IC chip from Hewlett Packard 34C Calculator, 1979-83. Some of the calculator’s revolutionary features were designed by UC Berkeley computer scientist William M. Kahan.
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
Verilog Introduction

- A **module** definition describes a component in a circuit.
- Two ways to describe module contents:
  - **Structural Verilog**
    - List of sub-components and how they are connected.
    - Just like schematics, but using text.
    - Tedious to write, hard to decode.
    - You get precise control over circuit details.
    - May be necessary to map to special resources of the FPGA/ASIC.
  - **Behavioral Verilog**
    - Describe what a component does, not how it does it.
    - Synthesized into a circuit that has this behavior.
    - Result is only as good as the tools.
  - Build up a hierarchy of modules. Top-level module is your entire design (or the environment to test your design).
Logic Synthesis

• Verilog and VHDL started out as simulation languages but soon programs were written to automatically convert Verilog code into low-level circuit descriptions (netlists)

• Synthesis converts Verilog (or other HDL) descriptions to a logic mapping by using technology-specific primitives:
  • For FPGA: LUTs, flip-flops, and BRAMs
  • For ASICs: standard cells and memory macros

• In addition, synthesis algorithms optimize the implementation for delay and power
Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.

Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure. Testbench is typically the top-level module.
Structural Verilog
module xor_gate (out, a, b);
  input     a, b;
  output    out;
  wire      aBar, bBar, t1, t2;
  not invA (aBar, a);
  not invB (bBar, b);
  and and1 (t1, a, bBar);
  and and2 (t2, b, aBar);
  or  or1 (out, t1, t2);
endmodule
/* 2-input multiplexer in gates */
module mux2 (in0, in1, select, out);
    input in0, in1, select;
    output out;
    wire s0, w0, w1;
    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or  (out, w0, w1);
endmodule // mux2
module mux4 (in0, in1, in2, in3, select, out);
input in0, in1, in2, in3;
input [1:0] select;
output out;
wire w0, w1;

mux2
   m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
   m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
   m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4

// 2-input multiplexor in gates
module mux2 (in0, in1, select, out);
   input in0, in1, select;
   output out;
   wire s0, w0, w1;
   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);
endmodule // mux2

---

module mux4 (in0, in1, in2, in3, select, out);
input in0, in1, in2, in3;
input [1:0] select;
output out;
wire w0, w1;

mux2
   m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
   m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
   m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4

/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
   input in0, in1, select;
   output out;
   wire s0, w0, w1;
   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);
endmodule // mux2

---

Signal array. Declares select[1], select[0]

Named ports. Highly recommended.
Aside: Netlists

- **Netlist** is a description of the connectivity of an electronic circuit
  - Netlist consists of a list of components in a circuit and a list of the nodes they are connected to. A wire (net) connects two or more components

- Structural Verilog is one form of describing a netlist

- Most common netlist format is EDIF (Electronic Design Exchange Format)
  - Established in 1985, still in use
Administrivia

• Lab 2 this week
  • Lab 3 starts next week

• Homework 1 due this Friday.

• Homework 2 out this Thursday.
  • More involved!
Behavioral Verilog
module foo (out, in1, in2);
input in1, in2;
output out;
assign out = in1 & in2;
endmodule

“continuous assignment”
Connects out to be the logical “and” of in1 and in2.

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.
module FullAdder(a, b, ci, r, co);
  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
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>reduction OR</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction NAND</td>
<td>Reduction</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>reduction NOR</td>
</tr>
<tr>
<td>^</td>
<td>reduction XOR</td>
<td>Reduction</td>
</tr>
<tr>
<td>^~ or ^~</td>
<td>reduction XNOR</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>
<tr>
<td>Values</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logic</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>Logic ‘0’ or false</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Logic ‘1’ or true</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>Don’t care or unknown value</td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>High impedance state</td>
<td></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

assign \( R = X \mid (Y \& \neg Z) \);

assign \( r = \& X \);

assign \( R = (a == 1'b0) ? X : Y \);

assign \( P = 8'hff \);

assign \( P = X \times Y \);

assign \( P[7:0] = \{4\{X[3]\}, X[3:0]\} \);

assign \( \{cout, R\} = X + Y + cin \);

assign \( Y = A \ll 2 \);

assign \( Y = \{A[1], A[0], 1'b0, 1'b0\} \);
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 (out, in1, in2, in3);
    input in1, in2, in3;
    output out;
    reg out;

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

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

Why bother?
Always Blocks

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

case statement example:

```verilog
module mux4 (in0, in1, in2, in3, select, out);
    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:

module mux4 (in0, in1, in2, in3, select, out);
    input in0, in1, in2, in3;
    input [1:0] select;
    output      out;
    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(a, b, ci, r, co);
    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.

```verilog
module Adder(A, B, R);
  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:bit
      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
```

Declarative 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)

```verilog
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 (bin, gray);
    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:bit
            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.
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 F1 F0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>A + B</td>
</tr>
<tr>
<td>0 0 1</td>
<td>A + 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>A - B</td>
</tr>
<tr>
<td>0 1 1</td>
<td>A - 1</td>
</tr>
<tr>
<td>1 0 X</td>
<td>A * B</td>
</tr>
</tbody>
</table>

A[31:0] B[31:0]

R[31:0]
Module Definitions

2-to-1 MUX

```verilog
module mux32two(i0, i1, sel, out);
  input [31:0] i0, i1;
  input sel;
  output [31:0] out;

  assign out = sel ? i1 : i0;
endmodule
```

3-to-1 MUX

```verilog
module mux32three(i0, i1, i2, sel, out);
  input [31:0] i0, i1, i2;
  input [1:0] sel;
  output [31:0] out;
  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

```verilog
module add32(i0, i1, sum);
  input [31:0] i0, i1;
  output [31:0] sum;

  assign sum = i0 + i1;
endmodule
```

32-bit Subtractor

```verilog
module sub32(i0, i1, diff);
  input [31:0] i0, i1;
  output [31:0] diff;

  assign diff = i0 - i1;
endmodule
```

16-bit Multiplier

```verilog
module mul16(i0, i1, prod);
  input [15:0] i0, i1;
  output [31:0] prod;

  assign prod = i0 * i1;
  // this is a magnitude multiplier
  // signed arithmetic later
endmodule
```
module alu(a, b, f, r);
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:

```verilog
module alu(a, b, f, r);
  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
```

Will this synthesize into 2 adders and 2 subtractors or 1 of each?
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
  - Transparent
  - LOW
  - Level sensitive if there is no ‘edge’
Timing

• Combinational logic timing

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

A is arriving late
(is in the critical path)

B is arriving late
(is in the critical path)

HL and LH transition differ
$t_{A\text{-Out}}$ and $t_{B\text{-Out}}$ differ
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
Summary

• Verilog is the most-commonly used HDL

• We have seen combinatorial constructs
  • Assign statement
  • Always blocks
  • Sequential – next week

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