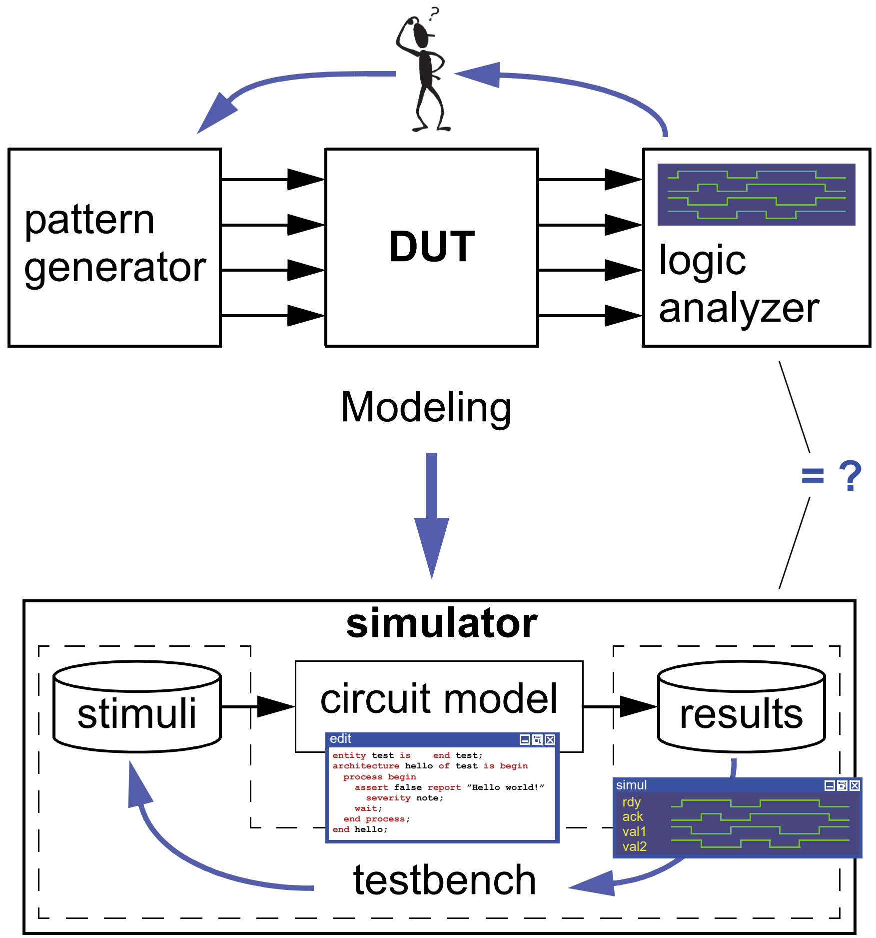
Simulation environment

- **Simulation = modeling + analysis**
- **Environment**
  - **design under test (DUT)**
    - **different abstraction levels**
  - **stimuli generator**
    - **different input-data sequences**
  - **results analyzer**
    - **is DUT responding correctly?**
- **Different combinations exist...**



Use of HDL –> Simulation

- **Simulation = modeling + analysis**
  - **Logic / register-transfer / functional (behavioral) / system level simulation**

concurrent / parallel modules

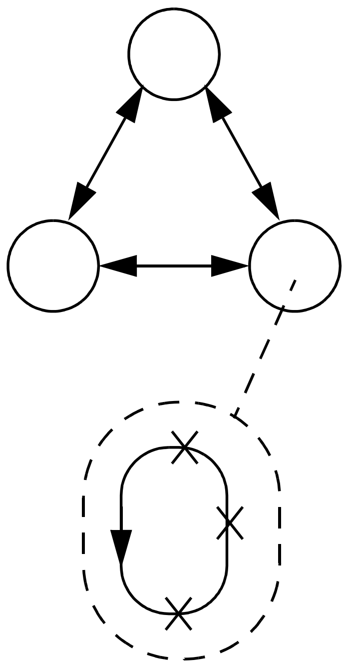
connected via signal / channels

sequential vs. concurrent execution?

execution order?!

current / new values to avoid non-determinism

event queue history+future



module / unit / process

continuous execution is slow

only when needed?

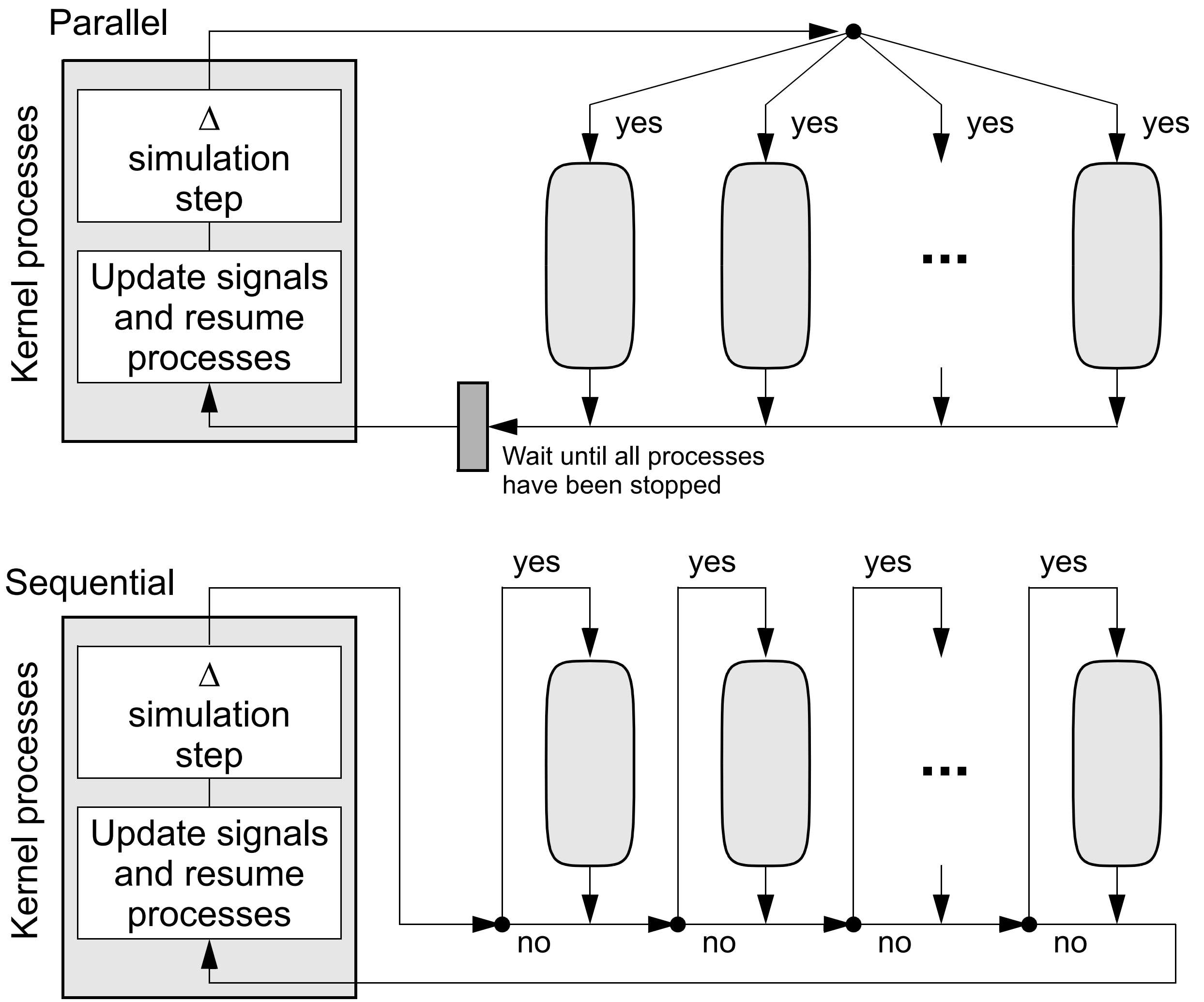
time / event triggered

different simulation engines

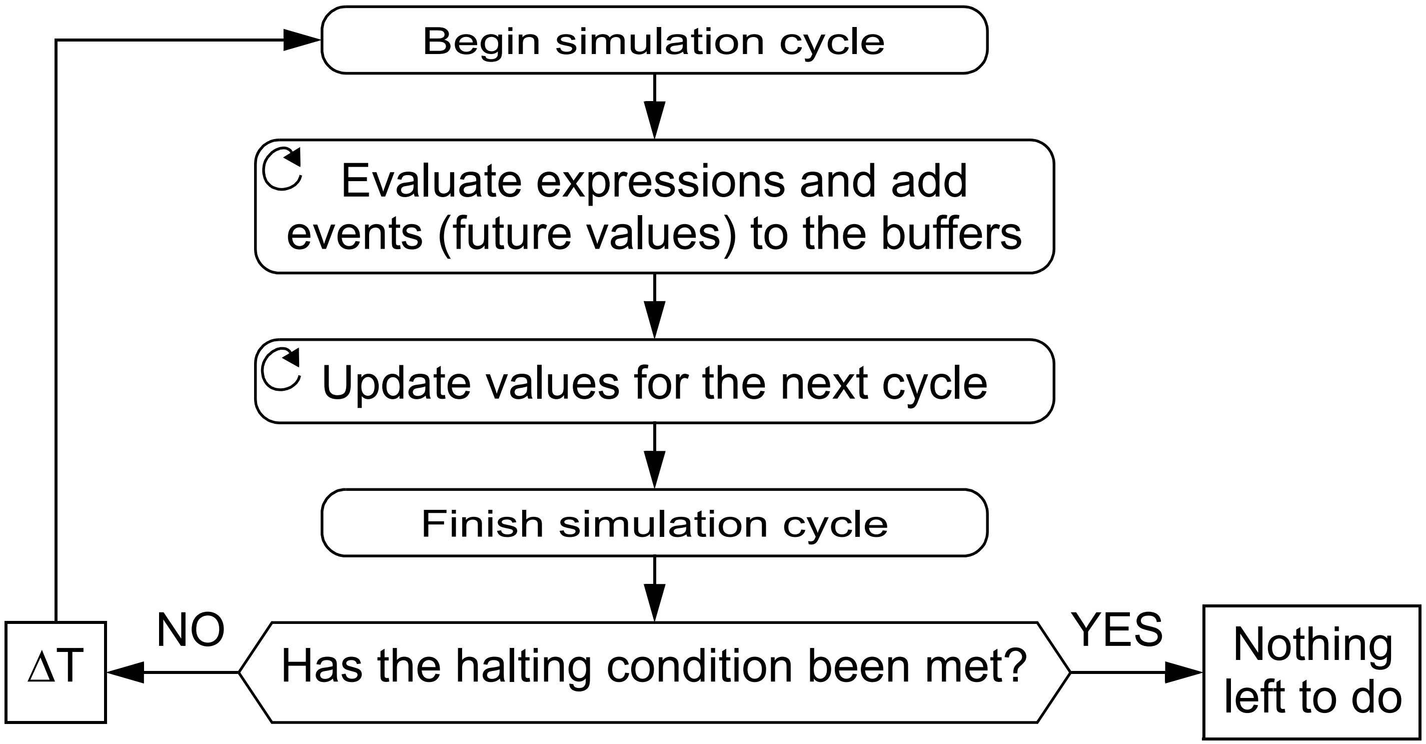
Simulators & timing/delay models

- Time & events
  - *Time-driven*:
    all components of the digital logic system are evaluated at every time step
  - *Event-driven*:
    system input events are kept in an time-ordered event queue
- Delay models
  - unit-delay (RTL simulator)
  - zero-delay (Verilog)
  - *delta*-delay (VHDL) – δ-delay, Δ-delay
- Simulation engines
  - all make use of the three following steps but details differ...
    - (1) calculate (and remember) new values for signals
    - (2) update signal values
    - (3) update time

Sequential and parallel simulation

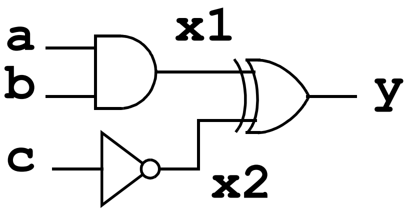


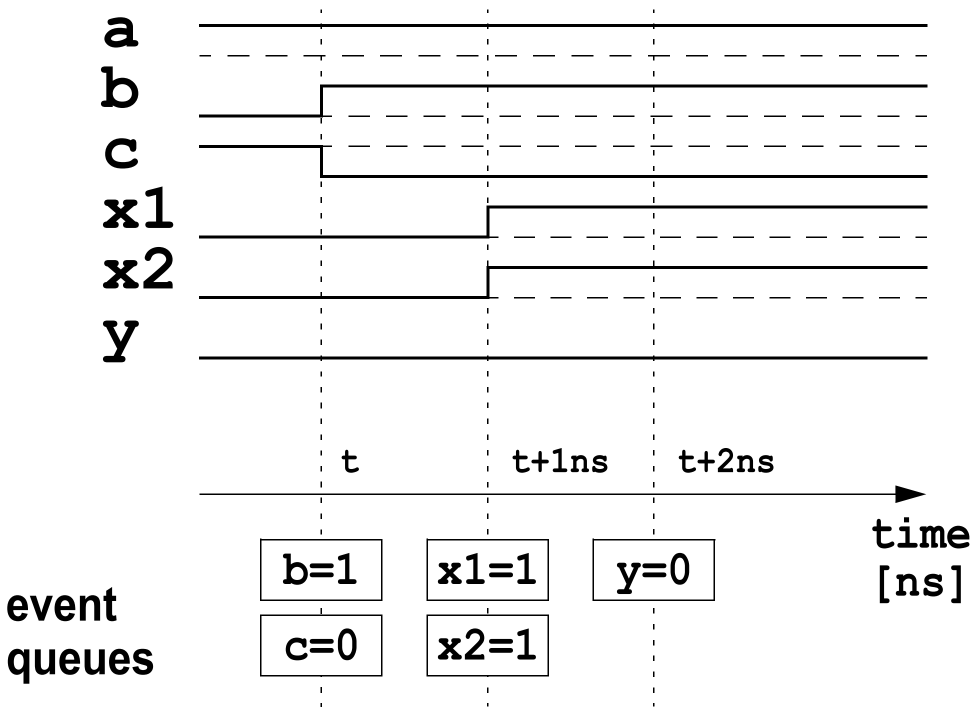
Unit-delay simulation model



Unit-delay simulation model (example)

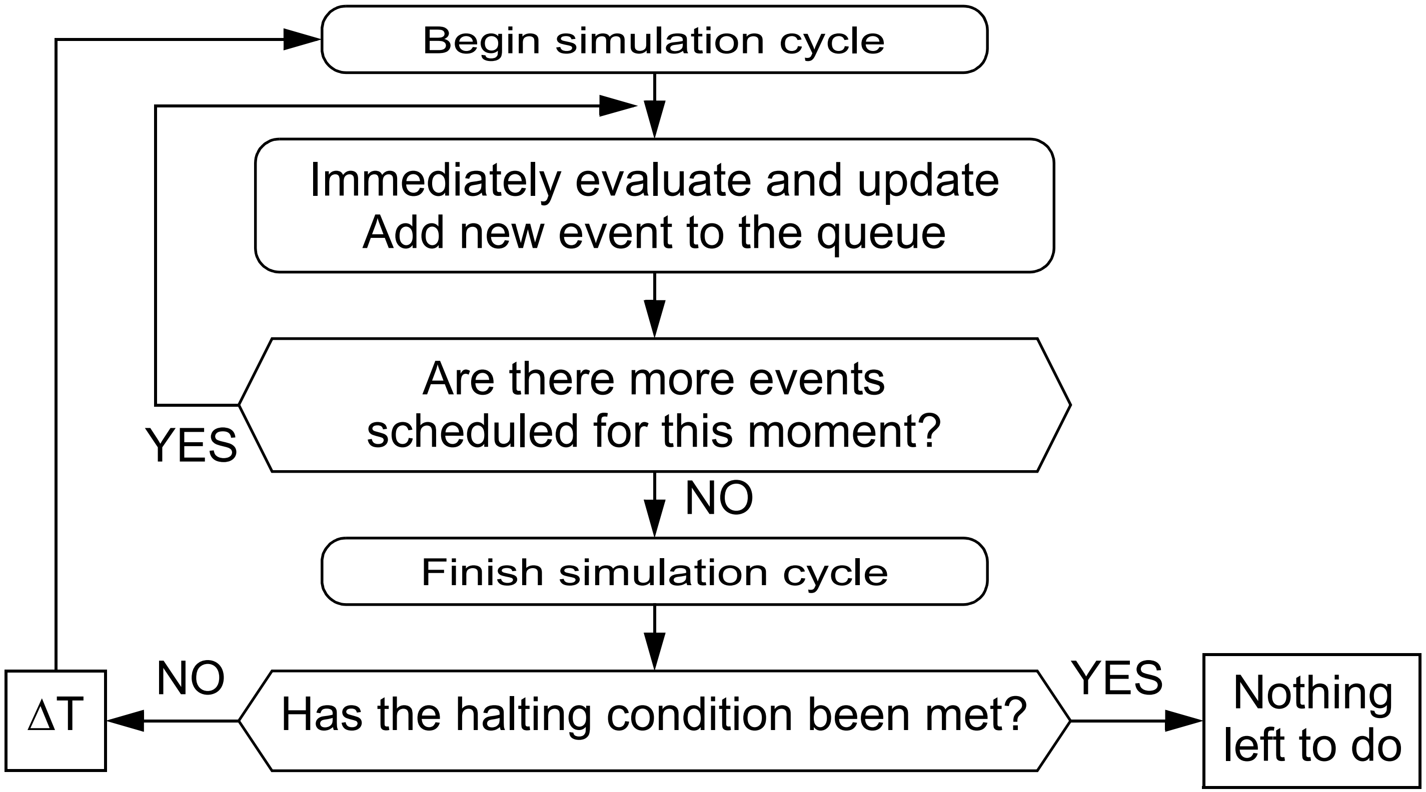
```
x1 <= a and b;
x2 <= not c;
y <= x1 xor x2;
```





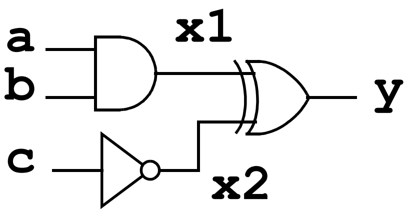
- Very fast but does not allow immediate signal changes

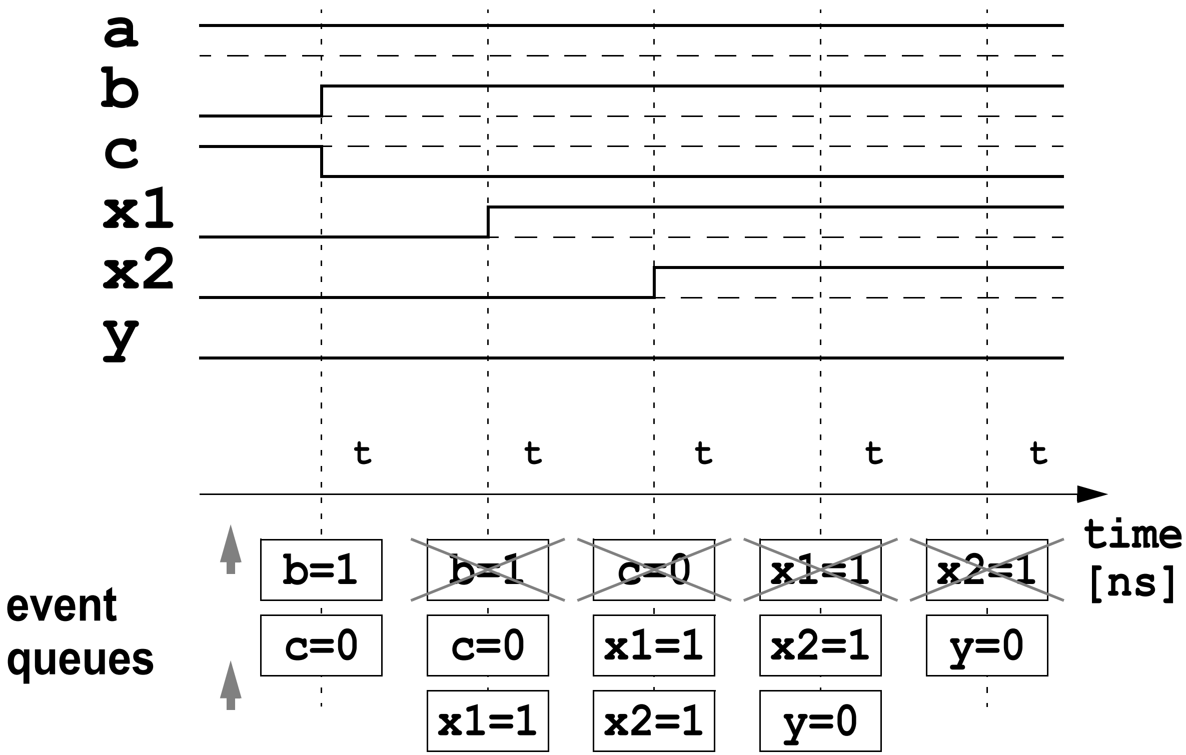
Zero-delay simulation model (Verilog)



Zero-delay simulation model (example #1)

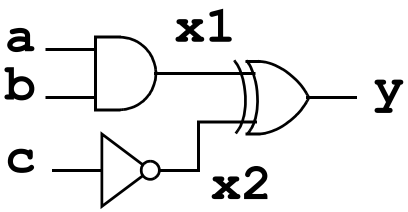
```
x1 <= a and b;
x2 <= not c;
y <= x1 xor x2;
```

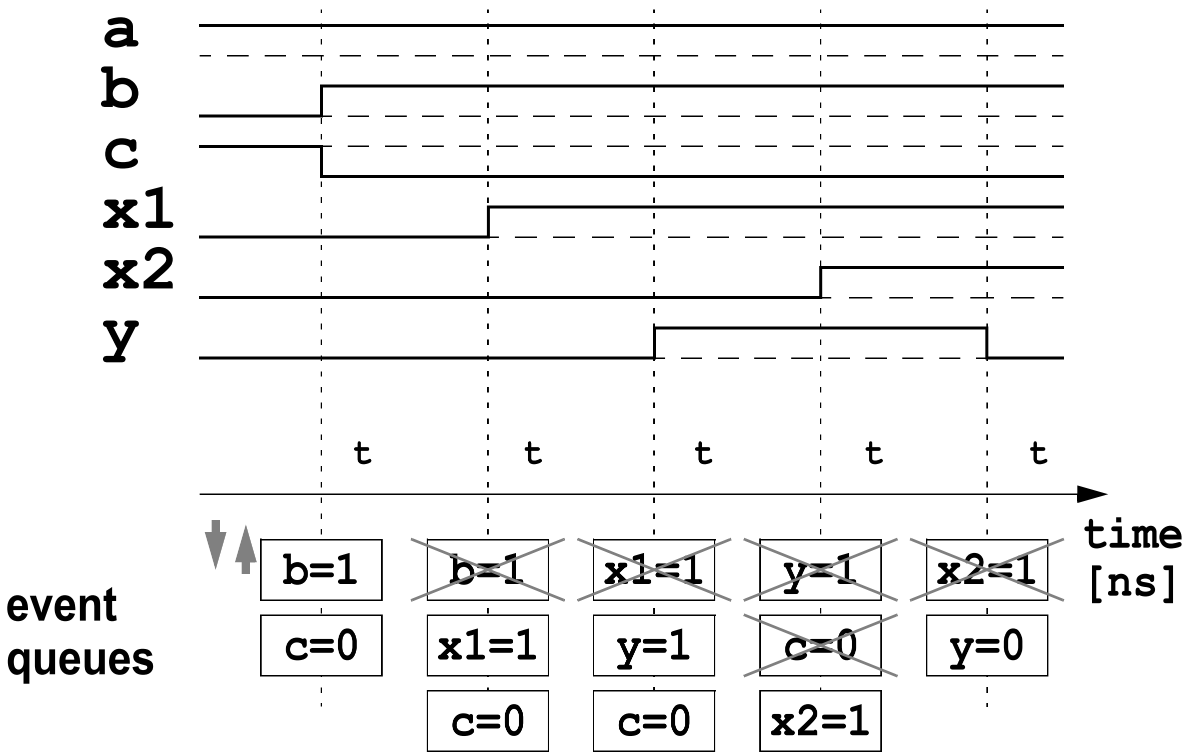




Zero-delay simulation model (example #2)

```
x1 <= a and b;
x2 <= not c;
y <= x1 xor x2;
```



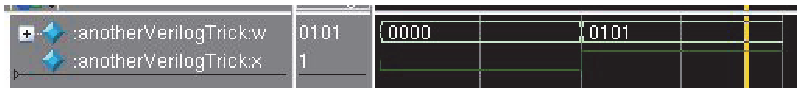


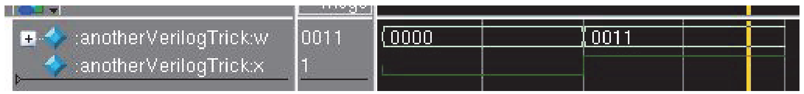
- **Fast and allows immediate signal changes, may suffer from non-determinism**

Non-deterministic behavior

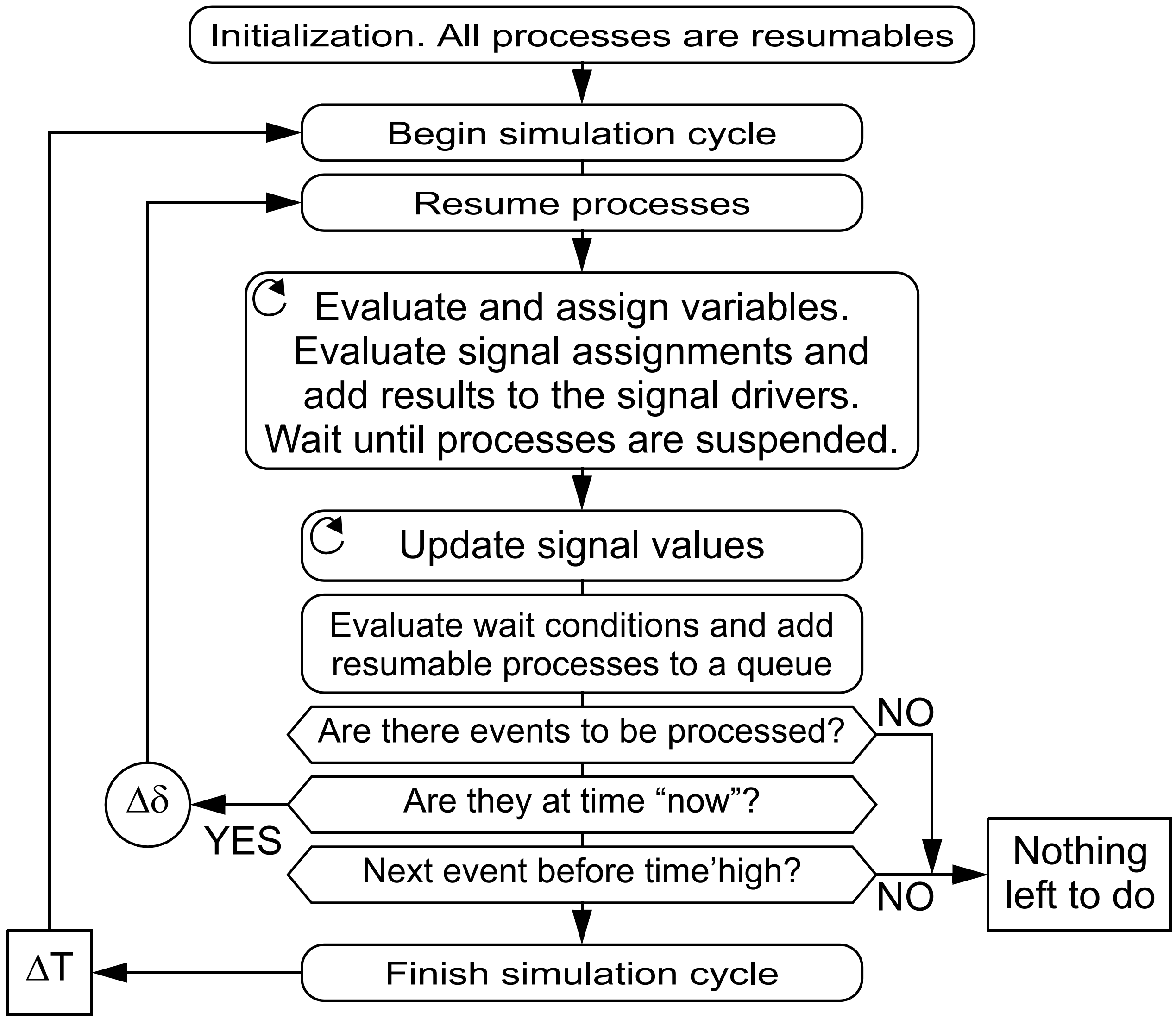
```
module anotherVerilogTrick;
  reg [3:0] w;
  reg x;
  initial begin
    x = 0;  w = 0;
    #100 x = 1;  #100;
  end
  always @(posedge x) w = 3;
  always @(posedge x) w = 5;
endmodule
```

```
module anotherVerilogTrick;
  reg [3:0] w;
  reg x;
  initial begin
    x = 0;  w = 0;
    #100 x = 1;  #100;
  end
  always @(posedge x) w = 5;
  always @(posedge x) w = 3;
endmodule
```



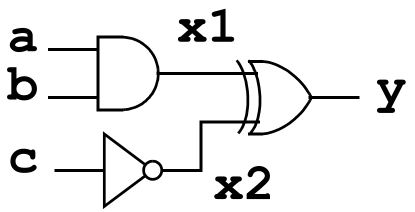


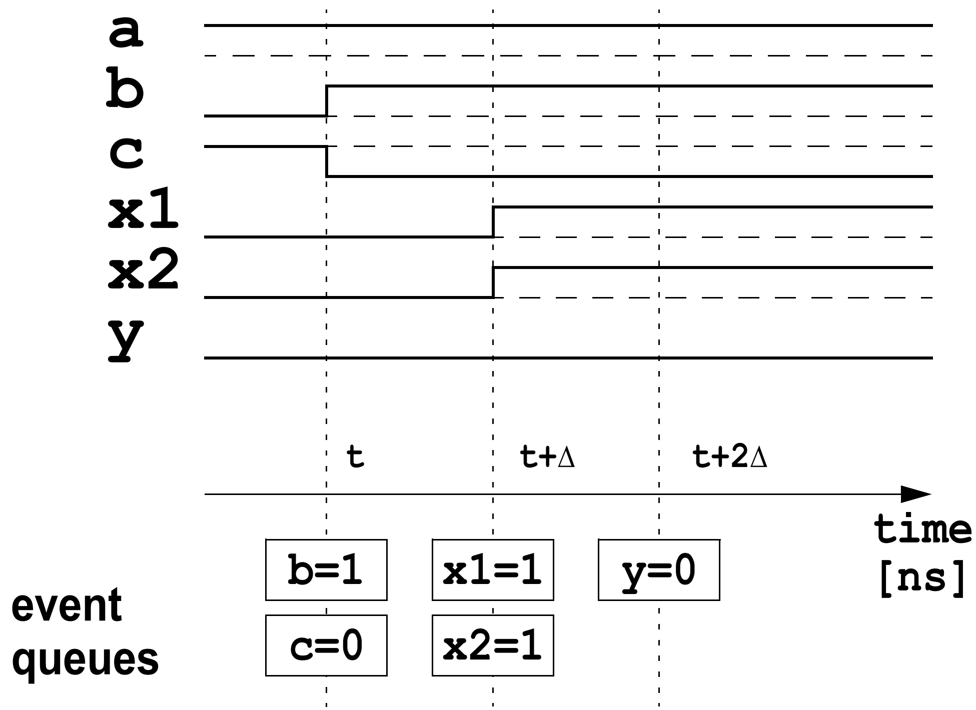
Delta-delay (VHDL) simulation model



Delta-delay (VHDL) simulation model (example)

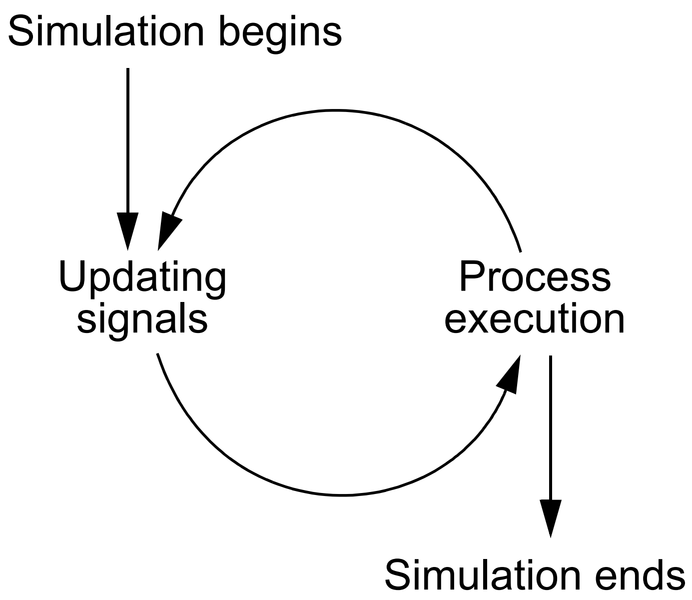
```
x1 <= a and b;
x2 <= not c;
y <= x1 xor x2;
```

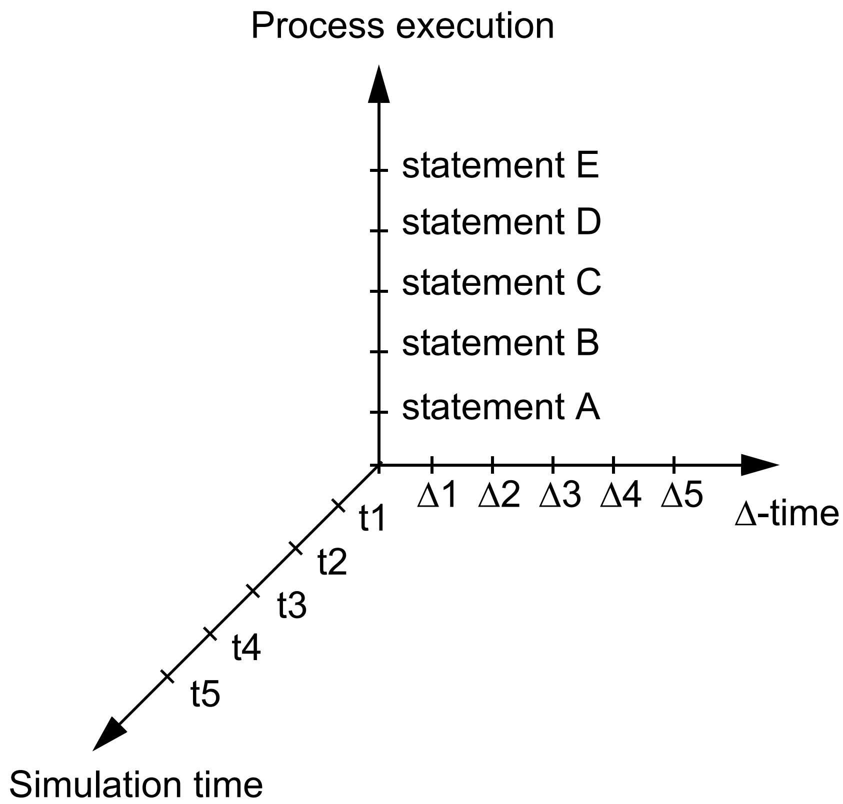




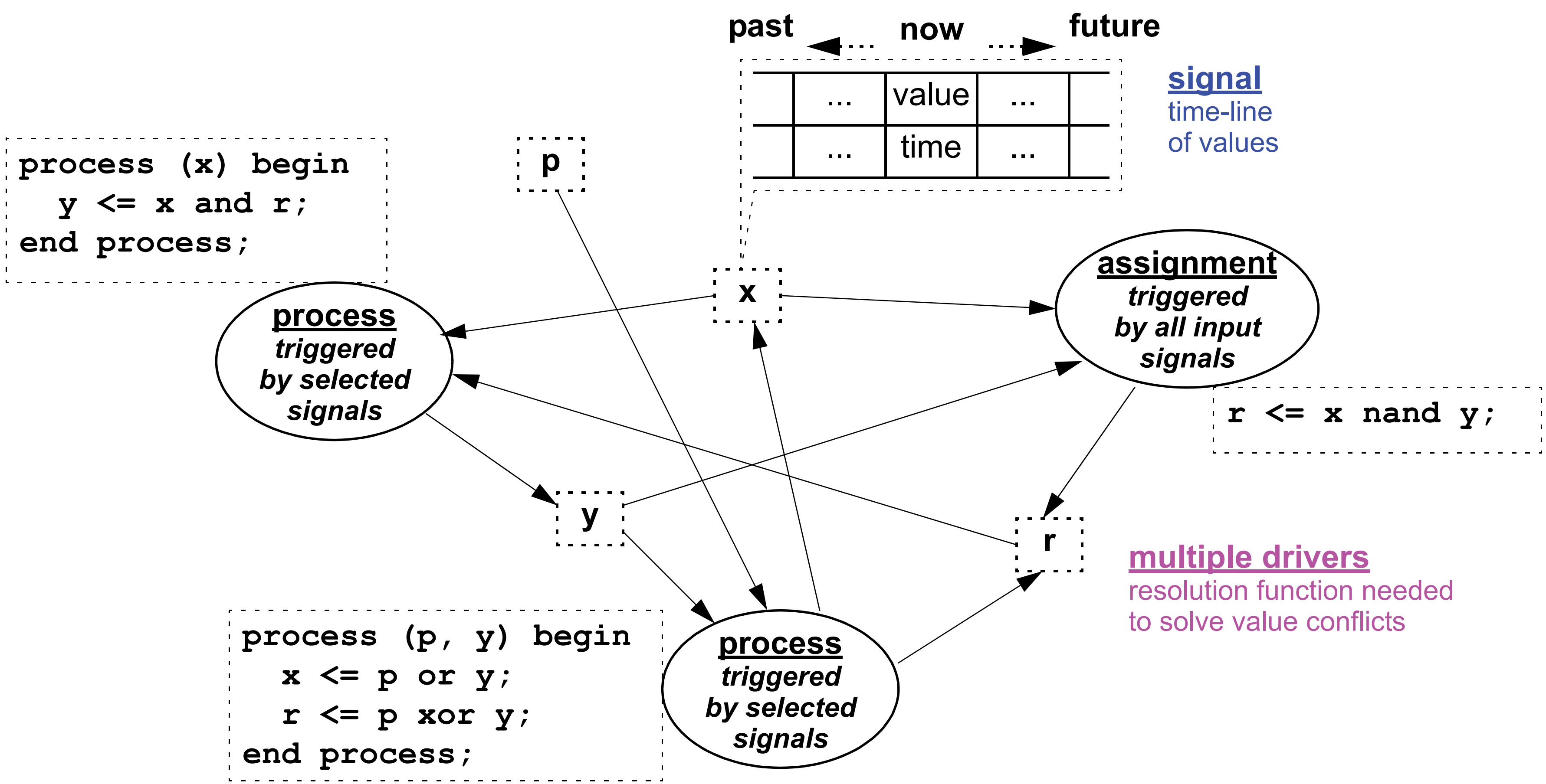
- Deterministic but slower and may suffer from delta-cycle oscillation

VHDL simulation scheme





VHDL simulation scheme



Delta delay example #1

```
-- SR flip-flop
x <= not (y and lset);   -- (1)
y <= not (x and reset);  -- (2)
```

- **Note the different initial values!**

time	lset	x	y	reset	stm.
20 ns	_	0	1	1	(1)
20 ns + 1Δ	0	_/	1	1	(2)
20 ns + 2Δ	0	1	_	1	(1)
20 ns + 3Δ	0	1	0	1	-
30 ns	_/	1	0	1	(1)
30 ns + 1Δ	1	1	0	1	-
40 ns	1	1	0	_	(2)
40 ns + 1Δ	1	1	_/	0	(1)
40 ns + 2Δ	1	_	1	0	(2)
40 ns + 3Δ	1	0	1	0	-

Delta delay example #2

```
-- SR flip-flop + delays
x <= not (y and lset)
     after 2 ns;          -- (1)
y <= not (x and reset)
     after 2 ns;          -- (2)
```

- Note the different initial values!

time	lset	x	y	reset	stm.
20 ns	_	0	1	1	(1)
22 ns	0	_/	1	1	(2)
24 ns	0	1	_	1	(1)
24 ns + 1Δ	0	1	0	1	-
30 ns	_/	1	0	1	(1)
30 ns + 1Δ	1	1	0	1	-
40 ns	1	1	0	_	(2)
42 ns	1	1	_/	0	(1)
44 ns	1	_	1	0	(2)
44 ns + 1Δ	1	0	1	0	-

Delta delay example #3

- **Dangers of default initialization**

```
-- SR flip-flop & oscillation
x <= not (y and lset);  -- (1)
y <= not (x and reset); -- (2)
```

- **Equal initial values will result in oscillation even with after clause!**

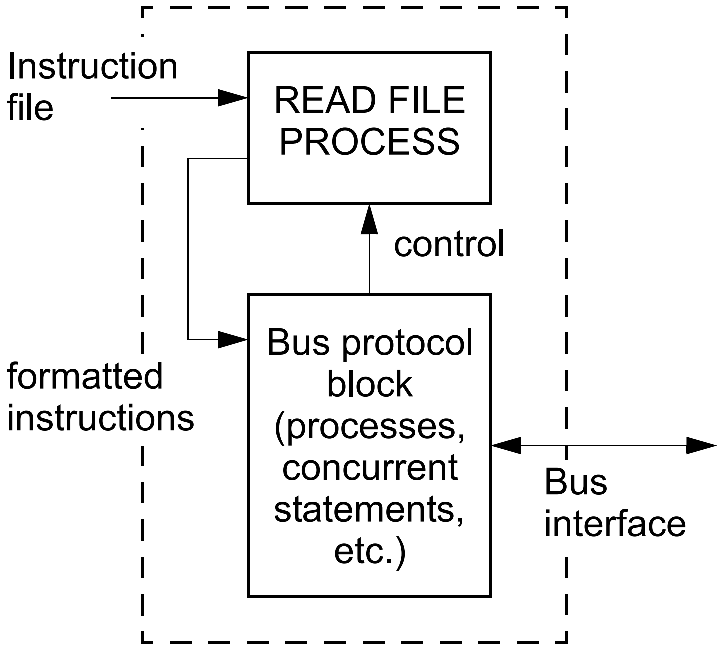
time	lset	x	y	reset	stm.
0 ns	_/	_	_	_/	(1),(2)
0 ns + 1Δ	1	_/	_/	1	(1),(2)
0 ns + 2Δ	1	_	_	1	(1),(2)
0 ns + 3Δ	1	_/	_/	1	(1),(2)
etc.	1	...	...	1	(1),(2)
0 ns	_/	_/	_/	_/	(1),(2)
0 ns + 1Δ	1	_	_	1	(1),(2)
0 ns + 2Δ	1	_/	_/	1	(1),(2)
0 ns + 3Δ	1	_	_	1	(1),(2)
etc.	1	...	...	1	(1),(2)

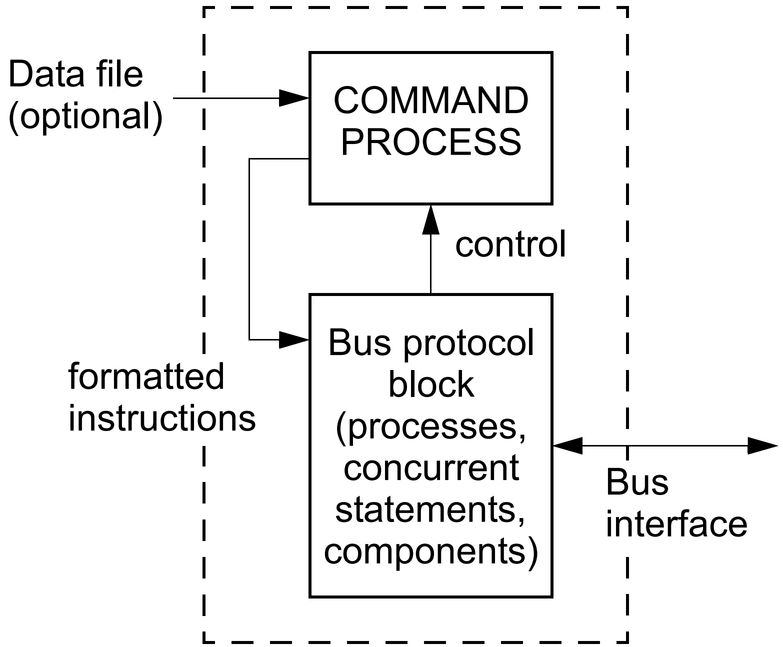
Simulation environment

- Testbench & the unit to be tested
  - *Testbench* (TB) is a VHDL component which instantiates the UUT
  - *Unit Under Test* (UUT) represents a design itself
- Testbench may make use of functional models
  - *Functional Model* (FM) is a model of a component which represents both the interfaces and the internal operation or structure of the component
  - *Bus Functional Model* (BFM) is a subset of the FM in that it only models the bus interfaces and bus transactions of the component
- Testbench's purposes
  - Stimuli generator(s)
  - Verifier against UUT specification
  - Report generation (human interface)

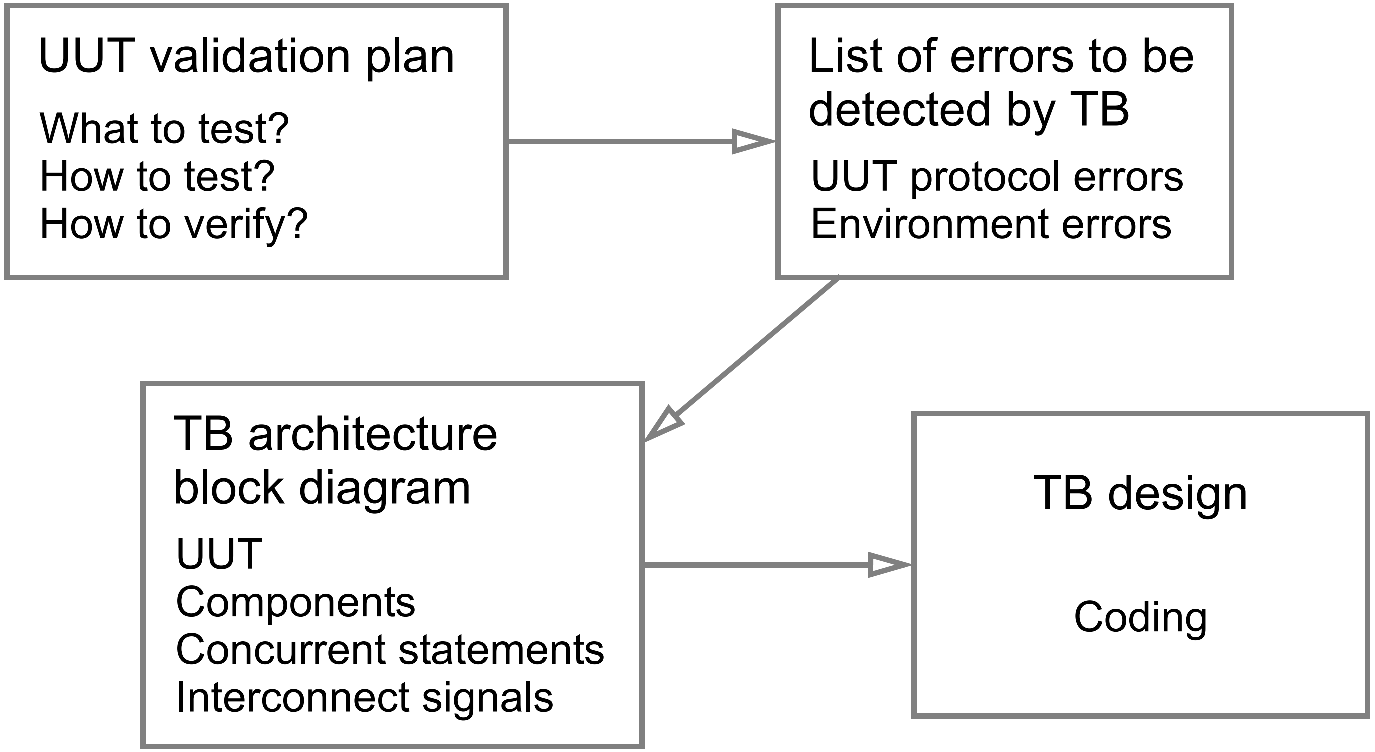
BFM modeling

- **Instruction file command format**
- **Architectural command format**





Testbench design methodology



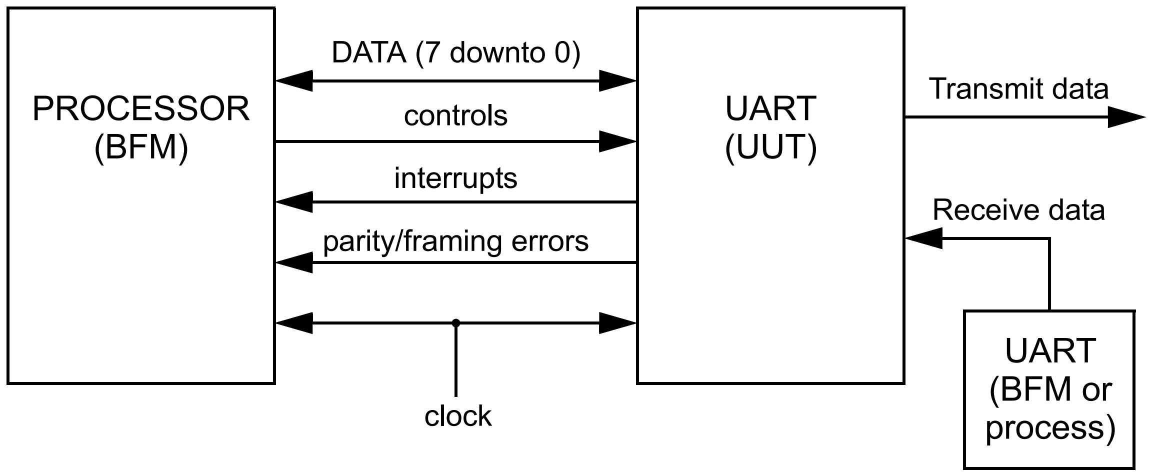
TB - TestBench
UUT - Unit Under Test

FM - Functional Model
BFM - Bus Functional Model

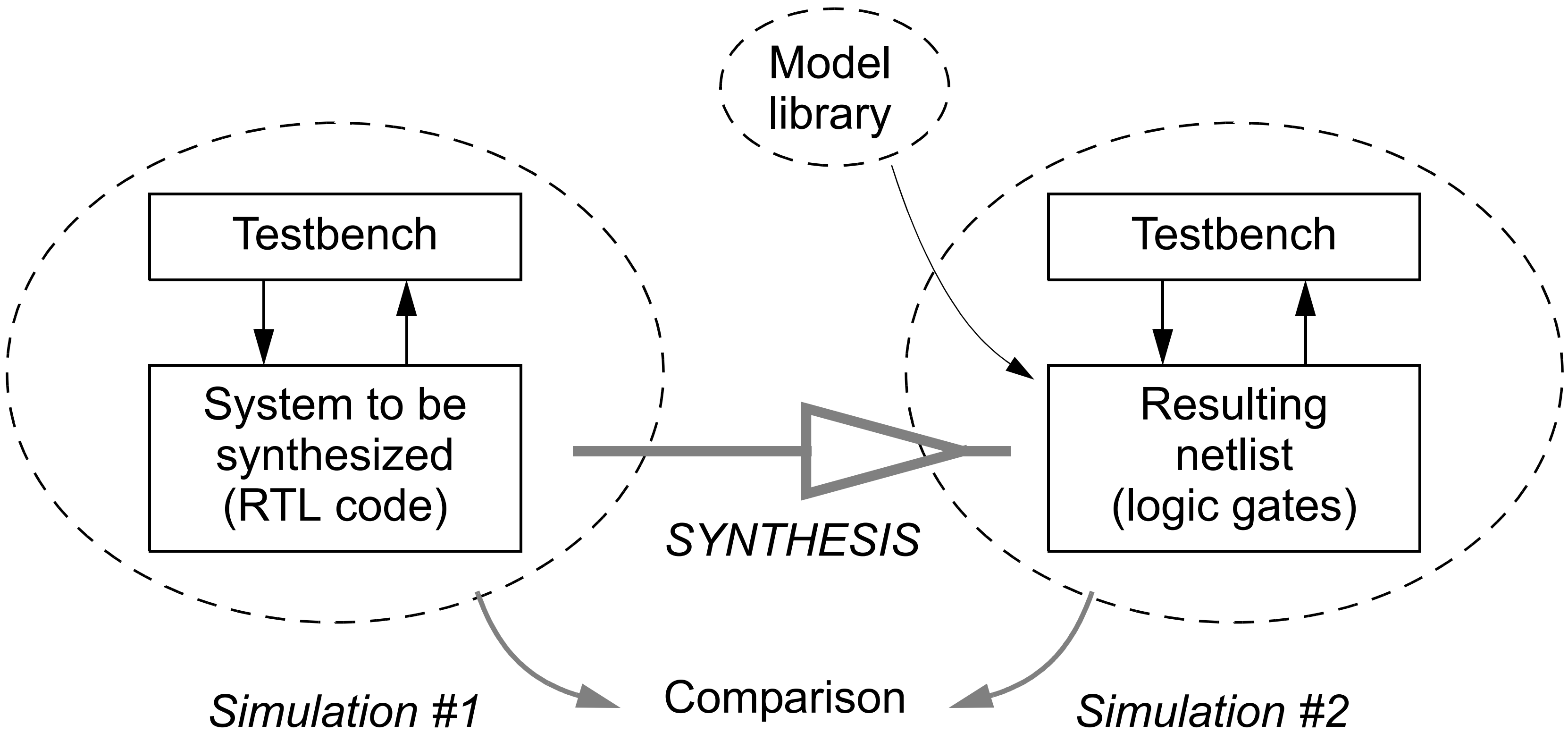
Testbench elements

- *UUT* (any abstraction level)
- Set of models that emulate *bus interfaces* and *bus transactions* to the UUT
- A *clock generator* for the system
- A *bus verifier* to perform timing and protocol checks (+ reports)

Example – UART testbench



Result validation methodology

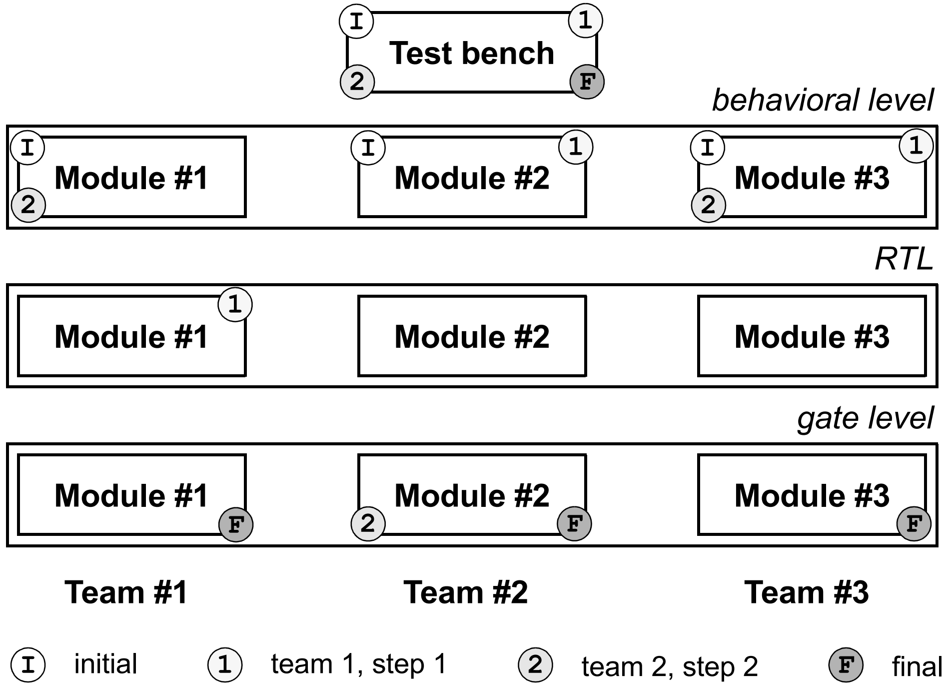


Result validation methodology

- How to compare the simulation results?
  - E.g., behavioral level modules vs. RT level modules
- Two simulations –> two signal traces
  - At which moments to compare?
    - Use clock flanks as “synchronization” points...
    - But behavioral level description may be without the clock signal?!
- Single simulation but two units to be tested in the same testbench
  - Two (or more) components of the same entity but with different architectures
    - One of them is the reference model, a.k.a. the Golden Device
    - Configuration declarations/specifications must be used
  - Synchronization is still a problem
    - The use of intelligent testbenches helps
    - Transaction Level Modeling (TLM) approaches can be used

Design refinement validation

- Large projects & multiple teams
  - one team – one module
    - behavioral –> RTL –> gate level
- Validating intermediate steps?
  - the same test bench for all teams
    - refining & replacing the module under design
  - interactive / intelligent testbenches
  - test sequence extraction

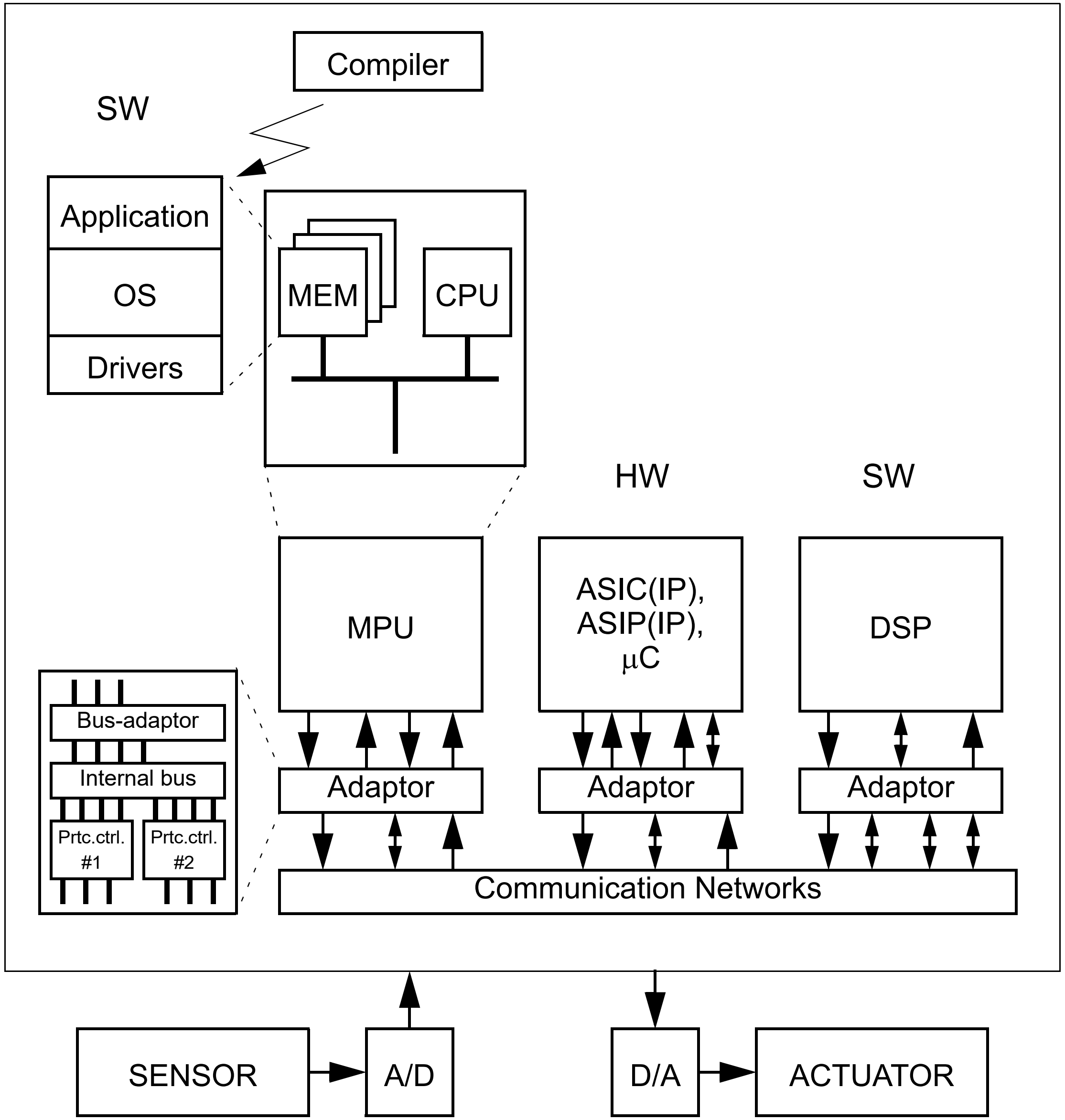


- "Sub-testbench" for components
  - stored test sequences + BFM (+ intelligent testbenches)

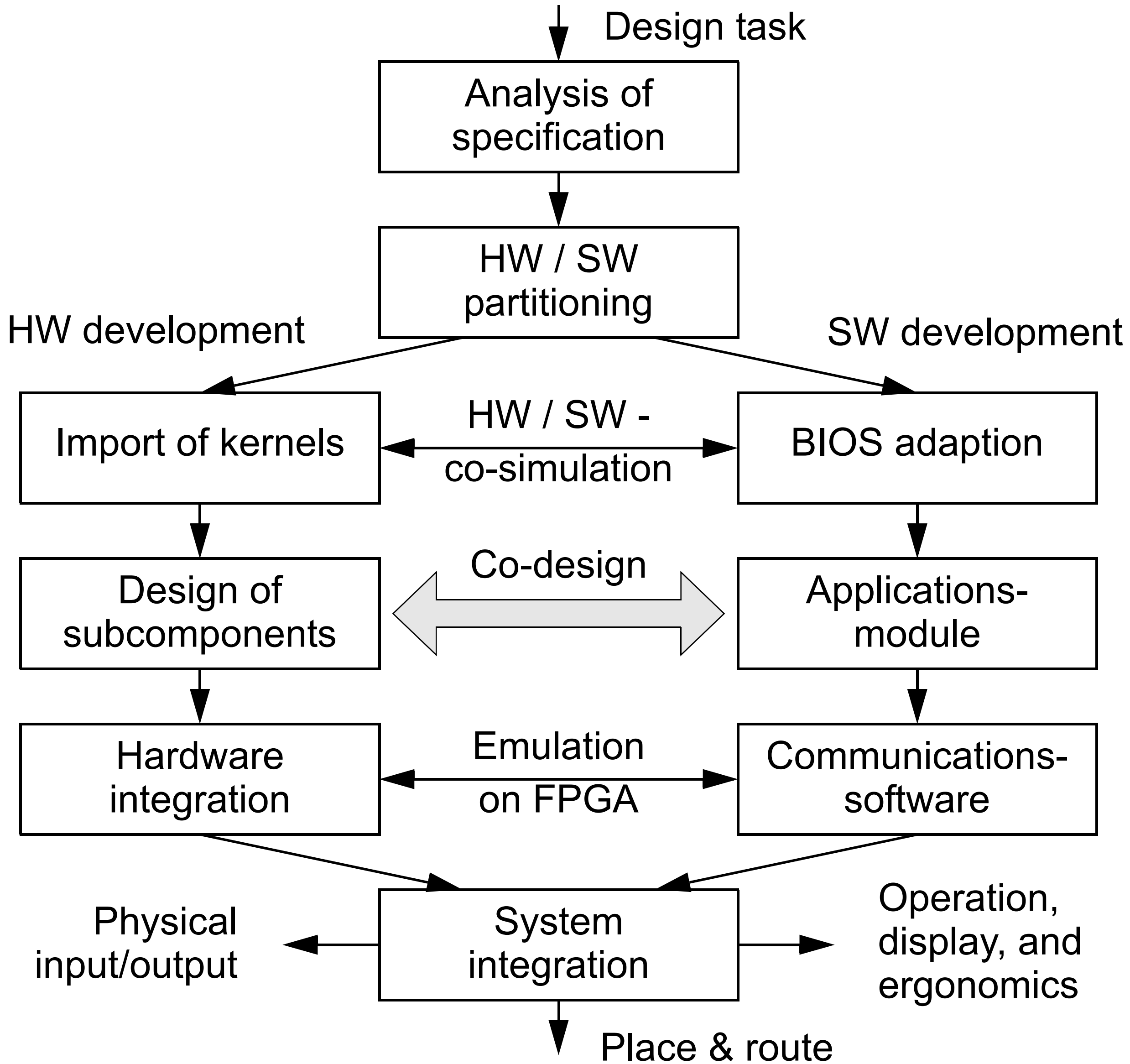
Co-simulation

- Manipulating simulated hardware with software
- The goal of co-simulation:
  To verify as much of the product functionality, hardware and software, as possible before fabricating the ASIC.
- In the past, co-simulation was adopted late in the process
  - after hardware is deemed to be working and stable
  - painful integration process, design flaw and could re-spin the silicon
- Today, behavioral model simulation has matured and simulation tools have improved to allow better simulation throughout the development cycle
  - Rabi N. Mahapatra (Texas A&M University) http://codesign.cs.tamu.edu/teaching/csce617/

Embedded systems components



Concurrent design of software and hardware



Simulation components

- Hardware design: Memory, CPU or many ASICs each with one or more CPUs
- Simulation platform:
  - PC or workstation. Everything exist as processes.
  - Hybrid platforms with co-processors: off-load part of the load to co-processor, peripheral and test benches remain in software.
- Emulation
  - Special simulation environment with hardware
    - runs whole design
    - expensive
    - 10% of real time
    - FPGA arrays may be the hardware
    - allow designers of large products to find a class of problem that cannot be found in simulation
    - can attach to real devices

Algorithms

- Event driven simulation (gate level simulation)
  - Most accurate - every active signal is calculated for every device as signals propagate
  - Each signal is simulated for its value and its time of occurrence
  - Excellent for timing analysis and to verify race conditions
  - Computation intensive and therefore very slow
- Cycle-based simulation
  - Calculates the state of the signals at active clock edge
  - Suitable for complex design that needs large number of tests
  - ~10 times faster than event driven simulation
- Data-Flow Simulator
  - Signals represented as stream of values (without notion of time)
  - Blocks are executed when signals present at the input
  - Scheduler in the simulator determines the order of block executions
  - High level abstraction simulation used in the early stages of verification

Hardware requirements

- Most simulators can handle behavioral models
- Emulators require synthesizable codes
- Some simulators may not handle HDLs
- Cycle-based simulators can handle asynchronous designs at severe performance penalty

Software requirements

- Simulation environment has effects on application software
- Programmers certainly need alternate version of application that do not have user interface code or any references to chips that is not part of the simulation environment
- Reduce size of functionality and tables for speed

Co-simulation methods

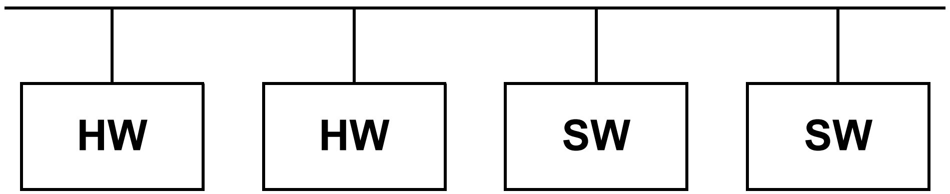
- Co-simulation is a way to simulate at a very high level of abstraction
- By creating a functional model that can be tested, system designers can make sure the requirements are clear
- Making a single model of both hardware and software functionality, the design boundary between the two is effectively removed
- Running model allows engineers to test different hardware/software functionality splits (mapping) for performance and get some rough timing estimates for various ideas
- Functional model also allows engineers to find fundamental bugs in the design

Co-simulation methods

- POLIS (UC Berkeley)
  - Cadence's Cierto VCC is based on ideas from POLIS
- Synopsy's COSSAP and Eaglei tools
  - promise a way to check the implementation against the original algorithmic specification for function equivalence
- The standard method - running software directly on simulated hardware
  - it is implied that the CPU is part of the ASIC -->
    CPU is simulated at the same level as other hardware
  - good when designing the CPU
  - waste of simulation results when using a core from the vendor

Heterogeneous co-simulation

- **Network different type of simulators together to attain better speed**
- **Claims to be actual co-simulation strategy as it affords better ability to match the task with the tool, simulates at the level of details.**
- **Synopsys' Eaglei**
  - **let HW run in many simulators**
  - **let SW on native PC/workstation or in instruction-set-simulator (ISS)**
  - **Eaglei tool interfaces all these**



Heterogeneous co-simulation

- How about performance?
- Complex enough to describe any situation
  - Proponents: since software is not running at hardware simulation speed, the actual performance will be higher
    - How fast is the software running when not doing hardware related task?
  - If target CPU is not PC cross compiler should be used
  - When software runs directly on PC/WS, it runs at the speed of PC/WS
  - When software can not run directly as processes on WS, instruction set simulator (ISS) is needed
    - ISS interprets assembly language at instruction level as long as CPU details are not an issue
    - ISS usually runs at 20% of the speed of actual or native processes

Hardware density of heterogeneous simulation

- How much time software accesses hardware?
- Hardware density depends on applications
- In loosely coupled CPU system, the block responsible for hardware initializations has 30% instructions to access the hardware
- In tightly coupled system, every memory reference could go through simulated hardware
- In general hardware density is important for simulation speed
- The base hardware and tools that communicate between the heterogeneous environment can attribute to the speed also
- If simulation is distributed (rather common these days), the network bandwidth, reliability and speed matters also

Co-simulation strategies

- What you simulate is what you get
  - Simulation is important for bug free test of the product
  - The product schedule forces suitable strategies
- Due to decrease in feature size and increase in die size, more functionality are pushed into hardware (could never happened in the past)
  - Creates challenges for testing due to increased functionality
    - Formal design methods, code reviews and code reuse have help
    - Emulation engine is also of help but expensive
- For typical strategies, we need to know the thoroughness of testing
  - Details of the surrounding environment
  - If it involves health and safety, then detailed testing strategy is needed

Co-simulation strategies

- Multi-pronged functional test strategy to build levels of assurance
  - Basic initial tests prove functionality and complex tests are built upon working
  - Any single test method has some coverage hole
  - Event driven tests are closest to the real hardware but its slowness is coverage hole!
  - Make balance between required test coverage and what might be avoided

- A simulation strategy might call for the functional specification to be written as a functional model (co-design)
  - Hardware designer could use event driven tests for hardware blocks
  - Software designer could do basic debug using ISS or cross compiler and with fake hardware calls
    - For detailed functional blocks, software could interface
    - After, completion of blocks, these can be dropped into the functional model for regression tests

Co-simulation strategies

- **Simulation speed**
  - **Degrades when real components replace the functional blocks**
  - **The simulation speed depends on simulation engine, the simulation algorithm, the number of gates in the design, and whether the design is primarily synchronous or asynchronous**
- **Low cost cycle based simulation is a good compromise**
  - **Since it can not test physical characteristic of a design, event driven simulator may be used in conjunction**
- **Cycle based simulators and emulators may have long compilation**
  - **Hence, not suitable for initial tests that needs many changes.**
  - **Event driven and cycle based simulators have fairly equal debugging environments, all signals are available at all times**
  - **Emulators on the other hand, require the list of signals to be traced to be declared at compilation time**

Co-simulation strategies

- If the next problem can be found in a few microseconds of simulated time, then slower simulators with faster compilation times are appropriate
- If the current batch of problems all take a couple hundred milliseconds, or even seconds of simulated time, then the startup overhead of cycle based simulation or even an emulator is worth the gain in run time speed
- How about the portability of test benches?
- Test after fabrication?
  - Fast simulators are useful
  - It is difficult to track down the hardware fault

Co-simulation strategies

- Determining which parts of the system software to run and how much software debug can be done without the hardware
- SW engineer need to go through the code and disable functionality that is too costly for simulation, or if the sequence is important, find ways to reduce its execution time
- The degree of fidelity between the simulated environment and the real world is both a requirement of simulation and a constantly shifting target throughout the simulation effort

How to co-simulate?

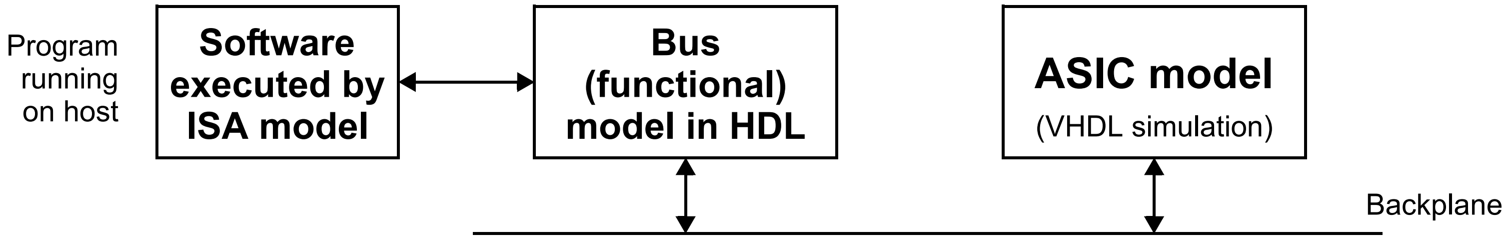
- **How to simulate hardware components of a mixed hardware-software system within a unified environment?**
  - **This includes simulation of the hardware module, the processor, and the software that the processor executes**
- **How to simulate hardware and software at same time?**
- **What are various challenges?**
  - **Software runs faster than hardware simulator.**
    How to run the system simulation fast keeping the above synchronized?
  - **Slow models provide detailed and accurate results than fast models.**
    How to balance these effects?
  - **Use of different platforms for simulations.**

Detailed processor model

- Processor components (memory, datapath, bus, instruction decoder, etc.) are discrete event models as they execute the embedded software
- Interaction between processor and other components is captured using native event-driven simulation capability of hardware simulator
- Gate level simulation is extremely slow (~tens of clock cycles/sec), behavioral model is ~hundred times faster
- Most accurate and simple model

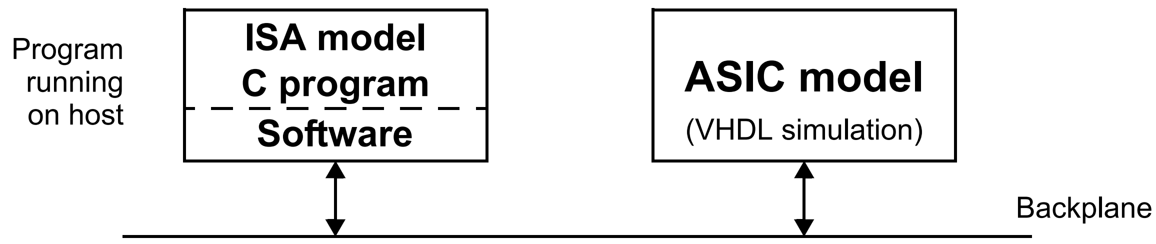
Bus model

- Cycle based simulator
- Discrete-event shells that only simulate activities of bus interface without executing the software associated with the processor
  - Useful for low level interactions such as bus and memory interaction
- Software is executed on ISA model and provides timing information in clock cycles for given sequence of instructions between pairs of IO operation
- Less accurate but faster simulation model



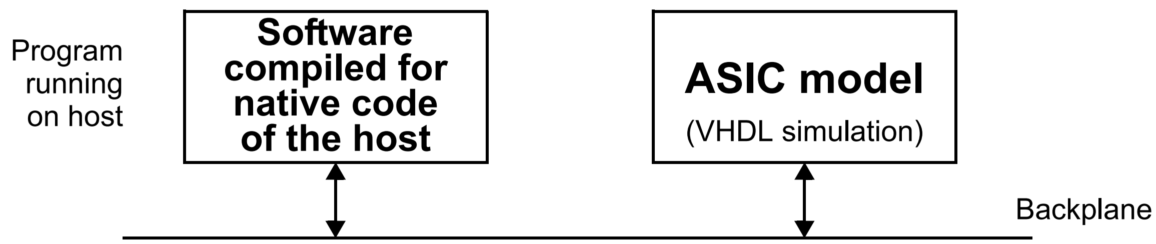
Instruction Set Architecture model

- ISA can be simulated efficiently by a C program
  - C program is an interpreter for the embedded software
- No hardware mode
  - Software is executed on ISA model
  - Provides timing (clock) details of the co-simulation
- Can be more efficient than detailed processor modeling because internals of the processor do not suffer at the expense of discrete-event scheduling



Compiled model

- Very fast processor models are achievable in principle by translating the executable embedded software specification into native code for processor doing simulation
  - Ex: Code for programmable DSP can be translated into Sparc assembly code for execution on a workstation
- No hardware, software execution provides timing details on interface to co-simulation
- Fastest alternative, accuracy depends on interface information

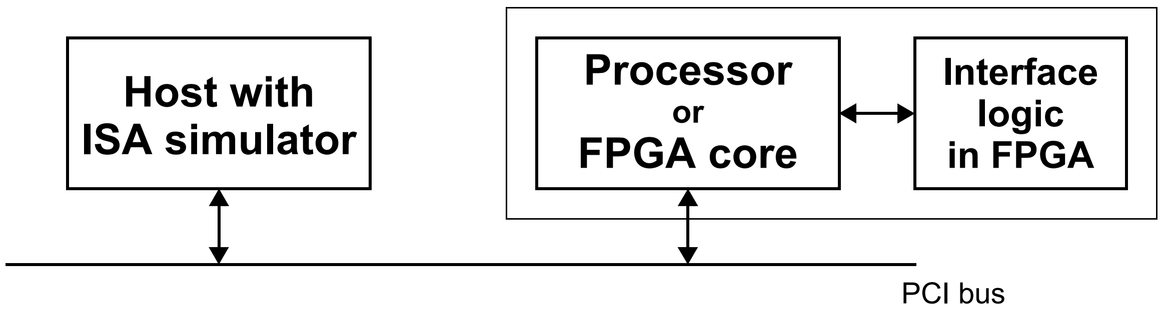


Hardware model

- If processor exists in hardware form, the physical hardware can often be used to model the processor in simulation
  - Alternatively, processor could be modeled using FPGA prototype (emulating)
- Advantage: simulation speed
- Disadvantage: availability of the physical processor

Combined HW/SW approach

- **The host is responsible of having OS, some applications and might have superset simulating environment (RSIM, SIMICS, SIMOID)**
- **Use of fast backplane (PCI) for communication**
  - **Real processor or processor core in FPGA as hardware model, and ASIC/FPGA for interface and interconnection for hardware modeler**
- **Good for fast complex architecture simulations including multiprocessor**

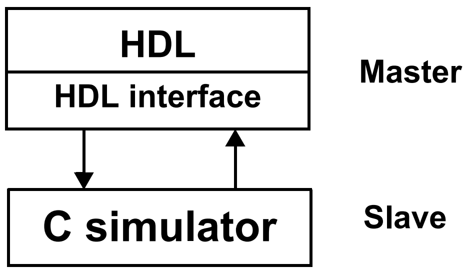


Domain coupling

- The host that runs software is required to interact with hardware model(s)
- Difficulties
  - providing timing information across the boundaries
  - coupling two domains with proper synchronization
- Simulation at different levels of abstraction
  - in the beginning of design process, hardware synthesis is not available
    - use functional model to study the interaction between HW and SW
  - after refinement(s), replace functional model with more detailed one(s)
  - when detailed operation of hardware is verified, swap back to the higher levels
    - this is to gain simulation speed
- The co-simulation environment should support different levels of abstraction
  - off-the-shelf components – design is not a part of the current design process
  - functional model is enough, no need to know internal details

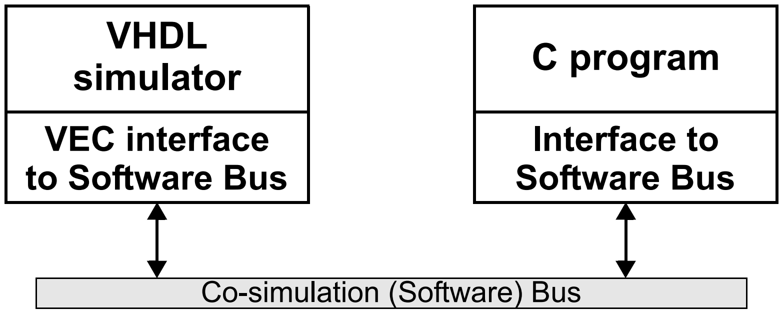
Master-slave co-simulation

- One master simulator and one or more slave simulators
  - Slave is invoked from master by a procedure call
- The language must have provision for interface with different language
  - Programming Language Interface (PLI)
- Difficulties:
  - No concurrent simulation possible
  - C procedures are reorganized as C functions to accommodate calls



Distributed co-simulation

- Software bus transfers data between simulators using procedure calls based on some protocol
- Implementation of System Bus is based on system facilities
  - Unix IPC or socket
  - It is only a component of the simulation tool.
- Allows concurrency between simulators



Synchronization and time in co-simulation

- In the case of a single simulator there is no problem for timing as single event queue is managed for simulation
- With several simulators and software programs in the domain:
  - hardware and software domain are using a handshaking protocol to keep their times (clocks) synchronized
    - signals (events) transferred from one side to the other should have attached a time stamp
  - it is possible to use a loosely coupled strategy that allows the two domain to proceed more independently
    - if a signal is received with a time stamp lower than the current clock in the respective domain, the respective simulator has to be back up

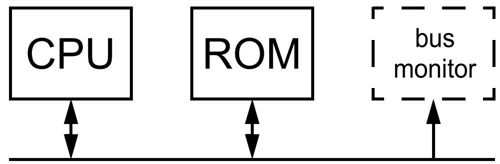
Levels of co-simulation technology

Abstraction level	Speed	Debug	Model	Turn-around	Soft-ware	Hard-ware
Nano-second accurate	1 - 100	Best	Hardest	Fast	OK	Yes
Cycle accurate	50 - 1000	Excellent	Hard	Fast	OK	Yes
Instruction level	2000 - 20,000	OK	Medium	Fast	Yes	OK
Synchronized handshake	Limited by hardware simulation	No processor state	None	Fast	Yes	OK

Levels of Co-simulation Technology

Abstraction level	Speed	Debug	Model	Turn-around	Soft-ware	Hard-ware
Virtual hardware	Fast	No processor or hardware state	None	Fast	Yes	No
Bus functional	Limited by hardware simulation	No processor state	Easier	Fast	No	Yes
Hardware modeler	10 - 50	No processor state	Timing only	Fast	OK	Yes
Emulation	Fast	Limited	None	Slow	OK	OK

Co-simulation example – different languages



- A small system – CPU + memory
  - CPU - bus functional model
  - instruction memory (ROM) - functional model
  - testbench - clock generator, reset circuitry and bus monitor
- All modules in VHDL
  - memory content - constant array
- All modules in Verilog (SystemVerilog)
  - memory content - dump file
- Co-simulation case
  - CPU in VHDL – easier to manage causality (no danger of non-determinism)
  - memory and testbench in Verilog – simpler code + memory content from file
  - data types, module names, etc. – no changes... [:-)]

TTÜ1918

All in VHDL

```
--
-- Bus-functional model of the processor
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity  processor  is
  port ( address: out unsigned(3 downto 0);
         data: in unsigned(3 downto 0);
         clk, res: in bit );
end processor;

architecture  BFM  of  processor  is

begin  --  BFM
  process
    variable pc, ir: unsigned(3 downto 0)  :=
      (others=>'0');
  begin
    address <= pc;
    wait on clk until clk='1';
    if res='0' then      pc := (others=>'0');
    else
      ir := data;
      if ir(3)='1' then  pc := ir(2 downto 0) & '0';
      else               pc := pc + '1';    end if;
    end if;
  end process;
end BFM;
```

```
--
-- Functional model of the instruction memory (ROM)
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity  memory  is
  port ( address: in unsigned(3 downto 0);
         data: out unsigned(3 downto 0) );
end memory;

architecture  FM  of  memory  is

begin  --  FM
  process
    type mem_array is array (integer range <>)
      of unsigned(3 downto 0);
    constant memo: mem_array(0 to 15) :=
    ( ”0101”, ”0110”, ”0010”, ”1110”,
      ”0001”, ”0010”, ”0011”, ”0100”,
      ”0101”, ”0110”, ”0111”, ”1000”,
      ”0101”, ”0110”, ”0010”, ”1010” );
  begin
    wait on address;
    data <= memo(conv_integer(address));
  end process;
end FM;
```

All in VHDL

```
--
-- Test-bench and bus monitor
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

use STD.textio.all;

entity  test  is
end test;
```

```
 > run 220 ns
 # 5 ns:  0000  0101
 # 15 ns:  0000  0101
 # 25 ns:  0000  0101
 # 35 ns:  0001  0110
 # 45 ns:  0010  0010
 # 55 ns:  0011  1110
 # 65 ns:  1100  0101
 # 75 ns:  1101  0110
 . . .
 # 155 ns:  1001  0110
 # 165 ns:  1010  0111
 # 175 ns:  1011  1000
 # 185 ns:  0000  0101
 # 195 ns:  0001  0110
 # 205 ns:  0010  0010
 # 215 ns:  0011  1110
```

```
architecture  bench  of  test  is
  signal clk, res: bit := '1';
  signal a_bus, d_bus: unsigned(3 downto 0);

  component processor
    port ( address: out unsigned(3 downto 0);
           data: in unsigned(3 downto 0);
           clk, res: in bit );
  end component;
  component memory
    port ( address: in unsigned(3 downto 0);
           data: out unsigned(3 downto 0) );
  end component;
begin  --  bench
  clk <= not clk after 5 ns;
  res <= '0', '1' after 22 ns;

  CPU: processor port map ( a_bus, d_bus, clk, res );
  MEM: memory port map ( a_bus, d_bus );

  process -- bus monitor
    variable out_buff: line;
    variable data_buff: bit_vector(3 downto 0) := ”0000”;
  begin
    wait on clk until clk='0';
    write(out_buff,now); write(out_buff,string'(”:  ”));
    data_buff := to_bitvector(std_logic_vector(a_bus));
    write(out_buff,data_buff); write(out_buff,string'(”  ”));
    data_buff := to_bitvector(std_logic_vector(d_bus));
    write(out_buff,data_buff); writeline(output,out_buff);
  end process;
end bench;
```

All in Verilog

```
// Bus-functional model of the processor
`timescale 1 ns / 1 ns

module processor ( address, data, clk, res );
  output [3:0] address;  reg [3:0] address;
  input  [3:0] data;
  input        clk, res;

  reg [3:0] pc, ir;
  initial begin pc=0; address=0; ir=0; end

  always begin
    address=pc;
    @(posedge clk);
    if (res==0)     pc=0;
    else begin
      ir=data;
      if (ir[3]==1) pc={ir[2:0],1'b0};
      else          pc=pc+1;
    end
  end
endmodule
```

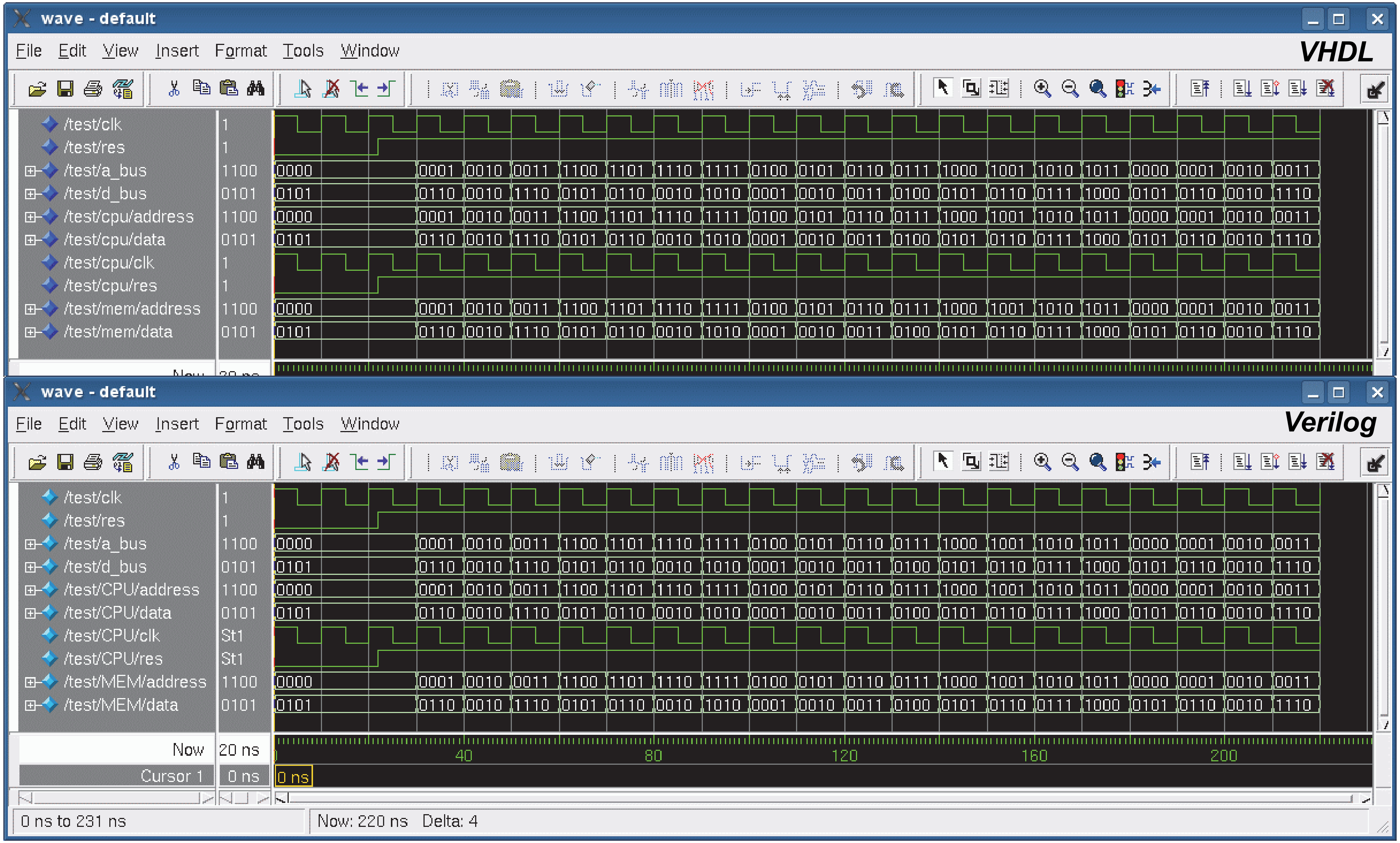
```
> run 220 ns
#    5:  0000  0101
#   15:  0000  0101
#   25:  0000  0101
#   35:  0001  0110
#   45:  0010  0010
. . .
#  195:  0001  0110
#  205:  0010  0010
#  215:  0011  1110
```

```
@0
0101
0110
0010
1110
0001
0010
0011
0100
0101
0110
0111
1000
0101
0110
0010
1010
```

```
// Functional model of the instruction memory (ROM)
`timescale 1 ns / 1 ns

module memory ( address, data );
  input  [3:0] address;
  output [3:0] data;  reg [3:0] data;

  reg [3:0] memo [0:15];
  initial $readmemb("co-memo.txt",memo);

  always @(address) data=memo[address];
endmodule
```

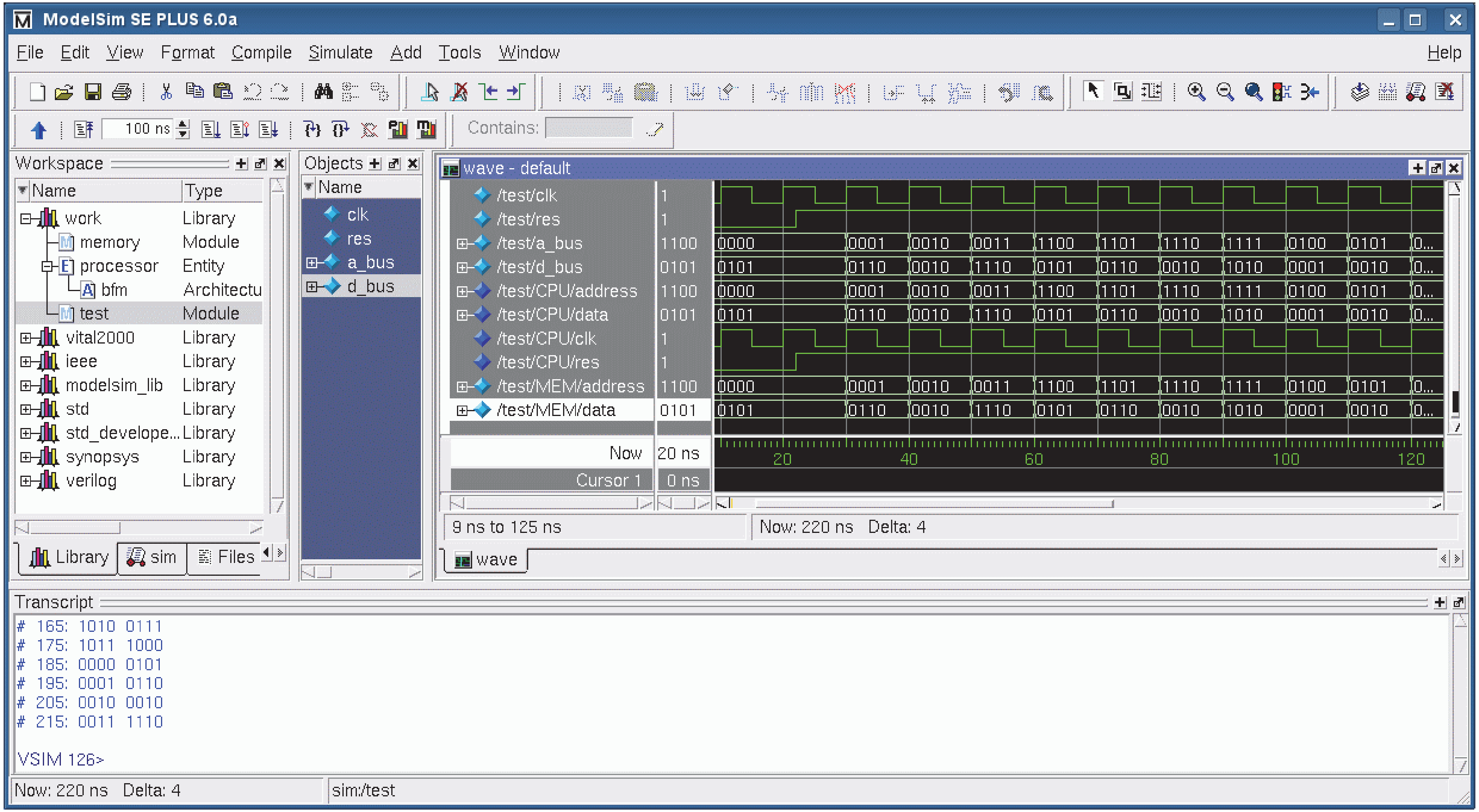
```
// Test-bench and bus monitor
`timescale 1 ns / 1 ns

module test;

  reg clk, res;
  wire [3:0] a_bus, d_bus;

  initial clk=1;
  always #5 clk=!clk;
  initial begin res=0; #22 res=1; end

  processor CPU ( a_bus, d_bus, clk, res );
  memory MEM ( a_bus, d_bus );

  always @(negedge clk) // bus monitor
    $display("%4t:  %b  %b",$time,a_bus,d_bus);
endmodule
```

Waveforms


wave - default
File Edit View Insert Format Tools Window
VHDL
/test/clk 1
/test/res 1
/test/a_bus 1100
/test/d_bus 0101
/test/cpu/address 1100
/test/cpu/data 0101
/test/cpu/clk 1
/test/cpu/res 1
/test/mem/address 1100
/test/mem/data 0101
0000 0001 0010 0011 1100 1101 1110 1111 0100 0101 0110 0111 1000 1001 1010 1011 0000 0001 0010 0011
0101 0110 0010 1110 0101 0110 0010 1010 0001 0010 0011 0100 0101 0110 0111 1000 0101 0110 0010 1110
wave - default
File Edit View Insert Format Tools Window
Verilog
/test/clk 1
/test/res 1
/test/a_bus 1100
/test/d_bus 0101
/test/CPU/address 1100
/test/CPU/data 0101
/test/CPU/clk St1
/test/CPU/res St1
/test/MEM/address 1100
/test/MEM/data 0101
0000 0001 0010 0011 1100 1101 1110 1111 0100 0101 0110 0111 1000 1001 1010 1011 0000 0001 0010 0011
0101 0110 0010 1110 0101 0110 0010 1010 0001 0010 0011 0100 0101 0110 0111 1000 0101 0110 0010 1110
Now 20 ns
Cursor 1 0 ns
40 80 120 160 200
0 ns
0 ns to 231 ns
Now: 220 ns Delta: 4

TTÜ1918

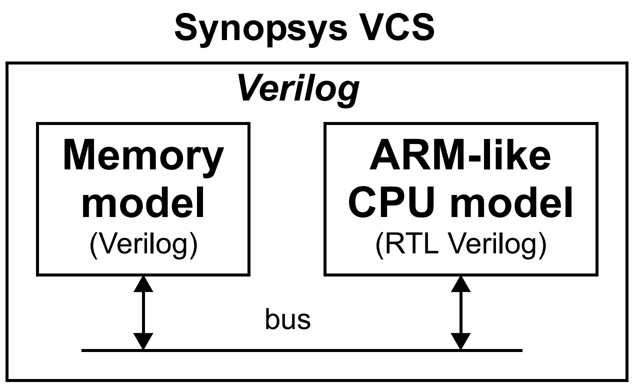
Co-simulating VHDL & Verilog



- **CPU in VHDL ; memory & testbench in Verilog**

Co-simulation example #2 – CPU & software

- Power consumption analysis of ARM-like processor
- Applications written in C
- Trimaran cross-compiler
- The main problem –
  are the applications running correctly?

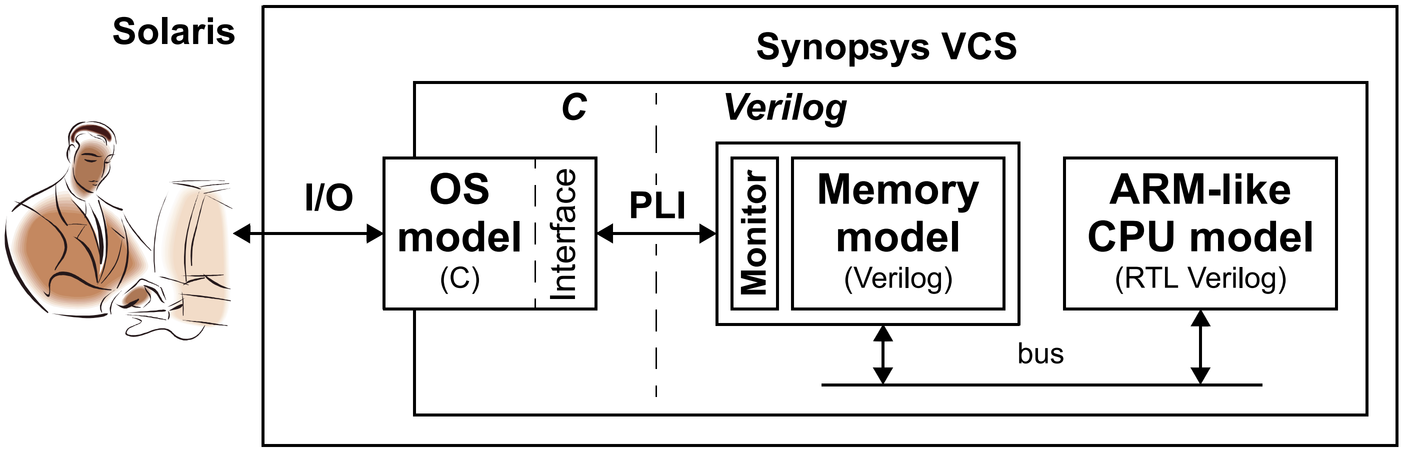


- An automated setup is needed – compiler and linker, plus OS kernel

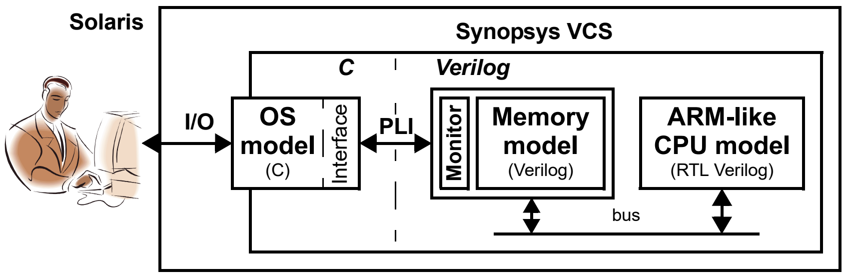
- K. Puttaswamy, K.-W. Choi, J. C. Park, V. Mooney, A. Chatterjee, P. Ellervee, "System Level Power-Performance Trade-Offs in Embedded Systems Using Voltage and Frequency Scaling of Off-chip Buses and Memory." The 15th International Symposium on System Synthesis (ISSS'2002), pp.225-230, Kyoto, Japan, Oct. 2002.

Co-simulation Example #2

- Memory mapped I/O like interfacing
  - monitor in Verilog to track memory writings
- Additional application SW (~3000 lines of C & assembly code)
  - scripts for compiler and linker to build the right memory mapping
  - OS kernel – I/O routines in C, boot-strap & system call in ARM assembly code
- Additional simulator SW (~750 lines of C code)
  - OS kernel system calls <--> Solaris I/O routines



Co-simulation Example #2



CPU bootstrap code (assembler)

```
    .section .boot, "ax"
    .align   3
    .global  __boot__
    .type    __boot__,%function
__boot__:    @ Reset
    b        .start
    b        .error
    ...
    .section .syscall, "aw"
    .align   3
    .global  __syscall_data__
    .type    __syscall_data__,%object
__syscall_data__:
    .word    0, 0, 0, 0, 0, 0, 0, errno
    .section .text
    .align   3
.start:
    mov      R0, #16
    msr      cpsr_all, R0
    mov      sp, #0x020000
    ...
    bl       main        @ "main();"
```

Memory access monitor (Verilog)

```
// Monitor
reg halt_cmd;
always @(posedge MMnWR)
  begin
    repeat (3) @(negedge GCLK);
    halt_cmd=0;  $syscall(halt_cmd);
    if (halt_cmd!=0) #10 $finish;
  end
```

Co-simulation Example #2

Function “fputc” (C)

```
int fputc(int c, FILE *stream)
{
  __syscall_data__ [0] = __SYSCALL_STDIO_FPUTC;
  __syscall_data__ [1] = c;
  __syscall_data__ [2] = (unsigned int)stream;
  return __syscall_ioproc__();
}
```

Memory access (assembler)

```
    .align   3
    .global  __syscall_ioproc__
    .type    __syscall_ioproc__,%function
__syscall_ioproc__:
    ldr      r0, .syscall.errno        @ flush & invalidate ”errno”
    stcl     p15, c0, [r0, #0]
    mcr      p15, 0, r0, c7, c6, 1
    nop                                @ a flush/invalidate problem?!
    ldr      r0, .syscall.data         @ location of parameters
    stcl     p15, c0, [r0, #0]         @ flush & invalidate ...
    mcr      p15, 0, r0, c7, c6, 1     @ ... cache-line (section .syscall)
    ldr      r0, [r0, #4]              @ return code
    mov      pc, lr
```

TTÜ1918

Co-simulation Example #2

```
int ReadMemory(const int addr) {
  int i,wd,value=0;
  for (i=0;i<SYSMEM_COUNT;i++) {
    wd=acc_getmem_int(mem[i],addr/SYSMEM_BYTES,SYSMEM_WD_BEG,SYSMEM_WD_LEN);
    value=(value<<SYSMEM_BITS)|(SYSMEM_MASK&wd);
  }
  return value;
}
static int SysCall_fputc(void) {
  FILE *fp; int c,ret;
  if ((fp=FilePointer(ReadMemory(syscall_addr+2*SYSMEM_BYTES),STREAM_WRITE))==NULL)
    { pli_errno=errno; return EOF; }
  c=ReadMemory(syscall_addr+SYSMEM_BYTES);
  ret=fprintf(fp,"%c",c); fflush(fp); pli_errno=errno; return ret==1?c:EOF;
}
void syscall_pli() {
  int exit_code,return_code=0;   unsigned int op_code;
  /* Setting parameters */
  DesignTimeScale();   syscall_addr=SYSCALL_ADDR;   SetUpMemory();
  ...
  op_code=ReadMemory(syscall_addr);                    /* Executing the operation */
  switch (op_code) {
  case __SYSCALL_NOP: return;
  case __SYSCALL_STDIO_FPUTC:   return_code=SysCall_fputc(); break;    /* "stdio" f-ns */
  ...
  }
  WriteMemory (pli_errno_addr, pli_errno);
  WriteMemory (syscall_addr+SYSMEM_BYTES, return_code);
  WriteMemory (syscall_addr, __SYSCALL_NOP);
}
```