Area is OK
  - Reduced Throughput is OK

- Implement functionality in combinatorial logic to remove pipelining

REDUCE LATENCY - EXAMPLE

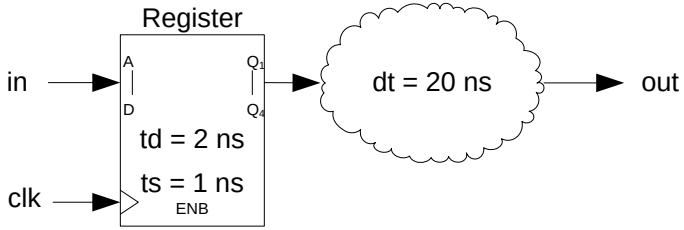
Pipelined Design



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(2+10+1 ns) = 77MHz

Latency = 3 * 1/fclk + (2+10)ns = 3 * 1/77MHz + 12 ns = 51 ns

De-Pipelined Design for low latency



Fmax = 1/(td,reg + ts,reg + td,comb) = 1/(23 ns) = 43MHz

Latency = 1 * 1/fclk + 22ns = 1 * 1/43MHz + 22 ns = 46 ns

OPTIMIZING SPEED SUMMARY

When a higher throughput is needed:

› Add registers – Pipeline the design

When Lower Latency is needed:

› Don't pipeline, use combinatorial logic

When a higher Throughput is needed, but area and latency must be maintained:

› Parallel structure – Split design in to parallel tasks
› Flatten Design – Remove unintentional priorities
› Register Balancing – Distribute combinatorial elements evenly in the pipeline.
› Path Re-Ordering – Shorten the critical timing path

Optimizing Area

OPTIMIZING AREA

- Several resources can be the limiting factor in an FPGA design:
- Logical elements
- Registers
- Memory
- Routing Resources (LAB interconnect)
- Clocks

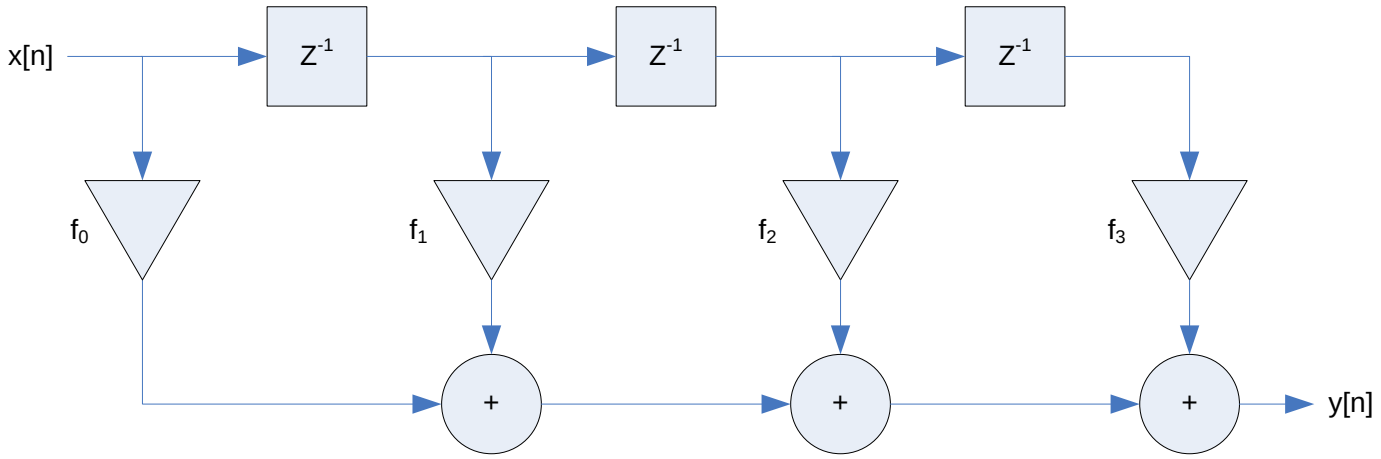
STRATEGIES FOR REDUCING AREA

- Reducing LE count:
- Rolling up the pipeline
- Resource sharing among functions
- Choosing the right reset strategy
- Using memory blocks
- Improving Routing
- Use a synchronous design
- Think “dataflow” when designing
- Reducing Clock Count
- Use few clocks, but a lot of *clock enable* signals

ROLL-UP PIPELINE

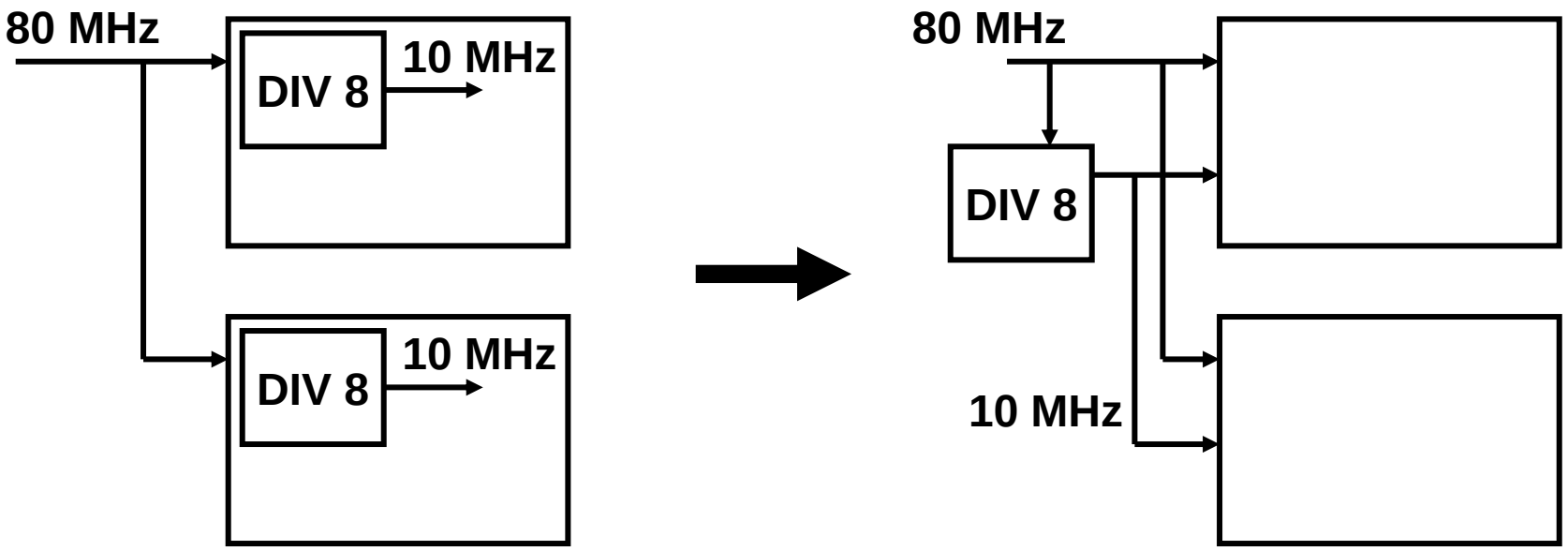
- Opposite to unrolling the pipeline
- Look into how more cycles can produce less logic
- Example: Multiplier
- Pipelined: 1-cycles, ~ 10 LE
- Multiplexed: 8-cycles, ~ 3 LE

CONTROL BASED LOGIC SHARING



- Using one multiplier per tap allows the filter to run at >200MHz
- If the throughput frequency required is lower, then the multiplier could be reused.
- Requires additional control logic

RESOURCE SHARING



- Local instances of a similar signal can be replaced by a "global" top-level signal.
- Thus reducing identical logic
- May not be detected automatically by the compiler

RESET STRATEGY

- Using reset for resources that does not have a build-in reset, will increase area consumption
- Using asynchronous reset for resources with synchronous reset will also increase area consumption

ASYNC / SYNC RESETS

Asynchronous Reset

```
If reset = 0 then
  sr <= 0;
Elsif rising_edge(clk) then
  sr <= shift_right(iDat, 1);
Endif;
```

Synchronous Reset

```
if rising_edge(clk) then
  if reset = 0 then
    sr <= 0;
  else
    sr <= shift_right(iDat, 1);
  endif;
Endif;
```

RESET OF RESOURCE WITH NO RESET

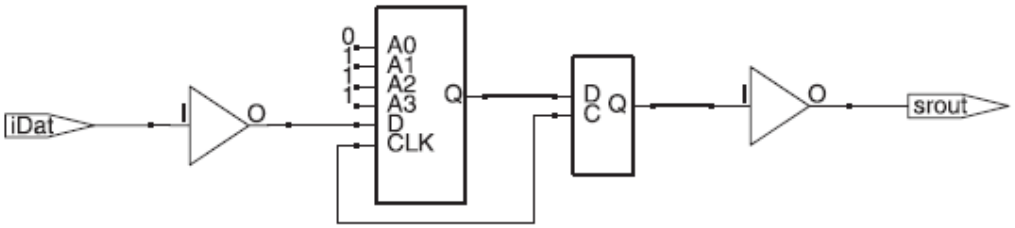


Figure 2.5 Shift register implemented with SRL16 element.

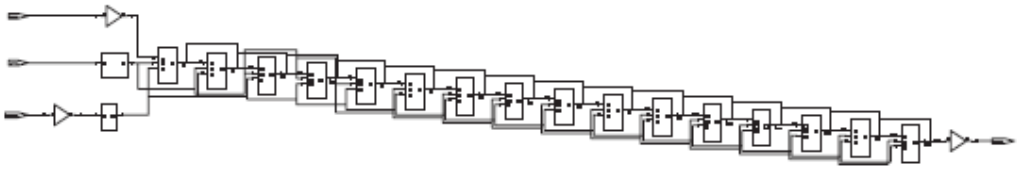


Figure 2.6 Shift register implemented with flip-flops.

- To use build-in functions
- Check if they have build in reset
- Consider if reset is necessary
- Check technology map viewer

RESOURCE WITH SYNC RESET (1)

- DSP Blocks have build-in sync reset inputs
- Using **async** reset adds logic to outputs
- Check blocks before adding reset signals

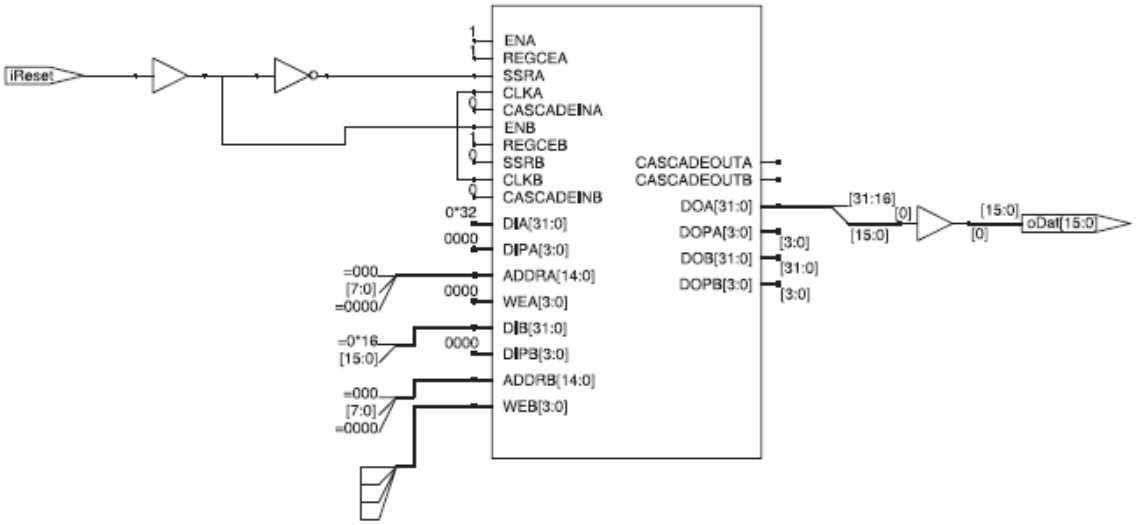


Figure 2.9 Xilinx BRAM with synchronous reset.

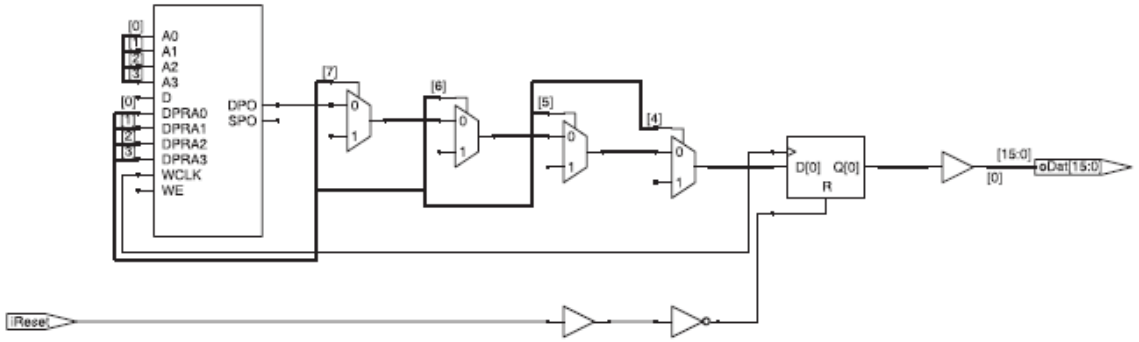


Figure 2.10 Xilinx BRAM with asynchronous reset logic.

RESOURCE WITH SYNC RESET (2)

Table 2.4 Resource Utilization for BRAM with Synchronous and Asynchronous Resets

Implementation	Slices slice	Flip-flops	4 Input LUTs	BRAMs
Asynchronous reset	3415	4112	2388	0
Synchronous reset	0	0	0	1

› DON'T USE ASYNC RESET TO RESET RAM!!!

USE THE RIGHT CODING STYLE

```
ARCHITECTURE arch OF shift_1x64 IS
TYPE sr_length IS ARRAY (63 DOWNTO 0) OF STD_LOGIC;
SIGNAL sr: sr_length;
BEGIN
PROCESS (clk)
BEGIN
IF (clk'EVENT and clk = '1') THEN
IF (shift = '1') THEN
sr(63 DOWNTO 1) <= sr(62 DOWNTO 0);
sr(0) <= sr_in;
END IF;
END IF;
END PROCESS;
sr_out <= sr(63);
END arch;
```

- Altera Recommended HDL Coding Style:
- http://www.altera.com/literature/hb/qts/qts_qii51007.pdf

USING MEMORY

- Compiler tries to implement registers in RAM where feasable
- Avoid large arrays of std_logic_vector = Matrix in registers => use Memory blocks instead
- Slow state machines may use time-multiplexing with coefficients stored in memory

OPTIMIZING AREA SUMMARY

- If you can allow more clock cycles, use time-division to re-use logic
- Consider using fewer central counters instead of multiple spread around the design hierarchy
- Be careful when choosing the reset strategy
- If resource does not support reset, then function will be implemented with std logic elements instead of dedicated hardware ➔ More logic!
- Async reset of resource with sync reset only adds logic to outputs. RAM example!
- Use memory block when feasible