Verilog and FPGAs

Coding Hardware?

April 22, 2015

Lincoln Roop

Presented By

#!/cmu/cc

Carnegie Mellon Computer Club

Sponsored By

Green Hills®

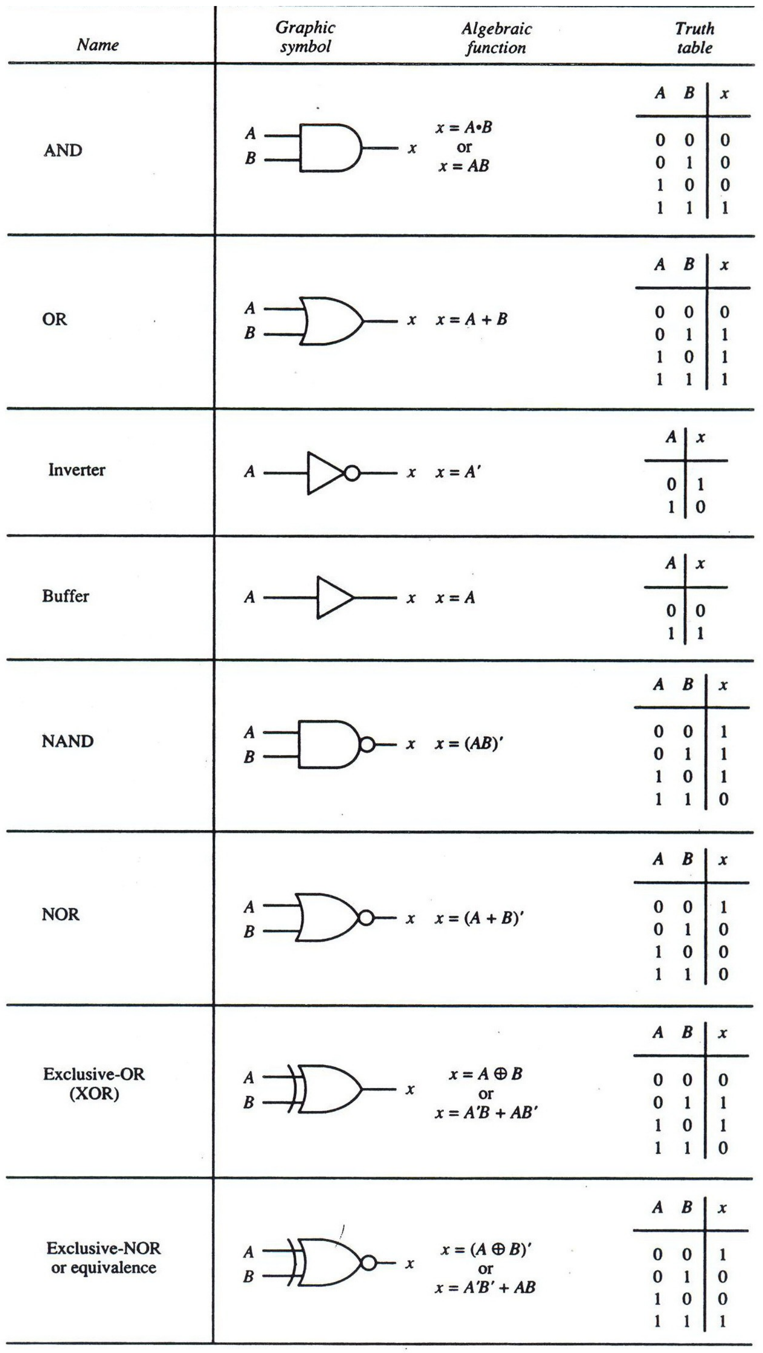
SOFTWARE

So, What's the point of this talk?

- To give an overview of hardware description languages, and explain how they differ from conventional software programming.
- To explain enough digital logic for the idea of HDLs to make sense.
  - We'll do this first even though it's listed second.
  - I apologize to any ECE majors in the room, this will sound a lot like an 18-240 lecture but less in depth.
- Unfortunately we can't cover everything.
  - There are entire semester-long classes in the ECE department about this stuff.

Boolean Logic

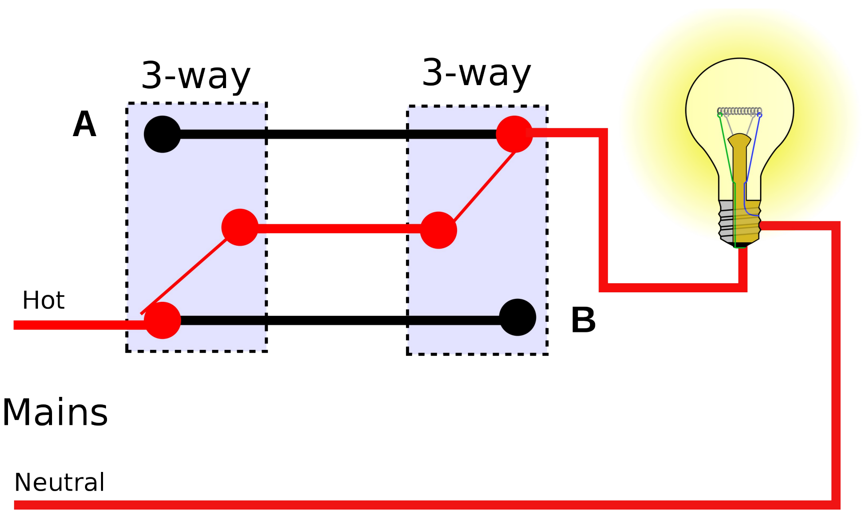
- Logic Gates – Devices that implement the Boolean logic functions (AND, OR, NOT, XOR, etc.)
- Easy to discuss and draw on paper, but can we make them in the real world?



Name	Graphic symbol	Algebraic function	Truth table
AND		$x = A \bullet B$ or $x = AB$	A B | x 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$x = A + B$	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
Inverter		$x = A'$	A | x 0 | 1 1 | 0
Buffer		$x = A$	A | x 0 | 0 1 | 1
NAND		$x = (AB)'$	A B | x 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$x = (A + B)'$	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
Exclusive-OR (XOR)		$x = A \oplus B$ or $x = A'B + AB'$	A B | x 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
Exclusive-NOR or equivalence		$x = (A \oplus B)'$ or $x = A'B' + AB$	A B | x 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

Boolean Logic

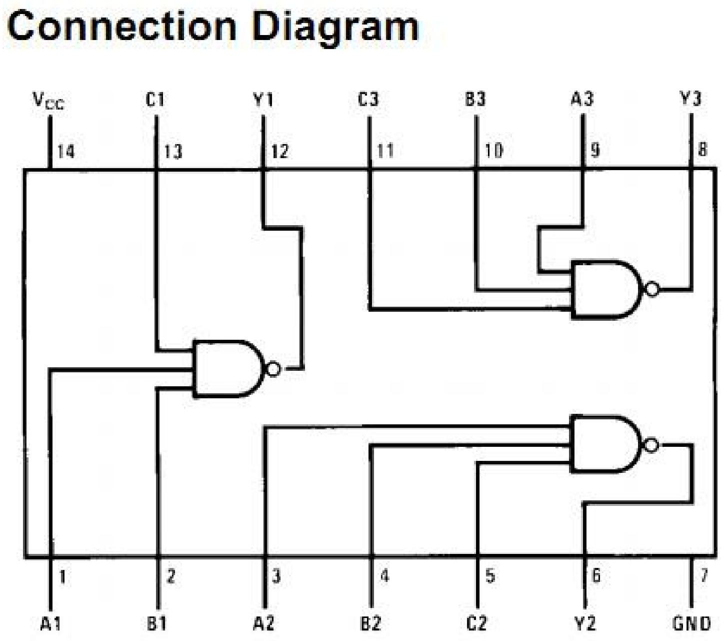
- Purely mechanical examples:
  - AND: If deadbolt is unlocked, and doorknob is turned, door opens.
  - XOR: Think of a 3 way light switch (the kind with switches at the top and the bottom of a flight of stairs).
- There are also plenty of more intentional examples of this: Old mechanical adding machines, early punchcard tabulators, etc.



Boolean Logic

- Small-Scale Integration ICs (7400 series, 4000 series).
  - These chips implement a few gates on a small chip.
  - Once upon a time, entire CPUs and motherboards were built out of these.
  - Today, they're mostly used when only a small amount of additional logic is needed in some more complex circuit.

74LS10 3x 3-input NAND Gate (in somewhat obsolete DIP-14 package)



74LS10 Functional Diagram

Is Something Missing Here?

- Interconnected Boolean gates can be used to implement any logic function, but there are some problems.
  - How long does it take for the output to be correct after we change inputs? In theory-land this is instantaneous. In the real world, the laws of physics don't allow for that.
  - Also, with just logic gates, we have no notion of **state**. We can't store the result of some function and do other things with it.

Memory Elements

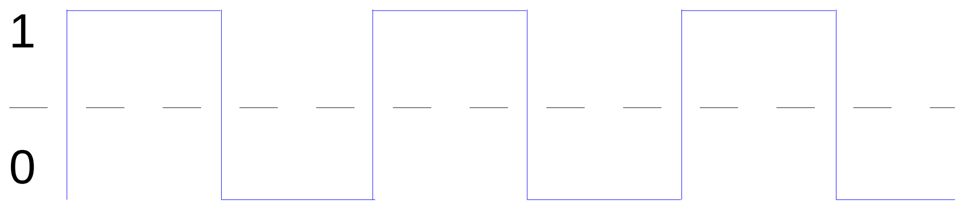
- There's a lot to explain about memory, but it's out of the scope of this talk.
- Instead, we'll just talk about *registers* as a concept.
- Registers act a lot like a variable in a computer program. We can store data in them and modify or reuse that data later.

Memory Elements

- We can also have arrays of registers. This is basically what RAM looks like from a programmer's perspective.
- In the real world it gets a bit more complicated.
  - DRAM requires periodic *refresh* cycles to keep the information stored in it.
  - Modern computers use a *memory hierarchy* with levels of *cache* (smaller, faster memory) between the CPU and main memory, and *swap space* (often called 'Virtual Memory') on disk for when programs want more memory than the system has.
- But we don't need to worry about that now.

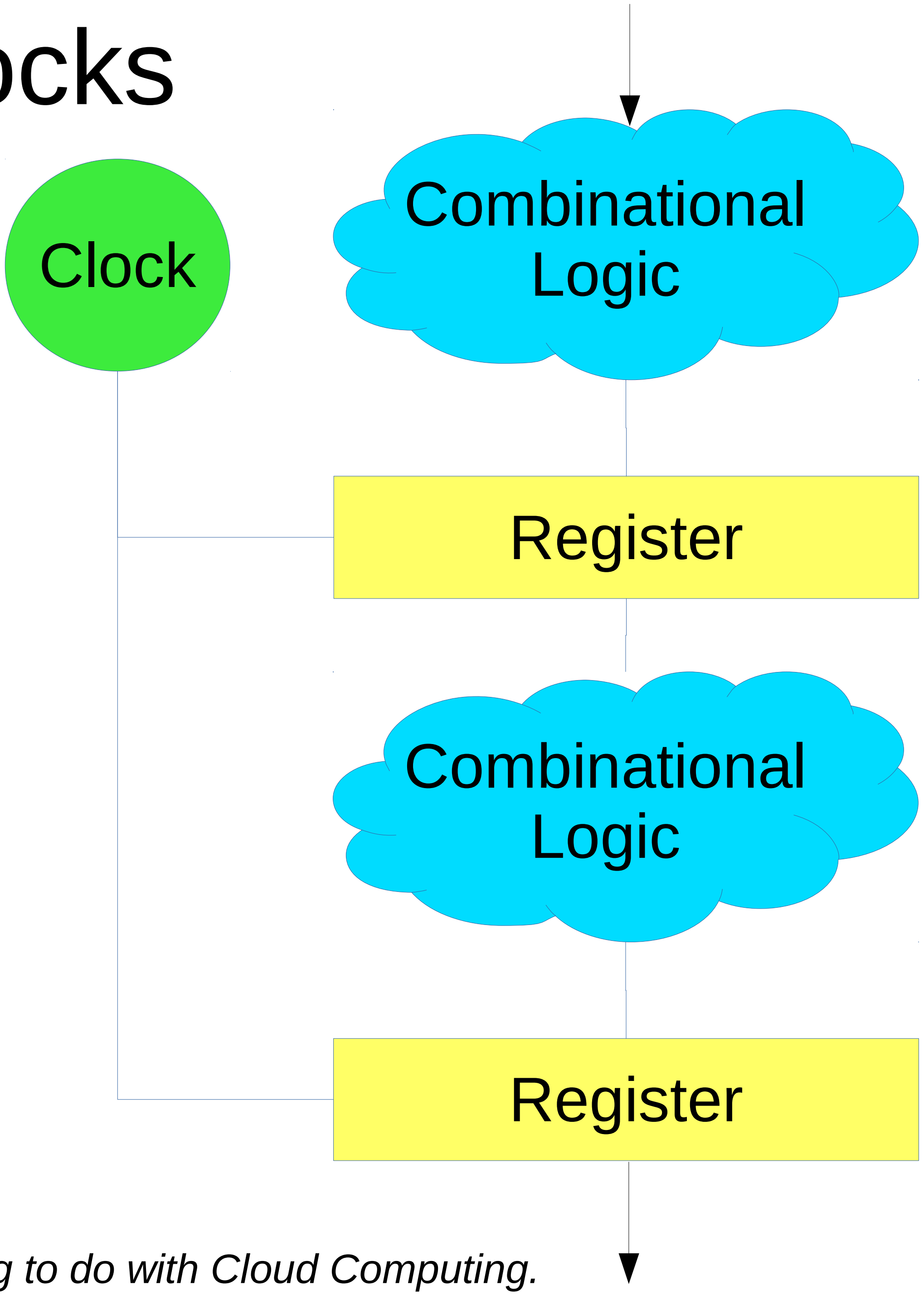
Clocks

- A *clock* in digital logic is a regular and periodic signal used to keep different parts of the circuit in time with each other.
- The clock in a digital circuit functions more like a metronome than an actual wall clock – its purpose is to keep parts of the circuit 'in time' with each other.



Clocks

- Using clock-controlled registers in a complex digital circuit lets us deal with the problem that combinational logic doesn't work instantly.
- Timing in digital circuits can get very complex, but for now we can just assume that as long as we don't run the clock faster than the combinational logic can work, things will be OK.



Note: Clouds of combinational logic have nothing to do with Cloud Computing.

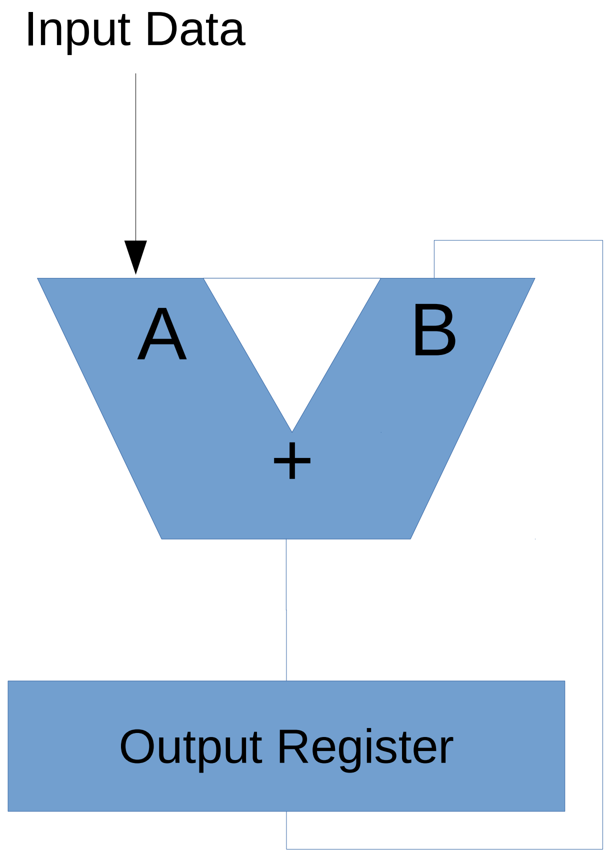
Example – Mixing Logic and Registers

- Say we want to add up a list of numbers.
- Using just combinational logic, we can construct an adder. Adding 2 terms is easy, 3 is doable, but beyond a certain point, this doesn't really make sense.

```
    1,000
    3,272
      602
   14,860
+   2,713
---------
      ???
```

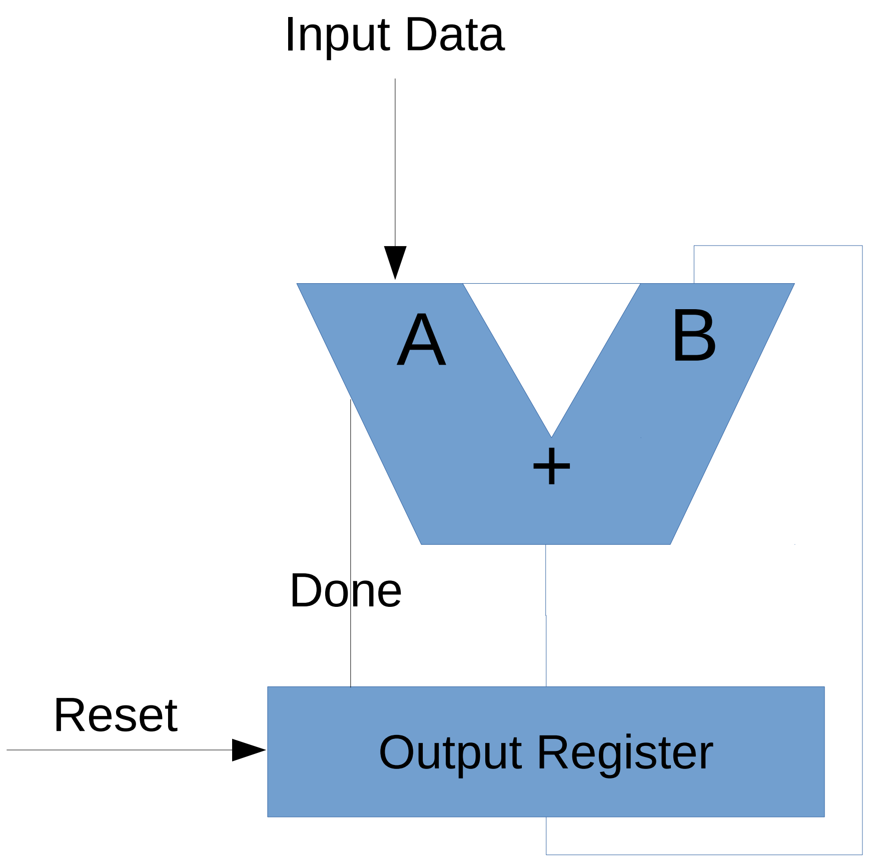
Example – Mixing Logic and Registers

- A 2-input adder will be enough to get the job done if we add some memory to the design.
- We can store the output of the adder in the register, and then use the register's output as one of the inputs to the adder.
- The other input will be the numbers that we need to add.



Example – Mixing Logic and Registers

- Wait? We're going to use the register as both an input and the output?!?
- Sure, we can do this, but we have to be careful.
- We need to be able to store 0 in the register on reset.
- And we need to clock the register such that it will latch the output of the adder only when the adder is done with each addition.

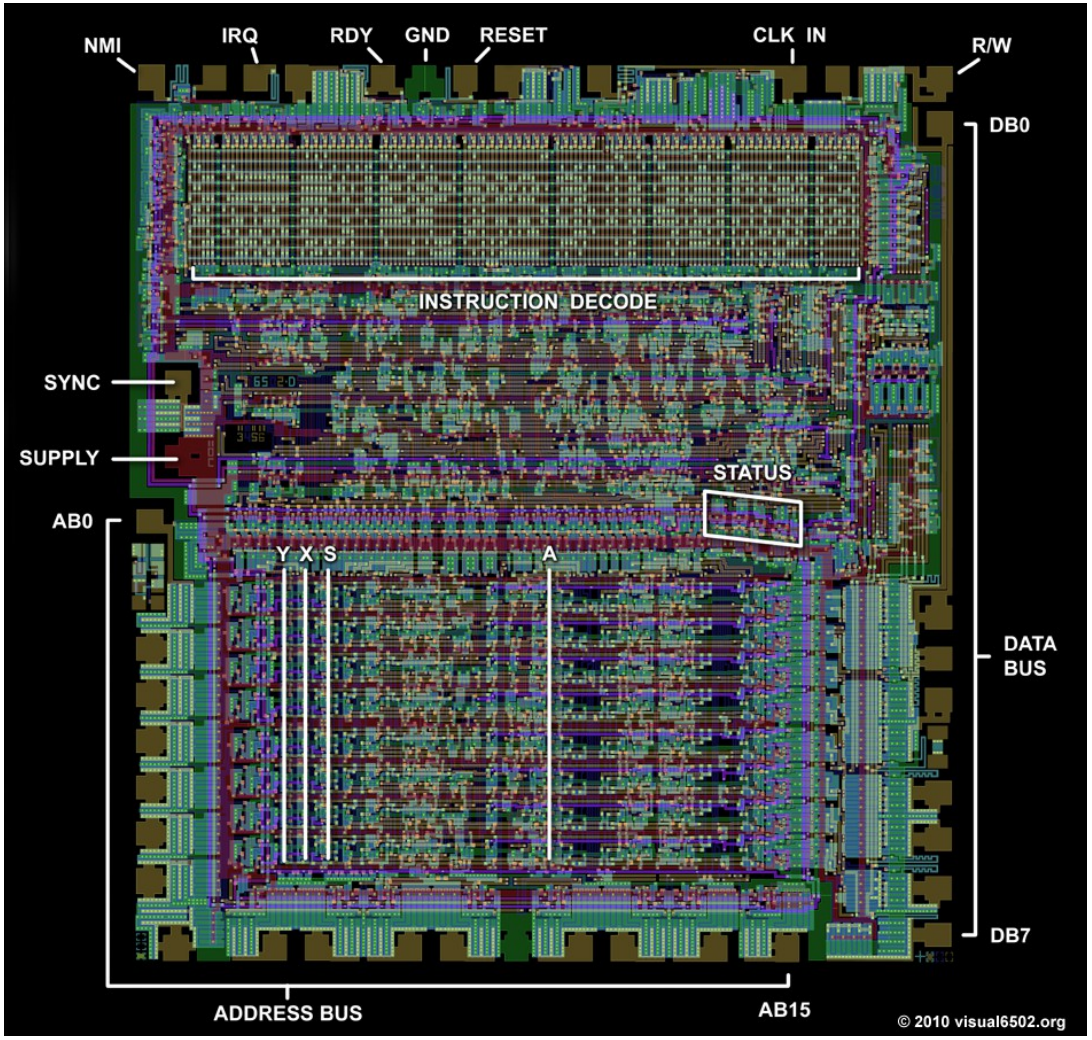


IC Design before automation

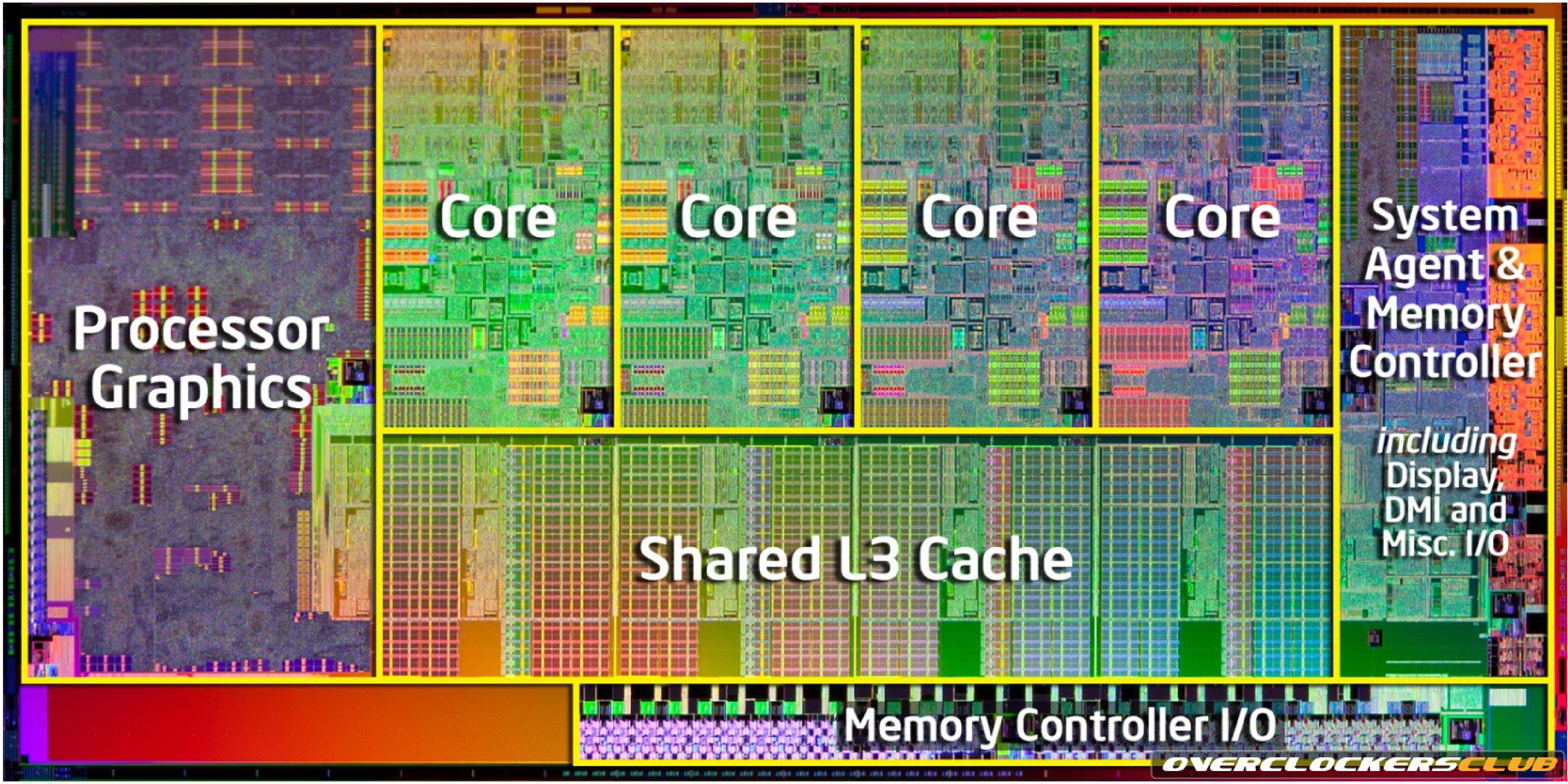
- Designing an IC is a complex process. You have to design the high-level logic, and then translate that into physical layout (actual transistors on silicon).
- Before automation tools, this was done entirely by hand.
  - Block diagrams were drawn on paper.
  - Gate-Level 'simulation' was done with breadboards or wire wrap and lots of individual logic ICs.
  - Actual silcon layout was done with colored tape and rulers on very large pieces of paper at many many times actual size, and then photo-reduced into masks for making chips.

IC Design before automation

- Manual design worked fine for simple things. Most 8-bit CPUs like the 6502 and 8080 would have been designed this way.
- It was still quite involved and time-consuming though.



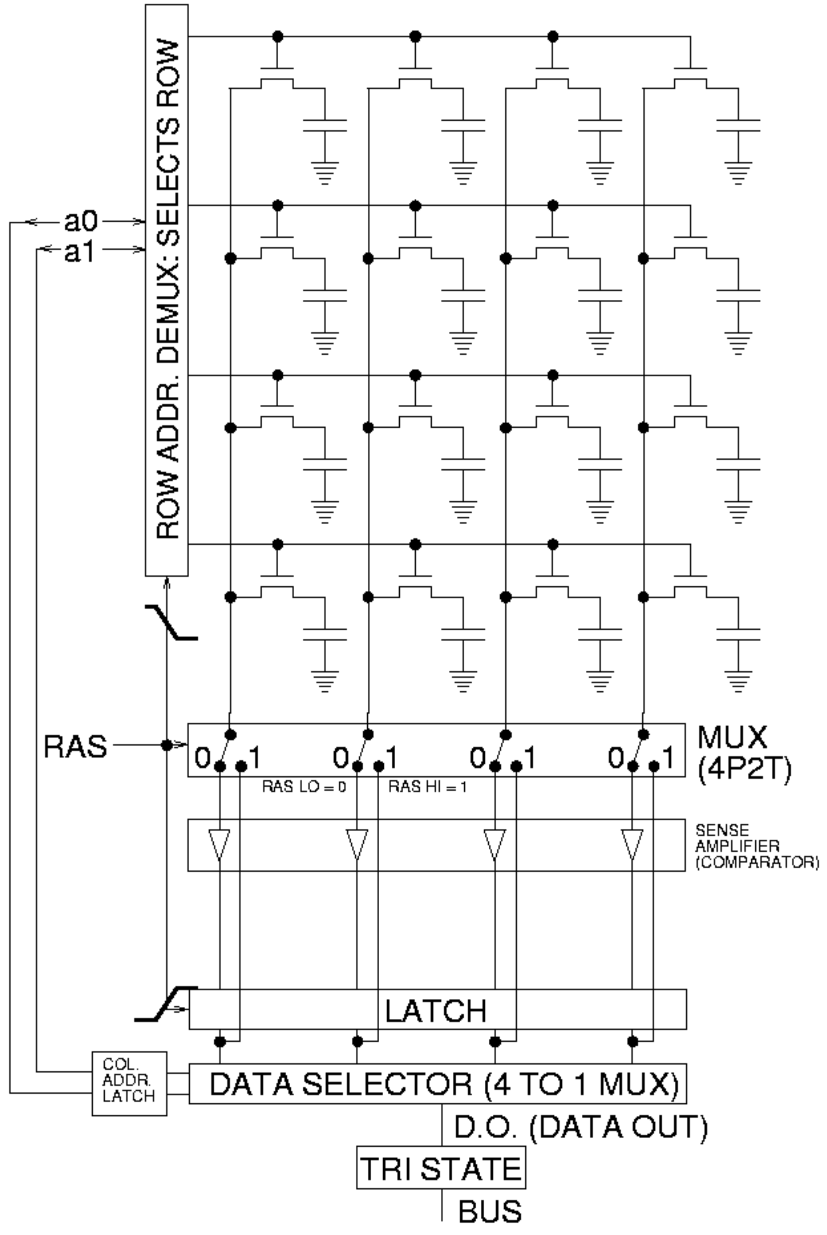
MOS Technology 6502 layout (1976). This die contains 3,510 individual transistors.



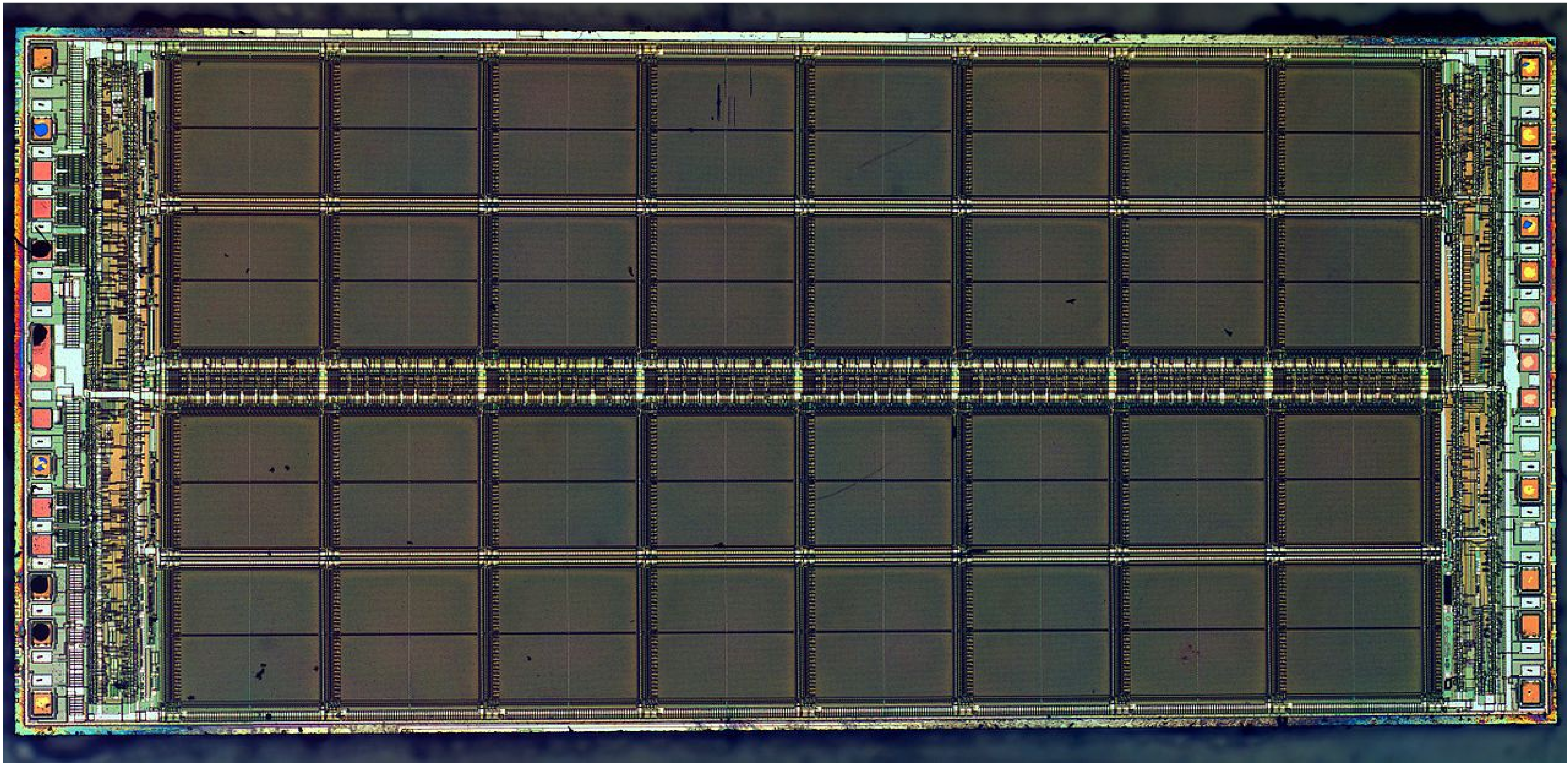
Intel *Sandy Bridge* CPU with integrated graphics (2011). This chip includes four x86-64 CPU cores, an L3 cache, GPU, and memory controller all on a single die. This whole die contains over 1 billion individual transistors (compare to 3,510 in the 6502).

IC Design before automation

- Manual design at the silicon level is still done for repetitive stuff where even a small improvement over what automation can do is significant – mostly memory devices like DRAM and Flash.



4x4 Array of DRAM cells



Micron MT4C1024 1Mbit DRAM (Circa 1990). Most of the space on the chip is taken up by a bunch of identical looking blocks of DRAM cells. This is a lot larger than our 4x4 cell example (1 million cells arranged in a 2048 x 512 array as opposed to 16), but still tiny compared to a modern DRAM IC which can easily contain 4Gbits or more.

IC Design Automation

- As you can see, designing a modern IC by hand simply wouldn't be practical.
- So, as computers became more powerful, Electronic Design Automation (EDA) tools were developed, and used to design subsequent generations of computers.
- Which then led to more complex computers, and more complex design tools for designing the next generation of computers.

Introduction – Hardware Description Languages

- We'll talk more about the physical logic later when we get to FPGAs.
- A Hardware Description Language (HDL) is similar to a programming language, but is used to model the behavior of hardware.
- The most common HDLs are **Verilog** and **VHDL**. I'm only going to discuss Verilog here since it's the language used at CMU, and the one I am most familiar with.

Basic Verilog

- I'm not going to start with “Hello, World”.
- That's actually kind of tricky to do in hardware.
  - We'd have to deal with interfacing to a display device.
  - Or we could cheat and print “Hello, World” on a piece of colored glass and put a light bulb behind it. But that would be boring.
- Here's some basic Verilog. Now let's go over what it does.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

Basic Syntax

- Looks a lot like C.
- With enough differences to be annoying.
- 'begin' and 'end' instead of curly braces.
- Except when you use other words.
  - Module/endmodule is the big one.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

Basic Syntax

- Modules are like a class in an object-oriented language.
- Modules can easily be reused, or instantiated inside other modules.
- Can you write an entire large design in one module? Sure, but that's basically the same as writing an entire large C program inside int main().

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

Input and Output

- An **input** to a module is a signal that comes from outside the module.
- An **output** of a module is a signal that the module drives for the outside world.
- There's also **inout**, which is both. With this, you have to be careful not to have multiple things driving the same signal at the same time.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

Inputs and outputs of the module are like parameters of a function, except the 'return values' are the outputs.

Signal Types

- There are many types of signal in Verilog, but I'm just going to cover **wire** and **reg**.
- A **wire** behaves like a literal piece of wire. Wires have no sense of memory, and are driven by whatever they're connected to.
- A **reg** behaves like a latch, or a variable in a software language.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

This is a 4-bit wire named 'a'.

This is a 5-bit reg named 'out'.

Signal Widths

- Software languages typically have different size types (char, int, long, double, etc.)
- Since we're working in hardware here, we have to manually specify the width of signals in bits.
- When we declare a signal (wire, reg, etc.) we do this by putting some numbers in square brackets after the type.
- When we assign a value to something, we use the notation N'*base*.
- If your signal isn't wide enough to hold a particular value, the extra bits get cut off.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

This is a 4-bit wire named 'a'.

This is a 5-bit reg named 'out'.

This assigns the value 0 (5 bits wide) to 'out'.

Useful Tip

- So what's this **`default_nettype none** business?
- One annoying default in Verilog is that if you don't explicitly declare a signal, it is assumed to be a 1-bit wire.
- This can lead to very annoying bugs if you make a typo in a signal name.
- The ` character denotes a preprocessor directive (similar to # in C). default_nettype none switches the default net type from **wire** to **none**, so if you typo a signal name you will get a compiler error instead of a broken design.
- Use it. I guarantee it will save you from hours of head scratching and quality time spent with the waveform viewer that could be better spent doing other things.

```
`default_nettype none
```

This simple line of code will save you much anguish.

```
module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

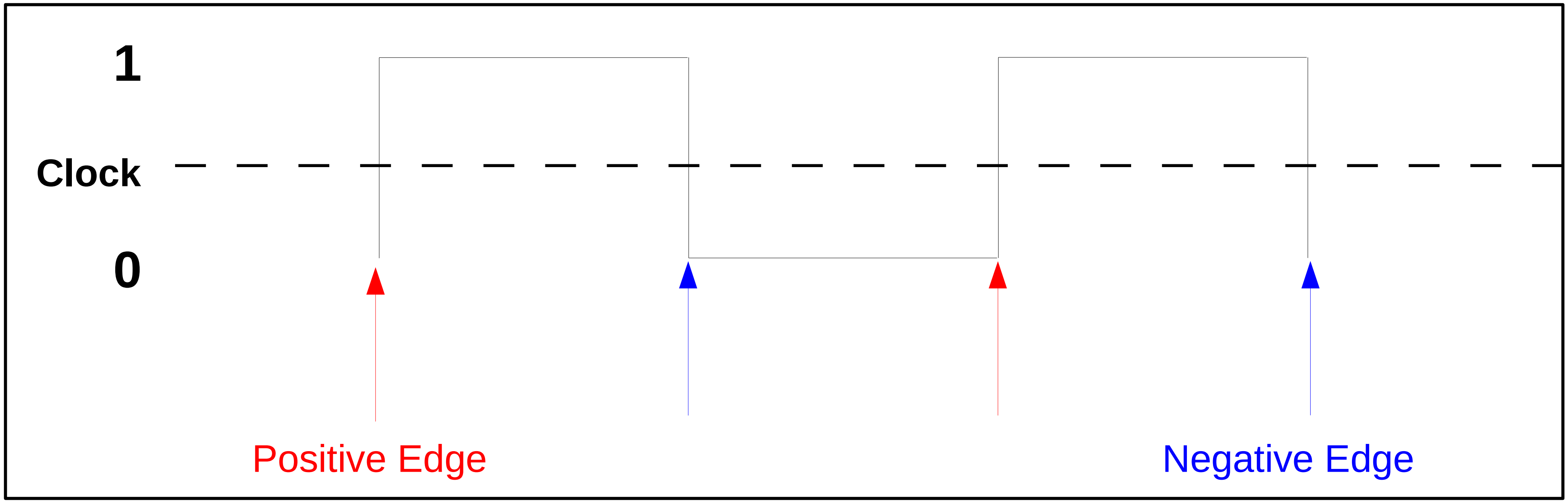
Always Blocks

- Code inside an **always** block is evaluated by the simulator (or implemented in the FPGA) whenever items in its **sensitivity list** change.
- The sensitivity list is the list of items inside parentheses after the @ symbol.
- This list can contain any number of signals, as well as a few conditions on the signals. The main ones are **posedge** and **negedge**, which mean that the block should only be evaluated on the positive or negative edge of the signal, respectively.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

What's an edge?

- An **edge** is a transition in a signal. We usually think about edges when we're dealing with clocks.
- A **positive** edge is a transition from low (0) to high (1).
- A **negative** edge is a transition from high (1) to low (0).
- Signals that change on an edge (for instance the stuff in our **always @ (posedge clock)** block are known as **edge-triggered** signals.



What's going on inside here?

- In our basic example, there's one **if...else** statement inside our **always** block. What it does is pretty simple.
- The behavior for **if (reset)** will happen if the **reset** signal is high. Here, we set the **out** register to all zeros, like hitting the clear button on a calculator.
- The **else** block handles the case where **reset** is not active. Here, **out** is assigned to be the value of **a+b**. Remember that this is within an **always @ (posedge clock)** block, so the output at the **out** register only changes on the clock edge, and the **a** and **b** inputs must be valid on the clock edge.

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

So what's this <= thing?

- You may have noticed that the variable assignments inside the block use a “<=” operator instead of just “=”.
- This indicates *non-blocking* assignment, which means that the assignments are not dependent on any other assignments during the same clock period.
- Standard 'good practices' are to use non-blocking assign in clocked logic (for instance **always @ (posedge clock)**), and blocking assign in combinational logic (for instance a continuous assign statement).
- You can read more about this at http://www.asic-world.com/tidbits/blocking.html

```
`default_nettype none

module fourbit_adder(
  input wire[3:0] a,
  input wire[3:0] b,
  input wire reset,
  input wire clock,
  output reg[4:0] out
);

  always @ (posedge clock) begin
    if(reset) begin
      out <= 5'b0;
    end else begin
      out <= a + b;
    end
  end

endmodule //fourbit_adder
```

OK, so how does this become hardware?

- We've seen some basic Verilog code, but how does this code turn into actual hardware?
- We could design a custom IC, but the initial set-up costs for that are on the order of millions of dollars.
- We could manually wire together a bunch of 7400-series logic like it's 1985, but then why bother with Verilog?
- Or, we could use some form of programmable logic.

Definition: Programmable Logic

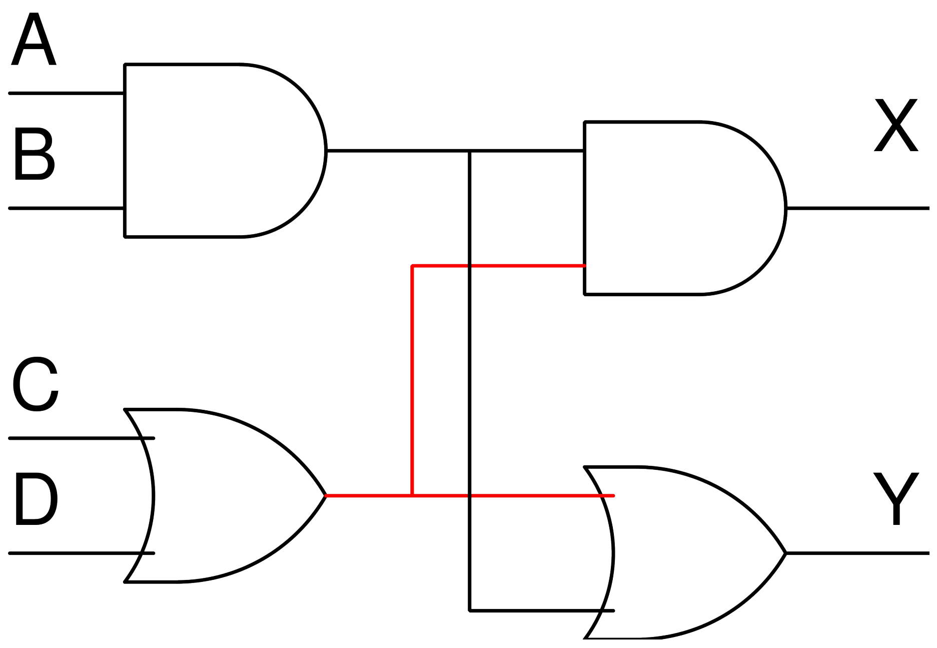
- Programmable logic circuits are digital circuits (computer chips) that can be configured after manufacturing.
- As opposed to a full custom ASIC that requires custom masks.
- Or a device called a ULA (Uncommitted Logic Array), which is sort-of programmable, but only in manufacturing.
  - These are less expensive than an ASIC in terms of setup cost, but less flexible than an FPGA since they are configured at manufacture time.
  - Some FPGA vendors (Altera for instance) sell ULAs that are designed to be similar to their FPGAs, so someone building a large enough volume of a design can switch from an FPGA to a semi-custom part and realize some cost savings.

Simple Programmable Logic

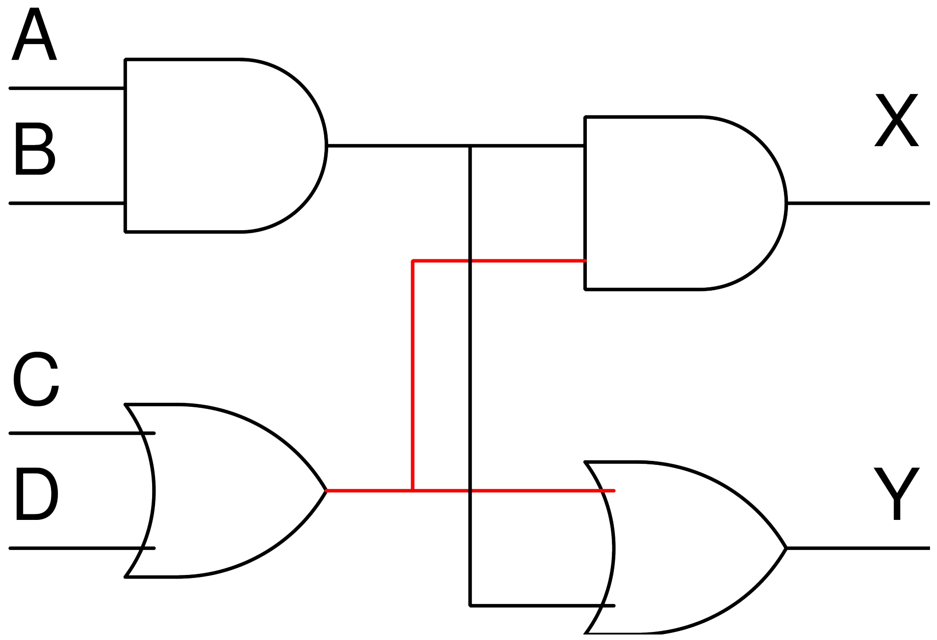
- Given a Boolean logic function of N inputs and M outputs, how could we implement this function in a way that we could reprogram the logic circuit if desired?
- Answer: Use memory to store the *truth table* of the logic circuit. Then, if we want to change the behavior of the circuit, all we have to do is reprogram the memory.

Simple Programmable Logic

- How could this work?
- Let's take a look at a simple Boolean logic circuit.
- This circuit has 4 inputs and 2 outputs, so our truth table will have 16 rows.



Truth Table for this circuit



Inputs				Outputs	
A	*B*	*C*	*D*	*X*	*Y*
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	1
1	0	1	0	0	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	1	1
1	1	1	1	1	1

Implementing this as memory

- To implement this truth table in a block of memory, our inputs A through D will be the address.
- The outputs X and Y will be the data.
- So, our ROM would be programmed as follows.

Address	Data
0	00
1	01
2	01
3	01
4	00
5	01
6	01
7	01
8	00
9	01
A	01
B	01
C	01
D	11
E	11
F	11

What's the point?

- Is this more efficient than actually wiring up 4 logic gates? Certainly not.
- So why bother?
- Well, what if we wanted to change the behavior of the circuit.
- With actual logic gates we'd have to replace the gate. But since this is just a truth table in memory, all we have to do is rewrite some memory.

Types of Programmable Logic

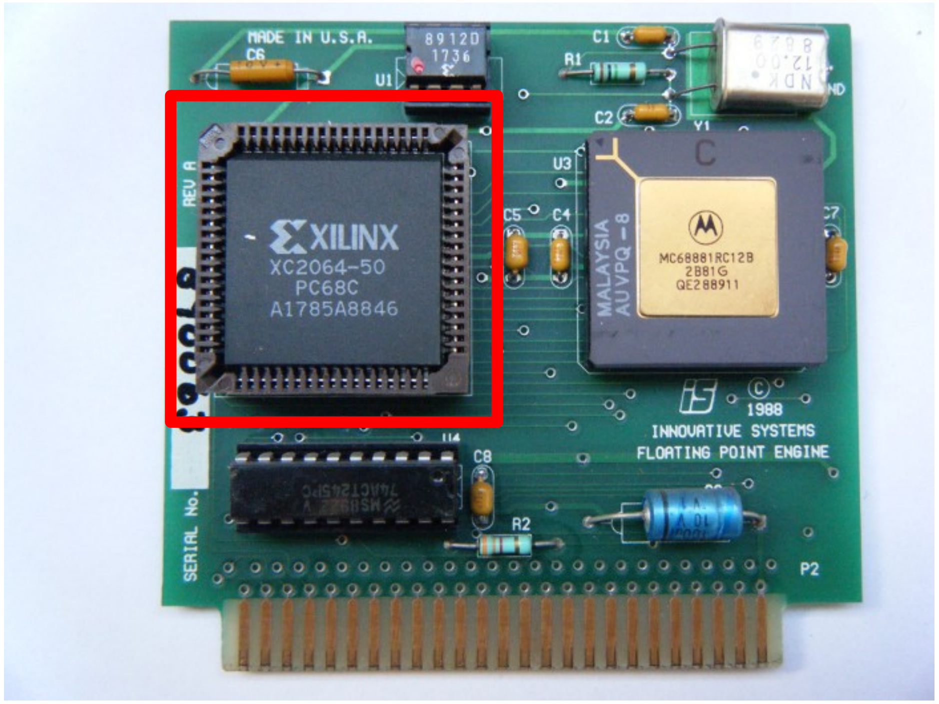
- There are various programmable logic devices that increase in performance, capacity, and cost.
- The main types encountered today are CPLDs (Complex Programmable Logic Devices) and FPGAs (Field Programmable Gate Arrays).
- There are also smaller, less-complex devices known as PALs (Programmable Array Logic) and GALs (Generic Array Logic, basically a reprogrammable PAL) but these are not commonly used in new designs.

What's an FPGA?

- Programmable logic, but a lot more advanced than the simple example we just reviewed.
- FPGAs are composed of three main components
  - **Programmable Logic Blocks** – A look-up table similar to our truth table example, and some memory (registers).
  - **Interconnect Blocks** – Configurable signal routing logic that can be used to connect PLBs to each other and to the I/O blocks.
  - **I/O Blocks –** Logic for interfacing to the physical pins on the FPGA.
- Modern FPGAs often contain other components, but we'll discuss that later.

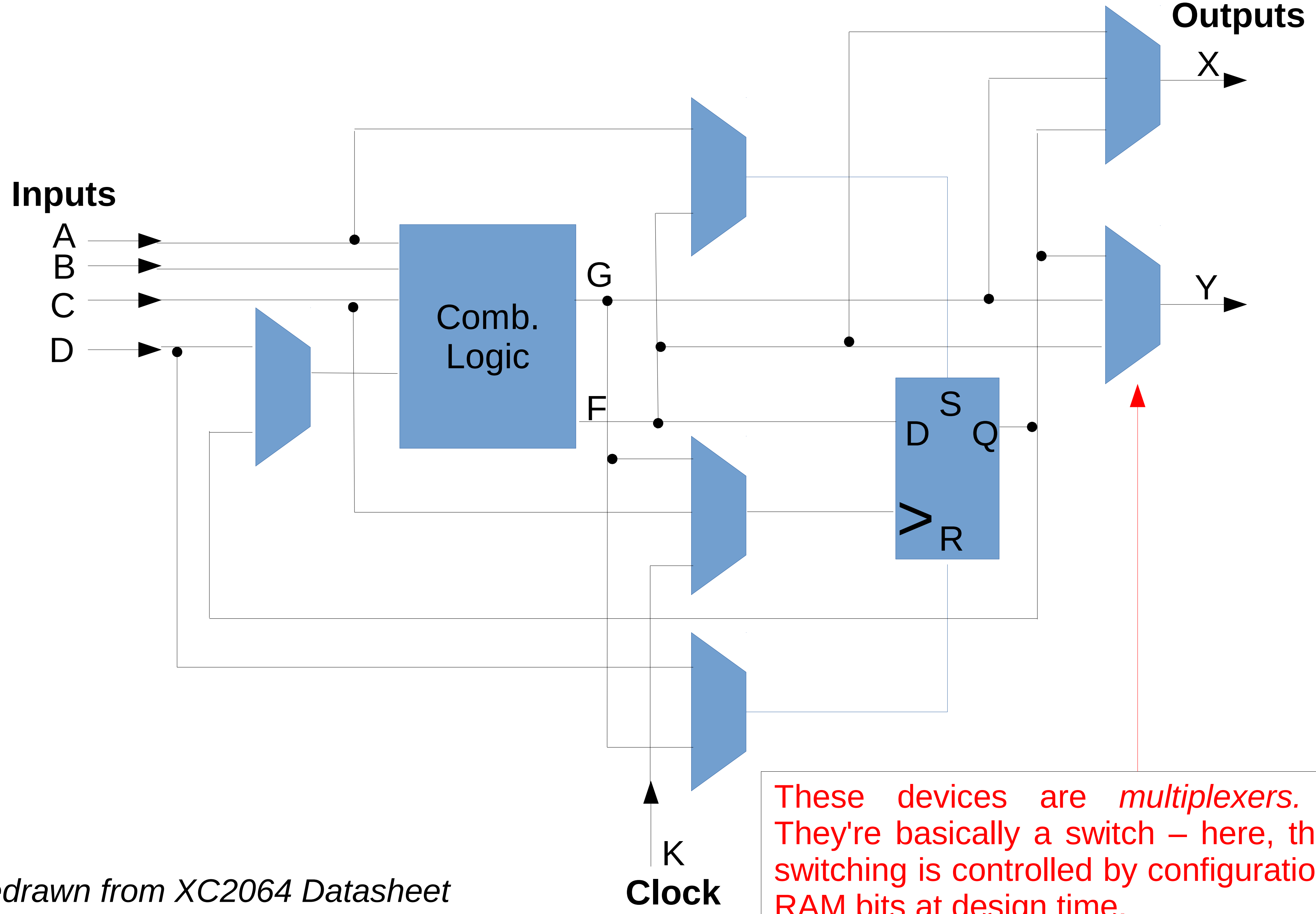
Understanding a Simple FPGA

- We'll take a look at the Xilinx XC2064.
- This is the first commercial FPGA, from 1985. It contains 64 PLBs, 58 I/O pins, and has a *capacity* of 1,200 logic gates.
  - This means the XC2064 can replace up to 1,200 individual AND, OR, etc. gates.
- This explanation won't be all-inclusive, but if you want more, I'll have a link to the XC2064 datasheet (which explains how the chip works in great detail) at the end of my slides.



XC2064 manufactured in week 46 of 1988, on an Apple IIgs math coprocessor card. The other large IC on the board is a Motorola 68881 Floating-Point Unit. The small IC above the FPGA is a serial EEPROM that the FPGA loads configuration data from.

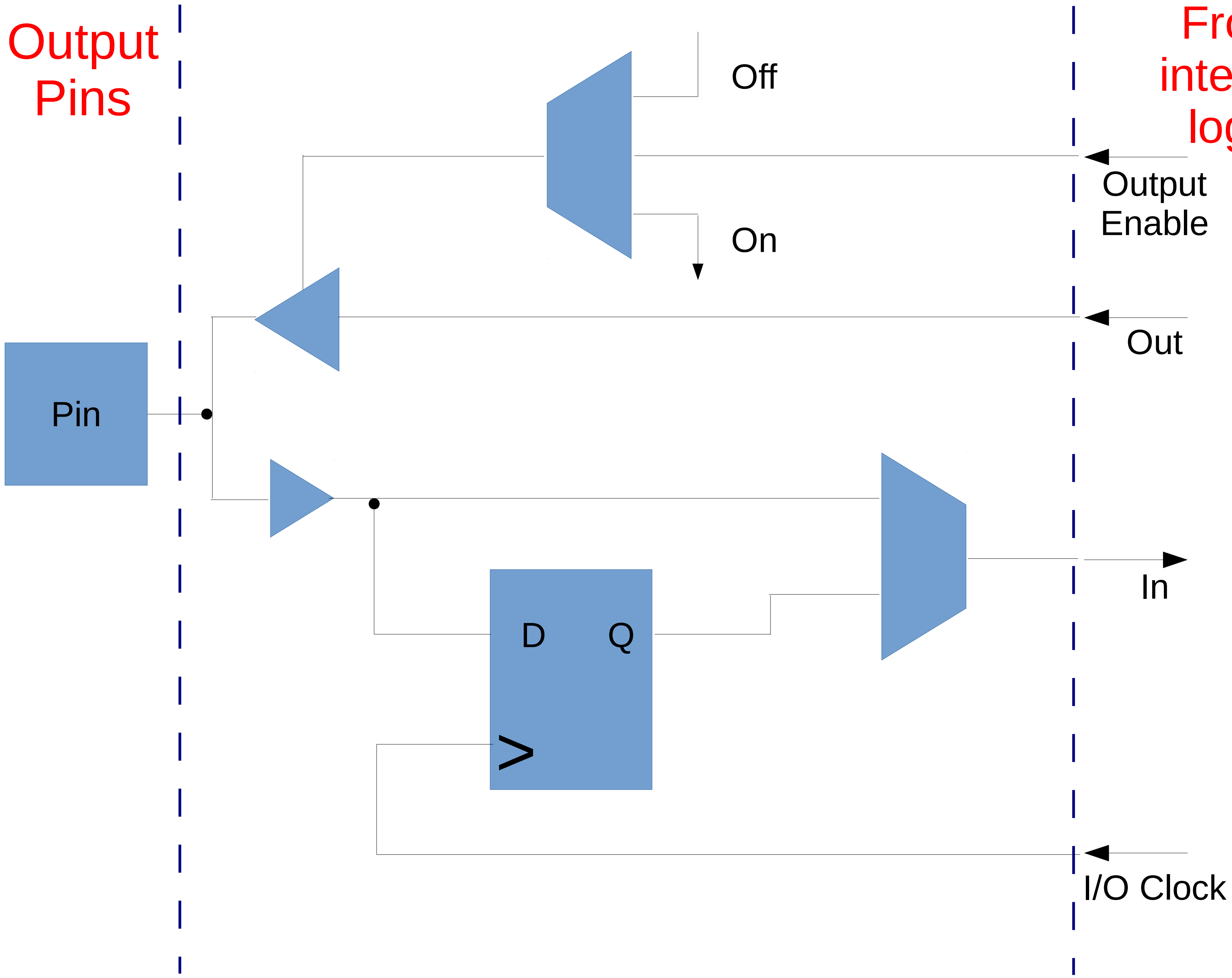
XC2064 Programmable Logic Block



These devices are *multiplexers.* They're basically a switch – here, the switching is controlled by configuration RAM bits at design time.

Redrawn from XC2064 Datasheet

XC2064 I/O Block



Interconnect

- The way the XC2064 did it was pretty simple.
  - Basically a 'grid' of horizontal and vertical wires with programmable switching to connect logic blocks to each other and to I/O blocks.
  - Also a few long lines for signals that have to travel a long distance and/or need minimal skew, and one global clock line that can be routed to all of the 'B' and 'K' inputs of the PLBs (or these can use other clock sources).
  - I didn't have time to do detailed diagrams on this, but we can go through some stuff from the datasheet at the end if people are interested.

Comparison to modern FPGAs

- Modern FPGAs still incorporate the same basic parts, but typically more of them.
  - For instance, Xilinx's current top of the line FPGA has over 3 million PLBs (and they're probably bigger than 4 inputs, 2 outputs as well), over 800 I/O pins, and over 40 megabits of registers just in the PLBs.
- Modern FPGAs also often incorporate some *hard logic* ranging from simple things like multipliers to entire ARM CPU cores, floating-point units, PCIe and Ethernet interfaces, etc.
- Modern FPGAs also often contain *block RAM*, small (but still larger than the register in an PLB) blocks of memory that your design can use. But if you need more RAM than that you're stuck interfacing to an external RAM chip.
  - The Xilinx chip I mentioned earlier has close to half a gigabit of block RAM.

Want to learn more?

- If you feel like taking an entire 12-unit class:
  - 18-240 would be a good starting point.
  - If you decide you really like this stuff, take 18-447 (Undergraduate Computer Architecture) and/or consider 18-545 (FPGA design) as a capstone course if you're an ECE major.
- If you just want to play with Verilog:
  - Icarus Verilog and Verilator are both free software (libre **and** gratis).
  - There are plenty of good Verilog references online.
- If you want to play with FPGAs:
  - The Computer Club owns a Novena (open hardware project with a fairly nice ARM CPU and an FPGA) that you're welcome to play with at Hacking Hours.
  - And/or you can find basic FPGA development boards relatively inexpensively <$50 for the really basic stuff, $100-200 for more powerful stuff. Both Xilinx and Altera provide free (but only as in gratis) development tools for all but their most expensive FPGAs.

Useful Links

- Xilinx XC2064 Datasheet
  - http://www.cmucc.org/~lroop/hdl_talk/xc2064.pdf
- Icarus Verilog (official site)
  - http://iverilog.icarus.com/
- Verilator (official site)
  - http://www.veripool.org/wiki/verilator
- Recipes for the cookies served at the talk
  - http://www.cmucc.org/~lroop/hdl_talk/
    - Two obviously named text files.
    - Some whole wheat flour was used, and the oatmeal raisin cookies were made with the full ½ teaspoon of cinnamon and the optional nutmeg.

Any Questions?