EE242

System Verilog & FPGA’s

What's an FPGA (or CPLD)?

- Field Programmable Gate Array
- Complex Programmable Logic Device
- These are names for…
  - Programmable devices
  - Can contain lots of gates/etc.
- CPLD's often non-volatile
  - Don't need to reload them after power-up
- FPGA's may be volatile

We'll Use: Altera MAX II

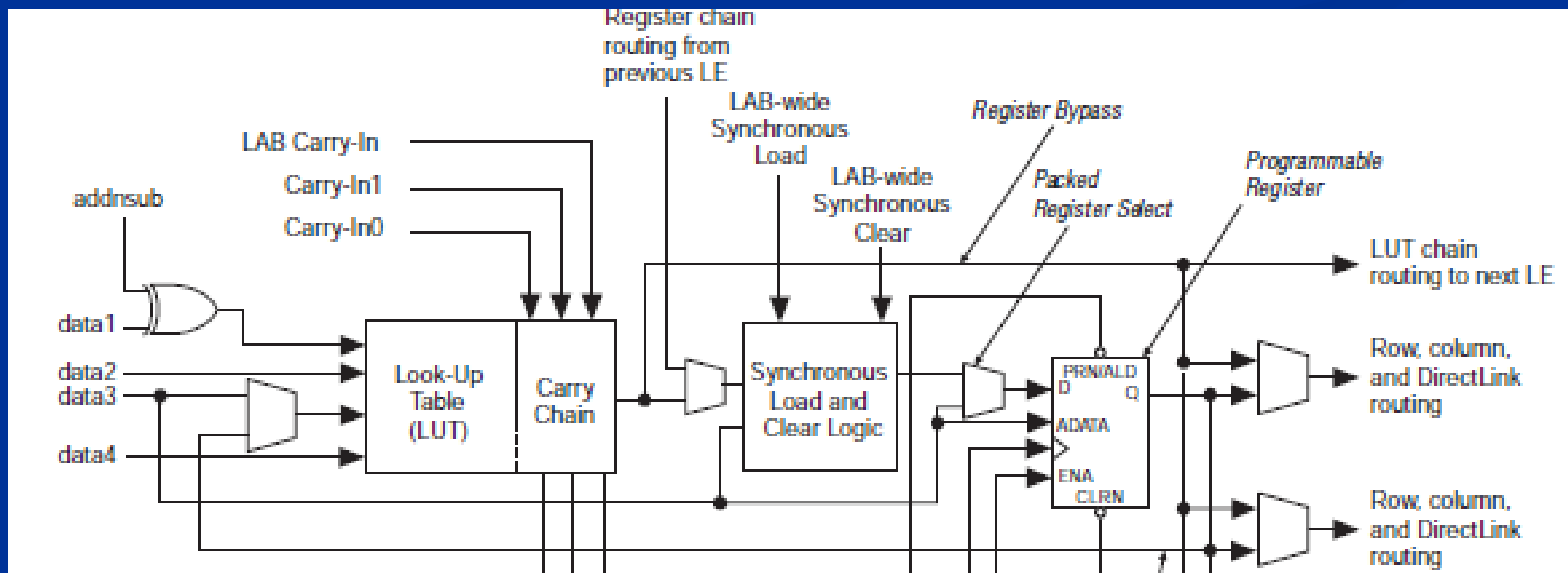
- Altera (now part of Intel) calls this a CPLD
- MAX II is a family of devices
  - Varying speed and capacity

Feature	EPM240/G/Z	EPM570/G/Z	EPM1270/G	EPM2210/G
Logic Elements (LEs)	240	570	1,270	2,210
Equivalent Macrocells	192	440	980	1,700
Maximum User I/O Pins	80	160	212	272

- Capacity
  - How much logic you can pack into the device
  - Expressed in Logic Elements
- What's a Logic Element? (next slide)

Logic Element

- The fundamental building block in MAX II
- Contains a flip-flop
- Contains a Look Up Table (LUT)
  - Can implement any logic function of up to 4 inputs



Working with CPLD/FPGA

- Don’t worry about configuring at the LE level
  - Tools will automatically manage these
- Instead, we work at a higher level
  - Schematic capture or
  - Hardware Description Language (HDL)

Verilog

What's an HDL?

- Hardware Description Language
  - Use text to describe hardware
  - Alternative to schematic capture
  - Top HDL's: VHDL and Verilog
    - Some SystemC also
- VHDL
  - Looks like Ada programming language
- Verilog
  - Looks like 'C' programming language
- System C
  - Built on C++ programming language

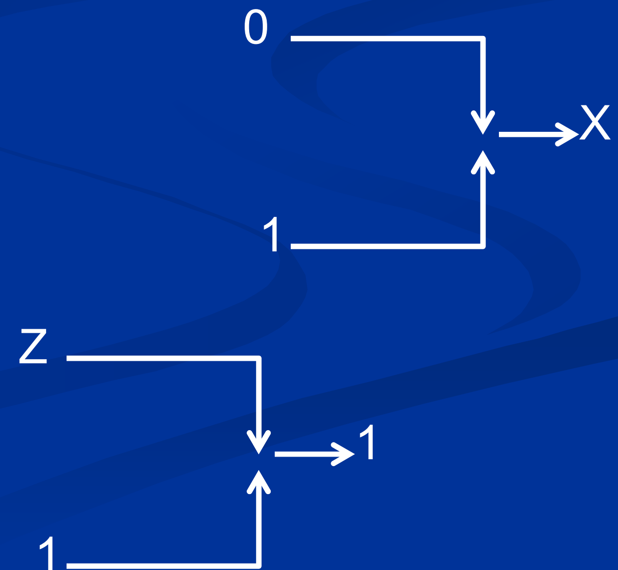
Verilog

- In my experience, Verilog is more common than VHDL for CPU design
- We'll use the latest variant: SystemVerilog
- IEEE standard 1800 (published 2012)
  - http://standards.ieee.org/getieee/1800/download/1800-2012.pdf
- This is a "big" language
  - Extends Verilog (IEEE 1364, 2005)
  - We'll only be using parts of it

Verilog Logic Values

- Four-value logic
  - 0: logic zero
  - 1: logic one
  - Z: high-impedance (float)
  - X: contention or uninitialized

- These combine intuitively



Single-bit or Multi-bit Logic Values

- "logic" is the 4-value type
  - To "declare" is to name something, assign a type

```
           // This is a comment
logic output;              // Single-bit "output"
logic [7:0] counter;       // 8-bit "counter"
```

- Verilog code will determine what the names do
  - Flop, wire, etc.
  - We'll see this later

Values

- Single-bit: just do it
  - X, Z, 0, 1

```
output = 1;   // drive '1' onto output
triSig = Z;       // Float triSig
```

- Multi-bit: can specify length and radix

```
bus = 32'bZ;   // 32 bits of Z,'b' = binary
value = 8'hA5; // 8b, hex
```

Expressions

- There are *many* operators
  - We typically just use a handful of them
- The usual math expressions are available

```
x = x + 2;       // add 2 to x
abc = def * 4; // def multiplied by 4
```

- Note that you can imply a lot of hardware
  - E.g., * operator: may turn into a multiplier
  - Think about what will “come out”

Expressions: Bitwise

- ~ (negation)

```
logic [7:0] x;
x = 8’b00110101;
x = ~x;      // x gets 11001010
```

- &, |, ^ (and, or, exclusive-or)

```
logic [7:0] x, y, z;
x = 8’b11000011;
y = 8’b00000010;
z = x & y;        // z gets 00000010
z = x | y;        // z gets 11000011
z = x ^ y;        // z gets 11000001
```

Expressions: Relational

```
logic rel;

rel = A == B;     // rel is '1' if A equals B
rel = A != B;     // rel is '1' if A not equal B
…
```

Combinatorial Logic Modules

Combinatorial Logic

- Also known as
  - Random logic
  - Combinational logic
- AND, OR, etc…
  - Equations, or expressions
- Verilog has certain places where you can put these

always_comb

- Used to contain logic that is *only* combinatorial
  - Not flip-flops or latches
- Example (from an EE345 CPU)

```
always_comb begin
    valid = memAccess.op.size != szNone;     // non-0  # of bytes to access

    memReq.memRd = valid & memAccess.op.isLoad;
    memReq.memWr = valid & ~ memAccess.op.isLoad;
        // …
end
```

Start of block

Intermediate signal

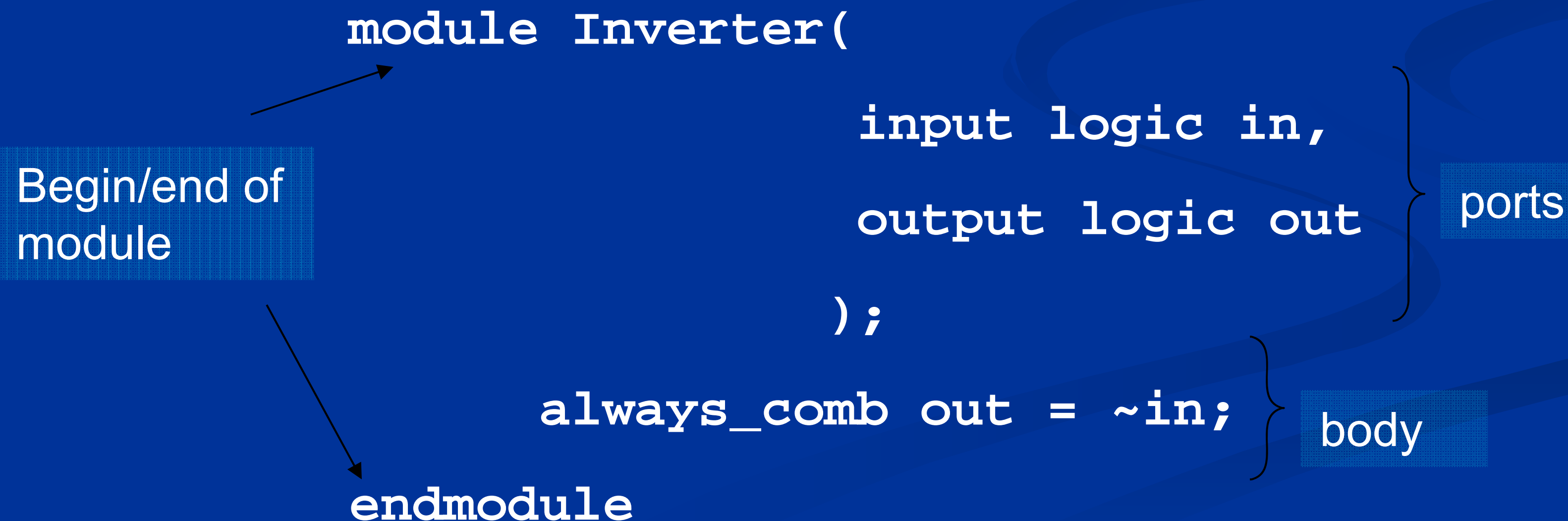
End of block

Combinatorial equations

Note: “blocking” assignment uses “=“. Effect is for assignments to happen in order.

Modules

- A "module" is like an IC for Verilog
  - Collection of logic
  - Defined inputs/outputs (ports)
  - May be instantiated multiple times



Using Modules

- Create an instance, connect ports

```
logic sigInput, sigOutput;


Inverter inv1(
            sigInput,  // in
            sigOutput  // out
            );
```

Modules and MAX II

- When creating a design for MAX II…
  - Make it a module
- The ports names are the pins on MAX II

Convention: main module named "top"

```
module top(
            input logic in,
            output logic out
        );
    always_comb out = ~in;
endmodule
```

"in" and "out" will be the pins on your MAX II design

Sequential Logic

Always_ff

- Controls code that results in flops
- You supply "sensitivity" list
  - Signals that can change the flop's state
  - Usually just a clock

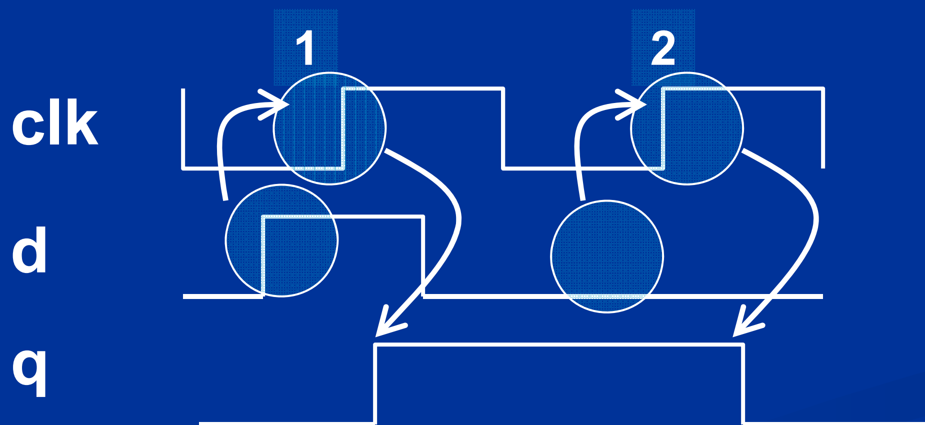
```
always_ff @(posedge clk)
  // put logic here
```

- Flops trigger on clock edges
  - Can specify *posedge* or *negedge*
  - We normally uses posedge

Example: D flop

- Output Q gets value of D on positive edge
- Maintains value until next positive edge

```
always_ff @(posedge clk)
      q <= d;
```



At edge 1

- q follows d to 1

At edge 2

- q follows d to 0

What’s up with <= ?

- “<=“ is another assignment operator
  - “non blocking assign”
- “=“: assignments happen in linear order
- “<=“: happen in random order
- Use “=“ for combinational logic
  - Because you’re building up layers of logic
  - Gate A feeds gate B: evaluate “A =“ before “B = A…”
- Use “<=“ for sequential logic
  - Because everything happens next clock

Example: Synchronous Reset

- Synchronous reset is preferred
  - Avoid metastability/race issues
  - Map into more PLD/FPGA devices

```
always_ff @(posedge clk)
      if(rst)
        q <= 0;
      else
        q <= d;
```

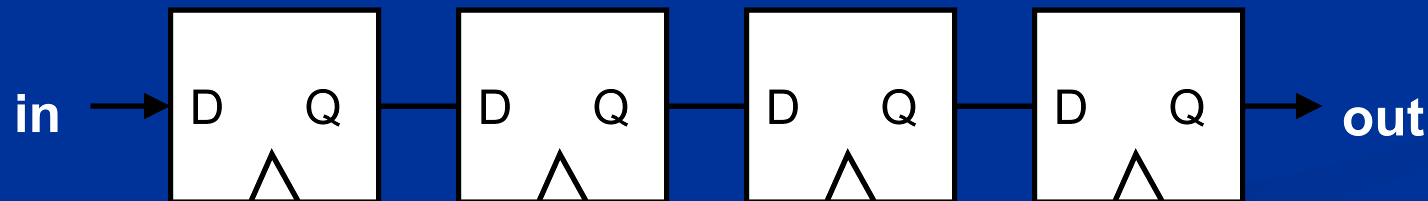
Example: Binary Counter

- Looks like an adder
- Tools will optimize into counter

```
always_ff @(posedge clk)
      count <= count + 1;
```

Example: Shift Register

- Shift register moves data one step on each clock

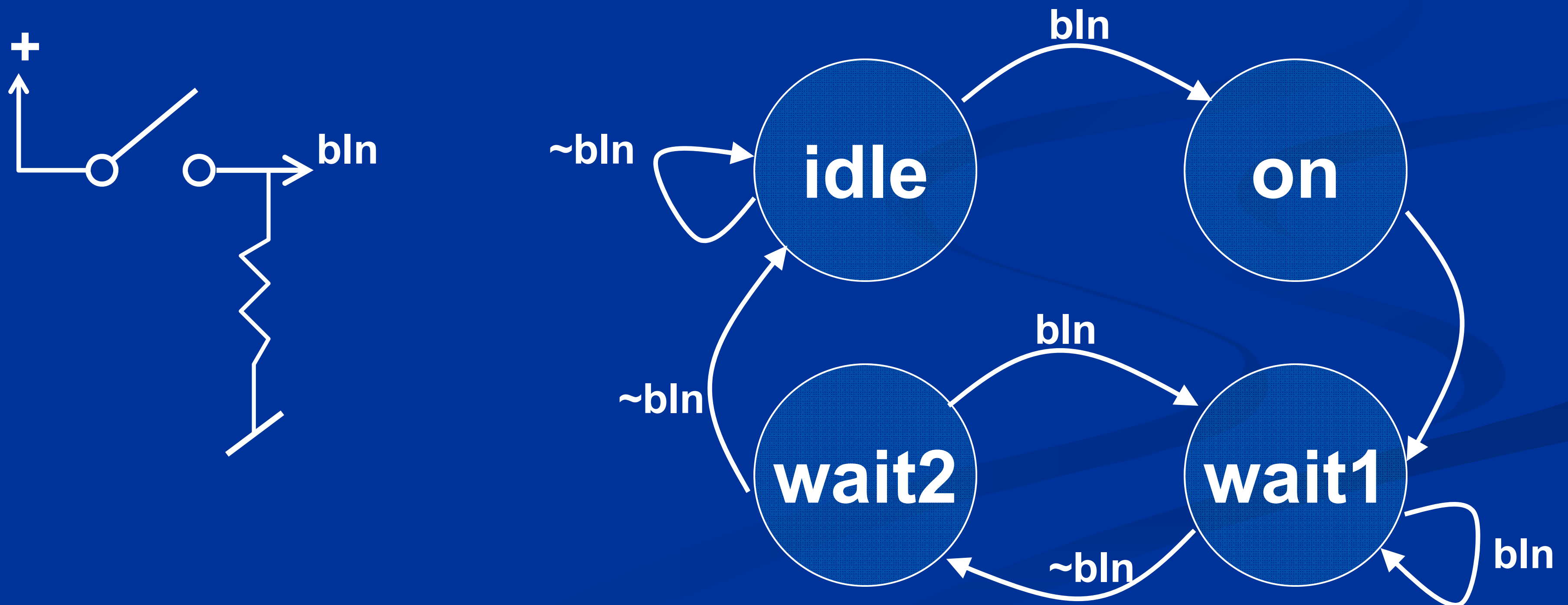



```
logic [3:0] sreg;

always_ff @(posedge clk)begin
  sreg[0] <= in;
  sreg[1] <= sreg[0];
  sreg[2] <= sreg[1];
  sreg[3] <= sreg[2];
end
```

```
logic [3:0] sreg;

always_ff @(posedge clk)
begin
  sreg[0] <= in;
  sreg[3:1] <= sreg[2:0];
end
```

Example: state machine

- Debounce a switch
  - switch pressed: *bIn* is 1
  - “Bounce” if bIn has short pulse after switch released



Example: State Machine

```
module Debounce  (
   input logic clk,         // 10-20 Hz for debounce
   input logic bIn,         // 1 = "button active"
   output logic pressed     // 1 = button-press
   );
   enum         {idle, on, wait1, wait2 } state;

   always_ff @(posedge clk)
      case(state)
        idle:  state <= bIn ? on : idle;
        on:    state <= wait1;
        wait1: state <= bIn ? wait1 : wait2;
        wait2: state <= bIn ? wait1 : idle;
      endcase

   assign pressed = (state == on);
endmodule // Debounce
```