

ECE382M.20: System-on-Chip (SoC) Design

Lecture 2 – Electronic System-Level (ESL) Design

*with sources from:
Christian Haubelt, Univ. of Erlangen-Nuremberg*

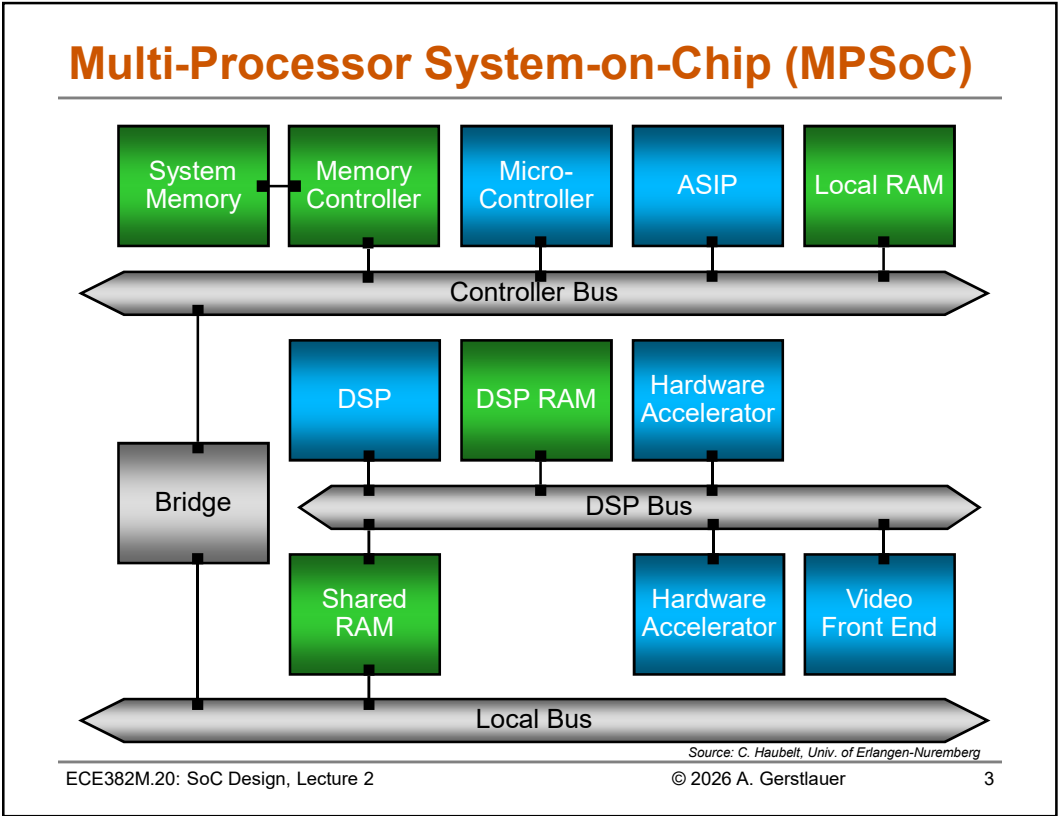
Andreas Gerstlauer
Electrical and Computer Engineering
The University of Texas at Austin
gerstl@ece.utexas.edu



The University of Texas at Austin
Chandra Department of Electrical
and Computer Engineering
Cockrell School of Engineering

Lecture 2: Outline

- Introduction
- Electronic system-level design (ESL/SLD)
 - System-level design flow
 - HW/SW co-design
- System-level design tasks
 - Synthesis
 - Modeling
- Summary and conclusions



SoC Design Challenges

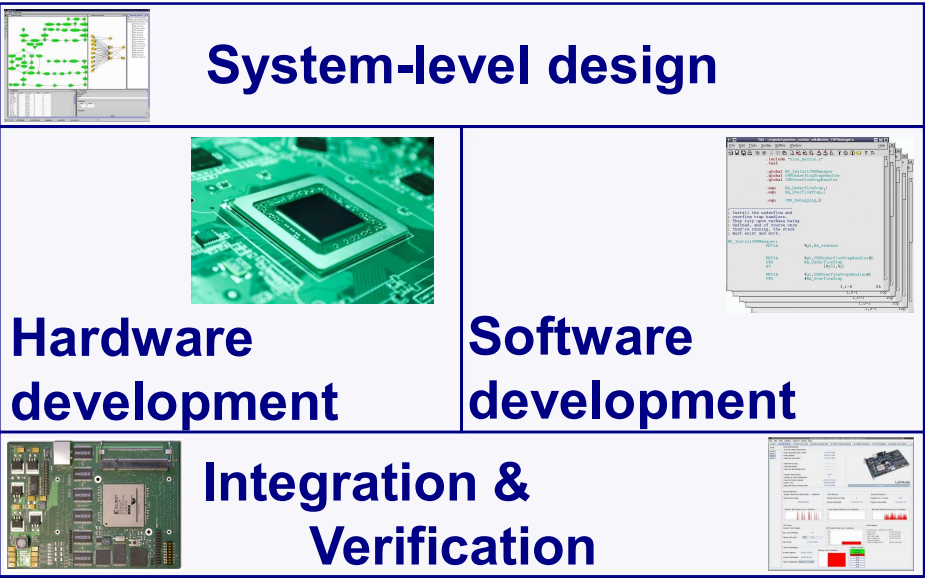
- **Complexity**
 - High degree of parallelism at various levels
- **Heterogeneity**
 - Of components
 - Of tools
- **Semantic gap**
 - Applications
 - Architectures

➤ **Raise abstraction level in design**

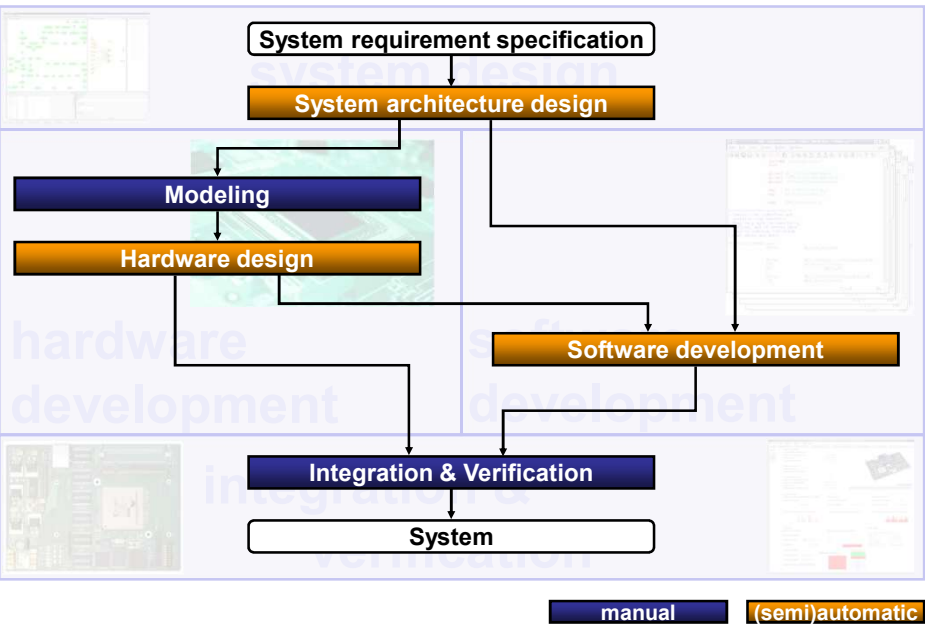
The diagram shows a flow from **Applications** (represented by a yellow cloud) to **Mapping?** (represented by a green hexagon). Below this, a simplified block diagram shows a hierarchy of components (green and blue blocks) connected by buses, representing the hardware architecture.

Source: C. Haubelt, Univ. of Erlangen-Nuremberg
ECE382M.20: SoC Design, Lecture 2 © 2026 A. Gerstlauer 4

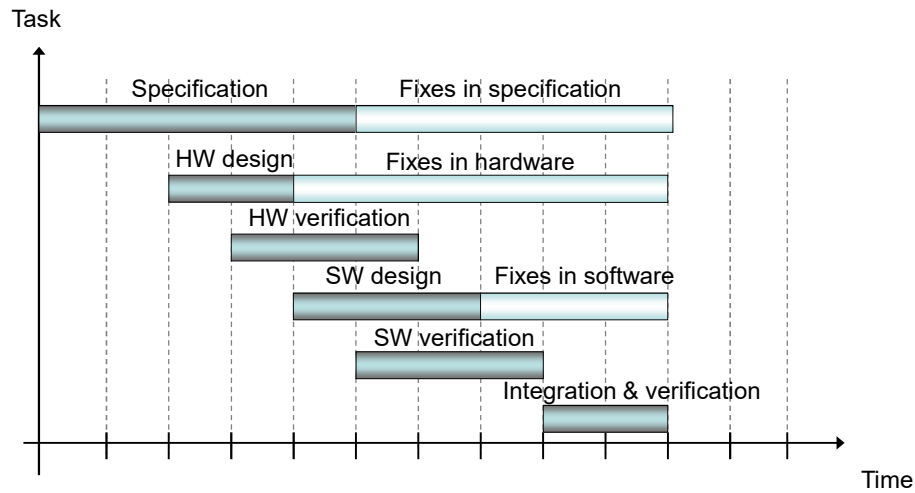
Electronic System-Level (ESL) Design



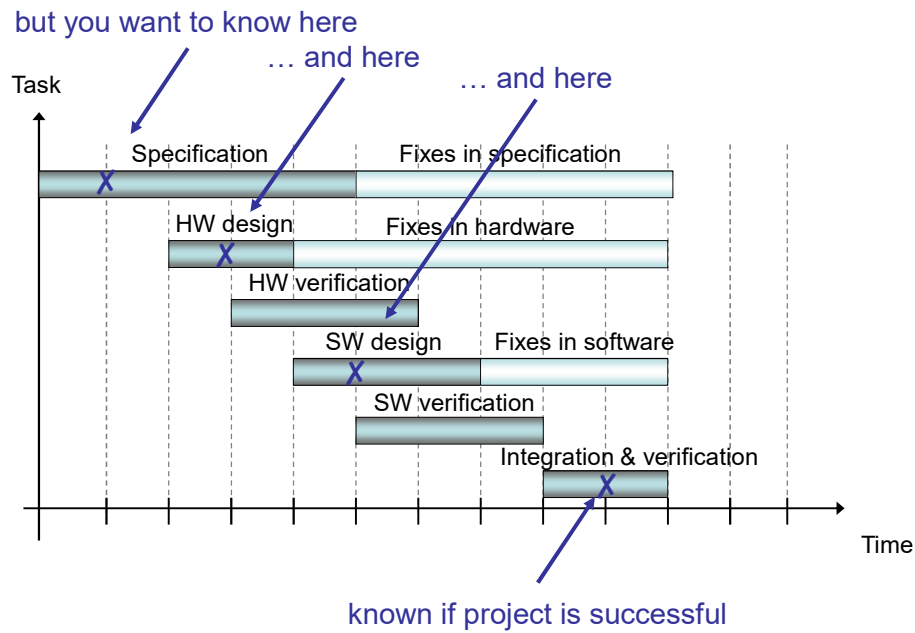
Classical System Design Flow



Hardware-Centric Design Cycle



Hardware-Centric Design Cycle



```

graph TD
    A[System requirement specification] --> B[System-level design]
    B --> C[High-level model]
    C --> D[Hardware design]
    C --> E[Software development]
    D --> F[Integration & Verification]
    E --> F
    F --> G[System implementation]
  
```

Legend:

- Dark blue box: manual
- Orange box: (semi)automatic

9

Task

Specification
(high-level & arch. models)

Fixes in specification

HW design

Fixes in hardware

HW verification

SW design

Fixes in software

SW verification

Integration & verification

Time

Find good design options here -> **high-level models as crucial enablers!**

10

Design Methodologies

• Top down design

- Starts with functional system specification
 - Application behavior
 - Models of Computation (MoC)
- Successive refinement
- Connect the hardware and software design teams earlier in the design cycle.
- Allows hardware and software to be developed concurrently
- Goes through architectural mapping
- The hardware and software parts are either manually coded or obtained by refinement from higher model
- Ends with HW-SW co-verification and System Integration

• Platform based design

- Starts with architecting a processing platform for a given vertical application space
 - Semiconductor, ASSP vendors
- Enables rapid creation and verification of sophisticated SoC designs variants
- PBD uses predictable and pre-verified firm and hard blocks
- PBD reduces overall time-to-market
 - Shorten verification time
- Provides higher productivity through design reuse
- PBD allows derivative designs with added functionality
- Allows the user to focus on the part that differentiate his design

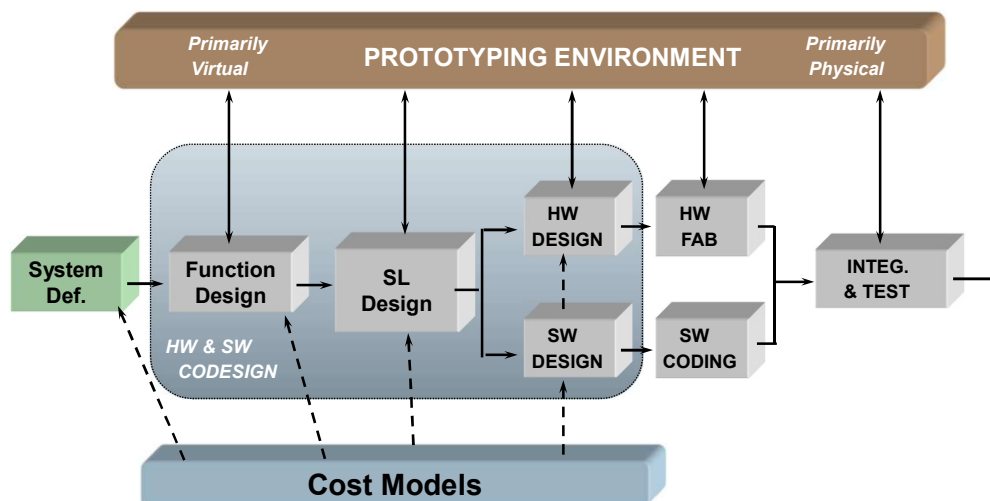
Source: Coware, Inc., 2005

ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

11

Top-Down ESL Design Environment



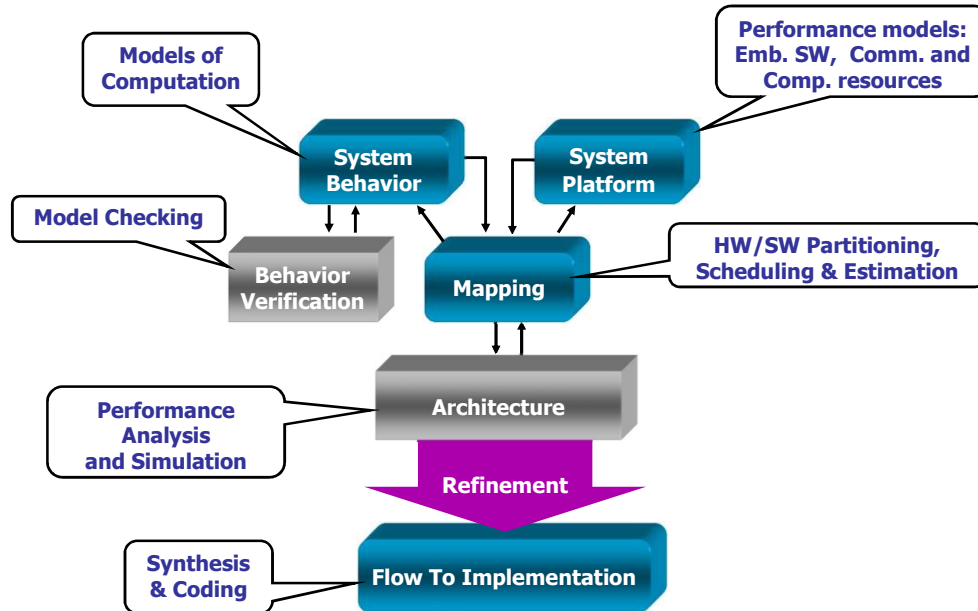
Copyright © 1995-1999 SCRA Used with Permission

ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

12

Platform-Based Design (PBD)



Source: UC Berkeley, EECS249

ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

13

Lecture 2: Outline

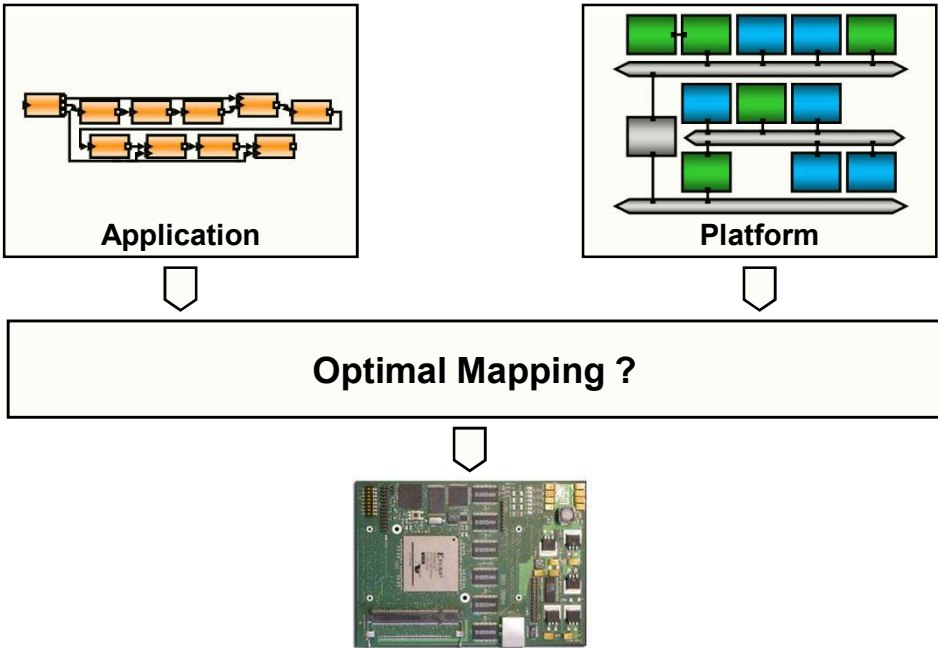
- ✓ Introduction
- ✓ ESL design methodology
- **System-level design tasks**
 - Synthesis
 - Modeling
- Summary and conclusions

ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

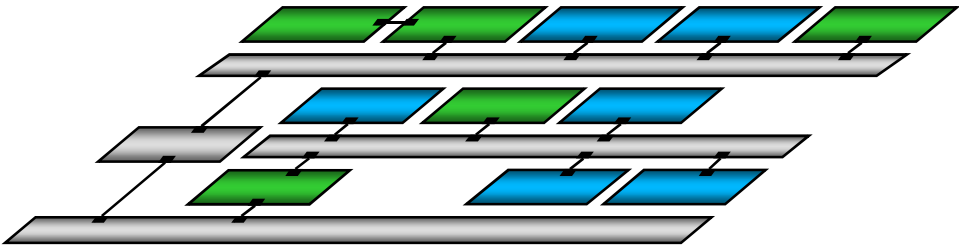
14

Platform-Based System Design Tasks



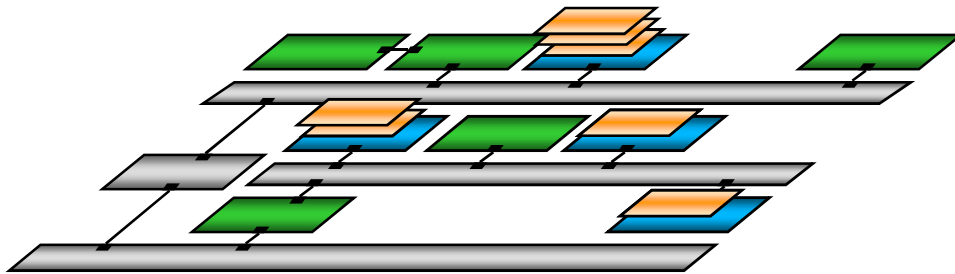
Resource Allocation

- Resource allocation, i.e., select resources from a platform for implementing the application



Process Mapping

- **Process mapping, i.e., partition & bind processes onto allocated computational resources**



ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

17

Communication Mapping

- **Channel mapping, i.e., assign channels to paths over busses and address spaces**



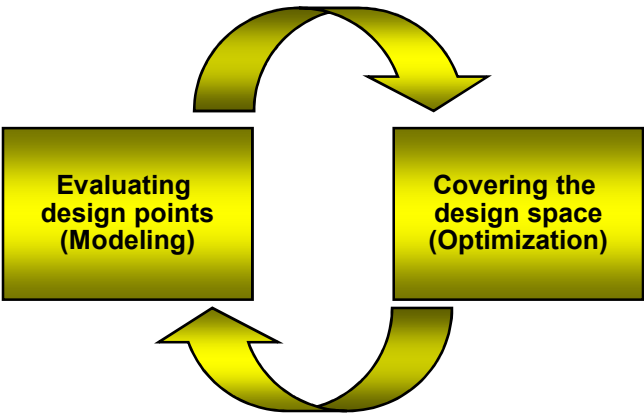
ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

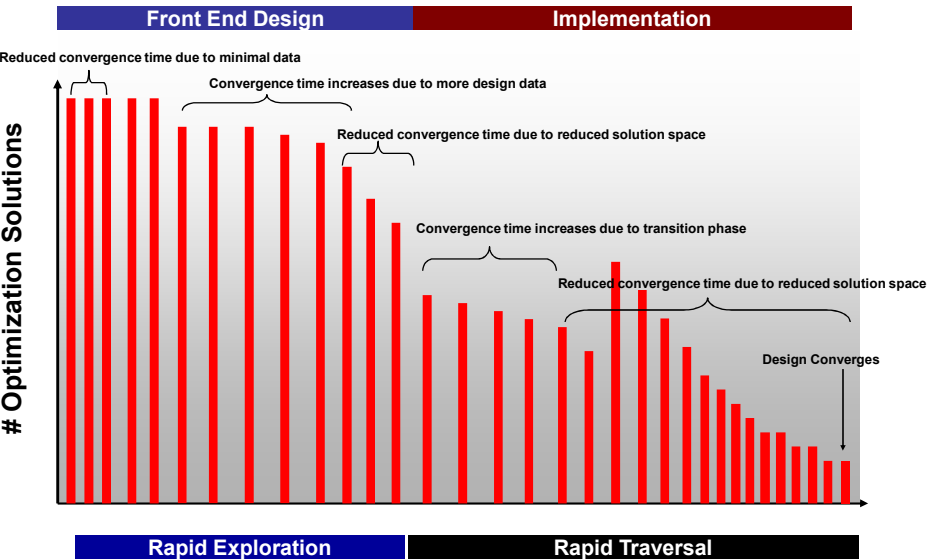
18

Design Space Exploration (DSE)

- **Design is an iterative process**
 - Make design decisions to explore the design space
 - Evaluate the impact of decisions on design metrics



Design Convergence



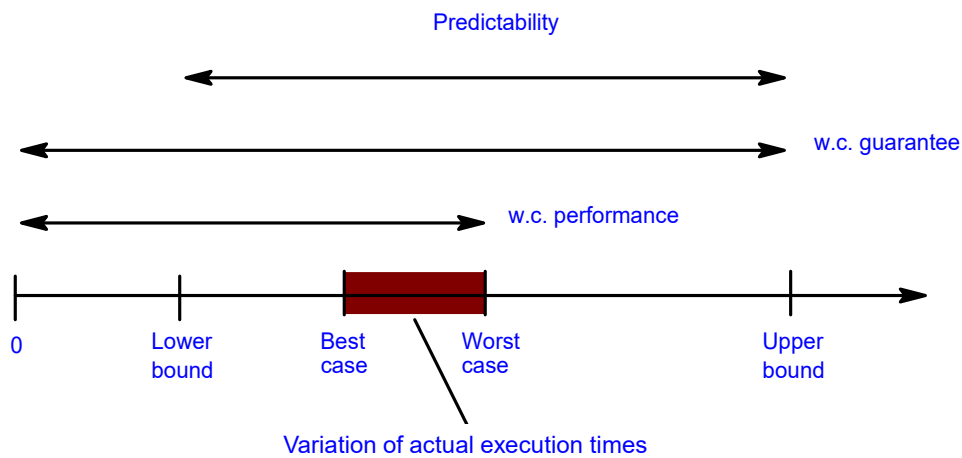
Lecture 2: Outline

- ✓ Introduction
- ✓ ESL design methodology
- **System-level design tasks**
 - ✓ Synthesis
 - Modeling
- Summary and conclusions

System Modeling

- **Design models are the core of any design flow**
 - Representation for evaluation and analysis
 - Specification for further implementation
- **System-level design models**
 - Support HW/SW co-design
 - Support early SoC architecture design
 - Support rapid design space exploration
- **System-level design languages**
 - Capture system-level models
 - Hardware and software

Performance Analysis



➤ Simulation (dynamic) vs. analysis (static)

- Tightness of bounds, under- vs. over-estimation

Performance of a System

• Depends on many factors

- System design (algorithms and data structures)
- Implementation (code)
- Execution platform architecture
- **The workload to which it is subjected**
- The metric used in the evaluation

➤ Interactions between these factors

Design Models

- **Algorithm and application level**
 - Profiling on existing hardware
 - Analytical complexity models
 - Identify bottlenecks, evaluate tradeoffs
- **Component level**
 - Software & hardware partition
 - Simulation-based component models
 - Analytical performance/power/... estimates
- **System level**
 - Simulation-based virtual platform prototyping
 - Physical prototyping

Class Labs & Project Modeling

- **Application profiling (Lab 1)**
 - Profiling of software code on board (**gprof**)
- **Accelerator synthesis (Lab 2)**
 - Accelerator performance/area estimation
 - Accelerator simulation model
- **Virtual platform prototyping (Lab 3)**
 - SystemC-based SoC simulation model
 - HW/SW co-simulation
- **Physical prototyping (Project)**
 - FPGA-based SoC prototyping

Algorithm-Level Analysis

- **Theoretical complexity scaling analysis**
 - $O()$ notation (“order-of”)
 - Example: Sorting
 - “Bubble” sort
 - Merge sort
 - Example: Fourier transform
 - Discrete Fourier transform
 - Fast Fourier transform
- **Code analysis**
 - Closed-form analytical complexity model
 - Example: BCH* encoding
 - Find the number of XOR and AND operations performed in the loop as a function of k
 - Assume *length* is 1024, and in any bit position, 0 and 1 are equally likely

* A BCH code is a multilevel, cyclic, error-correcting, variable-length digital code used to correct multiple random error patterns. BCH codes may also be used with multilevel phase-shift keying whenever the number of levels is a prime number or a power of a prime number.

BCH Code

```
encode_bch()
{
    register int    i, j;
    register int    feedback;
    for (i = 0; i < length - k; i++)
        bb[i] = 0;
    for (i = k - 1; i >= 0; i--) {
        feedback = data[i] ^ bb[length - k - 1];
        if (feedback != 0) {
            for (j = length - k - 1; j > 0; j--)
                if (g[j] != 0)
                    bb[j] = bb[j - 1] ^ feedback;
            else
                bb[j] = bb[j - 1];
            bb[0] = g[0] && feedback;
        } else {
            for (j = length - k - 1; j > 0; j--)
                bb[j] = bb[j - 1];
            bb[0] = 0;
        }
    }
}
```

Application-Level Profiling

- **Execute code on physical or simulated machine**
 - (Cross-)Compile down to binary
 - Run on real processor or component-level simulator
 - Benchmarks and input data test vectors
- **Instrument code and collect metrics at runtime**
 - Include effect of processor instruction set and architecture
 - For the given runtime platform (not necessarily the intended target)
 - Many profiling tools for data gathering and analysis
 - `gprof` measures where program spends its time and which functions call other functions while it was executing
 - Various interfaces, levels of automation, and approaches to information presentation
 - Flat profile: total amount of time program spends executing each function
 - Call graph: how much time was spent in each function and its children
 - A lot of work in the high performance computing community
 - Effect of instrumentation on measured results?

ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

29

Instrumentation Techniques

- **Program instrumentation techniques**
 - Manual: Programmer inserted directives
 - Automatic: No direct user involvement
 - Sampling-based [`gprof`], binary rewriting [`PIN`], dynamic Instrumentation
- **Hardware instrumentation techniques**
 - Hardware timers
 - Hardware counters
 - Cycles, cache misses, pipeline stalls, memory reads/writes, memory stalls, etc.
 - Available mostly through special registers or memory mapped location
 - Hardware assisted signal tracing
- **Operating system instrumentation techniques**
 - Virtual memory, file system, file cache, etc. statistics
 - Access via APIs
- **Network instrumentation techniques**
 - Passive, e.g. RMON protocol packet header fields for monitoring
 - Active, e.g. ping, NWS in grid style computing.

ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

30

Component-Level Estimation

- **Static analysis and estimation of performance**
 - Ability to provide hard guarantees
 - Reliability, safety, etc.
- **Worst-Case Execution Time (WCET) estimation of software**
 - Tightness of bounds?
- **Analysis of modern processors is very difficult**
 - Dynamic effects
 - Pipelines, caches, branch predictors
 - Hazards lead to timing accidents & penalties
 - Structural (resource being used by another)
 - Data (dependence for data calculations)
 - Control (calculating next address – branches, interrupts)
- **ML-based predictive modeling approaches**

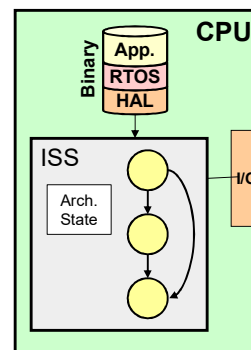
ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

31

Component-Level Simulation

- **Cycle-accurate**
 - Describe micro-architecture/RTL in C
- **Instruction-set simulation (ISS)**
 - Cycle-accurate/-approximate
 - Functionality/behavior only
- **Compiled ISS**
 - Binary translation
 - Offline vs. just-in-time (self-modifying code)
 - Functional only (no/rough timing, e.g. CPI=1)
- **Source-level/host-compiled simulation**
 - C model describing functionality/behavior
 - Back-annotate with timing and other metrics



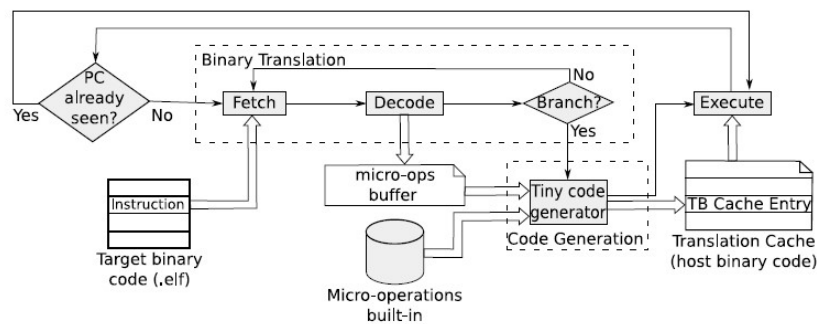
ECE382M.20: SoC Design, Lecture 2

© 2026 A. Gerstlauer

32

Quick EMUlator (QEMU)

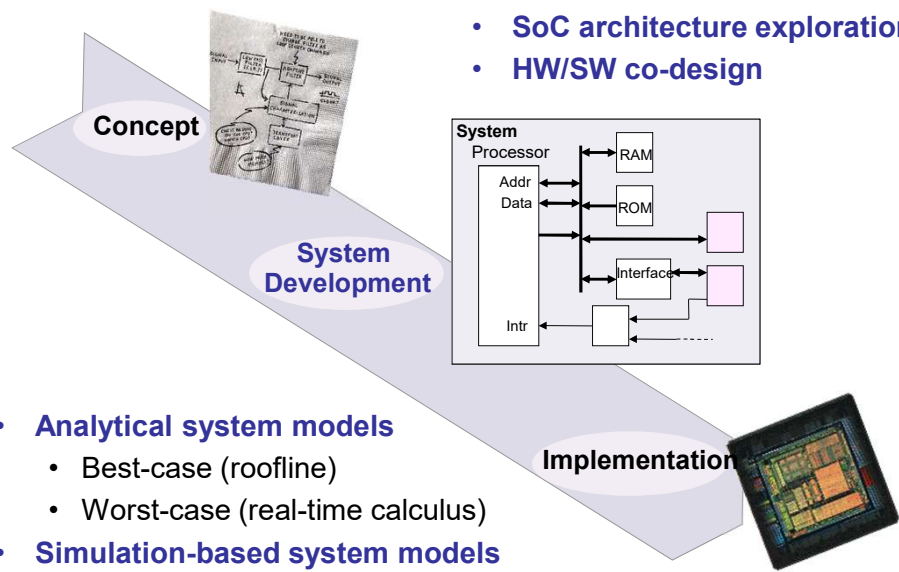
- **Open-source, binary-translating ISS [Bellard'05]**
 - Emulates a variety of architectures (x86, ARM, PowerPC)
 - Stand-alone or full-system model (peripherals to boot OS)



Source: M. Gilgor, N. Fournel, F. Pétrot, "Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation," CODES+ISSS'09

System-Level Modeling

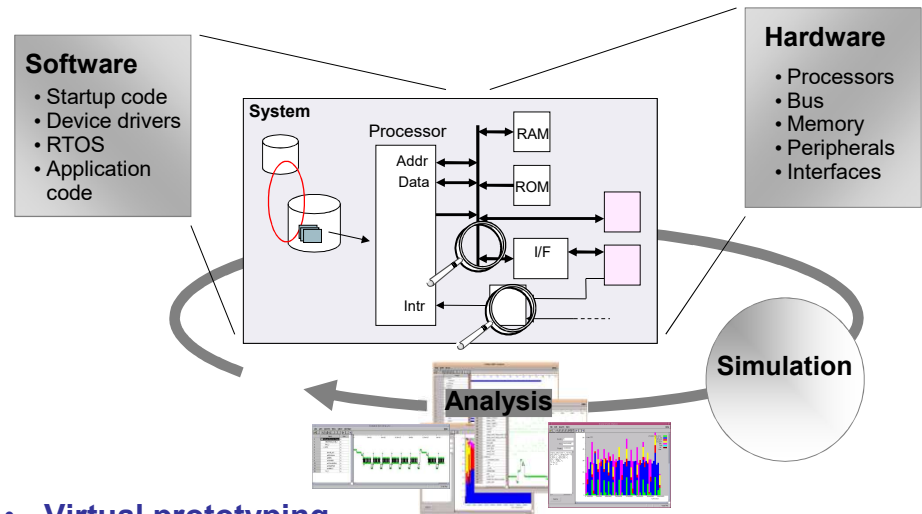
- **SoC architecture exploration**
- **HW/SW co-design**



- **Analytical system models**
 - Best-case (roofline)
 - Worst-case (real-time calculus)
- **Simulation-based system models**
 - Virtual platform prototyping

Source: CoWare, Inc.

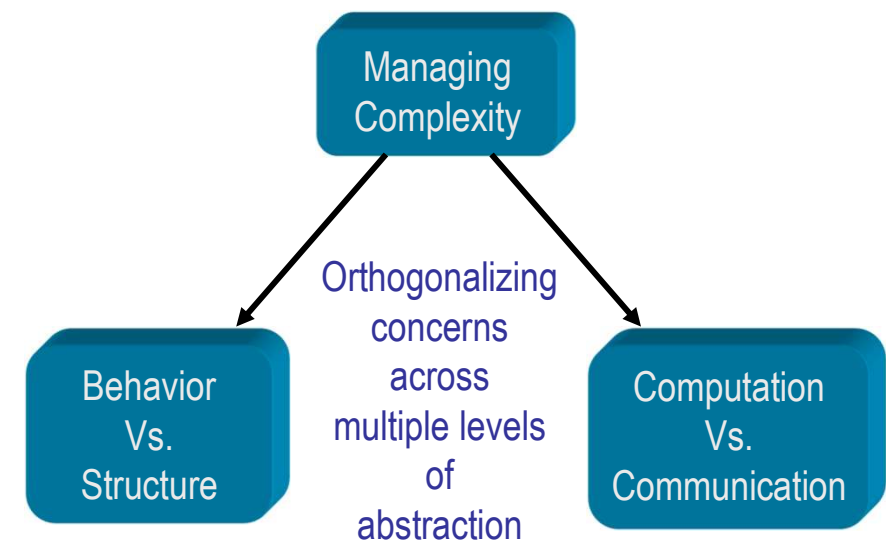
Virtual Platforms



- **Virtual prototyping**
 - Full-system co-simulation of different components
 - Observability of software, bus, memory, hardware behavior

Source: CoWare, Inc.

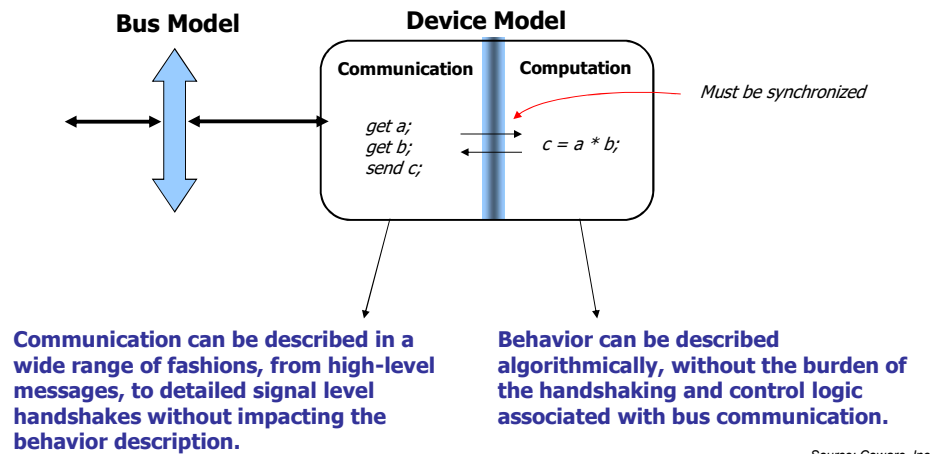
System-Level Modeling Concerns



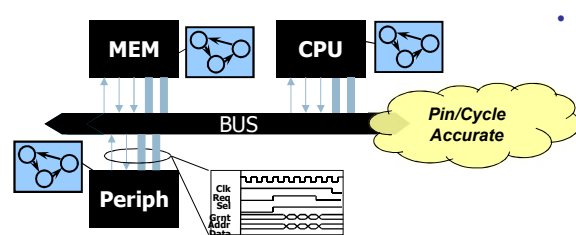
Source: UC Berkeley, EECS249

Computation vs. Communication

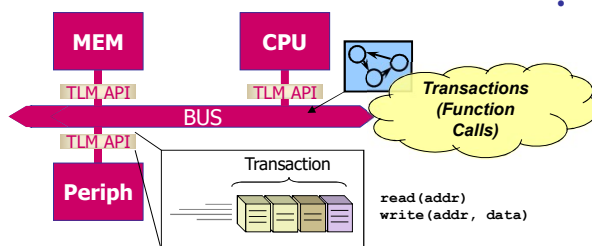
- **Separation of concerns**
 - Flexibility in modeling, IP reuse
 - Design computation & communication separately



Communication Models

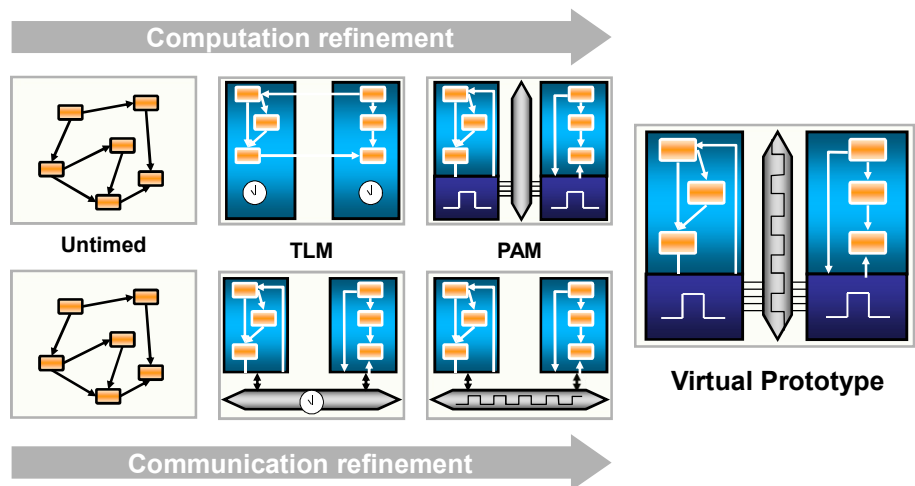


- **Pin-Accurate Model (PAM)**
 - Redundant RTL complexity results in slow simulation
 - Each device interface must implement the bus protocol
 - Each device on the bus has a pin-accurate interface
 - Detailed signal handshaking



- **Transaction-Level Model (TLM)**
 - Each device communicates via function-level API
 - Protocol is modeled as a single bus model instead of in each device
 - Abstracted communication, less code, no wires, fewer events
 - 100x-10,000x faster than PAM

Virtual Platform Prototyping



Abstraction Levels

