EL 6463
Advanced Hardware Design

```
ENTITY function1 IS
      PORT (
            C, x, y: IN  STD_LOGIC;
            s, p   : OUT   STD_LOGIC
      ) ;
END function1 ;

ARCHITECTURE abc OF function1 IS
BEGIN
   s <= x XOR y XOR C ;
   p <= (x AND y) OR (C AND x) OR (C AND y) ;
END abc
```

Draw the gate level diagram.
Rewrite P or Rewrite S using with-select construct
Can you guess what this function is?

```
module function2(
        input wire a,b, c_in,
        output wire q, c_out
);

assign q = a ^ b ^ c_in;
assign c_out = (a & b) | (q & c_in);

endmodule
```

Full Adder
Less Gate Required

Draw the gate level diagram.
Can you guess what this function is?
Are there any differences with the previous VHDL?

Quiz

1. Describe entity definition for a 5-bit input and a 4-bit output gate.
2. If b is 1, output = input. If b is 0, output= complement of input. write a vhdl architecture for this function. Assume that the input is 2 bits wide.
3. Now rewrite this in Verilog.

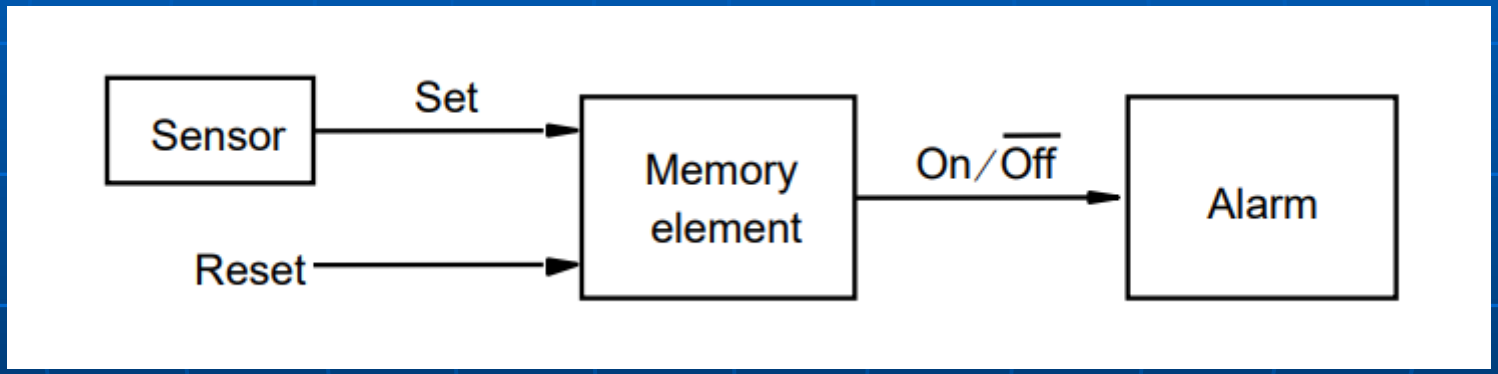
Remaining

Key concepts in modeling sequential circuits in VHDL/Verilog

flip flops, registers, synchronous, asynchronous, rising edge clock, falling edge clock, active high, active low signals……

Why have sequential circuits?

- Sometimes digital systems need to remember things:



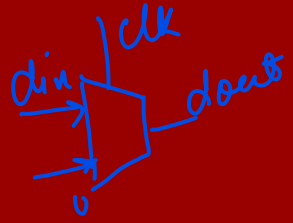
E.g. alarm system to sense burglar and maintain alarm signal until reset:
The **OUTPUT** depends on its **STATE**

Towards a 1-bit register (if-then-else)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 1_bit_notareg IS
 PORT  ( clk: IN STD_LOGIC;  -- Clock/enable signal
          din: IN STD_LOGIC);--1-bit input
          dout: OUT STD_LOGIC) --1-bit output
  );
END 1_bit_notareg;

ARCHITECTURE RTL OF 1_bit_notareg IS
BEGIN
PROCESS (din,clk)  BEGIN
IF (clk='1')
    THEN dout<=din;
ELSE
    dout <='0';
END IF;
END PROCESS;
END RTL
```

Sensitivity list

* Anything sequential needs to be inside Process and HDL's synthesise



STLA Blocks

A Clock-gated Latch

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY latch IS
 PORT  ( clk: IN STD_LOGIC;  -- Clock/enable signal
          din: IN STD_LOGIC;--1-bit input
          dout: OUT STD_LOGIC --1-bit output
  );
END latch;

ARCHITECTURE RTL OF latch IS
BEGIN
PROCESS (din,clk)  BEGIN
IF (clk='1') THEN
  dout<=din;
ELSE
 dout <='0';
END IF;
END PROCESS;
END RTL
```

due to this it is a memory block becomes a Latch

Latch or ?

A Clocked Flip Flop

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY flipflop IS
 PORT  ( clk: IN STD_LOGIC;  -- Clock/enable signal
          din: IN STD_LOGIC;--1-bit input
          dout: OUT STD_LOGIC --1-bit output
  );
END flipflop;

ARCHITECTURE RTL OF flipflop IS
BEGIN
PROCESS (din,clk)  BEGIN
IF (clk'EVENT AND clk='1') THEN
  dout<=din;
END IF;
END PROCESS;
END RTL
```

* what is the difference b/w all of these?

← Rising Edge

⇒ Model them all and study them

STD Revision

1-bit register (asynchronous clear)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 1_bit_reg IS
 PORT  ( clr: IN STD_LOGIC;  -- asynchronous reset
          clk: IN STD_LOGIC;  -- Clock signal
          din: IN STD_LOGIC;--1-bit input
          dout: OUT STD_LOGIC --1-bit output
  );
END 1_bit_reg;

ARCHITECTURE RTL OF 1_bit_reg IS
BEGIN
PROCESS (clr, clk)  BEGIN
  IF (clr='1') THEN dout <= 0; ---active high clear
  ELSIF (clk'EVENT AND clk='1') THEN dout<=din; --positive edge trigger
  END IF;
END PROCESS;
END RTL
```

1-bit register (synchronous clear)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 1_bit_reg IS
 PORT  ( clr: IN STD_LOGIC;  -- asynchronous reset
          clk: IN STD_LOGIC;  -- Clock signal
          din: IN STD_LOGIC);--1-bit input
          dout: OUT STD_LOGIC) --1-bit output
  );
END 1_bit_reg;

ARCHITECTURE RTL OF 1_bit_reg IS
BEGIN
PROCESS (clk)  BEGIN
  IF (clk'EVENT AND clk='1')
  then IF (clr='0') THEN dout <= 0;
        ELSE dout<=din;
        END IF;
 END IF;
END PROCESS;
END RTL
```

Active Low

VHDL signal data object

Signal

- ✓ Models a physical wire or a data storage element
- ✓ If not initialized, default value is the left-most value of its type
- ✓ Ports are signals
- ✓ "<=" is used for signal assignment
- ✓ "event" attribute detects change in signal (high to low or low to high)

✓ Some Predefined VHDL Data Types

- ✓ Bit: '0' or '1'
- ✓ Boolean: FALSE or TRUE
- ✓ std_logic and std_logic_vector
- ✓ Integer: range $-(2^{-31}-1)$ to $+(2^{+31}-1)$

Towards Verilog Register (if-then-else)

```
module 1_bit_notareg (
        input wire clk, //clock/enable signal
        input wire din, //1-bit input
        output reg dout //1-bit output – notice use of REG
);

always @(din, clk) begin
        if (clk==1) begin
                dout<=din;
        end else begin
                dout <=0;
        end
end

endmodule
```

→ not register

⇒ if you need to assign some value to output inside of always block need to be reg

⇒ Reg can be register but not always have to be

Verilog		VHDL
always	*functions like*	*process*
reg	*functions like*	*signal*

A Clock-gated Latch

```
module latch(
        input wire clk, //clock/enable signal
        input wire din, //1-bit input
        output reg dout //1-bit output – notice use of REG
);

always @(din, clk) begin
        if (clk==1) begin
                dout<=din;
        end else begin
                dout <=0;
        end
end

endmodule
```

→ Non clocked are Latch ?

A Clocked Flip Flop

```
module flipflop(
        input wire clk, //clock/enable signal
        input wire din, //1-bit input
        output reg dout //1-bit output – notice use of REG
);

always @(posedge clk) begin
        if (clk==1) begin
                dout<=din;
        end
end

endmodule
```

Verilog
posedge clk *functions like*

VHDL
clk'event and clk='1'
OR
rising_edge(clk)

1-bit register (asynchronous clear)

```
module 1_bit_reg (
        input wire clk, //clock/enable signal
        input wire rst, //reset signal
        input wire din, //1-bit input
        output reg dout //1-bit output – notice use of REG
);

always @(posedge rst or posedge clk) begin
        if (rst==1) begin
                dout <= 0;
        end else begin
                dout <= din;
        end
end

endmodule
```

Verilog will use *posedge/=1* on **active high** asynchronous resets, *negedge/=0* on **active low** asynchronous resets

1-bit register (synchronous clear)

```
module 1_bit_reg (
        input wire clk, //clock/enable signal
        input wire rst, //reset signal
        input wire din, //1-bit input
        output reg dout //1-bit output – notice use of REG
);

always @(posedge clk) begin
        if (rst==1) begin
                dout <= 0;
        end else begin
                dout <= din;
        end
end

endmodule
```

Synchronous reset very easy in Verilog.

Verilog reg variable object

Reg

- ✓ Models a physical wire or data storage element
- ✓ If not initialized, default value is the left-most value of its type
- ✓ Ports are only Reg type when declared (default is Wire)
- ✓ "<=" is used for **non-blocking** assignment
  - ✓(function like VHDL signal)
- ✓ "=" is used for **blocking** assignment
- ✓posedge/negedge for change in signal (high to low or low to high)
- ✓ Can take any of the std_logic types
- ✓ Integers and other Verilog types function the same as Reg

⇒ Everything is STD_logic in Verilog:

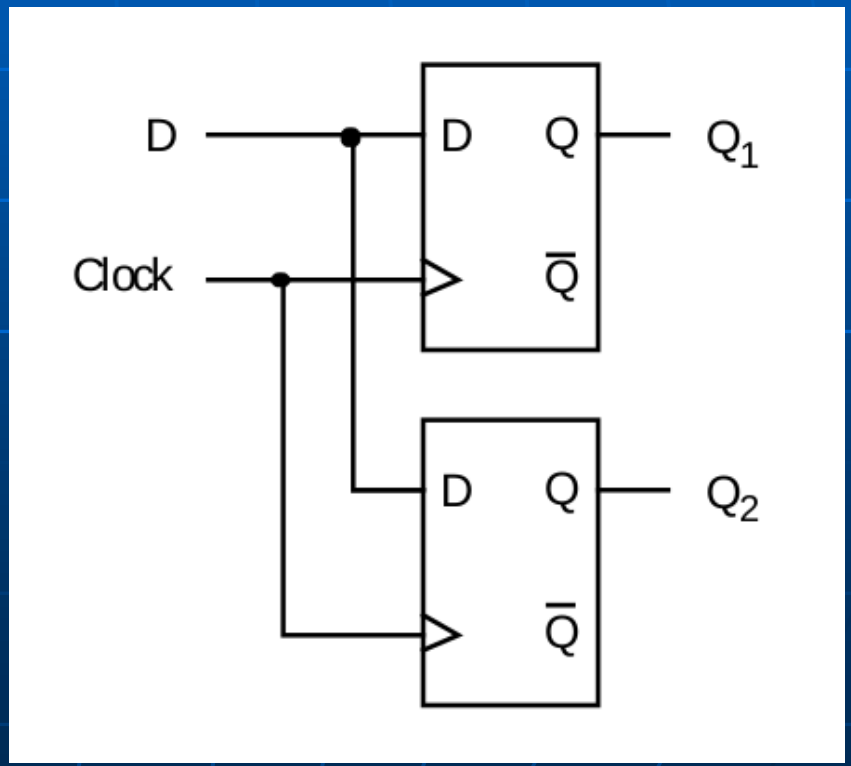
diff in Latch/Mux/FF

Lvl triggered FF is a Latch

Verilog reg BLOCKING vs NON-BLOCKING

```
module blockingRegs(
        input wire Clock, D,
        output reg Q1, Q2
);

always @(posedge Clock)
begin
        Q1 = D;          //blocking assignment
        Q2 = Q1;         //blocking assignment
end
endmodule
```

The "=" operator is blocking
It functions **IMMEDIATELY**

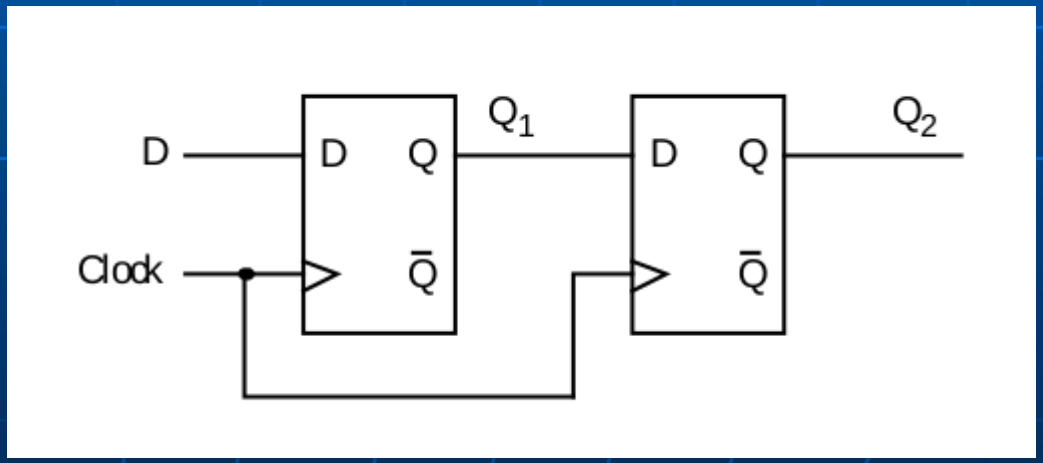


Q1 and Q2 will have the SAME VALUE

Verilog reg BLOCKING vs NON-BLOCKING

```
module nonBlockingRegs(
        input wire Clock, D,
        output reg Q1, Q2
);

always @(posedge Clock)
begin
        Q1 <= D;        //non-blocking assignment
        Q2 <= Q1;       //non-blocking assignment
end
endmodule
```

The "<=" operator is **non-blocking**
It functions **AT THE END** of always



Q2 will always be ONE CLOCK CYCLE behind Q1

VHDL BLOCKING vs NON-BLOCKING

VHDL simplifies matters

- ✓ SIGNALS are always NON-BLOCKING

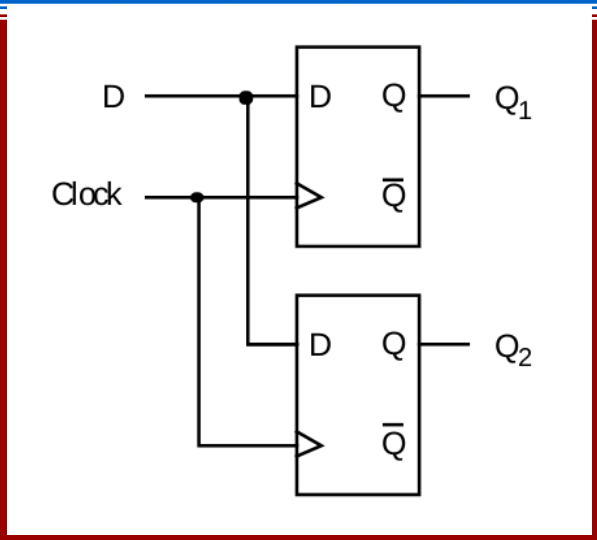
But, VHDL also includes VARIABLE type

- ✓ VARIABLES are always BLOCKING

- ✓ Example on the next slide

VHDL BLOCKING vs NON-BLOCKING

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY block_demo IS
 PORT  ( Clock: IN STD_LOGIC;
          D: IN STD_LOGIC;
          Q1, Q2: OUT STD_LOGIC);
END block_demo;

ARCHITECTURE RTL OF block_demo IS
BEGIN
PROCESS (Clock)
variable t1, t2: STD_LOGIC; --variables only exist inside processes
BEGIN
  IF (Clock'EVENT AND Clock ='1') THEN
          t1 := Q1; --BLOCKING assignment via variables
          t2 := t1;
 END IF;
Q1 <= t1; --assign variables into signals (ports)
Q2 <= t2;
END PROCESS;
END RTL
```



More examples

VHDL 4-bit register (asynchronous clear)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 4_bit_reg IS
PORT  ( clr: IN STD_LOGIC;  -- asynchronous reset
          clk: IN STD_LOGIC;  -- clock
          din: IN STD_LOGIC_VECTOR(3 DOWNTO 0);--4-bit input
          dout: OUT STD_LOGIC_VECTOR(3 DOWNTO 0) --4-bit output
  );
END 4_bit_reg;

ARCHITECTURE RTL OF 4_bit_reg IS
BEGIN
PROCESS (clr, clk)  BEGIN
  IF (clr='0') THEN dout <= 0;
  ELSIF (clk'EVENT AND clk='1') THEN dout<=din;
  END IF;
END PROCESS;
END RTL
```

VHDL 4-bit register (synchronous clear)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 4_bit_reg IS
 PORT  ( clr: IN STD_LOGIC;  -- asynchronous reset
          clk: IN STD_LOGIC;  -- Clock signal
          din: IN STD_LOGIC_VECTOR(3 DOWNTO 0));--1-bit input
          dout: OUT STD_LOGIC_VECTOR(3 DOWNTO 0) --1-bit output
  );
END 4_bit_reg;

ARCHITECTURE RTL OF 4_bit_reg IS
BEGIN
PROCESS (clr, clk)  BEGIN
  IF (clk'EVENT AND clk='1')
  then IF (clr='0') THEN dout <= 0;
        ELSE dout<=din;
        END IF;
 END IF;
END PROCESS;
END RTL
```

VHDL 32-bit register (asynchronous clear)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 32_bit_reg IS
 PORT  ( clr: IN STD_LOGIC;  -- asynchronous reset
          clk: IN STD_LOGIC;  -- Clock signal
          din: IN STD_LOGIC_VECTOR(31 DOWNTO 0);--32-bit input
          dout: OUT STD_LOGIC_VECTOR(31 DOWNTO 0) --32-bit output
  );
END 32_bit_reg;

ARCHITECTURE RTL OF 32_bit_reg IS
BEGIN
PROCESS (clr, clk)  BEGIN
  IF (clr='0') THEN dout <= 0;
  ELSIF (clk'EVENT AND clk='1') THEN dout<=din;
  END IF;
END PROCESS;
END RTL
```

VHDL 32-bit register (synchronous clear)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY 32_bit_reg IS
 PORT  ( clr: IN STD_LOGIC;  -- asynchronous reset
          clk: IN STD_LOGIC;  -- Clock signal
          din: IN STD_LOGIC_VECTOR(31 DOWNTO 0));--1-bit input
          dout: OUT STD_LOGIC_VECTOR(31 DOWNTO 0) --1-bit output
  );
END 32_bit_reg;

ARCHITECTURE RTL OF 32_bit_reg IS
BEGIN
PROCESS (clr, clk)  BEGIN
  IF (clk'EVENT AND clk='1')
  then IF (clr='0') THEN dout <= 0;
        ELSE dout<=din;
        END IF;
 END IF;
END PROCESS;
END RTL
```

Verilog 4-bit register (asynchronous clear)

```
module 4_bit_reg (
        input wire clr,  // asynchronous reset
        input wire clk, // clock
        input wire [3:0] din, //4-bit input
        output reg [3:0] dout //4-bit output
  );

always @(posedge clr or posedge clk) begin
  if (clr=1) dout <= 0;
  else if (clk=1) dout <= din;
end

endmodule
```

Verilog 4-bit register (synchronous clear)

```
module 4_bit_reg (
        input wire clr,  // asynchronous reset
        input wire clk, // clock
        input wire [3:0] din, //4-bit input
        output reg [3:0] dout //4-bit output
  );

always @(posedge clk) begin
  if (clr=1) dout <= 0;
  else if (clk=1) dout <= din;
end

endmodule
```

Verilog 32-bit register (asynch clear)

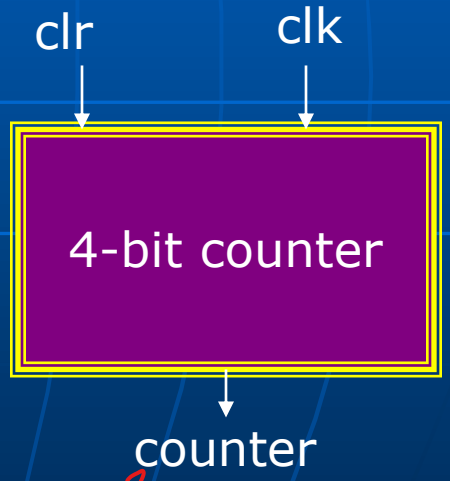
```
module 32_bit_reg (
        input wire clr,  // asynchronous reset
        input wire clk, // clock
        input wire [31:0] din, //32-bit input
        output reg [31:0] dout //32-bit output
  );

always @(posedge clr or posedge clk) begin
  if (clr=1) dout <= 0;
  else if (clk=1) dout <= din;
end

endmodule
```

Verilog 32-bit register (synchronous clear)

```
module 32_bit_reg (
        input wire clr,  // asynchronous reset
        input wire clk, // clock
        input wire [31:0] din, //32-bit input
        output reg [31:0] dout //32-bit output
  );

always @(posedge clk) begin
  if (clr=1) dout <= 0;
  else if (clk=1) dout <= din;
end

endmodule
```

VHDL 4-bit up counter (a new example !!)

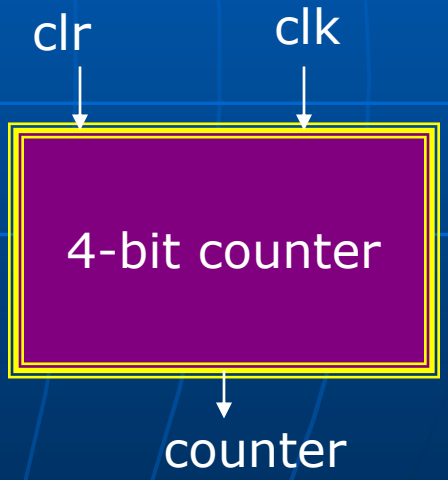
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
entity syn_count4 is
port (clk: in std_logic; reset:  in
std_logic;
counter: out std_logic_vector(3 downto 0));
end syn_count4;

architecture do_it of synch_count4 is
Signal i_cnt: unsigned (3 downto 0);
Begin
PROCESS(reset, clk)
BEGIN
   IF(reset='0') THEN  i_cnt<="0011";
   ELSIF(clk'EVENT AND clk='1') THEN
     IF(i_cnt="1101") THEN i_cnt<="0011";
     ELSE i_cnt<=i_cnt+'1';
     END IF;
   END IF;
END PROCESS;
counter <= i_cnt;
```



VHDL 4-bit down counter (new !)

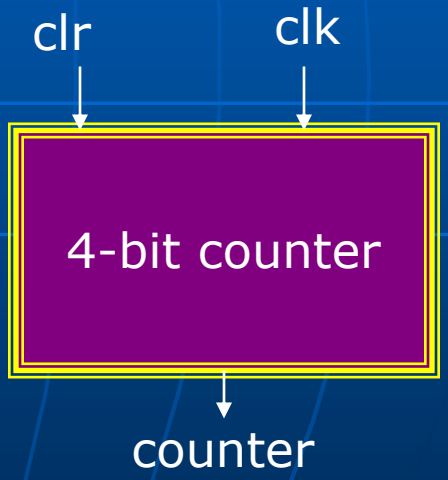
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
 entity syn_count4 is
 port (clk: in std_logic; reset:  in std_logic;
        counter: out std_logic_vector(3 downto 0));
end syn_count4;
architecture do_it of synch_count4 is
Signal i_cnt: unsigned (3 downto 0);
Begin
PROCESS(reset, clk)
BEGIN
  IF(reset='0') THEN  i_cnt<=”1011”;
  ELSIF (clk'EVENT AND clk='1') THEN
    IF (i_cnt=”0000") THEN i_cnt<=“1011";
    ELSE i_cnt<=i_cnt-'1';
    END IF;
  END IF;
END PROCESS;
counter <= i_cnt;
End do_it;
```

use Numeric_STD
IEEE. .all



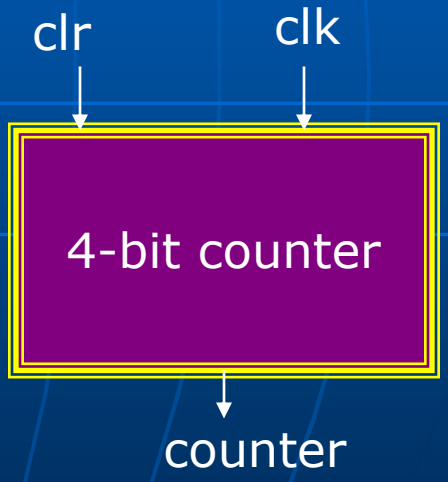
Verilog 4-bit up counter (a new example !!)

```
module syn_count4 (
        input wire clk, reset,
        output wire [3:0] counter
);

reg unsigned [3:0] i_cnt;

always @(posedge clk or posedge reset) begin
  if(reset==1) i_cnt <= 4'b0011;
  else begin
    if(i_cnt==4'b1101) i_cnt <= 4'b0011;
    else i_cnt <= i_cnt + 4'b1;
  end
end

assign counter = i_cnt;

endmodule
```



Verilog 4-bit down counter (new !)

```
module syn_count4 (
        input wire clk, reset,
        output wire [3:0] counter
);

reg unsigned [3:0] i_cnt;

always @(posedge clk or posedge reset) begin
  if(reset==1) i_cnt <= 4'b1011;
  else begin
    if (i_cnt==4'b0) i_cnt <= 4'b1011;
    else i_cnt <= i_cnt – 4'b1;
  end
end

assign counter = i_cnt;

endmodule
```



```
always @ (posedge clk) begin
    if (enable)
      qa <= qa + 1;
  end

always @ (posedge clk) begin
    if (!ld)
      qb <= d;
    else
      qb <= qb + 1;
  end

always @ (posedge clk) begin
    if (!clear)
      qc <= 0;
    else
      qc <= qc + 1;
  end
```

1. Enable counter
2. Up/down counter
3. Enable up/down counter
4. Synchronous load counter
5. Asynchronous load counter
6. Synchronous clear counter
7. Asynchronous clear counter
8. Synchronous load enable counter
9. Asynchronous load enable counter
10. Synchronous clear enable counter
11. Asynchronous clear enable counter
12. Synchronous load clear counter
13. Asynchronous load clear counter
14. Synchronous load up/down counter
15. Asynchronous load up/down counter
16. Synchronous load enable up/down counter
17. Asynchronous load enable up/down counter
18. Synchronous clear load enable counter
19. Asynchronous clear load enable counter
20. Synchronous clear up/down counter
21. Asynchronous clear up/down counter
22. Synchronous clear enable up/down counter
23. Asynchronous clear enable up/down counter

```
always @ (posedge clk) begin
    if (up_down) begin
      direction = 1;
    end
    else begin
      direction = -1;
    end
    qd <= qd + direction;
  end

always @ (posedge clk) begin
    if (!ld)
      qe <= d;
    else if (enable)
      qe <= qe + 1;
  end

always @ (posedge clk) begin
    if (up_down)
      direction = 1;
    else
      direction = -1;
    if (enable)
      qf <= qf + direction;
  end
```

1. Enable counter
2. Up/down counter
3. Enable up/down counter
4. Synchronous load counter
5. Asynchronous load counter
6. Synchronous clear counter
7. Asynchronous clear counter
8. Synchronous load enable counter
9. Asynchronous load enable counter
10. Synchronous clear enable counter
11. Asynchronous clear enable counter
12. Synchronous load clear counter
13. Asynchronous load clear counter
14. Synchronous load up/down counter
15. Asynchronous load up/down counter
16. Synchronous load enable up/down counter
17. Asynchronous load enable up/down counter
18. Synchronous clear load enable counter
19. Asynchronous clear load enable counter
20. Synchronous clear up/down counter
21. Asynchronous clear up/down counter
22. Synchronous clear enable up/down counter
23. Asynchronous clear enable up/down counter

```
always @ (posedge clk) begin
    if (!clear)
      qg <= 0;
    else if (enable)
      qg <= qg + 1;
  end

always @ (posedge clk) begin
    if (up_down)
      direction = 1;
    else
      direction = -1;
    if (!clear)
      ql <= 0;
    else if (enable)
      ql <= ql + direction;
  end

always @ (posedge clk) begin
    if (up_down)
      direction = 1;
    else
      direction = -1;
    if (!ld)
      qi <= d;
    else
      qi <= qi + direction;
  end
```

1. Enable counter
2. Up/down counter
3. Enable up/down counter
4. Synchronous load counter
5. Asynchronous load counter
6. Synchronous clear counter
7. Asynchronous clear counter
8. Synchronous load enable counter
9. Asynchronous load enable counter
10. Synchronous clear enable counter
11. Asynchronous clear enable counter
12. Synchronous load clear counter
13. Asynchronous load clear counter
14. Synchronous load up/down counter
15. Asynchronous load up/down counter
16. Synchronous load enable up/down counter
17. Asynchronous load enable up/down counter
18. Synchronous clear load enable counter
19. Asynchronous clear load enable counter
20. Synchronous clear up/down counter
21. Asynchronous clear up/down counter
22. Synchronous clear enable up/down counter
23. Asynchronous clear enable up/down counter

```
always @ (posedge clk) begin
    if (!clear)
      qh <= 0;
    else if (!ld)
      qh <= d;
    else
      qh <= qh + 1;
  end


always @ (posedge clk) begin
    if (up_down)
      direction = 1;
    else
      direction = -1;
    if (!clear)
      qm <= 0;
    else if (enable)
      qm <= qm + direction;
  end
```

1. Enable counter
2. Up/down counter
3. Enable up/down counter
4. Synchronous load counter
5. Asynchronous load counter
6. Synchronous clear counter
7. Asynchronous clear counter
8. Synchronous load enable counter
9. Asynchronous load enable counter
10. Synchronous clear enable counter
11. Asynchronous clear enable counter
12. Synchronous load clear counter
13. Asynchronous load clear counter
14. Synchronous load up/down counter
15. Asynchronous load up/down counter
16. Synchronous load enable up/down counter
17. Asynchronous load enable up/down counter
18. Synchronous clear load enable counter
19. Asynchronous clear load enable counter
20. Synchronous clear up/down counter
21. Asynchronous clear up/down counter
22. Synchronous clear enable up/down counter
23. Asynchronous clear enable up/down counter

```
always @ (posedge clk) begin
    if (up_down)
      direction = 1;
    else
      direction = -1;
    if (!ld)
      qj <= d;
    else if (enable)
      qj <= qj + direction;
  end
```

```
always @ (posedge clk) begin
    if (!clear)
      qk <= 0;
    else if (!ld)
      qk <= d;
    else if (enable)
      qk <= qk + 1;
  end
```

1. Enable counter
2. Up/down counter
3. Enable up/down counter
4. Synchronous load counter
5. Asynchronous load counter
6. Synchronous clear counter
7. Asynchronous clear counter
8. Synchronous load enable counter
9. Asynchronous load enable counter
10. Synchronous clear enable counter
11. Asynchronous clear enable counter
12. Synchronous load clear counter
13. Asynchronous load clear counter
14. Synchronous load up/down counter
15. Asynchronous load up/down counter
16. Synchronous load enable up/down counter
17. Asynchronous load enable up/down counter
18. Synchronous clear load enable counter
19. Asynchronous clear load enable counter
20. Synchronous clear up/down counter
21. Asynchronous clear up/down counter
22. Synchronous clear enable up/down counter
23. Asynchronous clear enable up/down counter

VHDL 16x4 ROM (new !)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; --use CONV_INTEGER
ENTITY 16x4ROM IS
 PORT  (clk: IN STD_LOGIC;  -- Clock signal
        addr: IN STD_LOGIC_VECTOR(3DOWNTO 0);--4-bit address
        data: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)); --4-bit output
END 16x4ROM;

ARCHITECTURE RTL OF 16x4ROM IS

TYPE MARRAY IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(3DOWNTO 0);
CONSTANT rom: MARRAY := (X"4",X"C",X"8", X"B", X"E", X"F",
X"3",X"1",X"D", X"7", X"9", X"2", X"0", X"E", X"3", X"F");

BEGIN
PROCESS(clk)
BEGIN
IF(clk'EVENT AND clk='1')  THEN data <= rom(CONV_INTEGER(addr));
  END IF;
END PROCESS;
END RTL;
```

Typedef

Read Only ("Constant")

Route

VHDL 16x4 ROM (new !)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; --use CONV_INTEGER
ENTITY 16x4ROM IS
 PORT  (clk: IN STD_LOGIC;  -- Clock signal
addr: IN STD_LOGIC_VECTOR(3DOWNTO 0);--4-bit address
data: OUT STD_LOGIC_VECTOR (3 DOWNTO 0)); --4-bit data output);
END 16x4ROM;

ARCHITECTURE RTL OF 16x4ROM IS
TYPE MARRAY IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(3DOWNTO 0);
CONSTANT rom: MARRAY := (“0000”,”1100",”1010", “1110",
“1111",”0001”, “0011",”0000”,”1111", “1110", “0011",”1111", “0101",
“1100",”1100",”1111”);

BEGIN
PROCESS(clk)
BEGIN
IF(clk'EVENT AND clk='1')  THEN data <= rom(CONV_INTEGER(addr));
  END IF;
END PROCESS;
END RTL;
```

Verilog 16x4 ROM (new !)

```
module ROM16x4 (
        input wire clk,
        input wire [3:0] addr,
        output reg [3:0] data
);

reg [3:0] rom [0:15];

initial begin
        rom[0] = 4'b0000;
        rom[1] = 4'b0110;
        //etc…
        rom[15] = 4'b1010;
end

always @(posedge clk) begin
  data <= rom[addr];
end

endmodule
```

*It is cumbersome to initialize large constant arrays in Verilog, so we often use the readmemh (hex strings) and readmemb (binary) "magic functions".

End of modeling a sequential circuit in VHDL/Verilog

A simple function

```
A = A + S[0];
B = B + S[1];
for i = 1 to 12 do
    A = ((A xor B) <<< B) + S[2×i];
    B = ((B xor A) <<< A) + S[2×i + 1];
```

- A and B are 32-bit halves of the input
- S[0], S[1], S[2]…S[25] are 26 values
  - 32-bits each → How many bits altogether?

and its inverse…..

```
for i = 12 downto 1 do
  B = ((B - S[2×i +1]) >>> A) xor A;
  A = ((A - S[2×i]) >>> B) xor B;
B = B - S[1];
A = A - S[0];
```

- Inverse of function on previous slide (why?)
  - F(G(x)) = x ⇒ F and G Inverses of each other
- A and B are 32-bit halves of input
- S[25], S[24], S[23]… are used in that order

Simple function	And its Inverse…
A0 = A + S[0]	A = A0 - S[0]
B0 = B + S[1]	B = B0 - S[1]
A1 = ((A0 xor B0) <<< B0) + S[2]	A0 = ((A1 – S[2])>>> B0) xor B0
B1 = ((B0 xor A1) <<< A1) + S[3]	B0 = ((B1 - S[3]) >>> A1) xor A1
A2 = ((A1 xor B1) <<< B1) + S[4]	A1 = ((A2 - S[4]) >>> B1) xor B1
B2 = ((B1 xor A2) <<< A2) + S[5]	B1 = ((B2 – S[5]) >>> A2) xor A2
A12=((A11 xor B11)<<<B11)+S[24]	A11=((A12- S[24])>>>B11)xorB11
B12=((B11 xor A12)<<<A12)+S[25]	B11=((B12–S[25])>>>A12)xor A12

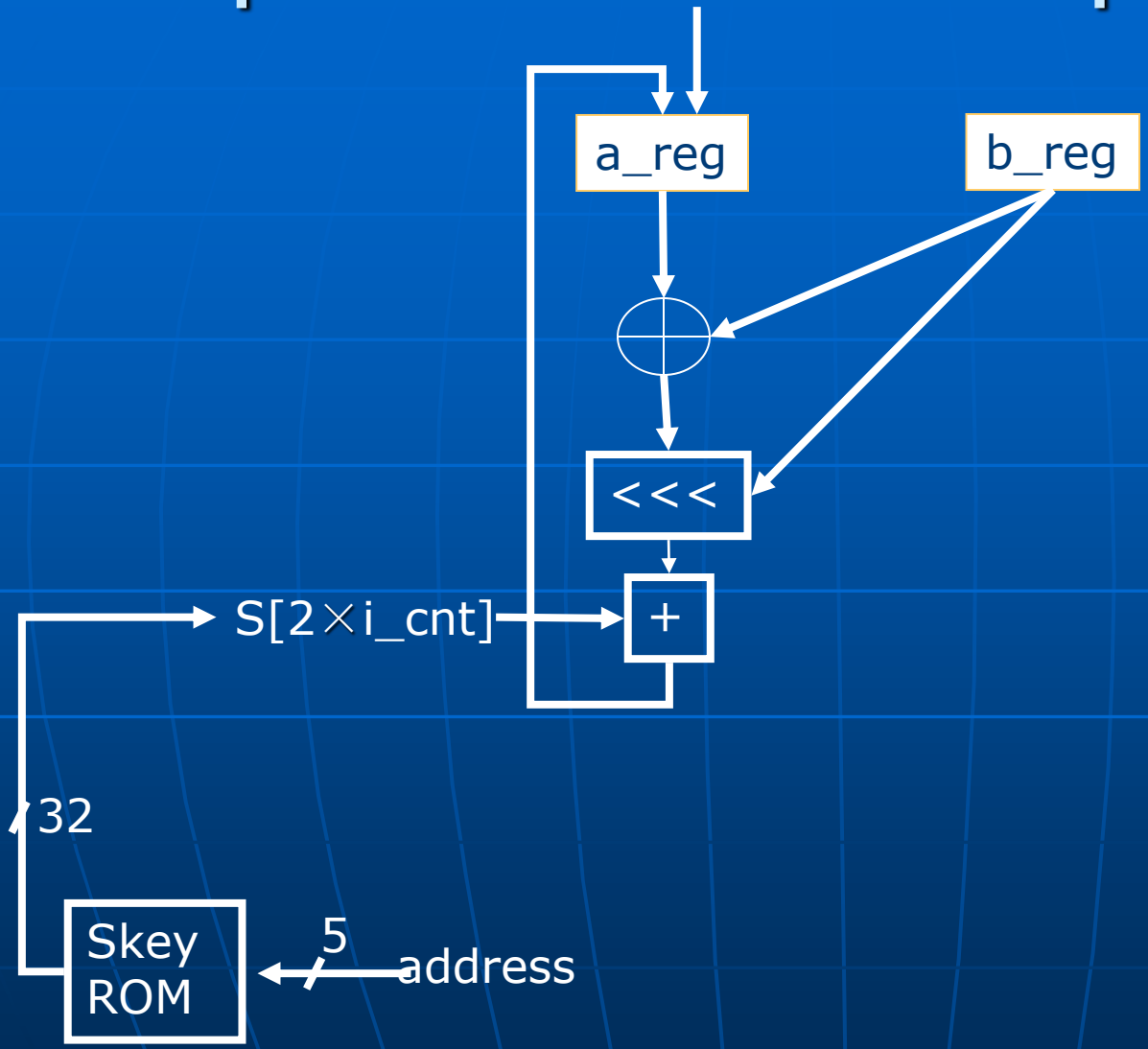
Simple function: entity definition

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; --use CONV_INTEGER

ENTITY rc5_enc IS
 PORT  (
  clr: IN STD_LOGIC;  -- asynchronous reset
  clk: IN STD_LOGIC;  -- Clock signal
  din: IN STD_LOGIC_VECTOR(63 DOWNTO 0);--64-bit input
  dout: OUT STD_LOGIC_VECTOR(63 DOWNTO 0) --64-bit outpu
  );
END rc5_enc;
```

Simple function: Verilog module I/O

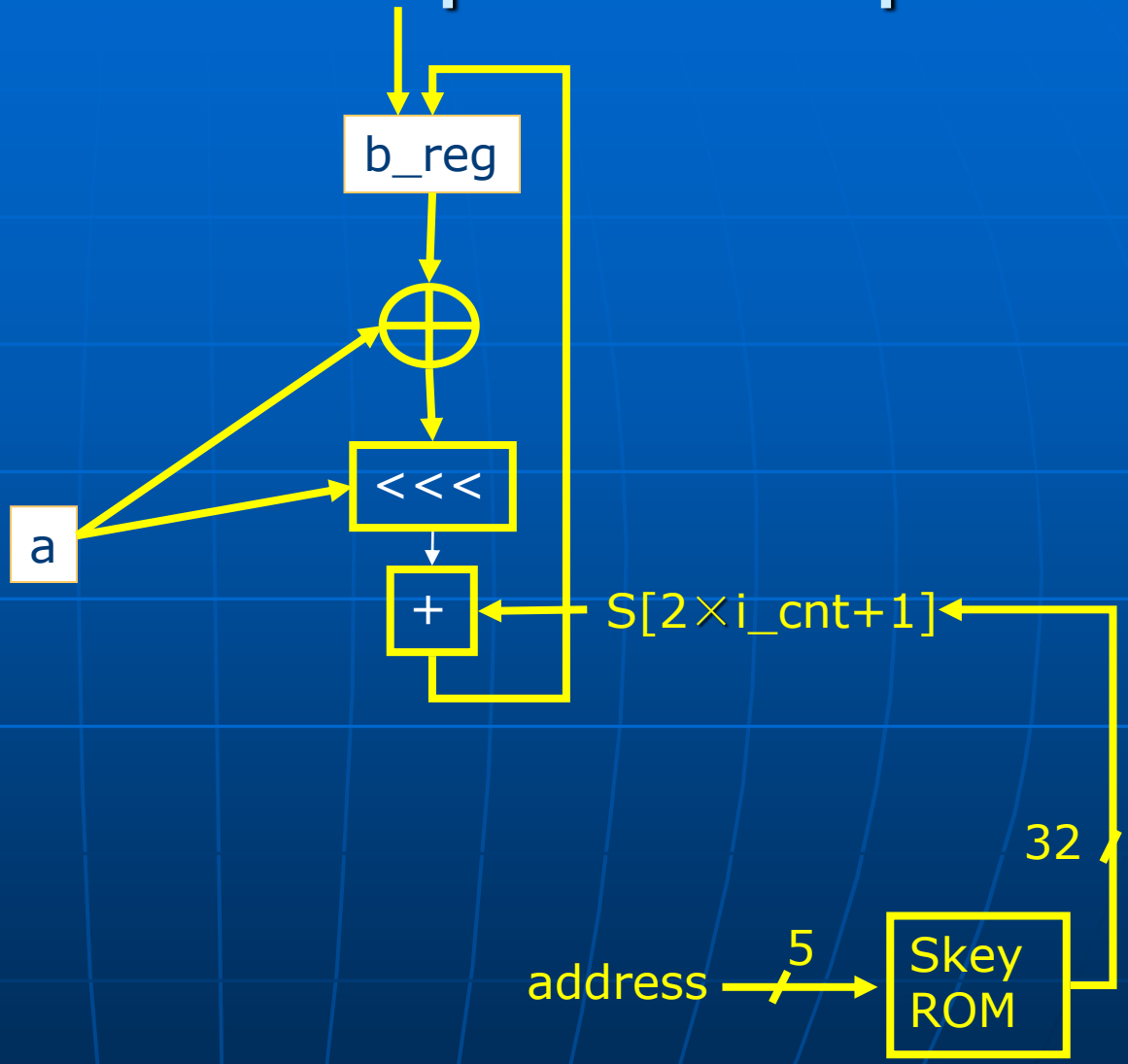
```
module rc5_enc (
        input wire clr,  // asynchronous reset
        input wire clk, //Clock signal
        input wire [63:0] din, //64-bit input
        output wire [63:0] dout //64-bit output
 );
```

Simple function data path: step a



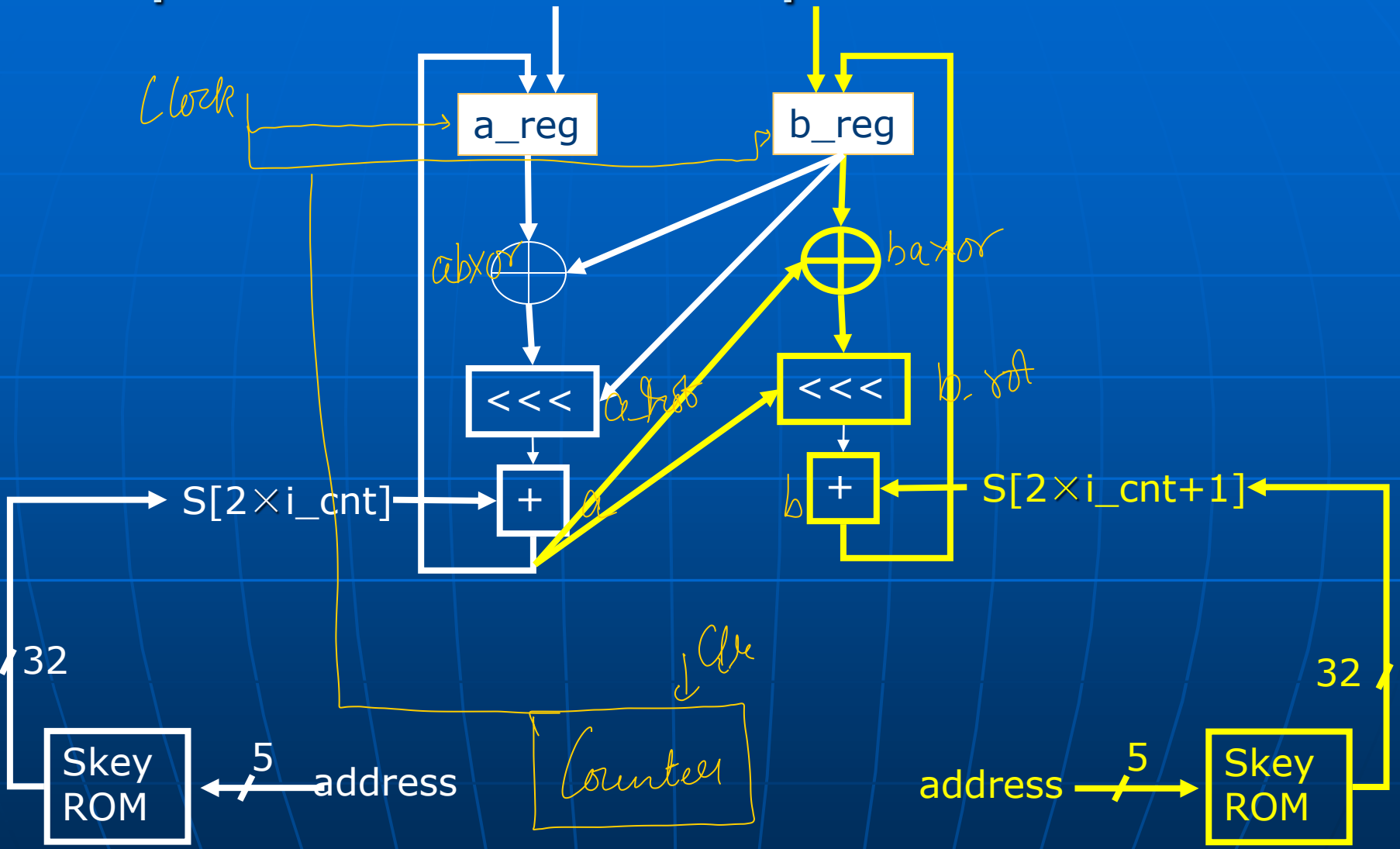
How are the addresses generated?

Simple function data path: step b



How are the addresses generated?

Simple function data path



How are the addresses generated?

Simple function (for loop part only): architecture description (internal signals)

```
ARCHITECTURE rtl OF rc5_enc IS
  --round counter
  SIGNAL i_cnt: STD_LOGIC_VECTOR(3 DOWNTO 0);
  SIGNAL ab_xor: STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL a_rot: STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL a: STD_LOGIC_VECTOR(31 DOWNTO 0);
  --register to store value A
  SIGNAL a_reg: STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL ba_xor: STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL b_rot: STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL b: STD_LOGIC_VECTOR(31 DOWNTO 0);
  --register to store value B
  SIGNAL b_reg: STD_LOGIC_VECTOR(31 DOWNTO 0);
```

Simple function (for loop part only): architecture description (internal signals)

```
//round counter
  reg [3:0] i_cnt;
  wire [31:0] ab_xor;
  wire [31:0] a_rot;
  wire [31:0] a;
//register to store value A
  reg [31:0] a_reg;
  wire [31:0] ba_xor;
  wire [31:0] b_rot;
  wire [31:0] b;
  //register to store value B
  reg [31:0] b_reg;
```

Simple function: hardcode 24 values

```
TYPE rom IS ARRAY (2 TO 25) OF STD_LOGIC_VECTOR(31
   DOWNTO 0);
```

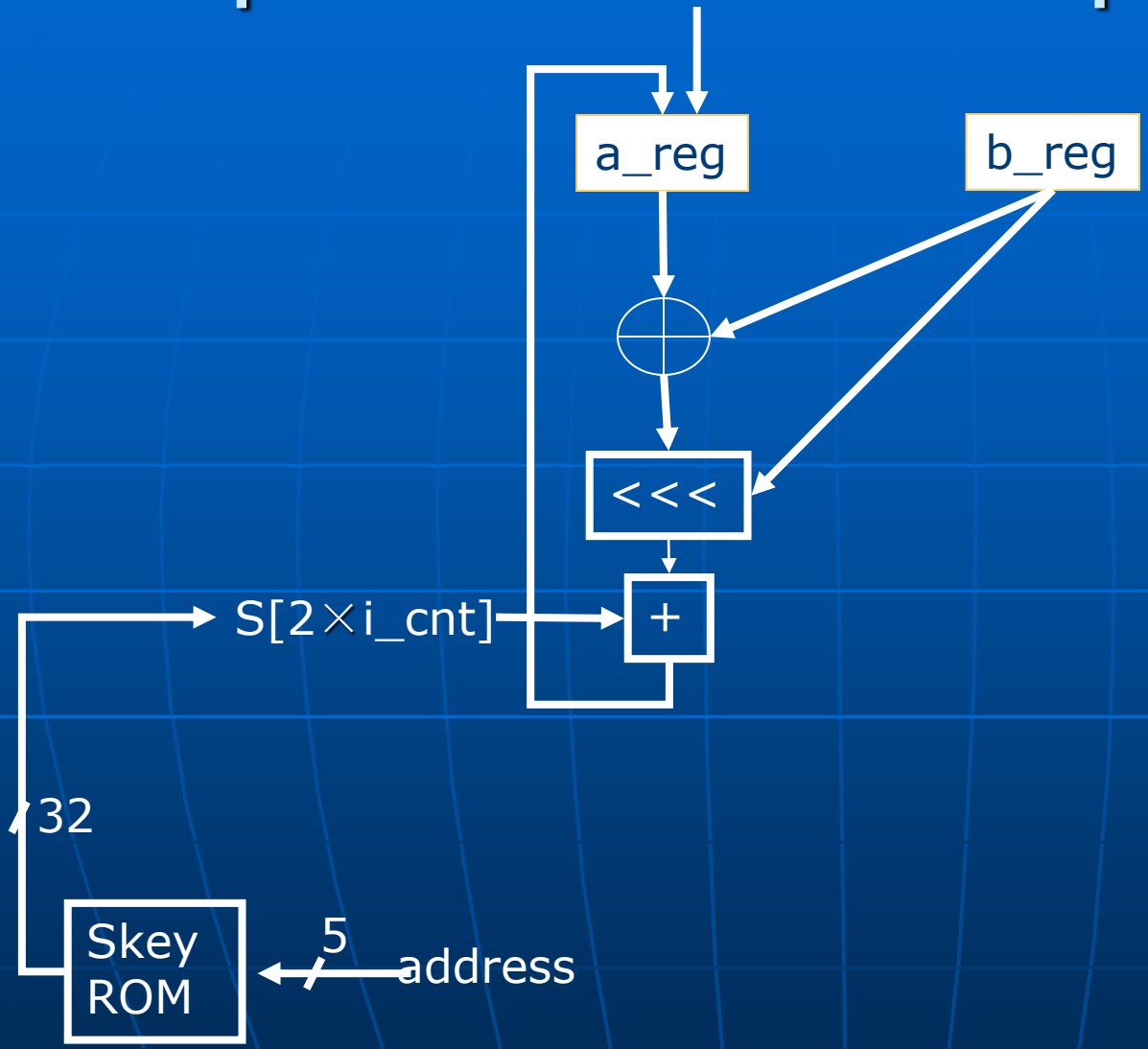
```
CONSTANT skey:
   rom:=rom`(X"46F8E8C5",X"460C6085",X"70F83B8A",
   X"284B8303", X"513E1454", X"F621ED22",
   X"3125065D",X"11A83A5D",X"D427686B", X"713AD82D",
   X"4B792F99", X"2799A4DD", X"A7901C49", X"DEDE871A",
   X"36C03196", X"A7EFC249", X"61A78BB8", X"3B0A1D2B",
   X"4DBFCA76", X"AE162167", X"30D76B0A", X"43192304",
   X"F6CC1431", X"65046380");
```

Simple function: hardcode 24 values

```
reg [31:0] rom [2:25];
```

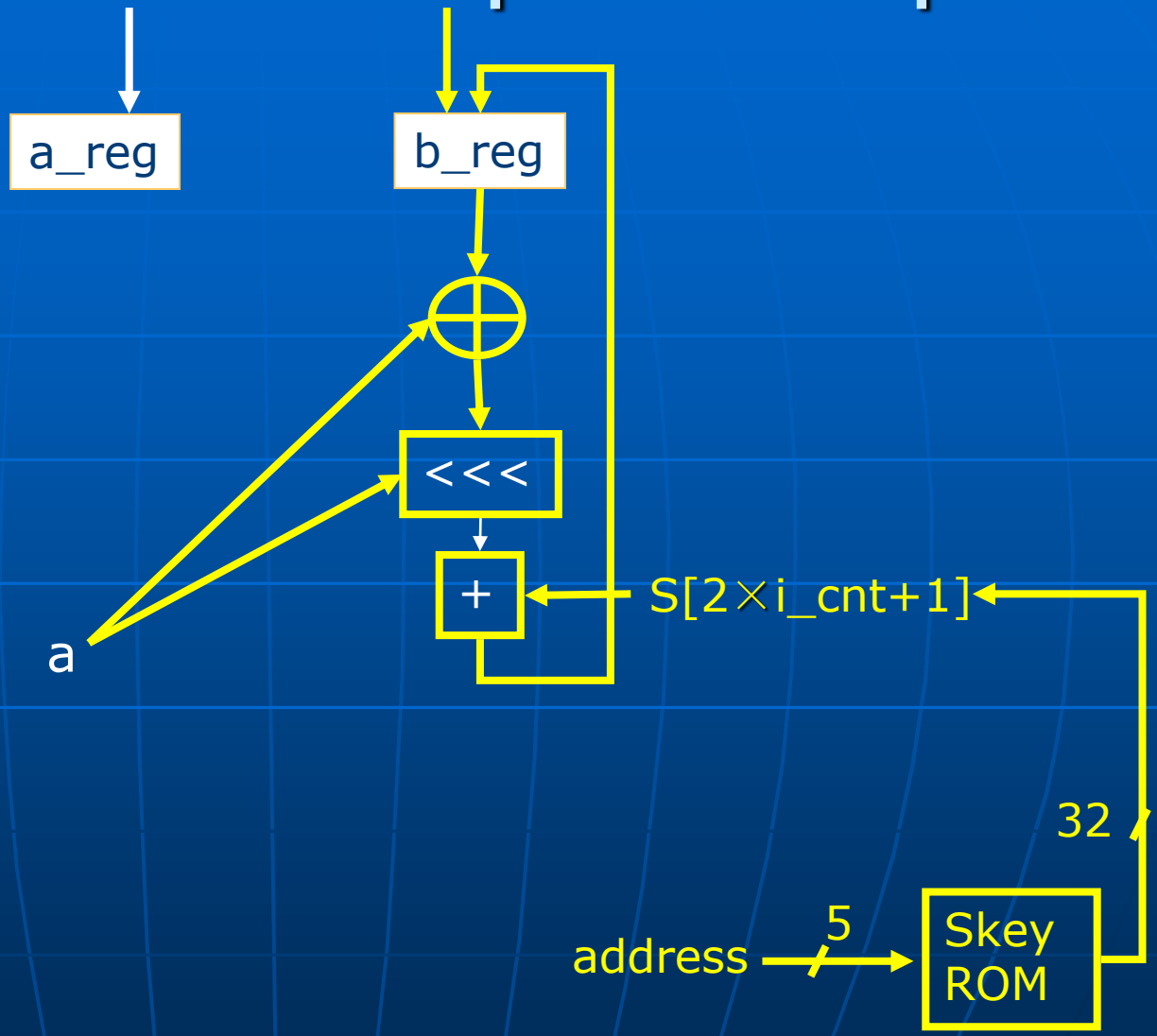
```
initial begin
   rom[2] = 32'h46F8E8C5;
   rom[3] = 32'h460C6085;
   rom[4] = 32'h70F83B8A;
   rom[5] = 32'h284B8303;
   rom[6] = 32'h513E1454;
   rom[7] = 32'hF621ED22;
   rom[8] = 32'h3125065D;
   rom[9] = 32'h11A83A5D;
   rom[10] = 32'hD427686B;
   rom[11] = 32'h713AD82D;
   rom[12] = 32'h4B792F99;
   rom[13] = 32'h2799A4DD;
   rom[14] = 32'hA7901C49;
   rom[15] = 32'hDEDE871A;
   rom[16] = 32'h36C03196;
   rom[17] = 32'hA7EFC249;
   rom[18] = 32'h61A78BB8;
   rom[19] = 32'h3B0A1D2B;
   rom[20] = 32'h4DBFCA76;
   rom[21] = 32'hAE162167;
   rom[22] = 32'h30D76B0A;
   rom[23] = 32'h43192304;
   rom[24] = 32'hF6CC1431;
   rom[25] = 32'h65046380;
end
```

Simple function data path: step a



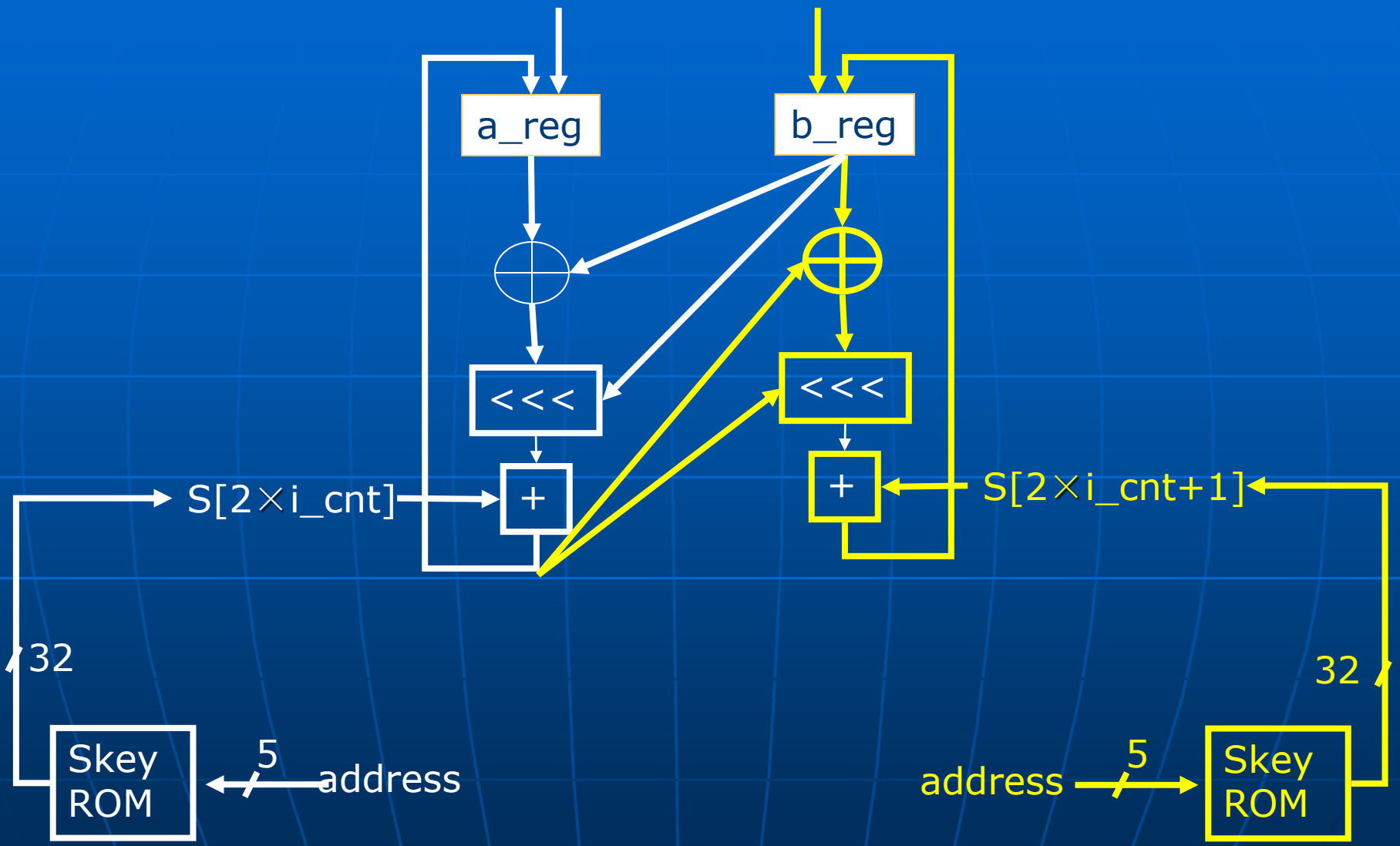
How are the addresses generated?

Simple function data path: step b



How are the addresses generated?

Simple function data path



How are the addresses generated?

Simple VHDL function (architecture body): step a: A=((A XOR B)<<<B) + S[2×i]

```
BEGIN
   -- A=((A XOR B)<<<B) + S[2×i];
  ab_xor <= a_reg XOR b_reg;
  WITH b_reg(4 DOWNTO 0) SELECT
     a_rot<=ab_xor(30 DOWNTO 0) & ab_xor(31) WHEN
"00001",
                    ……
                 ab_xor WHEN OTHERS;
  a<=a_rot + skey(CONV_INTEGER(i_cnt & '0')); --S[2×i]
```

We discussed this in previous lecture and was part of your homework!!!

Simple VHDL function (architecture body): step b: B=((B XOR A)<<<A) + S[2×i+1])

```
ba_xor <= b_reg XOR a;
  WITH a(4 DOWNTO 0) SELECT
    b_rot<=ba_xor(30 DOWNTO 0) & ba_xor(31) WHEN
"00001",
                  ......
              ba_xor WHEN OTHERS;
 b<=b_rot+skey(CONV_INTEGER(i_cnt & '1'));--S[2×i+1]
```

Modify the model in exercise 2 homework 1 !!!

Simple Verilog function (architecture body): step a: A=((A XOR B)<<<B) + S[2×i]

```
// A=((A XOR B)<<<B) + S[2×i];
assign ab_xor = a_reg ^ b_reg;

assign a_rot = (b_reg[4:0] == 5'b0001) ?
                              {ab_xor[30:0], ab_xor[31]} :
               // etc…… :
                ab_xor;

assign a = a_rot + skey[{i_cnt, 1'b0'}]; //S[2×i]
```

We discussed this in previous lecture and was part of your homework!!!

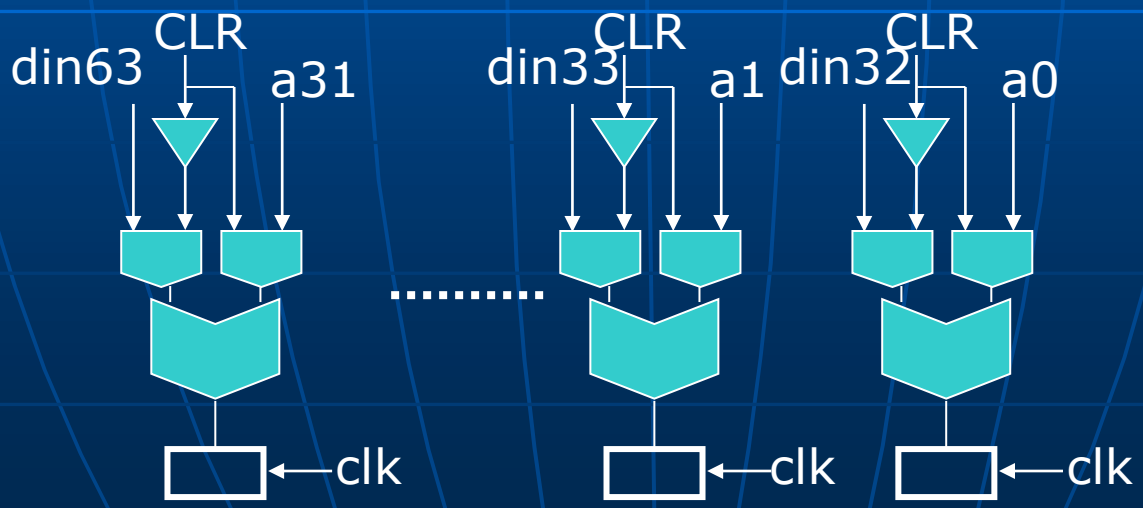
Simple Verilog function (architecture body): step b: B=((B XOR A)<<<A) + S[2×i+1])

```
assign ba_xor = b_reg ^ a;

assign b_rot = (a[4:0] == 5'b00001) ?
                          {ba_xor[30:0], ba_xor[31]} :
            // etc  ..... :
            ba_xor;

assign b = b_rot + skey[{i_cnt, 1'b1}]; //S[2×i+1]
```

Modify the model in exercise 2 homework 1 !!!

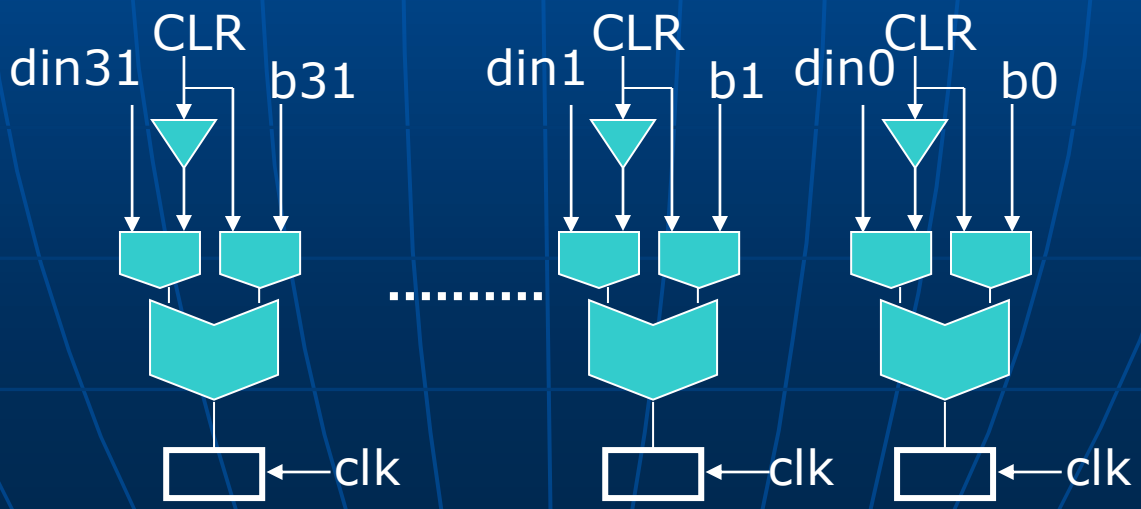
Simple VHDL function: modified 32-bit register

```
-- a_reg
PROCESS(clr, clk)  BEGIN
  IF(clr='0') THEN a_reg <= din(63 DOWNTO 32);
  ELSIF(clk'EVENT AND clk='1') THEN a_reg<=a;
  END IF;
END PROCESS;
```



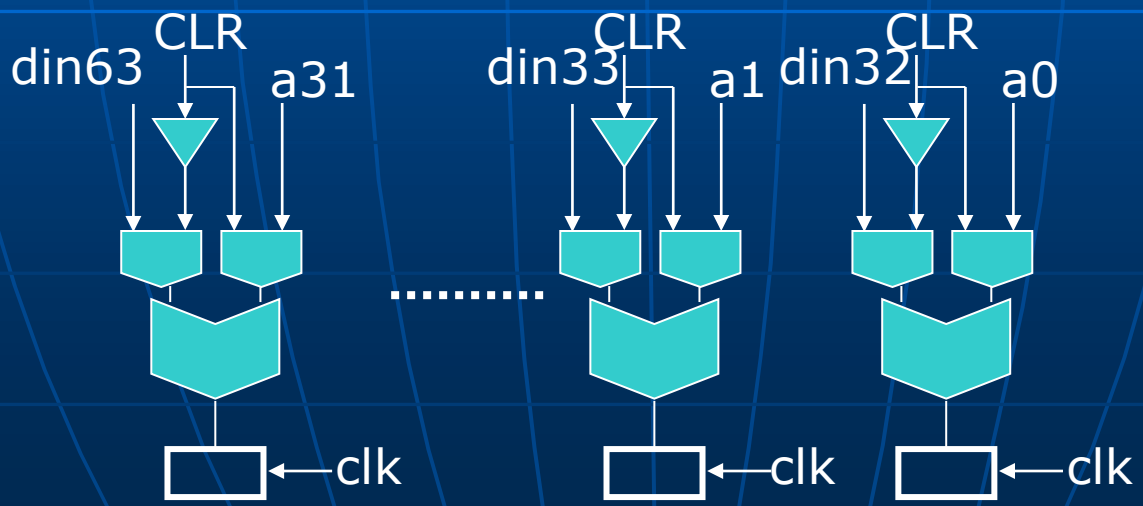
Simple VHDL function: modified 32-bit register

```
-- b_reg
   PROCESS(clr, clk)  BEGIN
      IF(clr='0') THEN b_reg<=din(31 DOWNTO 0);
      ELSIF(clk'EVENT AND clk='1') THEN b_reg<=b;
      END IF;
   END PROCESS;
```



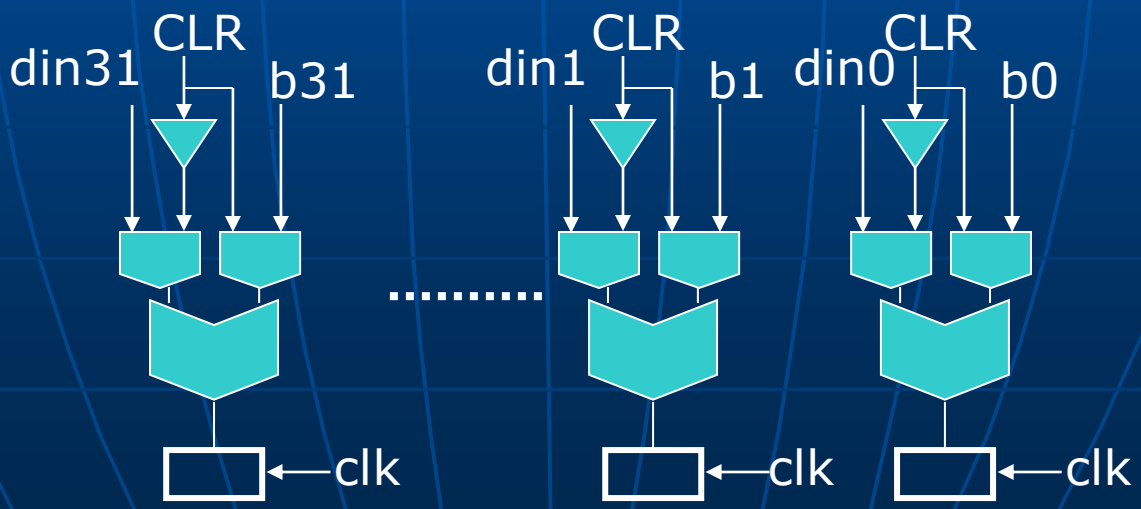
Simple Verilog function: modified 32-bit register

```
// a_reg
always @(negedge clr or posedge clk)  begin
  if(clr==0) a_reg <= din[63:32];
  else a_reg <= a;
end
```



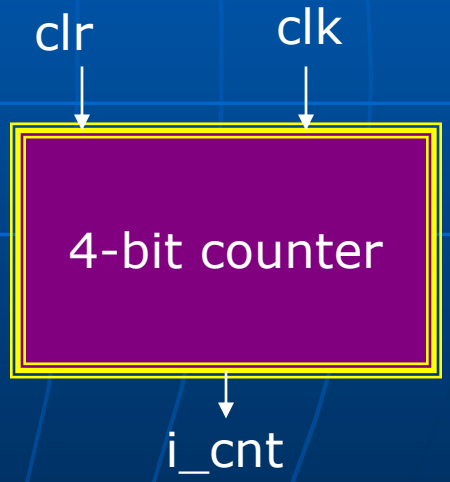
Simple Verilog function: modified 32-bit register

```
// b_reg
always @(negedge clr or posedge clk) begin
  if(clr==0) b_reg<=din[31:0];
  else b_reg <= b;
end
```



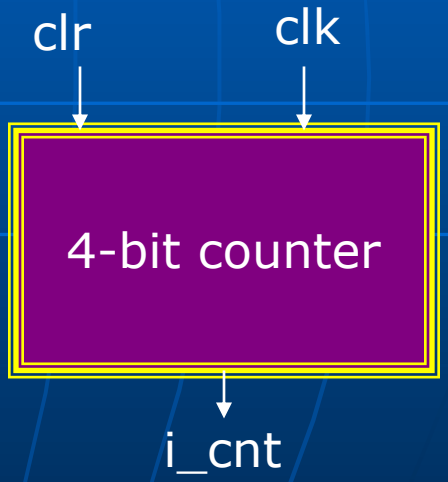
Simple VHDL function: a 4-bit counter

```
PROCESS(clr, clk)
BEGIN
  IF(clr='0') THEN
        i_cnt<="0001";
  ELSIF(clk'EVENT AND clk='1') THEN
      IF(i_cnt="1100") THEN
        i_cnt<="0001";
      ELSE
        i_cnt<=i_cnt+'1';
      END IF;
    END IF;
END PROCESS;
```



Simple Verilog function: a 4-bit counter

```
always @(negedge clr or posedge clk)
begin
  if (clr==0) begin
        i_cnt <= 4'b0001;
  end else begin
       if(i_cnt==4'b1100)
         i_cnt<=4'b0001;
       else
         i_cnt<=i_cnt+4'b1;

   end
end
```



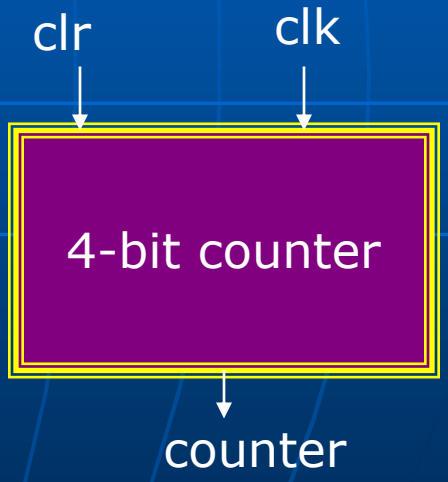
Simple function: assign results to output

```
dout<=a_reg & b_reg;
END rtl;
```

```
assign dout= {a_reg, b_reg};
endmodule
```

4-bit up counter: Quiz

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;
entity syn_count4 is
 port (clk: in std_logic; clr:  in std_logic;
        counter: out std_logic_vector(3 downto 0));
end syn_count4;
architecture do_it of synch_count4 is
Signal i_cnt: std_logic_vector (3 downto 0);
Begin
PROCESS (clr, clk)
BEGIN
  IF(clr='0') THEN  i_cnt<=”0000”;
  ELSIF (clk'EVENT AND clk='1') THEN
     i_cnt<=i_cnt+1;
 END IF;
END PROCESS;
counter <= i_cnt;
End do_it;
```



Modify the counter to implement these two sequences

Sequence 1: 2, 4, 6, 8, 10, 12, 14, 2

Sequence 2: 15, 13, 11, 9, 7, 5, 3, 1, …

HW2: Simple function iterative arch.

- Implement "for loop" part of the simple function
  - In **BOTH** VHDL and Verilog
  - Input: 64-bit plaintext (A, B are 32 bit input words)
  - Output: 64-bit ciphertext (A, B are 32 bit output words)
  - S [2], S[3], …, S[24], S[25] are 32-bit values (these will be provided)

HW3: Simple inverse func. iterative arch.

- Implement "for loop" part of inverse function
  - In **BOTH** VHDL and Verilog
  - Input: 64-bit ciphertext (A, B are 32 bit input words)
  - Output: 64-bit plaintext (A, B are 32 bit output words)
  - S [25], S[24], …, S[3], S[2] are 32-bit round keys (these will be provided)

VHDL Case Statement

- alternative to “with-select” (concurrent statement)
- sequential statement
  - can only be used inside a process

O =(A XOR B) <<<B + SKEY

```
LIBRARY        IEEE;
USE            IEEE.STD_LOGIC_1164.ALL;
ENTITY         leftrotate IS
  PORT (a: in STD_LOGIC_VECTOR(31 DOWNTO 0);
          b: in STD_LOGIC_VECTOR(31 DOWNTO 0);
           skey: out STD_LOGIC_VECTOR(31 DOWNTO 0);
          o: out STD_LOGIC_VECTOR(31 DOWNTO 0));
END leftrotate;

ARCHITECTURE rtl OF leftrotate IS
SIGNAL ab_xor: STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL ab_rot: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
ab_xor <= a XOR b;
```

O =(A XOR B) <<<B + SKEY

```
ROCESS(b, ab_xor)
EGIN
ASE b(4 DOWNTO 0) IS
WHEN “00001”=> ab_rot<= ab_xor(30 DOWNTO 0)&ab_xor(31);
WHEN “00010”=> ab_rot<=ab_xor(29 DOWNTO 0)&ab_xor(31DOWNTO 30);
WHEN "00011”=> ab_rot<= ab_xor(28 DOWNTO 0) & ab_xor(31 DOWNTO 29)
WHEN "00100“=> ab_rot<= ab_xor(27 DOWNTO 0) & ab_xor(31 DOWNTO 28)
WHEN "00101“=> ab_rot<= ab_xor(26 DOWNTO 0) & ab_xor(31 DOWNTO 27)
WHEN "00110“=> ab_rot<= ab_xor(25 DOWNTO 0) & ab_xor(31 DOWNTO 26)
WHEN "00111" => ab_rot<= ab_xor(24 DOWNTO 0) & ab_xor(31 DOWNTO 25
WHEN "01000" => ab_rot<= ab_xor(23 DOWNTO 0) & ab_xor(31 DOWNTO 24
WHEN "01001" => ab_rot<= ab_xor(22 DOWNTO 0) & ab_xor(31 DOWNTO 23
HEN "01010" => ab_rot<= ab_xor(21 DOWNTO 0) & ab_xor(31 DOWNTO 22)
WHEN "01011" => ab_rot<= ab_xor(20 DOWNTO 0) & ab_xor(31 DOWNTO 21
WHEN "01100" => ab_rot<= ab_xor(19 DOWNTO 0) & ab_xor(31 DOWNTO 20
WHEN "01101" => ab_rot<= ab_xor(18 DOWNTO 0) & ab_xor(31 DOWNTO 19
WHEN "01110" => ab_rot<= ab_xor(17 DOWNTO 0) & ab_xor(31 DOWNTO 18
WHEN "01111" => ab_rot<= ab_xor(16 DOWNTO 0) & ab_xor(31 DOWNTO 17
WHEN "10000" => ab_rot<= ab_xor(15 DOWNTO 0) & ab_xor(31 DOWNTO 16
HEN "10001" =>ab_rot<= ab_xor(14 DOWNTO 0) & ab_xor(31 DOWNTO 15);
HEN "10010" =>ab_rot<= ab_xor(13 DOWNTO 0) & ab_xor(31 DOWNTO 14);
HEN "10011" =>ab_rot<= ab_xor(12 DOWNTO 0) & ab_xor(31 DOWNTO 13);
```

O =(A XOR B) <<<B + SKEY

```
WHEN "10100" =>ab_rot<= ab_xor(11 DOWNTO 0) & ab_xor(31 DOWNTO 12)
WHEN "10101" =>ab_rot<= ab_xor(10 DOWNTO 0) & ab_xor(31 DOWNTO 11)
WHEN "10110" =>ab_rot<= ab_xor(9 DOWNTO 0) & ab_xor(31 DOWNTO 10);
WHEN "10111" =>ab_rot<= ab_xor(8 DOWNTO 0) & ab_xor(31 DOWNTO 9);
WHEN "11000" =>ab_rot<= ab_xor(7 DOWNTO 0) & ab_xor(31 DOWNTO 8);
WHEN "11001" =>ab_rot<= ab_xor(6 DOWNTO 0) & ab_xor(31 DOWNTO 7);
WHEN "11010" =>ab_rot<= ab_xor(5 DOWNTO 0) & ab_xor(31 DOWNTO 6);
WHEN "11011" =>ab_rot<= ab_xor(4 DOWNTO 0) & ab_xor(31 DOWNTO 5);
WHEN "11100" =>ab_rot<= ab_xor(3 DOWNTO 0) & ab_xor(31 DOWNTO 4);
WHEN "11101" =>ab_rot<= ab_xor(2 DOWNTO 0) & ab_xor(31 DOWNTO 3);
WHEN "11110" =>ab_rot<= ab_xor(1 DOWNTO 0) & ab_xor(31 DOWNTO 2);
WHEN "11111" =>ab_rot<= ab_xor(0) & ab_xor(31 DOWNTO 1);
WHEN OTHERS =>ab_rot<=ab_xor;
END CASE;
END PROCESS;
O <=ab_rot+skey
END rtl;
```

Verilog implementation left as exercise!

VHDL 16x8 RAM (new !)

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL; --use CONV_INTEGER
ENTITY 16x8RAM IS
 PORT  (clk: IN STD_LOGIC;
readwrite: in STD_LOGIC; --read =0; write =1
addr: IN STD_LOGIC_VECTOR(3DOWNTO 0);--4-bit address
datain: in STD_LOGIC_VECTOR(7 DOWNTO 0) ;--8-bit datain
dataout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0)); --8-bitdataout);
END 16x4RAM;
ARCHITECTURE RTL OF 16x8RAM IS
TYPE ram IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7DOWNTO 0);
signal skey: ram;
BEGIN
PROCESS(clk)
BEGIN
IF(clk'EVENT AND clk='1')  THEN
    if (readwrite ==‘1’) then skey(CONV_INTEGER(addr)) <= datain;
endif;
 END IF;
END PROCESS;
Dataout<= skey(CONV_INTEGER(addr));
END RTL;
```

Verilog 16x8 RAM (new !)

```
module RAM16x8 (
        input wire clk,
        input wire readwrite, //read =0; write =1
        input wire [3:0] addr, //4-bit address
        input wire [7:0] datain, //8-bit datain
        input wire [7:0] dataout //8-bitdataout
);

reg [7:0] skey [0:15];

always @(posedge clk) begin
  if(readwrite == 1)
    skey[addr] <= datain;
end

assign dataout = skey[addr];

endmodule
```

Hardware Security

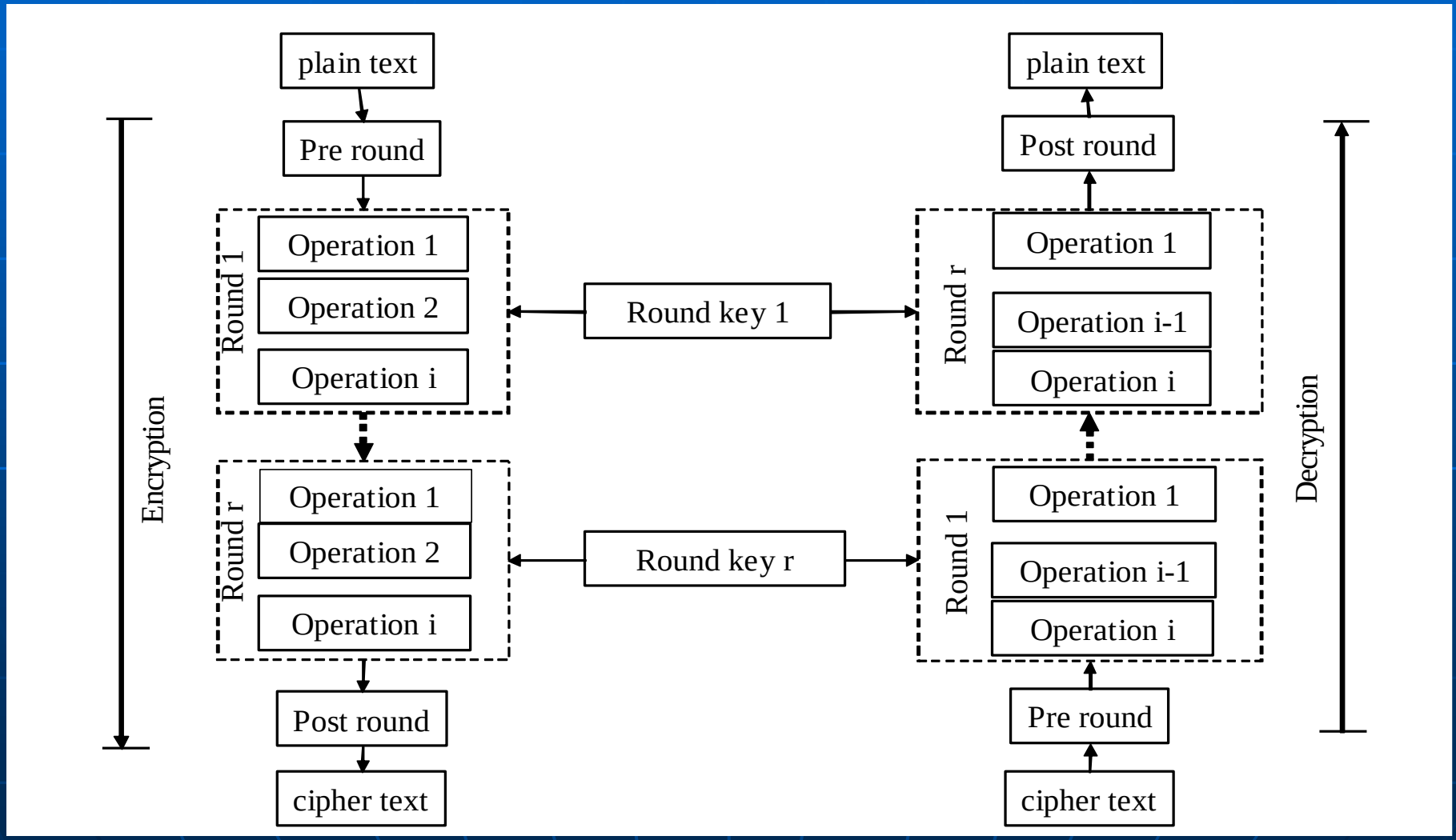
- Security and Privacy imply support for
  - Data confidentiality
  - Data integrity
  - Authentication
  - Non-repudiation
  - Symmetric Key Crypto is the building block

Data confidentiality

- Symmetric block cipher
  - Encrypt (plaintext block, key)= ciphertext block
  - Decrypt (ciphertext block, key)= plaintext block
  - Encryption key = Decryption key
  - Data Encryption Standard (DES)
    - 64-bit plaintext block, 56-bit secret key
  - Advanced Encryption Standard (AES)
    - 128-bit plaintext block, 128-bit secret key
  - RC5
    - 64-bit plaintext block, 128-bit key

RC5 Symmetric block cipher

Symmetric Block Cipher



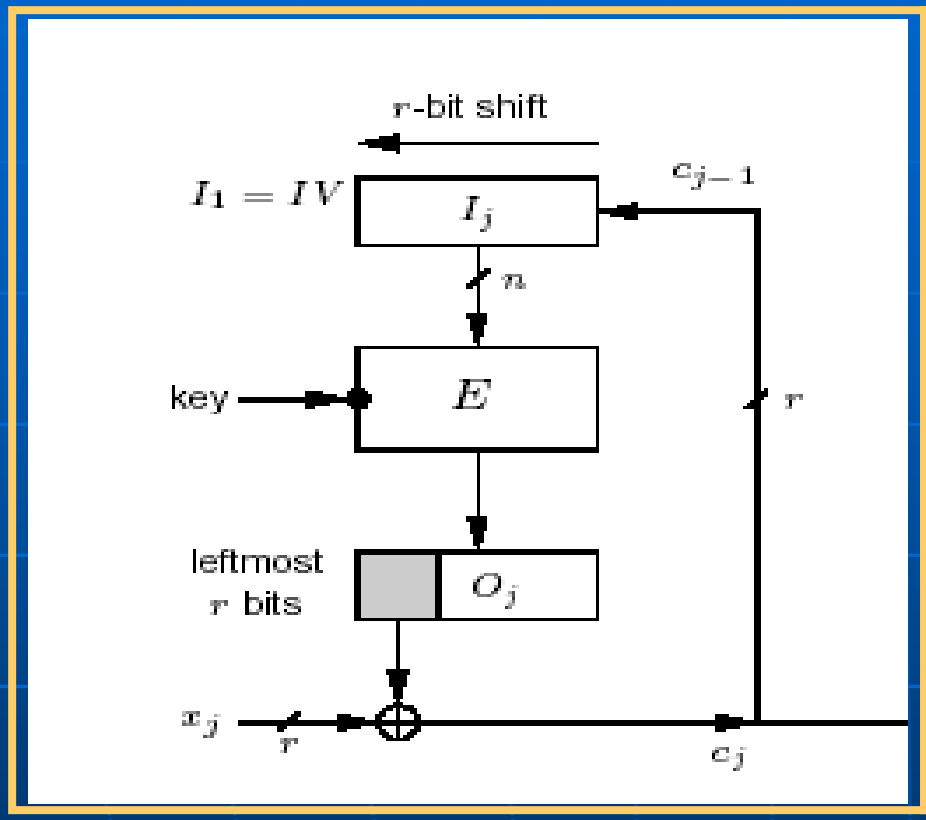
RC5

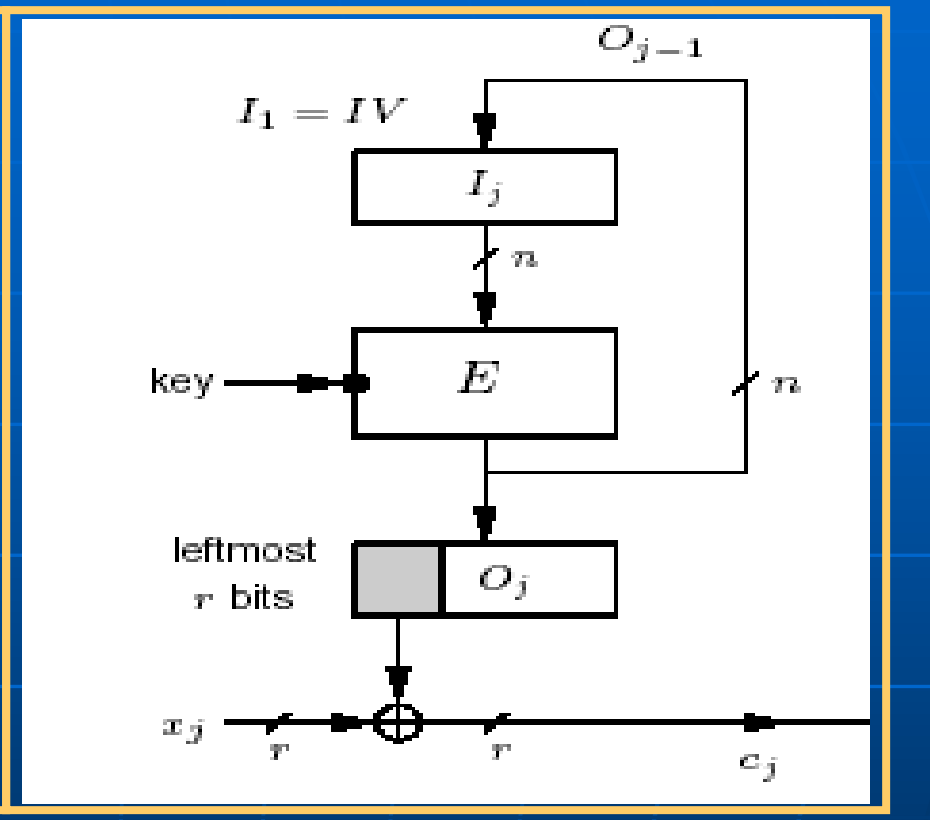
- Designed by Ronald Rivest, MIT
- Design principles
  - Simple
  - Secure
  - Fast in hardware
  - Fast in software
- RC5
  - 64-bit input; 64 bit output
  - 12 (encryption/decryption) rounds
    - Each round uses two round keys
  - 16 bytes in user secret key
  - Data dependent rotate

RC5 Components

- Encryption
- Decryption
- Decryption = Encryption^{-1}
- Round key generation

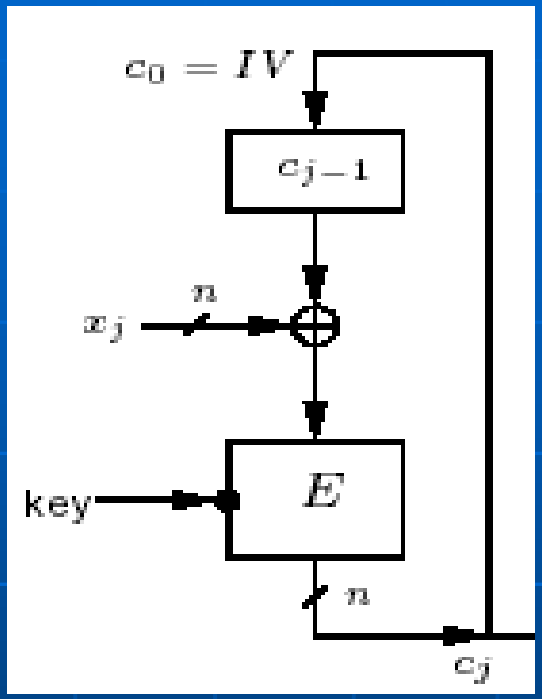
Data Confidentiality





- Stream ciphers
  - Key stream generator; encryption is a simple xor operation
  - Cipher feedback mode of DES, AES, RC5 etc..
  - Output feedback mode of DES, AES, RC5 etc..

Data Integrity



- Message authentication code
  - Detect modification to messages (how?)
  - Cipher block chaining mode (E= AES,DES,RC5)