Models, Architectures, Languages

- Introduction
- Models
  - State, Activity, Structure, Data, Heterogeneous
- Architectures
  - Function-Architecture, Platform-Based
- Languages
  - Hardware: VHDL / Verilog / SystemVerilog
  - Software: C / C++ / Java
  - System: SystemC / SLDL / SDL
  - Verification: PSL (Sugar, OVL)
**Design Methodologies**

- Capture-and-Simulate
  - Schematic Capture
  - Simulation
- Describe-and-Synthesize
  - Hardware Description Language
  - Behavioral Synthesis
  - Logic Synthesis
- Specify-Explore-Refine
  - Executable Specification
  - Hardware-Software Partitioning
  - Estimation and Exploration
  - Specification Refinement

---

**Motivation**

Executable specification

```
if (x = 0) then
  y = a + b / 2
```

System implementation

- Processor
- Memory
- Video accelerator
- ASIC
- I/O

Models Languages

- Partitioning
- Estimation
- Refinement

Software compilation
- Behavioral synthesis
- Logic synthesis

Physical design
- Test generation
- Manufacturing
**Models & Architectures**

Models are conceptual views of the system’s functionality
Architectures are abstract views of the system’s implementation

**Behavior Vs. Architecture**

Performance models: Emb. SW, comm. and comp. resources

HW/SW partitioning, Scheduling

SW estimation
Models of an Elevator Controller

"If the elevator is stationary and the floor requested is equal to the current floor, then the elevator remains idle. If the elevator is stationary and the floor requested is less than the current floor, then lower the elevator to the requested floor. If the elevator is stationary and the floor requested is greater than the current floor, then raise the elevator to the requested floor."

(a) English description

(b) Algorithmic model

loop
  if (req_floor = curr_floor) then
    direction := idle;
  elseif (req_floor < curr_floor) then
    direction := down;
  else (req_floor > curr_floor) then
    direction := up;
  end if;
end loop;

(c) State-machine model

Architectures Implementing the Elevator Controller

(a) Register level

(b) System level
Current Abstraction Mechanisms in Hardware Systems

Abstraction
The level of detail contained within the system model

- A system can be modeled at
  - System Level,
  - Algorithm or Instruction Set Level,
  - Register-Transfer Level (RTL),
  - Logic or Gate Level,
  - Circuit or Schematic Level.

- A model can describe a system in the
  - Behavioral domain,
  - Structural domain,
  - Physical domain.

Abstractions in Modeling: Hardware Systems

<table>
<thead>
<tr>
<th>Level</th>
<th>Behavior</th>
<th>Structure</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>PMS (System)</td>
<td>Communicating Processes</td>
<td>Processors Memories</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Switches (PMS)</td>
<td>Cabinets, Cables</td>
</tr>
<tr>
<td>Instruction Set (Algorithm)</td>
<td>Input-Output</td>
<td>Memory, Ports Processors</td>
<td>Board Floorplan</td>
</tr>
<tr>
<td>Register-Transfer</td>
<td>Register Transfers</td>
<td>ALUs, Regs, Muxes, Bus</td>
<td>ICs Macro Cells</td>
</tr>
<tr>
<td>Logic</td>
<td>Logic Eqns.</td>
<td>Gates, Flip-flops</td>
<td>Std. cell layout</td>
</tr>
<tr>
<td>Circuit</td>
<td>Network Eqns.</td>
<td>Trans., Connections</td>
<td>Transistor layout</td>
</tr>
</tbody>
</table>

© IEEE 1990

[McFarland90]
Current Abstraction Mechanisms for Software Systems

Virtual Machine
A software layer very close to the hardware that hides the hardware’s details and provides an abstract and portable view to the application programmer

Attributes
– Developer can treat it as the real machine
– A convenient set of instructions can be used by developer to model system
– Certain design decisions are hidden from the programmer
– Operating systems are often viewed as virtual machines

Abstractions for Software Systems

Virtual Machine Hierarchy
● Application Programs
● Utility Programs
● Operating System
● Monitor
● Machine Language
● Microcode
● Logic Devices
UNIFIED HW/SW REPRESENTATION

- Unified Representation –
  - High-level system (SoC) architecture description
  - Implementation (hardware or software) independent
  - Delayed hardware/software partitioning
  - Cross-fertilization between hardware and software
  - Co-simulation environment for communication
  - System-level functional verification
Abstract Hardware-Software Model

Unified representation of system allows early performance analysis

HW/SW System Models

- State-Oriented Models
  - Finite-State Machines (FSM), Petri-Nets (PN), Hierarchical Concurrent FSM
- Activity-Oriented Models
  - Data Flow Graph, Flow-Chart
- Structure-Oriented Models
  - Block Diagram, RT netlist, Gate netlist
- Data-Oriented Models
  - Entity-Relationship Diagram, Jackson’s Diagram
- Heterogeneous Models
  - UML (OO), CDFG, PSM, Queuing Model, Programming Language Paradigm, Structure Chart
State-Oriented: Finite-State Machine (Mealy Model)

\[ S = \{ s_1, s_2, s_3 \} \]
\[ I = \{ r_1, r_2, r_3 \} \]
\[ O = \{ d_2, d_1, n, u_1, u_2 \} \]
\[ f: S \times I \rightarrow S \]
\[ h: S \times I \rightarrow O \]

State-Oriented: Finite State Machine (Moore Model)
State-Oriented: Finite State Machine with Datapath

فئات اللعبة: ما هي العلاقات بين الفئات؟

Finite State Machines

- **Merits**
  - Represent system's temporal behavior explicitly
  - Suitable for control-dominated systems
  - Suitable for formal verification

- **Demerits**
  - Lack of hierarchy and concurrency
  - State or arc explosion when representing complex systems
State-Oriented: Petri Nets

- System model consisting of places, tokens, Petri Nets: a transitions, arcs, and a marking
  - Places - equivalent to conditions and hold tokens
  - Tokens - represent information flow through system
  - Transitions - associated with events, a “firing” of a transition indicates that some event has occurred
  - Marking - a particular placement of tokens within places of a Petri net, representing the state of the net

Example:

Example: Input Places → Transition → Output Place

State-Oriented: Petri Nets

(a) Sequence

(b) Branch

(c) Synchronization

(d) Resource contention

(e) Concurrency
**Petri Nets**

- **Merits**
  - Good at modeling and analyzing concurrent systems
  - Extensive theoretical and experimental works
  - Used extensively for protocol engineering and control system modeling
- **Demerits**
  - “Flat Model” that becomes incomprehensible when system complexity increases

**State-Oriented:**

*Hierarchical Concurrent FSM*
Hierarchical Concurrent FSM

- **Merits**
  - Support both hierarchy and concurrency
  - Good for representing complex systems

- **Demerits**
  - Concentrate only on modeling control aspects and not data and activities

Activity-Oriented: Data Flow Graphs (DFG)
Data Flow Graphs

- **Merits**
  - Support hierarchy
  - Suitable for specifying complex transformational systems
  - Represent problem-inherent data dependencies

- **Demerits**
  - Do not express control sequencing or temporal behaviors
  - Weak for modeling embedded systems

Activity-Oriented: Flow Charts

```
start

J = 1
MAX = 0

J > N

J = J+1

MEM(J) > MAX

MAX = MEM(J)

end
```
Flow Charts

- Merits
  - Useful to represent tasks governed by control flows
  - Can impose an order to supersede natural data dependencies

- Demerits
  - Used only when the system’s computation is well known

Structure-Oriented: Component Connectivity Diagrams
**Component Connectivity Diagrams**

- **Merits**
  - Good at representing system’s structure

- **Demerits**
  - Behavior is not explicit

- **Characteristics**
  - Used in later phases of design

---

**Data-Oriented:**

**Entity-Relationship Diagram**

![Entity-Relationship Diagram](image_url)
**Entity-Relationship Diagrams**

- **Merits**
  - Provide a good view of the data in a system
  - Suitable for representing complex relationships among various kinds of data

- **Demerits**
  - Do not describe any functional or temporal behavior of a system

---

**Data-Oriented: Jackson’s Diagram**

```
Drawing
  / \  AND
 Color  Shape
 /     \  OR
Circle  Rectangle
/       \  AND
   Radius   Width   Height
```

Users *

Name
Jackson’s Diagrams

- Merits
  - Suitable for representing data having a complex composite structure

- Demerits
  - Do not describe any functional or temporal behavior of the system

Heterogeneous: Control/Data Flow Graphs (CDFG)

- Graphs contain nodes corresponding to operations in either hardware or software
- Often used in high-level hardware synthesis
- Can easily model data flow, control steps, and concurrent operations because of its graphical nature

Example:

```
5 X 4 Y
```

Control Step 1
Control Step 2
Control Step 3
Control/Data Flow Graphs

- **Merits**
  - Correct the inability to represent control dependencies in DFG
  - Correct the inability to represent data dependencies in CFG

- **Demerits**
  - Low level specification (behavior not evident)

---

Heterogeneous: Structure Chart
Structure Charts

- Merits
  - Represent both data and control

- Demerits
  - Used in the preliminary stages of system design

Heterogeneous:
Object-Oriented Paradigms (UML, ...)

- Use techniques previously applied to software to manage complexity and change in hardware modeling
- Use OO concepts such as
  - Data abstraction
  - Information hiding
  - Inheritance
- Use building block approach to gain OO benefits
  - Higher component reuse
  - Lower design cost
  - Faster system design process
  - Increased reliability
Heterogeneous: Object-Oriented Paradigms (UML, …)

Object-Oriented Representation

Example:

3 Levels of abstraction:

- Register
  - Read
  - Write
- ALU
  - Add
  - Sub
  - AND
  - Shift
  - Mult
  - Div
- Processor
  - Load
  - Store

UML in the SoC Design Methodology
Object-Oriented Paradigms

- **Merits**
  - Support information hiding
  - Support inheritance
  - Support natural concurrency

- **Demerits**
  - Not suitable for systems with complicated transformation functions

---

Heterogeneous: Program State Machine (PSM)

```
variable A: array[1..20] of integer

variable i, max: integer;
max = 0;
for i = 1 to 20 do
  if ( A[i] > max ) then
    max = A[i];
  end if;
end for
```
Program State Machine

- Merits
  - Represent a system’s state, data, control, and activities in a single model
  - Overcome the limitations of programming languages and HCFSM models

Heterogeneous: Queuing Model

(a) One server

(b) Multiple servers
**Queuing Models**

- **Characteristics**
  - Used for analyzing system’s performance
  - Can find utilization, queuing length, throughput, etc.

---

**Codesign Finite State Machine (CFSM)**

- CFSM is FSM extended with
  - Support for data handling
  - Asynchronous communication

- CFSM has
  - FSM part
    - Inputs, outputs, states, transition and output relation
  - Data computation part
    - External, instantaneous functions
**Codesign Finite State Machine (CFSM)**

- CFSM has:
  - Locally synchronous behavior
    - CFSM executes based on snap-shot input assignment
    - Synchronous from its own perspective
  - Globally asynchronous behavior
    - CFSM executes in non-zero, finite amount of time
    - Asynchronous from system perspective

- GALS model
  - Globally: Scheduling mechanism
  - Locally: CFSMs

---

**Network of CFSMs: Depth-1 Buffers**

- Globally Asynchronous, Locally Synchronous (GALS) model
**Typical DSP Algorithm**

- **Traditional DSP**
  - Convolution/Correlation
  - Filtering (FIR, IIR)
    \[ y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \]
  - Adaptive Filtering (Varying Coefficient)
    \[ y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M-1} b_k x[n] \]
  - DCT
    \[ x[k] = e(k) \sum_{n=0}^{N-1} x[n] \cos\left(\frac{(2n+1)k\pi}{2N}\right) \]

**Specification of DSP Algorithms**

- **Example**
  \[ y(n)=ax(n)+bx(n-1)+cx(n-2) \]
- **Graphical Representation Method 1: Block Diagram**
  (Data-path architecture)
  - Consists of functional blocks connected with directed edges, which represent data flow from its input block to its output block
Graphical Representation Method 2: Signal-Flow Graph

- SFG: a collection of nodes and directed edges
- Nodes: represent computations and/or task, sum all incoming signals
- Directed edge (j, k): denotes a linear transformation from the input signal at node j to the output signal at node k
- Linear SFGs can be transformed into different forms without changing the system functions.
  - Flow graph reversal or transposition is one of these transformations (Note: only applicable to single-input-single-output systems)

Signal-Flow Graph

- Usually used for linear time-invariant DSP systems representation
- Example:
Graphical Representation Method 3: Data-Flow Graph

- DFG: nodes represent computations (or functions or subtasks), while the directed edges represent data paths (data communications between nodes), each edge has a nonnegative number of delays associated with it.
- DFG captures the data-driven property of DSP algorithm: any node can perform its computation whenever all its input data are available.

\[
\begin{align*}
\text{DFG: nodes represent computations (or functions or subtasks), while the directed edges represent data paths (data communications between nodes), each edge has a nonnegative number of delays associated with it.} \\
\text{DFG captures the data-driven property of DSP algorithm: any node can perform its computation whenever all its input data are available.}
\end{align*}
\]

Data-Flow Graph

- Each edge describes a precedence constraint between two nodes in DFG:
  - Intra-iteration precedence constraint: if the edge has zero delays
  - Inter-iteration precedence constraint: if the edge has one or more delays (Delay here represents iteration delays.)
  - DFGs and Block Diagrams can be used to describe both linear single-rate and nonlinear multi-rate DSP systems

Fine-Grain DFG

\[
\begin{align*}
\text{DFG: nodes represent computations (or functions or subtasks), while the directed edges represent data paths (data communications between nodes), each edge has a nonnegative number of delays associated with it.} \\
\text{DFG captures the data-driven property of DSP algorithm: any node can perform its computation whenever all its input data are available.}
\end{align*}
\]
Examples of DFG

- Nodes are complex blocks (in Coarse-Grain DFGs)
  
  \[
  \begin{align*}
  &\text{FFT} \quad \text{Adaptive} \quad \text{IFFT} \\
  &\text{filtering}
  \end{align*}
  \]

- Nodes can describe expanders/decimators in Multi-Rate DFGs

\[
\begin{align*}
\text{Decimator} & \quad \frac{\downarrow 2}{N \text{ samples}} \quad \frac{\downarrow}{\text{N/2 samples}} \quad \equiv \quad 2 \quad 1 \\
\text{Expander} & \quad \frac{\uparrow 2}{\text{N/2 samples}} \quad \frac{\uparrow}{\text{N samples}} \quad \equiv \quad 1 \quad 2
\end{align*}
\]

Graphical Representation Method 4: Dependence Graph

- DG contains computations for all iterations in an algorithm.
- DG does not contain delay elements.
- Edges represent precedence constraints among nodes.
- Mainly used for systolic array design.
"You hear the planned possibilities, but it is nothing like the visual concept of a model. You get the impact of the complete vision."

Galyn Fish
Director of PR, Southwest Medical Center

Southwest Medical Center
Oklahoma City, Oklahoma
System Level Design Science

- Design Methodology:
  - Top Down Aspect:
    - Orthogonalization of Concerns:
      - Separate Implementation from Conceptual Aspects
      - Separate computation from communication
    - Formalization: precise unambiguous semantics
    - Abstraction: capture the desired system details (do not overspecify)
    - Decomposition: partitioning the system behavior into simpler behaviors
    - Successive Refinements: refine the abstraction level down to the implementation by filling in details and passing constraints
  - Bottom Up Aspect:
    - IP Re-use (even at the algorithmic and functional level)
    - Components of architecture from pre-existing library

Separate Behavior from Micro-architecture

- System Behavior
  - Functional specification of system
  - No notion of hardware or software!

- Implementation Architecture
  - Hardware and Software
  - Optimized Computer
Example of System Behavior

IP-Based Design of the System Behavior
The next level of Abstraction ...

IP Block Performance
Inter IP Communication Performance Models

IP Blocks
RTL
SW Models

RTL Clusters
SDF
Wire Load

Gate Level Model
Capacity Load

Transistor Model
Capacity Load

Cluster

1970's
1980's
1990's
Year 2000 +

IP-Based Design of the Implementation

Which Bus? PI? AMBA? Dedicated Bus for DSP?

Which DSP Processor? C50? Can DSP be done on Microcontroller?

Which Microcontroller? ARM? HC11?

How fast will my User Interface Software run? How Much can I fit on my Microcontroller?

Can I Buy an MPEG2 Processor? Which One?

Do I need a dedicated Audio Decoder? Can decode be done on Microcontroller?

Can decode be done on Microcontroller?

Processor Bus

DSP Processor

DSP RAM

Control Processor

System RAM

Audio Decode

Peripheral

MPEG

External I/O

Fall 2002
Architectural Choices

1/Efficiency (power, speed)

Map Between Behavior and Architecture

Transport Decode Implemented as Software Task Running on Microcontroller

Communication Over Bus

Audio Decode Behavior Implemented on Dedicated Hardware

Fall 2002
**Classic A/D, HW/SW tradeoff**

- RF Front End
- Can trade custom analog for hardware, even for software
  - Power, area critical criteria, or easy functional modification

**Example: Voice Mail Pager**

- Design considerations cross design layers
- Trade-offs require systematic methodology and constraint-based hierarchical approach for clear justification
Where All is Going

- Create paradigm shift - not just link methods
  - New levels of abstraction to fluidly tradeoff HW/SW, A/D, HF/IF, interfaces, etc - to exploit heterogeneous nature of components
  - Links already being forged

Deep Submicron Paradigm Shift

- Cell Based Design
  - Minimize Area
  - Maximize Performance
  - Optimize Gate Level
- Virtual Component Based Design
  - Minimize Design Time
  - Maximize IP Reuse
  - Optimize System Level
- 90% New Design
- 90% Reused Design
Implementation Design Trends

Platform Based
- Consumer
- Wireless
- Automotive

Hierarchical
- Microprocessors
- High end servers & W/S

Flat Layout
- Net & Compute
- Servers
- Base stations

Platform-Based System Architecture Exploration

Level of Abstraction
- Function
- HW/SW
- Architecture
- RTL - SW
- Mask - ASM

Effort/Value
- Today
- Tomorrow

Application Space
- System Platform
- Architectural Space
Digital Wireless Platform

Source: Berkeley Wireless Research Center

Will the system solution match the original system spec?

- Limited synergies between HW & SW teams
- Long complex flows in which teams do not reconcile efforts until the end
- High degree of risk that devices will be fully functional

• Concept
  • Development
  • Verification
  • System Test

• IP Selection
• Design
• Verification

Fall 2002
EDA Challenge to Close the Gap

- Industry averaging 2-3 iterations SoC design
- Need to identify design issues earlier
- Gap between concept and logical/physical implementation

AMBA-Based SoC Architecture

- High-performance ARM processor
- High-bandwidth on-chip RAM
- UART
- APB
- Timer
- Keypad
- PIO
- DMA bus master
- AHB to APB Bridge
- AHB Bridge

Source: GSRC
Languages

- Hardware Description Languages
  - VHDL / Verilog / SystemVerilog
- Software Programming Languages
  - C / C++ / Java
- Architecture Description Languages
  - EXPRESSION / MIMOLA / LISA
- System Specification Languages
  - SystemC / SLDL / SDL / Esterel
- Verification Languages
  - PSL (Sugar, OVL) / OpenVERA
Hardware Description Languages (HDL)

- VHDL (IEEE 1076)
- Verilog 1.0 (IEEE 1364)
- Verilog 2.0 (IEEE 1364)
- SystemVerilog 3.0 (to be sent for IEEE review)
- SystemVerilog 3.1 (to be sent for IEEE review)

SystemVerilog 3.0

- Built-in C types
- Synthesis improvements (avoid simulation and synthesis mismatches)
- Enhanced design verification
  - procedural assertions
- Improved modeling
  - communication interfaces
SystemVerilog 3.1

- Testbench automation
  - Data structures
  - Classes
  - Inter-process communication
  - Randomization
- Temporal assertions
  - Monitors
  - Coverage Analysis

SystemVerilog Environment for Ethernet MAC
**Architecture Description Language in SOC Codesign Flow**

- Design Specification
- Estimators
- Hw/Sw Partitioning
- IP Library
- HW S/W Memory
- Processor Core
- On-Chip Memory
- Synthesized HW Interface
- SW Compiler
- Cosimulation
- Synthesis
- Rapid design space exploration
- Quality tool-kit generation
- Design reuse

**Architecture Description Languages**

**Objectives for Embedded SOC**

- Support automated SW toolkit generation
  - exploration quality SW tools (performance estimator, profiler, …)
  - production quality SW tools (cycle-accurate simulator, memory-aware compiler..)

- Specify a variety of architecture classes (VLIWs, DSP, RISC, ASIPs…)
- Specify novel memory organizations
- Specify pipelining and resource constraints
Architecture Description Languages

- Behavior-Centric ADLs *(primarily capture Instruction Set (IS))*
  - ISPS, nML, ISDL, SCP/Valenc, ...
  - good for regular architectures, provides programmer’s view
  - tedious for irregular architectures, hard to specify pipelining, implicit arch model
- Structure-Centric ADLs *(primarily capture architectural structure)*
  - MIMOLA, ...
  - can drive code generation and architecture synthesis, can specify detailed pipelining
  - hard to extract IS view
- Mixed-Level ADLs *(combine benefits of both)*
  - LISA, RADL, FLEXWARE, MDes, ...
  - contains detailed pipelining information
  - most are specific to single processor class and/or memory architecture
  - most generate either simulator or compiler but not both

EXPRESSION ADL

Verification

Feedback

EXPRESSION ADL

Exploration Phase

Refinement Phase

Application

Toolkit Generator

Exploration Simulator

Profile

Toolkit Generator

Retargetable Compiler

Profile

Feedback
SystemC History

- Synopsys ATG
- UC Irvine
- Synopsys “Scenic”
- Synopsys “Fridge”
- Frontier Design A/RT Library
- imec
- CoWare “N2C”

SystemC v0.90 Sep. 99
SystemC v1.0 Apr. 00
SystemC v1.1 Jun. 00

VSIA SLD Data Types Spec (draft)

SystemC Highlights

- Features as a codesign language
  - Modules
  - Processes
  - Ports
  - Signals
  - Rich set of port and signal types
  - Rich set of data types
  - Clocks
  - Cycle-based simulation
  - Multiple abstraction levels
  - Communication protocols
  - Debugging support
  - Waveform tracing
Current System Design Methodology

- Problems
  - Errors in manual conversion from C to HDL
  - Disconnect between system model and HDL model
  - Multiple system tests
SystemC Design Methodology

- SystemC Model
- Simulation
- Refinement
- Synthesis
- Rest of Process

SystemC Design Methodology (cont’d)

- Advantages
  - Refinement methodology
  - Written in a single language
  - Higher productivity
  - Reusable testbenches
SystemC Programming Model

- A set of *modules* interacting through *signals*.
- *Module* functionality is described by *processes*.

SystemC *programming model* (cont’d)

- System (program) debug/validation
  - Testbench
    - Simulation, Waveform view of signals
  - Normal C++ IDE facilities
    - Watch, Evaluate, Breakpoint, ...

- `sc_main()` function
  - instantiates all *modules*
  - initializes *clocks*
  - initializes output waveform files
  - starts simulation kernel
A Simple Example: Defining a Module

- Complex-number Multiplier
  \[(a+bi)\cdot(c+di) = (ac-bd)+(ad+bc)i\]

```
SC_MODULE(cmplx_mult) {
    sc_in<int> a, b;
    sc_in<int> c, d;
    sc_out<int> e, f;
    void calc();
    SC_CTOR(cmplx_mult) {
        SC_METHOD(calc);
        sensitive<<a<<b<<c<<d;
    }
}
```

A Simple Example: Defining a Module (cont’d)

```
void cmplx_mult::calc()
{
    e = a*c-b*d;
    f = a*d+b*c;
}
```
Completing the Design

input_gen module

\[
\begin{array}{c}
\text{M1} & \text{M2} & \text{M3} \\
\text{input_gen} & \text{Complex Multiplier} & \text{display}
\end{array}
\]

\begin{Verbatim}
SC_MODULE(input_gen) {
    sc_in<bool> clk;
    sc_out<int> a,b;
    sc_out<int> c,d;
    void generate();
    SC_CTOR(input_gen) {
        SC_THREAD(generate);
        sensitive_pos(clk);
    }
}

void input_gen::generate()
{
    int a_val=0, c_val=0;
    while (true) {
        a = a_val++;
        wait();
        c = (c_val+=2);
        wait();
    }
}
\end{Verbatim}
Completing the Design: display module

```cpp
SC_MODULE(display) {
    sc_in<int> e, f;
    void show();
    SC_CTOR(display) {
        SC_METHOD(show);
        sensitive<<e<<f;
    }
}

void display::show()
{
    cout<<e<<'+'<<f<<"i\n";
}
```

Putting it all together: sc_main function

```cpp
#include <systemc.h>

int sc_main()
{
    input_gen M1("I_G");
    cmplx_mult M2("C_M");
    display M3("D");
    sc_signal<int> a, b, c, d, e, f;
    sc_clock clk("clk", 20, 0.5);
    M1.clk(clk.signal());
    M1.a(a); M1.b(b);
    M1.c(c); M1.d(d);
    M2.a(a); M2.b(b);
    M2.c(c); M2.d(d);
    M2.e(e); M2.f(f);
    M3.e(e); M3.f(f);
    sc_start(100);
    return 0;
}
```
Property Specification Language (PSL)

- Accellera: a non-profit organization for standardization of design & verification languages
- PSL = IBM Sugar + Verplex OVL
- System Properties
  - Temporal Logic for Formal Verification
- Design Assertions
  - Procedural (like SystemVerilog assertions)
  - Declarative (like OVL assertion monitors)
- For:
  - Simulation-based Verification
  - Static Formal Verification
  - Dynamic Formal Verification