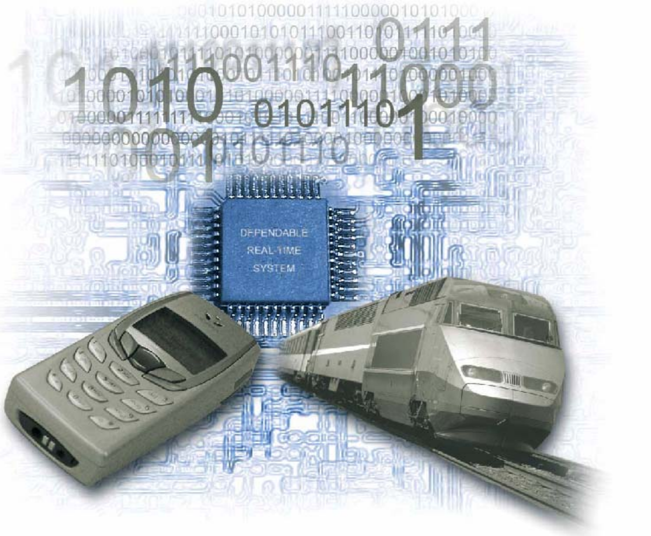
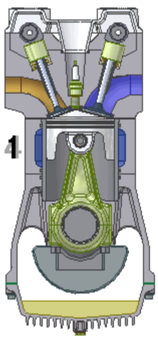
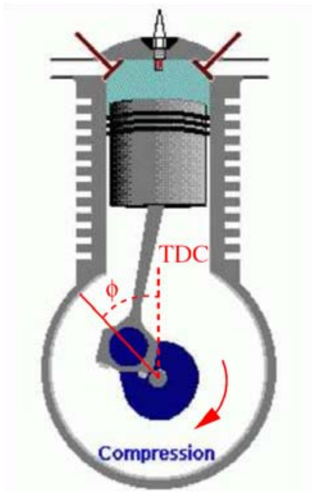
Embedded Systems 10

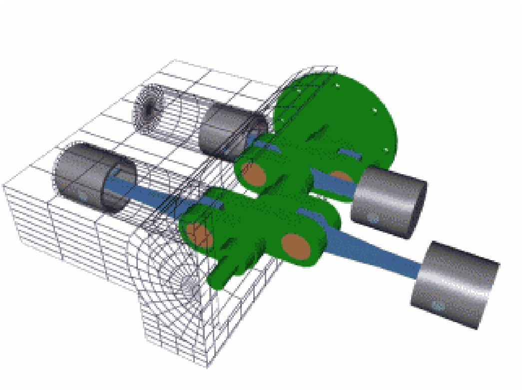


Project: Ignition Controller

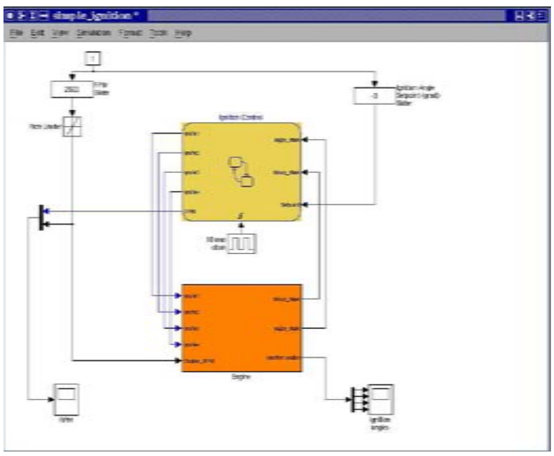




Project: Ignition Controller



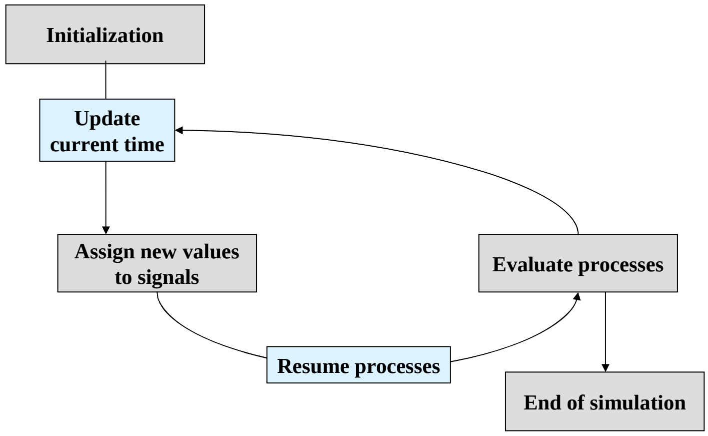
Project: Ignition Controller



Due: December 11, 2008

Overview of simulation

REVIEW



Initialization

REVIEW

- At the beginning of initialization, the current time, t_{curr}, is assumed to be 0 ns.
- An initial value is assigned to each signal.
  - Taken from declaration, if specified there, e.g.,
    - **signal** s : std_ulogic := \`0\`;
  - Otherwise: First value in enumeration for enumeration based data types, e.g.
    - **signal** s : std_ulogic
      with
      type std_ulogic **is** (\`U\`, \`X\`, \`0\`, \`1\`, \`Z\`, \`W\`, \`L\`, \`H\`, \`-\`);
      ⇒ initial value is \`U\`
  - This value is assumed to have been the value of the signal for an infinite length of time prior to the start of the simulation.
- Initialization phase executes each process exactly once (until it suspends).
- During execution of processes: Signal assignments are collected in transaction list (**not** executed immediately!) – more details later.
- If process stops at „wait for"-statement, then update process activation list – more details later.
- After initialization the time of the next simulation cycle (which in this case is the first simulation cycle), t_{next} is calculated:
  - Time t_{next} of the next simulation cycle = earliest of
    1. time'high (end of simulation time).
    2. Earliest time in transaction list (if not empty).
    3. Earliest time in process activation list (if not empty).

Signal assignment phase – first part of step

REVIEW

- Each simulation cycle starts with setting the current time to the next time at which changes must be considered:
- $t_{curr} = t_{next}$
- This time t_{next} was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time would exceed its maximum, time'high.
- For all (s, v, t_{curr}) in transaction list:
  - Remove (s, v, t_{curr}) from transaction list.
  - *s* is set to *v*.
- For all processes p_i which wait on signal s:
  - Insert (p_i, t_{curr}) in process activation list.
- Similarly, if condition of „**wait until**"-expression changes value.

Process execution phase – second part of step (1)

REVIEW

- Resume all processes p_i with entries (p_i, t_{curr}) in process activation list.
- Execute all activated processes „in parallel" (in fact: in arbitrary order).
- Signal assignments
  - are collected in transaction list (**not** executed immediately!).
  - Examples:
    - *s* <= *a* **and** *b*;
      - Let *v* be the conjunction of current value of *a* and current value of *b*.
      - Insert (s, v, t_{curr}) in transaction list.
    - *s* <= ´1´ **after** 10 ns;
      - Insert (*s*, ´1´, t_{curr} + 10 ns) into transaction list.
- Processes are executed until wait statement is encountered.
- If process p_i stops at „wait for"-statement, then update process activation list:
  - Example:
    - p_i stops at „**wait for** 20 ns;"
    - Insert (p_i, t_{curr} + 20 ns) into process activation list

Process execution phase – second part of step (2)

REVIEW

If some process reaches last statement and

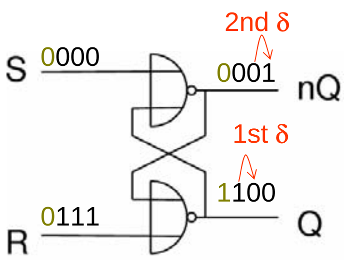
- does not have a sensitivity list and
- last statement is not a wait statement,

then it continues with first statement and runs until wait statement is reached.

- When all processes have stopped, the time of the next simulation cycle t_{next} is calculated:
  - Time t_{next} of the next simulation cycle = earliest of
    1. time'high (end of simulation time).
    2. Earliest time in transaction list (if not empty)
    3. Earliest time in process activation list (if not empty).
- Stop if t_{next} = time'high and transaction list and process activation list are empty.

Delta delay - Simulation of an RS-Flipflop

REVIEW



	0ns	0ns+δ	0ns+2δ
R	1	1	1
S	0	0	0
Q	1	0	0
nQ	0	0	1

```
entitiy RS_Flipflop is
   port (R, S : in std_logic;
            Q, nQ : inout std_logic);
end RS_FlipFlop;

architecture one of RS_Flipflop is
   begin
   process (R,S,Q,nQ)
   begin
            Q := R nor nQ;
            nQ := S nor Q;
   end process;
end one;
```

δ cycles reflect the fact that no real gate comes with zero delay.

„Write-write-conflicts“

REVIEW

```
signal s : bit;
…
p : process
begin
    …
    s <= ‘0‘;
    …
    s <= ‘1‘;
    wait for 5 ns;
end process p;
```

- **Case 1:**
  Write-write-conflicts are restricted to the same process
  (i.e. they occur inside the same process)
  - Then the second signal assignment overwrites the first one.
  - This is the only case of „non-concurrency“ of signal assignments
  - Note that writing to different signals occurs concurrently, however!

„Write-write-conflicts“

REVIEW

```
signal s : dt;
…
s<= v1;
…
p : process
begin
    …
    s <= v2;
    …
end process p;

q : process
begin
    …
    s <= v3;
    …
end process q;
```

- **Case 2:**
  Write-write-conflicts between different processes (explicit or implicit processes)
  - If there is no **„resolution function“** for the data type *dt,* then writing the same signal by different processes in the same step is **forbidden**.
  - If there is a resolution function, then the resolution function computes the value of s at time t_{curr}:
    - Value for s in the current step is computed for each process separately,
    - „resolution function“ for different values is used to compute final result.
- In the following:
  Data type std_ulogic with resolution function
  ⇒ data type std_logic

Multi-valued logic and standard IEEE 1164

REVIEW

- How many logic values for modeling?
- Two ('0' and '1') or more?
- If real circuits have to be described, some abstraction of the resistance (inversely-related to the strength) is required.

⇒ We introduce the distinction between:
  - the **logic level** (as an abstraction of the voltage) and
  - the **strength** (as an abstraction of the current drive capability) of a signal.
- Both logic level and strength are encoded in **logic values**.

1 signal strength

REVIEW

- Logic values '0' and '1'.
- Both of the same strength.
- Encoding false and true, respectively.

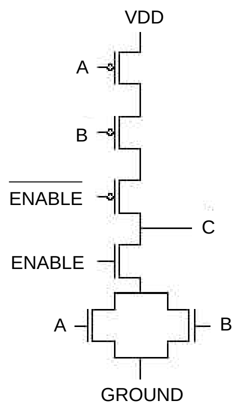
- No meaningful “resolution function” possible, if `0` and `1` are written to the same signal at the same time.

2 signal strengths (1)

REVIEW

- Many subcircuits can be effectively disconnected from the rest of the circuit (they provide „high impedance“ values to the rest of the circuit).
- Example: subcircuits with tri-state outputs.

Example: Tristate NOR



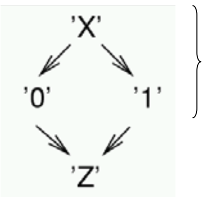
ENABLE = \`0\`
⇒ C is disconnected from the rest of the circuit

☞ We introduce signal value 'Z', meaning „high impedance “

2 signal strengths (2)

REVIEW

- We introduce an operation #, which generates the effective signal value whenever two signals are connected by a wire (“resolution”).
- #('0','Z')='0'; #('1','Z')='1'; '0' and '1' are „stronger“ than 'Z'



1 strength

According to the partial order in the diagram, # returns the *larger of the two arguments*.

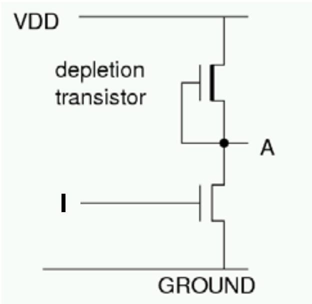
In order to define #('0','1'), we introduce 'X', denoting an undefined signal level.
'X' has the same strength as '0' and '1'.

3 signal strengths

REVIEW

Current set of values insufficient for describing real circuits:

Example:
nMOS-Inverter

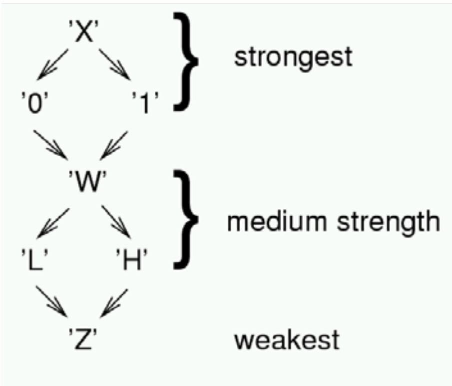


Depletion transistor (resistor) contributes a weak value to be considered in the #-operation for signal A
☞ Introduction of 'H', denoting a weak signal of the same level as '1'.
#('H', '0')='0'; #('H,'Z') = 'H'

3 signal strengths

REVIEW

- There may also be weak signals of the same level as '0'
- ☞ Introduction of 'L', denoting a weak signal of the same level as '0':
#('L', '0')='0'; #('L,'Z') = 'L';
- ☞ Introduction of 'W', denoting a weak signal of the same level as 'X':
#('L', 'H')='W'; #('L,'W') = 'W';
- # reflected by the partial order shown.



IEEE 1164

REVIEW

- VHDL allows user-defined value sets.

⇒ Each model could use different value sets (unpractical)

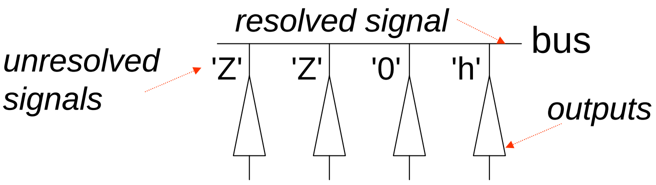
⇒ Definition of standard value set according to standard IEEE 1164:

{'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}

- First seven values as discussed previously.
- 'U': un-initialized signal; used by simulator to initialize all not explicitly initialized signals:
  type std_ulogic **is** (`U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, `-`);
- '-': is used to specify don't cares:
  - Example: **if** a /= '1' **or** b/='1' **then** f <= a **exor** b; **else** f <= '-';
  - '-' may be replaced by arbitrary value by synthesis tools.

Outputs tied together

- In hardware, connected outputs can be used:



Modeling in VHDL: resolution functions
type std_ulogic **is** ('U', 'X','0', '1', 'Z', 'W', 'L', 'H', '-');
subtype std_logic **is** resolved std_ulogic;

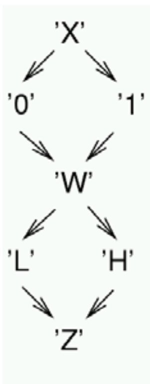
Resolution function for IEEE 1164

```
type std_ulogic_vector is array(natural range<>)of std_ulogic;

function resolved (s:std_ulogic_vector) return std_logic is
  variable result: std_ulogic:='Z';   --weakest value is default
  begin
    if (s'length=1) then return s(s'low) --no resolution
    else for i in s'range loop
      result:=resolution_table(result,s(i))
    end loop
    end if;
   return result;
  end resolved;
```


Resolution function for IEEE 1164

```
constant resolution_table : stdlogic_table := (
--U   X   0   1   Z  W    L   H  –
('U', 'U', 'U', 'U', 'U', 'U',  'U', 'U', 'U'),  --| U |
('U', 'X', 'X', 'X', 'X', 'X',  'X', 'X', 'X'),  --| X |
('U', 'X', '0', 'X', '0', '0',  '0', '0', 'X'),  --| 0 |
('U', 'X', 'X', '1', '1', '1',  '1', '1', 'X'),  --| 1 |
('U', 'X', '0', '1', 'Z', 'W',  'L', 'H', 'X'),  --| Z |
('U', 'X', '0', '1', 'W', 'W',  'W', 'H', 'X'),  --| W |
('U', 'X', '0', '1', 'L', 'W',  'L', 'W', 'X'),  --| L |
('U', 'X', '0', '1', 'H', 'W',  'W', 'H', 'X'),  --| H |
('U', 'X', 'X', 'X', 'X', 'X',  'X', 'X', 'X')   --| - |
);
```



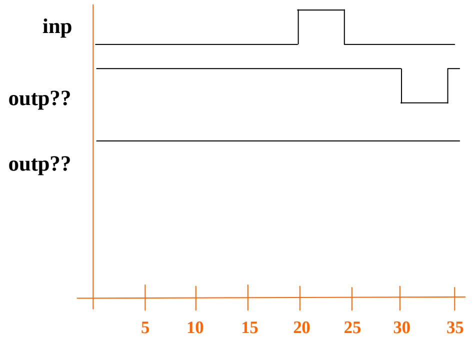
Inertial and transport delay model

- Signal assignment:
  - ```
    signal_assignment ::=
            target <= [ delay_mechanism ] waveform_element
                                       { , waveform_element }
    ```
  - ```
    waveform_element ::=
            value_expression [ after  time_expression ]
    ```
  - ```
    delay_mechanism ::=
            transport | [ reject  time_expression ] inertial
    ```
- Example:
  - Inpsig <= ´0´, ´1´after 5 ns, ´0´ after 10 ns, ´1´ after 20 ns;

Inertial and transport delay model

- Example for signal assignment:

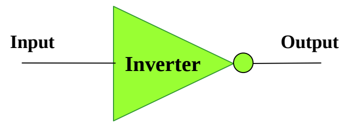
  outp <= **not** inp **after** 10 ns;


    ```

Inertial and transport delay model

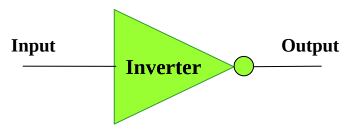
Two delay models in VHDL:

- **Inertial delay** („*träge Verzögerung*“)
- **Transport delay** („*nichtträge Verzögerung*“)



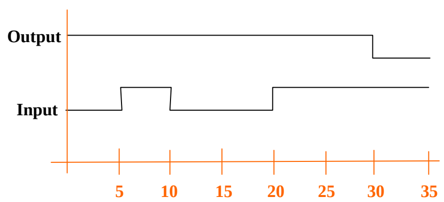
- Inertial delay model is motivated by the fact that physical gates absorb short pulses (spikes) at their inputs (due to internal capacities)

Inertial delay model



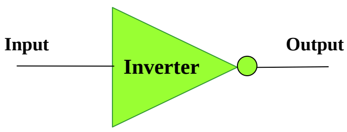
- … is the **default** model!
- Allows to specify the delay of a gate or operation
- Absorbs pulses at the inputs which are shorter than the delay specified for the gate / operation

```
-- INERTIAL is the default
Output <= NOT input AFTER 10 ns;
```

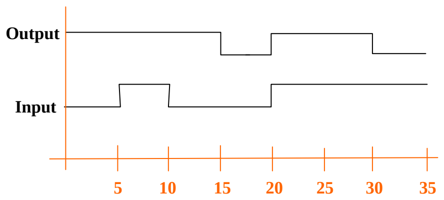


Transport delay model

- Allows to specify the delay of a gate or operation
- Transmits all pulses at the inputs ideally




```
-- TRANSPORT must be specified
Output <= TRANSPORT NOT input AFTER 10 ns;
```

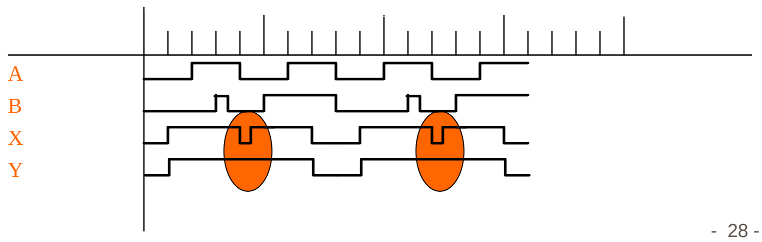


Inertial and transport delay model

```
entity DELAY is
end DELAY;

architecture RTL of DELAY is
  signal A, B, X, Y: bit;
begin
  p0: process (A, B)
  begin
    Y <= A nand B after 10 ns;
    X <= transport A nand B after 10 ns;
  end process;

  p1: process
  begin
    A <= '0', '1' after 20 ns, '0'
         after 40 ns, '1' after 60 ns;
    B <= '0', '1' after 30 ns, '0'
         after 35 ns, '1' after 50 ns;
    wait for 80 ns;
  end process
end RTL;
```

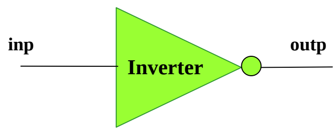


Semantics of transport delay model

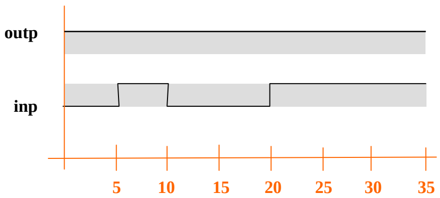
- Restriction (at first):
  - Do not consider resolution etc., i.e., assignments to a fixed signal only made in one process
- Signal assignments change transaction list.
- Before transaction (s, t_1, v_1) is inserted into transaction list, all transactions in the transaction list (s, t_2, v_2) with $t_2 \geq t_1$ are removed from transaction list.

Example for transport delay model

```
inv : process(inp)
begin
   if inp='1' then
      outp <= transport '0' after 20 ns;
   elsif inp='0' then
      outp <= transport '1' after 12.5 ns
   end if;
end process inv;
```



- Transaction list:
  - At 5ns:
    (outp, 25ns, `0`)
  - At 10 ns:
    (outp, 22.5ns, `1`), (outp, 25ns, `0`)
    Remove (outp, 25ns, `0`)!
    ⇒
    (outp, 22.5ns, `1`)



Semantics of inertial delay model

- Semantics for more general version of inertial delay statement:
  - Inertial delay absorbs pulses at the inputs which are shorter than the delay specified for the gate / operation.
  - Key word **reject** permits absorbing only pulses which are shorter than specified delay:
    - Example:
      - outp <= **reject** 3 ns **inertial not** inp **after** 10 ns;
      - Only pulses smaller than 3 ns are absorbed.
      - outp <= **reject** 10 ns **inertial not** inp **after** 10 ns;
        and
        outp <= **not** inp **after** 10 ns;
        are equivalent.

Semantics of inertial delay model

- Same restriction as for transport model (at first):
  - Do not consider resolution etc., i.e., assignments to a fixed signal only made in one process
- Rule 1 as for transport delay model:
  Before transaction (s, t_1, v_1) is inserted into transaction list, all transactions in the transaction list (s, t_2, v_2) with $t_2 \geq t_1$ are removed from transaction list.
- Rule 2 removes also some transactions with times < t_1:
  - Suppose the time limit for reject is rt.
  - Transactions for signal s with time stamp in the intervall (t_1 – rt, t_1) are removed.
  - Exception:
    If there is in (t_1 – rt, t_1) a subsequence of transactions for s immediately before (s, t_1, v_1) which also assign value v_1 to s, then these transactions are preserved.

Example

```
process
begin
   o1 <= transport '0', '0' after 5ns, '1' after 15 ns, '0' after 20ns,
                        '1' after 25 ns, '1' after 30ns, '1' after 45 ns,
                        '0' after 50 ns;
   -- same signal assignment for o2
   o2 <= transport '0', '0' after 5ns, '1' after 15 ns, '0' after 20ns,
                        '1' after 25 ns, '1' after 30ns, '1' after 45 ns,
                        '0' after 50 ns;
   wait for 15 ns;
    o2 <= reject 22 ns inertial '1' after 25 ns;
   wait;
end process;
```

- Transaction list until „`wait for 15 ns`":
  (o1, 0ns, `0`), (o1, 5ns, `0`), (o1, 15ns, `1`), (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 0ns, `0`), (o2, 5ns, `0`), (o2, 15ns, `1`), (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 45ns, `1`), (o2, 50ns, `0`)
- Transaction list when process is reactivated at time 15ns:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 45ns, `1`), (o2, 50ns, `0`)
- …

Example

```
process
begin
   o1 <= transport '0', '0' after 5ns, '1' after 15 ns, '0' after 20ns,
                        '1' after 25 ns, '1' after 30ns, '1' after 45 ns,
                        '0' after 50 ns;
   -- same signal assignment for o2
   o2 <= transport '0', '0' after 5ns, '1' after 15 ns, '0' after 20ns,
                        '1' after 25 ns, '1' after 30ns, '1' after 45 ns,
                        '0' after 50 ns;
   wait for 15 ns;
    o2 <= reject 22 ns inertial '1' after 25 ns;
   wait;
end process;
```

- At time 15ns:
  - insert transaction (o2, 40ns, `1`).
  - Remove transactions with time stamp $\geq$ 40ns.
- Results in preliminary transaction list:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 40ns, `1`)
- …

Example

```
process
begin
   o1 <= transport ‘0‘, ‘0‘ after 5ns, ‘1‘ after 15 ns, ‘0‘ after 20ns,
                    ‘1‘ after 25 ns, ‘1‘ after 30ns, ‘1‘ after 45 ns,
                    ‘0‘ after 50 ns;
   -- same signal assignment for o2
   o2 <= transport ‘0‘, ‘0‘ after 5ns, ‘1‘ after 15 ns, ‘0‘ after 20ns,
                    ‘1‘ after 25 ns, ‘1‘ after 30ns, ‘1‘ after 45 ns,
                    ‘0‘ after 50 ns;
   wait for 15 ns;
    o2 <= reject 22 ns inertial ‘1‘ after 25 ns;
   wait;
end process;
```

- Results in preliminary transaction list:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 20ns, `0`), (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 40ns, `1`)
- Rule 2:
  - (o2, 25ns, `1`), (o2, 30ns, `1`) are preserved,
  - (o2, 20ns, `0`), is removed.
- Resulting transaction list:
  (o1, 20ns, `0`), (o1, 25ns, `1`), (o1, 30ns, `1`), (o1, 45ns, `1`), (o1, 50ns, `0`),
  (o2, 25ns, `1`), (o2, 30ns, `1`), (o2, 40ns, `1`)

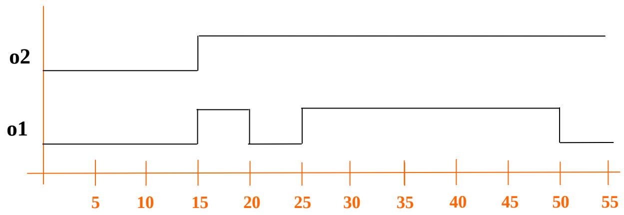
Rule 2:

- Transactions for signal o2 with time stamp in the intervall (40ns – 22ns, 40ns) = (18ns, 40ns) are removed.
- Exception:
  If there is in (18ns, 40ns) a subsequence of transactions for o2 immediately before (o2, 40ns, `1`) which also assign value `1` to o2, then these transactions are preserved.

Example

```
process
begin
   o1 <= transport ‘0‘, ‘0‘ after 5ns, ‘1‘ after 15 ns, ‘0‘ after 20ns,
                    ‘1‘ after 25 ns, ‘1‘ after 30ns, ‘1‘ after 45 ns,
                    ‘0‘ after 50 ns;
   -- same signal assignment for o2
   o2 <= transport ‘0‘, ‘0‘ after 5ns, ‘1‘ after 15 ns, ‘0‘ after 20ns,
                    ‘1‘ after 25 ns, ‘1‘ after 30ns, ‘1‘ after 45 ns,
                    ‘0‘ after 50 ns;
   wait for 15 ns;
    o2 <= reject 22 ns inertial ‘1‘ after 25 ns;
   wait;
end process;
```

- Resulting wave form:



Inertial and transport delay model

- For signal assignments of form
  Inpsig <= ´0´ after 5 ns, ´1´after 10 ns, ´0´ after 15 ns, ´1´ after 20 ns;
  only the first assignment follows the inertial delay model.

- If there are assignments to a signal s in several processes $p_1, \ldots, p_n$:
  - Insert entries of form (s^{P_i}, t, v) into transaction list („for each signal driver separate entries")
  - Apply rules for inertial/transport delay model as defined above (separately) to signals s^{P_i}.
  - If there are several entries (s^{P_i}, t_{curr}, v_i) in current assignment phase:
    - Apply resolution function to compute resulting value for assignment to s.

Some additional language elements

- VHDL supports usual elements of imperative programming languages, e.g.,
  - Various data types
    - scalar data types like integers, reals, enumeration types, physical types,
    - arrays,
    - pointers,
    - records,
    - files
  - Various control structures (if, case, when … else, with … select etc.)
  - Loops (loop, for, while)
  - Functions and procedures
  - …

Functions and procedures

- Apart from entities / architectures there are also functions and procedures in the usual (software) sense.
- Functions are typically used for providing conversion between data types or for defining operators on user-defined data types.
- Procedures may have parameters of directions **in**, **out** and **inout**.
  - **in** comparable to *call by value*,
  - **out** for providing results,
  - **inout** comparable to *call by reference*.

Example

```
architecture RTL of TEST is
  function BOOL2BIT (BOOL: boolean) return bit is
  begin
    if BOOL then return '1'; else return '0'; end if;
  end BOOL2BIT;

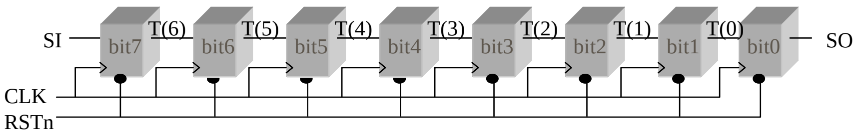
  procedure EVEN_PARITY (
      signal D: in bit_vector(7 downto 0);
      signal PARITY : out bit ) is
    variable temp : bit;
  begin
    ....
  end;

  signal DIN : bit_vector(7 downto 0);
  signal BOOL1 : boolean;
  signal BIT1, PARITY : bit;
 begin
  do_it: process (BOOL1, DIN)
  begin
    BIT1 <= BOOL2BIT(BOOL1);
    EVEN_PARITY(DIN, PARITY);
  end process;
  ....
 end;
```

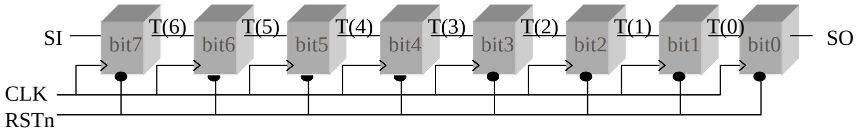
Parameterized hardware

- Conditional component instantiation with **if … generate** construct.
- Iterative component instantiation with **for … generate** construct.
- Parameterized design with **generic** parameters.

Example: 8-bit shift register



entity SHIFT8 **is**
port (RSTn, CLK, SI : **in** std_logic;
    SO : **out** std_logic);
end SHIFT8;




```
architecture RTL1 of SHIFT8 is
    component DFF
    port ( RSTn, CLK, D: in std_logic;
           Q         : out std_logic );
    end component;
    signal T: std_logic_vector(6 downto 0);
begin
    bit7 : DFF
        port map (RSTn => RSTn, CLK => CLK,
                  D => SI, Q => T(6) );
    bit6 : DFF
        port map (RSTn => RSTn, CLK => CLK,
                  D => T(6), Q => T(5) );
    bit5 : DFF
        port map (RSTn, CLK, T(5), T(4) );
    ...
    bit1 : DFF
        port map (RSTn, CLK, T(1), T(0) );
    bit0 : DFF
        port map (RSTn, CLK, T(0), S0 );
end RTL1;
```


Example: 1024-bit shift register

```
architecture RTL2 of SHIFT1024 is
    component DFF
    port ( RSTn, CLK, D: in std_logic;
           Q         : out std_logic );
    end component;
    signal T: std_logic_vector(1022 downto 0);
begin
    g0: for i in 1023 downto 0 generate
        g1: if (i = 1023) generate
            bit1023 : DFF port map (RSTn,CLK,SI,T(1022));
          end generate;
        g2: if (i>0) and (i<1023) generate
            bitm : DFF port map (RSTn,CLK,T(i),T(i-1));
          end generate;
        g3: if (i=0) generate
            bit0 : DFF port map (RSTn,CLK,T(0),S0);
          end generate;
      end generate;
end RTL2;
```

Example: n-bit shift register

```
entity SHIFTn is
generic ( n : positive);
port ( RSTn, CLK, SI : in std_logic;
        SO : out std_logic );
end SHIFTn;
```

```
architecture RTL3 of SHIFTn is
  component DFF
  port ( RSTn, CLK, D: in std_logic;
      Q          : out std_logic );
  end component;
  signal T: std_logic_vector(n-2 downto 0);
begin
  g0: for i in n-1 downto 0 generate
      g1: if (i = n-1) generate
            bit_high : DFF port map (RSTn,CLK,SI,T(n-2));
          end generate;
      g2: if (i>0) and (i<n-1) generate
            bitm : DFF port map (RSTn,CLK,T(i),T(i-1));
          end generate;
      g3: if (i=0) generate
            bit0 : DFF port map (RSTn,CLK,T(0),S0);
          end generate;
    end generate;
end RTL3;
```


Example: n-bit shift register

- Component instantiation

```
…
  component SHIFTn is
  generic ( n : positive);
  port ( RSTn, CLK, SI : in std_logic;
       SO : out std_logic );
  end component;
```

```
…
begin
  …
  Shift32comp : SHIFTn
    generic map (n => 32)
    port map(RSTn => …,
             CLK => …,
             SI => …,
             SO => …);
  …
end;
```

Recursive descriptions

- If parametrized hardware is described recursively, then
  - **generic**-parameters,
  - **if … generate**-constructs for conditional component instantiation and
  - recursive component instantiation are used.
- Example: Conditional Sum Adder

Conditional Sum Adder

- A conditional sum adder CSA_n computes both sum and sum + 1 of two operand, i.e., it implements a Boolean function

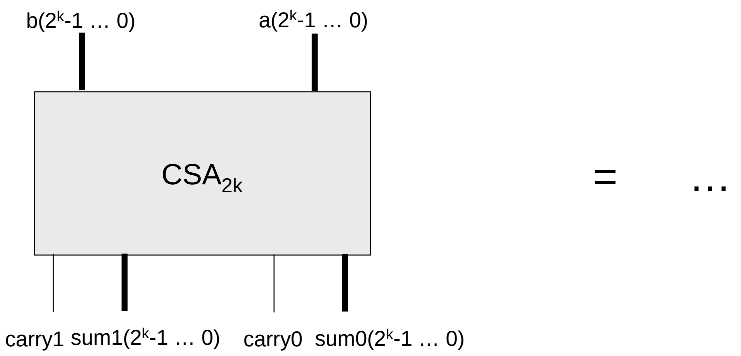
  $+_n : \mathbf{B}^{2n} \rightarrow \mathbf{B}^{2n+2}$,

  $(a_{n-1}, ..., a_0 , b_{n-1}, ..., b_0) \rightarrow$
  $(carry1, sum1_{n-1}, \dots, sum1_0, carry0, sum0_{n-1}, ..., sum0_0)$ with

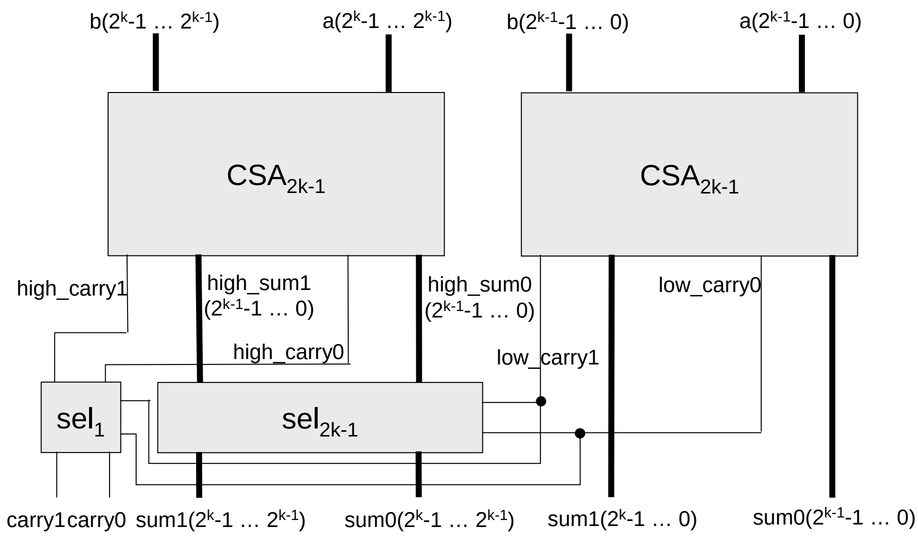
  $<carry0, sum0_n ... sum0_0> = <a_{n-1} ... a_0> + <b_{n-1} ... b_0>$

  $<carry1, sum1_n ... sum1_0> = <a_{n-1} ... a_0> + <b_{n-1} ... b_0>+1$.
- It can be realized by
  - Two conditional sum adders $CSA_{n/2}$
  - One n/2-bit select circuit $sel_{n/2}$
  - One 1-bit select circuit sel_1
- Let $n = 2^k$.

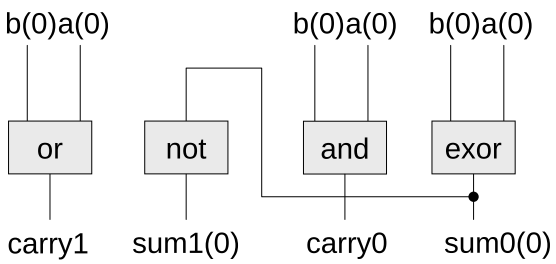
Conditional Sum Adder – recursive definition



Conditional Sum Adder – recursive definition

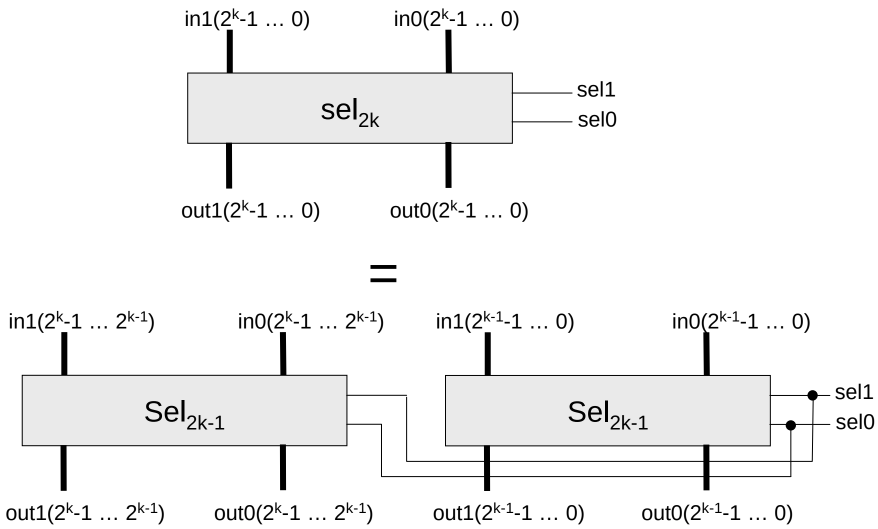


CSA_1



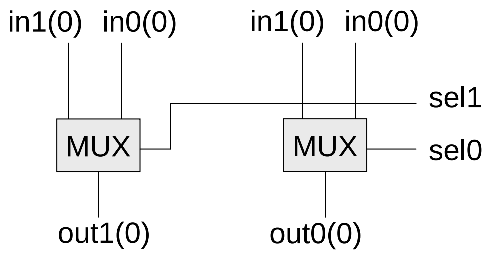
b(0)a(0)
b(0)a(0)
b(0)a(0)
or
not
and
exor
carry1
sum1(0)
carry0
sum0(0)

sel_{2k}

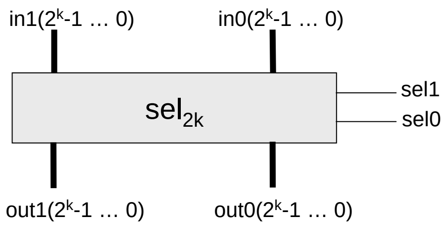


in1(2k-1 … 0)
in0(2k-1 … 0)
sel2k
sel1
sel0
out1(2k-1 … 0)
out0(2k-1 … 0)
=
in1(2k-1 … 2k-1)
in0(2k-1 … 2k-1)
in1(2k-1-1 … 0)
in0(2k-1-1 … 0)
Sel2k-1
Sel2k-1
sel1
sel0
out1(2k-1 … 2k-1)
out0(2k-1 … 2k-1)
out1(2k-1-1 … 0)
out0(2k-1-1 … 0)

sel_1



Recursive description of sel_{2k}



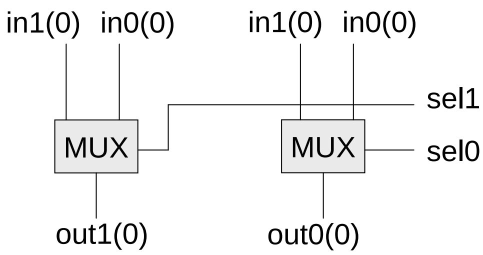

```
ENTITY select_2_power_k IS
  GENERIC(k : natural);
  PORT(in0  : IN std_logic_vector((2**k)-1 DOWNTO 0);
       in1  : IN std_logic_vector((2**k)-1 DOWNTO 0);
       sel0 : IN std_logic;
       sel1 : IN std_logic;
       out0 : OUT std_logic_vector((2**k)-1 DOWNTO 0);
       out1 : OUT std_logic_vector((2**k)-1 DOWNTO 0));
END select_2_power_k ;
```

```
ARCHITECTURE netlist OF select_2_power_k IS

COMPONENT mux
  PORT (m1, m0, sel : IN std_logic; res : OUT std_logic);
END COMPONENT;

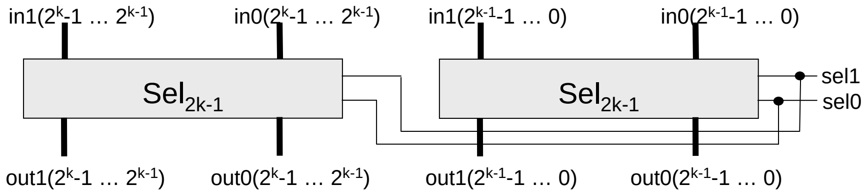
COMPONENT select_2_power_k
  GENERIC(k : natural);
  PORT(in0 : IN std_logic_vector(2**k-1 DOWNTO 0);
       in1  : IN std_logic_vector(2**k-1 DOWNTO 0);
       sel0 : IN std_logic;
       sel1 : IN std_logic;
       out0 : OUT std_logic_vector(2**k-1 DOWNTO 0);
       out1 : OUT std_logic_vector(2**k-1 DOWNTO 0));
END COMPONENT;

...
```

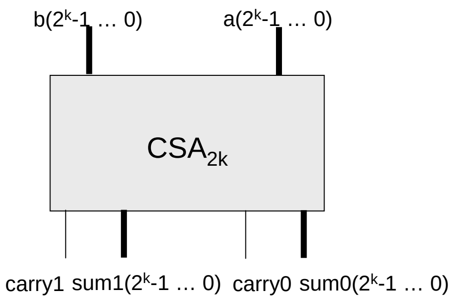

```
…
BEGIN
  basisblock: IF k = 0 GENERATE
  -- Erzeuge sel_1
    mux1 : mux
      PORT MAP(in1(0), in0(0), sel1, out1(0));
    mux0 : mux
      PORT MAP(in1(0), in0(0), sel0, out0(0));
  END GENERATE;

  …
```




```
…
recursion:  IF k > 0 GENERATE
  sel_high : select_2_power_k
    GENERIC MAP(k => k-1)
    PORT MAP (in0 => in0(2**k-1 DOWNTO 2**(k-1)),
              in1 => in1(2**k-1 DOWNTO 2**(k-1)),
              sel0 => sel0, sel1 => sel1,
              out0 => out0(2**k-1 DOWNTO 2**(k-1)),
              out1 => out1(2**k-1 DOWNTO 2**(k-1)));
  sel_low : select_2_power_k
    GENERIC MAP(k => k-1)
    PORT MAP (in0 => in0(2**(k-1)-1 DOWNTO 0),
              in1 => in1(2**(k-1)-1 DOWNTO 0),
              sel0 => sel0, sel1 => sel1,
              out0 => out0(2**(k-1)-1 DOWNTO 0),
              out1 => out1(2**(k-1)-1 DOWNTO 0));
END GENERATE;
END netlist;
```


Recursive description of CSA_{2k}




```
ENTITY csa_2_power_k IS
  GENERIC (k : natural);
  PORT (a   : IN std_logic_vector(2**k-1 DOWNTO 0);
        b   : IN std_logic_vector(2**k-1 DOWNTO 0);
        sum0 : OUT std_logic_vector(2**k-1 DOWNTO 0);
        carry0 : OUT std_logic;
        sum1 : OUT std_logic_vector(2**k-1 DOWNTO 0);
        carry1 : OUT std_logic);
END csa_2_power_k;
```

```
ARCHITECTURE csa_netlist OF csa_2_power_k IS

COMPONENT and2
  PORT (a, b : IN std_logic; y : OUT std_logic);
END COMPONENT;

COMPONENT xor2
  PORT (a, b : IN std_logic; y : OUT std_logic);
END COMPONENT;

COMPONENT or2
  PORT (a, b : IN std_logic; y : OUT std_logic);
END COMPONENT;

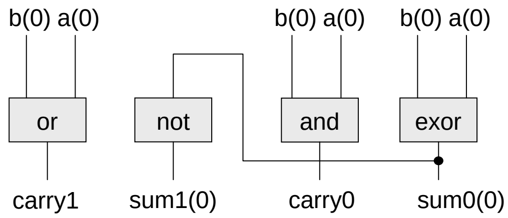
COMPONENT inv
  PORT (a : IN std_logic;  y : OUT std_logic);
END COMPONENT;

...
```



```
...
COMPONENT select_2_power_k
  GENERIC (k : natural);
  PORT(in0  : IN std_logic_vector(2**k-1 DOWNTO 0);
       in1  : IN std_logic_vector(2**k-1 DOWNTO 0);
       sel0 : IN std_logic;
       sel1 : IN std_logic;
       out0 : OUT std_logic_vector(2**k-1 DOWNTO 0);
       out1 : OUT std_logic_vector(2**k-1 DOWNTO 0));
END COMPONENT;

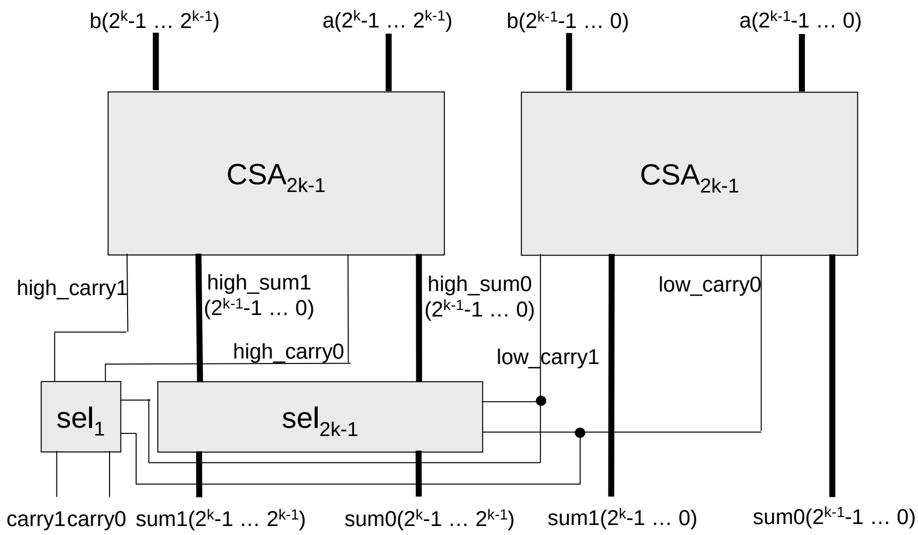
COMPONENT csa_2_power_k
  GENERIC (k : natural);
  PORT(a  : IN std_logic_vector(2**k-1 DOWNTO 0);
       b  : IN std_logic_vector(2**k-1 DOWNTO 0);
       sum0 : OUT std_logic_vector(2**k-1 DOWNTO 0);
       carry0 : OUT std_logic;
       sum1 : OUT std_logic_vector(2**k-1 DOWNTO 0);
       carry1 : OUT std_logic);
END COMPONENT;
...
```




```
...

BEGIN
  one_bit: IF k = 0 GENERATE
    SIGNAL int0 : std_logic;
  BEGIN
    exor_cell : xor2
      PORT MAP(b(0), a(0), int0);
    sum0(0) <= int0;
    inv_cell : inv
      PORT MAP(int0, sum1(0));
    and_cell : and2
      PORT MAP(b(0), a(0), carry0);
    or_cell : or2
      PORT MAP(b(0), a(0), carry1);
  END GENERATE;

...
```

```
...
more_bit:  IF k > 0 GENERATE
    SIGNAL high_sum0 : std_logic_vector(2**(k-1)-1 DOWNTO 0);
    SIGNAL high_sum1 : std_logic_vector(2**(k-1)-1 DOWNTO 0);
    SIGNAL high_carry0 : std_logic_vector(0 DOWNTO 0);
    SIGNAL high_carry1 : std_logic_vector(0 DOWNTO 0);
    SIGNAL carry_out0 : std_logic_vector(0 DOWNTO 0);
    SIGNAL carry_out1 : std_logic_vector(0 DOWNTO 0);
    SIGNAL low_carry0 : std_logic;
    SIGNAL low_carry1 : std_logic;
  BEGIN
    csa_high : csa_2_power_k
      GENERIC MAP(k => k-1)
      PORT MAP(a => a(2**k-1 DOWNTO 2**(k-1)),
               b => b(2**k-1 DOWNTO 2**(k-1)),
               sum0 => high_sum0, carry0 => high_carry0(0),
               sum1 => high_sum1, carry1 => high_carry1(0));
    csa_low : csa_2_power_k
      GENERIC MAP(k => k-1)
      PORT MAP(a => a(2**(k-1)-1 DOWNTO 0),
               b => b(2**(k-1)-1 DOWNTO 0),
               sum0 => sum0(2**(k-1)-1 DOWNTO 0), carry0 => low_carry0,
               sum1 => sum1(2**(k-1)-1 DOWNTO 0), carry1 => low_carry1);
...
```



```
...
    sel_sum : select_2_power_k
      GENERIC MAP(k => k-1)
      PORT MAP(in0 => high_sum0, in1 => high_sum1,
               sel0 => low_carry0, sel1 => low_carry1,
               out0 => sum0(2**k-1 DOWNTO 2**(k-1)),
               out1 => sum1(2**k-1 DOWNTO 2**(k-1)));
    sel_carry : select_2_power_k
      GENERIC MAP (k => 0)
      PORT MAP (in0 => high_carry0, in1 => high_carry1,
               sel0 => low_carry0, sel1 => low_carry1,
               out0 => carry_out0, out1 => carry_out1);
    carry0 <= carry_out0(0);
    carry1 <= carry_out1(0);
  END GENERATE;
END csa_netlist;
```

VHDL: Evaluation

- Hierarchical specification by entities / architectures / components, (procedures and functions)
- no nested processes,
- No specification of non-functional properties,
- No object-orientation,
- Static number of processes,
- Complicated simulation semantics,
- May be too low level for initial, abstract specification of very large systems.
- Mainly used for hardware generation (but not necessarily!).

(Other) Languages and Models

- **UML (Unified Modelling Language) [Rational 1997]**
  "systematic" approach to support the first phases of the design process

  - UML 1.xx not designed for embedded systems
    UML 2.xx supports real-time applications

  - several diagram types included
    9 (UML 1.4)
    13 (UML 2.0)
    in particular variants of
    StateCharts, MSCs, Petri Nets (called acticity diagrams)

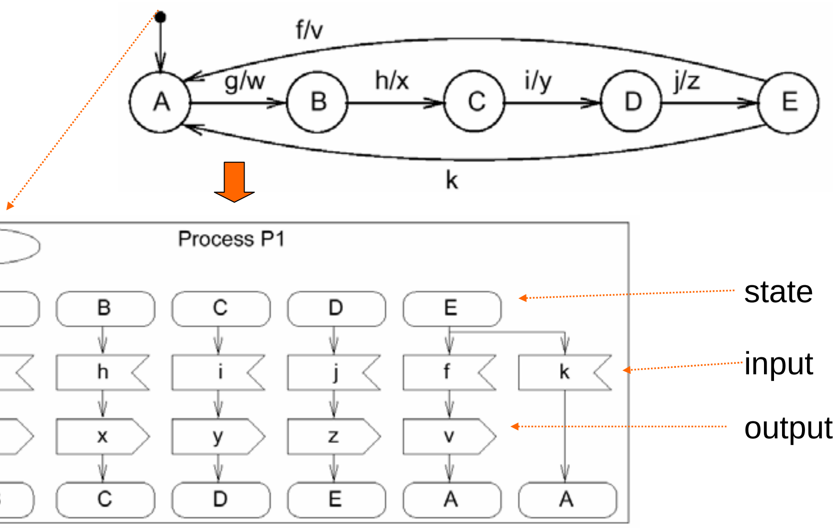
SDL

- Language designed for specification of distributed systems.
- Dates back to early 70s,
- Formal semantics defined in the late 80s,
- Defined by ITU (International Telecommunication Union): Z.100 recommendation in 1980
  Updates in 1984, 1988, 1992, 1996 and 1999

SDL

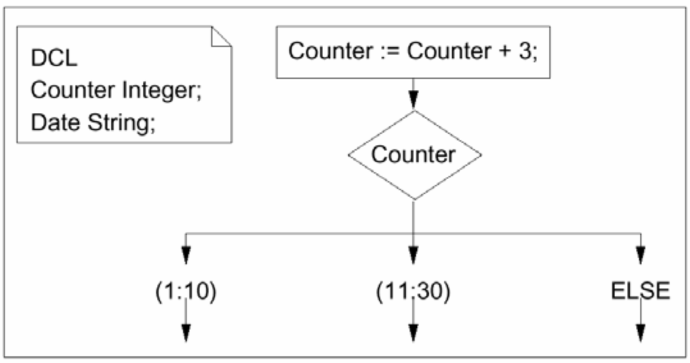
- Provides textual and graphical formats to please all users,
- Just like StateCharts, it is based on the CFSM model of computation; each FSM is called a **process**,
- However, it uses message passing instead of shared memory for communications,
- SDL supports operations on data.

SDL-representation of FSMs/processes



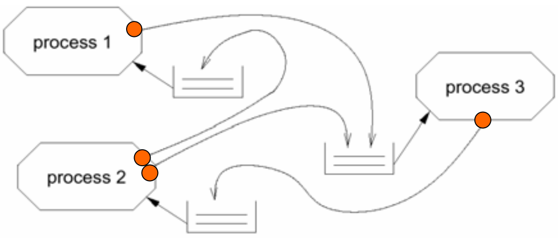
Operations on data

- Variables can be declared locally for processes.
- Their type can be predefined or defined in SDL itself.
- SDL supports abstract data types (ADTs). Examples:



Communication among SDL-FSMs

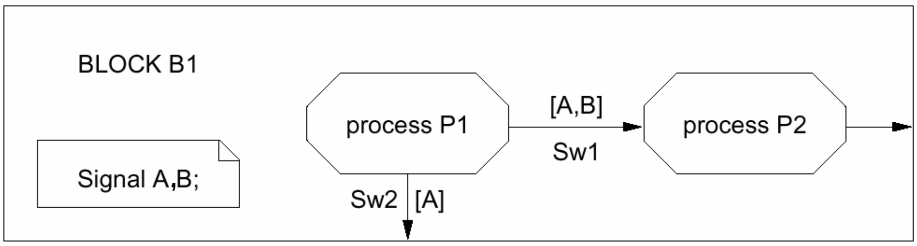
- Communication between FSMs (or „processes“) is based on message-passing, assuming a potentially indefinitely large FIFO-queue.



- Each process fetches next entry from FIFO,
- checks if input enables transition,
- if yes: transition takes place,
- if no: input is ignored (exception: SAVE-mechanism).

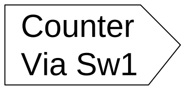
Process interaction diagrams

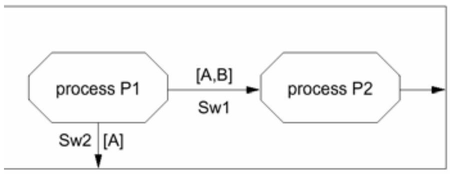
- Interaction between processes can be described in process interaction diagrams (special case of block diagrams).
- In addition to processes, these diagrams contain channels and declarations of local signals.



Designation of recipients

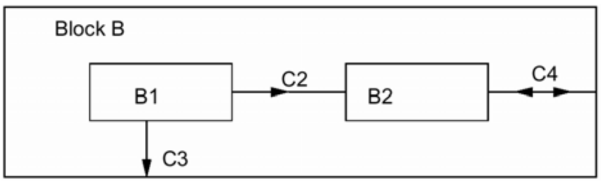
1. **Through process identifiers:** Example: OFFSPRING represents identifiers of processes generated dynamically.
2. **Explicitly:** By including the channel name.
3. **Implicitly:** If signal names imply channel names (B → Sw1)

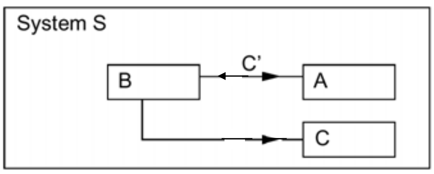


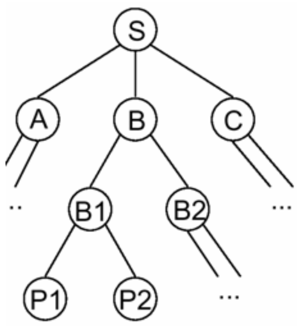


Hierarchy in SDL

- Process interaction diagrams can be included in **blocks.** The root block is called **system.**

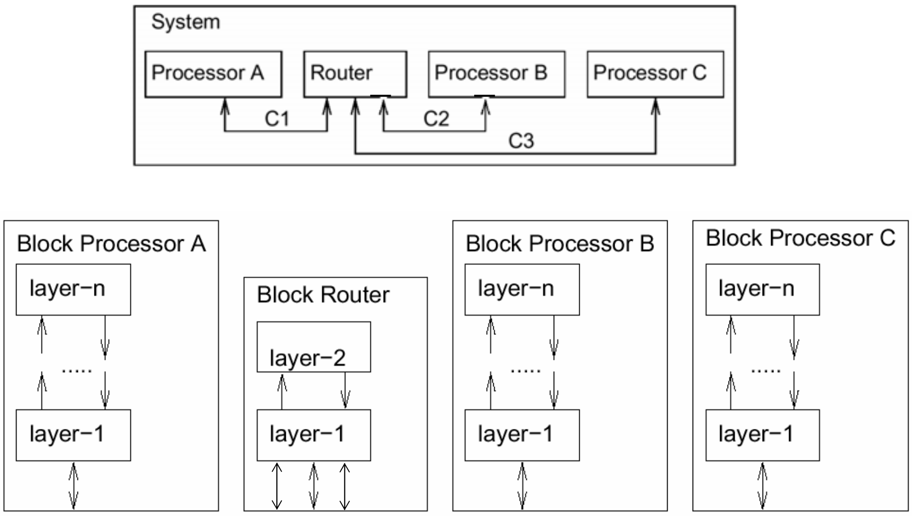






Processes cannot contain other processes, unlike in StateCharts.

Application: description of network protocols



Java

Java 2 Micro Edition (J2ME)

CardJava

Real-time specification for Java (JSR-1), see //www.rtj.org

SystemC

Attempts to describe software and hardware in the same language. Easier said than implemented.
Various C dialects used for hardware description.

Verilog

- HW description language competing with VHDL
- More popular in the US (VHDL common in Europe)

SystemVerilog

- Additional language elements for modeling behavior

SpecC [Gajski, Dömer et. al. 2000]

- SpecC is based on the clear separation between communication and computation. Enables „*plug-and-play*“ for system components; models systems as hierarchical networks of behaviors communicating through channels

Many other languages

- **Pearl:** Designed in Germany for process control applications. Dating back to the 70s. Popular in Europe.
- **Chill**: Designed for telephone exchange stations. Based on PASCAL.
- **IEC 60848, STEP 7**:
  Process control languages using graphical elements

Other languages (2)

- **LOTOS, Z**: Algebraic specification languages
- **Silage**: functional language for digital signal processing.
- **Rosetta**: Efforts on new system design language
- **Esterel:** reactive language; synchronous;
  all reactions are assumed to be in 0 time;
  communication based on ("instantaneous") broadcast;
  //www.esterel-technologies.com

Language Comparison

Language	Behavioral Hierarchy	Structural Hierarchy	Programming Language Elements	Exceptions Supported	Dynamic Process Creation
StateCharts	+	-	-	+	-
VHDL	+	+	+	-	-
SpecCharts	+	-	+	+	-
SDL	+-	+-	+-	-	+
Petri nets	-	-	-	-	+
Java	+	-	+	+	+
SpecC	+	+	+	+	+
SystemC	+	+	+	- (2.0)	- (2.0)
ADA	+	-	+	+	+