Outline

• HDL and Embedded Systems Design
• SystemC and VHDL
• Platform Based Design
• Transaction Level Modeling
• Verification by Simulation
• Example: a real case
• Summary

Introduction

• Electronic systems consist of:
  – HW platform
  – SW application layers
  – Interfaces
  – Analog components
  – Sensors and transducers
• Main trends:
  – Migration from analog to digital processing
  – Broader system-level integration to support System-On-a-Chip (SOC) approach

Embedded system architecture

Challenges in the design of embedded systems

• Increasing application complexity even in standard and large volume products
  – large systems with legacy functions
  – mixture of event driven and data flow tasks
  – flexibility requirements
  – examples: multimedia, automotive, mobile communication
• Increasing target system complexity
  – mixture of different technologies, processor types, and design styles
  – large systems-on-a-chip combining components from different sources (IP market)
• Numerous constraints and design objectives
• Reduced and overlapping design cycles

Hardware/software co-design

• Hardware/software co-design:
  – combined design of hardware and software
• Goals
  – design process optimization
  – increased design productivity
  – design optimization
  – improved product quality
• Tasks
  – co-specification and co-modeling
  – co-verification
  – co-design process integration and optimization
  – design optimization and co-synthesis
Co-design advantages

- Explore different design alternatives in the architectural design space
- Tune HW to SW and vice-versa
- Reduce the system design time
- Support coherent design specification at the system-level
- Facilitate the re-use of HW and SW parts
- Provide integrated environment for the synthesis and validation of HW and SW components

Co-design of embedded systems

- Design of dedicated computing and control systems
- Embedded controllers
  - On-line control of manufacturing process
  - Robots guidance and control
  - Aircraft, automobile and ship control
- Data processing and communication systems
  - Telecom
  - Radio-navigation

Co-design of embedded systems

- Design of dedicated HW parts
  - Different design styles:
    - Co-processors, embedded cores, ASIPs, ...
  - Widely varying design scale
- Design of dedicated SW parts
  - Special-purpose operating systems
  - Drivers of peripheral devices

HDL motivation

- Does it really work?
- Functional specifications failures?
- Performance expectations missing?
- Re-spins!

Main reason: lacking of a concretely usable view of the complete system before the tape-out phase!
State of the practice

- Co-simulation as a support of design (process) integration
  - extension of simulation techniques to combined simulation of hardware and software components
  - allows permanent control of hardware and software component consistency
  - supports early validation of reused component integration
- Integration validation more costly with increasing level of detail
  - current focus on co-simulation for lower levels of a design
  - simulation with models of specific processors, memories, busses, ...
  - reduction of accuracy mainly to improve simulation performance
  - examples: Mentor Seamless CVS, Viewlogic Eagle

State of the practice

- “Executable” co-specification used as a basis for system validation
- Virtual prototyping
  - simulation based validation
  - many commercial examples for different applications
    - Statemate (i-Logix), MatrixX (ISI), MATLAB (MathWorks)
    - RASSP program (DARPA)
- Rapid prototyping with “hardware-in-the-loop”
  - hardware supported system emulation in real environment
  - often custom design

Specification languages

- Different communities:
  - VLSI system design: VHDL, VERILOG, Specchart, ...
  - DSP: COSSAP, SPW, ...
  - Continuous design: MATLAB, MATRIXX, ...
  - Synchronous system design: Esterel, Lustre, Statechart
  - Classical programming: C, C++, Java, ...
  - Functional and algebraic: VDM, Z, B, Funmath, ...
  - Structured design methods: SART, OMT, ...

Example of specification Language

- SDL
  - well-suited for control-intensive, real-time systems
  - flow chart FSM, both graphics and text
  - abstract data types
  - dynamic process creation
  - synchronization via blocking, RPC
  - can monitor performance constraints

Concepts for system level specification

- CONCURRENCY
  - different levels (bit, operation, statement, process, system)
  - two types: data-driven, control-driven
- HIERARCHY
  - needed for structured design methodologies
  - Two types: behavior, structure
- COMMUNICATION
  - data exchange between concurrent subsystems
  - two types: message passing, shared memory
- SYNCHRONIZATION
  - two models: synchronous, asynchronous
Example of specification Language

- **StateCharts, SpecCharts**
  - graphical FSM of states and transitions
  - addition of hierarchical states for modeling complex reactive behaviors
  - SpecCharts adds:
    - behavioral completion
    - exceptions
  - may attach VHDL code to states and transitions arcs
  - extended with arithmetics
  - Easy to use for control-dominated systems

Simulation and debugging requirements

- **Embedded controllers:**
  - ASICs plus SW running on a processor
  - VHDL or Verilog plus C programs
  - Weakly heterogeneous systems
- **Embedded data processing and communication systems**
  - ASICs plus SW running on a processor or ASIP
  - Environmental modeling (e.g. telephone lines)
  - Strongly heterogeneous systems

Co-simulation

- **Simulate at the same time both hardware and software**
- **Two conflicting requirements:**
  - execute the software as fast as possible
  - keep hardware and software simulations synchronized so they interact as they will in the target system.

Co-simulation paradigms

- **Homogeneous** modeling:
  - HW models in HDL
  - Processor model in HDL
  - SW in assembly code
- **Usage of HDL simulator for the whole system including the processor model**
- Simple method but quite inefficient

Co-simulation paradigms

- **Weakly heterogeneous** systems
  - a) HDL simulators with processor model
  - b) Compiled SW
  - c) HW emulation
- **Strongly heterogeneous** systems
  - Require specialized simulation environments (e.g. Ptolemy)
  - Communication mechanisms among domains and their corresponding schedulers
HDL processor modeling

- Precise timing model
  - Accurate timing and complete functionality
  - Event-driven simulation
- Zero-Delay Model (ZDM) for timing
  - Correct transitions at clock edges
  - Cycle-based simulation
- Instruction-set simulator
  - Model emulates processor while insuring correct register and memory values

Compiled SW

- Basic assumption:
  - HW/SW communication protocol such that communication delay has no effect on functionality
- SW is compiled and linked to simulator
- HW/SW communication is replaced by handshake
- Simulation speed is limited by HW simulation speed

HW emulation

- HW mapped onto programmable HW
  - One order of magnitude loss in speed
- Programmable HW boards connected to workstations
- Limited visibility of internal states

Software versus Hardware Design

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB</td>
<td>MHz</td>
</tr>
</tbody>
</table>

Example: GCD modeled in C

```c
#include <stdio.h>

int gcd(int xi, int yi)
{
    int x, y, temp;
    x = xi; y = yi;
    while (x > 0){
        if (x <= y){
            temp = y;
            y = x;
            x = temp;
        } else {
            scanf("%d %d", &xi, &yi);
            y = y;
        }
    }
    return(y);
}

main()
{
    int xi, yi, ou;
    scanf("%d %d", &xi, &yi);
    ou = gcd(xi, yi);
    printf("%d\n", ou);
}
```

Hardware requirements (1)

- Input/Output
  - S printf, scanf...
  - Hcomponent interface must be defined
- Timing
  - CPU instructions are executed at the CPU clock speed
  - Hone or more explicit CLOCK signals must be defined
Hardware requirements (2)

- Variables size
  Shidden implicit definition (integer 4bytes, char 1byte, ...)
  Hall pre-defined and user-defined types must be translated into bit vectors

- Relationships operands/operators
  Sall operators in the C libraries are accepted
  Hexplicit mapping of operands on operators

Hardware requirements (3)

- Memory elements identification
  Sthe optimization module of the compiler transparently maps variables onto CPU registers and memory elements
  Hthe synthesis tool identifies memory elements by analyzing the algorithmic semantics

- Modules synchronization
  Ssequential execution of instructions
  Hinherently parallel execution of all components

Entity/Architecture

```
ENTITY gcd IS
  PORT ( clock, reset : IN bit;
         xi,yi : IN unsigned (size-1 DOWNTO 0);
         ou : OUT unsigned (size-1 DOWNTO 0));
END gcd;
```

Example: GCD modeled in VHDL

```
ARCHITECTURE behavioral OF gcd IS
BEGIN
  PROCESS
  VARIABLE x, y,temp : unsigned (size-1 DOWNTO 0);
  BEGIN
    WAIT UNTIL clock = '1';
    x := xi;
    y := yi;
    WHILE (x >= y) LOOP
      IF (x <= y) THEN
        temp:=y;
        y := x;
        x:=temp;
      END IF;
      x := x - y;
    END LOOP;
    ou <= y;
  END PROCESS;
END behavioral;
```

Algorithmic (Behavioral) Synthesis

- Identification of a target architecture
  – FSM + Data-Path (FSMD)
- Identification of time instants for each operation:
  – time order
  – time length
- Identification of operators:
  – data size
  – time performance

FSMD Model
**Scheduling (FSM)**

```vhdl
ARCHITECTURE behavioral OF gcd IS
BEGIN
    PROCESS
        VARIABLE x, y, temp : unsigned (size-1 DOWNTO 0);
        BEGIN
            WAIT UNTIL clock = '1';
            x := xi;
            y := yi;
            WHILE (x > 0) LOOP
                IF (x <= y) THEN
                    temp := y;
                    y := x;
                    x := temp;
                END IF;
                x := x - y;
            END LOOP;
            ou <= y;
        END PROCESS;
    END behavioral;
```

**Allocation (Data-Path)**

```vhdl
ARCHITECTURE behavioral OF gcd IS
BEGIN
    PROCESS
        VARIABLE x, y, temp : unsigned (size-1 DOWNTO 0);
        BEGIN
            WAIT UNTIL clock = '1';
            x := xi;
            y := yi;
            WHILE (x > 0) LOOP
                IF (x <= y) THEN
                    temp := y;
                    y := x;
                    x := temp;
                END IF;
                x := x - y;
            END LOOP;
            ou <= y;
        END PROCESS;
    END behavioral;
```

**Register Transfer Level (RTL)**

- Interconnection of FSM + Data-path
  - control/condition signals are identified
  - FSM is represented by states and transitions
  - Data-path is represented by registers, multiplexers and operators
- Automatic translation into a set of registers and library components (RTL)

**Logic Synthesis**

- Each RTL module is optimized by means of:
  - area
  - delay
  - power
- Efficient synthesis algorithms exist for:
  - two-level synthesis
  - multiple-level synthesis
- A logic representation of each RTL module is generated and optimized

**Standard C-based design flow**

1. **System Level Model C++**
2. **Analysis**
3. **Results**
4. **Manual conversion**
5. **VHDL/Verilog**
6. **Simulation**
7. **Synthesis**
8. **FMD description**

**SystemC-based design flow**

1. **SystemC Model System Level**
2. **Simulation**
3. **Refinement**
4. **FMD Logic description**
5. **Automatic translation**
6. **VHDL/Verilog**
7. **Synthesis**
8. **SystemC Model RT Level**
SystemC story

- Open SystemC Initiative (OSCI)
  - a standard for modeling digital systems
  - founders:
    - Synopsys, CoWare, Frontier Design ... ARM,
      Cygnus, Ericsson, Fujitsu, Infineon, Lucent, Sony,
      ST, TI ...
    - Free use of the language
    - Controlled language extension
    - Open market for tools

SystemC key features

- Concurrency:
- Communication:
- Notion of time:
- Reactivity:
- Hardware data types:
- Simulation support:
- Debugging support:
  - Processes (syn and asyn)
  - Signals, channel
  - Multiple clocks with arbitrary phases
  - Waiting for events
  - Bit vectors, arbitrary precision integer
  - Simulation kernel
  - C++ debugging tools

SystemC: main characteristics

- Modules and Hierarchy (cap. 3)
- Processes (cap. 4)
- Ports and Signals (cap. 5)
- Data Types (cap.6)
- Hardware Examples (app. A)

Use of SystemC distribution

- Header files:
  - SystemC class definition
- Libraries:
  - Class library
  - Simulation kernel
- Building strategy:
  - compilation of home-made classes
  - linking of libraries

Executable program with simulation capabilities

SC_Module example

```cpp
SC_MODULE(alu) {
  // input/output ports
  sc_in<bool> clock;
  sc_in<sc_int<N>> op1;
  ...
  sc_out<sc_int<N>> o;
  // method
  void reg_par_par();
  void calcola();
  // internal signals
  sc_signal<sc_int<N>> acc;
  sc_signal<sc_int<N>> t6;
  // constructor
  SC_CTOR(alu) {
    SC_METHOD(reg_par_par);
    sensitive_pos(clock);
    SC_METHOD(calcola);
    sensitive(op1);
  }
};
```

Proces types

- SystemC supports three kinds of proces:
  - methods (SC_METHOD):
    - executed from the start to the end
    - sensitive to signals
  - threads (SC_THREAD):
    - executed up to a wait()
  - clocked threads (SC_CTHREAD):
    - sensitive to clocks
Process example

```cpp
SC_MODULE(my_module) {
    // ports declaration
    sc_in<int> a;
    sc_in<bool> b;
    sc_out<int> x;
    // signals declaration
    sc_signal<bool> c;
    // process declaration
    void my_method_proc();
    // constructor
    SC_CTOR(my_module) {
        // process record
        SC_METHOD(my_method_proc);
        // sensitivity list declaration
    }
};
```

Simulation kernel

- SystemC scheduler works as follows:
  1. all clock signals are updated
  2. all SC_METHOD's and SC_THREAD's with modified input values are executed
  3. all SC_CTHREAD's which must be executed are inserted into a queue
  4. Steps 2 and 3 are repeated up a fixed point
  5. SC_CTHREAD's on queue are executed
  6. increase execution time and goto 1

Port and signal types

- `sc_int<n>` e `sc_uint<n>`
- `sc_bigint<n>` e `sc_biguint<n>`
- `sc_bit`
- `sc_logic`
- `sc_bv<n>` e `sc_lv<n>`
- `sc_fixed` e `sc_ufixed`
- `sc_fix` e `sc_ufix`
- end user self defined structures

Port and signal types example

- `sc_in<port_type>;
  – // input port of type port_type
- `sc_out<port_tipe> x[32];`
  – // output port ranging from x[0] to x[31] of type port_type
- `sc_signal<port_type> i[4];`
  – // signal ranging from i[0] to i[3] of type port_type

Synchronous D-flip-flop

```cpp
// dff.h
#include "systemc.h"
SC_MODULE(dff) {   //module declaration
    sc_in<bool> clock; //input declaration
    sc_in<bool> din;  
    sc_out<bool> dout; //output declaration
    void doit() {
        dout = din;
    }
    SC_CTOR(dff) {   //declaration of a SC_METHOD
        // process sensitive to clock
        SC_METHOD(doit);
        sensitive_pos(clock);
    }
};
```

Asynchronous D-flip-flop

```cpp
// dffa.h
#include "systemc.h"
SC_MODULE(dffa) {   //module declaration
    sc_in<bool> clock; //input declaration
    sc_in<bool> reset;  
    sc_in<bool> din;  
    sc_out<bool> dout; //output declaration
    void do_ffa() {
        if (reset)
            dout = false;
        else if (clock.event())
            dout = din;
    }
    SC_CTOR(dffa) {   //declaration of a SC_METHOD
        // process sensitive to clock
        SC_METHOD(dffa);
        sensitive(reset);
        sensitive_pos(clock);
    }
};
```
Parallel-parallel register

```cpp
SC_MODULE(reg_par_par) { //module declaration
    SC_METHOD(register_par_par); //method to build the register
    SC_CTOR(reg_par_par) { //constructor declaration
        // declaration of a SC_METHOD
        // process sensitive to clock
        sensitive_pos(clock);
    };
}
```

Serial/serial register

```cpp
SC_MODULE(reg_ser_ser) { //module declaration
    SC_METHOD(register_ser_ser); //method to build the register
    SC_CTOR(reg_ser_ser) { //constructor declaration
        // declaration of a SC_METHOD
        // process sensitive to clock
        sensitive_pos(clock);
    };
}
```

Parallel/serial register

```cpp
SC_MODULE(reg_par_ser) { //module declaration
    SC_METHOD(register_par_ser); //method to build the register
    SC_CTOR(reg_par_ser) { //constructor declaration
        // declaration of a SC_METHOD
        // process sensitive to clock
        sensitive_pos(clock);
    };
}
```
Shifter

```cpp
// shifter.h
#include "systemc.h"
#define N 8
SC_MODULE(shifter) {
    sc_in<bool> ds;
    sc_in<sc_bv<N>> a;
    sc_in<bool> i0;
    sc_out<sc_bv<N>> o;
    void shift();
    SC_CTOR(shifter) {
        SC_METHOD(shift);
        sensitive(ds);
        sensitive(a);
        sensitive(i0);
    }
};
```

```cpp
// shifter.cpp
#include "shifter.h"
void shifter::shift() {
    bool ds1, i01;
    sc_bv<N> a1, c1;
    i01 = i0.read();
    ds1 = ds.read();
    a1 = a.read();
    // right shift
    if(ds1 == 1) {
        c1.range(N-2,0) = a1.range(N-1,1);
        c1[N-1] = i01;
    } else {
        c1.range(N-1,1) = a1.range(N-2,0);
        c1[0] = i01;
    }
    o.write(c1);
}
```

ALU

```cpp
// alu.h
#include "systemc.h"
#define N 8
#define P 2
SC_MODULE(alu) {
    sc_in<bool> clock;
    sc_in<sc_int<N>> op1;
    sc_in<sc_int<N>> op2;
    sc_in<bool> stored;
    sc_in<sc_uint<P>> oper;
    sc_out<sc_int<N>> o;
    // methods
    void reg_par_par();
    void calcola();
    // signals
    sc_signal<sc_int<N>> acc;
    sc_signal<sc_int<N>> t6;
    SC_CTOR(alu) {
        SC_METHOD(reg_par_par);
        sensitive_pos(clock);
        SC_METHOD(calcola);
        sensitive(op1);
        sensitive(op2);
        sensitive(stored);
        sensitive(oper);
        sensitive(acc);
        // signals
        sc_signal<sc_int<N>> acc;
        sc_signal<sc_int<N>> t6;
    }
};
```

```cpp
// alu.cpp
#include "alu.h"
void alu::reg_par_par() {};
void alu::calcola() {
    sc_int<N> op11, op21, acc1, sel, t61;
    bool stored;
    op11 = op1.read(); op21 = op2.read();
    acc1 = acc.read(); stored = stored.read();
    sel = stored == 1 ? acc1 : op21;
    switch(sel) {
    case 0: t61 = op11+sel; break;
    case 1: t61 = op11-sel; break;
    case 2: if(op11 < sel) t61=op11; else t61=sel;
    case 3: if(op11 > sel) t61=op11; else t61=sel;
    }
    t6.write(t61); o.write(t61);
}
```

Platform Based Design

Definition: platform-based design is the creation of a stable microprocessor-based architecture that can be rapidly extended, customized for a range of applications and delivered to customers for quick deployment. (J.M. Chateau – STMicroelectronics)
To implement SW

To implement HW

Software

22/05/2006 SFM06-HW 69

To implement SW

To implement HW

Specs (UNL, Matlab, C/C++)

Module (Object)

To implement SW

To implement HW

SoC design (SystemC)

Profile and HW/SW partitioning

Functional test

Real-time, area, costs, and power constraints?

Performance analysis

22/05/2006 SFM06-HW 70

Platform based design: summary

• High-level description for functional verification.
• Timing estimation for performance analysis.
• Hierarchy for HW/SW partitioning.
• Modularity for IPs re-use and/or refinement.
• Simulation speed vs. implementation details.
• Synthesizable descriptions for HW modules.

Transaction Level Modeling (TLM)

• Goal: enabling Software development to start very early in the design flow

Classical design flow


Transaction Level Modeling (TLM)

• Transaction Level is the new design and verification abstraction above RTL.

Abstraction

TLM

RTL

Synthesis

22/05/2006 SFM06-HW 71
Transaction Level Modeling (TLM)

- Key concept: more emphasis on the data transfer functionality and less on their implementation details at the early design stage.

```
write (data, addr);
```

TLM: design languages

- SystemC 2.0.1 → IEEE 1666.
- Open SystemC Initiative (OSCI) Standard TLM: released in February 2005, rigorously defines implementation rules:
  - API defined for every TLM layer, since they form the heart of the TLM standard.
  - Transactor: to allow communications between components implemented at different abstraction levels.

TLM design languages

- Other implementation rules:
  - Argument passing semantics in function calls, to assure safety in concurrent environments.
  - Blocking/non-blocking calls to characterize communication semantics (i.e., polling, interrupt, etc.).
  - Pipelined communication mechanism, to implement different architectural choices to meet throughput requirements.
  - Uni/Bidirectional communication channels, to implement FIFO’s or hierarchical channels.

TLM layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Specific Model</th>
<th>Communication Processes</th>
<th>Implementation Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Message</td>
<td>TLM_Architecture</td>
<td>(Timed/Un“Ifed)</td>
<td>(Timed/Un“Ifed)</td>
</tr>
<tr>
<td>2 Transaction</td>
<td>PE assembly model</td>
<td>Programmers View</td>
<td></td>
</tr>
<tr>
<td>1 Transfer</td>
<td>Bus arbitration model</td>
<td>Cycle Accurate</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TLM key concepts

- Factoring out the common elements, the key concepts are:
  - To implement a system at higher layer means to implement the system in a more abstract way:
    - Leave implementation details in order to speed-up simulation (for functional verification purpose).
  - To implement a system at lower layer means to add implementation details to the system:
    - In order to simulate the system in a more accurate way (for performance analysis purpose and architectural exploration).
TLM: architectural exploration

Examples:
- **Sequential re-timing**: balancing combinational logic to maximize clock speed.
- **Resources sharing**: area vs. latency (High-level synthesis).
- **Pipeline optimization**.

TLM layer 3

- Executable specifications and first level of data and control functional partitioning.
- System proof of concepts.
- Implementation architecture abstract.
- Untimed functionalities modeling.

Example:
- Event-driven simulation semantics.
- Point-to-point Initiator-Target connection.
- Abstract data types.
- Functional verification goal.
- Simulation speed!

TLM layer 2

- Hardware architectural performance and detailed behavior analysis.
- HW/SW partitioning and co-development.
- Cycle performance estimation.
- Split pipeline with time delays.

TLM layer 1

- Detailed analysis and low level SW development.
- Modeling CA interface for abstract simulation models of IP blocks such as embedded processors.
- Clock-accurate performance simulation
- Clock-accurate protocols mapped to the chosen HW interfaces and bus structures.
- Interface pins are hidden.
- Parameterizable to model different bus protocol and signal interfaces.
- Performance analysis goal

TLM: transactor (TLM-TLM example)

Reference block

- write (Address, Data)

Refined block

- write (Address, Data)

Transactor examples

**write/read API (i.e., TL3)**

```c
status = port.write(SLAVE_ADDR, 27);
status = port.read(SLAVE_ADDR, mem);
```

**put/get API (i.e., TL2)**

```c
request.data = &data_pkt;
request.address = SLAVE_ADDR;
request.type = WRITE;
out port.put(request);
response = in port.get();
```
SoC verification

- **Validation:**
  - Does the design correctly work?
  - Are there implementation design errors?
  - No reference models!

- **Verification:**
  - Refinement (or abstraction) step verification: ascertaining that system functionality is kept unaltered throughout its representation hierarchy.
  - Golden Model as reference.

Dynamic verification (simulation)

- Golden model
- Refined model

- System level and block level verification
- Results quality
- Eased
- Suitable for TLM & RTL

Dynamic verification: faults injection

IF (5 % 2) output = true;
ELSE output = false;

Fault injected

IF (5 % 2) output = true;
ELSE output = false;

Found!

IF (5 % 2) output = true;
ELSE output = false;

Fault injected

IF (5 % 2) output = true;
ELSE output = false;

Not found!

TL3-RTL transactor

```cpp
void fsm_protocol(){
    while (true){
        switch (fsm_state) {
            case RESET:
                ...
                ST_1:
                out_data.write(data);
                out_len.write(len);
                state = ST_2;
                ST_2:
                ...
        }
    }
}
```

SoC verification techniques

- Dynamic verification (simulation):
  - High performance
  - Advanced commercial tools
  - No-exhaustive technique

- Static verification (formal):
  - Quality
  - Exhaustive technique
  - Long verification time
  - For small size designs

Semi-formal verification
Dynamic verification: faults injection

- Black level validation
- Results quality
- Suitable for TLM level 1 & RTL

Property Specification Language (PSL)

- Synthesizable design required for faults injection

Model checking

- Synthesized embedded system
- Design dependent temporal properties (CTL/CTT)
- Model Checker
- Properties hold/don't hold

- Block level validation
- Results quality
- Suitable for TLM level 1 & RTL

Equivalence checking (EC)

- Combinational Equivalence Checking
  - Block level verification
  - Results quality
  - Synthesizable design required
  - Suitable only for RTL vs Gate level verification
  - For small size designs

- Sequential Equivalence Checking
  - Block level verification
  - Suitable for TLM & RTL
  - Results quality
  - Synthesizable design required
  - For very small size design

Example: face recognition system

- (STMicroelectronics)
- Captured Image
- 1st step
- 2nd step
- 3rd step
- CPU
- Camera
- Memory
Summary

- Platform Based Design (PBD): the customization approach for design complexity, manufacturing cost and time-to-market challenges.
- Transaction Level Model (TLM) as SoC modeling style.
- SystemC 2.1 as the de-facto reference languages for SoC design: it rocks!
- Assertion Based Verification (ABV) and Property Specification Language (PSL) as the new validation and verification trend.
- Dynamic approach for system level verification and static approach for block level verification.