EECS 470: Computer Architecture

Lab 1

(while you're waiting, please boot into Linux)

My Name is Pat

- Undergraduate Senior
- Primary Interests:
  - Embedded Systems
  - Sensor Networks
  - Novel Hardware Platforms
    - (really tiny stuff)

My Name is Pulkit

- I am a Masters student in CSE.
- My interests lies in
  - Android OS
  - Compilers
- Current Research
  - Improving the JIT compiler that comes with the Android OS.

10 worst times for computers to fail

10. About to beat angry birds high score
9. Banking
8. Automated car
7. Nuclear weapons manufacture

A problem has been detected and Windows has been shut down to prevent damage to your computer.

The problem seems to be caused by the following file: SPCMDCON.SYS

PAGE_FAULT_IN_NONPAGED_AREA

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x00000050 (0xFD3094C2,0x00000001,0xFBFE7617,0x00000000)

*** SPCMDCON.SYS - Address FBFE7617 base at FBFE5000, DateStamp 3d6dd67c

10 worst times for computers to fail

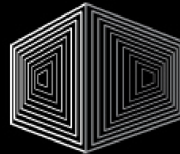
6. During your first discussion of the semester

5. To be continued …

Impact of errors

- Functional bugs

17 Jan 1995	**FDIV bug:** Some of the defective chips were later turned into



HITBSECCONF2008
27th - 30th October 2008 MALAYSIA

Kris Kaspersky: Remote Code Execution Through Intel CPU Bugs

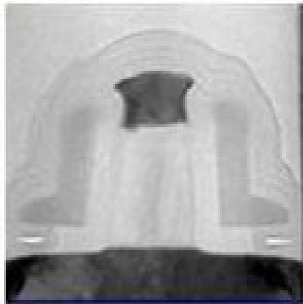
- Electrical failures

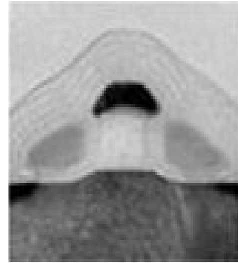
1024-bit RSA secret key extracted in **100 hours**

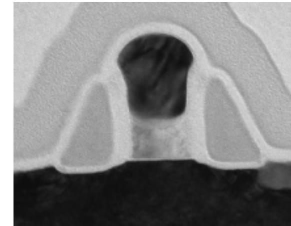
- Transistor faults

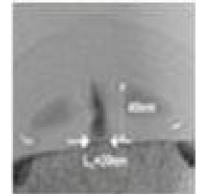
Sandy Bridge Bug 2X Costly as Pentium FDIV Bug

Trends in today's processors


90nm


65nm


45nm


32nm

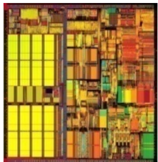
Shrinking transistor size

waning reliability

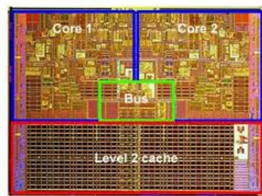
Increasing cores and complexity

verification challenges

Pentium4

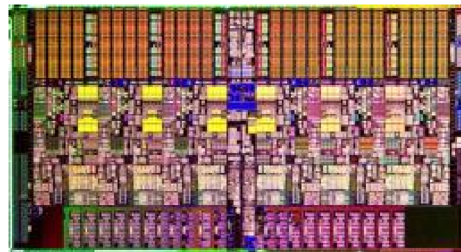

1 core 2000

Core2Duo

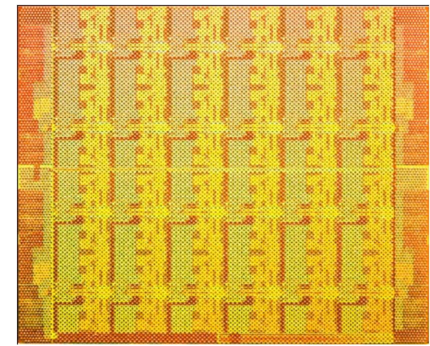


2 cores 2007

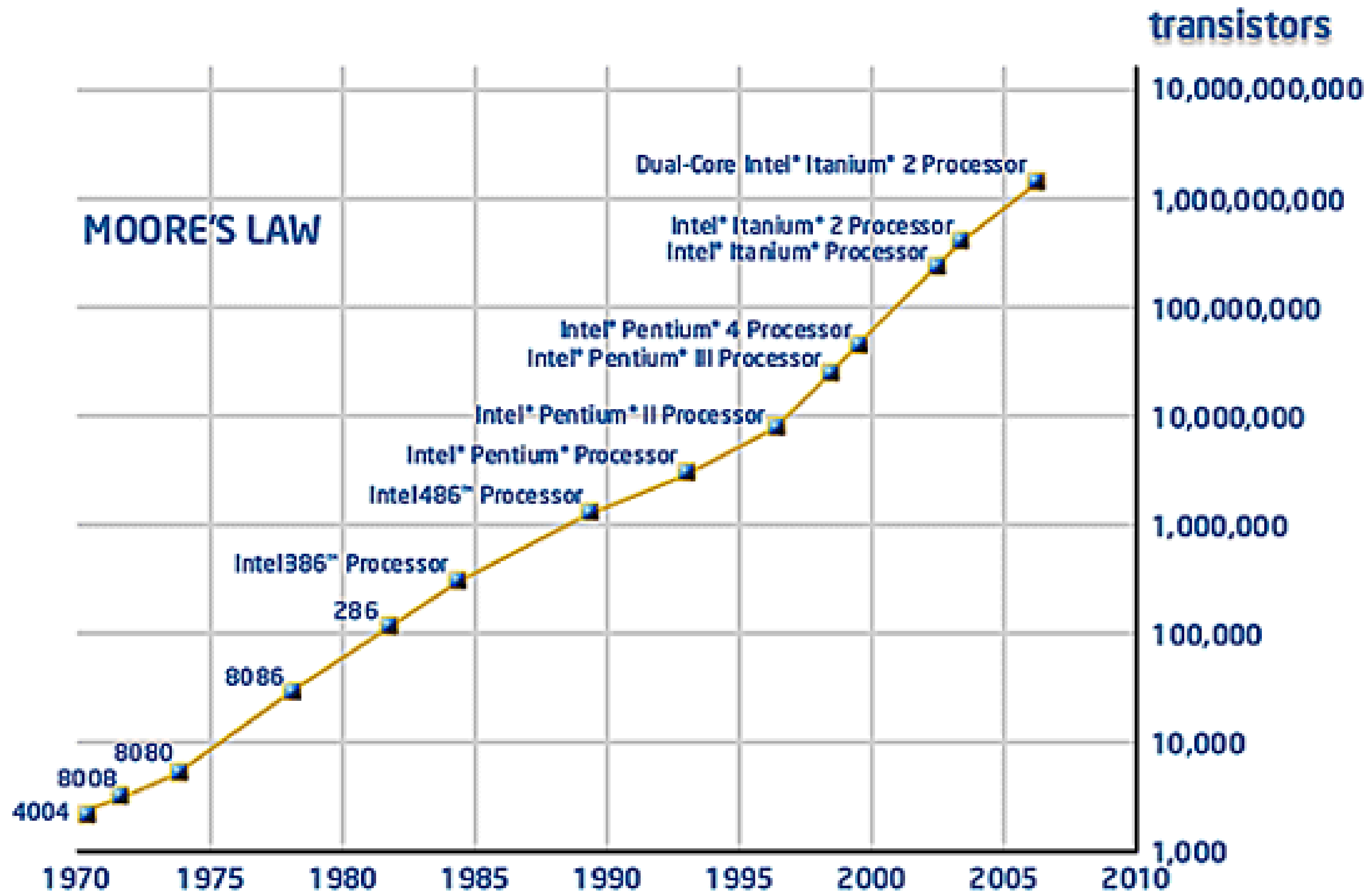
Core i7 980X


6 cores 2010

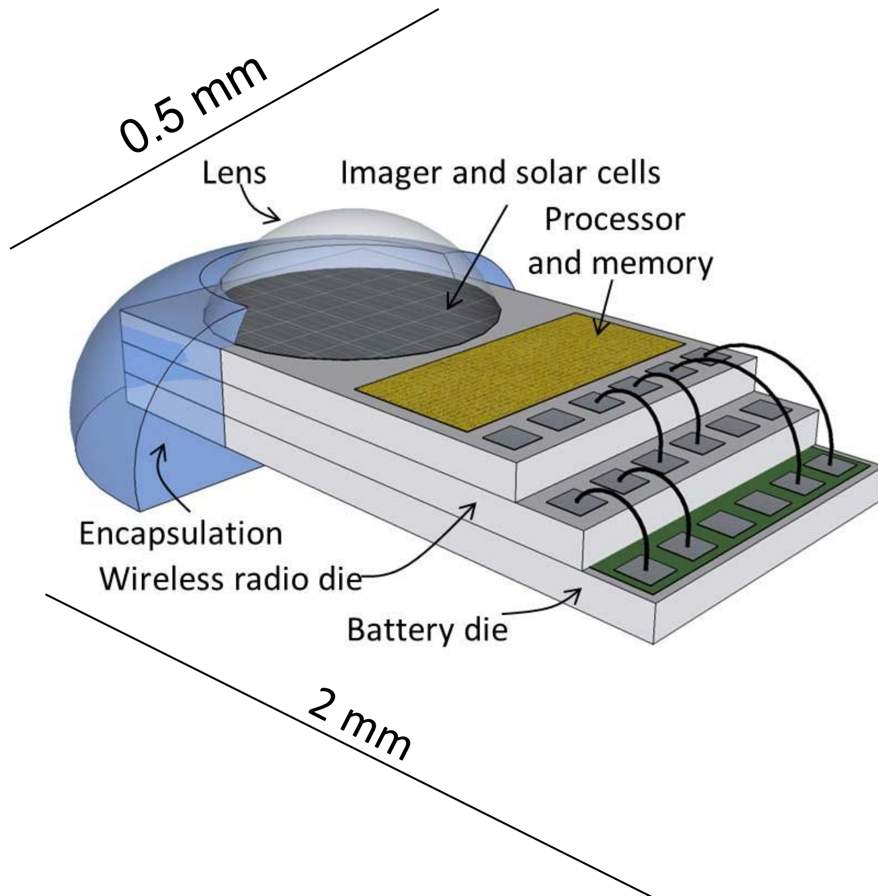
Core SCC


48 cores 2010

Because It Wouldn't Be a First EECS Lecture Without It...



Systems Are Shrinking Too



- Knowing the whole stack lets us build completely new technology
- In this course
  - You'll build the better part of a Pentium III
  - In 2 months...!

Administrative

- Pulkit Gupta: gpulkit@umich.edu
- Pat Pannuto: ppannuto@umich.edu
  - Please put EECS470 in the subject
  - Post technical questions to the forum
    - https://phorum.eecs.umich.edu

- InLabs
  - Fri 10:30-12:30 in CSE 1620
  - Fri 2:30-4:30 in CSE 1620

- Office Hours
  - (Pat) Sun 1-4, Mon 10:30-12:30
  - (Pulkit) Wed 4:30-7, Sat 2-4:30
  - DC3NE (3rd floor of the Dude, NE corner)

Why Come to InLabs?

- Cause its mandatory.
- LEARN TO VERILOG (HANDS ON SESSIONS)
- Project information/hints
- About 6 InLabs this semester
  - Substitute InLabs with more office hours later in the semester
- Chat about computers

Programming Projects

- Project 1: priority selector
- Project 2: pipelined multiplier, square root
- Project 3: Very Simple 5-stage Pipeline
- Final Project: out-of-order processing core

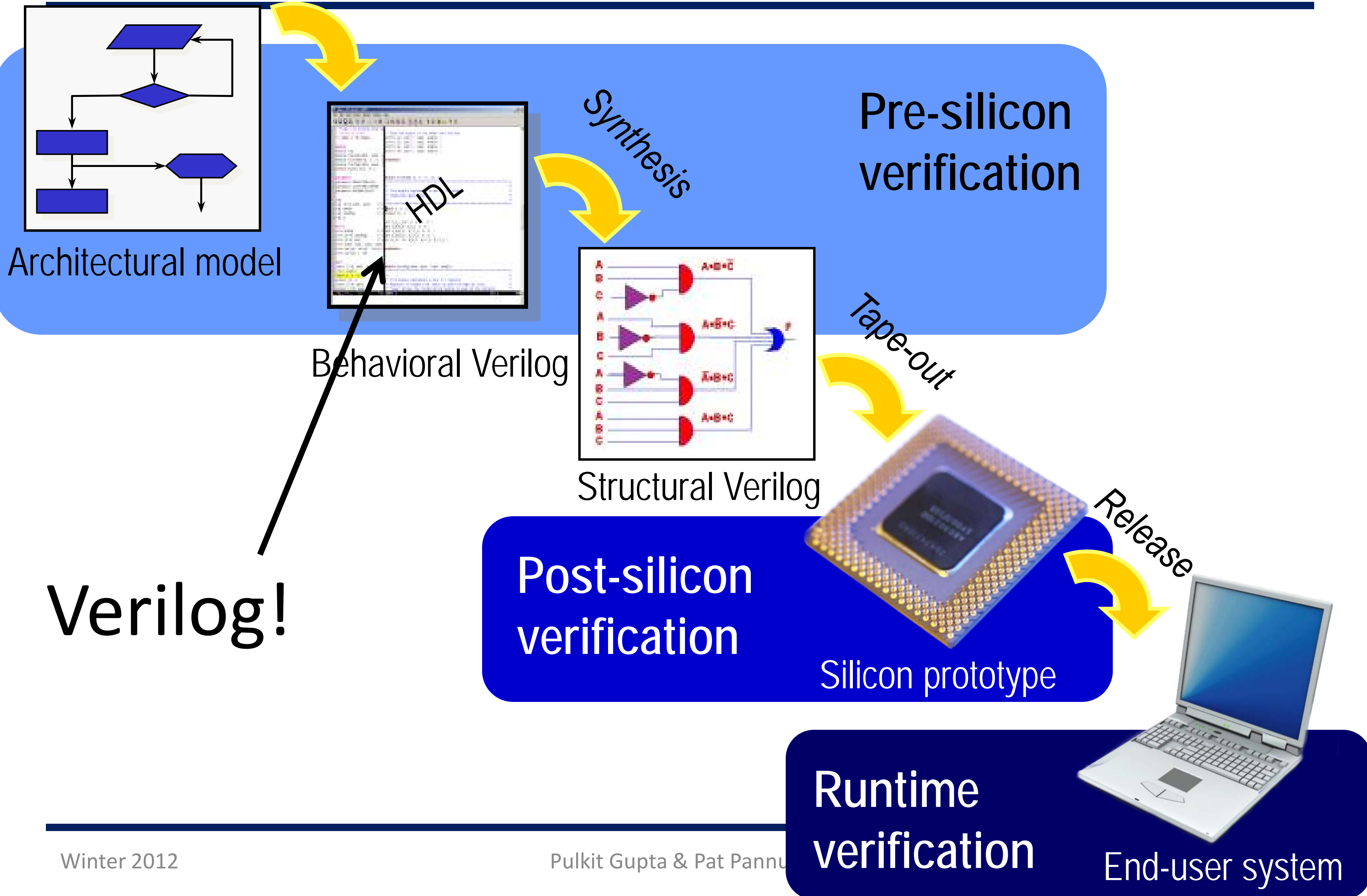
- Each of these will take a non-trivial amount of time especially if you're learning Verilog

Final Project

- Processing Core – subset of Alpha instructions
- Groups of 4-5
  - Start thinking about your groups soon
  - You'll be spending a lot of time with these people – work with people you get along with
- LOTS OF WORK
  - 100 hours/member will be a minimum
  - More Likely 150 – 300 hours/member
  - Many groups have spent much more time than that
- Class is heavily back loaded

YOU SHOULD START EARLY

How computers are born



What is Verilog?

- Hardware Description Language
  - Statements in Verilog describe real hardware

```
module mux(select_in,a_in,b_in,muxed_out);
    // all inputs and outputs listed
    input select_in, a_in, b_in;
    output muxed_out;

    // some logic
    assign muxed_out = select_in ? b_in : a_in;
endmodule
```

VERILOG IS NOT PROGRAMMING

SERIOUSLY.

IT ISN'T.

Thinking about Verilog

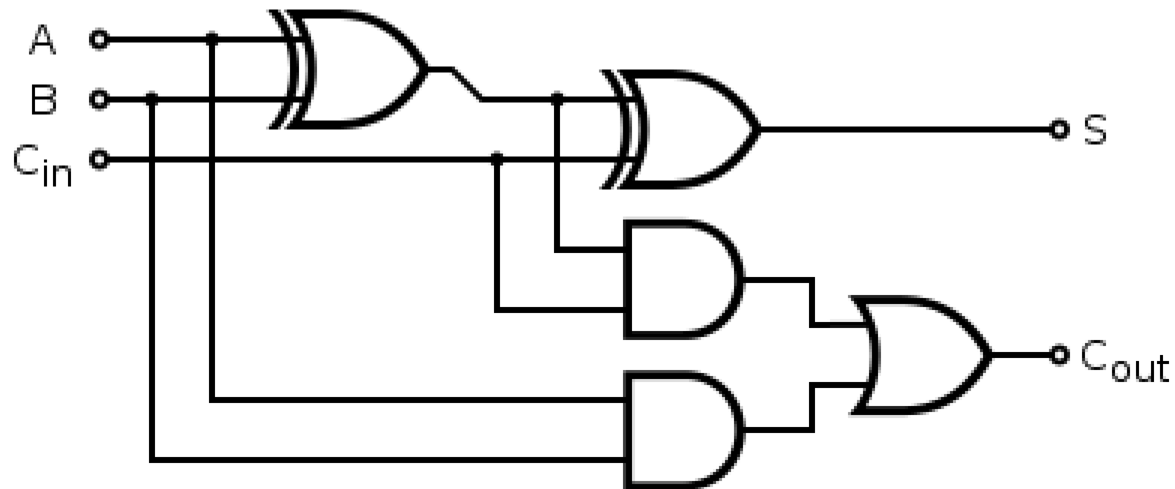
- Think in parallel

- Be able to explain at least one way that your code might be converted into hardware

- Consider drawing a diagram for each block

Example

- Build a simple 1-bit full adder



Answer #1

Simple Adder

```
module add(a,b,cin,sum,cout);
    input a, b, cin;
    output sum, cout;

    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

parallel

Modules

- Basic organizing units in Verilog
- Black boxes with inputs, outputs

```
module add(a,b,cin,sum,cout);
    input a, b, cin;
    output sum, cout;
...
endmodule
```

```
module add_two_bit(a,b,cin,sum,cout);
    input [1:0] a, b, cin;
    output [1:0] sum, cout;
...
endmodule
```

Using modules: two methods

1. Order of inputs/outputs is important

```
add my_add(a,b,cin,sum,cout);
```

2. Order is not important -- **USE THIS ONE**

```
add my_add
(
   .a(a),
   .b(b),
   .cin(cin),
   .sum(sum),
   .cout(cout)
);
```

Lexical

- Comments // and /* */
- Case sensitive
- Identifiers must start with: A-Z,a-z,_
- May contain: A-Z,a-z,0-9,_,$

```
// this is a comment!
module add(a,b,cin,sum,cout);
    input a, b, cin;
    output sum, cout;
```

Variables

- Wires – `wire a_wire, another_wire;`
  - Cannot hold state
- Registers – `reg a_register;`
  - Can hold state – but might not

```
module add(a,b,cin,sum,cout);
    input a, b, cin;   // defaults to wire
    output sum, cout; // defaults to reg
```

- Multiple bits
  - `wire[7:0] a_8bit_wire;`
- Other variables – Only for use in test benches
  - `integer` – signed 32-bit variable
  - `time` – unsigned 64-bit variable
  - `real` – double-precision floating point

Literals

0	1	Z, z	X, x
false, zero, low	true, one, high	high-impedance	unknown, invalid

- Written in the format `<width>'<base><constant>`
- Examples all represent decimal 10:
  - `8'b00001010` – 8 bit
  - `32'd10` – 32 bit decimal
  - `24'o12` – 24 bit octal
  - `16'hA` – 16 bit hexadecimal
  - '0 – initialize to all zeros, regardless of width

Operators

- +, -, *, %, /, >, <, <=, >=, ==, !=, &&, ||
  - Same as C operators
  - + – an efficient adder
  - / – a divider big/slow/did you really want one?
- <<, >> – Shifting operators
- ~, &, |, ^ – Bit-wise NOT, AND, OR, XOR
  - Can put these together for other gates (e.g. | – NOR)
- !==, === – Equivalence (different from !=)
- ? : – Conditional: <condition>? <if-true>:<if-false>
- {a,b,c} – concatenation
- {n{m}} – concatenate n copies of m

```
assign sum = a ^ b ^ cin;
assign cout = ((a ^ b) & cin) | a & b;
```

Setting Values – assign statements

- Descriptions of combinational logic
- `assign <wire> = <expression>;`
- Left hand side must be a wire, right hand side can be anything
- Can use the `? :` operator, but not other conditionals
  - They can be nested

```
module add(a,b,cin,sum,cout);
    input a, b, cin;
    output sum, cout;

    assign sum = a ^ b ^ cin;
    assign cout = ((a ^ b) & cin) | a & b;
endmodule
```

Setting values with always blocks

Simple Adder

```
module add(a,b,cin,sum,cout);
    input a, b, cin;
    output sum, cout;

    reg sum, cout;

    always @*
    begin
        sum = a ^ b ^ cin;
        cout = 0;
        if (((a^b) & cin) | (a & b))
            cout = 1;
    end
endmodule
```

Setting values – always block examples

Blocking	Non-blocking

Blocking

```
always @*
begin
    x = a + b;
    y = x + 8'd5;
end
```

Non-blocking

```
always @(posedge clock)
begin
    x <= next_x;
    y <= next_y;
end
```

- Update "wires" in order
  - Combinational logic
- Update registers
  - Sequential logic

Blocking and non-blocking operators

Blocking

- Update “wires” in order
  - Combinational logic
- `always @*`

  = operator
- Each line is processed in order (earlier lines block later lines)
- counter-intuitive: declared as “reg”, but behaves as “wire”

Non-Blocking

- Update registers
  - Sequential logic
- `always @(posedge clock)`

  <= operator
- All assignments occur at (approximately) the same time
- Use delays! (`SD or #1)

Blocking and Non-Blocking Examples

```
Bad Example
always @(posedge clock)
begin
    x <= y;
    z = x;
end
```

- What does this do?
- Note that z is updated before x!
- Mixing assignments has non-intuitive results.

Bad example

- Always assign all variables on all paths
- Otherwise you could get state in a block that you thought was combinational

Bad Example

```
always @*
begin
    if(cond)              //Synth: IMPLICIT LATCH WARNING
        next_x = y;
end
```

Bad example fixed

- Always assign all variables on **all paths**
- Otherwise you could get state in a block that you thought was combinational

Fixed Example

```
always @*
begin
    next_x = x; //default value
    if(cond)
        next_x = y;
end
```

Bad example fixed another way

Fixed Example

```
always @*
begin
    if(cond)
        next_x = y;
     else
        next_x = x;
end
```

Verilog References

- EECS 470 Verilog overview
  - http://www.eecs.umich.edu/courses/eecs470/tools/verilogOverview.pdf
  - **READ THIS**
- Quick reference
  - http://www.stanford.edu/class/ee108b/labs/VerilogQuickRef.pdf
- Examples and tutorials
  - http://www.asic-world.com/verilog
- Complete Synopsys reference
  - `sold` command

Verilog Tutorial

- Download InLab 1 from:
  - http://www.eecs.umich.edu/eecs/courses/eecs470/assignments.html

Basic Stuff We Assume You Know

- You should be familiar with GNU/Linux commands
  - i.e. bash, cd, mv, tar, make
- You should also be familiar with text editors
  - (Yay! Vim, Boo: Emacs)
  - But seriously: Something with good syntax highlighting that you are **comfortable with**
- You should be familiar with version control
  - This class traditionally uses svn
  - But you may use others if you like

Tutorial + InLab1

- The “lecture” portion is over
- Do the Tutorial
- When you finish, begin Lab 1
  - Labs are normally designed to finish **DURING LAB**
  - Since this is week 1…
  - Special Due Date : TODAY before midnight

Begin extra slides

Flow Control

- Can only be used inside procedural blocks.
- Syntax similar to C
  - Statement (something that ends with ;)
  - `begin … end`
- Flow control you should use:
  - `if`
  - `if/else`
  - `case`
  - `casez`

Flow Control you shouldn't use

- `while`
- `for`
- `repeat`
- `forever`
- Except in:
  - Test benches
  - Specific cases that we'll list later
  - If you look at something and immediately think of a loop this is the wrong case

Flow Control Examples

If/Else

```
always @*
begin
    if (muxy == 0)
        y = a;
    else
        y = b;
end
```

Flow Control Examples

casez

```
always @*
begin
    casez(alu_op)
        3'b000: r = a + b;
        3'b001: r = a – b;
        3'b010: r = a * b;
        …
        3'b1??: r = a ^ b;
    endcase
end
```

Initial Blocks

- Executed once at the beginning of the simulation
- Should only be used in test benches
- If you want to reset values outside of a testbench use a reset signal

Example

```
initial
begin
    @(negedge clock);
    reset = 1;
    in0 = 0; in1 = 0;
    @(negedge clock); reset = 0;
    @(negedge clock); in0 = 1;
    …
end
```

Tasks – Debugging help

- Tasks are procedures
  - Not synthesizable
  - Useful in test benches for debugging/modular code
- We'll discuss writing tasks later

Built-in Tasks

- `$monitor("format", signal,...);`
- `$display("format", signal,...);`
- `$write("format", signal,...);`
- `$strobe("format", signal,...);`
- `fd = $fopen("filename");`
- `$fclose(fd);`
- All the above can take a fd as their first parameter as well.
- `$time` – Current simulation time
- `$random(seed);` – Returns a 32 bit random value
- `$finish;` – End the simulation now
- `$stop;` – Halt the simulation now

`$monitor(), $display(), $strobe()`

- `$monitor()`
  - Best used in test benches so you get a print statement every time something happens.
  - Useful for small things, but not so much with big projects
    - Better to use waveform viewer, too
- `$display()`
  - Display formatted text at this instant, very useful when trying to debug something
  - Don’t need to print just variables, you could have higher level things in there as well (e.g. Entering state X)
- `$strobe()`
  - Similar to `$display()`
  - Waits until end of tick to print though, after assignments have been made
  - Good for always `@(posedge)` blocks

The Actual Tests

- Hand-written/directed
- Exhaustive
- Random