18-643 Lecture 7: Structural RTL Design

James C. Hoe

Department of ECE

Carnegie Mellon University

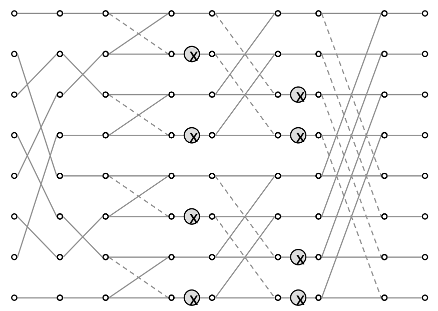
Housekeeping

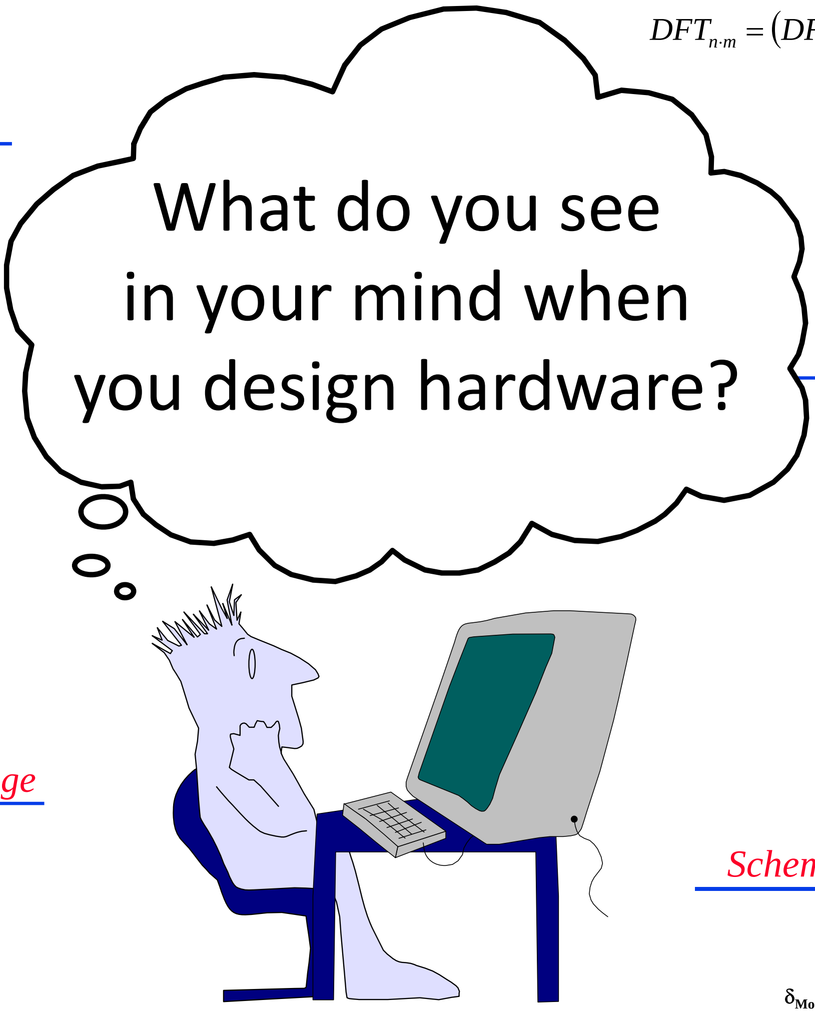
- Your goal today: think about what you think about when you design "RTL"
- Notices
  - Handout #3: lab 1, due noon, 9/22
- Readings
  - HDL Compiler for Verilog Reference Manual and others at /afs/ece/support/synopsys/2004.12/share/image/usr/local/synopsys/2004.12/doc/online/dc/print
  - Vivado Design Suite User Guide: Synthesis (UG901)

Math

$$DFT_{n\cdot m} = (DFT_n \otimes I_m) D_n^{n\cdot m} (I_n \otimes DFT_m) L_n^{n\cdot m}$$

Dataflow





What do you see in your mind when you design hardware?

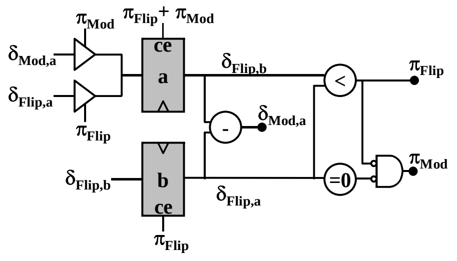
Program/Algorithm

```
for (m = 0; m < mmax; m += 1) {
    for (i = m; i < n; i += istep) {
      j =i + mmax;
      tempr = wr*data[2*j]   - wi*data[2*j+1];
      tempi = wr*data[2*j+1] + wi*data[2*j];
      data[2*j]   = data[2*i]   - tempr;
      data[2*j+1] = data[2*i+1] - tempi;
      data[2*i] += tempr; data[2*i+1] += tempi;
    }
```

Hardware Description Language

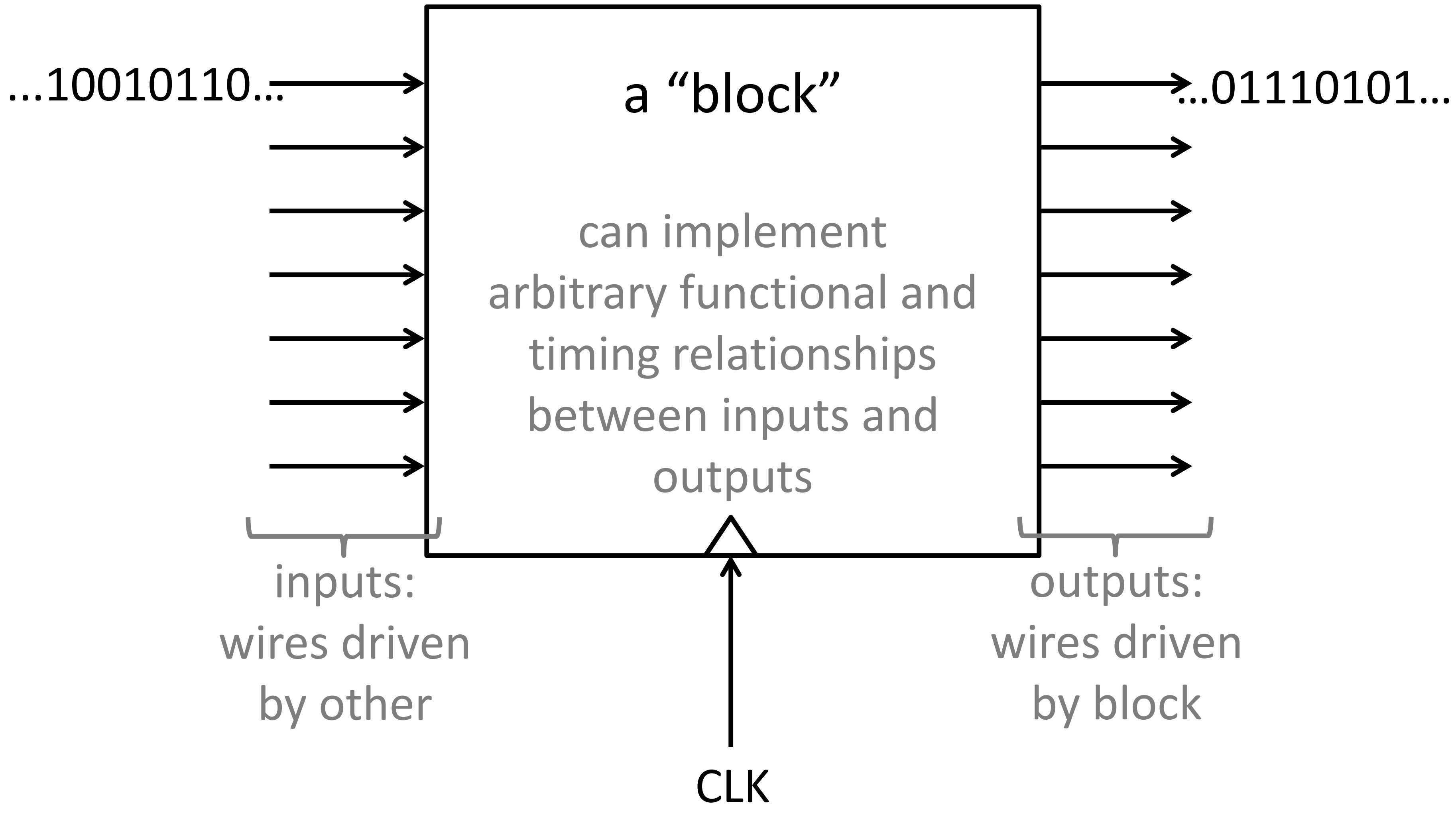
```
always @ (posedge Clk) begin
      if (a >= b) begin
            a <= a - b;
            b <= b;
      end else begin
            a <= b;
            b <= a;
      end
end
```

Schematic Capture

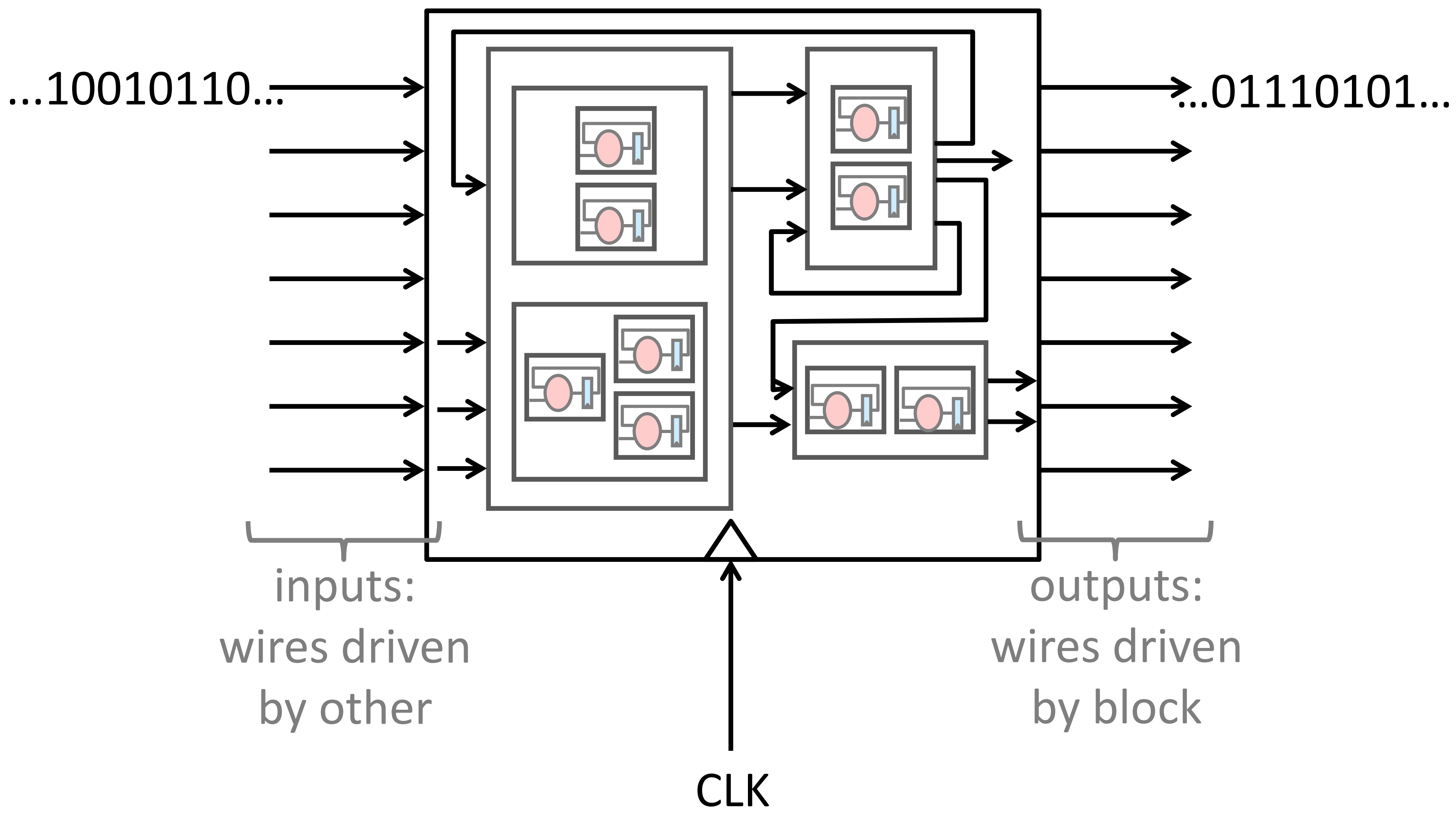


What is Structural Design?

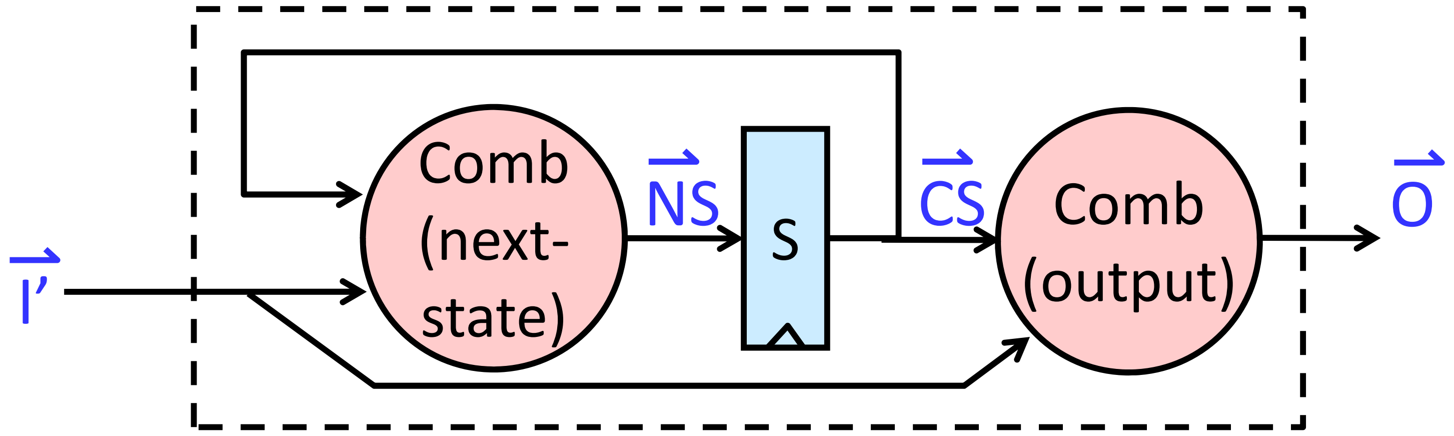
Hardware Design is Structural


...10010110...
..01110101...
a “block”
can implement arbitrary functional and timing relationships between inputs and outputs
inputs: wires driven by other
outputs: wires driven by block
CLK

Hardware Design is Hierarchical

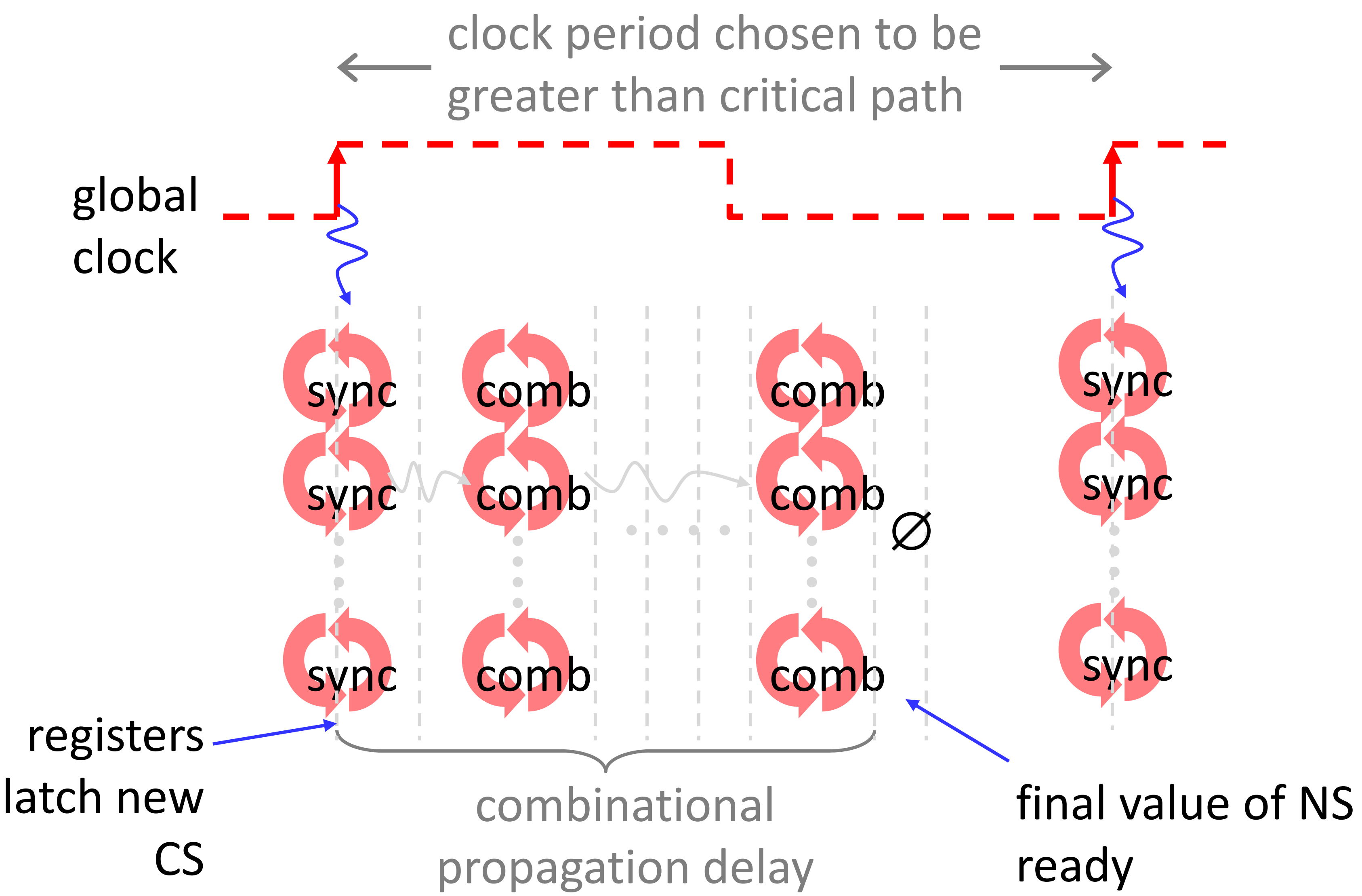


It all boils down to this

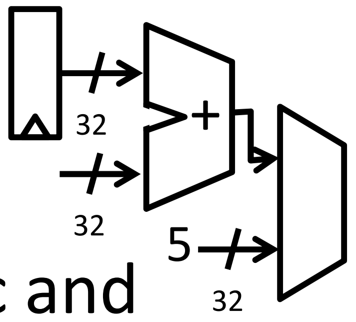


- a collection of synchronous state elements (updates on clock edge)
- a collection of combinational logic that computes next-state (NS) from current-state (CS) and input (I)
- a collection of combinational logic that computes output (O) from current-state (CS) and input (I)

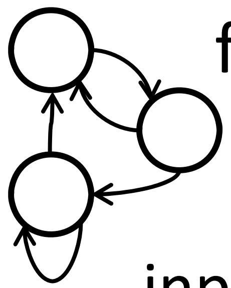
Synchronous Timing

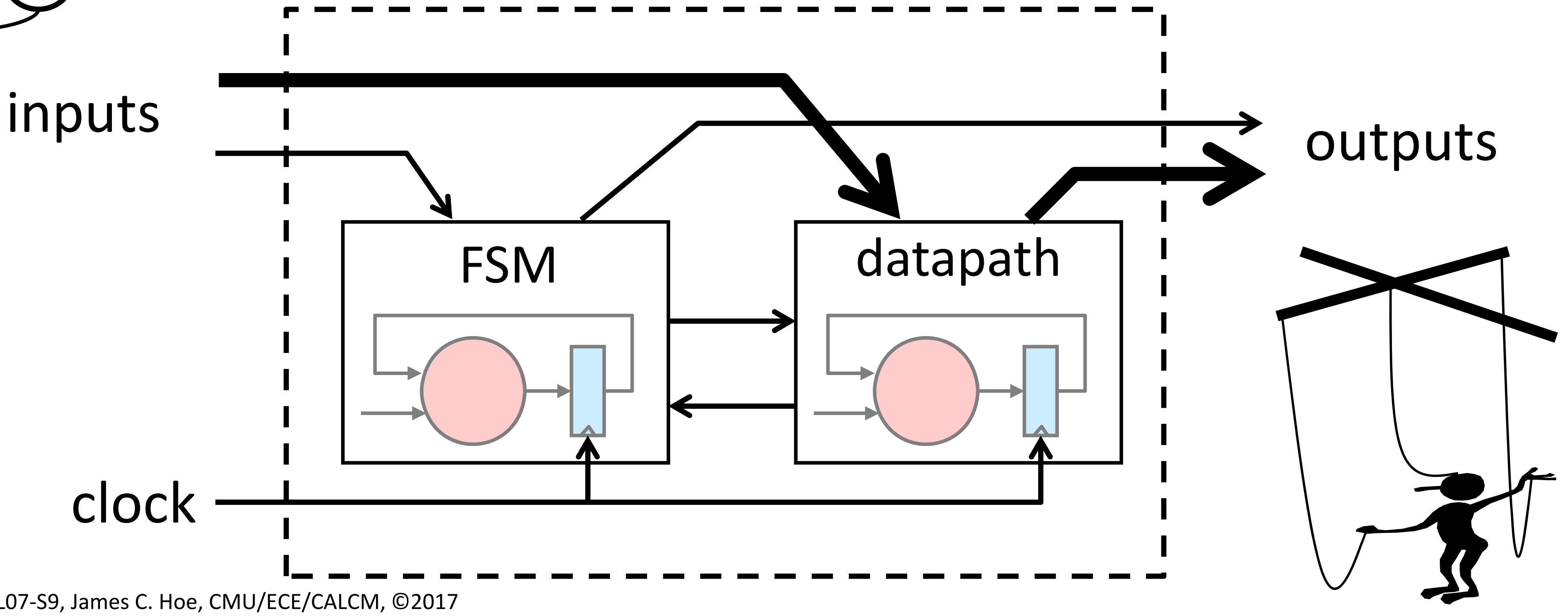


FSM-D



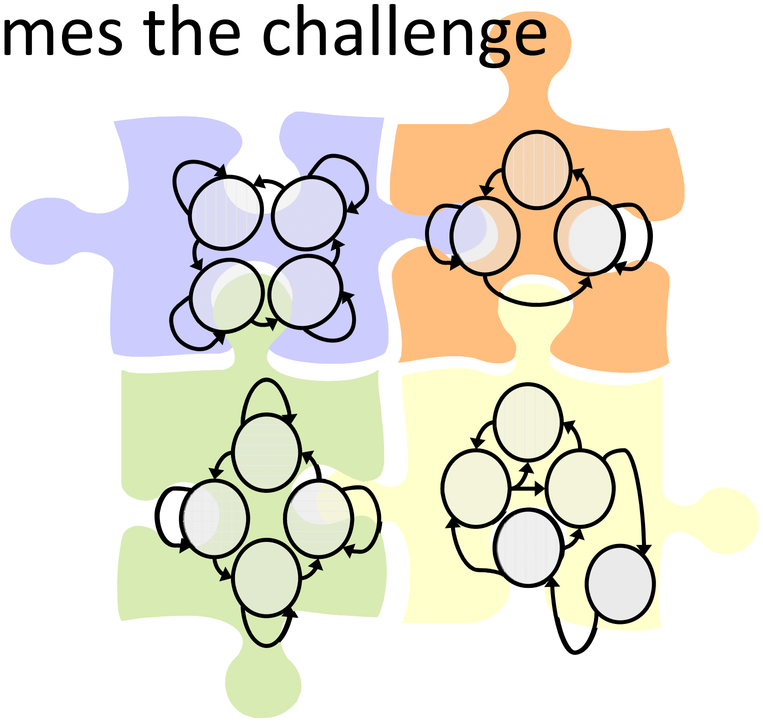
- datapath = "organized" combinational logic and registers to carry out computation (puppet)
- FSM = "stylized" combinational logic and registers for control and sequencing (puppeteer)





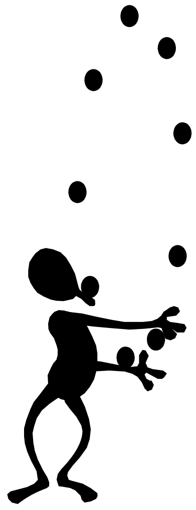
Cooperating FSM-Ds

- Partitioning large design into manageable chunks
  - natural decomposition by functionalities
  - inherent concurrency and replications
- Correct decomposition leads to simpler parts but coordination of the parts becomes the challenge
  - synchronization: having two FSM-Ds in the right state at the right time
  - communication: exchange information between FSM-D (requires synchronization)



Crux of the HW Design Difficulty

- We design FSM-Ds separately
  - liable to forget what one machine is doing when focusing on another
- No language support for coordination
  - no explicit way to say how state transitions of two FSMs must be related
- Coordination hardcoded into design implicitly
  - leave little room for automatic optimization
  - hard to localize design changes
  - (unless decoupled using request/reply-style handshakes)



"RTL" vs Schematics

- Same design abstraction
  - synchronous state, combinational next-state logic
  - hierarchy of modules with ports
- So why HDL more productive
  - textual description is easier, more compact
  - contemporary development in logic optimization (especially combinational)
  - procedural description of combinational logic
  - adopted PL know-hows: types, structs, operators, parameters, . . .
  - behavioral testbench

Better but not fundamentally different

Verilog an RTL language?

Verilog is not RTL

- Verilog in essence
  - a multithreaded programming language +
  - modeled time +
  - scheduling queue +
  - modules and ports
- Verilog describes how a module behaves, not its construction
  - no notion of “combinational” vs “sequential” logic
  - no notion of a register or even of a clock
  - perfectly happy describing non-hardware

Verilog Synthesis is Interpretation

```
// Ex 1
always@(a or b)
  if (a)
    c = b;
```

```
// Ex 2
always@(a)
  if (a)
    c = b;
  else
    c = 0;
```

```
// Ex 3
always@(a or b)
  if (a)
    c = b;
  else
    c = 0;
```

```
// Ex 4
always@(a or b)
begin
  c = 0;
  if (a)
    c = b;
end
```

```
// Ex 5
always@(a)
  if (a)
    c = b;
  else
    c = 0;

always@(b)
  if (a)
    c = b;
  else
    c = 0;
```

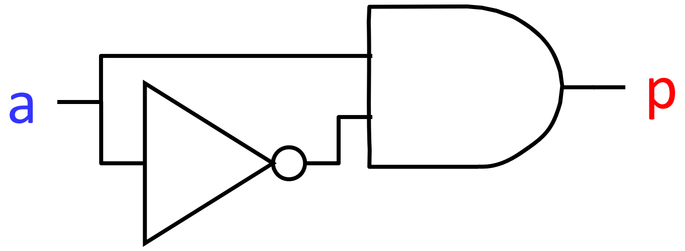
- All have well-defined behaviors
- According to Verilog semantics, c depends combinationally on a and b in Ex 3, 4 and 5
- Verilog doesn't say they are "combinational" or they are synthesizable

Synthesizable Verilog

- Verilog becomes an RTL language and becomes synthesizable only when used in a stylized way dictated by the synthesis tool
- So called "synthesizable subset" is really a different language
- Difficult even to define what is "correct" synthesis with respect to simulation behavior

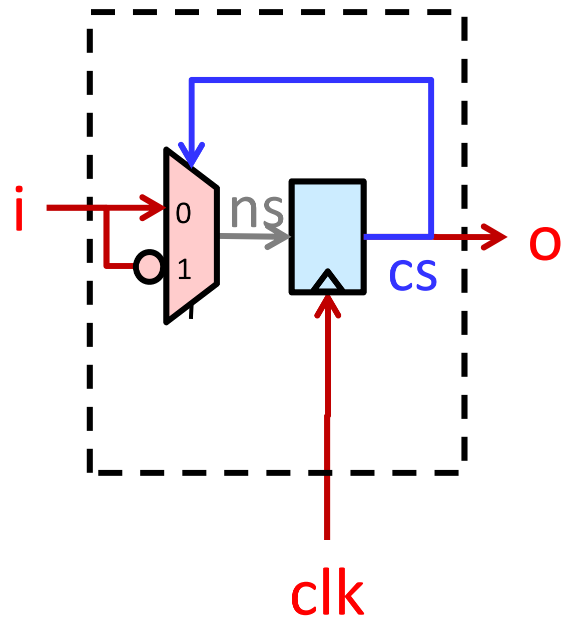
```
always@(a)
   p = a & (!a);
```

=?



Is p combinational?

RTL by Discipline



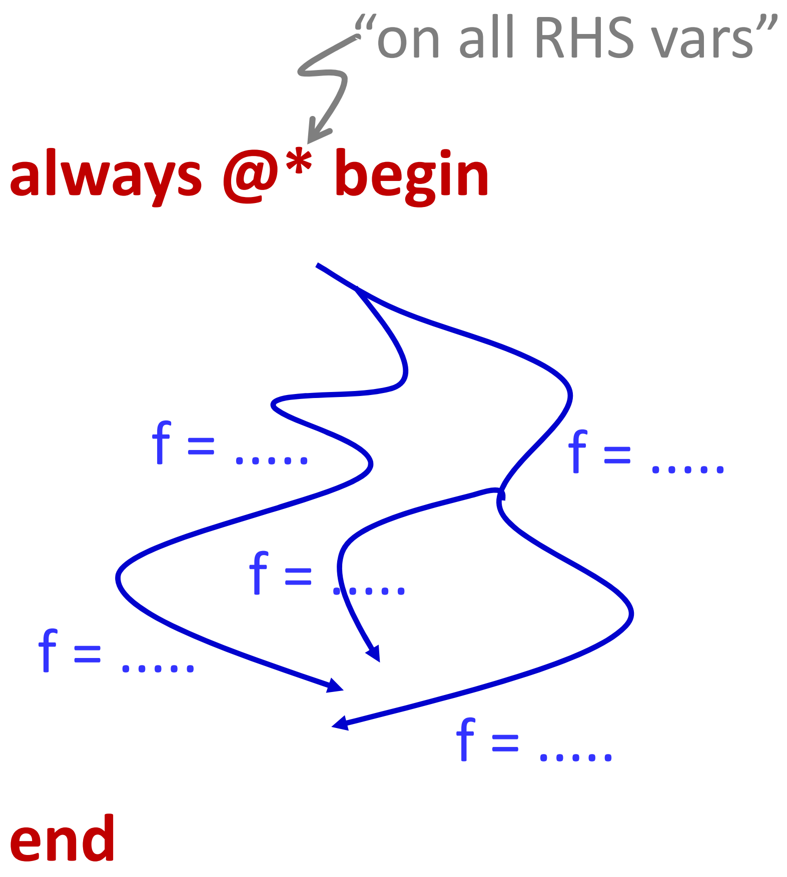
```
module contrived(input i, clk,
                 output o);
  reg cs; // sequential
  reg ns; // combinational

  assign o = cs;

  always @ ( i or cs)
      if (cs) ns = ~i;
      else    ns = i;

  always @ ( posedge clk ) begin
         cs <= ns;
  end

endmodule
```

combinational

synch. sequential

Crib sheet: Combinational "always"

- f must be assigned in all possible control paths; use "blocking" assigns
- f can depend on f only if f has been assigned
- repeated assigns to f okay; the last one holds
- f cannot be assigned in any other process
- multiple LHS vars okay; all rules above apply

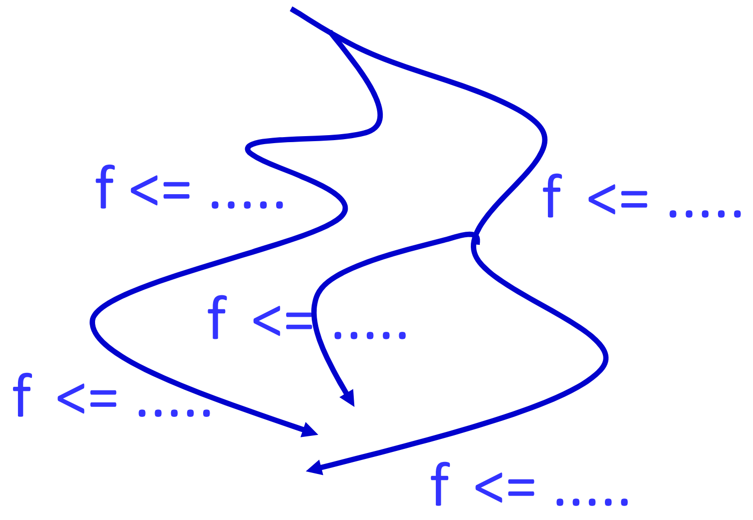


Use continuous assigns for simple expressions

Crib sheet: Synchronous “always”

- use “non-blocking” assigns; effect of assign not visible until after all triggered processes are done
- f can depend on f; RHS f stays at starting value
- repeated assigns to f okay; the last one holds
- f cannot be assigned in any other process
- multiple LHS vars okay; all rules above apply

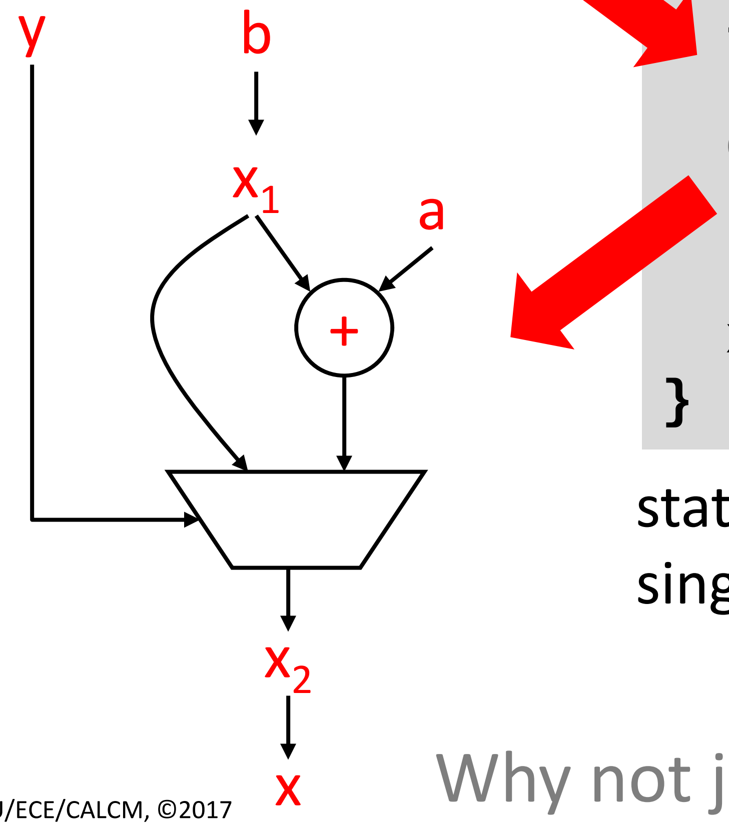
always @(posedge clk) begin



end

Procedural Block to Combinational

```
{
   x = b;
   if (y)
      x = x + a;
}
```



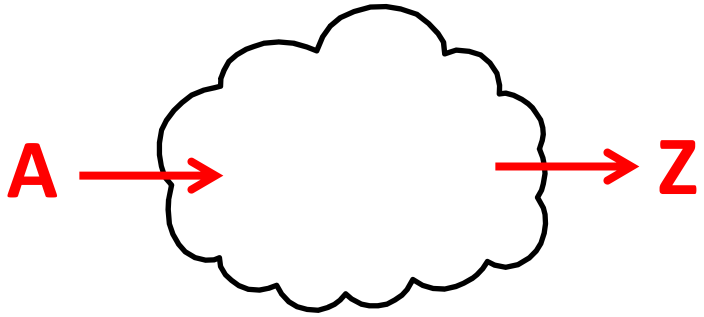
```
{
   x1 = b;

   if (y)
      x2 = x1 + a;
   else
      x2 = x1

   x = x2
}
```

static elaboration to single-assignment

Why not just: assign x=y?(b+a):b;

Why Procedural Block Wins

```
reg [5:0] Z;
wire [31:0] A;

integer i;

always @(A) begin

   Z=0;

   for(i=0;i<=31;i=i+1) begin
      if (A[i]) begin
         Z=Z+1;
      end
   end

end
```



Try saying this in continuous assign

Synthesizable Loops

- Loop must be statically unrolled, i.e.,
  - loop index must be integer type
  - loop index initial value must statically resolve to a constant
  - valid loop index operations are +, -
  - the valid loop condition test must test against a static limit using relational operators (<, <=, ==, etc)
- Precise limitations vary by tools and versions

Through static elaboration, even bounded-recursion should be okay

Don’t try this at home

```
module fib( output [7:0] z, input [7:0] n );

   function [7:0] recur;
      input [7:0] n;
      begin
         if (n==0)
            recur=0;
         else if (n==1)
            recur=1;
         else
            recur=recur(n-1)+recur(n);
      end
   endfunction // recur

   assign z = recur(n);

endmodule
```

This is correct Verilog; does it synthesize?

Modern Synthesis Plays Tricks

- Standard compiler passes
  - constant propagation
  - common-subexpression elimination
  - deadcode elimination
  - strength reduction
- Bit-wise logic also Boolean optimized

Most of the time, good enough to write understandable expressions with clear intent

Reserve Black Arts for Timing Closure

```
wire a, b, c, d;
wire [3:0] sel;
reg z;

always@* begin
   z = 0;
   if (sel[0]) z = a;
   if (sel[1]) z = b;
   if (sel[2]) z = c;
   if (sel[3]) z = d;
end
```

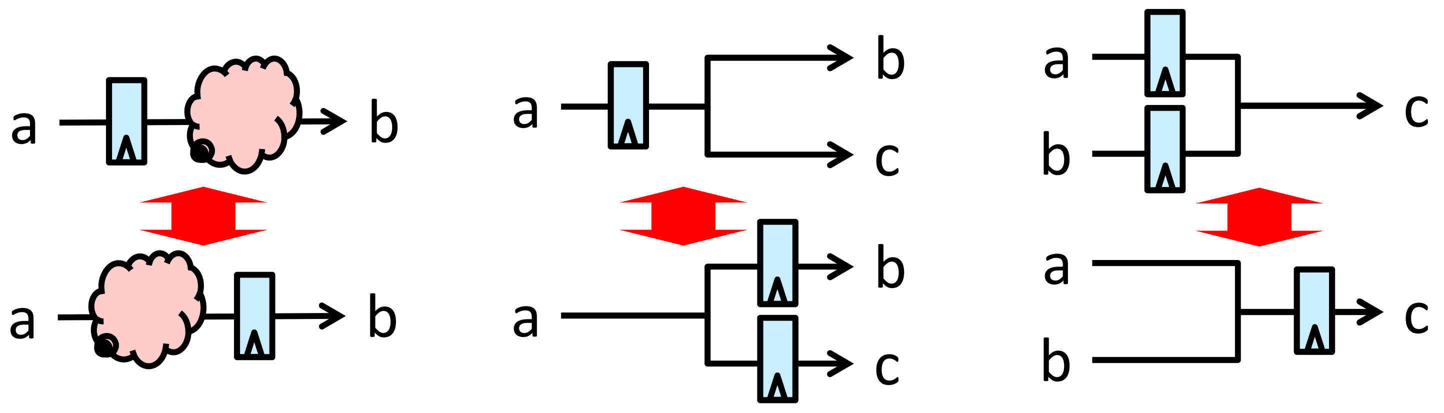
$\neq$

```
wire a, b, c, d;
wire [3:0] sel;
reg z;

always@* begin
   z = 0;
   if (sel[3])
      z = d;
   else if (sel[2])
      z = c;
   else if (sel[1])
      z = b;
   else if(sel[0])
      z = a;
end
```

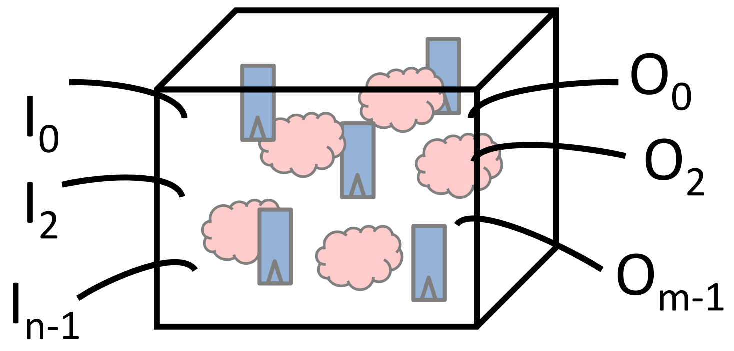
Tool specific

Retiming

- Local transformations



- Preserves I/O relationships
- Tools use retiming
  - balance critical paths
  - absorb FFs into hard macros



```
always@(posedge clk) begin
   a1<=a; b1<=b;
   a2<=a1; b2<=b1;
   c<=a2*b2;
end
```

Some Best Practices

- #1 Read the style guides
- Develop a mental model of synthesis
- Know what optimizations tools do and don't do
- Know the special inference rules: FF, RAM, FSM
- Know pragmas to control the tool
- Read the synthesis reports (inference and warnings)

- Have a good naming convention
- Keep combinational and sequential distinct
- Use modules and hierarchies
- Embed assertions

FPGA Inference Extra Gotcha's

- FPGA Macros (especially RAM and DSP)
  - coarse functions and structures
  - some powerful but arbitrarily specific features
  - penalty is too huge to not get it right
- Very specific guideline for inferring hard macros
  - hard macros only does what it does
  - tools cannot recognize all "functional equivalent" descriptions

Always check inference report to see if you got what you expected

Just on Flip-Flops

- Use asynch set or reset
  - ⇒ not all FFs have async reset; prevents DSP retiming
- Use both set and reset
  - ⇒ no FF has this; emulated externally with LUTs
- Use set and reset operationally
  - ⇒ set/reset cannot use special global lines
- Active-low set/reset and enables
  - ⇒ need LUTs to turn active-high

How could you know this?

- BRAM cannot be used if combinational read
- Shift registers can be made out of LUTs

  BUT! no set/reset and can't read middle bits
- Registers will retime into multiplier and DSP (if no asynch reset)
- Use "initial" for power-on reset
- Timing analysis doesn't do "latches"
- Many, many more like this. . .

Read the inference report and warnings

RTMF!

Why is Structural Design Hard?

Reason #1: Low Level of Abstraction

- When writing a "high-level" program,
  - specify functionality using abstract constructs: loops, double, '+', . . .
  - can ignore non-functional details: timing, data representation, data placement, ...
  - even assembly programming is in a virtualized realm
- When designing RTL
  - direct control of structure and operation
  - bit-level and cycle-level building blocks

Say anything you want and get what you want;
but takes a whole lot more work

Reason #2: Unrestricted Design Freedom

- Total freedom in how to realize a functionality

```
for(i=0; i<10; i=i+1)
     sum+=i;
```

  – is `sum` an integer or float, exactly how many bits?
  – how to ‘`+`’? ripple, c-select, c-look-ahead . . .
  – do ‘`+`’ one at a time or all at once?
  – where to store `i` and `sum`? may be not at all

- Can choose anything in between as “cheap as possible” (area or energy?) and as “fast as possible” (latency or throughput?)

Find the one right design in a haystack of bad ones

Reason #3: Massive Concurrency

- Extremely high degree of concurrency
- Extremely fine granularity of concurrency
- Explicit everything (synchronization, communication, . . .); poor language abstraction
- Nothing is disallowed (irregular parallelism, mixing parallelism, . . .)

Everything that makes parallel programming hard is much worse in hardware design

Reason #4: Because it is hard

- Only design hardware as the last resort
  - if not performance critical, just write software
  - if not energy (or cost or weight) constrained, run on faster processors, bigger computers
  - if embarrassingly SIMD parallel, run on GPUs
- Unfortunately, what makes hardware design hard (#1~#3) is what makes hardware the ultimate weapon when all else fails

How to make hardware design easier without making the results less efficient?

Parting Thoughts

- Know your tools and practice your craft
- Structural RTL design operates explicitly at the bit- and cycle-level
  - ➡ high workload requires large designs to be broken hierarchically into manageable modules
  - ➡ coordinating concurrent operations of distributed modules introduces its own "high" complexity
- Need better hardware design methodologies (without taking away hardware's advantages)