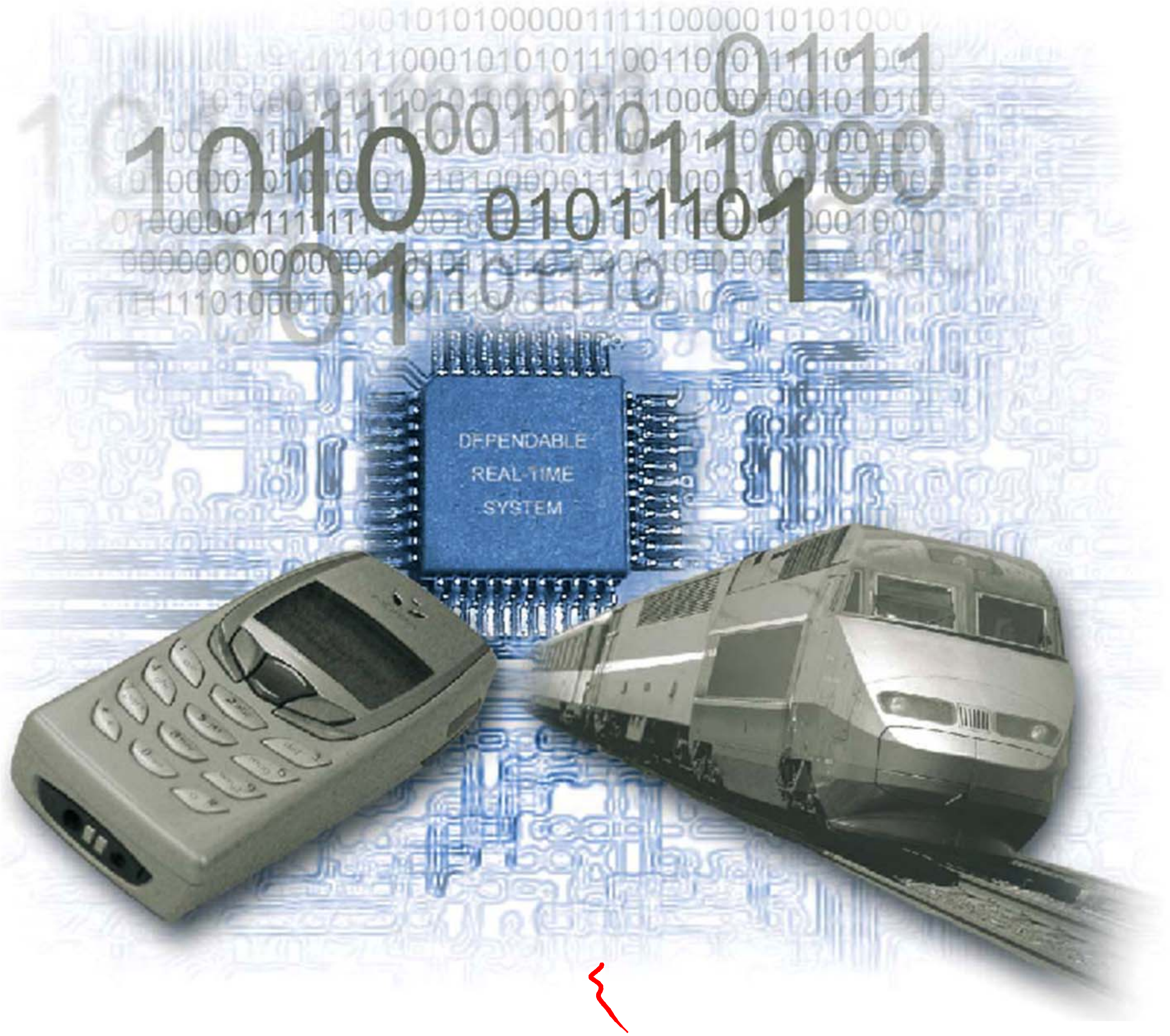
Embedded Systems



SDF Compiler

REVIEW

Task for an SDF compiler:

- Allocation of memory for the passing of data between nodes
- Scheduling of nodes onto processors in such a way that data is available for a block when it is invoked

Assumptions on the SDF graph:

- The SDF graph is nonterminating and does not deadlock
- The SDF graph is connected

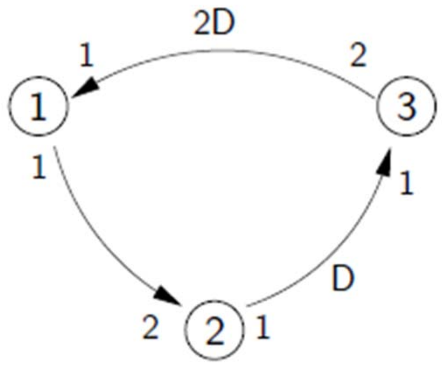
Goal:

- Development of a periodic admissible parallel schedule (PAPS)
- or a periodic admissible sequential schedule (PASS)

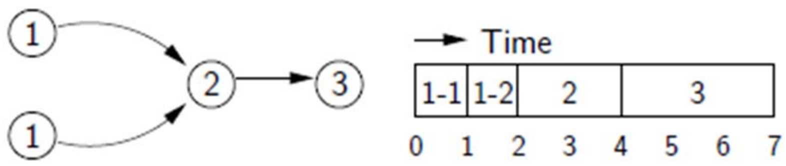
(admissible = correct schedule, finite amount of memory required)

PAPS

REVIEW



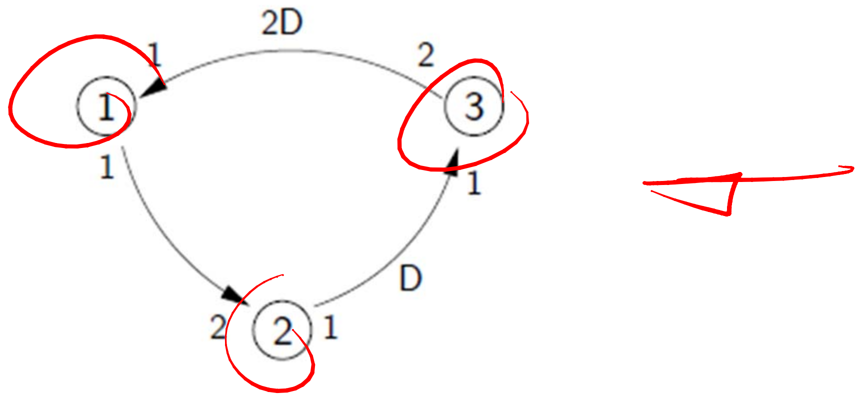
- Assumption: Block 1 : 1 time unit
  Block 2 : 2 time units
  Block 3 : 3 time units



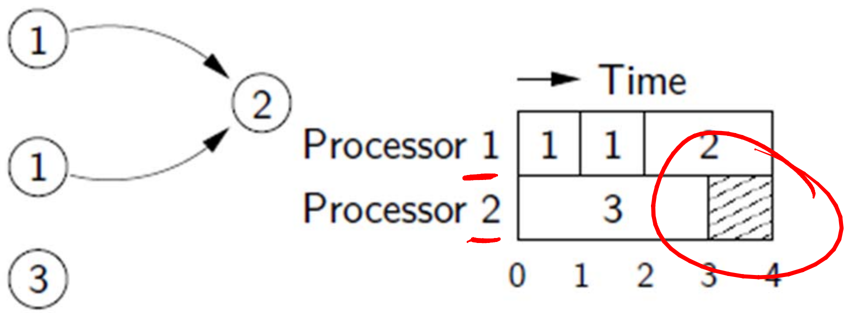
Trivial Case - All computations are scheduled on same processor

PAPS

REVIEW



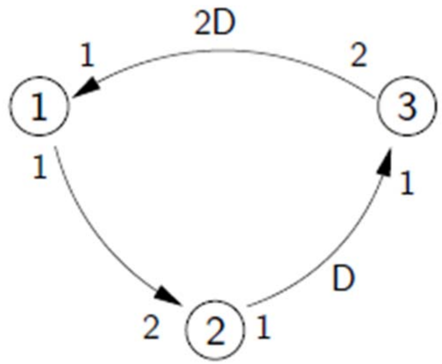
- The performance can be improved, if a schedule is constructed that exploits the potential parallelism in the SDF-graph. Here the schedule covers one single period.



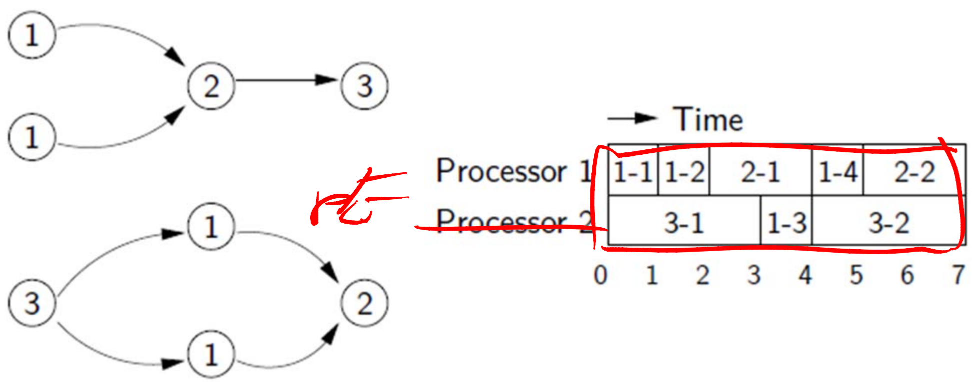
Single Period Schedule

PAPS

REVIEW



- The performance can be further improved, if the schedule is constructed over two periods.



Double Period Schedule

Scheduling Choices

REVIEW

- SDF Scheduling Theorem guarantees a schedule will be found if it exists
- Systems often have many possible schedules
- How can we use this flexibility?
  - Reduced code size
  - Reduced buffer sizes

Looped Code Generation

REVIEW

- Obvious improvement: use loops
- Rewrite the schedule in "looped" form:

  (3 B) C (4 D) (2 A)

- Generated code becomes

```
for ( i = 0 ; i < 3; i++) B;
C;
for ( i = 0 ; i < 4 ; i++) D;
for ( i = 0 ; i < 2 ; i++) A;
```

Conclusion SDF

REVIEW

- The SDF model is very useful for regular DSP applications
- Used for: simulation, scheduling, memory allocation, code generation for Digital Signal Processors (HW and SW)
- There is a mathematical framework to calculate a PASS or a PAPS and to determine the maximum size of buffers, if a PASS/PAPS exists
- The work on SDF can be used to derive single and multiple processor implementations

Selected Models of computation

REVIEW

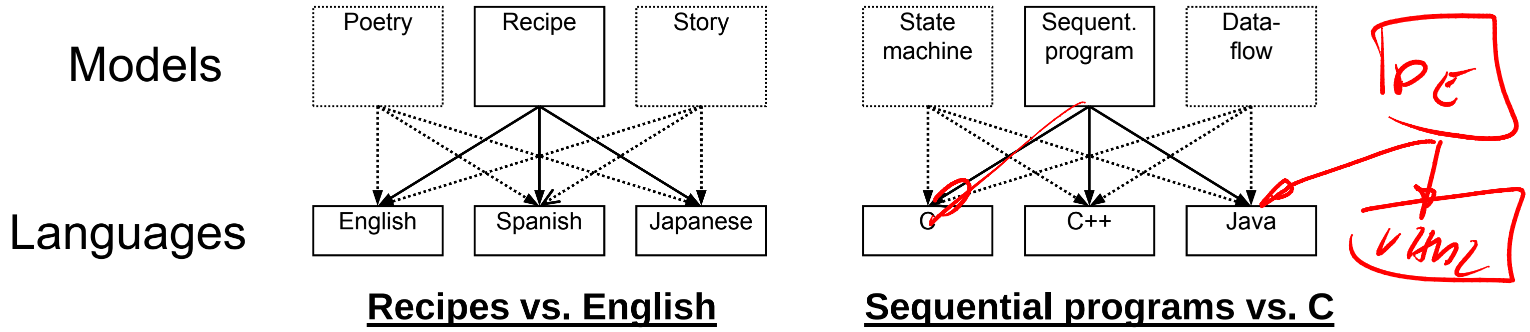
Communication/ local computations	**Shared memory**	**Message passing Synchronous**	**| Asynchronous**
Undefined components	Plain text, use cases	| (Message) sequence charts	
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Imperative (Von Neumann) model	C, C++, Java	C, C++, Java with libraries CSP, ADA	|

* Classification based on the **implementation** of HDLs

Models vs. languages

REVIEW

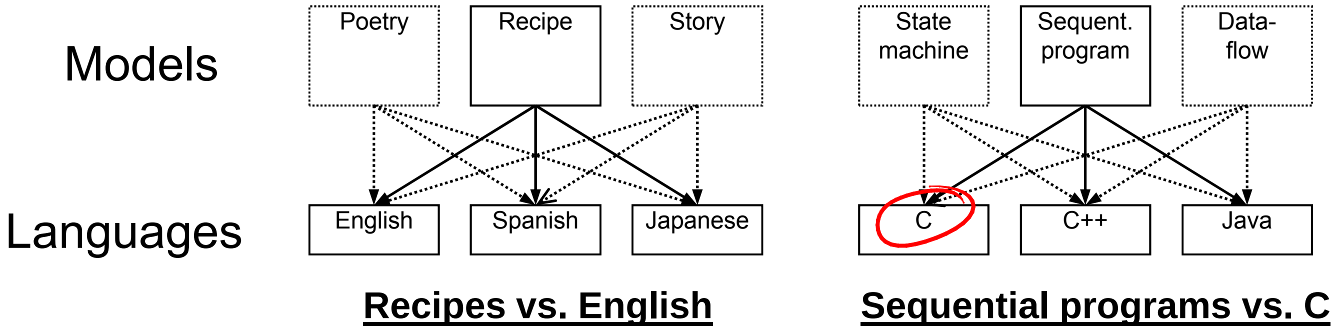
- How can we (precisely) capture behavior?
  - We may think of languages (C, C++), but *computation model* is the key



- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C

Models vs. languages

REVIEW

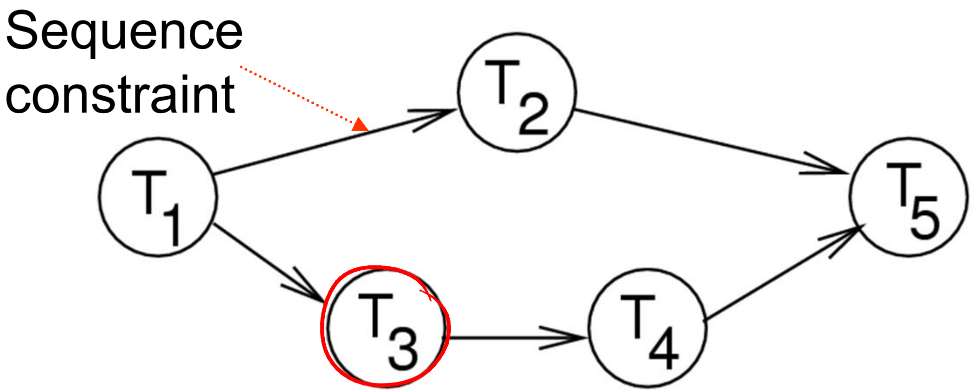


- Variety of languages can capture one model
  - E.g., sequential program model → C,C++, Java
- One language can capture variety of models
  - E.g., C++ → sequential program model, object-oriented model, state machine model
- Certain languages better at capturing certain computation models

Architecture Design – Models

Task graphs or dependency graph (DG)

REVIEW



Nodes are assumed to be a „program" described in some programming language, e.g. C or Java.

- **Def.:** A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.

- If $(v1, v2) \in E$, then $v1$ is called an **immediate predecessor** of $v2$ and $v2$ is called an **immediate successor** of $v1$.

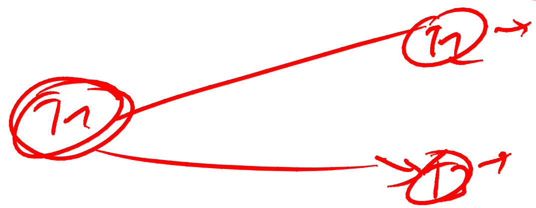
Dependence Graph (DG)

REVIEW

- A dependence graph describes **order relations** for the execution of single operations or tasks. **Nodes** correspond to **tasks or operations, edges** correspond to **relations** („executed after").

- Usually, a dependence graph describes a **partial ordering** between operations and therefore, leaves freedom for scheduling (parallel or sequential). It represents **parallelism** in a program **but no branches** in control flow.

- A dependence graph is acyclic.



- Often, there are additional quantities associated to edges or nodes such as
  - **execution times**, **deadlines**, **arrival times**
  - **communication demand**

Single Assignment Form

REVIEW

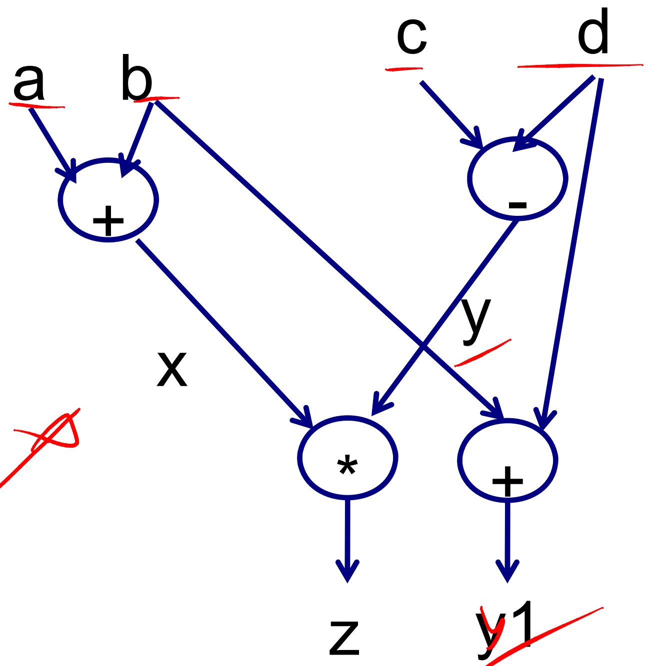
Basic block

```
x = a + b;
y = c - d;
z = x * y;
y = b + d;
```

Single assignment form

```
x = a + b;
y = c - d;
z = x * y;
y1 = b + d;
```

dependence graph



sequential program → optimized hardware

Control-Data Flow Graph (CDFG)

REVIEW

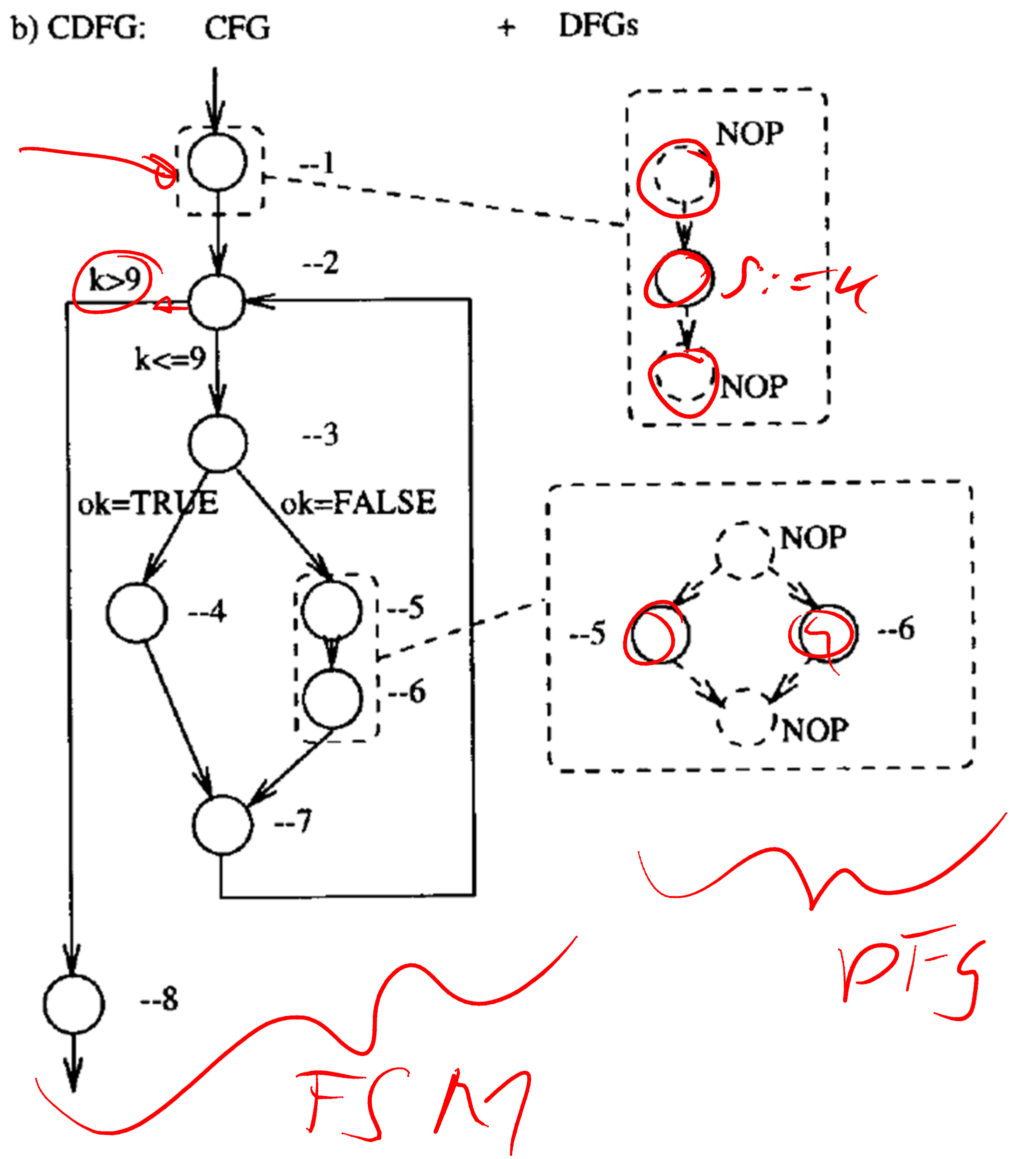
- Goal:
  - Description of control structures (for example branches) and data dependencies.
- Applications:
  - Describing the semantics of programming languages.
  - Internal **representation in compilers** for hardware and software.
- Representation:
  - Combination of control flow (sequential state machine) and dependence representation.
  - Many variants exist.

CDFG

REVIEW

a) VHDL-Code:

```
...
s := k;                  --1
LOOP
   EXIT WHEN k>9;        --2
   IF (ok = TRUE)        --3
      j:=j+1;            --4
   ELSE
      j:= 0;             --5
      ok:= TRUE;         --6
   END IF;
   k:=k+1;               --7
END LOOP;
r := j;                  --8
...
```



Sequence graph

REVIEW

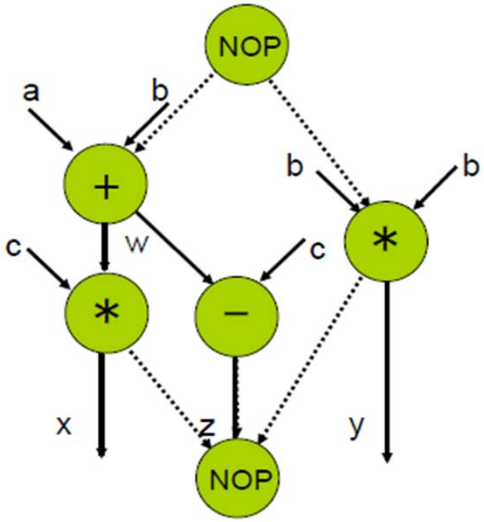
- Hierarchy of chained units
  - units model data flow
  - hierarchy models control flow
- Special nodes
  - start/end nodes: NOP (no operation)
  - branch nodes (BR)
  - iteration nodes (LOOP)
  - module call nodes (CALL)
- Attributes
  - nodes: computation times, cost, ...
  - edges: conditions for branches and iterations

Sequence Graph (SG)

REVIEW

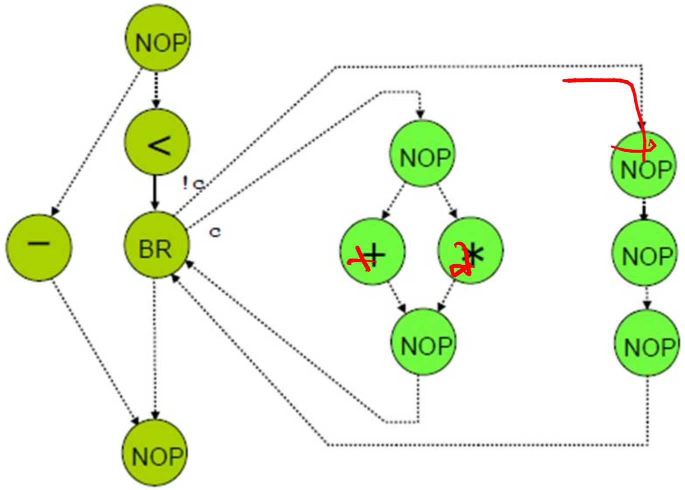
Unit

```
w = a + b;
x = w * c;
y = b * b;
z = w - c;
```



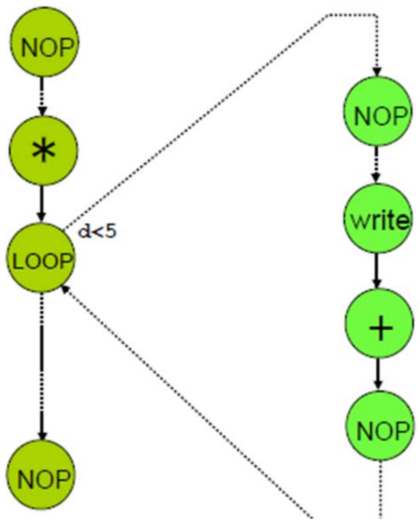
Branch

```
c = a < b;
IF (c) THEN
  p = m + n;
  q = m * n;
ENDIF
x = a - b;
```



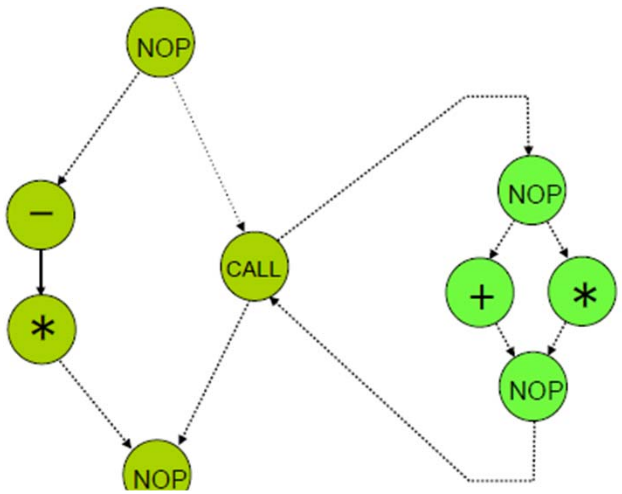
Loop

```
d = 2*x;
WHILE (d<5)DO
  write(d);
  d = d + 1;
ENDWHILE
```



Call

```
d = x - y;
e = d * x;
sub(x, y);
...

PROCEDURE sub (m,n)
  p = m + n;
  q = m * n;
END sub
```



Selected Models of computation

REVIEW

Communication/ local computations	Shared memory	Message passing Synchronous |	Asynchronous
Undefined components	Plain text, use cases	| (Message) sequence charts	
Communicating finite state machines	StateCharts		SDL
Data flow	(Not useful)		Kahn networks, SDF
Petri nets		C/E nets, P/T nets, …	
Discrete event (DE) model	VHDL*, Verilog*, SystemC*, …	Only experimental systems, e.g. distributed DE in Ptolemy	
Von Neumann model	C, C++, Java	C, C++, Java with libraries CSP, ADA |	

* Classification based on implementation

Hardware/System description languages

- VDHL
  - VHDL-AMS
- SystemC
  - TLM

Discrete event semantics

REVIEW

- Basic discrete event (DE) semantics
  - Queue of future actions, sorted by time
  - Loop:
    - Fetch next entry from queue
    - Perform function as listed in entry
      - May include generation of new entries
  - Until termination criterion = true

queue

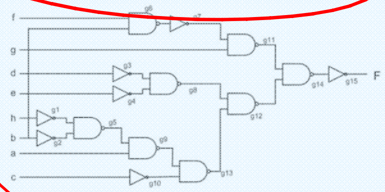
time	5	10	13	15	19
action	a:=5	b:=7	c:=8	a:=6	a:=9

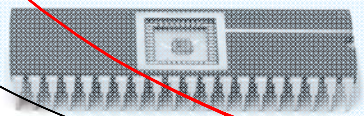
6 a
7 b
8 c

Methods for executing algorithms

REVIEW

Hardware (Application Specific Integrated Circuits)





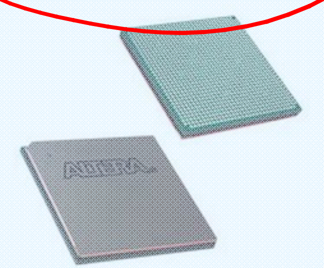
Advantages:

- very high performance and efficient

Disadvantages:

- not flexible (can't be altered after fabrication)
- expensive

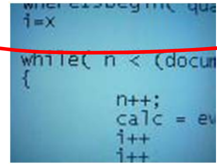
Reconfigurable computing



Advantages:

- fills the gap between hardware and software
- much higher performance than software
- higher level of flexibility than hardware

Software-programmed processors



Advantages:

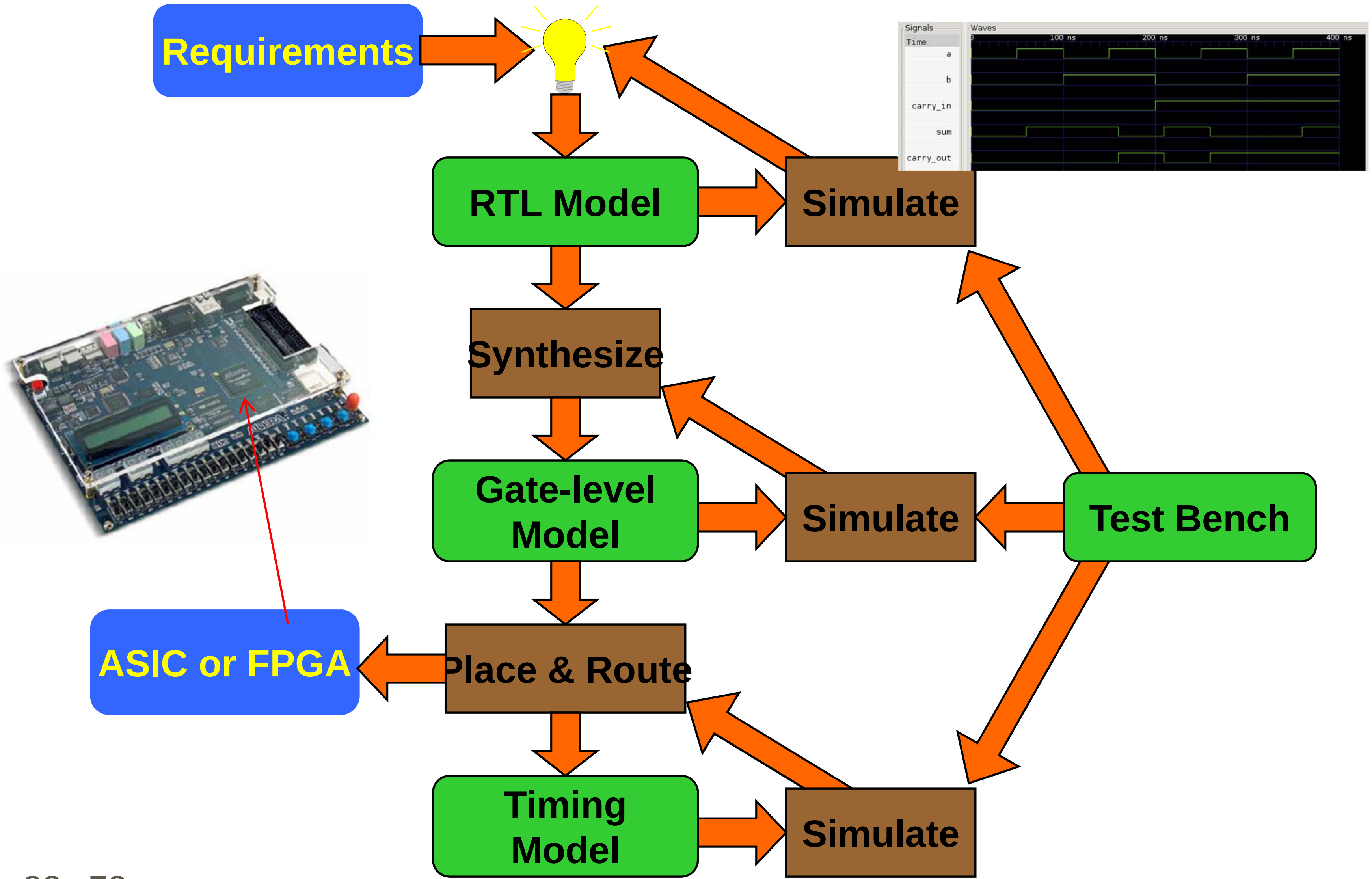
- software is very flexible to change

Disadvantages:

- performance can suffer if clock is not fast
- fixed instruction set by hardware

Basic Design Methodology

REVIEW



HDLs using discrete event (DE) semantics

- Used in hardware description languages (HDLs):
- Description of concurrency is a must for HW description languages!
  - Many HW components are operating concurrently
  - Typically mapped to "processes"
  - These processes communicate via "signals"
  - Examples:
    - MIMOLA [Zimmermann/Marwedel], ~1975
    - …
    - VHDL (very prominent example in DE modeling)
      One of the 3 most important HDLs:
      VHDL, Verilog, SystemC

VHDL

- HDL = hardware description language
- VHDL = VHSIC hardware description language
- VHSIC = very high speed integrated circuit
  - Consortium which developed VHDL (Intermetrics Inc., IBM, Texas Instruments)
  - Early 80's, initiated by US Department of Defense

- Modeling of digital circuits

- 1987 IEEE Standard 1076
- Reviews of standard: 1993, 2000, 2002, 2008

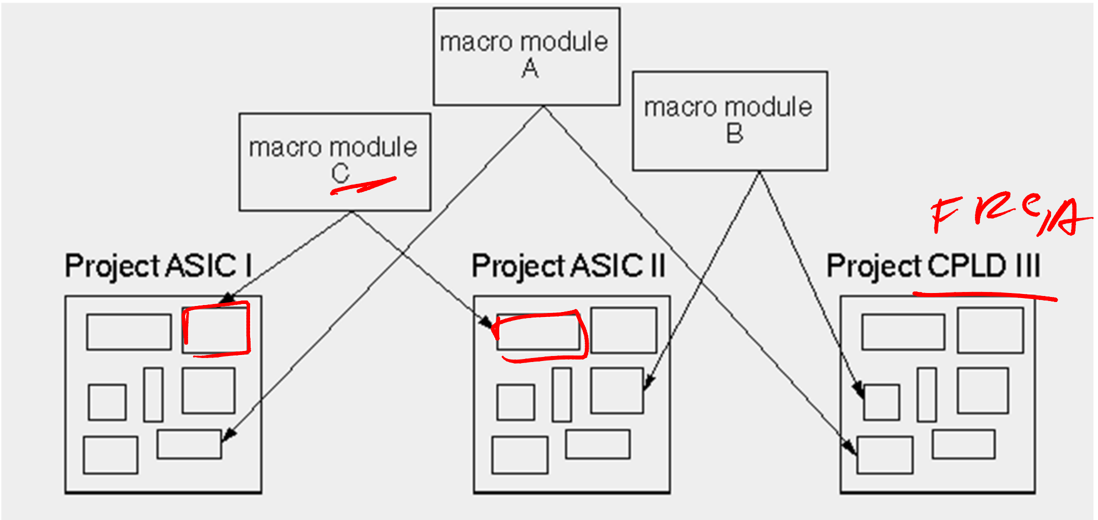
Standard in (European) industry

- Extension: VHDL-AMS, includes analog modeling

VHDL

- Main goal was modeling of digital circuits
  - Modelling at various levels of abstraction
  - Technology-independent

    Re-Usability of specifications



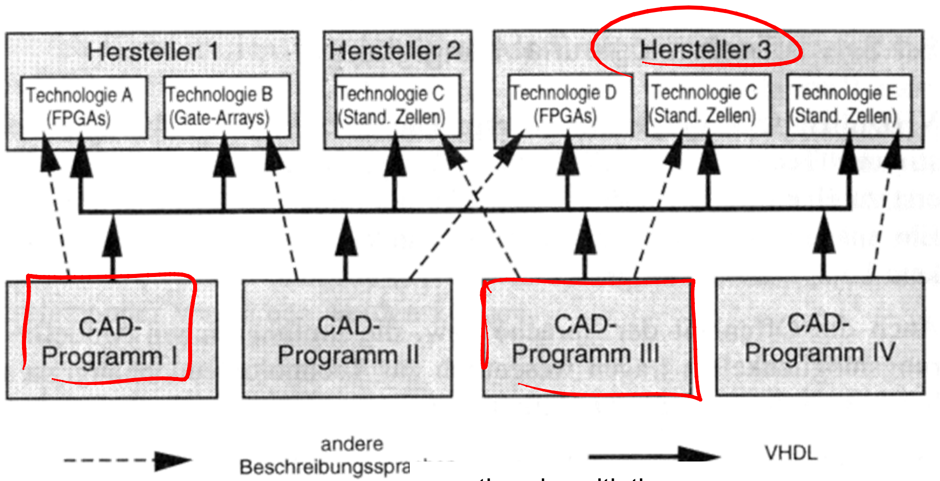
VHDL

- Standard

  Portability (different synthesis and analysis tools possible)

- Validation of designs based on the same description language for different levels of abstraction

- **Powerful** description language



other descritiption language

Hersteller – producer

Technologie – technology

Stand. Zellen – stand. Cells

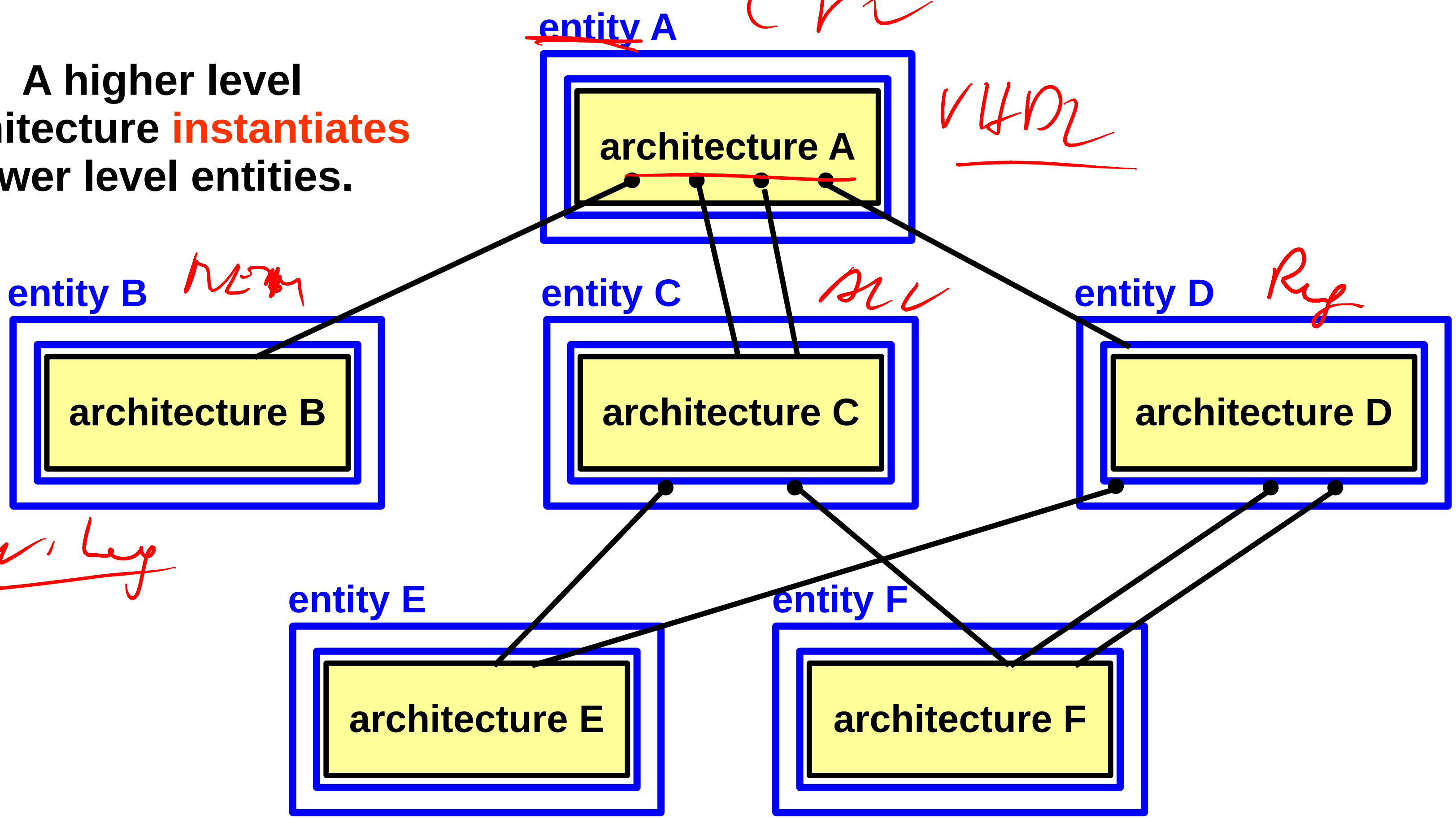
Programm - program

Modeling Digital Systems

- Reasons for modeling
  - requirements specification
  - documentation
  - testing using simulation
  - formal verification
  - synthesis
- Goal
  - most reliable design process, with minimum cost and time
  - avoid design errors!

VHDL Hierarchical Program Structure

A higher level architecture instantiates lower level entities.

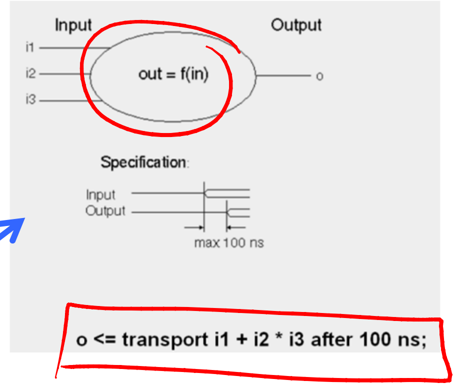


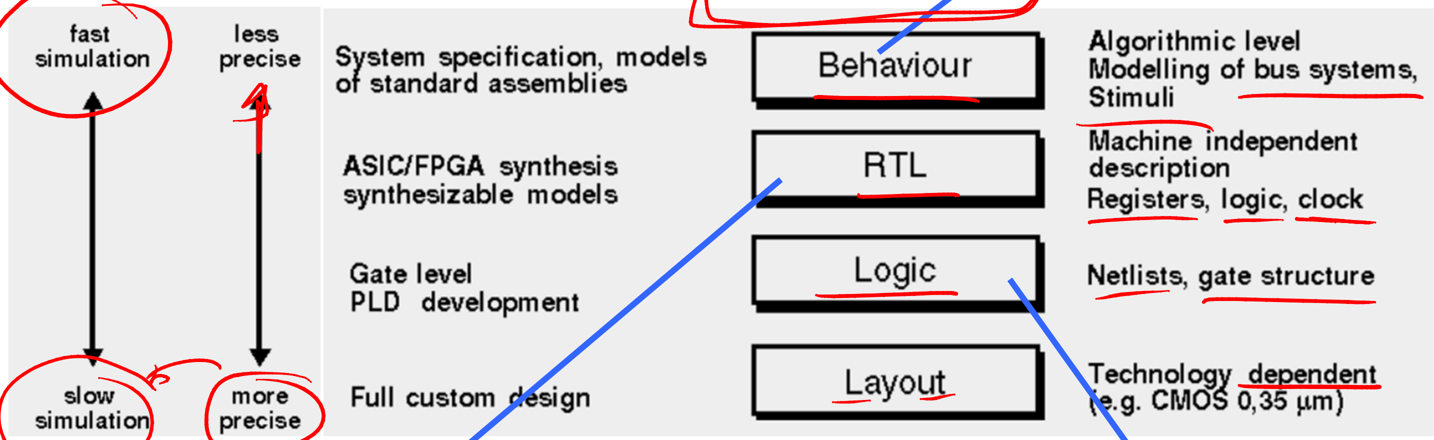
Abstraction

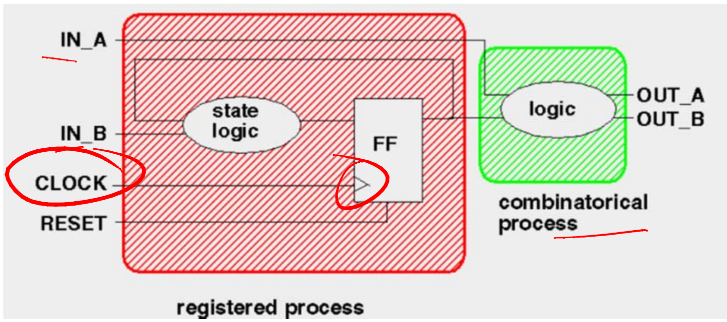
- Abstraction is hiding of details:
  Differentiation between essential and nonessential information

- Creation of abstraction levels:
  On every abstraction level only the essential information is considered, nonessential information is left out

Abstraction Levels







```
U86 : ND2 port map( A => n192, B => n191, Z => n188);
U87 : ND2 port map( A => I3_2, B => I2_0, Z => n175);
U88 : ND2 port map( A => I2_2, B => I3_0, Z => n173);
U89 : NR2 port map( A => mul_36_PROD_not_0,
      B => n174, Z => n185);
U90 : EN port map( A => n181, B => n182, Z => n180);
U91 : ND2 port map( A => I3_2, B => I2_1, Z => n181);
U92 : ND2 port map( A => I2_2, B => I3_1, Z => n182);
U93 : IVP port map( A => n180, Z => n192);
U94 : AO6 port map( A => n173, B => n174, C => n175,
      Z => n172);
U95 : NR2 port map( A => n174, B => n173, Z => n176);
U96 : ND2 port map( A => I3_1, B => I2_1, Z => n174);
U97 : EN port map( A => n183, B => n178,
      Z => product64_4);
U98 : ND3 port map( A => I2_2, B => I3_2, C => n174,
      Z => n183);
```

VHDL

- **Disadvantages:**
  - A change of culture
    - Away from Schematic-based Design
    - towards Language-based Design

"We don't know if to 'harden' a Software engineer or to 'soften' a Hardware engineer",

  - Cost of getting started
    - Selecting and paying for tools

Things to Remember

- **VHDL is a programming language**
  - Many good and bad programs have been (will be) written
  - Contains also many aspects of imperative programming languages

    VHDL is able to describe software, too.

- **Functionality is important BUT not enough!**
  - Style is important (“VHDL cookbook”)
  - Clarity is important

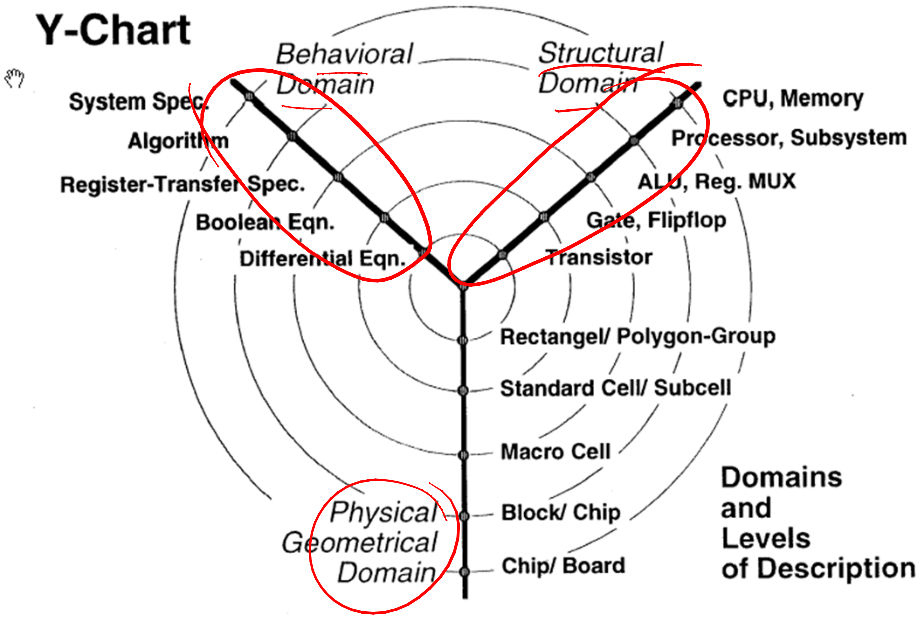
- **Synthesis is hard**

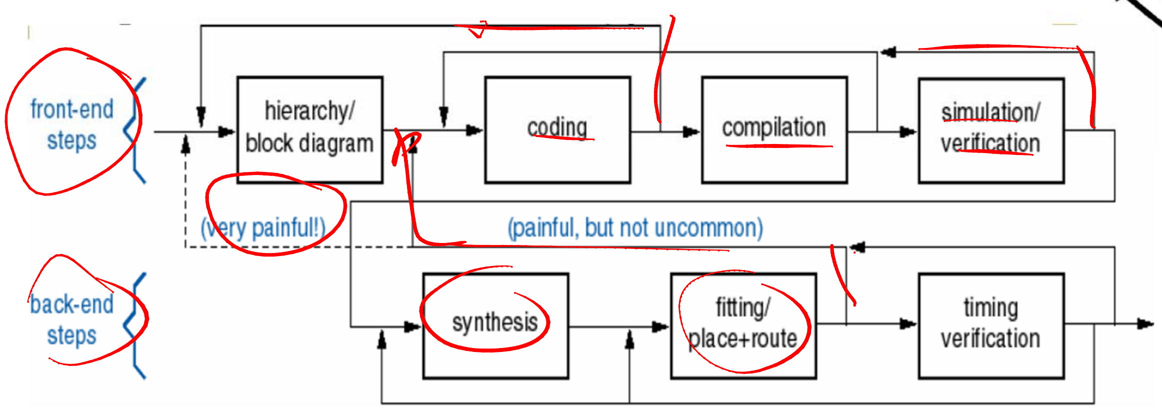
- **Decomposition of a large design into smaller, understandable sub-parts is essential**

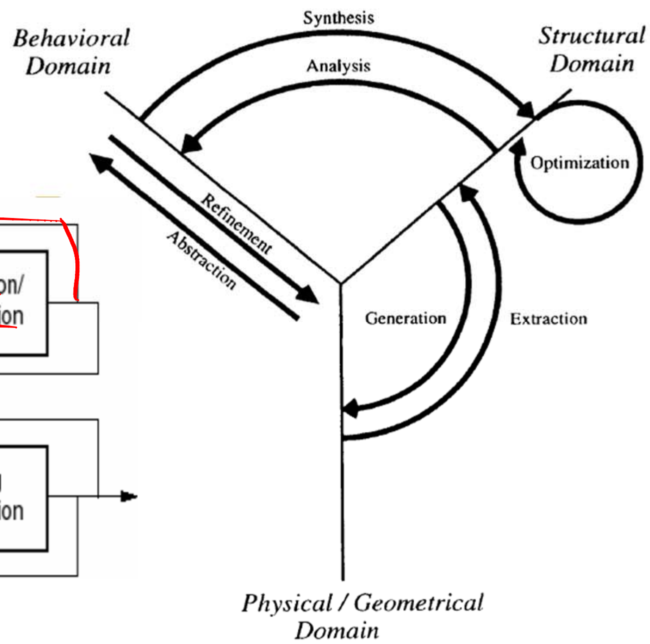
Y-Chart

- 3 design views
  - Behavior (functionality)
  - Structure (netlist)
  - Physical (layout)

- 5 abstraction levels







- Basic VHDL
- Structural VHDL
- Behavioral VHDL
- VHDL-AMS

ES cource: Only some aspects of VHDL, not complete language.

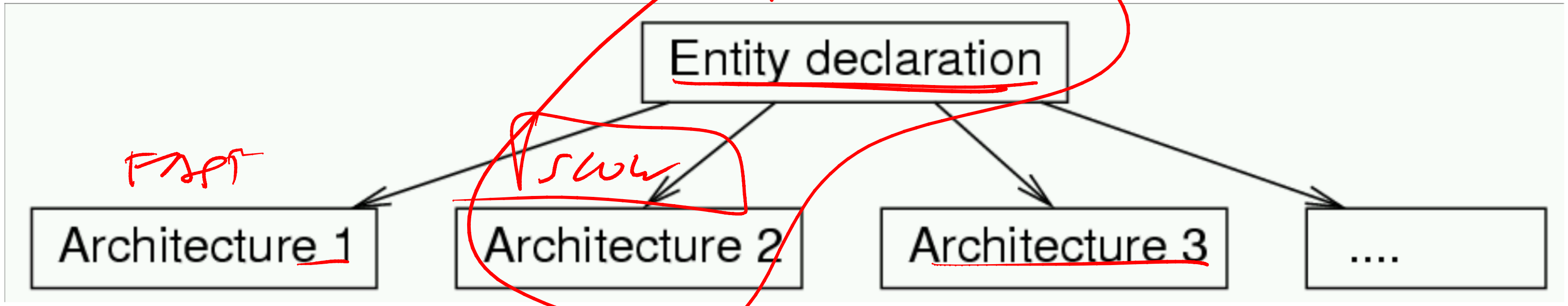
- Basic VHDL

Module Outline

- VHDL Design Example
- VHDL Model Components
  - Entity Declarations
  - Architecture Descriptions
- Basic VHDL Constructs
  - Data types
  - Objects
  - Sequential and concurrent statements
  - Packages and libraries
  - Attributes
  - Predefined operators
- Summary

Entities and architectures

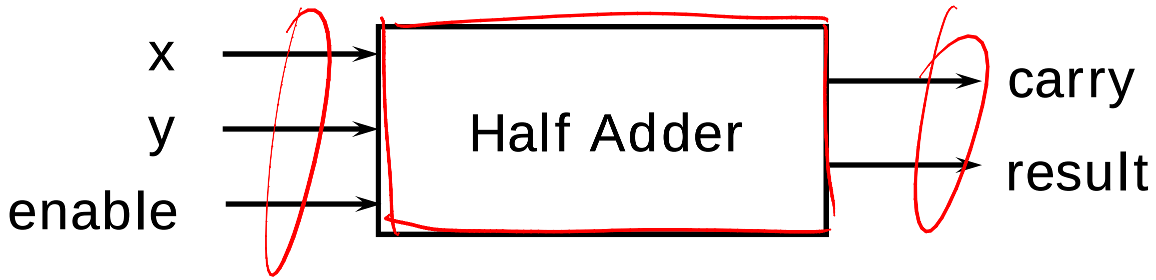
- In VHDL, HW components correspond to "entities"
- Entities comprise processes
- Each design unit is called an **entity**.
- Entities are comprised of **entity declarations** and one or several **architectures.**



Each architecture includes a model of the entity. By default, the most recently analyzed architecture is used. The use of another architecture can be requested in a **configuration**.

VHDL Design Example

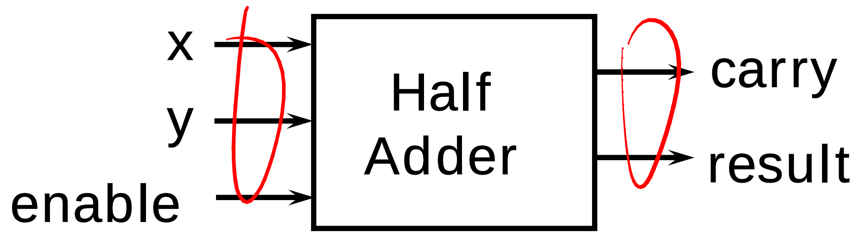
- Problem: Design a single bit half adder with carry and enable
- Specifications
  - Inputs and outputs are each one bit
  - When enable is high, result gets x plus y
  - When enable is high, carry gets any carry of x plus y
  - Outputs are zero when enable input is low



VHDL Design Example
Entity Declaration

- As a first step, the entity declaration describes the interface of the component
  - input and output *ports* are declared

```
ENTITY half_adder IS

    PORT( x, y, enable: IN BIT;
          carry, result: OUT BIT);

END half_adder;
```



VHDL Design Example
Behavioral Specification

- A high level description can be used to describe the function of the adder

```
ARCHITECTURE half_adder_a OF half_adder IS
                BEGIN
            PROCESS (x, y, enable)
                BEGIN
              IF enable = '1' THEN
                  result <= x XOR y;
                  carry <= x AND y;
                ELSE
                    carry <= '0';
                    result <= '0';
                END IF;
              END PROCESS;
          END half_adder_a;
```

The model can then be simulated to verify correct functionality of the component

VHDL Design Example
Data Flow Specification

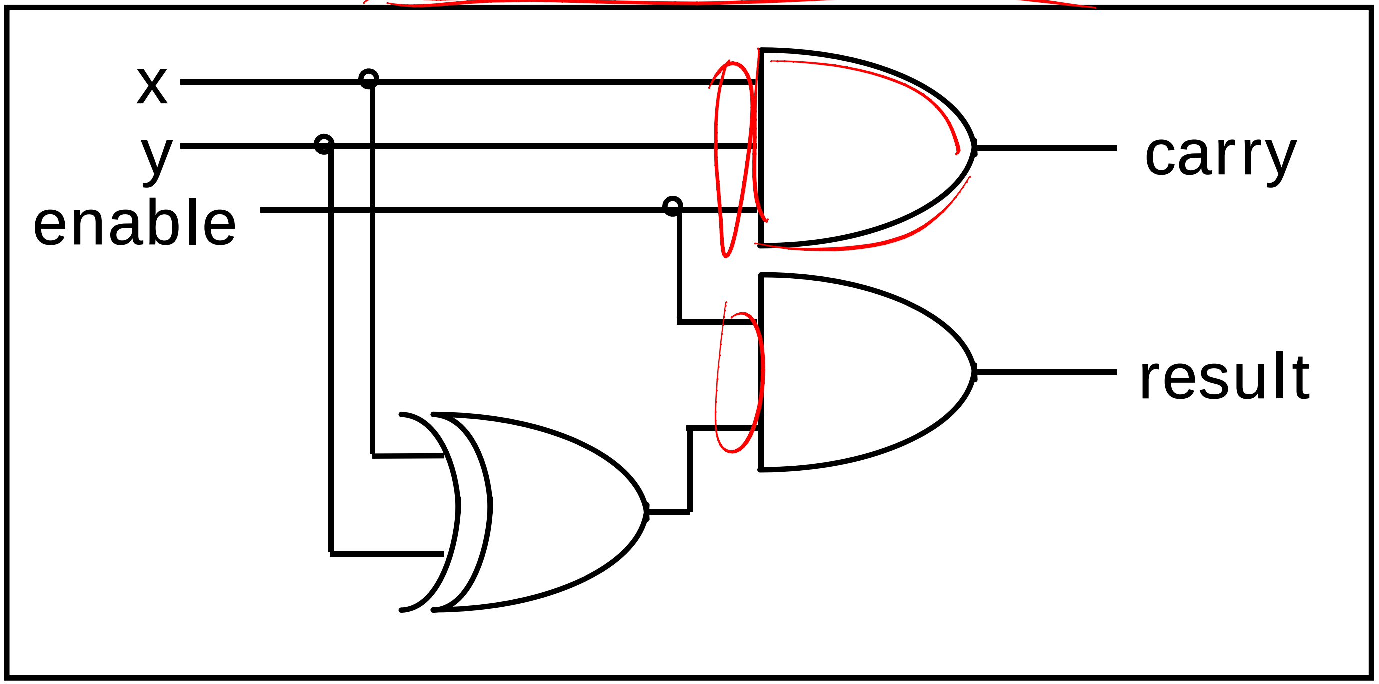
- A second method is to use logic equations to develop a data flow description

```
ARCHITECTURE half_adder_b OF half_adder IS
    BEGIN
      carry <= enable AND (x AND y);
      result <= enable AND (x XOR y);
    END half_adder_b;
```

Again, the model can be simulated at this level to confirm the logic equations

VHDL Design Example
Structural Specification

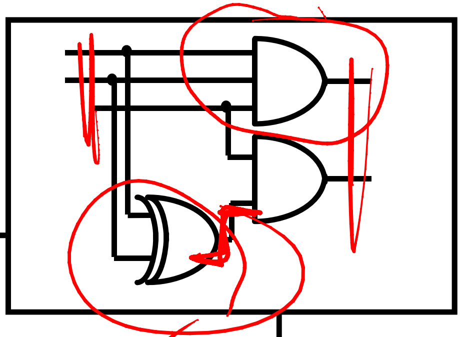
- As a third method, a structural description can be created from predescribed components



These gates can be pulled from a library of parts

VHDL Design Example

Structural Specification (Cont.)



```
ARCHITECTURE half_adder_c OF half_adder IS

    COMPONENT and2
      PORT (in0, in1 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    COMPONENT and3
      PORT (in0, in1, in2 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    COMPONENT xor2
      PORT (in0, in1 : IN BIT;
            out0 : OUT BIT);
    END COMPONENT;

    FOR ALL : and2 USE ENTITY gate_lib.and2_Nty(and2_a);
    FOR ALL : and3 USE ENTITY gate_lib.and3_Nty(and3_a);
    FOR ALL : xor2 USE ENTITY gate_lib.xor2_Nty(xor2_a);

-- description is continued on next slide
```

A number of locally defined idealized *components* are declared

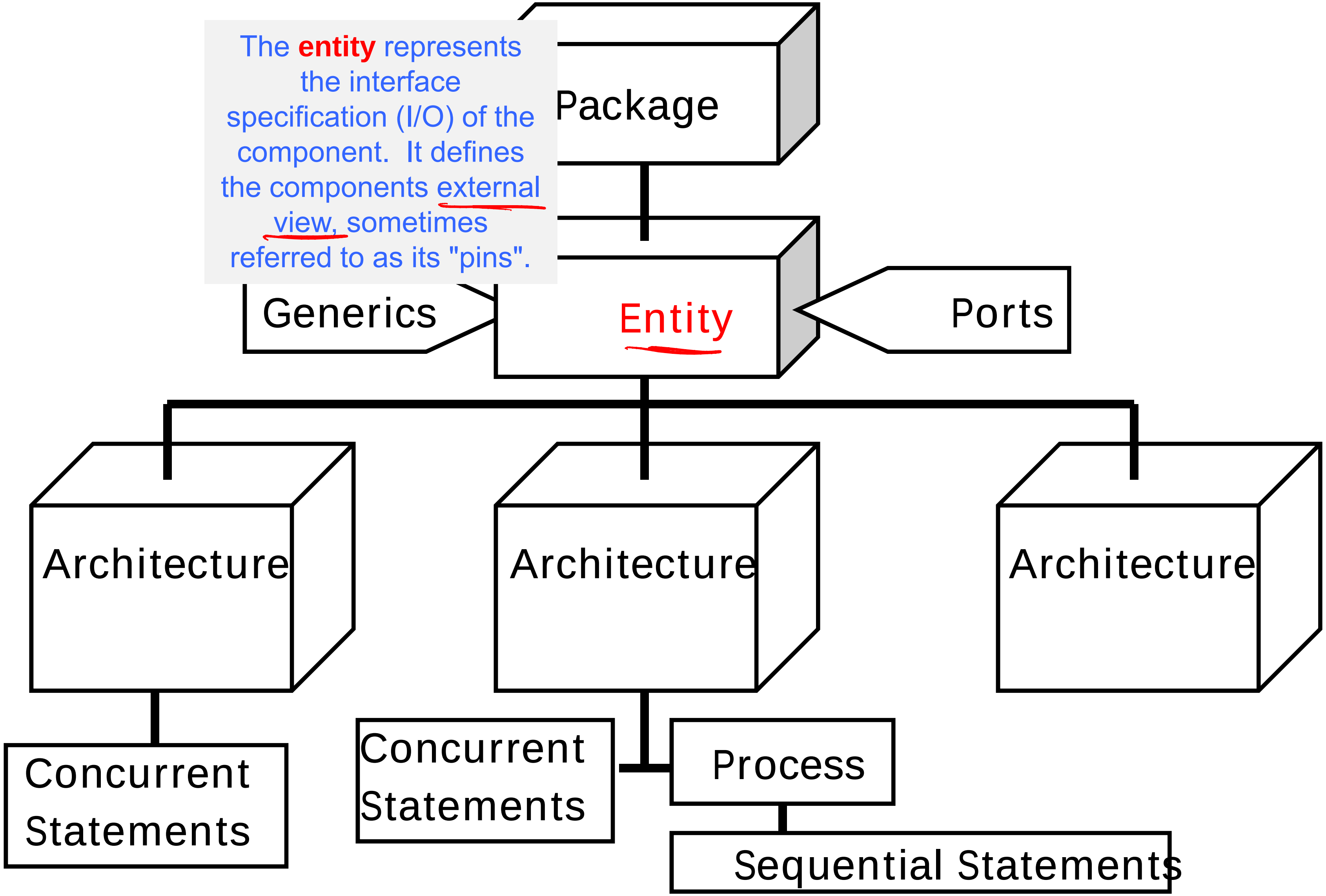
These components are then *bound* to VHDL entities found a library called *gate_lib*

VHDL Design Example
Structural Specification (cont.)

```
-- continuing half_adder_c description

   SIGNAL xor_res : BIT; -- internal signal
   -- Note that other signals are already declared in entity

   BEGIN

      A0 : and2 PORT MAP (enable, xor_res, result);
      A1 : and3 PORT MAP (x, y, enable, carry);
      X0 : xor2 PORT MAP (x, y, xor_res);

   END half_adder_c;
```

body of the architecture shows the component *instantiations* and how they are interconnected to each other and the outside world via the attaching of signals in their PORT MAPs

Putting It All Together



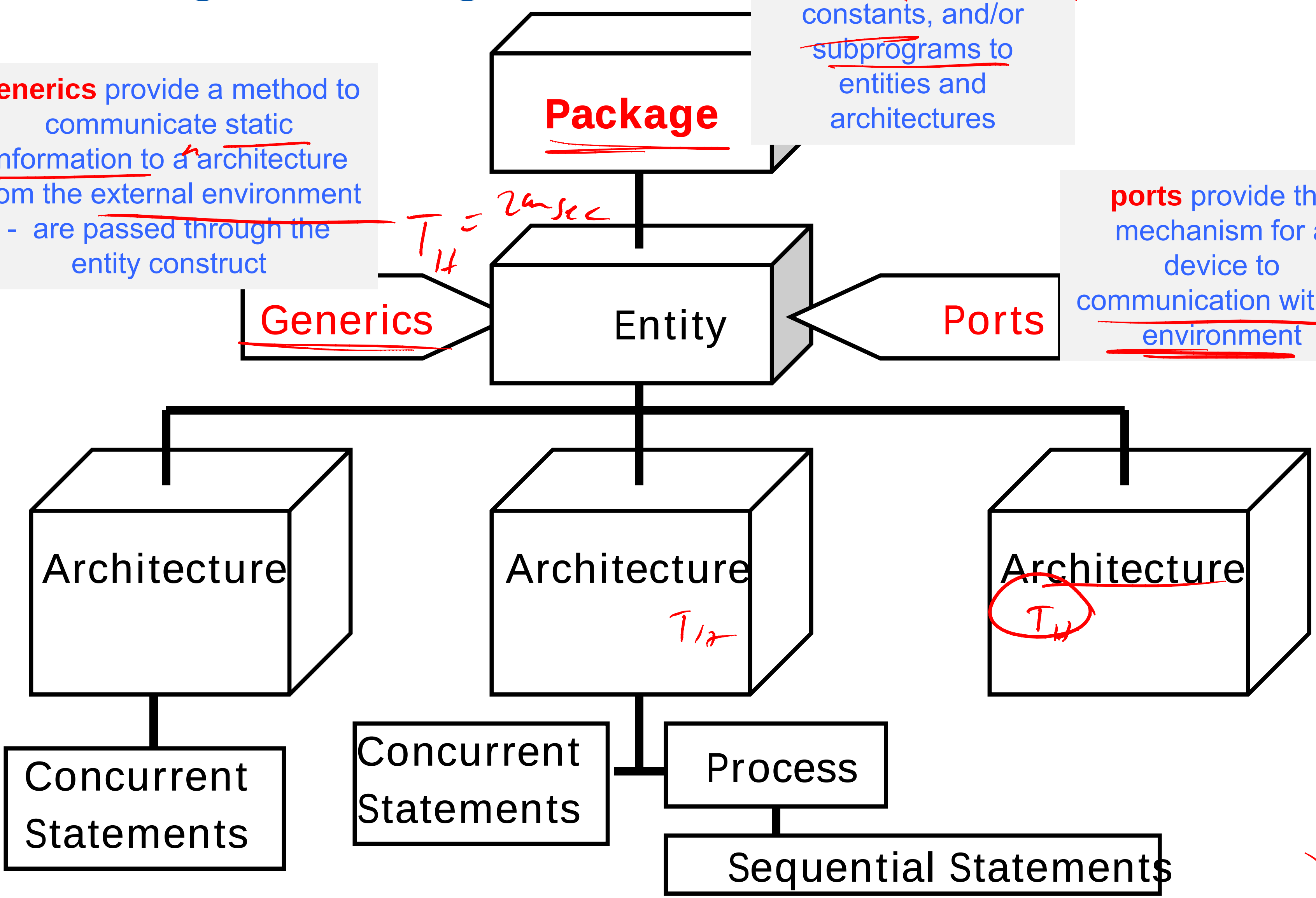
Putting It All Together

packages are used to provide a collection of common declarations, constants, and/or subprograms to entities and architectures

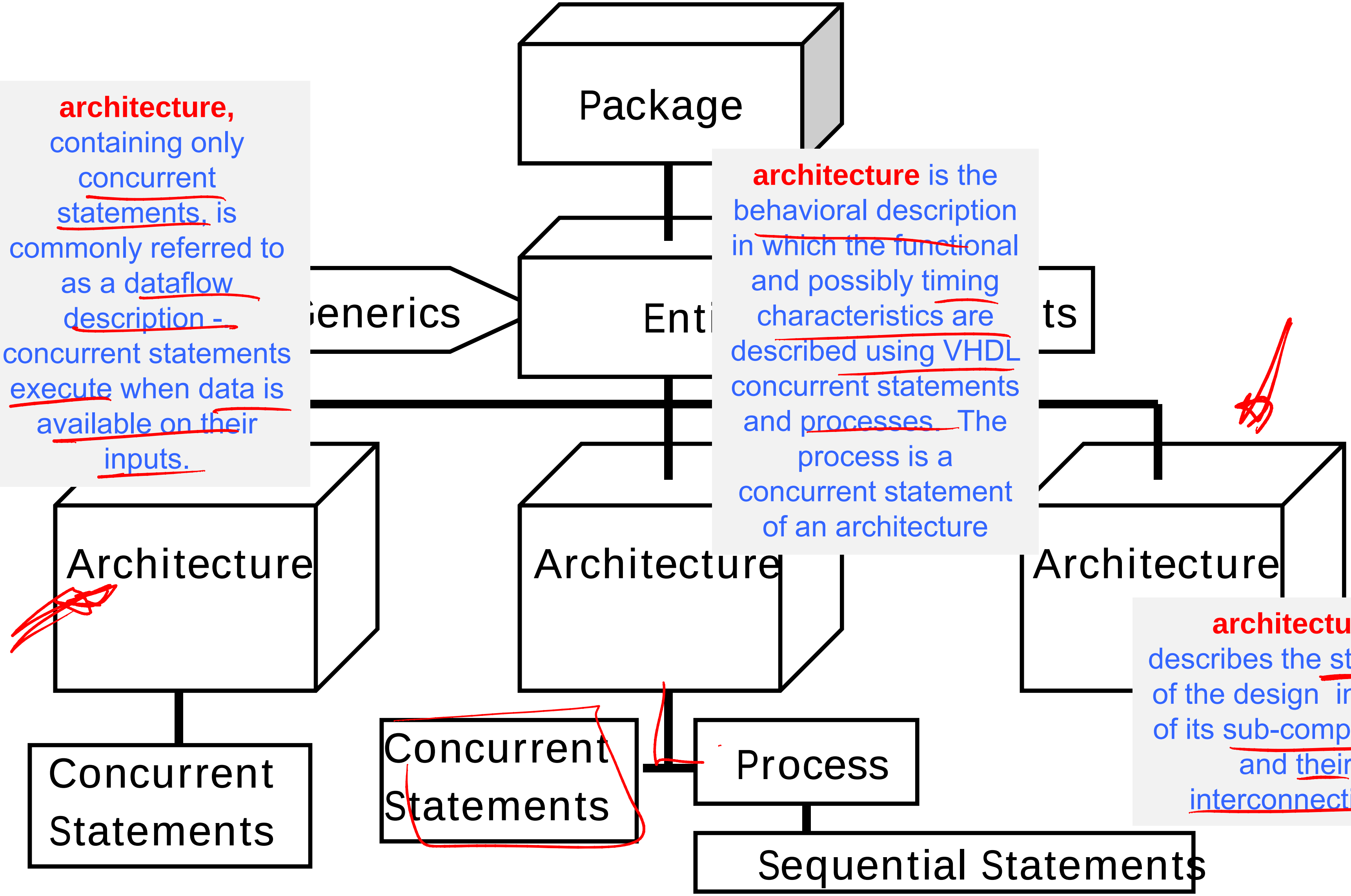
generics provide a method to communicate static information to a architecture from the external environment

- are passed through the entity construct

ports provide the mechanism for a device to communication with its environment



Putting It All Together



architecture, containing only concurrent statements, is commonly referred to as a dataflow description - concurrent statements execute when data is available on their inputs.

architecture is the behavioral description in which the functional and possibly timing characteristics are described using VHDL concurrent statements and processes. The process is a concurrent statement of an architecture

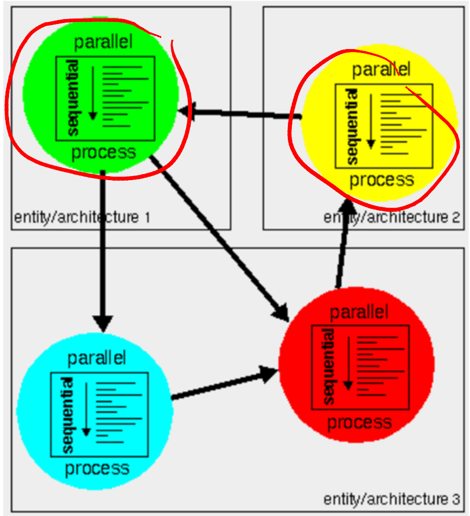
architecture describes the structure of the design in terms of its sub-components and their interconnections

Simulation Cycle
Sequential vs Concurrent Statements

- VHDL is inherently a concurrent language
  - All VHDL processes execute concurrently
  - Concurrent signal assignment statements are actually one-line processes
- VHDL statements execute sequentially *within a process*
- Concurrent processes with sequential execution within a process offers maximum flexibility
  - Supports various levels of abstraction
  - Supports modeling of concurrent and sequential events as observed in real systems

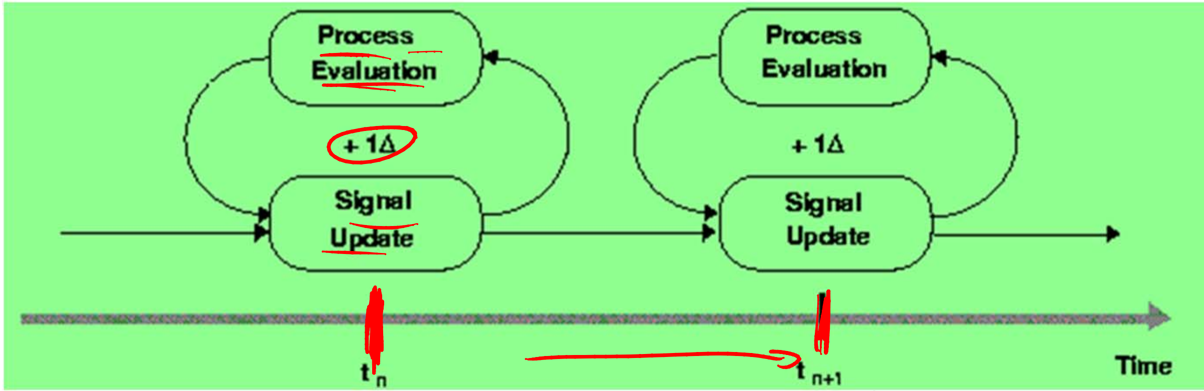
Concurrent Statements

- Basic granularity of concurrency is the *process*
  - Processes are executed concurrently
  - Concurrent signal assignment statements are one-line processes



- Mechanism for achieving concurrency :
  - Processes communicate with each other via signals
  - Signal assignments require delay before new value is assumed
  - Simulation time advances when all active processes complete
  - Effect is concurrent processing
    - I.e. order in which processes are actually executed by simulator does not affect behavior

Delta Delay

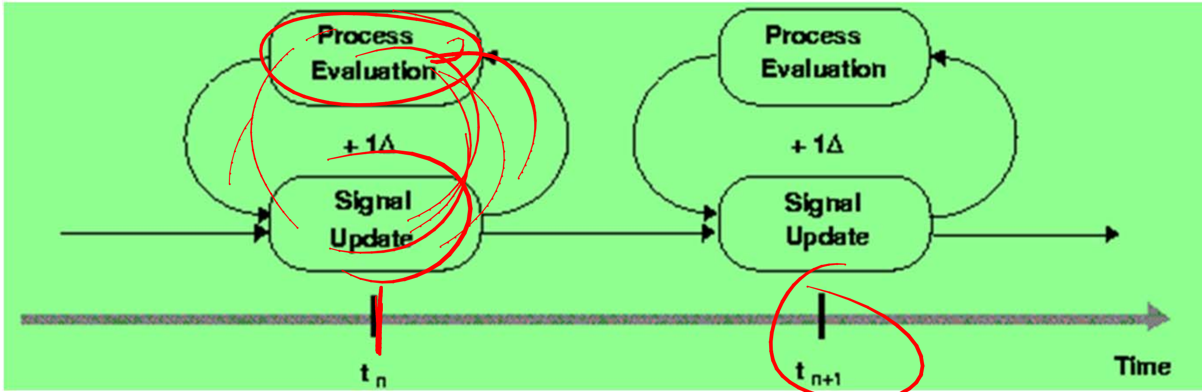


- Default signal assignment propagation delay if no delay is explicitly prescribed
  - VHDL signal assignments do not take place immediately
  - Delta is an infinitesimal VHDL time unit so that all signal assignments can result in signals assuming their values at a future time
  - E.g.

```
Output <= NOT Input;
-- Output assumes new value in one delta cycle
```

- Supports a model of concurrent VHDL process execution
- Order in which processes are executed by simulator does not affect simulation output

Delta Delay



1) all active processes can execute in the same simulation cycle

2) each active process will suspend at wait statement (sensitive list → process finish)

3) when all processes are suspended simulation is advanced the minimum time necessary so that some signals can take on their new values

4) processes then determine if the new signal values satisfy the conditions to proceed from the wait statement at which they are suspended

5) all processes are suspended and no signal update:

$t_n \rightarrow t_{n+1}$ (new entries in the event queue)