Outline

- Introduction
- Datapath Operators
- Control Structures
System-Level Hierarchy
Categories of Components

- Types of digital component
  - Datapath operators
  - Memory elements
  - Control structures
  - I/O cells

- Tradeoff of selection
  - Speed
  - Density
  - Programmability
  - Easy of design
  - etc
## Datapath – Adder

The adder is a fundamental building block in digital circuits. It performs the basic arithmetic operation of addition. A full adder can add two 1-bit numbers and a carry input to produce a 1-bit sum and a carry output. The truth table for an adder is as follows:

<table>
<thead>
<tr>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A.B</th>
<th>A+B</th>
<th>A⊕B</th>
<th>SUM</th>
<th>CARRY</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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</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>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Generate Signal G(A.B):** occurs when a carry output (CARRY) is internally generated within the adder.

**Propagate Signal P(A+B):** when it is true, the carry in signal C is passed to the carry output (CARRY) when C is true.
Datapath – Adder

\[
\begin{align*}
\text{SUM} &= A \oplus B \oplus C \\
\text{CARRY} &= AB + AC + BC
\end{align*}
\]

Single-bit schematic of SUM
Datapath – *Adder*

Single-bit schematic of CARRY

![Single-bit schematic of CARRY](image)
Datapath – **Adder**

Optimized combinational adder schematic

\[
C_{i+1} = A_i B_i + A_i C_i + B_i C_i \\
S_i = (A_i + B_i + C_i) \cdot \overline{C_{i+1}} + A_i B_i C_i
\]
Datapath – Adder

Symmetrical optimized combinational adder schematic
Datapath – *Bit-Parallel Adder*

Parallel adder implementations

```
\[\begin{array}{c}
A_{n} \\
B_{n} \\
C_{n}
\end{array}\]
\[\begin{array}{c}
S_{n} \\
C_{n+1}
\end{array}\]

```

```
\[\begin{array}{c}
A_{3} \\
B_{3} \\
C_{3}
\end{array}\]
\[\begin{array}{c}
S_{3} \\
C_{3}
\end{array}\]

```

```
\[\begin{array}{c}
A_{2} \\
B_{2} \\
C_{2}
\end{array}\]
\[\begin{array}{c}
S_{2} \\
C_{2}
\end{array}\]

```

```
\[\begin{array}{c}
A_{1} \\
B_{1} \\
C_{1}
\end{array}\]
\[\begin{array}{c}
S_{1} \\
C_{1}
\end{array}\]

```

```
\[\begin{array}{c}
A_{0} \\
B_{0} \\
C_{0}
\end{array}\]
\[\begin{array}{c}
S_{0} \\
C_{0}
\end{array}\]

```

Advanced Reliable Systems (ARES) Lab.

Jin-Fu Li, EE, NCU
Datapath – *Bit-Parallel Adder*

\[
\begin{align*}
&\text{A-B} \\
&\text{If (Subtract==0)} \\
&\quad \{ S=A+B; \} \\
&\text{else} \\
&\quad \{ S=A-B; \}
\end{align*}
\]
Datapath – *Bit-Serial Adder*

**Addend:**

- $A$: 01101
- $B$: 01001
- Addend: 10

**Result:**

- Cout: 00
- Result: 10010010

**Cin:**

- 00

**Diagram:**

- Bit-serial adder circuit with inputs $A$, $B$, addend, and augend, and outputs Cout and Result.
Datapath – Carry Look-Ahead Adder (CLA)

Objective

- To avoid the linear growth of the carry delay, we use a Carry Look-Ahead Adder (CLA) in which the carries can be generated in parallel.

Feature

- The Carry of each bit is generated from the propagate and the generate signals as well as the input carry.
- The propagate and the generate signals are derived from the operand $A_i$ and $B_i$ by
  - $G_i = A_i \cdot B_i$
  - $P_i = A_i + B_i$
Datapath – Carry Look-Ahead Adder

\[ C_{i+1} = A_iB_i + (A_i + B_i)C_i = G_i + P_iC_i \]

\[ C_1 = G_0 + P_0C_0 \]

\[ C_2 = G_1 + P_1G_0 + P_1P_0C_0 \]

\[ C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \]

\[ C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \]
Datapath – *Carry Look-Ahead Adder*

CLG1

![Carry Look-Ahead Adder Diagram](image-url)
Datapath – Carry Look-Ahead Adder

CLG4

\[ C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \]
Datapath – *Carry Look-Ahead Adder*

Manchester Carry Chain

\[
C_{i+1} = G_i + P_i C_i \\
G_i = A_i B_i \\
P_i = A_i + B_i
\]

Introduce the carry-kill bit \( K_i \), this term gets its name from the fact that if \( K_i = 1 \), then \( P_i = 0 \) and \( G_i = 0 \), so that \( C_{i+1} = 0 \); \( K_i = 1 \) thus “kills” the carry-out bit.

\[
K_i = \overline{A_i} \cdot \overline{B_i}
\]

<table>
<thead>
<tr>
<th>( A_i )</th>
<th>( B_i )</th>
<th>( P_i )</th>
<th>( G_i )</th>
<th>( K_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>1 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 1</td>
<td>0 1</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>
Datapath – Carry Look-Ahead Adder

Manchester circuit styles

Static circuit

Dynamic circuit

Dynamic Manchester chain
Extension to wide adders

If we use a brute-force approach for an 8-bit design, then the carry-out bit $C_8$ would have a term of the form

$$P_7 P_6 P_5 P_4 P_3 P_2 P_1 P_0 C_0$$

Multilevel CLA networks can improve this problem.
Datapath – *Carry Look-Ahead Adder*

\[ G_{[i,i+3]} = G_{i+3} + P_{i+3} G_{i+2} + P_{i+3} P_{i+2} G_{i+1} + P_{i+3} P_{i+2} P_{i+1} G_{i} \]

\[ P_{[i,i+3]} = P_{i+3} P_{i+2} P_{i+1} P_{i} \]
A carry-skip adder is designed to speed up a wide adder by aiding the propagation of a carry bit around a portion of the entire adder.

\[
P_{[i,i+3]} = p_{i+3}p_{i+2}p_{i+1}p_i
\]

\[
\text{Carry} = c_{i+4} + P_{[i,i+3]}c_i
\]
Datapath – Conditional-Sum Adder

Advanced Reliable Systems (ARES) Lab.  Jin-Fu Li, EE, NCU
Datapath – 8-bit Conditional-Sum Adder
Bit-level multiplier

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>axb</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>

Multiplication of two 4-bit words

\[
\begin{array}{cccc}
 a_3 & a_2 & a_1 & a_0 \\
b_3 & b_2 & b_1 & b_0 \\
\end{array}
\]

\[
\begin{array}{cccc}
 a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\
p_7 & p_6 & p_5 & p_4 \\
p_3 & p_2 & p_1 & p_0 \\
\end{array}
\]
The product $axb$ is given by the 8-bit result

$$p = p_7p_6p_5p_4p_3p_2p_1p_0$$

The $i$th product term $p_i$ can be expressed as

$$p_i = \sum_{i=j+k} a_j b_k + c_{i-1}$$

Alternate view of multiplication process

$$\begin{array}{cccccc}
 a_3 & a_2 & a_1 & a_0 \\
 b_3 & b_2 & b_1 & b_0 \\
\end{array}$$

$$\begin{array}{cccc}
 (a_3) & (a_2) & (a_1) & (a_0)xb_0 \\
 (a_3) & (a_2) & (a_1) & (a_0)xb_1 \\
 (a_3) & (a_2) & (a_1) & (a_0)xb_2 \\
 (a_3) & (a_2) & (a_1) & (a_0)xb_3 \\
\end{array}$$

$$\begin{array}{cccccc}
 p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
\end{array}$$

$$(axb_0)2^0$$

$$(axb_1)2^1$$

$$(axb_2)2^2$$

$$(axb_3)2^3$$
Using a product register for multiplication

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(xb)_{0}^{0}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(xb)_{1}^{1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(xb)_{2}^{2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(xb)_{3}^{3}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Datapath – Multipliers

#### Shift-right multiplication sequence

<table>
<thead>
<tr>
<th>Add (axb&lt;sub&gt;0&lt;/sub&gt;)</th>
<th>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</th>
<th>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</th>
<th>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</th>
<th>a&lt;sub&gt;0&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift right</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
<td>a&lt;sub&gt;0&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Add (axb&lt;sub&gt;1&lt;/sub&gt;)</th>
<th>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</th>
<th>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</th>
<th>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</th>
<th>a&lt;sub&gt;0&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&lt;sub&gt;x&lt;/sub&gt;</td>
<td>C&lt;sub&gt;x&lt;/sub&gt;</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;0&lt;/sub&gt;</td>
</tr>
<tr>
<td>Shift right</td>
<td>C&lt;sub&gt;x&lt;/sub&gt;</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Add (axb&lt;sub&gt;2&lt;/sub&gt;)</th>
<th>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</th>
<th>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</th>
<th>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</th>
<th>a&lt;sub&gt;0&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&lt;sub&gt;y&lt;/sub&gt;</td>
<td>C&lt;sub&gt;y&lt;/sub&gt;</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>Shift right</td>
<td>C&lt;sub&gt;y&lt;/sub&gt;</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Add (axb&lt;sub&gt;3&lt;/sub&gt;)</th>
<th>p&lt;sub&gt;7&lt;/sub&gt;</th>
<th>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;3&lt;/sub&gt;</th>
<th>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;3&lt;/sub&gt;</th>
<th>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;3&lt;/sub&gt;</th>
<th>a&lt;sub&gt;0&lt;/sub&gt;b&lt;sub&gt;3&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift right</td>
<td>p&lt;sub&gt;7&lt;/sub&gt;</td>
<td>a&lt;sub&gt;3&lt;/sub&gt;b&lt;sub&gt;4&lt;/sub&gt;</td>
<td>a&lt;sub&gt;2&lt;/sub&gt;b&lt;sub&gt;4&lt;/sub&gt;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;b&lt;sub&gt;4&lt;/sub&gt;</td>
<td>a&lt;sub&gt;0&lt;/sub&gt;b&lt;sub&gt;4&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

---

Advanced Reliable Systems (ARES) Lab. Jin-Fu Li, EE, NCU

28
Datapath – Register-Based Multiplier

Product register (2n)

Multiplicand

Multiplier

n-bit adder

MUX
Consider two unsigned binary integers $X$ and $Y$

$$X = \sum_{i=0}^{n-1} X_i 2^i \quad Y = \sum_{j=0}^{n-1} Y_j 2^j$$

$$P = X \times Y = \sum_{i=0}^{n-1} X_i 2^i \cdot \sum_{j=0}^{n-1} Y_j 2^j$$

$$= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (X_i Y_j) 2^{i+j}$$

$$= \sum_{k=0}^{n+n-1} P_k 2^k$$
Datapath – Array Multipliers

Advanced Reliable Systems (ARES) Lab.

Jin-Fu Li, EE, NCU
Datapath – Array Multipliers
Booth’s algorithm takes advantages of the fact that an adder-subtractor is nearly as fast and small as a simple adder.

Consider the two’s complement representation of the multiplier $y$

- $y = -2^n y_n + 2^{n-1} y_{n-1} + 2^{n-2} y_{n-2} + \cdots$

The representation can be rewritten as

- $y = 2^n (y_{n-1} - y_n) + 2^{n-1} (y_{n-2} - y_{n-1}) + 2^{n-2} (y_{n-3} - y_{n-2}) + \cdots$

Extract the first two terms

- $y = 2^n (y_{n-1} - y_n) + 2^{n-1} (y_{n-2} - y_{n-1})$

The right-hand term can be used to add $x$ to partial product

The left-hand term add $2x$
Datapath – *Booth Multiplier*

Actions during Booth multiplication

<table>
<thead>
<tr>
<th>( y_i )</th>
<th>( y_{i-1} )</th>
<th>( y_{i-2} )</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Add 0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Add x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Add x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Add 2x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Sub 2x</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Sub x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Sub x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Add 0</td>
</tr>
</tbody>
</table>

For example, \( x=011001 \) (\( 25_{10} \)), \( y=101110 \) (-\( 18_{10} \))

1. \( y_1y_0y_{-1}=100 \), so \( P_1=P_0-2\times.1=11111001110 \)
2. \( y_3y_2y_1=111 \), so \( P_2=P_1+0.4=11111001110 \)
3. \( y_5y_4y_3=101 \), so \( P_3=P_2-x.16=11000111110 \)
Datapath – *Booth Multiplier*

Structure of a Booth multiplier

![Booth Multiplier Diagram](image-url)
Datapath – Wallace Tree Multiplier

- A Wallace tree is a full adder tree structured specially for a quick addition of the partial products

- Example
  - A 16x16 Booth multiplier
  - 8 partial products are generated
  - Assume that all partial products are negative so all sign extension bits are 1’s
  - Sign extension correction vector is 1010101010101011

```
1111111111111111 1111111111111111 111111111111 11111111 111111 1111 11
1010101010101011
```
Wallace tree multiplication

Partial Products

1st stage 4-2 compression

2nd stage 4-2 compression

1 0 1 0 1 0 1 0 1 0 1 0 1 1

Sign Extension Correction

Final Addition
Datapath – Wallace Tree Multiplier

4-2 compressor

Outputs

Inputs

Inputs

Outputs

Carry-save adder
Serial multiplier

1. Require MN clock cycles to produce a product for an N-bit multiplier and a M-bit multiplicand
Datapath – **Serial Multiplication**

Serial/parallel multiplier

1. Require $M+N$ clock cycles to produce a product for an $N$-bit multiplier and a $M$-bit multiplicand
2. The critical path consists of the adders
Control – *FSM*

Moore

![Moore FSM Diagram](image)

Mealy

![Mealy FSM Diagram](image)
Control – FSM

- FSM design procedure
  - Draw the state-transition diagram
  - Check the state diagram
  - Write state equations (Write HDL)
- An example of state-transition diagram

IDLE: \((S1,S0)=(00)\)
WAIT: \((S1,S0)=(01)\)
EXIT: \((S1,S0)=(10)\)
A: car-in
C: change-ok
R: rst
Control – *FSM*

- Check the state-transition diagram
  - Ensure all states are represented, including the IDLE state
  - Check that the OR of all transitions leaving a state is TRUE. This is a simple method of determining that there is a way out of a state once entered.
  - Verify that the pairwise XOR of all exit transitions is TRUE. This ensures that there are not conflicting conditions that would lead to more than one exit-transition becoming active at any time.
  - Insert loops into any state if it is not guaranteed to otherwise change on each cycle.
- Formal FSM verification method
  - Perform conformance checking
module toll_booth(clk,rst,car_in,change_ok,green);
input          clk,rst,car_in,change_ok; output        green; reg[1:0]      state_reg, next_state; parameter IDLE  = 2'b00; parameter WAIT = 2'b01; parameter EXIT  = 2'b11;
always @(posedge clk or posedge rst) begin
  If (rst==1'b1) state_reg<=IDLE; else state_reg<=next_state;
end always @(state_reg or car_in or change_ok) begin
  case(state_reg):
    IDLE: if (car_in==1'1) begin
      next_state=WAIT;
      green=1'b0;
    end else begin
      next_state=IDLE;
    end
    WAIT: if (change==1'b1) begin
      next_state=EXIT;
      green=1'b1;
    end else begin
      next_state=WAIT;
    end
    default: begin
      next_state=IDLE;
      green=1'b0;
    end
  endcase
endmodule

EXIT: if (car_in==1'1) begin
  next_state=EXIT;
  green=1'b1;
end else begin
  next_state=IDEL;
  green=1'b0;
end
default: begin
  next_state=IDLE;
  green=1'b0;
end
endcase
endmodule
Control – *PLA*

**Structure of a PLA**

A PLA represents an expression of sum-of-product (SOP)

\[
f_i = \sum_i m_i(a, b, c, d)
\]

\[
f_1 = \overline{a} \cdot \overline{b} \cdot c \cdot \overline{d} + \overline{a} \cdot b \cdot c \cdot \overline{d} + a \cdot b \cdot c \cdot d
\]
Control – PLA

Fuse-programmable PLA
Logic gate diagram of a PLA