Outline

1. Administration
2. Implementation technologies
3. Scripting languages
4. Review of implementation technologies
5. Homework
Today’s topics

- Can use today’s class for Q&A
- PALs/PLAs
- Review, Q&A on MOS transistors
- Multiplexers, Demultiplexers
- Transmission gates
- Perl/Python
Lab two

- Lab two is more substantial than lab one
- If you find a problem and figure out the solution, yourself, please send me an email or post to the newsgroup
- Don’t think you’ll be able to fully understand the effects all the SIS commands, options, and sequences will have
- If you have a basic understanding of how to get reasonably good results with the software, that’s good
Midterm exam

Suggesting 21 or 23 October
Outline

1. Administration
2. Implementation technologies
3. Scripting languages
4. Review of implementation technologies
5. Homework
Section outline

2. Implementation technologies
   PALs and PLAs
   CMOS for logic gates
   Transmission gates and MUXs
Programmable arrays of logic gates

- We have considered implementing Boolean functions using discrete logic gates
  - NOT, AND, OR, NAND, NOR, XOR, and XNOR
- Can arrange AND and OR gates (or NAND and NOR gates) into a general array structure
- Program array to implement logic functions
- Two popular variants
  - Programmable logic arrays (PLA) and programmable array logic (PAL)
PALs and PLAs

- Pre-fabricated building block of many AND and OR (or NAND and NOR) gates
- “Personalized” (programmed) by making or breaking connections among the gates
SOP programmable array block diagram

- Inputs
- Dense array of ANDs
- Product terms
- Dense array of ORs
- Outputs
PLAs can share terms – Share product terms

Consider the following set of functions

\[ f_0 = a + \overline{b} \overline{c} \]
\[ f_1 = a\overline{c} + ab \]
\[ f_2 = \overline{b} \overline{c} + ab \]
\[ f_3 = \overline{b} \overline{c} + a \]

Personality matrix

<table>
<thead>
<tr>
<th>Product term</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ab )</td>
<td>1 1 X</td>
<td>0 1 1 0</td>
</tr>
<tr>
<td>( \overline{b} \overline{c} )</td>
<td>X 0 1</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>( a\overline{c} )</td>
<td>1 X 1</td>
<td>0 1 0 0</td>
</tr>
<tr>
<td>( \overline{b} \overline{c} )</td>
<td>X 0 0</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>( a )</td>
<td>1 X X</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>
PLAs efficiency

- PLAs can share terms – *Share product terms*
- Consider the following set of functions

$$f_0 = a + \overline{b} \overline{c}$$
$$f_1 = a\overline{c} + ab$$
$$f_2 = \overline{b} \overline{c} + ab$$
$$f_3 = \overline{b} \overline{c} + a$$

<table>
<thead>
<tr>
<th>Product term</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_0$</td>
<td>$f_1$</td>
<td>$f_2$</td>
</tr>
<tr>
<td>$a \overline{b}$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\overline{b} c$</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>$a \overline{c}$</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>$\overline{b} \overline{c}$</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>$a$</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>
PLA programming

- All connections available
  - All exist
  - Some removed
- None exist
  - Connections made
PLA programming
PLA programming
PLA programming

\[
\begin{align*}
ab \\
abc d
\end{align*}
\]
PLA programming

Diagram showing the PLA programming with inputs a, b, c, d and outputs for expressions ab, abc’d, bc’d.
PLA programming
PLA programming

\[ \overline{a}b + b\overline{c}d \]
PLA programming

\[ \overline{ab} + \overline{bcd} \]
PLA programming

\[
\begin{align*}
\overline{a}b & \quad \overline{a}bc \quad \overline{a}b \overline{c} \quad \overline{a}bcd \\
b \overline{c}d & \quad b \overline{c}d \\
a \overline{b} \overline{c}d & \quad ab \overline{c} \overline{d} \\
a \overline{b} \overline{c} \overline{d} & \quad ab \overline{c}d
\end{align*}
\]
PLA programming
PLA programming

\begin{align*}
\overline{ab} + \overline{b\overline{c}d} = & \\
\overline{a\overline{b}c\overline{d}} = & \\
b\overline{c}d + a\overline{b}c\overline{d} = & \\
\overline{ab} = & \\
abc\overline{d} + bc\overline{d} = & 
\end{align*}
PLA diagram shorthand

\[ \overline{ab} + \overline{b\overline{c}d} \]

\[ \overline{a\overline{b}c\overline{d}} \]

\[ bc\overline{d} + \overline{a\overline{b}c\overline{d}} \]

\[ \overline{ab} \]

\[ ab\overline{c}d + b\overline{c}d \]
Shorthand – Draw subset of wires

\[ \begin{align*}
ab &+ bc \cdot d \\
\overline{a}b &+ ab \cdot c \cdot d
\end{align*} \]
PAL/PLA differences

**PAL**
- Only the AND array is programmable
- A column of the OR array only has access to a subset of the product terms
- Generally, no sharing of product terms

**PLA**
- A column has access to any desired product terms
- Can share product terms
PAL/PLA differences
PAL/PLA differences

[Diagram showing PAL and PLA circuit, with inputs a, b, c, d and output logic gates with red boxes indicating differences.]
## BCD-Gray code converter

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<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>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
BCD-Gray code converter
Minimized BCD-Gray functions

\[ W = A + BD + BC \]
\[ X = B \overline{C} \]
\[ Y = B + C \]
\[ Z = \overline{A} \overline{B} \overline{C} D + BCD + \overline{A}D + \overline{B} CD \]
BCD-Gray discrete logic

(1) 7404 six inverter
(2, 5) 7400 four NAND2
(3) 7410 three NAND3
(4) 7420 two NAND4
BCD-Gray PAL
Comparator example

Determine whether the first two-bit number (AB) is
- Equal to (EQ),
- Not equal to (NE),
- Less than (LT),
- Or greater than (GT)

a second two-bit number (CD)
Comparator Karnaugh map

EQ:

NE:

LT:

GT:
Comparator PLA

EQ, NE, LT, GT
Section outline

2. Implementation technologies
   PALs and PLAs
   CMOS for logic gates
   Transmission gates and MUXs
Transistors

- Basic device in NMOS and PMOS (CMOS) technologies
- Can be used to construct any logic gate
NMOS transistor

source (N)  oxide  drain (N)
silicon bulk (P)
gate
NMOS transistor
NMOS transistor

Metal–oxide semiconductor (MOS)

- Then, it was polysilicon–oxide semiconductor
- Now, it is MOS again

P-type bulk silicon doped with positively charged ions

N-type diffusion regions doped with negatively charged ions

Gate can be used to pull a few electrons near the oxide
Forms channel region, conduction from source to drain starts
CMOS

- NMOS turns on when the gate is high
- PMOS just like NMOS, with N and P regions swapped
- PMOS turns on when the gate is low
- NMOS good at conducting low (0s)
- PMOS good at conducting high (1s)
- Use NMOS and PMOS transistors together to build circuits
  - Complementary metal oxide silicon (CMOS)
CMOS NAND gate

- **A**
- **B**
- **Z**
- **V_{DD}**
- **V_{SS}**
CMOS NAND gate

- PMOS
- NMOS

- Pull-up network
- Pull-down network
CMOS NAND gate

[Diagram of CMOS NAND gate with labels for VDD, VSS, A, B, Z, PMOS, and NMOS]
CMOS NAND gate

pull-up network

pull-down network

A B Z

V_{SS} V_{DD}
CMOS NAND gate layout
CMOS inverter operation

A B
V_{DD}
A=? B=?
CMOS inverter operation

A = 0

A = 1

B = 0

B = 1

A = ?

B = ?
CMOS inverter operation
CMOS inverter operation

A=1
B=0

A=0
B=1

A=?
B=?
CMOS inverter operation
CMOS inverter operation
NAND operation

- **A = 0**: Output is high (1)
- **B = 0**: Output is high (1)
- **Z = 1**: Output is high (1)

- **A = 0** and **B = 0** result in a high (1) output, indicating a NAND operation.
NAND operation
NAND operation

A = 0, B = 0 → Z = 1

A = 0, B = 1 → Z = 0, blocked

A = 1, B = 0 → Z = 0, blocked

A = 1, B = 1 → Z = 0
NAND operation

A B

\( \overline{A \cdot B} \)

\( \overline{0 \cdot 0} = 1 \)

\( \overline{0 \cdot 1} = 1 \)

\( \overline{1 \cdot 0} = 1 \)

\( \overline{1 \cdot 1} = 0 \)
NAND operation

A = 0
B = 1
Z = 0
VSS
VDD
A B
Z
blocked

A = 1
B = 0
Z = 1
VSS
VDD
A B
Z

A = 1
B = 0
Z = 1
VSS
VDD
A B
Z

A = 1
B = 1
Z = 1
VSS
VDD
A B
Z

A = 0
B = 1
Z = 1
VSS
VDD
A B
Z
NAND operation

A: 0, Z = 1
B: 0, Z = 1

A: 1, Z = 0
B: 1, Z = 0

A: 1, Z = 0
B: 1, Z = 0
NAND operation

A = 1
B = 0
Z = 1

A = 0
B = 1
Z

V_{SS}  V_{DD}
NAND operation

![NAND gate diagram]

- **A**: Input 1
- **B**: Input 2
- **Z**: Output
- **V_{DD}**: Power supply high
- **V_{SS}**: Power supply low

### NAND Operation
- **Z = 1** when **A = 0** and **B = 0**
- **Z = 0** when **A = 1** or **B = 1**
- **Z = 0** when **A = 1** and **B = 0**
- **Z = 0** when **A = 0** and **B = 1**
- **Z = 1** when both **A** and **B** are **1**
NOR operation

A = 0

B = 0

VDD

Z = 1

VSS

A = 1

B = 1

Z = 0

A = 1

B = 0

Z = 0

VDD

VSS

A B

Z

Z = 1
NOR operation

\[ A = 0, B = 0 \Rightarrow Z = 1 \]

\[ A = 0, B = 1 \Rightarrow Z = 0 \]

\[ A = 1, B = 0 \Rightarrow Z = 0 \]

\[ A = 1, B = 1 \Rightarrow Z = 0 \]
NOR operation

- **A = 0**
  - **Z = 1**
  - **B = 0**
  - **Z** is blocked

- **A = 1**
  - **Z = 0**
  - **B** is blocked

- **A = 1** and **B = 1**
  - **Z = 0**

- **V_{DD}**
- **V_{SS}**
- **A**
- **B**
- **Z**
NOR operation
NOR operation

A = 0, Z = 1

A = 1, Z = 0

A = 0, B = 1, Z blocked

A = 1, B = 1, Z = 0
NOR operation

A B
Z

VSS
VDD

A=0
B=1
Z=0

A=1
B=0
Z=0

A=1
B=1
Z=0

A=0
B=0
Z=1
NOR operation

\[
\begin{array}{c}
A = 1 \\
Z = 0 \\
B = 1 \\
V_{SS} \\
V_{DD}
\end{array}
\]
NOR operation
CMOS inefficient for ANDs/ORs

- Recall that NMOS transmits low values easily...
- ...transmits high values poorly
- PMOS transmits high values easily...
- ...transmits low values poorly
CMOS inefficient for ANDs/ORs

- $V_T$, or threshold voltage, is commonly 0.7 V
- NMOS conducts when $V_{GS} > V_T$
- PMOS conducts when $V_{GS} < -V_T$
- What happens if an NMOS transistor’s source is high?
- Or a PMOS transistor’s source is low?
- Alternatively, if one states that $V_{TN} = 0.7$ V and $V_{TP} = -0.7$ V then NMOS conducts when $V_{GS} > V_{TN}$ and PMOS conducts when $V_{GS} < V_{TP}$
NMOS transistor

gate

oxide

source (N)   drain (N)

silicon bulk (P)

silicon bulk (P)
drain (N)
source (N)

38 Robert Dick Advanced Digital Logic Design
CMOS inefficient for ANDs/ORs

- If an NMOS transistor’s input were $V_{DD}$ (high), for $V_{GS} > V_{TN}$, the gate would require a higher voltage than $V_{DD}$.
- If an PMOS transistor’s input were $V_{SS}$ (low), for $V_{GS} < V_{TP}$, the gate would require a lower voltage than $V_{SS}$.
Implications of using CMOS

NAND/NOR easy to build in CMOS

VDD
A B
Z
Z=1
B=0A=0
VSS
VDD
A B
Z
Z=0
B=1
A=1
VSS
VDD
A B
Z
VSS
VDD
A B
Z
A=1
B=0
Z=1

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

NAND/NOR easy to build in CMOS
Implications of using CMOS

AND/OR requires more area, power, time
CMOS transmission gates (switches)

NMOS is good at transmitting 0s
Bad at transmitting 1s

PMOS is good at transmitting 1s
Bad at transmitting 0s

To build a switch, use both: CMOS
Section outline

2. Implementation technologies
   PALs and PLAs
   CMOS for logic gates
   Transmission gates and MUXs
CMOS transmission gate (TG)
CMOS transmission gate (TG)

\[ C = 1 \]

\[ C = 0 \]
CMOS transmission gate (TG)

\[ A \rightarrow B \] when \( C = 0 \)

\[ A \rightarrow \overline{B} \] when \( C = 1 \)
CMOS transmission gate (TG)

\[
\begin{align*}
A & \rightarrow B = A \\
C = 1 & \rightarrow C = 0
\end{align*}
\]
CMOS transmission gate (TG)
CMOS transmission gate (TG)

\[ C = \begin{cases} 0 & \text{for } \bar{C} = 1 \\ 1 & \text{for } \bar{C} = 0 \end{cases} \]
CMOS transmission gate (TG)

When $C = 0$, the gate is blocked.

When $C = 1$, the gate is transparent.

$A$ and $B$ signals pass through the gate when $C = 1$.

$A$ and $B$ signals are blocked when $C = 0$. 

Diagram:

- $C = 0$: Blocked
- $C = 1$: Transparent

Symbols: $A$, $B$, $C$
CMOS transmission gate (TG)

- **Blocked** when $C = 0$
- **Blocked** when $C = 1$
- $B = \text{High-Z}$

Diagram showing the CMOS transmission gate with labeled states for $A$, $B$, and $C$.
CMOS transmission gate (TG)
Other TG diagram
Multiplexers (MUX) definitions

- Also called selectors
- $2^n$ inputs
- $n$ control lines
- One output
### MUX functional table

<table>
<thead>
<tr>
<th>C</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### MUX truth table

<table>
<thead>
<tr>
<th>$I_1$</th>
<th>$I_0$</th>
<th>C</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
MUX using logic gates

A B

I₀ I₁ I₂ I₃

Z

Robert Dick  Advanced Digital Logic Design
MUX using TGs
MUX

A
B
C
D

PALs and PLAs
CMOS for logic gates
Transmission gates and MUXs

Administration
Implementation technologies
Scripting languages
Review of implementation technologies
Homework
MUX

\[ \text{MUX} \]

\[ A \quad \bar{C} \quad C \quad D \]

\[ B \quad C \quad \overline{C} \]

\[ C = 1 \]

PALs and PLAs
CMOS for logic gates
Transmission gates and MUXs

Review of implementation technologies
Scripting languages
Implementation technologies
Administration
MUX

C = 1

A

B

D

C

\overline{C}

\overline{C}

\overline{C}

C

DC = 1

Robert Dick

Advanced Digital Logic Design
MUX

C = 1

D = A

A

B

C

D

C

C

PALs and PLAs
CMOS for logic gates
Transmission gates and MUXs

Advanced Digital Logic Design

Robert Dick

51
MUX

A \rightarrow \bar{C} \rightarrow C \rightarrow \bar{C} \rightarrow \rightarrow D

B \rightarrow C \rightarrow D

C \rightarrow \bar{C} \rightarrow \bar{C}
MUX

![MUX diagram]

- A
- B
- C
- D

When C = 0:

- C = 0
- C = 0

When C = 1:

- C = 0
- C = 0
MUX

A

C

B

C

D

C=0

PALs and PLAs
CMOS for logic gates
Transmission gates and MUXs

Review of implementation technologies
Scripting languages
Implementation technologies
Administration
MUX

\[ C = 0 \Rightarrow D = B \]

\[ \overline{C} \]

\[ C \]

\[ A \]

\[ B \]

\[ D \]
**MUX**

![MUX Diagram](image)

**PALs and PLAs**
- CMOS for logic gates
- Transmission gates and MUXs

**Review of implementation technologies**
- Scripting languages

**Homework**
- PALs and PLAs
- CMOS for logic gates
- Transmission gates and MUXs
Hierarchical MUX implementation

The diagram shows a hierarchical MUX implementation using 4:1 and 2:1 MUXs. The inputs are labeled I_0 to I_7, with selection signals S_0 and S_1 controlling the 4:1 MUXs. The outputs of the 4:1 MUXs are connected to the 2:1 MUX, with inputs B and C connected to the selection signals S_0 and S_1, respectively. The output Z of the 2:1 MUX is the final output of the hierarchical MUX implementation.
Alternative hierarchical MUX implementation
MUX examples

\[ Z = \overline{A} I_0 + AI_1 \]
MUX examples

\[ Z = \overline{A} \overline{B} I_0 + \overline{A} B I_1 + A \overline{B} I_2 + AB I_3 \]
MUX examples

\[ Z = \overline{A} \overline{B} \overline{C} I_0 + \overline{A} \overline{B} C I_1 + \overline{A} B \overline{C} I_2 + \overline{A} B C I_3 + \]
\[ A \overline{B} \overline{C} I_4 + A \overline{B} C I_5 + A B \overline{C} I_6 + A B C I_7 \]
**MUX properties**

- A $2^n : 1$ MUX can implement any function of $n$ variables
- A $2^{n-1} : 1$ can also be used
  - Use remaining variable as an input to the MUX
MUX example

\[ F(A, B, C) = \sum (0, 2, 6, 7) \]
\[ = \overline{A}B\overline{C} + \overline{A}B\overline{C} + AB\overline{C} + ABC \]
### Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Lookup table implementation
MUX example

\[ F(A, B, C) = \sum(0, 2, 6, 7) \]
\[ = \overline{A} \overline{B} \overline{C} + \overline{A} B \overline{C} + A \overline{B} \overline{C} + A B C \]

Therefore,

\[ \overline{A} \overline{B} \rightarrow F = \overline{C} \]
\[ \overline{A} B \rightarrow F = \overline{C} \]
\[ A \overline{B} \rightarrow F = 0 \]
\[ A B \rightarrow F = 1 \]
### Truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
### Truth Table

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

\[ F = \overline{C} \]
Lookup table implementation

MUX

S1 S0

0 1 2 3

A B F

4:1

Robert Dick
Advanced Digital Logic Design
Demultiplexer (DMUX) definitions

- Closely related to *decoders*
- $n$ control signals
- Single data input can be routed to one of $2^n$ outputs
Decoders vs. demultiplexers

- Decoders have $n$ inputs and $2^n$ outputs.
- They activate only the output indicated by the binary input value.
- Demultiplexers have one input, $2^n$ select lines, and $2^n$ outputs.
- They route the input to the output indicated by the binary select value, and inactivate the other outputs.
- In practice, decoders have an output enable input.
- If you treat a decoder output enable as a demultiplexer input and treat the decoder inputs as demultiplexer select lines, the two are equivalent.
- In practice decoders and demultiplexers are interchangeable.
Active high 1:2 decoder

- g
- select
- output 0
- output 1
Outline

1. Administration
2. Implementation technologies
3. Scripting languages
4. Review of implementation technologies
5. Homework
Perl/Python

- Lab two has a tedious portion
- You’ll need to lookup gates in a library
- I wrote a perl and python script for you to accelerate this process
  - ...and serve as examples
- You’ll still need to do this manually once
- Can use my scripts afterward
  - Read and understand it
Perl/Python

- A genius colleague is an ASIC (Application-Specific Integrated Circuit) design engineer at PMC-Sierra
- He glues together standard cells to make high-performance special-purpose circuits
- When he talks about perl, his eyes get all watery
- Why do digital design engineers get so excited about a system administrator’s scripting language?
Commercial CAD tool flows are often a mess

- Different tools that don’t quite work together
  - Translation
- Tools that don’t quite finish the job
  - Pre-post processing
- Poor support for complex testing, e.g., comparing a high-level language model’s behavior with circuit simulation
  - IO processing and command scripting
- Perl (python, etc.) allow quick (although sometimes inelegant) solutions to these problems
Perl code example motivation – Library

GATE "1310:physical" 16  O=!1A;
PIN * INV 1 999 1 .2 1 .2

GATE "1120:physical" 24  O=!1A+1B);
PIN * INV 1 999 1 .2 1 .2

GATE "1130:physical" 32  O=!1A+1B+1C);
PIN * INV 1 999 1 .2 1 .2

GATE "1220:physical" 24  O=!1A*1B);
PIN * INV 1 999 1 .2 1 .2
Perl code example motivation – Gates

[348] 2310:physical  40.00
{cout} 1970:physical  56.00
[345] 1310:physical  16.00
[364] 1310:physical  16.00
{sum} 1860:physical  40.00
Perl code example

# Make sure number of args is correct
if (scalar @ARGV != 2) {
    die "Usage: lookup-gate.perl " .
        "[library] [file]\n"
;
}

my $lib = $ARGV[0];
my $file = $ARGV[1];

my %lab_op = ();
Perl code example

# Read in library
open LIB, "< $lib";
while (<LIB>) {
    if (m/^GATE\s+"([\^]+)"\s+\d+\s+(.+)$/) {
        # Put the data into a hash map
        my ($label, $op) = ($1, $2);
        $lab_op{$label} = $op;
    }
}
close LIB;
Perl code example

# Loop on input file
open FILE, "< $file";
while (<FILE>) {
    # Chop up the line on whitespace
    my @ln = split ' ', $_;
    if (scalar(@ln) == 3) {
        # Grab the node and label
        my ($node, $lab) = @ln;
    }
Perl code example

# Look it up in the hash map and
# display the results
print "Node $node implemented with " .
"gate $lab_op{$lab}\n"

} }
close FILE;
Perl code output

Node [348] implemented with gate $O=!(A\cdot B+!A\cdot!B)$;
Node \{cout\} implemented with gate $O=A\cdot B+2C\cdot 2D$;
Node [345] implemented with gate $O=!A$;
Node [364] implemented with gate $O=!A$;
Node \{sum\} implemented with gate $O=!((A+1B)\cdot(2C+2D))$;
# Make sure number of args is correct
if len(sys.argv) < 3:
    print 'Usage: lookup-gate.py [library] [file]'
    sys.exit(0)
lib, file = sys.argv[1:]
lab_op = dict()
Python code example

# Read in library
fl = open(lib)
for ln in fl.readlines():
    m = re.match('^GATE\s+"([^"
]+)"\s+\d+\s+(.+)$', ln)
    if m:
        # Put the data into a hash map
        label, op = m.group(1, 2)
        lab_op[label] = op
fl.close()
Python code example

```python
# Loop on input file
fl = open(file)
for ln in fl.readlines():
    # Chop up the line on whitespace
    ln_ar = ln.split()
    if len(ln_ar) == 3:
        # Grab the node and label
        node, lab = ln_ar[:2]
        # Look it up in the hash map and display the results
        print 'Node %s implemented with gate %s' % (node, lab_op[lab])
fl.close()
```
Outline

1. Administration
2. Implementation technologies
3. Scripting languages
4. Review of implementation technologies
5. Homework
Back to implementation technologies

What is this?
What is this?

Select

Output0

Output1
Active-high 2:4 decoder/demultiplexer

G

Select0

Select1

Output0

Output1

Output2

Output3

Robert Dick

Advanced Digital Logic Design
Active-low 2:4 decoder/demultiplexer
Dangers when implementing with TGs

What if an output is not connected to any input?
Consider undriven inverter inputs
Consider undriven inverter inputs

A = 0
B = 1

A = ?
B = ?

VDD
VSS
Consider undriven inverter inputs

\[ A = 0, B = 1 \]

\[ A = 1, B = 0 \]

\[ A = ?, B = ? \]
Consider undriven inverter inputs

- \( V_{DD} \)
- \( V_{SS} \)

- \( A=0 \)  \( B=1 \)
- \( A=1 \)  \( B=0 \)
- \( A=? \)  \( B=? \)
Consider undriven inverter inputs
Consider undriven inverter inputs
Dangers when implementing with TGs
Set all outputs
Demultiplexer

However, this implementation is dangerous

C = 1
Demultiplexer

However, this implementation is dangerous
However, this implementation is dangerous.
Demultiplexer

However, this implementation is dangerous
However, this implementation is dangerous.
Demultiplexer

However, this implementation is dangerous.
Demultiplexer

\[ \text{D} = \text{High-Z} \]

\[ \text{E} = \text{A} \]

\[ \text{C} = 0 \]
Demultiplexer

However, this implementation is dangerous...
Demultiplexer

However, this implementation is dangerous.
TG decoder/demultiplexer implementation

Consider alternative paths
Demultiplexers as building blocks

Generate minterm based on control signals
Example function

\[ F_1 = \overline{A} \overline{B} CD + \overline{A} B \overline{C} D + ABCD \]
\[ F_2 = AB \overline{C} \overline{D} + ABC = AB \overline{C} \overline{D} + ABCD + ABCD \]
\[ F_3 = \overline{A} + \overline{B} + \overline{C} + \overline{D} = \overline{ABCD} \]
Demultiplexers as building blocks

If active-low, use NAND gates
1:32 demultiplexer
Multiple I/O circuit

Diagram:
- A0, A1 inputs to MUX
- B0, B1 inputs to MUX
- Outputs from MUXs are A and B
- Sum of A and B
- Ss input to DEMUX
- Outputs from DEMUX are S0 and S1
Summary

- Q&A
- PALs/PLAs
- Review, Q&A on MOS transistors
- Multiplexers, Demultiplexers
- Transmission gates
- Perl/Python
Outline

1. Administration
2. Implementation technologies
3. Scripting languages
4. Review of implementation technologies
5. Homework
Recommended reading
- Chapters 3 and 4

Lab two
Espresso and SIS logic minimization
Next lecture

- ROMs
- Multilevel logic minimization
- Review: NAND/NOR implementation