A crash course in Digital Logic

Computer Architecture

1DT016 distance
Fall 2017

http://xyx.se/1DT016/index.php

Per Foyer
Mail: per.foyer@it.uu.se
We start from here...

Gates
Flip-flops
Multiplexers (MUX)
Demultiplexers (DEMUX)
Registers
Latches
Shift registers
RAM
ROM
...

Per.Foyer@it.uu.se 2017
... and get here!

The MIPS CPU
int addmul( int t )
{
    return (t + 2) * 2;
}

addmul: addi $r1, $zero, 2
    mul  $r1, $r1, 2
    jr   $ra

li $v0, 4
syscall

0x24020004 0x0000000c
0x03E00008

0x24020004 0x0000000c
0x03E00008

110110101111010000010110
000100010011111010100001

11011010101110100000101110110111010010111111010100001
Binary representation

Arbitrary numbers can be represented in binary format.

\[ x = 2^n - 1 \] where \( n \) is the number of bits and \( x \) the maximum (positive) decimal number that can be represented.

So:

Byte (8 bits): \[ 2^8 - 1 = 0 \text{ to } 255 \]
Half word (16 bits): \[ 2^{16} - 1 = 0 \text{ to } 65.535 \]
Word (32 bits): \[ 2^{32} - 1 = 0 \text{ to } 4.294.967.295 \]

Bits are numbered from right to left.
One’s complement:

The most significant bit used as sign bit (negative number when set):

![Binary representation](image)

This gives two representations of zero: -0 and +0 which is a problem in arithmetic calculations.

Example:

- 00000000 = 10000000 = 0
- ... so “0” + “0” = 127 → 0 ≠ 127
- ... which is seriously wrong.
Negative binary numbers (2)

Two’s complement:

\[-m_2 = \text{inv}(|n_2|) + 1\]

...where \(n\) is the (negative) number and \(\text{inv}()\) is bitwise invert for a specific number of bits.

Example:
Convert \(3\) to negative in 8 bit representation:
\[\text{inv}(00000011) + 1 \Rightarrow 11111100 + 1 = 11111101\]

Asymmetric representation!

Examples:

For 8 bits: \(-128\) to \(127\)
For 16 bits: \(-32.768\) to \(32.767\)

We can still see that a number is positive or negative by inspecting the most significant bit.
Binary arithmetic

Example:

69\textsubscript{10} and 12\textsubscript{10} are to be added binary:

\[
\begin{array}{cccccccccccc}
  &  &  &  &  &  &  &  &  &  &  &  & 1 & 1 \\
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0100 & 0101 & \text{Carry Row} \\
+ 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 1100 & \text{(69)} \\
\hline
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0101 & 0001 & \text{(81)} \\
\end{array}
\]

Example:

12\textsubscript{10} is to be subtracted from 69\textsubscript{10} using add \( \rightarrow \) 69+(-12)

Use two’s complement:

Convert to binary: 0000 0000 0000 0000 0000 0000 0000 1100 \( \text{(12)} \)
Invert: 1111 1111 1111 1111 1111 1111 1111 0011
Add one: 1111 1111 1111 1111 1111 1111 1111 0100 \( \text{(-12)} \)

So:

\[
\begin{array}{cccccccccccc}
  &  &  &  &  &  &  &  &  &  &  &  & 1 & 1 \\
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0100 & 0101 & \text{Carry Row} \\
+ 1111 & 1111 & 1111 & 1111 & 1111 & 1111 & 1111 & 0100 & \text{(69)} \\
\hline
0000 & 0000 & 0000 & 0000 & 0000 & 0000 & 0011 & 1001 & \text{(57)} \\
\end{array}
\]
Boolean algebra (1)

The algebraic logic behind gate logic.

Three operators:

- **OR** – logical sum, written +
- **AND** – logical product, written *
- **NOT** – logical negation (inversion)

Let A and B be two inputs and R output. Then:

- R is true only if A is true AND B is true ( \( R = A \times B \) )
- R is true if A is true OR B is true ( \( R = A + B \) )
- R is true only if A is NOT true ( \( R = \overline{A} \) )
There are several laws of Boolean algebra that are helpful in manipulating logic equations.

Identity law: \( A + 0 = A \) and \( A * 1 = A \)

Zero and One laws: \( A + 1 = 1 \) and \( A * 0 = 0 \)

Inverse laws: \( \overline{A} + A = 1 \) and \( \overline{A} * A = 0 \)

Commutative laws: \( A + B = B + A \) and \( A * B = B * A \)

Associative laws: \( A + (B + C) = (A + B) + C \)
and \( A * (B * C) = (A * B) * C \)

Distributive laws: \( A * (B + C) = (A * B) + (A * C) \)
and \( A + (B * C) = (A + B) * (A + C) \)

Remember: **AND** is *, **OR** is +
## Numbers in different bases

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
</tbody>
</table>
Buffers and inverters

In = Out with more current

In = Out with more current if E is true. Else Hi-Z

A = NOT A

A = NOT A if E is true. Else Hi-Z

Hi stands for High (voltage)
Lo stands for Low voltage (often) zero volts
Hi-Z means High Impedance (Z)
OR and AND

\[
\begin{array}{ccc}
A & B & R \\
\hline
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

(\( R = A + B \))

\[
\begin{array}{ccc}
A & B & R \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

(\( R = A \cdot B \))
NOR and NAND

\[ R = \overline{A + B} \]

\[ R = \overline{A \cdot B} \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR and XNOR

**XOR**

- eXclusive OR – True if only one of A and B is true but not both

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**XNOR**

- eXclusive NOR – The opposite of the above.

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

*(XNOR is very seldom used)*
Negative logic

It’s sometimes easier and more convenient to invert inputs than rewriting logic using boolean algebra.

\[
\overline{A} \quad \overline{B} \quad R = \quad A \quad \overline{B} \quad \overline{R}
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
1-bit full adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C\text{IN}</th>
<th>R</th>
<th>C\text{OUT}</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>1</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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2-bit adder?

Right or **Wrong**?
Square waves

![Diagram of square waves with labels for rising and falling edges, logic levels (1 and 0), duty cycle, and period (t).]

- Rising edge
- Falling edge
- Logic 1
- Logic 0
- Duty cycle
- Period (t)

\[ f = \frac{1}{t} \]
Demo: How to use LogSim

Our first digital playground:

• Testing simple gates
• Toying with a four bit counter
Buses

Bus is a number of parallel lines (bus width), often binary, that has something in common.

Examples:

• Data bus \((1, 2, 4, 8, 16, 32, 64, \ldots \ 2^n \text{ bits wide})\)
• Address bus \((1, 2, 4, \ldots, 2^n \text{ bits wide})\)
• Control bus \((1, 2, 4, \ldots, 2^n \text{ bits wide})\)
The Arithmetic Logic Unit (ALU)

Input devices → Control unit → Arithmetic logic unit → CPU → Output devices

Arithmetic
- A + B
- A - B
- A * B
- A / B
- A < B
- A == B

Logic
- A AND B
- A OR B
- A XOR B
module MIPSALU (ALUctrl, A, B, ALUOut, Zero);
    input [3:0] ALUctrl;
    input [31:0] A,B;
    output reg [31:0] ALUOut;
    output Zero;
    assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0
always @(ALUctrl, A, B) begin //reevaluate if these change
    case (ALUctrl)
        0: ALUOut <= A & B;
        1: ALUOut <= A | B;
        2: ALUOut <= A + B;
        6: ALUOut <= A - B;
        7: ALUOut <= A < B ? 1 : 0;
        12: ALUOut <= ~(A | B); // result is nor
    default: ALUOut <= 0;
    endcase
end
endmodule

Look at the picture.
What’s wrong with the Verilog code?
LogSim – A 4-bit ALU

Exterior design view

Status flags (out)

Operand 1 (in) → Result (out)
Operand 2 (in)

ALU function (in)
Demo: LogSim: 4-bit ALU

Testing the ALU
Finito la musica!