Critical Path

- Each logic gate has a propagation delay. This is the delay between when the inputs are applied and the correct logical output is available.
- The critical path of a circuit is the path between the input values and the output value which has the longest delay.

- To calculate the critical path assume that each input value was available at time 0. For each gate which the input values propagate through, calculate the time when the output will be available. For each internal gate, take the input value with the largest delay and add the gate delay to produce the output time. Continue for all gates, until the output.

- The critical path delay time is the length of time for the output value. The critical path is the set of gates which dictate this longest time. If there is a tie between inputs for the longest delay, both inputs are required in the critical path.
**Multiplexor**

- Multiplexors are selectors which, based on the value of the ‘selector’ bits, the corresponding input is placed on the output line.

- Multiplexors come in a variety of sizes, 2-input, 4-input, 8-input, *etc.* $n$ inputs means $\lfloor \log_2 n \rfloor$ ‘selector’, or ‘control’, bits.

- **Ex:** 2-input multiplexor.
  - In this case, the inputs $A$ & $B$ are both 1-bit, the selector input $S$ is 1-bit, and the output $C$ is 1-bit.
  - If the selector $S$ has value 0, then $C = A$. If $S = 1$, then $C = B$.
  - This function can be created using the AND–OR gate network shown on the right. Note that the Boolean expression for a multiplexor is $C = S' \ A + S \ B$.

- **Ex:** 8-input multiplexor

- **Ex:** A multiplexor that selects between a pair of 32-bit buses, built as an array of 1-bit-wide multiplexors
a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit-wide multiplexors

A multiplexor is arrayed 32 times to perform a selection between two 32-bit inputs.

Note that there is still only one data selection signal used for all 32 1-bit multiplexors.

**Side Discussion - Universal (Complete) Sets**

- A set of gates is *complete/universal* if you can implement *any* logical function using only the types of gates in the set.
  - You can use as many gates as you want to implement the function.
- *Minimal complete set* is a complete set with no redundant elements.
- Some examples of complete sets:
  - \{AND, OR, NOT\} – this is a complete set, but not a *minimal* complete set.
  - \{AND, NOT\}
  - \{OR, NOT\}
  - \{NAND\}
  - \{NOR\}

- **Ex:** Proof that NAND is universal by building gates that form a universal set.
  - It is sufficient to build a NOT gate and either an OR or an AND gate.

- **Ex:** Proof that NOR is universal by building gates that form a universal set.
- It is sufficient to build a NOT gate and either an OR or an AND gate.

Implementing Gate Networks with Multiplexors

- Multiplexors are a minimal complete set \{MUX\} and therefore can be used to implement any Boolean expression/function.
- To do so, a single or set of the input variables are chosen as the selectors for the multiplexors.
  - This results in factoring out the selector input variables from all terms in the Boolean expression.
  - The remaining terms are then inputted into the MUX at the corresponding inputs.
  - Note: Any term which does not have the selector variables, must be placed on all inputs. Why? Because the selector variables are “hidden” (or multiplied by 1) for the selector variables.

- Ex: Implement the function \( F = abc' + a'bc' + a'b'c' + bc \) using 2-input MUXes

  **Step 1:** Select a single variable as the selector (2-input MUXes) and factor the variable out of all terms
  \[
  F = a (bc' + bc) + a' (bc' + b'c' + bc)
  \]
  **Note:** term \( bc \) must appear in both factorings. Why? Because term \( bc \) is really \( bc(a + a') \)

  **Step 2:** For each of the remaining sub boolean expressions, ie expressions in parentheses, additional MUXes must be used and the process is repeated. If the expression is only a single variable or constant value 0 or 1, additional MUXes are not required.
  **Note:** If complemented forms of variables are not given, then NOT gates must be built using the MUXes to invert the uncomplemented variable.
  **Note:** Simplify expression whenever possible to reduce the number of required gates

  \[
  G = bc' + bc = b
  \]
  \[
  H = bc' + b'c' + bc = c' + bc
  \]
  For \( H \) we select from variables \( b \) or \( c \) to be the selector.
  \[
  H = c'(1) + c (b)
  \]

  The gate network is:
Note: selecting different variables as the selectors will result in different networks and possibly different number of required gates. By selecting a variable which appears in most terms first and variables which appear in close to equal number of complemented and uncomplemented forms, you will result in a smaller implementation and possible a shorter critical path (to be discussed later).

- Ex: Implement the function \( F = abc' + a'bc' + a'b'c' + bc \) using 4-input MUXes
  
  Step 1: Select two variables as the selector (4-input MUX) and factor the variable out of all terms
  
  \[ F = abc' + a'bc' + a'b'c' + bc \ (a + a') \]
  
  Remember \( bc \) also contains both forms of \( a \). Therefore must be included in both \( ab \) and \( a'b \) terms
  
  \[ F = ab (c'+c) + a'b(c'+c) + ab' (0) + a'b'(c') \]
  
  Note: we have no terms containing \( ab' \) therefore the input to this combination should be 0
  
  \[ F = ab (1) + a'b(1) + ab' (0) + a'b'(c') \]
  
  All sub expressions are either constant values or a single variable. DONE.
  
  The gate network is:

Decoder

- Logical device which has \( n \) input bits, \( 2^n \) output bits.
- Only 1 of the output bits is 1 for each input combination.
- Selected output corresponds to the binary value of its input; \( i.e., \) if input is 3-bits with binary value of \( \{0,1,1\} \) then there are 8 output lines with binary value of \( \{0,0,0,1,0,0,0,0\} \).
Ex: 2-to-4 single bit decoder implementation

- In essence, a multiplexor is actually a decoder plus some extra gates.
  - Ex: 4 input multiplexor

```plaintext
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

```
Ex: 4 input multiplexor
```

```plaintext
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>O_0</td>
<td>O_1</td>
<td>O_2</td>
<td>O_3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

Minterm Equations

- D_0 = \overline{A}_1 \cdot \overline{A}_0
- D_1 = \overline{A}_1 \cdot A_0
- D_2 = A_1 \cdot \overline{A}_0
- D_3 = A_1 \cdot A_0