module mux2to1 (w0, w1, s, f);
  input w0, w1, s;
  output f;
  reg f;
  always @(w0 or w1 or s)
    f = s ? w1 : w0;
endmodule

module mux4to1 (w0, w1, w2, w3, S, f);
  input w0, w1, w2, w3;
  input [1:0] S;
  output f;
  assign f = S[1] ? (S[0] ? w3 : w2) : (S[0] ? w1 : w0);
endmodule

Combinational network
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;
    reg f;

    always @(w0 or w1 or s)
        if (s==0)
            f = w0;
        else
            f = w1;

endmodule

module mux4to1 (w0, w1, w2, w3, S, f);
    input w0, w1, w2, w3;
    input [1:0] S;
    output f;
    reg f;

    always @(w0 or w1 or w2 or w3 or S)
        if (S == 2'b00)
            f = w0;
        else if (S == 2'b01)
            f = w1;
        else if (S == 2'b10)
            f = w2;
        else if (S == 2'b11)
            f = w3;

endmodule

module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output f;
    reg f;

    always @(W or S)
        if (S == 0)
            f = W[0];
        else if (S == 1)
            f = W[1];
        else if (S == 2)
            f = W[2];
        else if (S == 3)
            f = W[3];

endmodule

module mux16to1 (W, S16, f);
    input [0:15] W;
    input [3:0] S16;
    output f;
    wire [0:3] M;

    mux4to1 Mux1 (W[0:3], S16[1:0], M[0]);
    mux4to1 Mux2 (W[4:7], S16[1:0], M[1]);
    mux4to1 Mux3 (W[8:11], S16[1:0], M[2]);
    mux4to1 Mux4 (W[12:15], S16[1:0], M[3]);
    mux4to1 Mux5 (M[0:3], S16[3:2], f);

endmodule
module mux4to1 (W, S, f);
  input [0:3] W;
  input [1:0] S;
  output f;
  reg f;
  always @(W or S)
    case (S)
      0: f = W[0];
      1: f = W[1];
      2: f = W[2];
      3: f = W[3];
    endcase
endmodule

module dec2to4 (W, Y, En);
  input [1:0] W;
  input En;
  output [0:3] Y;
  reg [0:3] Y;
  always @(W or En)
    begin
      if (En == 0)
        Y = 4'b0000;
      else
        case (W)
          0: Y = 4'b1000;
          1: Y = 4'b0100;
          2: Y = 4'b0010;
          3: Y = 4'b0001;
        endcase
    end
endmodule

module dec4to16(W, Y, En);
  input [3:0] W;
  input En;
  output [0:15] Y;
  wire [0:3] M;
  dec2to4 Dec1 (W[3:2], M[0:3], En);
  dec2to4 Dec2 (W[1:0], Y[0:3], M[0]);
  dec2to4 Dec3 (W[1:0], Y[4:7], M[1]);
  dec2to4 Dec4 (W[1:0], Y[8:11], M[2]);
  dec2to4 Dec5 (W[1:0], Y[12:15], M[3]);
endmodule
module seg7 (bcd, leds);
input [3:0] bcd;
output [1:7] leds;
reg [1:7] leds;
always @(bcd)
case (bcd)       //abcdefg
0: leds = 7'b1111110;
1: leds = 7'b0110000;
2: leds = 7'b1101101;
3: leds = 7'b1111001;
4: leds = 7'b0110011;
5: leds = 7'b1011011;6: leds = 7'b1011111;
7: leds = 7'b1110000;
8: leds = 7'b1111111;
9: leds = 7'b1111111;
default: leds = 7'bx;
endcase
endmodule

module alu(s, A, B, F);
input [2:0] s;
input [3:0] A, B;
output [3:0] F;
reg [3:0] F;
always @(s or A or B)
case (s)
0: F = 4'b0000;
6: F = A & B;
7: F = 4'b1111;
endcase
endmodule

module priority (W, Y, z);
input [3:0] W;
output [1:0] Y;output z;
reg [1:0] Y;reg z;
always @(W)
begin
z = 1;casex(W)
4'b1xxx: Y = 3;
4'b01xx: Y = 2;
4'b001x: Y = 1;
4'b0001: Y = 0;
default: begin
z = 0;
Y = 2'bx;
end
endcase
end
endmodule

module dec2to4 (W, Y, En);
input [1:0] W;
input En;
output [0:3] Y;reg [0:3] Y;integer k;
always @(W or En)
for (k = 0; k <= 3; k = k+1)
if ((W == k) && (En == 1))
Y[k] = 1;
else
Y[k] = 0;
endfor
endmodule
module priority (W, Y, z);
input [3:0] W;
output [1:0] Y;
output z;
reg [1:0] Y;
reg z;
integer k;
always @(W)
begin
    Y = 2'bx;
    z = 0;
    for (k = 0; k < 4; k = k+1)
        begin
            if(W[k])
                begin
                    Y = k;
                    z = 1;
                end
        end
endmodule

Topics

- Number representation.
- Shifters.
- Adders and ALUs.

Signed number representations

- One’s complement:
  - 3  = 0101
  - -3 = ~(0101) = 1010
  - Two zeroes: 0000, 1111

- Two’s complement:
  - 3  = 0101
  - -3 = ~(0101) +1 = 1011
  - One zero: 0000

Representations and arithmetic

- \( N = \sum 2^i b_i \)
- Test for zero: all bits are 0.
- Test for negative: sign bit is 1.
- Subtraction: negate then add.
  - \( a - b = a + (-b) = a + (~b + 1) \)
Combinational shifters

- Useful for arithmetic operations, bit field extraction, etc.
- Latch-based shift register can shift only one bit per clock cycle.
- A multiple-shift shifter requires additional connectivity.

Barrel shifter

- Can perform n-bit shifts in a single cycle.

Barrel shifter structure

Accepts $2n$ data inputs and $n$ control signals, producing $n$ data outputs.

Barrel shifter operation

- Selects arbitrary contiguous $n$ bits out of $2n$ input bits.
- Examples:
  - right shift: data into top, 0 into bottom;
  - left shift: 0 into top, data into bottom;
  - rotate: data into top and bottom.
Verilog for barrel shifter

```verilog
module shifter(data,b,result);
    parameter Nminus1 = 31; /* 32-bit shifter */
    input [Nminus1:0] data; /* compute parity of these bits */
    input [3:0] b; /* amount to shift */
    output [Nminus1:0] result; /* shift result */
    assign result = data << b;
endmodule
```

Adders

- Adder delay is dominated by carry chain.
- Carry chain analysis must consider transistor, wiring delay.
- Modern VLSI favors adder designs which have compact carry chains.

Full adder

- Computes one-bit sum, carry:
  - \( s_i = a_i \ XOR \ b_i \ XOR \ c_i \)
  - \( c_{i+1} = a_i b_i + a_i c_i + b_i c_i \)
- Half adder computes two-bit sum.
- Ripple-carry adder: n-bit adder built from full adders.
- Delay of ripple-carry adder goes through all carry bits.

Verilog for full adder

```verilog
module fulladd(a,b,carryin,sum,carryout);
    input a, b, carryin; /* add these bits*/
    output sum, carryout; /* results */
    assign {carryout, sum} = a + b + carryin;
endmodule
```
Verilog for ripple-carry adder

```verilog
module nbitfulladd(a, b, carryin, sum, carryout)
  input [7:0] a, b; /* add these bits */
  input carryin; /* carry in*/
  output [7:0] sum; /* result */
  output carryout; /* transfers the carry between bits */
  wire [7:1] carry; /* transfers the carry between bits */

  fulladd a0(a[0], b[0], carryin, sum[0], carry[1]);
  fulladd a1(a[1], b[1], carry[1], sum[1], carry[2]);
  ...  
  fulladd a7(a[7], b[7], carry[7], sum[7], carryout));
endmodule
```

Carry-lookahead adder

- First compute carry propagate, generate:
  - $P_i = a_i + b_i$
  - $G_i = a_i \cdot b_i$

- Compute sum and carry from $P$ and $G$:
  - $s_i = c_i \oplus P_i \oplus G_i$
  - $c_{i+1} = G_i + P_i c_i$

Carry-lookahead expansion

- Can recursively expand carry formula:
  - $c_{i+1} = G_i + P_i (G_{i-1} + P_{i-1} c_{i-1})$
  - $c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} (G_{i-2} + P_{i-1} c_{i-2})$

- Expanded formula does not depend on intermediate carries.
- Allows carry for each bit to be computed independently.

Depth-4 carry-lookahead

![Depth-4 carry-lookahead diagram with symbols and connections between inputs and outputs]
Analysis

- Deepest carry expansion requires gates with large fanin: large, slow.
- Carry-lookahead unit requires complex wiring between adders and lookahead unit—values must be routed back from lookahead unit to adder.
- Layout is even more complex with multiple levels of lookahead.

Verilog for carry-lookahead carry block

```verilog
module carry_block(a, b, carryin, carry);
    input [3:0] a, b; /* add these bits*/
    input carryin; /* carry into the block */
    output [3:0] carry; /* carry for each bit in the block */
    wire [3:0] g, p; /* generate and propagate */
    assign g[0] = a[0] & b[0]; /* generate 0 */
    assign p[0] = a[0] ^ b[0]; /* propagate 0 */
    assign g[1] = a[1] & b[1]; /* generate 1 */
    assign p[1] = a[1] ^ b[1]; /* propagate 1 */
    ...
    assign carry[0] = g[0] | (p[0] & carryin);
    assign carry[1] = g[1] | p[1] & (g[0] | (p[0] & carryin));
endmodule
```

Verilog for carry-lookahead sum unit

```verilog
module sum(a, b, carryin, result);
    input a, b, carryin; /* add these bits*/
    output result; /* sum */
    assign result = a ^ b ^ carryin; /* compute the sum */
endmodule
```

Verilog for carry-lookahead adder

```verilog
module carry_lookahead_adder(a, b, carryin, sum, carryout);
    input [15:0] a, b; /* add these together */
    input carryin;
    output [15:0] sum; /* result */
    output carryout;
    wire [16:1] carry; /* intermediate carries */
    assign carryout = carry[16]; /* for simplicity */
    /* build the carry-lookahead units */
    carry_block b0(a[3:0], b[3:0], carryin, carry[4:1]);
    carry_block b1(a[7:4], b[7:4], carry[4], carry[8:5]);
    carry_block b2(a[11:8], b[11:8], carry[8], carry[12:9]);
    carry_block b3(a[15:12], b[15:12], carry[12], carry[16:13]);
    /* build the sum */
    sum a0(a[0], b[0], carryin, sum[0]);
    sum a1(a[1], b[1], carryin[1], sum[1]);
    ...
    sum a15(a[15], b[15], carryin[15], sum[15]);
endmodule
```
Carry-skip adder

- Looks for cases in which carry out of a set of bits is identical to carry in.
- Typically organized into $b$-bit stages.
- Can bypass carry through all stages in a group when all propagates are true: $P_i \ P_{i+1} \ \ldots \ P_{i+b-1}$.
  - Carry out of group when carry out of last bit in group or carry is bypassed.

Two-bit carry-skip structure

- Worst-case carry-skip
  - Worst-case carry-propagation path goes through first, last stages:
Verilog for carry-skip add with P

```verilog
module fulladd_p(a, b, carryin, sum, carryout, p);
    input a, b, carryin; /* add these bits*/
    output sum, carryout, p; /* results including propagate */

    assign {carryout, sum} = a + b + carryin;
    /* compute the sum and carry */
    assign p = a | b;
endmodule
```

Verilog for carry-skip adder

```verilog
module carryskip(a, b, carryin, sum, carryout);
    input [7:0] a, b; /* add these bits */
    input carryin; /* carry in */
    output [7:0] sum; /* result */
    output carryout;

    wire [8:1] carry; /* transfers the carry between bits */
    wire [7:0] p; /* propagate for each bit */
    wire cs4; /* final carry for first group */

    fulladd_p a0(a[0], b[0], carryin, sum[0], carry[1], p[0]);
    fulladd_p a1(a[1], b[1], carry[1], sum[1], carry[2], p[1]);
    fulladd_p a2(a[2], b[2], carry[2], sum[2], carry[3], p[2]);
    fulladd_p a3(a[3], b[3], carry[3], sum[3], carry[4], p[3]);
    fulladd_p a4(a[4], b[4], carry4, sum[4], carry5, p[4]);

endmodule
```

Delay analysis

- Assume that skip delay = 1 bit carry delay.
- Delay of k-bit adder with block size b:
  - \[ T = (b-1) + 0.5 + (k/b -2) + (b-1) \]
  - block 0 OR gate skips last block
- For equal sized blocks, optimal block size is \( \sqrt{k/2} \).

Carry-select adder

- Computes two results in parallel, each for different carry input assumptions.
- Uses actual carry in to select correct result.
- Reduces delay to multiplexer.
**Carry-select structure**

![Carry-select structure diagram](image)

**Carry-save adder**

- Useful in multiplication.
- Input: 3 n-bit operands.
- Output: n-bit partial sum, n-bit carry.
  - Use carry propagate adder for final sum.
- Operations:
  - \( s = (x + y + z) \mod 2 \).
  - \( c = \lfloor (x + y + z) - 2 \rfloor / 2 \).

**FPGA delay model**

- Xing/Yu—ripple-carry adder:
  - n-stage adder divided into x blocks;
  - each block has \( n/x \) stages;
  - block \( k \), \( 1 \leq k \leq x \).
- Delays:
  - ripple-carry \( R(y_k) = \lambda_1 + \delta y_k \)  
  - delay of a single stage
  - carry-generate \( G(y_k) = \lambda_2 + \delta(y_k-1) \)
  - carry-terminate \( T(y_k) = G(y_k) \)

**Carry-skip delay model**

- Consider only inter-CLB delay.
- Delay dominated by interconnect:
  - \( S(y_k) = \lambda_3 + \beta l^2 \)
- Wire length \( l \) is proportional to the number of carry-skip layers.
**Adder comparison**

- Ripple-carry adder has highest performance/cost.
- Optimized adders are most effective in very long bit widths (> 48 bits).

**Serial adder**

- May be used in signal-processing arithmetic where fast computation is important but latency is unimportant.
- Data format (LSB first):

```
0 1 1 0
```

**Serial adder structure**

LSB control signal clears the carry shift register:

```
D Q
```

a+b

LSB
ALUs

- ALU computes a variety of logical and arithmetic functions based on opcode.
- May offer complete set of functions of two variables or a subset.
- ALU built around adder, since carry chain determines delay.

ALU as multiplexer

- Compute functions then select desired one:

Verilog for ALU

```verilog
'define PLUS 0
'define MINUS 1
'define AND 2
'define OR 3
'define NOT 4

module alu(fcode, op0, op1, result, oflo);
    parameter n=16, flen=3; input [flen-1:0] fcode; [n-1:0] op0, op1; output [n-1:0] result; output oflo;

    assign
    {oflo, result} =
        (fcode == 'PLUS) ? (op0 + op1) :
        (fcode == 'MINUS) ? (op0 - op1) :
        (fcode == 'AND) ? (op0 & op1) :
        (fcode == 'OR) ? (op0 | op1) :
        (fcode == 'NOT) ? (~op0) : 0;

endmodule
```

Topics

- Combinational network delay.
- Combinational network energy/power.
Delay characteristics

- Measured from change in inputs to change in outputs.
- Data-dependent:
  - Some inputs give longer delays than others.
- May exercise different paths through the network.

Timing diagram

\[ t_c \geq t_x + t_y \]

Sources of delay

- Gate delay:
  - intrinsic;
  - drive;
  - load.
- Wire:
  - lumped load;
  - transmission line.

Basic gate delay model

- Gate delay \( t_g \).
- Wire delay \( t_w \).
Optimizing a single link

- Custom design---improve gate delay:
  - Transistor sizing.
  - Gate topology.
- FPGA or custom design---improve wire delay:
  - Shorten wire length.
  - Choose wire category.
  - Increase driver size.

Fanout

- Fanout adds capacitance.

Driving fanout

- Adding gates adds capacitance:

Ways to drive large fanout

- Increase sizes of driver transistors. Must take into account rules for driving large loads.
- Add intermediate buffers. This may require/allow restructuring of the logic.
Buffers

- Use layers with lower capacitance.
- Redesign layout to reduce length of wires with excessive delay.

Wire capacitance

Path delay

- Combinational network delay is measured over paths through network.
- Can trace a causality chain from inputs to worst-case output.

Path delay example

- Network
- Graph model
Critical path

- Critical path = path which creates longest delay.
- Can trace transitions which cause delays that are elements of the critical delay path.

Delay model

- Nodes represent gates.
- Assign delays to edges—signal may have different delay to different sinks.
- Lump gate and wire delay into a single value.

Critical path through delay graph

Reducing critical path length

- To reduce circuit delay, must speed up the critical path—reducing delay off the path doesn’t help.
- There may be more than one path of the same delay. Must speed up all equivalent paths to speed up circuit.
- Must speed up cutset through critical path.
False paths

- Logic gates are not simple nodes—some input changes don’t cause output changes.
- A false path is a path which cannot be exercised due to Boolean gate conditions.
- False paths cause pessimistic delay estimates.

False path example

Another false path example

Placement and delay

- Placement helps determine routing.
- Routing determines wire length.
- Wire length determines capacitive load.
Placement and wire capacitance

Optimizing network delay

- Identify the longest path.
- Improve delay along the longest path:
  - Driver delay.
  - Wire delay.
  - Logic restructuring.

Example: adder placement and delay

- N-bit adder:

Bad placement and routing
Better placement and routing

- Placement
- Routing

Logic rewrites

- Can rewrite by using subexpressions.
  - Simplifications affect the cost of rewrites.
- Flattening logic increases gate fanin.
- Logic rewrites may affect gate placement.

Logic transformations

- Deep logic
- Shallow logic

Power optimization

- Transitions cause power consumption.
- Logic network design helps control power consumption:
  - Minimizing capacitance;
  - Eliminating unnecessary glitches.
Glitching example

Gate network:

NOR gate produces 0 output at beginning and end:
- beginning: bottom input is 1;
- end: NAND output is 1;

Difference in delay between application of primary inputs and generation of new NAND output causes glitch.

Adder chain glitching

Explanation

Unbalanced chain has signals arriving at different times at each adder.
A glitch downstream propagates all the way upstream.
Balanced tree introduces multiple glitches simultaneously, reducing total glitch activity.
Power estimation tools

- Power estimator approximates power consumption from:
  - gate network;
  - primary input transition probabilities;
  - capacitive loading.
- May be switch/logic simulation based or use statistical models.

Factorization for low power

- Proper factorization reduces glitching.
  \[
  f = ab + bc + cd
  \]

Factorization techniques

- In example, a has high transition probability, b and c low probabilities.
- Reduce number of logic levels through which high-probability signals must travel in order to reduce propagation of glitches.

Layout for low power

- Place and route to minimize capacitance of nodes with high glitching activity.
- Feed back wiring capacitance values to power analysis for better estimates.