ECE 4514
Digital Design II
Spring 2008

Lecture 13: Logic Synthesis

A *Tools/Methods* Lecture

Patrick Schaumont

Second half of Digital Design II

9	10-Mar-08	L13	(T) Logic Synthesis	PJ2
	13-Mar-08	L14	(D) FPGA Technology	
10	18-Mar-08		No Class (Instructor on Conference)	PJ3
	20-Mar-08	L15	(D) Control in FPGA	
11	25-Mar-08	L16	(T) Memory and Datapath in FPGA	PJ4
	27-Mar-08	L17	(D) Design of a Reed Solomon Coder	
12	01-Apr-08	L18	(T) Optimizing Area	
	03-Apr-08	L19	(T) Optimizing Speed	
13	08-Apr-08	L20	(D) Timing Analysis and Simulation	PJ5
	10-Apr-08	L21	(L) Tasks and Functions, PLI	
14	15-Apr-08	L22	(T) Testing and Verification	
	17-Apr-08	L23	(D) Design of a RISC Processor	
15	22-Apr-08		Project Discussion/Presentation	
	24-Apr-08		Project Discussion/Presentation	
16	29-Apr-08	R2	Review Lecture	
	05-May-08		Final Exam	

How to write Verilog for FPGA Synthesis

Week	Date	Lecture	Topic	Project
9	10-Mar-08	L13	(T) Logic Synthesis	PJ2
	13-Mar-08	L14	(D) FPGA Technology	
10	18-Mar-08		No Class (Instructor on Conference)	PJ3
	20-Mar-08	L15	(D) Control in FPGA	
11	25-Mar-08	L16	(T) Memory and Datapath in FPGA	PJ4
	27-Mar-08	L17	(D) Design of a Reed Solomon Coder	
12	01-Apr-08	L18	(T) Optimizing Area	
	03-Apr-08	L19	(T) Optimizing Speed	
13	08-Apr-08	L20	(D) Timing Analysis and Simulation	PJ5
	10-Apr-08	L21	(L) Tasks and Functions, PLI	
14	15-Apr-08	L22	(T) Testing and Verification	
	17-Apr-08	L23	(D) Design of a RISC Processor	
15	22-Apr-08		Project Discussion/Presentation	
	24-Apr-08		Project Discussion/Presentation	
16	29-Apr-08	R2	Review Lecture	
	05-May-08		Final Exam	

How to Optimize the Synthesis Results

Week	Date	Lecture	Topic	Project
9	10-Mar-08	L13	(T) Logic Synthesis	PJ2
	13-Mar-08	L14	(D) FPGA Technology	
10	18-Mar-08		No Class (Instructor on Conference)	PJ3
	20-Mar-08	L15	(D) Control in FPGA	
11	25-Mar-08	L16	(T) Memory and Datapath in FPGA	PJ4
	27-Mar-08	L17	(D) Design of a Reed Solomon Coder	
12	01-Apr-08	L18	(T) Optimizing Area	
	03-Apr-08	L19	(T) Optimizing Speed	
13	08-Apr-08	L20	(D) Timing Analysis and Simulation	PJ5
	10-Apr-08	L21	(L) Tasks and Functions, PLI	
14	15-Apr-08	L22	(T) Testing and Verification	
	17-Apr-08	L23	(D) Design of a RISC Processor	
15	22-Apr-08		Project Discussion/Presentation	
	24-Apr-08		Project Discussion/Presentation	
16	29-Apr-08	R2	Review Lecture	
	05-May-08		Final Exam	

How to write better Testbenches

Week	Date	Lecture	Topic	Project
9	10-Mar-08	L13	(T) Logic Synthesis	PJ2
	13-Mar-08	L14	(D) FPGA Technology	
10	18-Mar-08		No Class (Instructor on Conference)	PJ3
	20-Mar-08	L15	(D) Control in FPGA	
11	25-Mar-08	L16	(T) Memory and Datapath in FPGA	PJ4
	27-Mar-08	L17	(D) Design of a Reed Solomon Coder	
12	01-Apr-08	L18	(T) Optimizing Area	
	03-Apr-08	L19	(T) Optimizing Speed	
13	08-Apr-08	L20	(D) Timing Analysis and Simulation	PJ5
	10-Apr-08	L21	(L) Tasks and Functions, PLI	
14	15-Apr-08	L22	(T) Testing and Verification	
	17-Apr-08	L23	(D) Design of a RISC Processor	
15	22-Apr-08		Project Discussion/Presentation	
	24-Apr-08		Project Discussion/Presentation	
16	29-Apr-08	R2	Review Lecture	
	05-May-08		Final Exam	

A couple more design examples

Week	Date	Lecture	Topic	Project
9	10-Mar-08	L13	(T) Logic Synthesis	PJ2
	13-Mar-08	L14	(D) FPGA Technology	
10	18-Mar-08		No Class (Instructor on Conference)	PJ3
	20-Mar-08	L15	(D) Control in FPGA	
11	25-Mar-08	L16	(T) Memory and Datapath in FPGA	PJ4
	27-Mar-08	L17	(D) Design of a Reed Solomon Coder	
12	01-Apr-08	L18	(T) Optimizing Area	
	03-Apr-08	L19	(T) Optimizing Speed	
13	08-Apr-08	L20	(D) Timing Analysis and Simulation	PJ5
	10-Apr-08	L21	(L) Tasks and Functions, PLI	
14	15-Apr-08	L22	(T) Testing and Verification	
	17-Apr-08	L23	(D) Design of a RISC Processor	
15	22-Apr-08		Project Discussion/Presentation	
	24-Apr-08		Project Discussion/Presentation	
16	29-Apr-08	R2	Review Lecture	
	05-May-08		Final Exam	

Projects - A Major Components in DD-II

- Projects account for 45% of the course grade
- Project 2 and 3 are individual projects
  - Development time one week
- Project 4 and 5 are team-based projects
  - Teamsize 2 persons
  - Development time 2 weeks
  - End with a final presentation in week 15
- Projects are not Homeworks
  - Projects will require a more substantial design effort.
  - Projects will address open-ended problems
- Deadlines are important
  - Remember - no late policy

Projects: A major part of DD-II

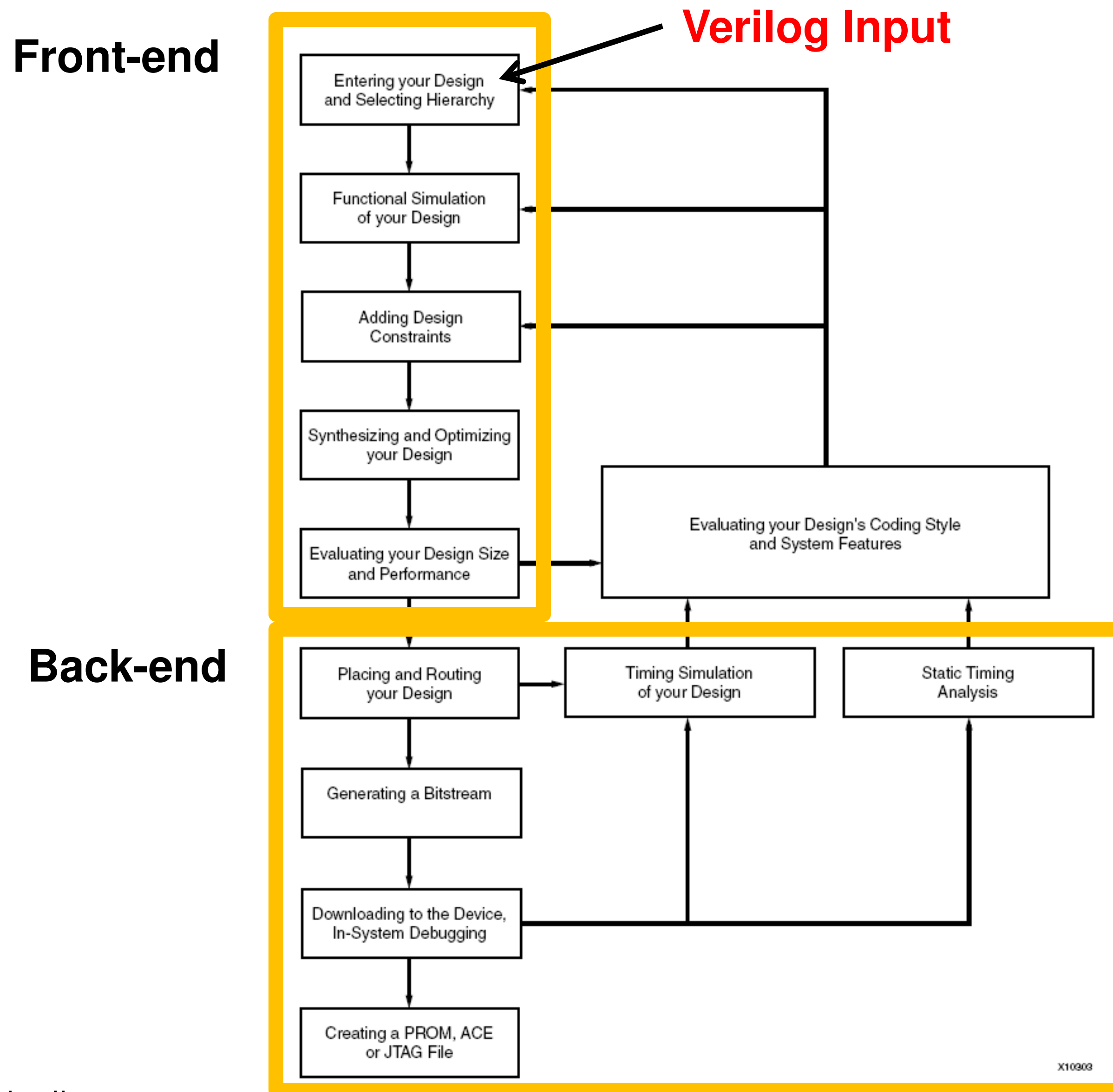
Week	Date	Lecture	Topic	Project
9	10-Mar-08	L13	(T) Logic Synthesis	PJ2
	13-Mar-08	L14	(D) FPGA Technology	
10	18-Mar-08		No Class (Instructor on Conference)	PJ3
	20-Mar-08	L15	(D) Control in FPGA	
11	25-Mar-08	L16	(T) Memory and Datapath in FPGA	PJ4
	27-Mar-08	L17	(D) Design of a Reed Solomon Coder	
12	01-Apr-08	L18	(T) Optimizing Area	
	03-Apr-08	L19	(T) Optimizing Speed	
13	08-Apr-08	L20	(D) Timing Analysis and Simulation	PJ5
	10-Apr-08	L21	(L) Tasks and Functions, PLI	
14	15-Apr-08	L22	(T) Testing and Verification	
	17-Apr-08	L23	(D) Design of a RISC Processor	
15	22-Apr-08		Project Discussion/Presentation	
	24-Apr-08		Project Discussion/Presentation	
16	29-Apr-08	R2	Review Lecture	
	05-May-08		Final Exam	

Today

- Logic Synthesis
  - Steps to convert Verilog Code into FPGA bitstream
  - Correspondence between Verilog and Hardware
  - Palnitkar Chapter 14

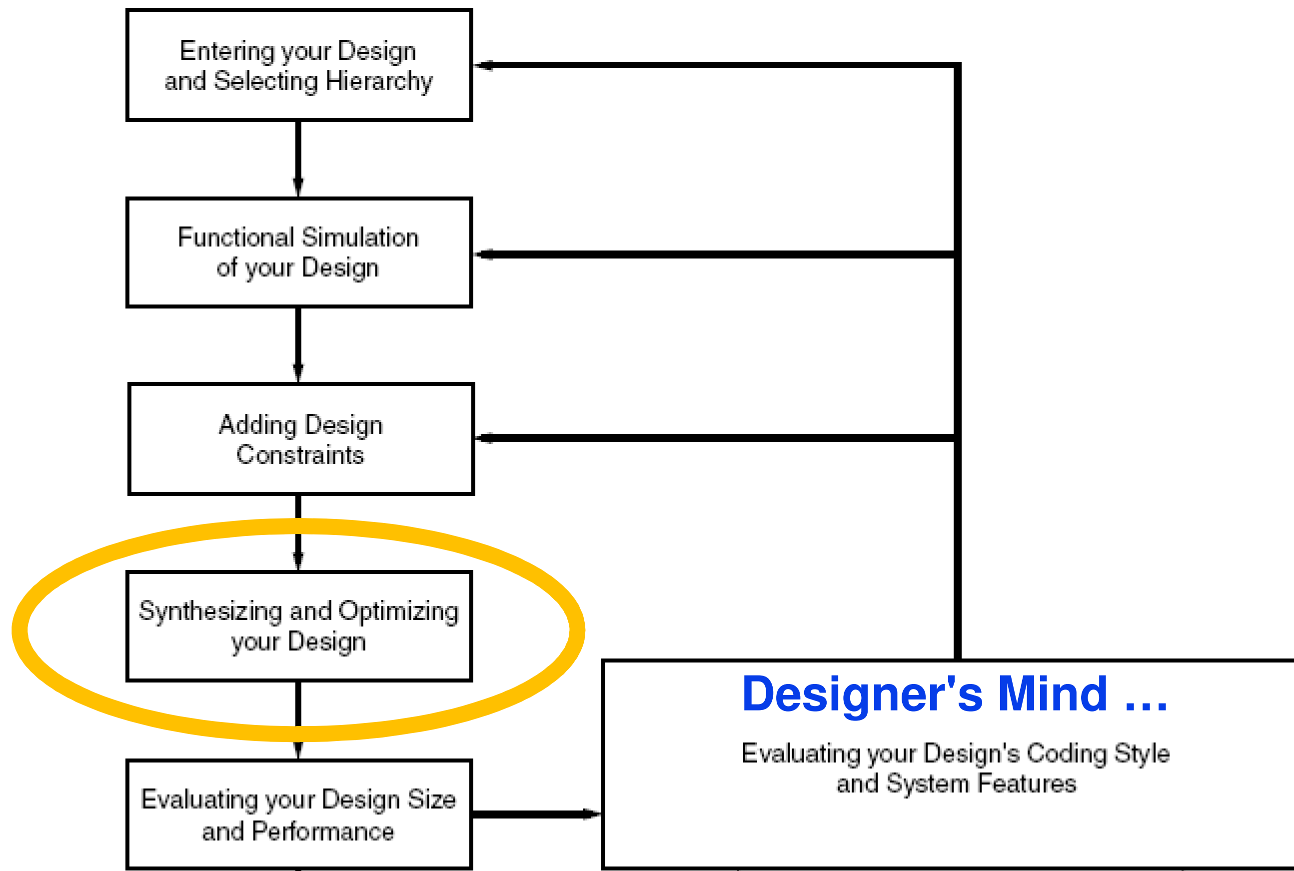
- Thursday: The FPGA fabric
  - A look at the features on an FPGA

FPGA Design Flow



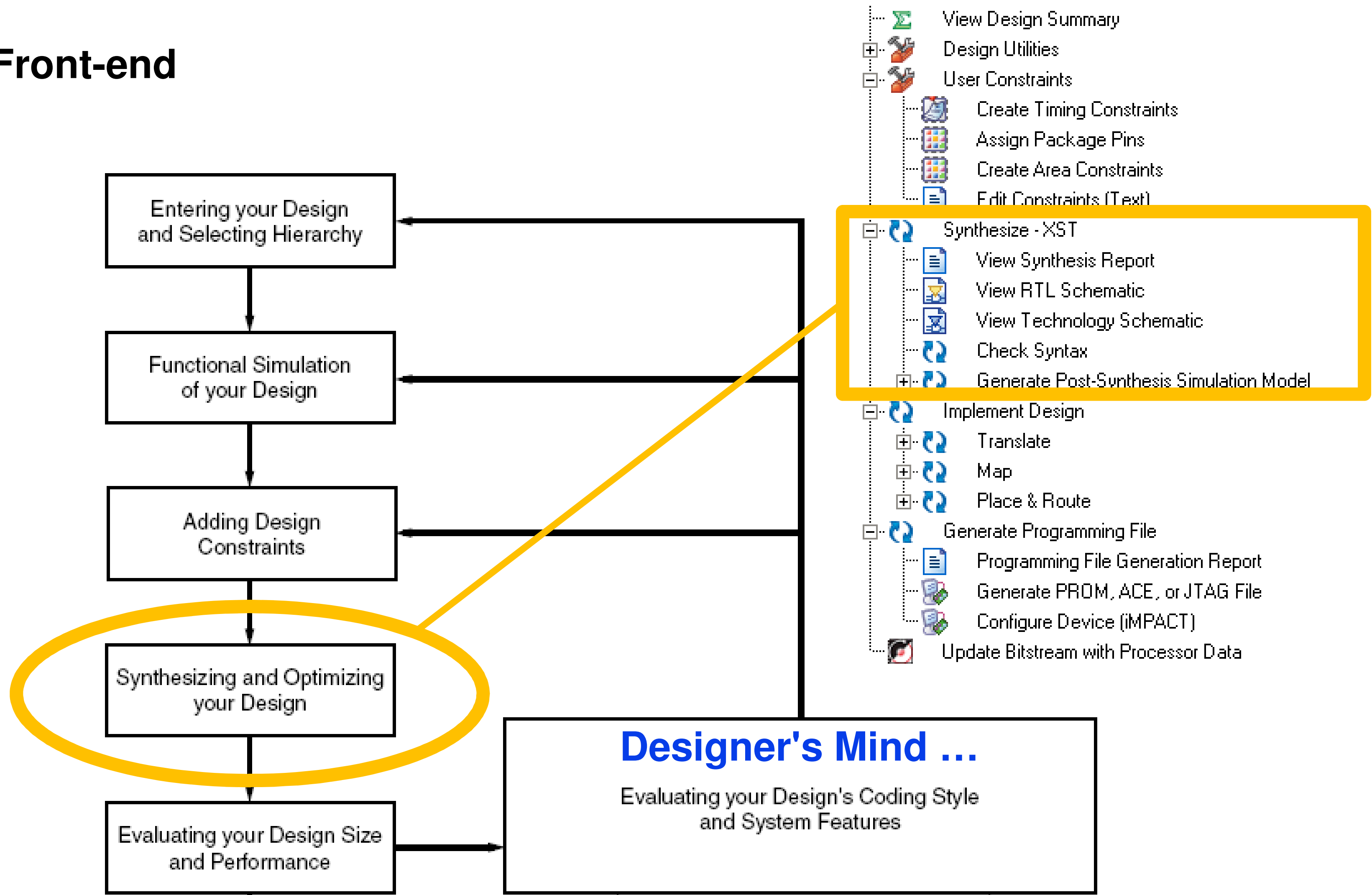
Logic Synthesis in the (FPGA) design flow

Front-end



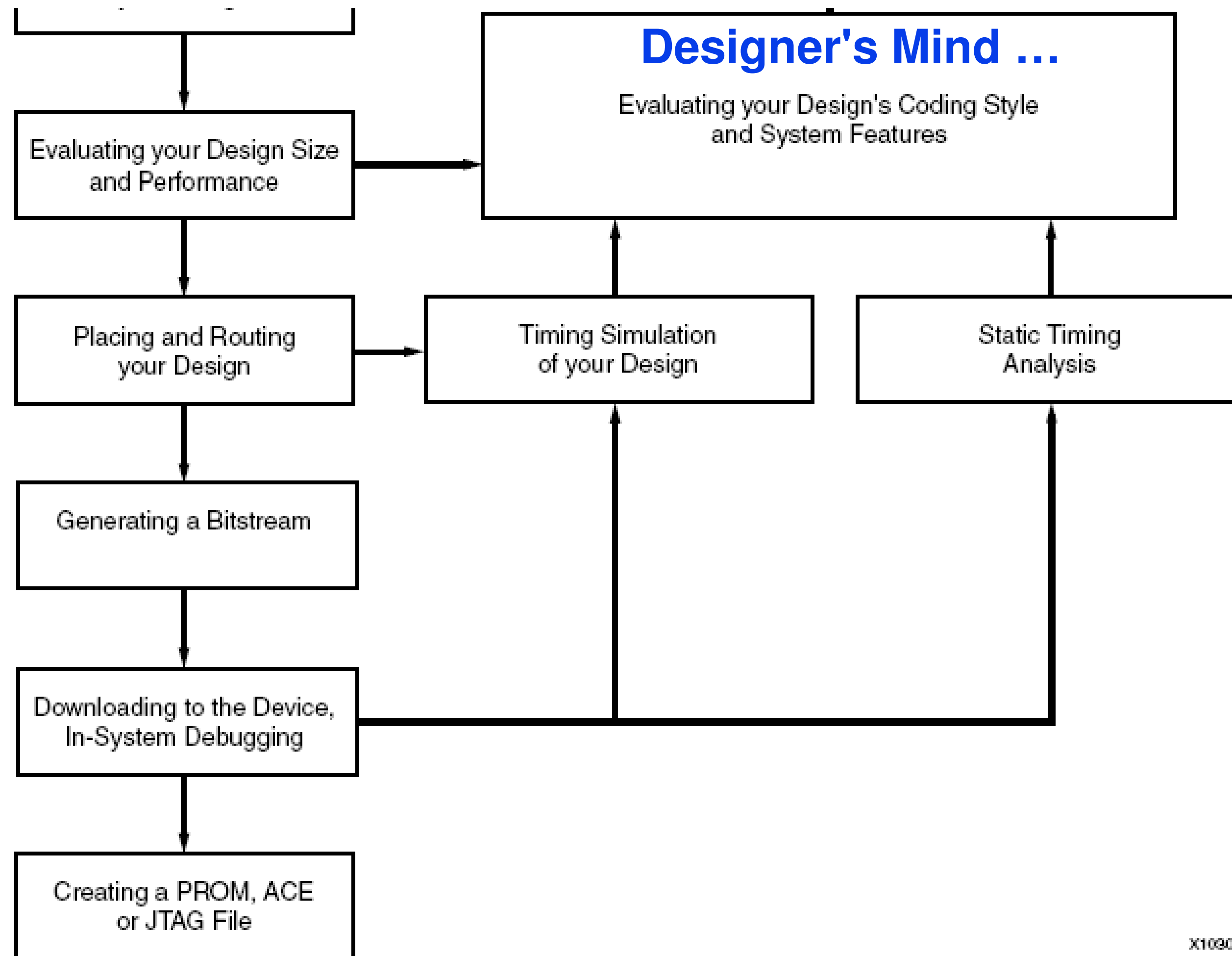
Logic Synthesis in the (FPGA) design flow

Front-end



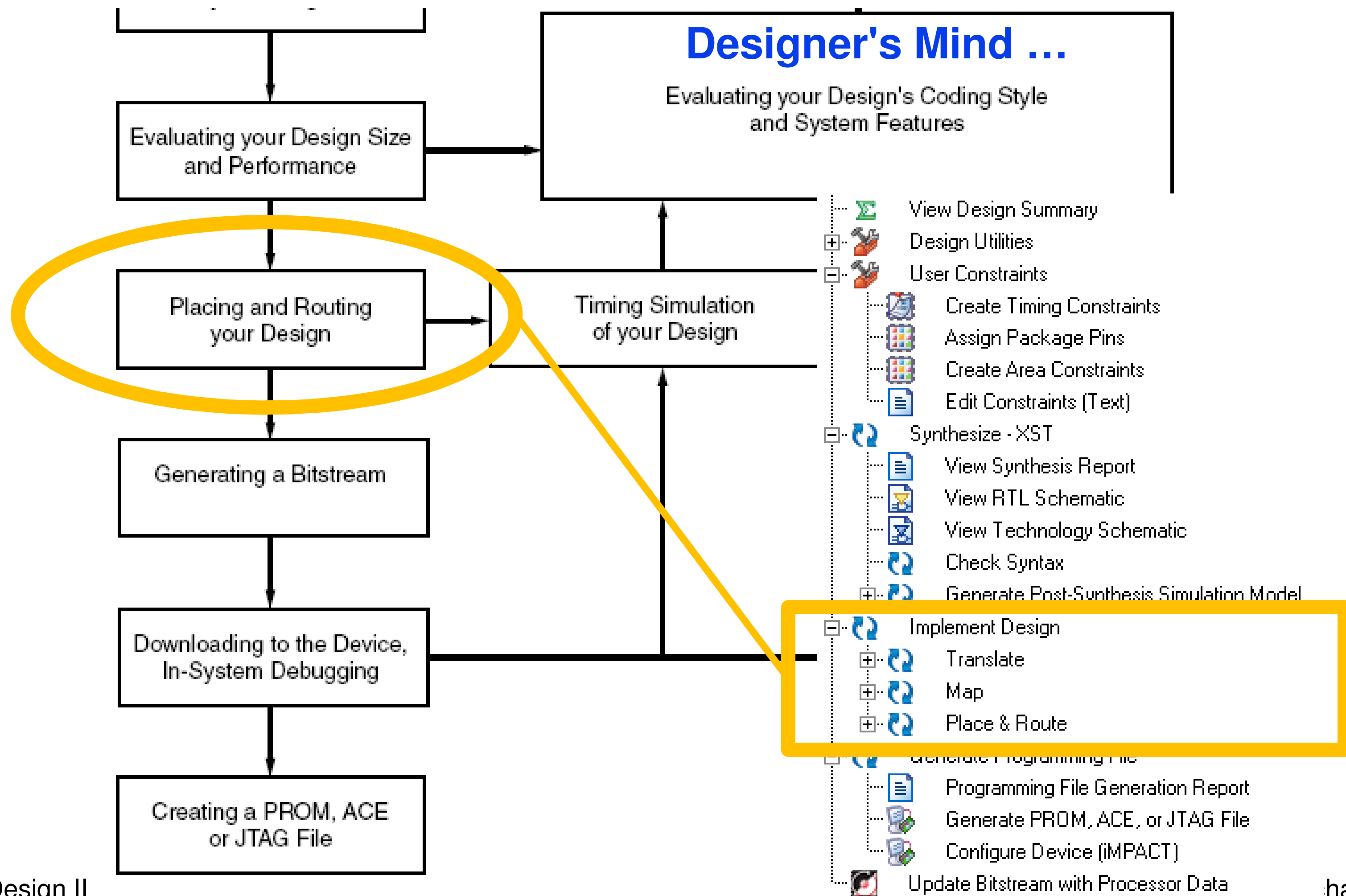
Logic Synthesis in the (FPGA) design flow

Back-end



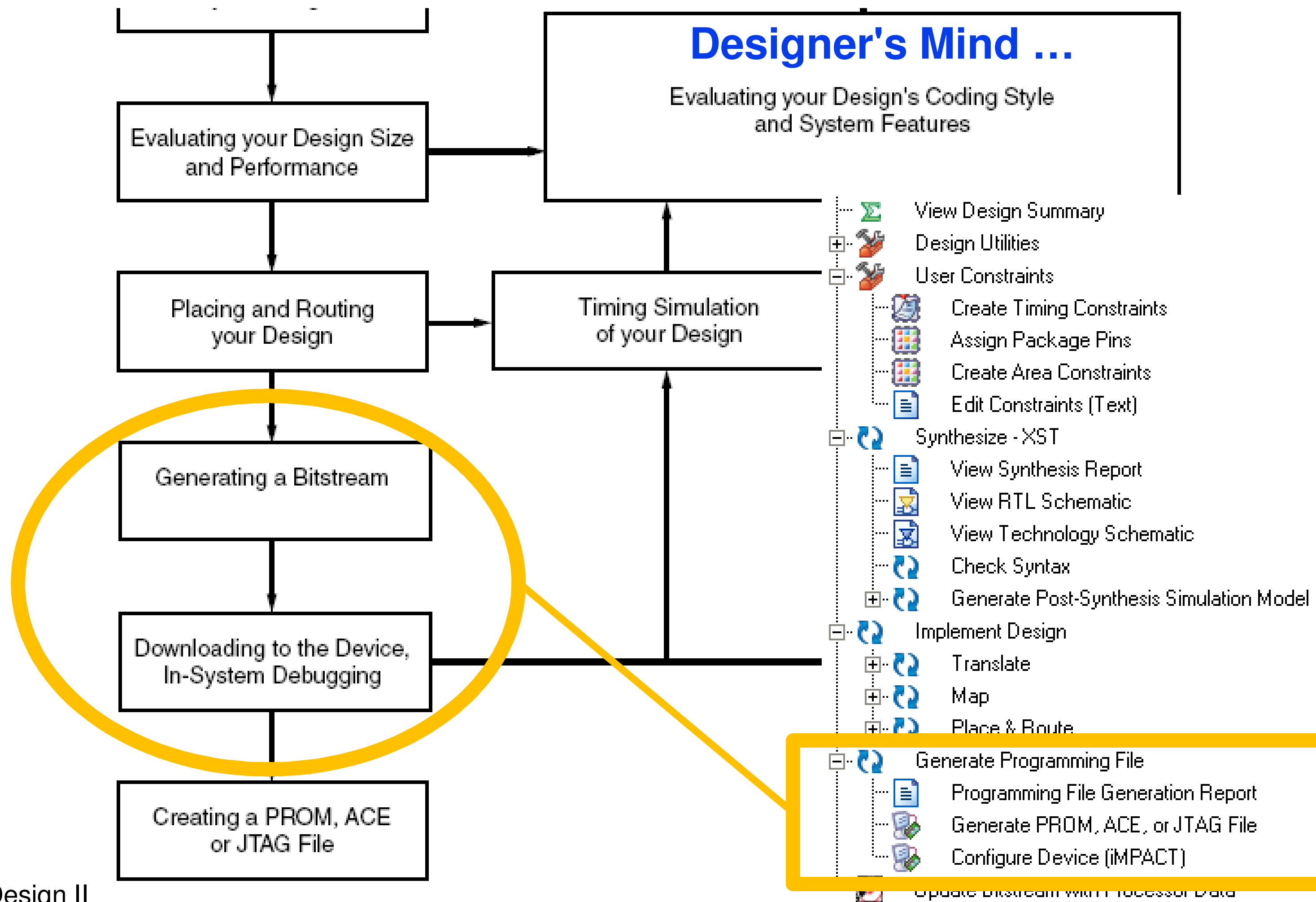
Logic Synthesis in the (FPGA) design flow

Back-end

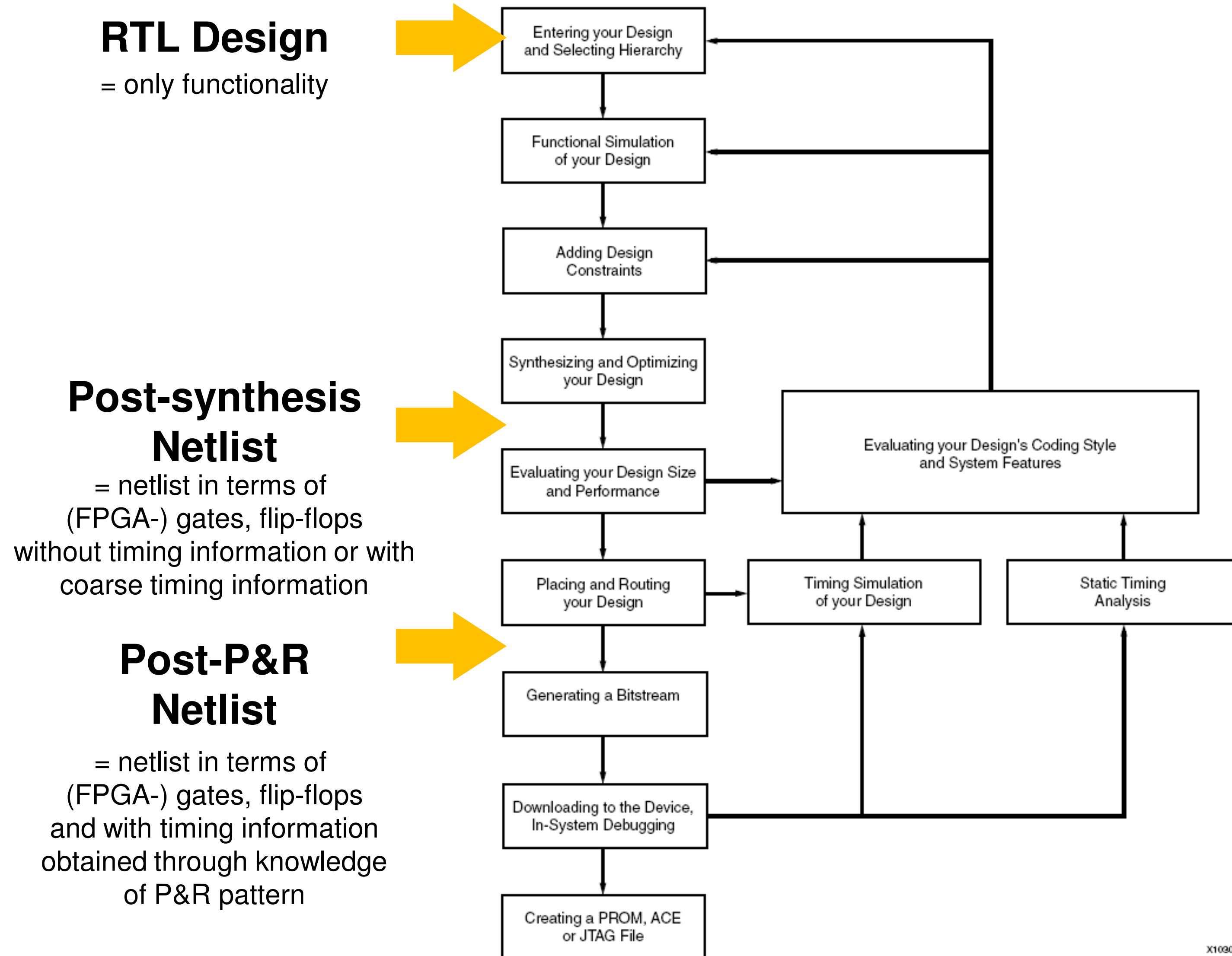


Logic Synthesis in the (FPGA) design flow

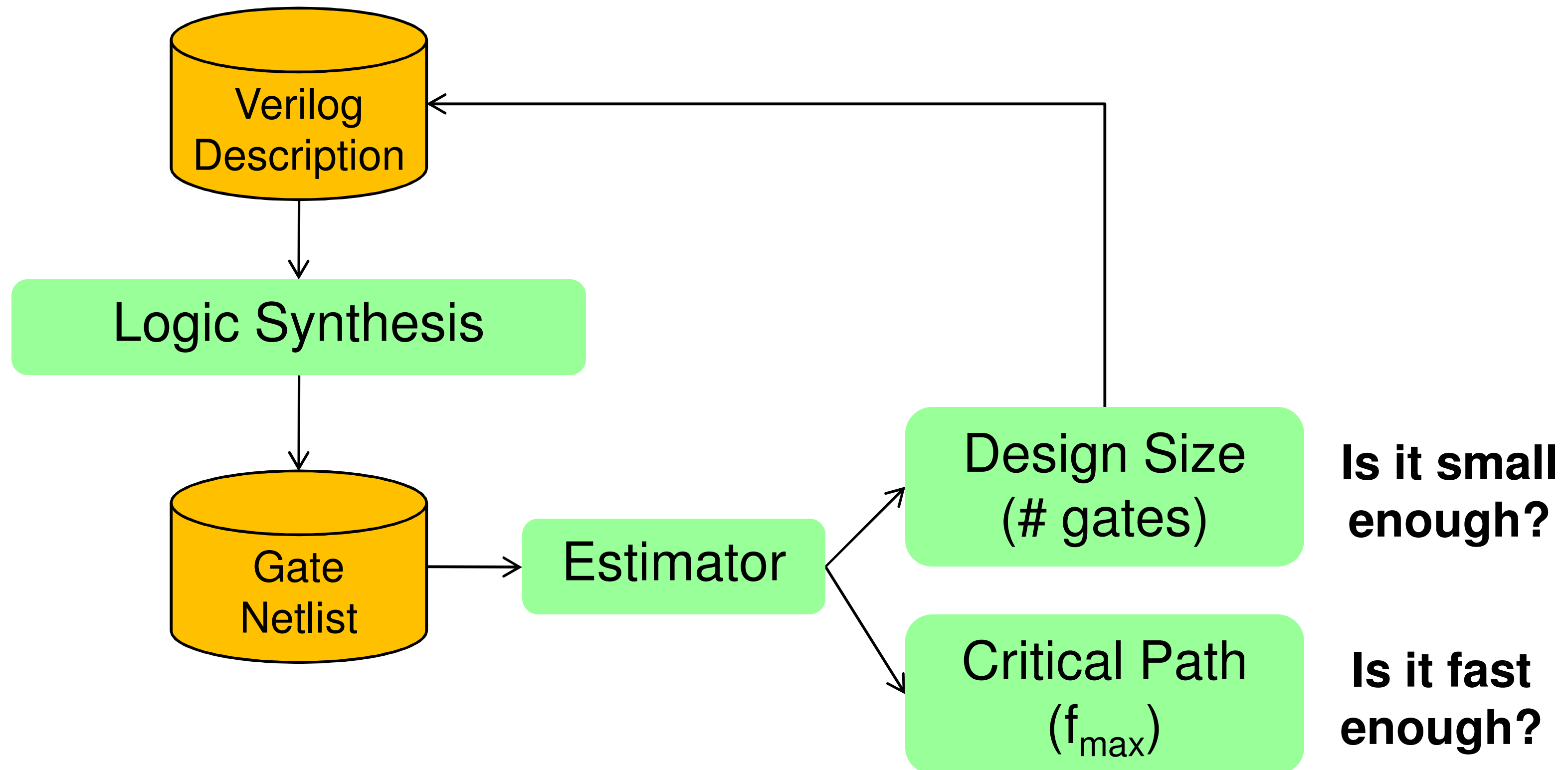
Back-end



Some Terms



How do we know synthesis quality is OK?

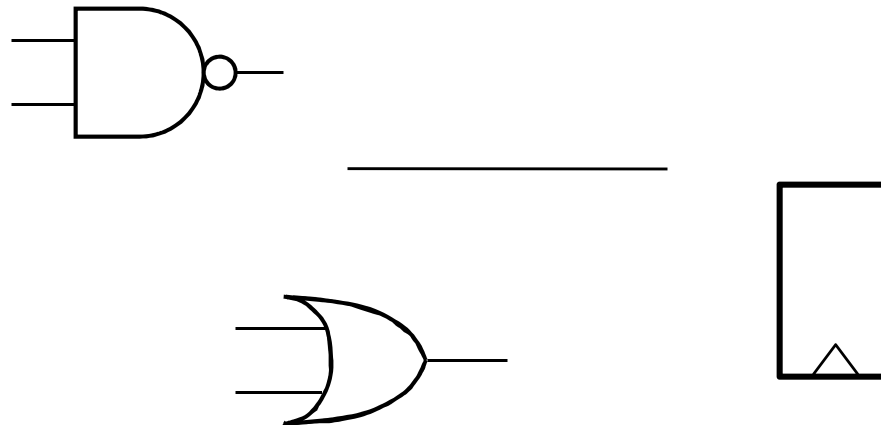


- Using ***estimators***, we obtain an early guess on the eventual performance of a design
- Estimators are built-in to synthesis tools
- Estimation is never 100% accurate until the final implementation is complete. However, more accuracy after P&R

How does Verilog map into Hardware ?

```
wire
reg
module
always @(posedge clk)
initial #15
```

Hardware Inference



Hardware Inference

- Unlike simulation, hardware inference is tool-dependent !
  - Not every Verilog construct which simulates will also synthesize
  - Different tools may use different translation techniques, what works in one tool may not in another
- There are some general rules to hardware inference which most tools will support
- We will describe these general rules, and occasionally highlight an FPGA-specific construct
  - This lecture: general synthesis guidelines
  - Next lecture: the FPGA fabric
  - Next-next lecture: designing FPGA controllers
  - Next-next-next lecture: FPGA datapaths and memories

Key Guide For HDL-based design on Xlx FPGA

3 / 246 100% Find

Bookmarks
Options

- Software Manuals
- Synthesis and Simulation Design Guide
  - About the Synthesis and Simulation Design Guide
  - Table of Contents
  - 1 Introduction to Synthesis and Simulation
  - 2 FPGA Design Flow
  - 3 General Recommendations for Coding Practices
  - 4 Coding for FPGA Flow
  - 5 Using SmartModels
  - 6 Simulating Your Design
  - 7 Design Considerations

XILINX®

Preface

About the Synthesis and Simulation Design Guide

This chapter (*About the Synthesis and Simulation Design Guide*) provides general information about this Guide, and includes:

- "Synthesis and Simulation Design Guide Overview"
- "Synthesis and Simulation Design Guide Design Examples"
- "Synthesis and Simulation Design Guide Contents"
- "Additional Resources"
- "Conventions"

Synthesis and Simulation Design Guide Overview

The *Synthesis and Simulation Design Guide* provides a general overview of designing Field Programmable Gate Arrays (FPGA) devices with Hardware Description Languages (HDLs). It includes design hints for the novice HDL user, as well as for the experienced user who is designing FPGA devices for the first time. Before using the *Synthesis and Simulation Design Guide*, you should be familiar with the operations that are common to all Xilinx tools.

The *Synthesis and Simulation Design Guide* does *not* address certain topics that are important when creating Hardware Description Language (HDL) designs, such as:

- Design environment
- Verification techniques
- Constraining in the synthesis tool
- Test considerations
- System verification

For more information, see your synthesis tool documentation.

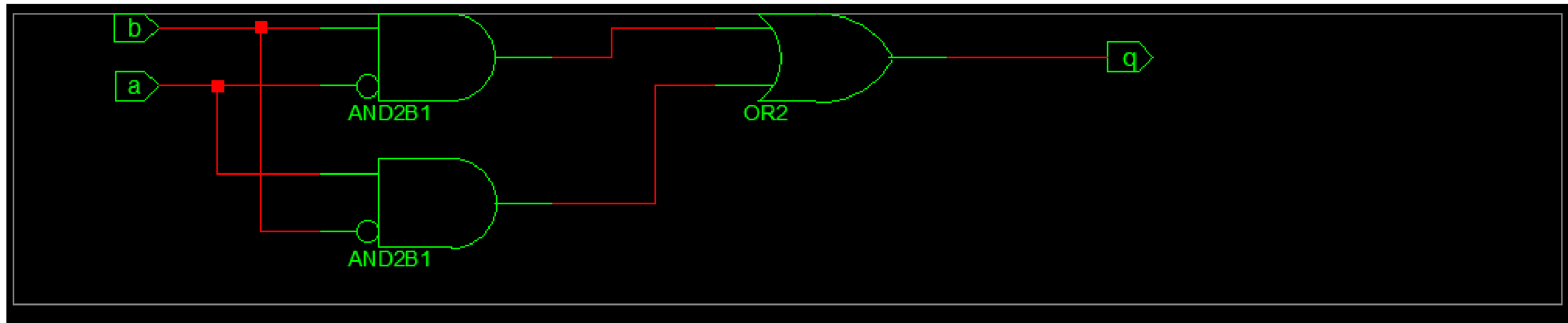
General Hardware Inference Rules

- Combinational Logic
  - Gates from dataflow
  - Gates from procedural blocks
  - Flow-control: if-then-else, for-loops
  - Pitfalls
- Sequential Logic
  - Registers vs wires
  - Registers vs Latches
  - Reset and initialization
- Constructs unsupported in synthesis
  - Initial Blocks
  - Delays

Combinational Logic

- Logic from Dataflow

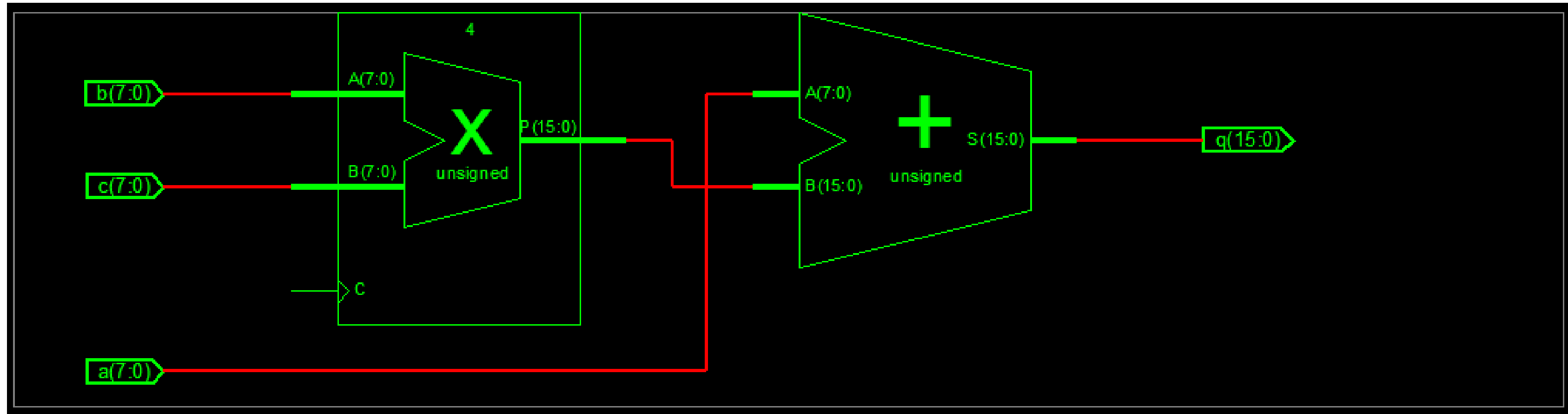
```
module andor(q, a, b);
  output q;
  input a, b;

  assign q = (a & ~b) | (~a & b);

endmodule
```



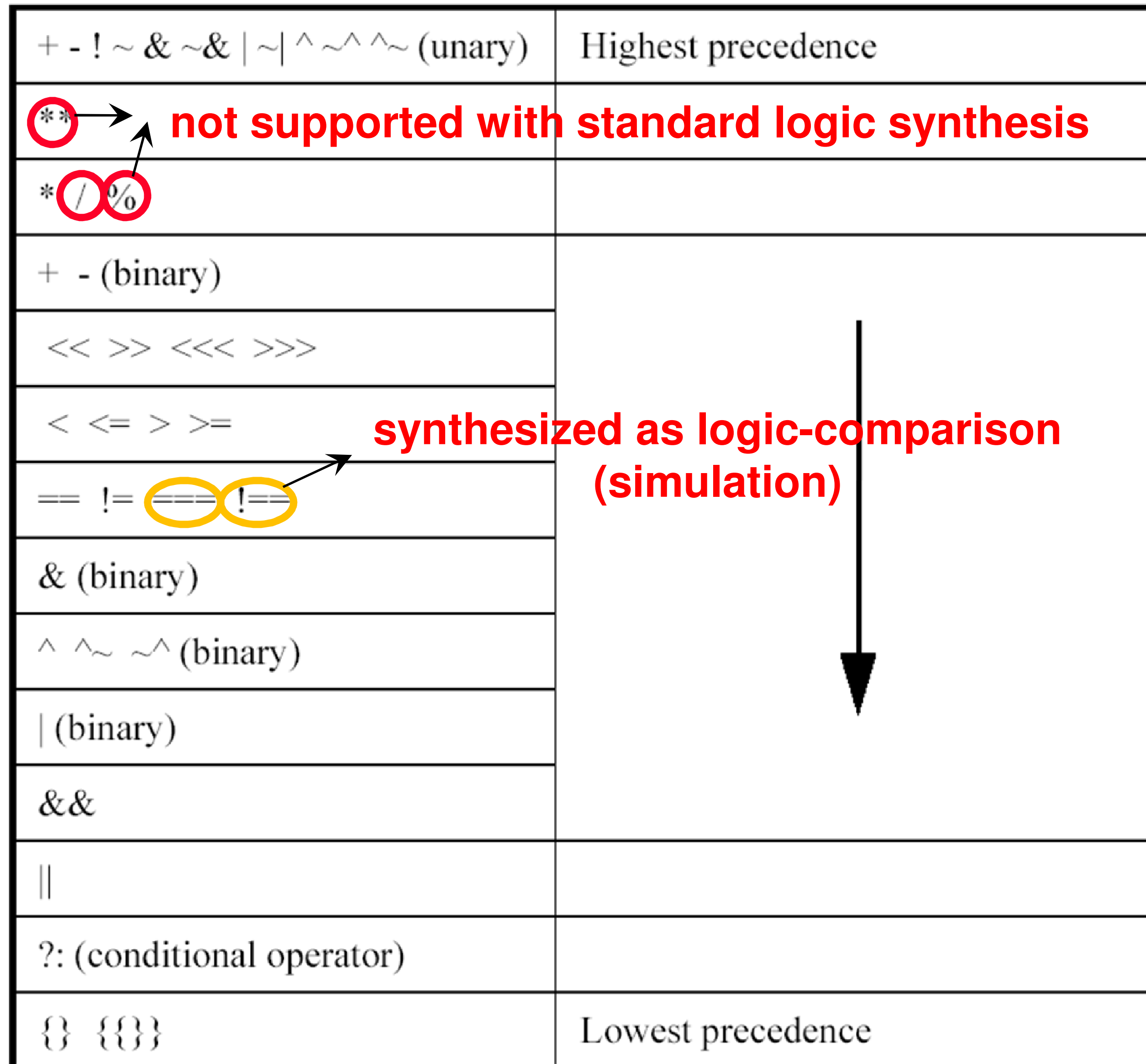
Logic from Dataflow

- Good for complex expressions

```
module muladd(q, a, b, c);
  output [15:0] q;
  input [7:0] a, b, c;
  assign q = a + b * c;
endmodule
```



Not all operators are supported



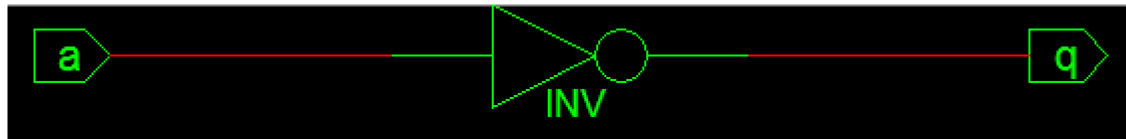
<table>
<tr><td>+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)</td><td>Highest precedence</td></tr>
<tr><td>**</td><td></td></tr>
<tr><td>* / %</td><td></td></tr>
<tr><td>+ - (binary)</td><td></td></tr>
<tr><td><< >> <<< >>></td><td></td></tr>
<tr><td>< <= > >=</td><td></td></tr>
<tr><td>== != === !==</td><td></td></tr>
<tr><td>& (binary)</td><td></td></tr>
<tr><td>^ ^~ ~^ (binary)</td><td></td></tr>
<tr><td>| (binary)</td><td></td></tr>
<tr><td>&&</td><td></td></tr>
<tr><td>||</td><td></td></tr>
<tr><td>?: (conditional operator)</td><td></td></tr>
<tr><td>{} {{}}</td><td>Lowest precedence</td></tr>
</table>

Logic from Procedures

- Use *always* block with sensitivity list
- A *reg* variable is required for procedural assignment, even though in combinational logic we have no storage

```
module t1(q , a);
  output q;
  reg q;
  input a;
  initial q = 25;
  always @(a)
    q = ~a;
endmodule
```

this will not generate logic (points to `initial q = 25;`)

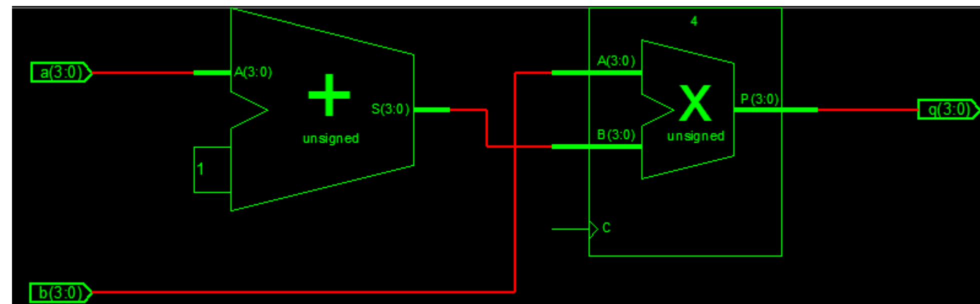


Logic from Procedures

- In multiline expressions, make sure the sensitivity list is complete

```
module t2(q , a, b);
  output [3:0] q;
  reg [3:0] q, t;
  input [3:0] a, b;
  always @(a) begin
    q = b * t;
    t = a + 1;
end
endmodule
```

In this example, synthesized netlist and input verilog will behave differently in a simulator




```
Analyzing top module <t2>.
WARNING:Xst:905 - "ex.v" line 46: The signals <b, t>
are missing in the sensitivity list of always block.
Module <t2> is correct for synthesis.
```

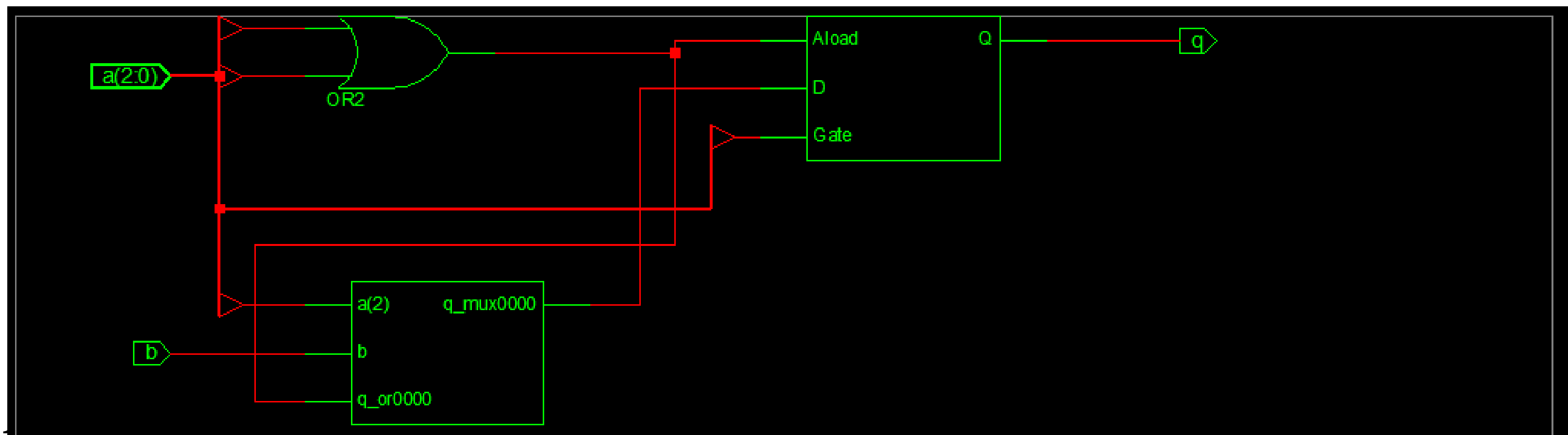
If-then and Case constructs

- If statement
  - Priority-encoded Logic
  - Multiple conditions can be tested
  - Use when speed is important
- Case statement
  - Creates balanced logic
  - Tests a common condition
  - Use for complex decoders

If-then for logic

```
module prio(q , a, b);
  output q;
  reg q;
  input [2:0] a;
  input b;
  always @(a) begin
    if (a[2])       q = b;
    else if (a[1]) q = ~b;
    else if (a[0]) q = 0;
  end
endmodule
```

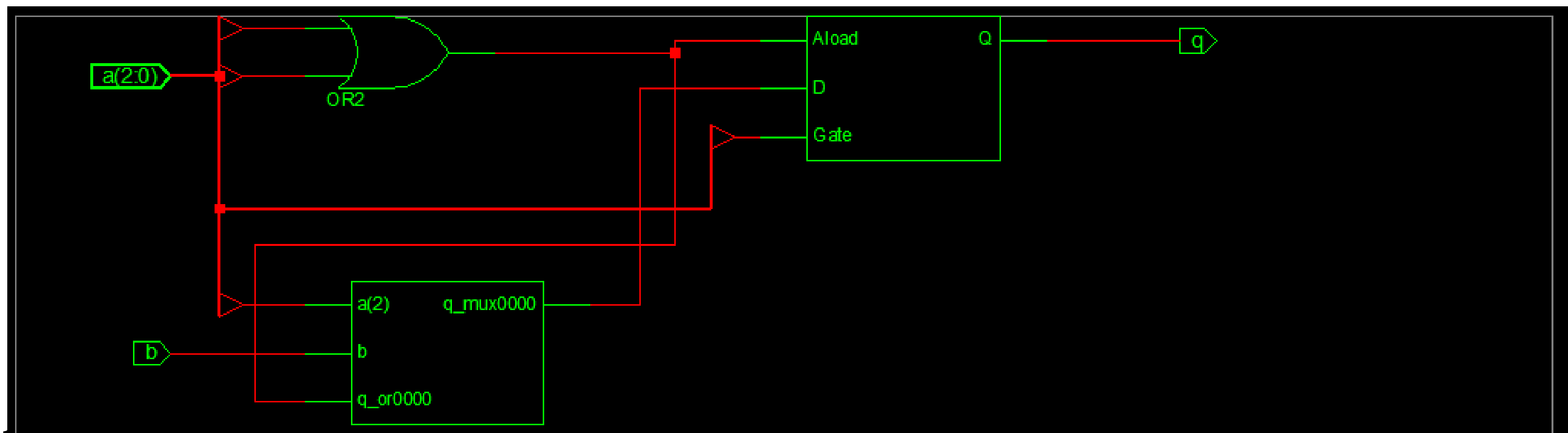
- Two Problems ?



If-then for logic

```
module prio(q , a, b);
  output q;
  reg q;
  input [2:0] a;
  input b;
  always @(a) begin
    if (a[2])       q = b;
    else if (a[1]) q = ~b;
    else if (a[0]) q = 0;
  end
endmodule
```

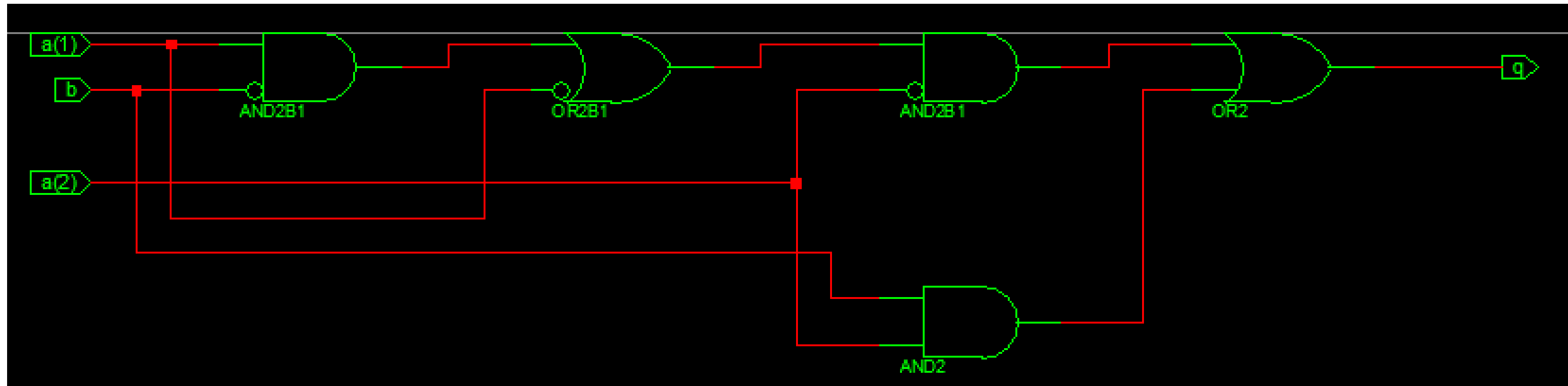
- Two Problems
  - Creates a latch (MUST be solved)
  - Incomplete sensitivity List (will synthesize to something different then the simulation)



If-then for logic

```
module prio(q , a, b);
  output q;
  reg q;
  input [2:0] a;
  input b;
  always @(a or b) begin
    if (a[2])        q = b;
    else if (a[1])   q = ~b;
    else             q = 0;
  end
endmodule
```

- Removing a latch
  - Solution 1: drop condition

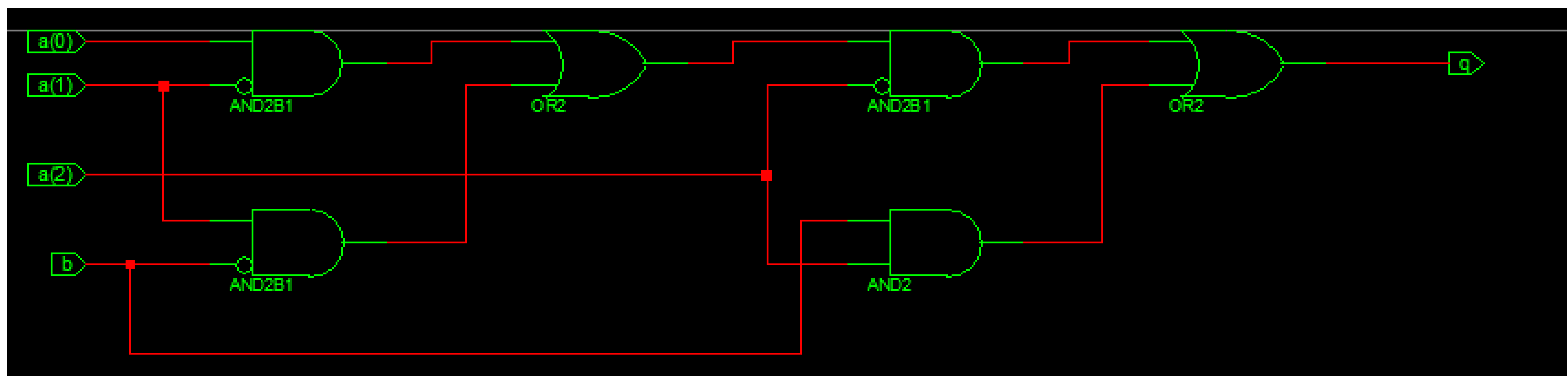


If-then for logic

```
module prio(q , a, b);
  output q;
  reg q;
  input [2:0] a;
  input b;
  always @(a or b) begin
    q = 0;
    if (a[2])       q = b;
    else if (a[1]) q = ~b;
    else if (a[0]) q = 0;
  end
endmodule
```

- Removing a latch
  - Solution 1: drop condition
  - Solution 2: default assignment

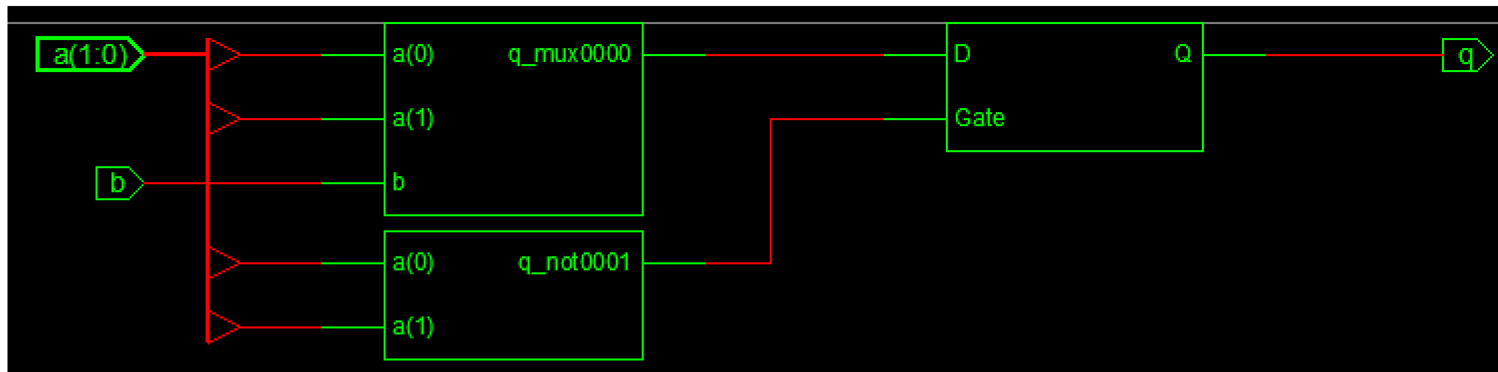
Note that both solutions change the specification! The original spec was incomplete for implementation.



Case for logic

```
module prio3(q , a, b);
  output q;
  reg q;
  input [1:0] a;
  input b;
  always @(a, b) begin
    case(a)
       1: q = b;
       2: q = ~b;
    endcase
  end
endmodule
```

- Another Problem?
- and the Solution(s)?



Case for logic

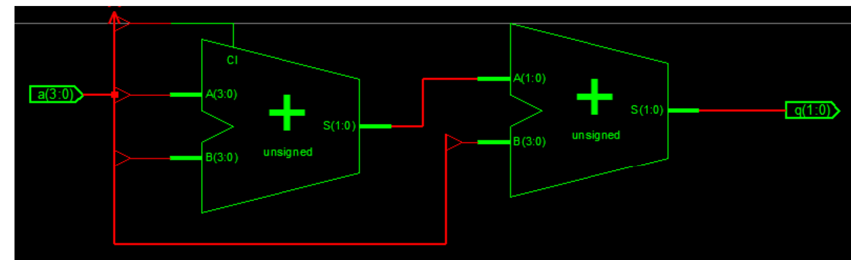
```
module prio3(q , a, b);
  output q;
  reg q;
  input [1:0] a;
  input b;
  always @(a, b) begin
    q = 0;
    case(a)
       1: q = b;
       2: q = ~b;
       default: q =0;
    endcase
  end
endmodule
```

- Another Problem?
  - No default assignment
- Solution 1: Add default entry
- (or) Solution 2: Do a default assignment

Loops

- Loops are treated as syntactical sugar and are unrolled into hardware

```
module sumbits(q , a);
  output [1:0] q;
  reg    [1:0] q, tmp;
  input  [3:0] a;
  reg    [2:0] i;
  always @(a) begin
    tmp = 0;
    for (i=0; i<4; i = i + 1)
      tmp = tmp + a[i];
      q = tmp;
  end
endmodule
```



Loop restrictions

- Synthesis tool must be able to find number of iterations at compile time, number of iterations must be finite

```
module sumaslong(q , a);
  output [1:0] q;
  reg    [1:0] q, tmp;
  input  [3:0] a;
  reg    [2:0] i;
  always @(a) begin
    i = a;
    while (i < 4)
      i = i + 1;
      q = i;
  end
endmodule
```

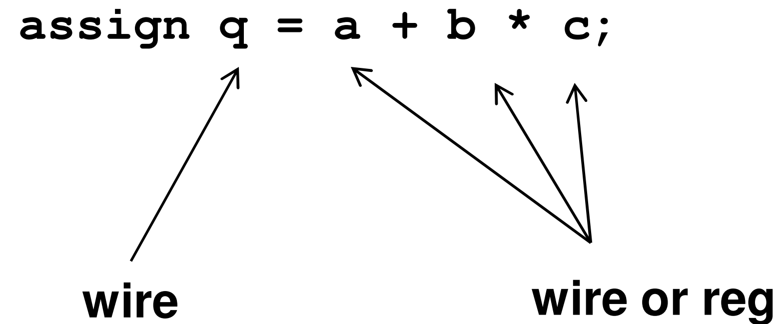
```
ERROR:Xst:1312 - Loop has iterated 64 times. Use "set -
loop_iteration_limit XX" to iterate more.

Found 1 error(s). Aborting synthesis.
```

Combinational Logic Pitfalls

- In procedural code, assign all outputs under all possible control flow paths
  - Otherwise, latches will appear
- In procedural code, make sure that sensitivity lists are complete

Hardware Inference of Combinational Logic

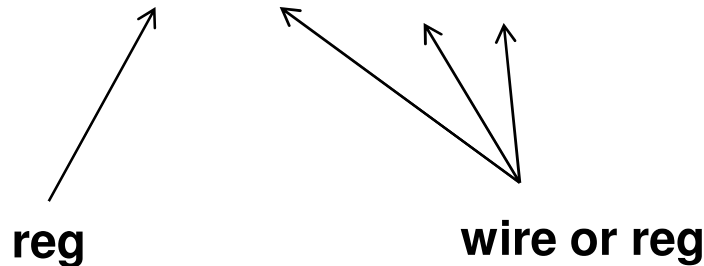
- Method 1: Dataflow Modeling



Hardware Inference of Combinational Logic

- Method 2: Use an always block
  - Assign block outputs under *all* possible control paths
  - Include block all inputs in sensitivity list

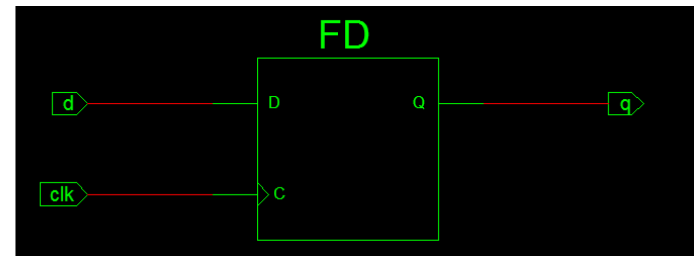
```
always @(a or b or c)
  if (a > 2)
    q = a + b * c;
  else
    q = a - b * c;
```



Hardware Inference of Sequential Logic

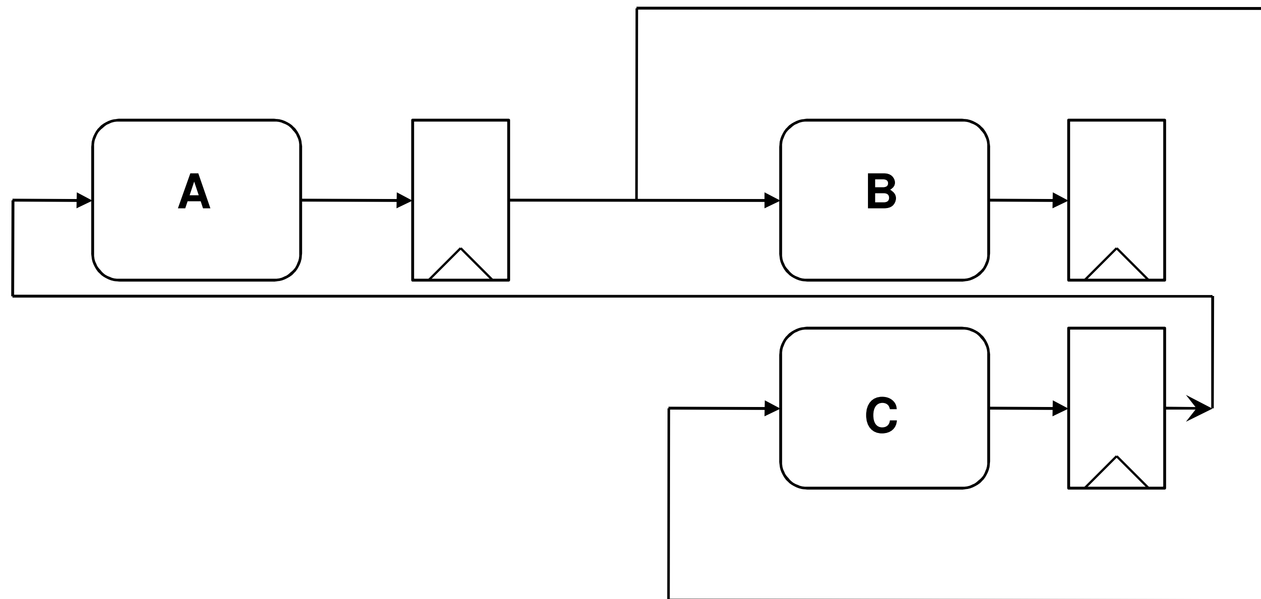
- Flip-flops require a clock signal and (optionally) a reset signal
- Standard modeling practice: Use always block and non-blocking assignment for flip-flop
  - Include the clock signal in the sensitivity list

```
module dff(q , clk, d);
  output q;
  reg q;
  input clk, d;
  always @(posedge clk)
    q <= d;
endmodule
```



Sequential Logic

- Standard modeling practice: Use always block and non-blocking assignment for flip-flop
- This ensures that all regs update at the same time



Flip-flops vs Latches

Edge-triggered Flip Flop

```
module dff(q , clk, d);
  output q;
  reg q;
  input clk, d;
  always @(posedge clk)
    q <= d;
endmodule
```

Level-sensitive Latch

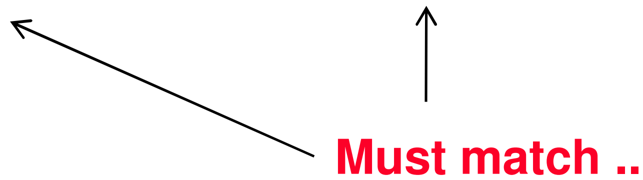
```
module dlatch(q, enable, d);
  output q;
  reg q;
  input enable, d;
  always @(enable)
     if (enable)
        q <= d;
endmodule
```

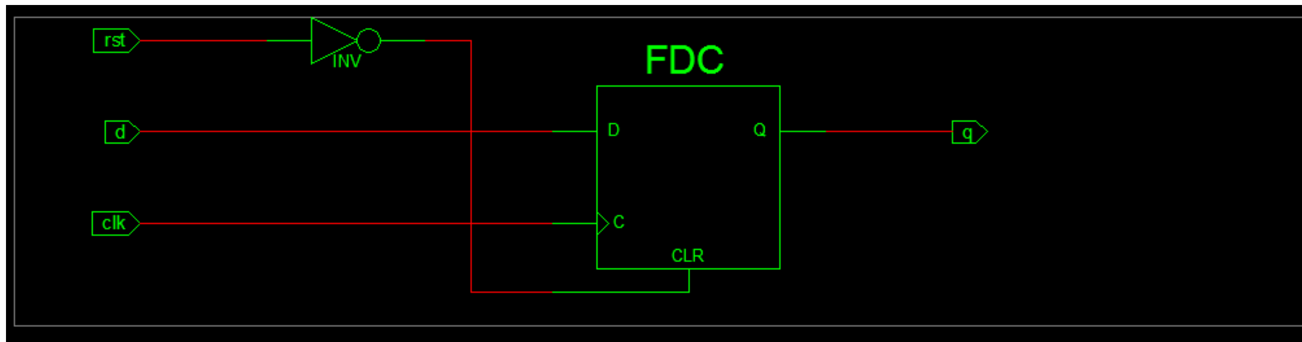
- Compared to Flip-flops ...
  - Latches have less area, have a more flexible clocking scheme, consume less power, but are more complex to use and verify
- Good Practice #1: Don't mix flip-flops and latches
  - use only a single type of storage
- Good Practice #2: Prefer flip-flops over latches
  - if you cannot motivate precisely why you need a latch, don't use it

Asynchronous Reset

- Extend the event list ..

```
module dffrst(q , clk, rst, d);
  output q;
  reg q;
  input clk, rst, d;
  always @(posedge clk or negedge rst)
    if (!rst)
      q <= 0;
    else
      q <= d;
endmodule
```

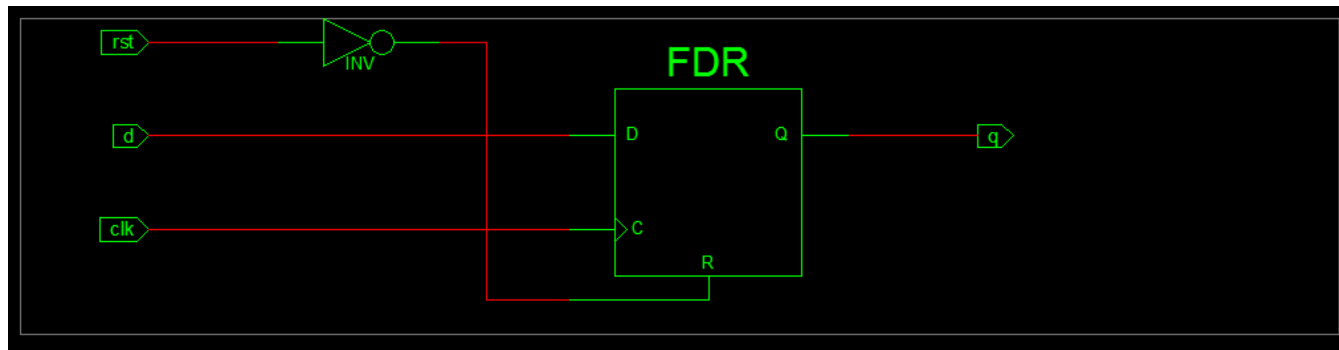




FDC = flip-flop with asynchronous clear

Synchronous Reset

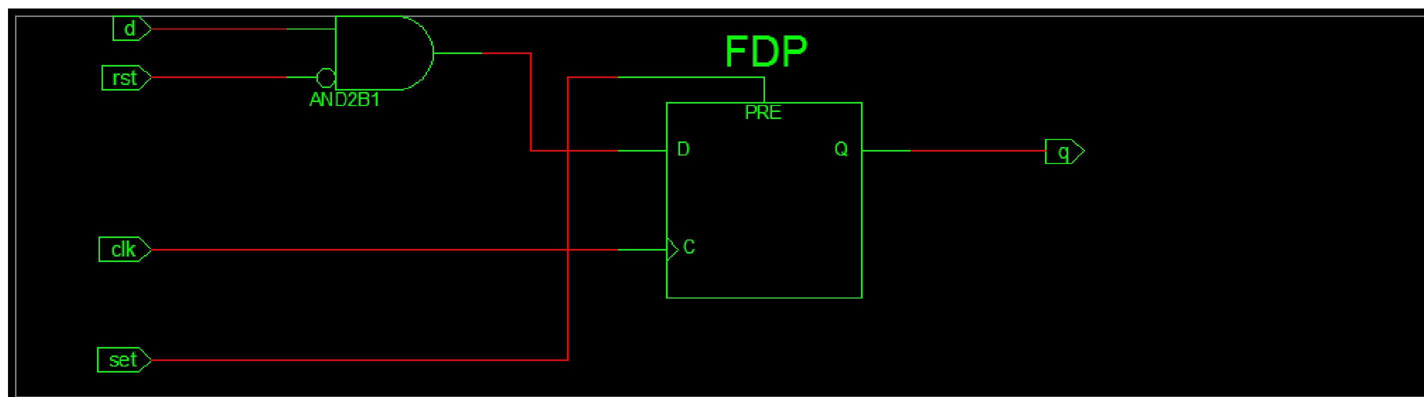
```
module dffrst(q , clk, rst, d);
  output q;
  reg q;
  input clk, rst, d;
  always @(posedge clk)
    if (!rst)
      q <= 0;
    else
      q <= d;
endmodule
```



FDR = flip-flop with synchronous reset

Mixed Synchronous/Asynchronous

```
module dffrst(q , clk, rst, set, d);
  output q;
  reg q;
  input clk, rst, d, set;
  always @(posedge clk or posedge set)
    if (set)        q <= 1;
    else if (rst) q <= 0;
    else            q <= d;
endmodule
```



FDP = flip-flop with asynchronous preset

Flip-flops are inferred by pattern matching

OK

```
module dffrst(q , clk, rst, d);
  output q;
  reg q;
  input clk, rst, d;
  always @(posedge clk or negedge rst)
    if (!rst)
      q <= 0;
    else
      q <= d;
endmodule
```

NOT OK

```
module dffrst(q , clk, rst, d);
  output q;
  reg q;
  input clk, rst, d;
  always @(posedge clk or negedge rst)
    if (rst)
      q <= d;
    else
      q <= 0;
endmodule
```

ERROR:Xst:898 - "ex.v" line 158: The reset or set test condition for <q> is incompatible with the event declaration in the sensitivity list.

Flip-flops are inferred by pattern matching

OK

```
module dffrst(q , clk, rst, set, d);
  output q;
  reg q;
  input clk, rst, d, set;
  always @(posedge clk or posedge set)
    if (set)       q <= 1;
    else if (rst)  q <= 0;
    else           q <= d;
endmodule
```

NOT OK

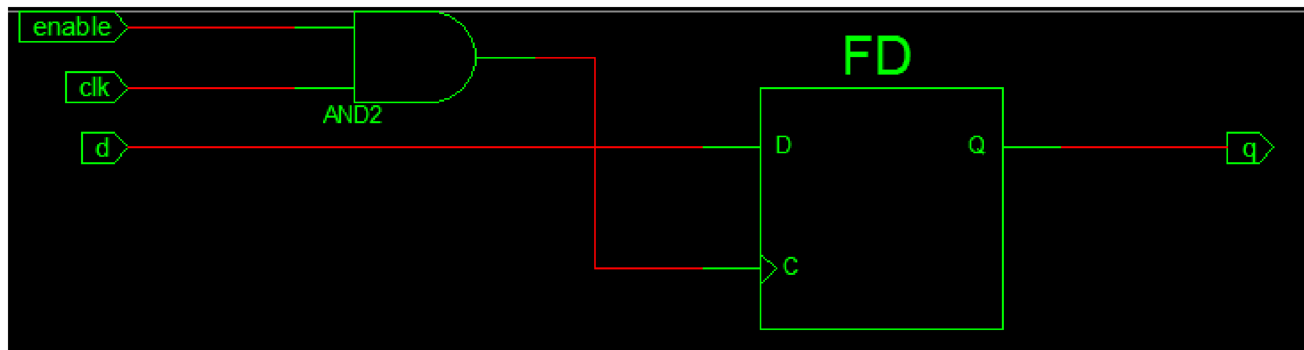
```
module dffrst(q , clk, rst, set, d);
  output q;
  reg q;
  input clk, rst, d, set;
  always @(posedge clk or set)
    if (set)       q <= 1;
    else if (rst)  q <= 0;
    else           q <= d;
endmodule
```

ERROR:Xst:902 - "ex.v" line 167: Unexpected set event in always block sensitivity list.

Gated clocks

- *this must be avoided !!*
  (skew & other asynchronous effects)

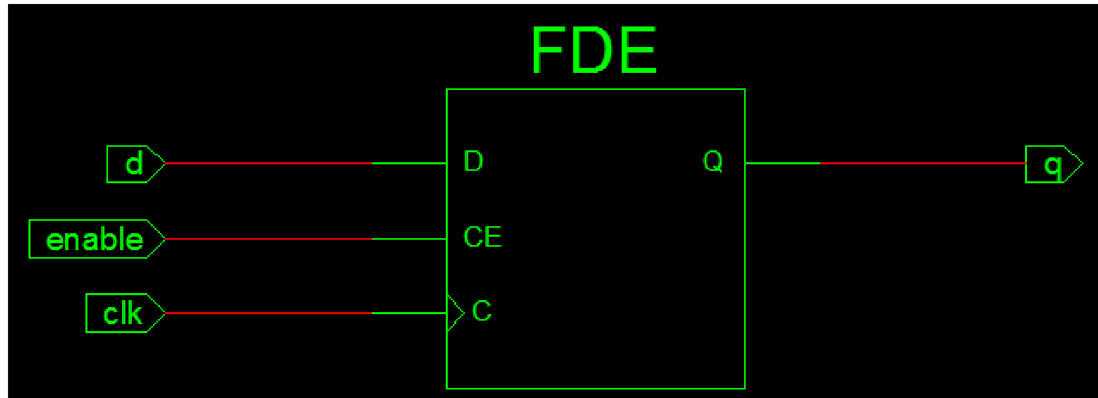
```
module dffgated(q , enable, clk, d);
  output q;
  reg q;
  input clk, d, enable;
  wire gatedclk;
  assign gatedclk = clk & enable;
  always @(posedge gatedclk)
    q <= d;
endmodule
```



Clock-enable

- Use this instead of gated clocks

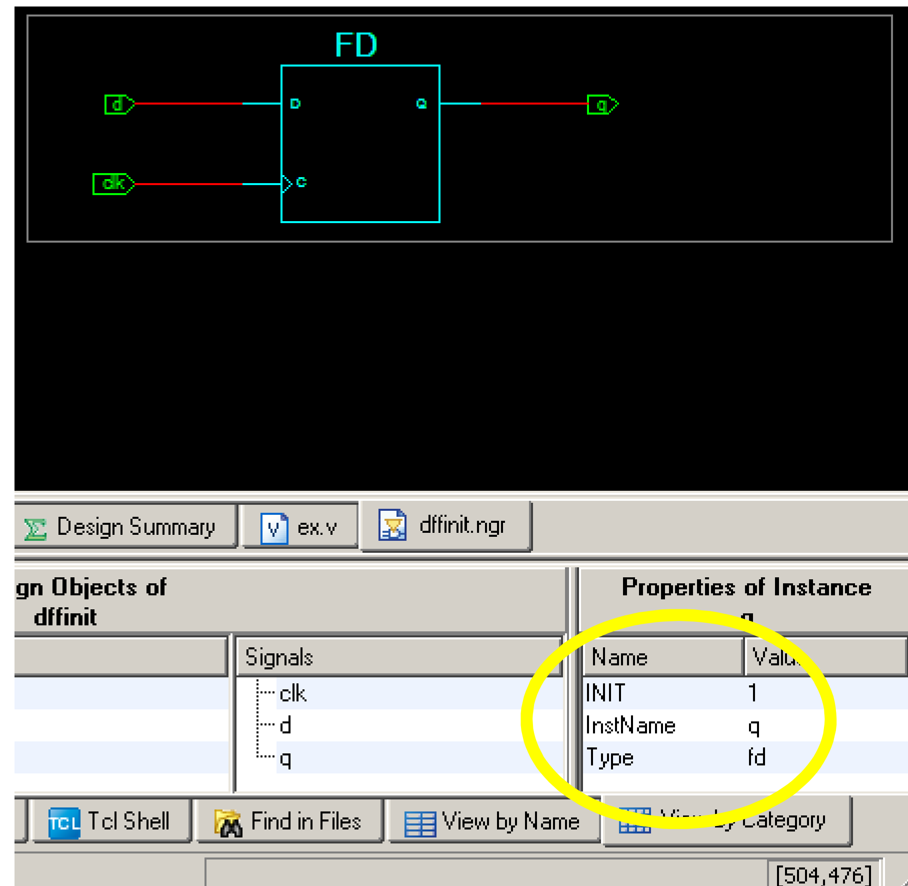
```
module dffgated(q , enable, clk, d);
  output q;
  reg q;
  input clk, d, enable;
  always @(posedge clk)
    if (enable)
      q <= d;
endmodule
```



Register Initalization

- In FPGA, flip-flops can be configured in an initialized state (not true for ASIC technology)
- FPGA Synthesis tools provide some support for modeling this

```
module dffinit(q, clk, d);
  output q;
  reg q = 1'b1;
  input clk, d, enable;
  always @(posedge clk)
    q <= d;
endmodule
```



Register Initalization

- Works as well (but only in FPGA):

```
module dffinit(q, clk, d);
  output q;
  reg q;
  input clk, d, enable;
  initial
    q = 1;
  always @(posedge clk)
    q <= d;
endmodule
```

Inferring Sequential Logic - Summary

- Flip-flops and Latches
  - @(posedge clk) .. -> flip-flop
  - @(en) + if (en) ... -> latch
- Synchronous and asynchronous set and reset
  - @(posedge rst) + if (rst) ... -> asynchronous
  - @(posedge clk) + if (rst) ... -> synchronous
- Gated clocks and Clock-enable
  - Avoid gated clocks
- Register initialization

Non-synthesizable Constructs

- Basically, everything which does not fit in the category 'combinational logic' or 'sequential logic'
- Event controls: wait, # (delay)
- Advanced data types: real, time
- Initial blocks (other then for flipflop initialization)
- System tasks ($display, ..)
- Many aspects of Verilog which we have *not* discussed:
  - procedural assign, deassign
  - specify
  - some switch-level data-types
  - named events

Summary Hardware Inference

- Logic Synthesis converts Verilog into Hardware by *Hardware Inference*
- Make sure you understand
  - How to capture combinational logic in an always block
  - How to capture registers in an always block
  - How to recognize warnings and errors from the tools when you synthesize your design
- When doing RTL design, ***first*** think about what you want to achieve, ***then*** think about how to model it in Verilog
  - Logic Synthesis/RTL Synthesis isn't magic:
  - Garbage In -> (Logic Synthesis) -> Garbage Out
  - Clean Code In -> (Logic Synthesis) -> Efficient Design Out
  - Logic synthesis takes away the tedious work of digital design (Karnaugh maps ..), but not the required intellectual effort!