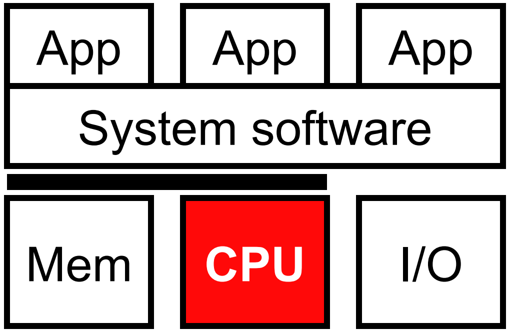
CIS 5710
Computer Organization and Design

Unit 2: Digital Logic & Verilog

Based on slides by Benedict Brown, Amir Roth,
Milo Martin & C.J. Taylor

This Unit: Digital Logic & Verilog

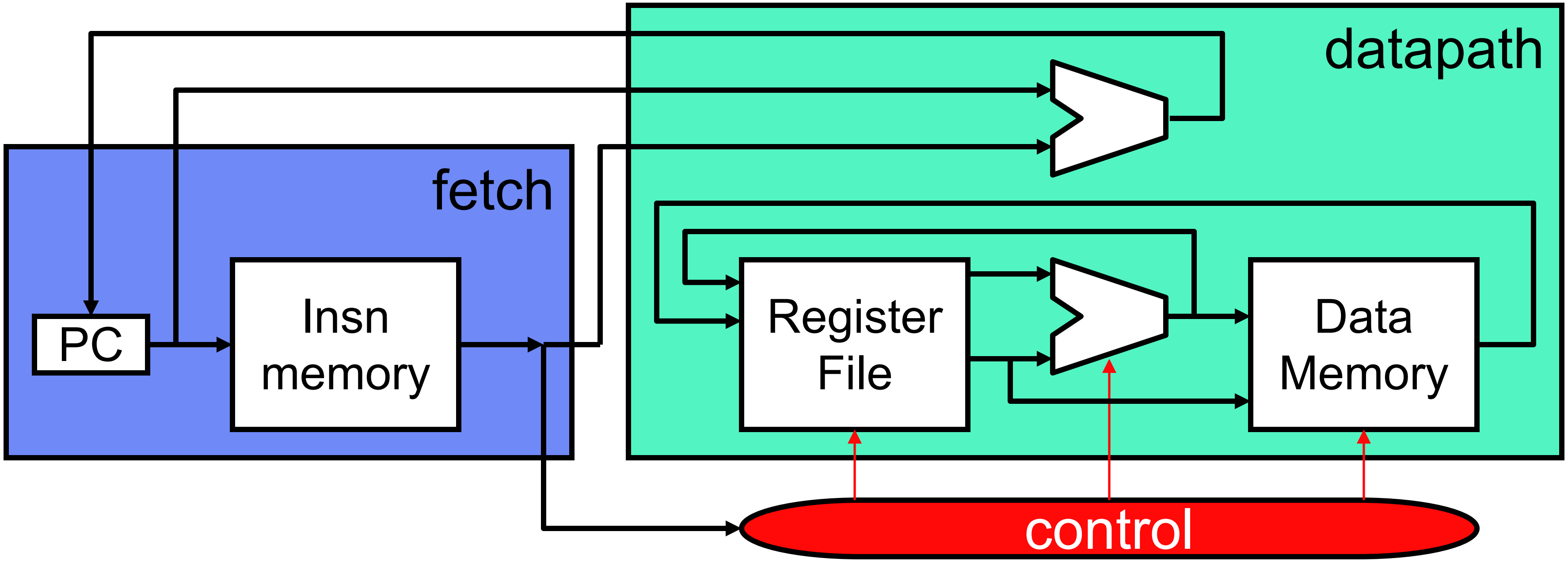


- Transistors & fabrication
- Digital logic basics
  - Focus on useful components
- Hardware design methods
  - Introduction to Verilog

Readings

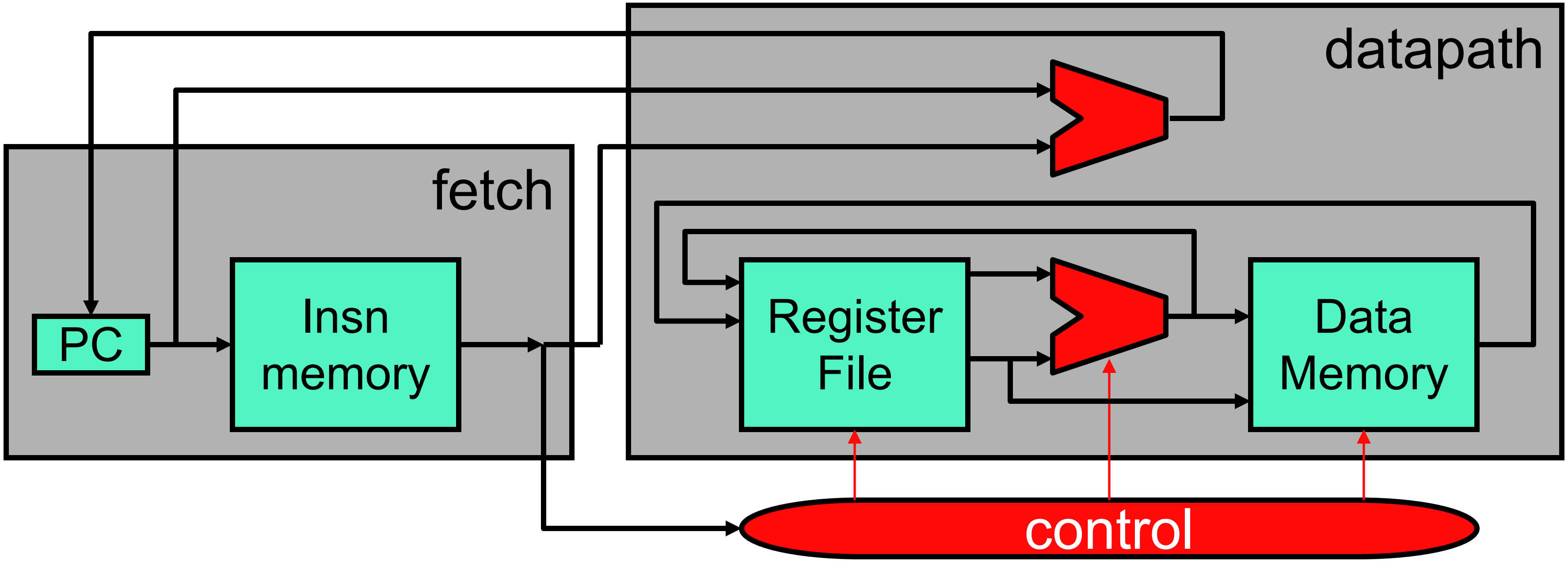
- Digital logic
  - P&H, Appendix C
- Manufacturing
  - P&H, Section 1.7
- Introduction to Logic Synthesis using Verilog HDL, Reese & Thornton
- See course homepage for other Verilog HDL resources

Motivation: Implementing a Datapath



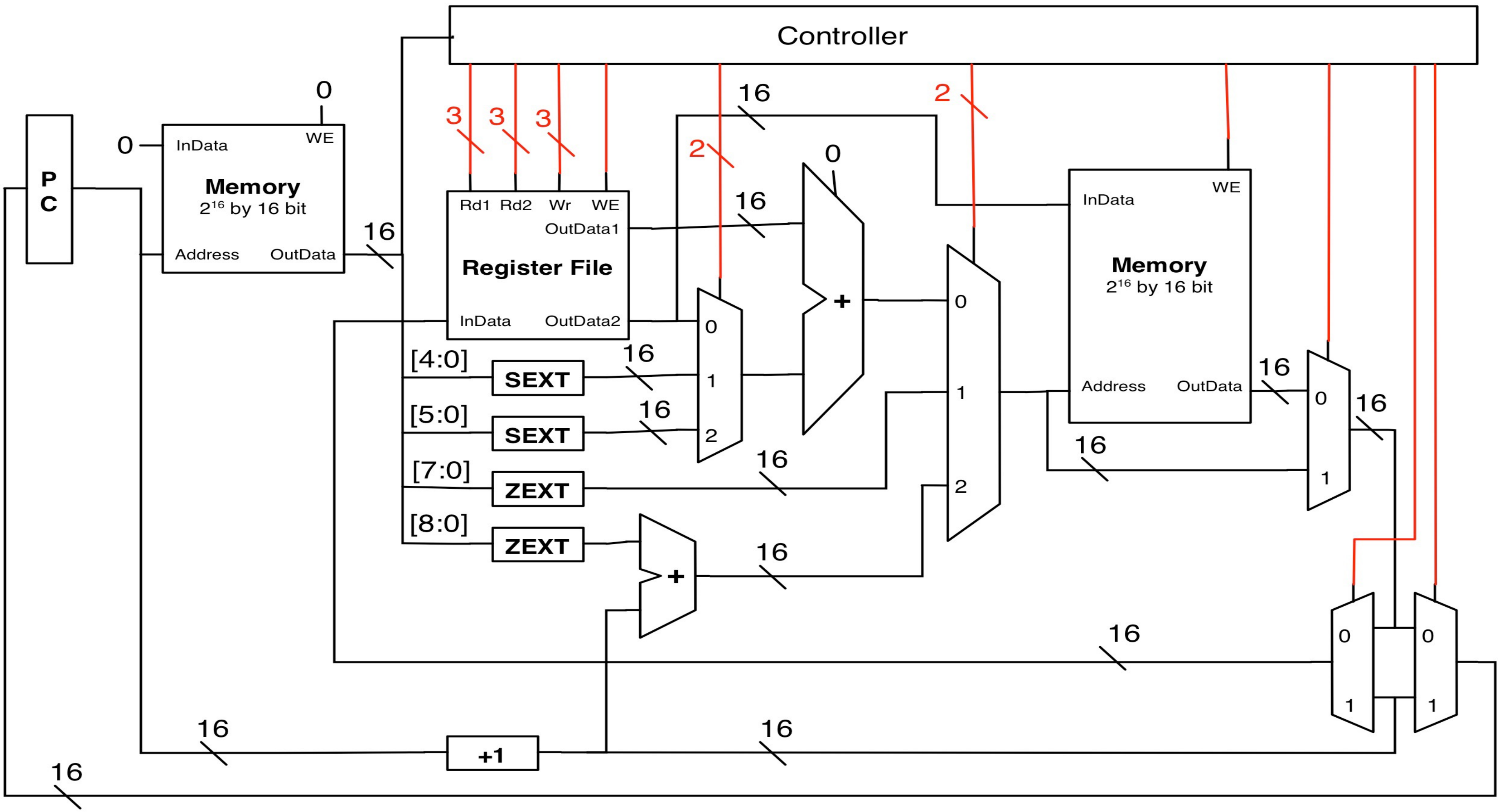
- **Datapath**: performs computation (registers, ALUs, etc.)
  - ISA specific: can implement every insn (single-cycle: in one pass!)
- **Control**: determines which computation is performed
  - Routes data through datapath (which regs, which ALU op)
- **Fetch**: get insn, translate opcode into control
- **Fetch** → **Decode** → **Execute** "cycle"

Two Types of Components

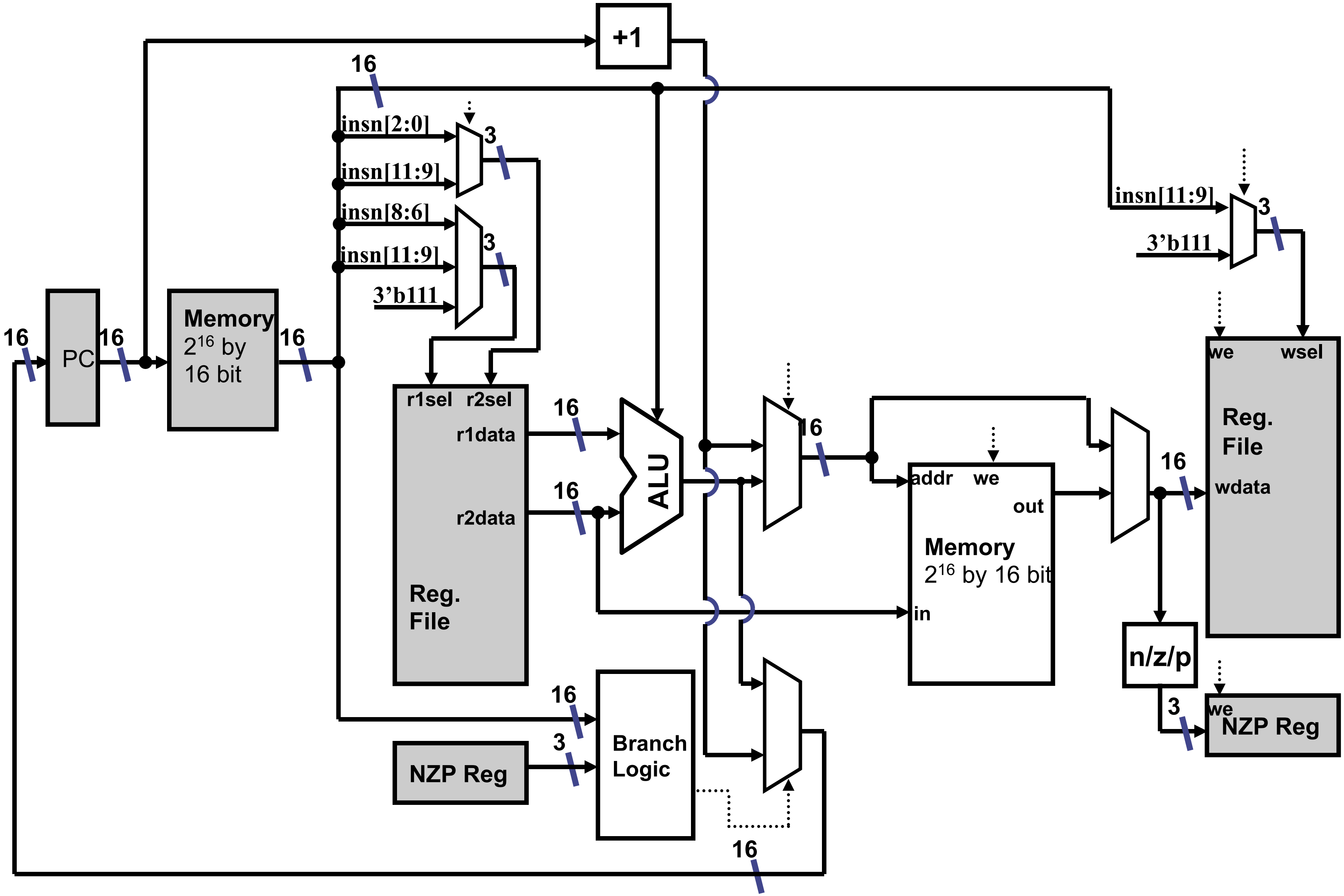


- **Purely combinational**: stateless computation
  - ALUs, muxes, control
  - Arbitrary Boolean functions
- **Combinational+sequential**: storage
  - PC, insn/data memories, register file
  - Internally contain some combinational components

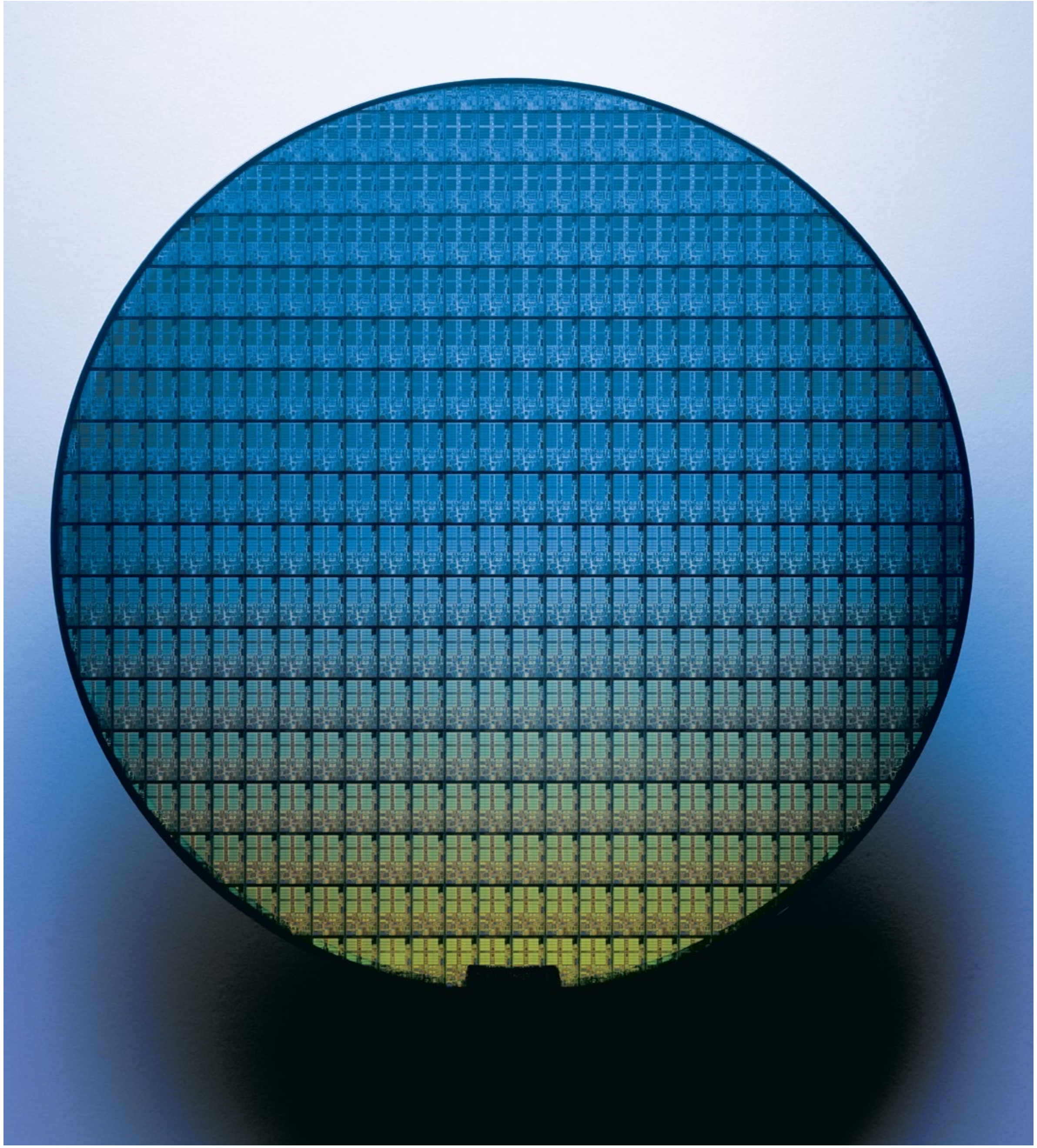
Example LC4 Datapath



LC4 Datapath

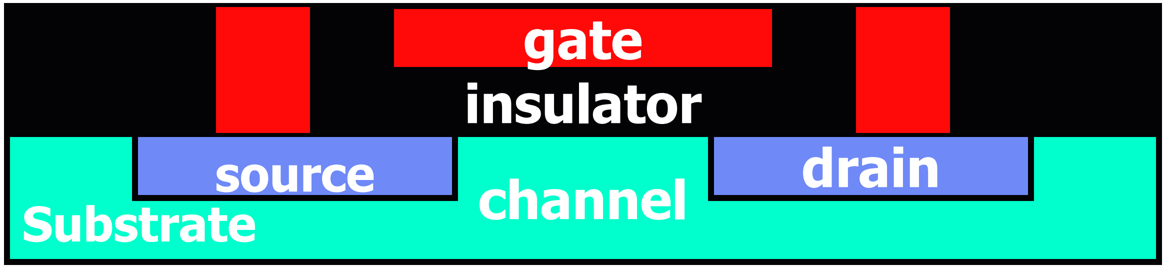


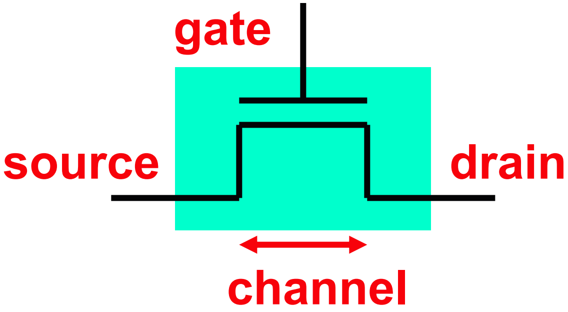
Transistors & Fabrication



Intel Pentium M Wafer

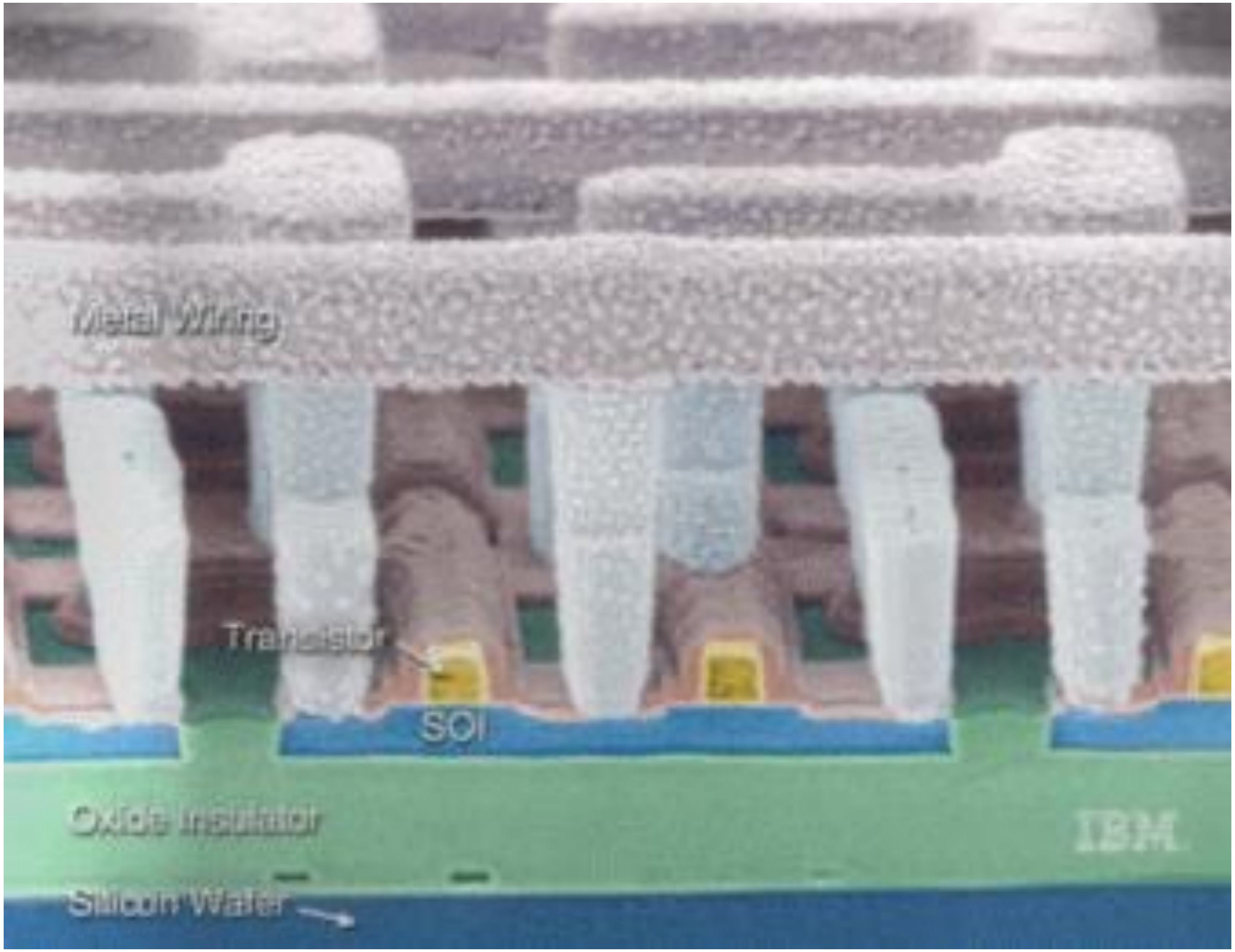
Semiconductor Technology





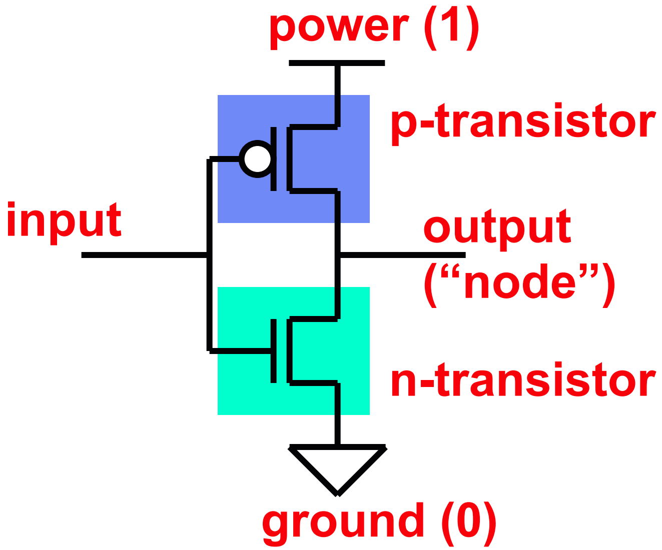
- Basic technology element: **MOSFET**
  - Solid-state component acts like electrical switch
  - **MOS**: metal-oxide-semiconductor
    - Conductor, insulator, semi-conductor
- **FET**: field-effect transistor
  - Channel conducts source→drain only when voltage applied to gate
- **Channel length**: characteristic parameter (short → fast)
  - Aka "feature size" or "technology"
  - Currently: "5 nanometers (nm)"
  - Continued miniaturization (scaling) known as "**Moore's Law**"
    - Won't last forever, physical limits approaching (or are they?)

Transistors and Wires



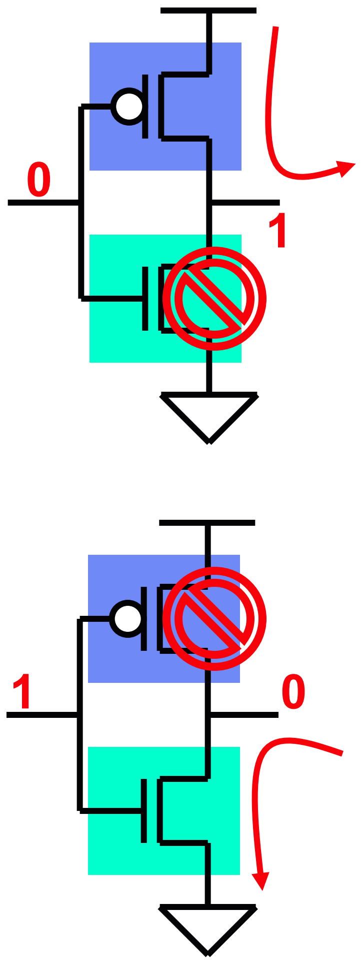
Complementary MOS (CMOS)

- Voltages as values
  - Power (V_{DD}) = "1", Ground = "0"
- Two kinds of MOSFETs
  - **N-transistors**
    - Conduct when gate voltage is 1
    - Good at passing 0s
  - **P-transistors**
    - Conduct when gate voltage is 0
    - Good at passing 1s
- **CMOS**
  - Complementary n-/p- networks form boolean logic (i.e., gates)
  - And some non-gate elements too (important example: RAMs)



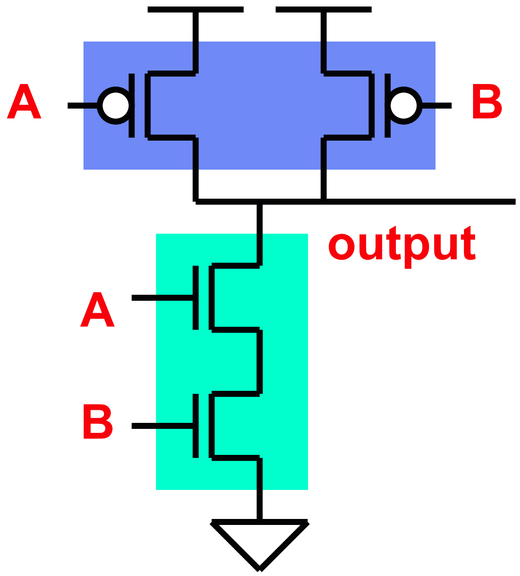
Basic CMOS Logic Gate

- **Inverter**: NOT gate
  - One p-transistor, one n-transistor
  - Basic operation
  - Input = 0
    - P-transistor closed, n-transistor open
    - Power charges output (1)
  - Input = 1
    - P-transistor open, n-transistor closed
    - Output discharges to ground (0)

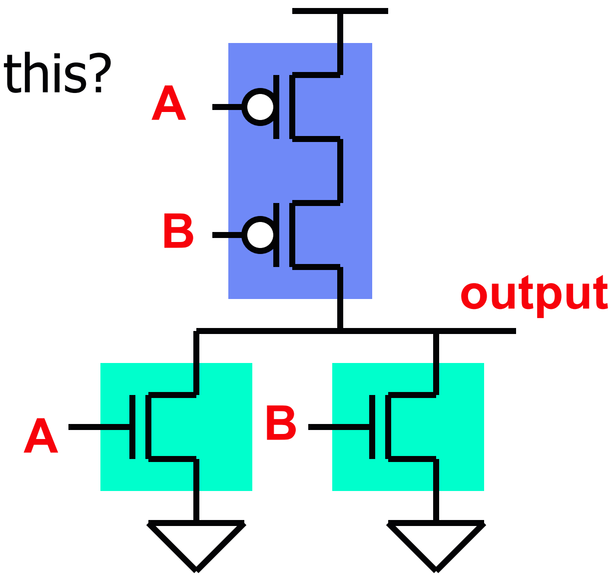


Another CMOS Gate Example

- What is this? Look at **truth table**
  - 0, 0 → 1
  - 0, 1 → 1
  - 1, 0 → 1
  - 1, 1 → 0
  - Result: **NAND** (NOT AND)
  - NAND is "universal"

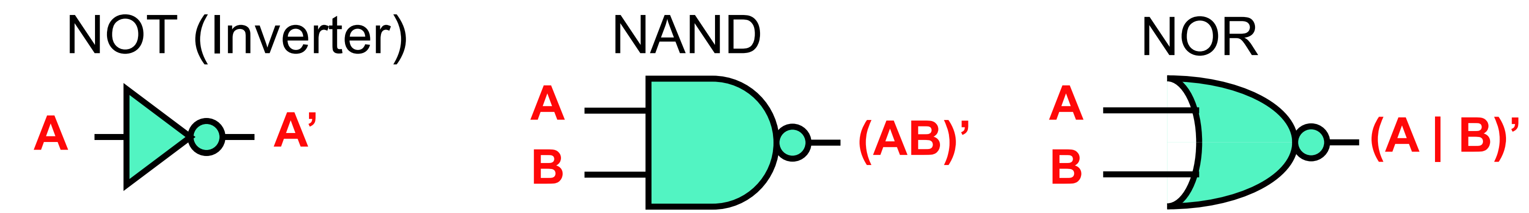


- What function is this?

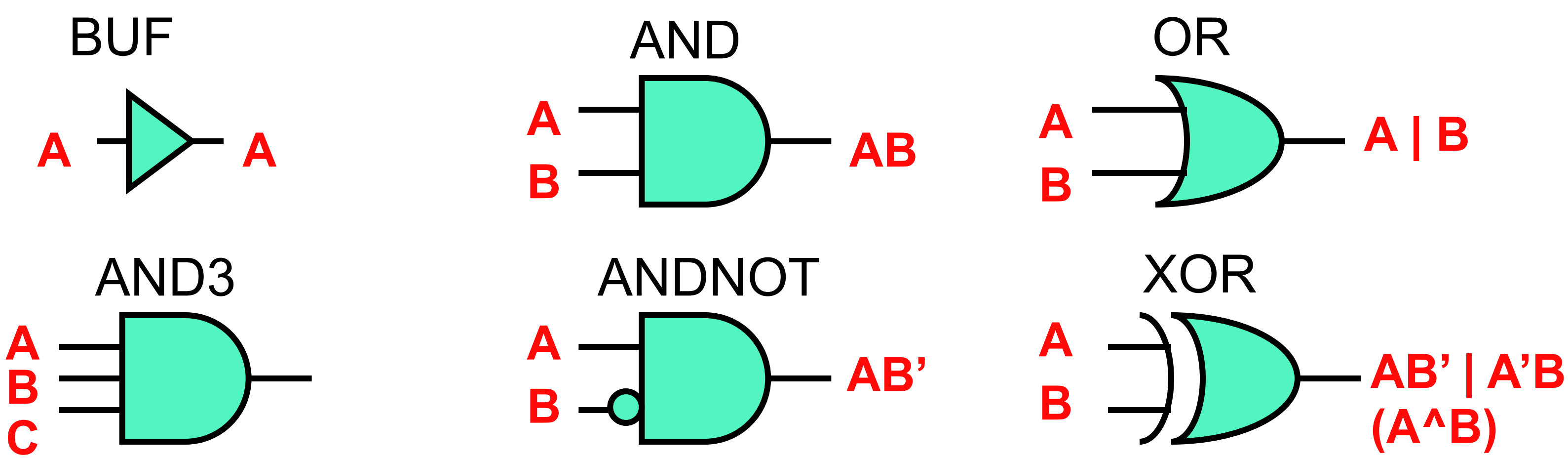


Digital Building Blocks: Logic Gates

- **Logic gates:** implement Boolean functions
  - Basic gates: NOT, NAND, NOR
    - Underlying CMOS transistors are naturally inverting (○ = NOT)



  - NAND, NOR are "Boolean complete"



Digital Logic Review

Boolean Functions and Truth Tables

- Any Boolean function can be represented as a truth table
  - **Truth table**: point-wise input → output mapping
  - Function is disjunction of all rows in which "Out" is 1

```
A,B,C → Out
0,0,0 → 0
0,0,1 → 0
0,1,0 → 0
0,1,1 → 0
1,0,0 → 0
1,0,1 → 1
1,1,0 → 1
1,1,1 → 1
```

- Example above: Out = AB'C | ABC' | ABC

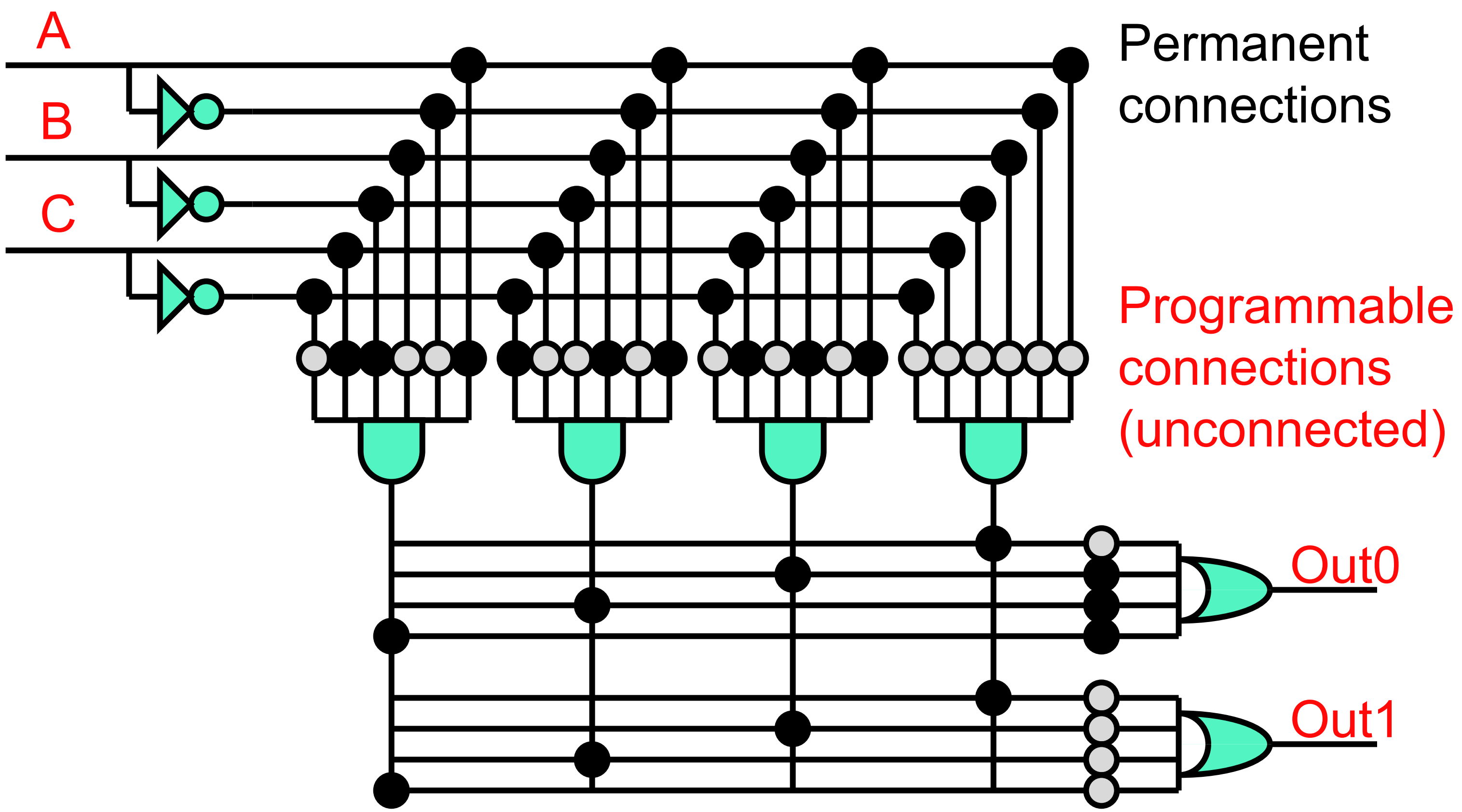
Truth Tables and PLAs

- Implement Boolean function by implementing its truth table
  - Takes two levels of logic
    - Assumes inputs and inverses of inputs are available (usually are)
  - First level: ANDs (product terms)
  - Second level: ORs (sums of product terms)

- **PLA (programmable logic array)**
  - Flexible circuit for doing this

PLA Example

- PLA with 3 inputs, 2 outputs, and 4 product terms
  - Out0 = AB′C | ABC′ | ABC



Boolean Algebra

- **Boolean Algebra**: rules for rewriting Boolean functions
  - Useful for simplifying Boolean functions
    - Simplifying = reducing gate count, reducing gate "levels"
  - Rules: similar to logic (0/1 = F/T)
    - **Identity**: A1 = A, A | 0 = A
    - **0/1**: A0 = 0, A | 1 = 1
    - **Inverses**: (A')' = A
    - **Idempotency**: AA = A, A | A = A
    - **Tautology**: AA' = 0, A | A' = 1
    - **Commutativity**: AB = BA, A | B = B | A
    - **Associativity**: A(BC) = (AB)C, A | (B | C) = (A | B) | C
    - **Distributivity**: A(B | C) = AB | AC, A | (BC) = (A | B)(A | C)
    - **DeMorgan's**: (AB)' = A' | B', (A | B)' = A'B'

Logic Minimization

- **Logic minimization**
  - Iterative application of rules to reduce function to simplest form
  - **Design tools do this automatically**

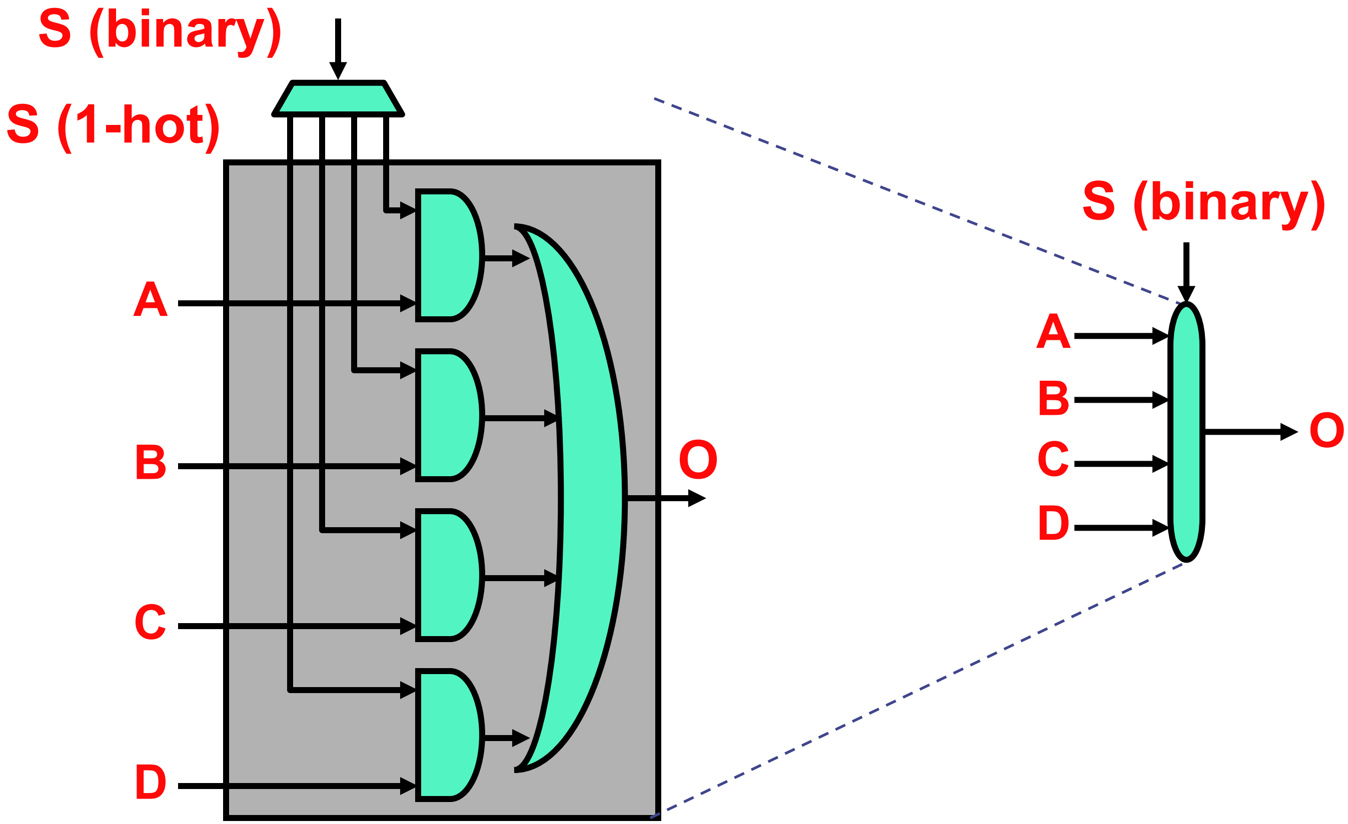
```
Out = AB′C  |  ABC′  |  ABC
Out = A(B′C  |  BC′  |  BC)        // distributivity
Out = A(B′C  |  (BC′  |  BC))      // associativity
Out = A(B′C  |  B(C′ | C))         // distributivity (on B)
Out = A(B′C  |  B1)                // tautology
Out = A(B′C  |  B)                 // 0/1
Out = A((B′ | B)(C | B))           // distributivity (on  | B)
Out = A(1(B | C))                  // tautology
Out = A(B | C)                     // 0/1
```

Non-Arbitrary Boolean Functions

- PLAs implement Boolean functions point-wise
  - E.g., represent f(X) = X+5 as [0→5, 1→6, 2→7, 3→8, …]
  - Mainly useful for "arbitrary" functions, no compact representation
- Many useful Boolean functions are not arbitrary
  - Have a compact implementation
  - Examples
    - Multiplexer
    - Adder

Multiplexer (Mux)

- **Multiplexer (mux)**: selects output from N inputs
  - Example: 1-bit 4-to-1 mux
  - Not shown: N-bit 4-to-1 mux = N 1-bit 4-to-1 muxes + 1 decoder



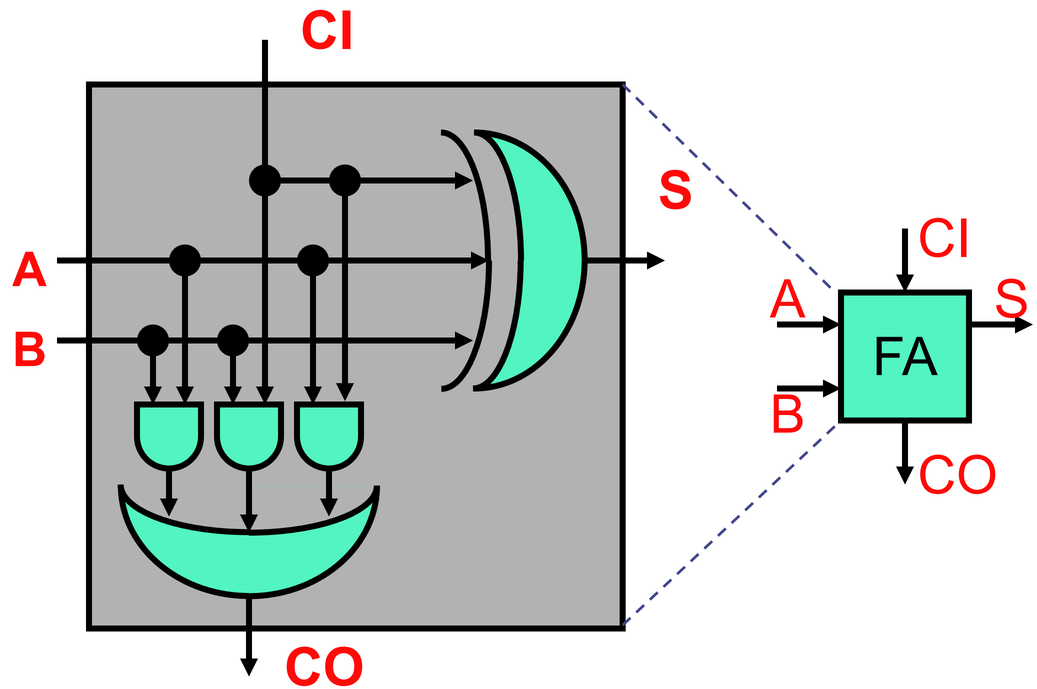
Adder

- **Adder**: adds/subtracts two binary integers in two's complement format
  - **Half adder**: adds two 1-bit "integers", no carry-in
  - **Full adder**: adds three 1-bit "integers", includes carry-in
  - **Ripple-carry adder**: N chained full adders add 2 N-bit integers
  - **To subtract**: negate B input, set bit 0 carry-in to 1

Full Adder

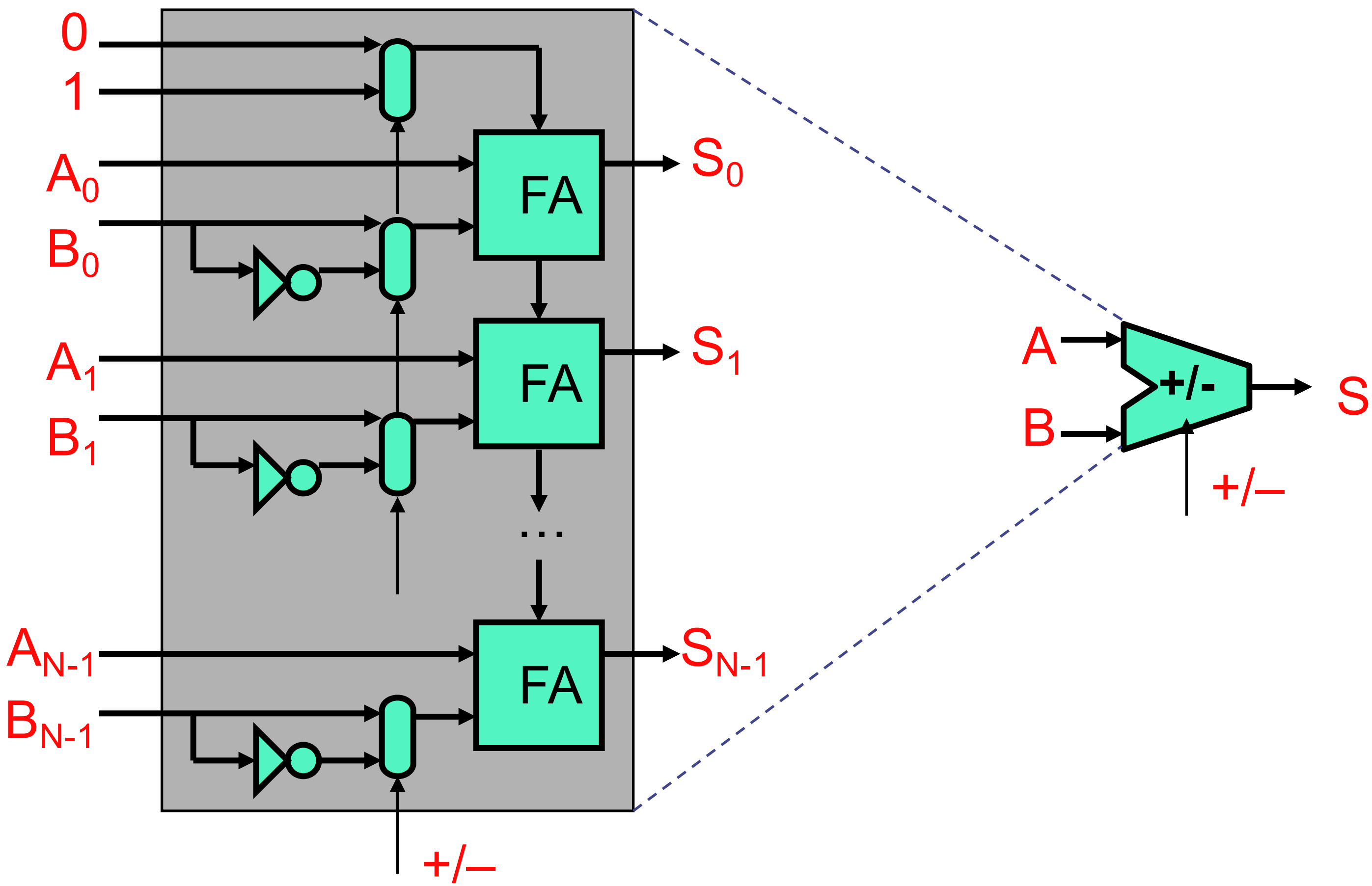
- What is the logic for a full adder?
  - Look at truth table

CI	A	B	→	CO	S
0	0	0	→	0	0
0	0	1	→	0	1
0	1	0	→	0	1
0	1	1	→	1	0
1	0	0	→	0	1
1	0	1	→	1	0
1	1	0	→	1	0
1	1	1	→	1	1



- **S = C′A′B | C′AB′ | CA′B′ | CAB = C ^ A ^ B**
- **CO = C′AB | CA′B | CAB′ | CAB = CA | CB | AB**

N-bit Adder/Subtracter



- More later when we cover arithmetic

FPGAs

Alternative to Fabrication: FPGA

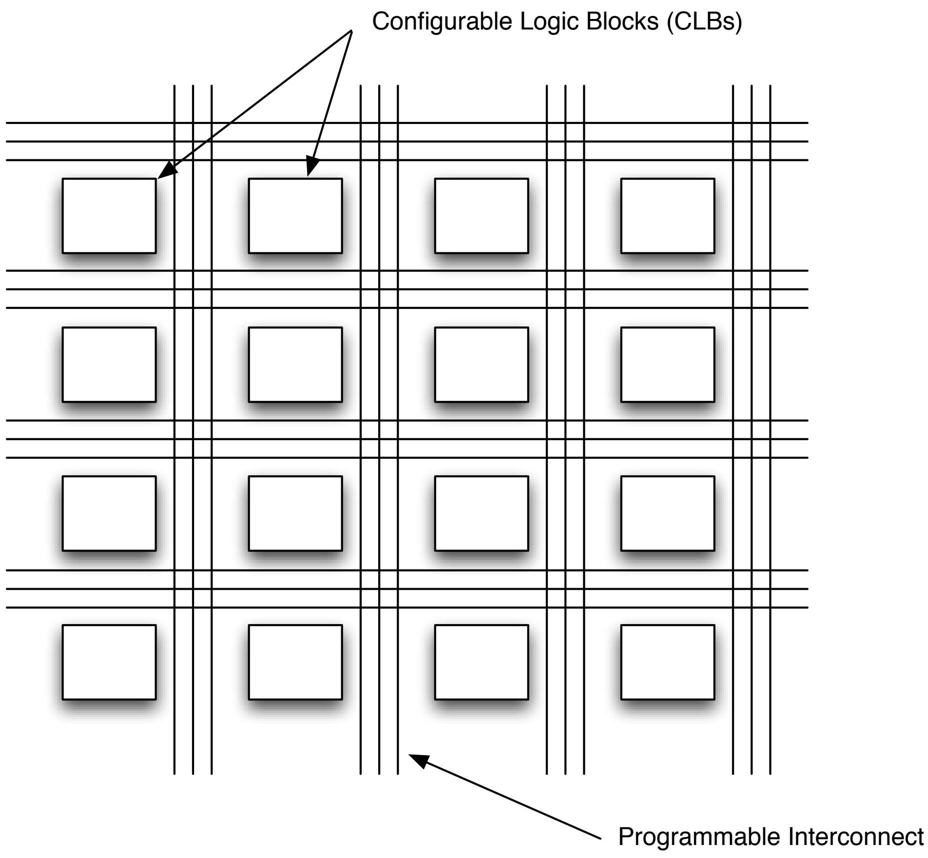
- We'll use **FPGAs (Field Programmable Gate Array)**
  - Also called Programmable Logic Devices (PLDs)

- An FPGA is a special type of programmable chip
  - Conceptually, contains a grid of gates
  - The wiring connecting them can be reconfigured electrically
    - Using more transistors as switches
  - Once configured, the FPGA can emulate any digital logic design
  - Tool converts **gate-level design** to configuration

- Uses
  - **Hardware prototyping** (what we're doing)
  - Low-volume special-purpose hardware
  - Network processing. FPGAs in AWS & Azure

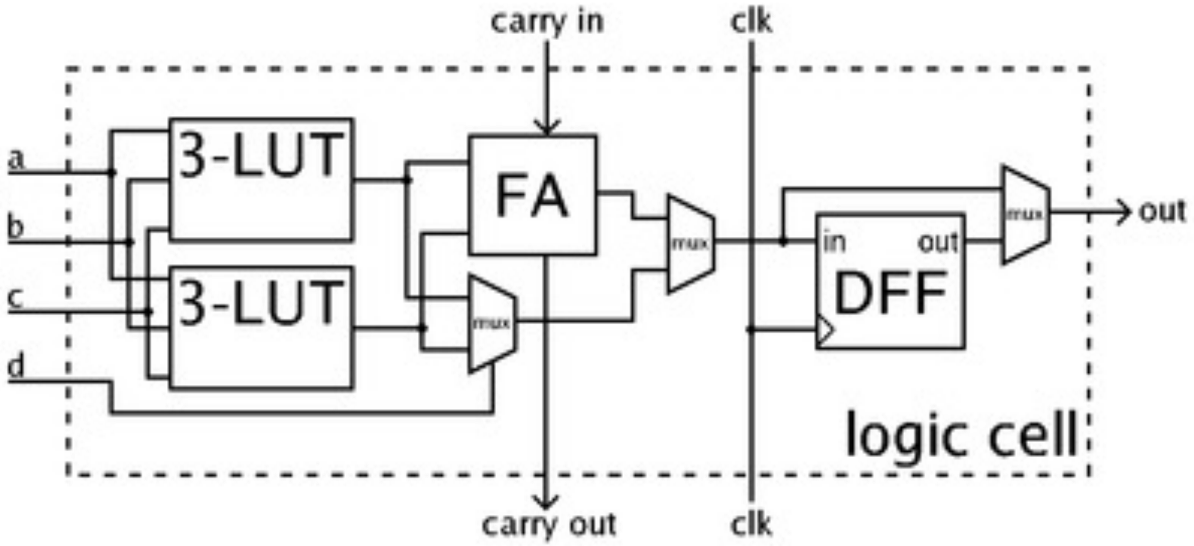
FPGA

- A Field Programmable Gate Array contains a collection of configurable logic elements and a programmable interconnect that can be set up to perform the desired logical operations.



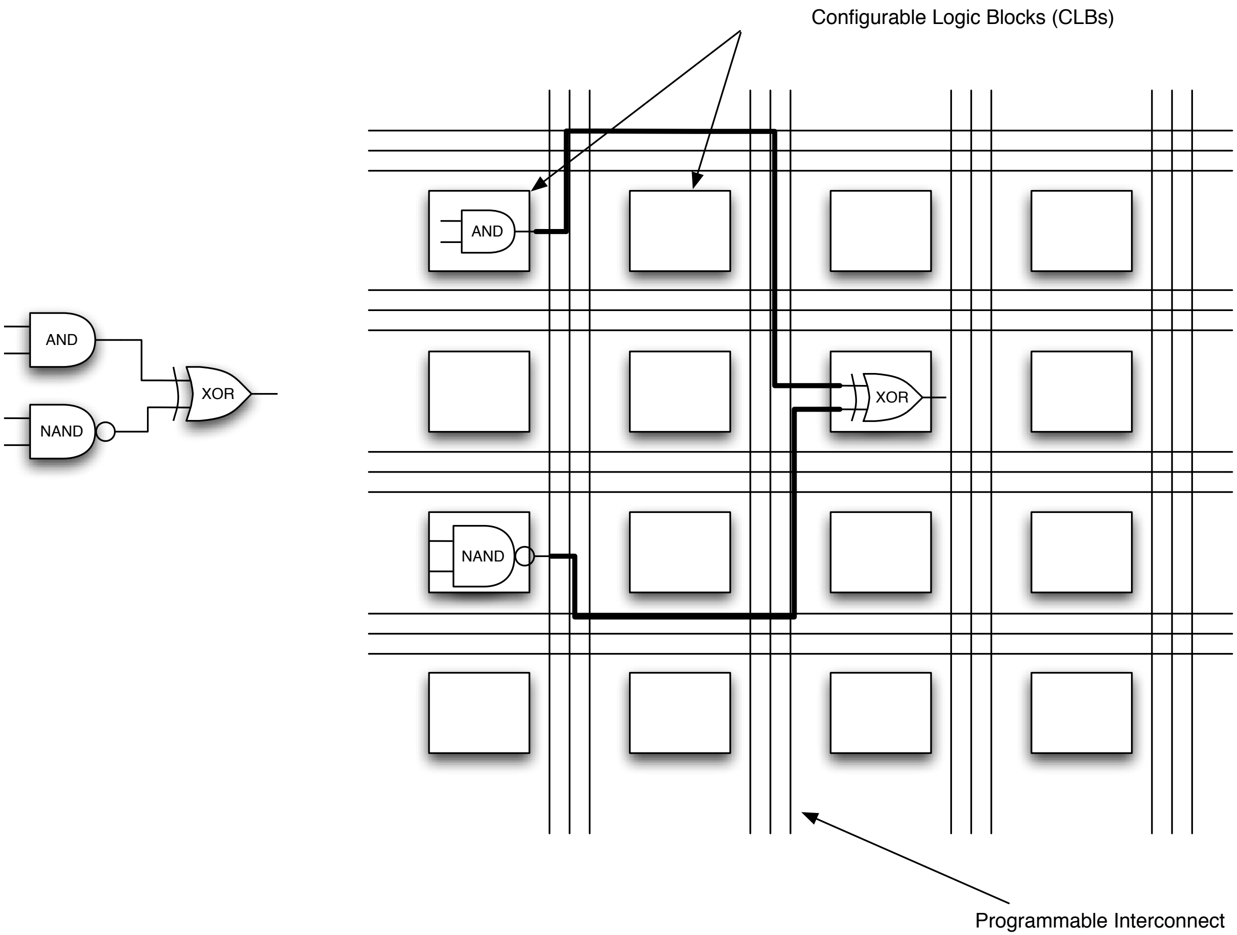
Configurable Logic Blocks

- Each of the configurable logic blocks (or logic cells) contains some lookup tables and one or more flip-flops.
- By setting the entries in the lookup tables (LUTs) these units can be programmed to implement arbitrary logical functions on their inputs.
- http://en.wikipedia.org/wiki/Field-programmable_gate_array
- ZedBoard has 85K logic cells

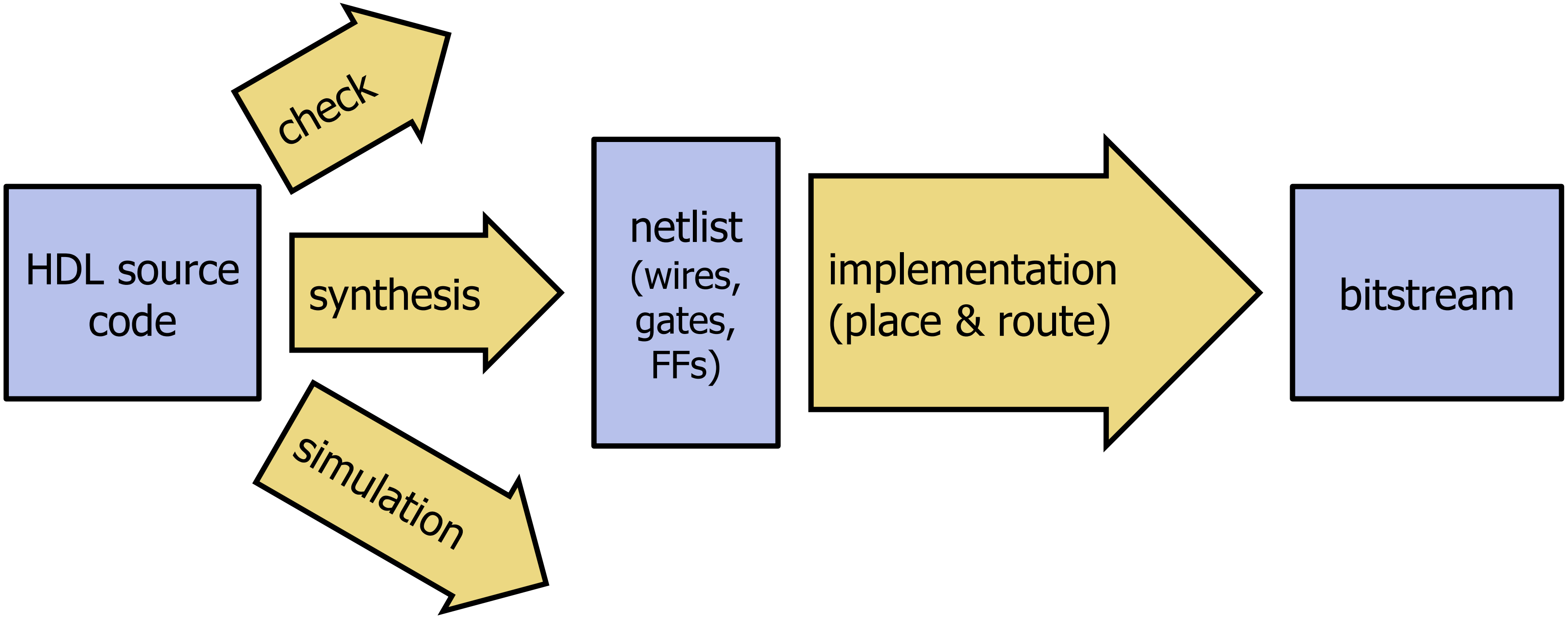


Configuring FPGAs

- By configuring the CLBs and the interconnect the FPGA can be ‘programmed’ to implement the desired operation.



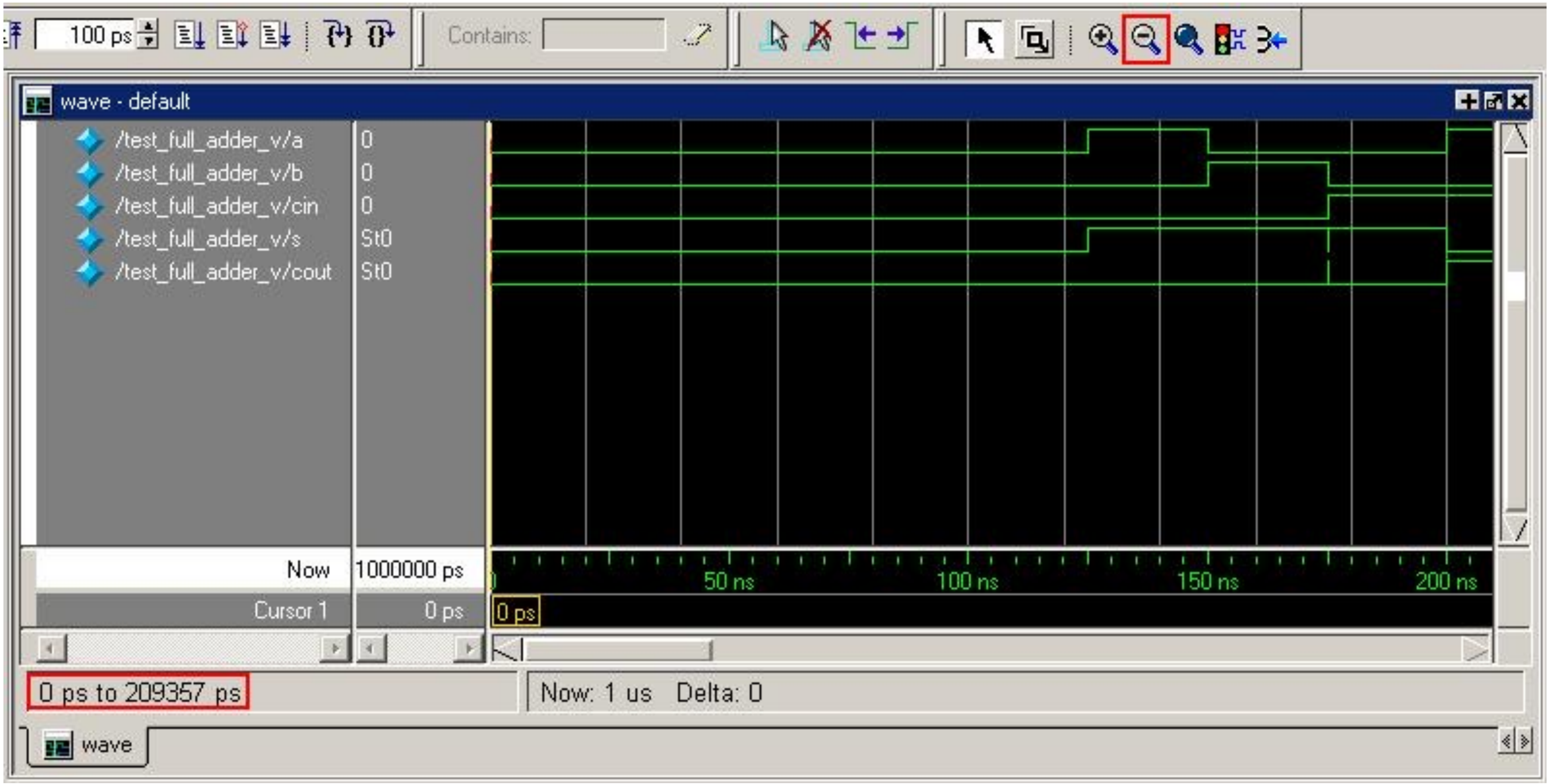
FPGA “Design Flow”



- Hardware compilers are generally **much slower** than their software counterparts
  - solving harder problems: many more choices, optimizing for area, power, picosecond-level timing

Simulation

- One way to test and debug designs
- Graphical output via waveforms



Hardware Design Methods

Hardware Design Methodologies

- Fabricating a chip requires a detailed layout
  - All transistors & wires
- How does a hardware designer describe such design?
  - (Bad) Option #1: draw all the masks “by hand”
    - All 10 billion transistors? Umm…
  - Option #2: use computer-aided design (CAD) tools to help
    - Layout done by engineers with CAD tools or automatically
- Design levels – uses **abstraction**
  - Transistor-level design – designer specifies transistors (not layout)
  - Gate-level design – designer specifics gates, wires (not transistors)
  - Higher-level design – designer uses higher-level building blocks
    - Adders, memories, etc.
    - Or logic in terms of and/or/not, and tools translates into gates

Describing Hardware

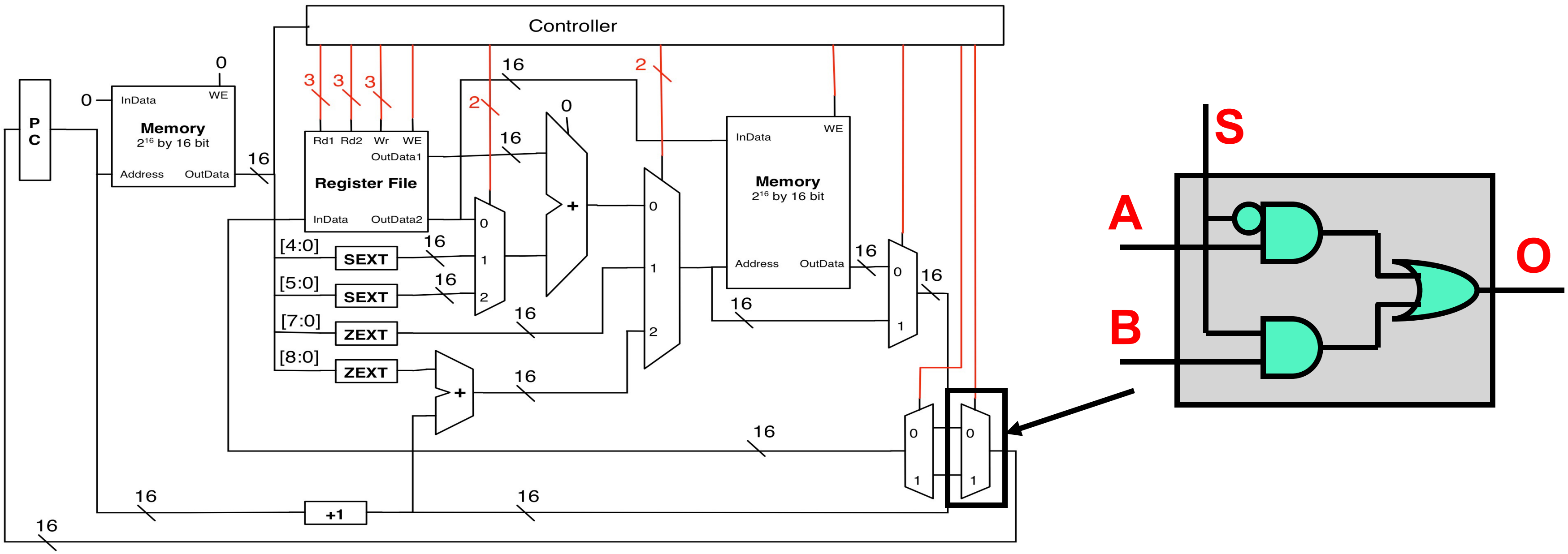
- Two general options

- Schematics
  - Pictures of gates & wires

- Hardware description languages
  - Use textual descriptions to specify hardware

- Translation process called "synthesis"
  - Textual description -> gates -> full layout
    - Tries to minimizes the delay and/or number of gates
  - Much like process of compilation of software
    - Much slower!

Schematics



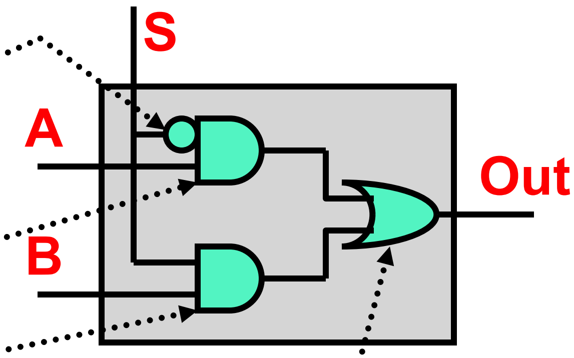
- Draw pictures
  - Use a schematic entry program to draw wires, logic blocks, gates
  - Support hierarchical design (arbitrary nesting)
  + Good match for hardware which is inherently spatial
  – Time consuming, “non-scalable” (large designs are unreadable)
  - Rarely used in practice (“real-world” designs are too big)

Hardware Description Languages (HDLs)

- Write "code" to describe hardware
  - HDL vs. SDL
  - Specify wires, gates, modules (also hierarchical)
  - \+ Easier to create, edit, modify, scales well
  - – Misleading "sequential" representation: must still "think" spatially (gets easier with practice)

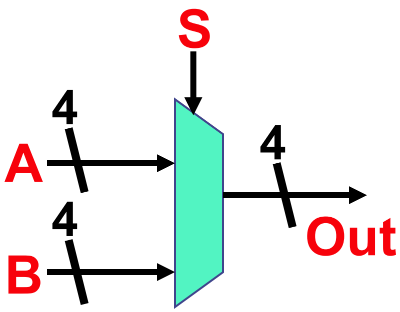
```
module mux2to1(S, A, B, Out);
   input S, A, B;
   output Out;
   wire S_, AnS_, BnS;

   not (S_, S);
   and (AnS_, A, S_);
   and (BnS, B, S);
   or (Out, AnS_, BnS);
endmodule
```



(Hierarchical) HDL Example

- Build up more complex modules using simpler modules
  - Example: 4-bit wide mux from four 1-bit muxes




```
module mux2to1_4(S, A, B, Out);
   input [3:0] A;
   input [3:0] B;
   input S;
   output [3:0] Out;

   mux2to1 mux0 (S, A[0], B[0], Out[0]);
   mux2to1 mux1 (S, A[1], B[1], Out[1]);
   mux2to1 mux2 (S, A[2], B[2], Out[2]);
   mux2to1 mux3 (S, A[3], B[3], Out[3]);
endmodule
```

Verilog HDL

- **Verilog**: HDL we will be using
  - Syntactically similar to C (by design)
  - ± Ease of syntax hides fact that this isn't C (or any software lang)
  - We will use a few lectures to learn Verilog

```
module mux2to1_4(S, A, B, Out);
   input [3:0] A;
   input [3:0] B;
   input S;
   output [3:0] Out;

   mux2to1 mux0 (S, A[0], B[0], Out[0]);
   mux2to1 mux1 (S, A[1], B[1], Out[1]);
   mux2to1 mux2 (S, A[2], B[2], Out[2]);
   mux2to1 mux3 (S, A[3], B[3], Out[3]);
endmodule
```

These aren't variables

These aren't function calls

HDLs are not "SDLs"

- SDL == Software Description Language (e.g., Java, C)
- Similar in some (intentional) ways …
  - Syntax
    - Named entities, constants, scoping, etc.
  - Tool chain: **synthesis** tool analogous to compiler
    - Multiple levels of representation
    - "Optimization"
    - Multiple targets (portability)
  - "Software" engineering
    - Modular structure and parameterization
    - Libraries and code repositories
- … but different in many others
  - One of the most difficult conceptual leaps of this course

Hardware is not Software

- Just two different beasts
  - Things that make sense in hardware, don’t in software, vice versa
  - One of the main themes of this course

- Software is sequential
  - **Hardware is inherently parallel and “always on”**
  - Have to work to get hardware to *not* do things in parallel
- Software is digital
  - Hardware has many analog properties
  - Including analog correctness properties!
- Software is mostly about functionality
  - Performance, power, area *and* functionality matter in hardware
- One reason that HDLs are not SDLs

HDL: Behavioral Constructs

- HDLs have **low-level structural** constructs
  - Specify hardware structures directly
  - Transistors, gates (**`and`**, **`not`**) and wires, hierarchy via modules
- Also have **mid-level behavioral** constructs
  - Specify operations, not hardware to perform them
  - Low-to-medium-level: **`&`**, **`~`**, **`+`**, **`*`**
- Also **higher-level behavioral** constructs
  - High-level: **`if-then-else`**, **`for`** loops
  - Some of these are synthesizable (some are not)
    - Tools try to guess what you want, often highly inefficient
    - – Higher-level $\rightarrow$ more difficult to know what it will synthesize to!
- HDLs are both high- and low-level languages in one!
  - And the boundary is not clear!

HDL: Simulation

- Another use of HDL: simulating & testing a hardware design
  - Cheaper & faster turnaround (no need to fabricate)
  - More visibility into design ("debugger" interface)

- **HDLs have features just for simulation**
  - Higher level data types: integers, FP-numbers, timestamps
  - Routines for I/O: error messages, file operations
  - Obviously, **these cannot be synthesized into circuits**

- Also another reason for HDL/SDL confusion
  - HDLs have "SDL" features for simulation

Side note: High-Level Synthesis

- Translate “C to gates”
- write hardware at a higher level of abstraction than conventional HDLs
  - greater programmer productivity
  - need to write stylized C that will synthesize well
  - tools are still slow
  - take ESE 5320 & 5390 to learn much more

Verilog HDL

HDL History

- 1970s:
  - First HDLs
- Late 1970s: VHDL
  - VHDL = VHSIC HDL = Very High Speed Integrated Circuit HDL
  - VHDL inspired by programming languages of the day (Ada)
- 1980s:
  - Verilog first introduced
  - Verilog inspired by the C programming language
  - VHDL standardized
- 1990s:
  - Verilog standardized (Verilog-1995 standard)
- 2000s:
  - Continued evolution (Verilog-2001 standard)
- Both VHDL and Verilog are evolving, still in use today

Modern HDLs

- BlueSpec
  - MIT startup from 2003
  - more functional style, richer types
  - inspired by Haskell
- Chisel
  - from Berkeley in 2012
  - embedded DSL in Scala

Verilog HDL

- Verilog is a (surprisingly) big language
  - Structural constructs at both gate and transistor level
  - Facilities for specifying memories
  - Precise timing specification and simulation
  - Lots of “behavioral” constructs
  - C-style procedural variables, including arrays
  - A pre-processor
  - VPI: Verilog programming interface
  - …

Our Verilog HDL

- **We're going to learn a focused *subset* of Verilog**
  - Focus on synthesizable constructs
  - Focus on avoiding subtle synthesis errors
  - Use as an educational tool
- For synthesis
  - Structural constructs at gate-level only
  - A few behavioral constructs
- Some testing and debugging features

Rule 1: if you haven't seen it in lecture, you can't use it!

Rule 1a: when in doubt, ask on Ed!

Basic Verilog Syntax

- Have already seen basic syntax, looks like C
  - C/C++/Java style comments
  - Names are case sensitive, and can use _ (underscore)
  - Avoid: clock, clk, power, pwr, ground, gnd, vdd, vcc, init, reset, rst
    - Some of these are "special" and will silently cause errors

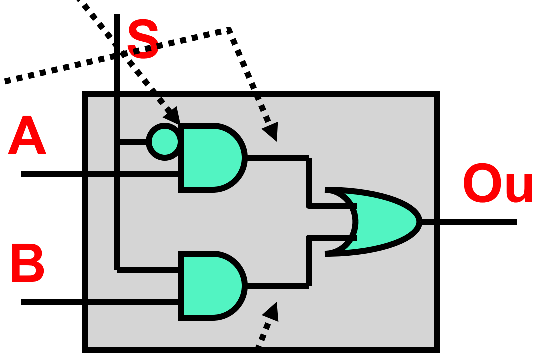
```
/* this is a module */
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
    wire S_, AnS_, BnS;
    // these are gates
    not (S_, S);
    and (AnS_, A, S_);
    and (BnS, B, S);
    or (Out, AnS_, BnS);
endmodule
```

(Gate-Level) Structural Verilog

- Primitive “data type”: **`wire`**
  - Have to declare it

Structural

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  wire S_, AnS_, BnS;
  not (S_, S);
  and (AnS_, A, S_);
  and (BnS, B, S);
  or (Out, AnS_, BnS);
endmodule
```

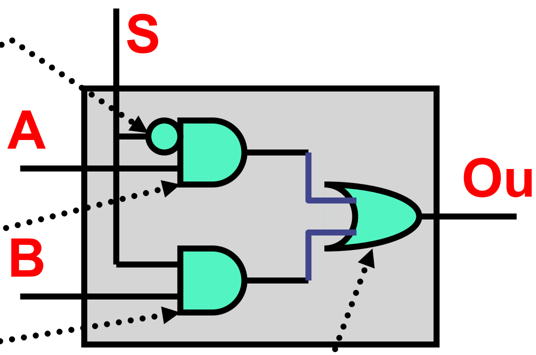


(Gate-Level) Structural Verilog

- Primitive "operators": gates
  - Specifically: **and**, **or**, **xor**, **nand**, **nor**, **xnor**, **not**, **buf**
  - Can be multi-input: e.g., **or (C, A, B, D)** (C= A | B | D)
  - "Operator" **buf** just repeats input signal (may amplify it)

Structural

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  wire S_, AnS_, BnS;
  not (S_, S);
  and (AnS_, A, S_);
  and (BnS, B, S);
  or (Out, AnS_, BnS);
endmodule
```

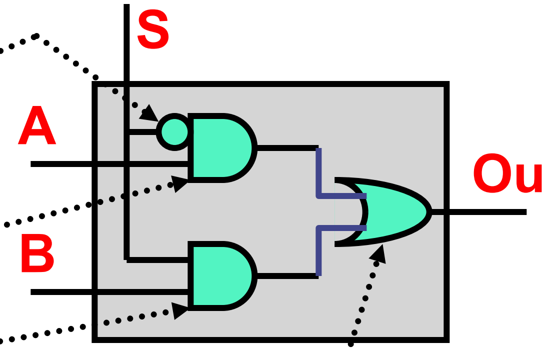


(Gate-Level) Behavioral Verilog

- Primitive “operators”: boolean operators
  - Specifically: **&**, **|**, **^**, **~**
  - Can be combined into expressions
  - Can be mixed with structural Verilog

“Behavioral” (Synthesizable)

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  wire S_, AnS_, BnS;
  assign S_ = ~S;
  assign AnS_ = A & S_;
  assign BnS = B & S;
  assign Out = AnS_ | BnS;
endmodule
```

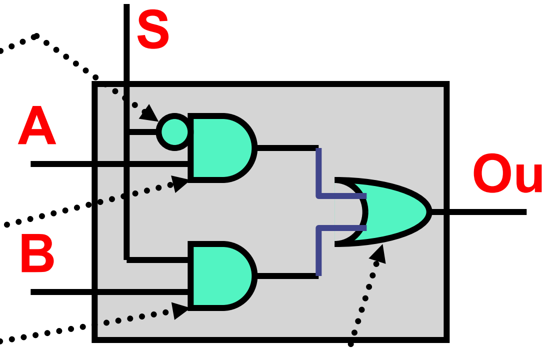


Wire Assignment

- **Wire assignment**:
  - Connect combinational logic block or other wire to wire input
  - **Order of statements not important**, executed totally in parallel
  - When right-hand-side changes, it is re-evaluated and re-assigned
  - Designated by the keyword `assign`

"Behavioral" (Synthesizable)

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  wire S_, AnS_, BnS;
  assign S_ = ~S;
  assign AnS_ = A & S_;
  assign BnS = B & S;
  assign Out = AnS_ | BnS;
endmodule
```



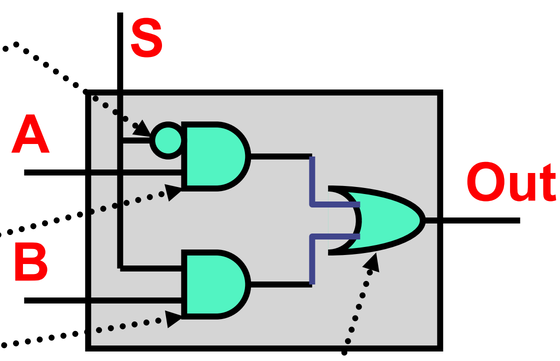
Wire Assignment

- Assignment can be combined with declaration

```
wire c = a | b;
```

"Behavioral" (Synthesizable)

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  wire S_ = ~S;
  wire AnS_ = A & S_;
  wire BnS = B & S;
  assign Out = AnS_ | BnS;
endmodule
```

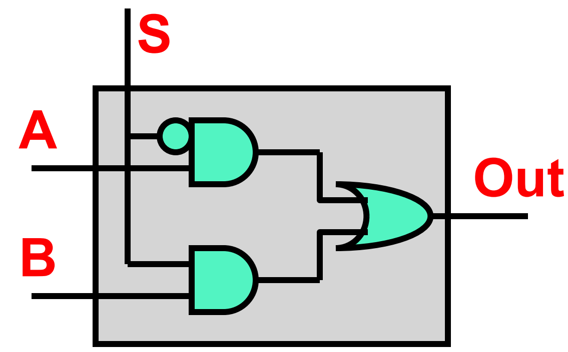


(Gate-Level) Behavioral Verilog

- Primitive "operators": boolean operators
  - Specifically: `&`, `|`, `^`, `~`
  - **Can be combined into expressions**
  - Can be mixed with structural Verilog

"Behavioral" (Synthesizable)

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  assign Out = (~S & A) | (S & B);
endmodule
```

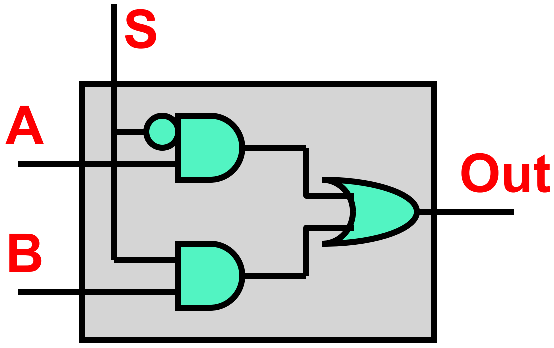


Best Way to do a Mux

- Verilog supports `?:` conditional assignment operator
  - Much more useful (and common) in Verilog than in C/Java

"Behavioral" (Synthesizable)

```
module mux2to1(input wire S,
    input wire A,
    input wire B,
    output wire Out);
  assign Out = S ? B : A;
endmodule
```



Wires Are Not C-like Variables!

- Order of assignment doesn't matter
  - This works fine

```
module mux2to1(input wire S,
  input wire A,
  input wire B,
  output wire Out);
  assign Out = AnS_ | BnS;
  assign BnS = B & S;
  assign AnS_ = A & S_;
  assign S_ = ~S;
endmodule
```

- Can't "reuse" a wire

```
assign temp = a & b;
assign temp = a | b;
```

  - Actually, you can; but doesn't do what you think it does

Wire Vectors

- Wire vectors: also called “arrays” or “buses”

```
wire [7:0] w1;  // 8 bits, w1[7] is most significant bit
wire [0:7] w2;  // 8 bits, w2[0] is most significant bit
```

- Example:

```
module mux2to1(input wire S,
  input wire [7:0] A,
  input wire [7:0] B,
  output wire [7:0] Out);
  assign Out = S ? B : A;
endmodule
```

Unlike C, array range is part of type, not variable!

- Operations
  - Bit select: vec[3]
  - Range select: vec[3:2]
  - Concatenate: `assign vec = {w, x, y, z};`

Wire and Wire Vector Constants

```
wire [3:0] w = 4'b0101;
```

- The "4" is the number of bits
- The "b" means "binary" - "h" for hex, "o" for octal, "d" for decimal
- The "0101" are the digits (in binary in this case)

```
wire [3:0] w = 4'd5; // same thing, effectively
```

- Here is a single wire constant

```
wire w = 1'b0;
```

- A useful example of wire-vector constants:

```
module mux4to1(input [1:0] Sel, input A, input B,
  input C, input D, output Out);
  assign Out = (Sel == 2'd0) ? A :
               (Sel == 2'd1) ? B :
               (Sel == 2'd2) ? C : D;
endmodule
```

Repeated Signals

- Concatenation

```
wire [2:0] vec = {x, y, z};
```

- Can also repeat a signal n times

```
wire [15:0] vec = {16{x}}; // 16 copies of x
```

- Example uses (what does this do?):

```
wire [7:0] out;
wire [3:0] A;
assign out = {{4{1'd0}}, A[3:0]};
```

- What about this?

```
assign out = {{4{A[3]}}, A[3:0]};
```

Gate-Level Vector Operators

- Verilog also supports behavioral vector operators

- **Logical bitwise and reduction**: `~,&,|,^`

```
wire [7:0] vec1, vec2;
wire [7:0] vec3 = vec1 & vec2; // bitwise AND
wire w1 = |vec1;               // OR reduction
```

- **Integer arithmetic comparison**: `+,–,*,/,%,==,!=,<,>`

```
wire [7:0] vec4 = vec1 + vec2; // vec1 + vec2
```

  - Important: all arithmetic is unsigned by default
  - Good: in signed/unsigned integers: `+`, –, `*` produces same output
    - Just a matter of interpretation
  - Bad: in signed/unsigned integers: `/ %` is not the same

Signed types

- All wires are unsigned by default
- Verilog supports signed types as well
  - changes the semantics of comparison operators **`< > <= >=`**
  - permits use of **`>>>`** arithmetic shift operator
  - useful for implementing **`CMP,SRA`** insns
- Vivado lets you intermix signed and unsigned types
  - use signed sparingly to avoid confusion

```
wire signed [3:0] s; // signed wire
assign s = …;
wire out = (s > 1) ? 1'b0 : 1'b1 ;
wire [3:0] sra = s >>> amt[1:0];
// $signed() operator on an unsigned wire
wire [3:0] u;
wire out2 = ($signed(u) > 2) ? 1'b0 : 1'b1;
```

Why Use a High-Level Operator?

- Abstraction
  - Why write assembly, when you can write C? (yay?)

- Take advantage of built-in high level implementation
  - Zedboard FPGAs have integer adders/multipliers on them
  - Xilinx will use these rather than synthesizing a multiplier from gates
    - Much faster and more efficient
  - How hard is it for Xilinx to figure out you were doing a multiply?
    - If you use "*": easy
    - If you "roll your own" using gates: nearly impossible

- Why *not* use high-level operators?
  - Less control over what they will synthesize to
  - …if they synthesize at all

Hierarchical Design using Modules

- Old-style interface specification

```
module mux2to1(Sel, A, B, Out);
  input Sel, A, B;
  output Out;
```

  - Can also have `inout`: bidirectional wire (we will not use this)
- Recommended Alternative: Verilog 2001 interfaces

```
module mux2to1(input wire Sel, A, B, output Out);
```

  A and B share same type as Sel. Convenient, but dangerous!
- Declarations
  - **Internal** wires, i.e., "locals"
  - Wires also called "nets" or "signals"

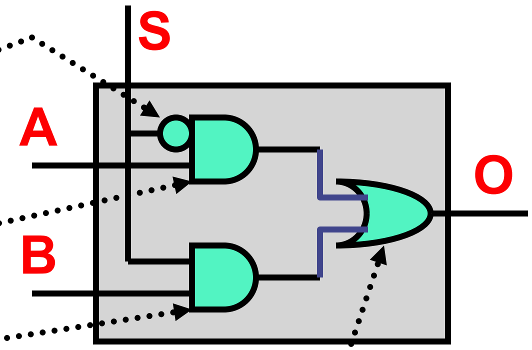
```
  wire S_, AnS_, BnS;
```

- Implementation: primitive and module instantiations

```
  and (AnS_, A, S_);
```

Verilog Module Example

```
module mux2to1(input wire Sel,
    input wire A,
    input wire B,
    output wire Out);
    wire S_, AnS_, BnS;
    not (S_, Sel);
    and (AnS_, A, S_);
    and (BnS, B, Sel);
    or (Out, AnS_, BnS);
endmodule
```



- Instantiation: `mux2to1 mux0 (cond, in1, in2, out);`
    - Non-primitive module instances must be named (helps debugging)
- Operators and expressions can be used with modules
    - `mux2to1 mux0 (cond1 & cond2, in1, in2, out);`

Hierarchical Verilog Example

- Build up more complex modules using simpler modules
- Example: 4-bit wide mux from four 1-bit muxes
  - Again, just “drawing” boxes and wires

```
module mux2to1_4(input wire Sel,
   input wire [3:0] A,
   input wire [3:0] B,
   output wire [3:0] Out);

   mux2to1 mux0 (Sel, A[0], B[0], Out[0]);
   mux2to1 mux1 (Sel, A[1], B[1], Out[1]);
   mux2to1 mux2 (Sel, A[2], B[2], Out[2]);
   mux2to1 mux3 (Sel, A[3], B[3], Out[3]);
endmodule
```

Connections via Named Association

- ALWAYS specify module connections by name
  - Like named parameters/keyword arguments in SDLs
  - Helps keep the bugs away
  - Bad example

```
mux2to1 mux0 (res, s, a, b);
```

  - Good example

```
mux2to1 mux0 (.Sel(s), .A(a), .B(b), .Out(res));
```

  - Also, order becomes irrelevant

```
mux2to1 mux1 (.A(a), .B(b), .Out(res), .Sel(s));
```

Generate construct

- basic metaprogramming to reduce repetition
- `for` loop must have a fixed bound
- think of this as “copy-and-paste”, **not** a SDL for loop

```
module mux2to1_4(input wire Sel,
    input wire [3:0] A,
    input wire [3:0] B,
    output wire [3:0] Out);
  genvar i;
  for (i = 0; i < 4; i = i+1) begin
    mux2to1 m(.Sel(Sel),.A(A[i]),.B(B[i]),
              .Out(Out[i]));
  end
endmodule
```

Per-Instance Module Parameters

- Module parameters: useful for defining varying bus widths

```
module Nbit_mux2to1 (input wire Sel,
  input wire  [N-1:0] A,
  input wire  [N-1:0] B,
  output wire [N-1:0] Out);
   parameter N = 1;
   assign Out = Sel ? B : A;
endmodule
```

- Two ways to instantiate: implicit

```
Nbit_mux2to1 #(4) mux1 (.Sel(S),.A(in1),.B(in2),.Out(out));
```

- And explicit

```
 Nbit_mux2to1 mux1 (.Sel(S),.A(in1),.B(in2),.Out(out));
 defparam mux1.N = 4;
```

- Multiple parameters per module allowed

Per-Instance Module Parameters

- localparam: a parameter that is only visible within a module
  - a constant, scoped to that module

Getting Fancy: Generate and Parameters

```
module rca
  #(parameter N = 4)
   (input wire [N-1:0] a,
    input wire [N-1:0] b,
    output wire [N-1:0] s);
   wire [N:0]          carry;
   assign carry[0] = 1'b0;

   genvar i; // skip optional "generate" keyword
     for (i=0; i<N; i = i+1) begin
       fulladder f(.cin(carry[i]),.a(a[i]),.b(b[i]),
                   .s(s[i]),.cout(carry[i+1]));
     end
endmodule
```

Wire Arrays

- Verilog supports multi-dimensional wire vectors
  - Useful with generate loops when you need lots of buses

```
wire[15:0] y[7:0]; // eight 16-bit buses
y[0]; // first 16-bit bus
y[0][0]; // first bit of the first bus
```

Verilog Pre-Processor

- Like the C pre-processor
  - But uses ` (back-tick) instead of **#**
  - Constants: **`define**
    - No parameterized macros
    - Use ` before expanding constant macro

```
`define letter_A 8'h41
wire w[7:0] = `letter_A;
```

  - Conditional compilation: **`ifdef**, **`endif**
  - File inclusion: **`include**

- Parameter vs `define
  - A parameter is scoped to a module instance
  - A `define is scoped to a file (potentially across modules)
  - Can use parameters as constants

Verilog Errata

- Wires have binary values: 0 or 1
  - except when they don't: **`x`** and **`z`** are undefined values
- No particular naming convention for modules and their files
  - Unlike, say, Java public classes
- No "imports" or "libraries"
  - There are ways to do this, to integrate 3rd-party Intellectual Property (IP) into your design. We won't explore this.

Verilog testing constructs

- **integer**, **reg** types
  - correspond to storage (unlike **wire** which is stateless)
  - DO NOT use these outside of testing!
    - we'll see this later: **reg** will synthesize into a latch/FF
- delay statement
  - **#10;** means "wait 10 cycles"
- **for/while** loops
  - allow iterating over test inputs
- **$display()**
  - printf-like output
  - $display("wire was %b", a);
- **$finish**
  - ends the simulation

Sequential Logic

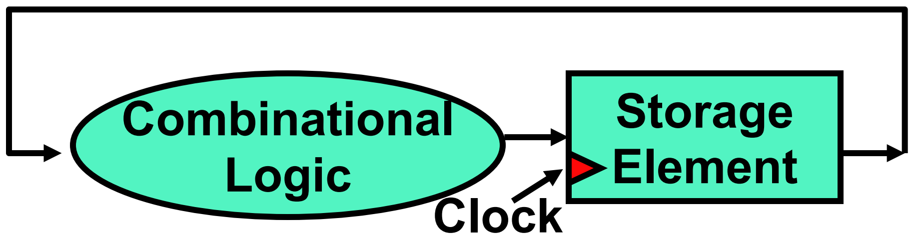
Two Types of Digital Circuits

- Combinational Logic
  - Logic **without state** variables
  - Examples: adders, multiplexers, decoders, encoders
  - **No clock involved**

- Sequential Logic
  - Logic **with state** variables
  - State variables: latches, flip-flops, registers, memories
  - **Clocked**
  - State machines, multi-cycle arithmetic, processors

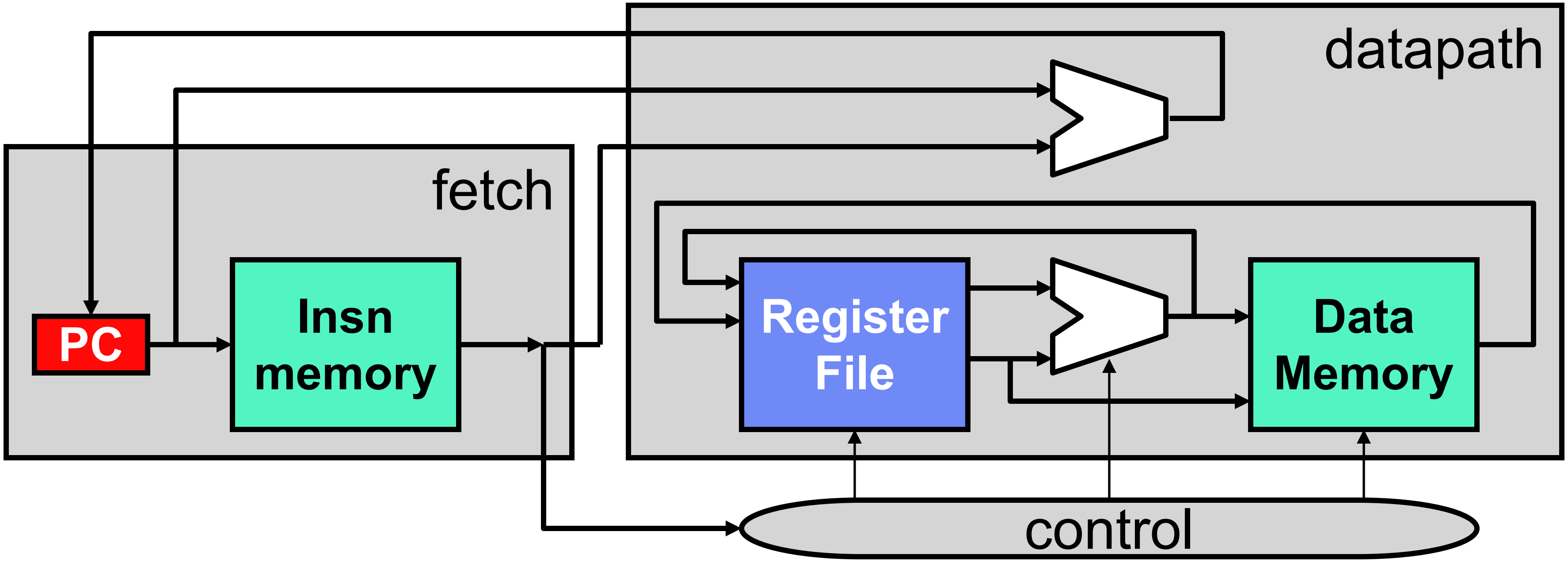
- Sequential Logic in Verilog
  - Special idioms using behavioral constructs that synthesize into latches, memories

Sequential Logic & Synchronous Systems



- Processors are complex fine state machines (FSMs)
  - Combinational (compute) blocks separated by storage elements
    - State storage: memories, registers, etc.
- **Synchronous systems**
  - **Clock**: global signal acts as write enable for all storage elements
    - Typically marked as triangle
  - All state elements write together, values move forward in lock-step
  - \+ Simplifies design: design combinational blocks independently
- Aside: asynchronous systems
  - Same thing, but … no clock
  - Values move forward using explicit handshaking
  - ± May have some advantages, but difficult to design

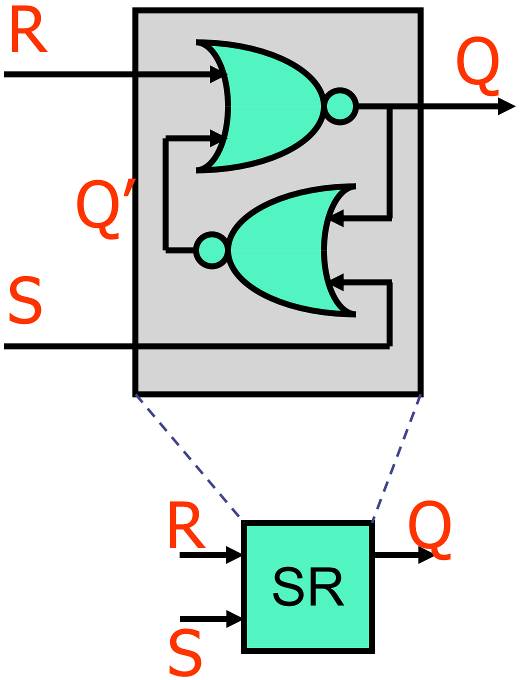
Datapath Storage Elements



- Three main types of storage elements
  - **Singleton registers**: PC
  - **Register files**: ISA registers
  - **Memories**: insn/data memory

S-R Latch

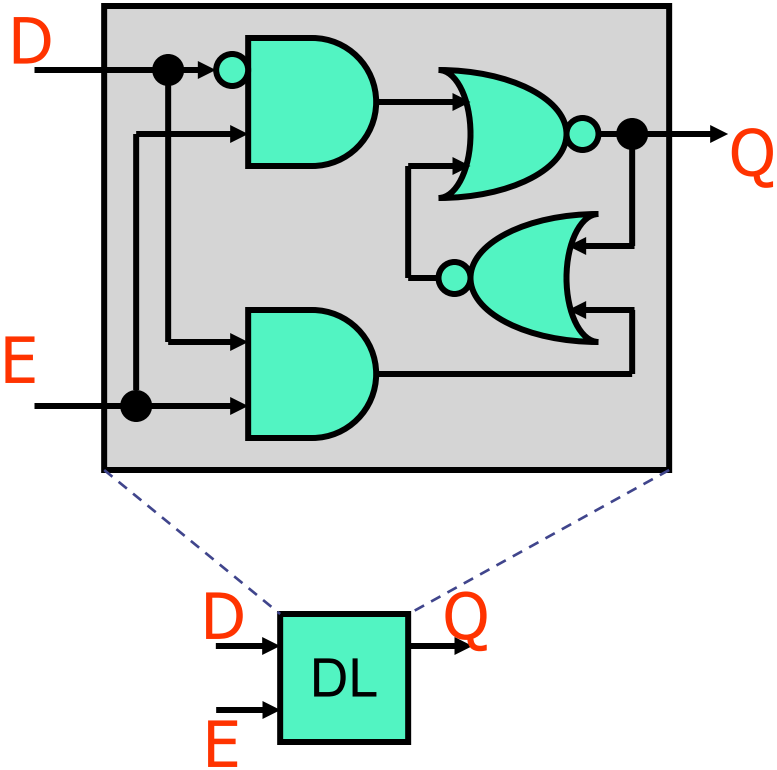
- **S-R (set-reset) latch**
  - Cross-coupled NOR gates
  - Distinct inputs/outputs




```
S,R → Q
0,0 → oldQ
0,1 → 0
1,0 → 1
1,1 → 0
```

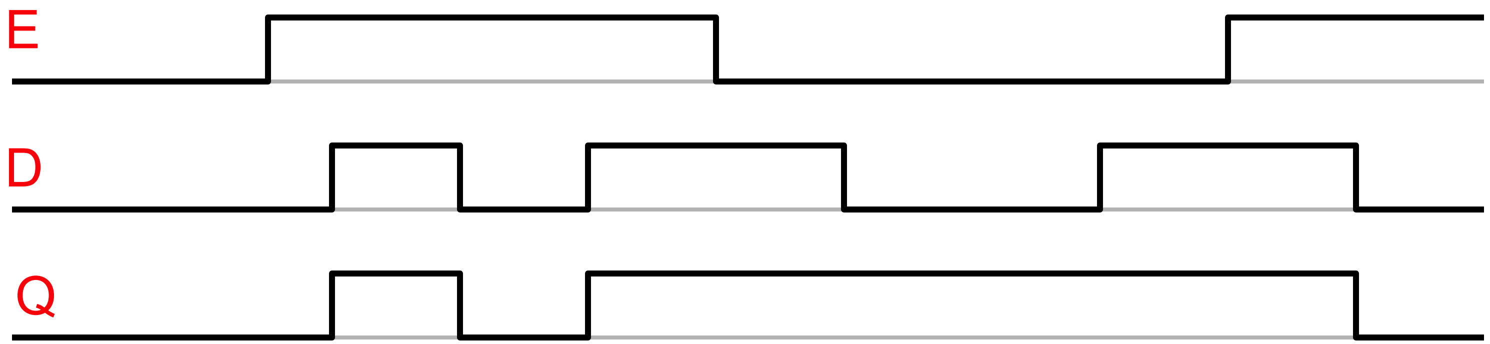
  - S=0, R=0? circuit degenerates to cross-coupled INVs
  - S=1, R=1? reset has "higher priority"
  - Not really used … except as component in something else

D Latch

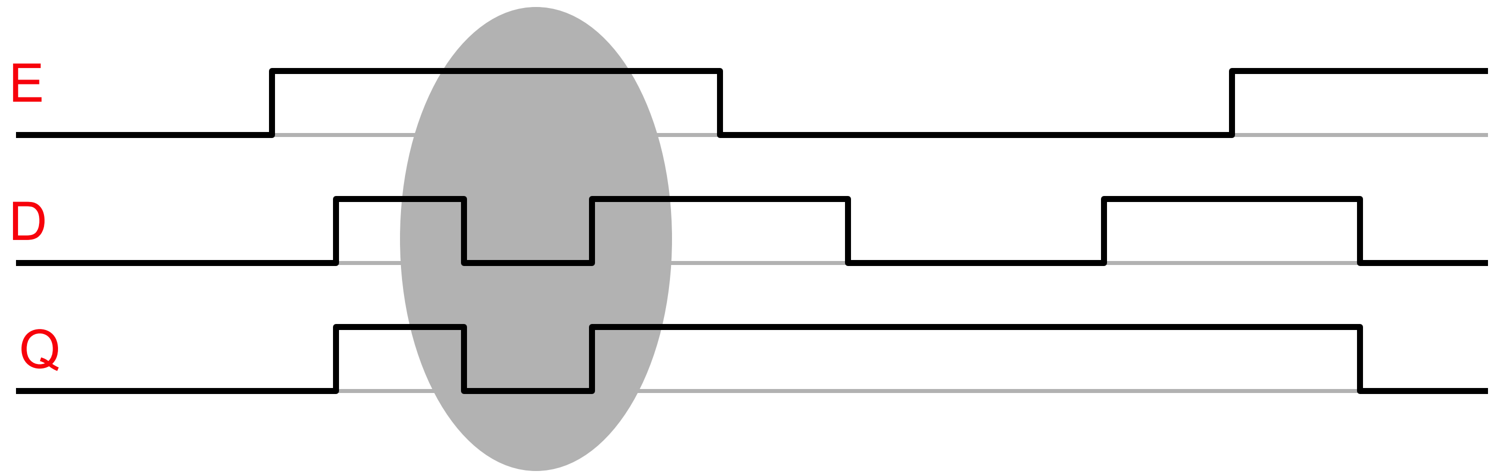


Timing Diagrams

- Voltage {0,1} diagrams for different nodes in system
  - “Digitally stylized”: changes are vertical lines (instantaneous?)
  - Reality is analog, changes are continuous and smooth
- Timing diagram for a D latch



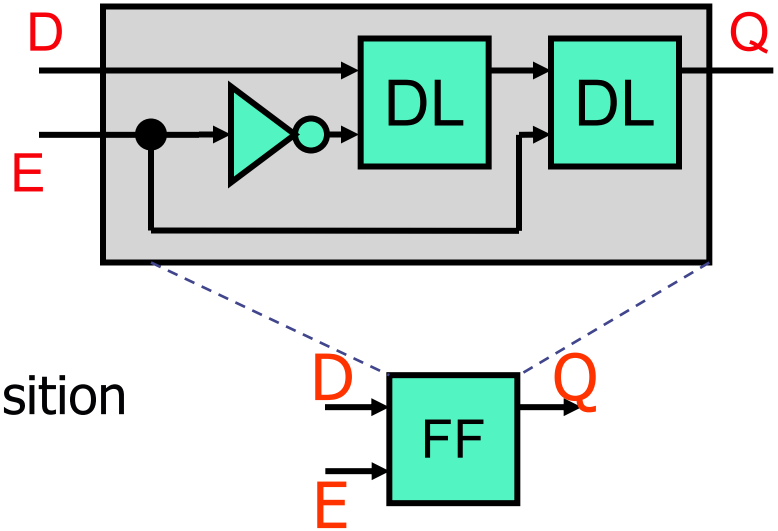
Triggering: Level vs. Edge

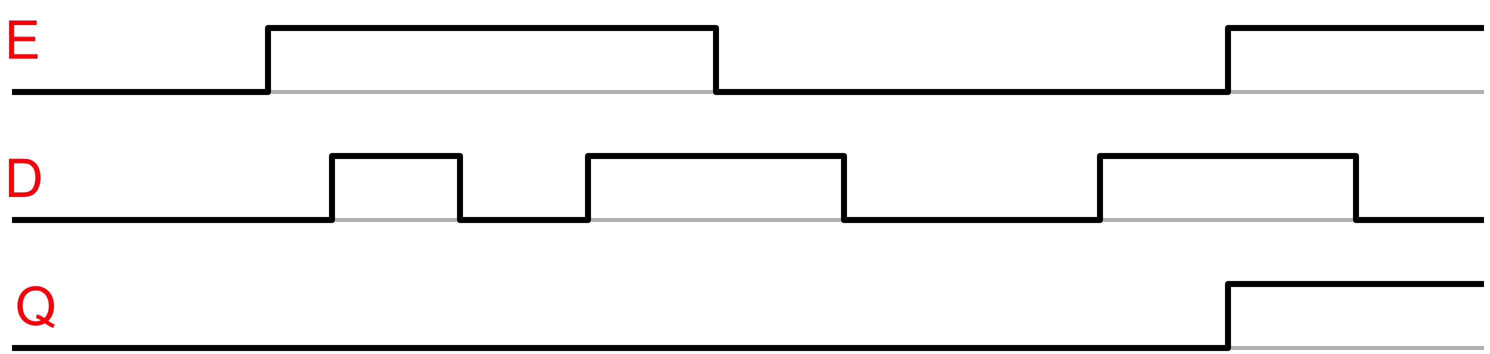


- The D-latch is **level-triggered**
  - The latch is open for writing as long as E is 1
  - If D changes continuously, so does Q
  - – Pretty hard to work with
- Much easier to use an **edge-triggered** latch
  - The latch is open for writing only on E transition (0 → 1 or 1 → 0)
  - + Don't need to worry about fluctuations in value of D

D Flip-Flop

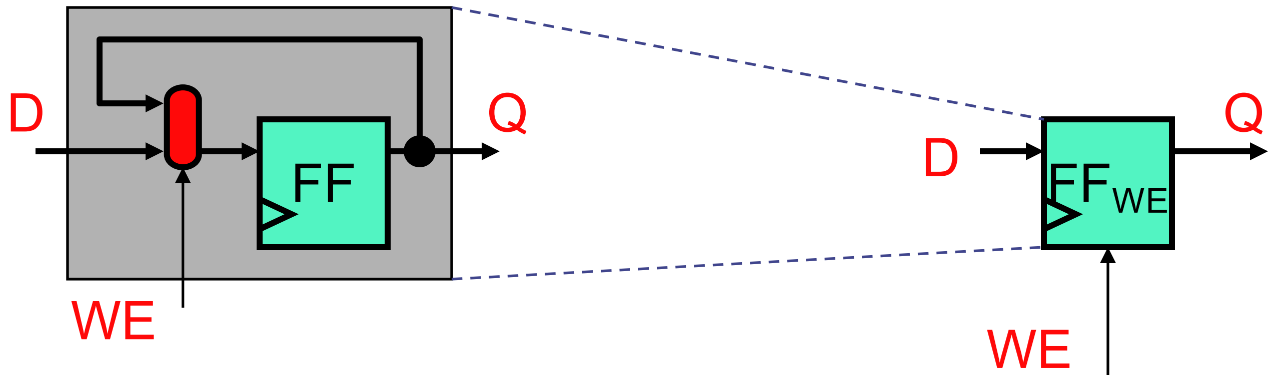
- **D Flip-Flop**:
  - Sequential D-latches
  - Enabled by inverse signals
  - First latch open when E = 0
  - Second latch open when E = 1
  - Overall effect?
    - D flipflop latches D on 0→1 transition
  - E is the "clock" signal input





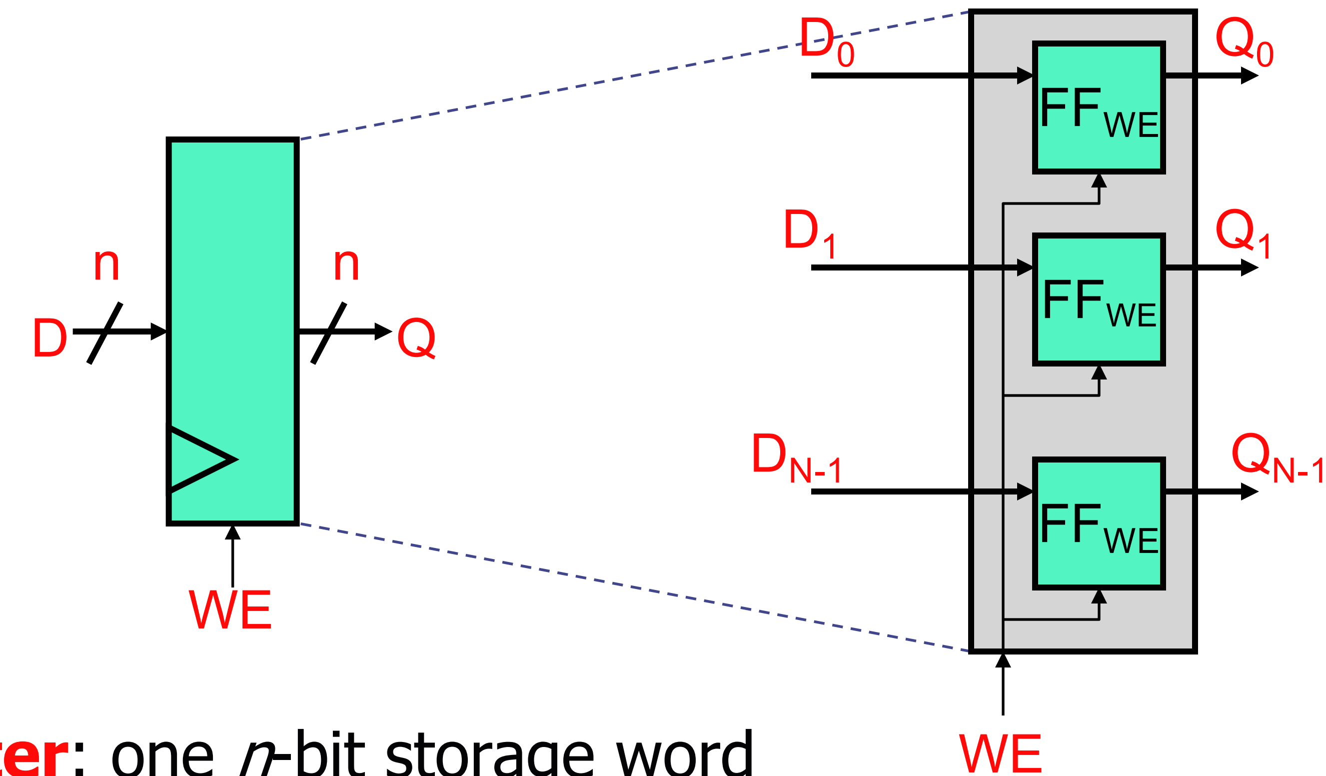
FF_{WE}: FF with Separate Write Enable

- **FF_{WE}**: FF with separate write enable
  - FF D(ata) input is MUX of D and Q, WE selects



  - Bad idea: why not just AND the CLK and WE?
    + Fewer gates
    – Creates timing problems
      - Do not try to do logic on CLK in Verilog
      - No, really. Never do this.

N-bit Register

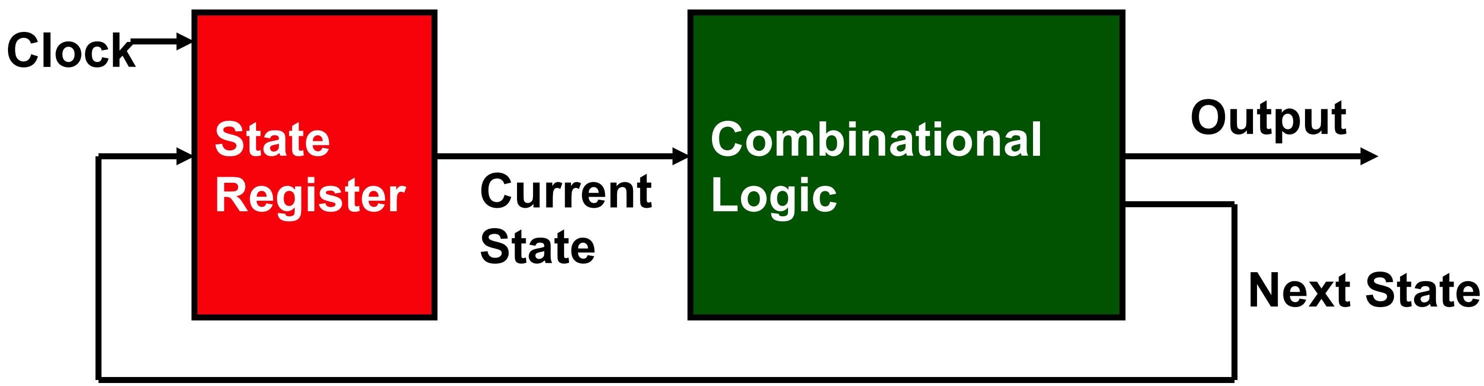


- **Register**: one *n*-bit storage word
  - Non-multiplexed input/output: data buses write/read same word
- Implementation: FF_{WE} array with shared write-enable (WE)
  - FFs written on CLK edge if WE is 1 (or if there is no WE)

Sequential Logic in Verilog

Designing Sequential Logic

- key design rule: separate combinational logic from sequential state elements
  - Not enforced by Verilog, but a very good idea
  - Possible exceptions: counters, shift registers
- We'll give you a flip-flop module (see next slide)
  - Edge-triggered, not a transparent latch
  - Parameterized to create an *n*-bit register
- Example use: state machine

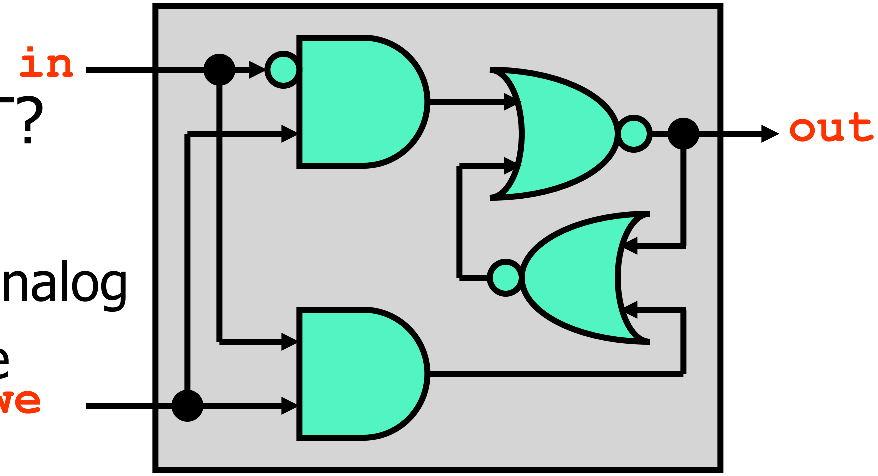


Sequential Logic In Verilog

- How are state-holding variables specified in Verilog?
  - First instinct: structurally
  - After all, real latches and flip-flops are made from gates…

```
module latch(out, in, we);
    output out; input in, we;
    wire not_out = ~(out | (we & ~in));
    assign out = ~(not_out | (we & in));
endmodule
```

- This should work, right? RIGHT?
  - Logically, yes… in practice, no
    - Storage elements are highly analog
    - FPGAs have dedicated storage



Verilog Flipflop (Behavioral Magic)

- How do we specify state-holding constructs in Verilog?

```
module dff (output wire out,
   input wire in, input wire writeEnable,
   input wire reset, input wire clock);

  reg out;
  always @(posedge clock)
    begin
      if (reset)
        out = 0;
      else if (writeEnable)
        out = in;
    end
endmodule
```

- **reg**: storage bit
- **always @ ()**: synthesizable behavioral sequential Verilog
  - Tricky: hard to know exactly what it will synthesize to
  - **We will give this to you, don't write your own**
  - "Creativity is a poor substitute for knowing what you're doing"

Verilog Register (Behavioral Magic)

- How do we specify state-holding constructs in Verilog?

```
module register (output wire [n-1:0] out,
   input wire [n-1:0] in, input wire writeEnable,
   input wire reset, input wire clock);
   parameter n = 1;

   reg [n-1:0] out;
   always @(posedge clock)
     begin
       if (reset)
         out = 0;
       else if (writeEnable)
         out = in;
     end
endmodule
```

- **reg**: interface-less storage bit
- **always @ ()**: synthesizable behavioral sequential Verilog
  - Tricky: hard to know exactly what it will synthesize to
  - **We will give this to you, don't write your own**
  - "Creativity is a poor substitute for knowing what you're doing"

Clock Signals

- Clocks & reset signals are ***not*** normal signals
- Travel on dedicated “clock” wires
  - Reach all parts of the chip
  - Special “low-skew” routing
- Ramifications:
  - **Never** do logic operations on the clocks
  - If you want to add a “write enable” to a flip-flop:
    - Use a mux to route the old value back into it
    - (or use the flip-flop with write enable we give you!)
    - Do not just “and” the write-enable signal with the clock!
- Messing with the clock can cause a errors
  - Often can only be found using detail low-level simulation

What does <…> do in Verilog?

- https://www.edaplayground.com
  - web-based Verilog editor+simulator
  - recommended simulator: Icarus Verilog 0.10.0
- https://hdlbits.01xz.net/wiki/Main_Page
  - online Verilog problem sets

Testbenches

- A more effective way to test & debug designs

- In Java?
  - Write test code in Java to test Java
  - "Test harness", "unit testing"

- For Verilog?
  - Write test code in Verilog to test Verilog
  - Verilog has advanced "behavioral" commands to facilitate this:
    - Delay for *n* units of time
    - Full high-level constructs: if, while, sequential assignment, ints
    - Input/output: file I/O, output to display, etc.

Common Errors

- Tools are from a less civilized time
  - More like C, less like Java
  - Assume that you mean what you say
- Common errors:
  - Not assigning a wire a value
  - Assigning a wire a value more than once
  - Implicit wire declarations (default to type "wire" 1-bit wide)
  - Mis-matched wire assignment widths
  - Combinational loops
- Avoid names such as:
  - clock, clk, power, pwr, ground, gnd, vdd, vcc, init, reset, rst
  - Some of these are "special" and will silently cause errors

Official Vivado Verilog Reference

XILINX
ALL PROGRAMMABLE.

from *Vivado Design Suite User Guide: Synthesis* UG901 (v2017.4)

List of Verilog keywords

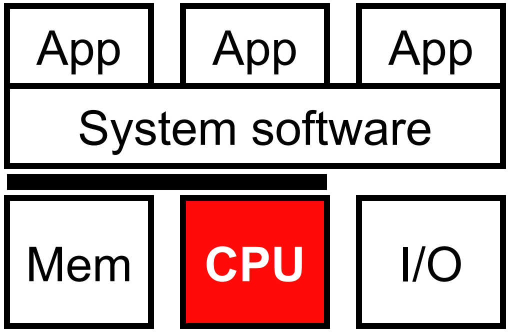
always	endspecify	medium	rnmos	vectored
and	endtable	module	rpmos	wait
assign	endtask	nand	rtran	wand
automatic	event	negedge	rtranif0	weak0
begin	for	nmos	rtranif1	weak1
buf	force	nor	scalared	while
bufif0	forever	noshow-cancelled*	show-cancelled*	wire
bufif1	fork	not	signed	wor
case	function	notif0	small	xnor
casex	generate	notif1	specify	xor
casez	genvar	or	specpa	
cell*	highz0	output	strong0	
cmos	highz1	parameter	strong1	
config*	if	pmos	supply0	
deassign	ifnone	posedge	supply1	
default	incdir*	primitive	table	
defparam	include*	pull0	task	
design*	initial	pull1	time	
disable	inout	pullup*	tran	
edge	input	pulldown*	tranif0	
else	instance*	pulsestyle_ondetect*	tranif1	
end	integer	pulsestyle_onevent*	tri	
endcase	join	rcmos	tri0	
endconfig*	larger	real	tri1	
endfunction	liblist*	realtime	triand	
endgenerate	library*	reg	trior	
endmodule	localparam	release	trireg	
endprimitive	macromodule	repeat	use*	

from Chapter 7 of *Vivado Design Suite User Guide: Synthesis* UG901 (v2017.4)

Additional Verilog Resources

- Elements of Logic Design Style by Shing Kong, 2001
  - Do's, do-not's, other tips
  - http://www.cis.upenn.edu/~milom/elements-of-logic-design-style/
- Verilog HDL Synthesis: A Practical Primer
  - By J. Bhasker, 1998
  - To the point (<200 pages)
- Advanced Digital Design with the Verilog HDL
  - By Michael D. Ciletti, 2003
  - Verilog plus lots of digital logic design (~1000 pages)

Summary



- Transistors & fabrication
- Digital logic basics
  - Focus on useful components
- Hardware design methods
  - Introduction to Verilog

- Next unit: fast arithmetic