
[CHICAGO] CODER CONFERENCE
INNOVATION. THOUGHT LEADERSHIP. TRAINING.
Design your own computer chips at home. Seriously.
Matt DeFano – STA Group
github.com/defano/digital-design
Monday, June 26, 2017 • 10:00am – 12:15pm • Room 208

Matt DeFano

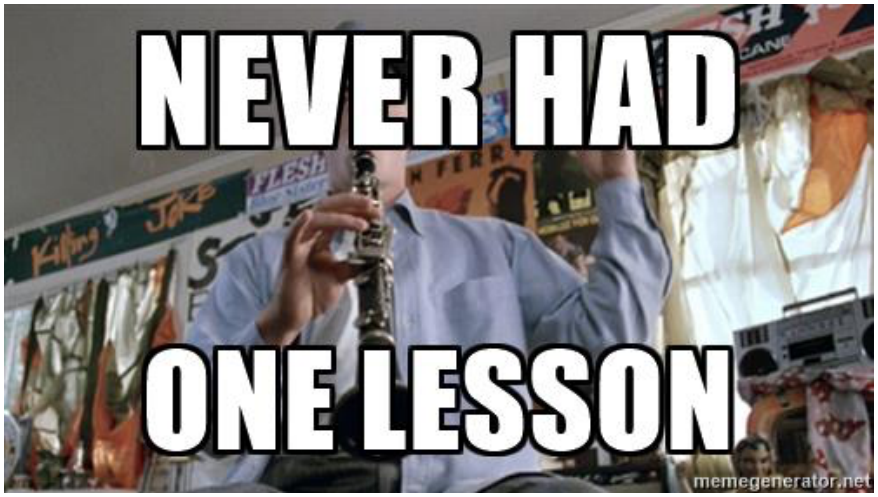
Sr. Architect, Manager

IOT TECHNOLOGY SOLUTIONS

An STA Group Company

222 South Riverside Drive
Suite 2800
Chicago, IL 60606

matt.defano@iottechnology.com
www.iottechnology.com

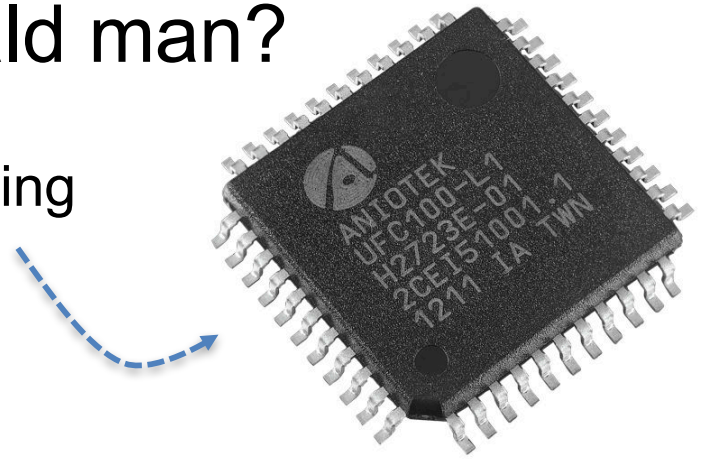


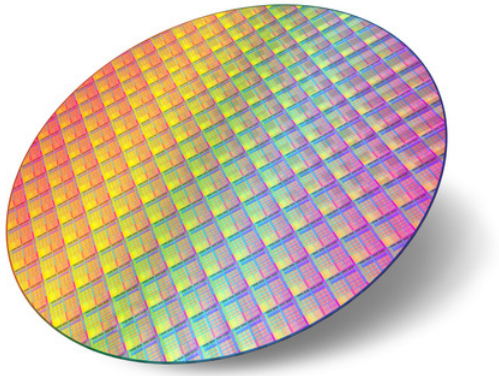
- ✓ Not an electrical engineer
- ✓ Never studied electronics
- ✓ Flunked algebra. Also, calculus.

🤔 Okay, I was employed as an ASIC / FGPA engineer for a Bay Area telecom startup.

What's the point of this talk, bald man?

- Help you understand what goes into creating a digital integrated circuit
- Introduce you to Verilog, a hardware description language used to design them
- Show nerds ("enthusiasts") how they can experiment at home with hobbyist hardware kits
- **Hands-on**: design and simulate a chip to drive LEDs in a "Knight Rider" pattern



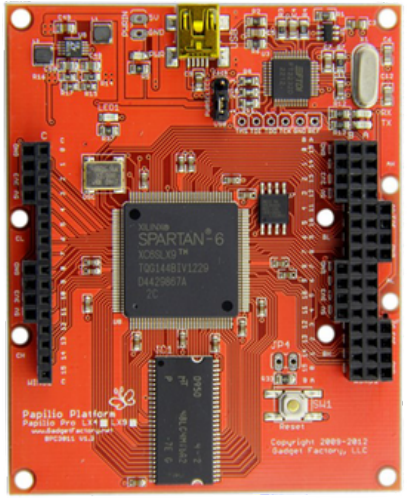


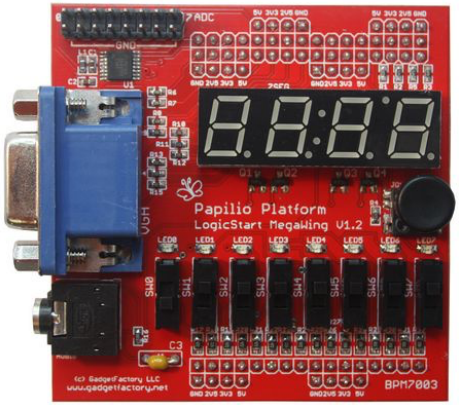
But we **can't cover everything.** We'll skip these more advanced topics:

- How chips are physically fabricated (manufacturing process)
- How to design analog or mixed-signal integrated circuits (chips containing wireless radios, micro-electro-mechanical systems, etc.)
- How to design complex digital circuits that involve multiple clock domains, PLL-synthesized clocks, or SerDes'
- Each and every aspect of the Verilog language (we'll focus on the constructs you're most likely to use)

Raspberry Pi is for kids. **Design integrated circuits** at home.

- The Papilio platform is an FPGA development kit for makers / hobbyists (others are on the market, too)
- Main board contains FPGA and / or CPU; auxiliary boards (called “wings”) provide various IO options (buttons, switches, displays, etc.)
- Can I use this to invent my own computer architecture? Of course!
- Setups cost ~$100 at gadgetfactory.net





While not strictly required, **some background** will be helpful…

- ✓ **You're Boolean bitwise**
  You understand truth tables, the behavior of logical operators (AND, OR, XOR, etc.), and bitwise expressions (i.e., `a & ~b | c`)
- ✓ **You know number bases and SI units**
  Never heard of binary or hexadecimal? That might be a problem. Milli, micro, nano, pico, femto…?
- ✓ **You've had some experience writing code**
  Ideally in a C-like language (Java, JavaScript, ObjectiveC, Swift, Perl, etc.)
- ✓ **You're comfortable with Linux (and a text editor)**
  Windows (and to a lesser degree, macOS) will work, but lab instructions are targeted at Ubuntu.

Lets get started with a quick refresher.

Information can be **represented numerically**.

Uppercase		Lowercase	
65	A	97	a
66	B	98	b
67	C	99	c
68	D	100	d
69	E	101	e
70	F	102	f
71	G	103	g
72	H	104	h
73	I	105	i
74	J	106	j

Hello *… your text message*

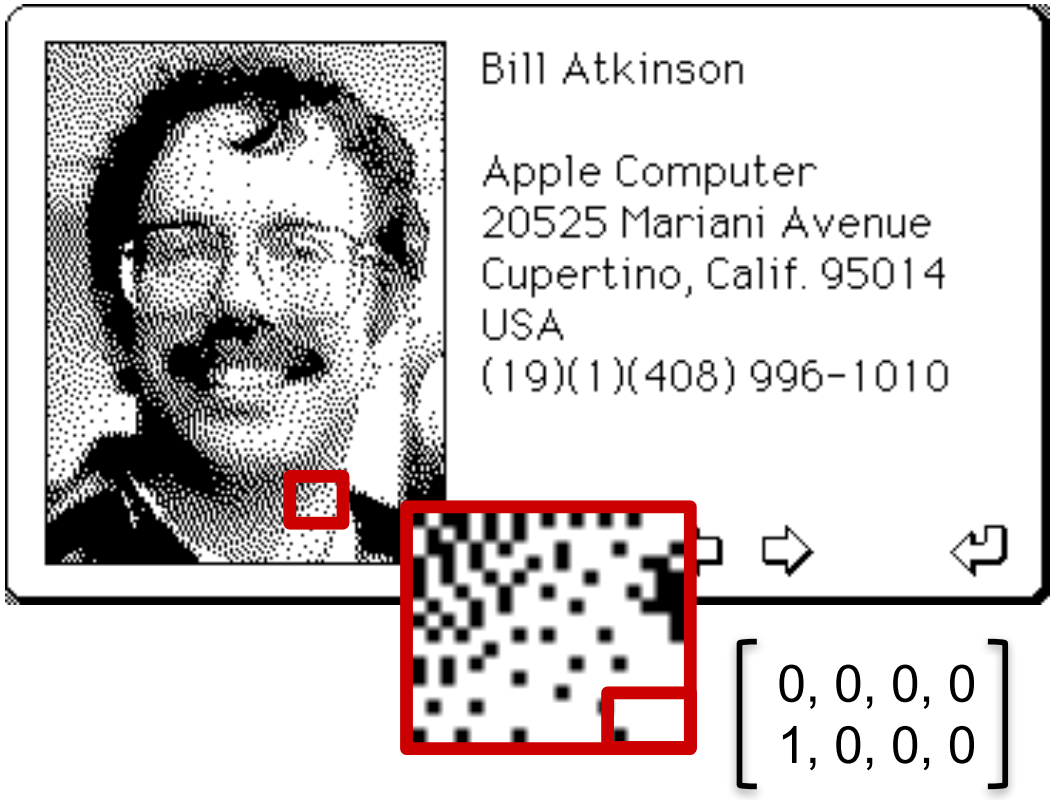
72, 101, 108, 108, 111 *… as decimal*

0x48, 0x65, 0x6c, 0x6c, 0x6f *… as hex*

01001000, 01101001, 01101100, 01101100, 10101111 *… as binary*

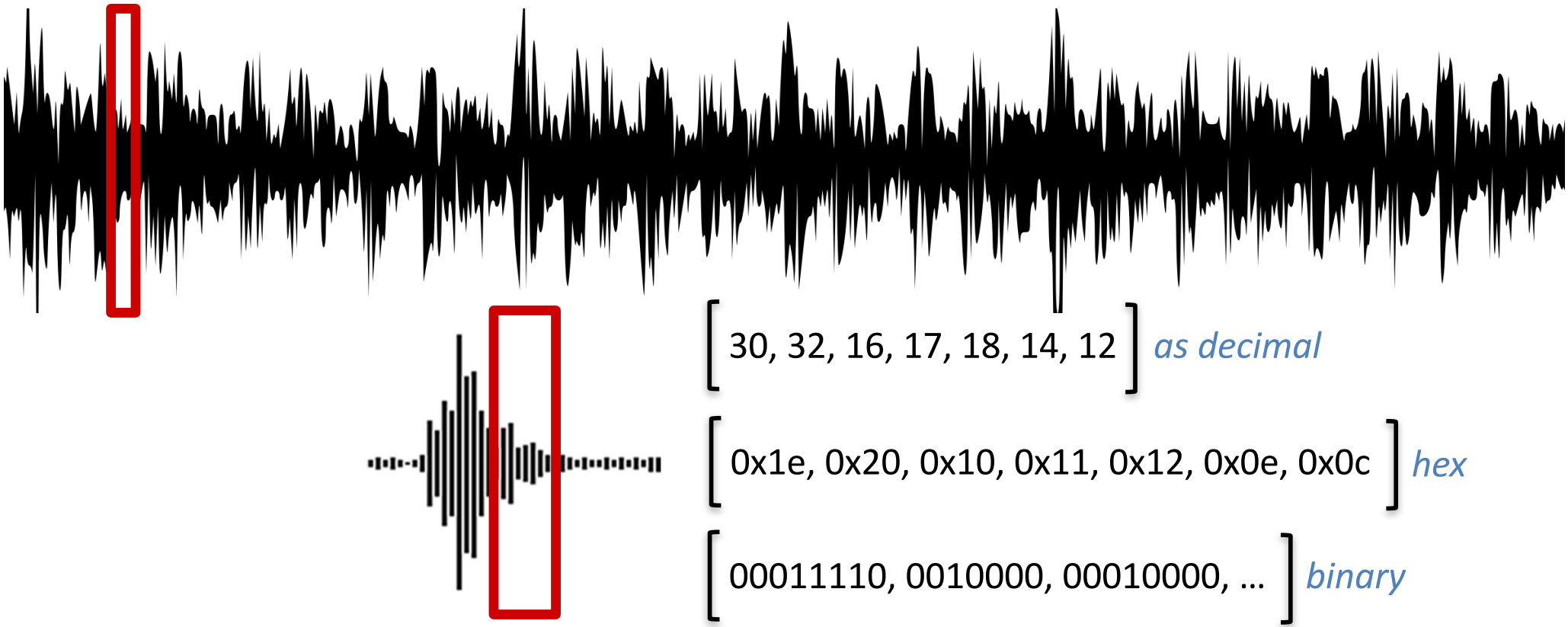
The American Standard Code for Information Interchange (ASCII), circa 1960s

Even information that **doesn't seem numerical**.



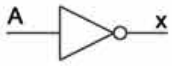
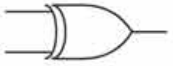
Rasterized "bitmap" image from HyperCard, circa 1987.

… **no matter how complex** the information.



Pulse code modulation (PCM), circa 1920 (used in compact discs).

Binary data **can be manipulated** with *Boolean algebra* and drawn as a network of *logic gates*.

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
Symbol							

Truth Table

A	X
0	**1**
1	**0**

B	A	X
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

B	A	X
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

B	A	X
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

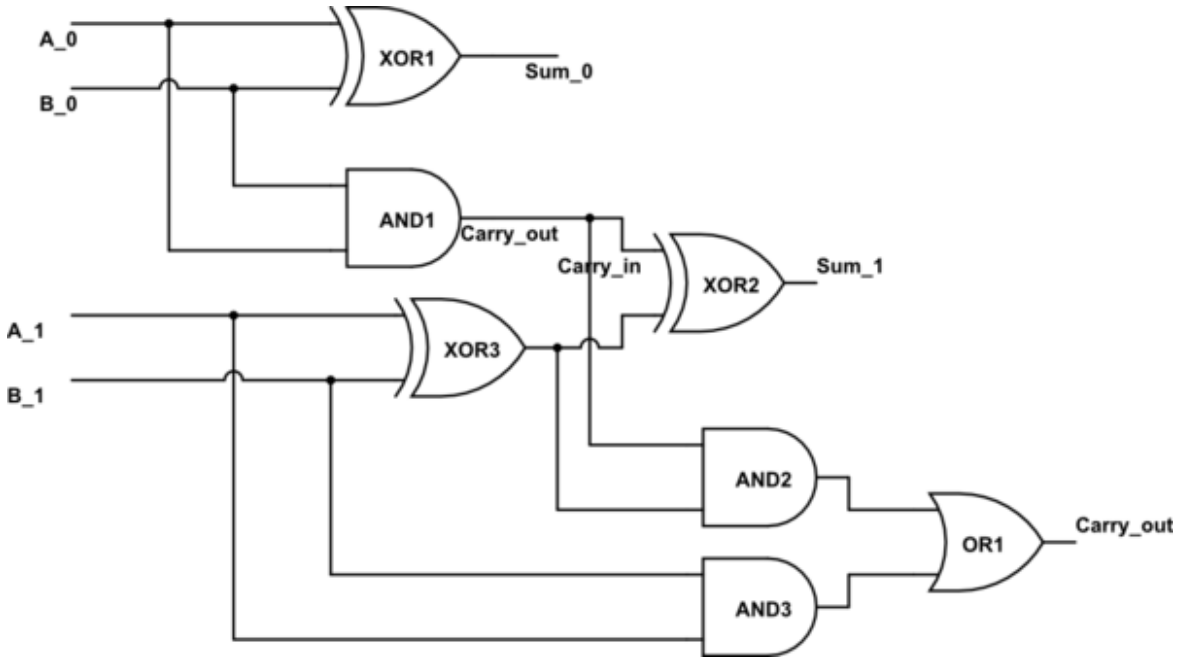
B	A	X
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

B	A	X
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

B	A	X
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

Proposed by George Boole circa 1847; you slept through this in college philosophy.

Logic gates can be arranged to **produce any transformation** of input to output.



Example of a circuit to add two-bit integers.

Every point in a digital circuit exists in **only one of two states**: 0 or 1.

State	Physical Representation
0 False, de-asserted	**Low Voltage** (usually 0v, but depends on logic standard)
1 True, asserted	**High Voltage** (commonly 5v, 3.3v, 1.8v or 1.2v, but depends on logic standard)
Well, not exactly. There's these two not-quite-a-state states, too:	
Z Disconnected	**High impedance** (disconnected, or "floating"); connected to neither power or ground
X Unknown	**Indeterminate** (value cannot be predicted or is unstable); logical value only; does not have a specific electrical representation

This is commonly referred to as *four-valued logic*.

What do X and Z mean when evaluating logical expressions?

- ✓ `X` combined with any value results in `X` unless short-circuit evaluation allows otherwise
- ✓ `Z` behaves like `X` when evaluated (although it represents a distinct electrical state in the circuit)

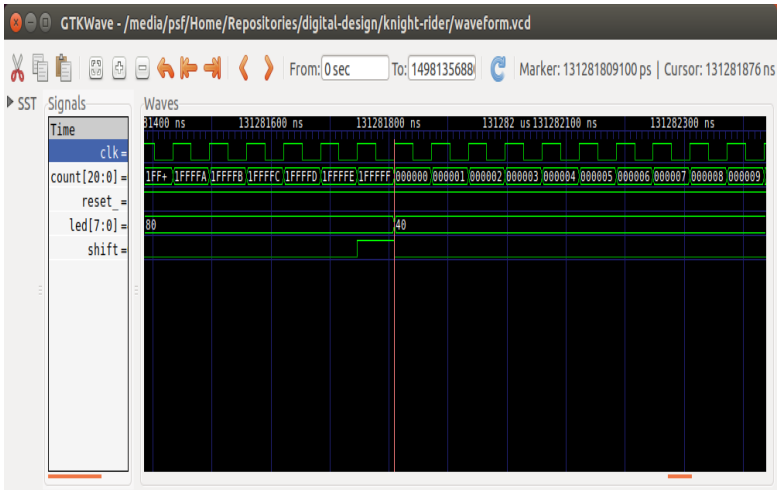
y = a & b		
0	0	0
0	0	1
0	1	0
1	1	1
X	1	**X**
0*	0	**X**
X	**X**	1
0*	**X**	0
y	**a**	**b**

y = a | b		
0	0	0
1	0	1
1	1	0
1	1	1
1*	1	**X**
X	0	**X**
1*	**X**	1
X	**X**	0
y	**a**	**b**

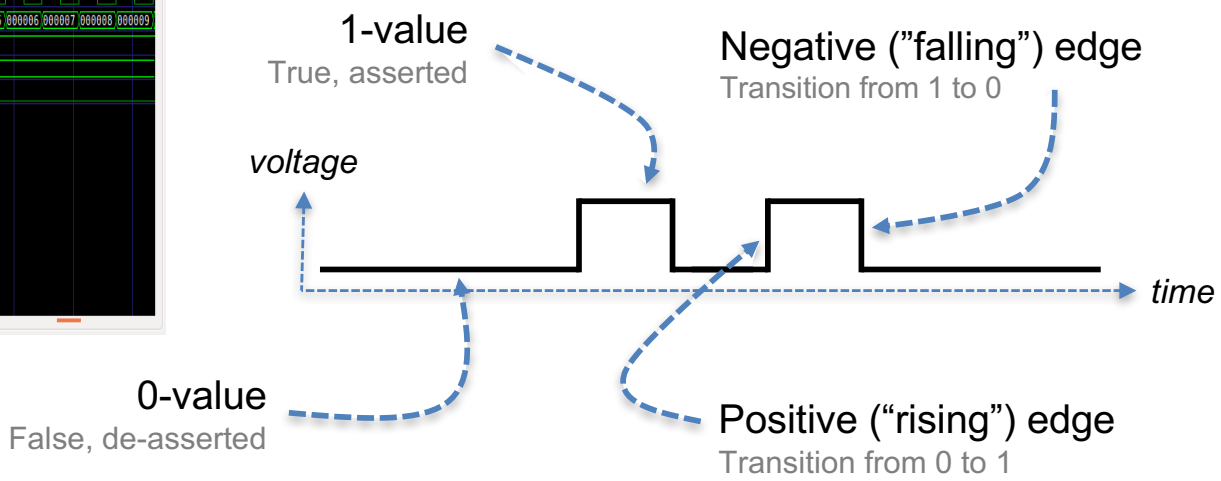
* Denotes short-circuit evaluation

A brief lesson in the **anatomy of a digital signal**.

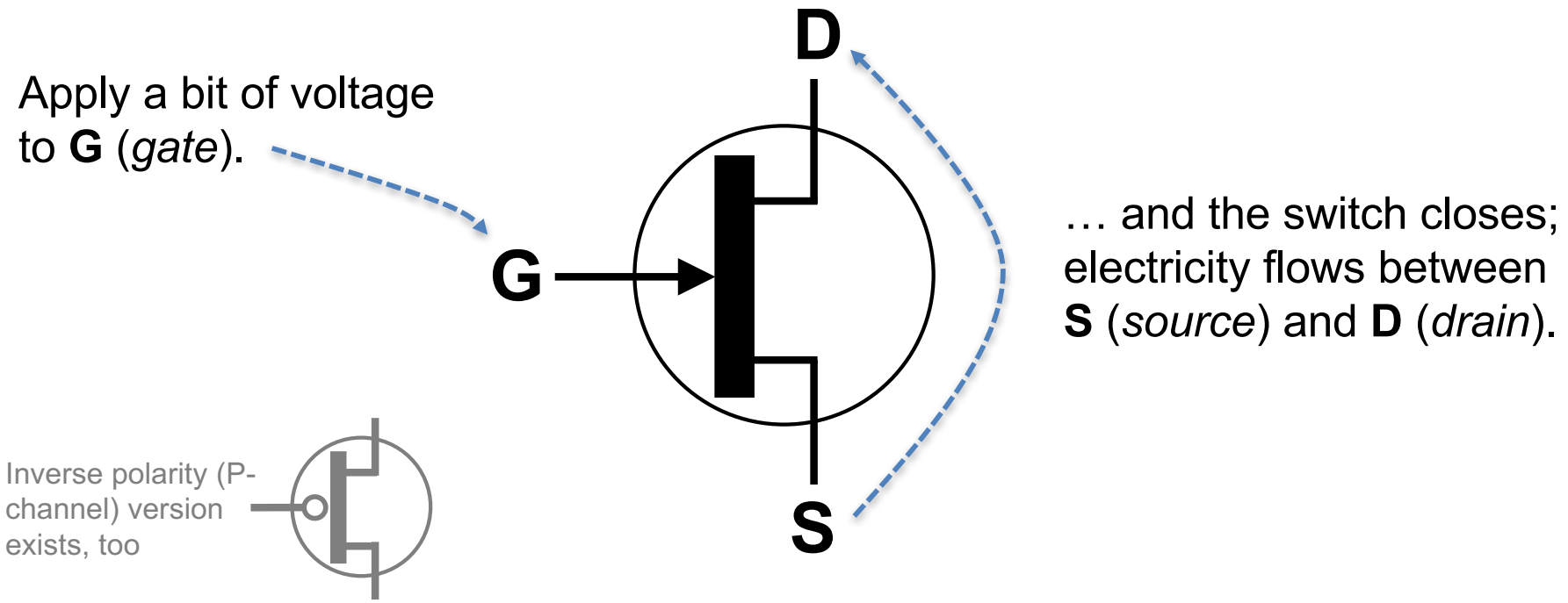
A digital waveform graphs voltage over time without any “analog” elements like curves or sine waves.



GTKWave - Waveform viewer

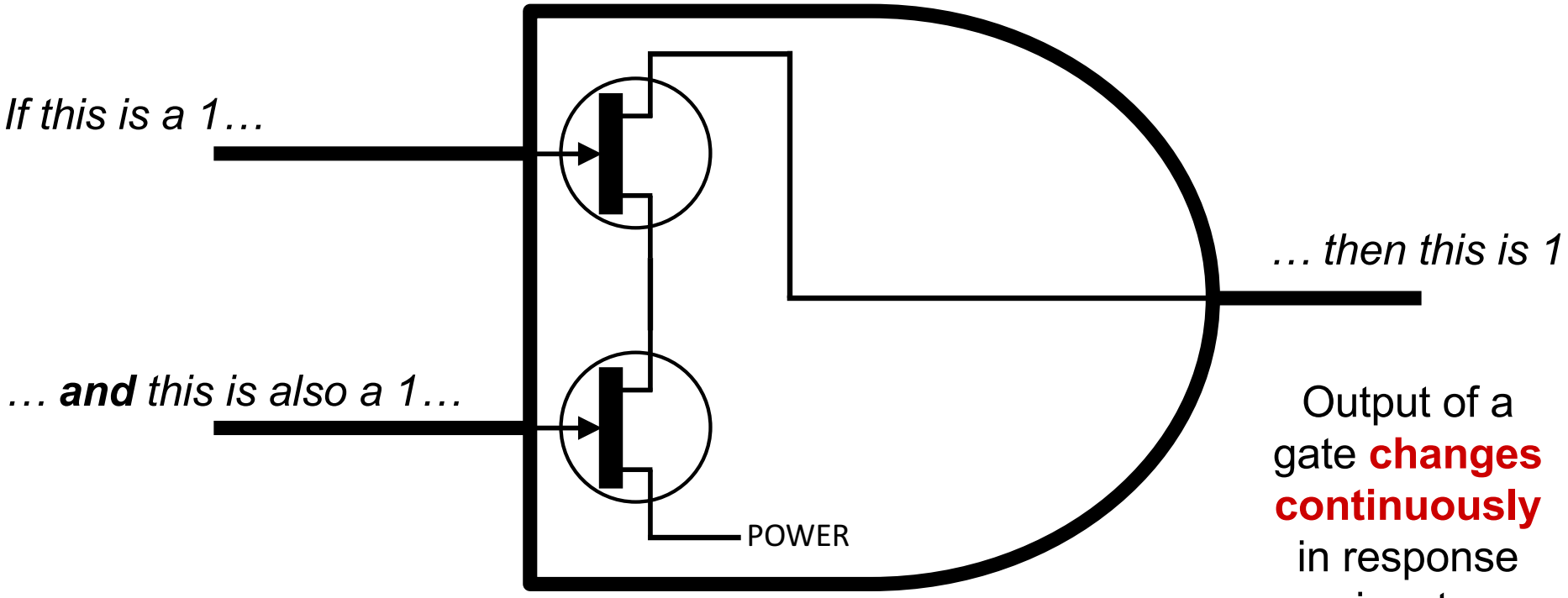


Gates are comprised of transistors, which work like **electrically controlled switches**.



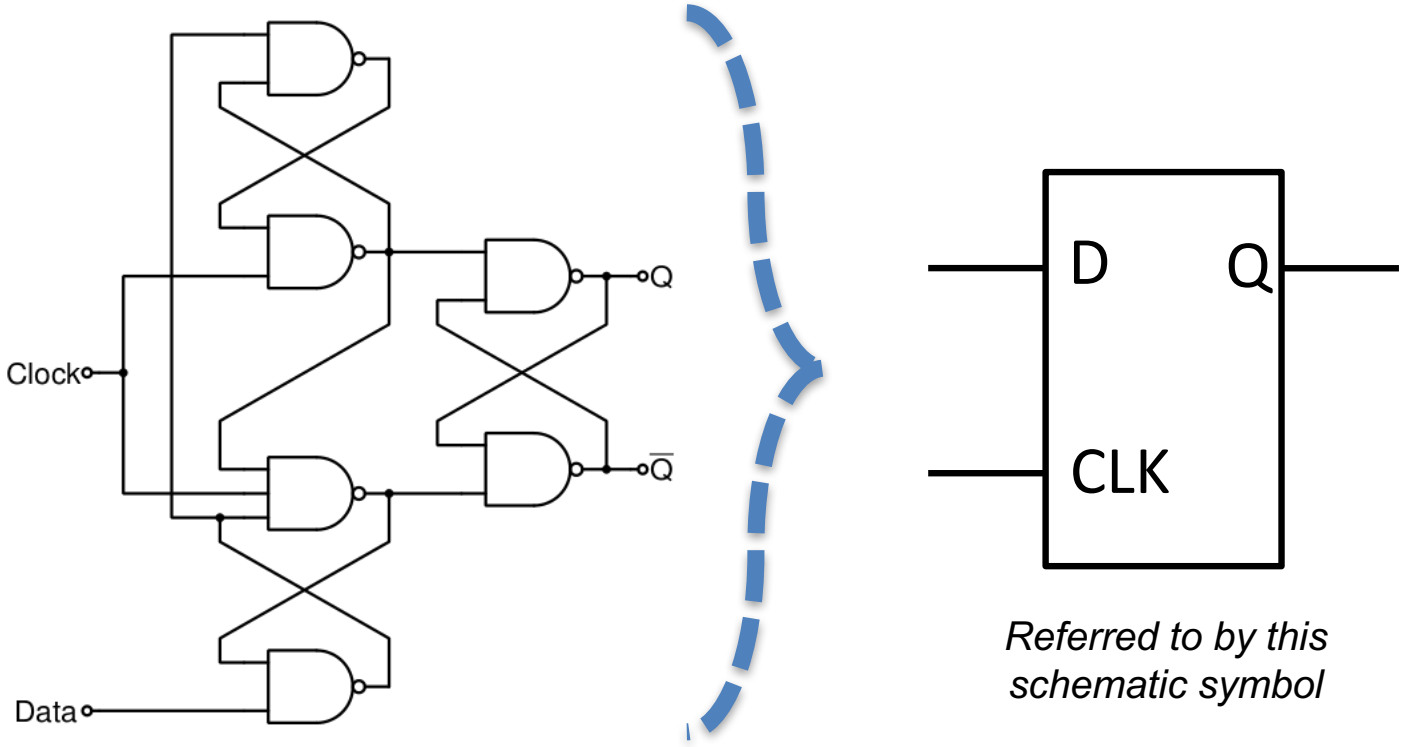
Schematic symbol for an N-channel field-effect transistor (FET)

Transistors can be **arranged to form logic gates**.



For illustration only; not an accurate schematic of a TTL AND gate. *What's wrong with it?*

Gates can be arranged to create **single-bit memories** called *flip-flops* (“registers”).

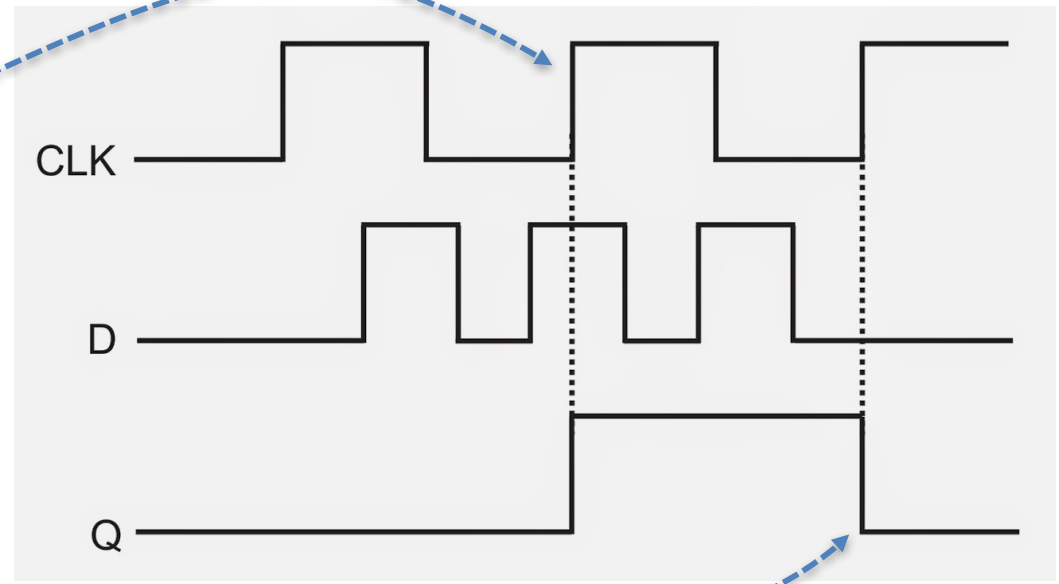


This talk will deal exclusively with positive-edge triggered, D-type, flip-flops with asynchronous reset.

Flip-flops store their input (*D*) the moment the clock (*CLK*) changes state.

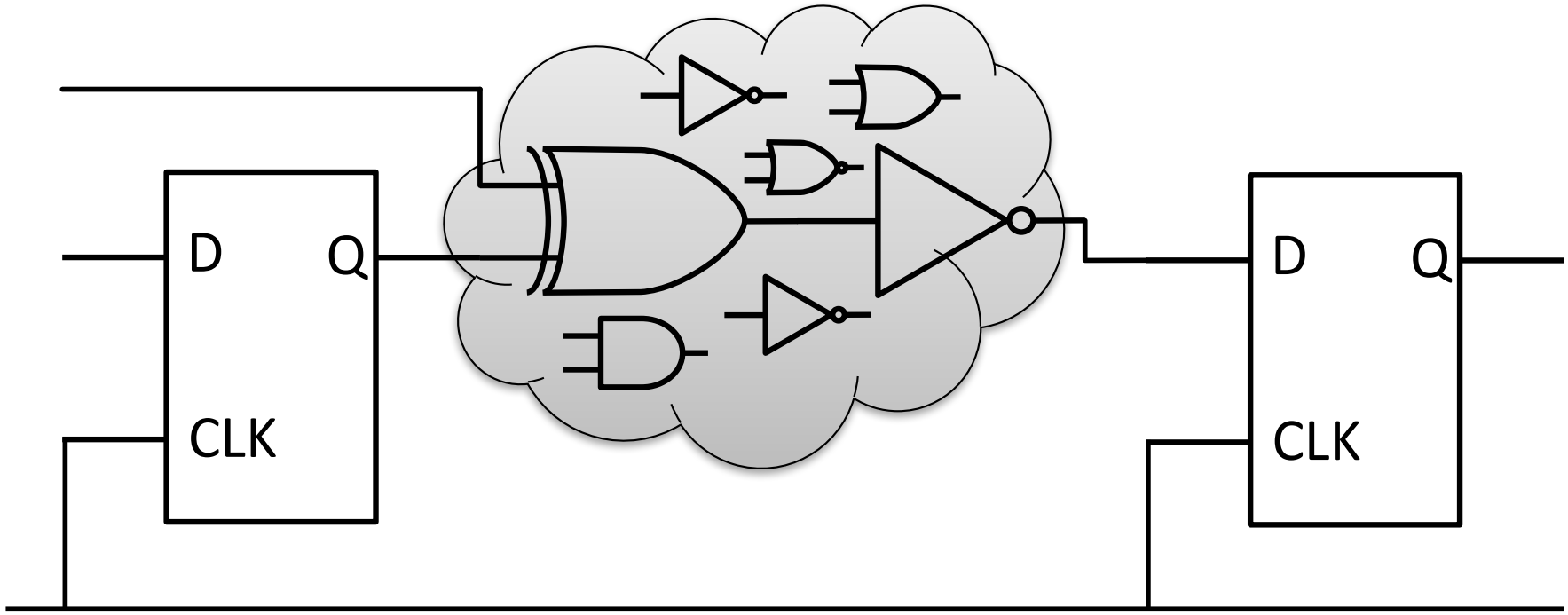
At the rising edge of the *CLK* input, the flip-flop samples the value of *D*.

The flip-flop produces a constant output until a new input is sampled at the next clock cycle.



Reset not illustrated.

Digital integrated circuits are simply **a very large combination** of gates and flip-flops.



How large a combination, you ask?

Macintosh Plus (c. 1986)
Motorola 68000

68 thousand transistors (~17 thousand gates)
8 million clock cycles per second (8 MHz)

About 50,000 times more circuitry

iPhone 7 (c. 2016)
Apple A10

3.3 billion transistors (~825 million gates)
2.4 billion clock cycles per second (2.4 GHz)

The complexity of these designs is managed by **expressing them in code**.

Verilog	VHDL
Based on C	Based on Ada
Models hardware better	Models behavior better
More popular in the USA	More popular in Europe
More popular for ASIC design	More popular for FPGAs
Both are relatively simple languages	

Chip design is actually software engineering (ssh, don't tell the hardware engineers).

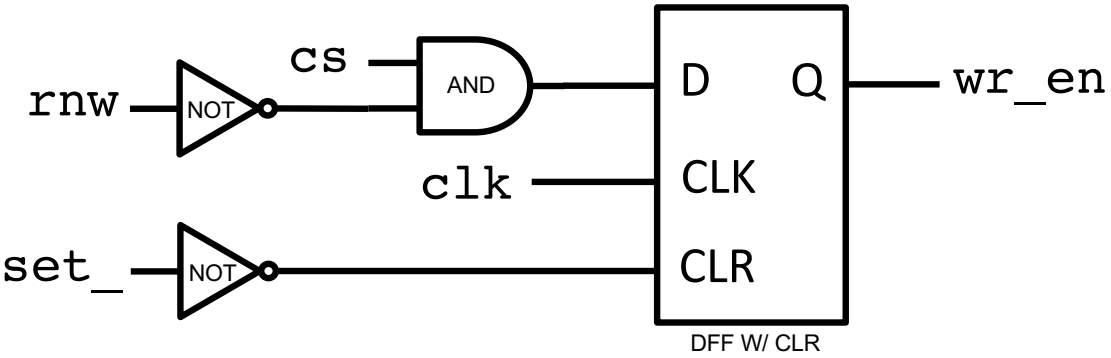
Code written in a hardware description language is ***synthesizable*** **into digital circuits**.

```
reg wr_en;

always@(posedge clk or negedge reset_)
  if (!reset_)
    wr_en <= 1'b0;
  else
    wr_en <= cs & !rnw;
```

Verilog

Code "compiles" into a hardware design



Cool. I've "written" my first integrated circuit. So **how does it become a chip?**

- ✓ Contract with a manufacturer to fabricate your design into an ASIC *(application-specific integrated circuit)*

  A custom chip that's manufactured to your specifications; approximately six-figure up-front cost (NRE) with per-unit costs dependent on size / complexity.

- ✓ Load your design into a programmable logic device such as an FPGA *(field-programmable gate array)*

  🤔 Deploy your chips in the cloud: AWS now offers FPGA-equipped EC2 instances.

  An off-the-shelf chip that contains thousands-to-millions of gates and flip-flops whose interconnections are programmable to match the behavior of your design. Each part costs somewhere between a few bucks and the price of a nice car.

Don’t kid yourself. An **FPGA is real hardware**, not some kind of circuit simulator.

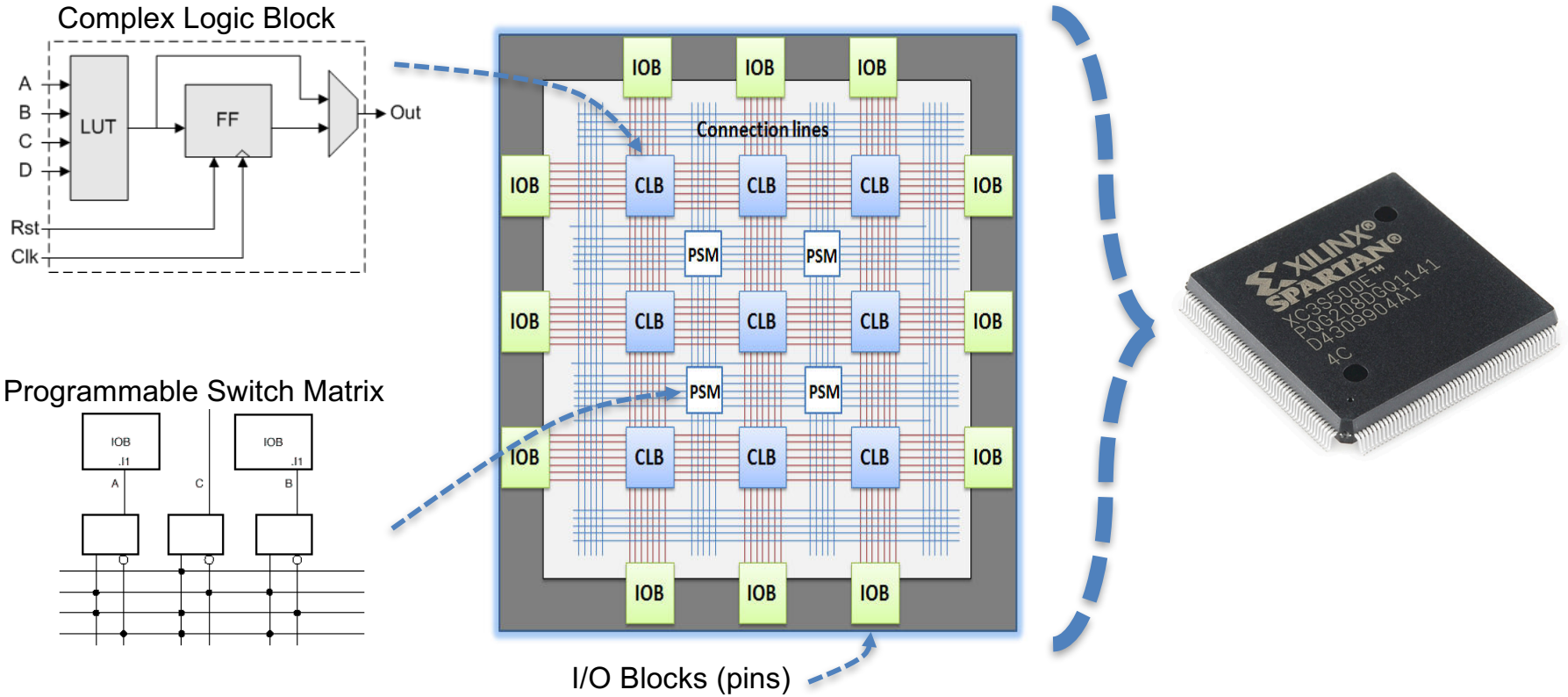
FPGA

Contains a whole pile of pieces. You assemble only those you need into your design (and the rest come along for the ride).

ASIC

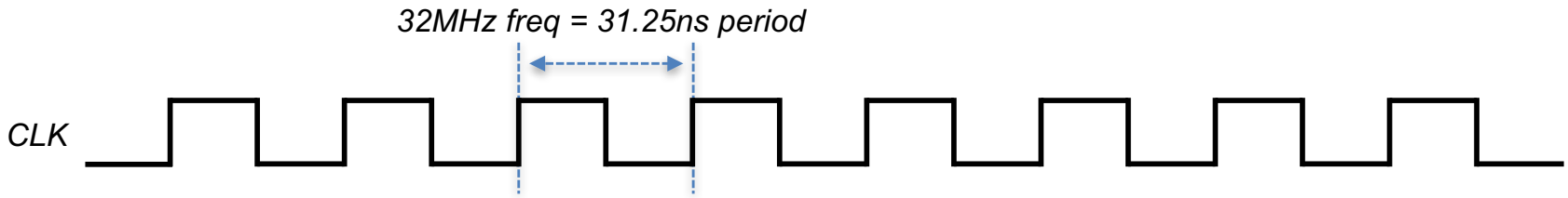
Contains only the pieces you utilize in your design.

An FPGA contains logic gates whose inter-connections are **electronically programmable**.

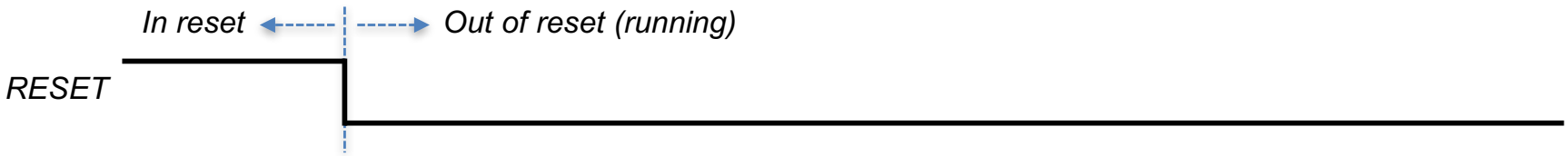


Every chip needs **two special inputs**:

A **clock** signal that oscillates between 0 and 1 at fixed frequency; causes flip-flops to sample their input. *(Commonly generated by a crystal oscillator external to the chip.)*

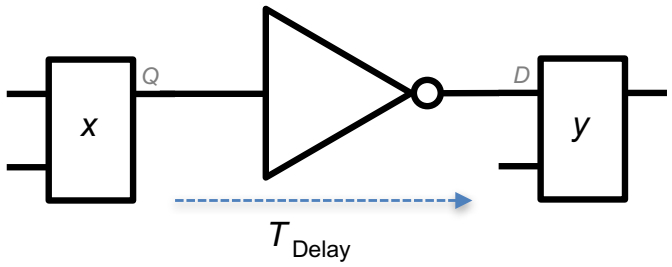


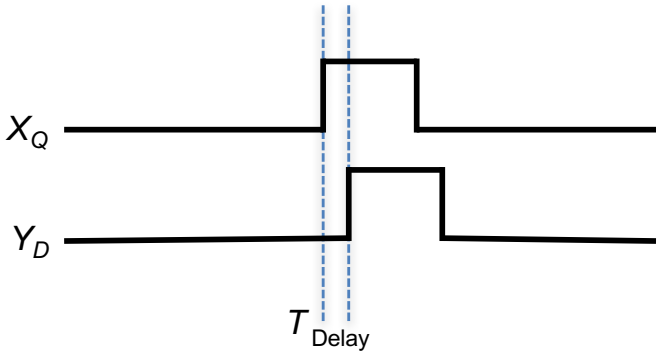
A **reset** signal used to initialize flip-flops to a default state. *(Typically driven by a power control circuit and/or a reset button.)*



Propagation delay limits how quickly logical expressions are evaluated in hardware.

```
y <= ~x;
```
Verilog



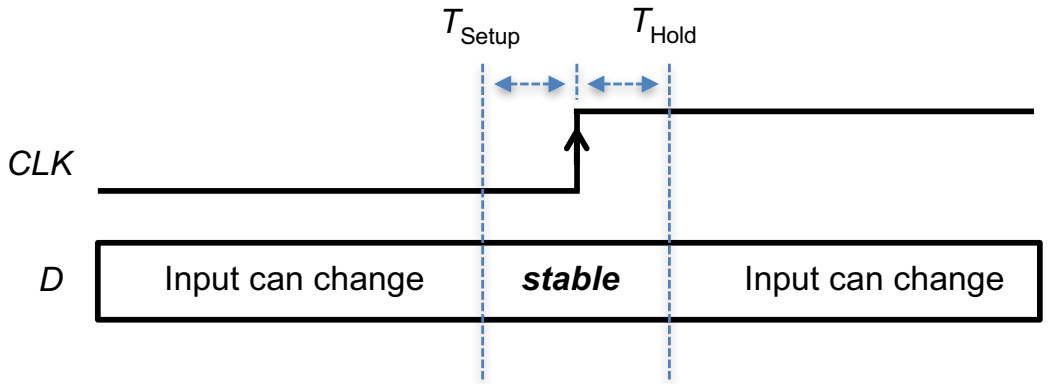


Each gate in the combinatorial pipeline introduces propagation delay. The total delay may not exceed the period of clock*.

* Except under unusual and intentionally-designed circumstances.

Setup and hold times are the traffic cops of the digital world.

The input to a flip-flop (*D*) must be stable for a brief moment before and after the clock ticks:



Otherwise, the output of the flip-flop is indeterminate (X): Could be a 1; could be a 0, or could be *metastable* (indeterminate).

Three **environmental variables** affect real-world propagation delay:

Process

Variations in the manufacturing process affect the speed at which transistors operate.

Voltage

As voltage increases, delay decreases.

Temperature

Higher temperatures result in greater delay.

If you've ever tried to overclock a PC, you probably fiddled with these variables.

Verilog for Software Engineers

Verilog is a hardware description language used to **design, simulate and describe** digital circuits.

Invented in 1985, standardized in ’95, and updated a few times since.

Software engineers will find Verilog quite simple:

- No complex types, abstractions or language semantics
- No standard libraries or frameworks to master

But also quite difficult:

- Entirely new programming paradigm; requires you think about problem solving in a different way
- Parallelism is free; sequential consistency is expensive

We’ll focus on Verilog ’95; later versions add syntactical sugar and sophisticated testing constructs.

Verilog is **like having three languages** in one! Order now!

- **Behavioral** syntax for simulating circuits
  The entire language syntax including constructs that could never be realized in hardware (i.e., no silicon equivalent of the `$display` task)
- **Register-transfer level** syntax for designing circuits
  Called "RTL"; this subset of the language is used to model the flow of data in a pipeline of *register* → *logic* → *register*. **We're mainly interested in this subset.**
- **Structural** syntax for describing synthesized circuits
  Used to describe circuits that have been "compiled" into *cells* (circuit components like flip-flops and gates), and the wires that interconnect them. Verilog compiles into Verilog.

Verilog is **converted to hardware** through a process called synthesis.

You write…

RTL Verilog

```
assign y = z & ~r;
```

Synthesis process replaces high-level behavioral language constructs with a network representation of pre-defined manufacturer-specific circuit components called "cells".

Synthesis software produces…

Gate Level Netlist

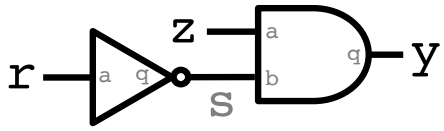
```
wire y, z, r, s;

NOT1 not00000 (
   .a(r),
   .q(s)
);

AND2 and00001 (
   .a(z),
   .b(s),
   .q(y)
);
```

Your chip contains…

Physical Hardware



Netlist has a one-to-one relationship with physical hardware.

Verilog "compiles" into Verilog. "Backend" EDA tools convert the compiled Verilog to physical hardware.

Electronic design automation (EDA) tools **convert HDL code into physical silicon**.

Simulation	Lets designers test their circuits before conversion to hardware
Synthesis	Translates HDL code into a list of gates and the interconnections between them
Verification	Assures that physical circuitry performs equivalently to simulation
Place & Route	Determines optimal physical location and placement of gates on silicon chip
Mask Prep	Generation of lithography photomasks for manufacturing

SYNOPSYS®

cadence

Mentor Graphics®

A Quick Look

This circuit outputs an 8-bit value (called `count`) that increments each clock cycle in which the `enable` signal is asserted (`1'b1`).

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input         reset_;
  input         clk;
  input         enable;
  output [7:0]  count;

  reg    [7:0]  count;
  wire   [7:0]  next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

Module Definition

Specifies the name of this circuit and the connections it makes to the outside world.

As a matter of convention, signals ending with an underscore (`reset_`) are "active low." That is, the circuit is in reset when this signal is zero.

Active high signals are asserted when their value is one.

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input         reset_;
  input         clk;
  input         enable;
  output [7:0]  count;

  reg    [7:0]  count;
  wire   [7:0]  next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

Port Definitions

Specifies the direction (`input`, `output` or `bidir`) of the ports and their width.

`[7:0]` implies that the `count` signal consists of eight physical wires grouped together as one logical value.

Each connection in the module definition should have a corresponding IO declaration.

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input         reset_;
  input         clk;
  input         enable;
  output [7:0]  count;

  reg    [7:0]  count;
  wire   [7:0]  next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

“Variable” Declarations

`reg` infers flip-flops; `wire` infers nets (connections)

Registers store data. The value of a register when evaluated in an expression is its output (*Q*).

Wires transmit data. The value of a wire changes immediately in response to the logical expression assigned to it.

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input         reset_;
  input         clk;
  input         enable;
  output [7:0]  count;

  reg    [7:0]  count;
  wire   [7:0]  next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

Combinatorial Logic

`assign` keyword used to "assign" a logical expression to a wire.

Note that the value of `next_count` changes continuously in response to the adder circuit (`count + 8'h1`).

Assignment is "delayed" by propagation delay through adder.

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input         reset_;
  input         clk;
  input         enable;
  output [7:0]  count;

  reg    [7:0]  count;
  wire   [7:0]  next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

Sequential Logic

Defines the behavior of all eight (`[7:0]`) `count` flip-flops.

Generally, all flip-flops should follow this template: activated by edge of clock or edge of reset.

Body is a single if-else statement defining the behavior of each condition.

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input        reset_;
  input        clk;
  input        enable;
  output [7:0] count;

  reg    [7:0] count;
  wire   [7:0] next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

Module Closure

`endmodule` keyword required to close module.

Verilog allows multiple modules to be defined in the same file, but considered a bad practice.

By convention, file is named after the module, i.e., `counter.v`

```
module counter (
    reset_,
    clk,
    enable,
    count
);

  input        reset_;
  input        clk;
  input        enable;
  output [7:0] count;

  reg    [7:0] count;
  wire   [7:0] next_count;

  assign next_count = count + 8'h1;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      count    <= 8'h00;
    else if (enable)
      count    <= next_count;

endmodule
```

Verilog is **based on C** (it may look familiar).

- Verilog is case sensitive and whitespace insensitive; uses C-standard `//` and `/*` for comments
- Loosely typed; everything is an unsigned integer*
- Has equivalent control flow constructs (`if/else`, `for`, `case`, etc.)
- Supports a superset of the C expression language (i.e., `+`, `-`, `*`, `/`, `~`, `!`, <, <<, etc.)
- One notable difference: Compound statements (blocks) are delimited Pascal-style with `begin` and `end`

```
if (x < y) begin
  // something interesting
end
```

* Not exactly true, but good enough for the purposes of this talk.

Remember that **you're describing hardware**, not executing instructions.

Therefore, the lexical order of statements usually doesn't matter.
(I'll wait while you get ahold of yourself.)

```
assign a = b & c;
assign d = ~e ^ f;
```

```
assign d = ~e ^ f;
assign a = b & c;
```

This is the same… as this!

```
always@(posedge clk)
  begin
    a <= b;
    b <= a;
  end
```

```
always@(posedge clk)
  begin
    b <= a;
    a <= b;
  end
```

whoa!

Verilog offers you **plenty of rope to hang yourself** (illustrated instructions included).

You can write perfectly valid Verilog that…

- 💩 Cannot be converted (“synthesized”) to hardware
- 💩 Behaves differently in simulation than in silicon
- 💩 Behaves differently depending on tool vendor
- 💩 Is completely impossible to debug or reason about*

This talk will describe Verilog as you ought to use it, not as it could be used (we’re not aiming for a definitive language guide).

* Not unique to Verilog.

A *module* is the **primary unit of abstraction** in Verilog.

- A module represents a complete (sub-)circuit that exposes a set of inputs and outputs
- All code must exist inside of a module (like Java)
- One module definition per file (by convention, also like Java)
- A signal inside the module whose name matches that of an input or output is considered connected to that IO.

```
module inverter(
   in,
   out
   );

   input  in;
   output out;

   wire out;
   assign out = ~in;

endmodule
```

A Verilog design consists of a **tree-hierarchy of modules**:

- The “top” (root) of the hierarchy represents the inputs / outputs of your chip (called the *pin-out*)
- A module can contain (*instantiate*) other modules…
- … but cannot instantiate itself or another module that instantiates it (no cycles)

```
module uart_top(
   clk,
   reset_,
      ...
   );

   transmitter tx1(
      .clk(clk),
      .reset_(reset_),
      .tx(tx),
      .data_out(data_out)
   );

   ...

endmodule
```

What's the **syntax for defining** a new module?

Declares a module called `my_chip` (by convention, defined in a file titled `my_chip.v`)

Input and output signals. No width specifiers in this list.

Each port (above) must have a corresponding `input` or `output` statement. Multi-bit values must have their width specified.

Your logic followed by the `endmodule` keyword.

```
module my_chip (
  clk,
  reset_,
  data_in,
  data_out);

  input         clk;
  input         reset_;
  input  [7:0]  data_in;
  output [7:0]  data_out;

  // Your logic here

endmodule;
```

How do I **instantiate a submodule** inside of an existing module?

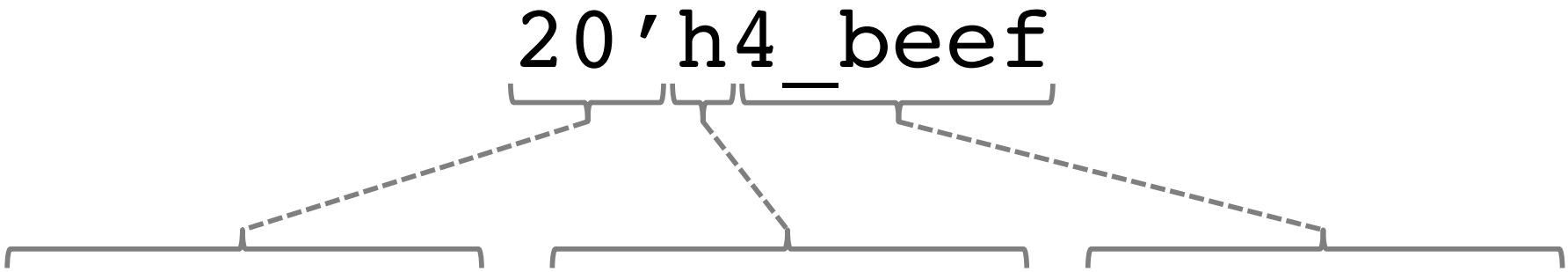
Instantiates a `receiver` named `rx1`. Module names are used to differentiate signals in waveform viewer.

Dot-value (`.data_in`) refers to the name of the input/output defined by `receiver`. Value in parentheses (`serdata`) is a signal defined in the current module.

```
module transceiver (
...
  receiver rx1 (
    .clk(clk),
    .reset_(reset_),
    .data_in(serdata),
    .data_out(pardata)
  );
...
endmodule
```

Verilog does not require all ports to be connected, but considered bad practice not to.

Integer literals in Verilog specify width and radix:



Size of integer (in bits)

Case-insensitive radix of literal expression.

Acceptable values:

H – Hexadecimal (base 16)
D – Decimal (base 10)
O – Octal (base 8)
B – Binary (base 2)

0..1 in binary expressions; **0..7** in octal; **0..9** in decimal; **0..9**, **a..f** in hexadecimal.

_ is for readability.

Also: `Z` or `?` to represent "high impedance" (nothing), or `X` to represent unknown (in simulation, only)

Verilog adds **new operators** to C's repertoire.

Concatenation { }

Combines two or more values into a single logical value.

```
{3'b111, 1'b0}          // 1110
```

Replication { { } }

Duplicates a value some number of times.

```
{2 {2'b10}}             // 1010
```

Reduction &, |, ^

Performs a bitwise operation across all bits of a value producing a single-bit result.

```
&4'b1011                // 0
|4'b1011                // 1
```

Four-valued Equality ===

Compares two values; == results in x if any bit in either value is x or z. (Simulation only.)

```
2'b11 == 2'bx1          // x
2'b11 == 2'b1z          // x
2'b1x === 2'b1x         // 1
2'bzx === 2'bzx         // 1
```

There are **two kinds of assignments** in Verilog. (It's best if you not think too much about this.)

<=	=
Non-blocking assignment	*Blocking* assignment
Use inside `always@` blocks	Use with `assign` statements
All assignments in a compound statement execute at the same time (in parallel)	All assignments in a compound statement execute sequentially
Behaves like hardware	Behaves like software

A common source of confusion among beginners; best to paint-by-numbers at first.

Verilog has **two variable types**: `wire` and `reg`.

`wire`	`reg`
Models a connection (*net*) between two circuit elements	Models a flip-flop or latch (like a variable in software)
Transmits value; does not store value	Transmits and stores value
Value changes continuously with respect to	Value changes when clock ticks or reset is asserted
Assigned using the `assign` statement	Assigned inside of an `always@` block
All module IOs are implicitly declared as wires (unless a reg of the same name exists)	Can drive module outputs but cannot be connected to an input (have the same name as an input signal)

Resist the urge to think of these as “data types;” they both hold the same type of data: bits.

Wires and regs **can be grouped together** into a single logical value called a *vector,* Victor.

- Bits in a vector can be accessed independently or as a subset using a *part select index.*
- Size of a vector declaration must be constant at compile time. (How come?)
- Part select index does not need to be constant; but the resulting width must be.

```
wire          single_bit;
wire [7:0]    eight_bit;
reg  [31:0]   thirtytwo_bit;
reg  [3:0]    four_bit;
```

```
assign eight_bit[0] = 1'b1;

assign x = eight_bit[7:4] &
           thirtytwo_bit[3:0];
```

```
always@(...)
  thirtytwo_bit <= 32'h00000000;
  four_bit[x]   <= y;
```

Wires **transmit value** but do not hold state

- Value changes continuously in response to assigned stimulus
- Assigned using the = operator and the `assign` keyword.
- May be connected to the input or output of an instantiated module or a `reg`.

```
wire wr_en;
assign wr_en = cs && !rnw;
```

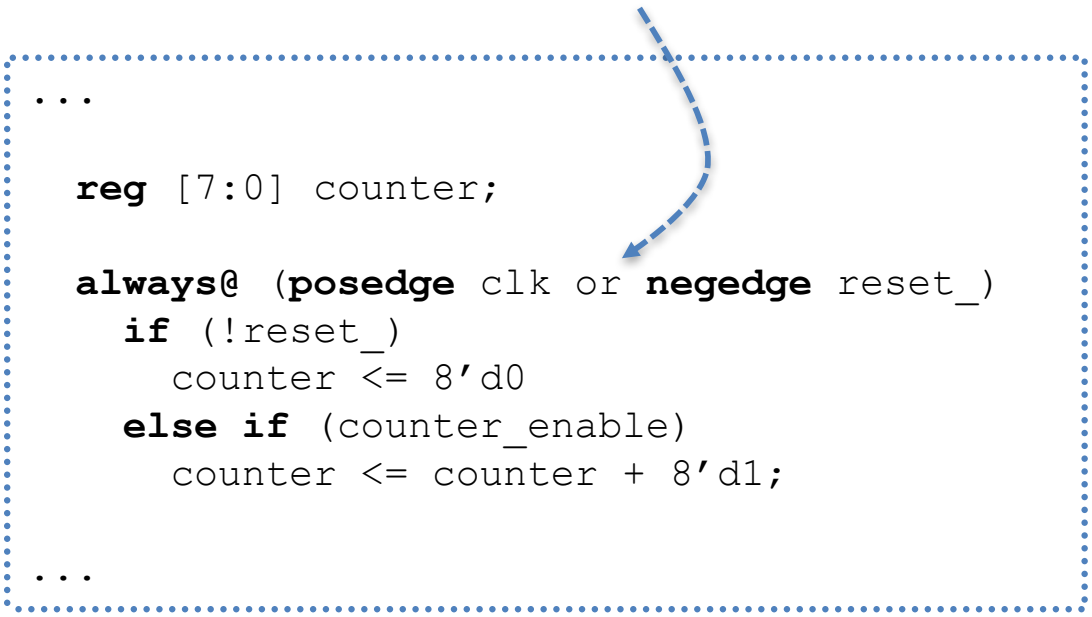
```
wire [15:0] counter_val;

counter counter_inst (
  .counter_out(counter_val)
);
```

- All inputs and outputs are inferred as wires (unless a `reg` of the same name exists—allowable only for outputs)

Registers **model flip-flops**; assignment is triggered by a clock edge or a reset signal.

- Value changes in response to a trigger (i.e., clock or reset)
- Assigned inside an `always@` block using the <= operator
- May be connected to the input of an instantiated submodule (but cannot accept the output).



```
...

  reg [7:0] counter;

  always@ (posedge clk or negedge reset_)
    if (!reset_)
      counter <= 8'd0
    else if (counter_enable)
      counter <= counter + 8'd1;

...
```

A **point of confusion** when using wires and regs:

Wires…

- Can be connected to the inputs or outputs of their own module or any instantiated submodule

Registers…

- Can be connected to the inputs of an instantiated submodule (that is, they can *drive* an input)
- Can be connected to the output of their own module
- Cannot be directly connected the output of an instantiated module (registers must be clocked with an `always@` block)

Combinatorial assignments in Verilog are modeled with an `assign` statement:

```
assign wr_en = cs & ~rnw;
```

Left-hand value **must be a `wire`** and only one assignment per wire.

Always use blocking assignment (=) in assign statements.

Terms on the right-hand side can be `wire`, `reg` or literal values.

```
assign a = b + c;
assign b = a + d;
```

No cycles allowed in assignments.

The `always@` statement is used to **model flip-flop assignments**.

```
always@(posedge clk or negedge rst)
```

`always@` is used to model flip-flop assignments in Verilog.

This form (with a `posedge x or negedge y` activation list) **models D flip-flops with async reset**.

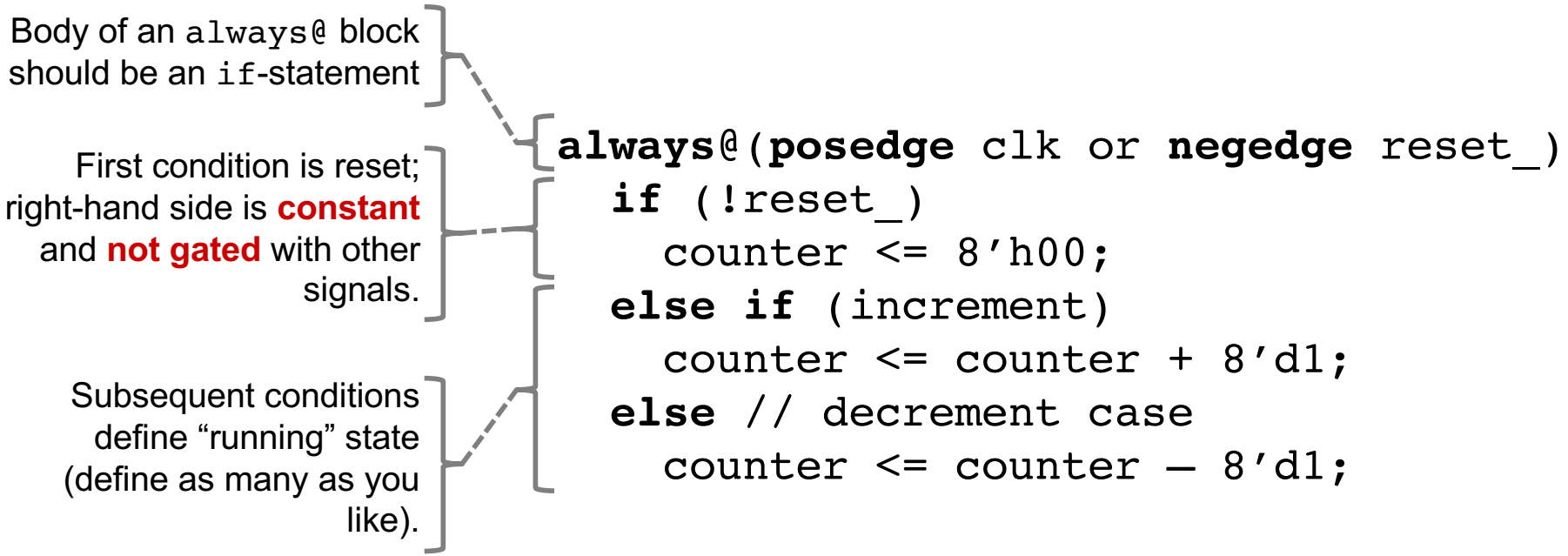
Indicates this block "executes" whenever the `clk` signal transitions from a 0 to a 1.

Used to specify the clock input to the flip-flop.

Indicates this block "executes" whenever the `rst` signal transitions from a 1 to 0.

Used to specify the async reset for the flip-flop.

By convention, your `always@` blocks should **follow this template**:



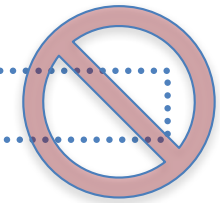
Do not assign the same `reg` from multiple `always@` blocks.

Clocks and resets are **special snowflakes** inside your chip. Don't mess with 'em (or Texas).

Don't gate the clock

The clock should not appear as a term in logical expressions (or really anywhere outside of a flip-flop's sensitivity list).

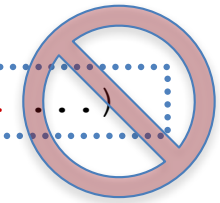
```
assign x = clk || y;
```



Clocks trigger flops

Only clock and reset belong in the `always@` sensitivity list. Don't use other signals! Use `if/else` conditionals instead.

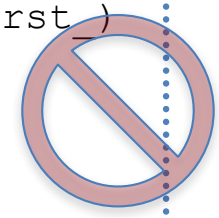
```
always@(posedge cntr_en ...)
```



Use constant reset values

The value assigned to a register in the reset condition should be an integer literal, not an expression.

```
always@(... or negedge rst_)
  if(!reset_)
    x <= y + z;
  else
    ...
```



These represent bad practices and code smells, not formal language requirements.

What's safe to use in synthesizable Verilog?

- ✓ `wire` and `reg` declarations
- ✓ `assign` statements (for combinatorial logic)
- ✓ `always@` blocks (triggered by `posedge clk or negedge reset_`) to model flip-flops
- ✓ Logical, bitwise and arithmetic expressions (division not allowed; multiply should be used with caution)
- ✓ `if` and `case` (switch) conditional constructs. Ternary conditional expression (`?:`) is your friend.
- ✓ Instantiation of other modules

Not an exhaustive list (but representative of the most common language elements).

A quick peek at simulating circuit designs.

A Verilog testbench is used to **simulate a circuit**.

A *testbench* provides stimulus (input) to a circuit under test, generates waveform output and verifies correctness:

- Can be written to verify correctness of a single module (like a unit test) or an entire design hierarchy (like an integration test)
- Generates input signals like clock and reset
- Testbenches are not intended to be synthesized into silicon so they're free to use the full Verilog syntax, including:
  - ✓ Timing and delay statements
  - ✓ System tasks
  - ✓ Procedural language constructs

Subsequent slides introduce syntax that can be used only in testbenches.

A Quick Look

This testbench generates clock and reset inputs to our Knight Rider circuit under test and produces waveform output (`waveform.vcd`)

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Module Definition

Note that our testbench module defines no inputs or outputs.

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Variable Declarations

Notice how inputs to the unit under test are `reg`; outputs are `wire`.

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Instantiation of Unit Under Test

knightrider is the name of the circuit we're testing. This connects that module's input to our test stimulus.

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Test Initialization

Initial block executes at the beginning of the simulation (time 0)

$dumpfile/$dumpvars instructs simulator to generate waveform output.

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Register Initialization

The value of `clk` and `reset_` are `1'bx` until we initialize them to some other value.

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Reset

Wait 100ns, then de-assert reset signal.

(Recall that `reset_` is asserted when 0)

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Clock Generation

This `forever` block is an infinite loop; each 31.25ns, toggle `clk`.

Produces a 32Mhz clock signal.

```
module testbench ();

  reg         clk;
  reg         reset_;
  wire [7:0] leds;

  knightrider uut (
    .clk(clk), .reset_(reset), .leds(leds)
  );

  initial begin
    $dumpfile("waveform.vcd"); $dumpvars();

    clk <= 1'b0;
    reset_ <= 1'b0;
    #100;
    reset_ <= 1'b1;
  end

  initial forever begin
    #31.25; clk <= ~clk;
  end

endmodule
```

Timing and delay statements control the **resolution and execution** of the simulation.

Timescale

Use the `` `timescale `` directive to specify a simulation resolution and normative timescale.

- First value specifies the unit of time denoted by delay statements (i.e., `#7` means 7ns)
- Second value indicates smallest unit of time "visible" to to the simulation (values rounded to this magnitude).

```
`timescale 1ns / 100ps
```

Delay

The # operator delays execution of subsequent statements in the block.

`x` is `16'h00` at beginning of simulation; `16'hff` 5ns later.

```
initial begin
  x[15:0] <= 16'h00;
  #5;
  x[15:0] <= 16'hff;
end
```

A variety of **system tasks provide hooks** to control the simulator.

System tasks are special commands whose name is preceded by $ (Verilog defines about a dozen more, too):

`$finish`

Ends the simulation.

`$dumpfile(<filename>)`

Specifies the name of the file that waveform data should be written to.

`$dumpvars`

Tells the simulator to begin writing waveform data to the dumpfile.

```
`timescale 1ns / 100ps

module my_testbench ();

  initial begin
    $dumpfile("waveform.vcd");
    $dumpvars();
    #1000;
    $finish
  end

endmodule
```

Initial and forever constructs are your primary building blocks for testbenches.

Initial

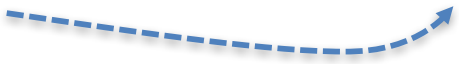
Executes at the beginning of the simulation (time zero). Useful for initialization and other test logic. Note that a testbench may have as many initial blocks as it wishes (each executes in parallel).

```
initial begin
  clk    <= 1'b0;
  reset_ <= 1'b1;
end
```

Forever

Executes statements in an infinite loop; useful for generating clock signals.

Generates a 32Mhz clock signal.



```
`timescale 1ns / 100ps

initial forever begin
  #31.25;
  clk   <= ~clk;
end
```

At last, you're ready to **synthesize to hardware**.

How do I **map physical pins** on the FPGA to inputs / outputs in my design?

A *user constraints file* (UCF) provides information about the physical pin-out of your chip (including a mapping to signals in the design).

Name of a top-level I/O in your design

Name of a pin on the Xilinx chip

```
NET CLK     LOC="P94"  | IOSTANDARD=LVTTL | PERIOD=31.25ns;
NET RESET_  LOC="P114" | IOSTANDARD=LVTTL;
NET LED<0>  LOC="P123" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=SLOW;
NET LED<1>  LOC="P124" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=SLOW;
NET LED<2>  LOC="P126" | IOSTANDARD=LVTTL | DRIVE=8 | SLEW=SLOW;
```

`LED<0>` is represented as `LED[0]` in Verilog.

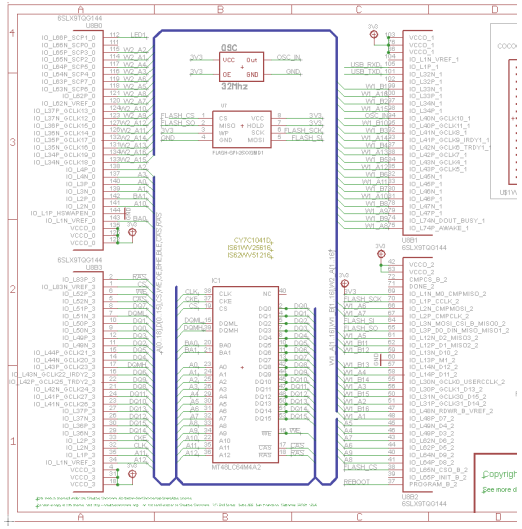
Logic standard (LVTTL is 0v-3.3v)

Extra information about the signal like drive strength (current) and clock period

How do I **know which pin on the FPGA** connects to the LEDs on my circuit board?

Sorry, Charlie. There's no magic. You'll have to read the electrical schematic.

- The Papilio site lists pin numbers and hardware connections in tabular format.
- Papilio schematics are also available on their website.
- **Don't screw up**. Misconnecting pins can damage the FPGA or circuit board.



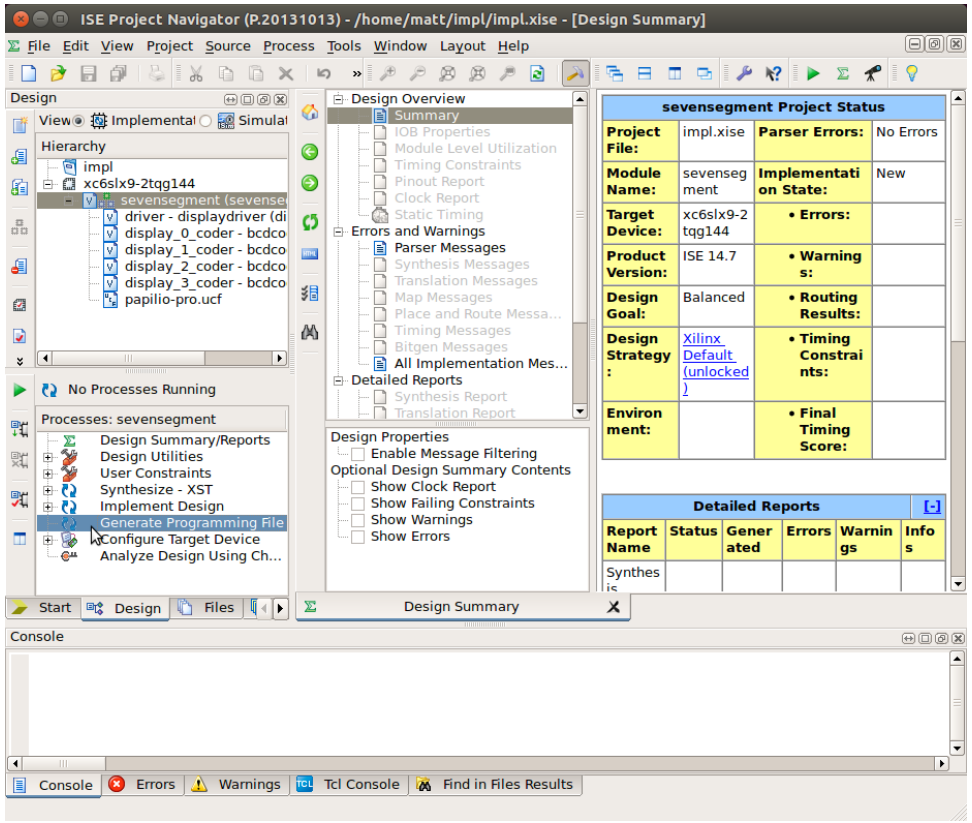
Perhaps more straightforward than it seems. This does not require an electrical engineering degree.

How do I **get my Verilog onto the FPGA**?

Xilinx provides GUI and command-line tools for "implementing" your design in hardware.

It's not hard. I promise.

Detailed installation and implementation instructions are available on my GitHub.



Knight Rider Hands-on Lab Exercise

Goals of the Knight Rider **hands-on lab exercise** are to:

- ✓ Author a circuit that animates eight LEDs in a back-and-forth pattern reminiscent of Kit from Knight Rider
- ✓ Simulate the circuit using Icarus Verilog
- ✓ View simulated waveform output in GTKWave
- ✓ Demonstrate how to synthesize and load the design on the Papilio hardware
- ✓ Please refrain from Hassling the Hoff during any point of this exercise

This circuit **can be decomposed** into the following pieces:

- An eight-bit register, each bit of which drives a LED. A 1 turns the LED on, a 0 turns it off.
- A counter that rolls over (saturates) at the frequency you want the LEDs to change. A frequency divider… How many bits must this register have if it increments every 31.25ns and should roll over approximately every 100ms?
- Logic to shift the active LED one spot left or right each time the counter saturates. Use << or >> to shift left or right.
- Logic to track whether we're shifting left or shifting right. A state variable of sorts.

How do I **get started**…

1. Install and launch the Ubuntu VM and log in as user `chicago`, password `coder`
2. Navigate to the `~/digital-design/knight-rider/` project directory
3. Modify the `rtl/knightrider.v` design file.
4. Execute `make` in the `knight-rider/` directory to run the simulation.
5. View the waveform using `$ gtkwave waveform.vcd`

Where to now? Detailed **instructions and sample projects** available on GitHub.

github.com/defano/digital-design

Detailed installation and setup instructions. Including instructions for loading designs onto Papilio hardware.

Learn by experimenting with working examples.

Several example projects that you can simulate, synthesize and modify.