Review: Elements of the Design Process

- Divide and Conquer (e.g., ALU)
  - Formulate a solution in terms of simpler components.
  - Design each of the components (subproblems)

- Generate and Test (e.g., ALU)
  - Given a collection of building blocks, look for ways of putting them together that meets requirement

- Successive Refinement (e.g., multiplier, divider)
  - Solve "most" of the problem (i.e., ignore some constraints or special cases), examine and correct shortcomings.

- Formulate High-Level Alternatives (e.g., shifter)
  - Articulate many strategies to "keep in mind" while pursuing any one approach.

- Work on the Things you Know How to Do
  - The unknown will become “obvious” as you make progress.

Review: ALU Design

- Bit-slice plus extra on the two ends
- Overflow means number too large for the representation
- Carry-look ahead and other adder tricks

Review: Carry Look Ahead (Design trick: peek)

\[
\begin{align*}
C_1 &= G_0 + C_0 \cdot P_0 \\
C_2 &= G_1 + G_0 \cdot P_1 + C_0 \cdot P_0 \cdot P_1 \\
C_3 &= G_2 + G_1 \cdot P_2 + G_0 \cdot P_1 + P_0 \cdot P_1 \cdot P_2 \\
C_4 &= \cdots
\end{align*}
\]

\[
\begin{align*}
A & \quad B & \quad C-out \\
0 & \quad 0 & \quad 0 & \quad \text{“kill”} \\
0 & \quad 1 & \quad C-in & \quad \text{“propagate”} \\
1 & \quad 0 & \quad C-in & \quad \text{“propagate”} \\
1 & \quad 1 & \quad \text{“generate”} \\
\end{align*}
\]

\[
\begin{align*}
G &= A \text{ and } B \\
P &= A \text{ xor } B
\end{align*}
\]
Review: Design Trick: Guess (or “Precompute”)

\[ CP(2n) = 2 \cdot CP(n) \]

\[ CP(2n) = CP(n) + CP(mux) \]

Carry-select adder

Why should you keep a design notebook?

- Keep track of the design decisions and the reasons behind them
  - Otherwise, it will be hard to debug and/or refine the design
  - Write it down so that can remember in long project: 2 weeks -> 2 yrs
  - Others can review notebook to see what happened

- Record insights you have on certain aspect of the design as they come up

- Record of the different design & debug experiments
  - Memory can fail when very tired

- Industry practice: learn from others mistakes

Why do we keep it on-line?

- You need to force yourself to take notes
  - Open a window and leave an editor running while you work
    1) Acts as reminder to take notes
    2) Makes it easy to take notes
  - 1) + 2) => will actually do it

- Take advantage of the window system’s “cut and paste” features

- It is much easier to read your typing than your writing

- Also, paper log books have problems
  - Limited capacity => end up with many books
  - May not have right book with you at time vs. networked screens
  - Can use computer to search files/index files to find what looking for

How should you do it?

- Keep it simple
  - DON’T make it so elaborate that you won’t use (fonts, layout, …)

- Separate the entries by dates
  - type “date” command in another window and cut&paste

- Start day with problems going to work on today

- Record output of simulation into log with cut&paste; add date
  - May help sort out which version of simulation did what

- Record key email with cut&paste

- Record of what works & doesn’t helps team decide what went wrong after you left

- Index: write a one-line summary of what you did at end of each day
On-line Notebook Example

- Refer to the handout “Example of On-Line Log Book” on cs 152 handouts page

1st page of On-line notebook (Index + Wed. 9/6/95)

* Index ===============================================================

Wed Sep 6 00:47:28 PDT 1995 - Created the 32-bit comparator component
Thu Sep 7 14:02:21 PDT 1995 - Tested the comparator
Mon Sep 11 12:01:45 PDT 1995 - Investigated bug found by Bart in comp32 and fixed it

Goal: Layout the schematic for a 32-bit comparator

I've laid out the schematics and made a symbol for the comparator.
I named it comp32. The files are

* ~/wv/proj1/sch/comp32.sch
* ~/wv/proj1/sch/comp32.sym

Wed Sep 6 02:29:22 PDT 1995

- ====================================================================

2nd page of On-line notebook (Thursday 9/7/95)

Goal: Test the comparator component

I've written a command file to test comp32. I've placed it
in *~/wv/proj1/diagnostics/comp32.cmd.*

I ran the command file in viewsim and it looks like the comparator
is working fine. I saved the output into a log file called

*~/wv/proj1/diagnostics/comp32.log*

Notified the rest of the group that the comparator
is done.

Thu Sep 6 16:15:32 PDT 1995

- ====================================================================

3rd page of On-line notebook (Monday 9/11/95)

Goal: Investigate bug discovered in comp32 and hopefully fix it

Bart found a bug in my comparator component. He left the following
e-mail.

------------------- From bart@simpsons.residence Sun Sep 10 01:47:02 1995
Received: by wayne.manor (NX5.67e/NX3.0S)
id AA00334; Sun, 10 Sep 95 01:47:01 -0800
Date: Wed, 10 Sep 95 01:47:01 -0800
From: Bart Simpson <bart@simpsons.residence>
To: bruce@wayne.manor, old_man@gokuraku, hojo@sanctuary
Subject: [cs152] bug in comp32
Status: R

Hey Bruce,

I think there's a bug in your comparator.
The comparator seems to think that ffffffff and ffffffff are equal.
Can you take a look at this?

Bart

-------------------
I verified the bug. Here’s a viewsim of the bug as it appeared.

(equal should be 0 instead of 1)

------------------ SIM>stepsize 10ns
SIM>v a_in A[31:0]
SIM>v b_in B[31:0]
SIM>w a_in b_in equal
SIM>a a_inffffffff
SIM>b b_in fffffff7
SIM>er 1
Simulation stopped at 10.0ns. -------------------

Ah. I’ve discovered the bug. I mislabeled the 4th net in the comp32 schematic.

I corrected the mistake and re-checked all the other labels, just in case.

I re-ran the old diagnostic test file and tested it against the bug Bart found. It seems to be working fine. Hopefully there aren’t any more bugs:

---

On second inspection of the whole layout, I think I can remove one level of gates in the design and make it go faster.

But who cares! The comparator is not in the critical path right now. The delay through the ALU is dominating the critical path. So unless the ALU gets a lot faster, we can live with a less than optimal comparator.

I e-mailed the group that the bug has been fixed.

Mon Sep 11 14:03:41 PDT 1995

---

Perhaps later critical path changes:
what was idea to make comparator faster? Check log book!

---

Representation Languages:

Hardware Representation Languages:

- Block Diagrams: FU's, Registers, & Dataflows
- Register Transfer Diagrams: Choice of busses to connect FU's, Regs
- Flowcharts
- State Diagrams

Fifth Representation "Language": Hardware Description Languages

E.G., ISP
VHDL
Verilog

Two different ways to describe sequencing & microoperations

Descriptions in these languages can be used as input to

- simulation systems
- "software breadboard"
- synthesis systems
generate hw from high level description

"To Design is to Represent"
Levels of Description

- **Architectural Simulation**: models programmer’s view at a high level; written in your favorite programming language.
- **Functional/Behavioral/Dataflow**: more detailed model, like the block diagram view.
- **Register Transfer**: commitment to datapath FU’s, registers, busses; capture xfer operations are clock phase accurate.
- **Logic**: model is in terms of logic gates; higher level MSI functions described in terms of these electrical behavior; accurate waveforms.
- **Circuit**: Schematic capture + logic simulation package like Xilinx ISE. Special languages + simulation systems for describing the inherent parallel activity in hardware.

Netlist

- A key data structure (or representation) in the design process is the “netlist”:
  - Network List
  - A netlist lists components and connects them with nodes:
    - Less Abstract
    - More Accurate
    - Slower Simulation

- Alternative format:
  - `n1 g1.in1 n2 g1.in2 n3 g2.in1 n4 g2.in2 n5 g1.out g3.in1 n6 g2.out g3.in2 n7 g3.out g1 "and" g2 "and" g3 "or"

  ° Netlist is what is needed for simulation and implementation.
  ° Could be at the transistor level, gate level, ...
  ° Could be hierarchical or flat.
  ° How do we generate a netlist?

Design Flow

- Circuit is described and represented:
  - Graphically (Schematics)
  - Textually (HDL)
  - Other (Special Compilers)
    - Memories
    - Error Correcting Circuits

- Result of circuit specification (and compilation) is a netlist of:
  - generic primitives - logic gates, flip-flops, or
  - technology specific primitives - LUTs/CLBs, transistors, discrete gates, or
  - higher level library elements - adders, ALUs, register files, decoders, etc.
High-level Analysis is used to verify:
- correct function
- rough:
  - timing
  - power
  - cost

Common tools used are:
- simulator - check functional correctness, and
- static timing analyzer
  - estimates circuit delays based on timing model and delay parameters for library elements (or primitives).

Technology Mapping:
- Converts netlist to implementation technology dependent details
  - Expands library elements,
  - Performs:
    - partitioning,
    - placement,
    - routing

Low-level Analysis
- Simulation and Analysis Tools perform low-level checks with:
  - accurate timing models,
  - wire delay
- For FPGAs this step could also use the actual device.

Netlist: used between and internally for all steps.

Schematics are intuitive. They match our use of gate-level or block diagrams.

Somewhat physical. They imply a physical implementation.
- This is why we use them for datapaths

Require a special tool (editor).

Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow.
High Level Design Languages (HDLs)

- **Basic Idea:**
  - Language constructs describe circuits with two basic forms:
    - **Structural descriptions** similar to hierarchical netlist.
    - **Behavioral descriptions** use higher-level constructs (similar to conventional programming).

- Originally designed to help in abstraction and simulation.
- Now "logic synthesis" tools exist to automatically convert from behavioral descriptions to gate netlist.
- Greatly improves designer productivity.
- However, this may lead you to falsely believe that hardware design can be reduced to writing programs!

**“Structural” example:**
```
Decoder(output x0, x1, x2, x3; inputs a, b)
{
    wire abar, bbar; inv(bbar, b); inv(abar, a); nand(x0, abar, bbar); nand(x1, abar, b);
    nand(x2, a, bbar); nand(x3, a, b);
}
```

**“Behavioral” example:**
```
Decoder(output x0, x1, x2, x3; inputs a, b)
{
    case [a b]
    00: [x0 x1 x2 x3] = 0x0;
    01: [x0 x1 x2 x3] = 0x2;
    10: [x0 x1 x2 x3] = 0x4;
    11: [x0 x1 x2 x3] = 0x8;
    endcase;
}
```

Verilog History

  - Invented as simulation language.
  - Synthesis was an afterthought. Many techniques for synthesis developed at Berkeley in 80's and applied commercially in the 90's.
- Around the same time as the origin of Verilog, the US Department of Defense developed VHDL.
  - Because it was in the public domain it began to grow in popularity.
  - VHDL is still popular within the government, in Europe and Japan, and some Universities.
- Standardization
  - Afraid of losing market share, Cadence opened Verilog to the public in 1990.
  - An IEEE working group was established in 1993, and ratified IEEE Standard 1394 (Verilog) in 1995.
  - Verilog is language of choice of Silicon Valley companies, initially because of high-quality tool support and its similarity to C-language syntax.
- Most major CAD frameworks now support both VHDL and Verilog.

Example: Structural XOR (xor built-in, but..)
```
module xor(Z, X, Y);
    input X, Y;
    output Z;
    wire notX, notY, XnotY, YnotX;
    not (notX, X), (notY, Y);
    or (Z, XnotY, YnotX);
    and (YnotX, notX, Y), (XnotY, X, notY);
endmodule
```

Says which “ports” input, output
Default is 1 bit wide data
“nets” to connect components

Note: order of gates doesn’t matter, since structure determines relationship

Example: Behavioral XOR in Verilog
```
module xorB(Z, X, Y);
    input X, Y;
    output Z;
    reg Z;
    always @(X or Y)
        Z = X ^ Y; // ^ is C operator for xor
endmodule;
```

- Unusual parts of above Verilog
  - “always @(X or Y)” => whenever X or Y changes, do the following statement
  - “reg” is only type of behavioral data that can be changed in assignment, so must redefine Z as reg
  - Default is single bit data types: X, Y, Z
Verilog big idea: Time in code

- Difference from normal prog. lang. is that time is part of the language
  - part of what trying to describe is when things occur, or how long things will take
- Underlying simulation system is event driven:
  - Changes on signals cause events
  - Events trigger firing of attached components
  - Changes to signals never visible in zero time!
    - But behavioral signals can only be visible
    - only visible if time advances at least one ‘tick’
- Simulation time does not advance without timing control of some sort.
  Examples of timing control:
  - Gate or wire delays can schedule events in the future
  - Delay controls, introduced with the “#” symbol
  - Event controls, introduced by the “@” symbol
  - The wait statement

Delay Specifications

`timescale 1ns/1ps
//Dataflow description of mux
module mux2 in0, in1, select, out;
  input in0, in1, select;
  output out;
  assign out = #5,10 select ? in1 : in0;
endmodule // mux2

Notes:
- Delay specifications relative to timescale specification
- May be placed in many different syntactical positions
- #single number
  - Delay specification for both edges
- #(rising,falling)
  - Delay specification for rising and falling edges

Time Example

module test(stream);
  output stream;
  reg stream;
  initial
    begin
      stream = 0; #2 stream = 1; #5 stream = 0; #3 stream = 1; #4 stream = 0;
    end
endmodule

- “Initial” means do this code once
- Note: Verilog uses begin ... end vs. {...} as in C
- #2 stream = 1 means wait 2 ns before changing stream to 1
- Output called a “waveform”

Time, variable update, and monitor

or #2(Z, X, Y);

The instant before the rising edge of the clock, all outputs and wires have their OLD values.

This includes inputs to flip flops.
- Therefore, if you change the inputs to a flip flop at a particular rising edge, that change will not be reflected at the output until the NEXT rising edge. This is because when the rising edge occurs, the flip flop still sees the old value. So when simulated time changes in Verilog, then ports, registers updated
### Sequential Logic

// Sequential Logic - involves an edge

**Notes:**
- "always @(posedge CLK)" forces Q register to be rewritten every simulation cycle.
- ">>" operator does right shift (shifts in a zero on the left).
- Shifts on non-reg variables can be done with concatenation:
  ```
  wire [3:0] A, B;
  assign B = (1'b0, A[3:1])
  ```

//Parallel to Serial converter

```verilog
module FF (CLK, Q, D);
  input D, CLK;
  output Q, reg Q;
  always @(posedge CLK) Q = D;
endmodule // FF
```

```
module ParToSer (LD, X, out, CLK);
  input [3:0] X;
  input LD, CLK;
  output out;
  reg out;
  reg [3:0] Q;
  assign out = Q[0];
  always @(posedge CLK)
    if (LD) Q = X;
    else Q = Q[1:3];
endmodule // mux2
```

### Example: Replicated XOR in Verilog

```
module 4xor (C, A, B);
  input [3:0] A, B;
  output [3:0] C;
  xor foo4xor[3:0] (.X(A), .Y(B), .Z(C));
endmodule;
```

**Note 1:** can associate ports explicitly by name,
- `(X (A), Y (B), Z(C))`
**Note 2:** must give a name to new instance of xors (`foo4xor`)
2-1 Mux in Dataflow Form

// Dataflow description of mux
module mux2 (in0, in1, select, out);
  input in0, in1, select;
  output out;
  assign out = (~select & in0) | (select & in1);
endmodule // mux2

Alternative:
assign out = select ? in1 : in0;

Notes:
• provides a way to describe combinational logic by its function rather than gate structure (similar to Boolean expressions).
• The assign keyword is used to indicate a continuous assignment. Whenever anything on the RHS changes the LHS is updated.

Combining modules: Hierarchy & Bit Vectors

// Assuming we have already
// defined a 2-input mux (either
// structurally or behaviorally,

// 4-input mux built from 3 2-input muxes
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)),
  m2 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4

Notes:
• instantiation similar to primitives
• select is 2-bits wide

2-to-1 mux Behavioral description

// Behavioral model of 2-to-1
// multiplexer.
module mux2 (in0, in1, select, out);
  input in0, in1, select;
  output out;
  reg out;
  always @(in0 or in1 or select)
    if (select) out = in1;
    else out = in0;
endmodule // mux2

• Behavioral: use keyword always followed by one procedural statement
  – Use Begin/End to place more statements after always
  – @() specifier: wait until an event (here, change on one of 3 sigs)
• Output of procedural assignments must of of type reg
  – a reg type retains its value until a new value is assigned
  – Not necessarily a real register: only for @(posedge signal)

Behavioral 4-to1 mux

// 4-input mux behavioral description
module mux4 (in0, in1, in2, in3, select, out);
  input in0, in1, in2, in3;
  input [1:0] select;
  output out;
  reg out;
  always @(in0 or in1 or in2 or in3 or select)
    case (select)
      2'b00: out = in0;
      2'b01: out = in1;
      2'b10: out = in2;
      2'b11: out = in3;
    endcase
endmodule // mux4

Notes:
• Case construct equivalent to nested if constructs.
• Definition: A structural description is one where the function of the module is defined by the instantiation and interconnection of sub-modules.
• A behavioral description uses higher level language constructs and operators.
• Verilog allows modules to mix both behavioral constructs and sub-module instantiation.
Behavioral with Bit Vectors

// Behavioral model of 32-bit wide 2-to-1 multiplexer.
module mux32 (in0, in1, select, out);
  input [31:0] in0, in1;
  input select;
  output [31:0] out;
  reg [31:0] out;
  always @ (in0 or in1 or select)
    if (select) out = in1; else out = in0;
endmodule

// Behavioral model of 32-bit adder.
module add32 (A, B, S, C);
  input [31:0] A, B;
  output [31:0] S;
  output C;
  reg [31:0] S;
  reg C;
  always @ (A or B)
    [31:0] {C, S} = A + B;
endmodule

Testing: Make sure that things work

° Testing methodologies
- Understand what correct behavior is when you design things
  - Collect vectors for later use
- Build monitor modules to check assertions of correct values
  - Produce a regression test
    - Set of tests to run each time something changes

° Types of test (Doug Clark):
  - Directed Vectors – test explicit behavior
  - Random Vectors – apply random values or orderings to device
  - Daemons – continuous error insertion

° Monitor modules:
  - Check to see if invariants are maintained during long running simulations

Monitor Modules: Passthrough testing

module monitorsum32(carry, sum, A, B);
  input [31:0] A, B;
  output [31:0] sum;
  output carry;
  reg [31:0] predsum;
  reg precarry;
  // The "real" adder
  sum32 mysum (.carry, .sum, A, B);
  `ifndef synthesis
    // This checker code only for simulation
    always @ (A or B)
      begin
        #100 // wait for output to settle (don't make too long!)
        predsum = A + B;
        if (cary != precarry) || (sum != predsum)
          $display(">>> Mismatch: 0x%x + 0x%x -> 0x%x carry %x", A, B, sum, carry);
      end
  `endif
endmodule

Testbench: Applying Directed Vectors

module testmux;
  reg a, b, s;
  wire f;
  reg expected;
  // Unit under test.
  mux2 myMux (.select(s), .in0(a), .in1(b), .out(f);
  initial
    begin
      s = 0; a = 0; b = 1; expected = 0; #10 a = 1; b = 0; expected = 1;
      s = 1; a = 0; b = 1; expected = 1;
    end
  initial
    $monitor("select=%b in0=%b in1=%b out=%b, expected out=%b time=%d", s, a, b, f, expected, $time);
endmodule

° Top-level modules written specifically to test sub-modules.
° Notes:
  - initial block similar to always except only executes once (at beginning of simulation)
  - #n's needed to advance time
  - $monitor - prints output
  - A variety of other "system functions", similar to monitor exist for displaying output and controlling the simulation.
Testbench: Randomized Vector Testing

```verilog
module testbench();
    reg [31:0] A, B;
    wire [31:0] sum;
    wire carry;
    reg [31:0] predsum;
    reg predcarry;
    // Device under test
    sum32 mysum (carry,sum,A,B);
    always begin
        A = $random; B = $random;
        #100 //wait for output to settle
        (predcarry,predsum) = A + B;
        if ((carry != predcarry) || (sum != predsum))
            $display(">>> Mismatch: 0x%x+0x%x->0x%x carry %x", A,B,sum,carry);
        else
            $display("Successful: 0x%x+0x%x=0x%x carry %x", A,B,sum,carry);
    end
endmodule
```

* Source of Vectors:
  - With $random-predicted result
  - Actual vectors

* Check actual results against predicted

More Verilog Help

* The lecture notes only cover the very basics of Verilog and mostly just the conceptual issues.
* The Mano textbook covers Verilog with many examples.
* The Bhasker book is a good tutorial.

On reserve in the Engineering

* Complete language spec from the IEEE available on handouts page
* Also, pretty good tutorial available on handouts page
* Synplify manual (for when we start using synthesis)

FPGA Variations

* Families of FPGA's differ in:
  - physical means of implementing user programmability,
  - arrangement of interconnection wires, and
  - basic functionality of logic blocks

* Most significant difference is in the method for providing flexible blocks and connections:
  - Anti-fuse based (ex: Actel)
  - Non-volatile, non-reprogrammable
  - fixed (non-reprogrammable)

(Almost used in 150 Lab: only 1-shot at getting it right!)

Where are FPGAs in the IC Zoo?

![Diagram of IC Zoo with FPGA, CPLDs, SPLDs, NAND arrays, and other logic devices]

Acronyms
- SPLD = Simple Prog. Logic Device
- PAL = Prog. Array of Logic
- CPLD = Complex PLD
- FPGA = Field Prog. Gate Array

(Standard logic is SSI or MSI buffers, gates)
**User Programmability**

- Latch-based (Xilinx, Altera, …)
  - Latches are used to:
    1. make or break cross-point connections in interconnect
    2. define function of logic blocks
    3. set user options:
      - within the logic blocks
      - in the input/output blocks
      - global reset/clock
  - “Configuration bit stream” loaded under user control:
    - All latches are strung together in a shift chain
    - “Programming” => creating bit stream
  - Note: Today 90% die is interconnect, 10% is gates

**Idealized FPGA Logic Block**

- 4-input Look Up Table (4-LUT)
  - implements combinational logic functions
  - Register
    - optionally stores output of LUT
    - Latch determines whether read reg or LUT

**4-LUT Implementation**

- n-bit LUT is actually implemented as a 2^n x 1 memory:
  - inputs choose one of 2^n memory locations.
  - memory locations (latches) are normally loaded with values from user’s configuration bit stream.
  - Inputs to mux control are the CLB (Configurable Logic Block) inputs.
- Result is a general purpose “logic gate”.
- n-LUT can implement any function of n inputs!

**LUT as general logic gate**

- An n-lut as a direct implementation of a function truth-table
  - Each latch location holds value of function corresponding to one input combination

**Example: n-lut**

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>0000</th>
<th>F(0,0,0,0)</th>
<th>store in 1st latch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0001</td>
<td>F(0,0,0,1)</td>
<td>store in 2nd latch</td>
</tr>
<tr>
<td></td>
<td>0010</td>
<td>F(0,0,1,0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0011</td>
<td>F(0,0,1,1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1001</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example: 2-lut**

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Implements any function of 2 inputs.
How many functions of n inputs?
More functionality for “free”? 

° Given basic idea
  • LUT built from RAM
  • Latches connected as shift register
° What other functions could be provided at very little extra cost? 
1. Using CLB latches as little RAM vs. logic
2. Using CLB latches as shift register vs. logic

1. “Distributed RAM”

° CLB LUT configurable as Distributed RAM
  • A LUT equals 16x1 RAM
  • Implements Single and Dual-Ports
  • Cascade LUTs to increase RAM size
° Synchronous write
° Synchronous/Asynchronous read
  • Accompanying flip-flops used for synchronous read

Block RAM (Extra RAM not using LUTs)

° Most efficient memory implementation
  • Dedicated blocks of memory
° Ideal for most memory requirements
  • Virtex-E XCV2000 has 160? blocks
    • 4096 x 1
    • 2048 x 2
    • 1024 x 4
    • 512 x 8
    • 256 x 16
° Increase memory depth or width by cascading blocks

Virtex-E Block RAM
2. Shift Register

- Each LUT can be configured as shift register
  - Serial in, serial out
- Saves resources: can use less than 16 FFs
- Faster: no routing
- Note: CAD tools determine with CLB used as LUT, RAM, or shift register, rather than up to designer

---

How Program: FPGA Generic Design Flow

- Design Entry:
  - Create your design files using:
    - schematic editor or
    - hardware description language (Verilog, VHDL)

- Design “implementation” on FPGA:
  - Partition, place, and route (“PPR”) to create bit-stream file
  - Divide into CLB-sized pieces, place into blocks, route to blocks

- Design verification:
  - Use Simulator to check function,
  - Other software determines max clock frequency.
  - Load onto FPGA device (cable connects PC to board)
    - check operation at full speed in real environment.

---

Example Partition, Placement, and Route

- Idealized FPGA structure:
- Example Schematic Circuit:
  - collection of gates and flip-flops

Circuit combinational logic must be “covered” by 4-input 1-output “gates”. Flip-flops from circuit must map to FPGA flip-flops. (Best to preserve “closeness” to CL to minimize wiring.) Placement in general attempts to minimize wiring.

---

Xilinx Virtex-E Routing Hierarchy

- Internal 3-state Bus
- Long lines and Global lines
- Buffered Hex lines (1/6 blocks)
- Single-length lines

Note:
CAD tools do PPR, not designers

- 24 single-length lines
  - Route GRM signals to adjacent GRMs in 4 directions
- 96 buffered hex lines
  - Route GRM (general routing matrix) signals to another GRMs six blocks away in each of the 4 directions
- 12 buffered Long lines
  - Routing across top and bottom left and right
**Virtex-E Configurable Logic Block (CLB)**

2 "logic slices" / CLB, two 4-LUTs / slice
=> Four 4-LUTs / CLB

- **Virtex-E CLB Slice Structure**
  - Each slice contains two sets of the following:
    - Four-input LUT
      - Any 4-input logic function
      - Or 16-bit x 1 sync RAM
      - Or 16-bit shift register
    - Carry & Control
      - Fast arithmetic logic
      - Multiplexer logic
      - Multiplier logic
    - Storage element
      - Latch or flip-flop
      - Set and reset
      - True or inverted inputs
      - Sync. or async. control

- **Details of Virtex-E Slice**
  - Very fast ripple carry: (24-bit @ 100 MHz)
  - Multiplexors to help combine CLBs into larger multiplexor

- **Virtex-E Dedicated Expansion Multiplexers**
  - Since 4-LUT has 4 inputs, max is 2:1 Mux (2 inputs, 1 control line)
  - MUXF5 combines 2 LUTs to create
    - 4x1 multiplexer
    - Or any 5-input function (5-LUT)
    - Or selected functions up to 9 inputs
  - MUXF6 combines 2 slices to form
    - 8x1 multiplexer
    - Or any 6-input function (6-LUT)
    - Or selected functions up to 19 inputs
  - Dedicated muxes are faster and more space efficient
**Xilinx Virtex-E Chip Floorplan**

- Input / Output Blocks (IOBs)
- Configurable Logic Blocks (CLBs)
- Block RAMs (BRAMs)
- Delay Locked Loop (DLL)
- VersaRing

---

**Virtex-E Delay Lock Loop (DLL) Capabilities**

- Easy clock duplication
  - System clock distribution
  - Cleans and reconditions incoming clock
- Quick and easy frequency adjustment
- Single crystal easily generates multiple clocks
- Excellent for advanced memory types
- De-skew incoming clock
- Generate fast setup and hold time or fast clock-to-outs

---

**DLL: Multiplication of Clock Speed**

- Have faster internal clock relative to external clock source
- Use 1 DLL for 2x multiplication
- Combine 2 DLLs for 4x multiplication
- Reduce board EMI
  - Route low-frequency clock externally and multiply clock on-chip

---

**DLL: Division of Clock Speed**

- Selectable division values
  - 1.5, 2, 2.5, 3, 4, 5, 8, or 16
- Cascade DLLs to combine functions
  - Combine DLLs to multiply and divide to get desired speed
- 50/50 duty cycle correction available
Clock Management Summary

- All digital DLL Implementation
  - Input noise rejection
  - 50/50 duty cycle correction
- Clock mirror provides system clock distribution
- Multiply input clock by 2x or 4x
- Divide clock by 1.5, 2, 2.5, 3, 4, 5, 8, or 16
- De-skew clock for fast setup, hold, or clock-to-out times

Virtex-E Family of Parts

<table>
<thead>
<tr>
<th>Device</th>
<th>System Gates</th>
<th>Logic Gates</th>
<th>CLB Array</th>
<th>Logic Cells</th>
<th>Differential I/O Pairs</th>
<th>User I/O</th>
<th>Block/RAM Bits</th>
<th>Distributed RAM Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCV1000E</td>
<td>128,236</td>
<td>32,400</td>
<td>20 x 30</td>
<td>2,700</td>
<td>83</td>
<td>196</td>
<td>81,020</td>
<td>38,400</td>
</tr>
<tr>
<td>XCV2000E</td>
<td>306,363</td>
<td>63,504</td>
<td>28 x 42</td>
<td>5,292</td>
<td>119</td>
<td>204</td>
<td>114,688</td>
<td>75,264</td>
</tr>
<tr>
<td>XCV3000E</td>
<td>411,055</td>
<td>82,044</td>
<td>32 x 48</td>
<td>6,012</td>
<td>137</td>
<td>316</td>
<td>131,072</td>
<td>98,364</td>
</tr>
<tr>
<td>XCV4000E</td>
<td>569,952</td>
<td>129,600</td>
<td>40 x 60</td>
<td>10,800</td>
<td>183</td>
<td>404</td>
<td>163,840</td>
<td>153,680</td>
</tr>
<tr>
<td>XCV4000E</td>
<td>983,663</td>
<td>186,634</td>
<td>48 x 72</td>
<td>15,557</td>
<td>247</td>
<td>512</td>
<td>294,912</td>
<td>221,184</td>
</tr>
<tr>
<td>XCV1000E</td>
<td>1,569,178</td>
<td>331,776</td>
<td>64 x 96</td>
<td>27,649</td>
<td>301</td>
<td>620</td>
<td>353,216</td>
<td>353,216</td>
</tr>
<tr>
<td>XCV1500E</td>
<td>2,188,742</td>
<td>419,904</td>
<td>72 x 128</td>
<td>34,993</td>
<td>344</td>
<td>724</td>
<td>589,674</td>
<td>497,664</td>
</tr>
<tr>
<td>XCV2000E</td>
<td>2,541,952</td>
<td>516,400</td>
<td>80 x 120</td>
<td>43,200</td>
<td>344</td>
<td>804</td>
<td>655,360</td>
<td>616,400</td>
</tr>
<tr>
<td>XCV2600E</td>
<td>3,263,735</td>
<td>685,564</td>
<td>92 x 138</td>
<td>57,132</td>
<td>344</td>
<td>804</td>
<td>753,654</td>
<td>812,744</td>
</tr>
<tr>
<td>XCV3200E</td>
<td>4,014,387</td>
<td>870,066</td>
<td>104 x 156</td>
<td>73,008</td>
<td>344</td>
<td>804</td>
<td>851,958</td>
<td>1,038,336</td>
</tr>
</tbody>
</table>

Summary I

- Design Process
  - Design Entry: Schematics, HDL, Compilers
  - High Level Analysis: Simulation, Testing, Assertions
  - Technology Mapping: Turn design into physical implementation
  - Low Level Analysis: Check out Timing, Setup/Hold, etc
- Verilog – Three programming styles
  - Structural: Like a Netlist
    - Instantiation of modules + wires between them
  - Dataflow: Higher Level
    - Expressions instead of gates
  - Behavioral: Hardware programming
    - Full flow-control mechanisms
    - Registers, variables
    - File I/O, console display, etc

Summary: Xilinx FPGAs

- How they differ from idealized array:
  - In addition to their use as general logic “gates”, LUTs can alternatively be used as general purpose RAM or shift register
    - Each 4-LUT can become a 16x1-bit RAM array
  - Special circuitry to speed up “ripple carry” in adders and counters
    - Therefore adders assembled by the CAD tools operate much faster than adders built from gates and LUTs alone.
  - Many more wires, including tri-state capabilities.
In conclusion, FPGAs...

* FPGAs are basically interconnect plus distributed RAM that can be programmed to act as any logical function of 4 inputs
* CAD tools due the partitioning, routing and placement functions onto CLBs
* FPGAs offer compromise of performance, unit cost, time to market vs. ASICs and microprocessors plus software