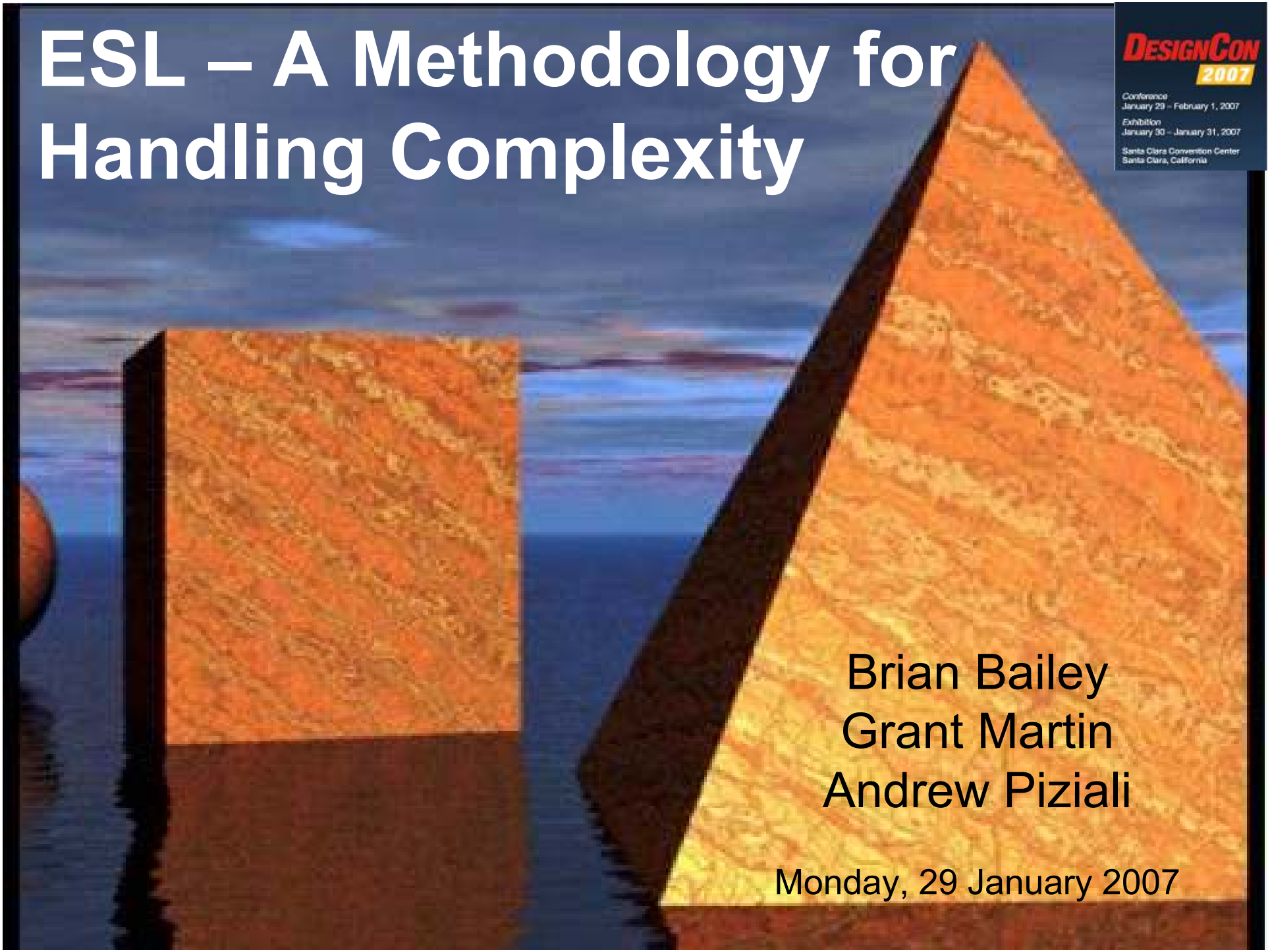
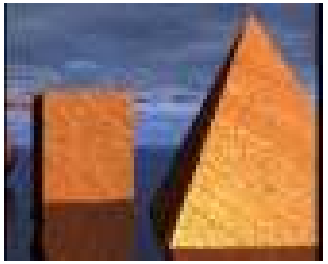
ESL – A Methodology for Handling Complexity



DesignCon 2007
Conference
January 29 – February 1, 2007
Exhibition
January 30 – January 31, 2007
Santa Clara Convention Center
Santa Clara, California

Brian Bailey
Grant Martin
Andrew Piziali

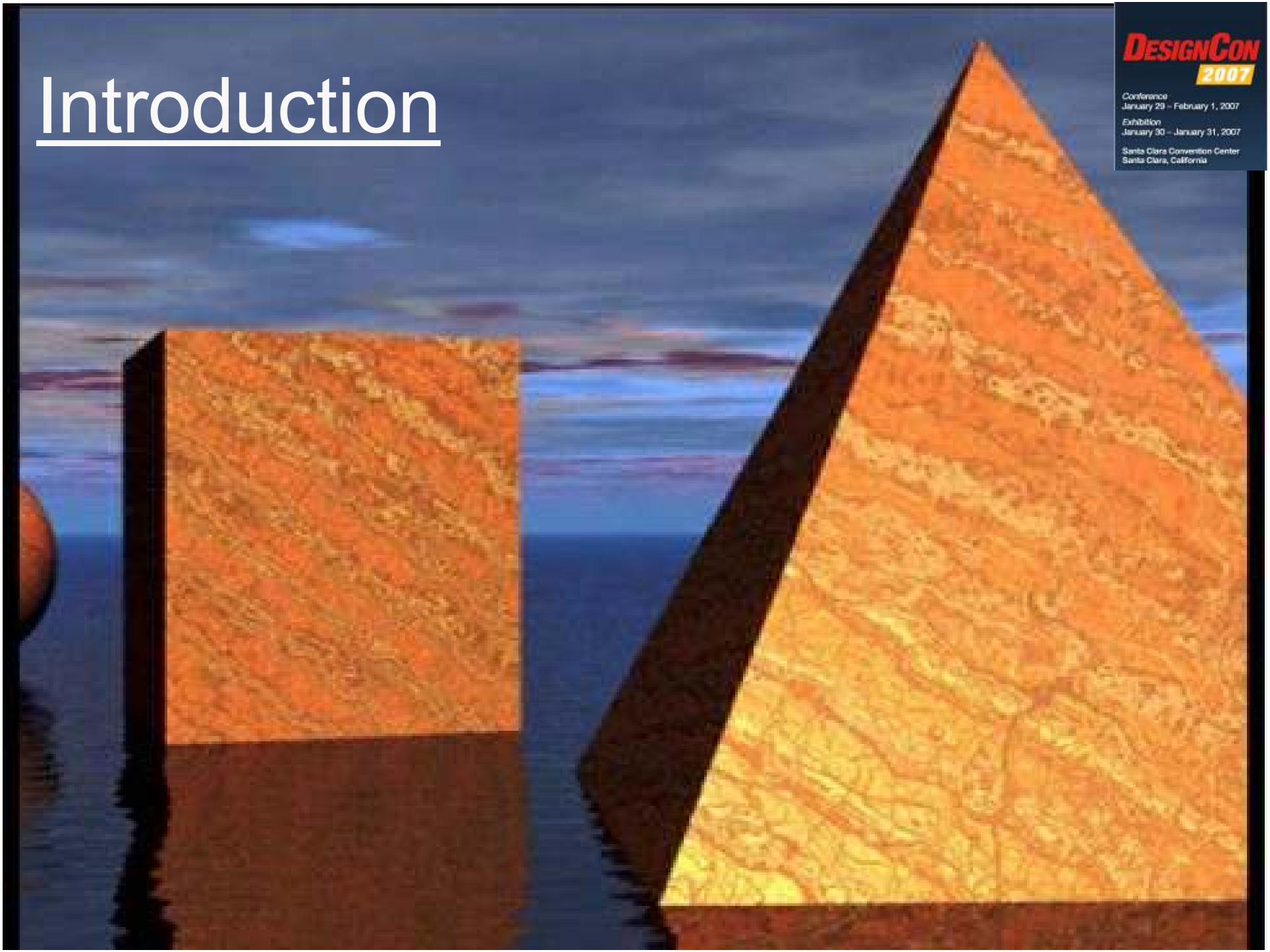
Monday, 29 January 2007

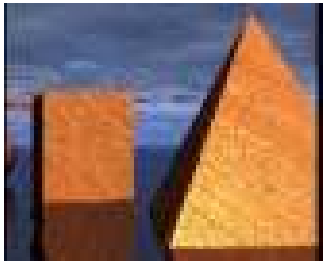


TecForum Outline

- Introduction – 30 min
- The ESL Flow – 10 min
- Specification and Analysis – 20 min
- Pre-Partitioning Analysis, and Partitioning – 25 min
- Break – 10 min
- Post-Partitioning Analysis – 15 min
- Verification – 30 min
- HW and SW Implementation – 25 min
- Summary, Futures and Conclusions – 15 min

Introduction



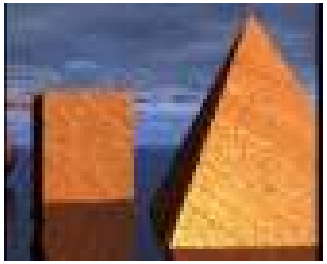


The Authors

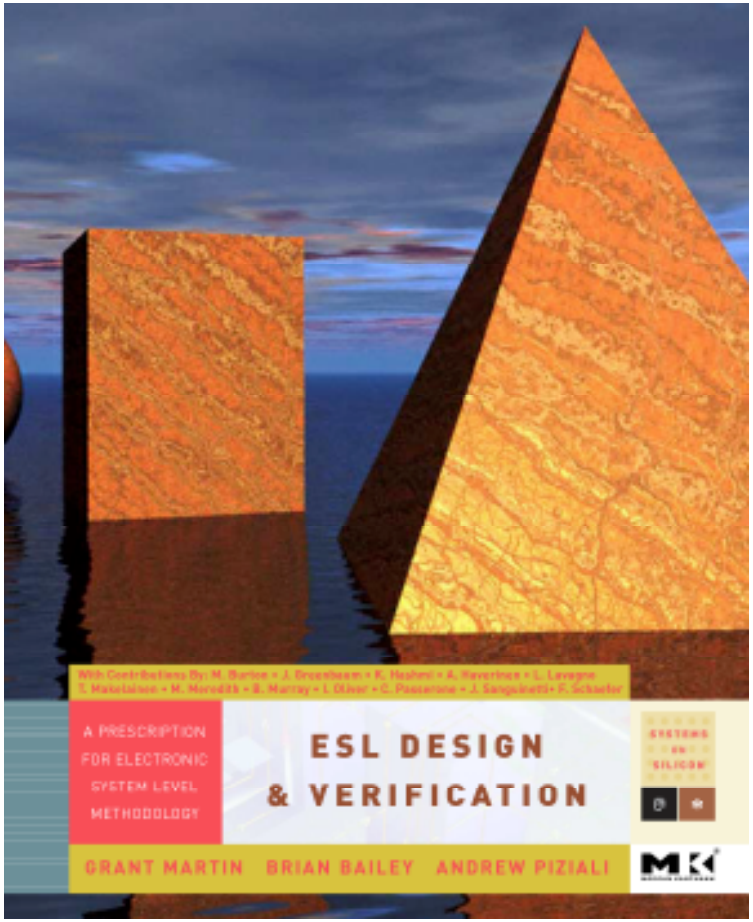
Brian Bailey

Grant Martin

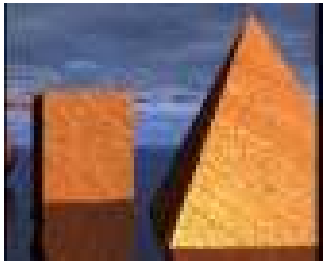
Andrew Piziali



The Book



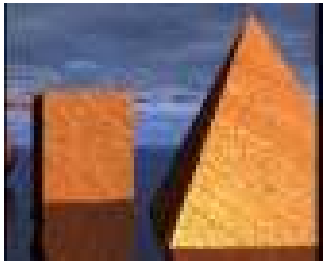
Due out March 2007
From Elsevier-Morgan Kaufmann



Why did we write it?

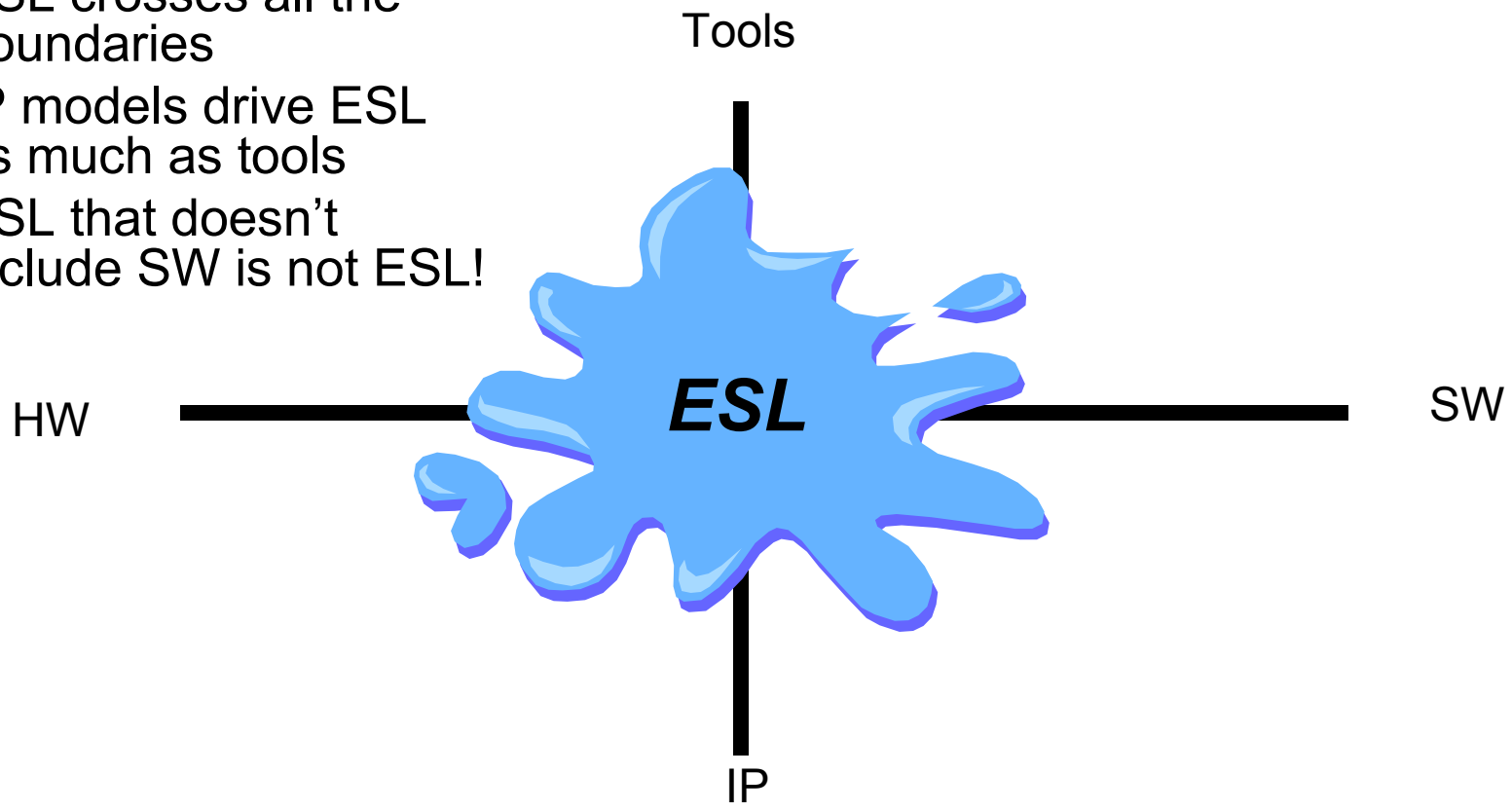
- "There is a tide in the affairs of men, Which taken at the flood, leads on to fortune. Omitted, all the voyage of their life is bound in shallows and in miseries. On such a full sea are we now afloat. And we must take the current when it serves, or lose our ventures."
  - William Shakespeare

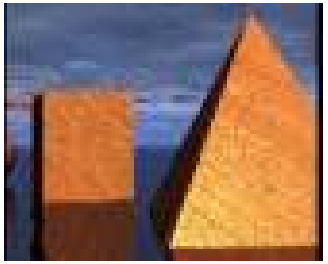
- ***The time is ripe***
  - We can see real ESL taking shape
  - We can see real usage of some of the current ESL tools occurring
  - Research concepts are now more ready to become practical steps in the design and verification flow



What makes ESL different?

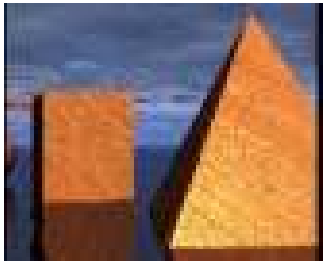
- ESL crosses all the boundaries
- IP models drive ESL as much as tools
- ESL that doesn't include SW is not ESL!





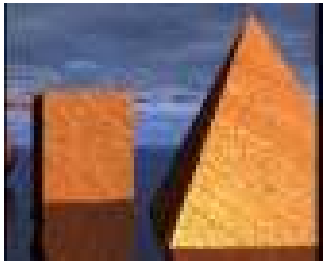
What makes ESL different?

- Abstraction is possible
  - Model speed-accuracy tradeoffs
    - Essential
    - Worthwhile
    - Possible
- Open Source
  - SystemC as an example (specific community source model)
  - Existence drove modeling and experiments
  - Standardization ensures value of tool and model investments
  - OSCI Reference has kept a lid on prices and revenues



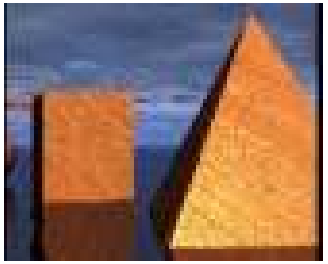
A brief look at history

- Those who do not remember the past are condemned to repeat it.
  - –George Santayana



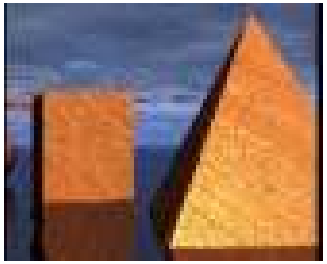
Motivations

- Consider a device:
- 3G cell phone/data terminal:
  - integrated Global Positioning System (GPS) device
  - digital camera
  - video/MP3
- Acts as:
  - entertainment center
  - web terminal
  - personal information management device
- With Wi-Fi or Bluetooth connectivity
- How to design, implement and verify?
- "…the increasing failure of traditional methodologies to cope with the burgeoning system algorithm content necessitated by the integration of so much functionality."

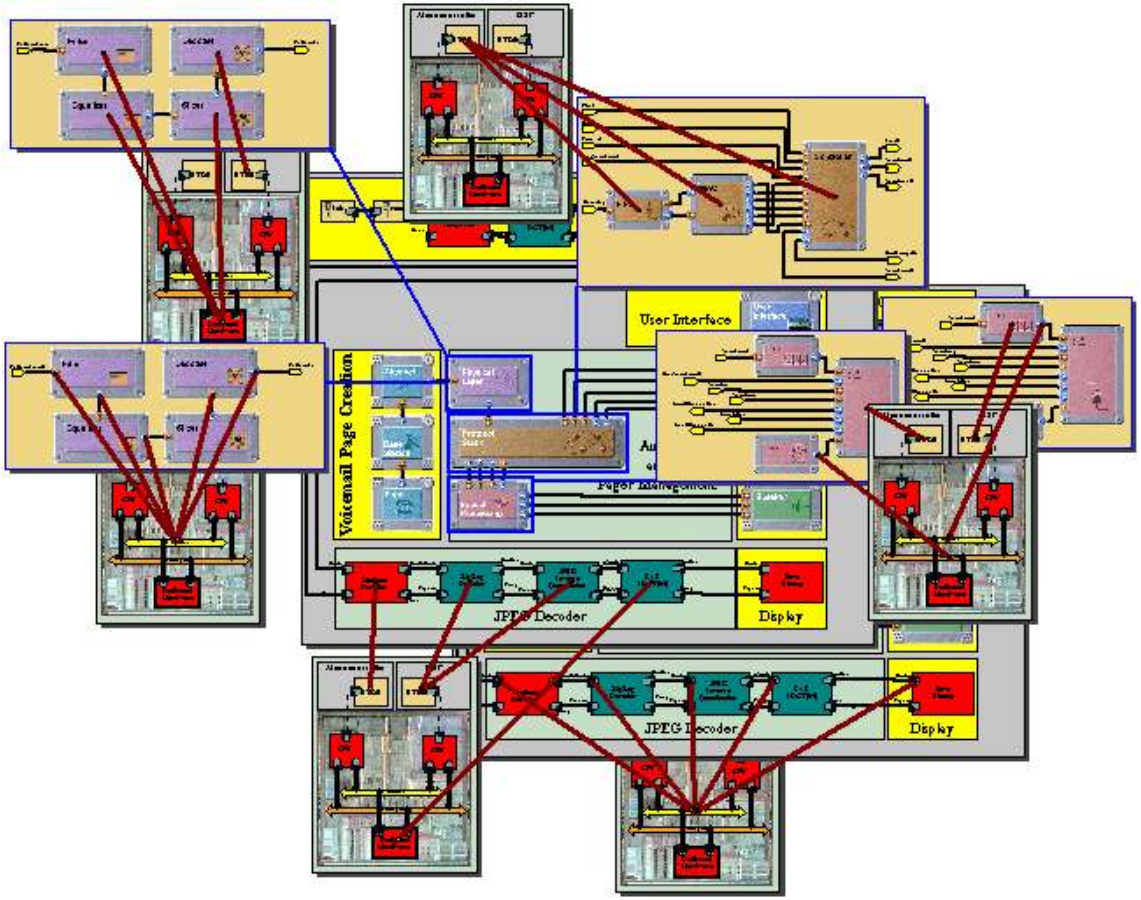


Historical Categories of ESL

- Behavioral Modeling
  - Leading to “Virtual System Prototypes” (VSP)
    - (after Graham Hellestrand, EST – Embedded Systems Technology)
- Automated Implementation of Fixed-Function Hardware
- Automated Implementation of Programmable Hardware



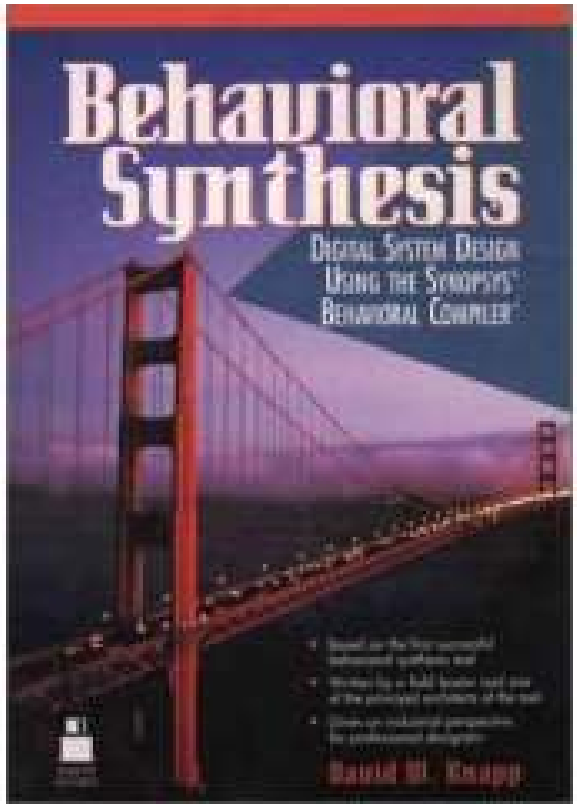
Historical Examples: Behavioral Modeling



Function-architecture Co-design:

The Late, Lamented 'Felix' (VCC) (1997+)

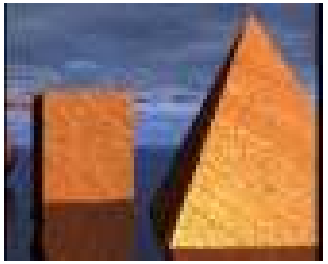
Historical Examples: Automated Implementation of Fixed-Function Hardware



Synopsys Behavioral Compiler

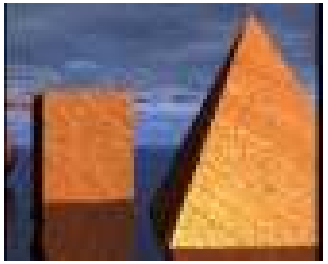
Book by David Knapp, Father of Behavioral Compiler

Prentice-Hall PTR
June ***1996!***



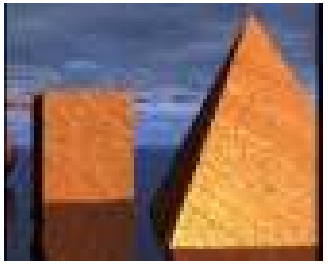
What can we learn from history?

- Standardized capture mechanisms (e.g. languages) are vital to promote model existence
  - SystemC
- Model interoperability is key
- IP-driven design at ESL level is driven by model availability
- Speed / accuracy tradeoffs are important
- The natural form for algorithm implementation is “C” (or variants)
- New implementation technologies fit design niches
  - Hyping them as universal solutions is counter-productive



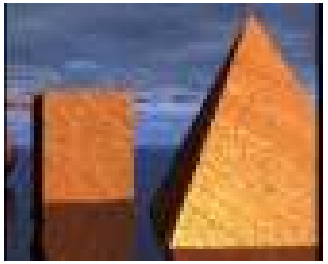
Examples: academic and open source

- Polis
- Ptolemy
- SpecC
- OSCI SystemC
- SPARK



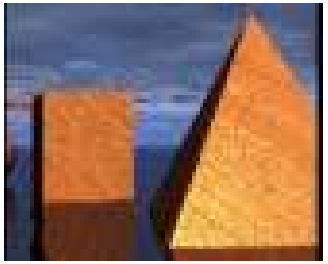
Current examples: industrial

- Behavioral modeling/VSP
  - CoWare Platform Architect
  - ARM Realview ESL
  - Synopsys System Studio/Virtio
  - VaST
  - Virtutech ….



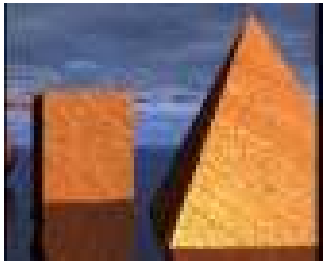
Current examples: industrial

- Automatic implementation of fixed-function HW:
  - Algorithm modeling: Matlab/Simulink, SPW, …
  - High-level synthesis: Forte, Mentor Catapult, Bluespec, …
- Automatic implementation of programmable HW:
  - Tensilica Xtensa/XPRES
  - Critical Blue
  - Synfora
  - Target Compiler Technologies
  - CoWare
  - ARC
  - ARM OptimoDE
  - Improv Systems



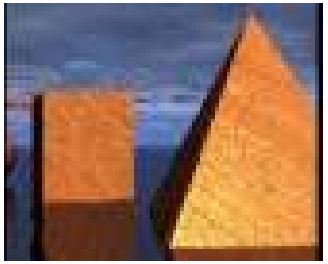
Example of possible value: VSP

- System architectural design, analysis, optimization, and verification
  - Estimate system performance before implementation
  - Analytical HW/SW and SW/SW partitioning over multiple processors
- Start application software development well in advance of hardware
- Early identification of system non-determinism
- Execute HW and HW/SW co-verification orders of magnitude faster than RTL/C
- Significantly reduce overall development time, effort and risk
  - “green field” or “blank sheet” designs
  - platform-based derivative designs

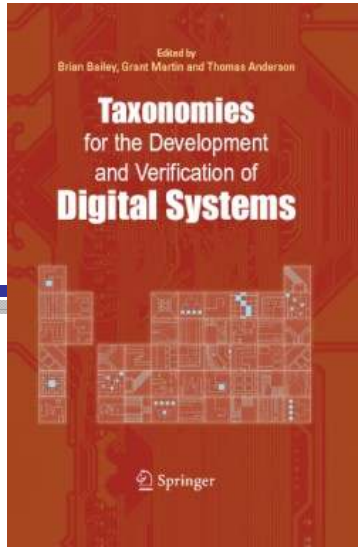


Entering the mainstream

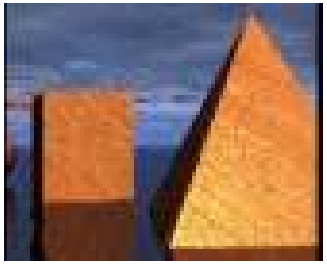
- Who bears the Risk?
  - System Architects
  - RTL Teams
  - SW Teams
  - ASIP design
- Impact of ESL on Commercial EDA
  - The “Big 3”



Taxonomy

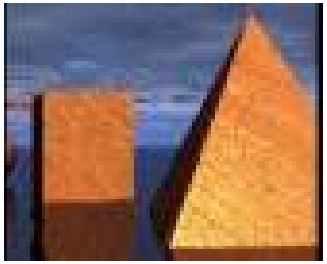


- Enables the definition of models and terms in a more precise manner
- Based on a long line of work (RASSP, VSIA)
  - B. Bailey, G. Martin, and T. Anderson, eds, *Taxonomies for the Development and Verification of Digital Systems*, Springer Science+Business Media, New York, 2005.
- Maintains most of the notion of abstraction from VSIA work
  - Temporal, Data abstraction

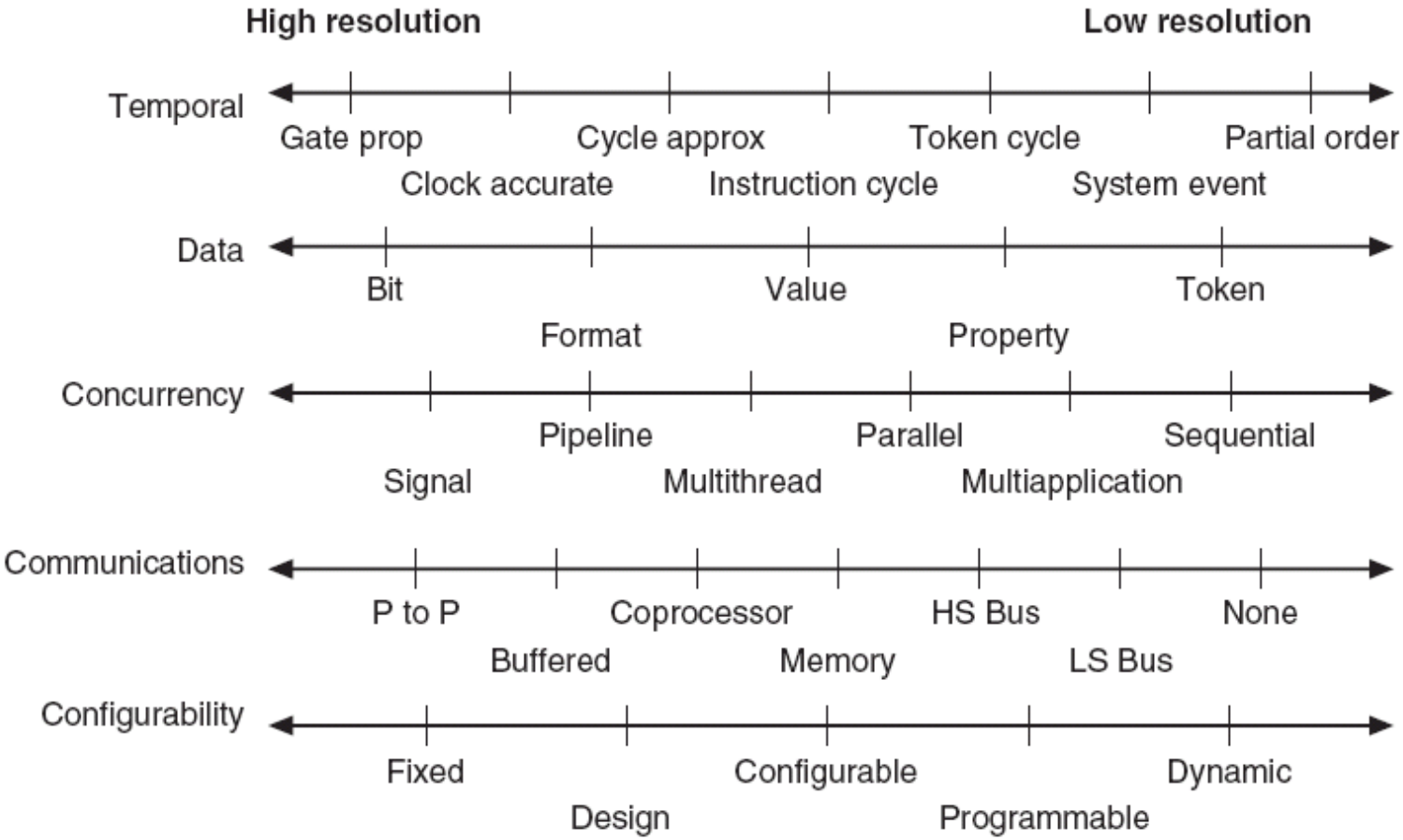


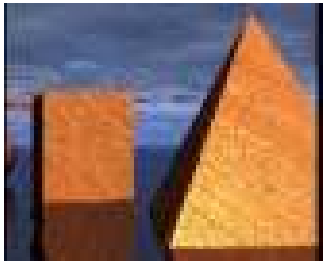
Taxonomy – for ESL

- Adds three new axes to define attributes of the system:
  - Computation
  - Communications
  - Configurability
- All attributes and abstractions are orthogonal
  - With some linking through practicalities

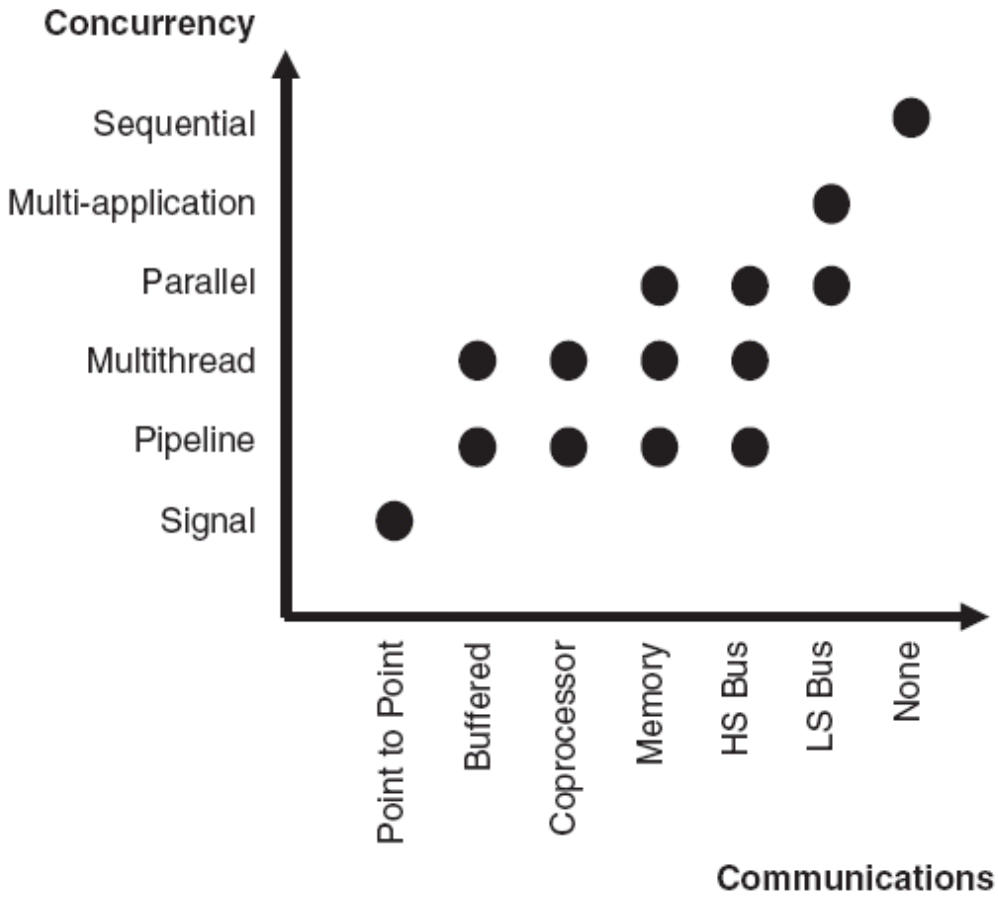


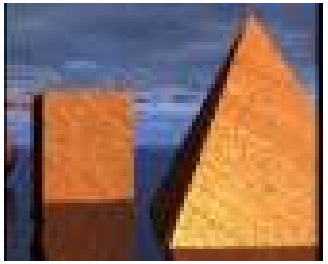
Taxonomy Axes



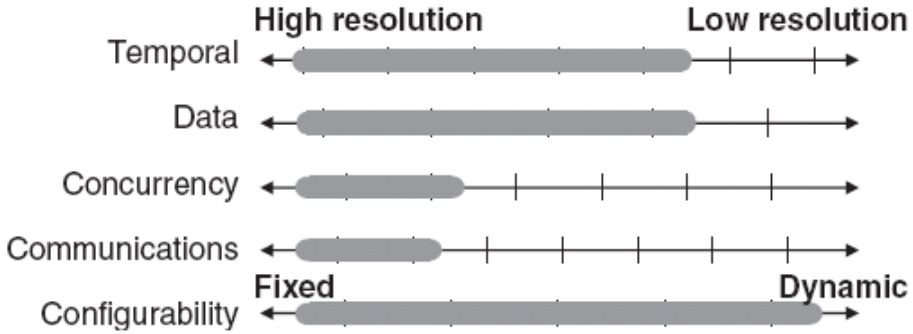


Communications vs Computation



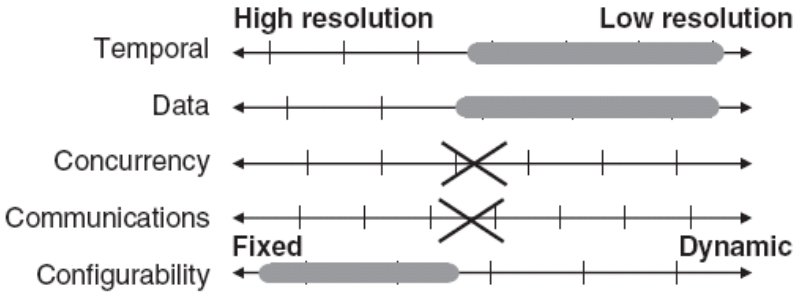


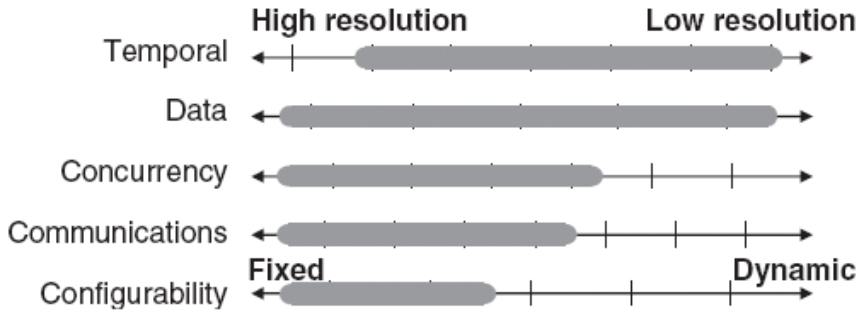
Taxonomy Examples



An HDL

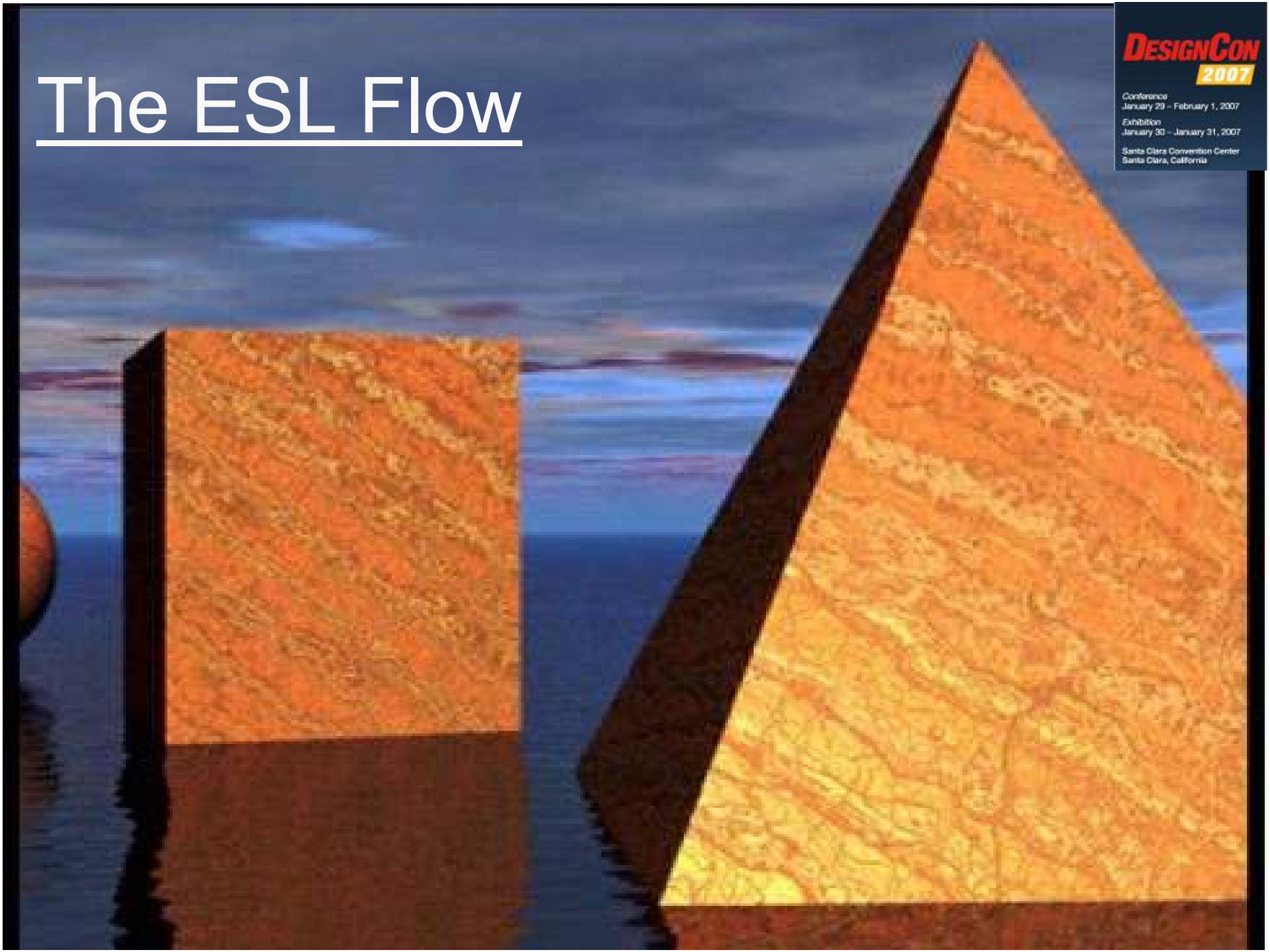
Generic C

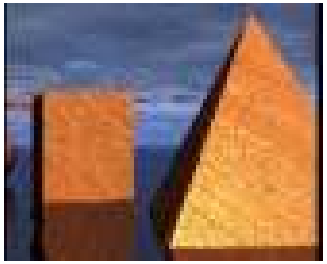




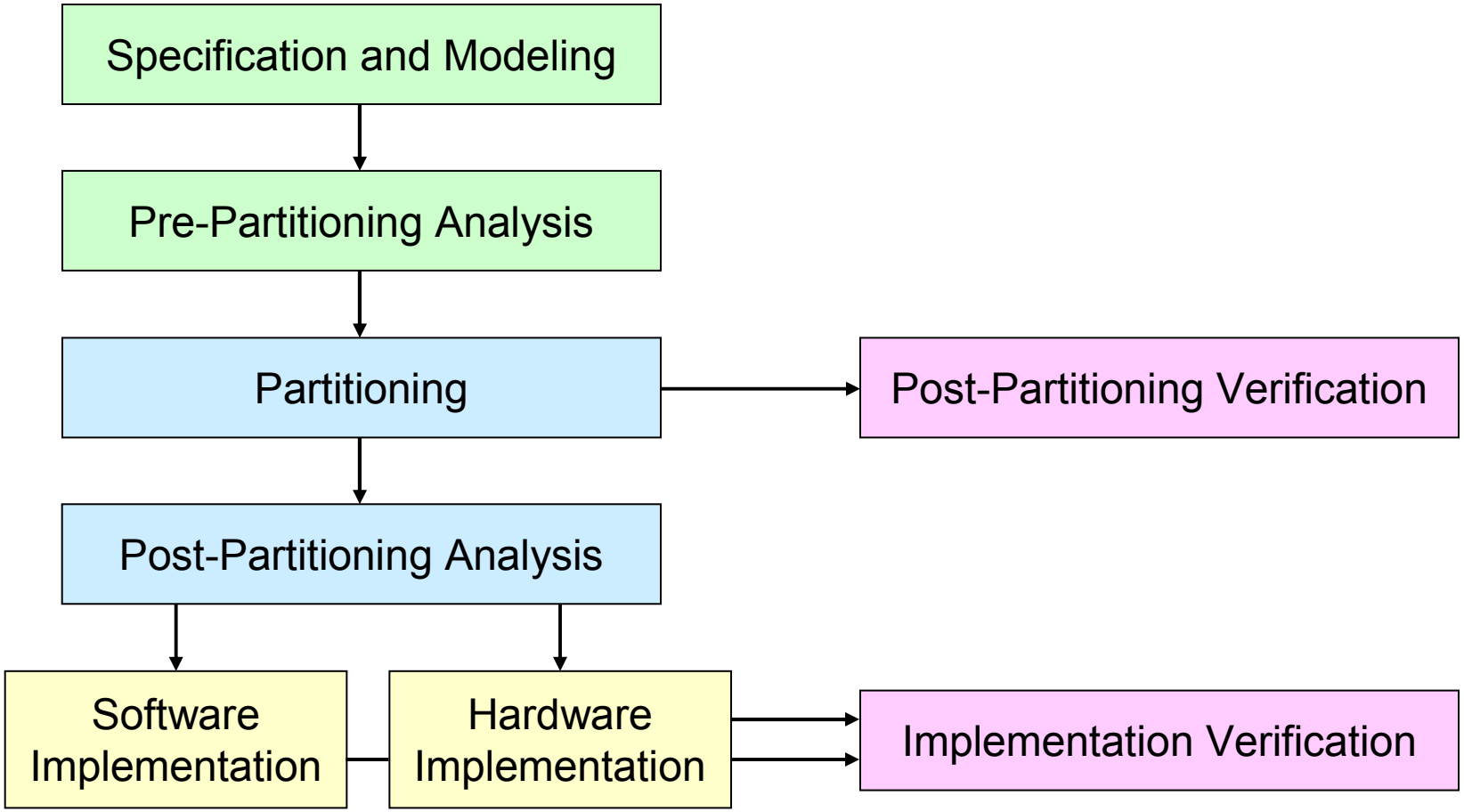
SystemC

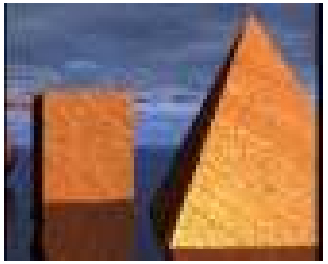
The ESL Flow


DESIGNCON
2007
Conference
January 29 – February 1, 2007
Exhibition
January 30 – January 31, 2007
Santa Clara Convention Center
Santa Clara, California



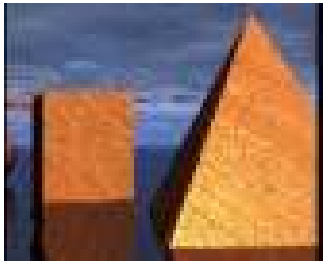
The Flow





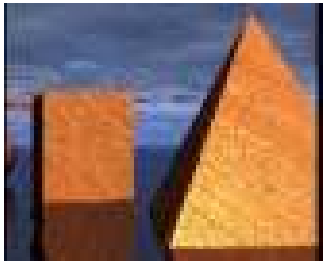
Specification and Modeling

- Use natural language specifications and executable specifications
- Manage complexity
- Track requirements with a tool
- Choose a specification language
- Consider model-based development



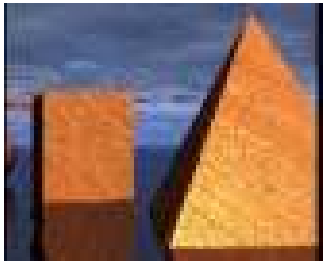
Pre-Partitioning Analysis

- Explore spectrum of algorithmic tradeoffs
- Time, space, power, complexity, TTM
- Dynamic analysis using executable specs
- Static analysis
  - Reliability, maintainability, usability and criticality
- Consider platform-based design



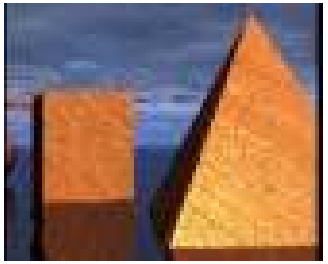
Partitioning

- Functional decomposition
- Architecture description (structural decomposition)
- Mapping (functional to architecture)
- Hardware partition
- Software partition
- Reconfigurable computing
- Communication implementation



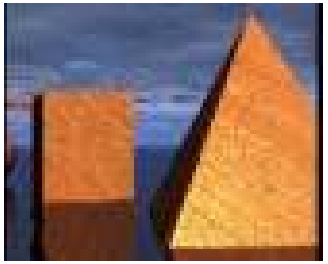
Post-Partitioning Verification

- Has the intended behavior been preserved?
- Verification planning
  1. Quantify scope of the verification problem
  2. Specify solution to the verification problem
- Implement verification environment
- Bring-up and regressions
- Analyze failures and coverage
- Employ abstract coverage



Post-Partitioning Analysis

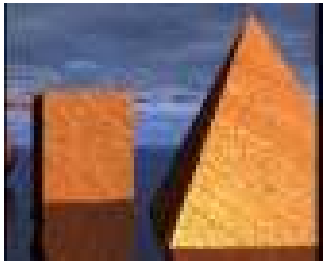
- Refine architectural models to reflect partitioning choices
- Choose appropriate HW and SW models
- Explore the design space
- Employ dynamic and static analysis
  - Functional, performance, interface, power, area, cost and debug capability analyses



Hardware Implementation

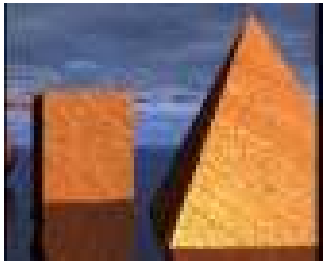
- Create HW model to be synthesized
- Choose a hardware implementation
  - extensible processors, DSP coprocessors, customized VLIW coprocessors, ...
- ESL synthesis piggybacks on RTL flow:

1. System specification
2. HW/SW partitioning
3. Virtual prototype
4. Transaction-level design
5. Transaction level
6. Verification
7. ESL synthesis to RTL
8. Verify RTL
9. Synthesize RTL to gates
10. Verify timing
11. Place and route gates
12. Design rule check
13. Generate GDSII



Software Implementation

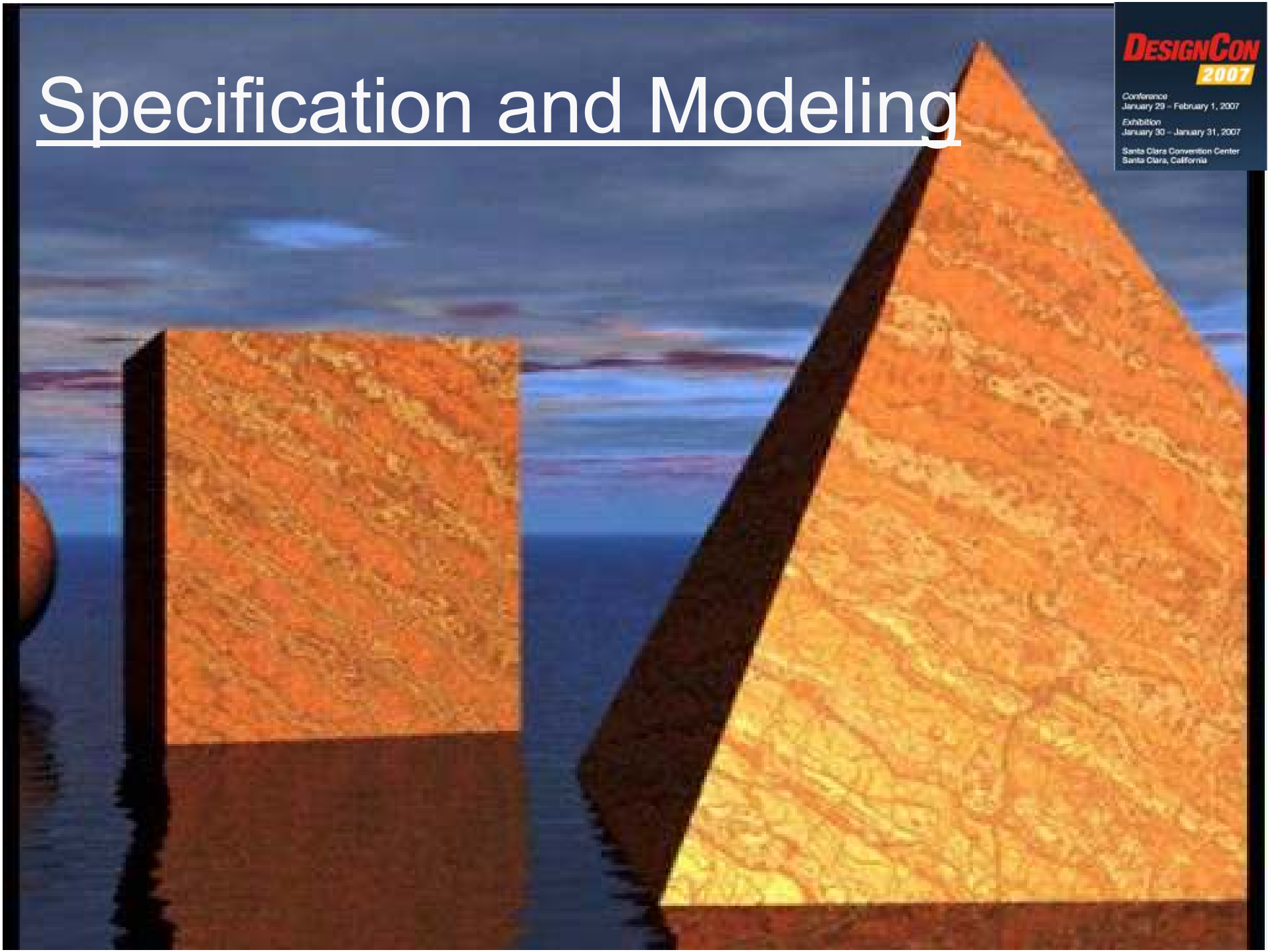
- Use ESL models to prototype software components
- Estimate algorithm performance
- Choose ESL specification language
- Consider debugging environment
- Use ESL model for runtime development

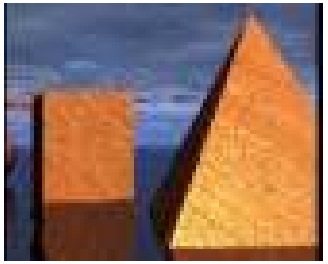


Implementation Verification

- *Clear box* vs. *opaque box* verification
- Compare implementation against post-partitioned models
- Employ positive and negative verification
- Use formal analysis (PSL, SVA)
- Use verification IP
- Measure and analyze coverage (again!)
- Accelerate execution when necessary

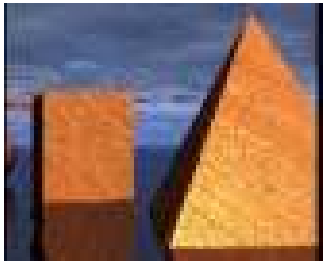
Specification and Modeling





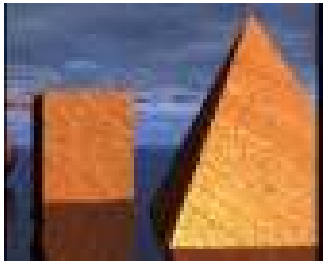
Specification

- A specification defines the functional and non-functional aspects of a system that is devoid of implementation decisions
  - For ESL it is important that decisions about HW, SW, Architecture etc., are not embedded in the specification
- Architectural decisions are made to refine a specification towards implementation
  - Architectural, micro-architecture, fabrication technologies…



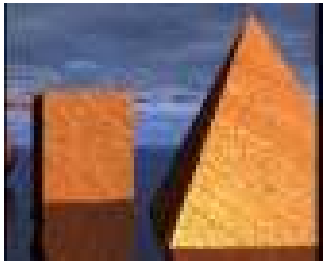
Natural language or executable?

- Natural languages are more expressive
  - Easier for humans to navigate
  - But subject to ambiguity
  - And more difficult for computers to navigate
- Requirements are easier to automate
  - And easier to be implementation independent
  - Most likely to lead into verification flow
    - Verification Planning
    - Coverage, Properties



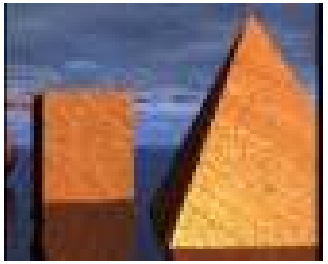
Multiple Aspects

- Need to specify multiple aspects
  - Functionality
    - Includes HW, SW, mechanical…
  - Architecture
    - Solution structure
    - Constraints
      - Power, performance, cost…
  - Mapping
    - Used to be called HW/SW co-design
    - Today the scope is broader



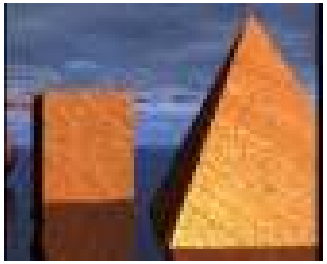
Modeling

- A model is a description or analogy used to help understand something that cannot be directly observed
  - a model employs abstraction that can hide unnecessary details and thus highlight the important aspects of the model, making them more comprehensible
  - In general we can only deal with a finite set of issues at a time, so we use abstraction to reduce the number visible
- Implies that you need different models to analyze different aspects of a system



Requirements

- Requirements emerge from the problem domain
- Requirements management is a process that:
  - Takes care of making all requirements visible and traceable
- A requirement management system depends on the size and complexity of the organisation
  - Placing trust only in paper documents will not suffice
  - Some degree of automation is required

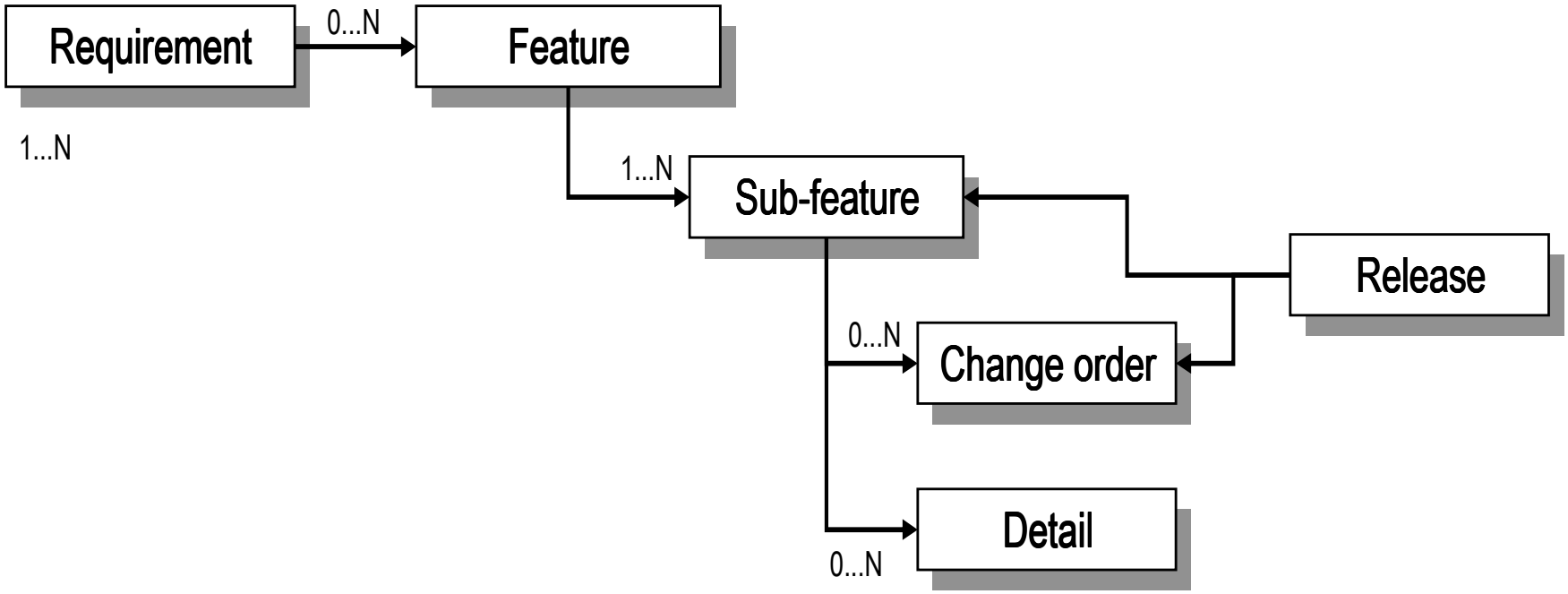


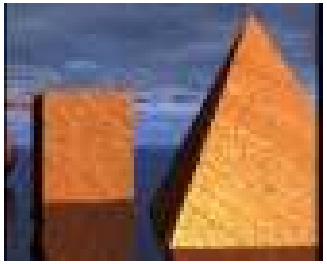
Requirements Management

Customer's understanding of the need

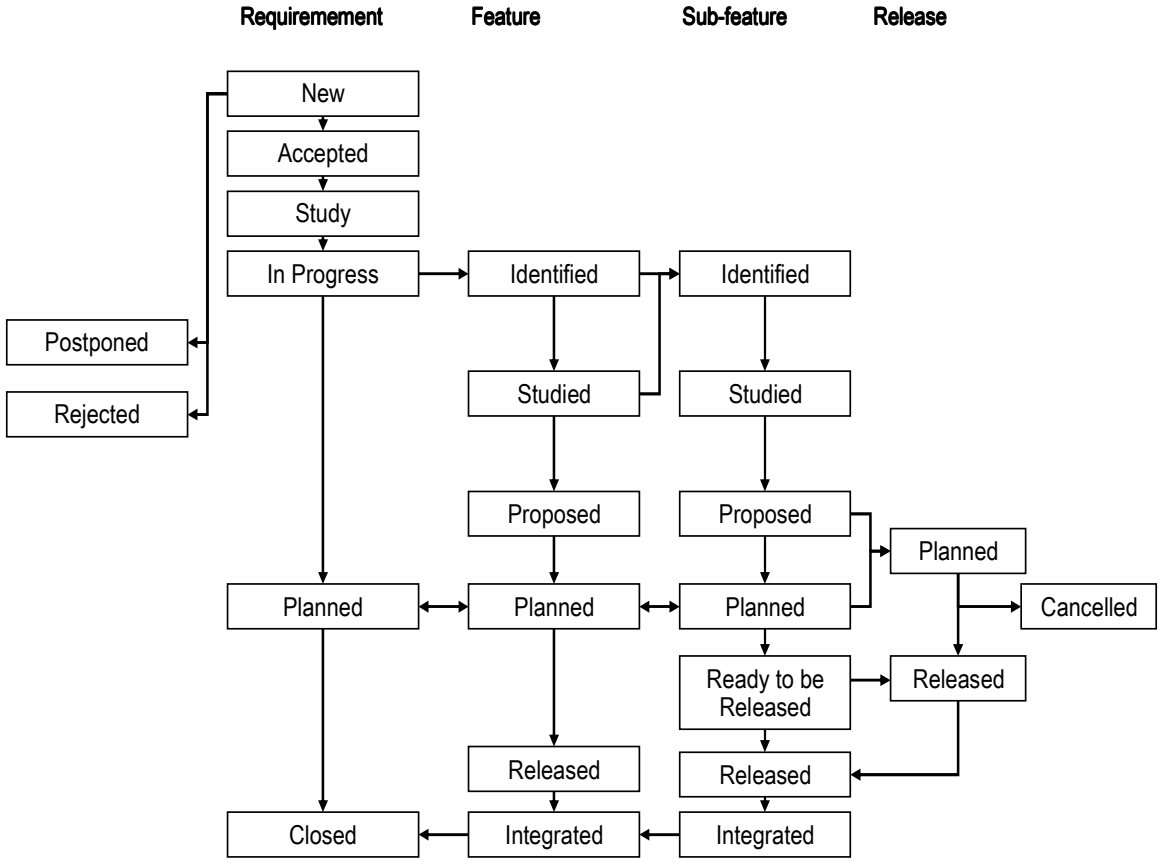
Features and Subfeatures represent the product/component management and implementation view of the supplier

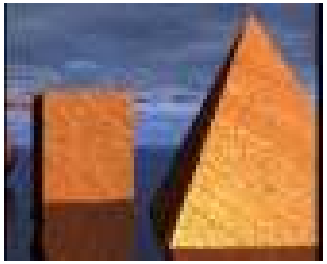
Releasing provides planning view to implementation





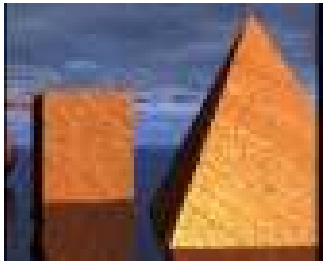
Requirements Documents





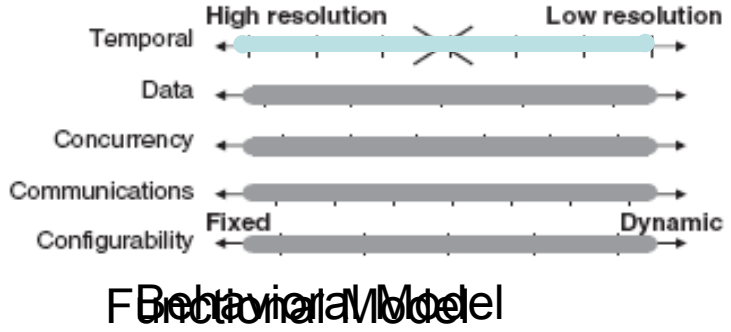
Solutions

- Commercial tools
  - DOORS,
  - Caliber-RM,
  - PACE,
  - RMTrack,
  - Team-Trace
- References
  - Woodruff, Wayne, “Requirements Management for Small Organizations”, A Field Guide to Effective Requirements Management Under SEI’s Capability Maturity Model, Rational Software Corporation, 1997
  - Sud, Rajat R. and Arthur, James, “Requirements Management Tools: A Qualitative Assessment”, Department of Computer Science, Virginia Tech, Blacksburg, VA 24060 USA, 2003



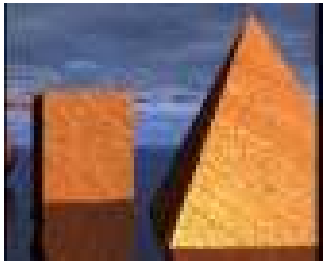
Specification Languages

- Multiple Domains
  - Dataflow / Control flow
  - Protocol Stacks
  - Embedded Systems
- Multiple Viewpoints
  - Algorithmic
  - Functional
  - Behavioral
    - Transaction level



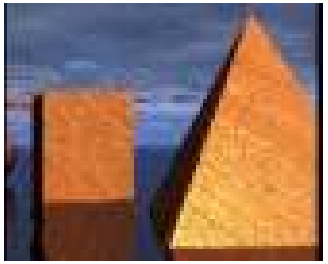
Note that neither defines abstraction

No one language today can properly address them all



Specification Languages

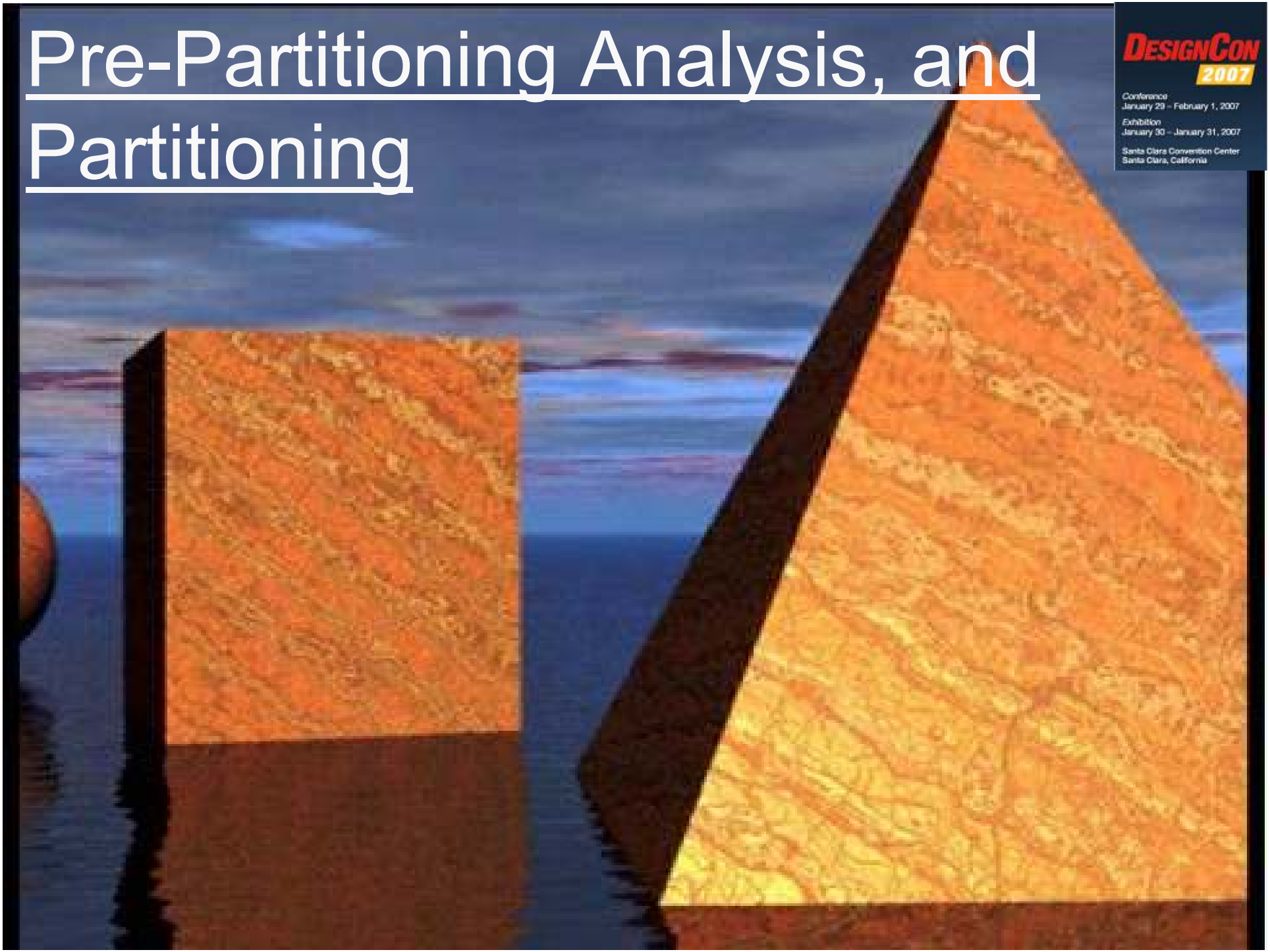
- Leading Candidates
  - MATLAB
  - Rosetta
  - SystemC
  - SDL
  - UML
  - Bluespec
- Ideal solution would to add aspect-oriented constructs to a specification language
  - An EDA example is the *e* language

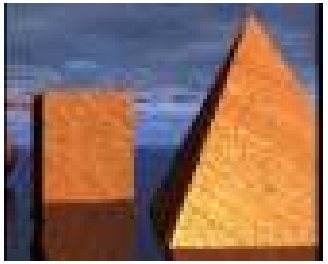


The Prescription

- Specification should be captured as formally *as possible*
  - Executable if it adds value and can be independent of implementation
  - Requirements should be formalized and tracked
  - Use natural language docs to fill in the blanks
- Concentrate on new functionality
- Specification should be refined over time

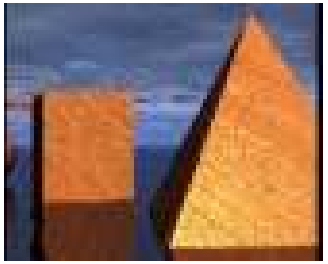
Pre-Partitioning Analysis, and Partitioning





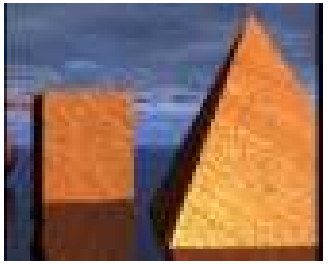
Pre-Partitioning Analysis

- Static analysis of specifications
- Impact of platform-based design
- Dynamic analysis
- Algorithmic analysis
- Analysis scenarios
- Downstream use of results
- Case study



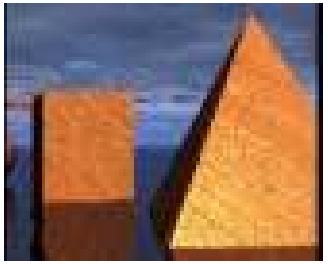
Static Analysis

- Software project estimation
  - AJ Albrecht, function point analysis (1979)
  - Tom DeMarco, function ! Metrics (1982)
  - International Function Point User Group (1986 to today)
- Analysis of HW and Systems
  - William Fornaciari and colleagues, CEFRIEL (Milano)
  - Designs in VHDL, Occam2, C, UML
  - Predicting:
    - Power estimation
    - Software execution time
    - Development cost, size, including reuse; product cost
  - Performance usually a constraint
  - Very difficult to separate pre- from post-partitioning



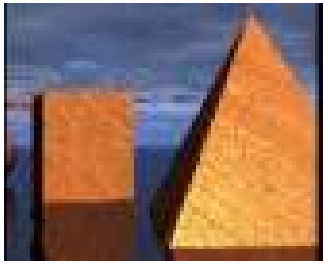
Static Analysis

- "ility" analysis
  - Reliability, maintainability, usability, criticality…
  - Mil/aerospace (MIL-STD-217)
  - Hierarchical combination of predictors for subsystems
  - Depends on accurate subsystem and component models
  - Difficult to gather usable historical data in many embedded systems and teams



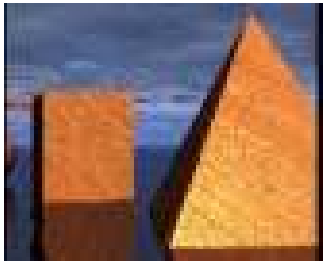
Requirements Analysis

- To support traceability
- To help define verification and validation tests
- Can be used to help define "implementation weight" of a specification



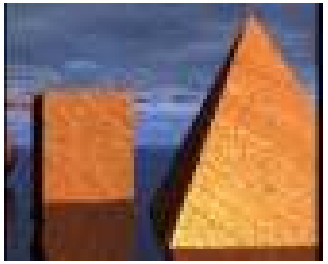
Impact of platform-based design

- If creating a derivative of a defined platform
  - Apply analysis methods to new or revised portions of platform
  - Try to avoid being biased by partitioning decisions already embedded in the platform
    - New functions tending to software do not have to just run on existing processors
    - Analyze before partitioning – establish requirements/needs before deciding on HW vs. SW



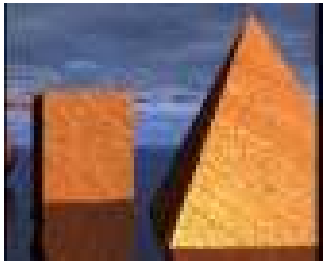
Dynamic Analysis

- Based on executable models
- Based on simulation
- Can estimate
  - Computational “burden”
  - Communications “burden”
  - Power/energy “burden”
- Avoid bias
  - Models are usually partitioned – need NOT imply final partitioning
  - Executable models contain implementation “artifacts”
    - Carefully separate out characteristics that are ESL level from those that are artifacts



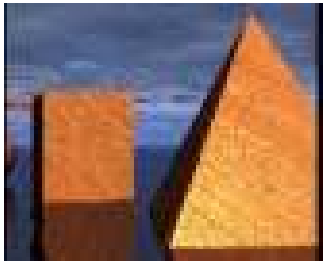
Algorithmic Analysis

- One of the oldest and most widely used areas of practical ESL
- Several long-standing commercial tools
  - SPW: Comdisco, Cadence (Alta), CoWare
  - The Mathworks Matlab/Simulink
  - Synopsys SystemStudio (many incarnations; best known was COSSAP)
  - State Machine tools from Mathworks (StateFlow), UML providers (IBM/Rational, Telelogic iLogix Rhapsody, Artisan SW tools, Esterel Technologies, …

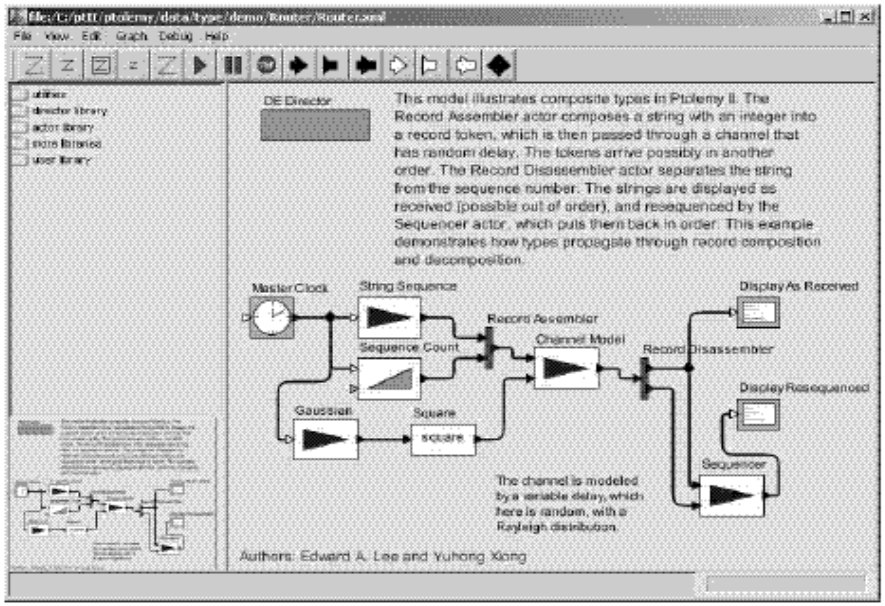


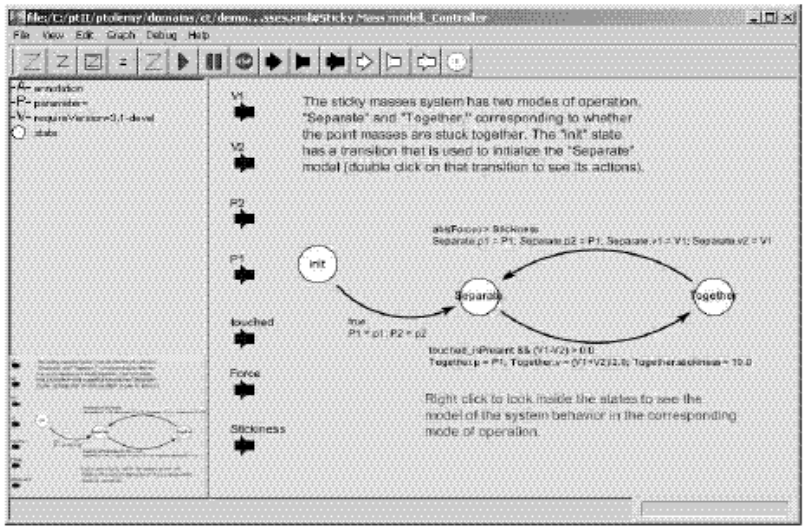
Research Tools

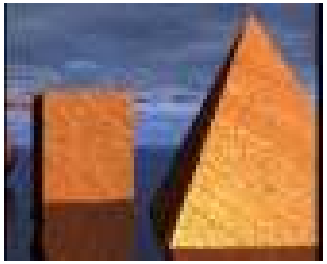
- Ptolemy – UC Berkeley
- POLIS/Metropolis – UC Berkeley
- SpecC – UC Irvine
- ….



Ptolemy Example

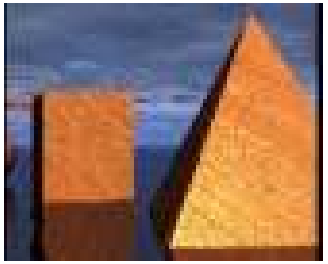






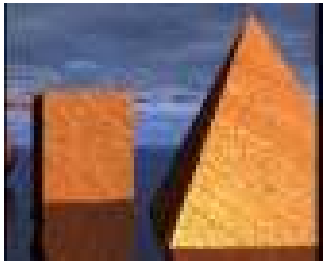
Analysis Scenarios

- Signal processing algorithms
  - Wireless and wired communications
  - Bit error rate (BER), Frame error rate (FER)
  - In the context of defined, parameterised channel models
  - Algorithms defined and inherent BER, FER determined when simulated with channel model for a particular communications protocol
  - Gradual refinement into partitioned, post-partitioned implementation possible
    - Floating point to fixed point mapping
    - DSP or custom HW targets
    - Eg. Iridium SPW example from mid-1990s
  - Filter design
  - Software-designed radio
- Most successful uses of these toolsets tends to focus on communications
  - Dataflow paradigm
  - Possible to take into implementations through a flow
  - Demonstrated success



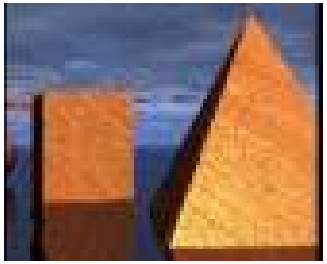
Downstream use of results

- Classic uses illustrated by SPW, Cossap, Mathworks Simulink/Matlab:
  - Floating-point, fixed-point models, results used as golden verification environments for HW/SW implementations
  - Algorithmic specifications drive software code generation for target processors and DSPs
  - Algorithmic specifications that can drive hardware code generation for RTL level synthesis
  - Co-simulation between system-level simulation, RTL simulation, and software execution of code on instruction set simulators
- Emerging: input into Behavioural/High-level/ESL synthesis

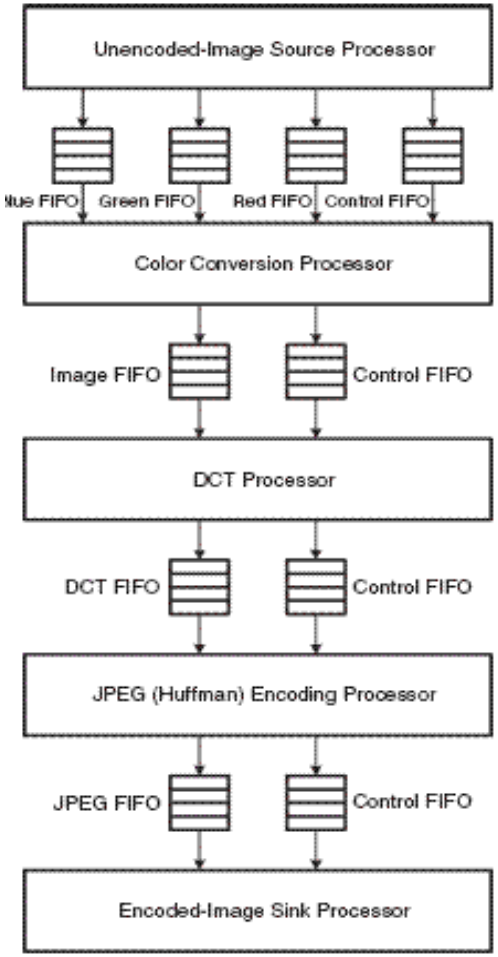


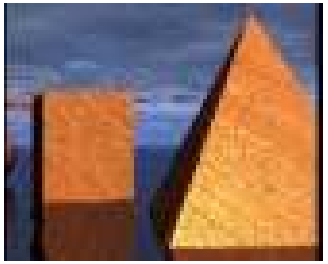
Case study: JPEG encoding

- Basic algorithm partitioned into sub-sections
- Mapped to configurable processors (5)
- Used to estimate computational and communications burden of algorithm as expressed in code
- Important not to be biased by particular case study partitioning in drawing conclusions



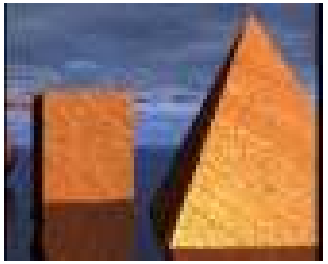
JPEG encoding





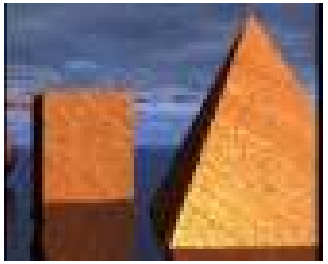
Computational Burden

Picture Size	Sum of system cycles	Color Conversion	DCT	JPEG (Huffman) encoding
32 x 32	636 K			
64 x 64	2.03 M	386 K	342 K	315 K
128 x 128	21.4 M			
256 x 256	85.5 M			
1024 x 1024		122.5 M	98.6 M	67.4 M



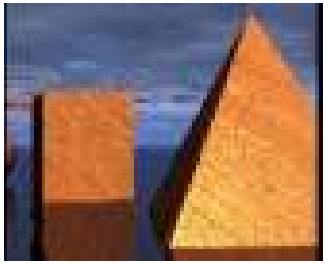
Case Study Design Space Exploration

- To process one 1024 x 1024 size picture in 1 second, assuming instructions per cycle ~ 1, run single processor at 300 MHz
- To do it in ½ second – HW or multiprocessor
- 2 processor solution – first at 250 MHz, second at 350 MHz (DCT+Huffman on 2nd)
- 3 processor solution – 400, 300, 200 MHz – does it in ~ 1/3 second



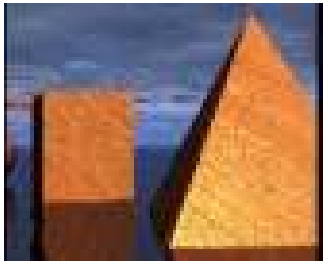
The Prescription

- Use specifications for analysis wherever possible
- Avoid “paralysis by analysis”
- Avoid “death by simulation”
- Simulate executable specs but separate out implementation artifacts
- Rich set of algorithmic analysis tools available
- Keep an eye on new methods and tools



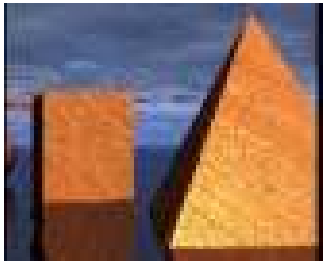
Partitioning

- "the process of subdividing an initial specification for a system, which is generally defined by a monolithic natural language document, executable specification, or legacy design, or a combination of all three, into a set of potentially concurrent cooperating processes, each of which may be described by documents, executable models, or legacy designs, or a combination of these forms, and of assigning them to a set of more or less abstract resources, representing processors for software, silicon area or IP blocks for hardware, communication channels, and storage resources (e.g., buses, memories)"
  - Take a specification
  - Chop it into pieces
  - Don't worry if the pieces need to operate concurrently
  - Assign the pieces to architectural resources
    - Processors
    - HW blocks
    - Communication channels
    - Storage



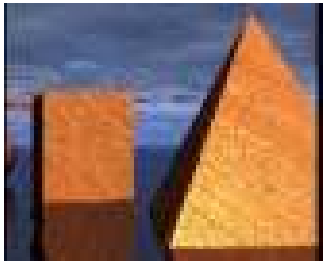
Partitioning

- Functional Decomposition
- Defining Target Architecture
- Mapping Function to Architecture
- Implementation
  - SW
  - HW
  - Reconfigurable
- Specify, implement and optimize the interfaces



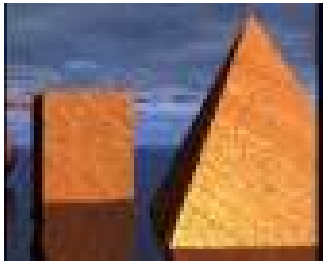
Functional Decomposition

- Use a (set of) functional concurrent executable specification language(s)
- Start from a sequential language and automatically extract concurrency



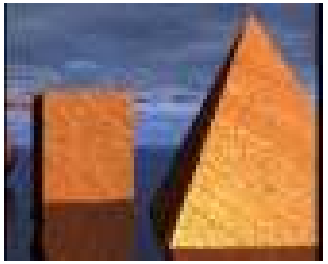
Many functional specification languages

- As discussed earlier:
  - Commercial tools and research languages
  - Simulink, MATLAB, Lustre, Esterel, UML, SDL, SPW, Ptolemy, SystemC…
- Divide the specification(s) into islands of "models of computation"
  - FSMs, discrete event, dataflow
- Link them together with specification models for each island

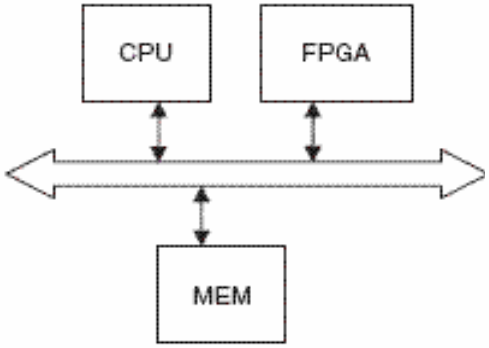


Extract Concurrency

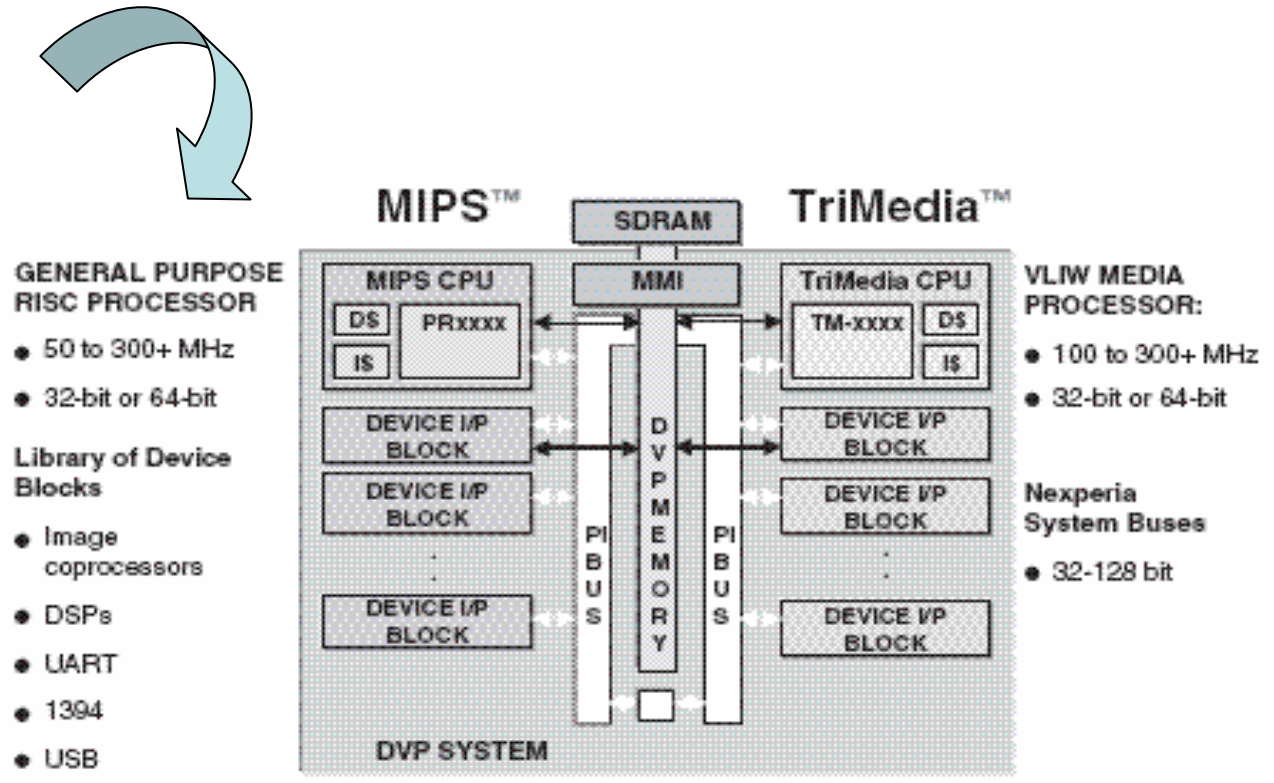
- Start with a single specification language, usually sequential:
  - E.g. C/C++/SystemC
- Research and commercial approaches to mapping this into partitioned systems
  - E.g. Synfora PicoExpress
  - Tensilica XPRES
  - Research using compiler technology at several universities
    - Convert loop nests into potentially concurrent HW



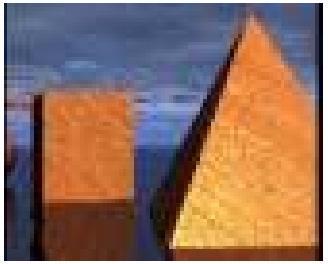
Defining Target Architecture



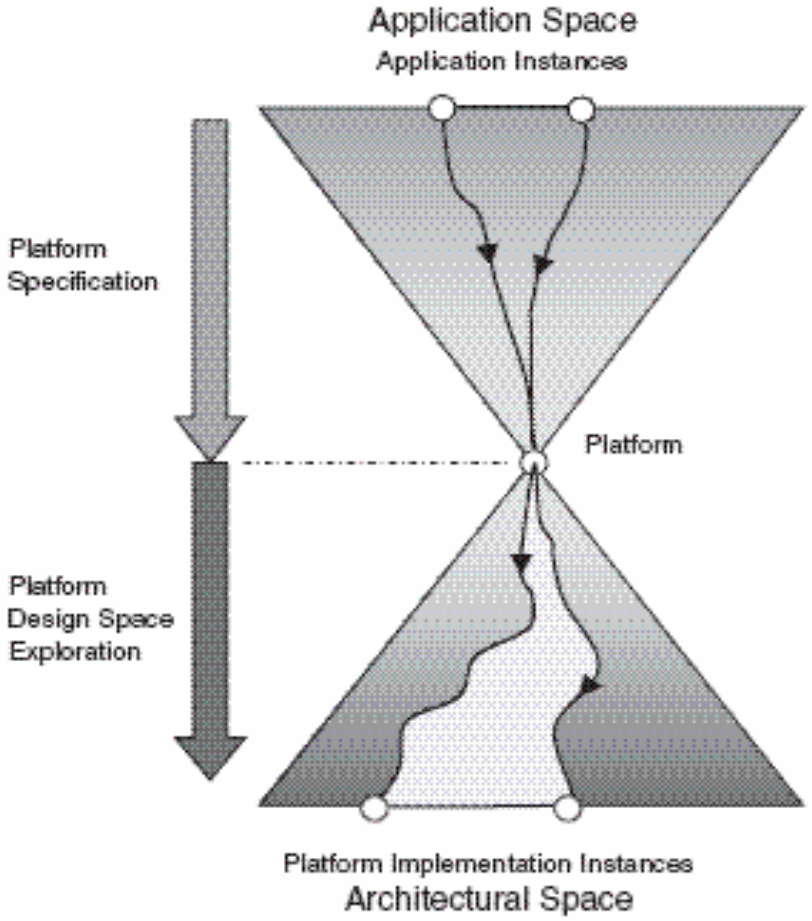
Stylized architecture

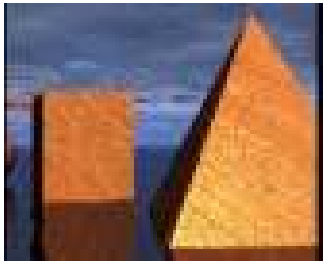


NXP Semiconductors Nexperia Platform



Platform based partitioning

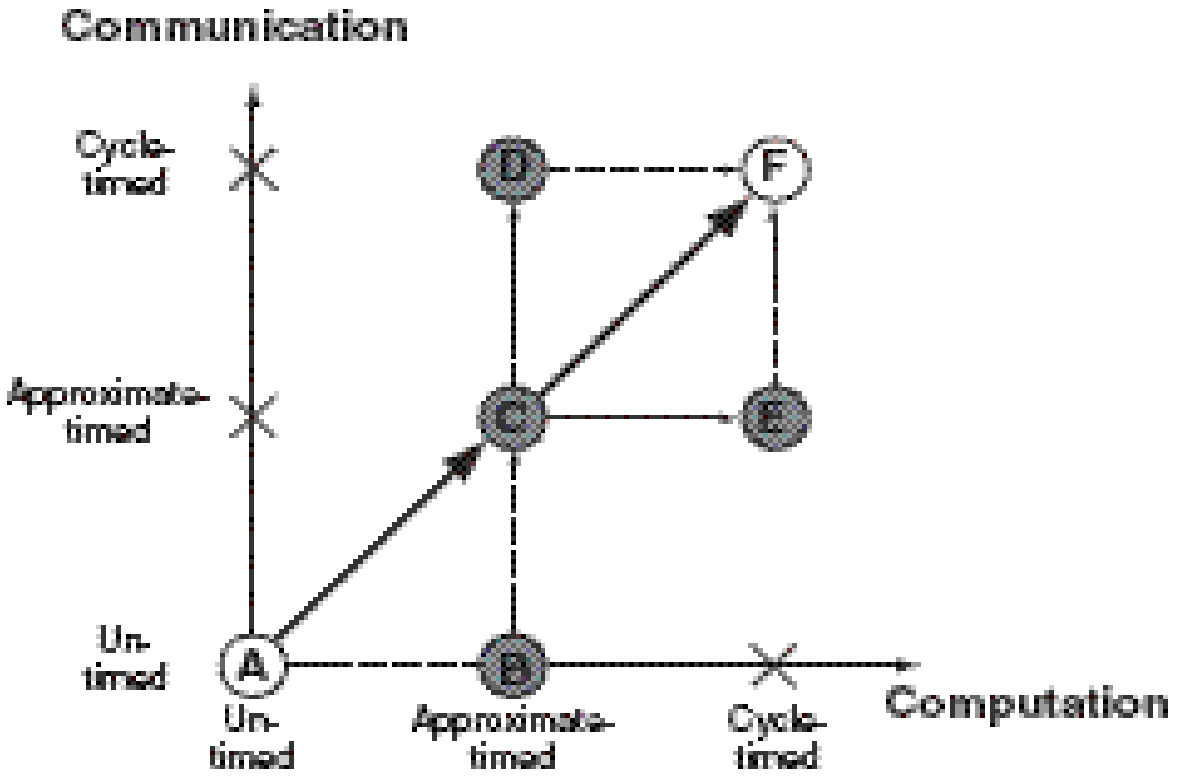


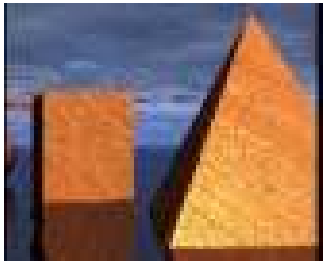


Mapping Function to Architecture

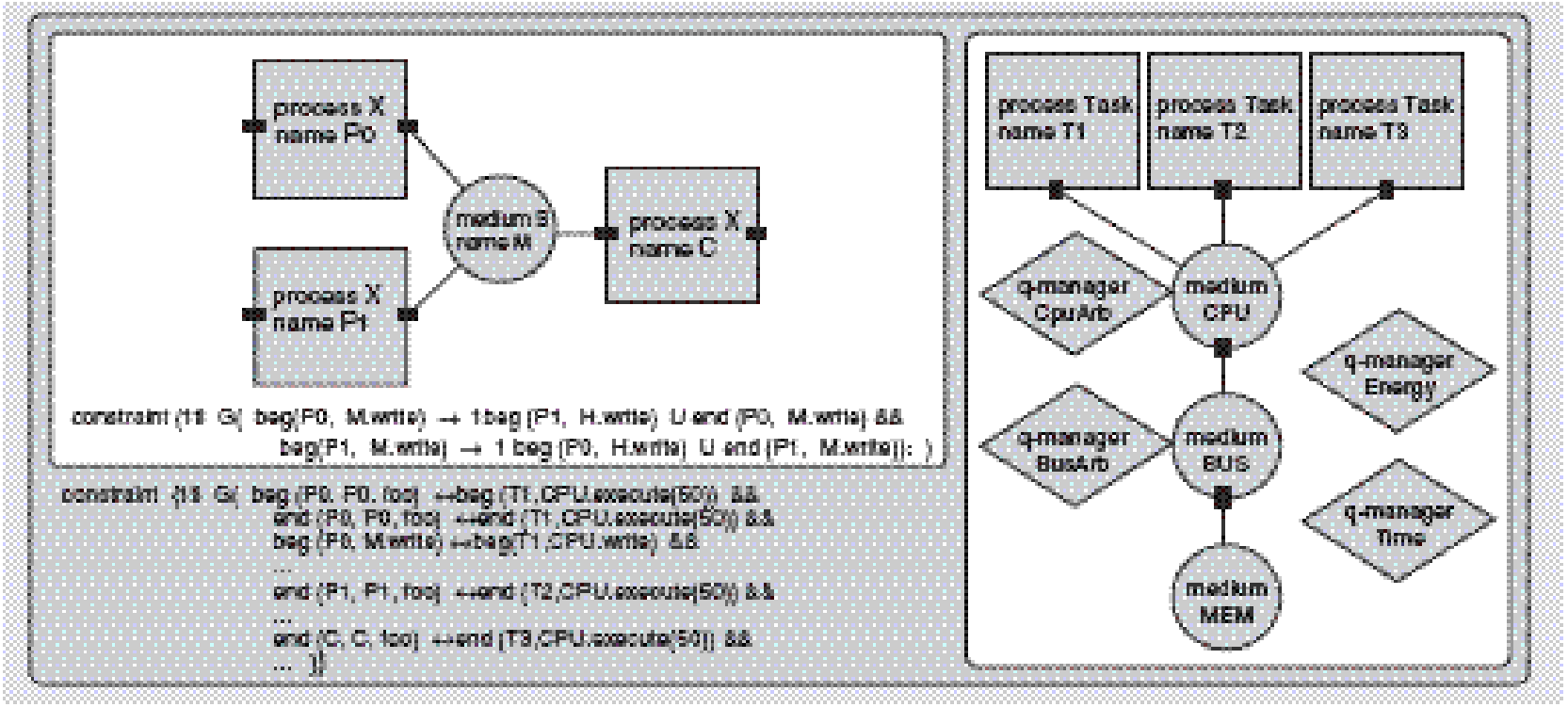
Walking through
The design space

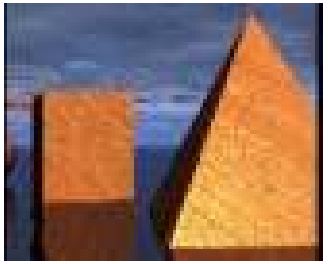
……..after Gajski





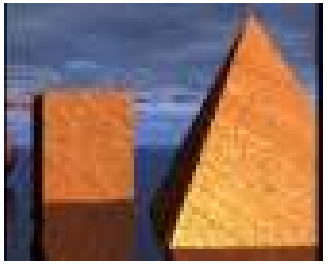
Metropolis





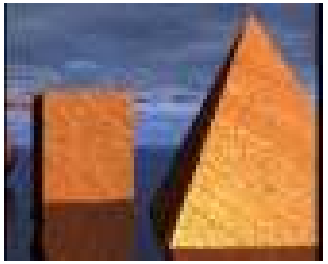
Implementation: HW

- Using a platform-based approach:
  - Becomes a configuration process for the platform, with
    - A minimal amount of new HW block creation
    - Avoid new HW at all costs
    - For flexibility and risk reduction, map new function to SW on processors if at all possible
    - Configurable processors are an interesting way to have SW often with HW performance



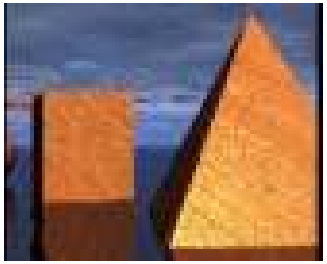
Implementation: SW

- "SW-SW Co-design": Partitioning over multiple processors
  - Heterogeneous
    - Classic: RISC + DSP
  - Homogeneous
    - Classic: Symmetric MultiProcessors (SMP)
- Partitioning into multiple tasks
  - Task scheduling and dependencies
  - Inter-task communications
    - APIs: Message passing, shared memory
  - Real-time dynamic vs. static or quasi-static scheduling
  - Worst case execution time (WCET) estimation



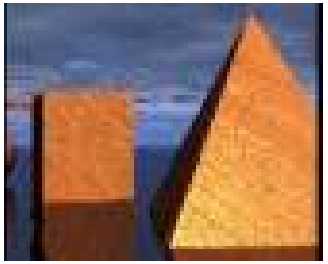
Operating systems and memory

- Provide resources for SW task management
- Commercial RTOSs
- Application layering and APIs
- Hardware dependent SW
- Custom (synthesized) OSs
- Memory partitioning and optimization
  - Atomium (IMEC – Catthoor et al)



Implementation: Reconfigurable

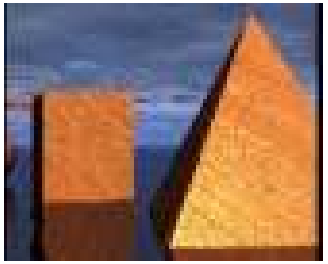
- New implementation option when added to classical HW and SW options
- Many possibilities exist between pure ASIC/ASSP, pure SW, pure FPGA
  - ASIC with FPGA region
  - FPGA with fixed cores (eg. Xilinx Virtex series)
  - Custom design with reconfigurable region
  - Configurable processor with instruction extensions mapped to reconfigurable logic (cf. Stretch)
- Various programming models and tools exist
  - Eg. Academic research (GARP), Simulink based (BEE)
  - Industrial: Stretch, Xilinx, Atmel, Altera



Interfaces

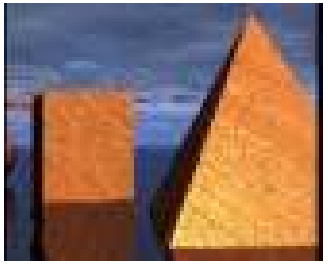
- Communication template instantiation
  - Early example is CoWare (Leuven research, mid-1990s; was commercial “N2C”)
  - Many other examples exist
- Interface synthesis
  - Automatic generation of adaptors between incompatible communications layers
  - E.g. FSM based adaptors
  - Research (Passerone et. Al.)

Post-Partitioning Analysis

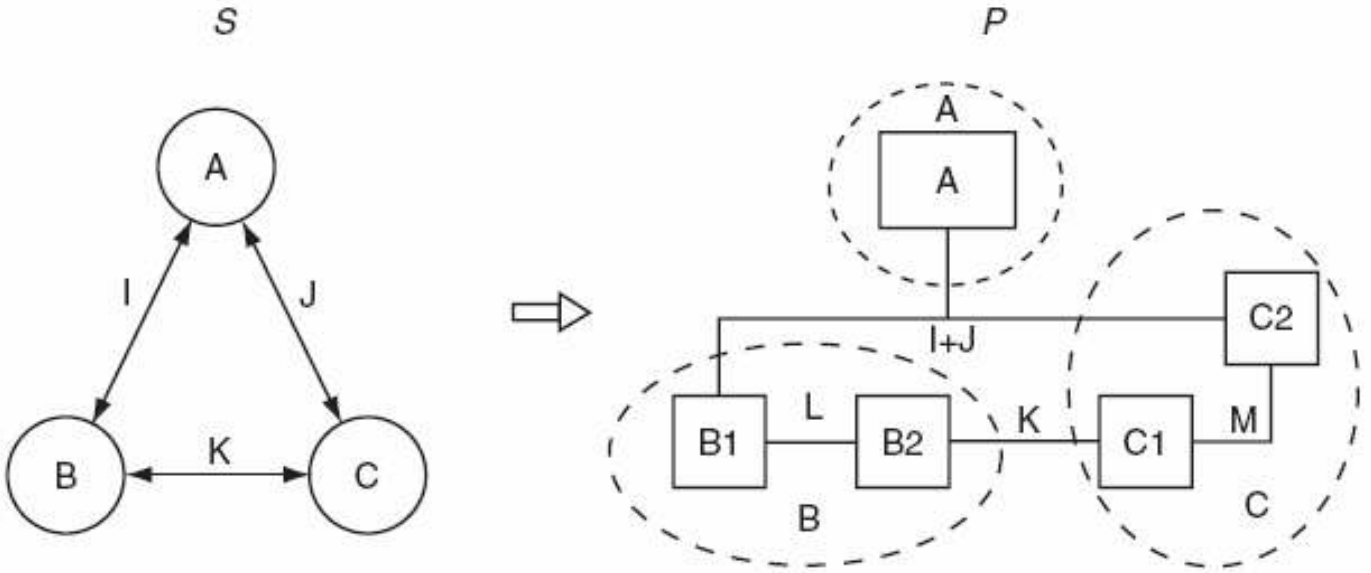


Post-Partitioning Analysis

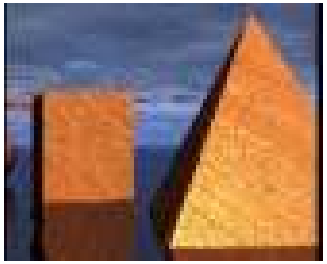
- We now have hardware and software
- Function and architecture have merged
- Need to verifying partitioning choices
- Need to establish the models and framework necessary for verification
- Interfaces become very important
  - Between functions
  - Between groups



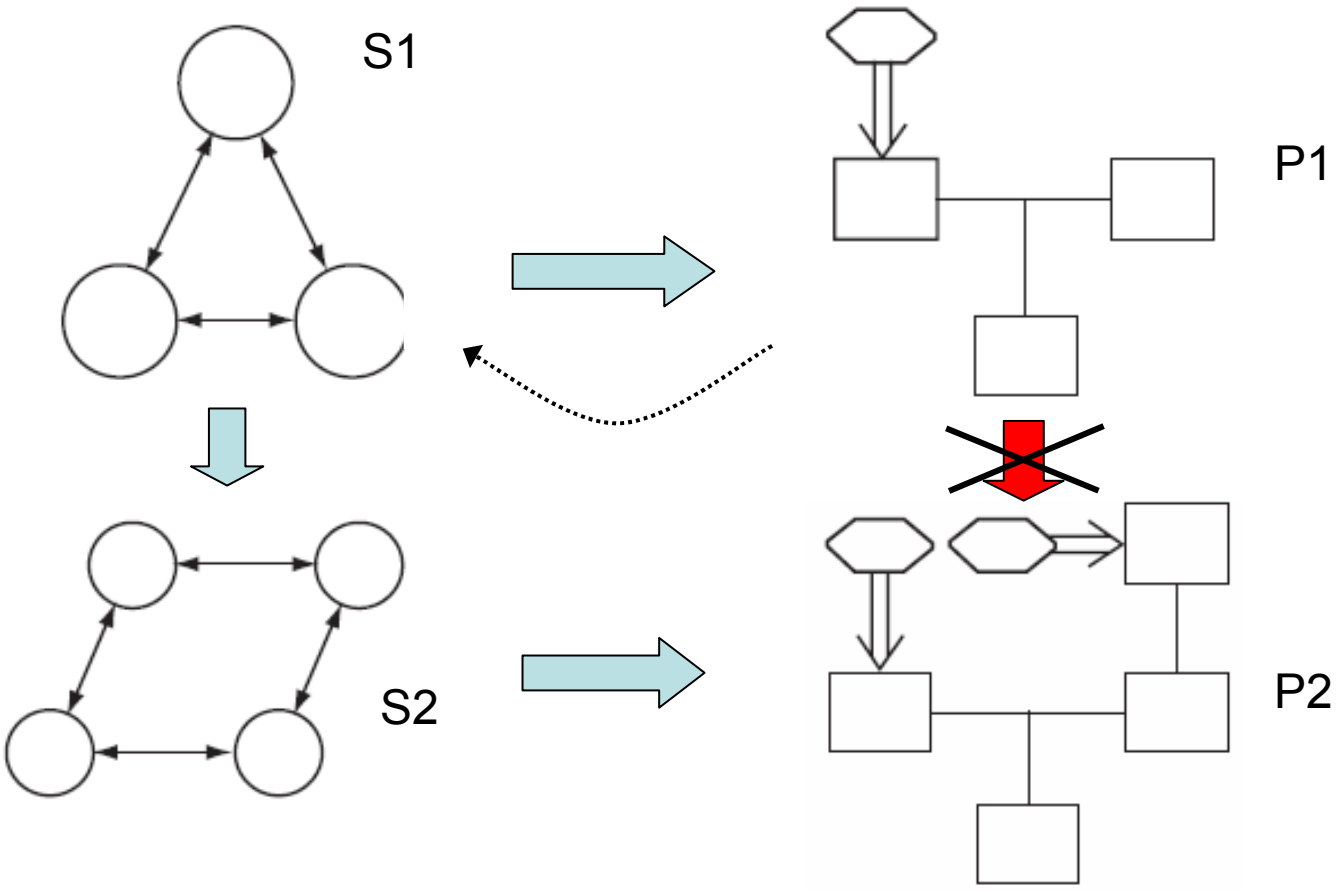
Interfaces

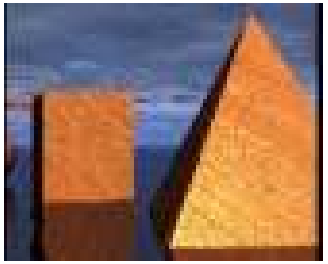


- Interfaces must be owned
- As partitioning continues new interfaces are created



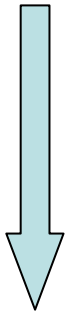
Maintaining the models



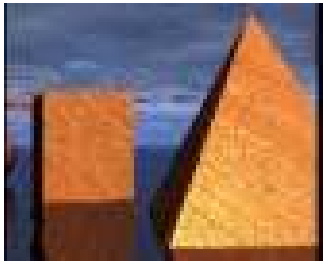


Hardware / Software

- Three modeling options
  - Model HW and SW in a single model
    - Virtual System Prototype
    - Usually used for SW verification
  - Filter / Translate HW/SW communications
    - Separates modeling concerns
    - Must be careful about implicit effects of interfaces
  - Model SW running on the HW
    - Traditional HW/SW co-verification
    - Possible performance issues

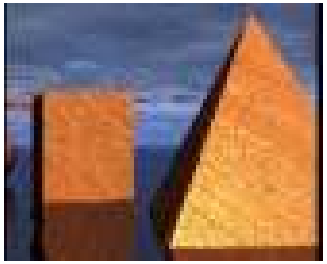


Migration Path

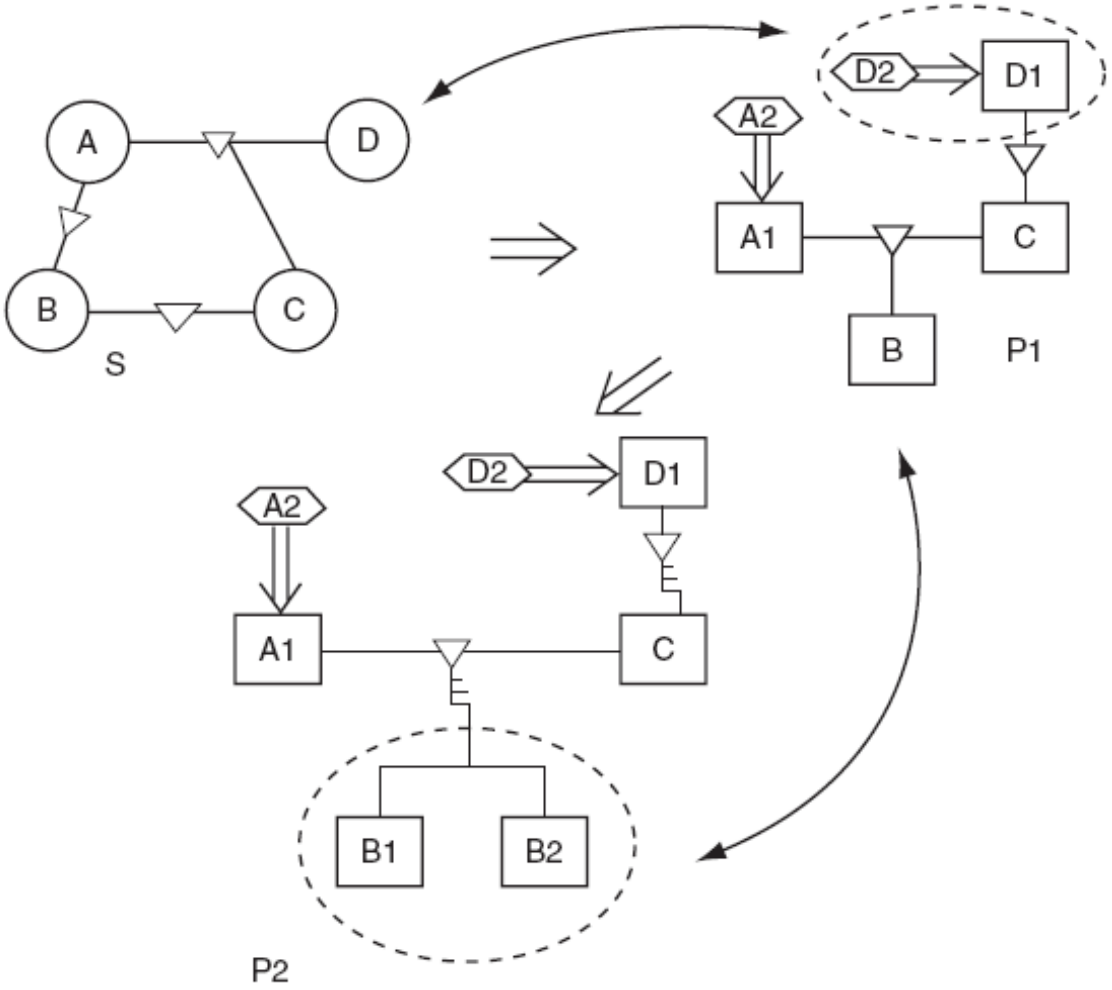


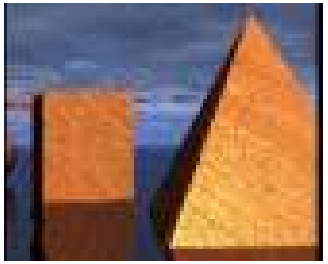
Interface Models

- Interface models are key to system model migration
  - Wire / Event level
  - Method / Transaction level
  - Need smooth migration between them
- Must be able to:
  - Evaluate system models with abstract interfaces
  - Evaluate interface implementation with system models
  - Evaluate implementation models with abstract interfaces



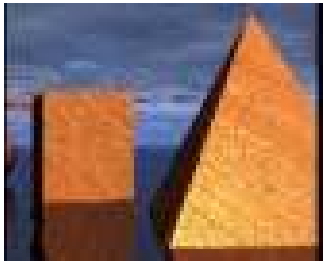
Model and Interface Refinement





Interface ‘languages’

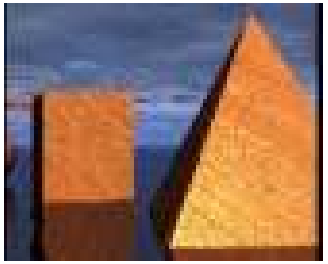
- OSCI – TLM
  - Defined interface views. Good start but not sufficient
- GreenSoCs – GreenBus
  - Separates interface protocol from transport
  - Concepts to be moved into OSCI
- Spiratech (Mentor) CY language
  - Declarative interface specification language
  - Supports abstraction migration



Analysis Possibilities

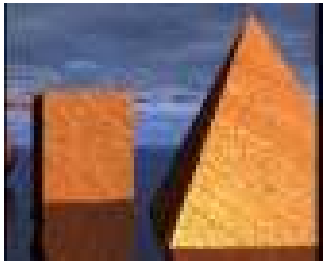
- Functional
- Performance
- Interface
- Power
- Area
- Cost
- Debuggability

Verification

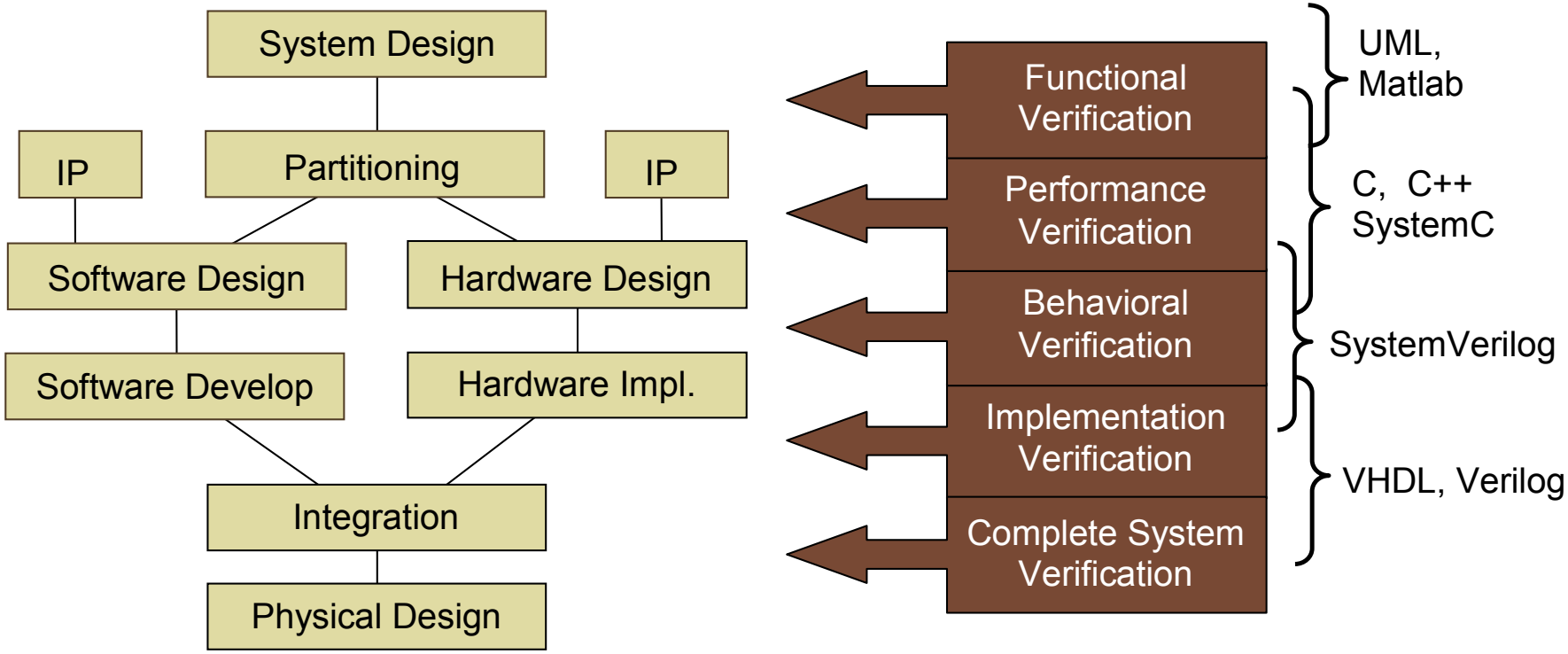


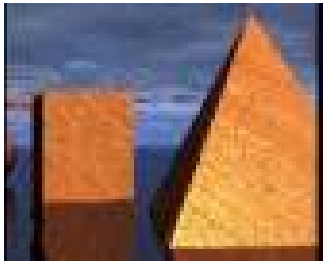
When Does Verification Begin?

- Verification starts as soon as a project is conceived
  - In early stages it is ad-hoc
  - Based on experimentation
- After partitioning verification becomes more formalized
  - Structure becomes more stable
    - Major decisions fixed
    - Enables verification plan creation



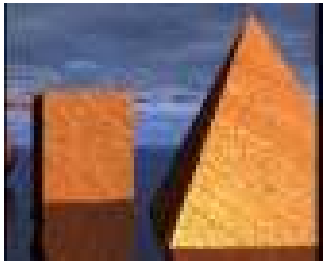
Facets of Verification





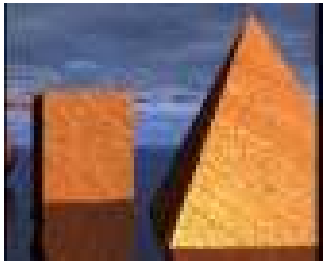
Verification Fundamentals

- Verification is the comparison of two independently obtained models
  - Formal verification is exhaustive and analytical
    - Need a partial model of the environment (constraints)
  - Simulation is a sampling approach
    - Random generation also requires a model of the environment (interaction model)
    - A set of directed tests is also a model
- An abstract model synthesized into an implementation cannot serve as a reference model
  - This would only verify the synthesis tool, not the function
  - This is equivalence checking



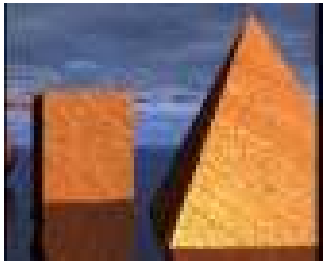
Positive and Negative Verification

- Negative Verification
  - Show the non-existence of bugs
    - Predominates today
    - Shows a block will function under all conditions
    - With simulation we cannot achieve 100% negative verification
      - Property checking not ready yet
    - There are many horror stories about bugs found late in the cycle
    - Tends to imply all bugs created equal
- Positive Verification
  - Show that the design actually does something useful
    - Prioritizes important functionality over others
    - More predictable schedules
    - BUT – changes may have catastrophic consequences
- Need to balance positive and negative verification



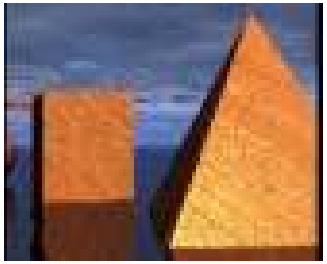
Verification Plan

- A verification plan is used to:
  - Formulate a strategy
  - Develop tactics
- The verification plan must answer two questions:
  1. What is the scope of the verification problem?
  2. What is the solution to the verification problem?



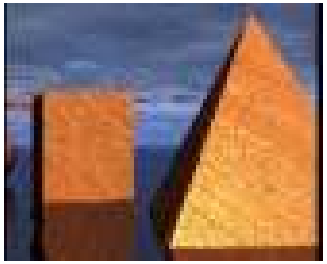
Verification Plan Outline

1. Introduction ..... *what does this document contain?*
2. Functional Requirements ..... *opaque box design behaviors*
   - 2.1 Functional Interfaces ..... *external interface behaviors*
   - 2.2 Core Features ..... *external design-indep behaviors*
3. Design Requirements ..... *clear box design behaviors*
   - 3.1 Design Interfaces ..... *internal interface behaviors*
   - 3.2 Design Cores ..... *internal block requirements*
4. Verification Views ..... *time-based or functional features*
5. Verification Environment Design ..... *functional spec of the verification env*
   - 5.1 Coverage ..... *coverage aspect functional spec*
   - 5.2 Checkers ..... *checking aspect functional spec*
   - 5.3 Stimuli ..... *stimulus aspect functional spec*
   - 5.4 Monitors ..... *data monitors functional spec*

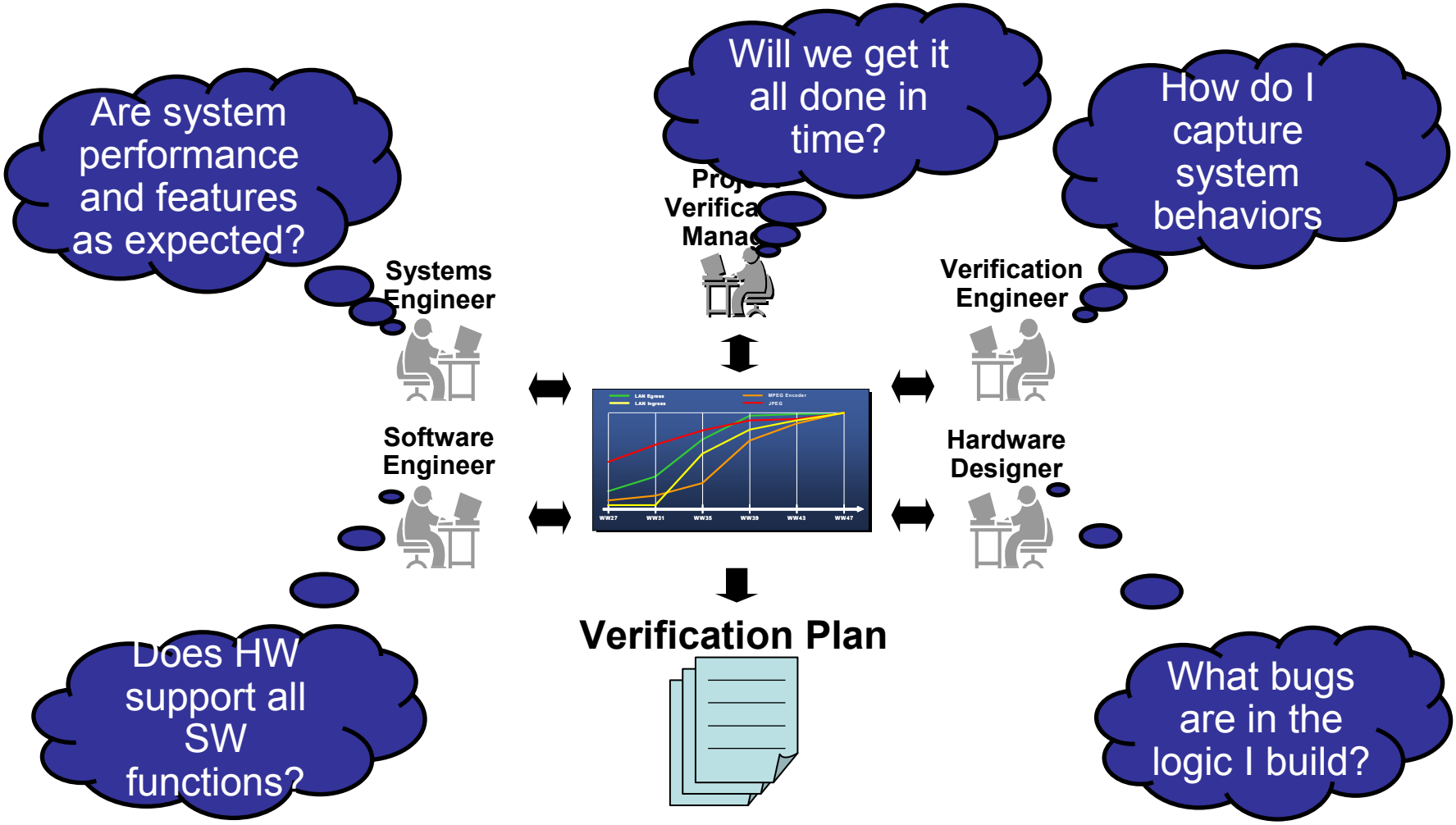


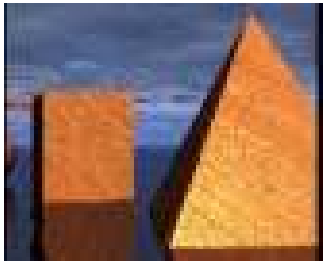
Specification Analysis

- Identify the feature set of the design and its corner cases
  - A corner case is one or more data values or sequential events that, in combination, lead to a substantial change in design behavior
- Can be done two ways
  - Bottom up
    - Suitable for small specifications
  - Top down
    - Preferred for most design specifications



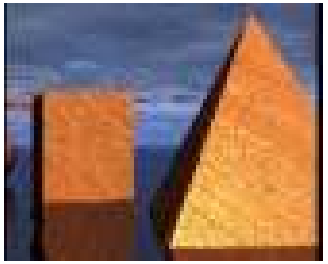
Top-Down Analysis Contributors





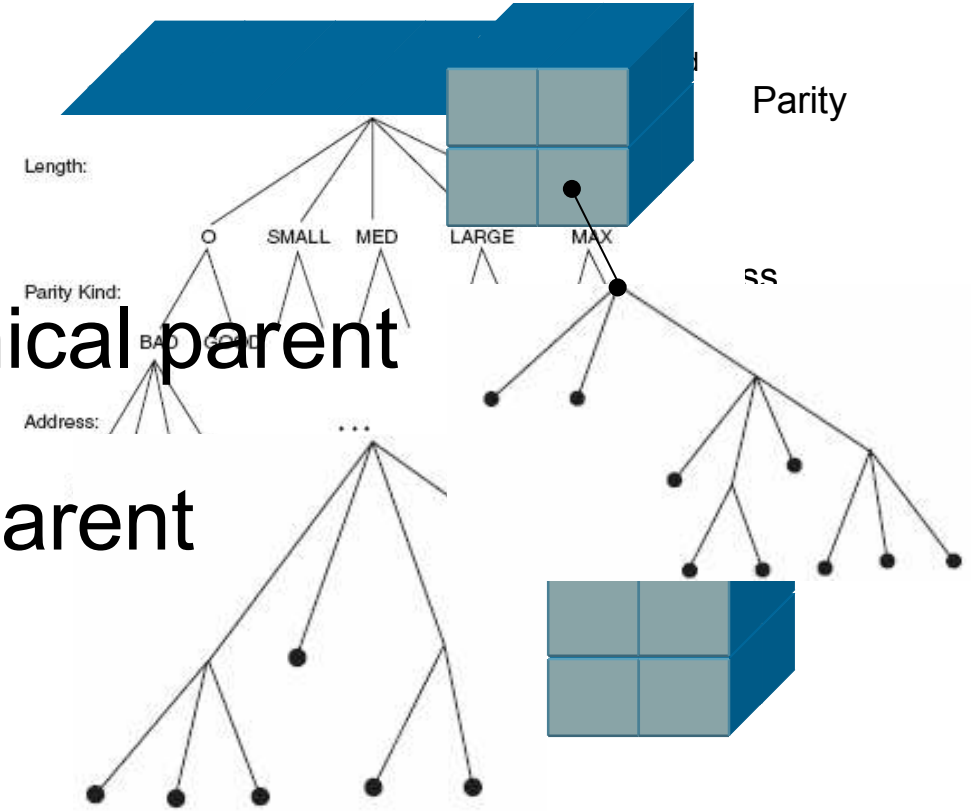
A Coverage Model is the Result

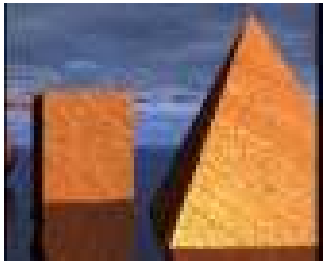
- A coverage model is an abstract representation of device behavior composed of attributes and their relationships. The relationships may be either data or temporal in nature.



Coverage Model Structures

- Matrix
- Hierarchical
- Hybrid with hierarchical parent
- Hybrid with matrix parent

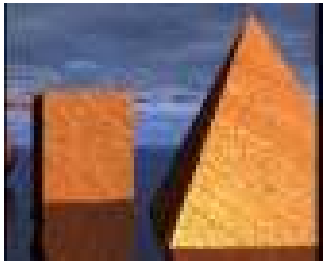




Coverage Model Detailed Design

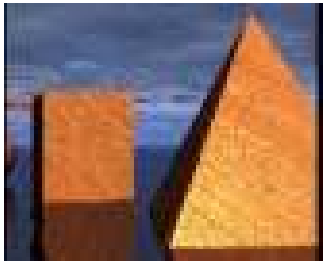
Attribute	Value	Sampling Time	Correlation Time		
			Packet sent		
Length	*[0, SMALL, MEDIUM, LARGE, MAX]*	Length computed	*0*	*SMALL, MEDIUM, LARGE*	*MAX*
Address	*[0..3]*	Packet sent	*0..2*	*3*	*0..2*
Parity kind	*[GOOD, BAD]*	Parity computed	*GOOD*	*GOOD*	*BAD*

- For each attribute, answer the questions:
  - What must be sampled for each attribute value?
  - Where in the verification environment or DUV should the value be sampled?
  - When should the data be sampled and correlated?



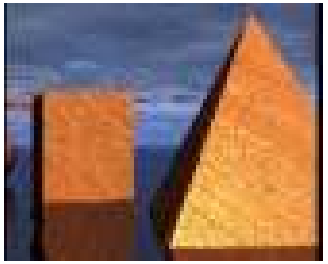
Verification Environment Components

- Coverage Models
- Dynamic Verification
  - Stimulus generator
  - Response checking
- Static Verification
  - Limited to implementation verification
  - Constraints
- Execution Management
- Result Analysis
  - Failure analysis
  - Coverage analysis



Post Silicon Debug

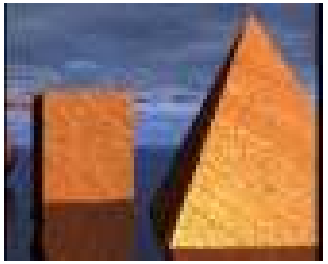
- The likelihood that your design will work first time ?
  - Small today
  - Getting smaller
- Must plan for silicon debug
  - Raise visibility within the chip
  - Add controllability
  - Potentially add modifiability
- Lots of progress in this area



The Prescription

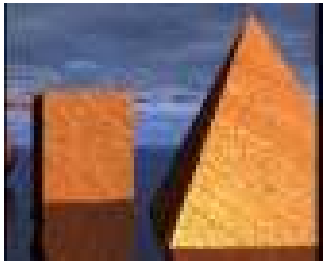
- Capture all design intent in specifications
  - Executable and natural language
- Perform rigorous verification planning
  - Quantify the scope of the problem
  - Specify the solution to the problem
- Modulate coverage model fidelity
- Use the plan to drive the verification process

HW and SW Implementation

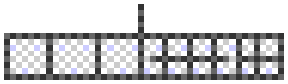
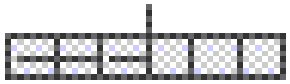
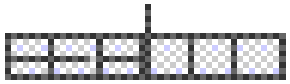
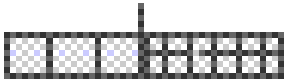
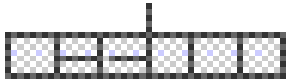
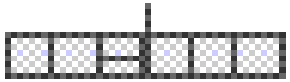
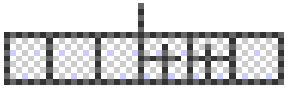
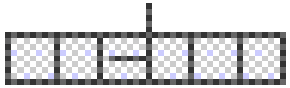
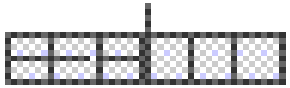
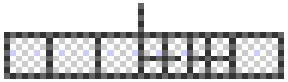
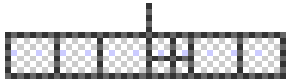
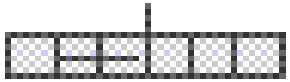
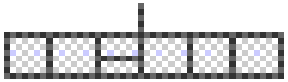
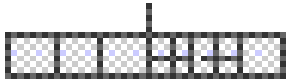
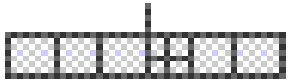
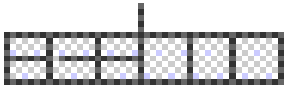
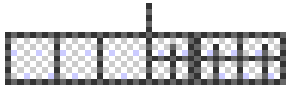
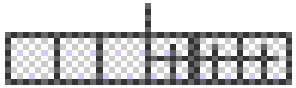


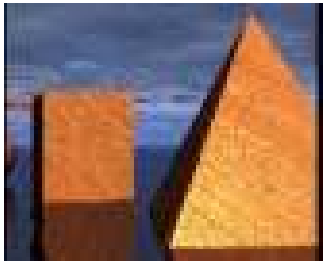
Hardware Implementation

- A range of implementation architectures are possible:
  - General-purpose fixed ISA CPU
  - Configurable and extensible processor tailored for the application
  - DSP (which may be based on an extensible processor)
  - VLIW processor (which may be based on an extensible processor)
  - FPGA (which may incorporate one or more processors)
  - ASIC/ASSP hardware blocks



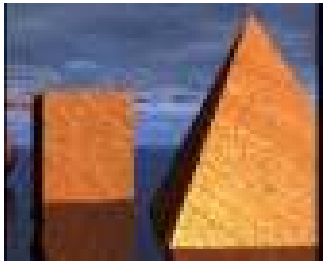
Comparison

Tech	Programmability	Performance	Power
CPU			
EPU			
DSP			
VLIW			
FPGA			
ASIC			

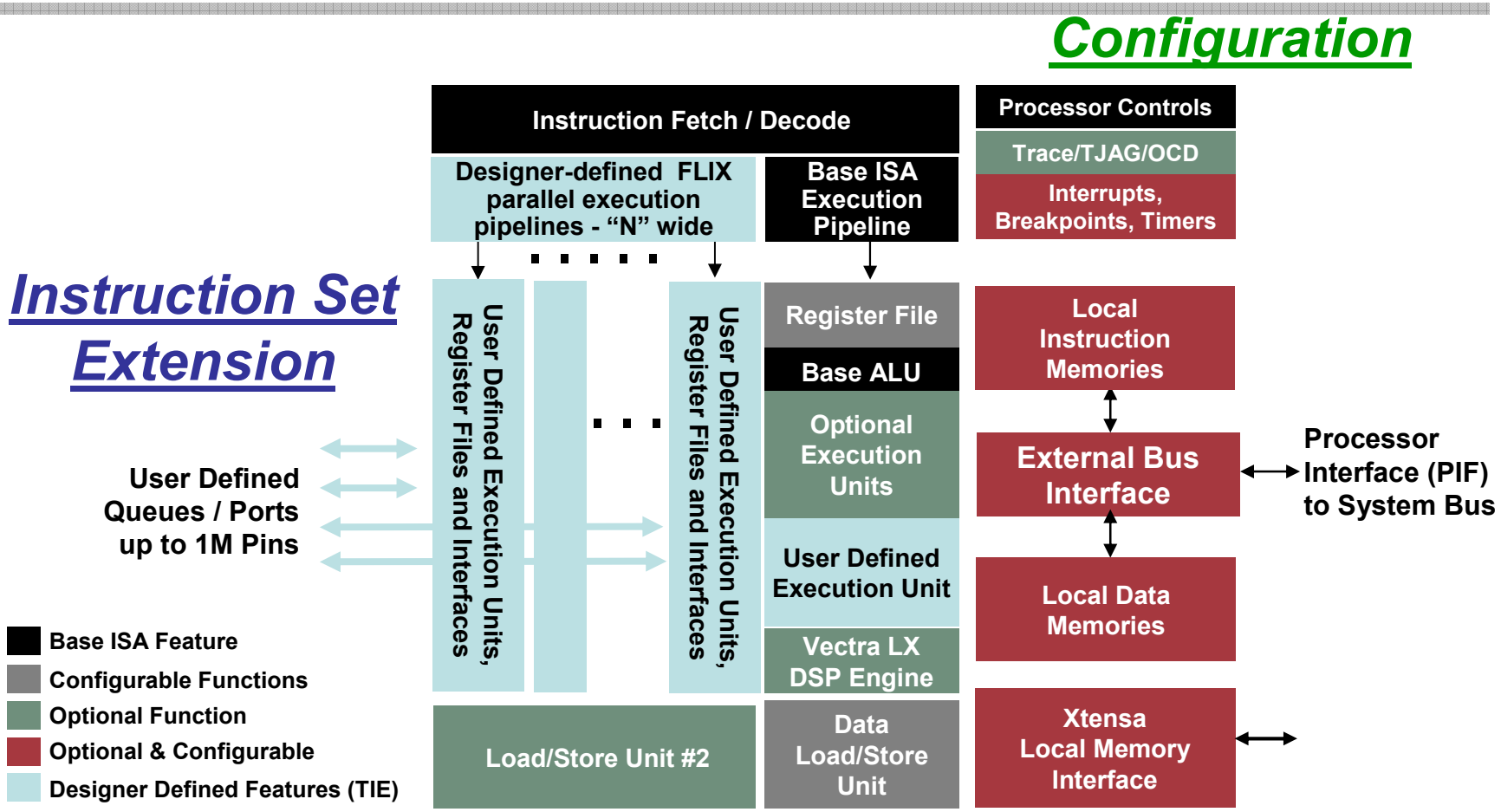


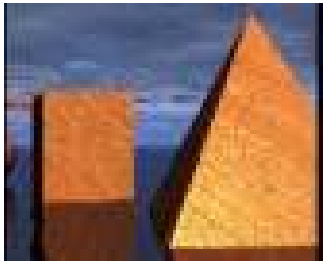
Processor Alternatives

- Configurable and Extensible
- DSP
- VLIW
- Application Specific Coprocessors
  - Note: The first category may subsume all the rest, depending on the technology offered

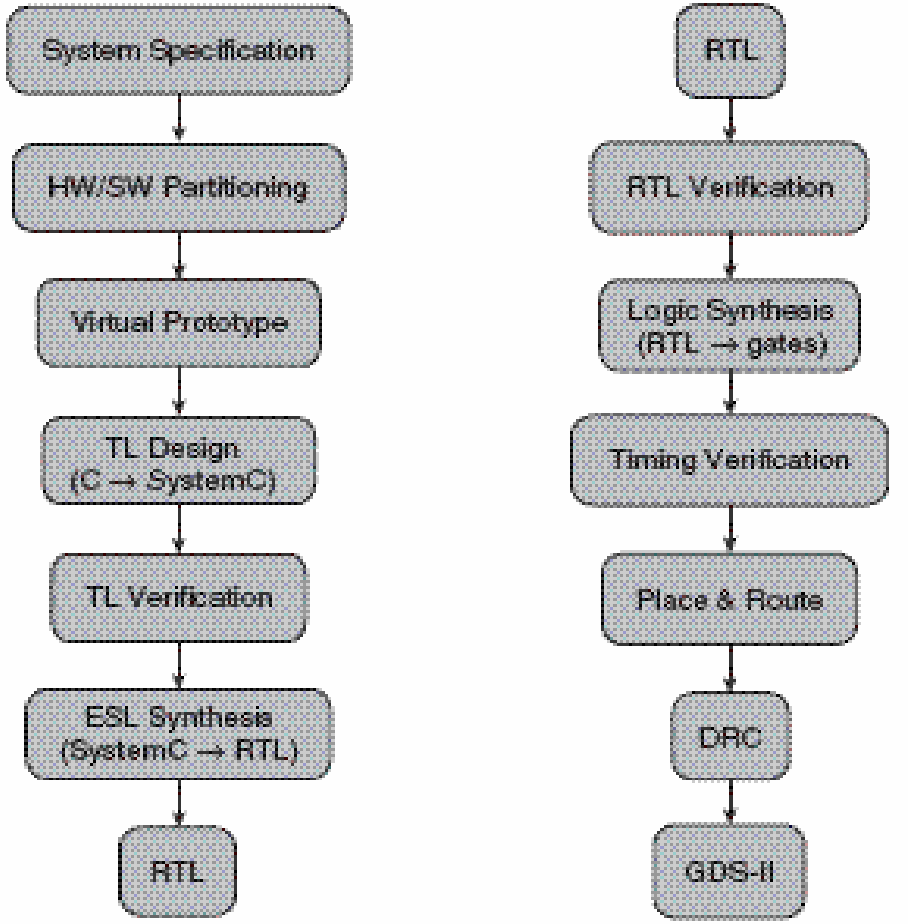


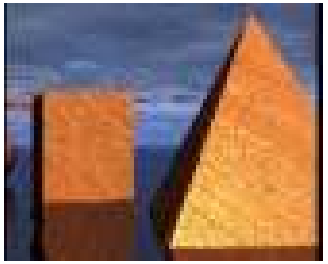
Example of configurable extensible processor





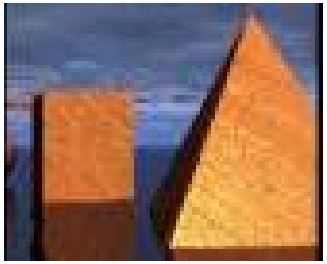
ESL Synthesis Design Flow





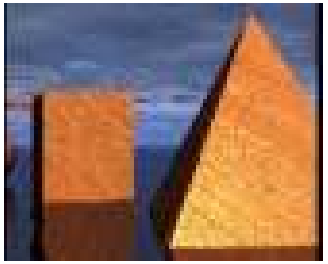
High-level/Behavioral synthesis: past

- Used different code than RTL synthesis although usually used Verilog or VHDL as inputs
  - Multicycle
  - Loops
  - Memory access via arrays
- But fell short due to:
  - Input language – HDLs not natural for algorithms
  - Timing convergence issues
  - Verification of RTL implementation



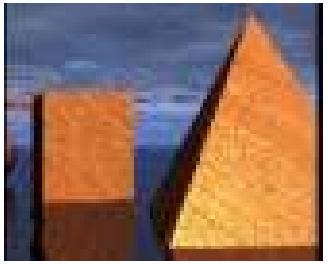
ESL Synthesis: present

- Overcomes limitations of past behavioral or high-level synthesis:
  - More natural input languages
    - C or a C-related language (C++, SystemC, special C dialects)
    - Support for:
      - Structure
      - Concurrency
      - Data types (e.g. bit-wise, fixed-point)
      - Operation overloading to support polymorphic typing



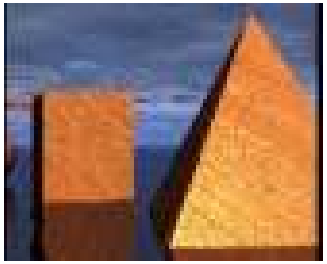
ESL Synthesis: Other requirements

- Natural input/output declarations
  - Types or pragmas
- Verification compatibility
  - E.g. using SystemC TLM models together with RTL-level adaptors
- Control over quality of results
  - Timing convergence
  - Scheduling/latency constraints
  - Resource allocation
  - Compatibility with RTL flow “back end”
  - General constraint handling
  - Design space exploration



Not your grandfather's behavioral synthesis

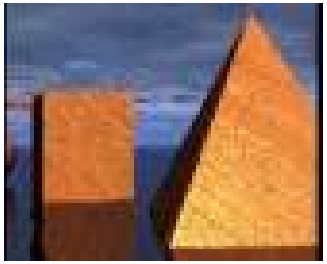
- New tools have emerged with more credible evaluation and some adoption results
  - Forte Cynthesizer
  - Mentor Catapult
  - NEC Cyber



Example code

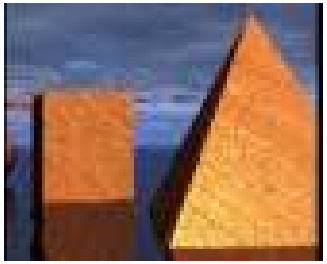
```
/* Metaports and port data types */
typedef dctelem< sc_uint<8>, DCT_SIZE, DCT_SIZE > UINT8_DATA;
typedef p2p< UINT8_DATA, IF_LEVEL > UINT8_IF;
typedef dctelem< sc_int<12>, DCT_SIZE, DCT_SIZE > INT12_DATA;
typedef p2p< INT12_DATA, IF_LEVEL > INT12_IF;

/* Module Definition */
SC_MODULE(dct)
        {
        public:
        sc_in< bool > clk;
        sc_in< bool > rst;
        UINT8_IF::base_in in;
        INT12_IF::base_out out;
        SC_CTOR(dct) : clk( "clk" ), rst( "rst" ), in( "in" ), out( "out" ) {
                SC_CTHREAD( thread0, clk.pos() );
                watching( rst.delayed() == 0 );
                }
        private:
        void thread0();
        void dct_2d( sc_int<16> data[DCT_SIZE][DCT_SIZE] );
        };
```



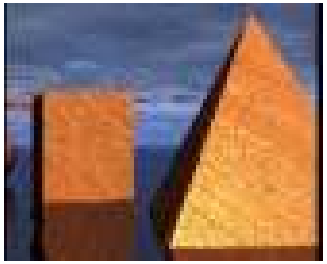
Example Code

```
void dct::thread0()
        {
        UINT8_DATA in_data;
        INT12_DATA out_data;
        sc_int<16> buf[DCT_SIZE][DCT_SIZE];
        {
                CYN_PROTOCOL( "reset" );
                in.reset();
                out.reset();
                wait();
        }
        while( true ) {
                for( int r = 0; r < DCT_SIZE; r++ ) {
                in_data = in.get();
                for( int c = 0; c < DCT_SIZE; c++ )
                buf[r][c] = in_data[c];
                }
                dct_2d( buf );
                for( int r = 0; r < DCT_SIZE; r++ ) {
                                for( int c = 0; c < DCT_SIZE; c++ )
                                out_data.d[c] = buf[r][c];
                                out.put( out_data );
                }
        }
}
```



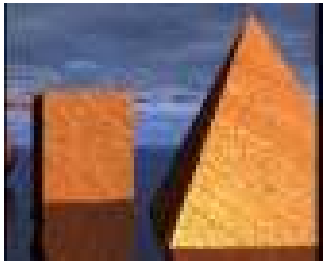
Example code – output ports

```
template <class T, typename L>
class p2p_base_out
        {
        public:
p2p_base_out(
        const char* name=sc_gen_unique_name("p2p_out") )
        : busy("busy")
        , vld( "vld")
        , data( "data")
        {}
// Interface ports
        sc_in<bool> busy;
        sc_out<bool> vld;
        sc_out<T> data;
// Binding functions
        template <class C>
        void bind( C& c ) {
        busy(c.busy);
        vld(c.vld);
        data(c.data);
        }
….
```

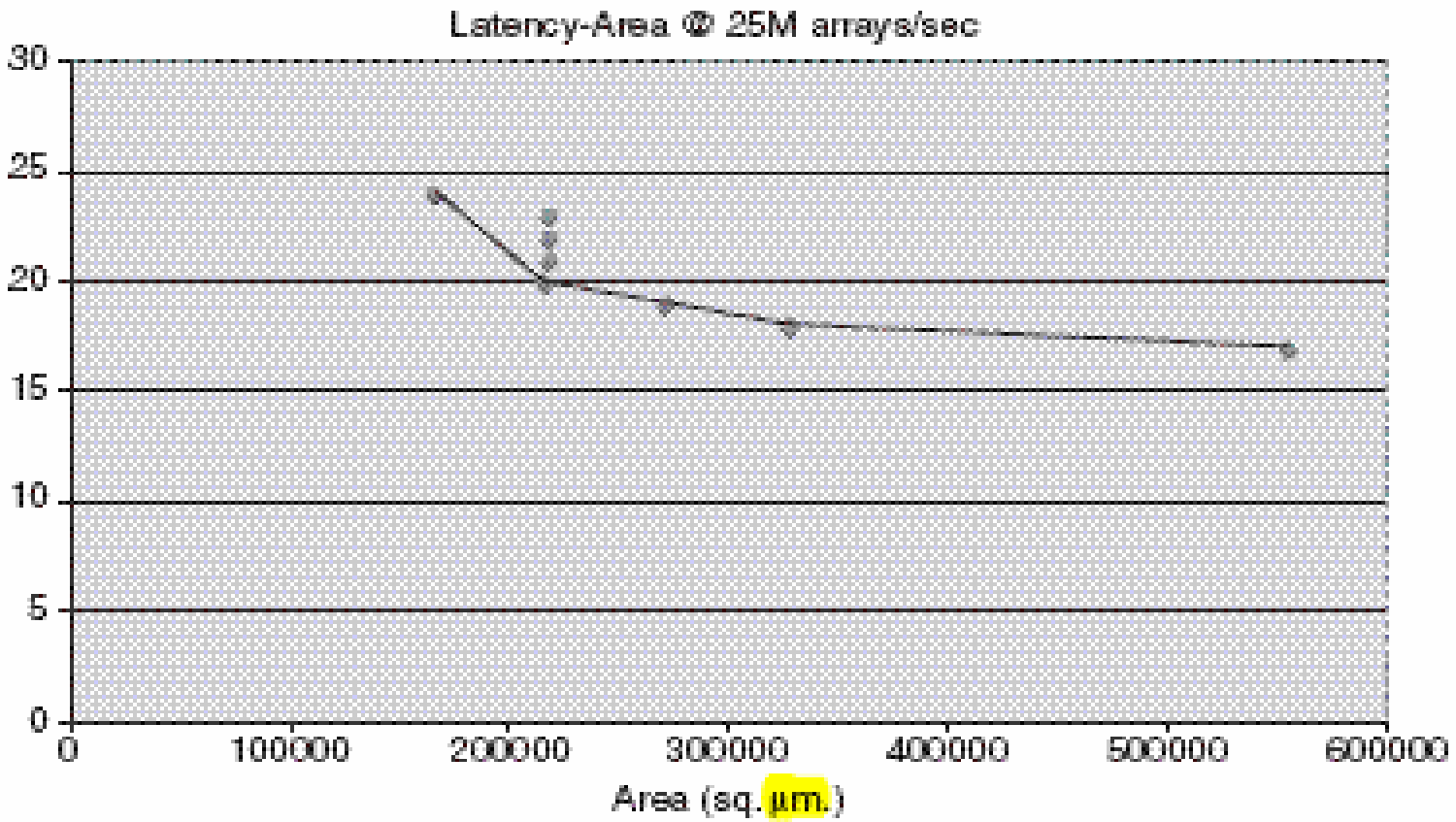


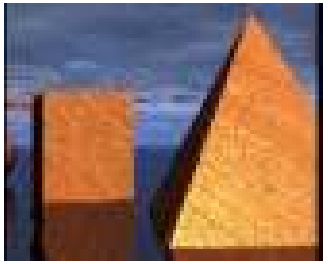
Constraints as pragmas

```
void dct::dct_1d( sc_int<16> data[DCT_SIZE] ) {
        CYN_DPOPT("dct_1d");
        …
        }
void dct::dct_2d( sc_int<16> buf[DCT_SIZE][DCT_SIZE] ) {
        …
        }
void dct::thread0()
        {
…
        while( true ) {
                CYN_INITIATE(8, "dct_pipe");
                for( int r = 0; r < DCT_SIZE; r++ ) {
                        in_data = in.get();
                        for( int c = 0; c < DCT_SIZE; c++ )
                        buf[r][c] = in_data[c];
                        }
                dct_2d( buf );
                for( int r = 0; r < DCT_SIZE; r++ ) {
                        for( int c = 0; c < DCT_SIZE; c++ )
                        out_data.d[c] = buf[r][c];
                        out.put( out_data );
                        }
                }
        }
```



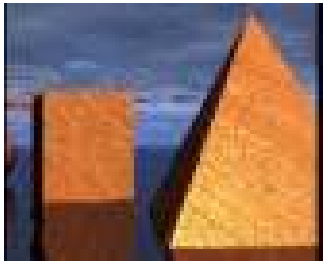
Design Space Exploration





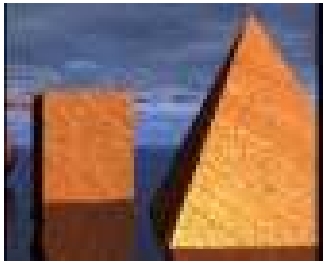
The Prescription

- Variety of implementation alternatives if function must be implemented in tuned hardware:
  - Reuse of IP blocks
  - Configuring RTL
  - Configurable extensible processor
  - Generating function-specific coprocessor
  - High-level or ESL synthesis
- This generation of ESL synthesis is real.
- ESL Synthesis uses C/C++/SystemC or other C dialects as input
- Verification environments that work between ESL and RTL levels have made progress
- Several commercial tools exist
- ESL Synthesis is thus a viable option for hardware implementation



Software Implementation

- Classical SW development methods
- Developing run-time software from ESL tools
- Developing software using ESL models as run-time environments (Virtual System Prototypes)



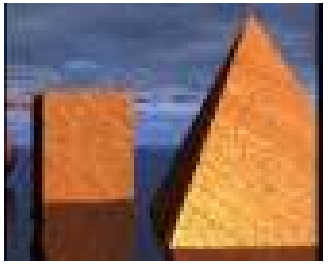
Classical SW development

- Performance Estimation
  - Historically, estimates often used processor "MIPS" ratings
  - But $MIPS_1 \neq MIPS_2$

➔Mandates use of ISSs

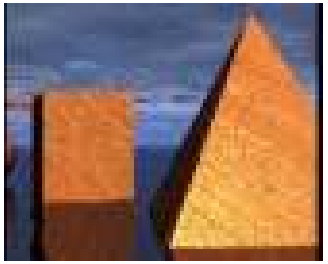
  - But standalone ISSs don't reflect the system environment, especially memory

➔Mandates use of Virtual System Prototypes



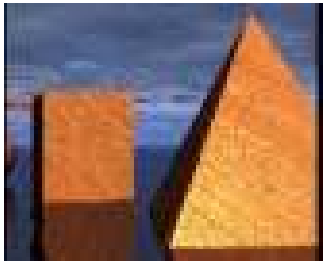
Classical Development Tools

- C is still dominant for embedded systems, with some C++, C# and Java
- Standard IDEs from various vendors
- Emulation, ICE, evolving to on-chip embedded processor trace



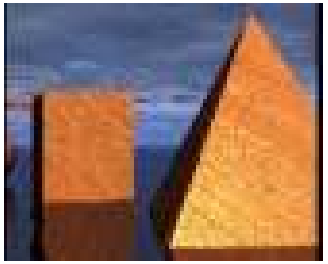
Developing run-time software from ESL tools

- Algorithmic, e.g.
  - MATLAB code generation to production code
    - Some companies (Accelchip, now part of Xilinx, was one)
    - Catalytic was offering this, but now focusing more on fast (C based) MATLAB modeling
    - Many issues of generating efficient executable code from actor libraries e.g. MATLAB, Simulink



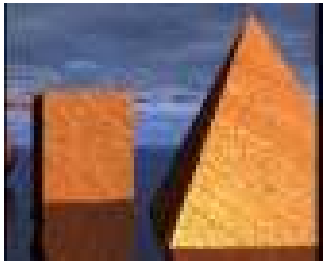
Developing run-time software from Models

- Control code viz. UML or SDL
  - Some success with this over the years
  - SDL has been used to generate real executable code as part of telecom protocol stacks
  - Code generation from UML is improving especially as recent versions, e.g. UML 2.0, have allowed better modeling and annotation of constraints to be incorporated
  - Continues to be a “Holy Grail” for the UML tools companies
  - Issues of code quality (especially optimizing across levels of the “stack”) and debugging continue to be issues
  - UML tools have become integrated with IDEs over time

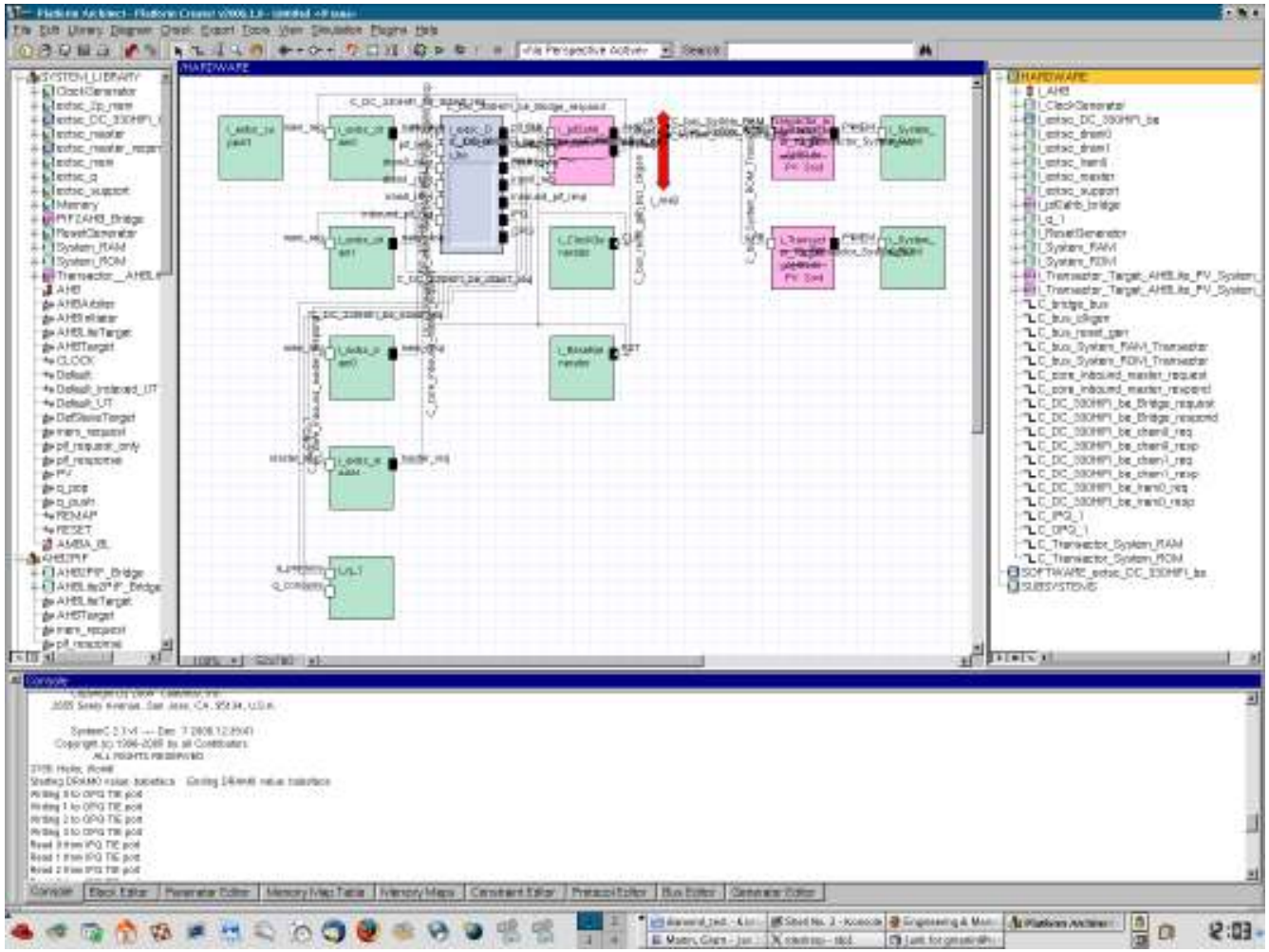


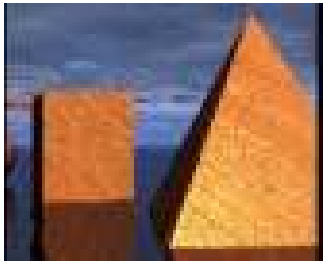
Developing software using ESL models

- Models are run-time environments (Virtual System Prototypes)
- This is the area that is “hottest” as a pragmatic capability
- Commercial tools available from several vendors:
  - CoWare, VaST, Synopsys (Virtio), ARM (Axys)
  - Need support from IP model vendors
    - ARM, Tensilica, Ceva, others
    - As much value lies in the standard bus libraries for AMBA AHB, AXI, OCP-IP as in the environment
    - Analysis capabilities useful to some
  - Often use both cycle-accurate, TLM and Fast functional models for processors
- Debug/observability/integration with IDEs important
- Likely to see rapid development in this area



Example

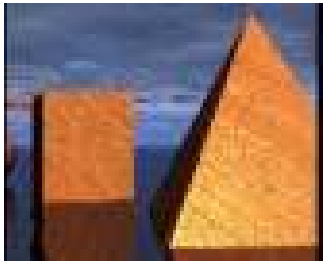




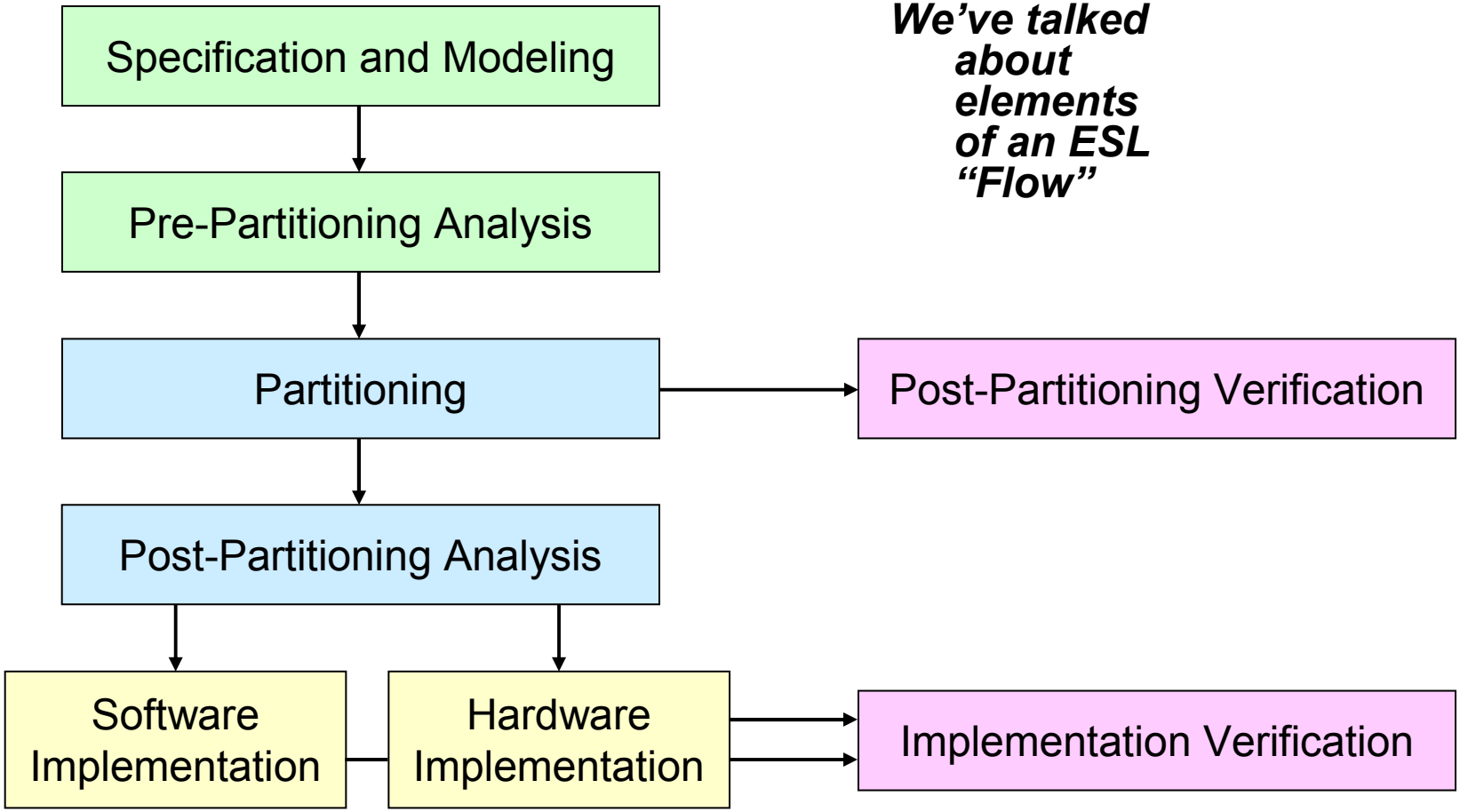
The Prescription

- The two key ESL capabilities important for SW today:
  - Code generation from Models
    - Keep watching the skies!
    - Progress is being made
  - Virtual System Prototype Models
    - A reality today
    - Acquire and use!

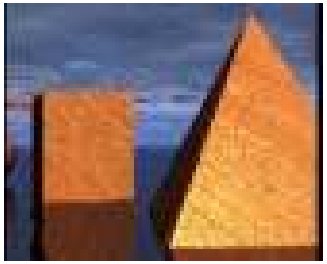
Summary, Futures and Conclusions



Summary

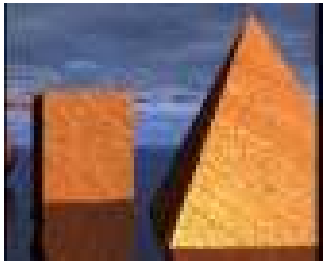


We've talked about elements of an ESL "Flow"



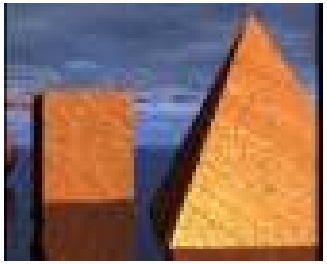
Summary

- Pieces of this ESL “flow” are at different stages of evolution
- Specification
  - Lots of languages to choose from
  - Practices not yet standardized
  - SystemC likely to become standard modeling glue
- Pre-Partitioning analysis
  - Static methods have history, few users
  - Dynamic (simulation) methods have history and lots of users, especially in dataflow/algorithmic space
  - Beware of implementation artifacts confusing early analysis
- Partitioning
  - Still more of a manual process, supported by various models and simulation
  - Output of hardened partitions may be a Virtual System Prototype (VSP)
  - Commercial modeling tools beginning to become credible
  - SystemC/OSCI playing a growing role with commercial tools



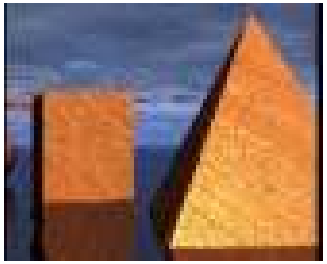
Summary

- Post-Partitioning analysis
  - May feedback into partitioning process
  - Availability of modeling/simulation environments and IP models important to back this up
- Post-Partitioning verification
  - Interacts with other kinds of modeling and analysis environments
  - Important step in building reusable verification plans that can migrate to implementation step
- HW implementation
  - Many different implementation options – configurable processors, coprocessor synthesis, high-level synthesis
  - Emerging ESL synthesis is “not your grandparent’s”
  - Growing and credible use of ESL synthesis for blocks that *must* be HW
- SW implementation
  - But HW takes the back seat to SW – SW everywhere we can; HW only where we must
  - SW implementation from ESL models (UML, SDL, etc.) still rare
  - SW verification on ESL models (VSPs) is growing
- Implementation Verification



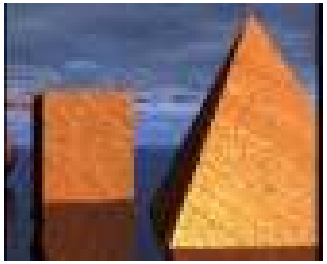
Futures

- Research
- Globalization
- Value Migration
- Education
- Commercial EDA



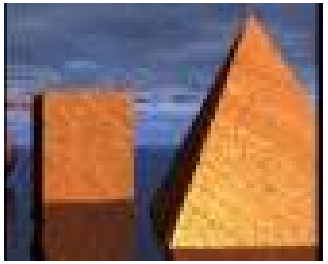
Research

- Metropolis
- SPACE
- Multi-processors
- Emerging architectures
  - Homogeneous systems
  - Heterogeneous systems
  - ROSES



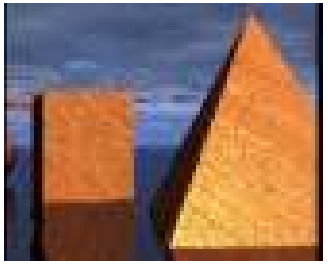
Globalization

- More people able to participate in the high-tech economy
- More people can contribute
- More people can consume
  - Products may need wider range of derivatives across wider global markets
- We have been here before
  - Railroads
  - Replacement of wind by steam at sea
  - Communications and air travel
- Needs:
  - IP policy harmonization to reasonable set of common accepted practices
  - Using standards to promote market development, not engage in short-term protectionism



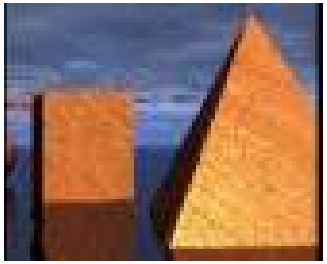
Value Migration

- Past: focus on EDA tools
- Emergence of IP industry
  - Star vs Less-than-stellar
  - IP value hard to maintain: migration to platforms
- Cost of verification
  - Can verification IP attract and maintain value?



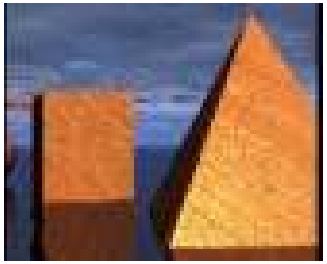
Education

- What do future designers need to know?
- If they need to understand ESL
  - What do they stop learning?
  - How do they cross the HW-SW (Electrical Engineering- Computer Science) divide?
  - How do we move people from the details of detail to the details of systems?



Commercial EDA

- Decline of ASIC/ASSP starts
- Rise of FPGA starts
- Decline of ASPs as products became commoditized
- FPGA pressures to lower ASPs
- Challenges of back end
- Complexity of front end
- IP industry – tools are an enabler, not a business in itself
- NOT the responsibility of designers to guarantee a viable ESL market for EDA tool companies
  - But if tools provide value, a viable market will emerge
  - Remember open source and standards! Constrains revenue of proprietary solutions



Conclusions

- ESL has made some significant progress in the last few years
- After fits and starts since at least the mid 1990s, we can see an ESL flow begin to take shape
- But a flow is more than just commercial tools
- Significant work is being done with research tools, models and open source modeling environments
- We urge everyone to
  - Educate themselves on ESL
  - Adopt what is usable now
  - Monitor new developments and adopt when ready

Thank You. Questions?

Contact Information

brian_bailey@acm.org
gmartin@tensilica.com
andy@piziali.dv.org