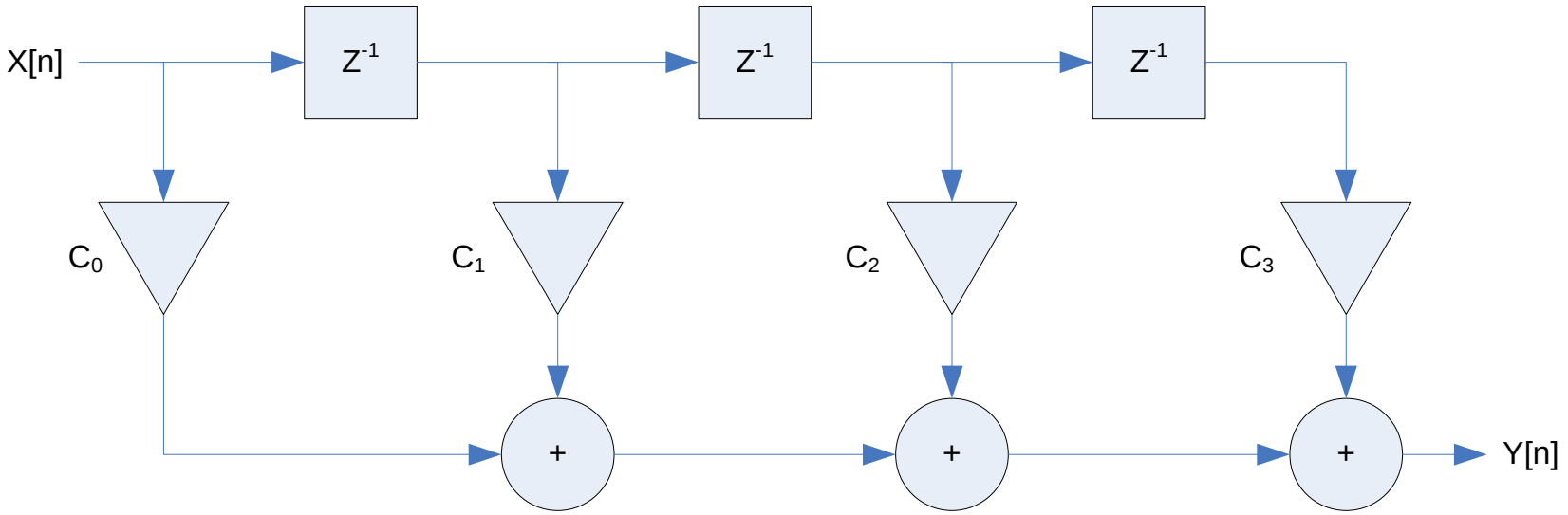
AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

PETER HØGH MIKKELSEN
29. JANUARY, 2016

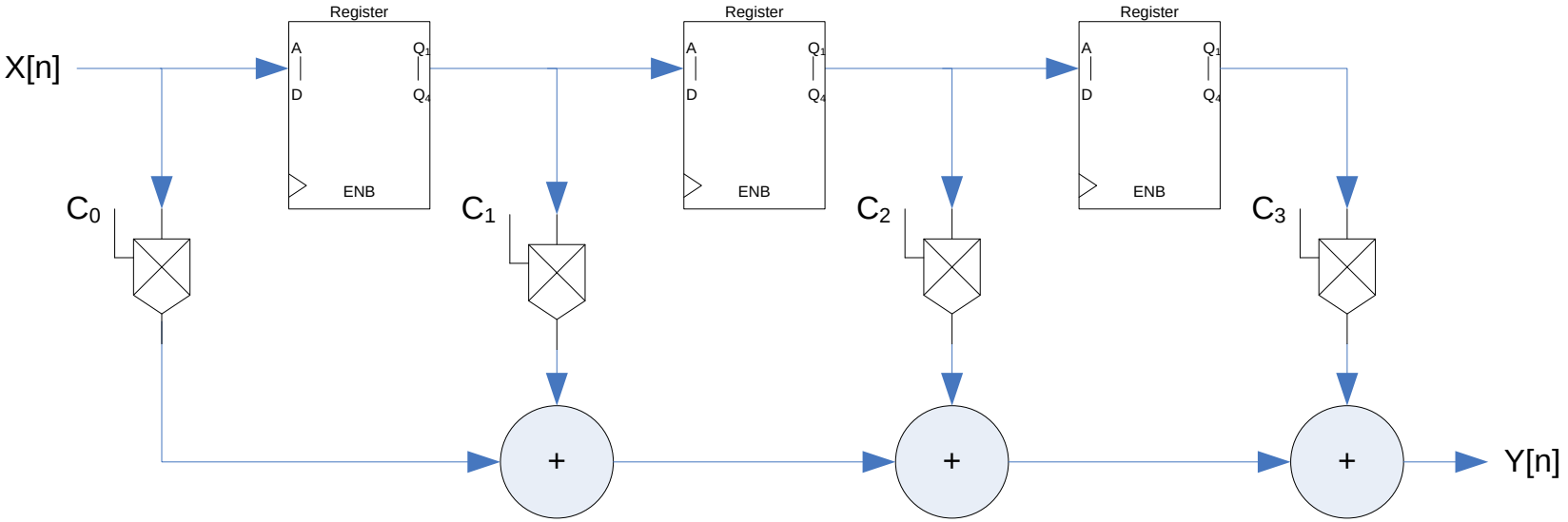
ARITHMETIC & FILTERS IN FPGAS

BASIC FIR FILTER

Direct Form FIR Filter

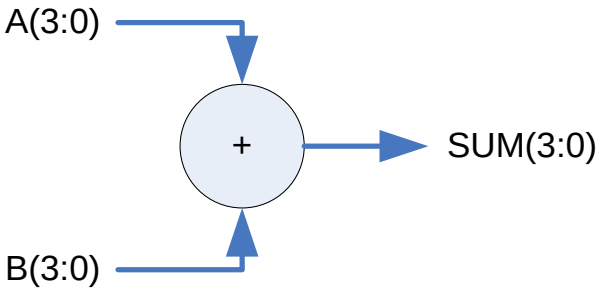


FILTER COMPONENTS

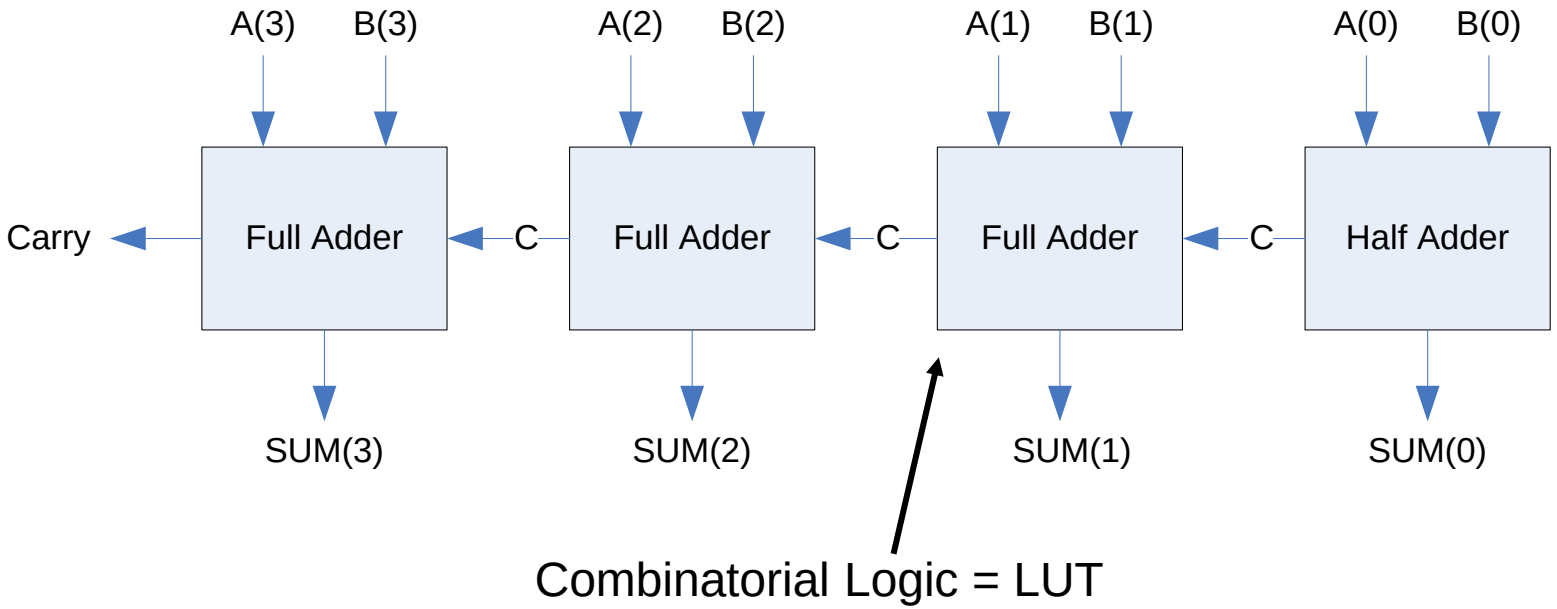


To build a filter we need: Registers, Multipliers and Adders

ADDERS

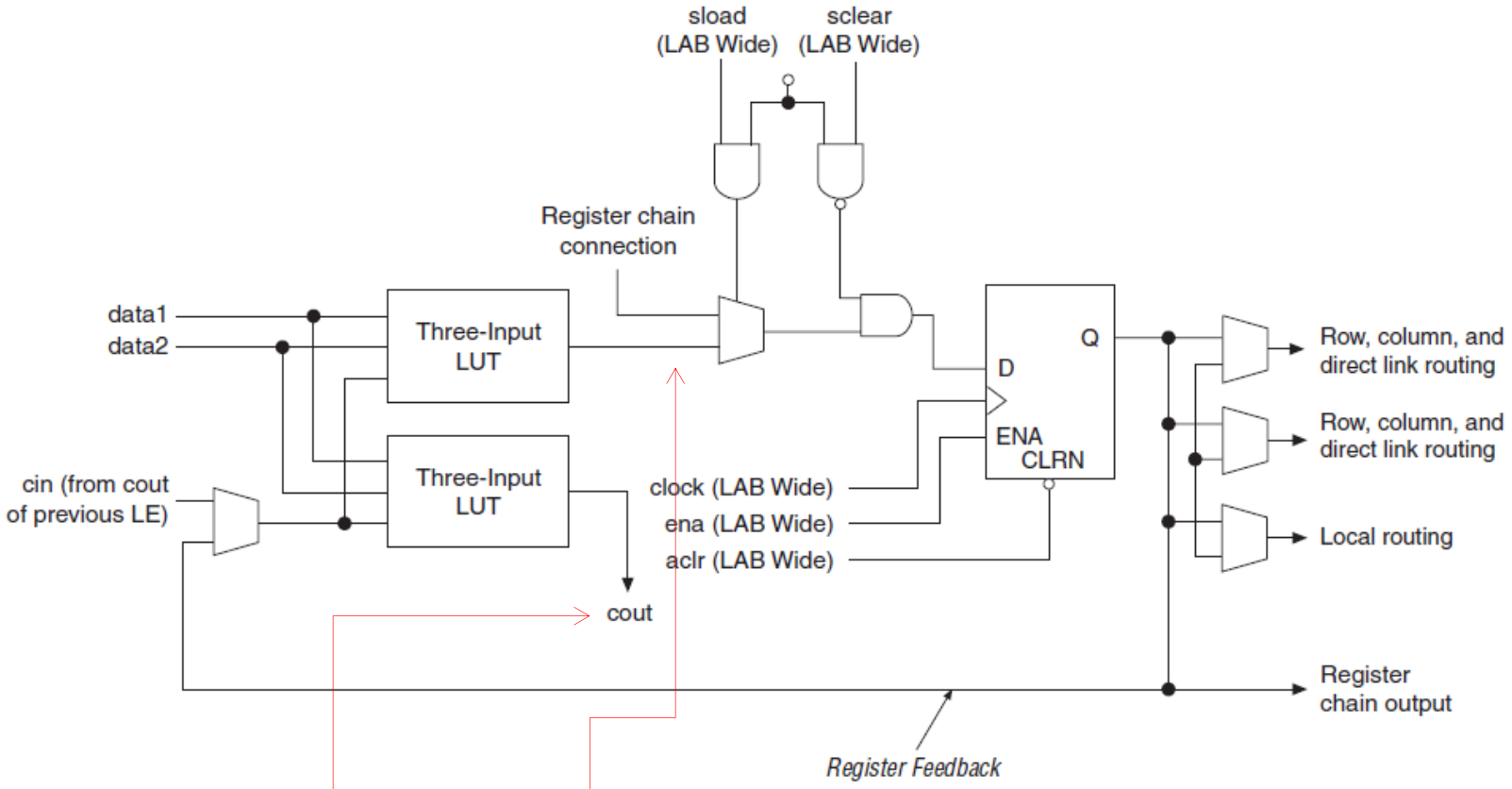



```
SUM <= A + B;
```



LE – ARITHMETIC MODE

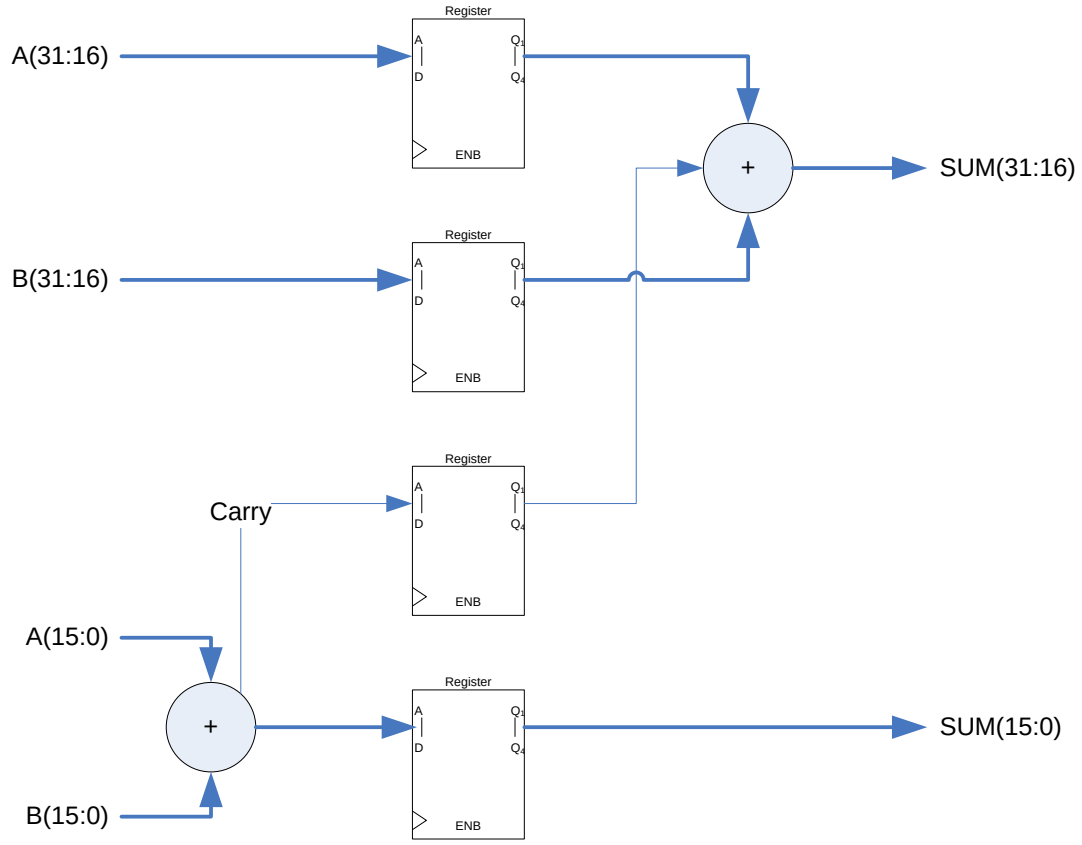
Figure 2–4. LE in Arithmetic Mode



SUM = A xor B xor Cin

Cout = (A and B) or (A and Cin) or (B and Cin)

PIPELINED ADDER



›Propagation delay too long: Add registers (pipelining)

MULTIPLIERS

- Multiplication is a sum of additions:
- P = A*X = 5 * 17 = 17 + 17 + 17 + 17 + 17

```
if reset = '0' then
  sum <= 0;
  sum_cnt <= 0;
elsif rising_edge(clk) then
  if sumCnt >= 5 then
    P <= sum;
  else
    sum <= sum + 17;
    sumCnt <= sumCnt + 1;
  end if;
end if;
```

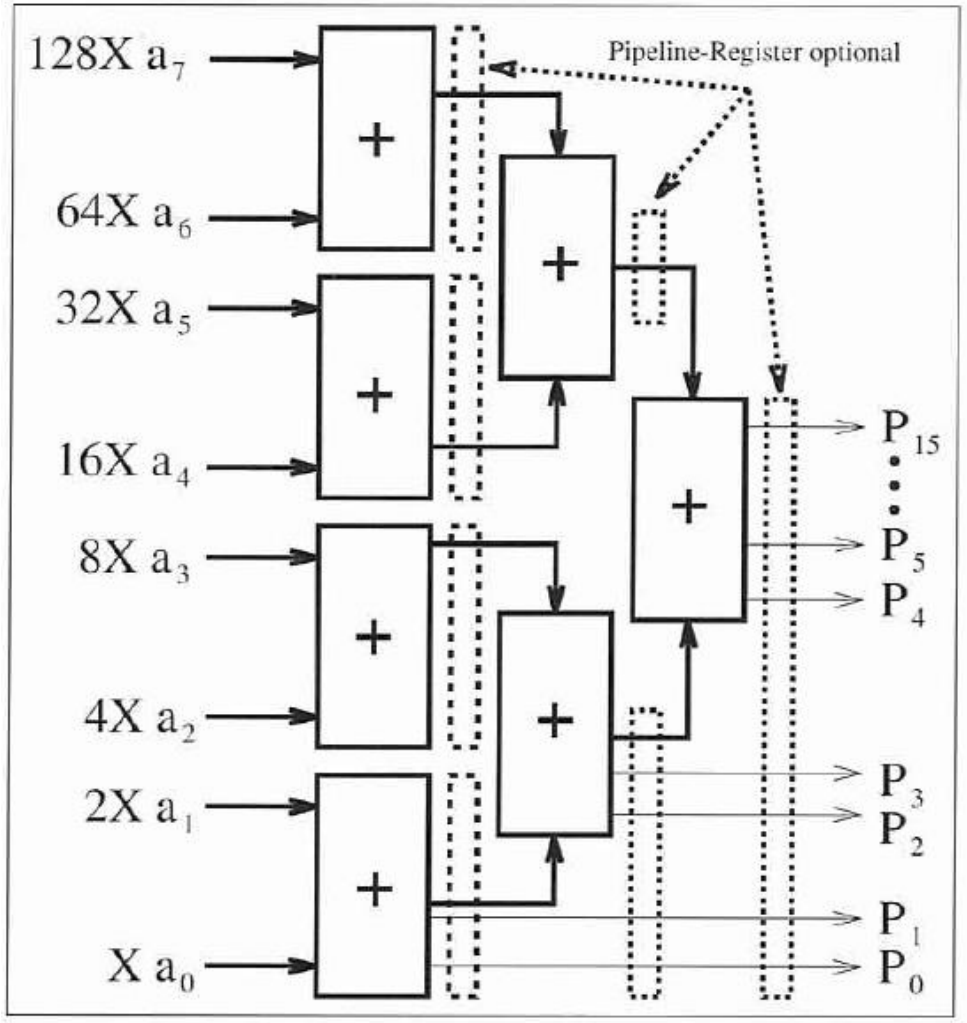
BINARY MULTIPLIER

$$Product = A \cdot X = \sum_{k=1}^{N-1} a_k \cdot 2^k \cdot X \qquad a_k \in [0,1]$$

$$P = 3 \cdot 17 = [011] \cdot 17$$
$$P = (1 \cdot 2^0 \cdot 17) + (1 \cdot 2^1 \cdot 17) + (0 \cdot 2^2 \cdot 17) = 51$$
$$P = (1 \cdot 17) \ll 0 + (1 \cdot 17 \ll 1) + (0 \cdot 17 \ll 2)$$

```
constant A : unsigned(4 downto 0) := "00011";
constant X : unsigned(4 downto 0) := "10001";
signal k : integer range 0 to 4 := 0;
if rising_edge(clk) then
  if A(k) = '1' then
    P <= P + shift_left(X, k);
  end if;
  k <= k + 1;
end if;
```

FAST MULTIPLIERS



- Parallized to increase performance
- Increased area
- Extensive use of shift registers

FAST MULTIPLIER CONT.

```
if rising_edge(clk) then
  for i in 0 to N/2 generate
    s1(i) <= a(2*i) and (X << 2*i) +
             a(2*i+1) and (X << (2*i+1));
  end generate;

  for i in 0 to N/4 generate
     s2(i) <= s1(2*i) + s2(2*i+1);
  end generate;

  s3 <= s2(0) + s2(1);
end if;

P <= s3 & s2(1 downto 0) & s1(1 downto 0);
```

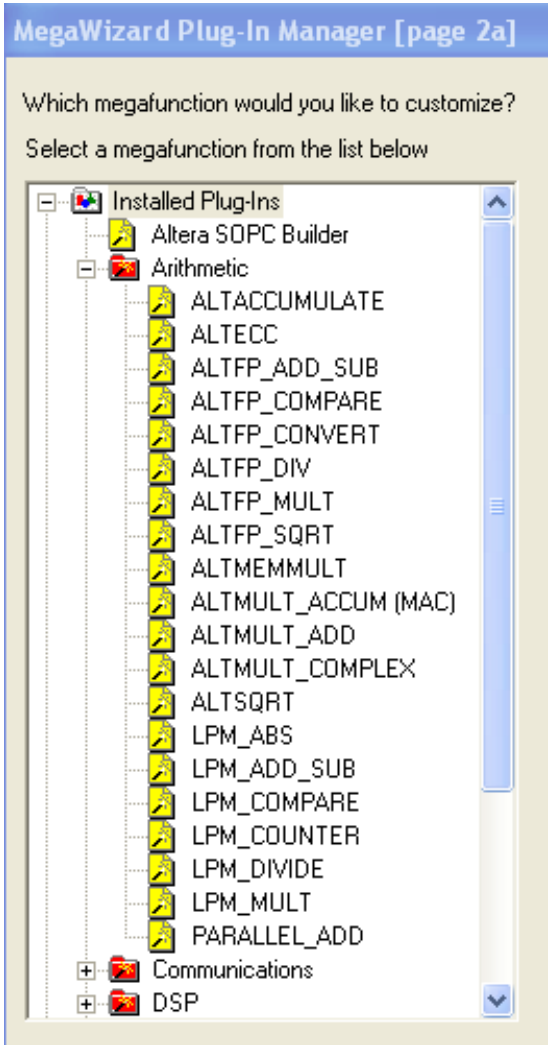
Note that this will unfold into N/4 VHDL statements and NOT be looped through at run-time

ARITHMETIC FUNCTIONS

- Silicon producers provide special components for arithmetic.
  Altera provides among other:
- LPM_ADD_SUB
- LPM_MULT

- These can be configured for:
- Pipelining, Latency, Constants..

- As they are vendor specific, you may have to rewrite code if switching to another vendor.
- LPM components are mature and portable.



ARITHMETIC OPERATORS

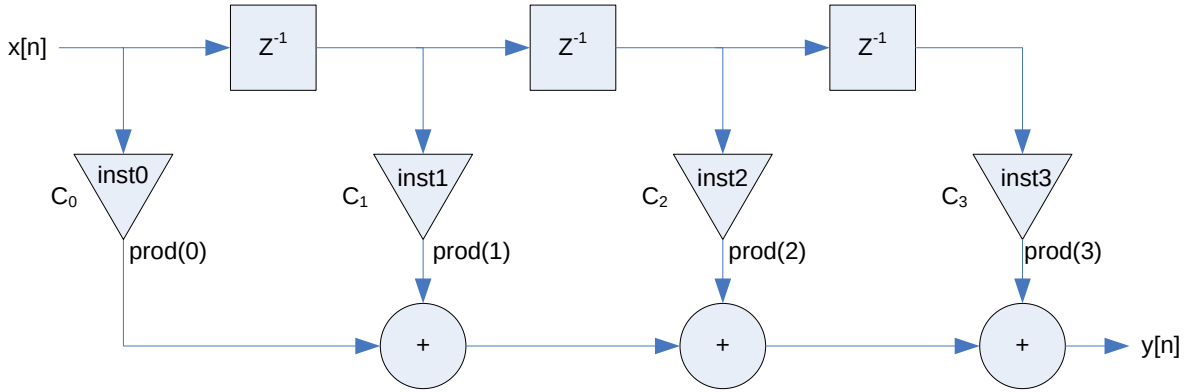
- Package NUMERIC_STD supports:
- **+** (signed, unsigned, integer)
- **-** (signed, unsigned, integer)
- ***** (signed, unsigned)
- **/** (Avoid!)

- Operators are functions (y=a+b ⇔ y="+"(a,b)) written in combinatorial style.
- Pipelining, memory a.o. is not possible.
- RTL functions are recognized during fitting and implemented as LPM functions

FIR FILTERS IN FPGAS

- FPGA allows us to implement structure directly
- Designs may be serialized / parallelized to need
  - Parallel: High throughput / High area consumption
  - Serial: Lower throughput / Low area consumption
- Filter structure and design choices affect efficiency

DIRECT FORM FIR IMPLEMENTATION



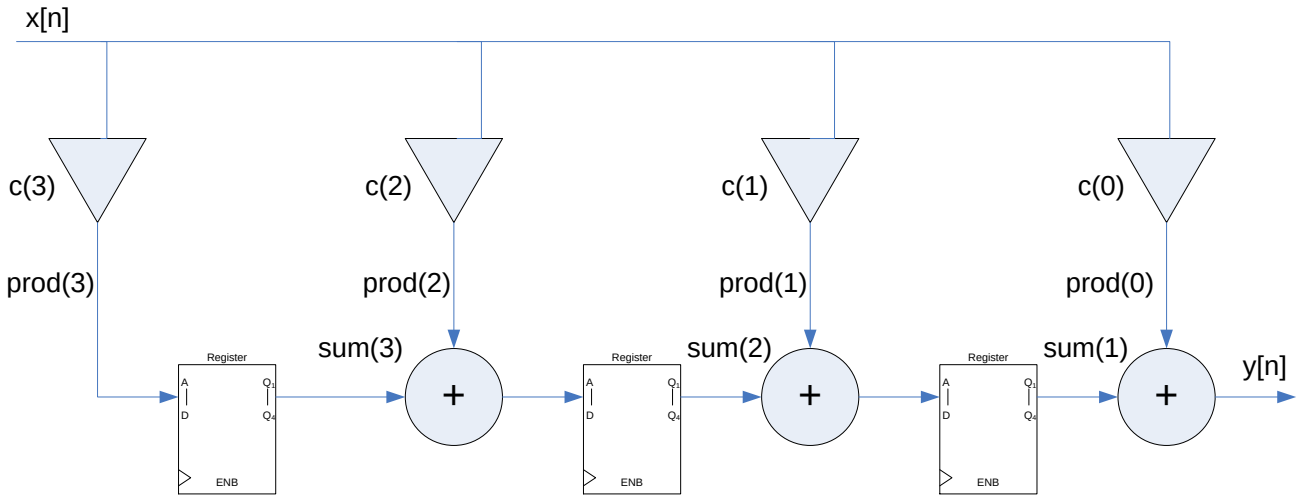

```
if rising_edge(clk) then
  y <= prod(0) + prod(1) + prod(2) + prod(3);
  x(1) <= x(0);
  x(2) <= x(1);
  x(3) <= x(2);
end if;

inst0: Lpm_mult port map(dataa => x(0), datab => c(0),
result => prod(0));
...
```

Sum of products

Delay Line for X[n]

TRANSPOSED FIR




```
if rising_edge(clk) then
  y       <= prod(0) + sum(1);
  sum(1)  <= prod(1) + sum(2);
  sum(2)  <= prod(2) + sum(3);
  sum(3)  <= prod(3);
end if;
Inst1: Lpm_mult port map(clock => clk,
dataa => x, datab => c(0), result => prod(0));
Inst2: ...
```

Sum of two + Delay Line

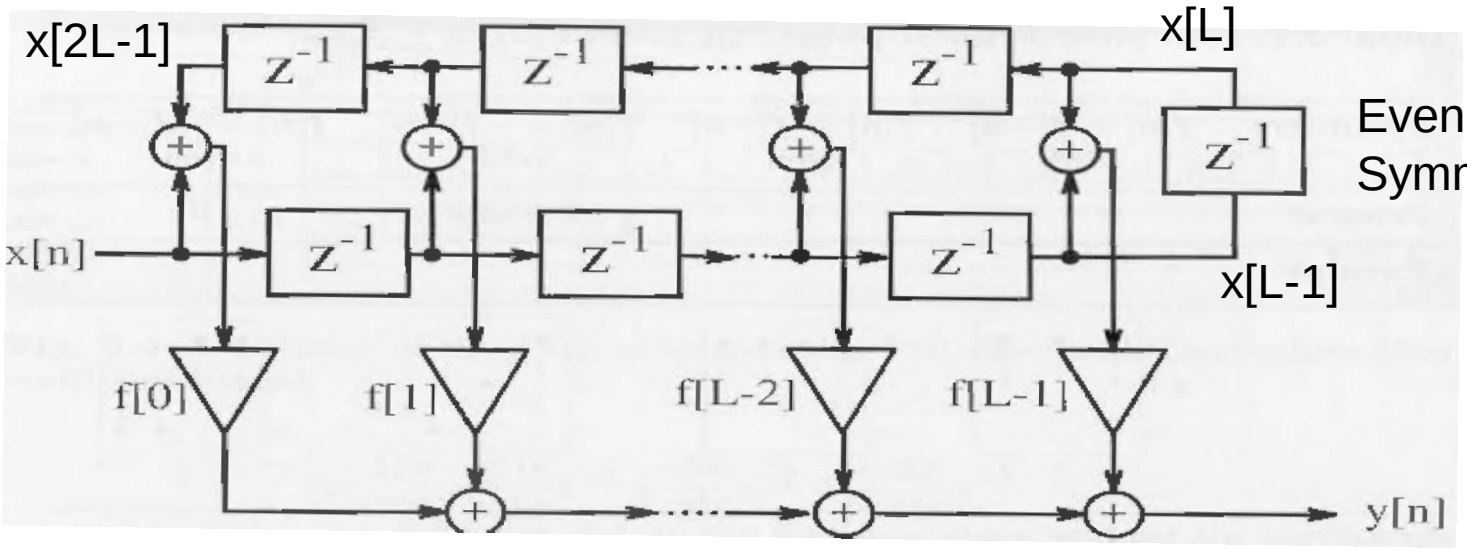
Same x input for all multipliers

FIR FILTER SYMMETRY

- Linear Phase Filters have inherent symmetries:
- Symmetric: f[n] = f[-n]
- Antisymmetric: f[n] = -f[-n]

- Symmetry yields a reduced multiplier implementation, as symmetric tabs can be summed before multiplying

FIR FILTER SYMMETRY

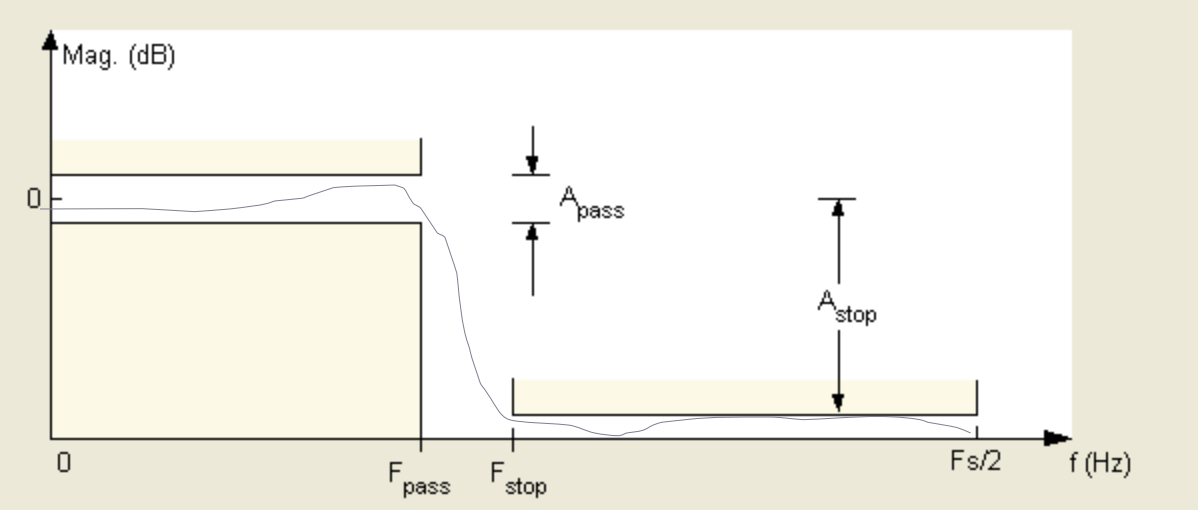



```
prod(0) := (others=>'0');

if rising_edge(clk) then
 for i in 0 to (L-1) loop
   prod(i+1) := prod(i) + f(i)*(x(i) + x(2*L-1 -i));
 end loop;  -- i
end if;
```

Nbr taps = 2*L
Nbr adders = 2*L
Nbr mult. = L

FIR FILTER DESIGN METHODS



- Equiripple / Least Squares:
- Linear phase
- Symmetric
- Use Matlab or GNU Octave to find coefficients!
- See the Wiki for a walk-through
- Verify fixed point conversion before implementation!!

FIXED POINT ARITHMETIC

- Consider the filter:
- $F(Z) = 0.4831 + 0.8365Z^{-1} + 0.2241Z^{-2} - 0.1294Z^{-3}$

- We are using 8-bit (+ 1 sign bit) fixed point 2's complement:

- Multiply coefficients by $2^8 = 256$
- $F_1(Z) = 256*(0.4831 + 0.8365Z^{-1} + 0.2241Z^{-2} - 0.1294Z^{-3})$ ➔
- $F_1(Z) = 123{,}67 + 214{,}144Z^{-1} + 57{,}370Z^{-2} - 33{,}126Z^{-3})$ ➔
- $F_1(Z) = 124 + 214Z^{-1} + 57Z^{-2} - 33Z^{-3})$ ➔ Now in 8-bit range!

- Divide the output by 256 (aka shift right 8 bit!)
- $Y[n] = x[n] * F_1(Z) / 256 = (x[n]* F_1(Z)) >> 8$

- Simulate quantized filter to verify result

DYNAMIC GROWTH

- As the coefficients are fixed and well-known, we can calculate the max number of bits used on the output:
- Growth < log2(sum(abs(coefficients)))
- Ex: Data in width N_in = 8, Coeffs = [10 34 -23 0]
- Growth < log2(10 + 34 + 23) = 6.06 → 7
- N_out = N_in + Growth = 8 + 7 bits = 15 bits

OPTIMIZATION

› We can approximate the coefficients using “Power og Two Arithmetic”

› Example:
› y = x[n] * 3.75
› y = x[n] * (2 + 1 + ½ + ¼)
› y = (x[n] * 2) + x[n] + (x[n] * ½) + (x[n] * ¼)
› y = (x[n] << 1) + x[n] + (x[n] >> 1) + (x[n] >> 2)
(Horners Method)

› -Or optimal CSD:
› y = x[n] * 4 – x[n] / 4 = (x[n] << 2) – (x[n] >>2)
› x/2, x/4 is allowed and will be translated to shifts

› May sacrify precision, but is fast and not too area consuming

OPTIMUM DIRECT FIR FILTER

Original

```
if rising_edge(clk) then
  y <= 2*tap(1) + tap(1) +
       tap(1)/2 + tap(1)/4 +
       2*tap(2) + tap(2) +
       tap(2)/2 + tap(2)/4 -
       tap(3) - tap(0);
  tap(3) <= tap(2);
  tap(2) <= tap(1);
  tap(1) <= tap(0);
  tap(0) <= x;
end if;
```

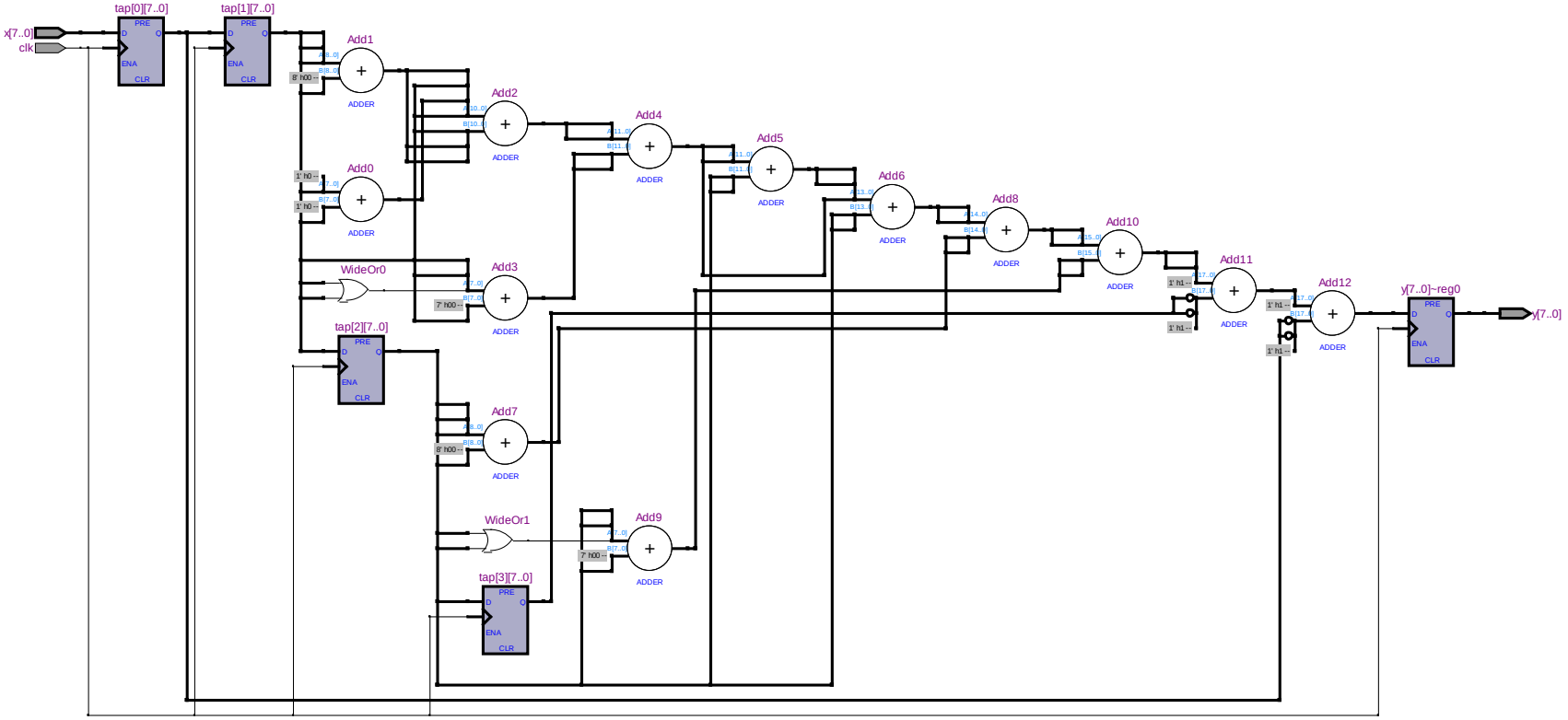
Optimized

```
if rising_edge(clk) then
  t1 <= tap(1) + tap(2);
  t2 <= tap(0) + tap(3);
  t3 <= 4*t1 - t1/4;
  t4 <= -t2;
  y <= t3 + t4;
  tap(3) <= tap(2);
  tap(2) <= tap(1);
  tap(1) <= tap(0);
  tap(0) <= x;
end if;
```

Coefficients: [-1.0 3.75 3.75 -1.0]

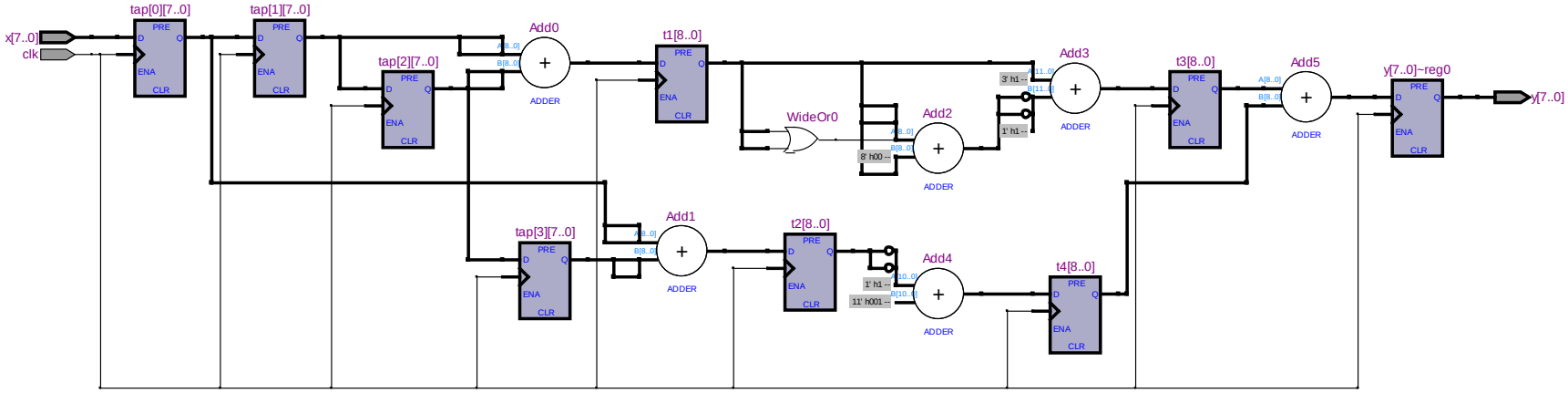
› The second is optimized for symmetry, CSD, pipelining

ORIGINAL FIR - RTL VIEW



› 107 LE (40 dedicated registers)
› Fmax = 89 MHz

OPTIMIZED FIR – RTL VIEW



- 83 LE (73 dedicated registers)
- Fmax = 220MHz

GNU OCTAVE / MATLAB

- Matlab = Matrix Laboratory
- Command prompt driven

- Horizontal vector : a = [1 2 3]

- Vertical vector: b = [1 ; 2 ; 3]

- Multiplying vectors: c = b * a = $a_1*b_1 + a_2*b_2 + a_3*b_3$

- b = fir1(N, w0); % w0 = $f/f_{nyquist}$