High-level modelling tools

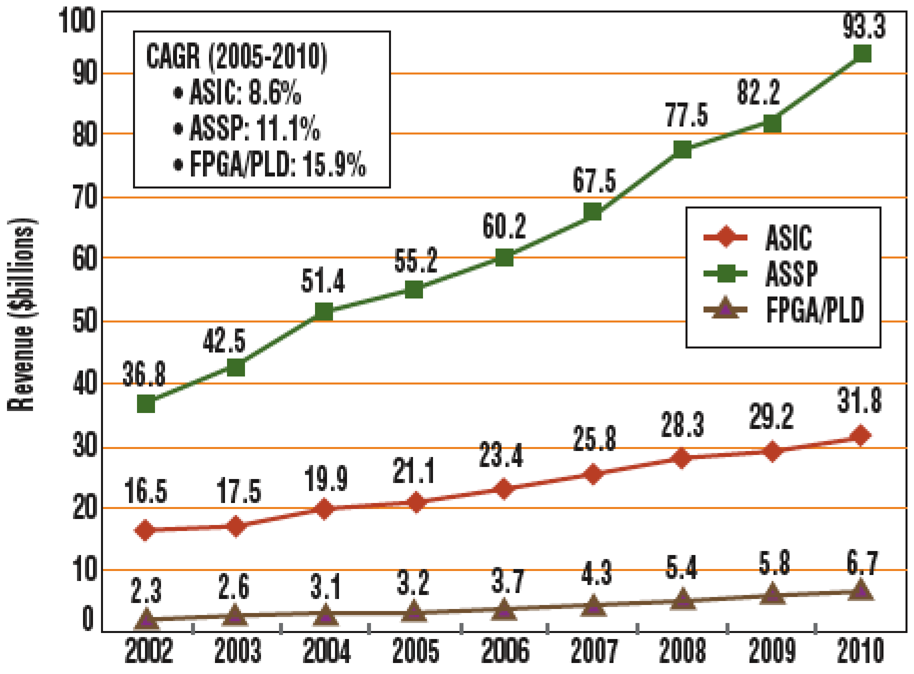
J. Evans CERN

Contents

Modern electronic design methodology

Modern tools and methodologies are needed to:

- Manufacture new devices
- Implement them in working circuits



1. ASICs, ASSPs, and FPGAs/programmable logic devices (PLDs) will all see a good compound annual growth rate (CAGR) through 2010, with FPGAs delivering the highest CAGR of nearly 16% per year. (courtesy of Gartner)

http://www.elecdesign.com/Articles/Print.cfm?AD=1&ArticleID=14442

A high-level modelling tool is..?

- A high-level modelling tool provides the ability to build and simulate ideal models. Once proper validation is complete on these ideal models, lower levels of abstraction can be added until the final real world model is designed
- This is the top-down design concept. Design first takes place with the basic functional knowledge of a system. The scope and use of high-level modelling tools is continually expanding
- High-level modelling tools and validation are almost mandatory for very large designs

History of EDA

- First design tools used for IC design (mid 70s) - then called CAD (layout)/CAE (design capture and verification) tools
- SPICE2 presented in 1975
- Beginning of 80s, company spun-out in-house design tools to form companies
- Mid 80s, Hardware Description Languages (HDL) emerge
- Electronics Design Automation (EDA) term now used expanded to cover all aspects of design from the IC through to the circuit to the PCB to system level

SPICE

- **SPICE** (Simulation Program with Integrated Circuit Emphasis) is a general purpose analog circuit simulator. It is used in IC, circuit and PCB simulations.
- An “analog” circuit simulator (solves for voltage between and current through circuit nodes)
- Developed at Berkeley, initially by funding from DOD
- Re-written to allow “free” distribution – many commercial simulators e.g. PSpice, HSPICE based on SPICE
- SPICE2 was big improvement as introduced variable time-step transient analysis

Hardware Description Languages in EDA

- An HDL is a standard text-based description of an electronic system. In contrast to a software programming language, it incorporates syntax and semantics to include the notions of time and concurrency, essential to describe hardware.
- The two major languages currently used are VHDL and Verilog. They are typically used to support the conception, simulation, and implementation of designs in FPGAs
- These are RTL languages

Verilog language

- Verilog HDL was developed in the 1980s by Gateway Design Automation as a proprietary hardware description language for use on its simulator.
- Cadence Design Systems acquired Gateway. In the early 90s, Verilog was put in the public domain.
- Verilog HDL became IEEE Std. 1364-1995 in 1995.
- Verilog HDL now maintained by Accellera (non-profit organisation). Accellera has also been developing a new standard, SystemVerilog.
- There also exists Verilog-AMS. The Verilog-AMS standard supports analogue and mixed signal designs at three levels: transistor/gate, transistor/gate- rtl /behavioural, and mixed transistor/gate- rtl /behavioural circuit levels.

VHDL language

- VHDL development started in 1981 by the United States Department of Defense
- Designs were inadequately documented to be reproduced in new technologies. The cost of verifying each updated individual component was prohibitive. VHDL developed as a language that:
- could adequately describe (document) the part
- could produce similar results on any simulator
- independent of technology or design methodology

- VHDL is a non-proprietary, comprehensively defined language. The Language Reference Manual defines the language completely. The LRM does not define any given simulator – however, it clearly defines how each language construct should be handled during simulation.
- VHDL can be used at any level of abstraction
- There also exists VHDL-AMS

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity casdsp_ex is
      generic (tpd_hl : TIME := 1 NS;
               tpd_lh : TIME := 1 NS);
      port     (in1, in2 : in std_logic;
               casdsp_out : out std_logic);
end casdsp_ex ;

architecture only of casdsp_ex is
begin
      p1: process (in1, in2)
           begin
      if ((in1='1') and (in2='1')) then
      casdsp_out <= '1' after tpd_lh;
      else
      casdsp_out <= '0'after tpd_hl;
      end if;
end process;
end only;
```

The **ENTITY** declaration defines the inputs to and outputs from the model, and any **GENERIC** parameters used by the different implementations

Entities and architectures are the only two design units that must exist in any VHDL design description

ARCHITECTURE defines a different implementation or behaviour of a given design unit

The **LIBRARY** statement is used to make specified libraries visible in a model description. A **USE** statement can precede the declaration of any entity or architecture which is to utilize items from the package.

beware of latches

AMS extensions

- The VHDL-AMS language is an extension of the IEEE 1076 (VHDL) standard that supports the description and the simulation of analog, digital, and mixed-signal circuits and systems.
- VHDL-AMS provides a mechanism for analogue behavior specification and mixed system modeling (conservative/non-conservative)
- Continuous models are based on differential algebraic equations (DAEs)
- DAEs need to be solved by a simulation kernel: the analogue solver
- VHDL-AMS supports the handling of initial conditions, piecewise-defined behavior, and discontinuities

- Need to solve analogue signals => different tools can give different results

AMS extensions

- Extended structural semantics
  - Conservative semantics to describe physical systems (e.g. respect Kirchhoff's law for electrical circuits)
  - Non-conservative semantics for abstract models (signal-flow descriptions)
  - Mixed-signal interfaces
- Mixed-signal semantics
  - Mixed-signal initialization and simulation cycle
  - Mixed-signal descriptions of behavior
  - Frequency domain support
  - Small-signal frequency and noise modeling and simulation

AMS extensions

A **TERMINAL** declares a terminal and its nature

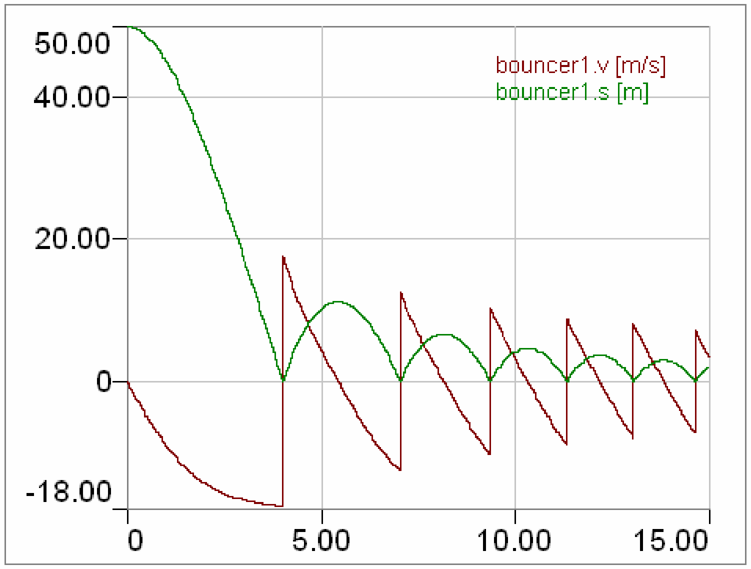
The **QUANTITY** declaration defines one or more identifiers as quantity objects. A branch quantity appears in the architecture declaration to specify across and through terminals

```
-- VHDL-AMS model
        -- (c) Southampton University 1997
        -- author: Tom Kazmierski
        -- Department of Electronics and Computer Science, University of Southampton
        -- e-mail: tjk@ecs.soton.ac.uk
        -- Last revised:  20 August 2005 (by Shaolin Wang)

library IEEE;
        use IEEE.electrical_systems.all;

        entity capacitor is
           generic (cap := 2007E-6:  capacitance); -- Capacitance [F]
           port (terminal p1, p2 : electrical);
        end entity capacitor;

        architecture ideal of capacitor is
           quantity v across i through p1 to p2;
        begin
           if domain=quiescent_domain use
              v == 0.0;        --initial condition
           else
              i == cap * v'dot; -- Fundamental equation
           end use;
        end architecture ideal;
```

AMS extensions

```
library ieee;
use   ieee.mechanical_systems.ALL;

entity bouncer is
        generic (
                    initial_position : displacement := 50.0;
                    gravity_accel : acceleration := 9.81;
                    air_resistance : real := 0.1
        );

end entity bouncer;
        architecture first_order of bouncer is
         quantity v : velocity;
         quantity s : displacement;

         begin
         break s => initial_position;
         break v => -v when not s'above(0.0);
         s'dot == v;
         if (v > 0.0) use
         v'dot == -gravity_accel - v ** 2*air_resistance;
        else
         v'dot == -gravity_accel + v ** 2*air_resistance;
        end use;

        end architecture first_order;
```

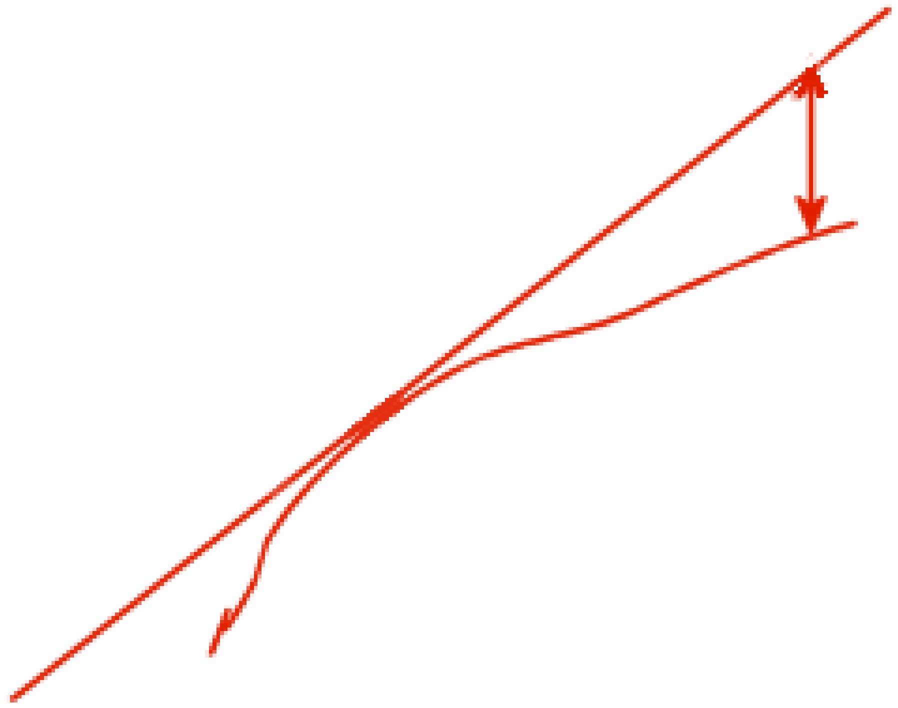
Model and simulation result using Ansoft Simplorer Student Version 7, www.ansoft.com



VHDL-AMS execution

- Elaboration of a design (generally) gives
  - Digital part => set of processes (digital simulation kernel)
  - Analogue part => set of equations (analogue solver)
- Execution of design simulation
- Initialization: find quiescent state of the model
- Simulation: time domain, small-signal frequency, or noise

SPICE Algorithms



SPICE Algorithms

- **Problem**:

  Knowing a function at some point in time t_n, how to approximate the function at a future time point t_{n+1}?

  - SPICE calculates an approximation to an analytical solution at discrete time points using numeric integration
  - Forward Euler is simplest. From a given point (t_n , y_n), next point is estimated from slope of known initial point.

    $y_{n+1} \sim y_n + h\, f(t_n , y_n)$ - an explicit method - y_{n+1} is estimated from known y_n

  - Backward Euler is often used:

    $y_{n+1} \sim y_n + h\, f(t_{n+1} , y_{n+1})$ - implicit method – needs more computation but can have better results

- Can develop further by taking averages of slopes of more points further in the future - Runge-Kutta method (we'll see later on in MATLAB, ode45 etc)

  $s_1 = f(t_n , y_n)$, $s_2 = f(t_n + (h/2), \; y_n + s_1(h/2))$
  $s_3 = f(t_n + (h/2), \; y_n + s_2(h/2))$ $s_4 = f(t_n + h, \; y_n + s_3 h)$

  $t_{n+1} = t_n + h$, $y_{n+1} = y_n + (s_1 + 2s_2 + 2s_3 + s_4)(h/6)$

SPICE Algorithms

- To save computational time, SPICE makes the timestep as large as possible while still providing an acceptable solution

What is the definition of "acceptable"? This is defined by the simulator parameters (PSpice):

  - **VNTOL** - best accuracy of voltages
  - **ABSTOL -** best accuracy of currents
  - **RELTOL -** relative accuracy of V and I

- Convergence problems arise when solution cannot be found. This is typically when timestep cannot be made small enough. Can solve by:
  - adding Cs (!)
  - changing simulator parameters e.g. reltol, abstol, vntol etc, but !!!

Why is it difficult to do mixed analogue/digital simulation?

- Digital simulation is done using fixed, well-defined time-steps
- Analogue simulation uses variable time-steps
- Getting both simulators working together is not trivial for efficient mixed-signal simulations. Techniques that have been used:
- ”Backplanes” between simulators – lots of interaction => possible inefficiency
- Synchronizing the simulators:
  - **Lockstep**

    This algorithm requires that the two engines are locked together in time throughout the simulation => the smallest timestep is used. Also, the first simulator to complete a time step must wait for the other to catch up.

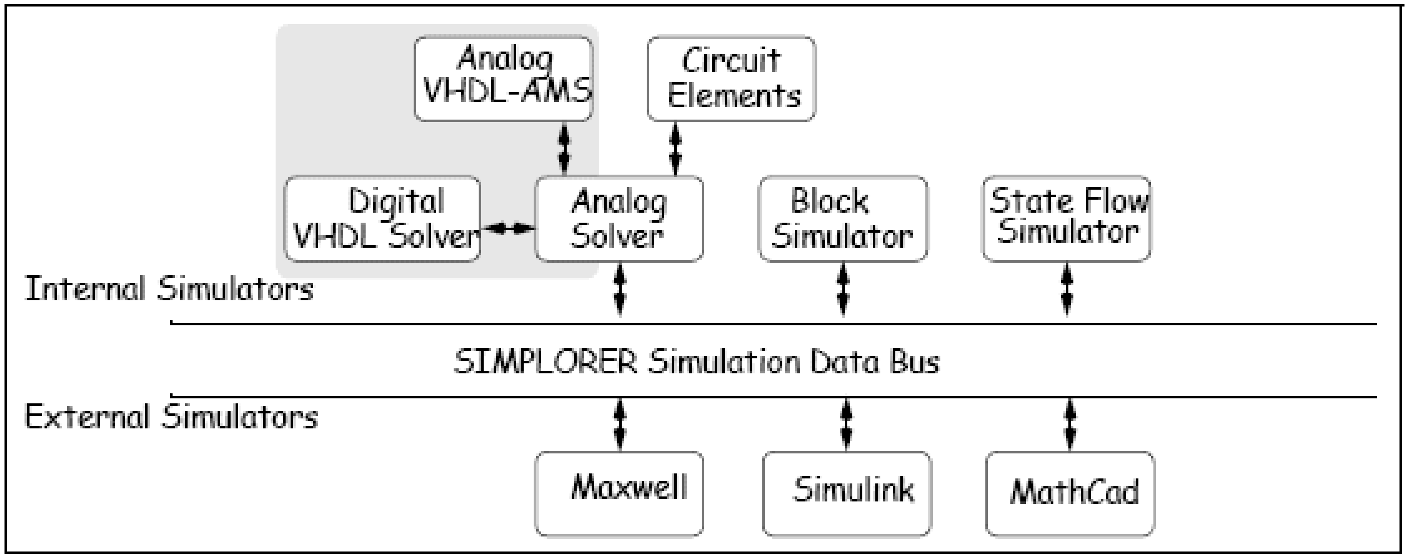
  Disadvantage: one simulator is usually waiting for the other => inefficiency
  - **Calaveras**

    The Calaveras algorithm allows one simulator to run ahead of the other. If the simulators subsequently determine that evaluations in one domain would have affected the other, the effected simulator “rewinds" and repeats simulation with the new data.

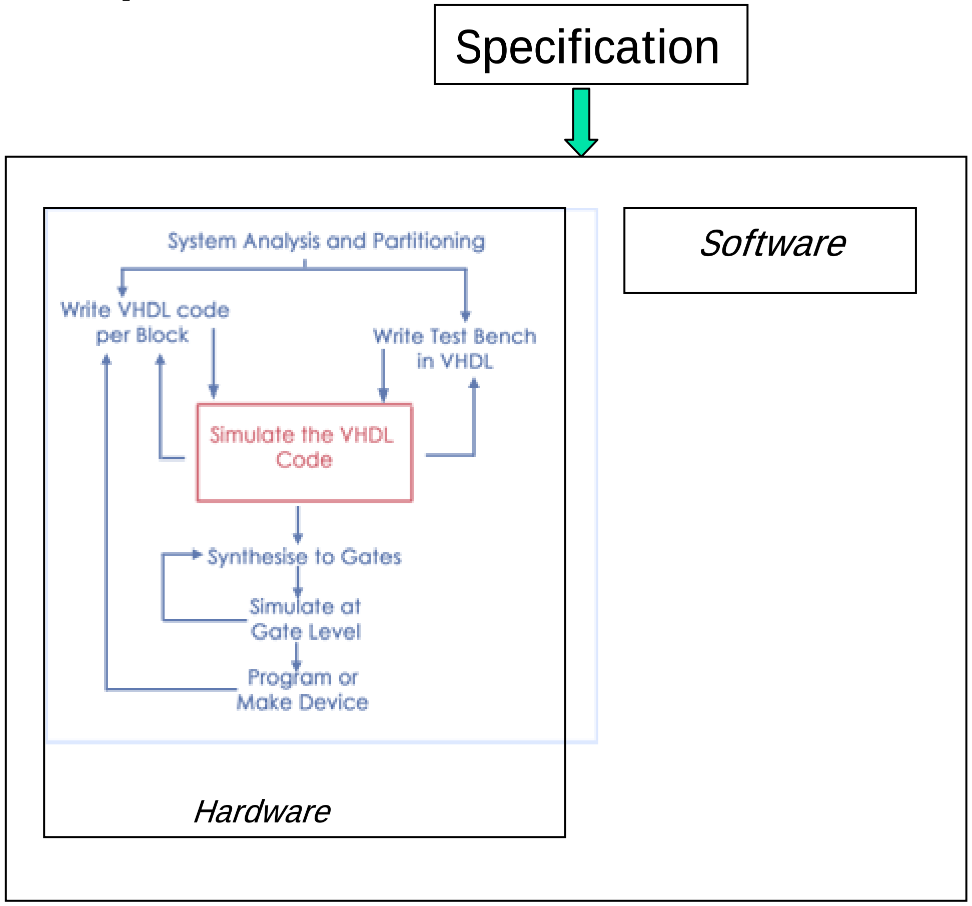
  Disadvantage: Often need to discard data => inefficiency

Mixed analogue/digital simulation

Simplorer® is a multi-domain, system simulator. It is also linked to their other tools.



“Traditional” design flow



A typical modern design has both hardware and software elements.

After system analysis and partitioning, HDL code is written and simulated for the block(s) while coding associated software.

This flow has several problems. Some of them are:

- the specification translation to the VHDL design must be done manually
- what is being tested is the perceived design intent, NOT the specification
- There is no obvious link between the hardware and software

Electronic System Level tools have been developed to try and overcome these problems.

Electronic System Level design

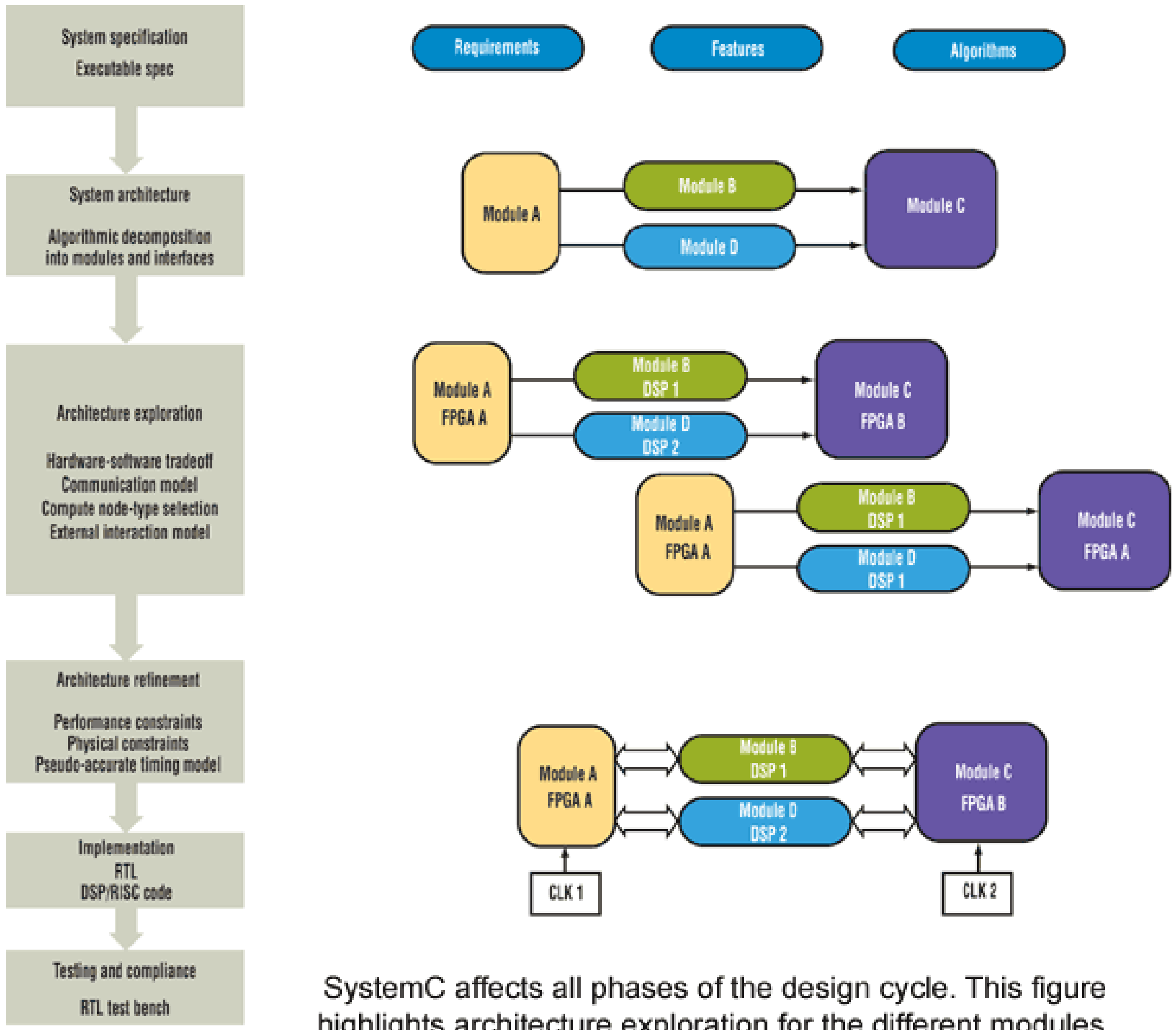
- (One) definition of ESL is:

  "The utilization of appropriate abstractions in order to increase comprehension about a system, and to enhance the probability of a successful implementation of functionality in a cost-effective manner, while meeting necessary constraints."

(from *"ESL Design and Verification: A Prescription for Electronic System Level Methodology"*, Elsevier Morgan Kaufmann, February 2007)

- What ESL seems to mean to most people:
  - Allows fast exploration of the solution space before detailed simulation
  - Easily change model level of sophistication
  - Co-simulate hardware and software
  - Simulate/Verify system behaviour

- Some solutions – *SystemC, SystemVerilog*



SystemC affects all phases of the design cycle. This figure highlights architecture exploration for the different modules.

http://www.elecdesign.com/Articles/Index.cfm?ArticleID=11083&pg=3

ESL Design flow

- An *executable specification* is a model that is a direct translation of a design specification. Executable specifications model the intended functionality of a design without taking into account any proposed implementation
- The specification can be used as a “golden model”. When verifying the implemented design, compare its output to that of the “golden model” for same test scenarios
- The executable specification can be timed or untimed – any included time delays represent timing constraints to be imposed on the (undefined) implementation
- Timed functional models are often used for early hardware-software trade-off analysis to evaluate impact of mapping processes to hardware (one delay figure) versus mapping to software (another delay figure)
- Communication delays of a target implementation modelled using timing delays on the communication between modules.

  Note - executable specifications and both untimed and timed functional models do not have any direct structural correspondence to a target implementation

- Timed / untimed functional and communication models usually done through *Transactional Level Modelling*

Transaction-level modeling

- Transaction level modelling (TLM) uses a higher abstraction level for modelling than RTL
- With TLM, it is possible to accurately model many aspects of a system at a higher level. Using TLM simplifies the modelling effort and gains simulation speed
- A TLM distinguishes between function and communication
- Model types trade off performance versus accuracy

  -> more accurate model, slower

  -> less accurate model, faster

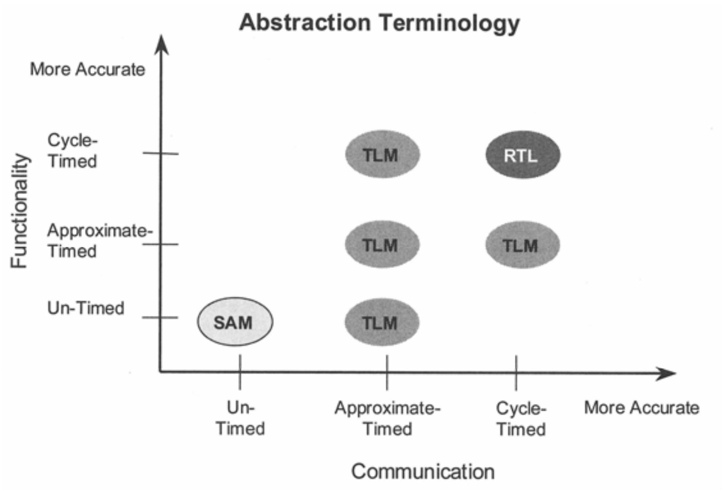
Transaction-level modeling

- TLM nomenclature can be confusing…
  - ***1.1 Programmers View (PV)***

    Intended to imply a pure function call based interface with no communication events and little timing information carried. Usually associated with Untimed Functional (UTF) behaviour description.

  - ***1.2 Programmers View with Timing (PV+T) (Architect's View)***

    A modelling style where a PV and a signal or FIFO interface co-exist –the simulation can switch between two interfaces, with and without timing. The goal of PV+T style is to allow model refinement without changing the functional description of the model's behaviour.



Ref: SystemC: From the Ground UP, Black, Donovan

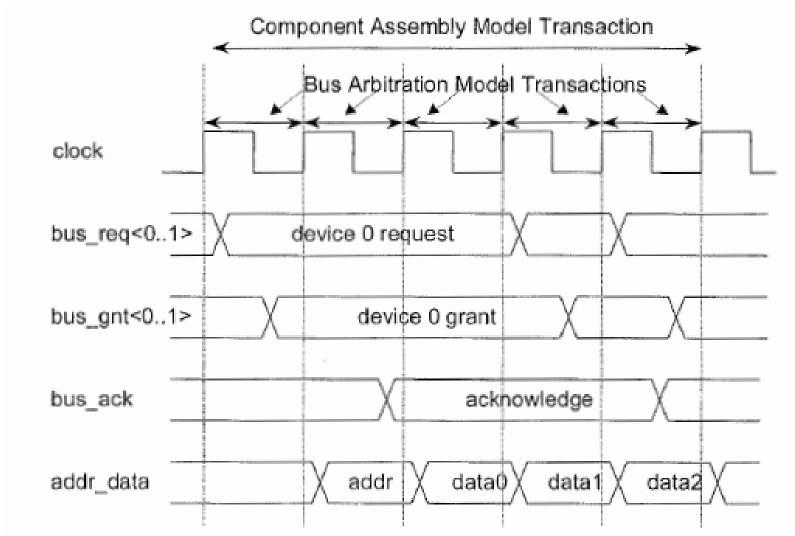
1.3 Cycle Callable (CC) , Verification View

A cycle accurate modelling style. The cycle-by-cycle behaviour of the interface is explicitly described. The behaviour of the model coupled with this interface would also include cycle timing – either a "cycle-count-accurate" Timed Functional (TF) model or fully cycle-accurate. Alternatively, the behaviour may be omitted or modelled in a very trivial manner to create a cycle-accurate performance model.

Transaction-level modeling

- **Examples on how to access a bus device**

- Highest-level modelling (component assembly)
  - Exact bus-timing details unimportant. All information is transferred using one interaction
- Lower level (more detail – bus arbitration or cycle-accurate)
  - Number of bus cycles becomes important so information for each bus clock-cycle information is transferred as one transaction
- Lowest level (most detail)
  - Communication and functional designs are fully described and timed



Ref: SystemC: From the Ground UP, Black, Donovan

Design Verification

- Verification is the task of verifying that the design conforms to specification. It should answer the question “did I build the right thing?”, not “did I build the thing right?”.
- It should check:
  - What is the design intent?
  - What does the design actually do?
  - Do the two things match?
  - How good is the testing? – *Coverage metrics*

How good is the testing?

- Aim is to know how well that the design intent has been achieved – not to know how well the implementation has been tested
- For a design of non-trivial complexity, cannot guarantee all functional bugs have been removed. Can use different methods to reduce bugs but… Also have to make decision when it's "good-enough"
- Code-coverage tools will report estimate of coverage (quality)
- How does they do this? Where do metrics come from?

Design verification (HW)

- ***Code Coverage***
  - Code coverage reflects how thorough the HDL code was exercised
  - Code coverage is a necessity. It is unacceptable to synthesize dead or unverified code
- ***Functional Coverage***
  - Functional coverage is more concerned with what the design does rather than how. It is generally done at a higher level of abstraction
  - Low-level details are often hidden from the report reviewer
- Functional and code coverage are complementary in nature

Verification – code coverage

- *Code coverage – Toggle coverage*
  - Tracks the value of every net and register and checks to see if it toggles. Some tools need both transitions high-low and low-high to pass.
  - Advantages –
    - very easy to implement and test – any net that hasn't toggled hasn't been exercised – FAIL
  - Disadvantages –
    - a lot of false positives. Most clock-based systems will cause a lot of toggling inside a design. Can indicate a significant coverage figure simply by resetting/clocking e.g. 4 bit counter – all outputs toggled after counting to 5 – tests returns OK even though other states have not been visited.

Verification – code coverage

- *Code coverage – Line Coverage*
  - Measures that each line has been exercised during a simulation run.
  - Advantages
    - again easy to implement and test. Can be more thorough than toggle coverage as at least registers if line has been executed
  - Disadvantages
    - again can return high rate of false positives. Simply clocking/resetting a design can cause a not insignificant amount of code to be executed.
    - dependent on coding style. A line that includes both an "if" and its corresponding "else" statement will pass if either branch is executed

elseif ((down==1) or reset==0) count becomes 0;

Verification – code coverage

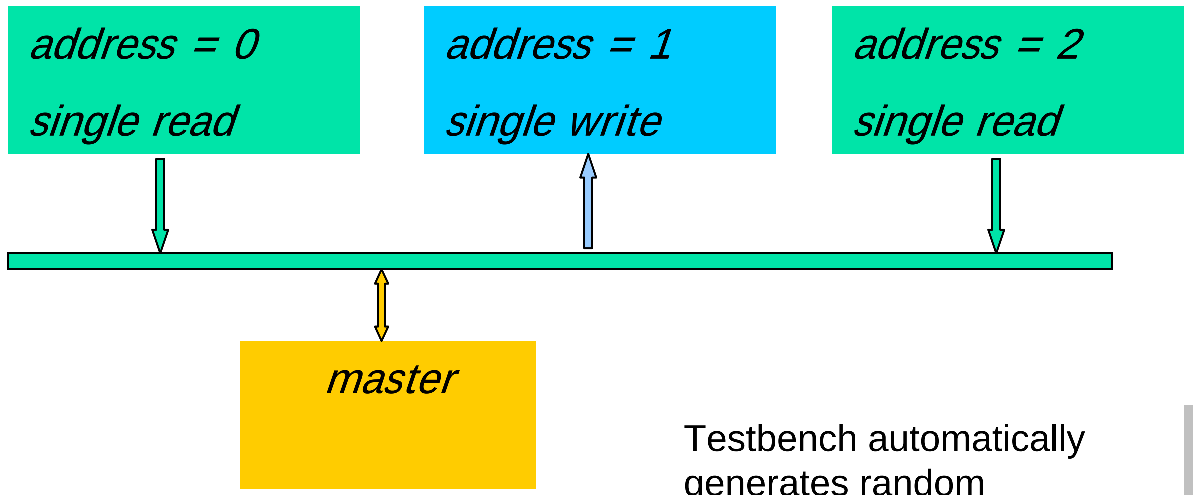
- *Code coverage – branch coverage*
    - Similar to Line coverage but determines if each possible branch on each line of code has been executed.
    - Advantages
        - Looks at many more conditions so more thorough testing

elseif ((down==1) or reset==0) count becomes 0;

Constrained-random data generation

- Simulation has long been used for verification. It has disadvantages:
  - Takes a “long” time to set up a testbench for verify each desired functionality
  - Practically impossible to write testbenches for huge designs
- To overcome these difficulties, *testbench automation* is used. The tool is intelligent enough to generate randomly a set of test patterns for different scenarios
- Verification languages (available in SystemC, SystemVerilog) allow the user to limit the generated tests. This is done using *constraints* – this ensures that the randomly produced test is valid.
- Re-running the simulation (with a different random seed) causes a different but still valid stimuli to be generated

Constrained-random data generation



Testbench automatically generates random read/write commands to an address. Only valid tests are eventually performed.

address	mode	
0	read	
0	write	X
1	write	
2	read	

Coverage-driven Verification

- From previous example, it is impossible to know which addresses and modes are exercised until the test is run. Recording the actual tests gives a *functional coverage* metric. This can be used to determine if a particular test verified a given feature, and to what extent. This information is then used to determine what to do next => *coverage-driven verification.*
- CDV is used to produce randomly multiple scenarios from a single test. Coverage data metrics are used to modify the constraints to try and uncover unexpected situations (corner cases).
- For CDV to work well, the user must specify the *coverage points* (or specific behaviours) that must be exercised
- How do we do this? - *Assertions*

Coverage-driven Verification

- The metric is produced from the analysis of two sets of data:
- *Data-oriented coverage:*
  - records a set of variable values at a particular point during simulation e.g. sampling the address and bus mode at the start of each bus cycle. Recording typically done during stimulus generation to verify that all transfer types to all address spaces have been created
  - can be used to record that the device responded with the correct results

  R*esult often shown as a matrix noting how many times each variable achieved a particular value when other variables had specific values. The number of occurrences of each unique combination is reported.*

- *Control-oriented coverage* records specific temporal behaviour
  - if a user-defined specific case has been exercised e.g. the data B was received less than 5 clocks after the read_request (in this last case, it can also record the actual result)

- This type of testing is often done using *assertions*.

Assertion-based verification

- *Assertions* are an important part of both simulation and *formal model checking* (see below).
- *Assertions* are concise, mathematically precise descriptions of behaviour that must hold or that constrain the operation of a design or block. e.g. handshake must follow request, data_ready must be present 3 to 5 cycles after bus_read
- In addition, they can also describe *coverage points -* cases that must be exercised by the verification process
- White-box and black-box assertions can be used
  - White-box assertions assume some knowledge of the inner-workings of the design.
  - Black-box assertions describe “general” behaviour and are independent of the actual implementation
- How many assertions are “enough”?
  - Enough to ensure that all the desired critical behaviours have been verified
  - Enough to ensure that all parts of the design are adequately related to one or more assertions. Rule of thumb: there should be sufficient *assertion density* such that every tested value propagates to an assertion within 2-3 clock cycles.

Formal Model Checking

- *Formal Model Checking* is an exhaustive examination all of all the possible states of a design to determine if any of them violate a specified set of properties. It is done through mathematical analysis
- Properties (often assertions) describes a precise description of sequential (“out happens after in”) or invariant (“out =‘0010’ will never happen”) behaviours about the design. Each property is considered a part of the specification
- While theoretically possible to use FMC to fully verify the functionality of an arbitrarily complex design, it is typically used for sub-blocks of a design
- FMC can exhaustively analyse all of the possible states the design can reach, without requiring the user to write a testbench.
- An advantage of using FMC at the block level is that it can be re-used in a larger system
- *cf equivalence checking - Equivalence checking refers to comparing two different implementations of a design to see if they are functionally equivalent.*

Formal Model Checking and Simulation

- FMC is a complementary technology to simulation.
- Properties defined at a block level can be used as monitors in simulation (assumes consistency between FMC and simulation)
- If a block has been proven to behave correctly for a set of inputs, and the block is only driven by inputs in this range, it is guaranteed to work in the complete system
- => for full-chip simulation, unnecessary to create scenarios that will stress the block. Simulation can focus on the end-to-end behaviour, knowing that the previously verified intermediate block is OK
- Theoretically, FMC by itself can be extended to full hierarchical design. Block2 has been formally verified to work correctly for a restricted set of inputs. Block1 drives Block2. Block1 has been shown to provide only inputs to Block2 within its restricted set of verified values => all OK. This is the *assume-guarantee* paradigm of formal verification.
- Now must guarantee that Block1's inputs can only be in the verified range etc. Theoretically simple, realisation of the assume guarantee relationship quickly becomes impractically complex.
- However, FMC can be extremely valuable in finding bugs due to its exhaustive analysis. As it is exhaustive, FMC will show up design errors (and incorrect assertions)

Coverage-driven Verification/ Formal Model Checking

- CDV generates random stimuli to automatically produce multiple tests for a given case. The inherent randomness can uncover corner cases (but might not). FMC is an exhaustive analysis of a model
- The two methods are combined in *Dynamic Formal Verification*
- Particularly useful for known "dangerous" cases e.g. FIFO is full. Using CDV only, FIFO_full might be tested but some other condition at the same time could cause a bug (unlikely that both are present using CDV alone)
- => When FIFO_full is recorded, perform a formal analysis on all states leading up to and beyond this time
- Has advantages that verification time is targeted on potentially dangerous situations

Electronic System Level Tools

- There are a number of tools and methodologies available for ESL design
- The two most prevalent “languages” are SystemC and SystemVerilog (but there are some others)
- Different tool vendors focus on different products

Introduction to SystemC

- SystemC is a C++ library used for supporting system level modeling. It is particularly strong in supporting various abstraction levels and can be used for fast, efficient designs and verification.
- The SystemC library is provided by the Open SystemC Initiative, a non-profit independent organisation. OSCI is composed of numerous companies, universities and individuals, all aiming to develop and standardize the language.
- SystemC libraries were developed to allow C++ to "understand" time and concurrent processes. Hardware specific ideas were also added: signals, ports.

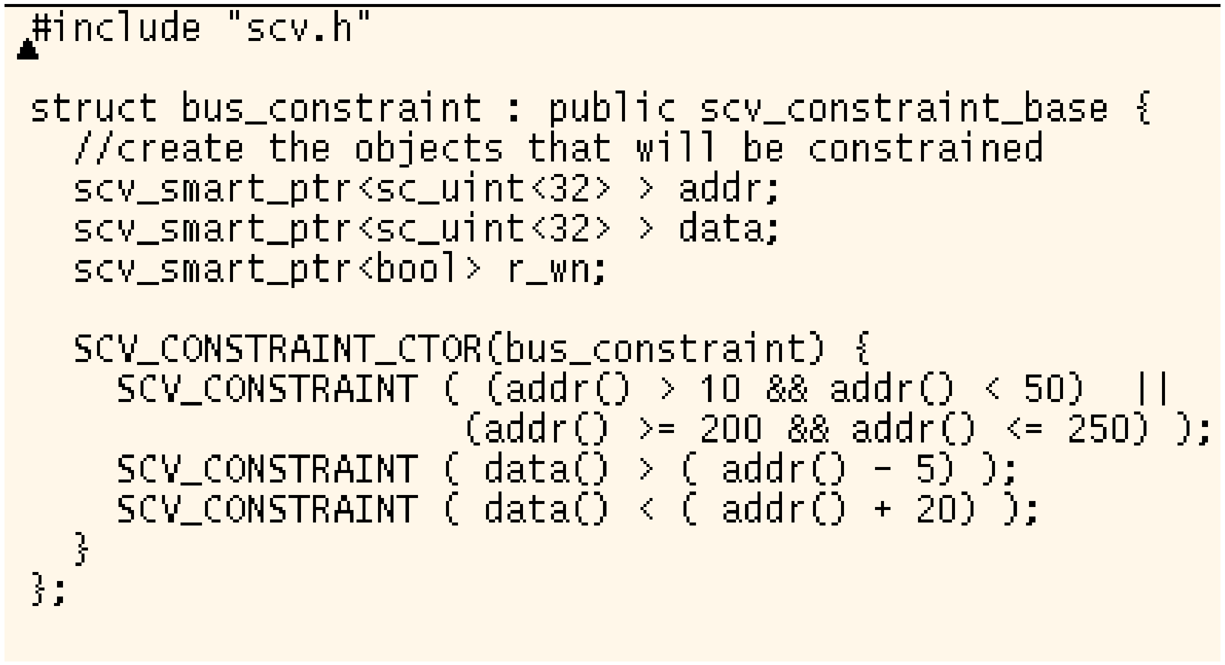
  They also define data types dedicated to hardware modelling e.g. bit, vector, fixed point types. Core language elements such as modules, processes, events, channels, event driven simulation kernel are available.
- Elementary channels such as signals or FIFOs are provided to implement communication mechanisms between concurrent objects
- A basic SystemC system is available freely from OSCI. Numerous EDA vendors provide their implementations of the SystemC language and support for mixed languages simulations. Other additional libraries are made available from the OSCI site.

SCV library

- SystemC verification library available for download from www.systemc.org.
- Provided library examples:
  - “This is a very simple example to show how a user can create a distribution of values when randomizing an object. In this case, a distribution is created for an enumerated data type. The distributed values that are generated will fall in the distribution that is applied.”
  - “This is a very simple example to show how a user can create a special enumerated data type. SystemC SCV uses a special C++ methodology called partial template specialization to support arbitrary data types with randomization, transaction recording, callbacks, and other features that deconstruct the elements of an object.”
  - “This is a very simple example to show how a user can create a distribution range to apply to the randomization of an object. The idea of creating ranges of values is to set up buckets of values and create probabilities over the buckets. That is, the distribution within a bucket is uniform, but the particular bucket being selected is set by the distribution. This technique can be used to create approximations of non-linear distributions.”

SCV library

- Example code: **addr** variable is limited between 10 - 50 and 200 - 250, while data is limited to be between addr-5 and addr+20**.**



```
#include "scv.h"

struct bus_constraint : public scv_constraint_base {
  //create the objects that will be constrained
  scv_smart_ptr<sc_uint<32> > addr;
  scv_smart_ptr<sc_uint<32> > data;
  scv_smart_ptr<bool> r_wn;

  SCV_CONSTRAINT_CTOR(bus_constraint) {
    SCV_CONSTRAINT ( (addr() > 10 && addr() < 50)  ||
                     (addr() >= 200 && addr() <= 250) );
    SCV_CONSTRAINT ( data() > ( addr() - 5) );
    SCV_CONSTRAINT ( data() < ( addr() + 20) );
  }
};
```

SystemVerilog

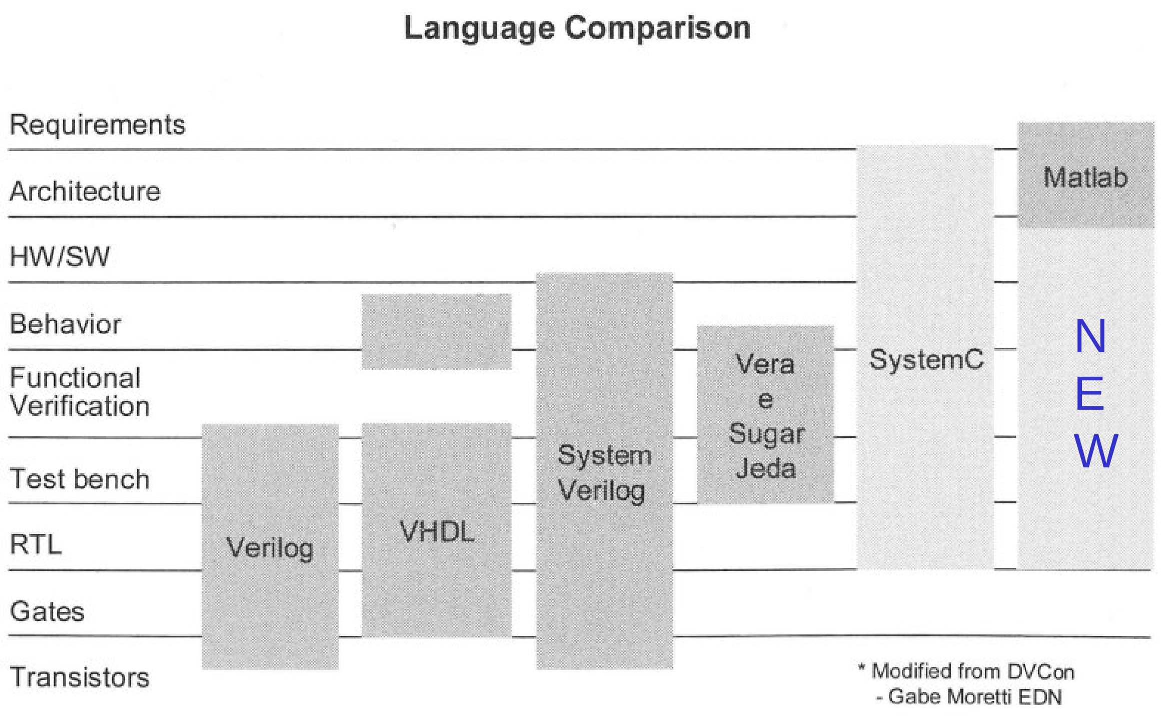
- SystemVerilog is a hardware description language with extensive verification capabilities. The language is based on Verilog with donations from several sources:
  - SUPERLOG Extended Synthesizable Subset from Co-Design Automation
  - OpenVERA verification language from Synopsys
  - OpenVERA Assertions from Synopsys
  - PSL assertions (began as a donation of Sugar assertions from IBM)
  - DirectC and coverage Application Programming Interfaces from Synopsys
  - Compilation and other extensions from Mentor Graphics
  - High-level language features from BlueSpec

- Accellera released a specification for SystemVerilog in 2003. The language has been eventually adopted as IEEE Standard 1800-2005 (in 2005)
- SystemVerilog has few unique capabilities compared to other languages but has the advantage that all aspects of the design and verification flow: design description, functional simulation, property specification, and formal verification, are based on one (very popular) HDL => led to easier user acceptance

Accellera "organigramme"

- A brief outline...
- Accellera formed by merger of OVI and VI in 2001
- Vera was an entire verification environment. Synopsys promoted it as a standalone tool. Synopsys then made available OpenVera and OpenVera Assertions (OVA)
- Due to interest in formal verification, Accellera looked for a language to standardize – they eventually chose IBM's Sugar language (became PSL)
- Synopsys then donated OVA to the Accellera committee in charge of SystemVerilog
  - *btw yet another committee looks after the Open Verification Library (available for both VHDL and Verilog)*
- VHDL200X being defined => ~ VHDL + better verification

How do languages compare?



Ref: based on figure from “SystemC: From the Ground Up”, Black, Donovan, 2003

“ESL” examples

- **HEP examples**
- Unified C/VHDL Model Generation of FPGA-based LHCb VELO algorithms / Muecke, Manfred; Szumlak, Tomasz
  http://doc.cern.ch//archive/cernrep/2007/2007-001/p492.pdf

  Also, poster presented at 12th Workshop on Electronics for LHC and Future Experiments (LECC'06), 25-29 September 2006, Valencia SPAIN”
  http://engineering-software.web.cern.ch/engineering-software/poster2006/posters/muecke.pdf

- “From Behavioral to RTL Design Flow in SystemC”, E. Vaumorin, T. Romanteau
  http://www.us.design-reuse.com/articles/article7354.html

- “Modeling of the architectural studies for the PANDA DAT system”, K. Korcyl, W. Kuehn, J. Otwinowski, P. Salabura, L. Schmitt
  http://conferences.fnal.gov/cgi-bin/rt2007/download.pl?paper_id=RTA-NEW04.pdf

- **Industry EDA vendor tools and usage**

“ESL” examples in HEP

- “Unified C/VHDL Model Generation of FPGA-based LHCb VELO algorithms” / Muecke, Manfred; Szumlak, Tomasz
  http://doc.cern.ch//archive/cernrep/2007/2007-001/p492.pdf

- “We show an alternative design approach for signal processing algorithms implemented on FPGAs. Instead of writing VHDL code for implementation and maintaining a C-model for algorithm simulation, we derive both models from one common source, allowing generation of synthesizable VHDL and cycle and bit-accurate C-Code. We have tested our approach on the LHCb VELO pre-processing algorithms and report on experiences gained during the course of our work.”

- Tools used – Confluence, HDCaml
  http://www.confluent.org/wiki/doku.php/hdcaml

“ESL” examples in HEP

- “From Behavioral to RTL Design Flow in SystemC”, E. Vaumorin, T. Romanteau
  http://www.us.design-reuse.com/articles/article7354.html

- “This paper reports the scientific collaboration between LLR and PROSILOG. The aim of this collaboration was to show the possibility to quickly implement a system into a FPGA, using SystemC as the unique description language. Starting from behavioral abstraction level, the model, before hardware synthesis, is refined down to RTL then automatically translated to the equivalent model into VHDL or Verilog.

  It will be shown that this design flow is less time consuming, more efficient and more reliable than the traditional C++ to HDL flow. “

“ESL” examples in HEP

- “Modeling of the architectural studies for the PANDA DAT system”
  - K. Korcyl, W. Kuehn, J. Otwinowski, P. Salabura, L. Schmitt *https://appora.fnal.gov/pls/rt07/JACoW.view_abstract?abs_id=1073*
- “Abstract— We present design studies of the DAQ and trigger system (DAT) for the PANDA detector proposed for the new FAIR facility at GSI. The broad physics program of PANDA requires a novel DAT system able to cope with high interaction rates (up to 2*10^7/s) and to trigger on various event topologies simultaneously. We used SystemC as modeling platform to investigate a candidate architecture for the PANDA DAT system.

  We simulated the behavior of the complete system with simplified models of all the components. The model covers detector buffers connected via Ethernet to farms of computing nodes constituting two filtering levels and an event building level. We present results from modeling illustrating the impact of the key architectural choices and parameters on the overall performance.”

Electronic System Level Tools

- “Usual” suspects, Cadence, Synopsys, Mentor
- Others:
- Celoxica (celoxica.com)
  - Agility Compiler - SystemC Behavioural Design and Synthesis
  - Handel-C - superset of ANSI-C
  - Xilinx ESL Starter Kit **-** C-based design for Xilinx FPGA
- CoWare (coware.com) Platform Architect, Model Designer, Virtual Platform Designer
- SystemCrafter (systemcrafter.com)
  - SystemCrafter SC “... a high-performance SystemC synthesis tool for Xilinx FPGAs. It synthesizes the industry standard SystemC language to RTL VHDL, and so allows you to design, debug and simulate hardware and systems using the SystemCrafter GUI or your existing C++ development environment. The breakthrough price of $2995 brings SystemC synthesis within reach of everyone.”

Survey results - HDL

- **2005** - "Does your project do mixed Verilog/VHDL simulations?"
  - Verilog only : ############################### 59%
  - mixed : ################### 38%
  - VHDL only : # 3%

- 2007 - "Does your project do mixed Verilog/VHDL simulations?"
  - Verilog only : ############################# 55.3%
  - mostly Verilog : ######### 18.0%
  - both equally : ### 6.5%
  - mostly VHDL : ######## 16.4%
  - VHDL only : ## 4.0%

"The VHDL stalwarts were mostly US military contractor companies plus some (not all, but some) European companies -- with the rest of the world being Verilog oriented. The biggest reason why there were VHDL stalwarts were due to legacy code reasons."

results from 818 engineers, www.deepchip.com

Survey results – HDL simulators

- **2005** - "Whose Verilog or VHDL simulator(s) do you currently use?"
  - Cadence NC-Sim : ########################### 27%
  - NC-Verilog : ##################### 21%
  - Verilog-XL : ## 2%
  - NC-VHDL : # 1%
  - Synopsys VCS : ### 43%
  - VCS-MX : #### 4%
  - Mentor ModelSim : ################################### 35%
  - Others : ##### 5%
- **2007** - "Whose Verilog or VHDL simulator(s) do you currently use?“
  - Cadence NC-Sim : ######################## 24.3%
  - NC-Verilog : ################## 18.0%
  - Verilog-XL : # 0.7%
  - NC-VHDL : # 1.1%
  - Synopsys VCS : ### 44.7%
  - VCS-MX : ######### 8.5%
  - Mentor ModelSim : ################################### 35.3%
  - Others : #### 4%

results from 818 engineers, www.deepchip.com

Survey results- SystemC

- **2005** - "Do you see your project using SystemC in the next 6 months?"
  - yes : ##################### 42%
  - no : ############################# 58%
- **2007** - "Is your project using SystemC? (Yes/No)"
  - yes : ########### 23.0%
  - no : ####################################### 77.0%

- **2005** - "Are you using SystemC for high level modeling, or verification, or for design?“
  - high level modeling : ####################### 70%
  - verification : ##################### 62%
  - design : ## 7%
- **2007** - "Are you using SystemC for high level modeling, or for verification, or for design? (Choose all that apply)"
  - high level modeling : ######################### 73.7%
  - verification : ###################### 64.2%
  - design : ## 5.8%

results from 818 engineers, www.deepchip.com

Survey results- SystemC tools

- **2007** - "Whose specific SystemC tools are you using?"
  - Cadence NC-SystemC : ################################## 33.6%
  - Cadence TestBuilder : # 0.9%
  - CoWare : ####### 6.5%
  - Free OSCI : ### 43.0%
  - Mentor ModelSim : ################# 16.8%
  - Mentor Summit Vista : ########### 11.2%
  - Synopsys : ################ 16.0%
  - Mentor Catapult C : 0%
  - Forte Cynthesizer : #### 3.7%%
  - Synfora Pico : 0%
  - all others combined : ##### 4.7%

results from 818 engineers, www.deepchip.com

Survey results- SystemVerilog

- **2005** - Do you see your project using SystemVerilog in the next 6 months?
  - yes : ########## 19%
  - no : ## 81%
- **2007** - Excluding assertions, is your project using SystemVerilog? (Y/N)
  - yes : ################## 35.1%
  - no : ################################ 64.9%

- **2005** - Do you plan on using the SystemVerilog design or the verification extensions, or both?
  - verification : #################################### 70%
  - design : ## 5%
  - both : ############# 26%
- **2007** - Are you using SystemVerilog for testbench or design or both?
  - testbench : ### 80.2%
  - design : ## 4.1%
  - both : ######## 15.8%

results from 818 engineers, www.deepchip.com

Survey results- SV tools

- **2005** - Whose System Verilog tools are you using?
  - Synopsys VCS : ## 79%
  - Mentor ModelSim : ######## 15%
  - Cadence : ### 6%
- 2007 - Whose specific System Verilog tool(s) are you using? (Include everything from simulators to synthesis.)
  - Synopsys DC : ##### 10.6%
  - Synopsys VCS : ################################## 65.6%
  - Synopsys Leda : # 1.8%
  - Synopsys Formality : # 2.6%
  - Synopsys VMM : ### 6.2%
  - Synopsys Magellan : 0.4%
  - Mentor ModelSim : ###### 12.3%
  - Mentor Questa : ######## 15.0%
  - Mentor AVM : # 0.9%
  - Cadence NC-Sim : ############ 24.7%
  - Cadence RTL Compiler : # 0.9%
  - Atrenta Spyglass : # 1.8%
  - Novas Verdi : # 2.2%
  - all others : # 1.3%

results from 818 engineers, www.deepchip.com

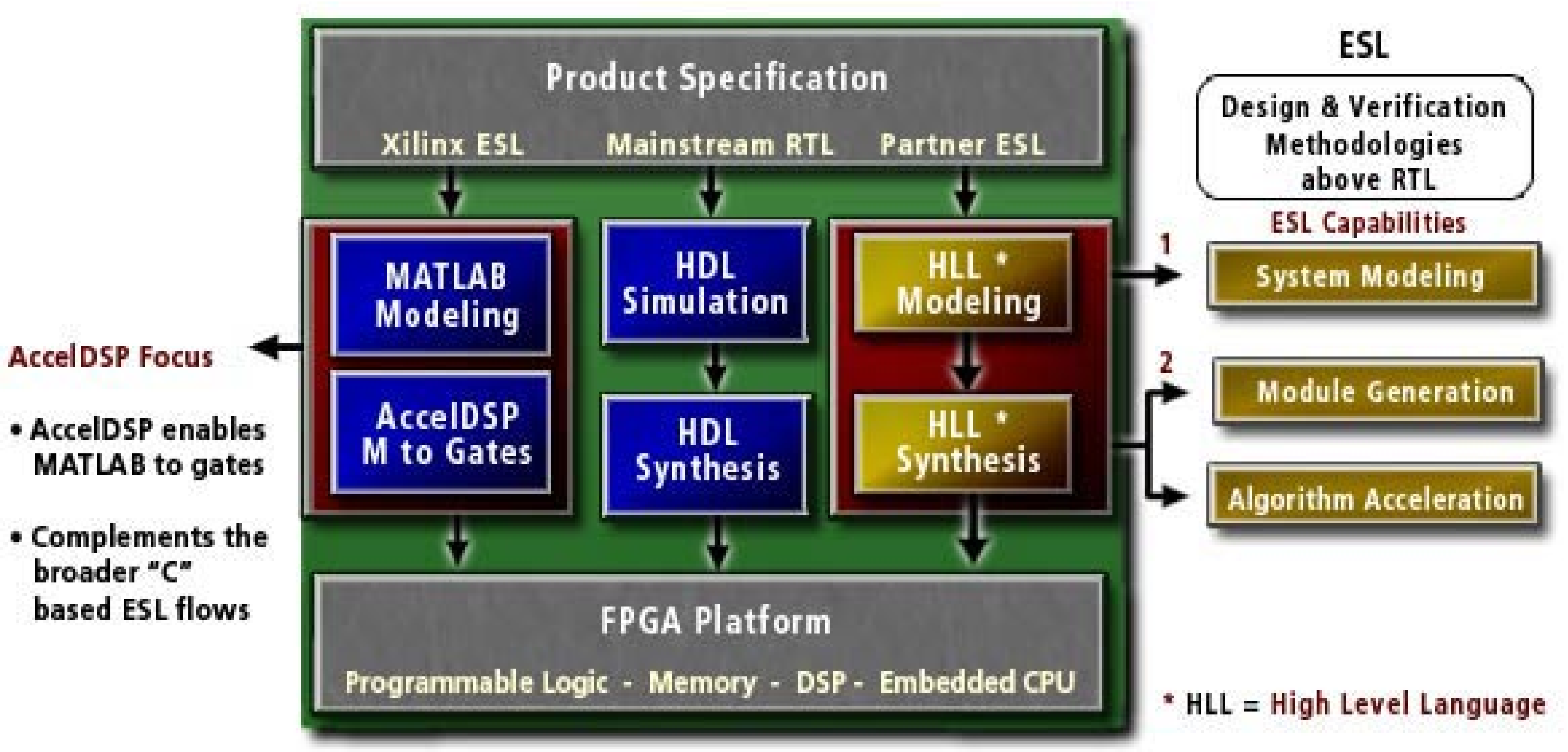
DSP Design Flows

- Two main ways of programming DSP functions – DSP or FPGA
  - A DSP is a specialised microprocessor – typically suited to extremely complex maths-intensive tasks with conditional processing. Can be limited in performance by clock-rate and/or number of useful operations per clock
  - FPGA can be programmed to perform parallel processing (if gates available) => can be very fast
- Rules of thumb
  - Low sampling rates, complex programs => DSP
  - High-sampling rates, lower complexity tasks => FPGA
- In both cases, use high-level languages to provide:
  - High productivity
  - Portability
  - Maintainability
  - Code reuse
  - Optimising system cost / performance
  - Rapid prototyping and algorithm proving
  - Integration with real-time kernels and operating systems
  - Ease of debug
  - Availability of algorithms

DSP design Tools

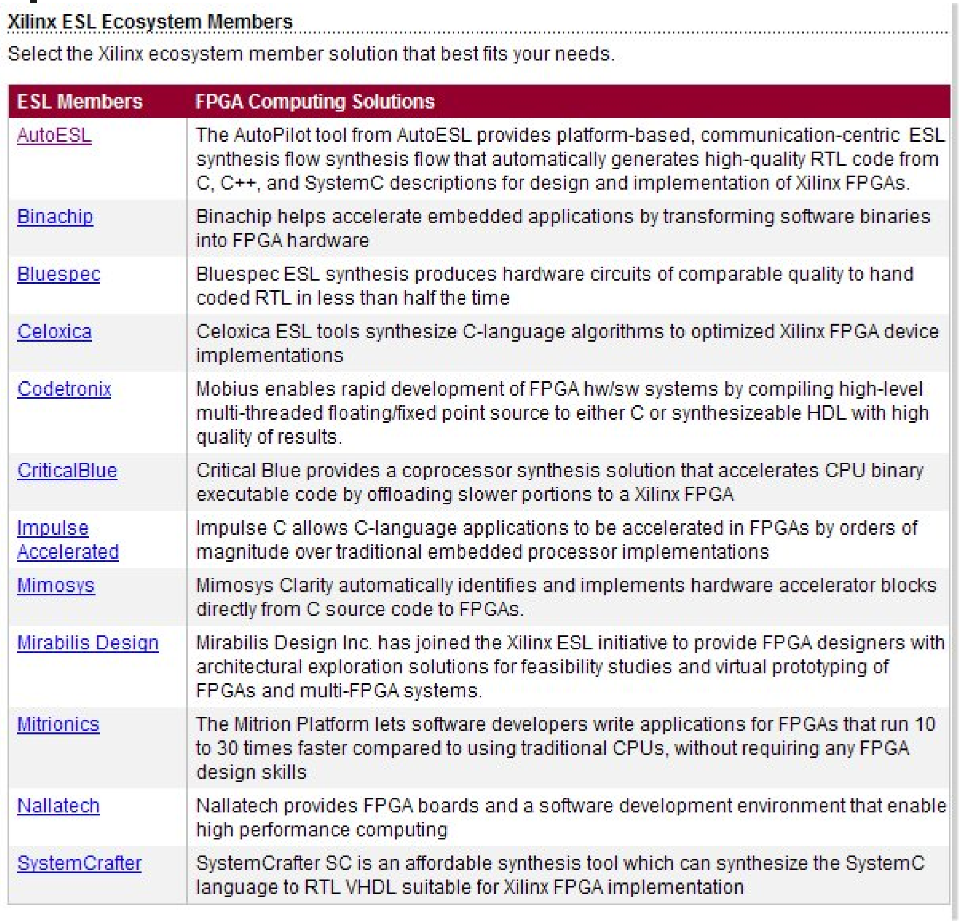
- There exists many tools for DSP design flows: CoWare Signal Processing Designer ("SPW"), Ptolemy, VisSim
- Why did we chose MATLAB/Simulink for the school?
  - Tools available to make it easier to go directly from algorithm to FPGA
  - Tools readily (and rather cheaply) available
  - Learning curve - "everybody" knows MATLAB/Simulink

Xilinx and ESL



www.xilinx.com/esl

Xilinx and ESL



Xilinx ESL Ecosystem Members

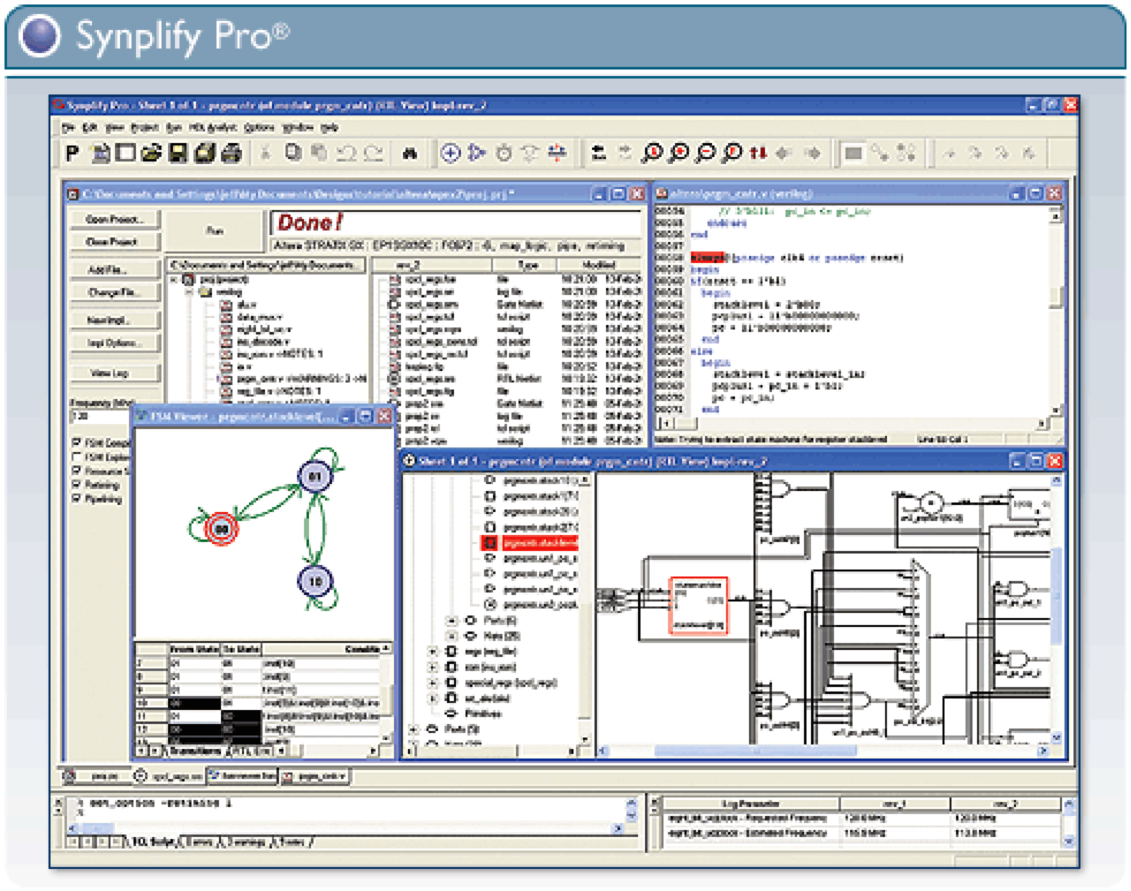
Select the Xilinx ecosystem member solution that best fits your needs.

ESL Members	FPGA Computing Solutions
AutoESL	The AutoPilot tool from AutoESL provides platform-based, communication-centric ESL synthesis flow synthesis flow that automatically generates high-quality RTL code from C, C++, and SystemC descriptions for design and implementation of Xilinx FPGAs.
Binachip	Binachip helps accelerate embedded applications by transforming software binaries into FPGA hardware
Bluespec	Bluespec ESL synthesis produces hardware circuits of comparable quality to hand coded RTL in less than half the time
Celoxica	Celoxica ESL tools synthesize C-language algorithms to optimized Xilinx FPGA device implementations
Codetronix	Mobius enables rapid development of FPGA hw/sw systems by compiling high-level multi-threaded floating/fixed point source to either C or synthesizeable HDL with high quality of results.
CriticalBlue	Critical Blue provides a coprocessor synthesis solution that accelerates CPU binary executable code by offloading slower portions to a Xilinx FPGA
Impulse Accelerated	Impulse C allows C-language applications to be accelerated in FPGAs by orders of magnitude over traditional embedded processor implementations
Mimosys	Mimosys Clarity automatically identifies and implements hardware accelerator blocks directly from C source code to FPGAs.
Mirabilis Design	Mirabilis Design Inc. has joined the Xilinx ESL initiative to provide FPGA designers with architectural exploration solutions for feasibility studies and virtual prototyping of FPGAs and multi-FPGA systems.
Mitrionics	The Mitrion Platform lets software developers write applications for FPGAs that run 10 to 30 times faster compared to using traditional CPUs, without requiring any FPGA design skills
Nallatech	Nallatech provides FPGA boards and a software development environment that enable high performance computing
SystemCrafter	SystemCrafter SC is an affordable synthesis tool which can synthesize the SystemC language to RTL VHDL suitable for Xilinx FPGA implementation

- Electronic System Level (ESL) design refers to evolving design and verification methodologies that begin at a higher level of abstraction than the current mainstream Register Transfer Level (RTL). Many of the ESL design languages are closer in syntax and semantics to the popular ANSI C than to hardware languages like Verilog and VHDL.
- A key focus of many of the ESL tools for FPGAs is to empower designers with software programming skills to be able to easily implement their ideas in programmable hardware without having to learn traditional hardware design techniques.
- A wide array of FPGA optimized ESL solutions are available from Xilinx ecosystem members.

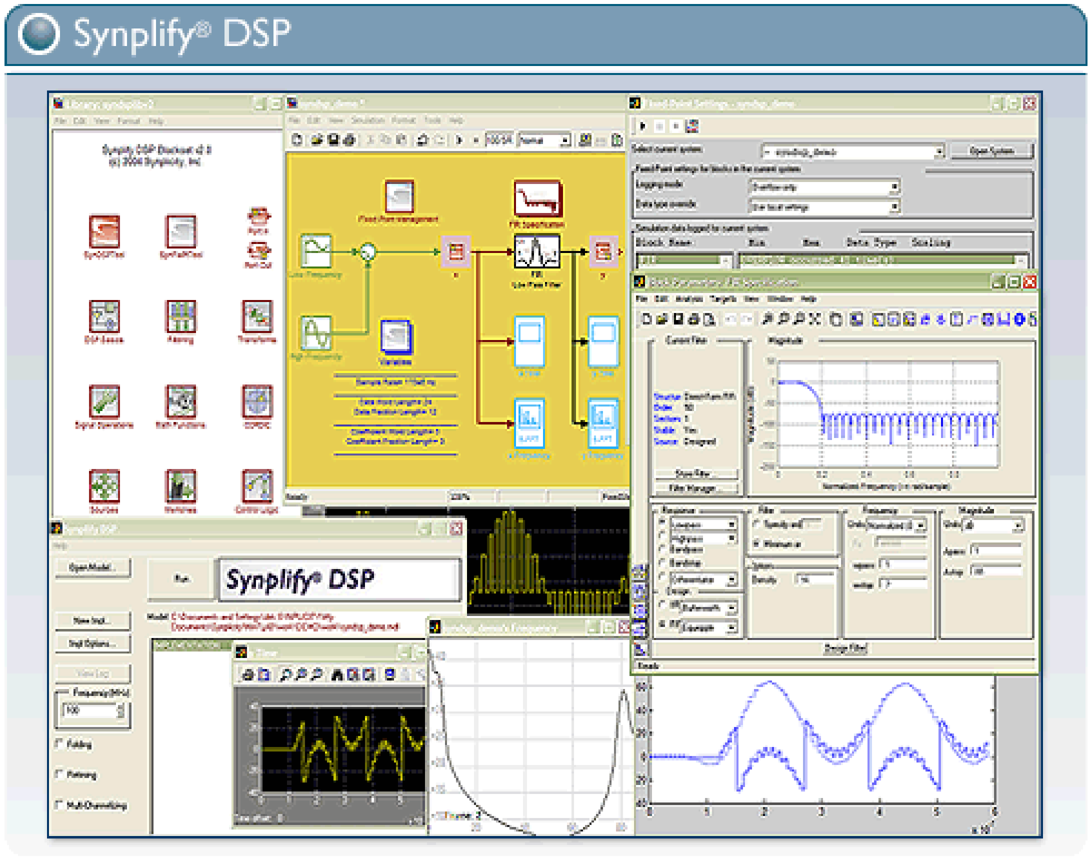
www.xilinx.com/esl

Synplify PRO



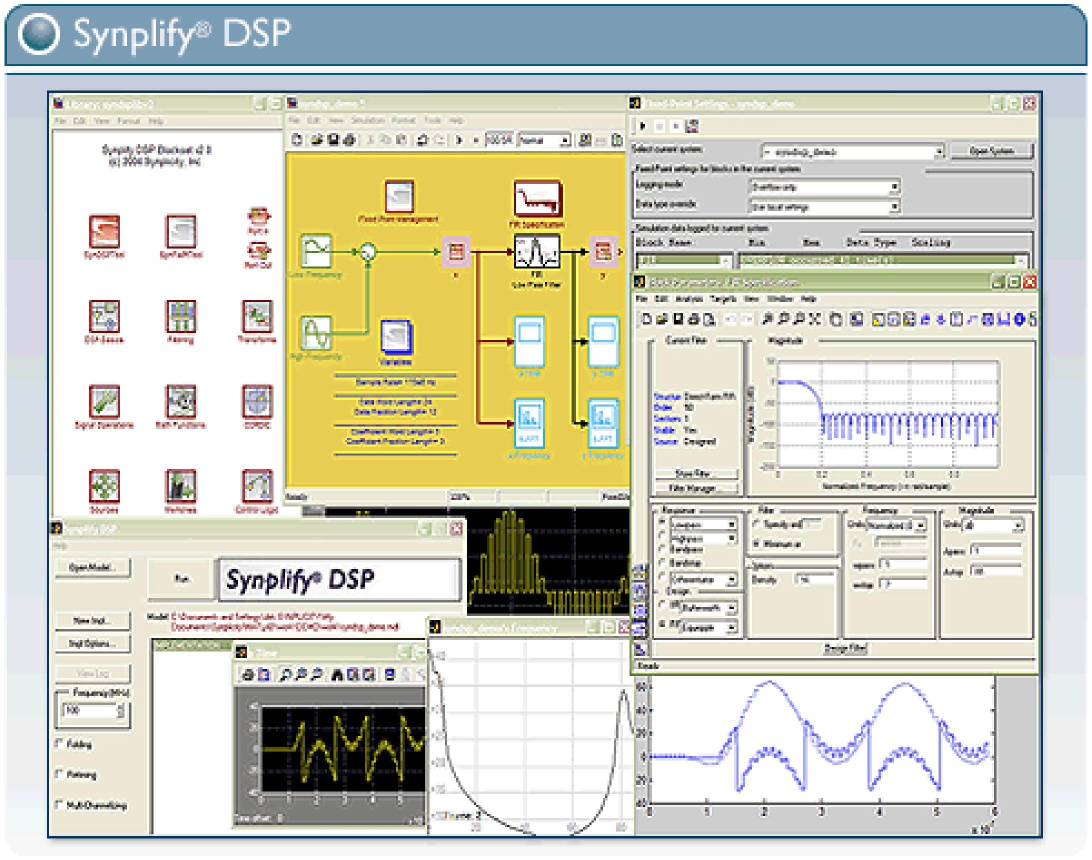
- High-density field programmable gate arrays (FPGAs) can contain millions of gates and operate at speeds in excess of 100 MHz.
- At this level of complexity, schedules, budgets and FPGA design tools all begin to feel the burden.
- By using the Synplify Pro solution, you can push the performance of challenging and complex designs while remaining comfortably on or ahead of schedule
- The Synplify solution is a high-performance, sophisticated logic synthesis engine that delivers fast, highly efficient FPGA and CPLD designs.
- The Synplify product takes Verilog and VHDL Hardware Description Languages as input and outputs an optimized netlist in most popular FPGA vendor formats

Synplify DSP



- DSP designers are targeting FPGA and ASIC hardware for implementation of their high-performance DSP designs. FPGAs and ASICs can achieve a performance of hundreds of millions of operations per second.
- Today's FPGAs contain large quantities of DSP blocks and multipliers facilitating efficient and parallel implementation of DSP functions in programmable logic. High volume DSP applications frequently use ASIC devices.
- There has been no good way to get a design specified at the algorithm level from tools such as MATLAB®/Simulink® by The MathWorks, into high-quality RTL code. A common implementation path has been to hand-code the RTL with numerous iterations between the DSP algorithm architect and the RTL hardware designer, which is error prone and time consuming.

Synplify DSP



- Synplify DSP software is a true DSP synthesis tool and the only one that performs high-level DSP optimizations from a Simulink specification.
- These special DSP optimizations allow designers to capture the behaviour needed for their DSP algorithm without worrying about the specific implementation in hardware
- The Synplify DSP solution automatically produces a highly optimized, technology independent implementation of the design ready for RTL synthesis.

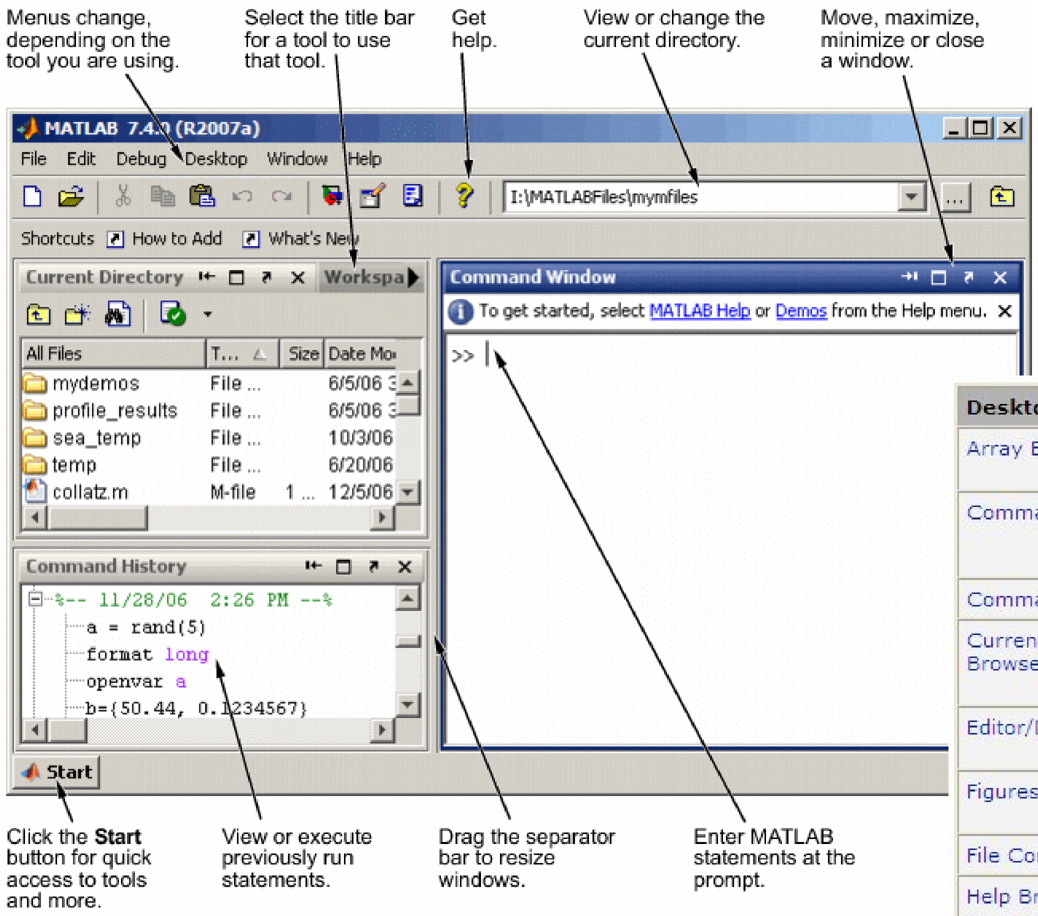
Xilinx AccelDSP

DSP modelling – Design, architectural exploration, and debug of high-level DSP algorithms with MATLAB for Xilinx FPGAs to reduce design cycles and costs.
IP-Explorer Technology – Heuristic-driven selection of hardware architecture at the algorithmic level to produce system-optimized designs.
Automated floating- to fixed-point conversion – Automated word width selection and propagation for floating- to fixed-point conversion.
Automatic code generation of synthesizable VHDL or Verilog – Bit-accurate code generated after fixed-point design meets system specifications.
Verification of bit-accuracy – Comparison of RTL and post-place and route model for automatic verification.
C++ simulation model generation – Improved simulations speeds of 1000x over standard fixed-point MATLAB.
System Generator integration – Generated blocks can be exported to System Generator for inclusion in a larger system.
Third party integration – Access to and integration of third party simulation and synthesis tools to simplify the design flow for algorithm designers unfamiliar with RTL simulation and synthesis tools.

Introduction to MATLAB

- MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Using MATLAB, you can solve technical computing problems faster than with traditional programming languages, such as C, C++, and Fortran.
- You can use MATLAB in a wide range of applications, including signal and image processing, communications, control design, test and measurement, financial modeling and analysis, and computational biology. Add-on toolboxes (collections of special-purpose MATLAB functions, available separately) extend the MATLAB environment to solve particular classes of problems in these application areas.

www.mathworks.com



Menus change, depending on the tool you are using.
Select the title bar for a tool to use that tool.
Get help.
View or change the current directory.
Move, maximize, minimize or close a window.
MATLAB 7.4.0 (R2007a)
File Edit Debug Desktop Window Help
I:\MATLABFiles\mymfiles
Shortcuts How to Add What's New
Current Directory
Workspa
Command Window
To get started, select MATLAB Help or Demos from the Help menu.
All Files
mydemos
profile_results
sea_temp
temp
collatz.m
Command History
Start
Click the **Start** button for quick access to tools and more.
View or execute previously run statements.
Drag the separator bar to resize windows.
Enter MATLAB statements at the prompt.

```
%-- 11/28/06  2:26 PM --%
a = rand(5)
format long
openvar a
b={50.44, 0.1234567}
```

Desktop Tool	Description
Array Editor	View array contents in a table format and edit the values.
Command History	View a log of or search for the statements you entered in the Command Window, copy them, execute them, and more.
Command Window	Run MATLAB statements.
Current Directory Browser	View files, perform file operations such as open, find files and file content, and manage and tune your files.
Editor/Debugger	Create, edit, debug, and analyze M-files (files containing MATLAB statements).
Figures	Create, modify, view, and print MATLAB figures.
File Comparisons	View line-by-line differences between two files.
Help Browser	View and search the documentation and demos for all your MathWorks products.
Profiler	Improve the performance of your M-files.
Start Button	Run tools and access documentation for all your MathWorks products, and create and use MATLAB shortcuts.
Web Browser	View HTML and related files produced by MATLAB.
Workspace Browser	View and make changes to the contents of the workspace.

Introduction to Simulink

- **Introduction**
- Simulink is a platform for multidomain simulation and Model-Based Design of dynamic systems. It provides an interactive graphical environment and a customizable set of block libraries that let you accurately design, simulate, implement, and test control, signal processing, communications, and other time-varying systems.

  Add-on products extend the Simulink environment with tools for specific modelling and design tasks and for code generation, algorithm implementation, test, and verification.

  Simulink is integrated with MATLAB, providing immediate access to an extensive range of tools for algorithm development, data visualization, data analysis and access, and numerical computation.

www.mathworks.com

Systems and Solvers

- Continuous-state systems are history-dependent dynamic systems that update continuously. Most natural processes and physical systems are continuous-states
- Discrete-state systems are updated in steps separated by a finite time interval

Fixed vs. variable step solvers

- Simulink solvers fall into two basic categories: fixed-step and variable-step.
- Fixed-step solvers solve the model at regular time intervals from the beginning to the end of the simulation. The size of the interval is known as the step size. You can specify the step size or let the solver choose the step size. Generally, decreasing the step size increases the accuracy of the results while increasing the time required to simulate the system.
- Variable-step solvers vary the step size during the simulation, reducing the step size to increase accuracy when a model's states are changing rapidly and increasing the step size to avoid taking unnecessary steps when the model's states are changing slowly. Computing the step size adds to the computational overhead at each step but can reduce the total number of steps, and hence simulation time, required to maintain a specified level of accuracy for models with rapidly changing or piecewise continuous states.

www.mathworks.com

Variable time-step solvers

- A variable-step solver adjusts the step-size to keep the error within acceptable limits. The exact definition of “acceptable” is dependent on the solver type
- ode45
  - This is based on Dormand-Prince (type of Runge-Kutta). ode45 evaluates the integral using both a fourth-order and a fifth-order routine. If the difference between both results is sufficiently small, simulation continues. Otherwise, the time-step is reduced and integrals re-calculated.
- “Acceptable” is varied by rtol (relative error) and atol (absolute error).
  - Calculate rel_error = abs(x) * rtol
  - Error e between two integration values must be less than rel_error or atol (whichever is largest)

Fixed time-step solvers

- A fixed time-step solver can simulate a model to an arbitrary accuracy by making the time-step small enough. However, an unnecessarily small time-step can lead to unacceptably long simulation times

- Using a fixed time-step solver, when is the result “correct”?
  - Simulate using a variable-step solver
  - Simulate with ode1
  - Are the results similar? If not, try ode2 etc. up to ode5. If ode5, still not good enough, reduce the time step

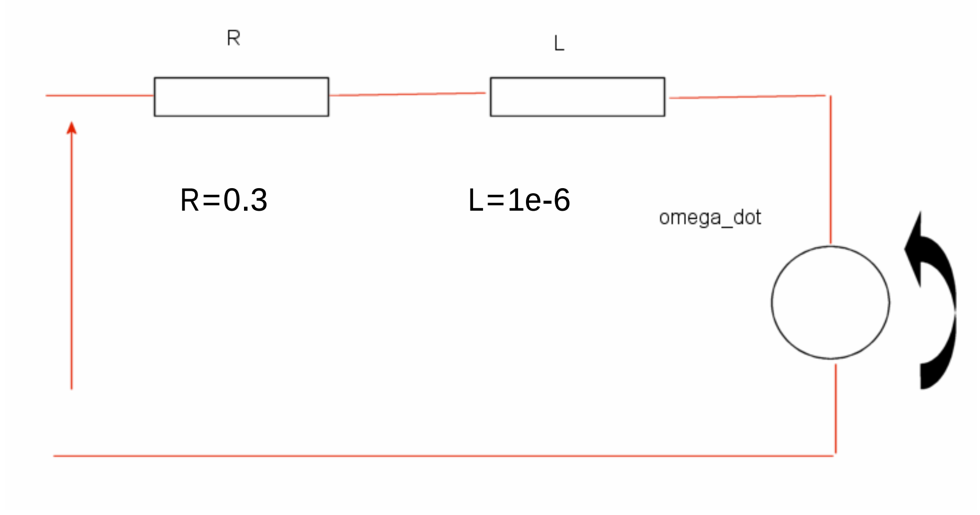
- Continue until sufficient accuracy is achieved with minimum computational time

Fixed vs. variable step solvers

- **Fixed time-step solvers**
  - Do not control integration errors
  - Do not detect discontinuities and events
  - Allow faster simulations
  - Are supported for real-time code generation

- **Variable-step solvers**
  - Control integration errors
  - Detect discontinuities and events
  - Provide more accurate results (but can increase simulation speed)
  - Are not supported for real-time code generation

Stiff solvers



J=6 kg*m2

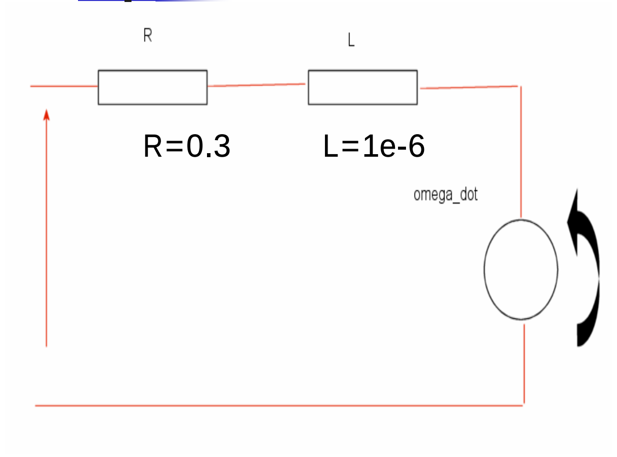
K=0.5 N*m/Amp

k=0.5 V*s/rad

$$\begin{cases} L\dot{i}(t) = -k\omega - Ri + V \\ J\dot{\omega}(t) = Ki \\ y(t) = x(t) \end{cases}$$

Stiff solvers

$$\begin{cases} L\dot{i}(t) = -k\omega - Ri + V \\ J\dot{\omega}(t) = Ki \\ y(t) = x(t) \end{cases}$$

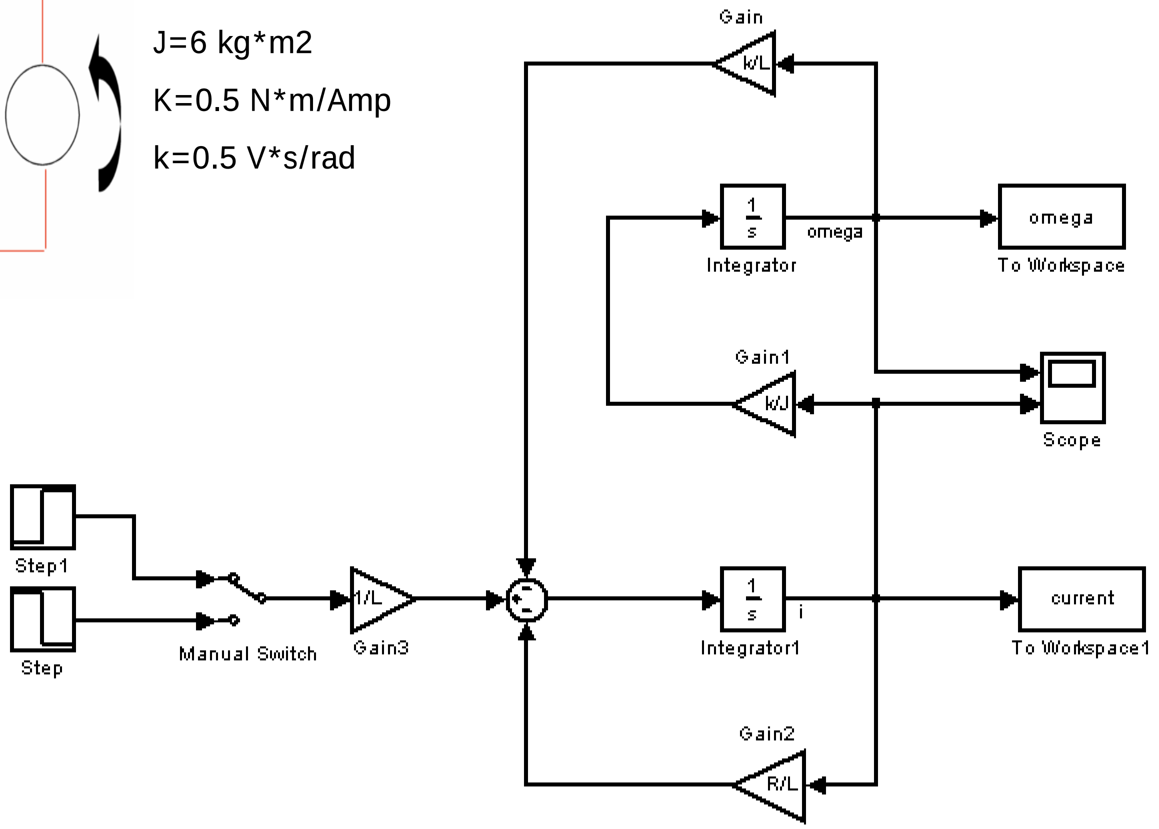


J=6 kg*m2

K=0.5 N*m/Amp

k=0.5 V*s/rad

casmotor.mdl



Solvers - stiffness

- There can be problems to find model solutions if the system shows *stiffness*
- Could be due to system containing time constants that are very different from each other
- In this case, use one of the Simulink stiff solvers, ODE23s, ODE15s

Zero-crossing

- Discontinuities are important events while simulating a dynamic system
- A fixed-step solver can overcome this problem by having a very small step size => long simulation times
- Even a variable time step can produce long simulation times.
- Solution - Simulink checks for discontinuities in the system's state variables at each time step using *zero-crossing detection*.
- Specific blocks incorporate zero-crossing detection. At the end of each simulation step, Simulink checks to see if any registered a discontinuity. If so, the time of crossing is estimated through interpolation between the two timesteps. Simulink then steps up to and over each zero crossing in turn. This avoids simulating exactly at the discontinuity, where the value of the state variable might be undefined.

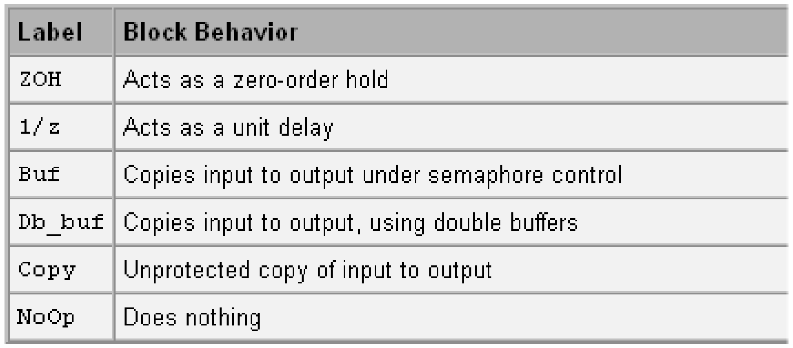
=> Zero-crossing detection enables Simulink to simulate discontinuities accurately without resorting to a large number of excessively small step sizes.

Rate Transition

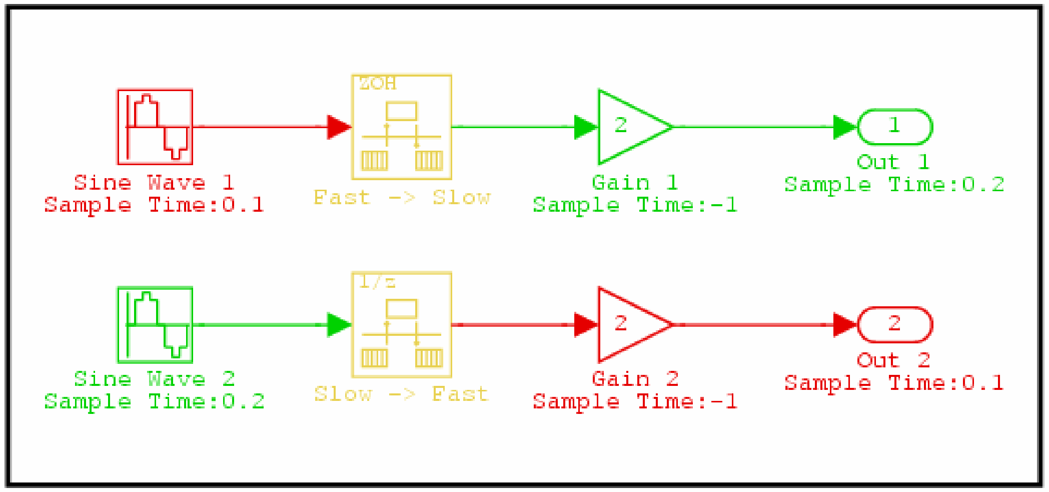
- The Rate Transition block transfers data from the output of a block operating at one rate to the input of another block operating at a different rate. The Rate Transition block's parameters allows you to specify options that trade data integrity and deterministic transfer for faster response and/or lower memory requirements.
- In particular, the block supports the following options:
  - Deterministic transfer of data with data integrity between blocks operating at different speeds at the cost of maximum latency of data transfer (DEFAULT)
  - Nondeterministic data transfer with minimum latency and assured data integrity but increased memory requirements. To specify this option, check the Ensure data integrity during data transfer parameter and uncheck the Ensure deterministic data transfer parameter.
  - Minimum latency and target size at the cost of nondeterministic data transfer and possible loss of data integrity. To specify this option, uncheck the Ensure data integrity during data transfer and Ensure deterministic data transfer parameters.

- The behaviour of the Rate Transition block depends on the sample times of the ports between which it is connected, the priorities of the tasks corresponding to the source and destination sample times (see Sample time properties), and whether the model specifies a fixed- or variable-step solver.
- Updating the diagram causes a label to appear on the block that indicates its behaviour during simulation

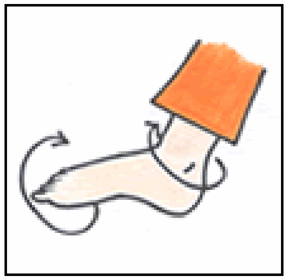
Rate Transition



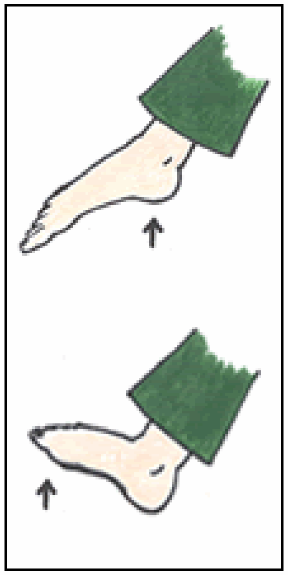
Label	Block Behavior
ZOH	Acts as a zero-order hold
1/z	Acts as a unit delay
Buf	Copies input to output under semaphore control
Db_buf	Copies input to output, using double buffers
Copy	Unprotected copy of input to output
NoOp	Does nothing



- THE END

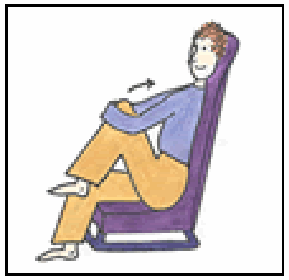


1. Foot circles. Lift your feet off the floor. Moving both feet together, draw an imaginary circle with each big toe so that each foot rotates about the ankle joint. Continue several times in one direction, then repeat in the other direction.

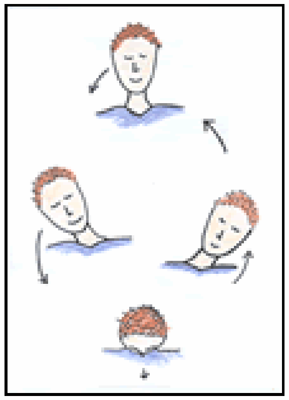


2. Heel lift/Toe lift. Start with your feet flat on the floor. Then lift your heels as high as is comfortable while leaving your toes and the balls of your feet on the floor. Lower your heels and repeat several times. Now leave your heels on the floor and gently lift your toes and the front of your feet off the floor, thus flexing your ankle as far as is comfortable. Lower and repeat several times.

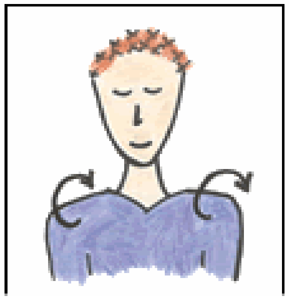
3. Knee raises. Sitting upright with your feet flat on the floor, lift one leg up while keeping your knee bent, hold for 2-3 seconds, then lower. Do the same with the other leg. Repeat the sequence at least 20 times for each leg.



4. Knee curls. Sitting upright, gently lean forward while at the same time raising one knee. Grasp the knee with both arms and gently pull the leg towards your chest as you then lean back. Hold for 15 seconds, then release and gently lower the leg. Do the same for the other leg. Repeat the sequence 10 times.



5. Neck stretches. Start with your head in an upright position. Gently drop your right ear towards your right shoulder as far as is comfortable. Then gently roll your head forwards until you are looking down at your lap. Finally roll your head gently up towards your left shoulder, then lift your head to the upright position. Alternate the direction and repeat several times.



6. Shoulder rolls. Sitting comfortably, roll your shoulders gently backwards, continuing in a circular motion, several times. Repeat the circles, rolling your shoulders forwards several times.