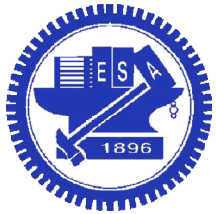
HW/SW Co-Simulation



National Chiao Tung University
Chun-Jen Tsai
5/23/2011

Note: Part of materials of these slides are from the book: J. R. Andrews, *Co-Verification of Hardware and Software for ARM SoC Design*, Elsevier, 2005

Reasons for Co-Simulation

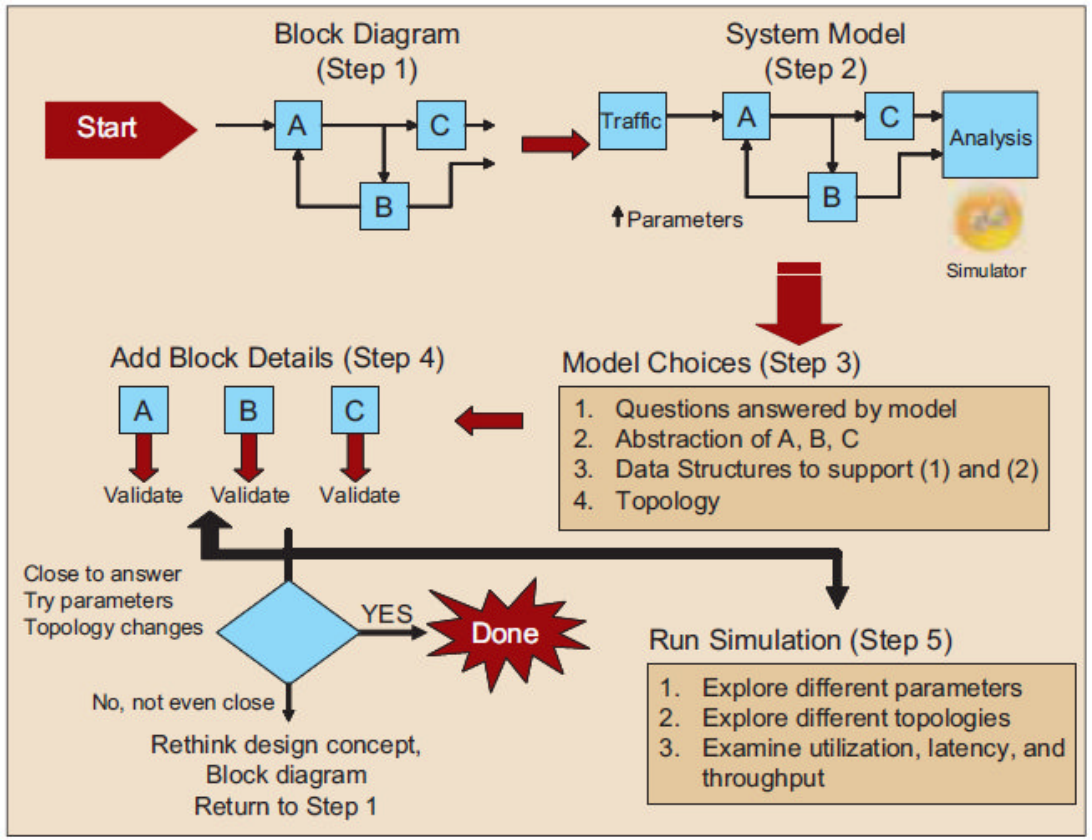
- There are many reasons for HW/SW co-simulation:
  - Full system integration
    - HW and SW engineers are usually different groups of people; mutual understanding is always an issue
  - Design exploration
    - Trial-and-error is never a good way to design systems, but we have to face the fact that, unfortunately, most engineers are trained to do trial-and-error development
  - Enable parallel development of HW and SW components
    - Nowadays, a development cycle is only 3 ~ 6 months
  - Enable replacing of a component late in the design stage
    - The market of IPs and target technologies is changing every minutes; we want to adapt to new environment fast

HW/SW System Integration

- Most HW/SW co-designed system are developed using the “plug-and-debug” model:
  - Once HW/SW partition and communication models are decided, HW team and SW team go on and develop systems independently
  - When both HW and SW are done, the system is integrated and debugged by lots of logic analyzer probing, “printf” analysis, educated guesses, etc.

Architecture Design Exploration

- Sometimes, we need fast prototyping for better design†:



† D. Shankar, "Accelerating Architecture Exploration for FPGA Selection and System Design," *Xcell Journal*, Issue 58, 2006.

System Performance Characteristics

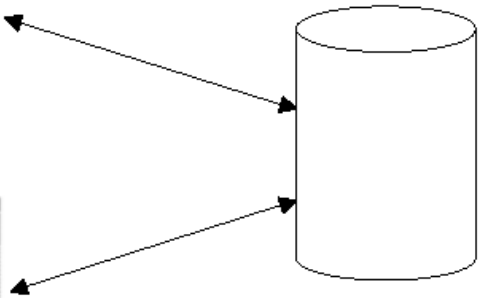
- We want to know some performance characteristics as early as possible in our design process
  - Processor performance characteristics
    - Instruction distribution
    - Cache behavior
    - Bus (memory) access behavior
  - Bus traffic behavior, bus transaction turn-around time
    - Communication distribution among IPs and memory banks
  - Memory performance characteristics
    - Memory transactions causing bottlenecks?

Co-Verification Model

- How do we verify SW and HW side-by-side?

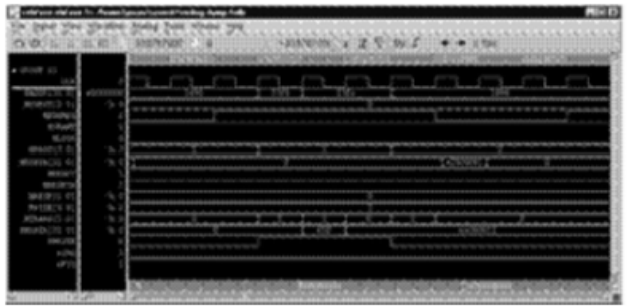
Software debugger

Correlates Software Source Code
to
Simulation Timestamp



transaction store

Logic simulator

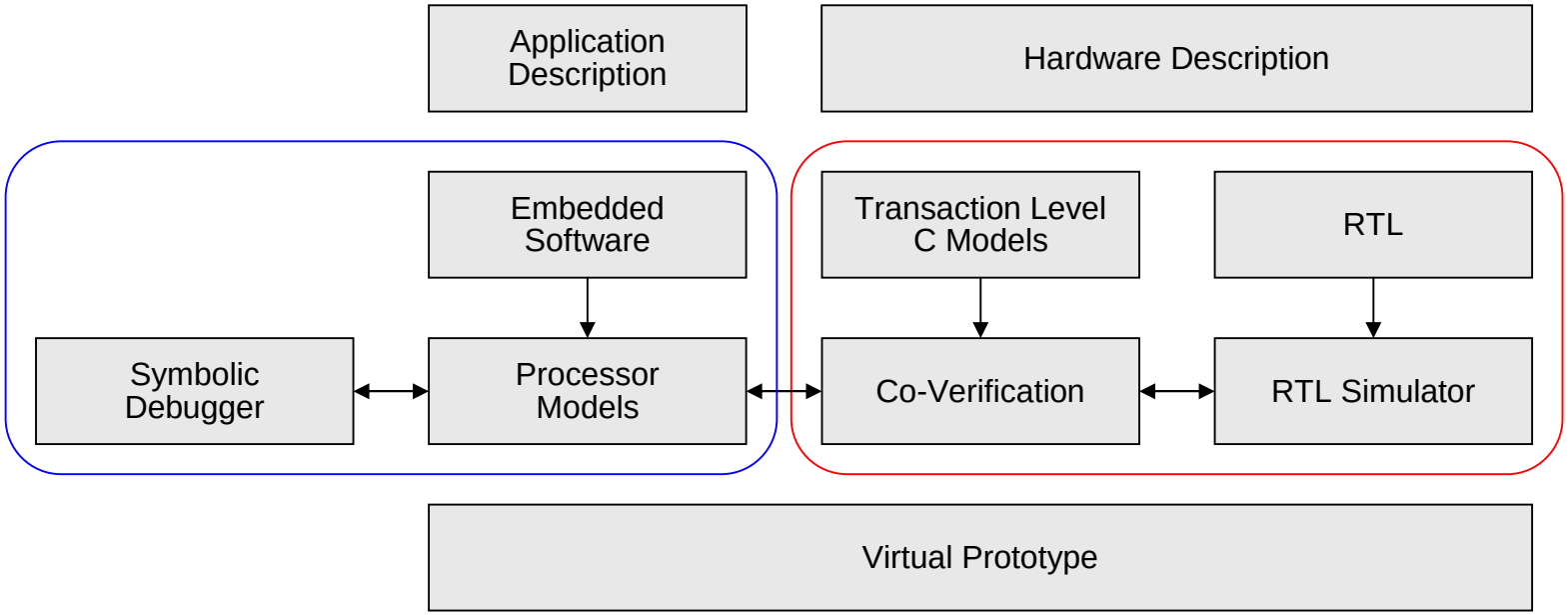


Benefits of Co-Simulation

- Benefits of Co-Simulation
  - Early access to the hardware design for software engineers
  - Provide more “realistic” stimulus for hardware simulation (as compared to “unrealistic” testbench waveforms)
  - Increase controllability and visibility
  - Increase understanding between hardware and software designers
- The first commercial co-simulation tools appears in 1995-1996 (Seamless)
  - Mentor Graphics Seamless & Seamless FPGA
  - CoWare ConvergenSC + Cadence Incisive
  - Synopsys CoCentric
  - ARM SoC Designer

Seamless Co-Design

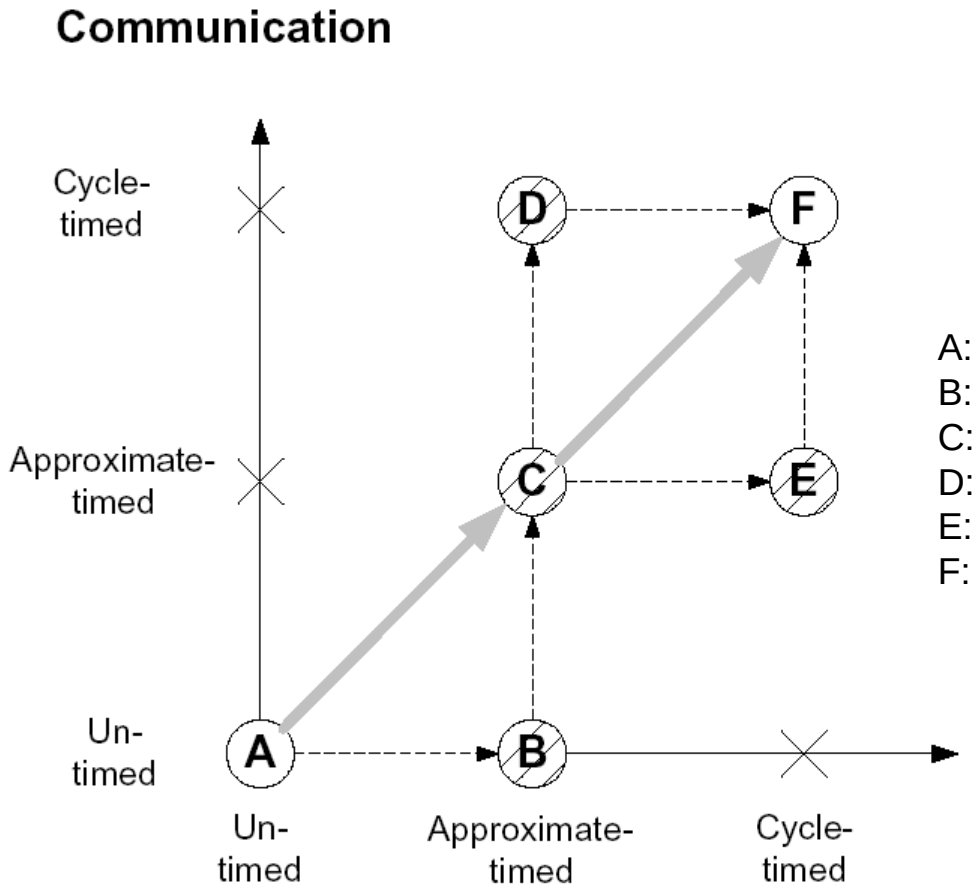
- Mentor Graphics Seamless HW/SW codesign flow:



Co-Verification and Co-Simulation

- Co-verification can be done on a simulated platform; or an emulated platform
  - For a simulated platform, simulators of software and hardware are required
  - For an emulated platform, an FPGA-based development board is often used

Resolutions of Co-Simulation†

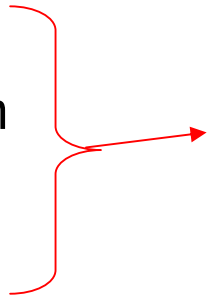


A: Specification model
B: Component-assembly model
C: Bus-arbitration model
D: Bus-functional model
E: Cycle-accurate computation model
F: Implementation model

† L. Cai and D. Gajski, "Transaction Level Modeling: An Overview," *Proc. of CODES+ISSS*, Newport Beach, 2003.

Different Levels of Abstraction

- Functional level
  - HW/SW independent
  - Timing is not crucial
- Transaction level
  - SW-like viewpoint of system
  - Rough timing
  - Defines both “data token” and “data flow”
- Pin level
  - HW-like viewpoint of system
  - Precise timing
  - Define RTL behavior

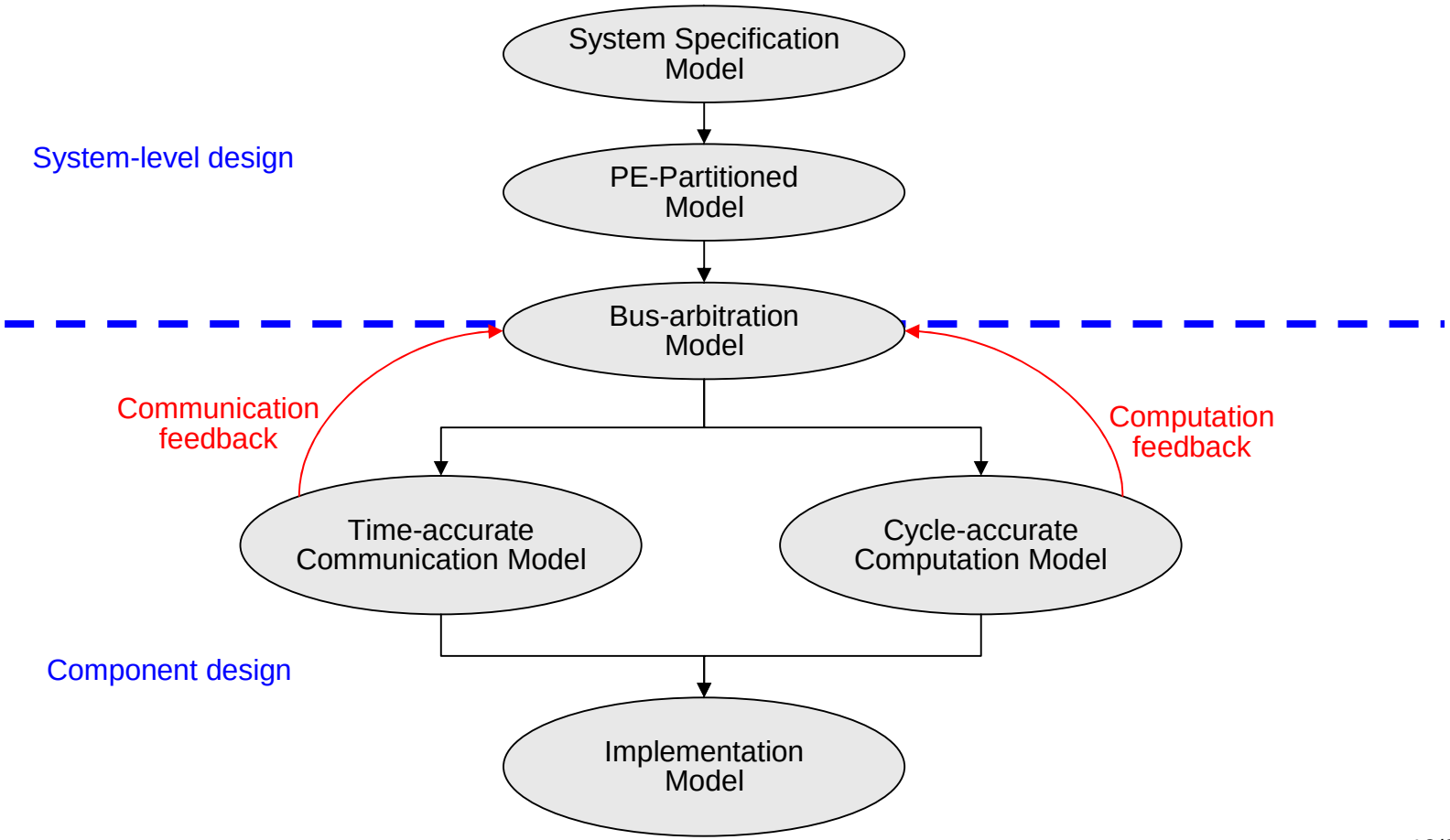


For example, the co-simulation environment we use for LEON has cycle- & pin-level accuracy

Transaction-Level Modeling

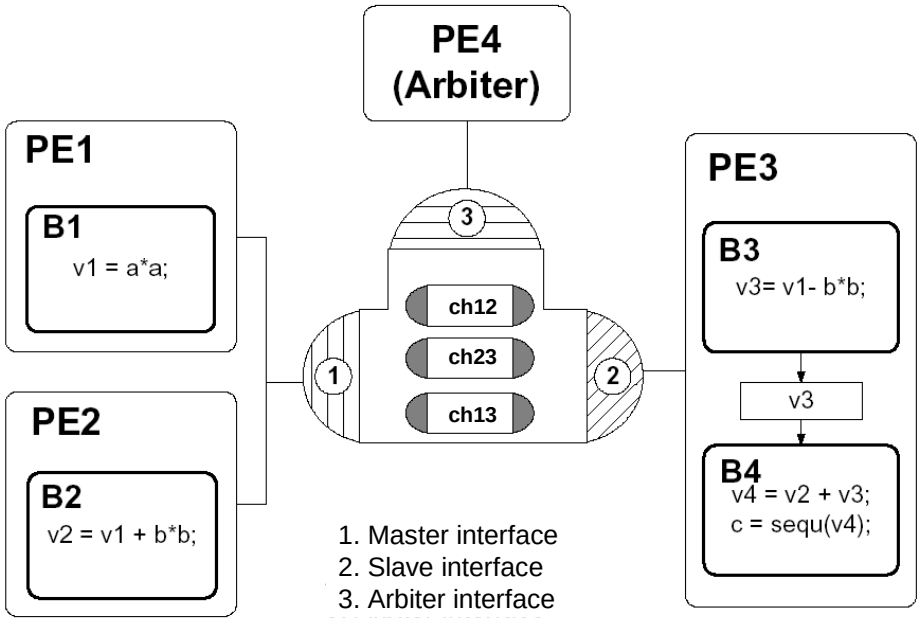
- In a transaction-level model (TLM), the details of communication among PEs are hidden in “channel models”
  - A **module** is a structural entity with computational processes and transaction processes
  - **Transaction** requests take place by calling **interface** functions of the **channel** models (each channel can have multiple interfaces) that belong to a **port** of a module
  - TLM speeds up simulation and allow implementation freedom through higher level of abstractions

TLM Model Types



Bus Arbitration Model

- The channels among PEs implement data transfer through message passing
- Bus protocols can be simplified as blocking and non-blocking I/O. No cycle-accurate and pin-accurate protocol details are specified.



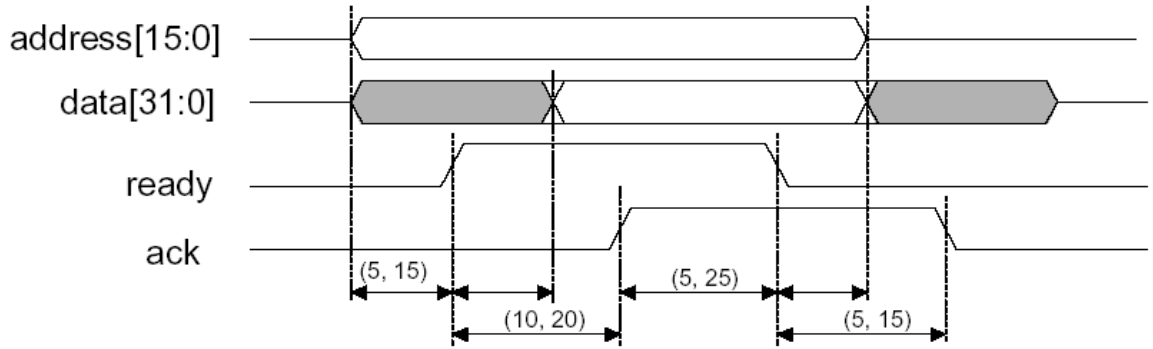
Bus Functional Model (1/2)

- A bus functional model contains time/cycle-accurate communication
- Two types of bus functional model are specified: time-accurate model and cycle-accurate model
  - Time-accurate model specifies the time constraint of communication, which is determined by the time diagram of component's protocol
  - Cycle-accurate model can specify the time in terms of the bus master's clock cycles
- The task of refining a time-accurate model to a cycle-accurate model is called protocol refinement.

Bus Functional Model (2/2)

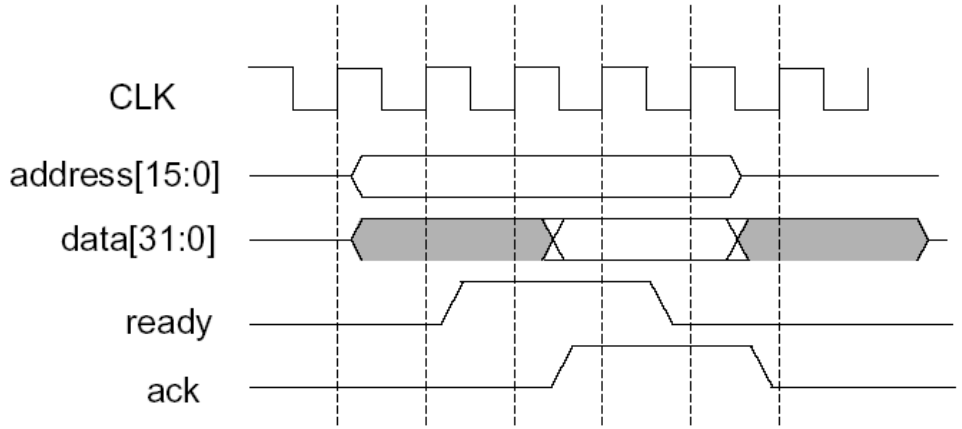
- In bus functional models, the message-passing channels are replaced by protocol channels
  - A protocol channel is time/cycle-accurate and pin-accurate
  - Inside a protocol channel, wires of the bus are represented by instantiating corresponding variables/signals
- Data is transferred following the time/cycle accurate protocol sequence.
- Bus functional model of a protocol channel provides interface functions for all abstract bus transactions

Time/Cycle-Accu. Timing Diagrams



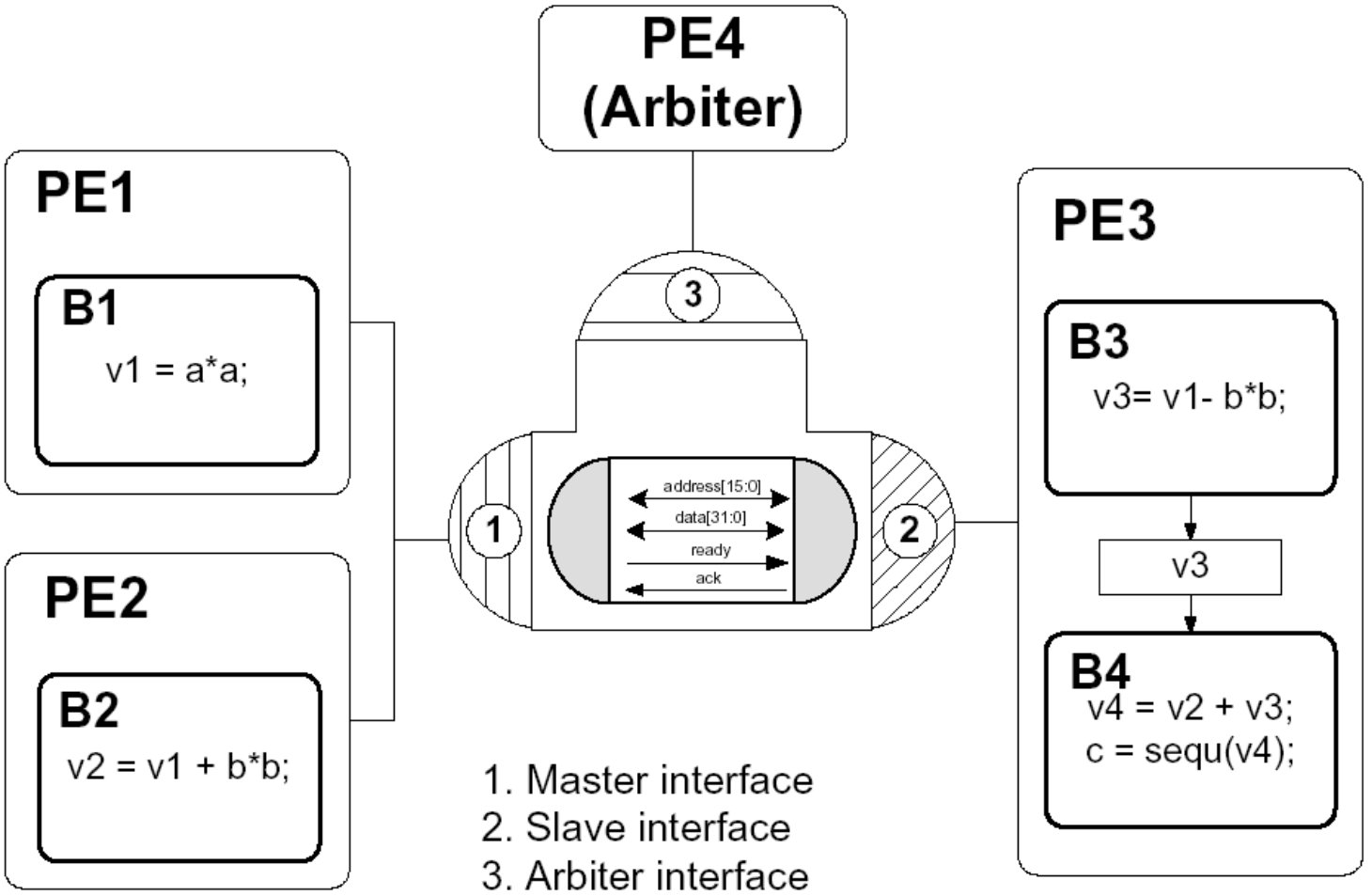
Lowerbound = 5 + 10 + 5 + 5 = 25
Upperbound = 15 + 20 +25 + 15 =75

(a)Time Diagram

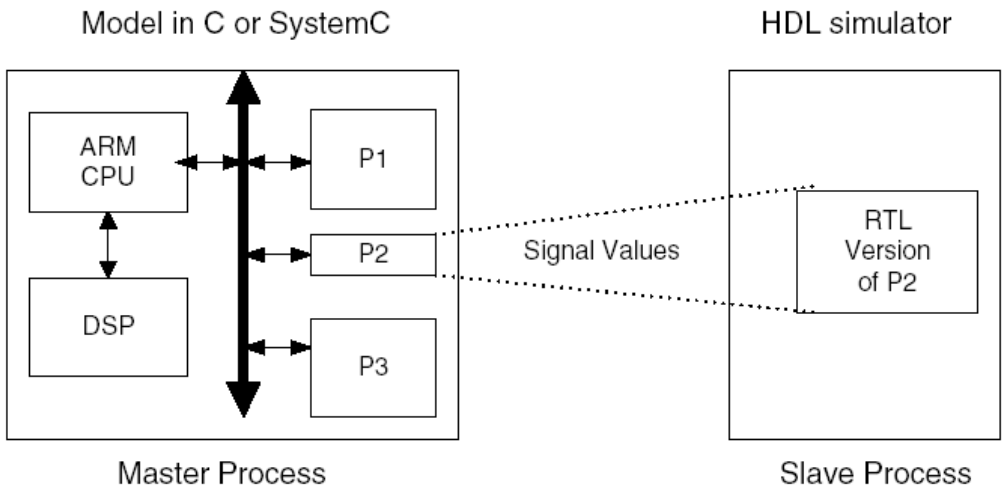


(b)Cycle accurate time diagram

Bus Functional Model Example

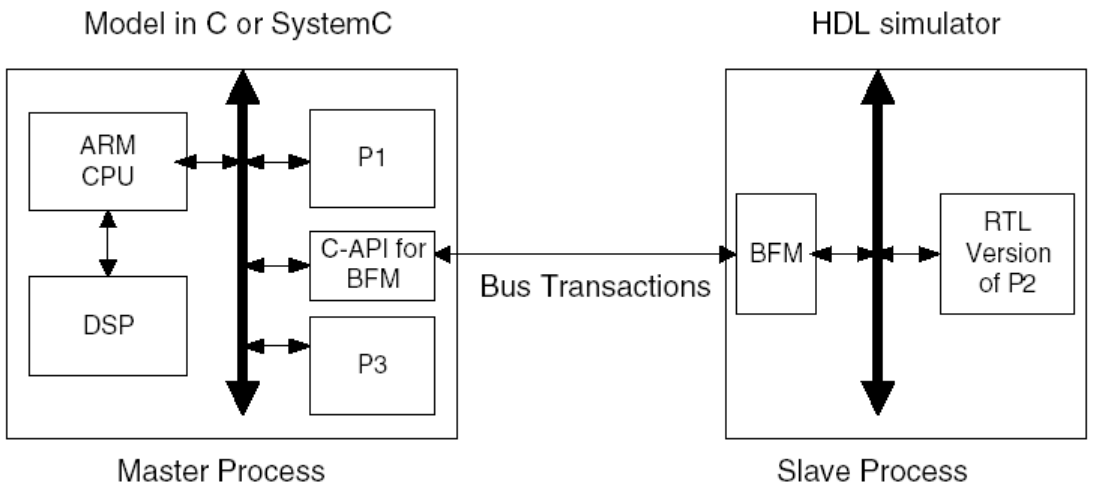


Pin- vs. Transaction-level Co-Simulation



Pin-level Co-simulation

Transaction-level Co-simulation



Comments on Model Languages (1/2)

- What languages should be used as the TLM modeling language?
- Hardware people doesn't like C/C++ because they thought:
  - Concurrency support is missing (HW is inherently parallel)
  - No notion of time (clock, delays)
  - Communication model (function calls & parameters) is very different from actual HW model (pins & signals)
  - Weak/complex reactivity to events
  - Some data types missing (logic values, bit vectors, fixed point)

Comments on Model Languages (2/2)

- Some people propose to extend C++ to include the following features
  - Concurrency support: modules
  - Notion of time: clocks, custom wait() calls
  - Support new communication model: signals, protocols, handshakes
  - Support for events, sensitivity list, watching() construct
  - Support new data types: logic values, bit vectors, fixed point
- The result is a modeling “language” called SystemC
  - From Computer Scientist’s point of view, SystemC is just plain C++, there is no new language created here!

Co-Simulation Techniques

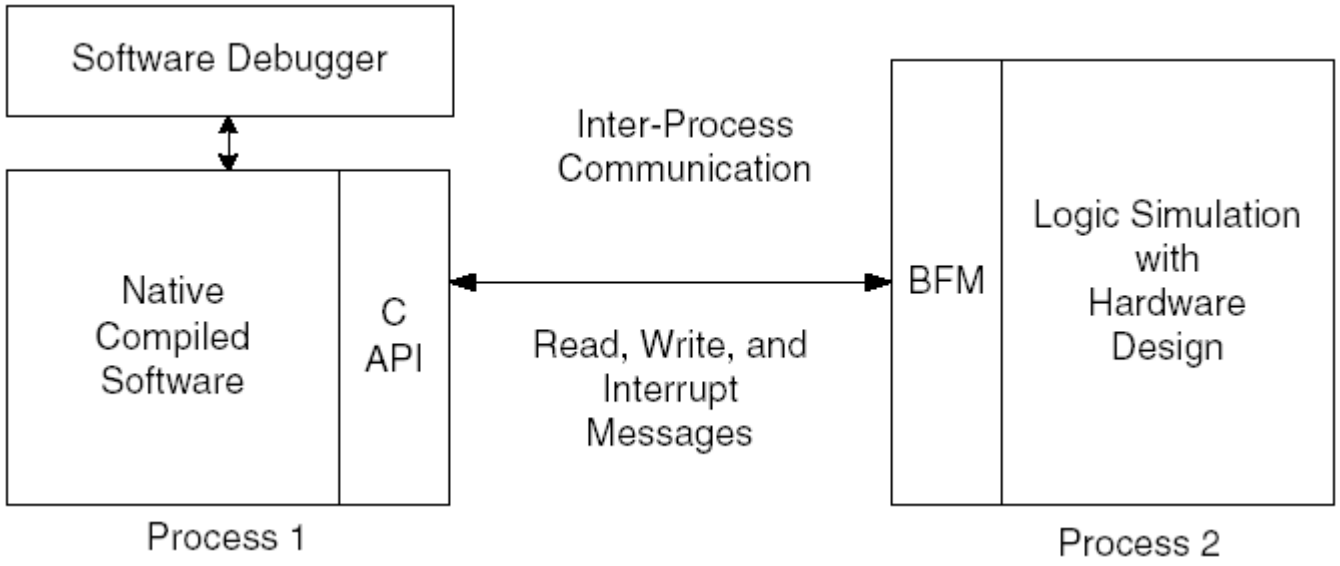
- There are different approaches to co-simulation:
  - Host-code with logic co-simulation
  - Joint instruction set and logic co-simulation
  - Purely C/C++ based co-simulation
  - Purely logic-based co-simulation

Host-Code with Logic Simulation (1/2)

- Use native software inserted with special function calls to trigger “events” to the logic simulator
- On the software side, this approach do not need full ISS, just a communication model simulator (or bus functional model)
- Handle only transaction-based simulation between software and hardware
- An OS simulator can be integrated into the software

Host-Code with Logic Simulation (2/2)

- Logic-related memory accesses are translated into bus transactions via the Bus Functional Model (BFM)

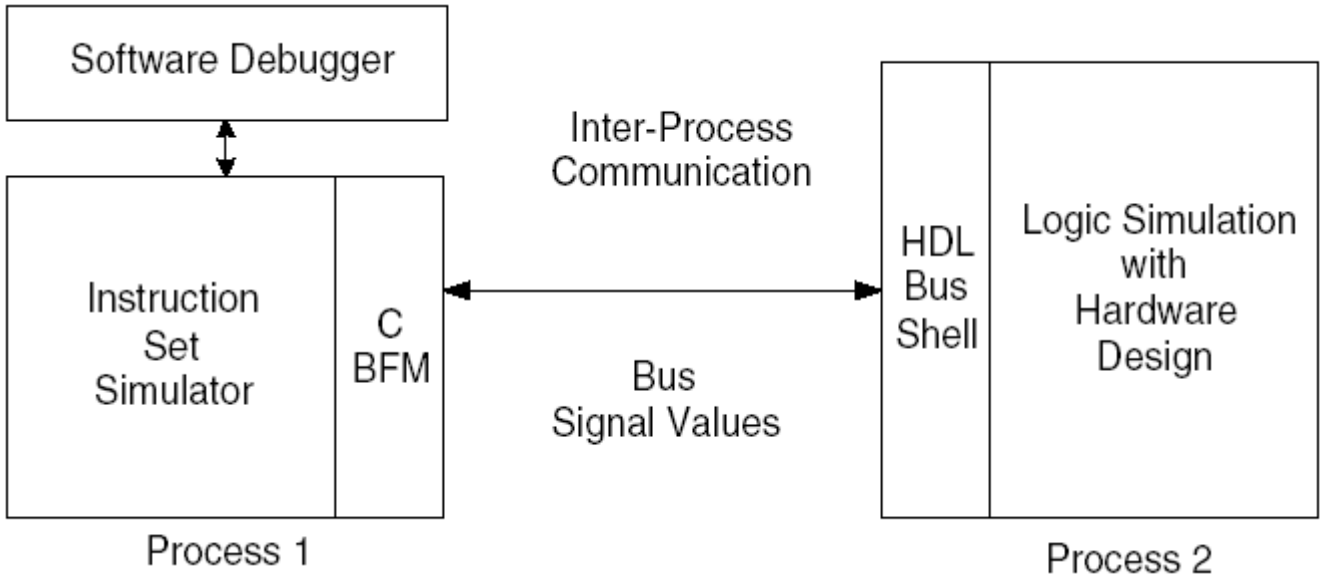


Joint ISS and Logic Simulation (1/2)

- Combine an Instruction Set Simulator (ISS) and a logic simulator to create a virtual platform
- Both software program and hardware logic can be executed in a simulated manner
- A big challenge is to get the OS running on this virtual platform
- Can handle transaction-based as well as cycle-based simulation

Joint ISS and Logic Simulation (2/2)

- A cycled-based co-verification can be done by exchange pin values between ISS and LS on every bus cycle



Pure C/C++ Co-Simulation (1/2)

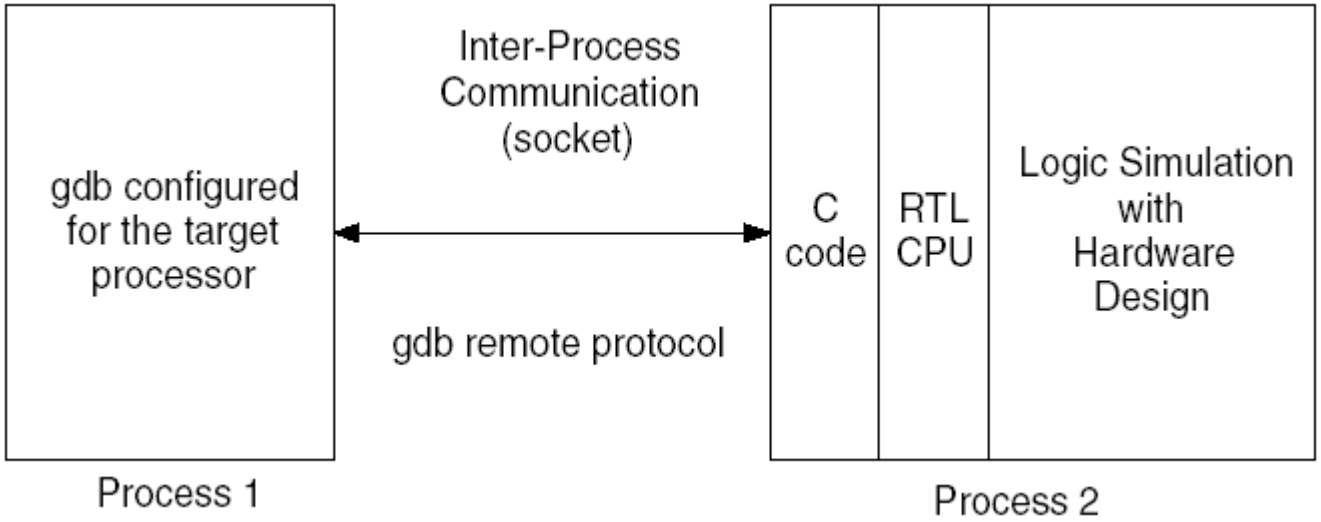
- Since logic simulators are expensive, it may not be cost-effective to include logic simulators in the co-verification flow
- Alternatives:
  - #1: Design cycle-based or transaction-base SystemC models for all the hardware logics
  - #2: Use a HDL-to-SystemC translator to convert all the RTL model in HDL to RTL model in SystemC
- In either cases, the co-verification can be done without logic simulators

Pure C/C++ Co-Simulation (2/2)

- Co-simulation based on C models can be fasters since the RTL model of any proven logics can be replaced by a functional model
- Big questions on C-only co-simulation:
  - Who is going to create all the models?
  - Who is going to maintain the consistency between C models and HDL models

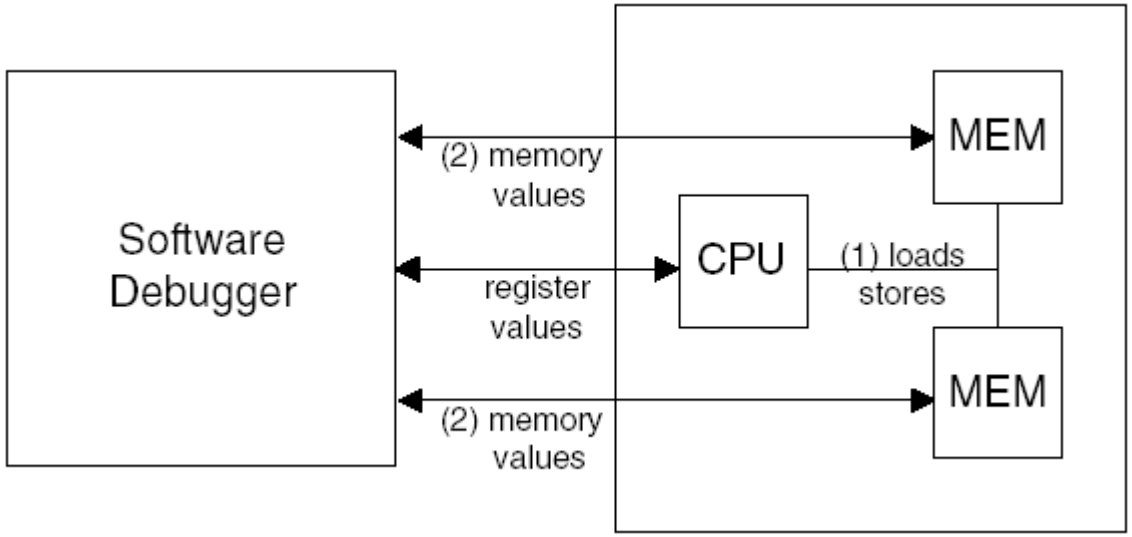
Pure Logic Simulation

- Today, the RTL models for many popular processor cores are available; does it make sense to replace ISS with the RTL model running on a logic simulator?



Software Memory Access Issues

- There are two types of memory accesses on the software side in a simulated environment
  - Memory accesses from the debugger are intrusive and should not generates simulated transactions to the simulated logics



Co-Verification Metrics

- Different projects have different requirements for co-verification environment
  - Performance (simulation speed)
  - Model accuracy
  - Synchronization accuracy (between HW and SW)
  - Type of software verification (e.g. multimedia vs. drivers)
  - Degree of hardware visibility
- Conclusion:
  There is no single best way to do co-verification!