Lecture 7:
RTL Coding Guidelines

Slides adapted from Ofer Shacham

with contributions from Zain Asgar

Overview

Reading

+ RTL Coding Styles That Yield Simulation and Synthesis Mismatches: https://www.synopsys.com/news/pubs/eurosnug/eurosnug2001/mills_rtl_coding_sfinal.pdf
+ Predicting Routability at the Register Transfer Level: http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf
+ RTL Coding & Optimization Guide: http://www.synopsys.com/news/pubs/sjsnug/sj02/marshall_slidesf.pdf
+ Retiming: http://www.eecs.berkeley.edu/~keutzer/classes/244fa2005/lectures/8-2-retiming.pdf
+ “System Verilog For Design,” Sutherland, Davidmann & Flake

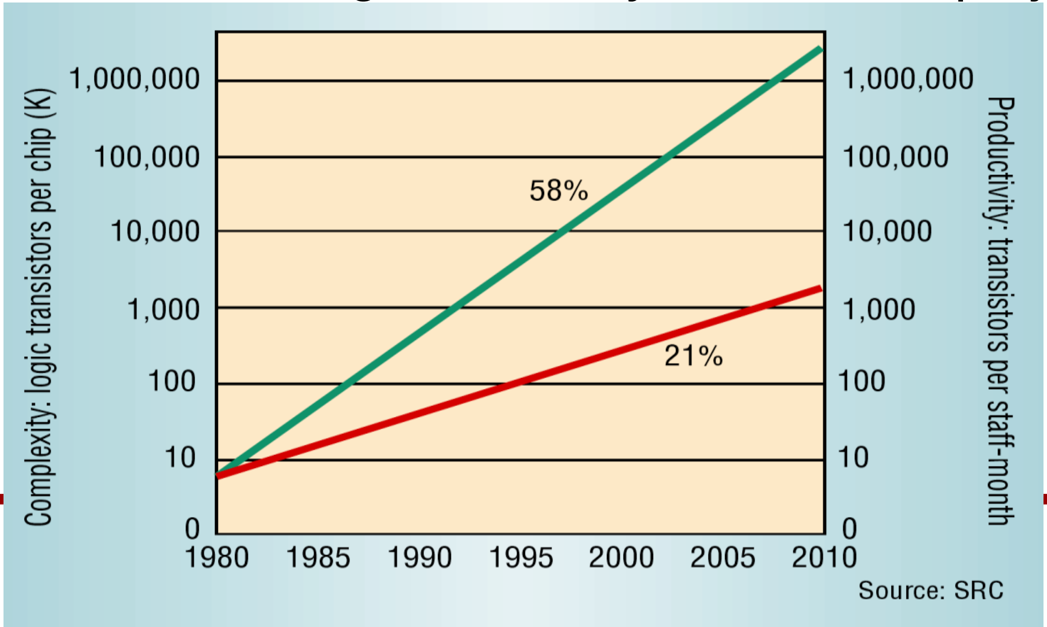
Background

While we assume you have all coded in Verilog before, if you want to create a complex system, you need to stylize the type of Verilog you create. We will talk about how to do a little preplanning, some general style rules, leveraging retiming to do some of your job, and then how to avoid / deal with communication issues.

Why Use Coding Guidelines: The Human Perspective

- Coding guidelines make the chip design process more efficient
  - Today’s designs are thousands of verilog lines
  - Written by tens/hundreds of people
  - Teams from all around the world design one chip

- Coding guidelines are set to save us from ourselves
  - Think about reading code one year after the project is done

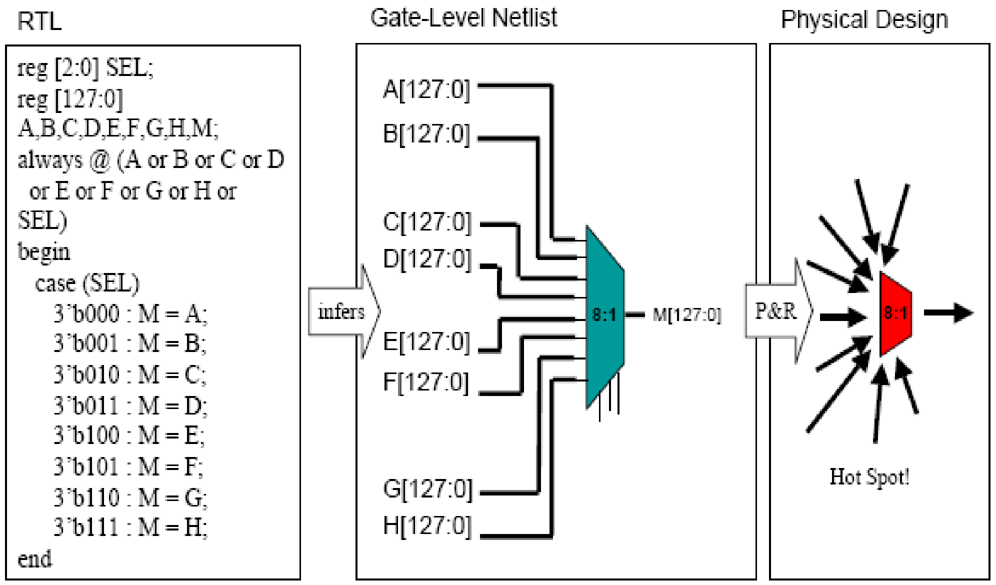


Why Use Coding Guidelines: The Tools Perspective

- Simulators can do magical things
  - But real hardware can't!

```
initial begin
        integer I;
        for (i=0; i<128; i++) begin
                mem[i] = 0;
        end
end
```

- Some issues may only be realized at the place & route phase



```
reg [2:0] SEL;
reg [127:0]
A,B,C,D,E,F,G,H,M;
always @ (A or B or C or D
  or E or F or G or H or
SEL)
begin
  case (SEL)
    3'b000 : M = A;
    3'b001 : M = B;
    3'b010 : M = C;
    3'b011 : M = D;
    3'b100 : M = E;
    3'b101 : M = F;
    3'b110 : M = G;
    3'b111 : M = H;
end
```

This Lecture is Based On…

- Lessons learned the hard way
  - Or the easy way (i.e. other people’s hard way ;-))
- Lessons learned from Zain Asgar’s presentation
- A collection of other people’s presentations/papers/books
  - See the reading list for this lecture!

Planning Is Critical: Things to Do Before RTL

- Do you have a specification for your design?
- Create a block level drawing first
  - Break the design down to manageable chunks of RTL code
  - Look at the communication paths between the chunks
- Any pre-existing physical requirements for the design?
  - Is area a limitation? Is power or frequency a limitation?
  - Do you have enough room for all your I/Os?
- What is the expected frequency of operation? what technology?
  - Roughly estimate number of logic levels between flops
  - Determine at what level(s) to register outputs
  - Where are the anticipated critical paths?

For Large Designs, You Must: Decompose & Decouple

- Think about how the blocks interact
  - Decouple state machines dependencies between blocks
  - Separate data path logic and control logic
- Start thinking about physical constraints
  - What size can your synthesis/physical design tools handle?
  - Currently it is around 200K gates (and rising)
  - Wires will be pretty long between these (large) blocks
- At the top level - constrain blocks to around 200K gates
  - This will be the layout partition
  - Flop all outputs (no need to flop inputs)
    - Decouples timing between blocks (critical path is within a block)

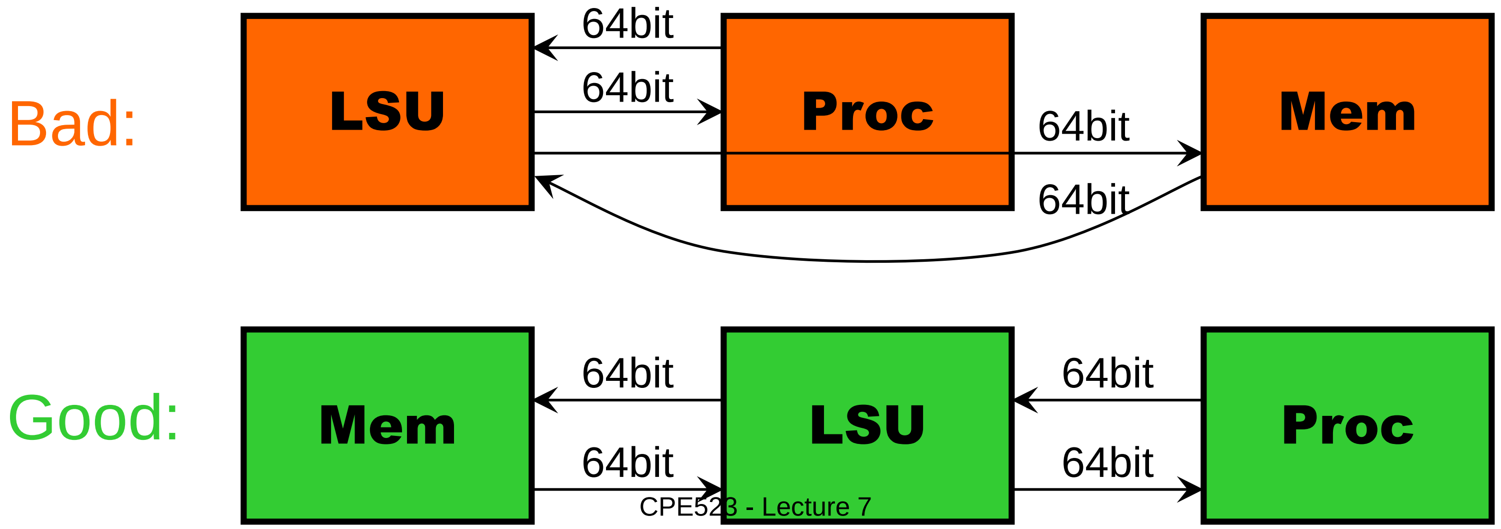
Flopping Outputs

- Want to create a modular design
  - Need to be able to build / evaluate each block
- Flopping all outputs sets the required timing on all outputs
  - Must settle before setup time of flop
- Input can arrive from other units
  - Top level designs need to estimate this time
  - But the timing of inputs is just from wiring
    - Can easily estimate from a rough layout plan
- But this means a signal can only go through one block/cycle
  - In some cases the communication takes a full cycle
    - Then you flop input and output

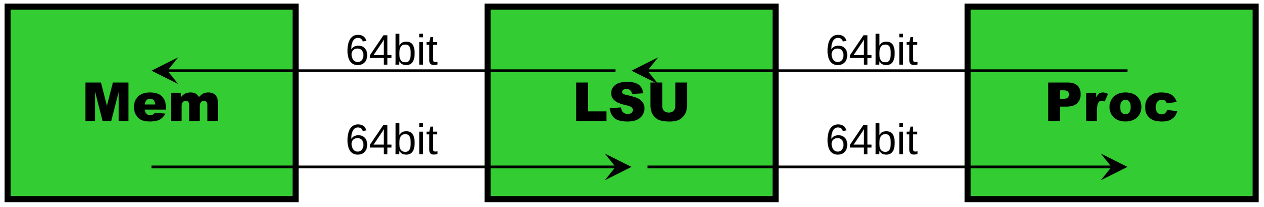
Start Planning Wires

- Create a rough floorplan
  - Units that communicate should be adjacent (when possible)
  - Make sure that not too many wires cross or loop around

- Think carefully about nets with large fanouts.
  - These can have a huge timing disparity in synthesis vs P&R
  - They also make routing hard



One More Point About Wire Planning



- While we tend to draw wires that
  - Start at the edge of a block
  - End at the edge of the next block
- Wires origins and destinations are random in a block

- So for top level wire cap / delay estimates
  - Assume all wires start from middle of the physical block

Keep Clean Interfaces

- Keep interfaces simple
  - Don’t overload signals unless absolutely necessary
  - Separate meaningful signals

- Key to successful **verification**, IP reuse, code robustness…

Bad:

```
   input [73:0] proc2lsu; // interface processor to ld/st unit
   output [89:0] lsu2proc;
   assign proc2lsu = {req, op, addr, data…};
   assign addr = lsu2proc[37:19];
```

Good:

```
   input proc2lsu_req;
   input [7:0] proc2lsu_op;
   input [31:0] proc2lsu_addr;
   input [48:0] proc2lsu_data;
```

Better: Use a SV interface

```
   interface proc2lsu (input logic Clk);
      logic req;
      …
   endinterface
```

Files and Folders

- Separate design from environment files!
  - Even if the environment files are in Verilog / System Verilog
  - Separate the synthesis / place-and-route scripts
- It is conventional to name files **exactly** as the module inside
  - Some tools expect it

Bad:

```
// File name: Arithmetic_Logic.v
module alu (…);
…
```

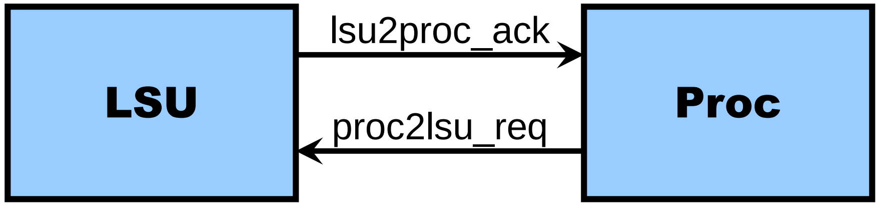
Good:

```
// File name: alu.v
// Description: arithmetic-logic unit …
module alu (…);
```

- For shared constants/parameters use headers (or SV packages)
  - Don’t define twice (different namespace resolution between tools)

Naming Conventions

- Have a naming convention!
- Verilog is case sensitive
  - Avoid mixing CapitalStyle with underscore_style
- Avoid names that look/sound like keywords
  - Bad: “wire module_proc_input;”
- Interface signals
  - Don’t use the signal_in / signal_out – it’s confusing
  - I like the “src2dst_signal” notation



Naming Conventions, Cont'd

- Pervasive signals should be named the same
  - Or you end-up with {Clk, CK, CLK, clk} or {rst, Reset, reset}
- Some signals are commonly named by their function
  - Enable: signal_en
  - Negation: signal_b or signal_n
  - Delayed by one clock cycle: signal_d or signal_del
  - Request / Acknowledge / Grant: signal_req / _ack / _gnt
- Some signals are named by their qualifying clock
  - signal_clk1 is clocked by clk1
  - signal_clk2 is clocked by clk2

Verilog Constructs

- Verilog (and VHDL) sucks…
  - Makes it easy to write un-synthesizable code
  - Or code which is hard to implement
  - Or code that will behave differently before/after synthesis
- It is your job to understand the limitation
- It is your job to use the right construct for the right purpose
  - Use “programming” extensions where it makes sense
    - Verification
  - Only use synthesizable subset for your design

Verilog Constructs: Tasks and Functions

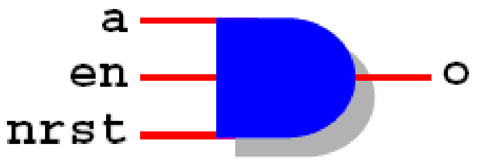
- Not the same as software methods/functions
  - Multiple function calls occur across die space, not time
- Don’t use tasks
  - They only work well in simulation
- Be careful with functions
  - A function can only be a block of combinatorial logic and hence synthesizable
    - Sequential logic not allowed in functions
  - If your logic infers a latch ➔ synthesis will not match simulation
    - Worse yet the synthesis tool will not complain

Verilog Constructs: Function Example

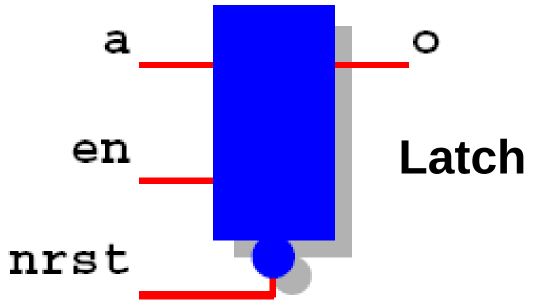
```
module bad_latch(
    output o,
    input a,
    input en,
    input nrst);

    function latch;
        input a, nrst, en;
        if      (!nrst)   o = 1'b0;
        else  if  (en)   o = a;
    endfunction

    assign o = latch(a, nrst, en);
endmodule
```

```
module good_latch(
    output reg o,
    input a,
    input en,
    input nrst);
always @(a or rst or en)
    if      (!nrst)   o = 1'b0;
    else  if  (en)   o = a;
endmodule
```

Synthesized result!



Synthesized result



Don Mills, EuroSnug http://www.synopsys.com/news/pubs/eurosnug/eurosnug2001/mills_rtl_coding_sfinal.pdf

Verilog Constructs: Loops

- Be careful with loops
  - They usually create more hardware
  - Make sure there is a constant termination
    - Loops with a variable termination are not synthesizable

- Rethink – do you really want a loop or a tree?
  - You are usually better of creating a tree structure yourself
  - The synthesis tool might try to create a tree for better timing
    - Don’t make the synthesis tool do the designers job

Verilog Constructs: Loop Examples (That Give Correct Results In Simulation)

```
module multiplier(input [31:0] A, input [31:0] B, output [63:0] Res);
    integer i;
    logic [63:0] partial_prod[32]; // two dimensional array to hold partial products
    logic [63:0] partial_sum[32]; // two dimensional array to hold partial sums
    always_comb begin : Good_Loop
        for (i=0; i<32; i=i+1)
            partial_prod[i] = (A[i] == 1'b1)?   {'0, B<<i}   :  '0;
    end

    always_comb begin   :  Bad_Loop
        partial_sum[0] = partial_prod[0];
        for (j=1; j<32; j++)
            partial_sum[j] = partial_sum[j-1] + partial_prod[j];
    end

    assign Res = partial_sum[31] ;
endmodule
```

This is parallel, independent logic

This creates a path of 32 adders!!

- How would you design the partial product summation?

Verilog Constructs: Always Blocks

- **Don't mix comb. and seq. logic in same always block!**
  - Logic is much clearer
  - Avoids weird issues with resets
  - Avoids latch/flop inferring that you did not plan

Bad1: mixing comb. and seq. logic

```
always @posedge Clk
    counter <= counter +1
```

Bad2: unintended latch inferred

```
always @ signal_a
    if (signal_a)
        signal_b = signal_a + 1;
```

Good1:

```
always @*
    nx_counter = counter + 1

always @posedge Clk
    counter <= nx_counter
```

Good2:

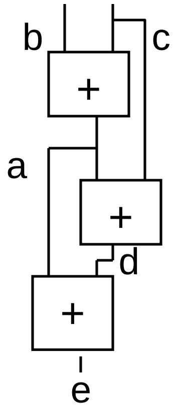
```
always @ signal_a
    if (signal_a)
        signal_b = signal_a + 1;
    else
        signal_b = signal_a + 2;
```

Verilog Constructs: Always Blocks

- For combinational logic only use blocking assignments

```
always @*
    begin
        a = b + c; // evaluates first
        d = a + c;
        e = d + a; // evaluates last
    end
```

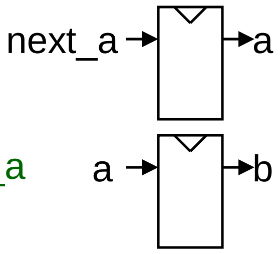
Blocking



- For sequential logic only use non-blocking assignments

```
always @(posedge Clk)
    begin
        a <= next_a;
        b <= a; // b receives a, not next_a
    end
```

Non-blocking



Verilog Constructs: Always Blocks In SV

- System Verilog attempts to make things clearer
  - Adds ‘always_comb’, ‘always_latch’ and ‘always_ff’
  - Designer intent is clearer

- Note that (unfortunately) tools are not required to check your code
  - If you put combinational logic in an ‘always_ff’ you get weird results
  - It is good practice to synthesize and trace back any flop/latch

Verilog Constructs: Case Statements

- A *case* by definition is an “if… else if… else if… [else]”
  - This means that there is priority implicitly in the code
  - Yet, tools would try to optimize away priority if they can
- Always have defaults
  - Unless you meant to infer a latch
  - But then again, “**Don’t mix comb. and seq. logic!**”
- Try to fully enumerate all cases

Verilog Constructs: Case Pragmas

- In verilog you might see “// synopsys parallel_case full_case”
  - This is a **synthesis** pragma which **should be avoided**
    - parallel_case ➔ synthesis tool should remove priority
    - full_case ➔ synthesis tool should optimize outputs for unspecified cases
- System Verilog extends verilog: ‘unique case’ and ‘priority case’
    - ‘unique case’ is the safe way to do “// synopsys parallel_case full_case”
    - ‘priority case’ is the safe way to do “// synopsys full_case”
- ‘unique case’ tells the tools (both sim. and synth.) that:
  - One and only one *case_select* matches the *case_expr*.
- ‘priority case’ tells the tools (both sim. and synth.) that:
  - At least one *case_select* matches the *case_expr*.
  - Must be evaluated in order in which it was written

Verilog Constructs: Onehot Case

- One hot state machines are tricky
  - There is no way for the tool to know you wanted one hot
  - Some libraries have special (faster) one hot mux cells

The right way to specify a one-hot mux in system verilog:

```
reg [2:0] state_vec;
typedef enum { IDLE=0, NEXT, PREV}  states_t;
always_ff @(posedge Clk)
    if (Reset) state_vec <= 3'b1;
    else state_vec <= next_state_vec;
always_comb begin
    next_state_vec = state_vec; // default
    unique case (1'b1)
        state_vec[IDLE]: …
        state_vec[NEXT]:…
        state_vec[PREV]: …
    endcase
end
assert property (@(posedge clk) $onehot(state_vec)) else $error("state is not one hot");
```

Similar to verilog's 'parameter' or '`define'

Separated sequential logic from combinational logic

Instead of the SV 'unique case', you can use:

case (1'b1) '// synopsys parallel_case full_case'

•Always verify with assertions

•More about assertions in coming lectures

A Word About Memories

- Careful with memories with too many read or write ports
  - It is easy create multi ported memories:
    - "A = Mem[a], B= mem[b], C=mem[c]" implies three ports
    - Each independent address that you read/write is another port

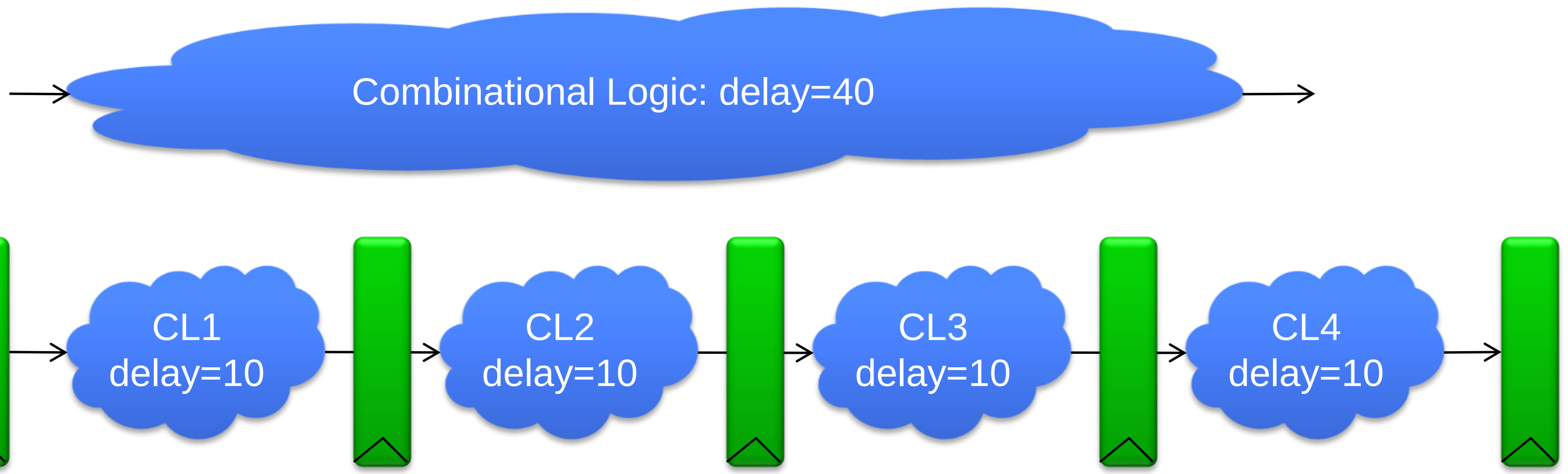
- Tradeoff: latch rams, flop rams, register files, block memories
  - Register files: full swing (no sense-amp)
    - Better for smaller memories and generally a bit faster
  - Block memories: low swing (with sense-amp)
    - Need BIST logic / support for test mode
  - Flop/latch rams:
    - Fastest, but poor density
    - Scanned flop rams are really big

Cell Type	Area@45nm
Reg. File	1-1.5um^2
SRAM*	0.346um^2
DFF / SDFF**	4.522 / 6.118um^2
Latch**	3.458um^2

* IEDM 2007 (Intel)
** Nangate Open Cell Library

Pipelining

- Remember why pipelining is useful
  - Take a function with long delay
    - Create many functions with shorter delay that run in parallel
  - Make the latency through the function slightly worse
    - But increase the throughput of the machine (concurrency)

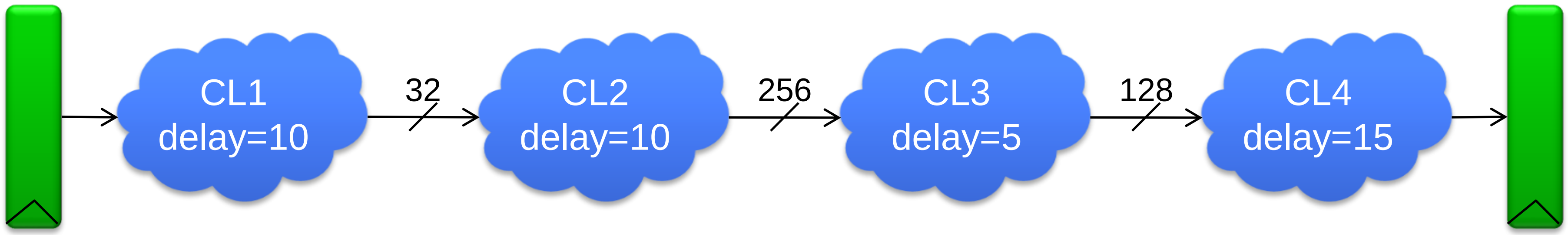


Limits to Pipelining

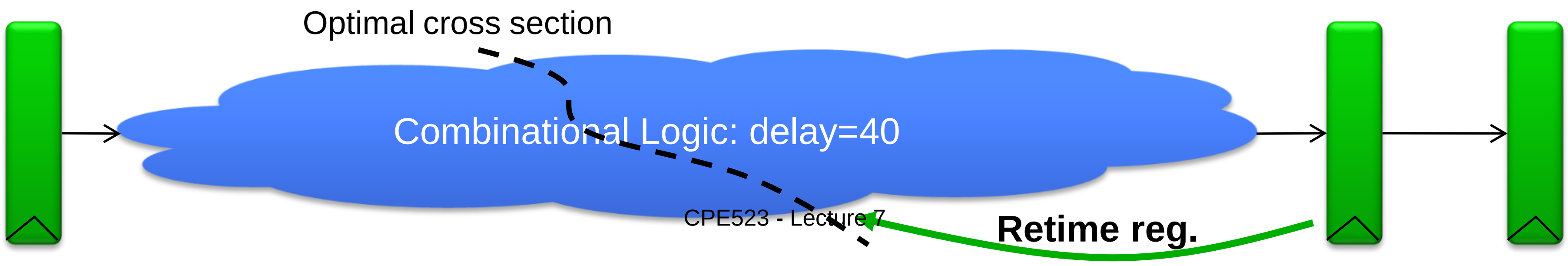
- What limits the degree of pipelining in hardware?

Retiming – A Method For Data Path Optimization

- Register location or pipe subdivision can be tricky
  - Example: where should you place an extra (middle) register?
    - Left → paths not even; middle → 8x registers; right → not even + 4x



- Solution: Don't worry about where the pipeline registers are
  - Let the tools do retiming for you
  - Tools see one big comb. cloud → can optimize cross sections



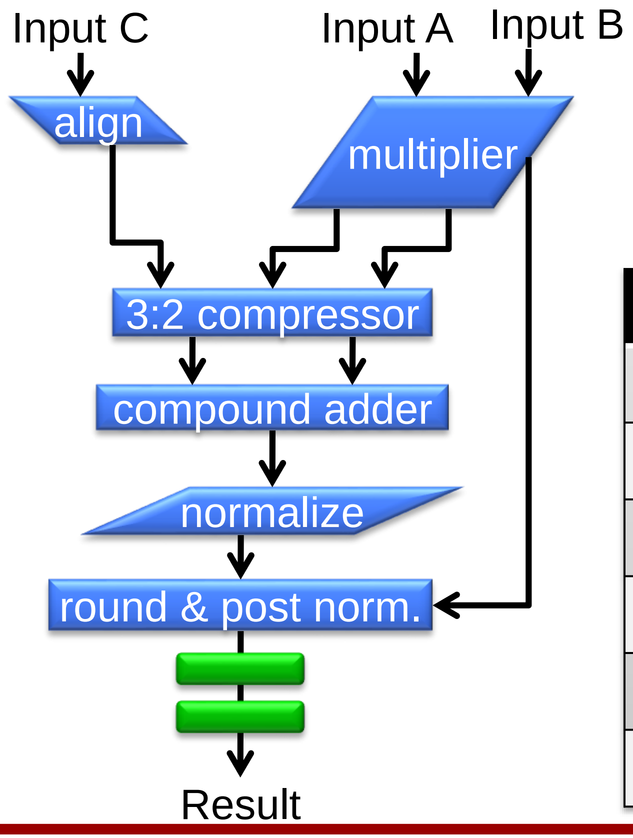
Retiming, Cont'd

- Goals
  - Clock period (min-period retiming)
  - Number of registers (min-area retiming)
  - number of registers for a target clock period (constrained min-area retiming)

- Problem definition: From circuit to a directed graph
  - Gate ←→ Vertex (V is the set of vertices)
  - Wire ←→ Edge (E is the set of edges)
  - d(v) denotes delay of a gate, d(v) >= 0
  - w(e) denotes #registers on edge, w(e) >= 0

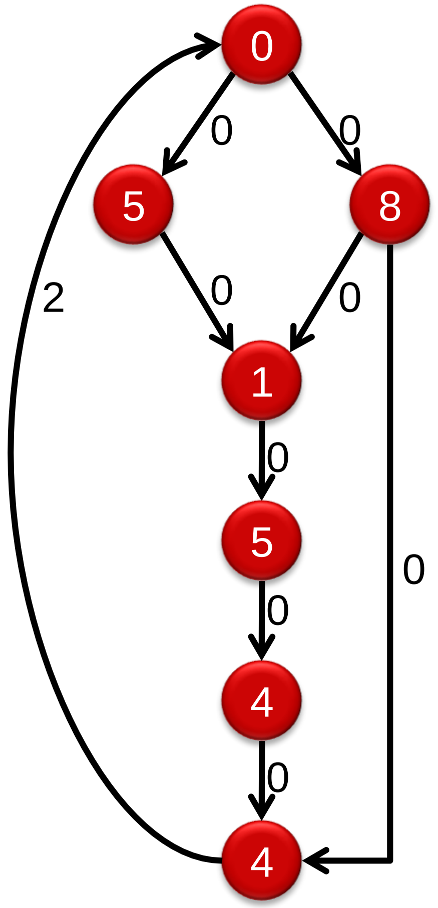
Retiming Circuit Representation (Example)

Circuit
(simplified floating point multiply accumulate unit)



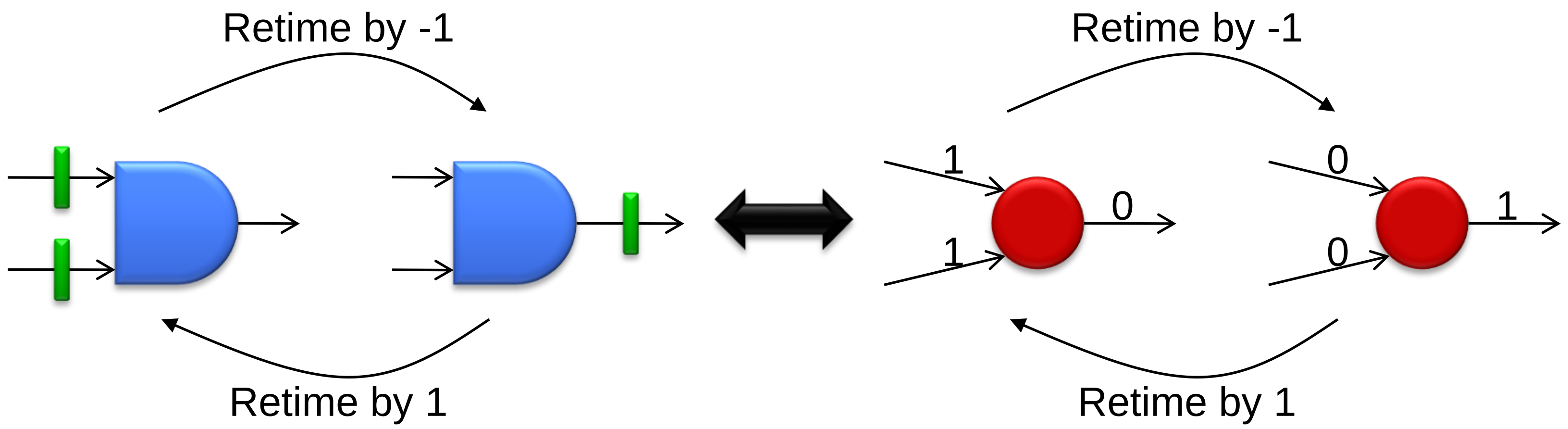
OP	Delay
align	5
mult.	8
3:2	1
adder	5
norm	4
round	4

Representation Graph



Basic Operation

- Registers can move from input to output of a gate (or vice versa)
- Does not affect gate functionalities
- Problem reduced to a graph optimization problem



Retiming Problem Formulation (Tools' Perspective)

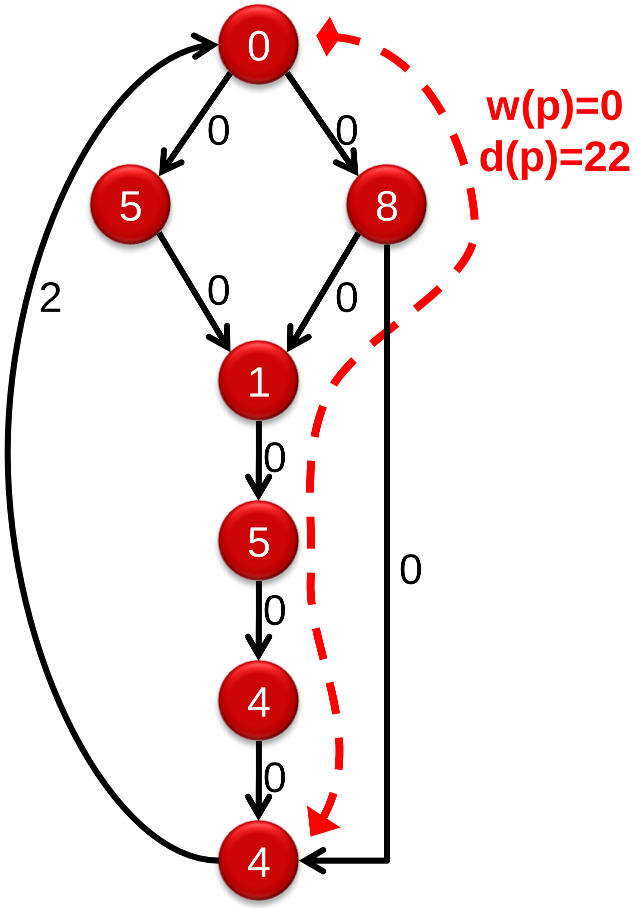
- For a path *p*: $v_0 \rightarrow v_k$

  Path delay is $d(p) = \sum_{i=0}^{k} d(v_i)$

  #Cycles is $w(p) = \sum_{i=0}^{k-1} w(e_i)$

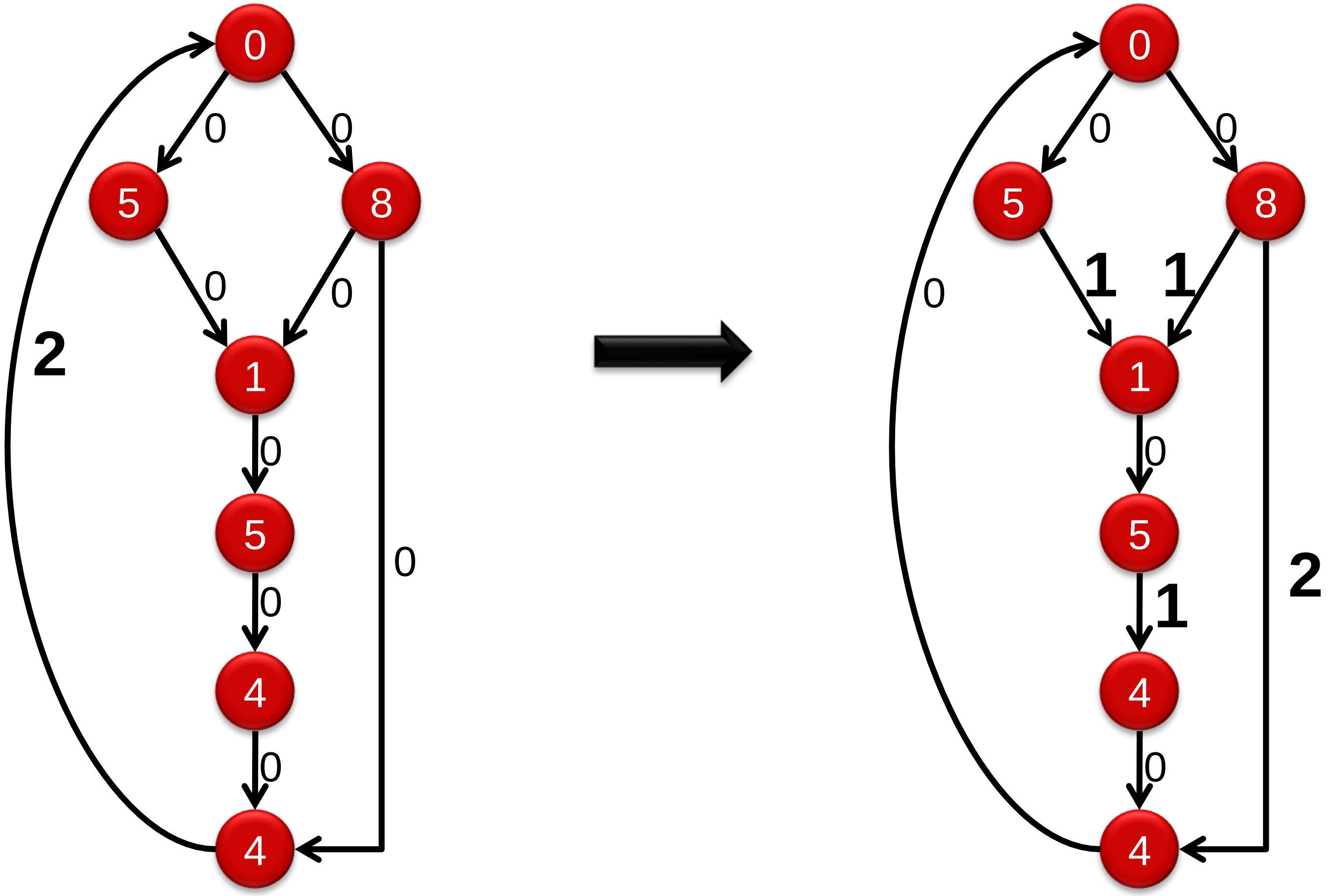
- To find the resulting clock cycle: find the longest path that has no flops: $c = max_{(p|w(p)=0)\{d(p)\}}$

- For the FMA, c = 22

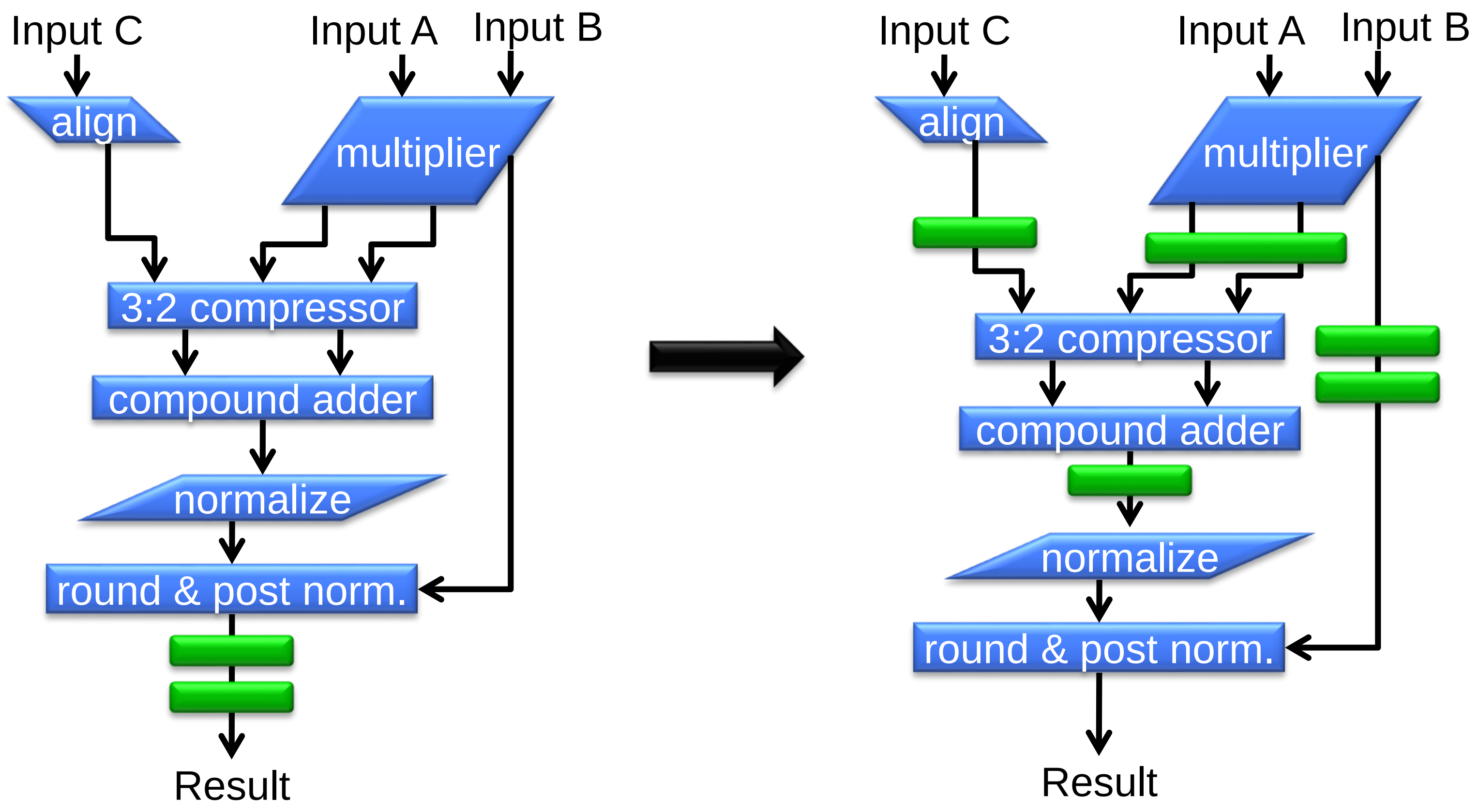


The Tool Crunch The Numbers

- Optimizing the FMA graph: c = 8



Retimed Netlist



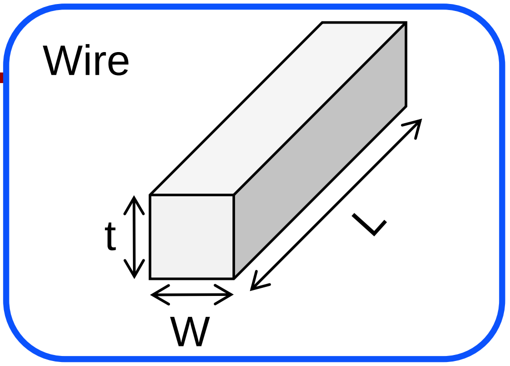
Retiming In Practice Today

- Retiming considerations
  - No combinational loops. Each loop must contain >=1 flops
  - Any power-up state of the design must be safely handled
- Retiming Achilles: Verification
  - Retiming does not change the signature values…
  - But it may change the cycle at which a signature appears!
  - Significantly more computationally intensive than combinational equivalence
- Verification technology needs to be improved for greater acceptance
- Mostly used in pipelined data-path

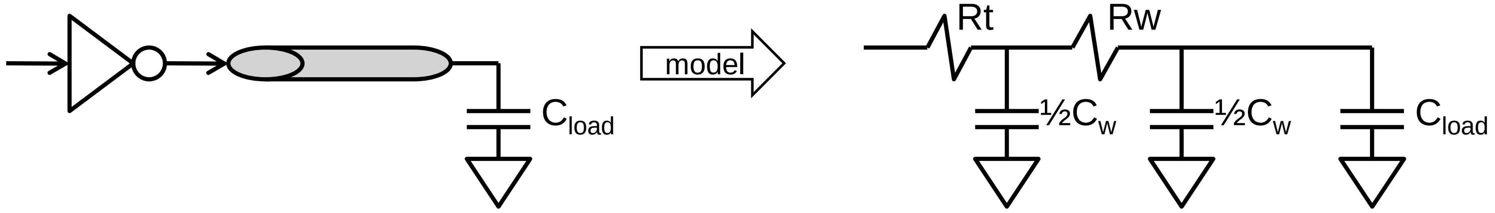
Dealing with Routing Issues Before they Become Huge Problems

- Communication can create nasty problems late in design
  - So it is good to think about them early on
  - And avoid doing things you will regret later
- Delay and power cost of long wires
- Dealing with large fan-in circuits
- Dealing with large fan-out circuits

Long Wires



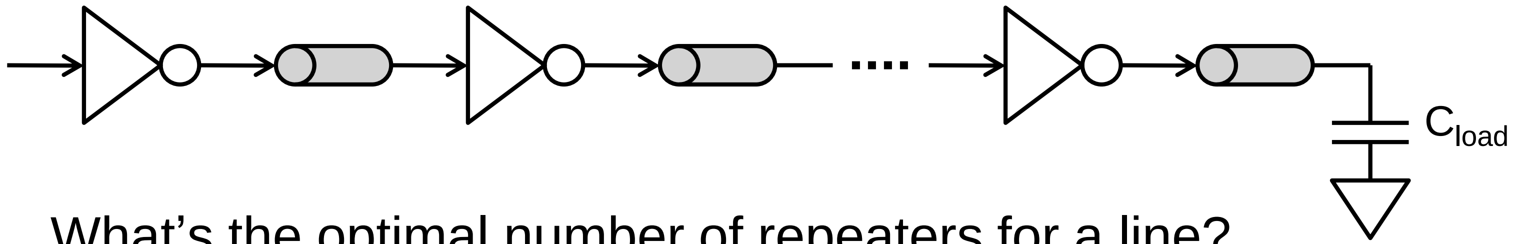
- Wires introduce noise coupling (cross-talk)
  - Why?
- Wires add both resistivity and capacitance
  - Proportional to length: $R_w = \rho/(t*W) * L$ and $C_w = \varepsilon*W/t * L$



  - For **long wires** delay is proportional to $\mathbf{L^2}$ since $\mathbf{D \sim R_w*C_w/2}$
  - Wires load and slow down the gate that drives them
- Adding distributed inverters to a wire can make it faster

Repeaters

- Repeaters: Break the wire into two or more pieces
  - Synthesis tools (should) do buffer insertion

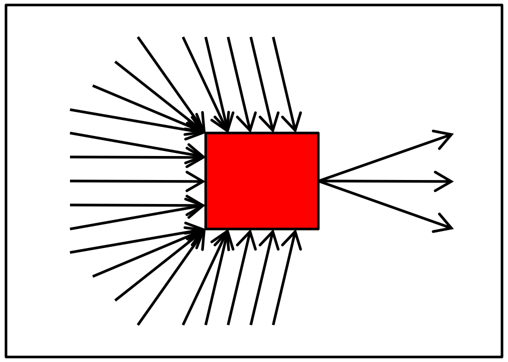


- What's the optimal number of repeaters for a line?
  - Add N identical repeaters → Delay = N * Repeater delay
  - Optimize repeater's size and find the optimal N:
  - Turns out it is **linear with L**: $N_{opt} = \boldsymbol{L} \cdot \sqrt{\frac{C_w \cdot R_w}{2 \cdot C_G \cdot R_T}}$
- This means **Delay is linear with L**!

Dealing With Large Fanin

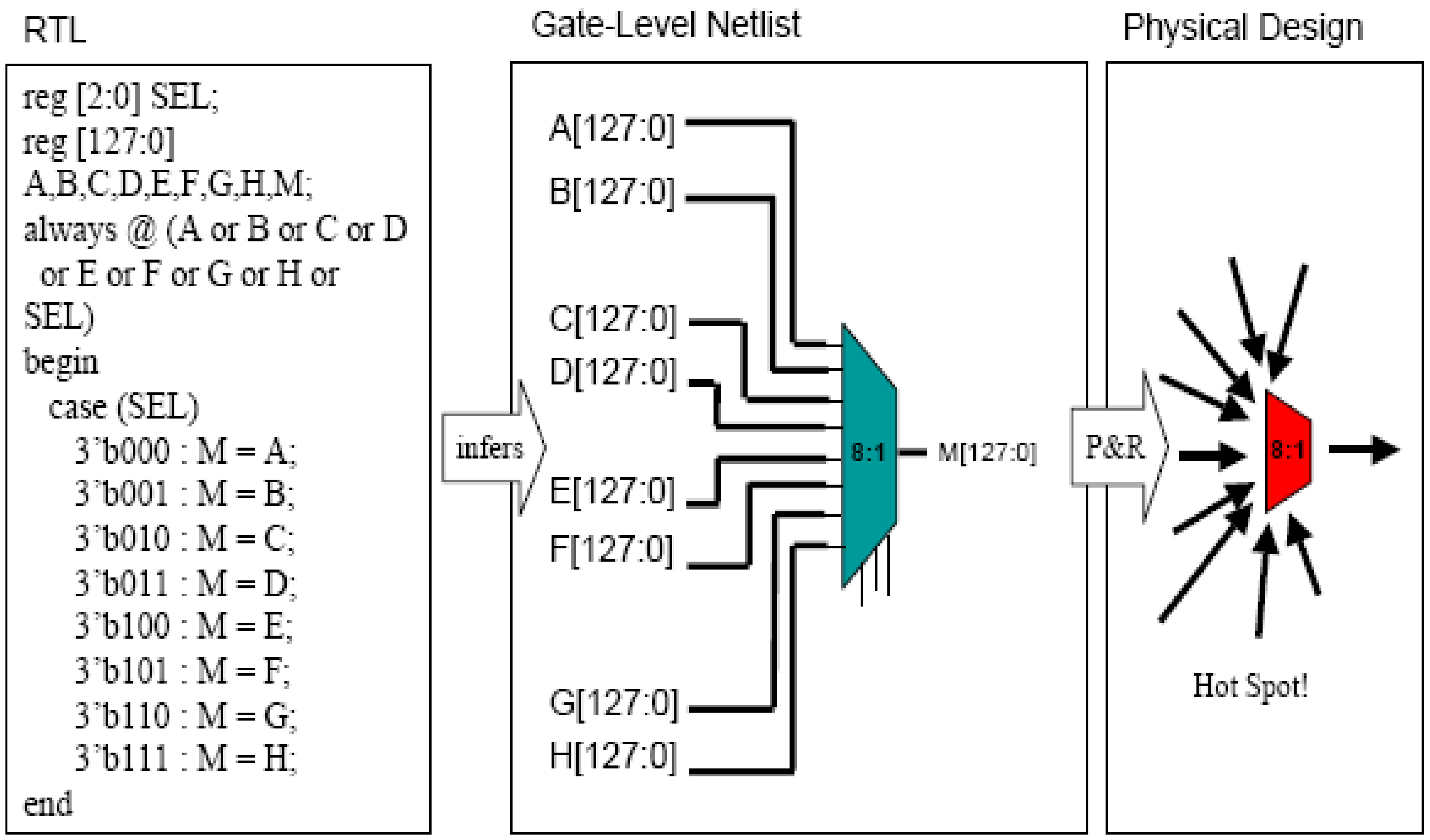
- Hot Spots
  - Huge demand for routing resources in a very small physical space
  - RTL Structures have high # input pins per unit area

- Designer should know where such structures exist (and avoid)



http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

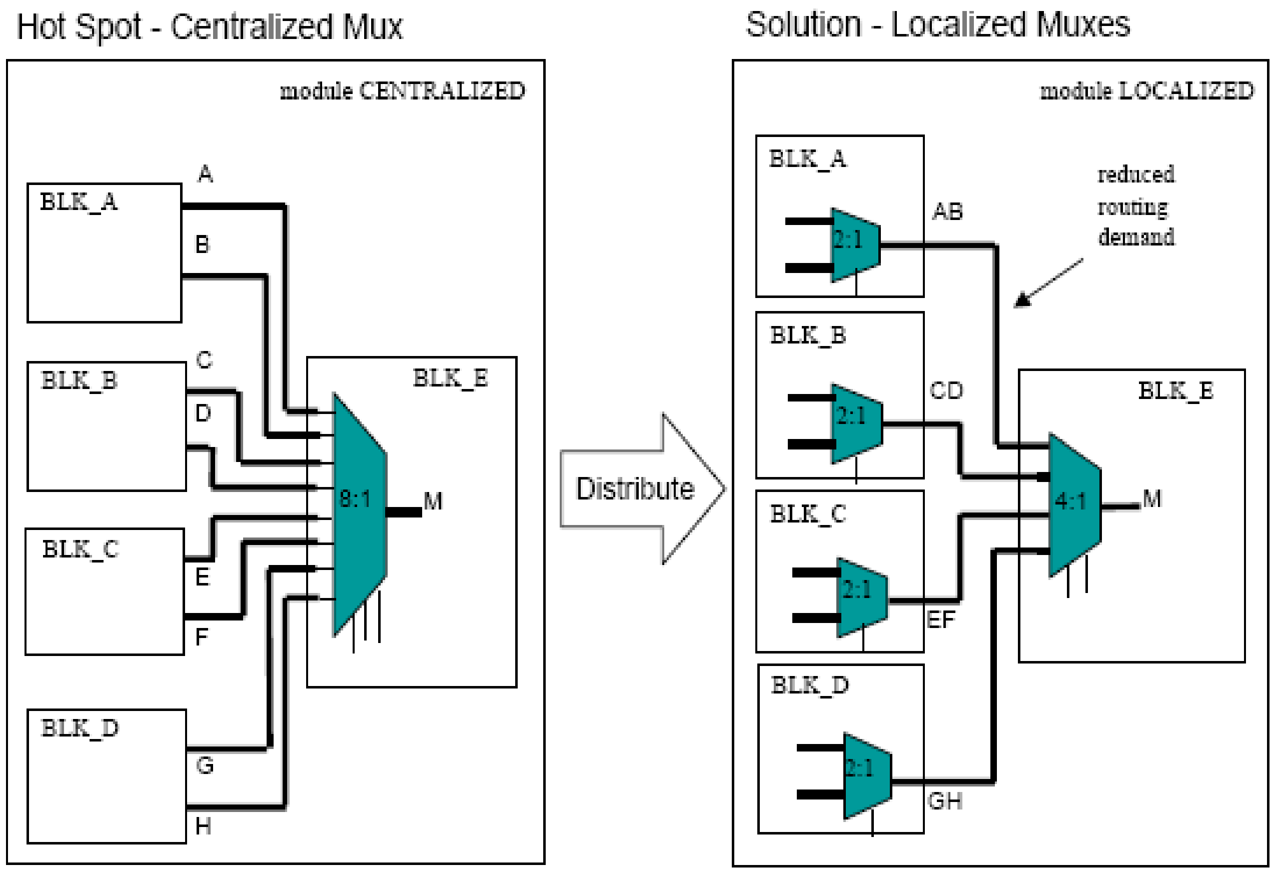
Example: Large Muxes



```
reg [2:0] SEL;
reg [127:0]
A,B,C,D,E,F,G,H,M;
always @ (A or B or C or D
  or E or F or G or H or
SEL)
begin
   case (SEL)
      3'b000 : M = A;
      3'b001 : M = B;
      3'b010 : M = C;
      3'b011 : M = D;
      3'b100 : M = E;
      3'b101 : M = F;
      3'b110 : M = G;
      3'b111 : M = H;
end
```

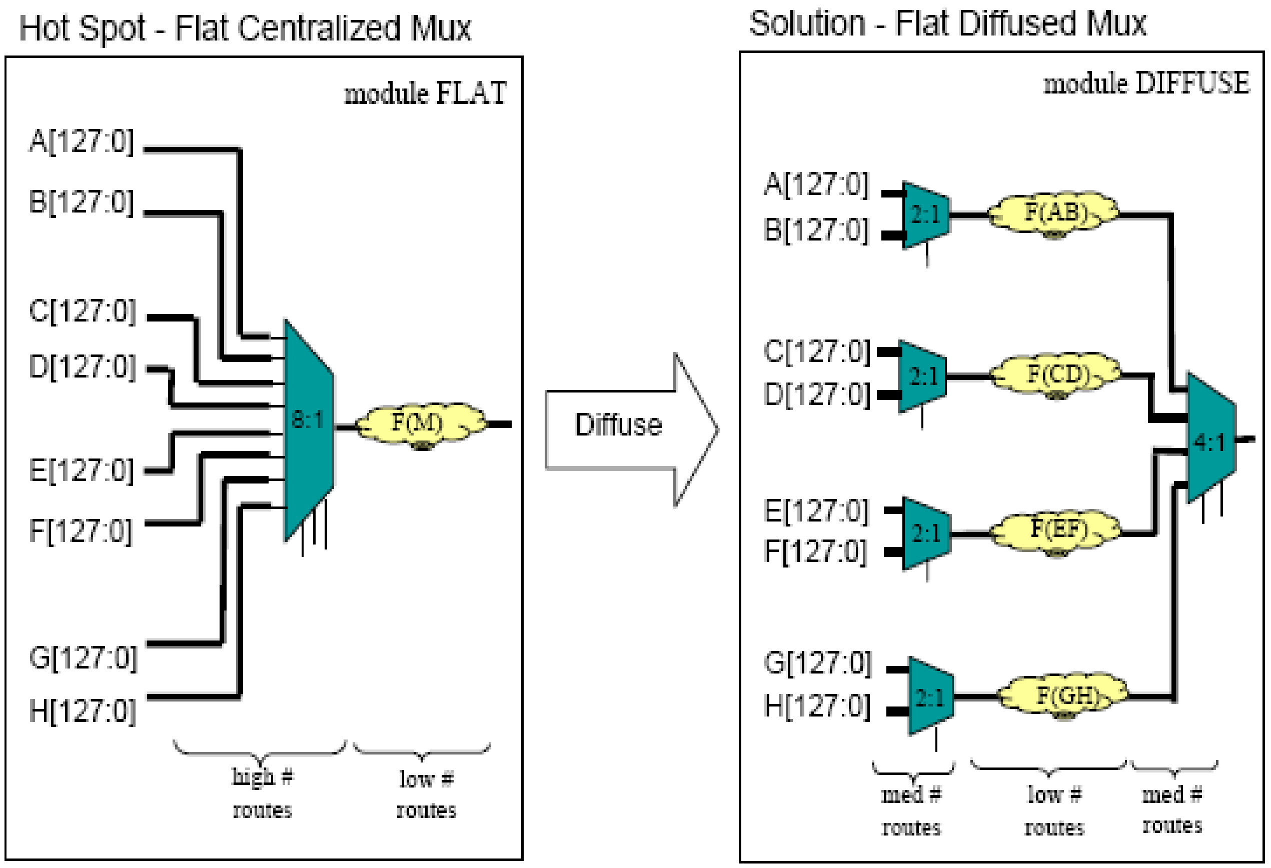
http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

Solution 1: Hierarchical Separation



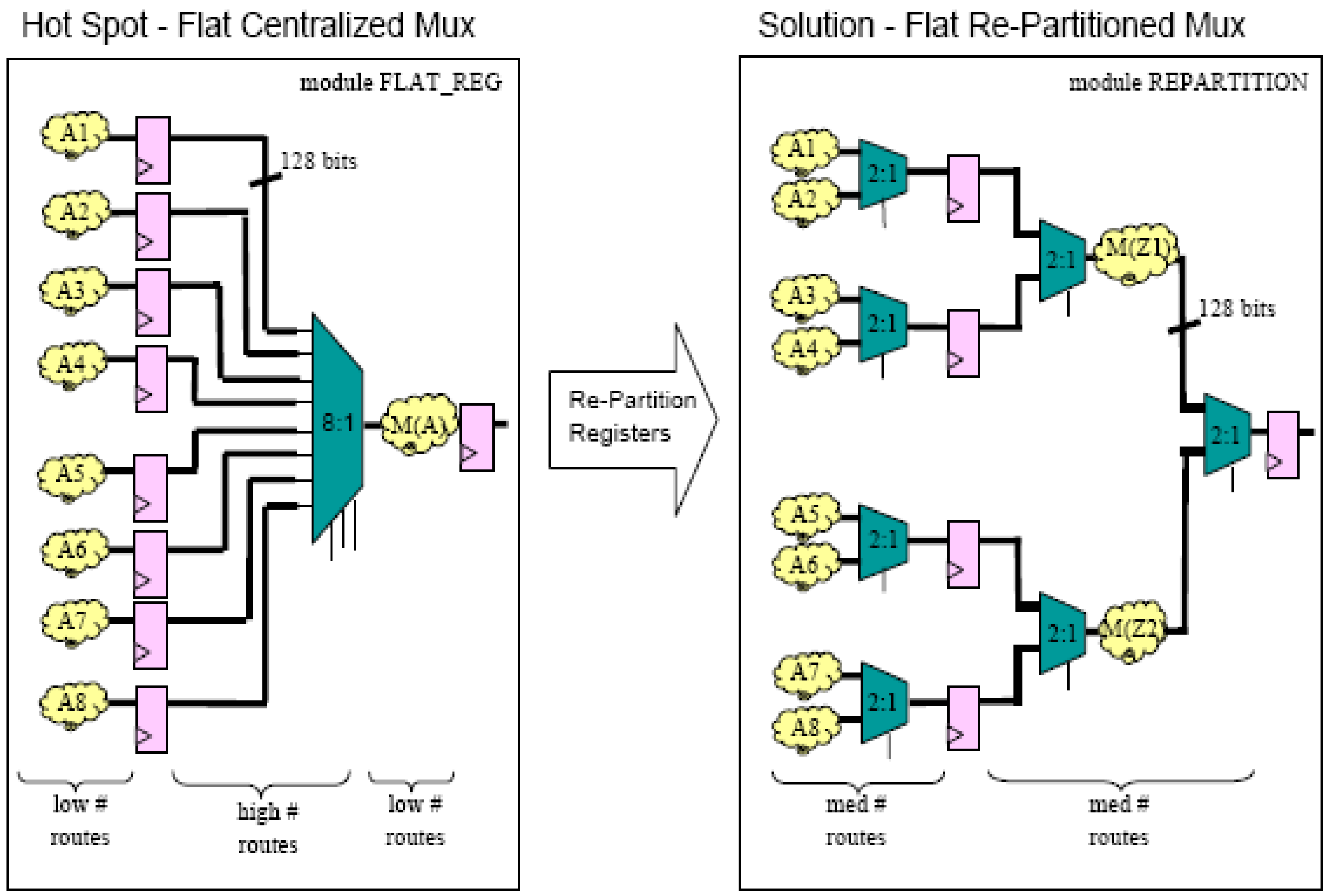
http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

Solution 2: HW Repetition & Logic Diffusion



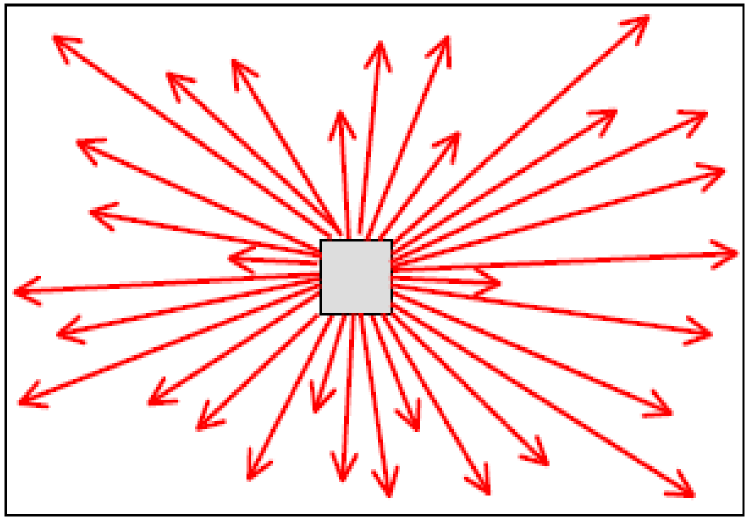
http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

Solution 3: HW Repetition & Logic Diffusion & Retiming



http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

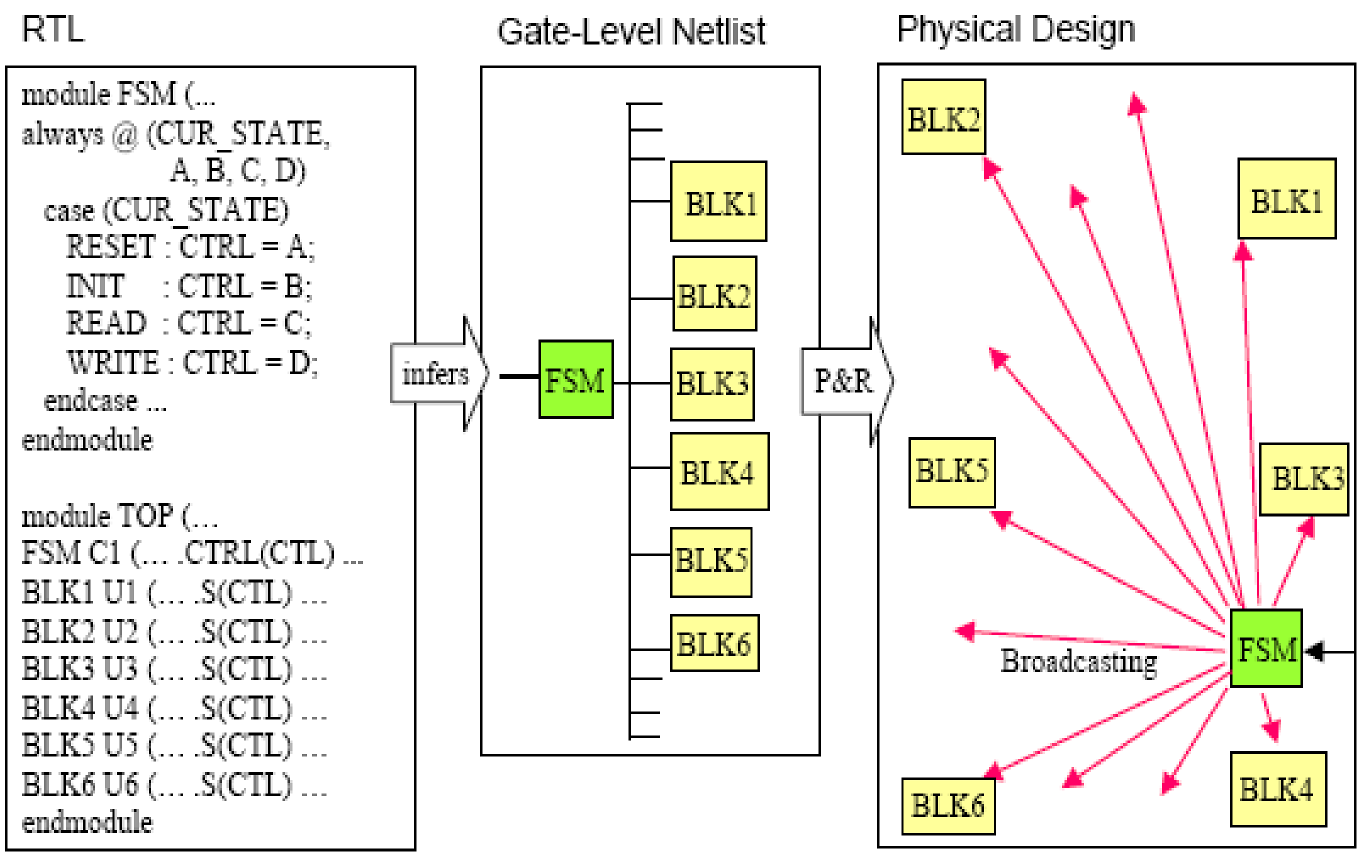
Broadcasts



- Non clock or reset signal that has high fanout, crosses many hierarchical boundaries, and is time critical
- Long routes cause timing problem & impact routability
- Early RTL floorplanning & RTL analysis can identify signals

http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

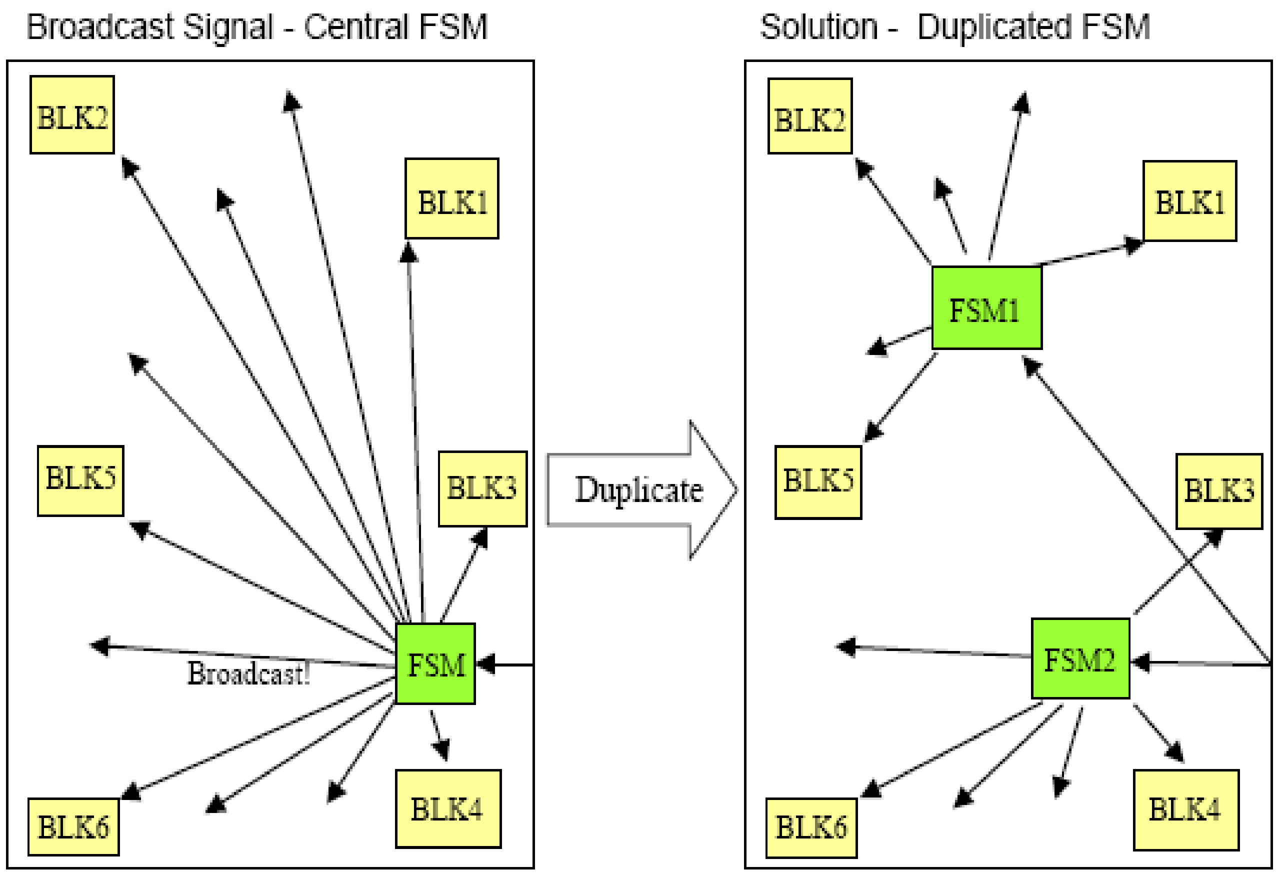
Example: Control logic



```
module FSM (...
always @ (CUR_STATE,
              A, B, C, D)
  case (CUR_STATE)
    RESET : CTRL = A;
    INIT    : CTRL = B;
    READ  : CTRL = C;
    WRITE : CTRL = D;
  endcase ...
endmodule

module TOP (...
FSM C1 (... .CTRL(CTL) ...
BLK1 U1 (... .S(CTL) ...
BLK2 U2 (... .S(CTL) ...
BLK3 U3 (... .S(CTL) ...
BLK4 U4 (... .S(CTL) ...
BLK5 U5 (... .S(CTL) ...
BLK6 U6 (... .S(CTL) ...
endmodule
```

http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

Solution: FSM Duplication



http://www.synopsys.com/news/pubs/sjsnug/sj03/marshall_pres.pdf

Bottom Line

- Gates are cheap

- Wires are expensive
  - Delay, energy and area, the big three

- Need to plan / optimize wires
  - It is ok to duplicate gates in most cases
  - Will end up being faster, lower power
    - And sometimes even smaller

Design For Verification

- Unit Interfaces – Be sure to make them simple and well defined
  - Litmus test: Can the module be replaced with a black box?

- Wrap any "magic" construct with appropriate compile flags
  - Make sure "magic" constructs are well documented

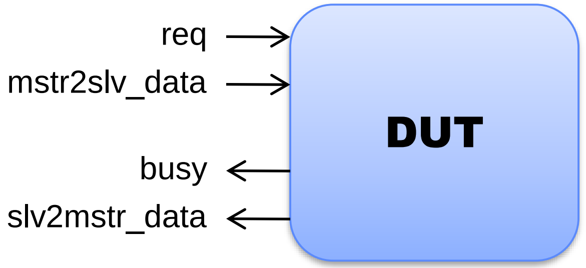
```
`ifdef MAGIC_MEM_INIT
      for (i=0; i<128; i++)  mem[i] = 0;
`endif   /* MAGIC_MEM_INIT */
```

- Where possible – parameterize
  - It is often useful to simulate smaller memories/queues/fifos
    - Can hit more corner cases in less simulation time
  - Enables mapping of a "reduced version" to emulation platforms

Points Where Verification and Design Interact: Assertions

- Don't wait to test your assumptions – **Add Assertions!**
  - For inputs:
    - E.g., Does your module receive requests **only** after *busy==0*?

```
Assumption1: assert  property (
   @(posedge clk)  req |-> $past(busy,1)==1'b0
)else
   $error("ERROR – req after busy asserted");
```



  - For things you're just not quite sure of
    - Remember the "one-hot" example from a few slides back…
  - For your block's output: to force protocol obedience

- Assertions are great for both uncovering bugs as well as for debugging/pinpointing them

Help Verification Sync With Design Intent

- Keep key signals/states explicit
  - In cases, the verification must know the current state

```
if (DUT.proc.init_done)
    Run_test()
```

Vs.

```
if (DUT.proc.no_interrupt  &  DUT.proc.alu_rdy  & DUT.proc.reg_file_init  & …)
    Run_test()
```

  - Similarly, use explicit arbitration signals

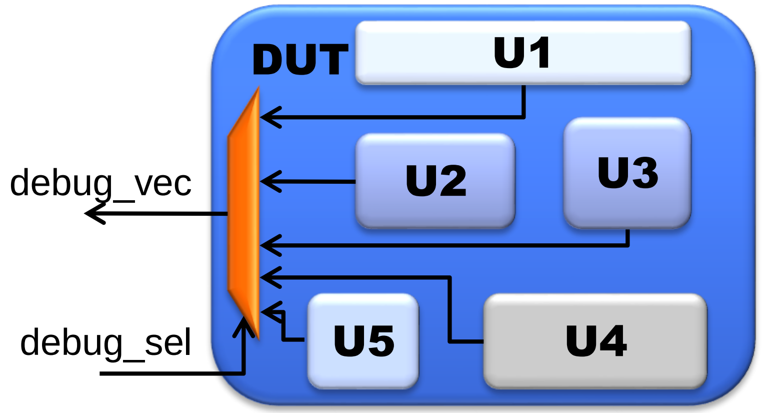
- Inline Comments – No one can read your mind
  - Especially if you optimized the heck out of the algorithm ;-)
  - Bonus: You’ll find many bugs as you write your thoughts

Design For Post-Silicon Testing
Debug Pathways - “Is it alive?”

- For *<u>key signals</u>*, and keep it *<u>as simple as possible</u>*
  - Key signals: various clocks, reset, select, etc.
  - Simple: control the debug path with minimal logic
  - Scope: unit or system level signals

- No logic should ever depend on a debug path
  - This is good news: Don’t worry about the debug path timing
    - If needed, run the machine slower

- I like the following scheme:
  - Zero synchronous logic
  - Minimal comb. logic
    - Can be hierarchical mux

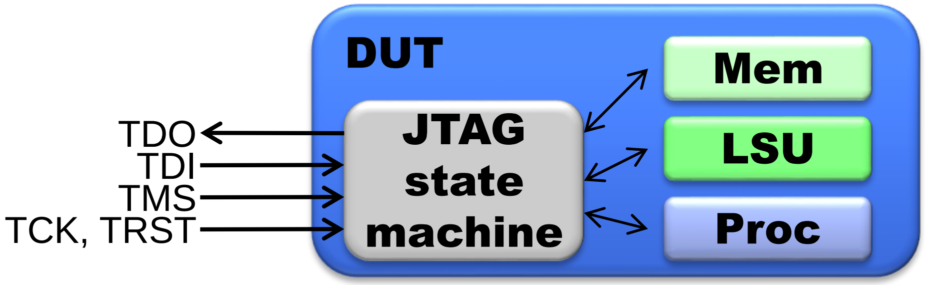


Internal State Visibility - “Does it work?”

- Is the memory being written? Does the configuration register hold the right value? Is the counter being updated?

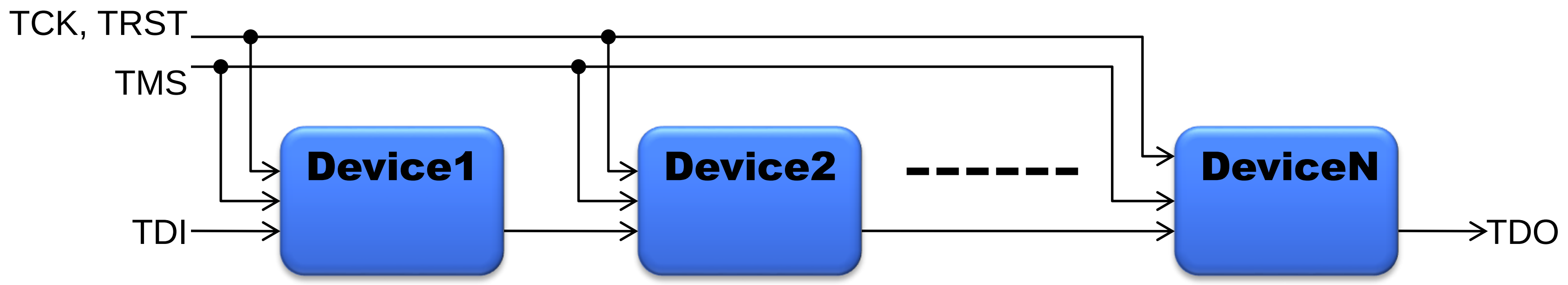
- Enable external <u>reads</u> of the explicit state of the machine
  - Explicit state = memories, register files, machine configuration…
  - Enabling writes (in addition to reads) is useful

- Typically done through the JTAG interface
  - JTAG = Joint Test Action Group, IEEE 1149.1 standard
  - Minimal additional I/O: {TCK, TDI, TDO, TMS, TRST}

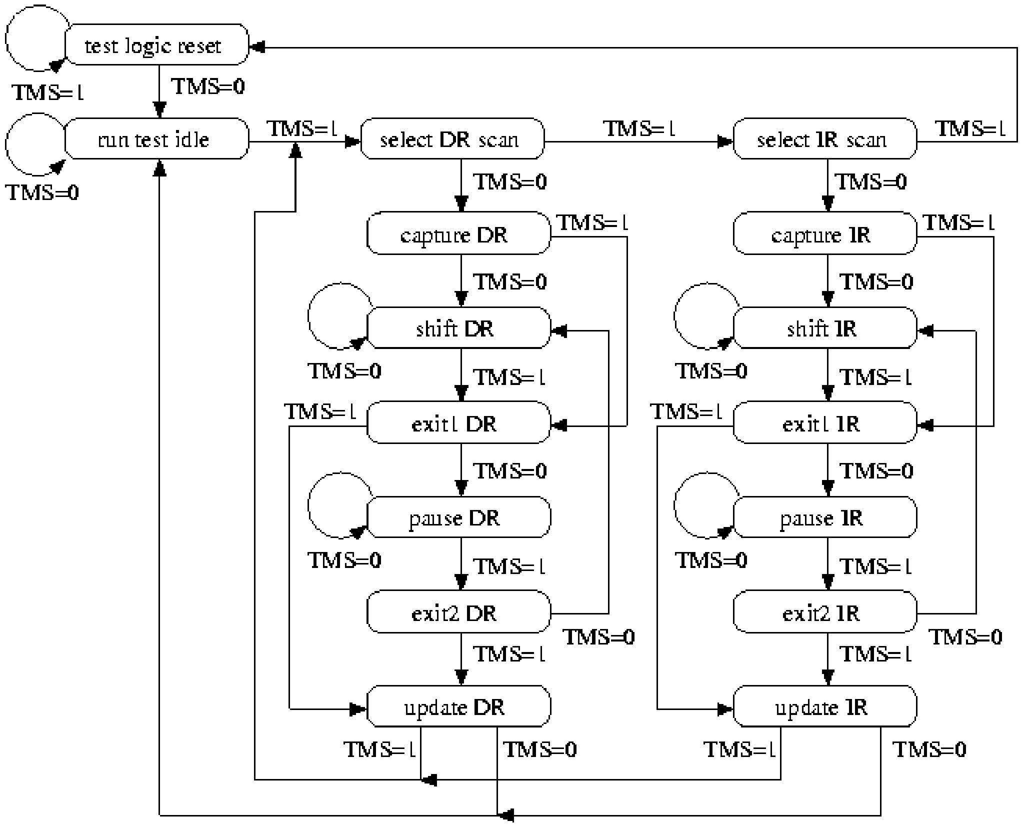


JTAG

- Works for any data width - serial interface
- Typically runs at 1/2 to 1/10 of core frequency
  - Not in critical path
  - beware of time domain crossing
- Multiple TAPs can be chained together (serially)



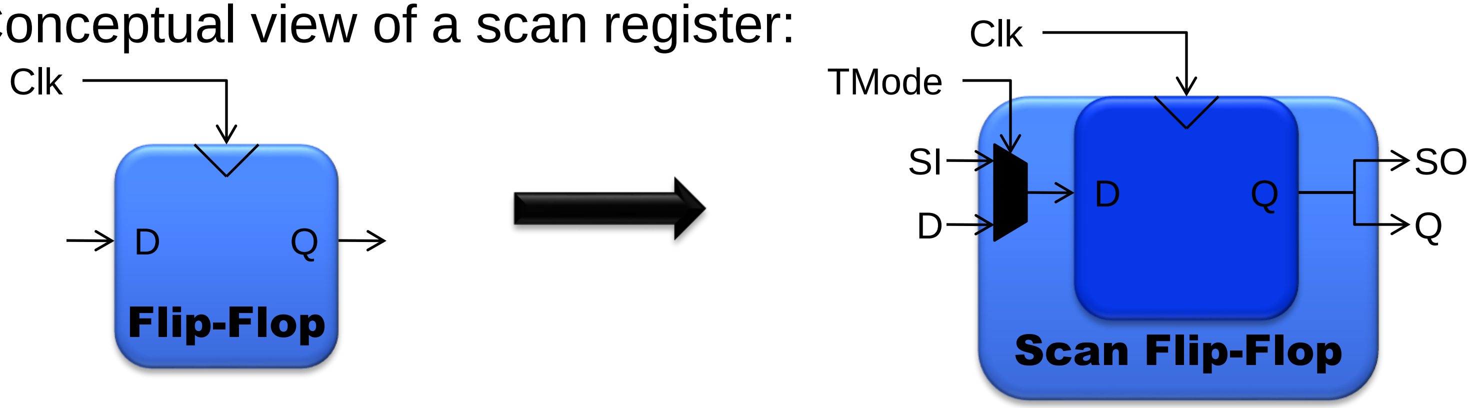
JTAG State Machine



JTAG Test Access Port (TAP) controller state transition diagram

Reading All Flops - “When all else fails…”

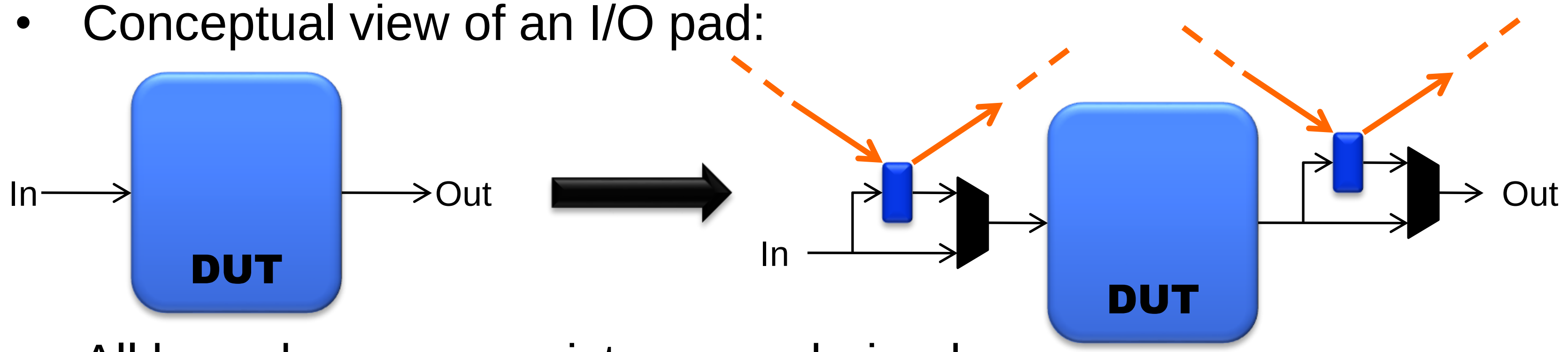
- Machine state is not only memories and register files
  - It is every flop along every path or pipeline
  - But not all of them have explicit read ports
- Conceptual view of a scan register:



- Automatic tools stitch all flops into a long chain (1000s/chain)
- **The exact machine state can be read at any point of time**

Controlling All I/Os – Boundary Scan Registers

- Conceptual view of an I/O pad:



- All boundary scan registers are chained

- Controlled via JTAG:
  – SAMPLE: sample all inputs and outputs; values are then scanned out.
  – INTEST: Force all inputs to a previously-scanned-in pattern
  – EXTEST: Force all outputs to a previously-scanned-in pattern

- What is it good for?

The End