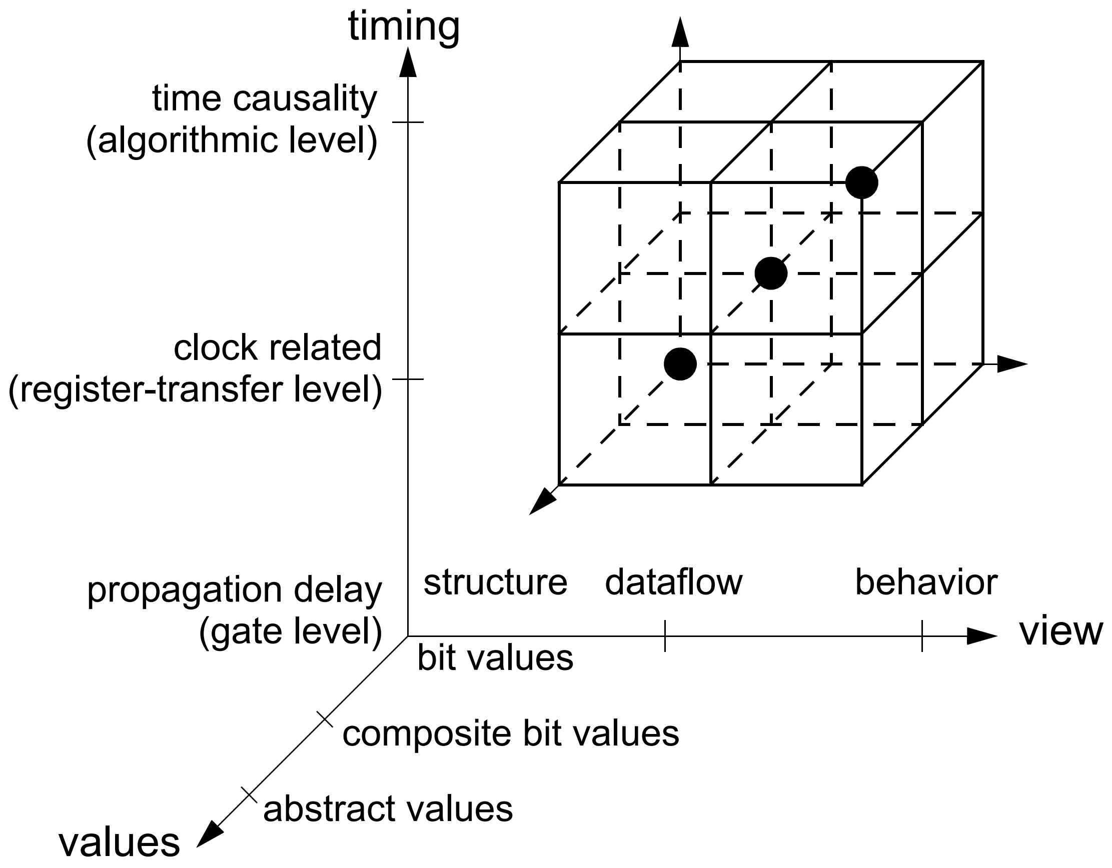
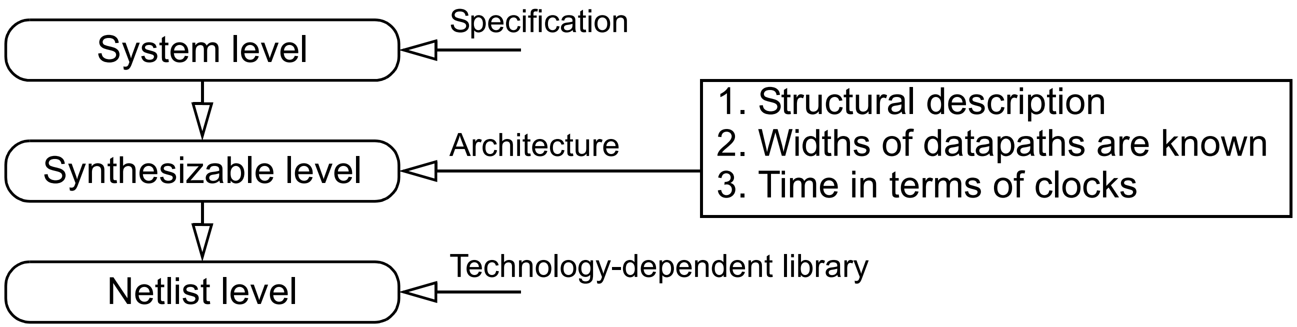
HDL Design Cube



Synthesis and VHDL

- **VHDL LRM defines only simulation semantics of the language.**
- **LRM – Language Reference Manual**



- **Synthesis restrictions:**
  - **the lack of maturity of synthesis tools**
  - **the state-of-art in synthesis targets RTL synthesis only**
  - **certain VHDL features are simply not synthesizable**

Synthesis style

- Delay expressions (*after* clauses, *wait for* statements are ignored)
- Certain restrictions on the writing of process statement occur
- Only a few types are allowed
  - integer, enumerated, e.g., bit, bit_vector, signed
- Type conversion and resolution functions are not interpreted
- Description is oriented towards synchronous styles with explicit clocks
- Types: enumeration, integer, one-dimensional array, record

```
type WORD is array (31 downto 0)   of BIT;
type RAM  is array (1023 downto 0) of WORD;
```

- In record, an item address becomes hardware coded
- !!! Time type is not supported !!!
- No explicit or default initialization
- Parenthesis in expressions have effect on HW generation
- Some arithmetic operations are supported partially only

Sensitivity list

- **Equivalent processes:**

```
process (A, B, C)
   ...
begin
   ...
end process;
```

==

```
process
   ...
begin
wait on A, B, C;
   ...
end process;
```

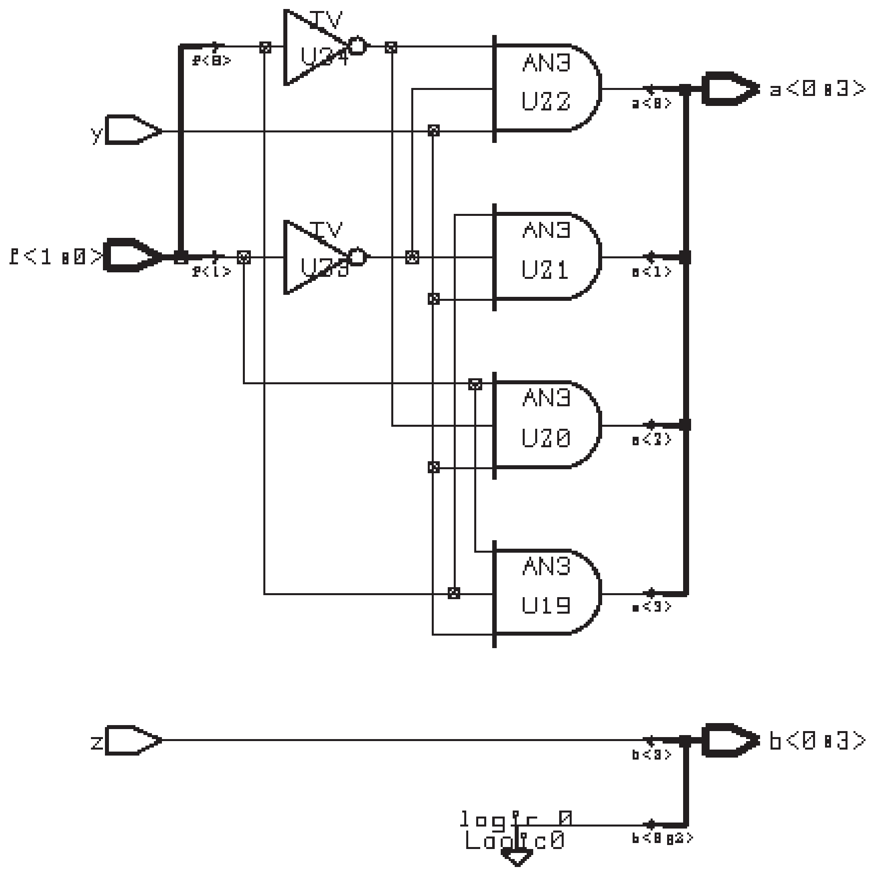
~~

```
process
   ...
begin
   ...
wait on A, B, C;
end process;
```

- **Some synthesizers support only sensitivity list for combinational logic**
- **In case of single synchronization process there is no need to "remember" at which synchronization point it was stopped –> such behavior does not imply memorization**
- **Process with multiple synchronization points, i.e. several wait states, infer memorization – FSM**

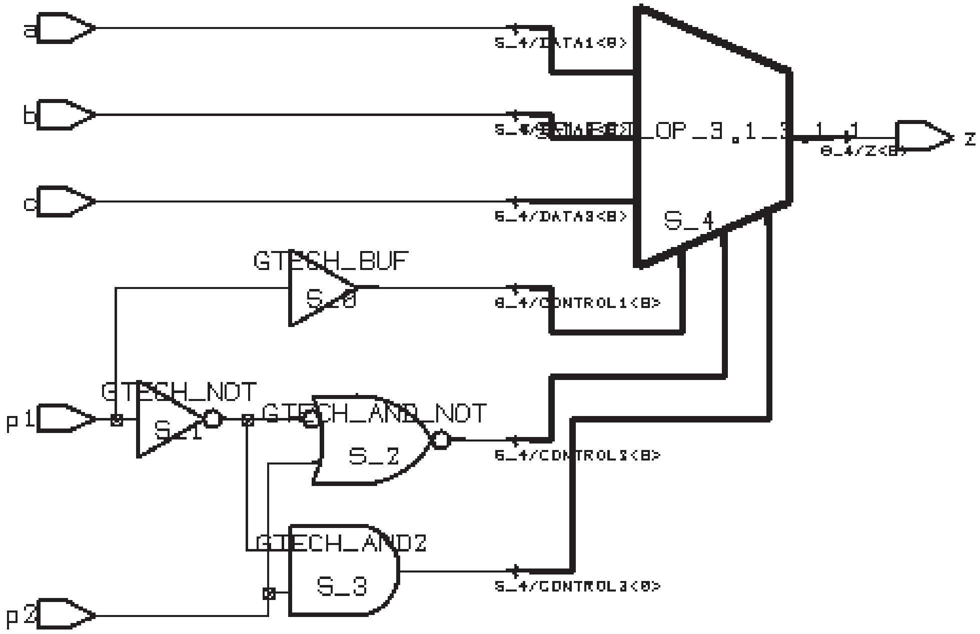
Assignment statement synthesis

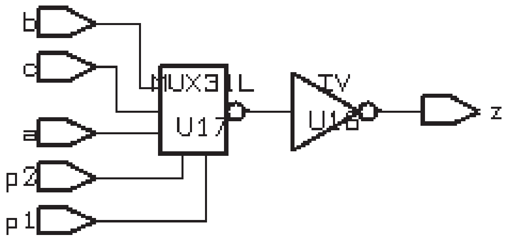
```
  signal A,B: BIT_VECTOR(0 to 3);
  signal I: INTEGER range 0 to 3;
  signal Y,Z: BIT;
  -- . . .
process ( I, Y, Z ) begin
  A<="0000";
  B<="0000";
  A(I)<=Y;   -- Computable index
  B(3)<=Z;   -- Constant index
end process;
```



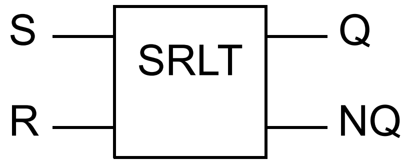
if statement synthesis

```
   signal A,B,C,P1,P2,Z: BIT;
   -- . . .
process (P1,P2,A,B,C) begin
   if (P1 = '1') then
       Z <= A;
   elsif (P2 = '0') then
       Z <= B;
   else
       Z <= C;
   end if;
end process;
```

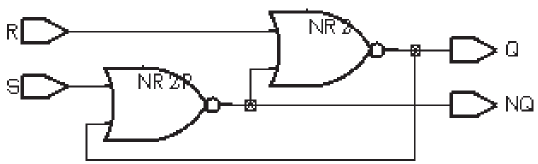




SR latch




```
use WORK.CHECK_PKG.all;
entity SRLT is
  port ( S, R:  in bit;
         Q, NQ: out bit );
begin
  NOT_AT_THE_SAME_TIME(S,R);
end SRLT;
```




```
architecture A1 of SRLT is
  signal LQ:  bit := '1';
  signal LNQ:  bit := '0';
begin
  LNQ <= S nor LQ;
  LQ  <= R nor LNQ;
  Q   <= LQ;
  NQ  <= LNQ;
end A1;
```

- **NB! Asynchronous feed-back is temporarily cut by synthesizers...**

Combinational circuit

- **A process is combinational, i.e. does not infer memorization, if:**
  - **the process has an explicit sensitivity list or contains a single synchronization point (waiting for changes on all input values); [1)]**
  - **no local variable declarations, or variables are assigned before being read;**
  - **all signals, which values are read, are part of the sensitivity list; [2)] and**
  - **all output signals are targets of signal assignments independent on the branch of the process, i.e. all signal assignments are covered by all conditional combinations.**

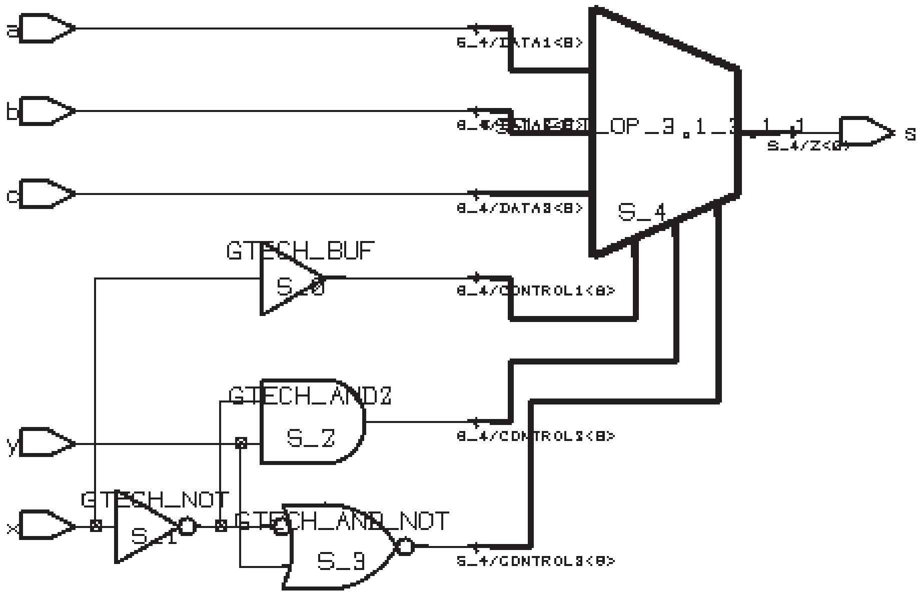
[1)] waiting on a clock signal, e.g., “ wait on clk until clk=’1’; ”, implies buffered outputs (FF-s)

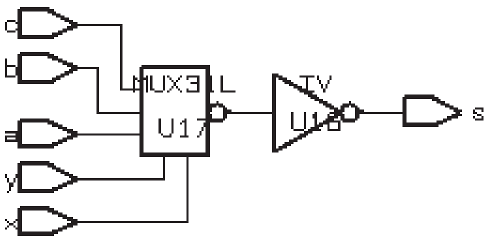
[2)] interpretation may differ from tool to tool

Complex assignments

- No memory

```
S <= A when X='1' else B when Y='1' else C;

process (A, B, C, X, Y) begin
  if    X='1' then   S <= A;
  elsif Y='1' then   S <= B;
  else               S <= C;
  end if;
end process;
```

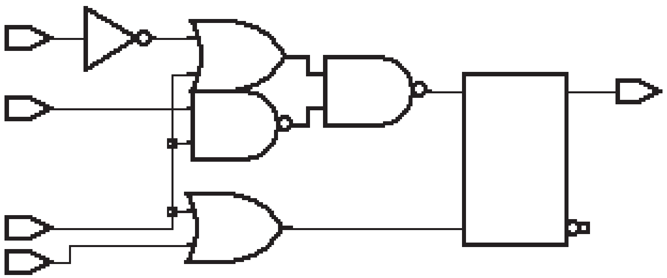


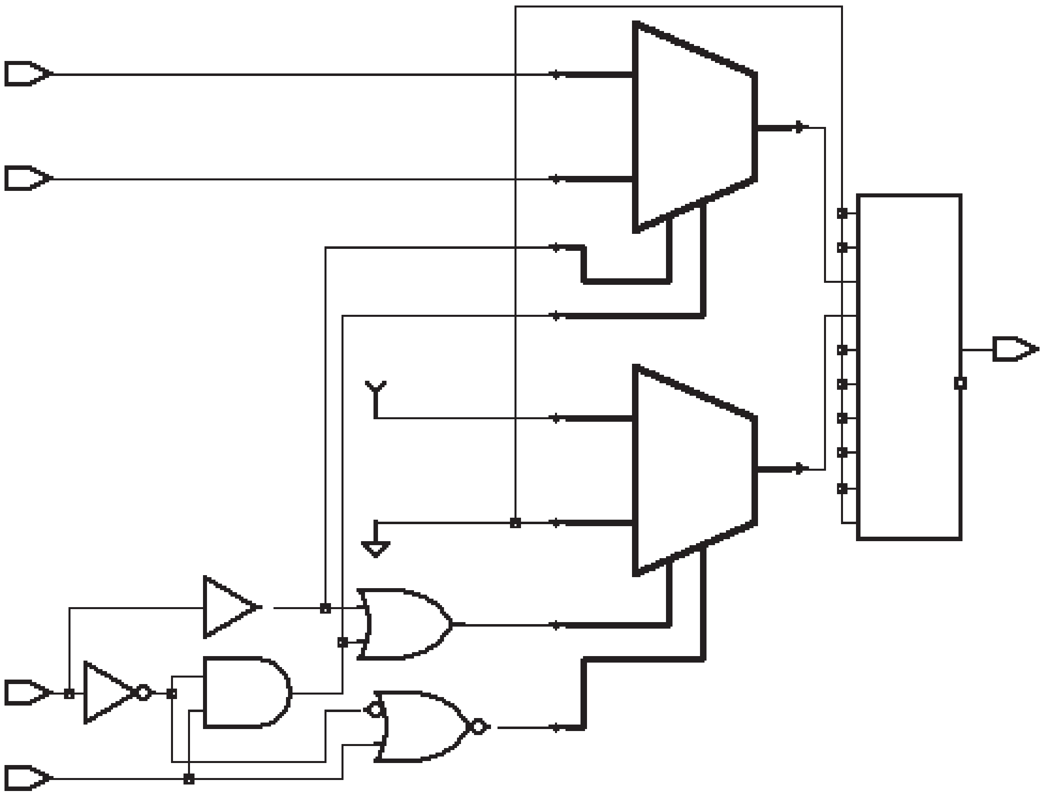


Complex assignments (#2)

- Memory element generated!

```
process (A, B, X, Y) begin
  if    X='1' then  S <= A;
  elsif Y='1' then  S <= B;
  end if;
end process;
```

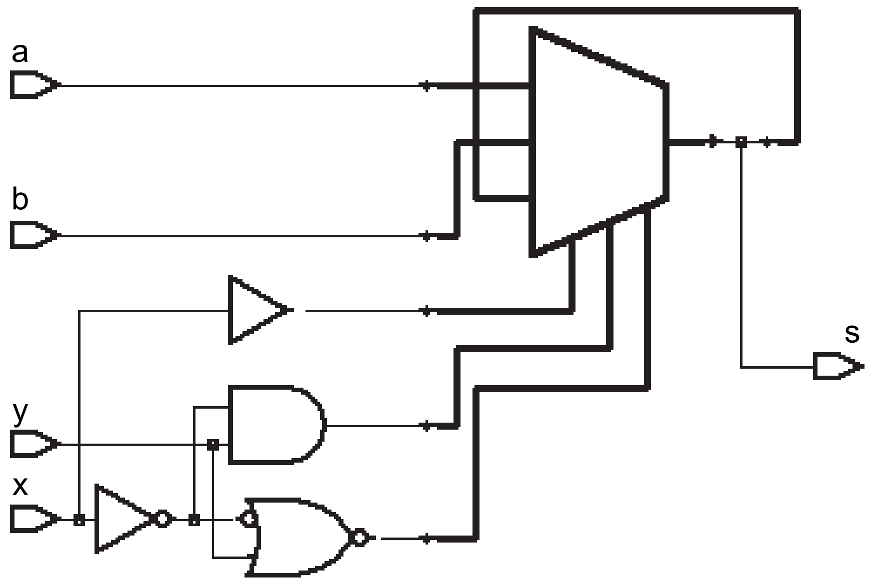


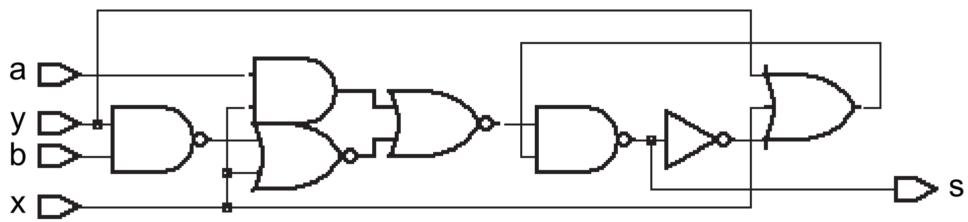


Complex assignments (#3)

- **Memory element generated!**

```
S <= A when X='1' else B when Y='1' else S;

process (A, B, X, Y) begin
  if    X='1' then   S <= A;
  elsif Y='1' then   S <= B;
  else               S <= S;
  end if;
end process;
```



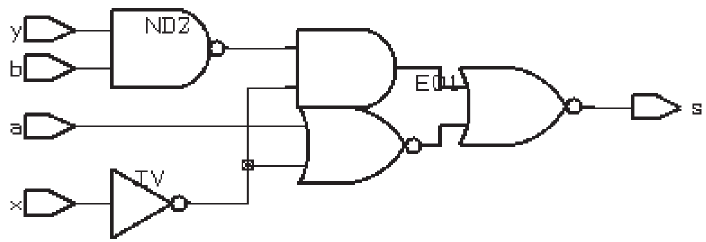


Default values

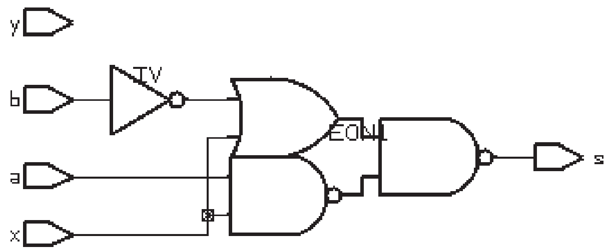
- The default values inherited from type or subtype definitions
- The explicit initialization that is given when the object is declared
- A value assigned using a statement at the beginning of a process
- Only the last case is supported by synthesis tools!
- Usually, a part of the synthesizable code is devoted to *set/reset* constructions

- Default values can be used to guarantee that the signal always gets a new value

```
process (A, B, X, Y) begin
  S <= '0';
  if    X='1' then   S <= A;
  elsif Y='1' then   S <= B;
  end if;
end process;
```



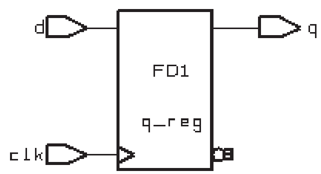

```
process (A, B, X, Y) begin
  S <= '-';
  if    X='1' then   S <= A;
  elsif Y='1' then   S <= B;
  end if;
end process;
```



Flip-flops

- **Process with the clock signal in the sensitivity list and explicit clock flank definition**

```
process (CLK) begin -- bit & std_logic
  if CLK='1' and CLK'event then    Q <= D;    end if;
end process;
                          -- std_logic only!
  if rising_edge(CLK) then    Q <= D;    end if;
```



- **Process with the clock signal and clock flank definition in the wait statement**

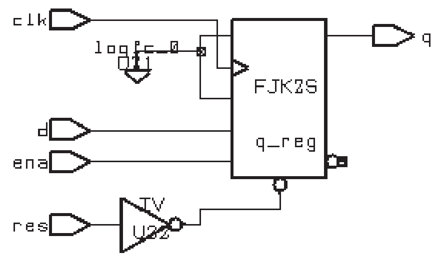
```
process begin
  wait on CLK until CLK='1';    Q <= D;
end process;
```

- **Concurrent assignment with the clock signal and clock flank definition**

```
Q <= D when CLK='1' and CLK'event;
```

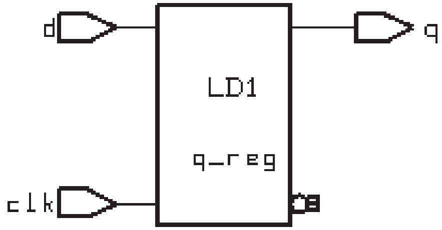
- **Asynchronous reset & synchronous enable**

```
process (RES,CLK) begin
  if    RES='1' then    Q <= '0';     -- asynchronous reset
  elsif CLK='1' and CLK'EVENT then
    if  ENA='1' then    Q <= D;    end if;
  end if;
end process P4_FF;
```



Latch vs. Flip-flop?

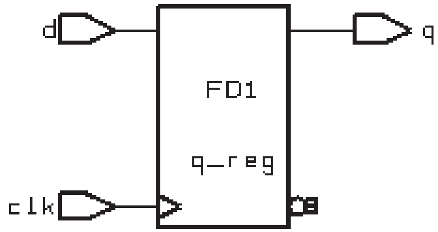
```
P1_L: process (CLK, D) begin
    if CLK='1' then    Q <= D;
    end if;
end process P1_L;

P2_FL: process (CLK) begin
    if CLK='1' then    Q<=D;
    end if;
end process P2_FL;

P1_FF: process (CLK) begin
    if CLK='1' and
       CLK'event then  Q<=D;
    end if;
end process P1_FF;
```



- **Simulation OK but not synthesis!**

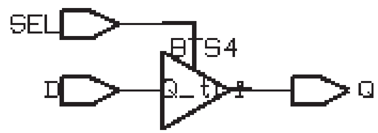
```
-- Warning: Variable 'd' is being read
-- in routine .. line .. in file '..',
-- but is not in the process sensitivity
-- list of the block which begins
-- there.    (HDL-179)
```

- **Result - latch**



Synthesis rules

- Guidelines in priority order:
  - the target signal(s) will be synthesized as flip-flops when there is a signal edge expression, e.g. CLK'event and CLK='1', in the process
  - usually, only one edge expression is allowed per process
    - different processes can have different clocks (tool depending)
  - the target signal will infer three-state buffer(s) when it can be assigned a value 'Z'
    - example: Q <= D when SEL='1' else 'Z';

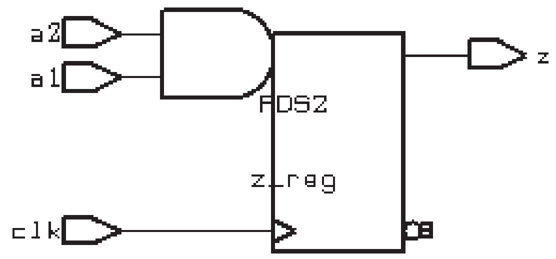


  - the target signal will infer a latch (latches) when the target signal is not assigned with a value in every conditional branch, and the edge expression is missing
  - a combinational circuit will be synthesized otherwise
- It is a good practice to isolate flip-flops, latches and three-state buffers inferences to ensure design correctness

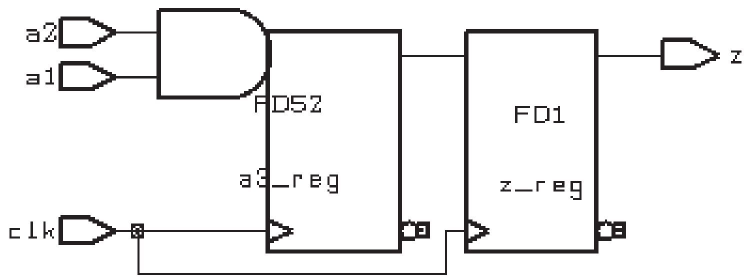
Signal versus variable

- The hardware resulting from synthesis of variables or signals differs: either nothing, wires, or memory elements

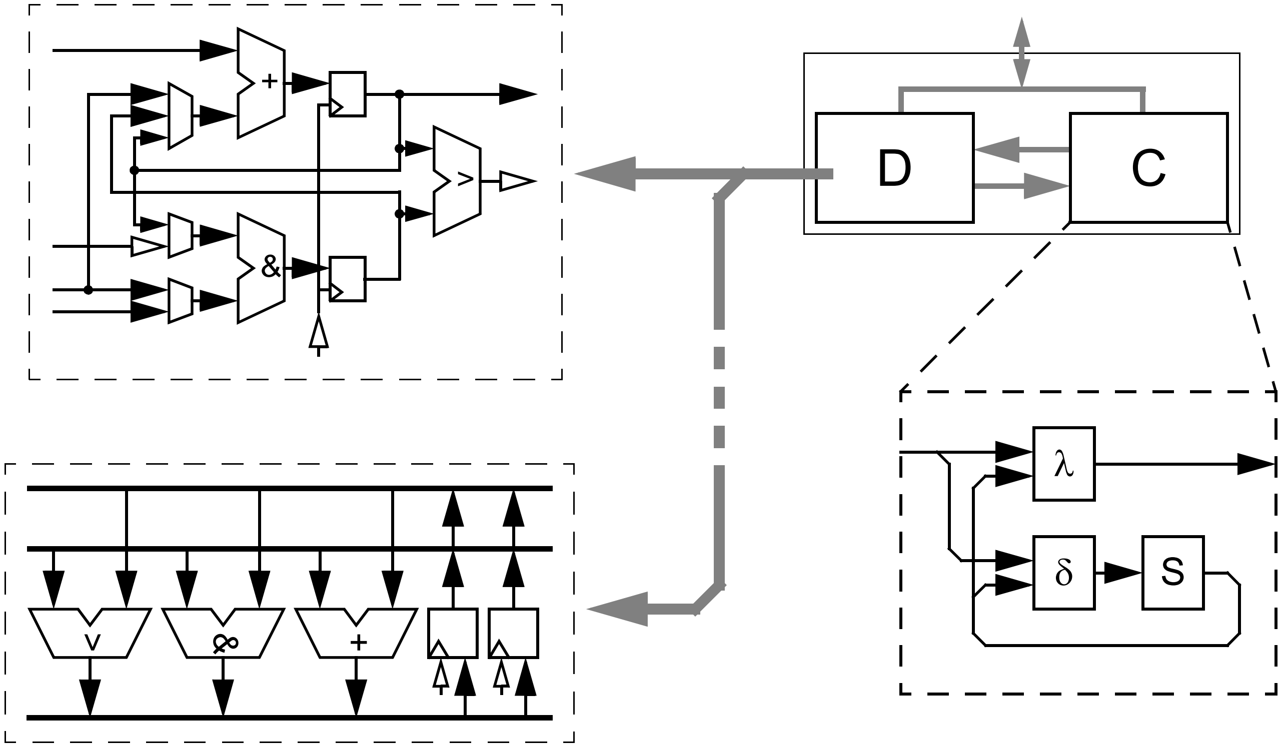
```
signal A1, A2: BIT;
-- . . .
process (CLOCK)
  variable A3: BIT;
begin
  if CLOCK='1' and CLOCK'event then
    A3 := A1 and A2;
    Z  <= A3;
  end if;
end process;
```




```
signal A1, A2, A3: BIT;
-- . . .
process (CLOCK)
begin
  if CLOCK='1' and CLOCK'event then
    A3 <= A1 and A2;     -- Next delta-cycle!!
    Z  <= A3;
    end if;
end process;
```



Data-part & control-part

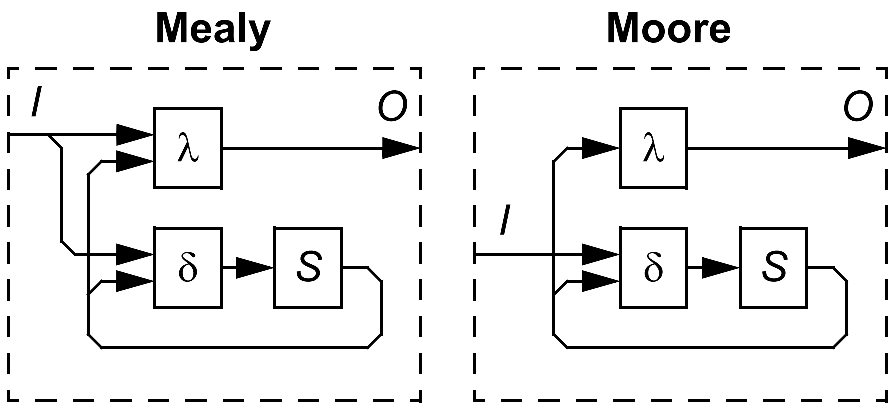


- one unit – one process
  - functional units – combinational processes [all inputs in the sensitivity list]
  - storage units – clocked processes [activation at clock edge]

TTÜ1918

FSM in VHDL

- **FSM: M = (S, I, O, δ, λ)**
  - **S: set of states**
  - **I: set on inputs**
  - **O: set of outputs**
  - **δ: transition function - δ: S × I → S**
  - **λ: output function - λ: S × I → O**



- **How many processes?**
  - **Process per block**
- **Three processes**
  - **(1) transition function, (2) output function, (3) state register**
- **Two processes**
  - **(1) merged transition and output functions, (2) state register** [*Mealy*]
- **One process**
  - **buffered outputs!** [*Moore*]

FSM as a single process

- Note that <u>all</u> signals assigned in the process will have flip-flops!

```
-- RESET is the asynchronous reset, CLK is the clock
-- STATE is a variable (or signal) memorizing the current state
process(RESET,CLK)
begin
    if RESET='1' then          -- asynchronous reset
        STATE <= S_INIT;
    elsif CLK='1' and CLK'EVENT then
        case STATE is
            when S_INIT => if I0='1' then STATE <= S5; end if;
            when ... => ...
        end case;
    end if;
end process;
```

Three process FSM

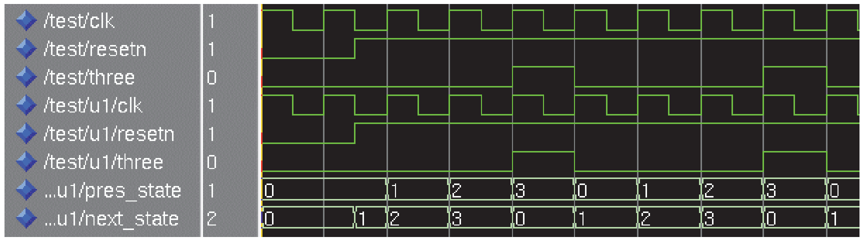
- **storage elements, transition function & output function**

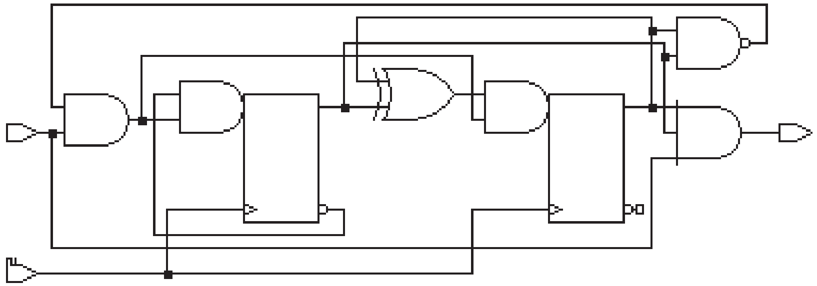
```
architecture B of FSM is
    type TYPE_STATE is (S_INIT,S1,...Sn);
    signal CURRENT_STATE, NEXT_STATE : TYPE_STATE;
begin
  P_STATE: process begin -- sequential process / storage elements
    wait until CLK'EVENT and CLK='1';
    if RESET ='1' then CURRENT_STATE <= S_INIT;
    else               CURRENT_STATE <= NEXT_STATE; end if;
  end process P_STATE;
  P_NEXT_STATE: process (I0, ..., CURRENT_STATE) begin -- next state function
    NEXT_STATE <= CURRENT_STATE;
    case CURRENT_STATE is
      when S_INIT => if I0='1' then NEXT_STATE <= S5; end if;
      when ... => ...
    end case;
  end process P_NEXT_STATE;
  P_OUTPUTS: process (CURRENT_STATE) begin -- output function
    case CURRENT_STATE is
      when S_INIT => O <= '0';
      when ... => ...
    end case;
  end process P_OUTPUTS;
end B;
```

FSM #2 – description styles & synthesis

Two processes (modulo-4 counter)

```
library IEEE; use IEEE.std_logic_1164.all;
entity counter03 is
  port ( clk: in bit;
         resetn: in std_logic;
         three: out std_logic );
end entity counter03;
architecture fsm2 of counter03 is
  subtype state_type is integer range 0 to 3;
  signal pres_state, next_state: state_type := 0;
begin
  process (clk) begin -- State memory
    if clk'event and clk = '1' then
      pres_state <= next_state;
    end if;
  end process;
-- Next state & output functions
  process (resetn, pres_state) begin
    three <= '0';
    if resetn='0' then    next_state <= 0;
    else
      case pres_state is
      when 0 to 2 => next_state <= pres_state + 1;
      when 3 => next_state <= 0;  three <= '1';
      end case;
    end if;
  end process;
end architecture fsm2;
```



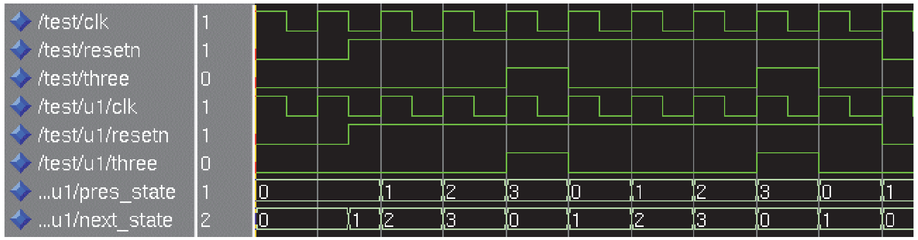


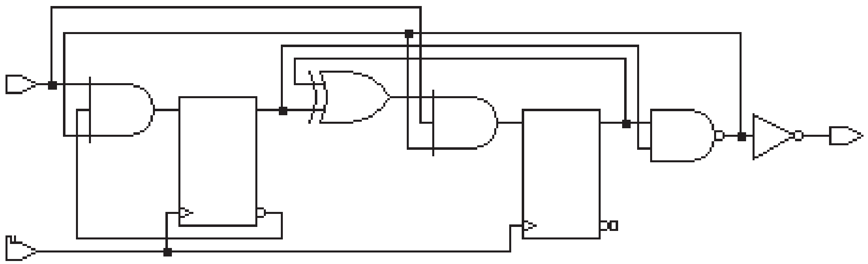
22 gates / 3.70 ns

FSM #2 – description styles & synthesis

Three processes (modulo-4 counter)

```
library IEEE; use IEEE.std_logic_1164.all;
architecture fsm3 of counter03 is
  subtype state_type is integer range 0 to 3;
  signal pres_state, next_state: state_type := 0;
begin
  process (clk) begin -- State memory
    if clk'event and clk = '1' then
      pres_state <= next_state;
    end if;
  end process;

  -- Next state function
  process (resetn, pres_state) begin
    if resetn='0' then    next_state <= 0;
    else
      if pres_state=3 then    next_state <= 0;
      else    next_state <= pres_state + 1;
      end if;
    end if;
  end process;
  -- Output function
  process (resetn, pres_state) begin
    if pres_state=3 then    three <= '1';
    else                    three <= '0';
    end if;
  end process;
end architecture fsm3;
```



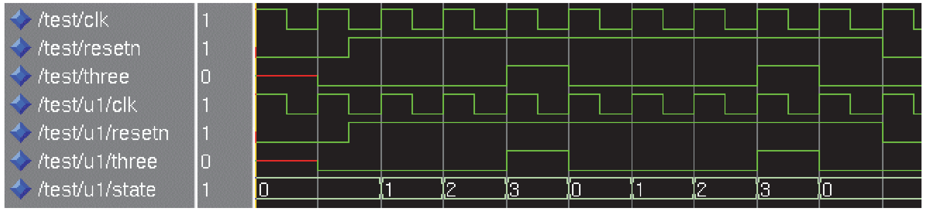


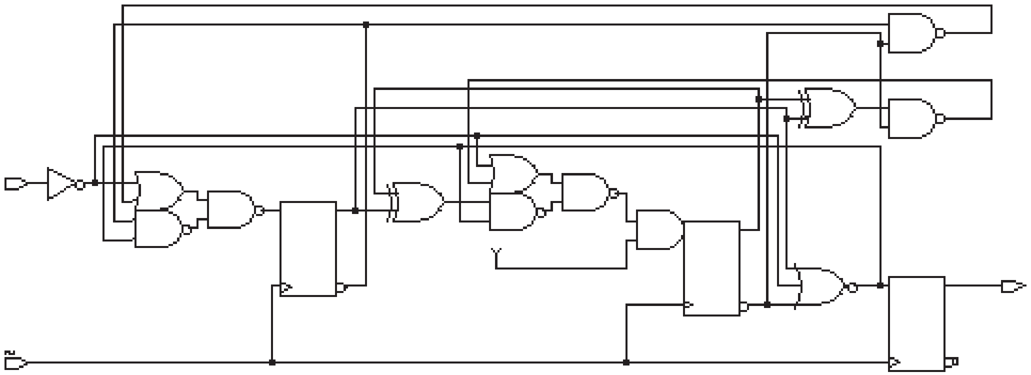
23 gates / 4.36 ns

FSM #2 – description styles & synthesis

One process (modulo-4 counter)

```
library IEEE; use IEEE.std_logic_1164.all;
architecture fsm1 of counter03 is
  subtype state_type is integer range 0 to 3;
  signal state: state_type := 0;
begin
  process (clk) begin
    if clk'event and clk = '1' then
      three <= '0';
      if resetn='0' then    state <= 0;
      else
        case state is
        when 0 | 1 => state <= state + 1;
        when 2 => state <= state + 1;  three <= '1';
        when 3 => state <= 0;
        end case;
      end if;
    end if;
  end process;
end architecture fsm1;

// Another version to build the process
  process begin
    wait on clk until clk='1';
    three <= '0';
    if resetn='0' then    state <= 0;
    else
    -- and so on...
```

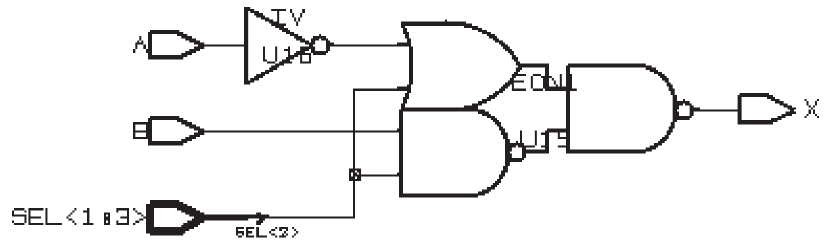




38 gates / 5.68 ns

Don’t care values and synthesis

```
process (A, B, SEL) begin
  case SEL is
  when ”001” =>    X <= A;
  when ”010” =>    X <= B;
  when others =>   X <= ’-’;
  end case;
end process;
```



Using generics

```
entity AND_N is
  generic (N: POSITIVE);    port (DIN: in BIT_VECTOR(1 to N); R: out BIT);
end AND_N;
architecture A1 of AND_N is
  signal INTER: BIT_VECTOR(1 to N);
begin
  INTER(1) <= DIN(1);
  L: for I in 1 to N-1 generate  INTER(I+1) <= DIN(I+1) and INTER(I);
  end generate;
  R <= INTER(N);
end A1;

C1: AND_N generic map (N=>12) port map(IN_DATA, OUT_DATA);
```

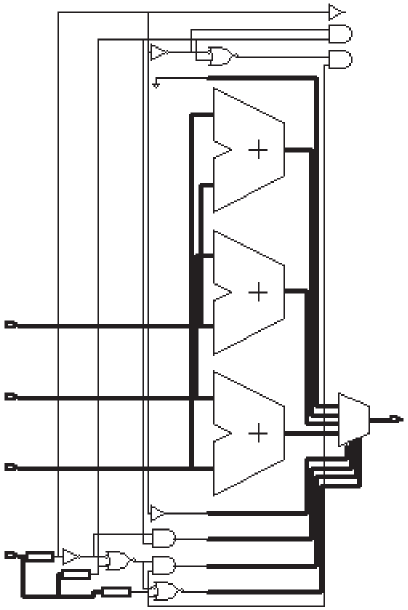
Arithmetics

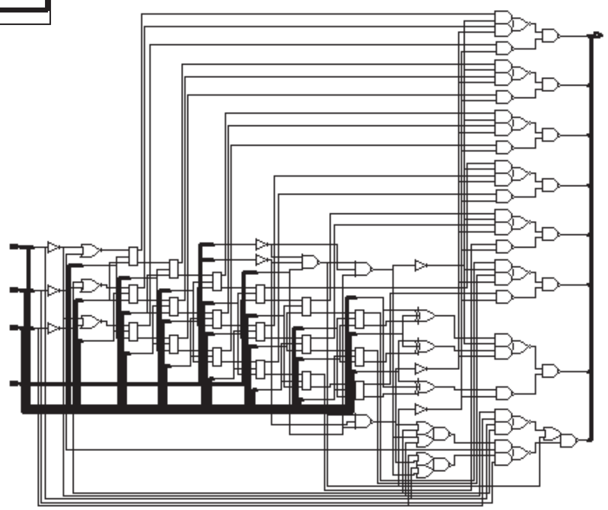
- Overloaded arithmetic operations:
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```
- Sources of the named packages are in the directory: $SYNOPSYS/packages/IEEE/src
  - $SYNOPSYS/packages is the root directory for all Synopsys packages
- Be careful with ‘*’, ‘/’, ‘**’ - extremely chip area consuming
  - Safe in some special cases - multiplication by power of two
- Use parenthesis to group a set of gates
- What you write is what you’ll get!

TTÜ1918

Behavioral RTL vs. “pure” RTL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity  test  is
  port ( a, b, c: in unsigned(7 downto 0);
         x: in unsigned(2 downto 0);
         o: out unsigned(7 downto 0) );
end test;
architecture bhv of test is begin
process (a, b, c, x)
  constant x2: unsigned(2 downto 0):="010";
  constant x3: unsigned(2 downto 0):="011";
  constant x6: unsigned(2 downto 0):="110";
begin
  if    x=x2 then  o <= a+b;
  elsif x=x3 then  o <= a+c;
  elsif x=x6 then  o <= b+c;
  else             o <= (others=>'0');
  end if;
end process;
end architecture bhv;
```

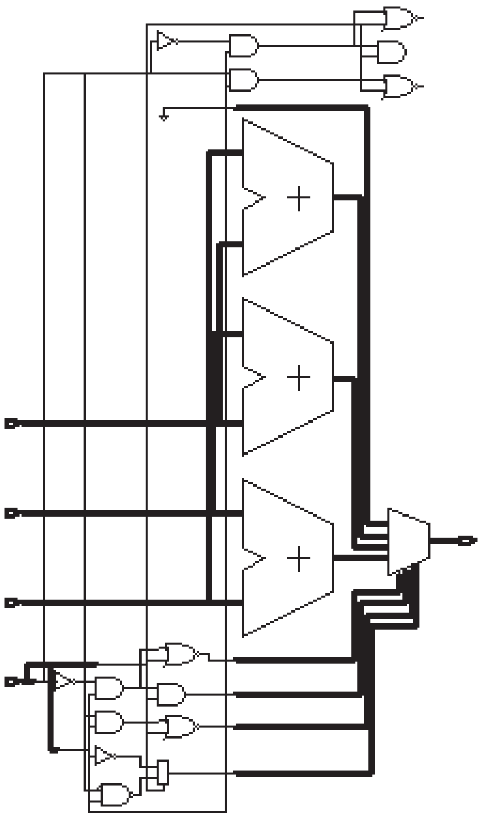




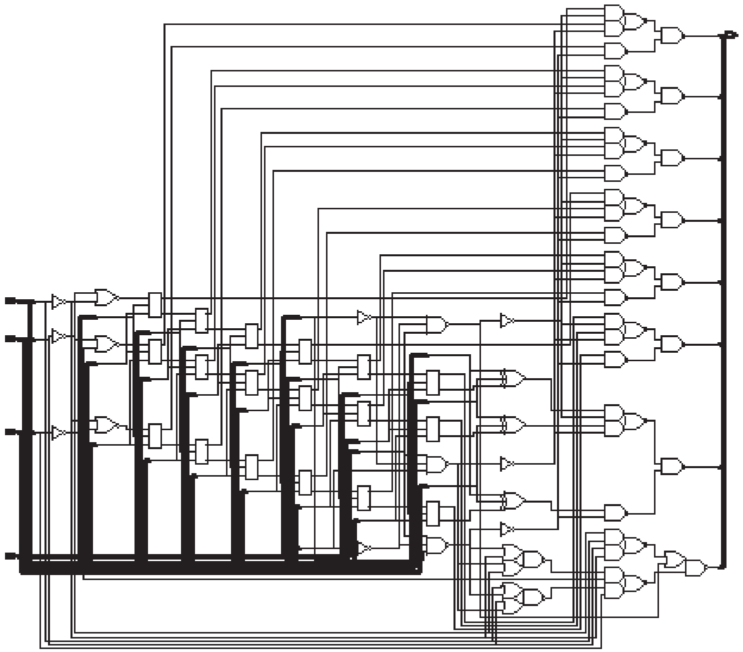
220 gates / 11.57 ns

Behavioral RTL vs. “pure” RTL

```
architecture bhv2 of test is begin
process (a, b, c, x) begin
  case x is
  when ″010″ =>   o <= a+b;
  when ″011″ =>   o <= a+c;
  when ″110″ =>   o <= b+c;
  when others =>  o <= (others=>'0');
  end case;
end process;
end architecture bhv2;
```

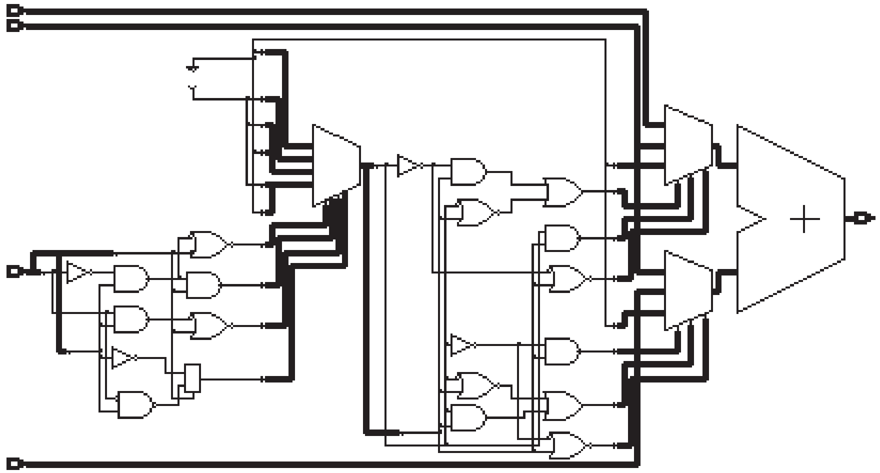


220 gates / 11.57 ns

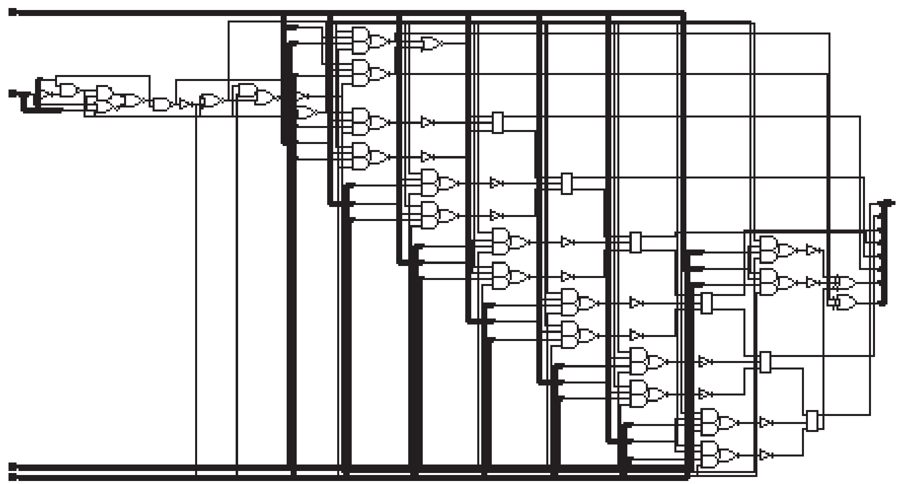


Behavioral RTL vs. “pure” RTL

```
architecture rtl of test is
  signal a1, a2: unsigned(7 downto 0);
  signal dc: unsigned(1 downto 0);
begin
dec: process (x) begin
  case x is
  when ”010” =>   dc <= ”01”;
  when ”011” =>   dc <= ”10”;
  when ”110” =>   dc <= ”11”;
  when others =>  dc <= ”00”;
  end case;
end process dec;
m1: process (a, b, dc) begin
  case dc is
  when ”01” =>    a1 <= a;
  when ”10” =>    a1 <= a;
  when ”11” =>    a1 <= b;
  when others =>  a1 <= (others=>'0');
  end case;
end process m1;
m2: process (b, c, dc) begin
  -- ...
end process m2;
o <= a1 + a2;
end architecture rtl;
```



117 gates / 19.2 ns



Adder / Subtracter

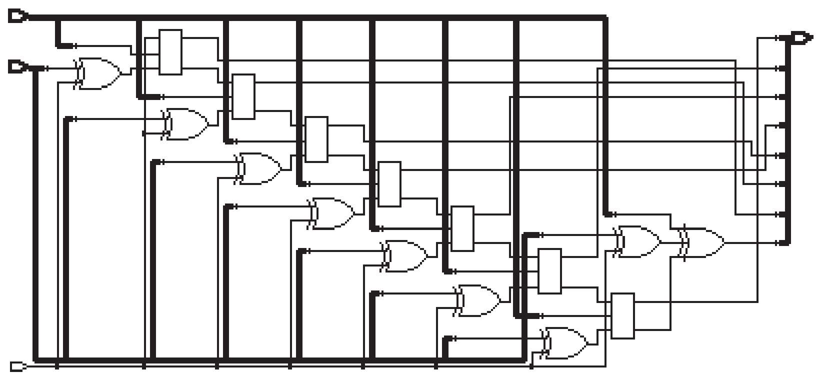
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity  add_sub  is
  port ( a, b: in unsigned(7 downto 0);
         x: in std_logic;
         o: out unsigned(7 downto 0) );
end add_sub;
architecture bhv of add_sub is begin
process (a, b, x) begin
  if x='0' then  o <= a+b;
  else           o <= a-b;  end if;
end process;
end architecture bhv;
```

Adder, subtracter & mux

145 gates / 11.64 ns

```
architecture dfl of test5 is
  signal a1, b1, o1: unsigned(8 downto 0);
begin
  a1 <= a & '1';
  b1 <= b & '0' when x='0' else
    unsigned(not std_logic_vector(b)) &
    '1';
  o1 <= a1+b1;
  o <= o1(8 downto 1);
end architecture dfl;
```

87 gates / 12.45 ns



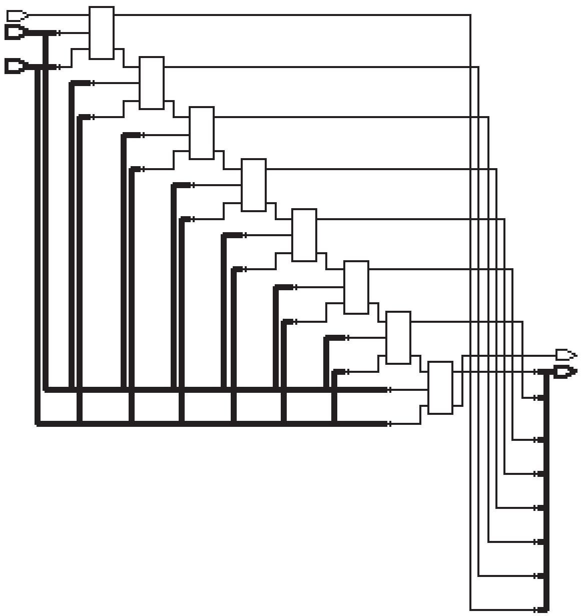
Adders & Subtracters

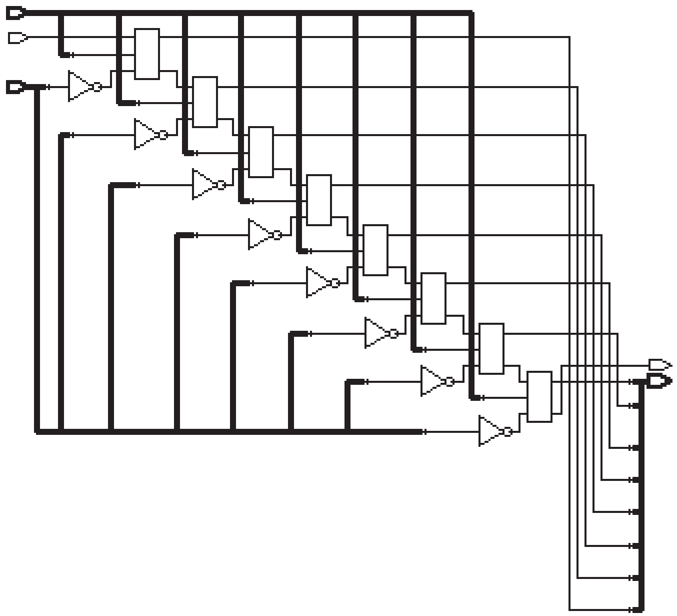
```
a1 <= '0' & a & '1';
b1 <= '0' & b & ci;
o1 <= a1 + b1;
o <= o1(8 downto 1);
co <= o1(9);
```

```
a1 <= '0' & a & '1';
b1 <= '0' &
  unsigned(not std_logic_vector(b)) & ci;
o1 <= a1 + b1;
o <= o1(8 downto 1);
co <= o1(9);
```

64 g. / 10.66 ns [60 g. / 10.08 ns w/o ci/co]

72 g. / 10.62 ns [66 g. / 10.35 ns w/o ci/co]





for-loop versus while-loop?

- May be tool dependent!
  - Design Vision (Synopsys): *for* - parallel, *while* - sequential
  - ISE (Xilinx): *for* / *while* - both parallel
  - Leonardo (Mentor Graphics): depending on the timing constructs

- for-loop
  - parallel implementation
  - no timing control (wait) in the loop body

```
for i in 0 to 7 loop
  x(i) := a(i) and b(i);
end loop;
```

- while-loop
  - sequential implementation
  - timing control (wait) required in the loop body

```
i := 0;
while i<7 loop
  data(i) := in_port;
  wait on clk until clk='1';
  i := i + 1;
end loop;
```

Multiple wait statements

- VHDL semantics must be preserved
  - different interpretations possible
- Distributing operations over multiple clock steps
- Algorithm
  - Inputs: a, b, c, d
  - Output: x
  - Coefficients: c1, c2
  - x = a + b*c1 + c*c2 + d
  - Timing constraint - 3 clock periods

```
process
  variable av, bv, cv, dv: ...;
begin
  av:=a; bv:=b; cv:=c; dv:=d;
  wait on clk until clk='1';
  wait on clk until clk='1';
  x <= av + bv * c1 + cv * c2 + dv;
  wait on clk until clk='1';
end process;
```

Multiple wait statements

- **Behavioral interpretation may lead to an unoptimal solution**

```
process
  variable av, bv, cv, dv: ...;
begin
  av:=a; bv:=b; cv:=c; dv:=d;
  wait on clk until clk='1';
  wait on clk until clk='1';
  x <= av + bv * c1 + cv * c2 + dv;
  wait on clk until clk='1';
end process;
```

2 multipliers & 3 adders

Behavioral Synthesis
(High-Level Synthesis)

```
process
  variable av, bv, cv, dv: ...;
  variable r1, r2: ...;
begin
  av:=a; bv:=b; cv:=c; dv:=d;
  r1 := av + dv;    r2 := bv * c1;
  wait on clk until clk='1';
  r1 := r1 + r2;    r2 := cv * c2;
  wait on clk until clk='1';
  x <= r1 + r2;
  wait on clk until clk='1';
end process;
```

1 multiplier & 1 adder

Inserting wait statements

- VHDL semantics preserved for inputs/outputs
  - targeting as-fast-as-possible (AFAP) schedules

- 16-tap FIR filter
  - new input and output data at every rising flank of sys_clk (sampling clock)
  - internal clock can be added
- How to implement loops?
  - 1st - in parallel (shift-register)
  - 2nd - sequentially
    - multiply-and-accumulate (MAC)
    - ROM for coefficients

```
process
  variable sum: ...;
  variable buff: ...; -- array (0 to 15)
begin
  for i in 15 downto 1 loop
    buff(i):=buff(i-1);
  end loop;
  buff(0):=data_in;    sum:=0;
  for i in 0 to 15 loop
    sum := sum + buff(i) * coeff(i);
  end loop;
  x <= sum;
  wait on sys_clk until sys_clk='1';
end process;
```

TTÜ1918

GCD (Greatest Common Divisor) example

- Specification ~~ behavioral description
  - input/output timing fixed – handshaking signals & clock

```
process -- gcd-bhv.vhdl
  variable x, y: unsigned(15 downto 0);
begin
  -- Wait for the new input data
  wait on clk until clk='1' and rst='0';
  x := xi;     y := yi;     rdy <= '0';
  wait on clk until clk='1';
  -- Calculate
  while  x /= y  loop
    if  x < y  then y := y - x;
    else            x := x - y;    end if;
  end loop;
  -- Ready
  xo <= x;   rdy <= '1';
  wait on clk until clk='1';
end process;
```

Problems
- inner loop not clocked
- complex wait statement

(- multiple wait statements)

What to look for?
- different synthesis tools
- minimizing resources
- maximizing performance

Target technologies – ASIC, FPGA

VHDL code & testbenches

http://mini.pld.ttu.ee/~lrv/gcd/gcd.html

GCD example – synthesizable code?

- **Clocked behavioral style**

```
process -- gcd-bhvc.vhdl
  variable x, y: unsigned(15 downto 0);
begin
  -- Wait for the new input data
  while  rst = '1'  loop
    wait on clk until clk='1';
  end loop;
  x := xi;     y := yi;     rdy <= '0';
  wait on clk until clk='1';
  -- Calculate
  while  x /= y  loop
    if  x < y  then y := y - x;
    else            x := x - y;    end if;
    wait on clk until clk='1';
  end loop;
  -- Ready
  xo <= x;   rdy <= '1';
  wait on clk until clk='1';
end process;
```

ASIC: synthesizable
961 e.g. / 20.0 ns
2 sub-s, 2 comp-s

FPGA: non-synthesizable
wait statements in loops :(
explicit FSM needed :(:(

Possible trade-offs
- functional unit sharing
- universal functional units
- out-of-order execution

GCD example – behavioral FSM

```
process begin -- gcd-bfsm.vhdl
  wait on clk until clk='1';
  case  state  is
  -- Wait for the new input data
  when S_wait =>
   if  rst='0'  then
     x<=xi; y<=yi; rdy<='0'; state<=S_start;
   end if;
  -- Calculate
  when S_start =>
    if  x /= y  then
      if  x < y  then  y <= y - x;
      else              x <= x - y;    end if;
      state<=S_start;
    else
      xo<=x; rdy<='1'; state<=S_ready;
    end if;
  -- Ready
  when S_ready =>    state<=S_wait;
  end case;
end process;
```

ASIC: synthesizable
911 e.g. / 19.4 ns
2 sub-s, 2 comp-s

FPGA: synthesizable
108 SLC / 9.9 ns
2 sub-s, 2 comp-s

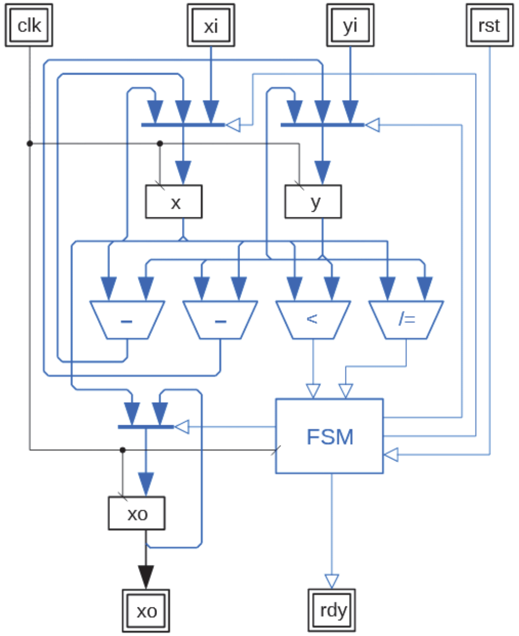
Can it be made better?

Again the possible trade-offs
- **functional unit sharing**
  - one operation per clock step
- **universal functional units**
  - A<B == A-B<0 / A/=B == A-B/=0
- **out-of-order execution**
  - subtracting first then deciding

TTÜ1918

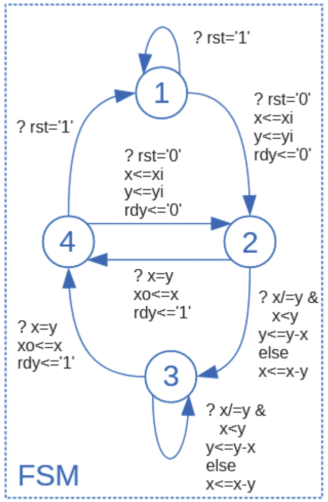
GCD example

Clocked behavioral



Only registers (signals) are described explicitly.

The rest (datapath & FSM) are described implicitly – subject for interpretations by synthesizers.

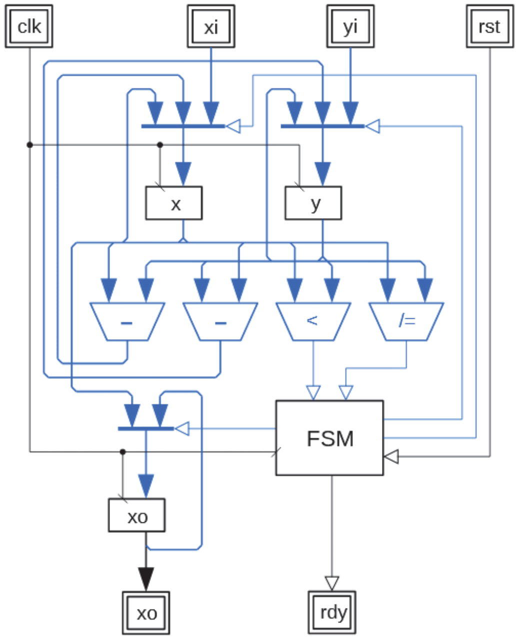


Code: gcd-bhvc.vhdl

2 subtracters, 2 comparators
[1 clock step per iteration]

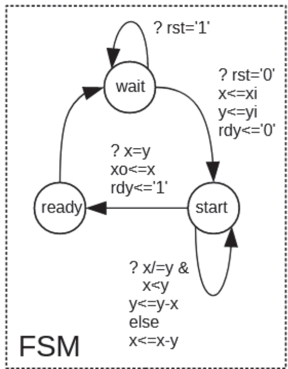
ASIC: 961 e.g. / 20.0 ns
FPGA: not synthesizable

Behavioral FSM



Registers (signals) and FSM are described explicitly.

The datapath is described implicitly – subject for interpretations by synthesizers.



Code: gcd-bfsm.vhdl

2 subtracters, 2 comparators
[1 clock step per iteration]

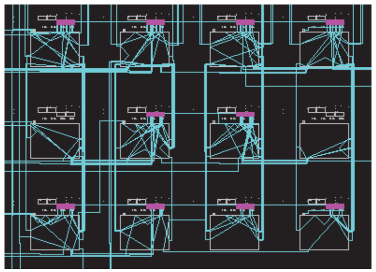
ASIC: 911 e.g. / 19.4 ns
FPGA: 108 SLC / 9.9 ns

GCD example – universal functional units?

- A<B == A-B<0 / A/=B == A-B/=0

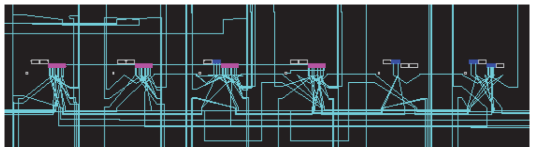
```
-- Three operations:
--   subtraction, and
--   comparisons not-equal &
--   less-than
xo <= xi - yi;

ne <= '1' when xi /= yi else '0';

lt <= '1' when xi < yi else '0';
```

ASIC: 209 e.g. / 19.9 ns
FPGA: 20 SLC / 12.1 ns
three adder chains!



```
-- ALU - subtracting and then comparing
x_out <= xi - yi;    xo <= x_out;
process (x_out)
  variable or_tmp: unsigned(15 downto 0);
begin
  or_tmp(15) := x_out(15);
  for  i in 14 downto 0  loop
    or_tmp(i) := or_tmp(i+1) or x_out(i);
  end loop;
  ne <= to_bit(or_tmp(0));
end process;
lt <= to_bit(x_out(15));
```

ASIC: 148 e.g. / 21.8 ns / FPGA: 12 SLC / 14.6 ns



GCD example – design space exploration

- Different solutions – `http://mini.pld.ttu.ee/~lrv/gcd/gcd.html`

  - gcd-bhv.vhdl – pure behavioral description, non-synthesizable
  - gcd-bhvc.vhdl – fully clocked behavioral style, some synthesis tools can handle
  - gcd-bfsm.vhdl – so called behavioral FSM (explicit FSM & behavioral data-path), synthesizable (but how efficient it is?)
  - gcd-rtl1.vhdl – single ALU, 3 clock cycles per iteration – 1) "not equal?", 2) "less than?", 3) subtract
  - gcd-rtl2.vhdl – single ALU, 2 clock cycles per iteration – 1) "not equal?" and "less than?", 2) subtract
  - gcd-rtl3.vhdl – comparator (less than) controls subtraction, 1 clock cycle per iteration – small but slow (sequential) data-path
  - gcd-rtl4.vhdl – out-of-order execution – both subtractions are calculated first then the decision is made (one subtracter compares for "less than", another for "not equal")
  - gcd-rtl5.vhdl – out-of-order execution – both subtractions are calculated first then the decision is made (one subtracter compares for "less than" but separate "not equal")

TTÜ1918

GCD example – single ALU, 2 clock cycles per iteration

gcd-rtl2.vhdl

```
-- Next state function of the FSM
process (state, rst, alu_ne, alu_lt) begin
  ena_x <= '0';      ena_y <= '0';       ena_r <= '0';
  set_rdy <= '0';   xi_yi_sel <= '0';   sub_y_x <= '0';
  next_state <= state;
  case state  is
  when S_wait =>    -- Wait for the new input data
    if  rst='0'  then
      xi_yi_sel <= '1';   ena_x <= '1';   ena_y <= '1';
      next_state <= S_start;
    end if;
  when S_start =>    -- Loop: ready?
    if  alu_ne='1'  then
      if  alu_lt='1'  then    next_state <= S_sub_y_x;
      else    next_state <= S_sub_x_y;    end if;
    else    next_state <= S_ready;
    end if;
  when S_sub_y_x =>    -- Loop: y-x
    ena_y <= '1';    sub_y_x <= '1';
    next_state <= S_start;
  when S_sub_x_y =>    -- Loop: x-y
    ena_x <= '1';    sub_y_x <= '0';
    next_state <= S_start;
  when S_ready =>    -- Ready
    ena_r <= '1';    set_rdy <= '1';
    next_state <= S_wait;
  end case;
end process;
```

```
-- ALU: subtract / less-than / not-equal
alu_o <= alu_1 - alu_2;
alu_lt <= to_bit(alu_o(15));
process (alu_o)
  variable or_tmp: unsigned(15 downto 0);
begin
  or_tmp(15) := alu_o(15);
  for  i in 14 downto 0  loop
    or_tmp(i) := or_tmp(i+1) or alu_o(i);
  end loop;
  alu_ne <= to_bit(or_tmp(0));
end process;

-- Multiplexers
x_i <= xi when xi_yi_sel='1' else alu_o;
y_i <= yi when xi_yi_sel='1' else alu_o;
alu_1 <= y when sub_y_x='1' else x;
alu_2 <= x when sub_y_x='1' else y;

-- Registers
process begin
  wait on clk until clk='1';
  state <= next_state;
  if  ena_x='1'  then    x <= x_i;    end if;
  if  ena_y='1'  then    y <= y_i;    end if;
  if  ena_r='1'  then    xo <= x;     end if;
  rdy <= set_rdy;
end process;
```

GCD example – out-of-order execution (2 sub-s)

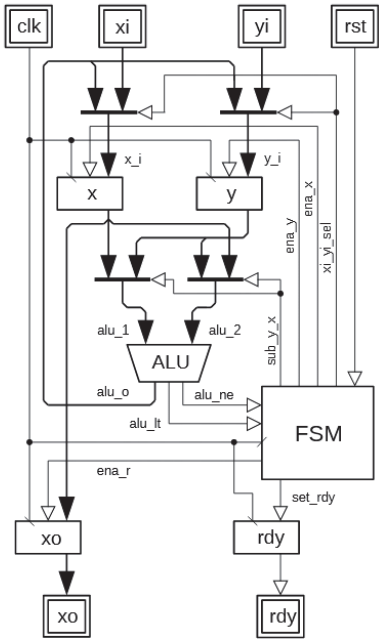
gcd-rtl5.vhdl

```
-- Next state function of the FSM
process (state, rst, alu_ne) begin
  ena_xy <= '0';     ena_r <= '0';
  set_rdy <= '0';    xi_yi_sel <= '0';
  next_state <= state;
  case state is
  -- Wait for the new input data
  when S_wait =>
    if  rst='0'  then
      xi_yi_sel <= '1';    ena_xy <= '1';
      next_state <= S_start;
    end if;
  -- Calculate
  when S_start =>
    if  alu_ne='1'  then
      ena_xy <= '1';
      next_state <= S_start;
    else
      ena_r <= '1';      set_rdy <= '1';
      next_state <= S_ready;
    end if;
  -- Ready
  when S_ready =>
    ena_r <= '1';      set_rdy <= '1';
    next_state <= S_wait;
  end case;
end process;
```

```
-- Subtracter (x-y) / comparator (x<y)
alu_o1 <= x - y;
sub_y_x <= '1' when alu_o1(alu_o1'high)='1' else '0';

-- Subtracter (y-x)
alu_o2 <= y - x;

-- Comparator (y/=x)
alu_ne <= '1' when x /= y else '0';

-- Multiplexers
x_i <= xi when xi_yi_sel='1' else alu_o1;
y_i <= yi when xi_yi_sel='1' else alu_o2;
ena_x <= '1' when (sub_y_x='0' and ena_xy='1') or
         xi_yi_sel='1' else '0';
ena_y <= '1' when (sub_y_x='1' and ena_xy='1') or
         xi_yi_sel='1' else '0';

-- Registers
process begin
  wait on clk until clk='1';
  state <= next_state;
  if  ena_x='1'  then    x <= x_i;    end if;
  if  ena_y='1'  then    y <= y_i;    end if;
  if  ena_r='1'  then    xo <= x;    end if;
  rdy <= set_rdy;
end process;
```

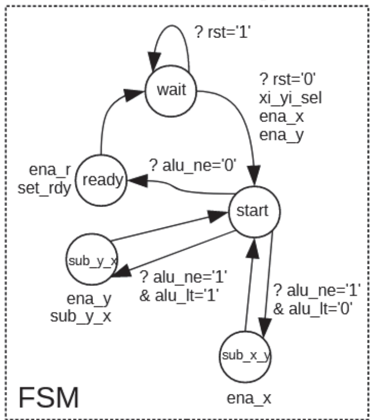
GCD example

Single ALU (RTL #2)



Register-transfer level description with universal ALU (operation reuse).

Default value for the control signals is '0'.

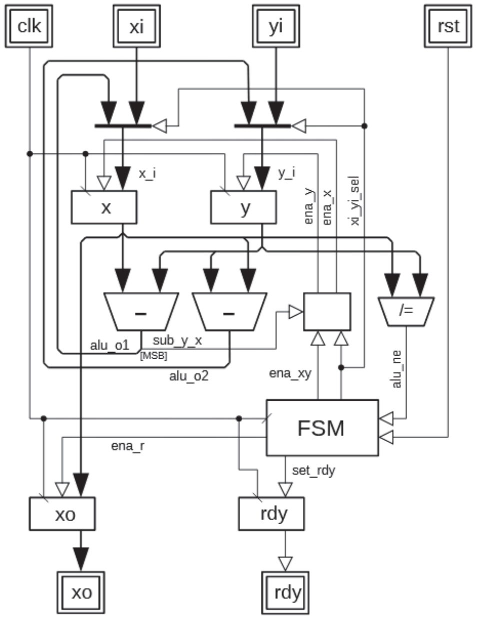


Code: gcd-rtl2.vhdl

Single ALU ("–", "<", "/=")
[2 clock steps per iteration]

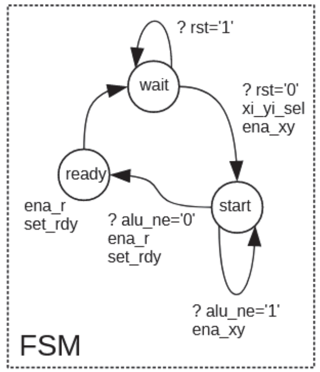
ASIC: 931 e.g. / 19.9 ns
FPGA: 48 SLC / 10.8 ns

Out-of-order execution (RTL #5)



Register-transfer level description with out-of-order subtractions. Only data-path differs from RTL #3 & #4.

Default value for the control signals is '0'.



Code: gcd-rtl5.vhdl

2 subtracters, 1 comparator
[1 clock step per iteration]

ASIC: 915 e.g. / 20.0 ns
FPGA: 58 SLC / 8.0 ns

TTÜ1918

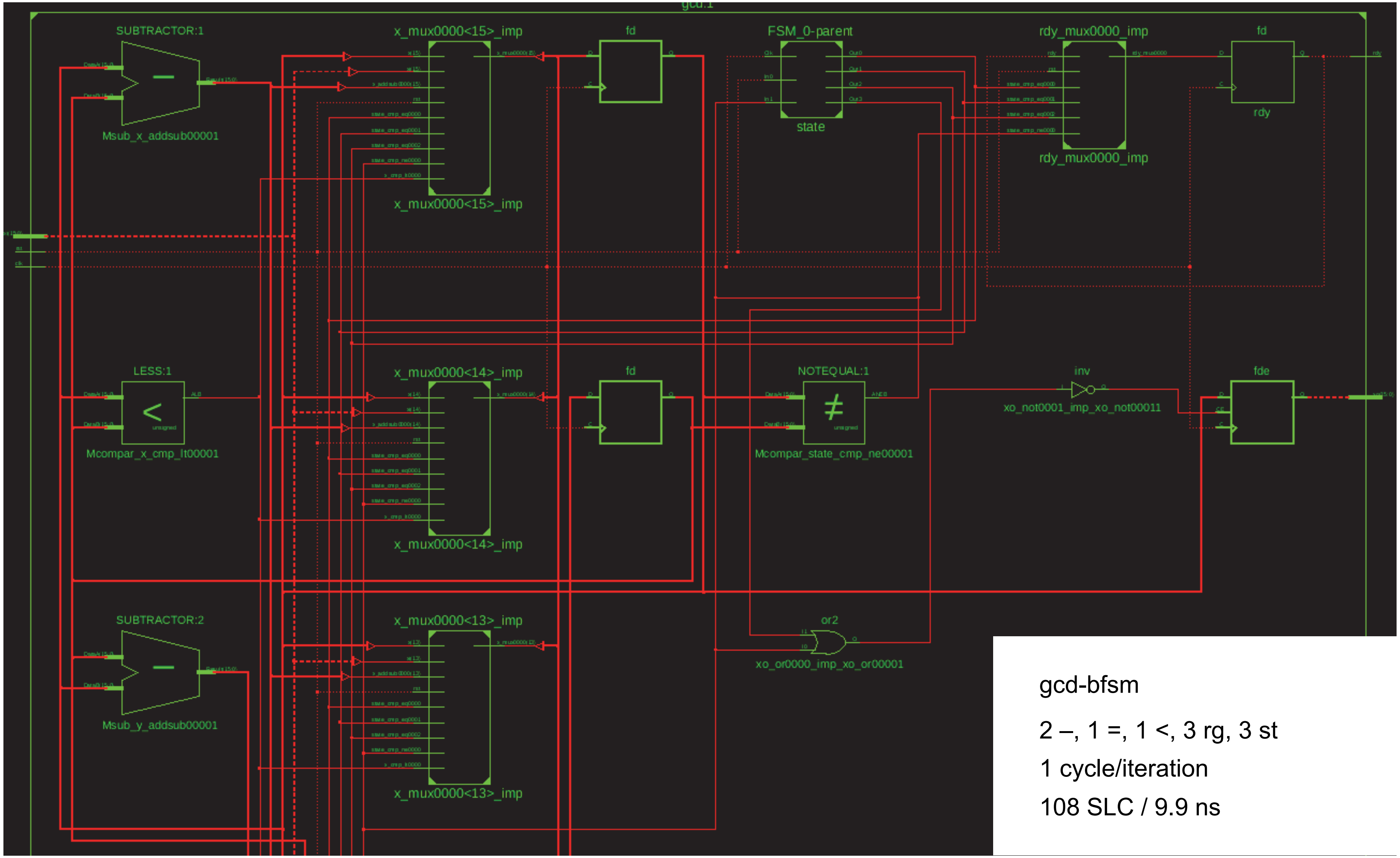
GCD example – synthesis results

Technology	FPGA				ASIC			
Constraint[1)]	50 MHz		100 MHz		50 MHz		25 MHz	
	[SLC]	[ns]	[SLC]	[ns]	[e.g.]	[ns]	[e.g.]	[ns]
gcd-bhv[2)]	*93*	*17.3*	-	-	*1141*	*20.0*	-	-
gcd-bhvc	-	-	-	-	961	20.0	977	31.1
gcd-bfsm	108	9.9	108	9.4	911	19.4	984	30.8
gcd-rtl1	50	10.8	50	9.7	986	19.8	883	32.4
gcd-rtl2	48	10.8	48	10.0	931	19.9	882	32.3
gcd-rtl3	58	17.0	58	14.6	1134	20.0	928	40.0
gcd-rtl4	78	12.6	78	9.0	976	19.9	928	29.0
gcd-rtl5	58	8.0	58	7.6	915	20.0	932	26.9

[1)] Clock period was the only constraint

[2)] gcd-bhv was synthesized using the help of prototype HLS tool xTractor

GCD example – why such differences?



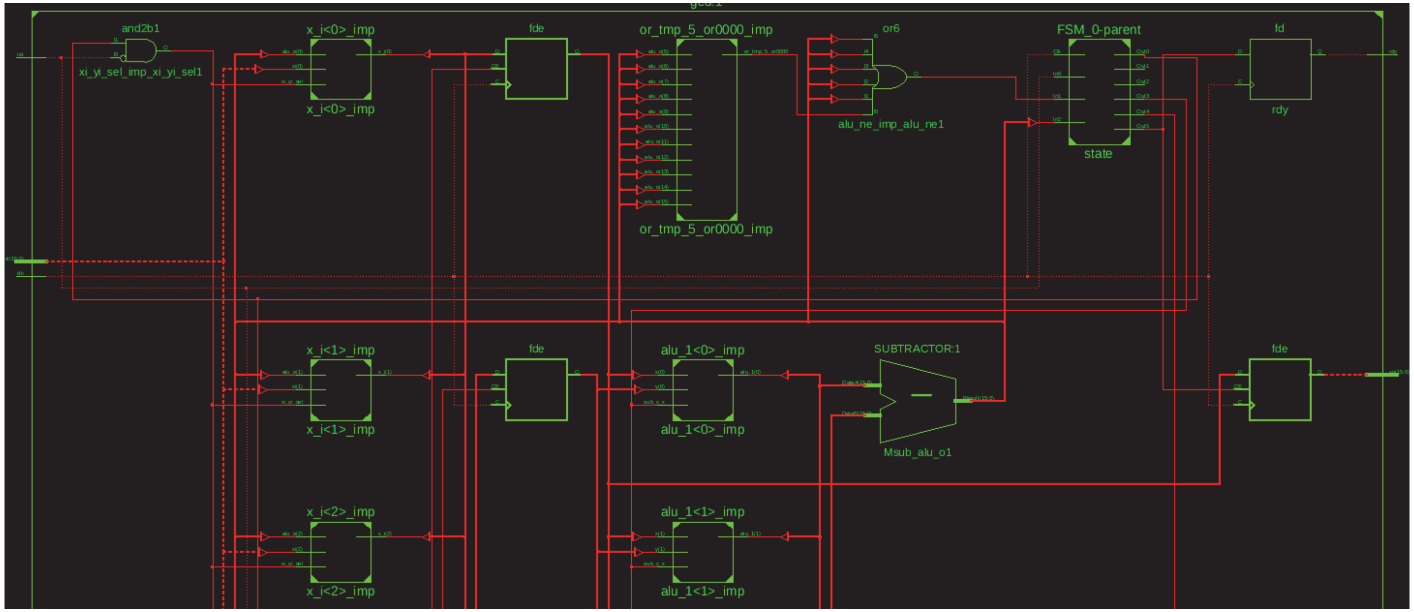
gcd-bfsm

2 –, 1 =, 1 <, 3 rg, 3 st

1 cycle/iteration

108 SLC / 9.9 ns

GCD example – why such differences?



gcd-rtl1

1 ALU, 3 rg, 6 st

3 cycles/iteration

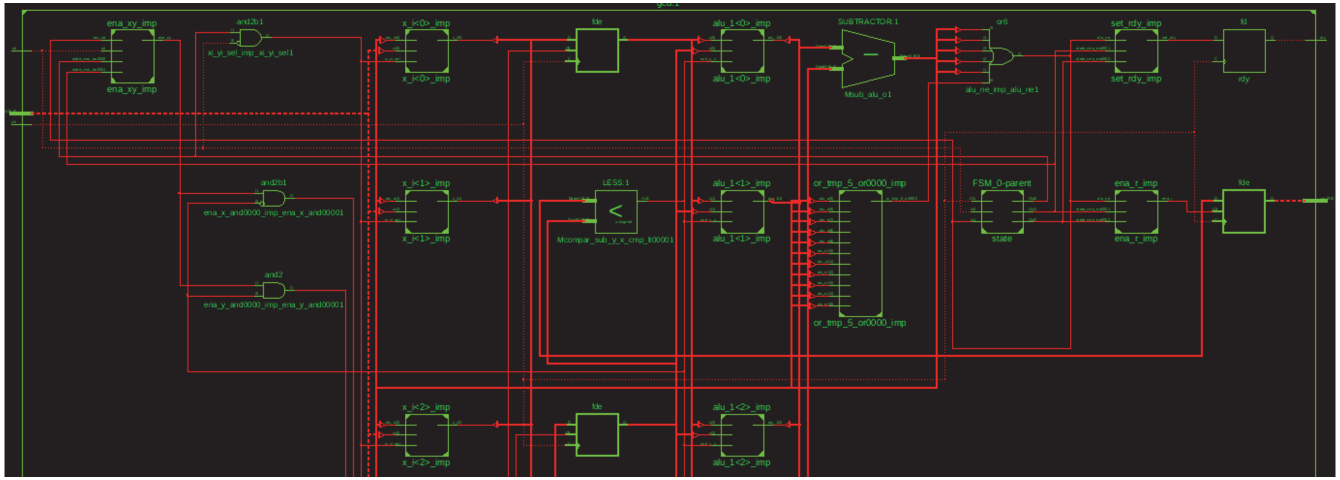
50 SLC / 10.8 ns

gcd-rtl2

1 ALU, 3 rg, 5 st

2 cycles/iteration

48 SLC / 10.8 ns



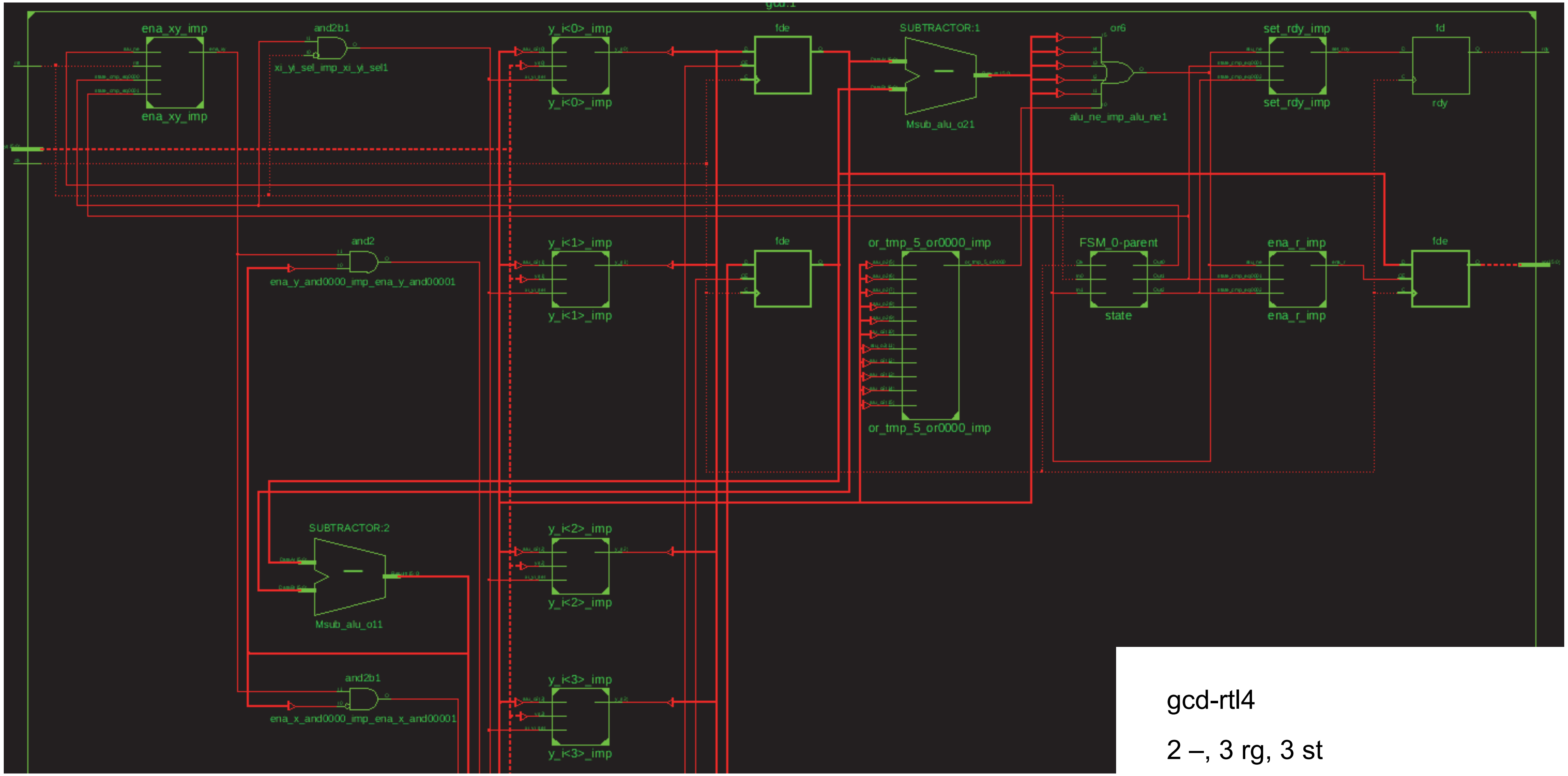
gcd-rtl3

1 –, 1 <, 3rg, 3st

1 cycle/iteration

58 SLC / 17.0 ns

GCD example – why such differences?



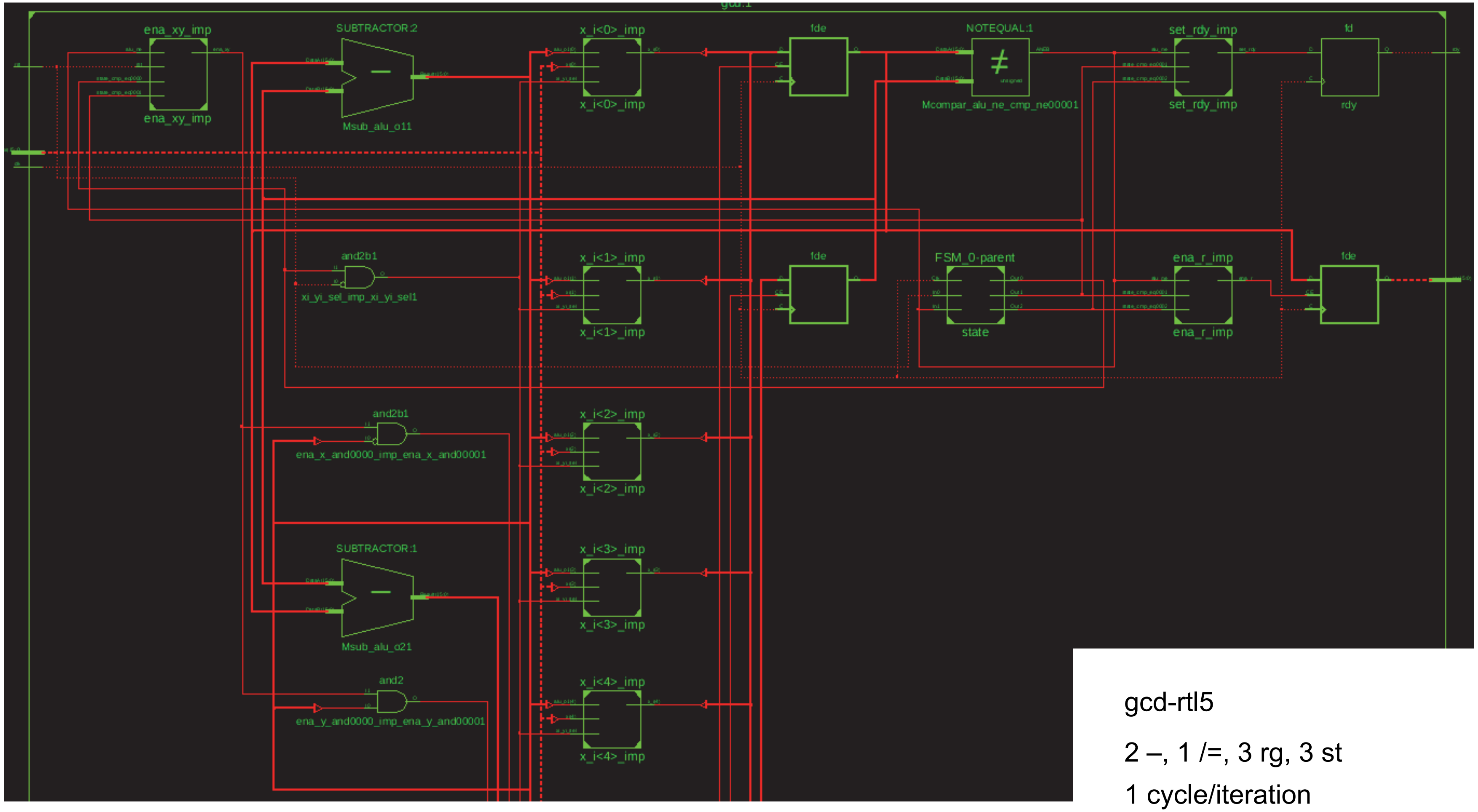
gcd-rtl4

2 –, 3 rg, 3 st

1 cycle/iteration

78 SLC / 12.6 ns

GCD example – why such differences?



gcd-rtl5

2 –, 1 /=, 3 rg, 3 st

1 cycle/iteration

58 SLC / 8.0 ns