Digital Building Blocks

CS31
Pascal Van Hentenryck
Overview

Digital Building Blocks

• Decoders and Multiplexers
• ALU (arithmetic and logic unit)
• ROM

ALU

• This is the brawn of the computer

ROM

• Useful for implementing the control
The Big Picture

compiler

computer

data path

control

input

output
Abstraction Hierarchy

- Programming Language
- Assembly Language
- Machine Language
- Sequential Circuit
- Combinational Circuit
- Binary Value
- Voltage
Two Building Blocks

Decoders

- \( n \) inputs / \( 2^n \) outputs
- Given the inputs, select a unique output
- Particularly useful for implementing memories

Multiplexers (selectors)

- \( 2^n \) data inputs / \( n \) selection inputs / 1 output
- The output is the data input selected by the selection input
- Particularly useful for implementing ALU and various other pieces of the machine
Decoders

Turn binary “coded” quantities into one unit vector for every possible value.
Decoders

Turn binary “coded” quantities into one unit vector for every possible value.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>d₀</th>
<th>d₁</th>
<th>d₂</th>
<th>d₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Look at \((x, y)\) as a binary number

The decoder
- assign \(d_{2^x+y}\) to 1
- all the others to zero
Decoders

Turn binary “coded” quantities into one unit vector for every possible value.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>d₀</th>
<th>d₁</th>
<th>d₂</th>
<th>d₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

![Diagram of a decoder circuit]
More Decoders

- Decoders can be in any size

\[ n \text{ is always } 2^m \]

- Two types of decoders: a single output 1 or a single output is 0
Multiplexers

Choose a particular input to pass through as specified by values on the selection (or address) lines.

Essentially a switch in hardware

Output = $l_{2x+y}$
Multiplexers

Choose a particular input to pass through as specified by values on the selection (or address) lines.

\[
\begin{array}{c|c|c|c}
S_1 & S_2 & O \\
\hline
0 & 0 & I_0 \\
0 & 1 & I_1 \\
1 & 0 & I_2 \\
1 & 1 & I_3 \\
\end{array}
\]
Multiplexers

Choose a particular input to pass through as specified by values on the selection (or address) lines.

What is inside a multiplexer?
• can you see it?
Multiplexers

Choose a particular input to pass through as specified by values on the selection (or address) lines.

What is inside a multiplexer?
- can you see it?
1-bit ALU

1-bit ALU with **and** and **or**

Add addition to this ALU now.
1-bit ALU
32-bit ALU

Carry-in     Operation

a_0
a_1
a_2
......
a_{31}
b_0
b_1
b_2
......
b_{31}

ALU

r_0
r_1
r_2
......
r_{31}

Carry-out
32-bit ALU

Operation

CarryIn

Alu0

Alu1

Alu31

a0
b0
a1
b1
a31
b31
32-bit ALU

Include subtraction
  • Add the negative version of \( b \)

How to obtain the negative version
  • Invert each bit
  • Add 1

How do to that simply
  • Generalize a bit
  • Use a simple trick

This shows why 2’s complement is a good representation
1-bit ALU (Revisited)
32-bit ALU

binvert \quad CarryIn \quad Operation

\begin{align*}
a_0 & \quad \rightarrow & \quad Alu0 \\
b_0 & \quad \rightarrow & \\
a_1 & \quad \rightarrow & \quad Alu1 \\
b_1 & \quad \rightarrow & \\
a_{31} & \quad \rightarrow & \quad Alu31 \\
b_{31} & \quad \rightarrow & \\
\end{align*}
ALU (Absolutely Lazy Unit)

Possible Modes

Operation 0: \( R = A + B \)
Operation 1: \( R = A - B \)
Operation 2: \( R = A \text{ and } B \)
Operation 3: \( R = A \text{ or } B \)
## Controlling the ALU

<table>
<thead>
<tr>
<th></th>
<th>$O_1$</th>
<th>$O_0$</th>
<th>$B_{\text{inv}}$</th>
<th>$C_{\text{in}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A + B$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$A - B$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$A \wedge B$</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>$A \vee B$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>

(Look Ma! I can build an ALU)
Implementing a Logic Function

Three main possibilities

- Specific circuit
- ROM
- PLA

Specific Circuit

- when there is a lot of structure (the Karnaugh map, remember?)

ROM

- when there is almost no structure

PLA

- in between
Read Only Memory (ROM)

Map addresses to fixed values

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a&lt;sub&gt;1&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>V&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>V&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>V&lt;sub&gt;3&lt;/sub&gt;</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>V&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>V&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>V&lt;sub&gt;6&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>V&lt;sub&gt;7&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>V&lt;sub&gt;8&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

Diagram of the 3x8 decoder and OR gate implementation.
Modern ROMs

Local outputs are associated with each pulldown

The local outputs are equal to their corresponding inputs

The output is the logical and of the local outputs
Modern ROMs

Modern ROMs are organized around

- a decoder
- a multiplexer
- an array of pulldowns

![Diagram of Modern ROMs]

- \( n \) address bits
- \( 2^n \times 2^m \) array of pulldowns
- \( m \) address bits
- Output
Example (stupid, I agree)

Assume that I need a ROM to store the prime numbers from 0 to 63.

Given an integer (between 0 and 63), build a circuit that returns 1 if the number is prime and 0 otherwise.

The integer is given as a sequence of bits

\[ A_5 \ A_4 \ A_3 \ A_2 \ A_1 \ A_0 \]

We use an array of 8 by 8
Storing Primes

Put a pulldown where you want a 0.

If you want more than one bit of output, just use more ROM circuits in parallel with different stored functions.
Not Chaining Adders

Can we speed this up?

\[ \begin{align*}
0 & \quad \rightarrow \quad S_0 \\
x_0 & \quad \rightarrow \quad FA \\
y_0 & \quad \rightarrow \quad FA \\
0 & \quad \rightarrow \quad S_1 \\
x_1 & \quad \rightarrow \quad FA \\
y_1 & \quad \rightarrow \quad FA \\
0 & \quad \rightarrow \quad S_2 \\
x_2 & \quad \rightarrow \quad FA \\
y_2 & \quad \rightarrow \quad FA \\
0 & \quad \rightarrow \quad S_3 \\
x_3 & \quad \rightarrow \quad FA \\
y_3 & \quad \rightarrow \quad FA \\
0 & \quad \rightarrow \quad C
\end{align*} \]