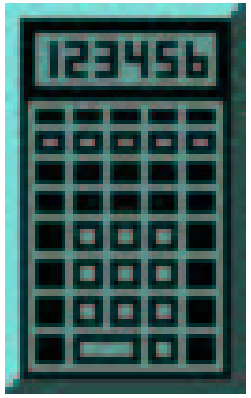
Arithmetische Schaltungen



Gliederung

- Addierer
  - Verschiedene Architekturen
- Multiplizierer
  - Verschiedene Architekturen
- Barrel Shifter
- Clock Gating
- Clock Skew

Motivation

- Addition und Multiplikation sind bereits vordefiniert
  - auf Integern (in VHDL) und
  - auf std_logic_vector (im Package IEEE)
- Warum kann man dies nicht verwenden?

```
signal a,b,c : integer;
...
c <= a + b;
```

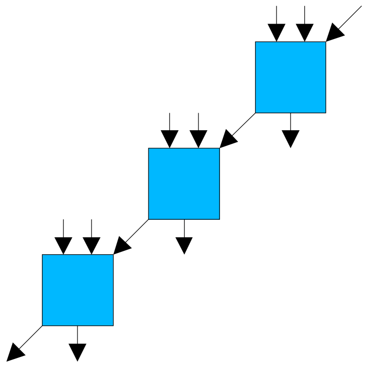
Motivation (2)

- ... weil vordefinierte Arithmetik nicht unbedingt optimal synthetisiert wird
  - Abhängig vom Synthesetool
  - Spezielle Architektur kann notwendig sein
    - Beschränkung des Delays
    - Beschränkung des Energieverbrauchs
- Außerdem: Praxis in VHDL

Addierer

```
library ieee;
use ieee.std_logic_1164.all;
entity adder is
  generic (n : integer := 32);
  port (a, b : in std_logic_vector(n-1 downto 0);
        cin  : in std_logic;
        s    : out std_logic_vector(n-1 downto 0);
        cout : out std_logic);
end adder;
```

Carry Ripple Adder

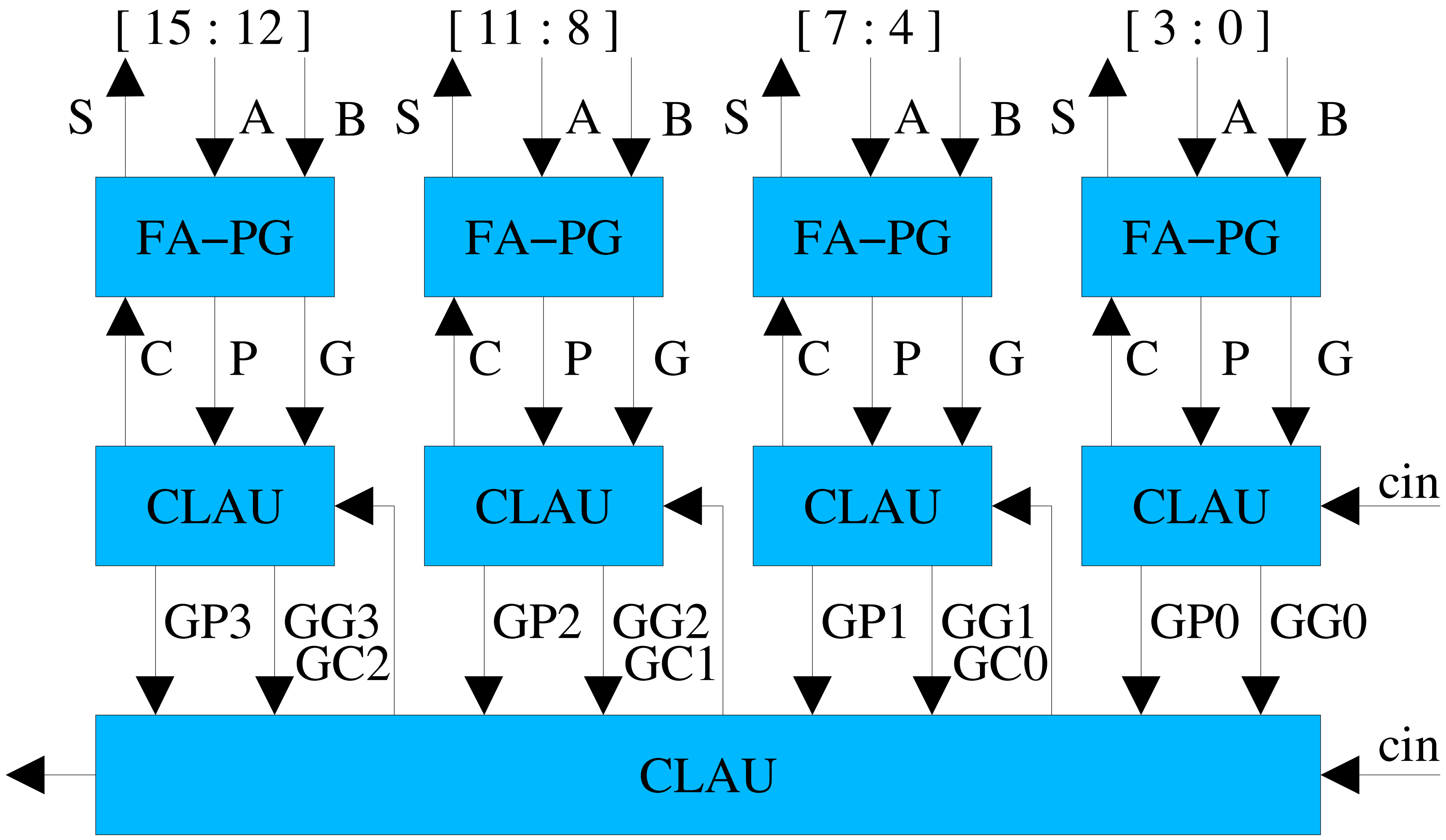


```
architecture carry_ripple of adder is
  component fa
     port(a, b, cin : in  std_logic;
          s, cout   : out std_logic);
  end component;
  signal c : std_logic_vector(a'length downto 0);
begin
  c(0) <= cin;
  cout <= c(a'length);
  gen : for j in a'range generate
    inst : fa port map(a => a(j), b => b(j),
                       cin => c(j), s => s(j),
                       cout => c(j+1));
  end generate;
end carry_ripple;
```

Carry Look Ahead Adder

- Ausgehendes Carry wird zerlegt
  - Propagate Carry
  - Generate Carry
- Dies kann schnell für jedes Bit bestimmt werden
- Daraus wird "richtiges" Carry berechnet

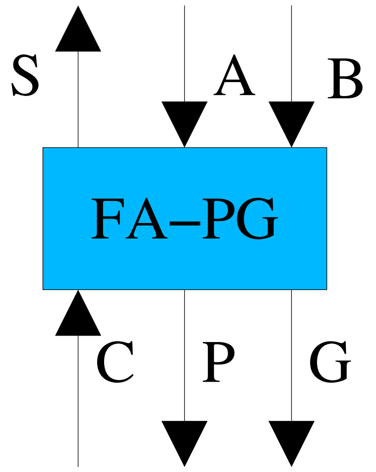
Carry Look Ahead, 16 Bit



FA–Propagate/Generate

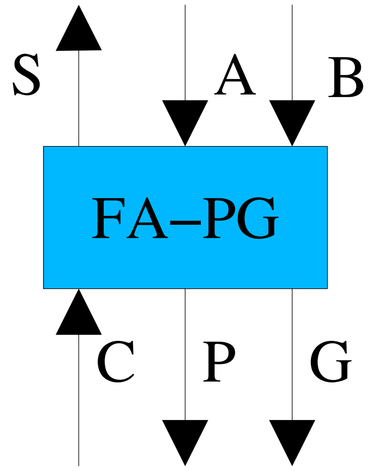
- Voll–Addierer mit carry propagate and generate

```
library ieee;
use ieee.std_logic_1164.all;

entity fapg is
   port(a, b, cin : in  std_logic;
        s, p, g   : out std_logic);
end fapg;
```



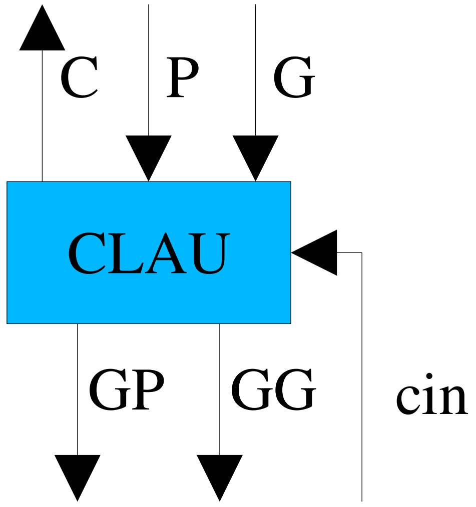
FA–Propagate/Generate (2)

```
architecture rtl of fapg is
signal gsig, psig : std_logic;
begin
   gsig <= a and b;
   psig <= a xor b;
   s    <= psig xor cin;
   g    <= gsig;
   p    <= psig;
   assert (g==’1’ nand p==’1’)
      report "wrong g or p"
      severity error;
end rtl;
```



Carry Look Ahead Unit

```
library ieee;
use ieee.std_logic_1164.all;
entity clau is
 port(p, g : in  std_logic_vector(3 downto 0);
      cin  : in  std_logic;
      cout : out std_logic_vector(3 downto 0);
    gp, gg : out std_logic);
end clau;
```



Carry Look Ahead Unit (2)

```
architecture rtl of clau is
begin
  cout(0) <= cin;
  cout(1) <= (cin and p(0)) or g(0);
  cout(2) <= (cin and p(0) and p(1))
          or (g(0) and p(1))
          or  g(1);
  cout(3) <= (cin and p(0) and p(1) and p(2))
          or (g(0) and p(1) and p(2))
          or (g(1) and p(2))
          or  g(2);
  ...
```

Carry Look Ahead Unit (3)

```
architecture rtl of clau is
begin
  ...
  gg <= (g(0) and p(1) and p(2) and p(3))
     or (g(1) and p(2) and p(3))
     or (g(2) and p(3))
     or  g(3);
  gp <= p(3) and p(2) and p(1) and p(0);
  assert (gg==’1’ nand gp==’1’)
      report "wrong g or p"
      severity error;
end rtl;
```

16–Bit CLA Adder

```
library ieee;
use ieee.std_logic_1164.all;

entity cla16 is
 port(a, b : in  std_logic_vector(15 downto 0);
      cin  : in  std_logic;
      s    : out std_logic_vector(15 downto 0);
      cout : out std_logic);
end cla16;
```

16–Bit CLA Adder (2)

```
architecture rtl of cla16 is
   component fapg
   port(a, b, cin : in  std_logic;
        s, p, g   : out std_logic);
   end component;

   component clau
   port(p, g : in  std_logic_vector(3 downto 0);
        cin  : in  std_logic;
        cout : out std_logic_vector(3 downto 0);
      gp, gg : out std_logic);
   end component;
...
```

16–Bit CLA Adder (3)

```
   ...
  signal p, g, c : std_logic_vector(15 downto 0);
  signal gp, gg, gc
                 : std_logic_vector(3 downto 0);
 begin
   fagen : for i in 0 to 15 generate
      fapg0 : fapg
            port map(a => a(i), b => b(i),
                     cin => c(i), s => s(i),
                     p => p(i), g => g(i));
   end generate; -- of fagen
   ...
```

16–Bit CLA Adder (4)

```
  claugen : for i in 0 to 3 generate
     clau0 : clau
             port map(p => p(i*4+3 downto i*4),
                      g => g(i*4+3 downto i*4),
                    cin => gc(i),
                   cout => c(i*4+3 downto i*4),
                     gp => gp(i),
                     gg => gg(i));
  end generate; -- of claugen
  clau1 : clau port map(p => gp, g => gg,
       cin => cin, cout => gc, gp => open,
       gg => cout);
end rtl;
```

Multiplizierer

- Multiplikation mit Carry Save Addern von natürlichen Zahlen
- Multiplikation von Zweierkomplementzahlen
- Booth Encoding
- Booth – Wallace Tree Multiplier

Entity

```
library ieee;
use ieee.std_logic_1164.all;

entity mult is
 generic(n : natural := 4);
 port(a, b : in  std_logic_vector(n-1 downto 0);
      prod : out std_logic_vector(
                                  2*n-1 downto 0));
end mult;
```

Beispiel

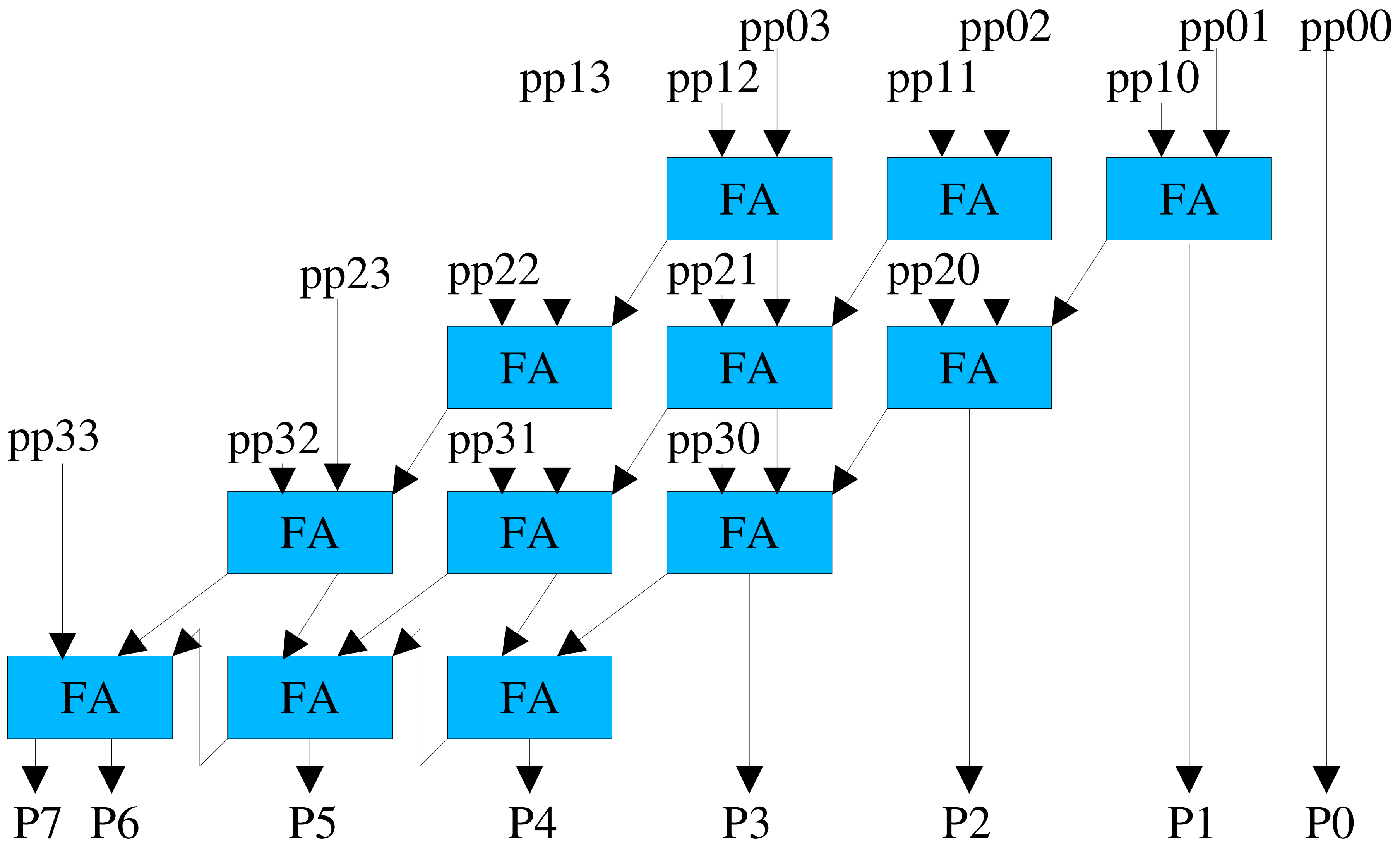
- 11 x 13 = 143

```
    1011x1101
-------------
    1011
   0000
  1011
 1011
--------
10001111
```

Partialprodukte

- Berechnung: AND–Gatter
- Addieren: mit Carry–Save–Adder

Carry Save Adder



Architecture 1

```
architecture rtl1 of mult is
subtype pary1 is std_logic_vector(n-1 downto 0);
type    pary is array(0 to n) of pary1;
signal  pp, pc, ps : pary;

component fa
   port(a, b, cin : in  std_logic;
        s, cout   : out std_logic);
end component;

begin
  ...
```

Architecture 1 (cont.)

```
begin
  pgen : for j in 0 to n-1 generate
     pgen1 : for k in 0 to n-1 generate
          pp(j)(k) <= a(k) and b(j);
     end generate;
  end generate;

  pc(0) <= (others => '0');
  ps(0) <= pp(0);
  prod(0) <= pp(0)(0);
  ...
```

Architecture 1 (cont.)

```
...
-- lines 1 to n-1
addr : for j in 1 to n-1 generate
   addc : for k in 0 to n-2 generate
      fa0 : fa port map
            (a => pp(j)(k), b => ps(j-1)(k+1),
             cin => pc(j-1)(k),
             s => ps(j)(k), cout => pc(j)(k));
   end generate; -- k
   prod(j) <= ps(j)(0);
   ps(j)(n-1) <= pp(j)(n-1);
end generate; -- j
...
```

Architecture 1 (cont.)

```
  ...
  -- last line (j = n)
  pc(n)(0) <= '0';
  addlast : for k in 0 to n-2 generate
      fa1 : fa port map
             (a => ps(n-1)(k), b => pc(n-1)(k-1),
              cin => pc(n)(k-1),
              s => ps(n)(k), cout => pc(n)(k));
  end generate; -- k
  prod(2*n-1) <= pc(n)(n-1);
  prod(2*n-2 downto n) <= ps(n)(n-1 downto 1);
end rtl1;
```

Multiplikation von Zweierkomplementzahlen

- Kann zurückgeführt werden auf die Multiplikation von unsigned Zahlen
  - `sign_a = a(n−1); a_abs <= abs(a);`
  - `sign_b = b(n−1); b_abs <= abs(b);`
  - `prod_abs = a_abs * b_abs;`

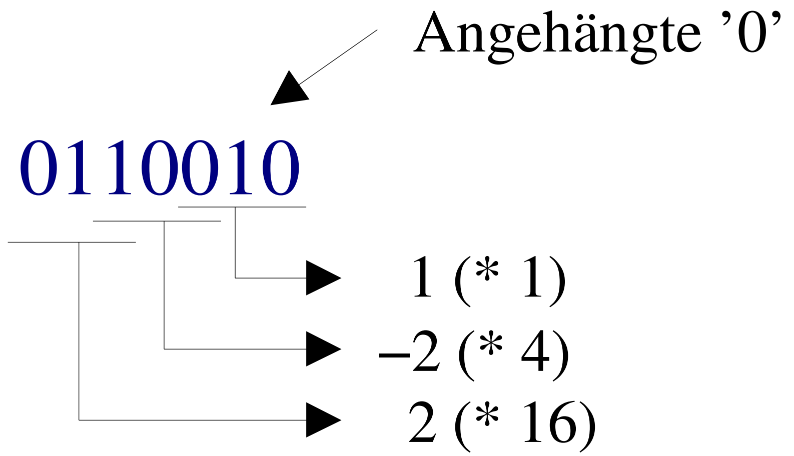
  -
    ```
    if sign_a xor sign_b = ’0’ then
        prod <= prod_abs;
    else
        prod <= − prod_abs;
    ```

Booth Encoding

- Reduktion der Partialprodukte auf die Hälfte
- Argumente können 2er–Komplementzahlen sein
- An zweiten Operand wird rechts eine Null angefügt
- Gruppieren von je 3 Bits, 1 Bit Überlappung

Beispiel

- 100101 x 011001 (–27 x 25)



recoding bits	partial product value
000	0
001	multiplicand
010	multiplicand
011	multiplicand * 2
100	multiplicand * –2
101	multiplicand * –1
110	multiplicand * –1
111	0

Beispiel (cont.)

- 100101 x 011001 (–27 x 25)

```
Vorzeichen– ──►  111111100101    (–27 * 1)
erweitert        0000110110      (–27 * –2 * 4)
                 11001010        (–27 * 2 * 16)
            ──────────────────
                 110101011101
```

Booth Encoder

```
library ieee;
use ieee.std_logic_1164.all;

entity encoder is
 generic(n : natural := 16);
 port(din  : in  std_logic_vector(n-1 downto 0);
      bit3 : in  std_logic_vector(2 downto 0);
      dout : out std_logic_vector(n downto 0));
end encoder;
```

Booth Encoder (cont.)

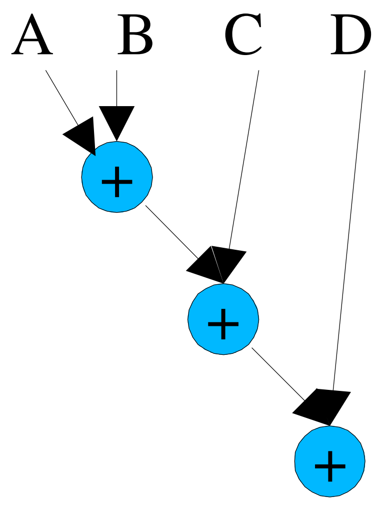
```
use ieee.std_logic_unsigned.all;

architecture rtl of encoder is
subtype bitn1 is std_logic_vector(n downto 0);

function comp2(d : in bitn1) return bitn1 is
   variable dn : bitn1;
begin
   dn := not d;
   return(dn + 1);
end comp2;

begin  ...
```

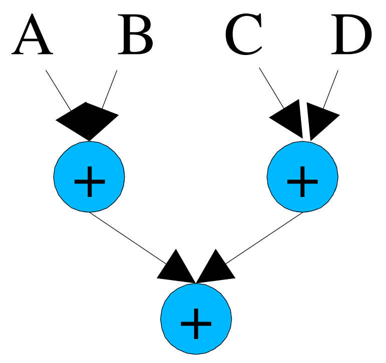
Booth Encoder (cont.)

```
...
begin
  process(din, bit3)
  begin
    case bit3 is
    when "001"|"010" => dout <= din(n-1) & din;
    when "101"|"110" => dout <= comp2(
                                din(n-1) & din);
    when "100" =>   dout <= comp2(din & '0');
    when "011" =>   dout <= din & '0';
    when others =>  dout <= (dout'range => '0');
end case; end process; end rtl;
```

Wallace Tree Adders

- $A + B + C + D = (A + B) + (C + D)$





Unit Adder

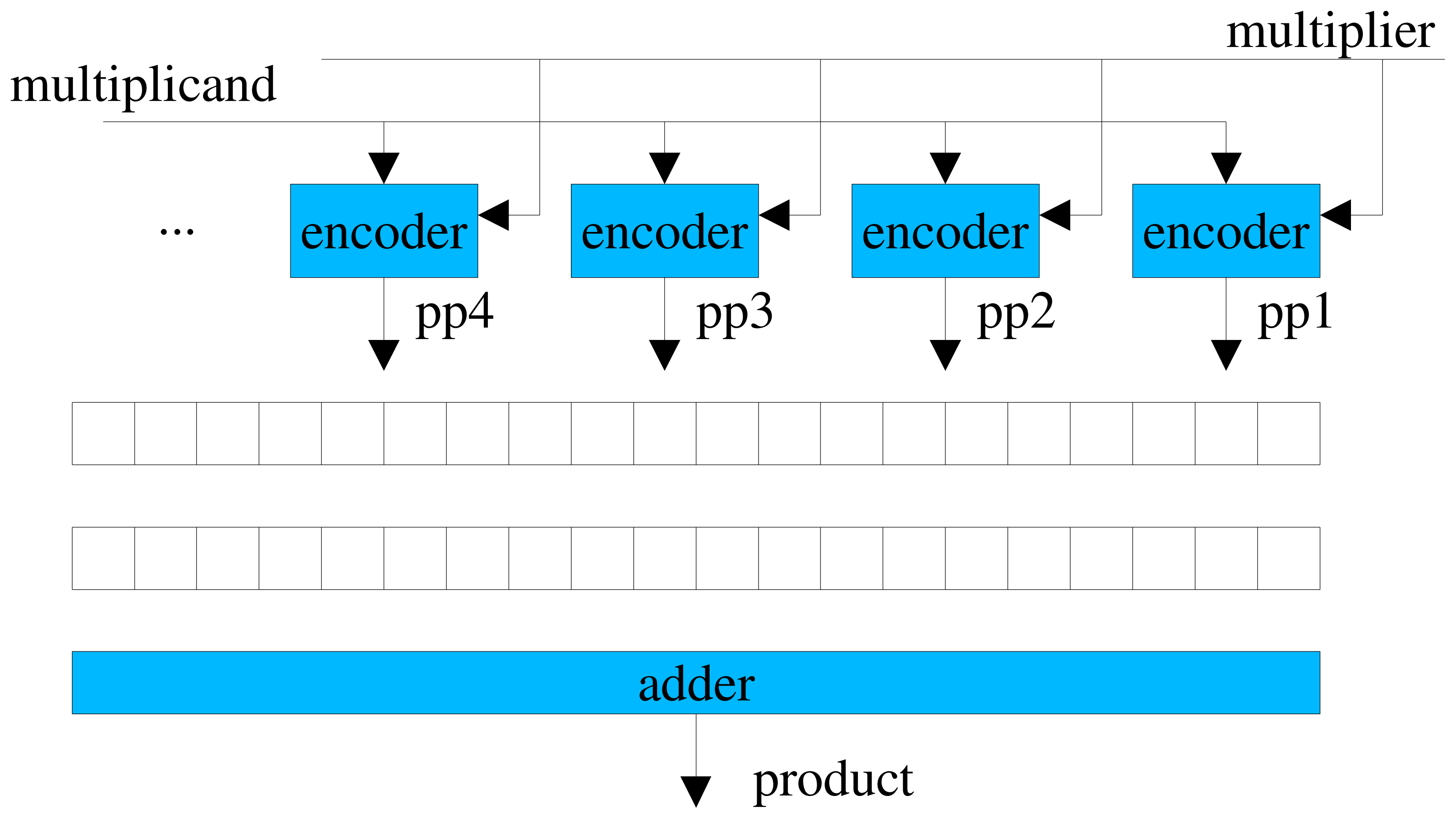
```
library ieee;
use ieee.std_logic_1164.all;

entity unitadd is
  port(x      : in  std_logic_vector(3 downto 0);
       cin    : in  std_logic;
       s      : out std_logic;
       c, co  : out std_logic);
end unitadd;
```

$$x(0) + \ldots + x(3) + cin = s + 2 \cdot c + 2 \cdot co$$

Unit Adder (cont.)

```
architecture rtl of unitadd is
begin
  process(x, cin)
     variable tmp1, tmp2 : std_logic;
  begin
   co   <= (x(0)  or x(1)) and (x(2)  or x(3));
   tmp1 := (x(0) xor x(1)) xor (x(2) xor x(3));
   s    <=  cin  xor tmp1;
   tmp2 := (x(0) and x(1)) nor (x(2) and x(3));
   c    <= (tmp1 and cin)  or  (tmp1 nor tmp2);
  end process;
end rtl;
```

Booth – Wallace Tree Multiplier



Booth – Wallace Tree Multiplier

```
architecture rtl of mult is
 constant w : integer := 2*n; -- prod.width
 constant d : integer := (n/2-4)/2+1; -- add.levels
 subtype prod_t is std_logic_vector(w-1 downto 0);
 type pp_t is array(n/2 downto 1) of prod_t;
 signal pp : pp_t; -- partial products
 signal prod1 : std_logic_vector(w downto 0);

 subtype col_t is std_logic_vector(w downto 0);
 type    int_t is array(d-1 downto 0) of col_t;
 signal  co_s, c_s, s_s : int_t; -- for co, c, s
 signal encode_bits : std_logic_vector(n downto 0);
...
```

Booth – Wallace Tree (2)

```
...
component encoder
  port(din   : in  std_logic_vector(n-1 downto 0);
       bit3 : in  std_logic_vector(2 downto 0);
       dout : out std_logic_vector(n downto 0));
end component;

component unitadd
  port(x    : in  std_logic_vector(3 downto 0);
     cin    : in  std_logic;
     s      : out std_logic;
     c, co : out std_logic);
end component;
```

Booth – Wallace Tree (3)

```
begin
 encode_bits <= b & '0'; -- encoding bits, add '0'

 -- initialize co_s, c_s, s_s signals column 0
 init : for i in 0 to d-1 generate
     co_s(i)(0) <= '0';
     c_s(i)(0)  <= '0';
     s_s(i)(0)  <= '0';
 end generate;

 ...
```

Booth – Wallace Tree (4)

```
-- generate booth encoders for partial products
ppgen : for k in 1 to n/2 generate
   encode0 : encoder
    port map(din => a,     -- shift 2 bits
             bit3 => encode_bits(k*2 downto k*2-2),
             dout => pp(k)(n+k*2-2 downto k*2-2));
   signext : for i in 2*n-1 downto n+k*2-1 generate
      pp(k)(i) <= pp(k)(n+k*2-2); -- sign ext. MSB
   end generate; -- i
   zeroext : for i in 0 to k*2-3 generate
      pp(k)(i) <= '0'; -- zero extend LSB
   end generate; -- i
end generate; -- k
```

Booth – Wallace Tree (5)

```
-- generate adders
colunns : for col in 1 to w generate -- LSB to MSB
 rows : for row in 0 to d-1 generate -- top->bottom
   oneunit : block -- one unit adder
      signal tempx : std_logic_vector(3 downto 0);
   begin
      -- setup inputs for the unit adder
      genx1 : if (row < d-1) generate
         tempx(3) <= pp(row*4 + 4)(col-1);
         tempx(2) <= pp(row*4 + 3)(col-1);
         tempx(1) <= pp(row*4 + 2)(col-1);
         tempx(0) <= pp(row*4 + 1)(col-1);
      end generate;
```

Booth – Wallace Tree (6)

```
      genx2 : if (row = d-1) generate
         tempx(3) <= s_s(row-2)(col);
         tempx(2) <= c_s(row-2)(col-1);
         tempx(1) <= s_s(row-1)(col);
         tempx(0) <= c_s(row-1)(col-1);
      end generate;
      unitadd1 : unitadd
           port map(x   => tempx,
                    cin => co_s(row)(col-1),
                    s   => s_s (row)(col),
                    c   => c_s (row)(col),
                    co  => co_s(row)(col));
end block; end generate; end generate;
```

Booth – Wallace Tree (7)

```
  add0 : block
  begin
     prod1 <=
          (s_s(d-1)(w  ) & s_s(d-1)(w   downto 1))
        + (c_s(d-1)(w-1) & c_s(d-1)(w-1 downto 0));
  end block; -- add0

  prod <= prod1(w-1 downto 0);
end rtl;
```

Barrel Shifter

```
library ieee;
use ieee.std_logic_1164.all;

entity barrel_shift is
  port(d : in  std_logic_vector(15 downto 0);
       s : in  std_logic_vector( 3 downto 0);
       y : out std_logic_vector(15 downto 0));
end barrel_shift;
```

Barrel Shifter: Verhalten

```
use ieee.std_logic_arith.all;

architecture behave of barrel_shift is
   signal sint : integer;
begin
  sint <= conv_integer(s);
  y <= d(15-sint downto 0)
     & (sint downto 1 => ’0’);
end behave;
```

Barrel Shifter: RTL

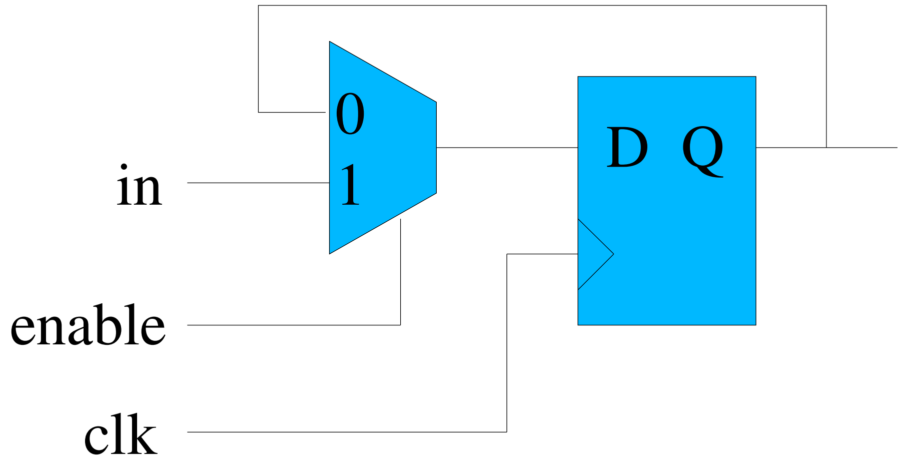
```
architecture rtl of barrel_shift is
   constant n : integer := 16;
   constant m : integer :=  4;
   type ary_t is array(m downto 0) of
                 std_logic_vector(n-1 downto 0);
   signal int_s, left, pass : ary_t;
   signal zero : std_logic_vector(n-1 downto 0);
begin
   zero <= (zero'range => '0');
   int_s(0) <= d;

   ...
```

Barrel Shifter: RTL (2)

```
...
  muxgen : for j in 1 to m generate
    pass(j) <= int_s(j-1);
    left(j) <= int_s(j-1)(n-2**(j-1)-1 downto 0)
             & zero(2**(j-1)-1 downto 0);
    int_s(j) <= pass(j) when s(j-1) = '0' else
                left(j);
  end generate;

  y <= int_s(m);
end rtl;
```

Clock Gating

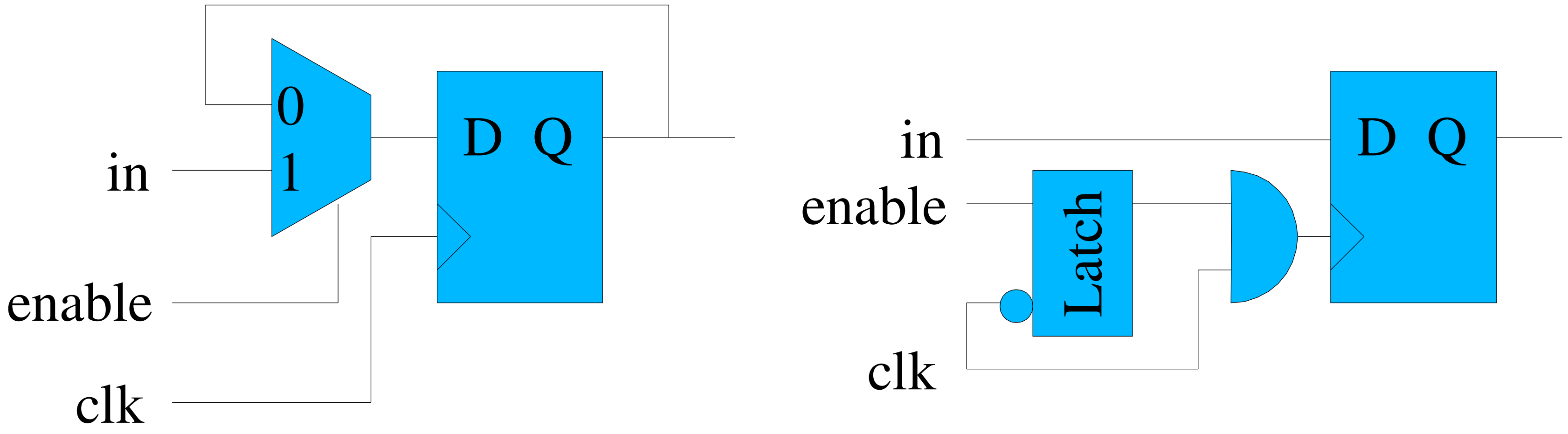
- Problem:



- Flipflop muss bei jedem Takt schalten
  - hoher Stromverbrauch

Clock Gating (2)

- Ersetzen durch (fast) äquivalente Schaltung:

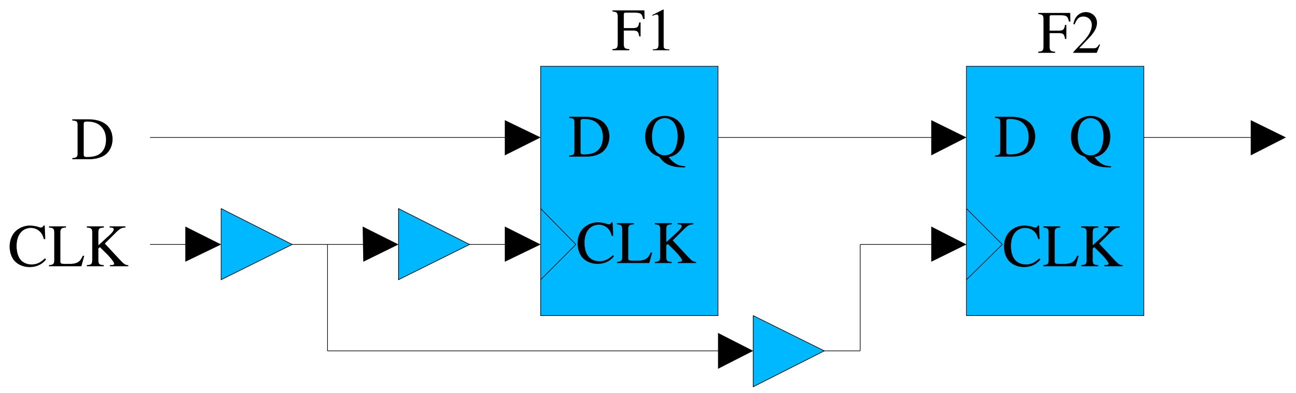


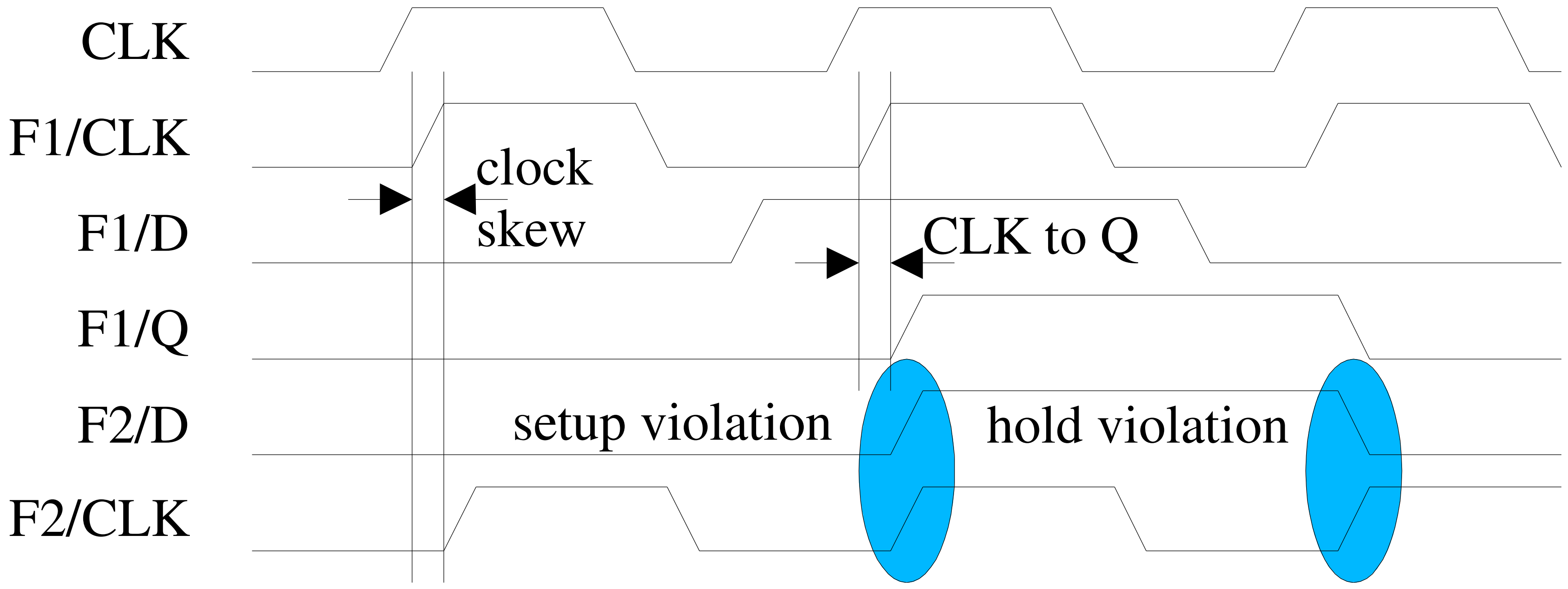
- Flipflop schaltet nur, wenn enable=’1’ und clk’event und clk=’1’

Clock Gating (3)

- Wann sollte "Clock Gating" verwendet werden?
  - nur wenn es unbedingt notwendig ist
- Wo kann "Clock Gating" verwendet werden?
  - In der RTL–Beschreibung
  - Automatische Erzeugung während der Logiksynthese
  - Mischformen sind möglich

Clock Skews

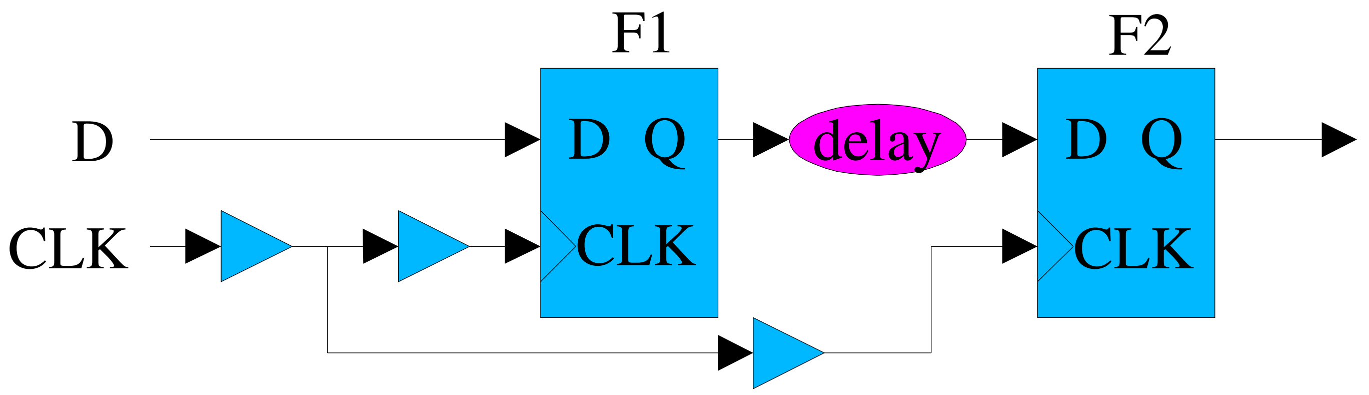

F1
F2
D
CLK
D Q
CLK
D Q
CLK

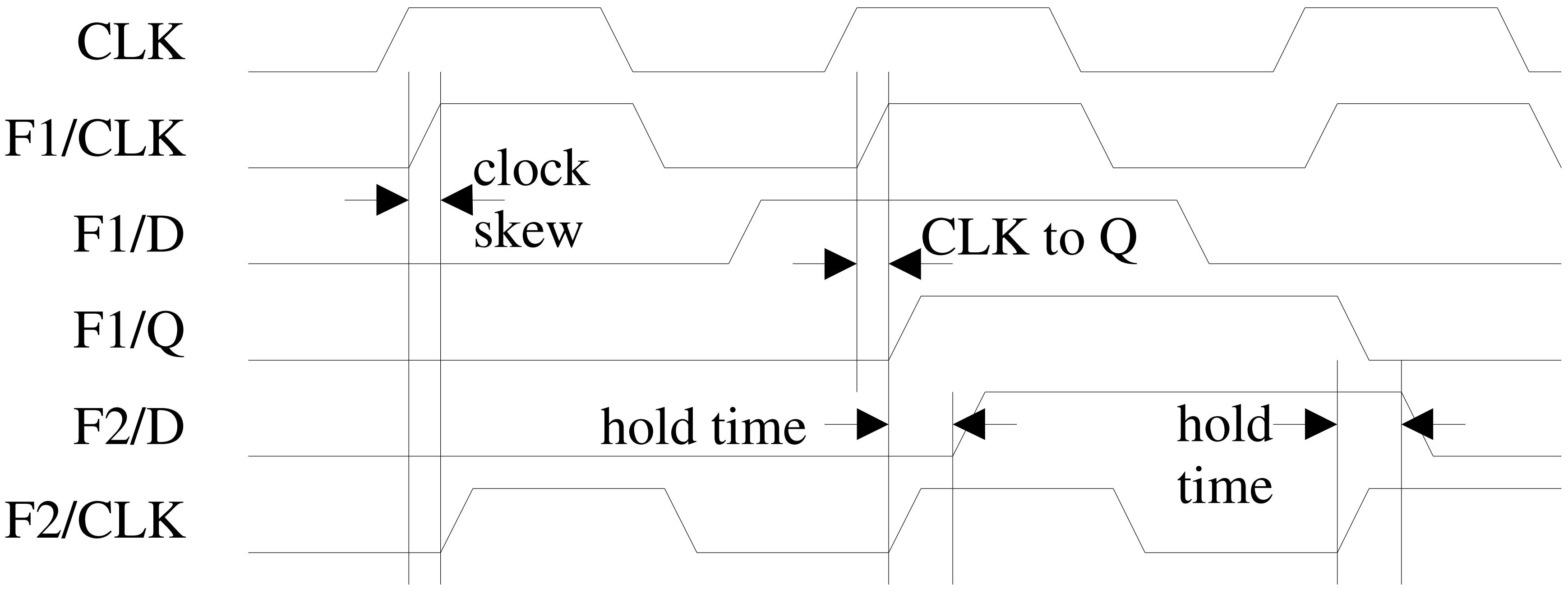

CLK
F1/CLK
F1/D
F1/Q
F2/D
F2/CLK
clock skew
CLK to Q
setup violation
hold violation

Clock Skews (2)

- Clock–Signale sind verzögert
  - durch hohe Last
  - durch Treiber (*Clock Tree*)
- Dadurch schalten Flipflops zu spät
  - Setup– und Hold–Zeiten werden nicht eingehalten
  - Wert am Ausgang ist unvorhersehbar oder falsch

Clock Skews: Lösung


F1
F2
D
CLK
D Q
CLK
delay
D Q
CLK


CLK
F1/CLK
F1/D
F1/Q
F2/D
F2/CLK
clock skew
CLK to Q
hold time
hold time