

7-Mar-08 (1)

CEG 5010: Reconfigurable Computing Distributed Arithmetic

Philip Leong

"The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it." - Harry Emerson Fosdick

7-Mar-08 (2)

Introduction

- Distributed arithmetic (DA) computes an inner product (also known as a dot product) efficiently
- Makes use of the fact that one of the inputs is constant to achieve 50-80% reduction in gates
- Applications in fixed point DSP: FIR/IIR filter, FFT, DCT etc

7-Mar-08 (3)

2's Complement Fractions

$$x_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \quad |x_k| \leq 1$$

- $b_{kn} \in \{0,1\}$ are the bits representing the number, $b_{k,N-1}$ is the LSB
- What is the 2's complement fraction 0100 in decimal?
- What is the 2's complement fraction 1100 in decimal?
- You can also think of these as being the normal integer 2's complement numbers scaled by $2^{-(N-1)}$
- We normally use these for signal processing as multiplication does not cause these numbers to overflow i.e. the product of two numbers ≤ 1 is ≤ 1

7-Mar-08 (4)

Inner Product

$$y = \sum_{k=1}^K A_k x_k. \quad (1)$$

The A_k are fixed coefficients, and the x_k are the input data words. If each x_k is a 2's-complement binary number scaled (for convenience, not as necessity) such that $|x_k| < 1$, then we may express each x_k as

$$x_k = -b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \quad (2)$$

where the b_{kn} are the bits, 0 or 1, b_{k0} is the sign bit, and $b_{k,N-1}$ is the least significant bit (LSB).

Now let us combine Equations 1 and 2 in order to express y in terms of the bits of x_k :

$$y = \sum_{k=1}^K A_k \left[-b_{k0} + \sum_{n=1}^{N-1} b_{kn} 2^{-n} \right]. \quad (3a)$$

7-Mar-08 (5)

Main idea

$$y = \sum_{n=1}^{N-1} \left[\sum_{k=1}^K A_k b_{kn} \right] 2^{-n} + \sum_{k=1}^K A_k (-b_{k0}). \quad (3b)$$

This is the crucial step: Equation 3b defines a distributed arithmetic computation. Consider the bracketed term in Equation 3b:

$$\sum_{k=1}^K A_k b_{kn}. \quad (3c)$$

- b_{kn} is either 0 or 1 and there are k of them. Hence (3c) only has 2^K possible values as A_k never changes. Compute it by table lookup using b_{kn} ($k=1..K$) as the address

7-Mar-08 (6)

Example (32 word mem)

Table 1

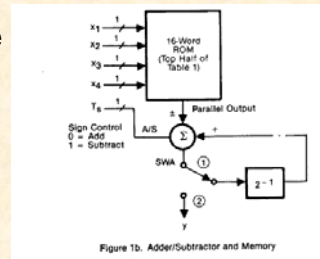
Input Code					32 Word Memory Contents
Ts	b1n	b2n	b3n	b4n	
0	0	0	0	0	0
0	0	0	0	1	A ₄ = 0.11
0	0	0	1	0	A ₃ = 0.95
0	0	0	1	1	A ₃ - A ₄ = 1.06
0	0	1	0	0	A ₂ = -0.30
0	0	1	0	1	A ₂ - A ₄ = -0.19
0	0	1	1	0	A ₂ - A ₃ = -0.65
0	0	1	1	1	A ₂ + A ₃ + A ₄ = 0.75
0	1	0	0	0	A ₁ = 0.72
0	1	0	0	1	A ₁ + A ₄ = 0.83
0	1	0	1	0	A ₁ + A ₃ = 1.67
0	1	0	1	1	A ₁ + A ₃ - A ₄ = 1.78
0	1	1	0	0	A ₁ + A ₂ = 0.42
0	1	1	0	1	A ₁ + A ₂ + A ₄ = 0.53
0	1	1	1	0	A ₁ + A ₂ + A ₃ = 1.37
0	1	1	1	1	A ₁ + A ₂ + A ₃ + A ₄ = 1.48
1	0	0	0	0	0
1	0	0	0	1	-A ₄ = -0.11
1	0	0	1	0	-A ₃ = -0.95
1	0	0	1	1	-(A ₃ + A ₄) = -1.06
1	0	1	0	0	-A ₂ = -0.30
1	0	1	0	1	-(A ₂ + A ₄) = -0.19
1	0	1	1	0	-(A ₂ + A ₃) = -0.65
1	0	1	1	1	-(A ₂ + A ₃ + A ₄) = -0.75
1	1	0	0	0	-A ₁ = -0.72
1	1	0	0	1	-(A ₁ + A ₄) = -0.83
1	1	0	1	0	-(A ₁ + A ₃) = -1.67
1	1	0	1	1	-(A ₁ + A ₃ - A ₄) = -1.78
1	1	1	0	0	-(A ₁ + A ₂) = -0.42
1	1	1	0	1	-(A ₁ + A ₂ + A ₄) = -0.53
1	1	1	1	0	-(A ₁ + A ₂ + A ₃) = -1.37
1	1	1	1	1	-(A ₁ + A ₂ + A ₃ + A ₄) = -1.48

- Input delivered serially, LSB first ($\{b_{k,N-1}\}$). The MSB is $\{b_{k0}\}$
- SWA is set to (1) except when MSB sent
- 2^{-1} is just a shift
- Note how apriori knowledge of A is needed to compute the table

7-Mar-08 (7)

Example with subtractor (16 word mem)

- In previous slide, top and bottom half of table are identical in magnitude, opposite in sign
- Since table is exponential in K, better to use a subtraction and halve table size
- $Ts=1$ on MSB 0 otherwise



7-Mar-08 (8)

Further memory reduction

- Using a $(-1,1)$ representation (instead of $(0,1)$) we can reduce memory by half again

$$x_k = \frac{1}{2} [x_k - (-x_k)], \quad (4)$$

and remember that in 2's-complement notation the negative of x_k is written as

$$-x_k = -\bar{b}_{k0} + \sum_{n=1}^{N-1} \bar{b}_{kn} 2^{-n} + 2^{-(N-1)} \quad (5)$$

where the overscore symbol indicates the complement of a bit. From Equations 2 and 5 we may rewrite Equation 4 as:

$$x_k = \frac{1}{2} \left[-(b_{k0} - \bar{b}_{k0}) + \sum_{n=1}^{N-1} (b_{kn} - \bar{b}_{kn}) 2^{-n} - 2^{-(N-1)} \right]. \quad (6)$$

7-Mar-08 (9)

$$c_{kn} = b_{kn} - \bar{b}_{kn} \quad n \neq 0 \quad (7)$$

and

$$c_{k0} = -(b_{k0} - \bar{b}_{k0}) \quad (8)$$

where the possible values of the c_{kn} , including $n = 0$, are ± 1 . Now (6) may be rewritten as

$$x_k = \frac{1}{2} \left[\sum_{n=0}^{N-1} c_{kn} 2^{-n} - 2^{-(N-1)} \right] \quad (9)$$

By substituting (9) into (1) we obtain

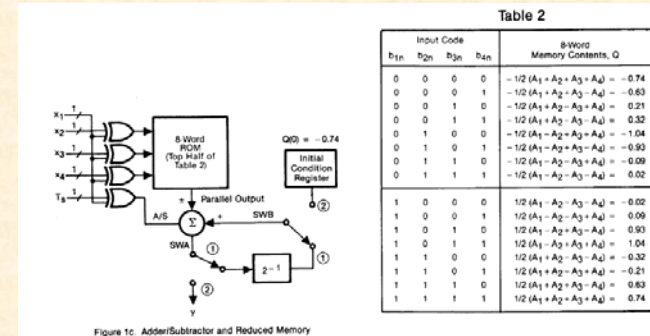
$$y = \frac{1}{2} \sum_{k=1}^K A_k \left[\sum_{n=0}^{N-1} c_{kn} 2^{-n} - 2^{-(N-1)} \right] \quad (10)$$

$$= \sum_{n=0}^{N-1} Q(b_n) 2^{-n} + 2^{-(N-1)} Q(0) \quad (11)$$

$$Q(b_n) = \sum_{k=1}^K \frac{A_k}{2} c_{kn} \quad \text{and} \quad Q(0) = \sum_{k=1}^K \frac{A_k}{2} \quad (12)$$

7-Mar-08 (10)

Example (8 word mem)



- Bits of the input represent (-1,1) weightings of A_i in the table
- Input to the ROM has changed so that the appropriate bit pattern is applied to the ROM (b_{1n} is also used to select sign)

7-Mar-08 (11)

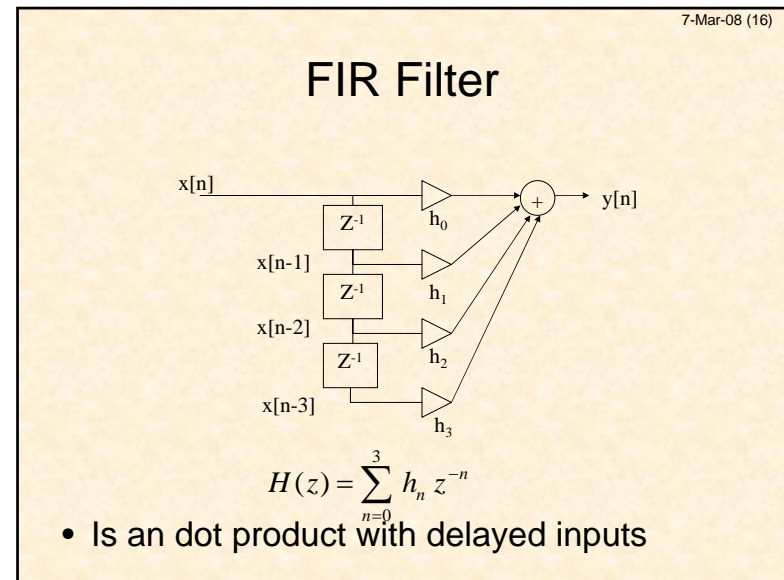
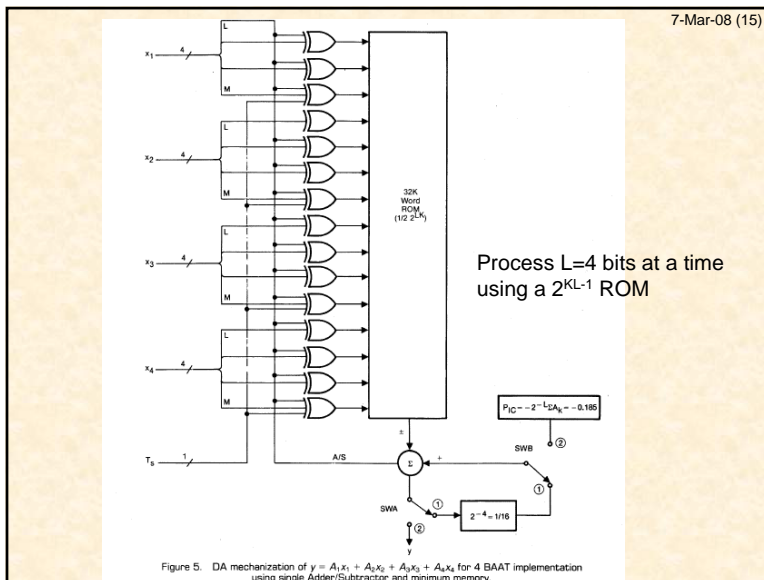
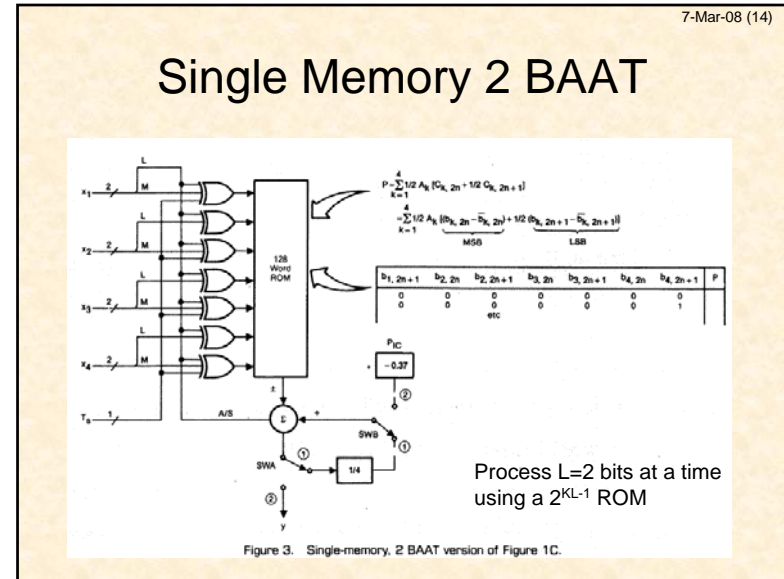
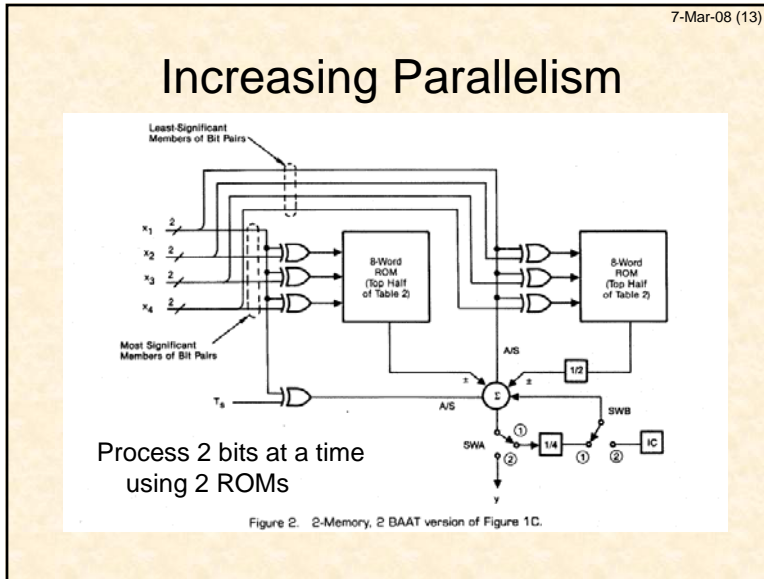
Resource Analysis

- Need 2^{K-1} entry ROM to compute inner product in N cycles
- For $K > N$ it is more efficient than a single multiplier/adder

7-Mar-08 (12)

Parallelism

- Should be apparent that the DA computation can be made more parallel
- There are two choices
 - Use L ROMs to compute L bits at a time
 - Use a single 2^{KL-1} address ROM to compute L bits at a time
 - Both cases use N/L cycles hence the speedup factor is L
- How does each scale with L?



7-Mar-08 (17)

IIR Filter

- Offer possibility of steeper slope filters than FIR at the expense of stability, overflow and higher sensitivity to coefficient values
- Called “infinite impulse response” as it uses feedback from the output
- Has both poles and zeros, FIR only has zeros
- Can be implemented as a cascade of biquad sections

7-Mar-08 (18)

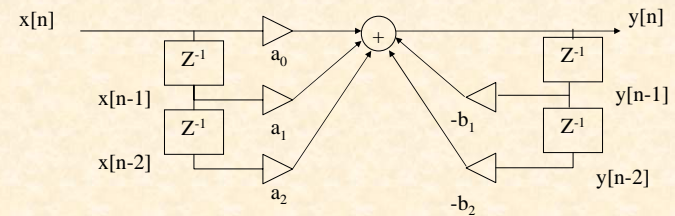
A typical biquadratic digital filter has a transfer function of the form

$$\frac{Y(z)}{X(z)} = \frac{A_0 + A_1z^{-1} + A_2z^{-2}}{1 + B_1z^{-1} + B_2z^{-2}} \quad (20)$$

where the poles are determined by the B_1 and B_2 , and the gain and zeros are determined by the A_0 , A_1 , and A_2 . The time-domain description is

$$y_n = [A_0 A_1 A_2 B_1 B_2]^T [x_n x_{n-1} x_{n-2} y_{n-1} y_{n-2}] \quad (21)$$

where the coefficient vector is $[A_0 A_1 A_2 B_1 B_2]^T$ and the data vector is $[x_n x_{n-1} x_{n-2} y_{n-1} y_{n-2}]^T$. A direct DA mechanization



7-Mar-08 (19)

IIR Filter Implementation

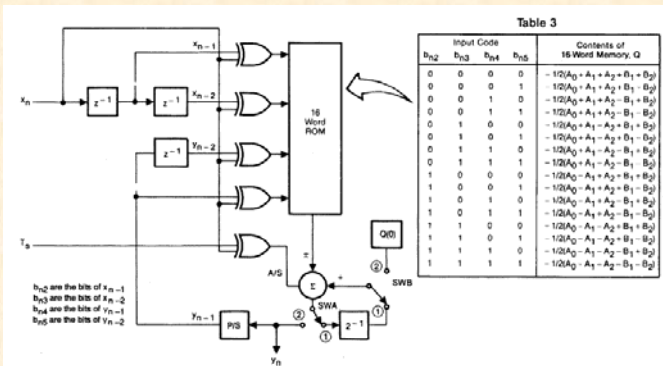


Figure 6. DA implementation of direct form of biquadratic digital filter (1 BAAT).

7-Mar-08 (20)

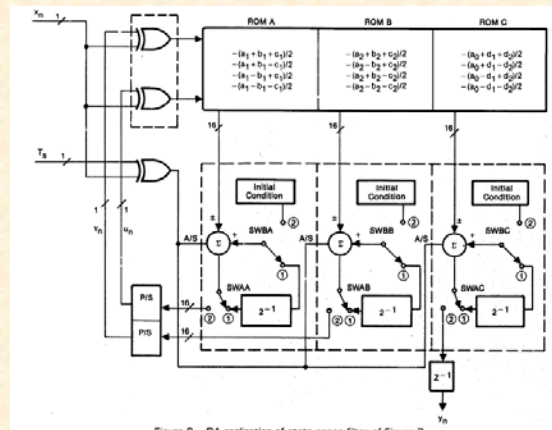


Figure 8. DA realization of state-space filter of Figure 7.

7-Mar-08 (21)

Conclusion

- Reformation of bit operations of dot product leads to DA representation where sums can be precomputed and stored in a table

7-Mar-08 (22)

Review Questions

- Make sure you understand 2's complement fractions
- Derive the DA formula of (3b) from (1) and (2)
- A moving average filter computes
 - $y(n)=0.25*x(n)+0.25*x(n-1)+0.25*x(n-2)+0.25*x(n-3)$
 - What are the contents of the 8 word ROM needed to implement this filter using the datapath of slide 10?
 - For inputs $x(n)=0.5$, $x(n-1)=0.75$, $x(n-2)=0.25$, $x(n-3)=0.5$, manually calculate the output on each cycle of execution

7-Mar-08 (23)

References

- S.A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," IEEE ASSP Magazine, July 1989, pp. 4-19.

7-Mar-08 (24)

Parity FSM

```

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity parity is
  port (clk, reset: in std_logic;
        first: in std_logic;
        din: in std_logic;
        dout: out std_logic;
        p: buffer std_logic;
        cnt: buffer std_logic_vector(2 downto 0));
end parity;

```

7-Mar-08 (25)

```

architecture rtl of parity is
type state_t is (out0, out1, odd, even);
signal curstate, nextstate : state_t;
signal inc : std_logic;
begin
  stateclkd: process(clk, reset)
  begin
    if (reset = '1') then
      curstate <= out0;
    elsif (rising_edge(clk)) then
      curstate <= nextstate;
      if (inc = '1') then
        cnt <= cnt + 1;
      else
        cnt <= "000";
      end if;
      dout <= p;      -- one cycle after entering out0/out1
    end if;
  end process;

```

7-Mar-08 (26)

```

statecomb: process(curstate, first, din, cnt)
begin
  inc <= '1'; p <= '0';
  case curstate is
    when out0 =>
      inc <= first;      -- when we enter non-out state, cnt=1
      if (first = '1' and din = '0') then
        nextstate <= even;
      elsif (first = '1' and din = '1') then
        nextstate <= odd;
      else
        nextstate <= out0;
      end if;
    when out1 =>
      p <= '1';
      inc <= first;      -- when we enter non-out state, cnt=1
      if (first = '1' and din = '0') then
        nextstate <= even;
      elsif (first = '1' and din = '1') then
        nextstate <= odd;
      else
        nextstate <= out1;
      end if;

```

7-Mar-08 (27)

```

    when even =>
      if (cnt = 7) then
        inc <= '0';
        if (din = '1') then
          nextstate <= out0;
        else
          nextstate <= out1;
        end if;
      else
        if (din = '1') then
          nextstate <= odd;
        else
          nextstate <= even;
        end if;
      end if;

```

7-Mar-08 (28)

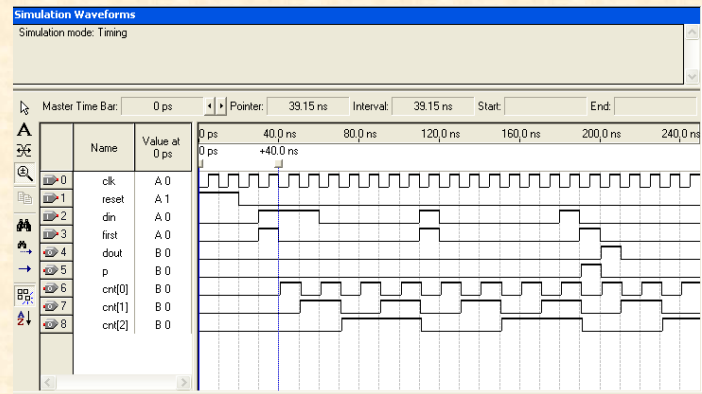
```

    when odd =>
      if (cnt = 7) then
        inc <= '0';
        if (din = '1') then
          nextstate <= out1;
        else
          nextstate <= out0;
        end if;
      else
        if (din = '1') then
          nextstate <= even;
        else
          nextstate <= odd;
        end if;
      end if;
    end case;
  end process;
end rtl;

```

7-Mar-08 (29)

Simulation



7-Mar-08 (30)

All Synchronous Solution

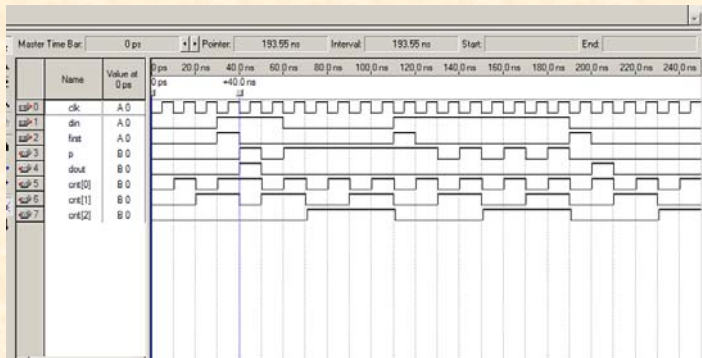
```

architecture rtl of parity is
  signal lp : std_logic; -- latched byte parity
  signal inc : std_logic;
begin
  stateclkd: process(clk) begin
    if (rising_edge(clk)) then
      if first = '1' then
        p <= din;
      elsif din = '1' then
        p <= not p;
      end if;
      if first = '1' then
        cnt <= "001";
        dout <= not p;
      else
        cnt <= cnt + 1;
        dout <= '0';
      end if;
    end if;
  end process;
end rtl;

```

7-Mar-08 (31)

Synchronous Parity FSM



7-Mar-08 (32)

Parity FSM

```

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

```

```

entity parity is
  port (clk, reset: in std_logic;
        first: in std_logic;
        din: in std_logic;
        p : buffer std_logic; -- parity output per bit
        dout: out std_logic;
        cnt: buffer std_logic_vector(2 downto 0));
end parity;

```

7-Mar-08 (33)

```

architecture rtl of parity is
type state_t is (init, odd, even);
signal curstate, nextstate : state_t;
signal lp : std_logic;      -- latched byte parity
signal inc : std_logic;
begin
stateclkd: process(clk, reset)
begin
if (reset = '1') then
curstate <= init;
elsif (rising_edge(clk)) then
curstate <= nextstate;
lp <= p;
dout <= lp;
if (inc = '1') then
cnt <= cnt + 1;
else
cnt <= "000";
end if;
end if;
end process;

```

7-Mar-08 (34)

```

statecomb: process(curstate, first, din, cnt)
begin
p <= '0'; inc <= '1';
case curstate is
when init =>
inc <= first;      -- when we enter non-init state, cnt=1
if (first = '1' and din = '0') then
nextstate <= even;
elsif (first = '1' and din = '1') then
nextstate <= odd;
else
nextstate <= init;
end if;

```

7-Mar-08 (35)

```

when even =>
if (cnt = 7) then
inc <= '0'; p <= not din;
nextstate <= init;
elsif (din = '1') then
nextstate <= odd;
else
nextstate <= even;
end if;
when odd =>
if (cnt = 7) then
inc <= '0'; p <= din;
nextstate <= init;
elsif (din = '1') then
nextstate <= even;
else
nextstate <= odd;
end if;
end case;
end process;
end rtl;

```

7-Mar-08 (36)

Parity FSM

