At HotChips’23, Berkeley presented two posters on the EE194/290 class chips: OSCIBear and BearlyML.

F. Guo

OSCIBear: 32b RISC-V + BLE + AES + Power

Y. Chi

BearlyML'22: 5 RISC-V cores: 4 Rocket with custom sparse matrix acc, NoC, PLL, L2
Review

• Electronic systems can be realized by using embedded processors, FPGAs and ASICs

• Verilog is a common design entry for synthesis

• Design flows for FPGAs and ASICs differ
  • But also have similarities
  • LA labs: ASIC
  • LB labs: FPGA
Behavioral Verilog
module foo (
    input in1, in2,
    output out
);

assign out = in1 & in2;
endmodule

Short-hand for explicit instantiation of bit-wise “and” gate (in this case).

The assignment continuously happens, therefore any change on the RHS is reflected in out immediately (except for the small delay associated with the implementation of the practical &).

Not like an assignment in C that takes place when the program counter gets to that place in the program.
Example - Ripple Adder

module full_adder (  
  input a, b, ci,  
  output r, co  
);
                                 
  assign r = a ^ b ^ ci;  
  assign co = a&ci | a&b | b&cin;  

endmodule

module adder(  
  input [3:0] A,  
  input [3:0] B,  
  output [4:0] R  
);

  wire c1, c2, c3;  
  full_adder  
  add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),  
  add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),  
  add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),  
  add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );

endmodule
## Verilog Operators

<table>
<thead>
<tr>
<th>Verilog Operator</th>
<th>Name</th>
<th>Functional Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>bit-select or part-select</td>
<td></td>
</tr>
<tr>
<td>( )</td>
<td>parenthesis</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation</td>
<td>Logical</td>
</tr>
<tr>
<td>~</td>
<td>negation</td>
<td>Bit-wise</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND</td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>^</td>
<td>reduction NOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^ or ^~</td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>+</td>
<td>unary (sign) plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>unary (sign) minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>{}</td>
<td>concatenation</td>
<td>Concatenation</td>
</tr>
<tr>
<td>{()}</td>
<td>replication</td>
<td>Replication</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>+</td>
<td>binary plus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>binary minus</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Shift</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Logic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic ‘0’ or false</td>
</tr>
<tr>
<td>1</td>
<td>Logic ‘1’ or true</td>
</tr>
<tr>
<td>x</td>
<td>Don’t care or unknown value</td>
</tr>
<tr>
<td>z</td>
<td>High impedance state</td>
</tr>
</tbody>
</table>
Verilog Numbers

**Constants:**

14 ordinary decimal number

-14 2’s complement representation

12’b0000_0100_0110 binary number (“_” is ignored)

12’h046 hexadecimal number with 12 bits

**Signal Values:**

By default, values are unsigned

e.g., \( C[4:0] = A[3:0] + B[3:0]; \)

if \( A = 0110 \) (6) and \( B = 1010 \) (-6)

\( C = 10000 \) (not 00000)

i.e., B is zero-padded, not sign-extended

wire signed [31:0] x;

Declares a signed (2’s complement) signal array.
Verilog Assignment Types
Continuous Assignment Examples

```verilog
assign R = X | (Y & ~Z);
assign r = &X;
assign R = (a == 1'b0) ? X : Y;
assign P = 8'hff;
assign P = X * Y;
assign P[7:0] = {4{X[3]}, X[3:0]};
assign {cout, R} = X + Y + cin;
assign Y = A << 2;
assign Y = {A[1], A[0], 1'b0, 1'b0};
```

use of bit-wise Boolean operators

conditional operator

example constants

arithmetic operators (use with care!)

(ex: sign-extension)

bit field concatenation

bit shift operator

equivalent bit shift
Non-Continuous Assignments

A bit unusual from a hardware specification point of view. Shows off Verilog roots as a simulation language.

“always” block example:

```verilog
module and_or_gate (input in1, in2, in3;
output reg out;)
endmodule

always @(in1 or in2 or in3) begin
  out = (in1 & in2) | in3;
end

Isn’t this just: assign out = (in1 & in2) | in3;?

Why bother?
```

“reg” type declaration. Not really a register in this case. Just a Verilog idiosyncrasy.

“sensitivity” list, triggers the action in the body.

brackets multiple statements (not necessary in this example).
Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

**case statement example:**

```verilog
module mux4 (
    input  in0, in1, in2, in3;
    input  [1:0] select;
    output out;
);
    reg out;
    always @ (in0 in1 in2 in3 select)
        case (select)
            2'b00: out = in0;
            2'b01: out = in1;
            2'b10: out = in2;
            2'b11: out = in3;
        endcase
endmodule // mux4
```

Couldn’t we just do this with nested “if”s?

Well yes and no!
Always Blocks

Nested if-else example:

```verilog
module mux4 (
    input in0,in1,in2,in3;
    input [1:0] select;
    output reg  out;
);

always @ (in0 in1 in2 in3 select)
    if (select == 2’b00) out=in0;
    else if (select == 2’b01) out=in1;
    else if (select == 2’b10) out=in2;
    else out=in3;
endmodule  // mux4
```

Nested if structure leads to “priority logic” structure, with different delays for different inputs (in3 to out delay > than in0 to out delay). Case version treats all inputs the same.
Verilog and SystemVerilog Data Types

• Verilog nets
  
  `reg` – can ‘hold’ a value. Often, but not always represents registers.
  
  LHS of signals assigned within ‘always’ statement.

  `wire` – cannot hold a value, has to be driven. Represents connections.
  
  LHS of signals assigned outside ‘always’ statements (continuous assignment).

• SystemVerilog
  
  `logic` – generic data type
  
  `net` – can have multiple drivers.
module FullAdder(
    input a, b, ci;
    output r, co;
);

assign r = a ^ b ^ ci;
assign co = a&ci + a&b + b&cin;

endmodule

module Adder(A, B, R);
    input [3:0] A;
    input [3:0] B;
    output [4:0] R;

    wire c1, c2, c3;

    FullAdder
        add0(.a(A[0]), .b(B[0]), .ci(1'b0), .co(c1), .r(R[0]) ),
        add1(.a(A[1]), .b(B[1]), .ci(c1), .co(c2), .r(R[1]) ),
        add2(.a(A[2]), .b(B[2]), .ci(c2), .co(c3), .r(R[2]) ),
        add3(.a(A[3]), .b(B[3]), .ci(c3), .co(R[4]), .r(R[3]) );

endmodule
Example - Ripple Adder Generator

Parameters give us a way to generalize our designs. A module becomes a “generator” for different variations. Enables design/module reuse. Can simplify testing.

module Adder
#(parameter N = 4)
    input [N-1:0] A;
    input [N-1:0] B;
    output [N:0] R;
    wire [N:0] C;

    genvar i;
    generate
        for (i=0; i<N; i=i+1) begin
            FullAdder add(.a(A[i], .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
        end
    endgenerate

    assign C[0] = 1'b0;
    assign R[N] = C[N];
endmodule

Declare a parameter with default value.

Note: this is not a port. Acts like a “synthesis-time” constant.

Replace all occurrences of “4” with “N”.

variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

For-loop creates instances (with unique names)

Adder adder4 ( ... );
Adder #( .N(64) )
adder64 ( ... );

Overwrite parameter N at instantiation.
More on Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```verilog
// Gray-code to binary-code converter
module gray2bin1
    #(parameter SIZE = 8)
    (output [SIZE-1:0] bin;
     input  [SIZE-1:0] gray
    );

    genvar i;
    generate for (i=0; i<SIZE; i=i+1)
        begin
            assign bin[i] = ^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

**generate if-else-if** based on an expression that is deterministic at the time the design is synthesized.

**generate case** : selecting case expression must be deterministic at the time the design is synthesized.

---

Variable exists only in the specification - not in the final circuit.

Keywords that denotes synthesis-time operations

For-loop creates instances of assignments

Loop must have constant bounds
Defining Processor ALU in 5 mins

- Modularity is essential to the success of large designs
- High-level primitives enable direct synthesis of behavioral descriptions (functions such as additions, subtractions, shifts (<< and >>), etc.

**Example: A 32-bit ALU**

**Function Table**

<table>
<thead>
<tr>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A + 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A - B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A - 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>A * B</td>
</tr>
</tbody>
</table>

### Diagram:

- **Inputs:** A[31:0], B[31:0]
- **Outputs:** R[31:0]
- **Functions:**
  - Addition (+)
  - Subtraction (-)
  - Multiplication (*)
  - 32-bit value (32'd1)

- **Logic Gates:**
  - XNOR gates
  - XOR gates
  - Or gates

- **Truth Table:**

<table>
<thead>
<tr>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A + B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A + 1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A - B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A - 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>A * B</td>
</tr>
</tbody>
</table>
Module Definitions

2-to-1 MUX
module mux32two(
    input [31:0] i0, i1;
    input sel;
    output [31:0] out
);

assign out = sel ? i1 : i0;
endmodule

3-to-1 MUX
module mux32three(
    input [31:0] i0, i1, i2;
    input [1:0] sel;
    output reg [31:0] out;
);
always @ (i0 or i1 or i2 or sel)
begin
    case (sel)
        2'b00: out = i0;
        2'b01: out = i1;
        2'b10: out = i2;
        default: out = 32'bx;
    endcase
end
endmodule

32-bit Adder
module add32(
    input [31:0] i0, i1;
    output [31:0] sum;
);

assign sum = i0 + i1;
endmodule

32-bit Subtractor
module sub32(
    input [31:0] i0, i1;
    output [31:0] diff
);

assign diff = i0 - i1;
endmodule

16-bit Multiplier
module mul16(
    input [15:0] i0, i1;
    output [31:0] prod;
);

// this is a magnitude multiplier
// signed arithmetic later
assign prod = i0 * i1;
endmodule
module alu(
    input [31:0] a, b;
    input [2:0] f;
    output [31:0] r);

wire [31:0] addmux_out, submux_out;
wire [31:0] add_out, sub_out, mul_out;

mux32two   adder_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(addmux_out));
mux32two   sub_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(submux_out));
add32      our_adder(.i0(a), .i1(addmux_out), .sum(add_out));
sub32      our_subtractor(.i0(a), .i1(submux_out), .diff(sub_out));
mul16      our_multiplier(.i0(a[15:0]), .i1(b[15:0]), .prod(mul_out));
mux32three output_mux(.i0(add_out), .i1(sub_out), .i2(mul_out), .sel(f[2:1]), .out(r));
endmodule

Top-Level ALU Declaration

• Given submodules:

module mux32two(i0,i1,sel,out);
module mux32three(i0,i1,i2,sel,out);
module add32(i0,i1,sum);
module sub32(i0,i1,diff);
module mul16(i0,i1,prod);

• Declaration of the ALU Module:

module alu(  
    input [31:0] a, b;
    input [2:0] f;
    output [31:0] r);

wire [31:0] addmux_out, submux_out;
wire [31:0] add_out, sub_out, mul_out;

mux32two   adder_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(addmux_out));
mux32two   sub_mux(.io(b), .i1(32'd1), .sel(f[0]), .out(submux_out));
add32      our_adder(.i0(a), .i1(addmux_out), .sum(add_out));
sub32      our_subtractor(.i0(a), .i1(submux_out), .diff(sub_out));
mul16      our_multiplier(.i0(a[15:0]), .i1(b[15:0]), .prod(mul_out));
mux32three output_mux(.i0(add_out), .i1(sub_out), .i2(mul_out), .sel(f[2:1]), .out(r));
endmodule
Top-Level ALU Declaration, take 2

• No Hierarchy:

• Declaration of the ALU Module:

```
module alu(a
  input [31:0] a, b;
  input [2:0] f;
  output [31:0] r
);

always @ (a or b or f)
  case (f)
     3'b000: r = a + b;
     3'b001: r = a + 1'b1;
     3'b010: r = a – b;
     3'b011: r = a – 1'b1;
     3'b100: r = a * b;
    default: r = 32'bx;
  endcase
endmodule
```
Administrivia

• Bora’s office hour (this week) on Thursday at 1pm
  • Regular office hour on Mondays at 10am

• Lab 1 finished
  • If you couldn’t attend the lab, please get checked off in office hours this week

• Lab 2 is this week
  • Cycle wraps up on Monday
  • Lab 3 starts on Tuesday – be prepared!

• Homework 1 due this Friday
  • Homework 2 posted today
Sequential Logic, Take 2
Latches and Flip-Flops

• Flip-flop is edge-triggered, latch is level-sensitive

• D Flip-flop
  • Rising edge
    - Signifies 'edge triggered'

• D Latch
  • Transparent
    - HIGH
      - Level sensitive if there is no 'edge'
  • Transparent
    - LOW
Timing

- Combinational logic timing

In CMOS, propagation delay depends on:
- Gate type, size (output resistance)
- Capacitive loading
- Input slope
Timing

• Flip-flop timing
  (latch timing will be covered later)

• Setup and hold times
  • Data cannot change in the interval of setup time before the clock edge to hold time after the clock edge
Register

- 4-bit register

- Accumulator
Sequential Logic in Verilog
Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

```verilog
module dff(q
    input d, clk, set, rst;
    output reg q;
);
    always @(posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
endmodule
```

Keywords: 
- `always @ (posedge clk)` is key to flip-flop inference.
- This gives priority to reset over set and set over d.
- On FPGAs, maps to native flip-flop.

Unlike logic gates, there are no primitive flip-flops in Verilog. Although, it is possible to instantiate FPGA or standard-cell specific flip-flops.
The Sequential *always* Block

**Combinational**

```verilog
code
module comb(input a, b, sel, output reg out);
    always @(*) begin
        if (sel) out = b;
        else out = a;
    end
endmodule
```

**Sequential**

```verilog
code
module seq(input a, b, sel, clk, output reg out);
    always @(posedge clk) begin
        if (sel) out <= b;
        else out <= a;
    end
endmodule
```
Latches vs. Flip-Flops

Flip-Flop

module flipflop
(
    input clk,
    input d,
    output reg q
);

    always @(posedge clk)
    begin
        q <= d;
    end
endmodule

Latch

module latch
(
    input clk,
    input d,
    output reg q
);

    always @(clk or d)
    begin
        if ( clk )
            q <= d;
    end
endmodule
Importance of the Sensitivity List

- The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered)

D-Register with **synchronous** clear

```verilog
module dff_sync_clear(  
  input d, clearb, clock,  
  output reg q);  
always @(posedge clock)  
begin  
  if (!clearb) q <= 1'b0;  
  else q <= d;  
end  
endmodule
```

D-Register with **asynchronous** clear

```verilog
module dff_async_clear(  
  input d, clearb, clock,  
  output reg q);  
always @(negedge clearb or posedge clock)  
begin  
  if (!clearb) q <= 1'b0;  
  else q <= d;  
end  
endmodule
```

Note: The following is incorrect syntax: `always @(clear or negedge clock)`

If one signal in the sensitivity list uses `posedge/negedge`, then all signals must.

- **Assign any signal or variable from only one `always` block.**
- **Be wary of race conditions:** `always` blocks with same trigger execute concurrently…
Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.

  - **Blocking assignment (=):** evaluation and assignment are immediate

```verilog
always @(*) begin
    x = a | b;     // 1. evaluate a|b, assign result to x
    y = a ^ b ^ c; // 2. evaluate a^b^c, assign result to y
    z = b & ~c;    // 3. evaluate b&(~c), assign result to z
end
```

  - **Nonblocking assignment (<=):** all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active `always` blocks)

```verilog
always @(*) begin
    x <= a | b;     // 1. evaluate a|b, but defer assignment to x
    y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
    z <= b & ~c;    // 3. evaluate b&(~c), but defer assignment to z
    // 4. end of time step: assign new values to x, y and z
end
```

Sometimes, as above, both produce the same result. Sometimes, not!
Assignment Styles for Sequential Logic

What we want:
Register-based digital delay line
(a.k.a. shift-register)

Will non-blocking and blocking assignments both produce the desired result?

```
module nonblocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule

module blocking(
  input in, clk,
  output reg out
);
  reg q1, q2;
  always @(posedge clk) begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule
```
Use Nonblocking for Sequential Logic

always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;  // uses old q1
    out <= q2;  // uses old q2
end

always @(posedge clk) begin
    q1 = in;
    q2 = q1;   // uses new q1
    out = q2;  // uses new q2
end

("old" means value before clock edge, "new" means the value after most recent assignment)

“At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2.”

“At each rising clock edge, q1 = in. After that, q2 = q1. After that, out = q2. Therefore out = in.”

- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignments for sequential always blocks
Example: A Simple Counter

```verilog
// 4-bit counter with enable and synchronous clear
module counter(input clk, enb, clr,
               output reg [3:0] count);
    always @(posedge clk) begin
        count <= clr ? 4’b0 : (enb ? count+1 : count);
    end
endmodule
```
module ParToSer(
    input [3:0] X;
    input ld, clk;
    output reg out;
);

reg [3:0] Q;
wire [3:0] NS;

assign NS = (ld) ? X : {Q[0], Q[3:1]};

always @ (posedge clk)
    Q <= NS;

assign out = Q[0];
endmodule

Example - Parallel to Serial Converter

Specifies the muxing with “rotation”

forces Q register (flip-flops) to be rewritten every cycle

connect output
Simplified Verilog Guidelines

• Combinational logic:
  • Continuous Assignment:
    ```verilog
    assign a = b & c;
    ```
  • Always block with `@(*)`
    ```verilog
    always @(*) begin
      a = b & c;  // blocking statement
    end
    ```

• Sequential logic:
  • Always block with `@posedge clk`
    ```verilog
    always @posedge clk begin
      a <= b & c;  // nonblocking statement
    end
    ```
Verilog in EECS 151/251A

• We use behavioral modeling at the bottom of the hierarchy

• Use instantiation to 1) build hierarchy and, 2) map to FPGA and ASIC resources not supported by synthesis.

• Favor continuous assign and avoid always blocks unless:
  • No other alternative: ex: state elements, case
  • Helps readability and clarity of code: ex: large nested if else

• Use named ports.

• Refer to the class style manual!

• Verilog is a big language. This is only an introduction.
  • Harris & Harris book chapter 4 is a good source.
  • Be careful of what you read on the web. Many bad examples out there.
  • We will be introducing more useful constructs throughout the semester. Stay tuned!
Verilog vs. SystemVerilog

• **always** statements in Verilog can be used to infer flip-flops, latches or logic
  • Depends on the sensitivity list and the statement
  • Easy to create confusion

• System Verilog adds disambiguation:
  • **always_ff** for flip-flops
  • **always_latch** for latches
  • **always_comb** for combinational logic
Summary

• Introduced behavioral Verilog
  • Examples of combinatorial logic
  • Continuous and non-continuous assignments

• Sequential logic uses flip-flops and (sometimes) latches

• Flip-flops and latches are inferred in Verilog
  • Always blocks

• Practice is the best way to learn a new language…

• Blocking and non-blocking assignments