COMP303 - Computer Architecture

Combinational Logic & Computer Arithmetic Review
Combinational Logic Review

AND gate

\[
\begin{array}{c|c|c}
A & B & F \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

NAND gate

\[
\begin{array}{c|c|c}
A & B & F \\
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

OR gate

\[
\begin{array}{c|c|c}
A & B & F \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

NOR gate

\[
\begin{array}{c|c|c}
A & B & F \\
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\end{array}
\]

NOT gate

\[
\begin{array}{c|c}
A & F \\
0 & 1 \\
1 & 0 \\
\end{array}
\]

XOR gate

\[
\begin{array}{c|c|c}
A & B & F \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

Logic symbol

Truth table
**Combinational Logic Review**

- **Input = 4-bit number**
- **Output = 1 if primary, 0 otherwise**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Karnaugh Map**

\[ BC'D + A'BD + A'B'C + B'CD \]
Shift Operations

The MIPS architecture defines various shift operations:

(a) `sll r1, r2, 3` 
   `r2 = 10101100` (shift left logical) 
   `r1 = 01100000` 
   - shift in zeros to the least significant bits

(b) `srl r1, r2, 3` 
   `r2 = 10101100` (shift right logical) 
   `r1 = 00010101` 
   - shift in zeros to the most significant bits

(c) `sra r1, r2, 3` 
   `r2 = 10101100` (shift right arithmetic) 
   `r1 = 11110101` 
   - copy the sign bit to the most significant bits

There are also versions of these instructions that take three register operands.
Logical Operations

In the MIPS architecture logical operations (and, or, xor) correspond to bit-wise operations.

(a) and r1, r2, r3    r3 = 1010 (r1 is 1 if r2 and r3 are both one)
                      r2 = 0110
                      r1 = 0010

(b) or  r1, r2, r3    r3 = 1010 (r1 is 1 if r2 or r3 is one)
                     r2 = 0110
                     r1 = 1110

(c) xor r1, r2, r3    r3 = 1010 (r1 is 1 if r2 and r3 are different)
                     r2 = 0110
                     r1 = 1100

Immediate versions of these instructions are also supported.
Two’s Complement Negation

To negate a two's complement integer, invert all the bits and add a one to the least significant bit.

What are the two’s complements of

\[
\begin{align*}
6 &= 0110 & \rightarrow & & 1001 \\
& & + & 1 & \rightarrow & 1010 = -6
\end{align*}
\]

\[
\begin{align*}
-4 &= 1100 & \rightarrow & & 0011 \\
& & + & 1 & \rightarrow & 0100 = 4
\end{align*}
\]
Two's Complement Subtraction

- To subtract two's complement numbers we first negate the second number and then add the corresponding bits of both numbers.

- \[ A - B = A + (2^n - B) \]

- For example:
  
  \[
  3 = 0011 \\
  -2 = 0010 \\
  1 = 0001
  \]
Overflow

- When adding or subtracting numbers, the sum or difference can go beyond the range of representable numbers.

- This is known as overflow. For example, for two's complement numbers,

  \[ 5 = 0101 \quad -5 = 1011 \]
  \[ +6 = 0110 \quad +(-6) = 1010 \]

  \[ \text{--} \quad \text{--} \]

  \[ -5 = 1011 \quad 5 = 0101 \]

- Overflow creates an incorrect result that should be detected.
When adding two's complement numbers, overflow will only occur if
- the numbers being added have the same sign
- the sign of the result is different

If we perform the addition
\[
\begin{array}{c}
\begin{array}{cccccccc}
  a_{n-1} & a_{n-2} & \ldots & a_1 & a_0 \\
  + & b_{n-1} & b_{n-2} & \ldots & b_1 & b_0 \\
\end{array}
\end{array}
\]
\[
\begin{array}{c}
\begin{array}{cccccccc}
  \hline
  s_{n-1} & s_{n-2} & \ldots & s_1 & s_0 \\
\end{array}
\end{array}
\]
Overflow can be detected as
\[
V = a_{n-1} \cdot b_{n-1} \cdot s_{n-1} + \overline{a_{n-1}} \cdot \overline{b_{n-1}} \cdot s_{n-1}
\]
Overflow can also be detected as
\[
V = c_n \otimes c_{n-1}, \text{ where } c_{n-1} \text{ and } c_n \text{ are the carry in and carry out of the most significant bit.}
\]
A fundamental building block in the ALU is a full adder (FA).

A FA performs a one bit addition.

\[ a_i + b_i + c_i = c_{i+1} s_i \]
Full Adder Logic Equations

- $s_i$ is ‘1’ if an odd number of inputs are ‘1’.
- $c_{i+1}$ is ‘1’ if two or more inputs are ‘1’.

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>$b_i$</th>
<th>$c_i$</th>
<th>$c_{i+1}$</th>
<th>$s_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

$s_i = a_i \overline{b_i} c_i + a_i b_i \overline{c_i} + a_i \overline{b_i} \overline{c_i} + a_i b_i c_i$

$s_i = a_i \otimes b_i \otimes c_i$

$c_{i+1} = a_i b_i c_i + a_i \overline{b_i} c_i + a_i b_i \overline{c_i} + a_i b_i c_i$

$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$

$c_{i+1} = a_i b_i + c_i (a_i + b_i)$

$c_{i+1} = a_i b_i + c_i (a_i \otimes b_i)$
Larger Adders

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]
One possible implementation of a full adder uses nine gates.

\[ s_i = a_i \otimes b_i \otimes c_i \]
\[ c_i + 1 = a_i b_i + c_i (a_i \otimes b_i) \]
\[ a_i \otimes b_i = (a_i + b_i) a_i b_i \]
We will be designing a 1-bit ALU with the following interface.

- Z = 1, if Result = 0
- V = 1, if Overflow
- \( C_1 = 1 \), if Carry-Out

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND</td>
</tr>
<tr>
<td>001</td>
<td>OR</td>
</tr>
<tr>
<td>010</td>
<td>ADD</td>
</tr>
<tr>
<td>110</td>
<td>SUBTRACT</td>
</tr>
<tr>
<td>111</td>
<td>XOR</td>
</tr>
</tbody>
</table>
The full adder, an xor gate, and a 4-to-1 mux are combined to form a 1-bit ALU.

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>AND</td>
</tr>
<tr>
<td>001</td>
<td>OR</td>
</tr>
<tr>
<td>010</td>
<td>ADD</td>
</tr>
<tr>
<td>110</td>
<td>SUBTRACT</td>
</tr>
<tr>
<td>111</td>
<td>XOR</td>
</tr>
</tbody>
</table>
The ALU for the MSB must also detect overflow and indicate the sign of the result.

\[ V = c_n \oplus c_{n-1} \]

\[ set = (A < B) \]
Fig 9-1. 2-to-1 Multiplexer and Switch Analog

logic equation for the 2 - to - 1 MUX

\[ Z = A' I_0 + A I_1 \]
logic equation for the 8 - to - 1 MUX

\[ Z = A' B' C' I_0 + A' B' CI_1 + A' BC' I_2 + A' BCI_3 \\
+ AB' C' I_4 + AB' CI_5 + ABC' I_6 + ABCI_7 \]
Multiplexers

Fig 9-3. Logic Diagram for 8-to-1 MUX
Assembly Example

.data
str1: .asciiz "\nEnter a number for summation:"
str2: .asciiz "Sum of numbers entered = 

.text
.globl main
main:
    li $t0, 0
loop:
    li $v0, 4       # system call code for print_str
    la $a0, str1    # address of string to print
    syscall
    li $v0, 5       # system call code for read_int
    syscall         # read int
    add $t0, $t0, $v0
    bne $v0, $zero, loop
    li $v0, 4       # system call code for print_str
    la $a0, str2    # address of string to print
    syscall
    li $v0, 1       # system call code for print_int
    move $a0, $t0   # move the result in $a0
    syscall         # print it