Combinational Logic Design

- Combinational Circuits
- Design Topics
- Analysis Procedure
- Design Procedure
- Common Building Blocks
- Hardware Design Languages
4 Bit ALU Design Elements

if \( S = 0 \) then \( D = B - A \)
if \( S = 1 \) then \( D = A - B \)
if \( S = 2 \) then \( D = A + B \)
if \( S = 3 \) then \( D = -A \)
Combinational Circuits

In combinational circuits, there is no way for a signal to flow from a gate output to one of its inputs.
- so, outputs depend only on current input values (not past)
- non-combinational circuits use feedback to implement storage

Combinational circuits are essential building blocks.

Each output of a combinational circuit is a function of the input values.
- each output can be specified by a truth table or Boolean exp.
- analysis: circuit → specification
- synthesis: specification → circuit
Hierarchical Design

- Complex systems are designed by assembling simpler parts in a systematic and (usually) hierarchical way.
  - complex function at top of hierarchy, simple gates at bottom
  - design process can be top-down or bottom-up
- Key concept is composition of simpler circuit blocks to produce more complex blocks.

9 input odd function

\[ Z_0 = \text{odd}(X_0, \ldots, X_8) \]

\[ \text{odd}(X_0, \ldots, X_8) = \text{odd}(\text{odd}(X_0, X_1, X_2), \text{odd}(X_3, X_4, X_5), \text{odd}(X_6, X_7, X_8)) \]

3 input odd function

\[ Z_0 = \text{odd}(X_0, X_1, X_2) \]

\[ \text{odd}(X_0, X_1) = \text{nand}(\text{nand}(X_0, \text{nand}(X_0, X_1)), \text{nand}(X_1, \text{nand}(X_0, X_1))) \]
Design Concepts

- Hierarchical design is essential for managing complexity & allows us to understand larger circuits.
- Design re-use is a key tool for reducing design effort.
  - apply common building blocks (functional blocks) to construct larger systems
  - large designs may contain many instances of a given block
  - parameterized design elements implement common functions but may differ based on parameter values
    - e.g. an odd function block, with number of inputs as a parameter
- Top-down design, goes from high level specification to simpler components using iterative refinement.
- In bottom-up design, we identify & construct common elements that can be re-used multiple times.
Computer-Aided Design

- **CAD tools are essential to the design of complex parts.**

- **Logic design**
  - schematic capture - interactive creation of logic diagrams
  - hardware description languages - textual representation of circuit function

- **Design verification**
  - logic simulation to check circuit behavior experimentally
  - formal verification tools - automated correctness proofs and assertion checking
  - timing analysis and simulation

- **Implementation**
  - logic synthesis - convert high level spec. to low level gates
  - circuit layout - placement of components, routing of wires
  - details - clock distribution, power, pads, testing
Analyzing Combinational Circuits

- Purpose of analysis is to determine what a circuit does.
- Procedure
  1. verify that circuit is combinational
  2. label all inputs, outputs and internal nets
  3. write logic equations for internal nets in terms of inputs
  4. write logic equations for outputs in terms of inputs and simplify

\[ T_1 = B' \cdot C \quad T_2 = A' \cdot B \]
\[ T_3 = A + T_1 = A + B' \cdot C \]
\[ T_4 = T_2 \oplus D = A' \cdot B \oplus D \]
\[ F_1 = T_3 + T_4 = A + B' \cdot C + B' \cdot D + BD' \]
\[ F_2 = T_2 + D = A' \cdot B + D \]
Derivation of Truth Tables

- Can derive truth tables directly from circuit.
- Procedure
  1. For \( n \) input circuit, truth table has \( 2^n \) rows, one for each binary number from 0 to \( 2^n - 1 \).
  2. Label internal nets and place columns in truth table for internal nets and outputs.
  3. Fill in columns for internal nets and outputs.

<table>
<thead>
<tr>
<th>ABCD</th>
<th>T_1</th>
<th>T_2</th>
<th>T_3</th>
<th>T_4</th>
<th>F_1</th>
<th>F_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Designing Combinational Circuits

- Procedure
  1. Determine number of inputs and outputs and assign a symbol to each.
  2. Derive truth table for each output.
  3. Obtain Boolean expressions for each output.
  4. Create an appropriate logic diagram.
  5. Verify correctness by analysis and/or simulation.

Example: design circuit with 3 inputs, 1 output; the output should be 1 when the binary value of the inputs is <3.

<table>
<thead>
<tr>
<th>XYZ</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ F = X'Y' + X'Z' \]
BCD to Excess 3 Code Converter

- Excess-3 code for a decimal digit is the binary value for the decimal number plus 3.

<table>
<thead>
<tr>
<th>input ABCD</th>
<th>output WXYZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0011</td>
</tr>
<tr>
<td>0001</td>
<td>0100</td>
</tr>
<tr>
<td>0010</td>
<td>0101</td>
</tr>
<tr>
<td>0011</td>
<td>0110</td>
</tr>
<tr>
<td>0100</td>
<td>0111</td>
</tr>
<tr>
<td>0101</td>
<td>1000</td>
</tr>
<tr>
<td>0110</td>
<td>1001</td>
</tr>
<tr>
<td>0111</td>
<td>1010</td>
</tr>
<tr>
<td>1000</td>
<td>1011</td>
</tr>
<tr>
<td>1001</td>
<td>1100</td>
</tr>
</tbody>
</table>

\[ W = A + BC + BD \]

\[ X = B' \overline{C} + B' \overline{D} + BC' \overline{D}' \]

\[ Y = CD + \overline{C} \overline{D} \]
Decoders

A binary-to-unary decoder converts a binary input value with \( n \) bits to one of \( 2^n \) possible output values.

<table>
<thead>
<tr>
<th>( A_2A_1A_0 )</th>
<th>( D_{7}...D_{0} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00000001</td>
</tr>
<tr>
<td>001</td>
<td>00000010</td>
</tr>
<tr>
<td>010</td>
<td>00000100</td>
</tr>
<tr>
<td>011</td>
<td>00001000</td>
</tr>
<tr>
<td>100</td>
<td>00010000</td>
</tr>
<tr>
<td>101</td>
<td>00100000</td>
</tr>
<tr>
<td>110</td>
<td>01000000</td>
</tr>
<tr>
<td>111</td>
<td>10000000</td>
</tr>
</tbody>
</table>

Alternative Implementation

Read MK 111-145
Decoder Schematic & Simulation
**Encoders**

- A unary-to-binary encoder converts one of $2^n$ input values to an encoded binary value.

<table>
<thead>
<tr>
<th>$D_3D_2D_1D_0$</th>
<th>$A_1A_0$</th>
<th>$A_1=D_2+D_3$</th>
<th>$A_0=D_1+D_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- A priority encoder converts the first of $2^n$ input values that are 1 to the corresponding encoded binary value.

<table>
<thead>
<tr>
<th>$D_3D_2D_1D_0$</th>
<th>$A_1A_0V$</th>
<th>$A_1=D_3+D_2$</th>
<th>$A_0=D_3'+D_2D_1$</th>
<th>$V=D_3+D_2+D_1+D_0$ -- valid output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>xx0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001x</td>
<td>011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01xx</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1xxx</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multiplexers

- A multiplexer (a.k.a. data selector) has \( n \) control inputs, \( 2^n \) data inputs & a single data output
  - control input value connects one data input to output
  - circuit similar to decoder
  - optional enable input (input to all AND gates) allows construction of larger muxes.
  - alternative implementation uses transmission gates
Implementing Logic with Multiplexers

To implement an \( n \) input logic function:

- use multiplexer with \( n-1 \) control inputs
- connect first \( n-1 \) inputs to multiplexer control inputs
- connect data inputs to “0”, “1”, \( n^{th} \) input or its complement, as dictated by function required
- this technique used in some programmable logic devices
A demultiplexer has $n$ control inputs, $2^n$ data outputs & a single data input

- control input value connects data input to one of the outputs

Mux & demux can be used to transmit several low speed signals on a single wire.
An increment circuit with \( n \) inputs and \( n+1 \) outputs computes binary value that is one larger than its input.

It can be implemented using \( n \) linked half-adder circuits.

- to obtain a selectable incrementer replace the constant 1 input with a control input
- time for increment grows with number of bits
Addition Circuit and Full Adders

- Addition circuit with $2n$ inputs & $n+1$ outputs computes the binary sum of two input values.
- It can be implemented using $n$ linked full-adder circuits.
- A full-adder can be built from 2 half-adders.

This addition circuit is called a ripple carry adder:
- takes time proportional to $n$ to add two $n$ bit numbers.
Simulation of Adder Circuit

Functional Simulation (0 gate delays)

Unit Delay Simulation (1 ns delay per gate)
Binary Multiplication

- Binary multiplication is done much like decimal multiplication.
  
  \[
  \begin{array}{c}
  \text{multiplicand} \quad 1101 \\
  \text{multiplier} \quad 1010 \\
  \text{partial products} \\
  \text{product} \quad 10000010
  \end{array}
  \]

- Requires 1 bit multipliers (AND gates) and addition circuits.
Incrementer with Carry Look-ahead

- Can speed up counter using carry lookahead.
- Compute carry out of each position directly from inputs.
  - redundant AND operations, but faster
- Speed comparison
  - assumptions: 2 input gate has 1 ns delay,
    3 or 4 input gate has 2 ns delay,
    5 to 7 input gate has 3 ns delay,
    ...
  - 64 bit ripple carry incrementer needs 64 ns in worst-case
  - 64 bit carry-lookahead incrementer needs 8 ns in worst-case
- So, what’s the catch?
  - carry lookahed uses 2000 “simple gate equivalents”
  - inputs must drive many gates
More Scalable Carry Lookahead

64 bit version has 8 ns delay, about 380 gates for carry, fanout=6.
Ripple carry adder is too slow for fast addition of large values (typical computer uses 32 bit arithmetic).

To get a faster circuit, replace long carry chain with a “shorter” circuit. First separate carry logic in FA.

Let $G_i$ be generate signal for bit $i$, $P_i$ be propagate signal and $C_i$ be carry out of bit $i$.

\[
C_1 = G_1 + P_1 C_0 = G_1 + P_1 G_0 + P_1 P_0 C_{in}
\]

where $C_{in}$ is carry into bit 0; and

\[
C_2 = G_2 + P_2 C_1 \\
= G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_{in}) \\
= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}
\]

and so forth.

So high order carries can be generated with low delay, at the cost of more gates.
Simulation of Carry Lookahead Adder

Functional Simulation (0 gate delays)

Unit Delay Simulation (1 ns delay per gate)
More Scalable Carry Lookahead Adder

- Simple carry lookahead design needs lots of gates to generate high order carries.
  - Number of simple gate equivalents for n bit adder grows in proportion to $n^3$
- For more scalable design, use lookahead idea with groups of bits.
  - $\text{group}_G = G_2 + P_2 G_1 + P_2 P_1 G_0$
  - $\text{group}_P = P_2 P_1 P_0$
  - Time for carry to propagate grows with number of blocks
  - Number of simple gate equivalents grows in proportion to $n^2$ if number of bits per block equals number of blocks
Modular and Signed Arithmetic

- If overflows are discarded, binary adders actually implement modulo arithmetic in which values wrap around circularly.
  - To add $A + B$, start at position for $A$ and then count clockwise $B$ positions.
  - Standard addition algorithm does exactly this.

- Associating certain bit patterns with negative values yields signed arithmetic.

- Negate a given value by flipping all bits and adding 1.

Read MK 145-158
2’s Complement and Subtraction

- In 2’s complement arithmetic with \( n \) bits:
  - the first bit represents the sign (0 for positive, 1 for negative)
  - for positive numbers, the remaining \( n \) bits give the magnitude in standard binary notation
  - to convert a positive number to corresponding negative number, flip all bits and add 1 (0011 → 1100+1=1101)
  - to convert a negative number to corresponding positive number, flip all bits and add 1 (1101 → 0010+1=0011)

- To subtract, take complement and add.
  - \( 4_{10} - 7_{10} = 0100 - 0111 = 0100 + (-0111) = 0100 + 1001 = 1101 = -3_{10} \)

- 2’s complement is most popular method for representing negative numbers.
  - requires no special subtraction circuit, just addition and complement
**Adder-Subtracter**

- **When** $\text{sub}=0$, result is $A + B$.

- **When** $\text{sub}=1$
  - bit flipper complements all bits of $B$
  - adder sums and adds 1
    - $A - B = A + (-B)$
    - $= A + (\neg B + 1)$
    - $= A + \neg B + 1$

- Takes just slightly more time than “plain” adder.
Alternative Negative Number Formats

- In 1’s complement arithmetic, negate a value by flipping bits (do not also add 1).
  - gives two different representations for zero
  - when adding two values, if carry out of most significant digit, increment to obtain final sum
  - comparable to 2’s complement but not quite as simple

- In sign-magnitude arithmetic, left-most bit is sign and remaining bits give magnitude.
  - most obvious representation for people
  - does not allow negative numbers to be directly added
  - requires separate subtraction hardware
Hardware Description Languages

- Hardware Description Languages (HDLs) allow designers to work at a higher level of abstraction than logic gates.
- As with programming languages, HDL descriptions are compiled into a lower level representation:
  - Low level form can be simulated for logical correctness
  - And, can be converted to a circuit specification using a library of primitive components and timing/area constraints
- But don’t confuse hardware design with software:
  - HDL descriptions must reduce to physical hardware that can be fit in the physical space available and meets timing specs.
  - Hardware designs are inherently parallel with many things going on at once
  - On the other hand, software can be used to implement much more complex functions than hardware alone.
VHDL Specification of Half Adder

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity half_adder is
  port ( 
    X: in STD_LOGIC;
    Y: in STD_LOGIC;
    S: out STD_LOGIC; -- sum: X+Y
    C: out STD_LOGIC; -- carry out
  );
end half_adder;

architecture dataflow of half_adder is
begin
  S <= X xor Y;
  C <= X and Y;
end dataflow;
```

- **Library**: The `library` statement imports the `IEEE` library, which provides commonly used types and functions.
- **Port Declaration**: The `port` section defines the inputs and outputs of the `half_adder` entity.
- **Signal Assignments**: The `architecture` section contains signal assignments that occur simultaneously. `STD_LOGIC` is the type used for signals.
- **Built-in Operators**: `xor` and `and` are built-in operators in VHDL.
- **Simulation**: Simulation of VHDL is used to verify correctness. (Could also synthesize circuit to gates, first.)
Generated Schematic for Half-Adder

- Synthesizer makes circuit for FPGA.
  - field programmable gate array
  - Xilinx tools can synthesize to Xilinx FPGAs
- FMAPs are Boolean function generators.
  - can generate any 4-input function
  - here, one does AND, one does XOR
- Remaining gates are built into FPGA cell.
- Generated FPGA schematics of limited use for designers.
library IEEE;
use IEEE.std_logic_1164.all;

entity fulladd is
  port ( 
    a, b, Ci: in STD_LOGIC;
    s, Co: out STD_LOGIC
  );
end fulladd;

architecture fulladd_arch of fulladd is
begin
  s <= (a xor b) xor Ci;
  Co <= (a and b) or (a and Ci) or (b and Ci);
end fulladd_arch;
VHDL for 4 Bit Adder

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity adder4 is
  port ( 
    X, Y: in STD_LOGIC_VECTOR(3 downto 0);
    Ci: in STD_LOGIC;
    S: out STD_LOGIC_VECTOR(3 downto 0);
    Co: out STD_LOGIC
  );
end adder4;

architecture mixed of adder4 is
  component fulladd 
    port (a, b, Ci: in STD_LOGIC; s, Co: out STD_LOGIC);
  end component;
  begin
    b0: fulladd port map(X(0), Y(0), Ci, S(0), Co(0));
    b1: fulladd port map(X(1), Y(1), C(0), S(1), c(1));
    b2: fulladd port map(X(2), Y(2), C(1), S(2), c(2));
    b3: fulladd port map(X(3), Y(3), C(2), S(3), c(3));
    Co <= C(3);
  end mixed;
```

- Vector notation for grouping signals.
- Component definitions required in every architecture using a component.
- Component statement used to form complex circuits from simpler parts.
- Positional association of signals used here. Explicit assignment (a=>X(0)) also allowed.
- Note effect of Ci.
- Note delays.
Key Concepts in VHDL

- VHDL developed for circuit specification & simulation.
  - synthesis tools developed later
  - not all VHDL specifications can be synthesized

- Signals correspond to wires in circuit.

- Signal assignments define logic circuits.
  - signals on left side of assignment change as signals on right side change (exceptions to be discussed later)
  - not like sequential program execution

- Strong typing in VHDL.
  - signal types in expressions must match exactly
    - no automatic type conversions
  - bit and integer are only built-in types
  - extensive support for user-defined types, such as std_logic
  - std_logic defines 9 values, including 0, 1 and undefined
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity arithuv is
port(
a, b: in STD_LOGIC_VECTOR (3 downto 0);
c: in STD_LOGIC_VECTOR (2 downto 0);
x: out STD_LOGIC_VECTOR (3 downto 0);
v: out STD_LOGIC);
end;

architecture arithuv_arch of arithuv is

signal result: STD_LOGIC_VECTOR (3 downto 0);
signal ax, bx: STD_LOGIC_VECTOR (3 downto 0);

begin
ax <= '0' & a; -- put a leading zero on a
bx <= '0' & b; -- put a leading zero on b
result <= ax when c = '000' else bx when c = '001' else (not ax)+1 when c = '010' else (not bx)+1 when c = '011' else ax+bx when c = '100' else ax-bx when c = '101' else ax-bx when c = '110' else bx-ax;
v <= result(3 downto 0);
v <= '1' when (c = "010" and a="1000") or (c = "011" and b="1000") or (c = "100" and result(3) = '1') or (c = "101" and a(3)=b(3) and a(3)=result(3)) or (c = "110" and a(3)=b(3) and a(3)=result(3)) or (c = "111" and a(3)=b(3) and b(3)=result(3)) else '0';
end arithuv_arch;

V bit indicates arithmetic overflow

c=0 means x=a,  c=1 means x=b,  c=2 means x=-a,  c=3 means x=-b,  
c=4 means x=a+b (unsigned),  c=5 means x=a+b (signed),  
c=6 means x=a-b,  c=7 means x=b-a