Combinational Building Blocks – Decoder

One-hot representation

- Represent a set of N elements with N bits
- Exactly one bit is set
- Example – encode numbers 0-7

Binary	One-hot
000	00000001
001	00000010
010	00000100
…	…
110	01000000
111	10000000

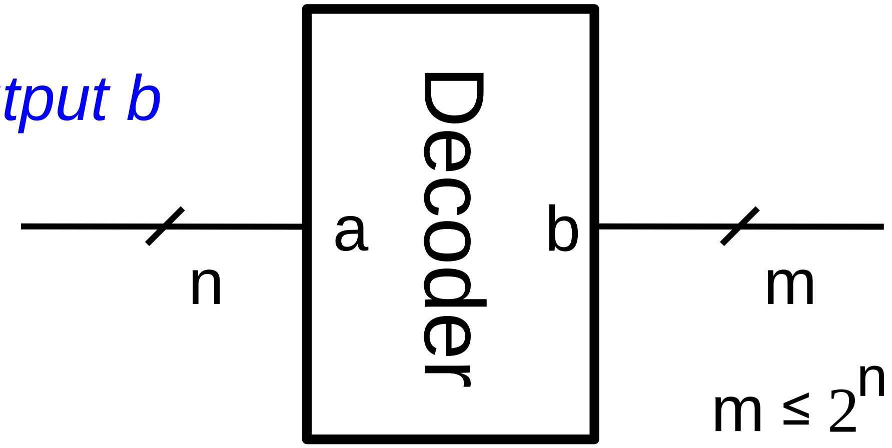
Decoder

- A decoder converts symbols from one code to another.
- A binary to one-hot decoder converts a symbol from binary code to a one-hot code.
  - One-hot code: exactly one bit is high at any given time and each bit represents a symbol.

Binary input a to one-hot output b

b[i] = 1 if a = i

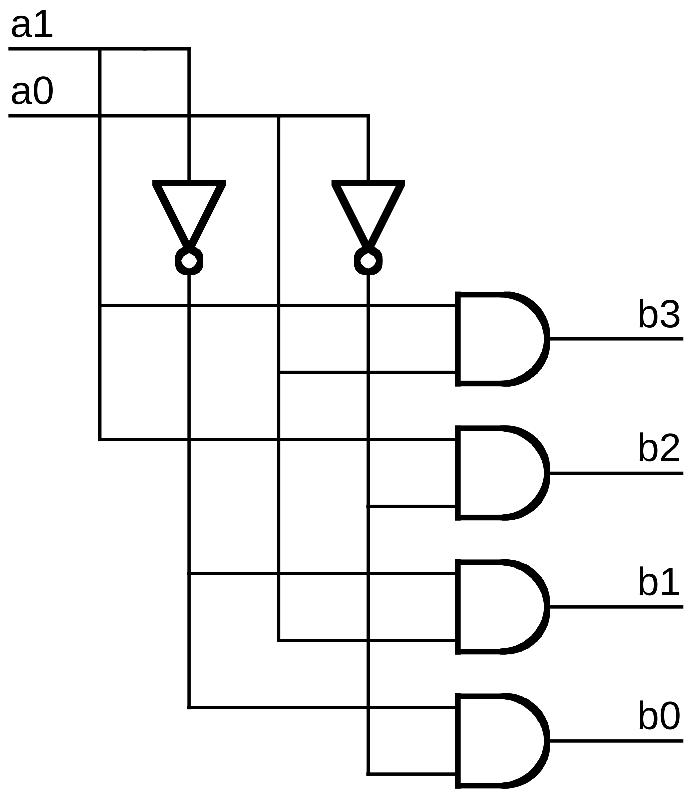
b = 1<<a



Example

$2 \longrightarrow 4$ Decoder

a1	a0	b3	b2	b1	b0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



Can build a large decoder from several small decoders

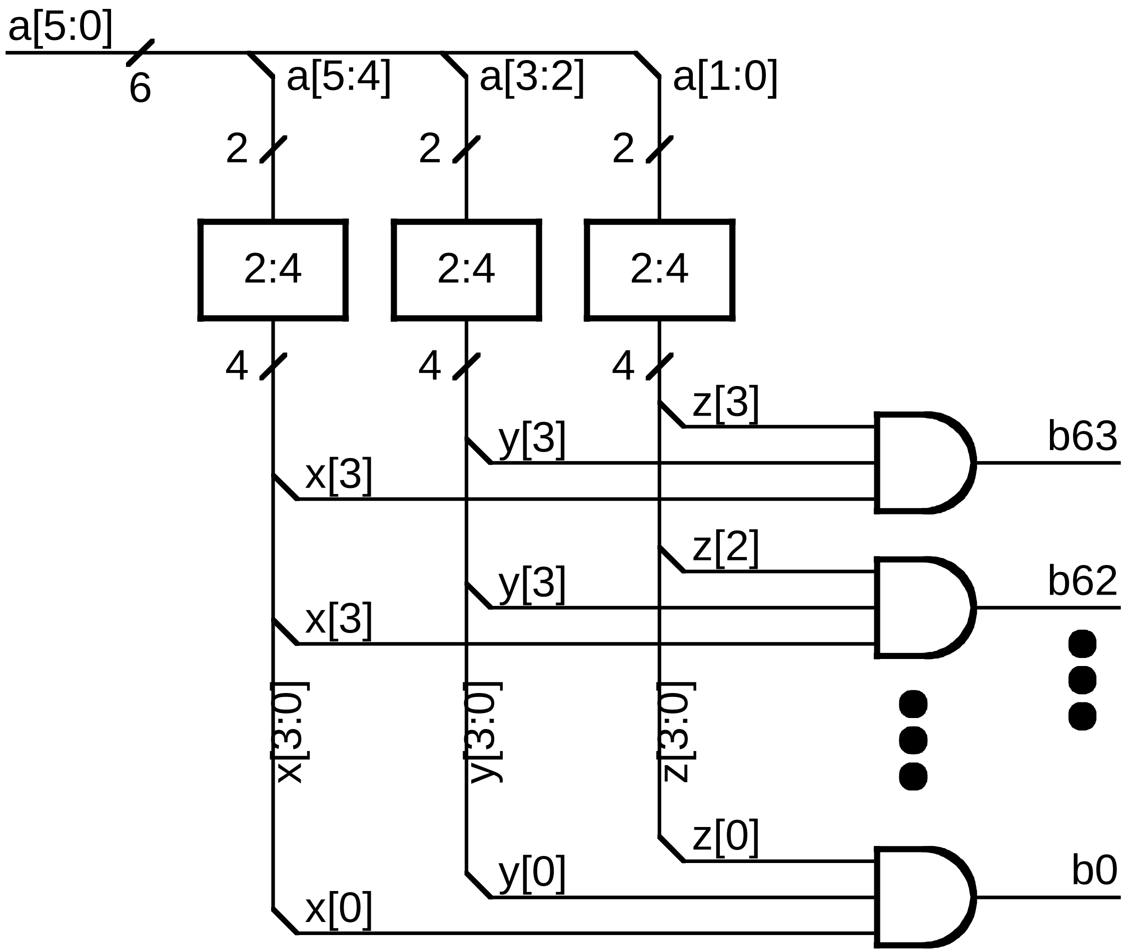
- Example – build a 6 : 64 decoder from three 2 : 4 decoders
- Each 2 : 4 decodes 2 bits
  - a[5:4]→x[3:0], a[3:2]→y[3:0], a[1:0]→z[3:0]
- AND one bit each of x, y, and z to generate an output
  - b[a] = x[a[5:4]] & y[a[3:2]] & z[a[1:0]]

a5	a4	x3	x2	x1	x0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

a3	a2	y3	y2	y1	y0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

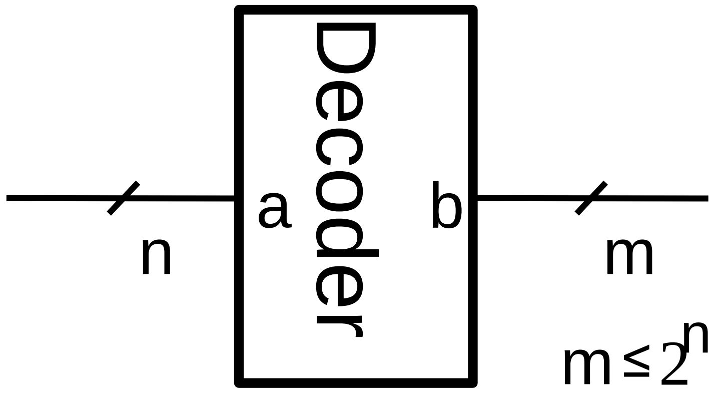
a1	a0	z3	z2	z1	z0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

2-Stage decoders

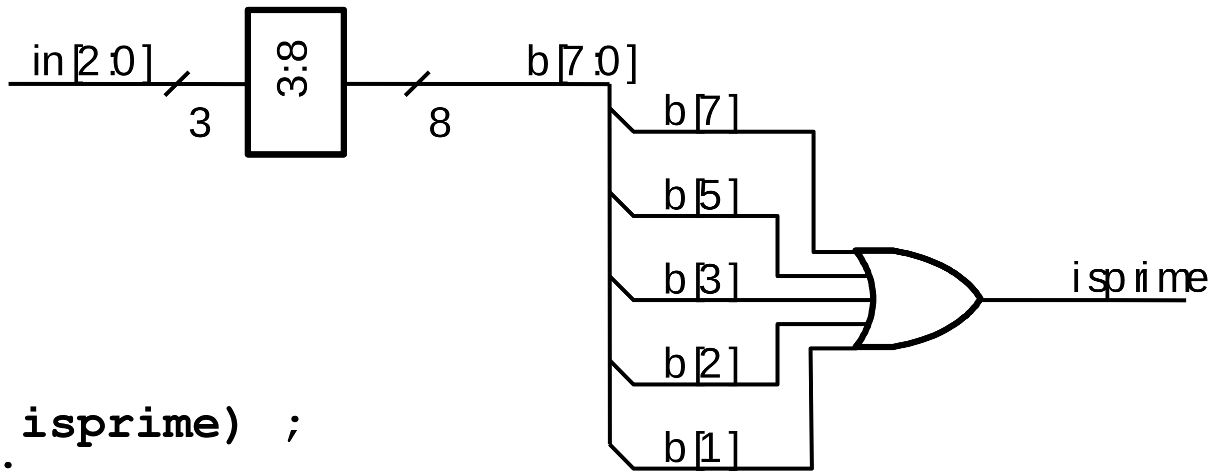


Verilog implementation of a decoder

```
// a - binary input    (n bits wide)
// b - one hot output (m bits wide)
module Dec(a, b) ;
  parameter n=2 ;
  parameter m=4 ;

  input  [n-1:0] a ;
  output [m-1:0] b ;

  wire [m-1:0] b = 1<<a ;
endmodule
```



Logic implementation using decoder
Example – 3-bit prime number function



```
module Primed(in, isprime) ;
  input [2:0] in ;
  output      isprime ;
  wire   [7:0] b ;

  // compute the output as the OR of the required minterms
  wire        isprime = b[1] | b[2] | b[3] | b[5] | b[7] ;

  // instantiate a 3->8 decoder
  Dec #(3,8) d(in,b) ;
endmodule
```

Combinational Building Blocks – Encoder

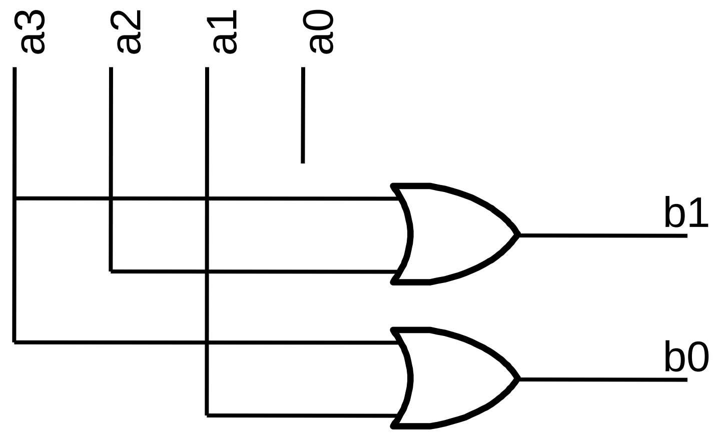
Encoder

- An encoder is an inverse of a decoder.
- Encoder is a logic module that converts a one-hot input signal to a binary-encoded output signal.
- Example: a 4 : 2 encoder.

a3	a2	a1	a0	b1	b0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

$b0 = a3 \vee a1$

$b1 = a3 \vee a2$



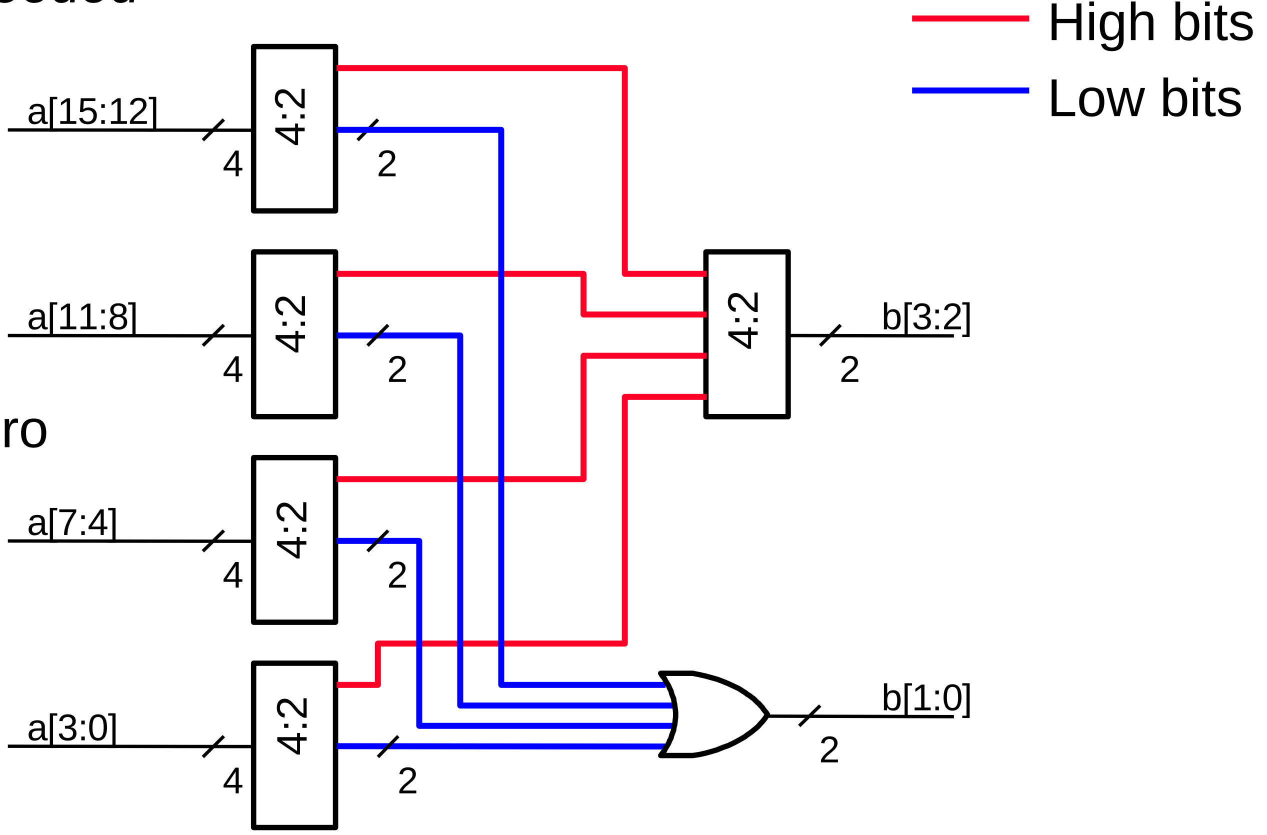
Designing a large encoder – 16 : 4 from five 4 : 2 encoders

Need an 'any input true' output on first rank of encoders

First rank encodes low bits, second rank encodes high bits

Repeat as needed

One hot or all zero



Summary

- Assemble combinational circuits from pre-defined building blocks
  - Decoder – converts codes (e.g., binary to one-hot)
  - Encoder – encodes one-hot to binary
  - Multiplexer – select an input (one-hot select)
  - Arbiter – pick first true bit
  - Comparators – equality and magnitude
  - ROMs
  - RAMs
  - PLAs
- Divide and conquer to build large units from small units
  - Decoder, encoder, multiplexer
- Logic with multiplexers or decoders
- Bit-slice coding style

Combinational Building Blocks – MUX

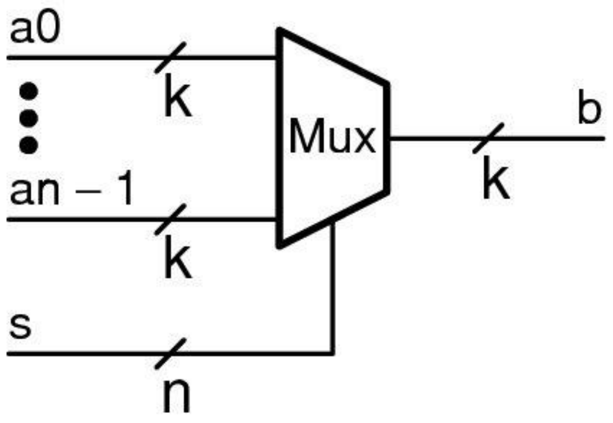
Multiplexer

- Multiplexer:
  - n k-bit inputs
  - n-bit one-hot select signal s
  - Multiplexers are commonly used as *data selectors*

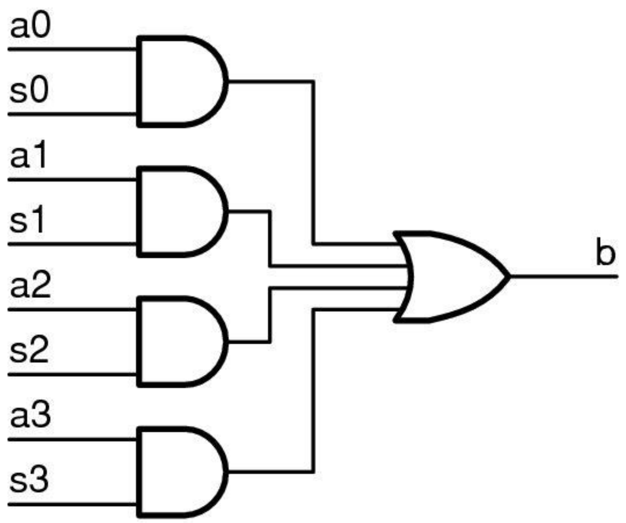
Selects one of n k-bit inputs

s must be one-hot

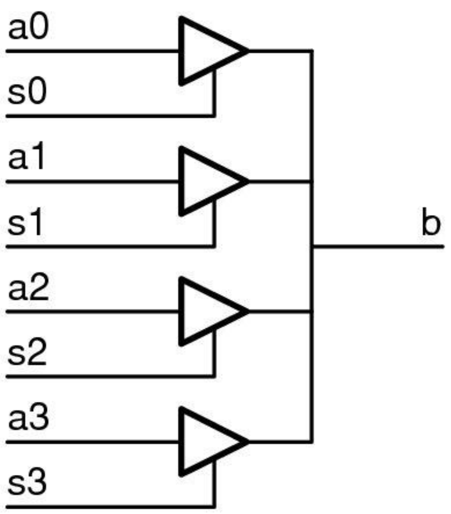
b = ai if s[i] = 1



Multiplexer Implementation



(a)

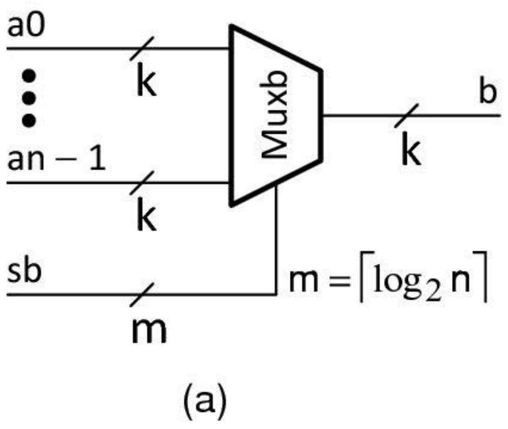


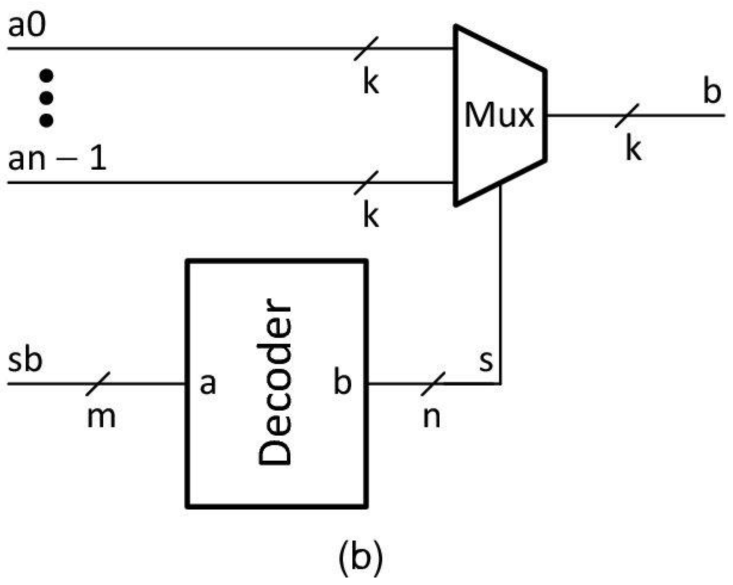
(b)

```
// four input k-wide mux with one-hot select
module Mux4(a3, a2, a1, a0, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2, a3 ;  // inputs
  input [3:0]   s ; // one-hot select
  output[k-1:0] b ;
  wire [k-1:0] b = ({k{s[0]}} & a0) |
                   ({k{s[1]}} & a1) |
                   ({k{s[2]}} & a2) |
                   ({k{s[3]}} & a3) ;
endmodule

Mux4 #(2) mx(2'd3, 2'd2, 2'd1, 2'd0, f, h) ;

  f    h
# 0001 00
# 0010 01
# 0100 10
# 1000 11
```

```
module Mux3a(a2, a1, a0, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2 ;  // inputs
  input [2:0]   s ; // one-hot select
  output[k-1:0] b ;
  reg [k-1:0] b ;

  always @(*) begin
    case(s)
      3'b001: b = a0 ;
      3'b010: b = a1 ;
      3'b100: b = a2 ;
      default: b =  {k{1'bx}} ;
    endcase
  end
endmodule
```

k-bit Binary-Select Multiplexer

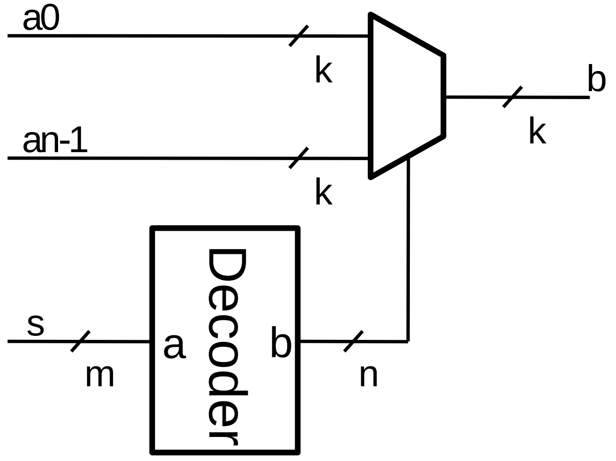


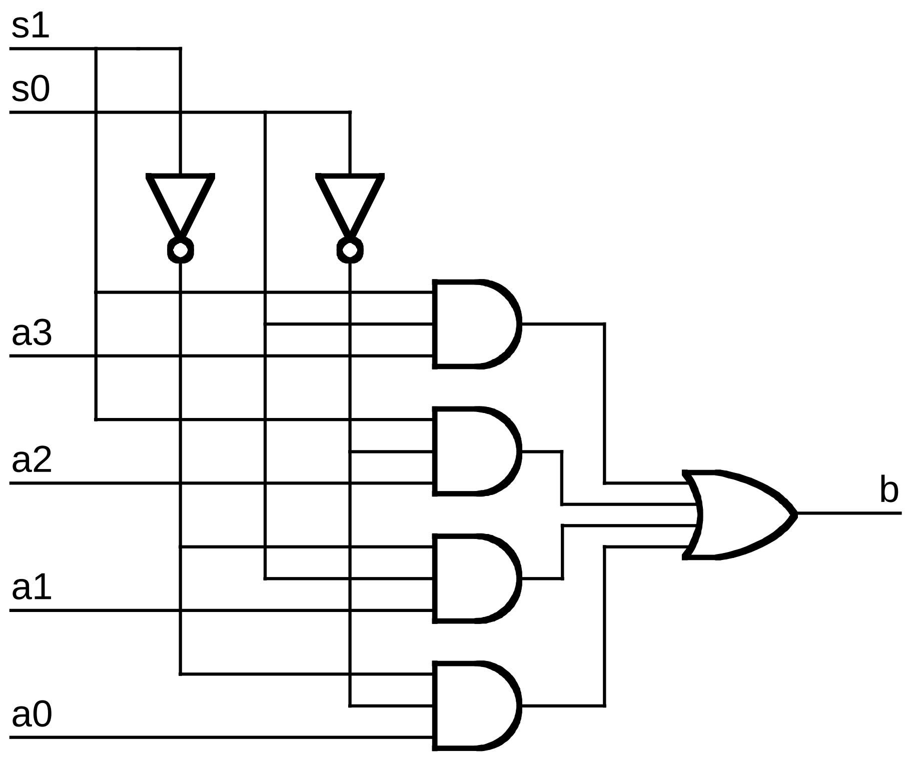


Selects one of n k-bit inputs

b = ai if sb = i

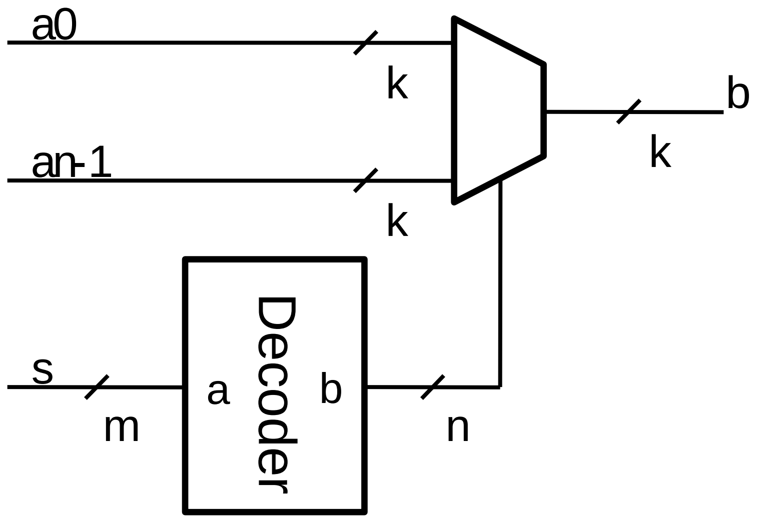
k-bit Binary-Select Multiplexer





Implementing k-bit Binary-Select Multiplexer using Verilog

```
// 3:1 multiplexer with binary select (arbitrary width)
module Muxb3(a2, a1, a0, sb, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2 ;  // inputs
  input [1:0]   sb ;          // binary select
  output[k-1:0] b ;
  wire  [2:0]   s ;

  Dec #(2,3) d(sb,s) ; // Decoder converts binary to one-hot
  Mux3 #(k)  m(a2, a1, a0, s, b) ; // multiplexer selects input
endmodule
```

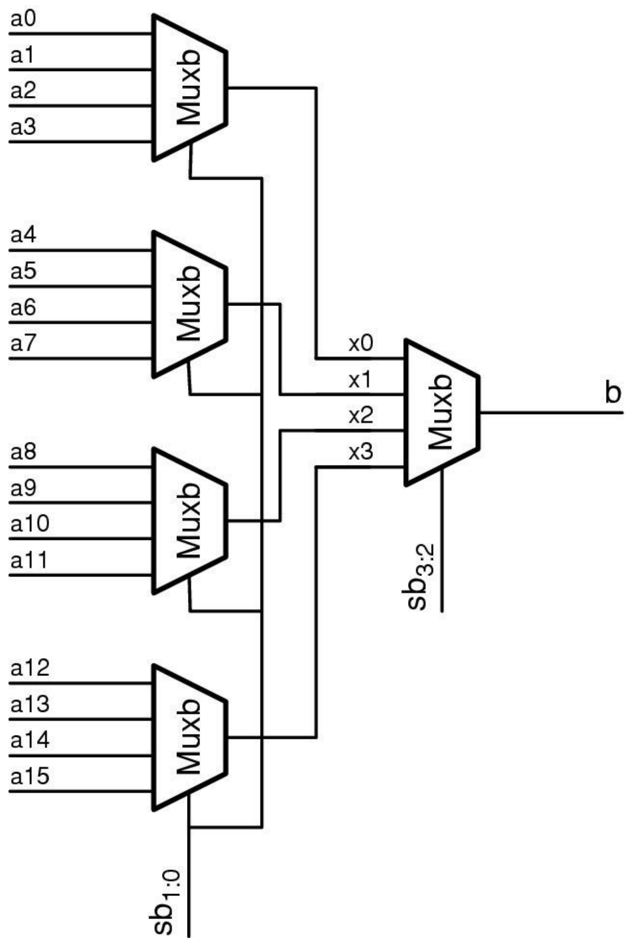


```
// 3:1 multiplexer with binary select (arbitrary width)
module Muxb3a(a2, a1, a0, sb, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2 ;  // inputs
  input [1:0]   sb ;          // binary select
  output[k-1:0] b ;
  reg   [k-1:0] b ;

  always @(*) begin
    case (sb)
     0 : b = a0 ;
     1 : b = a1 ;
     2 : b = a2 ;
     default: b = {k{1'bx}} ;
    endcase
  end

endmodule
```

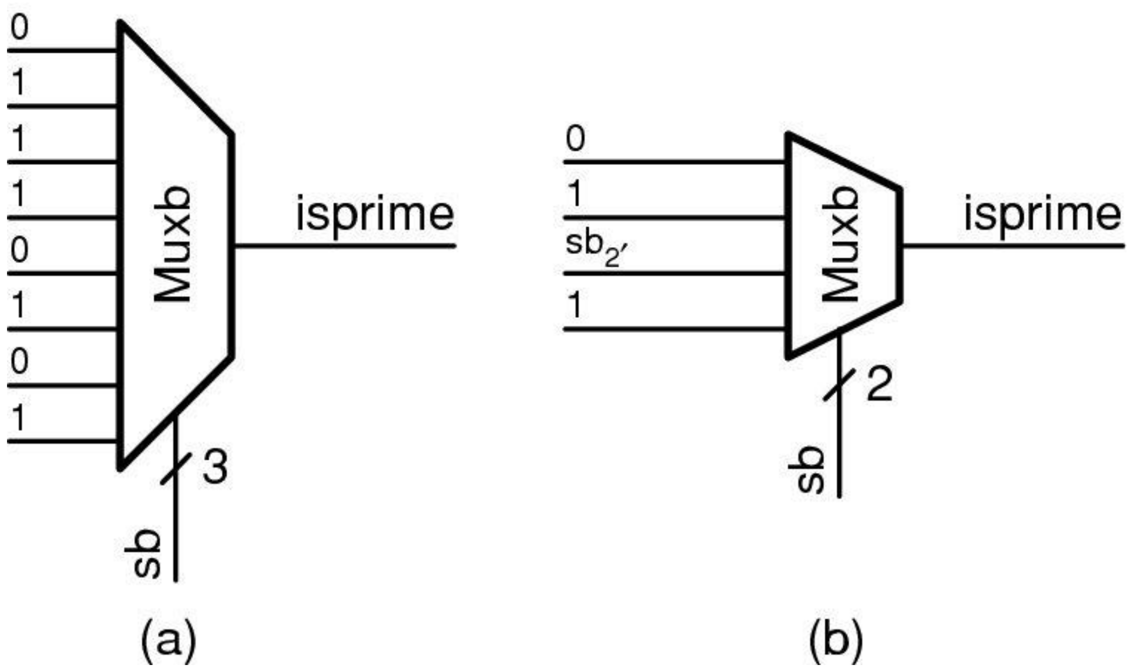
Implementing a large one-hot select mux using small ones

```
// 6:1 multiplexer with one-hot select (arbitrary width)
module Mux6a(a5, a4, a3, a2, a1, a0, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2, a3, a4, a5 ;  // inputs
  input [5:0]   s ;            // one-hot select
  output[k-1:0] b ;
  wire  [k-1:0] ba, bb ;

  assign b = ba | bb ;

  Mux3 #(k) ma(a2, a1, a0, s[2:0], ba) ;
  Mux3 #(k) mb(a5, a4, a3, s[5:3], bb) ;
endmodule
```

Implementing a large binary-select mux using small ones



Logic with Muxes



```
module Primem(in, isprime) ;
  input [2:0] in ;
  output      isprime ;

  Muxb8 #(1) m(1, 0, 1, 0, 1, 1, 1, 0, in, isprime) ;
endmodule
```

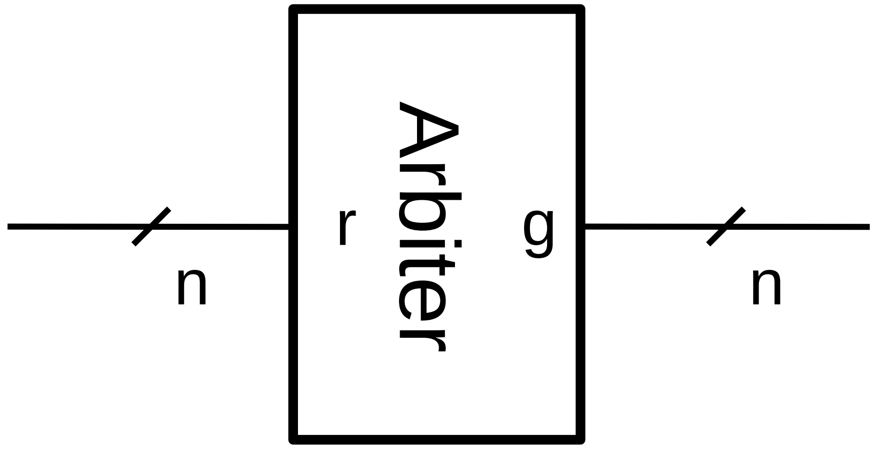
Summary

- Assemble combinational circuits from pre-defined building blocks
  - Decoder – converts codes (e.g., binary to one-hot)
  - Encoder – encodes one-hot to binary
  - Multiplexer – select an input (one-hot select)
  - Arbiter – pick first true bit
  - Comparators – equality and magnitude
  - ROMs
  - RAMs
  - PLAs
- Divide and conquer to build large units from small units
  - Decoder, encoder, multiplexer
- Logic with multiplexers or decoders
- Bit-slice coding style

Combinational Building Blocks - Arbiter

Arbiter

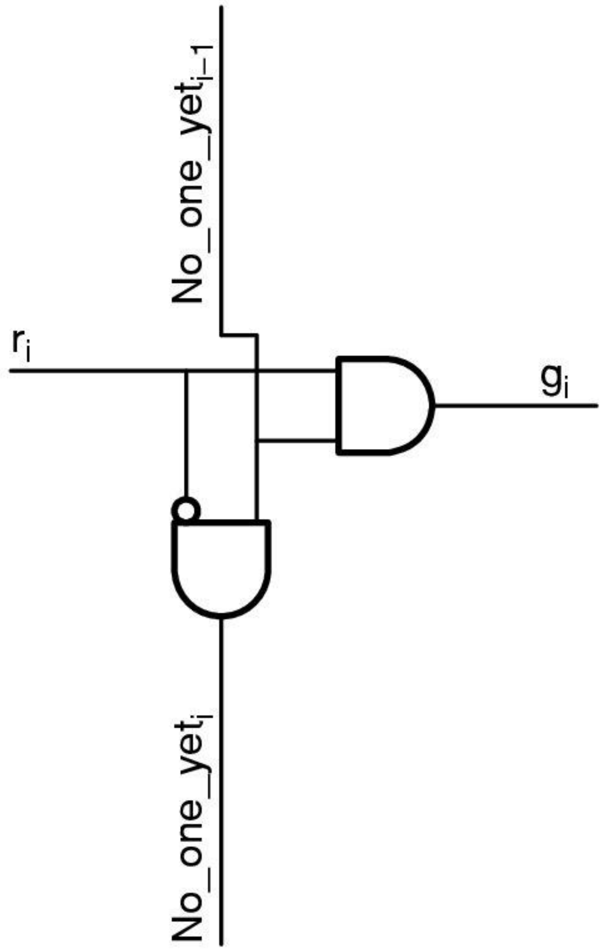
Arbiter handles requests from multiple devices to use a single resource



Finds first "1" bit in r

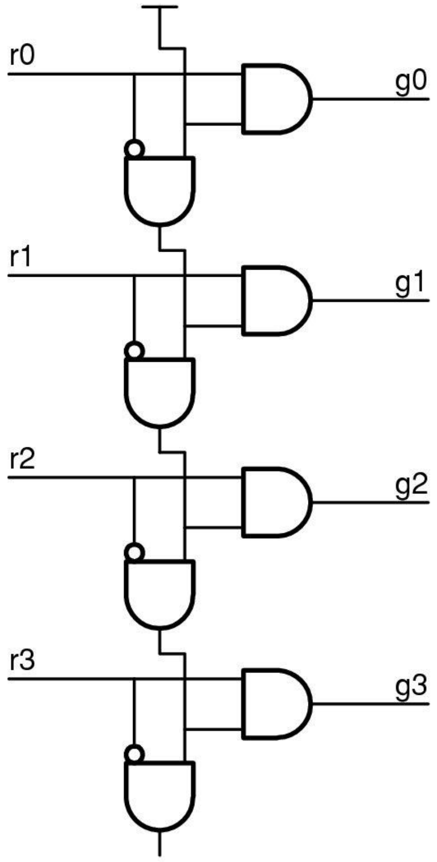
g[i]=1 if r[i]=1 and r[j]=0 for $j \geq i$

Logic Diagram of one Bit of an Arbiter



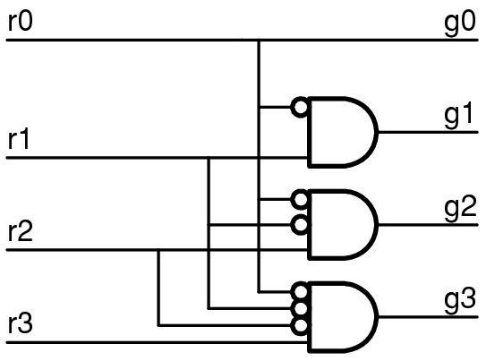
Two Implementations of a 4-bit Arbiter

Using bit-cell



(a)

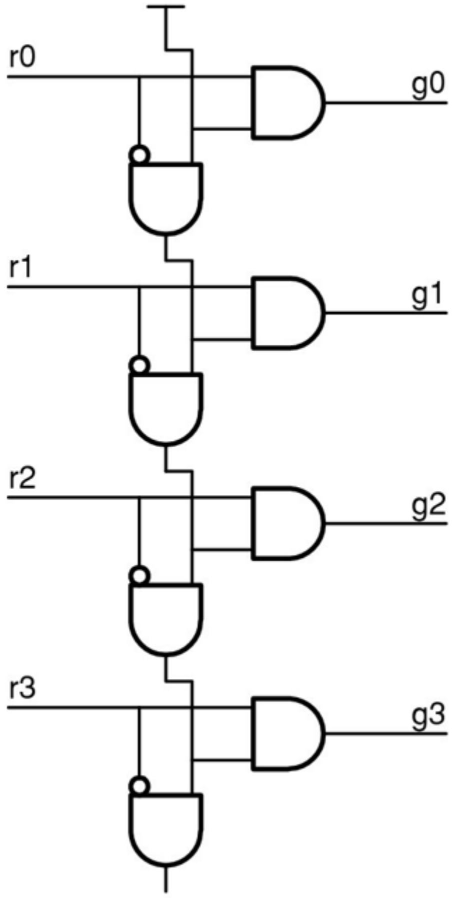
Using look-ahead



(b)

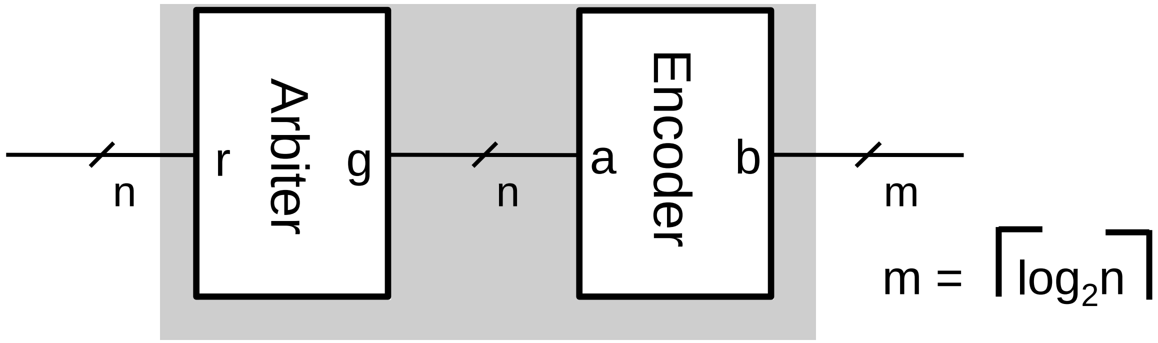
Arbiter using Verilog with LSB as the highest priority

```
// arbiter (arbitrary width)
module Arb(r, g) ;
  parameter n=8 ;
  input  [n-1:0] r ;
  output [n-1:0] g ;
  wire   [n-1:0] c ;

  assign c = {(~r[n-2:0] & c[n-2:0]),1'b1} ;

  assign g = r & c ;

endmodule
```

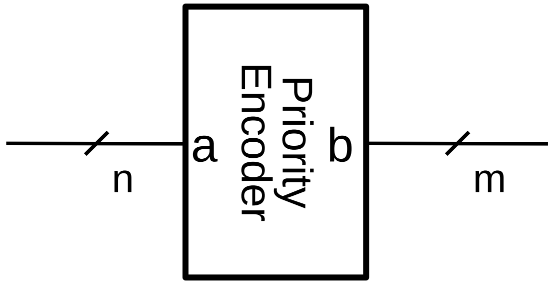


Priority Encoder

- Priority encoder:
  - n-bit input signal a
  - m-bit output signal b
    - b indicates the position of the first 1 bit in a

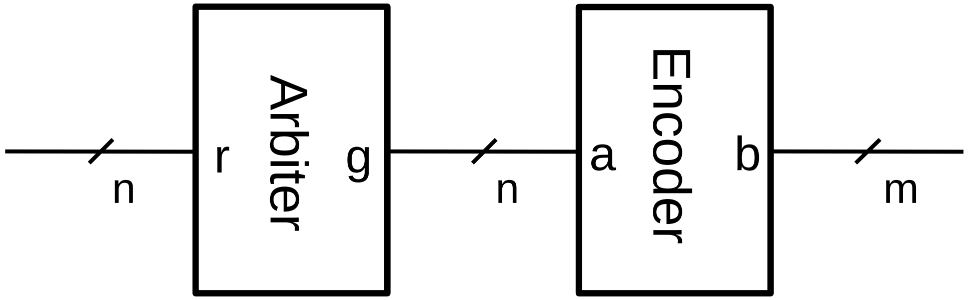


$$m = \lceil \log_2 n \rceil$$



Verilog for Priority Encoder

```
// priority encoder (arbitrary width)
module PriorityEncoder83(r, b) ;
  input  [7:0] r ;
  output [2:0] b ;
  wire   [7:0] g ;
  Arb #(8) a(r, g) ;
  Enc83    e(g, b) ;
endmodule
```

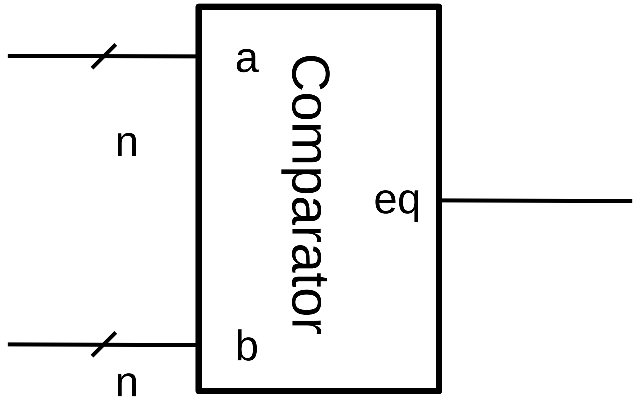


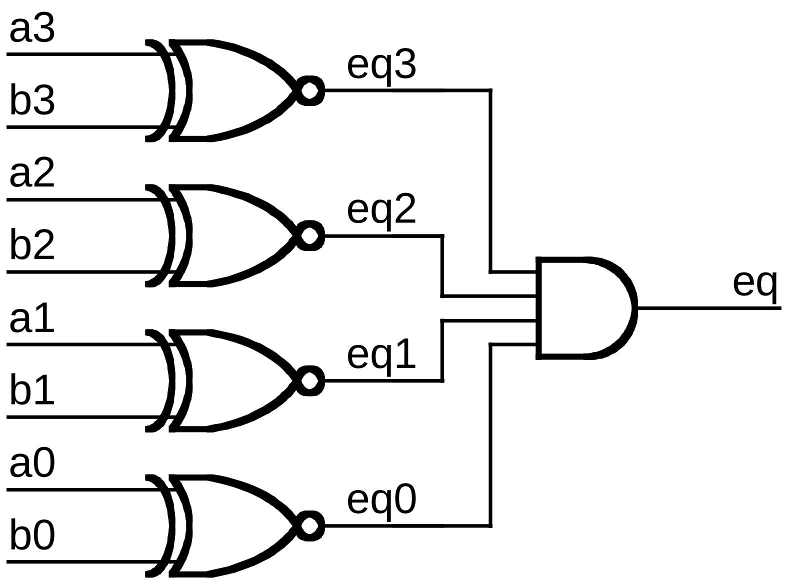
Summary

- Assemble combinational circuits from pre-defined building blocks
  - Decoder – converts codes (e.g., binary to one-hot)
  - Encoder – encodes one-hot to binary
  - Multiplexer – select an input (one-hot select)
  - Arbiter – pick first true bit
  - Comparators – equality and magnitude
  - ROMs
  - RAMs
  - PLAs
- Divide and conquer to build large units from small units
  - Decoder, encoder, multiplexer
- Logic with multiplexers or decoders
- Bit-slice coding style

Combinational Building Blocks
- Comparator and shifter

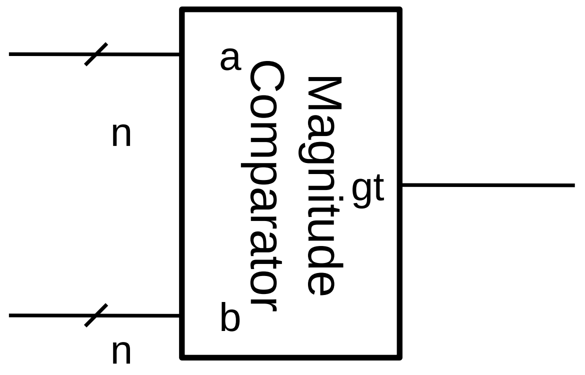
Equality Comparator

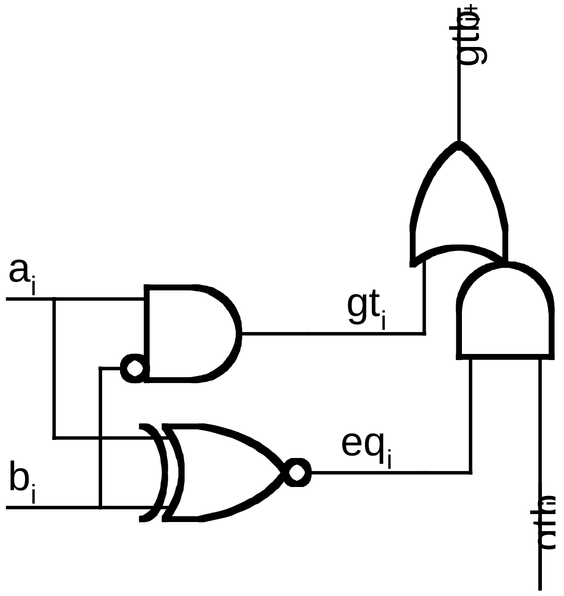




```
// equality comparator
module EqComp(a, b, eq) ;
  parameter k=8;
  input  [k-1:0] a, b;
  output eq ;

  assign eq = (a==b) ;
endmodule
```

Magnitude Comparator

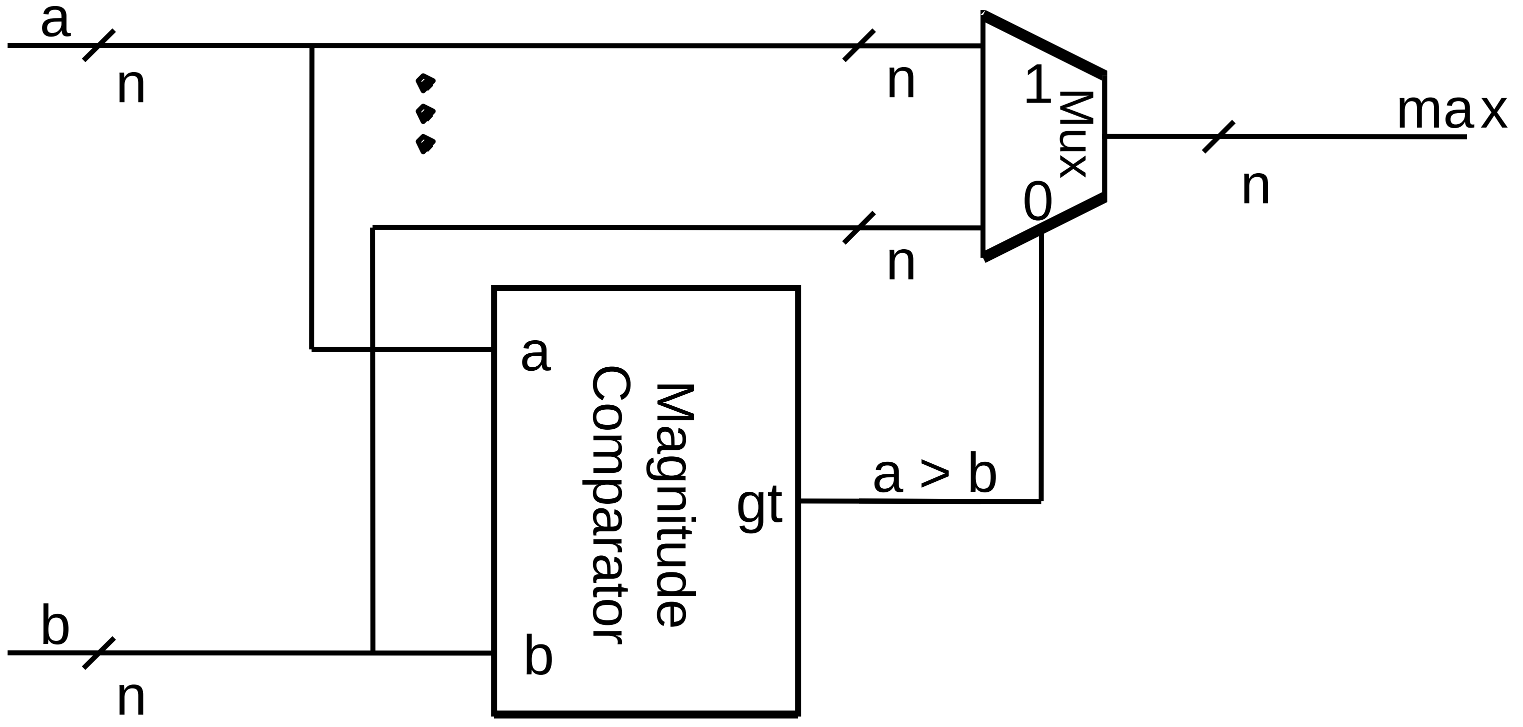




```
// magnitude comparator
module MagComp(a, b, gt) ;
  parameter k=8 ;
  input  [k-1:0] a, b ;
  output gt ;
  wire   [k-1:0] eqi = a ~^ b ;
  wire   [k-1:0] gti = a & ~b ;
  wire   [k:0]   gtb {((eqi[k-1:0] & gtb[k-1:0]) | gti[k-1:0]), 1'b0} ;
  wire   gt = gtb[k] ;
endmodule
```

```
// Behavioral Magnitude comparator
module MagComp_b(a, b, gt) ;
  parameter k=8 ;
  input  [k-1:0] a, b ;
  output gt ;

  assign   gt = (a > b) ;
endmodule
```

Putting things together – Maximum unit



Shifters

```
module ShiftLeft(n, a, b) ;
  parameter k=8 ;
  parameter lk=3 ;
  input  [lk-1:0] n ;
  input  [k-1:0] a ;
  output [2*k-2:0] b ;
  assign b = a<<n ;
endmodule
```

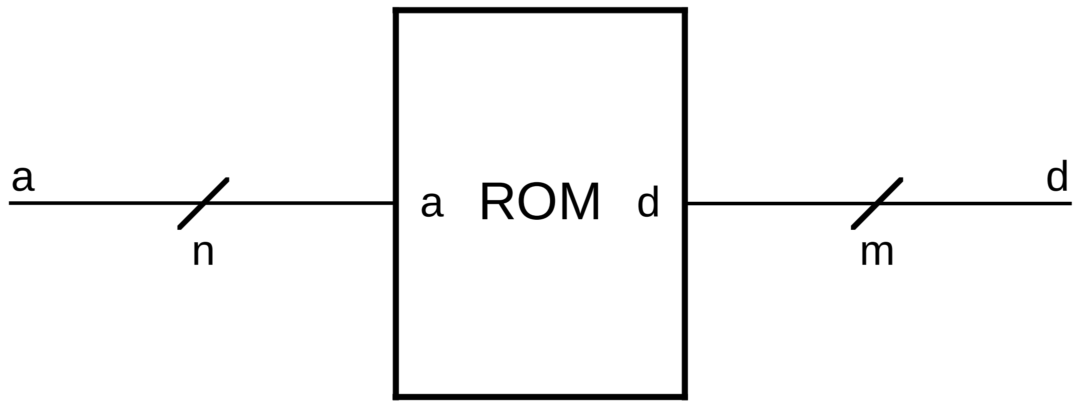
```
module BarrelLeftShift(n, a, b) ;
  parameter k=8 ;
  parameter lk=3 ;
  input  [lk-1:0] n ;
  input  [k-1:0] a ;
  output [k-1:0] b ;
  wire [2*k-2:0] x = a<<n ;
  assign b = x[k-1:0] | {1'b0, x[2*k-2:k]} ;
endmodule
```

Summary

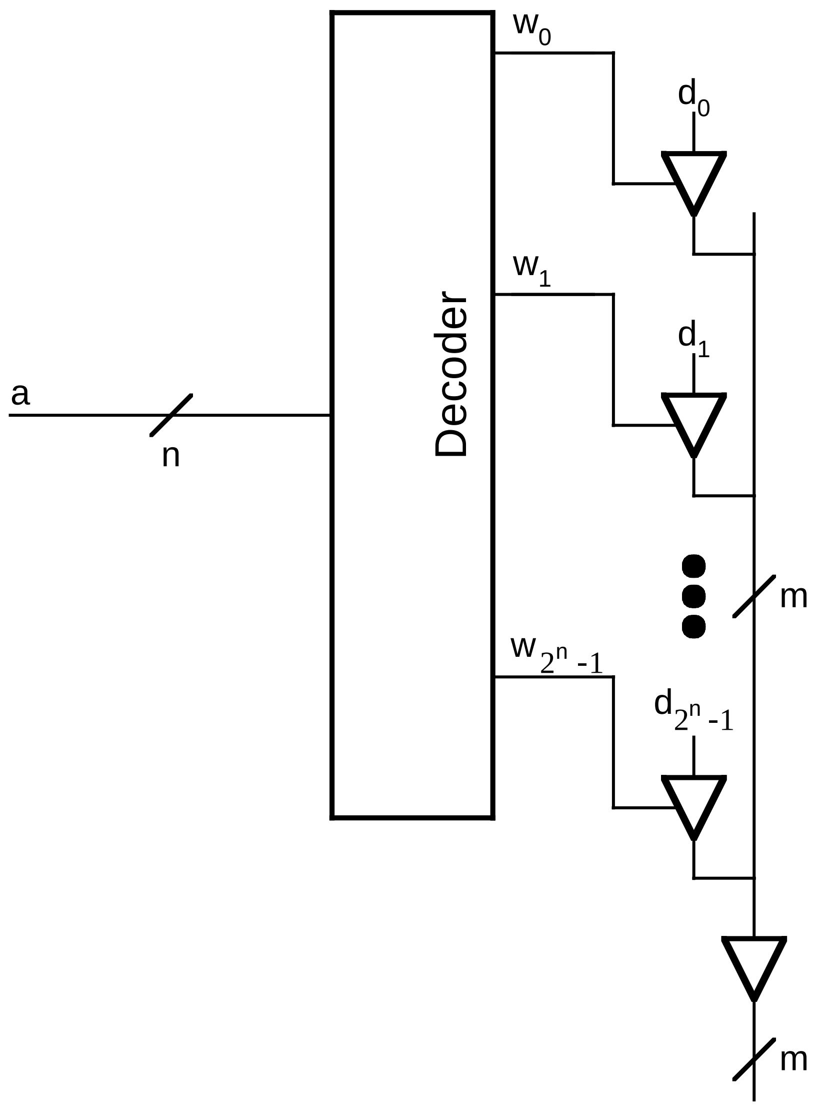
- Assemble combinational circuits from pre-defined building blocks
  - Decoder – converts codes (e.g., binary to one-hot)
  - Encoder – encodes one-hot to binary
  - Multiplexer – select an input (one-hot select)
  - Arbiter – pick first true bit
  - Comparators – equality and magnitude
  - Shifters
  - ROMs, RAMs
  - PLAs
- Divide and conquer to build large units from small units
  - Decoder, encoder, multiplexer
- Logic with multiplexers or decoders
- Bit-slice coding style

Combinational Building Blocks
- ROM, RAM, PLA

Read-only memory (ROM)

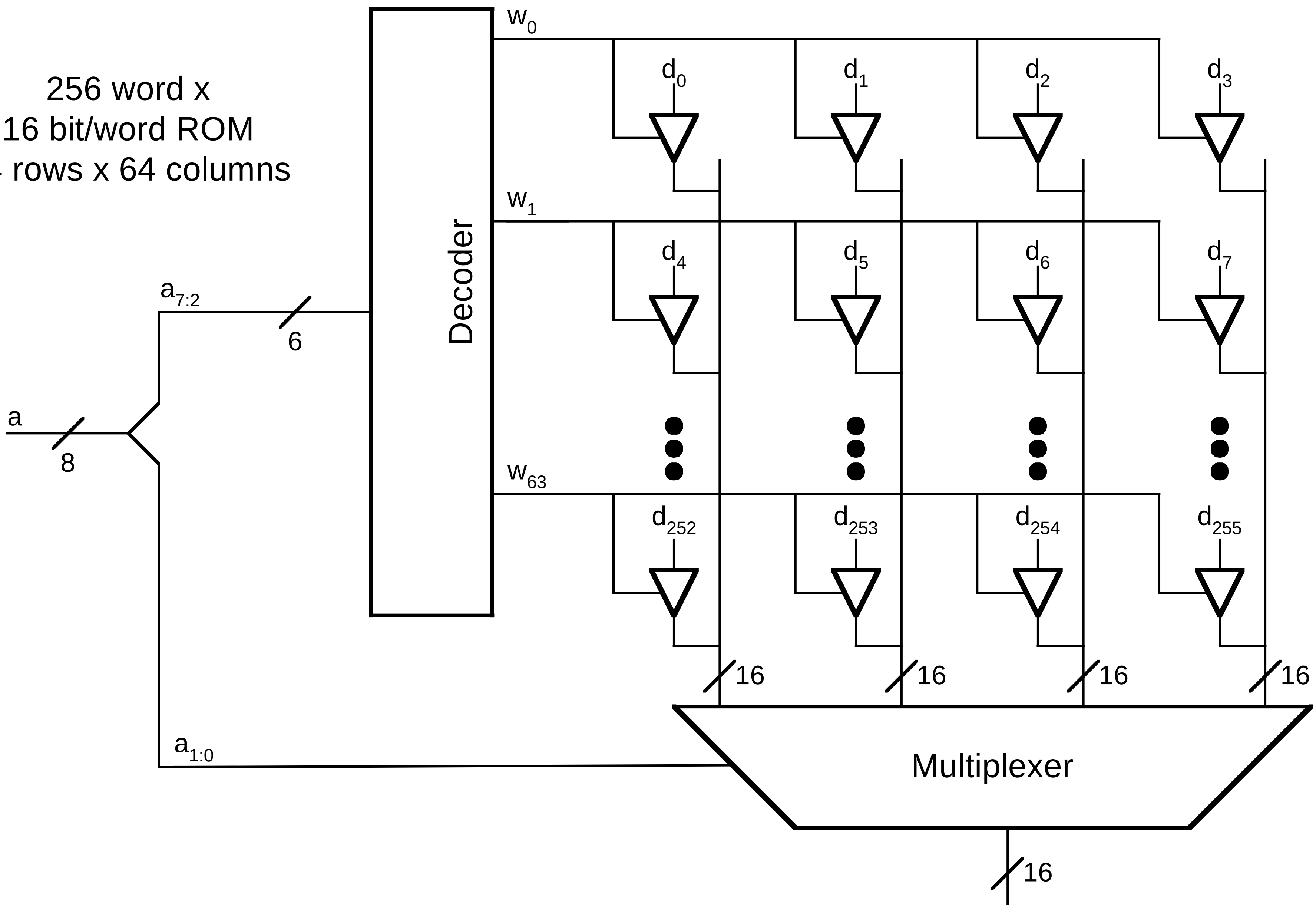


Conceptual block diagram

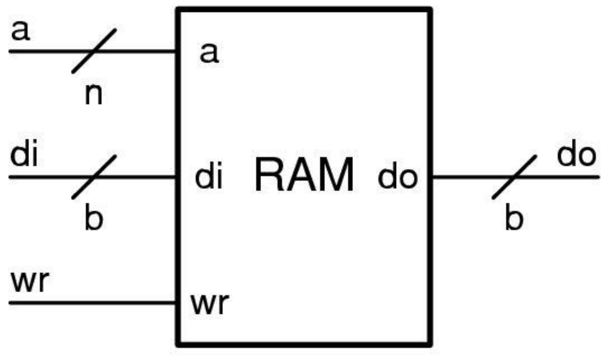


2-D array implementation

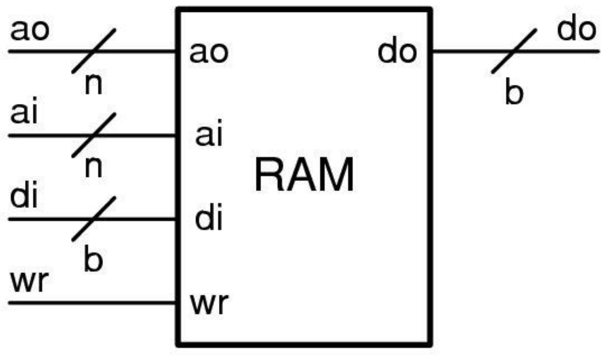
256 word x
16 bit/word ROM
64 rows x 64 columns



Read-write memory (RAM)

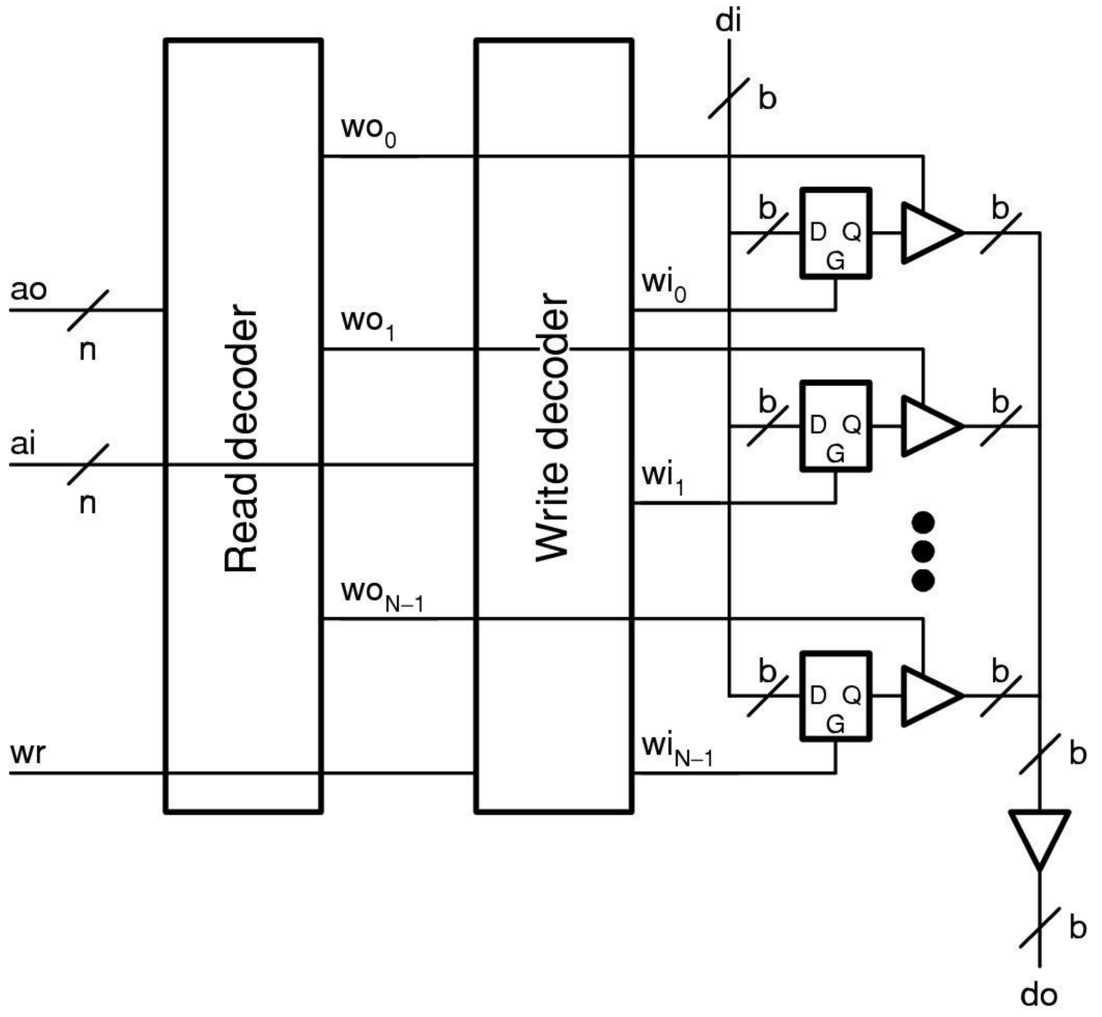


(a)

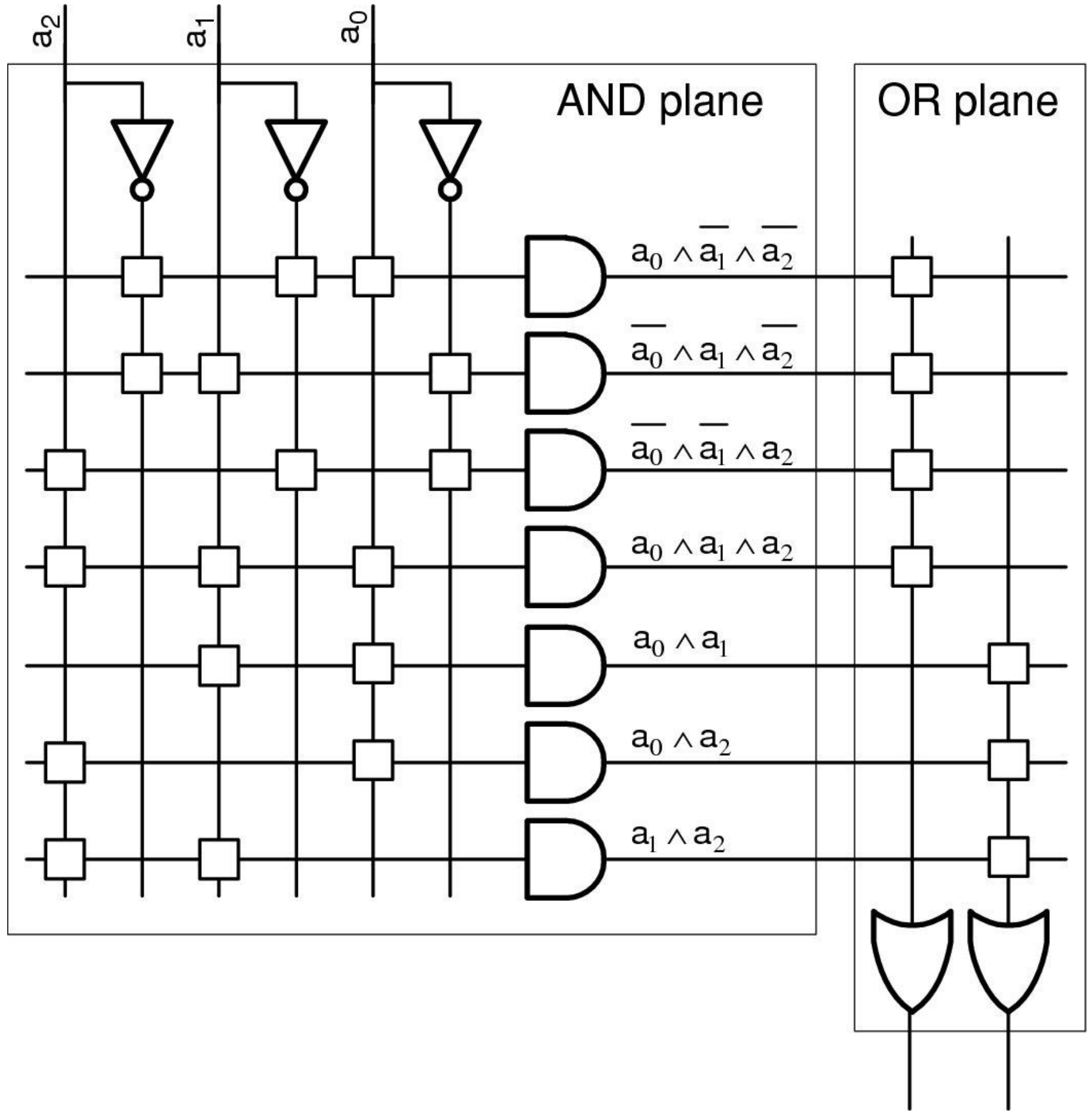


(b)

Dual-port RAM implementation



Programmable logic array



a_2
a_1
a_0
AND plane
OR plane
$a_0 \wedge \overline{a_1} \wedge \overline{a_2}$
$\overline{a_0} \wedge a_1 \wedge \overline{a_2}$
$\overline{a_0} \wedge \overline{a_1} \wedge a_2$
$a_0 \wedge a_1 \wedge a_2$
$a_0 \wedge a_1$
$a_0 \wedge a_2$
$a_1 \wedge a_2$