Embedded Systems Module 6

formlabs

What is Rapid Prototyping?

- *Rapid prototyping is the fast fabrication of a physical part, model or assembly using 3D computer aided design (CAD). Rapid prototyping (RP) includes a variety of manufacturing technologies, although most utilize layered additive manufacturing. However, other technologies used for RP include high-speed machining, casting, moulding and extruding.*

What is Emulation?

- *Emulation is termed as the ability of a computer programme to replicate (or imitate) another programme or system within an electronic device.*
- *Emulator is software or hardware that allows one computer system (host) to function like another computer system. Usually, it allows the host machine to run software or use guest system-designed peripheral devices.*

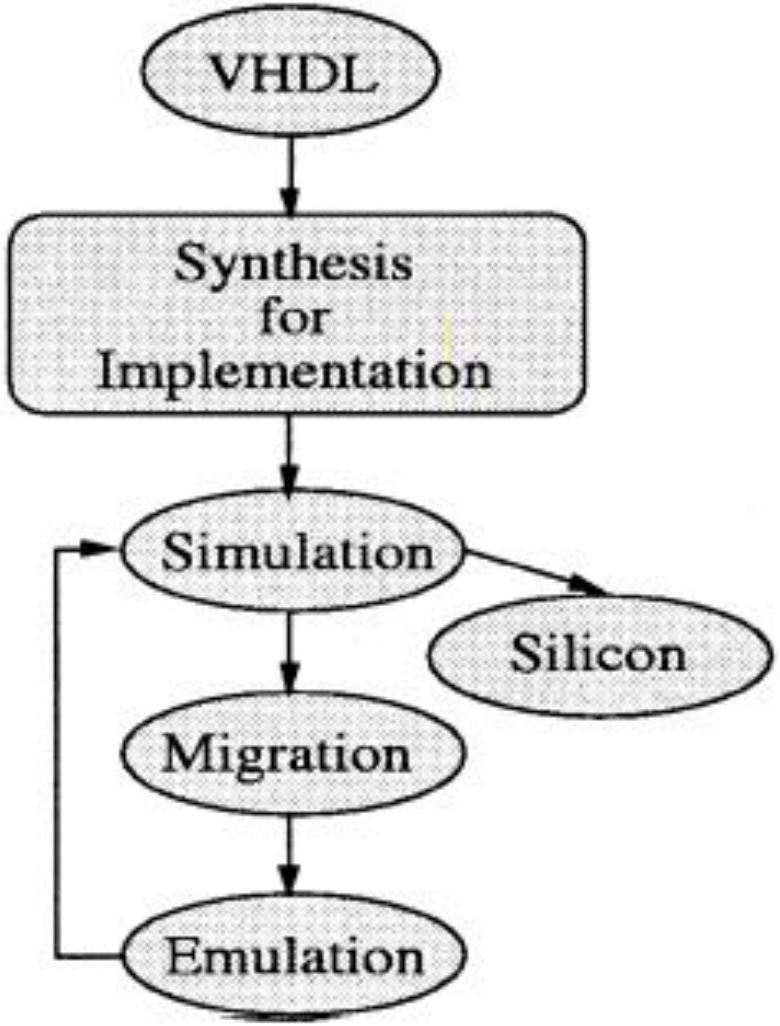
Prototyping and Emulation Techniques

- *Validation in hardware/software co-design has to solve two major problems. **First**, appropriate means for software and hardware validation are needed; **second**, these means must be combined for integrated system validation*
- *Since validation methods for software are well known, the main effort is spent on hardware validation methods and the integration of both techniques. Today, the main approaches for hardware validation are simulation, emulation and formal verification Simulation provides all abstraction levels from the layout level up to the behavioral level.*

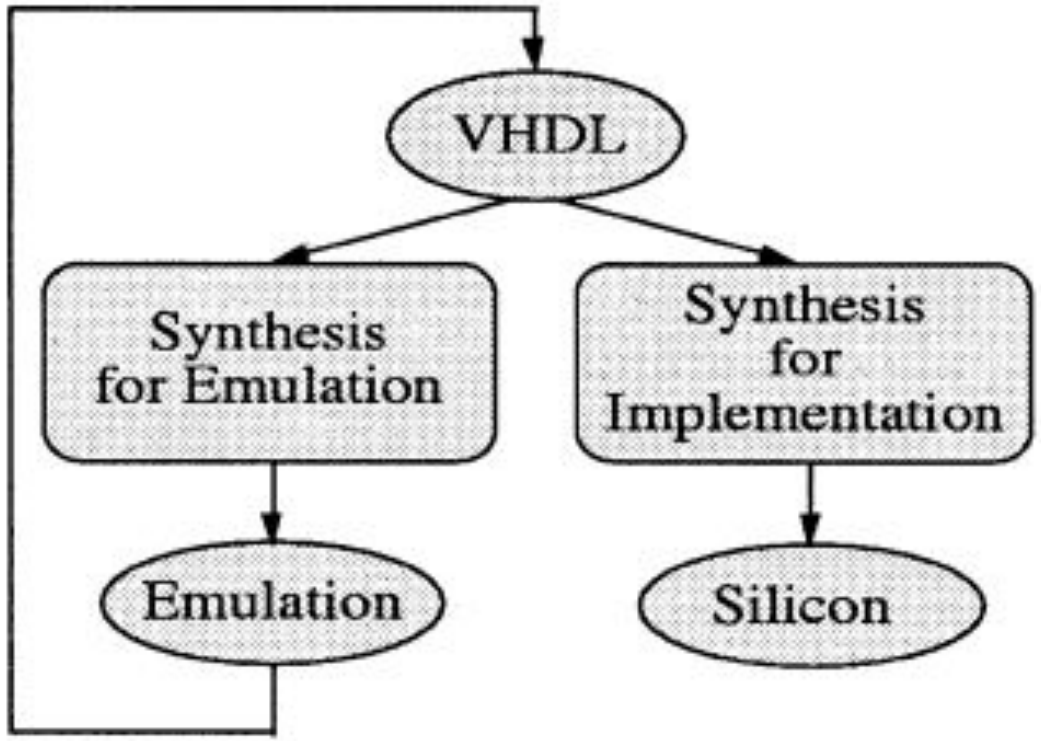
- *Formal verification of behavioral circuit specifications has up to now only limited relevance, since the specifications which can be processed are only small and can only use a very simple subset of VHDL.*
- *All these validation and verification are harder to implement during hardware/software co-design: low-level validation methods are not really useful, since hardware/software co-design aims at automatic hardware and software design starting from behavioral specifications*

Different design flow integrations for emulation

Alternative 1:



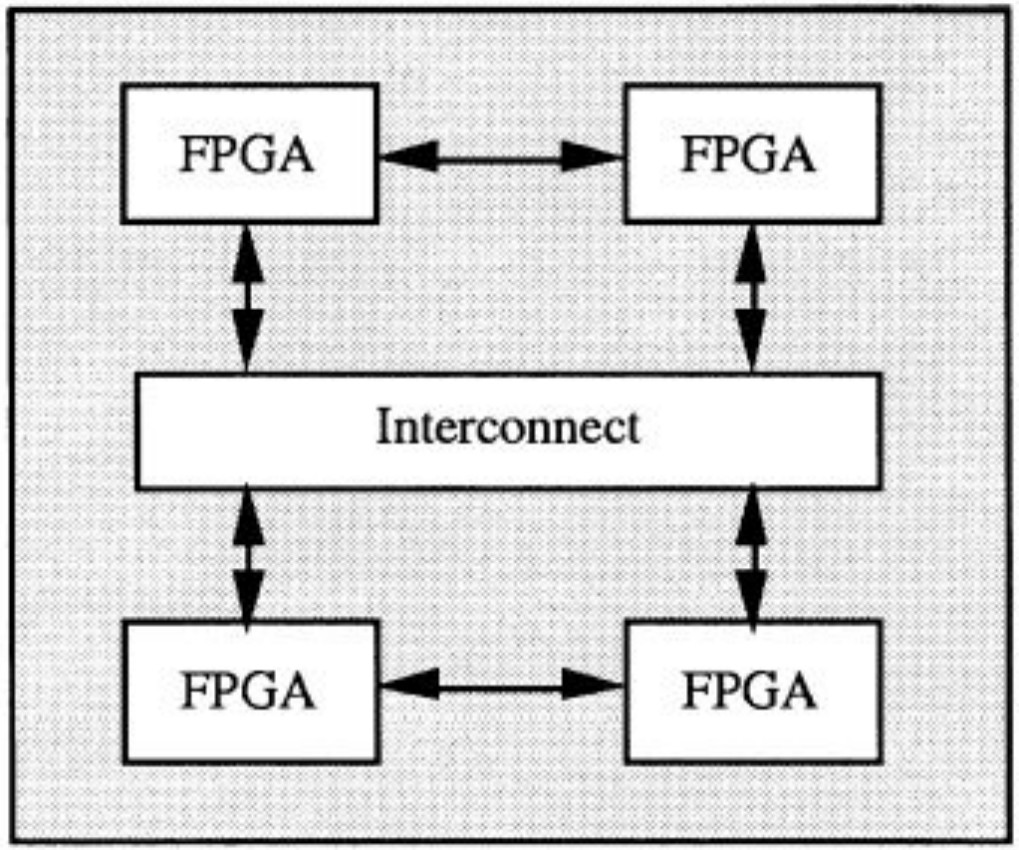
Alternative 2:



- *Figure demonstrates these varying design flows. Alternative 1 is today still very common. The cycle between simulation and emulation is passed for so long until no errors occur anymore.*
- *This is followed by a translation into the final ASIC implementation. The disadvantage of this alternative is that the migration which is required by emulation results in an implementation which is different from the subsequent translation into an ASIC. The ensuing validation problem is to be solved by an increased application of alternative 2.*

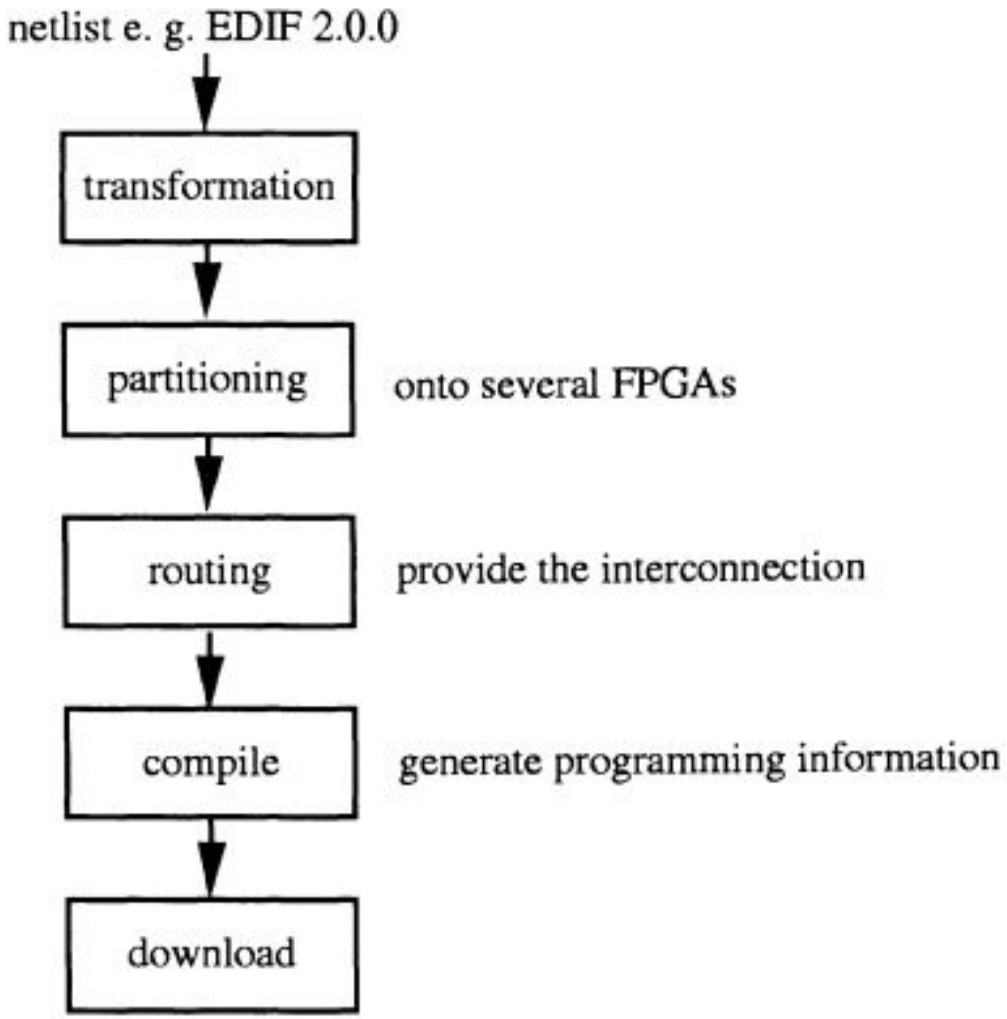
- *Here, a synthesis for emulation is implemented, for example on the basis of a gate level or a future RT level VHDL description. Accordingly, errors bring on changes in the VHDL source. This identical VHDL source file is then also converted by synthesis into a corresponding ASIC.*
- *Postulating an identical VHDL subset with identical semantics enables a sign-off on the basis of the VHDL source code.*
- *In the future, alternative 2 will support a RT level sign-off by means of a more abstract description on RT level.*

FPGA-based emulation



- *Just as individual emulation systems differ with respect to their use of FPGAs, there are differences between emulation systems on the market concerning their interconnection structure.*
- *Below concrete examples for each class are provided:*
- *Some systems also use programmable gate arrays for their interconnection structure, others take recourse to special interconnect switches (ex. Aptix), and there are some which apply especially developed custom interconnect chips for the connection of individual FPGAs*

The FPGA configuration process



Software fault injection

- *Software fault injection (SFI) denotes the artificial insertion—injection—of faults and error states into a running software system. It can be applied beyond the limits of formal verification methods because it does not assume a complete formal specification of the system under test.*
- *The experimental approach of SFI can be implemented efficiently and with little intrusiveness. In the simplest case, SFI can confront an interface with randomly generated values.*

Trigger mechanism of Software fault injection tool

- *Any fault injection tool relies on a trigger mechanism that causes the artificially generated fault or error to be inserted into normal program execution:*

- ***Time-based:*** *fault injection takes place at predetermined time intervals.*
- ***Location-based:*** *faulty values are written into predefined memory locations.*
- ***Execution-driven:*** *fault injection occurs dynamically, depending on the control flow.*

Different injection times of Software fault injection

- ***Before runtime:*** *the program is modified upfront, for instance, by using source code mutation to add faults (software bugs) to the code.*
- ***During runtime:*** *faults are injected during program execution.*
- *At* ***the loading time of*** *external components: here, injection triggers may be the dynamic binding of external libraries or the adding of other dependencies during runtime.*

Fault Simulation

- *We use fault simulation after we have completed logic simulation to see what happens in a design when we deliberately introduce faults.*
- *In a production test we only have access to the package pins—the primary inputs (PIs) and primary outputs (POs).*
- *As each fault is inserted, the fault simulator runs our test program.*
- *If the fault simulation shows that the POs of the faulty circuit are different than the PIs of the good circuit at any strobe time, then we have a detected fault ; otherwise we have an undetected fault .*

Fault coverage

- *The list of fault origins is collected in a file and as the faults are inserted and simulated, the results are recorded and the faults are marked according to the result. At the end of fault simulation we can find the fault coverage ,*

fault coverage = detected faults / detectable faults.

- *Fault coverage refers to the percentage of some type of fault*

Serial Fault Simulation

- *Serial fault simulation is the simplest fault-simulation algorithm. We simulate two copies of the circuit, the first copy is a good circuit.*
- *We then pick a fault and insert it into the faulty circuit. In test terminology, the circuits are called machines , so the two copies are a good machine and a faulty machine .*
- *We shall continue to use the term circuit here to show the similarity between logic and fault simulation (the simulators are often the same program used in different modes).*
- *We then repeat the process, simulating one faulty circuit at a time. Serial simulation is slow and is impractical for large ASICs.*

Parallel Fault Simulation

- *Parallel fault simulation takes advantage of multiple bits of the words in computer memory. In the simplest case we need only one bit to represent either a '1' or '0' for each node in the circuit.*
- *In a computer that uses a 32-bit word memory we can simulate a set of 32 copies of the circuit at the same time. One copy is the good circuit, and we insert different faults into the other copies.*
- *When we need to perform a logic operation, to model an AND gate for example, we can perform the operation across all bits in the word simultaneously.*
- *In this case, using one bit per node on a 32-bit machine, we would expect parallel fault simulation to be about 32 times faster than serial simulation.*
- *The number of bits per node that we need in order to simulate each circuit depends on the number of states in the logic system we are using. Thus, if we use a four-state system with '1' , '0' , 'X' (unknown), and 'Z' (high-impedance) states, we need two bits per node.*

Concurrent Fault Simulation

- *Concurrent fault simulation is the most widely used fault-simulation algorithm and takes advantage of the fact that a fault does not affect the whole circuit.*
- *Thus we do not need to simulate the whole circuit for each new fault. In concurrent simulation we first completely simulate the good circuit.*
- *We then inject a fault and re-simulate a copy of only that part of the circuit that behaves differently (this is the diverged circuit). For example, if the fault is in an inverter that is at a primary output, only the inverter needs to be simulated—we can remove everything preceding the inverter.*

Concurrent Fault Simulation

- *Keeping track of exactly which parts of the circuit need to be diverged for each new fault is complicated, but the savings in memory and processing that result allow hundreds of faults to be simulated concurrently.*
- *Concurrent simulation is split into several chunks, you can usually control how many faults (usually around 100) are simulated in each chunk or pass .*
- *Each pass thus consists of a series of test cycles. Every circuit has a unique fault-activity signature that governs the divergence that occurs with different test vectors.*
- *Thus every circuit has a different optimum setting for faults per pass . Too few faults per pass will not use resources efficiently. Too many faults per pass will overflow the memory.*

Nondeterministic Fault Simulation

- *Serial, parallel, and concurrent fault-simulation algorithms are forms of deterministic fault simulation . In each of these algorithms we use a set of test vectors to simulate a circuit and discover which faults we can detect.*
- *If the fault coverage is inadequate, we modify the test vectors and repeat the fault simulation. This is a very time-consuming process.*
- *As an alternative we give up trying to simulate every possible fault and instead, using probabilistic fault simulation , we simulate a subset or sample of the faults and extrapolate fault coverage from the sample.*
- *In statistical fault simulation we perform a fault-free simulation and use the results to predict fault coverage. This is done by computing measures of observability and controllability at every node.*

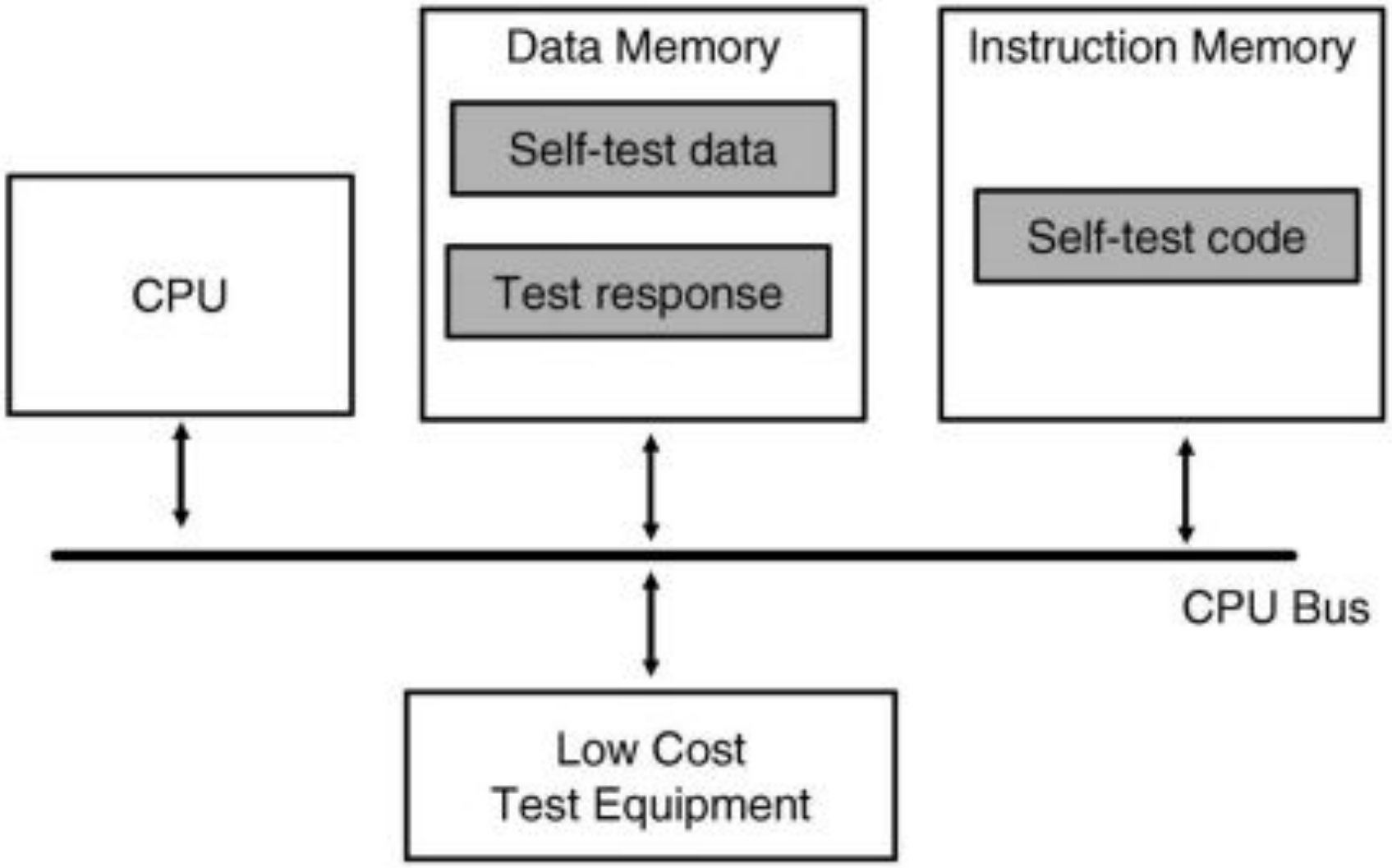
Nondeterministic Fault Simulation

- *We know that a node is not stuck if we can make the node toggle—that is, change from a '0' to '1' or vice versa. A toggle test checks which nodes toggle as a result of applying test vectors and gives a statistical estimate of vector quality , a measure of faults detected per test vector.*
- *There is a strong correlation between high-quality test vectors, the vectors that will detect most faults, and the test vectors that have the highest toggle coverage . Testing for nodes toggling simply requires a single logic simulation that is much faster than complete fault simulation.*
- *We can obtain a considerable improvement in fault simulation speed by putting the high-quality test vectors at the beginning of the simulation. The sooner we can detect faults and eliminate them from having to be considered in each simulation, the faster the simulation will progress.*
- *We take the same approach when running a production test and initially order the test vectors by their contribution to fault coverage. This assumes that all faults are equally likely. Test engineers can then modify the test program if they discover vectors late in the test program that are efficient in detecting faulty chips.*

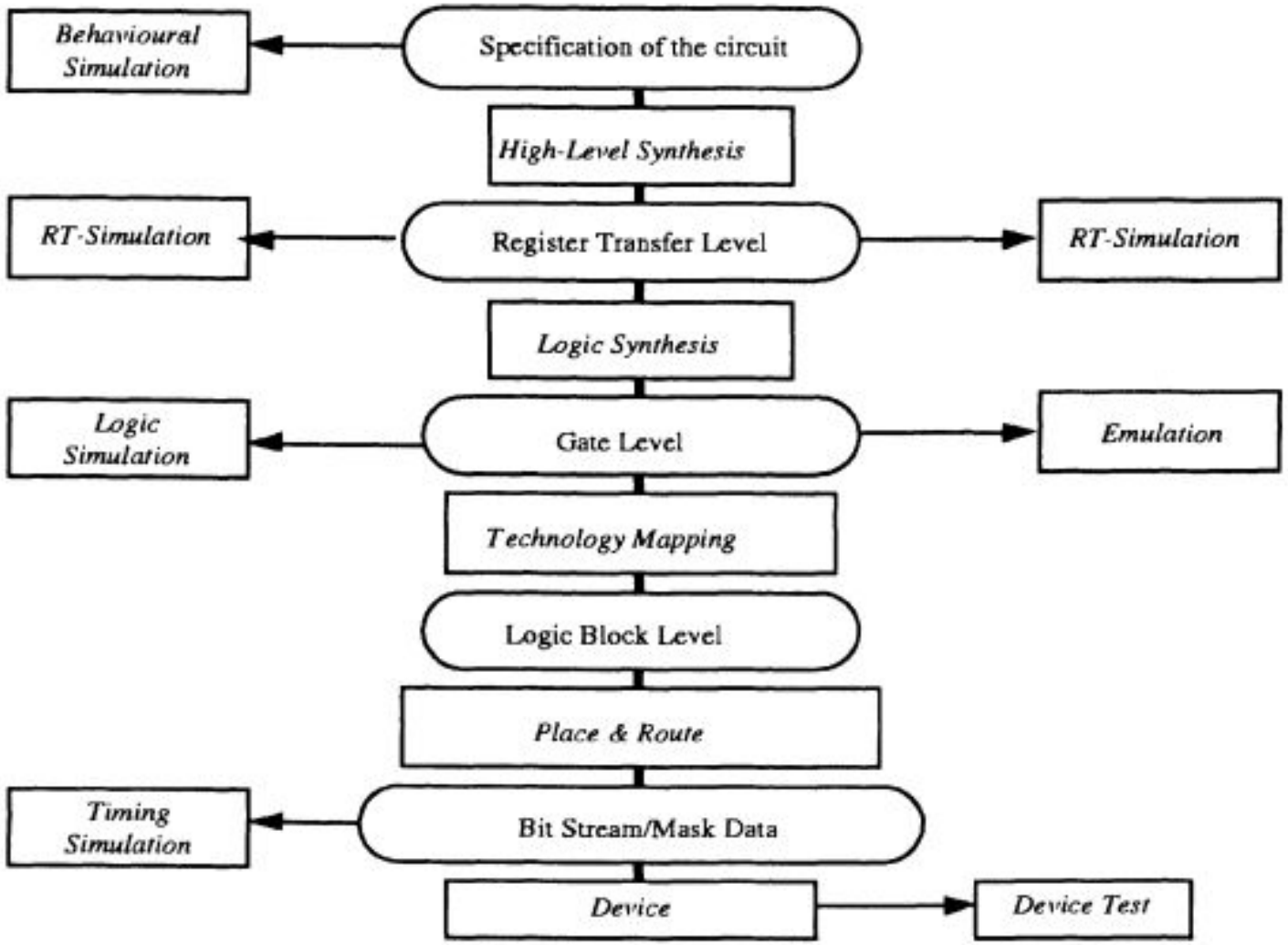
Software-based self-testing concept outline

- *Self-test routines and data are downloaded into instruction and data memories, respectively, from a low-speed, low-cost external mechanism. Subsequently, these self-test routines are executed at the processor actual speed (at-speed testing) and test responses are stored back in the on-chip RAM.*
- *These test responses may be in a compacted (self-test signatures) or uncompacted form and can be unloaded by the low-cost external ATE during manufacturing testing.*
- *Since today's microprocessors have a reasonable amount of on-chip cache, execution from the on-chip cache is considered a further improvement when targeting low cost testers, provided that a cache-loader mechanism exists to load the test program and unload the test response.*
- *As an alternative, self-test routines may be executed from an on-chip ROM dedicated to this task when periodic online testing is performed for the device*

Software-based self-testing concept outline



Rapid prototyping with design flow diagram of synthesis and validation



- *Flow diagram describes the necessary steps of synthesis and validation for the design of integrated circuits.*
- *There are four methods to reach the goal of "first-time right silicon":*
    - *specification on high levels of abstraction followed by automatic synthesis;*
    - *simulation on various levels;*
    - *formal verification;*
    - *prototyping and emulation.*

- *In the last few years the growing significance of synthesis has become apparent, that is synthesis in a general sense and-more specifically synthesis on higher levels of abstraction, such as RT level synthesis and the so-called high-level synthesis, i.e. synthesis from behavioral descriptions.*
- *The increasing number of available commercial tools for RT and high-level synthesis indicates that the abstraction level of the design entry will increase to higher levels in the near future.*

- *Especially with respect to hardware/software co-design high level synthesis gathers momentum. Only by integrating high-level synthesis in the hardware software co-design cycle real hardware/software co-design will be possible.*
- *The investigation of various possibilities with different hardware/software trade-offs is only sensible if the different hardware partitions can be implemented as fast as software can be compiled into machine code.*
- *High-level synthesis is, therefore, the enabling technology for hardware software co-design.*