Switching theory and logic design

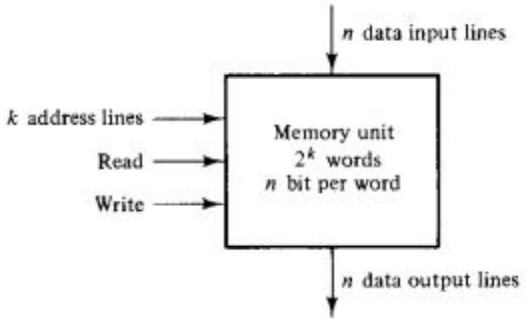
Module VI

Memory and Programmable Logic: Random-Access Memory (RAM)—Memory Decoding—Error Detection and Correction — Read only Memory (ROM), Programmable Logic Array (PLA).

HDL: fundamentals, combinational logic, adder, multiplexer.

Arithmetic algorithms: Algorithms for addition and subtraction of binary and BCD numbers, algorithms for floating point addition and subtraction

MEMORY AND PROGRAMMABLE LOGIC



nd from

which information is retrieved when needed for processing.

There are two types of memories that are used in digital systems: **random-access memory** (RAM) and **read-only memory** (ROM).

RAM can perform both **Write** and **Read** operations.

ROM can perform only the **Read** operation

RANDOM-ACCESS MEMORY

A memory unit is a collection of storage cells, together with associated circuits needed to transfer into and out of device. The architecture of memory is such that information can be selectively retrieved from any of its internal locations. The time it takes to transfer information to or from any desired random location is always the same hence the name random access memory. A memory unit stores binary information in groups of bits called words. A group of 8 bits is called a byte.

Write and Read Operations

Write to RAM

Apply the binary address of the desired word to the address lines

Apply the data bits that must be stored in memory to the data input lines

Activate the write control

Read from RAM

Apply the binary address of the desired word to the address lines

Activate the read control

Types of Memories

1. Access mode:
    a. Random access: any locations can be accessed in any order
    b. Sequential access: accessed only when the requested word has been reached (ex: hard disk)
2. Operating mode:
    a. Static RAM (SRAM)
    b. Dynamic RAM (DRAM)
3. Volatile mode:
    a. Volatile memory: lose stored information when power is turned off (ex: RAM)

sh, ROM,

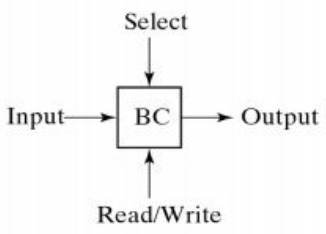
SRAM vs. DRAM

Static RAM:
- Use internal latch to store the binary information
- Stored information remains valid as long as power is on
- Shorter read and write cycles
- Larger cell area and power consumption

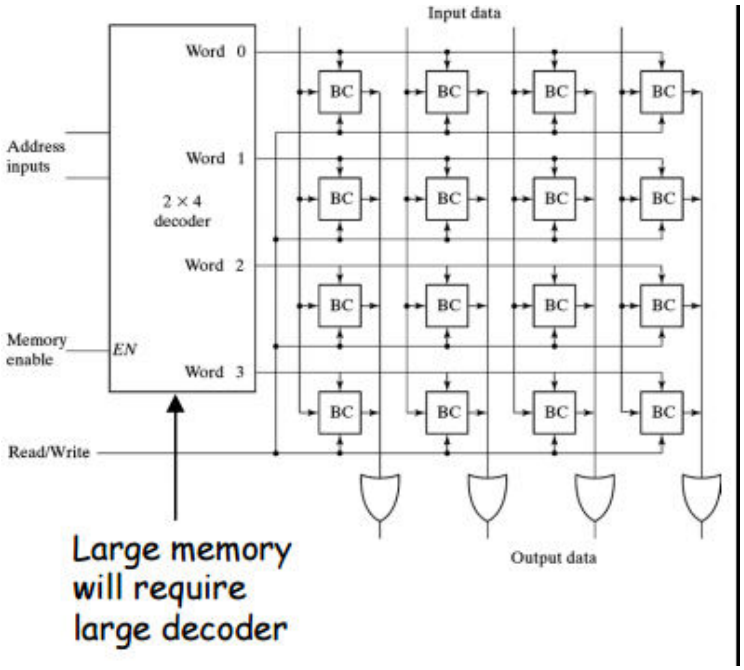
Dynamic RAM:
- Use a capacitor to store the binary information
- Need periodically refreshing to hold the stored info.
- Longer read and write cycles
- Smaller cell area and power consumption

MEMORY DECODING

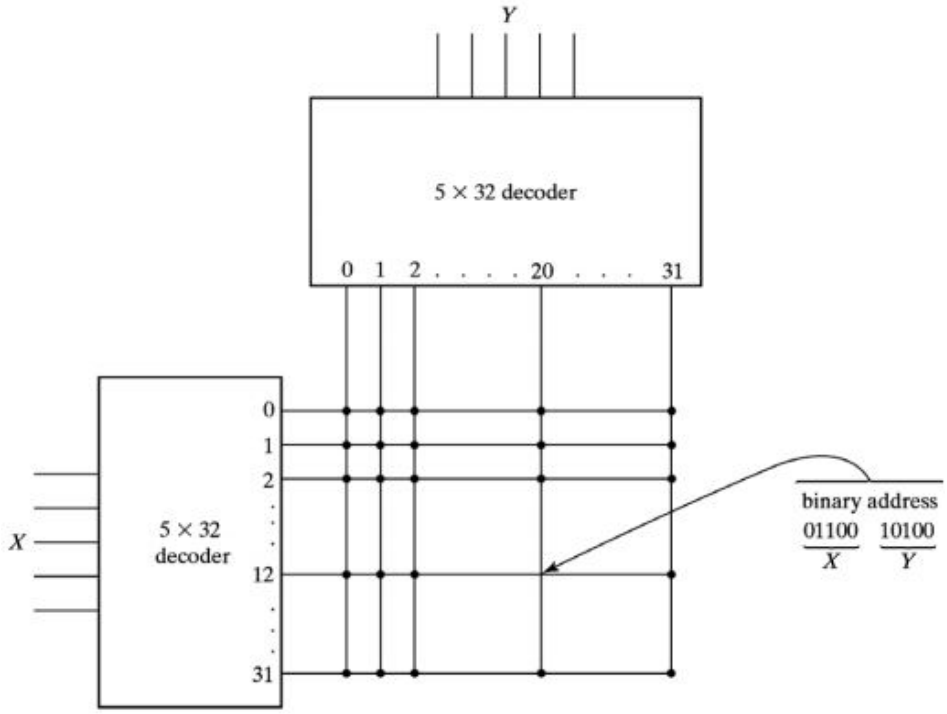
The equivalent logic of a binary cell that stores one bit of information is shown below



Memory construction 4x4 RAM



Two Dimensional Decoding Structure for a 1k word memory



ERROR DETECTION AND CORRECTION

Memory arrays are often very huge.May cause occasional errors in data access . Reliability of memory can be improved by employing error-detecting and correcting codes

Error-detecting code: only check for the existence of errors. Most common scheme is the parity bit

Error-correcting code: check the existence and locations of errors .Use multiple parity check bits to generate a syndrome that can indicate the erroneous bits .Complement the erroneous bits can correct the errors

Hamming Code

One of the most common used in RAM was devised by R. W. Hamming (called Hamming code).

In Hamming code: k = parity bits in n-bit data word,

forming a new word of n + k bits. Those positions numbered as a power of 2 are reserved for the parity bits. the remaining bits are the data bits.

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	X		X		X		X		X		X		X		X		X		X		...
	p2		X	X			X	X			X	X			X	X			X	X		
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

Error Detection and Correction Using Hamming Code

Encoding in the Hamming Code

Consider the 8-bit data word 11000100. we include four parity bits with it and arrange the 12 bits as follows:

Bit position:

D12	D11	D10	D9	P8	D7	D6	D5	P4	D3	P2	P1

D12 – 1 , D11 - 1 , D10 – 0 , D9 – 0 , D7 – 0 , D6 – 1 , D5 – 0 , D3 – 0

P1 = XOR of bits(3,5,7,9,11) = 0, 0, ,0, 0, 1 - 1

P2 = XOR of bits(3,6,7,10,11) = 0, 1 , 0, 0, 1 – 0

P4 = XOR of bits(5,6,7,12) = 0, 1,0,1 - 0

P8 = XOR of bits(9,10,11,12) – 0, 0, 1,1 – 0

Then the Encoded Data using Hamming Code = 110000100001

Error Detection

By the effect of noise the transmitted data 11000010000 01 changed 11100010000 01 in receiver side

The method of Error Detection is as follows

We know P1 = XOR of bits(3,5,7,9,11) = 0, 0, ,0, 0, 1 – 1 that means no error

P2 = XOR of bits(3,6,7,10,11) = 0, 1 , 0, 1, 1 – 0 error

P4 = XOR of bits(5,6,7,12) = 0, 1,0,1 – 0 – no error

P8 = XOR of bits(9,10,11,12) – 0, 1, 1,1 – 0 - Error

Error Correction

After checking the Parity bit we identified that there is an error in received bit. The next step is to correct that error. For error correction we need to find which bit contains the error. For this purpose

Assign 1 to Error found Parity bit other wise 0. In this case P2 and P8 shows Error and P1 and P4 shows no error. So P1=0, P4=0 and P2=1, P8 =1

That can be written as P8 P4 P2 P1 = 1010. Consider this number as binary and convert it in to decimal. 1010 = 10. The bit position 10 is the error bit. So change that bit.

READ ONLY MEMORY (ROM)

A memory device that can permanently keep binary data. Even when power is turned off and

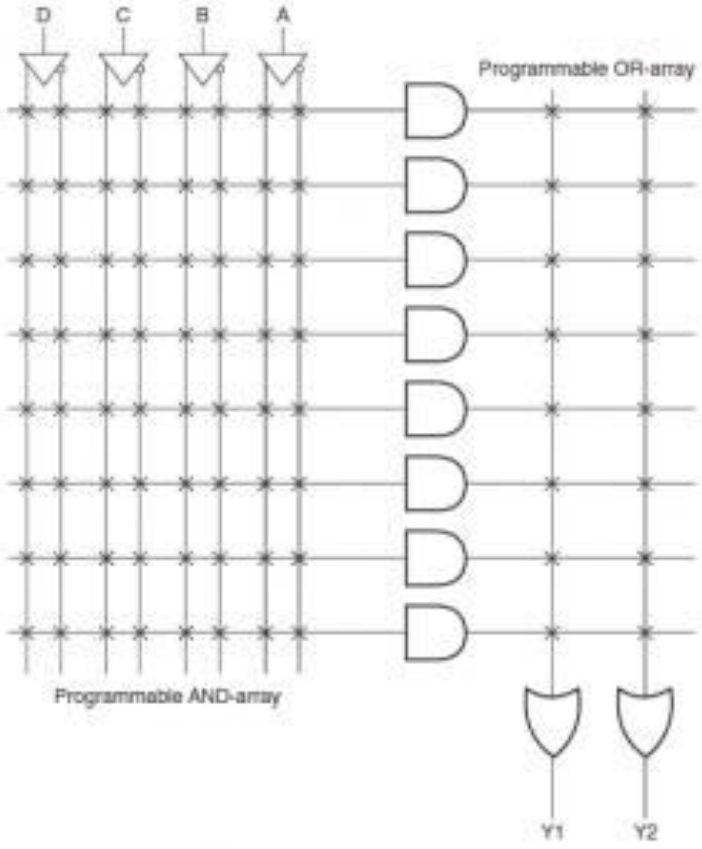
- Mask programming
  - Program the ROM in the semiconductor factory
  - Economic for large quantity of the same ROM
- Programmable ROM (PROM)
  - Contain all fuses at the factory ! Program the ROM by burning out the undesired fuses (irreversible process)
- Erasable PROM (EPROM)
  - Can be restructured to the initial state under a special ultraviolet light for a given period of time
- Electrically erasable PROM (EEPROM or E2PROM)
  - Like the EPROM except being erased with electrical signals

PROGRAMMABLE LOGIC ARRAY

A **programmable logic array** (PLA) is a type of fixed architecture logic device with programmable AND gates Followed by Programmable OR gates .

The number of AND gates in the programmable AND array are usually much less and the number of inputs of each of the OR gates equal to the number of AND gates. The OR gate generates an arbitrary Boolean function of minterms equal to the number of AND gates.

Figure below shows the PLA architecture with four input lines, a programmable array of eight AND gates at the input and a programmable array of two OR gates at the output.



Advantages

PLA architecture more efficient than a PROM.

Disadvantage

PLA architecture has two sets of programmable fuses due to which PLA devices are difficult to manufacture, program and test.

(Example Explained in Class Room)

HARDWARE DESCRIPTION LANGUAGES

In electronics, a **hardware description language** (**HDL**) is a specialized computer **language** used to describe the structure and behaviour of electronic circuits, and most commonly, digital logic circuits.

The two leading hardware description languages are *Verilog* and *VHDL.*

Verilog and VHDL are built on similar principles but have different syntax

Verilog

Verilog was developed by Gateway Design Automation as a proprietary language for logic simulation in 1984. Gateway was acquired by Cadence in 1989 and Verilog was made an open standard in 1990 under the control of Open Verilog International. The language became an IEEE standard1 in 1995 (IEEE STD 1364) and was updated in 2001.

A Verilog module begins with the module name and a listing of the inputs and outputs. The assign statement describes combinational logic. ~ indicates NOT, & indicates AND, and | indicates OR.

Verilog signals such as the inputs and outputs are Boolean variables (0 or 1).

```
assign y _ ~a & ~b & ~c |
a & ~b & ~c |
a & ~b & c;
endmodule
```

VHDL

VHDL is an acronym for the *VHSIC Hardware Description Language.* VHSIC is in turn an acronym for the *Very High Speed Integrated Circuits* program of the US Department of Defense. VHDL was originally developed in 1981 by the Department of Defense to describe the structure and function of hardware. Its roots draw from the Ada programming language. The IEEE standardized it in 1987 (IEEE STD 1076) and has updated the standard several times since. The language was first envisioned for documentation but was quickly adopted for simulation and synthesis

VHDL code has three parts: the library use clause, the entity declaration, and the architecture body. The entity declaration lists the module name and its inputs and outputs.
The architecture body defines what the module does.VHDL signals, such as inputs and outputs, must have a *type declaration.* Digital signals should be declared to be STD_LOGIC type. STD_LOGIC signals can have a value of '0' or '1', as well as floating and undefined values. The STD_LOGIC type is defined in the IEEE.STD_LOGIC_1164 library, which is why the library must be used. VHDL lacks a good default order of operations, so Boolean equations should be parenthesized

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity sillyfunction is
port (a, b, c: in STD_LOGIC;
y: out STD_LOGIC);
end;
architecture synth of sillyfunction is
begin
y __ ((not a) and (not b) and (not c)) or
(a and (not b) and (not c)) or
```

COMBINATIONAL LOGIC

Inverters

Verilog

```
module inv(input  [3:0] a,
           output [3:0] y);

  assign y = ~a;
endmodule
```

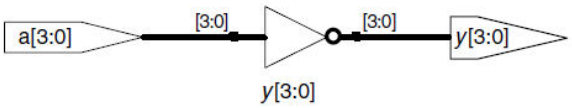
`a[3:0]` represents a 4-bit bus. The bits, from most significant to least significant, are `a[3]`, `a[2]`, `a[1]`, and `a[0]`. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have named the bus `a[4:1]`, in which case `a[4]` would have been the most significant. Or we could have used `a[0:3]`, in which case the bits, from most significant to least significant, would be `a[0]`, `a[1]`, `a[2]`, and `a[3]`. This is called *big-endian* order.

VHDL

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port(a: in  STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of inv is
begin
  y <= not a;
end;
```

VHDL uses `STD_LOGIC_VECTOR`, to indicate busses of `STD_LOGIC`. `STD_LOGIC_VECTOR (3 downto 0)` represents a 4-bit bus. The bits, from most significant to least significant, are 3, 2, 1, and 0. This is called *little-endian* order, because the least significant bit has the smallest bit number. We could have declared the bus to be `STD_LOGIC_VECTOR (4 downto 1)`, in which case bit 4 would have been the most significant. Or we could have written `STD_LOGIC_VECTOR (0 to 3)`, in which case the bits, from most significant to least significant, would be 0, 1, 2, and 3. This is called *big-endian* order.



Verilog

```
module gates(input  [3:0] a, b,
             output [3:0] y1, y2,
                          y3, y4, y5);

  /* Five different two-input logic
     gates acting on 4 bit busses */
  assign y1 = a & b;    // AND
  assign y2 = a | b;    // OR
  assign y3 = a ^ b;    // XOR
  assign y4 = ~(a & b); // NAND
  assign y5 = ~(a | b); // NOR
endmodule
```

`~`, `^`, and `|` are examples of Verilog *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a & b`, or `~(a | b)`, is called an *expression*. A complete command such as `assign y4 = ~(a & b);` is called a *statement*.

`assign out = in1 op in2;` is called a *continuous assignment statement*. Continuous assignment statements end with a semicolon. Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe combinational logic.

VHDL

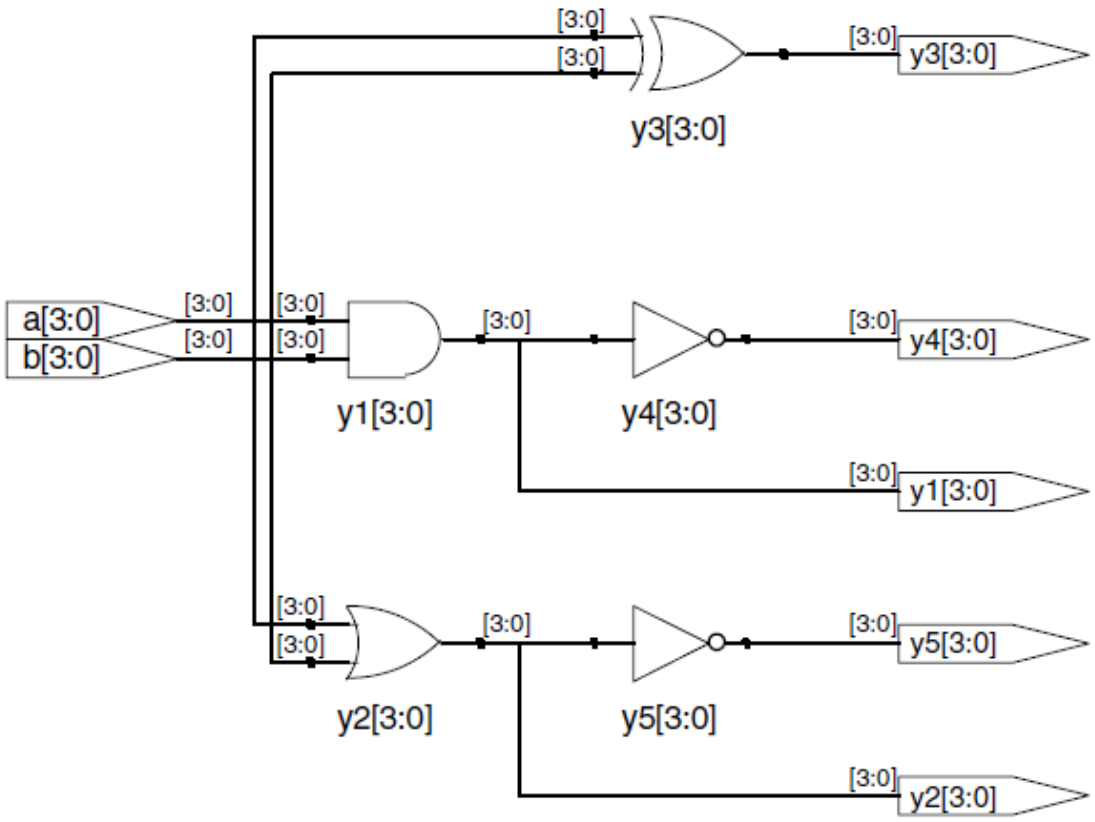
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity gates is
  port(a, b: in  STD_LOGIC_VECTOR(3 downto 0);
       y1, y2, y3, y4,
       y5:   out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth of gates is
begin
  -- Five different two-input logic gates
  --acting on 4 bit busses
  y1 <= a and b;
  y2 <= a or b;
  y3 <= a xor b;
  y4 <= a nand b;
  y5 <= a nor b;
end;
```

`not`, `xor`, and `or` are examples of VHDL *operators*, whereas `a`, `b`, and `y1` are *operands*. A combination of operators and operands, such as `a and b`, or `a nor b`, is called an *expression*. A complete command such as `y4 <= a nand b;` is called a *statement*.

`out <= in1 op in2;` is called a *concurrent signal assignment statement*. VHDL assignment statements end with a semicolon. Anytime the inputs on the right side of the <= in a concurrent signal assignment statement change, the output on the left side is recomputed. Thus, concurrent signal assignment statements describe combinational logic.



gates **synthesized circuit**

Verilog

A 4:1 multiplexer can select one of four inputs using nested conditional operators.

```
module mux4 (input  [3:0] d0, d1, d2, d3,
             input  [1:0] s,
             output [3:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                  : (s[0] ? d1 : d0);
endmodule
```

If `s[1]` is 1, then the multiplexer chooses the first expression, `(s[0] ? d3 : d2)`. This expression in turn chooses either `d3` or `d2` based on `s[0]` (`y = d3` if `s[0]` is 1 and `d2` if `s[0]` is 0). If `s[1]` is 0, then the multiplexer similarly chooses the second expression, which gives either `d1` or `d0` based on `s[0]`.

VHDL

A 4:1 multiplexer can select one of four inputs using multiple `else` clauses in the conditional signal assignment.

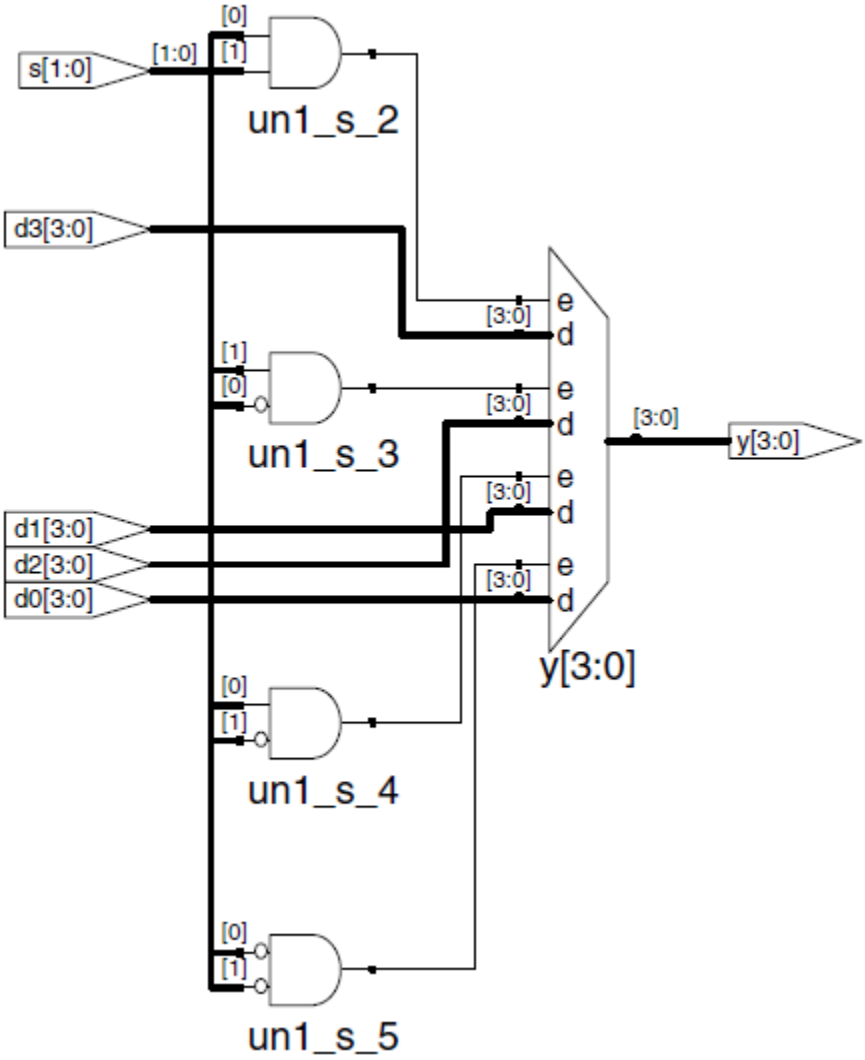
```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity mux4 is
  port(d0, d1,
       d2, d3:  in  STD_LOGIC_VECTOR(3 downto 0);
       s:       in  STD_LOGIC_VECTOR(1 downto 0);
       y:       out STD_LOGIC_VECTOR(3 downto 0));
end;

architecture synth1 of mux4 is
begin
  y <= d0 when s = "00" else
       d1 when s = "01" else
       d2 when s = "10" else
       d3;
end;
```

VHDL also supports *selected signal assignment statements* to provide a shorthand when selecting from one of several possibilities. This is analogous to using a `case` statement in place of multiple `if/else` statements in some programming languages. The 4:1 multiplexer can be rewritten with selected signal assignment as follows:

```
architecture synth2 of mux4 is
begin
  with a select y <=
    d0 when "00",
    d1 when "01",
    d2 when "10",
    d3 when others;
```

Mux synthesized circuit



ADDER

Verilog

In Verilog, *wires* are used to represent internal variables whose values are defined by `assign` statements such as `assign p = a ^ b;` Wires technically have to be declared only for multibit busses, but it is good practice to include them for all internal variables; their declaration could have been omitted in this example.

```
module fulladder(input  a, b, cin,
                 output s, cout);

  wire p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g  |  (p & cin);
endmodule
```

VHDL

In VHDL, *signals* are used to represent internal variables whose values are defined by *concurrent signal assignment statements* such as `p <= a xor b;`

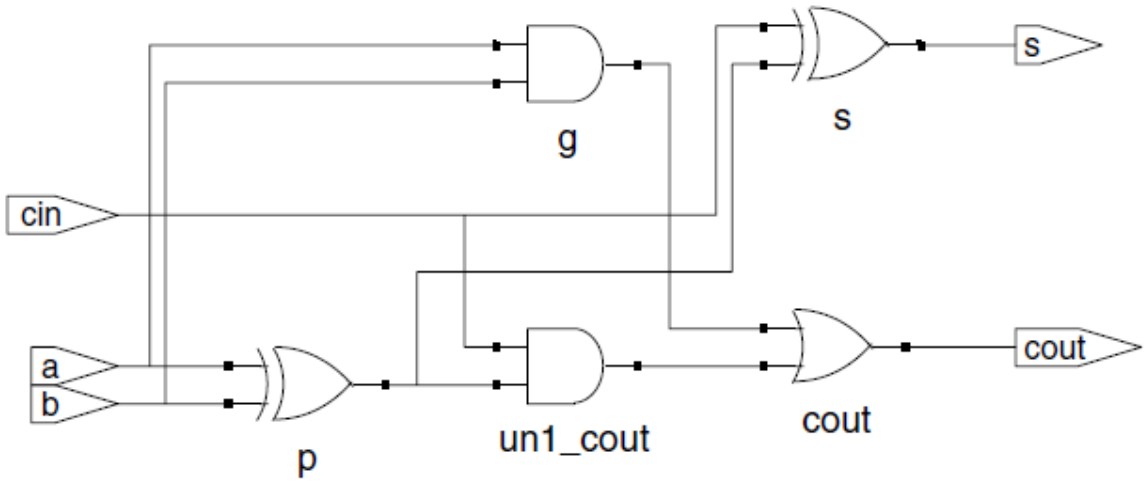
```
library IEEE; use IEEE.STD_LOGIC_1164.all;

entity fulladder is
  port(a, b, cin: in  STD_LOGIC;
       s, cout:   out STD_LOGIC);
end;

architecture synth of fulladder is
  signal p, g: STD_LOGIC;
begin
  p <= a xor b;
  g <= a and b;

  s <= p xor cin;
  cout <= g or (p and cin);
end;
```

Full adder Synthesized circuit



ARITHMETIC ALGORITHMS:

Addition and Subtraction with Signed-Magnitude Data The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations. The procedure for adding or subtracting two signed binary numbers with paper and pencil is simple and straightforward.

A review of this procedure will be helpful for deriving the hardware algorithm. We designate the magnitude of the two numbers by A and B. When the signed numbers are added or

on the sign

of the numbers and the operation performed. These conditions are listed in the first column of Table . The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to prevent a negative zero.

In other words, when two equal numbers are subtracted, the result should be +0 not -0. The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words inside parentheses should be used for the subtraction algorithm): Addition (subtraction) algorithm: when the signs of A and B are identical (different), add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different (identical), compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive. The two algorithms are similar except for the sign comparison. The procedure to be

followed for identical signs in the addition algorithm is the same as for different signs in the subtraction algorithm, and vice versa

Addition and Subtraction of Signed-Magnitude Numbers

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

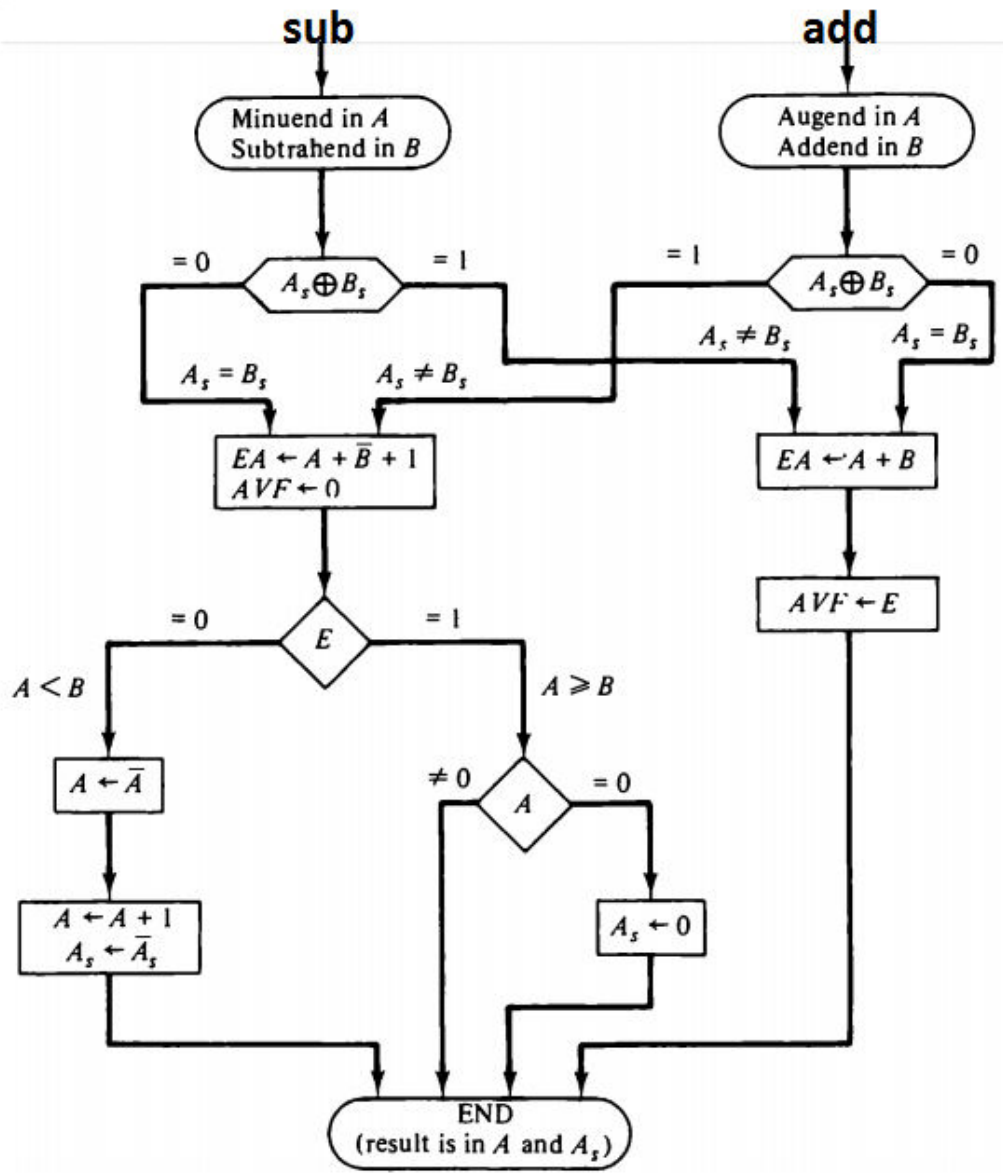
Algorithm Explanation

The flowchart for the hardware algorithm is presented in Fig. . The two signs A, and B,are compared by an exclusive-OR gate. If the output of the gate isO, the signs are identical; if it is I, thesignsaredifferenl Foranelddoperation, identical signs dictate that the magnitudes be added. For a subtrlld operation, different signs dictate that the magnitudes be added. The magnitudes are added with a mlcrooperation EA .-A + B. where EA is a register that

equal to

nitudes are

subtracted if the signs are different for an Cldd operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complement of B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0. A 1 in E indicates that A ? B a.nd the number in A is the correct result. If this nu.mber i.s zero, the sign A, must be made positive to avoid a negative zero. A 0 in E: indicates that A < B. For this case it is necessa ry to take the 2's complement of the value in A. This operation can be done with one mlcrooperation A .-A"+ 1. However, we assume that the A register has circuits for mlcrooperations complm.mt and inlTtment, so the 2' s complement is obtained from these two mlcroo perations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in A, is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign. The final result is found in register A and its sign in A,. The value inAVF provides an overflow indication. The final value of E is immaterial.



Description

- A_S Sign of A
- B_S Sign of B
- A_S & A Accumulator
- AVF Overflow bit for $A + B$
- E Output carry for parallel adder

ALGORITHM FOR FLOATING POINT ADDITION AND SUBTRACTION

During addition or subtraction, the two floating-point operands are in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

A floating-point number that is zero cannot be normalized. If this number is used during the computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be unnormalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory. The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 10-15. If BR is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer the content of BR into AC and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b provides three outputs

orm the

r exponent

is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal. The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E is equal to 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position. If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the AC is made zero. Otherwise, the mantissa must have at least one bit that is equal to 1. The mantissa has an underflow if the most significant bit in position A1 is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until it is equal to 1. When A1 = 1, the mantissa is normalized and the operation is completed.