Synthesis Lectures on
Digital Circuits & Systems

SYNTHESIS
COLLECTION OF TECHNOLOGY

Khaled Salah Mohamed

Heterogeneous SoC Design and Verification

HW/SW Co-Exploration, Co-Design, Co-Verification and Co-Debugging

Springer

Synthesis Lectures on Digital Circuits & Systems

Series Editor

Mitchell A. Thornton, Southern Methodist University, Dallas, USA

This series includes titles of interest to students, professionals, and researchers in the area of design and analysis of digital circuits and systems. Each Lecture is self-contained and focuses on the background information required to understand the subject matter and practical case studies that illustrate applications. The format of a Lecture is structured such that each will be devoted to a specific topic in digital circuits and systems rather than a larger overview of several topics such as that found in a comprehensive handbook. The Lectures cover both well-established areas as well as newly developed or emerging material in digital circuits and systems design and analysis.

Khaled Salah Mohamed

Heterogeneous SoC Design and Verification

HW/SW Co-Exploration, Co-Design, Co-Verification and Co-Debugging

Springer

Khaled Salah Mohamed
Siemens Digital Industries Software
Fremont, CA, USA

ISSN 1932-3166 ISSN 1932-3174 (electronic)
Synthesis Lectures on Digital Circuits & Systems
ISBN 978-3-031-56151-1 ISBN 978-3-031-56152-8 (eBook)
https://doi.org/10.1007/978-3-031-56152-8

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

To My Beloved Wife, Marwa
In every chapter of my life, you are the most beautiful story. This book is a testament to the love and inspiration you bring into my world. Your unwavering support and encouragement have filled the pages of our journey with joy, laughter, and countless cherished moments. Thank you for being the muse to my words, the melody to my heart, and the love that completes every sentence of my life. With all my love.

Contents

1 An Introduction to Heterogeneous SoC Design and Verification "A Conceptual-Level" 1
1.1 Introduction 1
1.1.1 Introduction: Digital Design Flow 2
1.1.2 Introduction: Software 6
1.1.3 Introduction: How to Develop HW/SW Together? 12
1.1.4 Introduction: How to Run/Communicate HW/SW Together? 15
1.1.5 Introduction: SoC Case Studies 17
1.1.6 Introduction: IP Management 23
1.1.7 Conclusions 23
References 24

2 SoC Design Methodologies 27
2.1 Introduction: Hardware Design Methodologies 27
2.1.1 FPGA-Centric SoC Design 29
2.1.2 Processor-Centric SoC Design 35
2.1.3 HLS-Centric SoC Design 45
2.1.4 Data-Centric/Memory-Centric SoC Design 47
2.1.5 Hardware Accelerators-Centric SoC Design 51
2.1.6 ASIC-Based SoC Design 53
2.1.7 PCB-Based SoC Design 55
2.1.8 Application-Centric SoC Design 57
2.1.9 Conclusions 58
References 59

3 HW/SW Co-Exploration and Co-Design 61
3.1 HW/SW Partitioning 64
3.1.1 Design Space Exploration (DSE): HW/SW Co-Exploration Tradeoff 66
3.1.2 HW/SW Interfacing 72
3.1.3 Task Graph and Cost Function: Problem Definition 72

3.1.4 Petri Nets ... 73
3.1.5 UML Diagrams ... 74
3.1.6 Optimization Techniques for Manual HW/SW Partitioning: ML-Based Approach ... 74
3.2 HW/SW Communication ... 76
3.2.1 SCEMI ... 77
3.2.2 DPI-C ... 79
3.3 HW/SW Synchronization ... 79
3.3.1 Semaphore ... 79
3.3.2 Handshake ... 80
3.3.3 Bus Locking ... 80
3.3.4 Mutex ... 80
3.3.5 Interrupt ... 81
3.3.6 FIFO ... 81
3.4 HW/SW Co-Design Metrics ... 81
3.5 Conclusions ... 83
References ... 83

4 Pre-Silicon Verification and Post-Silicon Validation Methodologies ... 85
4.1 Introduction ... 85
4.2 Pre-Silicon Verification ... 87
4.2.1 Virtual Prototyping ... 87
4.2.2 FPGA Prototyping ... 90
4.2.3 Physical Prototyping ... 91
4.2.4 Emulation-Based Verification: Using FPGAs to Simulate ASICs ... 91
4.2.5 Simulation-Based Verification ... 92
4.2.6 Functional Verification ... 93
4.2.7 Directed-Testing Verification ... 94
4.2.8 Coverage-Driven Verification ... 94
4.2.9 UVM ... 96
4.2.10 Formal Verification ... 102
4.2.11 Verification IP Versus Design IP ... 109
4.2.12 Power-Aware Verification ... 110
4.2.13 Timing Verification ... 111
4.2.14 Safety Verification: Fault Simulation ... 112
4.2.15 Performance Verification: *Meeting Bandwidth and Latency* ... 114
4.2.16 Verification Challenges ... 114
4.2.17 How to Verify the Verification ... 114
4.2.18 Regression Testing Techniques ... 115
4.2.19 IP-XACT Methodology ... 115
4.2.20 Portable Stimulus Standard: Graph-Based Testing ... 117

4.3 Post-Silicon Validation . . . 119
4.3.1 DFT Verification: Gate-Level Simulations . . . 120
4.3.2 Physical Probing/Trace-Based Technique . . . 125
4.3.3 Synthesizable Assertions for Post-Silicon Debug . . . 125
4.4 Full-Chip SoC Verification: SoC Integration Testing . . . 126
4.5 *Data-Driven* Verification: AI-Powered Verification . . . 127
4.6 Conclusions . . . 127
References . . . 128

5 HW/SW Co-Verification and Co-Debugging . . . 133
5.1 Co-Simulation . . . 134
5.2 Co-Emulation . . . 138
5.3 Co-Debugging . . . 141
5.3.1 GDB: A GNU Debugger . . . 144
5.3.2 Arm DS5 Debugger . . . 145
5.3.3 MIPS SP55 Debugger . . . 146
5.3.4 Lauterbach Trace32 Debugger . . . 146
5.3.5 OpenOCD Debugger . . . 147
5.3.6 Codelink . . . 148
5.4 HW/SW Co-Verification Metrics . . . 148
5.5 Conclusions . . . 149
References . . . 150

6 HW/SW Co-Optimization and Co-Protection . . . 153
6.1 HW/SW Co-Optimization . . . 153
6.1.1 Software Optimization . . . 155
6.1.2 Hardware Optimization . . . 157
6.1.3 HW/SW Compilation Time Optimization . . . 158
6.1.4 HW Maximum Frequency Optimization . . . 160
6.1.5 HW/SW Power Optimization . . . 161
6.1.6 HW/SW Speed Optimization . . . 162
6.1.7 HW Area Optimization . . . 163
6.2 HW/SW Co-Protection . . . 164
6.2.1 Hardware Oriented Security and Trust . . . 166
6.3 Conclusions . . . 167
References . . . 168

1 An Introduction to Heterogeneous SoC Design and Verification "A Conceptual-Level"

1.1 Introduction

System-on-chip (SoC) and intellectual property (IP) era started in 1995 by integration of more than 100M transistor per chip. Motivation behind SoC is to increase performance and to decrease costs and time-to-market. A typical SoC contains hardware and software. Hardware (HW) may provide higher performance over software (SW) but may be less **reliable** than software. On the contrary, software can be **patched** more easily. So, there is always a tradeoff between HW and SW. Any design can be implemented as pure HW, pure SW or HW/SW Co-Design (hybrid) or Silicon (Fig. 1.1). To model a system, transform your problem from English description or technical specifications to an algorithmic model. Then transform the algorithmic model to a FSM, flowchart, or an architecture. Designing a computer is different from designing a phone or a data-center as they have different platforms and different design requirements/constrains (area, power, and performance). However, there are many common fundamental concepts. Table 1.1 summarizes the solution hierarchy for any VLSI-based problem. The overall design flow of hybrid SoC design is shown in Fig. 1.2. To conquer the complexity of SoC, a pre-designed components are used (IP reuse). Hardware IP cores have emerged as an integral part of modern SoC designs. IP cores are pre-designed and pre-verified complex functional blocks. Based on their properties, IP cores can be distinguished into three types of cores: hard, firm, and soft as depicted in Table 1.2. Where Soft-cores are architectural modules which are synthesizable and offer the highest degree of modification flexibility, Firm-cores are delivered as a mixture of RTL code and a technology-dependent netlist, and are synthesized with the rest of ASIC logic, and Hard-cores are mask and technology-dependent modules. The complete picture for electronic systems is described in Figs. 1.3 and 1.4. For System with multiple SoC's, globally asynchronous locally synchronous (GALS) interconnect concept is used to simplify its design (Fig. 1.5). GALS aims at filling the gap between the purely

K. S. Mohamed, *Heterogeneous SoC Design and Verification*, Synthesis Lectures on Digital Circuits & Systems, https://doi.org/10.1007/978-3-031-56152-8_1

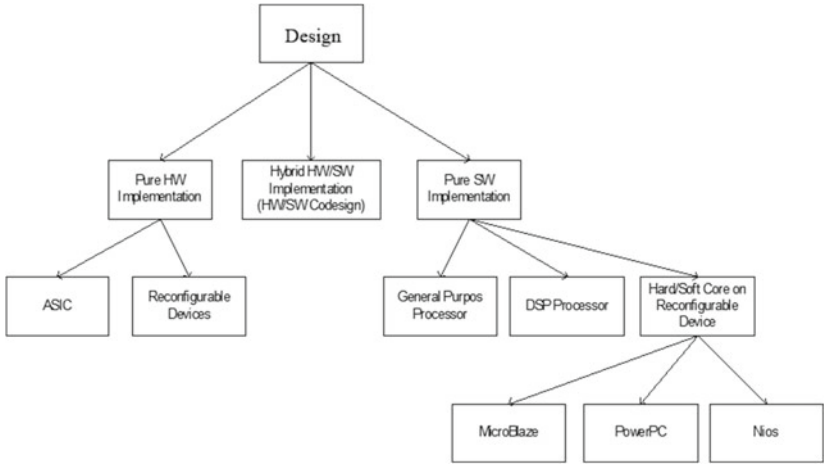


Fig. 1.1 Design implementation: alternatives

synchronous and asynchronous domain. A Full-stack SoC can boot full operating system kernels such as Linux with support for a broad range of applications [1–6].

Heterogeneous System-on-Chip (SoC) design refers to the integration of multiple types of processing elements, such as CPUs, GPUs, DSPs, FPGAs, and other specialized hardware accelerators, onto a single chip. The design of a heterogeneous SoC can be complex because it involves integrating different types of processing elements that have different architectures, performance characteristics, and communication interfaces. Verification is the process of ensuring that the design meets its functional and performance specifications. In the context of heterogeneous SoC design, verification involves checking that the different processing elements operate correctly and communicate with each other properly. This can involve simulating the behavior of the system, running test cases on prototypes or emulators, and using formal verification techniques to mathematically prove that the design is correct. The verification of a heterogeneous SoC design is challenging because of the complexity and diversity of the processing elements involved. It is often necessary to use a combination of simulation, emulation, and formal methods to verify the design thoroughly. In addition, the verification process must consider the power and thermal constraints of the system, as well as its reliability and security requirements [7–16].

1.1.1 Introduction: Digital Design Flow

During digital systems development, a digital design goes into multiple transformations, starting from the set of specifications to the final product. Each of these transformations corresponds a different description of the system, which is incrementally more detailed

Table 1.1 Solution hierarchy for any VLSI-based problem

Solution levels				
Level of Abstraction	SW	Choose a modeling language/Approach/Architecture (Pure SW/Pure HW/ SW-HW co-design) {based on design goals/requirements: performance, power, area, Security, reliability, safety}		
	SW/HW	Build the SW side	Application	
			OS, VM	
			ISA	
			Microarchitecture (processor architecture)	
			DRAM	
	RTL	Build the SW/HW interface		
		Build the HW side	Algorithmic level	Create an Algorithm
			Architecture level	Build a system-level based architecture
			Logic/gate level	Build a gate-level based architecture
	Silicon		Devices/transistors level	Build a device-level based architecture
			Layout/mask level	ASIC-level
			Electrons/physical/ technology level	Create a mathematical model for the problem according to the physics laws

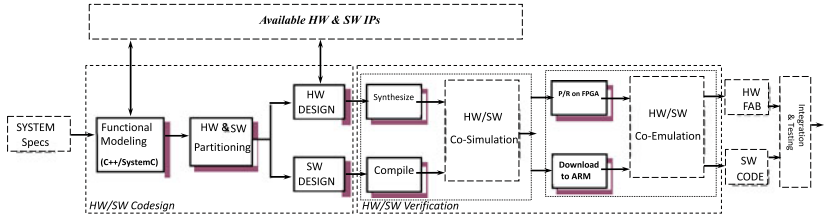


Fig. 1.2 The overall design flow of hybrid SoC design. Models are conceptual views of the system's functionality. Architectures are abstract views of the system's implementation

and more complex. The correctness of a digital circuit is a major consideration in the design of digital systems. Given the extremely high and increasing costs of manufacturing chips, the consequences of finding failures going unnoticed in system designs until after

Table 1.2 Classification of hardware IP

IP	Representation	Technology	Optimization	Reuse	Changes
Soft	RTL (HDL)	Independent (fabless level)	Low	Very high	Many
Firm	Gate level netlist	Independent	Medium	High	Some placement and routing
Hard	GDSII (layout)	Dependent (fab level)	Very high	Low	No

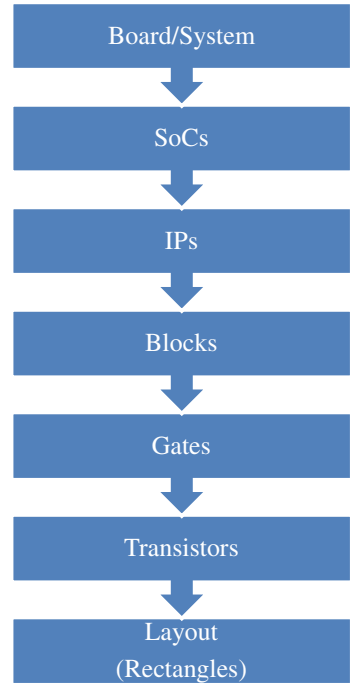


Fig. 1.3 Electronic systems level from board to transistors

the production phase are very expensive. So, functional specification is a major step in the digital design life cycle Fig. 1.6 shows a typical Digital IC design cycle.

The digital design flow life cycle can be summarized in the following steps: The functional design is the initial process of deriving a potential and realizable solution from this design specifications and requirements. During this phase, the architectural description is further refined: memory element and functional components of each model are designed using a Hardware Description Languages (HDL). RTL verification consists of acquiring a reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. The next design phase consists of the synthesis and

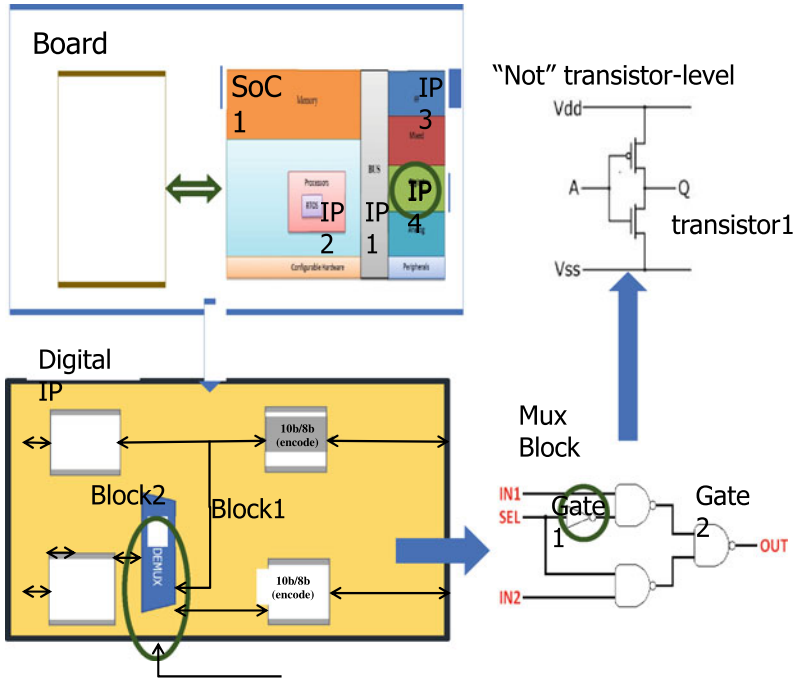


Fig. 1.4 Detailed electronic systems level, where a single board contains number of SoCs and each SoC consists of number of IPs, these IPs consists of number of blocks which consists of number of gates. Gates are consists of number of transistors [17]

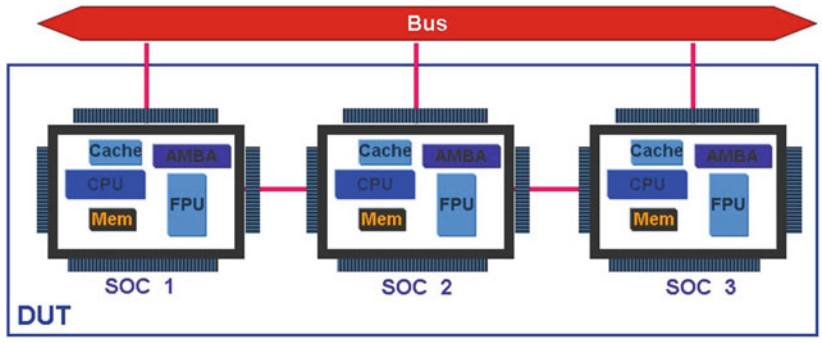


Fig. 1.5 System with Multiple SOC's. Synchronous modules on a chip communication asynchronously

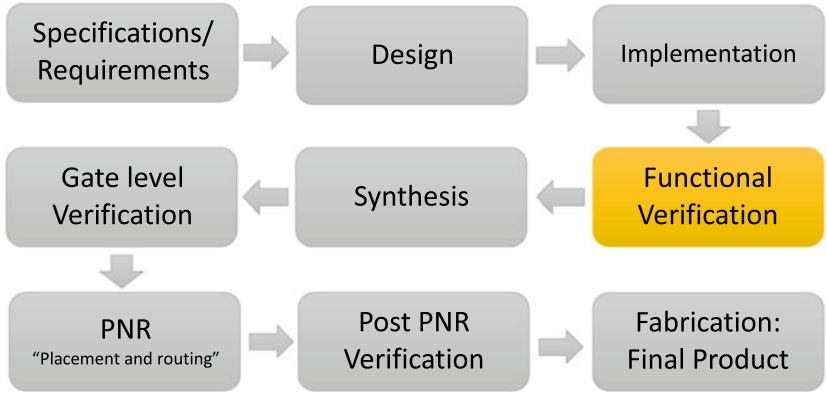


Fig. 1.6 Digital IC design cycle

optimization of the RTL design. The overall result of this phase is to generate a detailed model of the circuit which is optimized based on the design constraints. The objective of RTL versus gates verification, or equivalency checking, is to guarantee that no errors have been introduced during the synthesis phase. The result is a description of the circuit in terms of a geometrical layout used for the fabrication process. Finally, the design is fabricated, and the microchips are tested and packaged [13, 18–26].

Y chart describes the different levels of the digital system development. In each level of the design, the verification step is mandatory to maintain its compliance to the required specification as follows (Fig. 1.7):

- *Architecture level*: system verification for the functional model which is high level language model.
- *RTL level*: RTL functional verification for the RTL model which is hardware description language design.
- *Logic level*: formal verification and post place and route verification is needed to ensure that the design synthesized correctly and there are no timing violations.
- *Circuit level*: Physical verification is performed here to ensure the design is working after adding the physical effects such as parasitic capacitances.

1.1.2 Introduction: Software

"Software is the soul of the machine." Greg Lavender. Virtualization is the abstraction of details. Algorithms and programming languages provide abstraction, too. Computer architecture involves mainly the instruction set architecture (ISA) design and microarchitecture design. An ISA defines everything a machine language programmer needs to

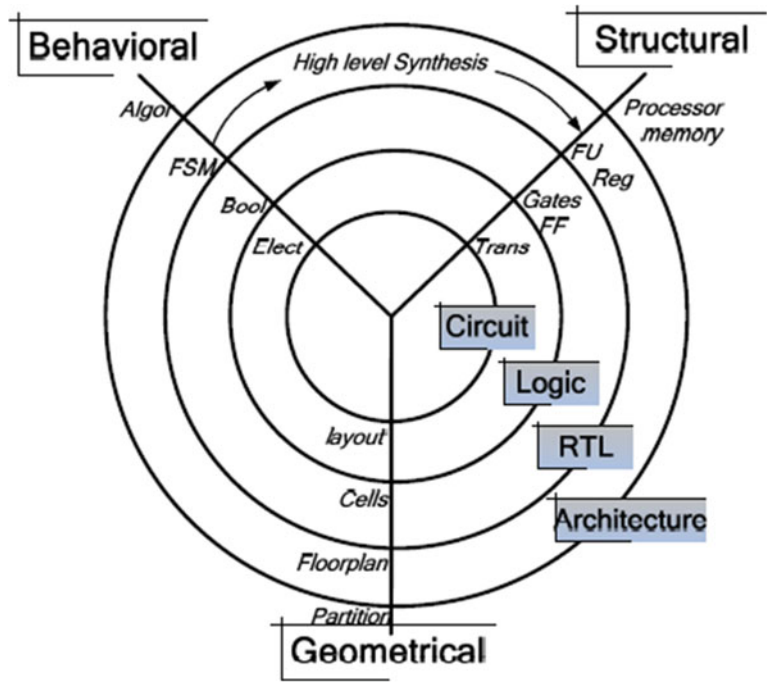


Fig. 1.7 Y chart which describes the different levels of the digital system development

know to program a computer. in general, ISAs define the supported data types, what state there is such as the main memory and registers and their semantics such as the memory consistency and addressing modes, the instruction set which is the set of machine instructions that comprises a computer's machine language, and the input/output model. ISA is an abstract model of a computer. An ISA permits multiple implementations that may vary in performance. the ISA serves as the interface between software and hardware. Software that has been written for an ISA can run on different implementations of the same ISA (Binary-code compatibility). Microarchitecture is the way a given instruction set architecture (ISA) is implemented in a particular processor. A given ISA may be implemented with different microarchitectures; implementations may vary due to different goals of a given design or due to shifts in technology [27, 28].

The typical design flow to convert software source code into instructions for a processor is shown in Fig. 1.8. A compiler or an assembler is used to convert source code into object code. An object code file contains opcodes /instructions and constants. A linker is used to combine several object code files into a single, stand-alone executable file. a loader program determines how the information in an executable file is organized in memory. The processor will fetch these instructions from memory and execute them on the processor datapath. In a RISC processor, the execution of a single instruction is split

in different stages, which are chained together as a pipeline. Each instruction operates on a set of registers contained within the processor. Processor registers are used as operands or as targets for the processor instructions, and for control. Each stage of a RISC pipeline takes one clock cycle to complete. A typical **RISC** pipeline has three or five stages. The five stages of the pipeline are called Instruction Fetch, Instruction Decode, Execute, Buffer, and Write-back:

- **Instruction Fetch**: The processor retrieves the instruction addressed by the program counter register from the instruction memory.
- **Instruction Decode**: The processor examines the instruction opcode.
- **Execute**: The processor executes the computational part of the instruction on a datapath.
- **Buffer**: the processor may access the data memory, for reading or for writing.
- **Write Back**: the processor registers are updated.

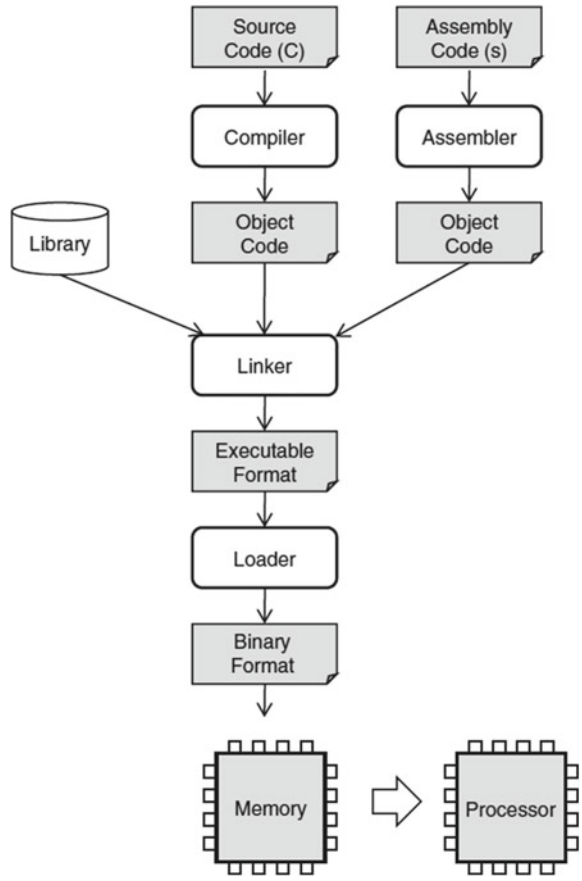


Fig. 1.8 Design flow of software source code

A **soft** microprocessor (or a soft core) is considered Register transfer logic (RTL) code that describes a specific design and capable of executing some sort of an instruction set. This code can then be synthesized into a net list and mapped onto a programmable logic such as FPGA. Unlike hard processors which are physically implemented as a structure in silicon. The main advantage of soft processors is the higher configurability, and adjustability, as all features are written in code and thus its instruction set could be easily extended, modified, and altered; on the other hand, increasing those capabilities will result in much more waste of resources and FPGA area, in addition to consuming more power while running at lower speeds as they are limited by fabric speed. Productivity increases with software; however, hardware is more efficient [29].

1.1.2.1 Software Ecosystem

A software ecosystem refers to a collection of software applications, tools, and services that interact with each other to deliver a comprehensive solution to users. In other words, it is a set of software products that work together to provide a complete solution for a particular task or domain. The ecosystem is characterized by the interdependence of various software products that interact with each other to deliver value to users. Software Ecosystem Components are summarized in Table 1.3. Here are some of the key components of a software ecosystem [30–34]:

1. Core Software Products: These are the primary software products that form the foundation of the ecosystem. They are often developed by the ecosystem owner or a major contributor and are responsible for providing the core functionality of the ecosystem. Examples of core software products include operating systems, databases, and development environments.
2. Add-on Products: These are additional software products that complement the core products by providing additional features or functionalities. They are often developed by third-party developers or vendors and are designed to work seamlessly with the core products. Examples of add-on products include plugins, extensions, and integrations.

Table 1.3 Software Ecosystem Components

Component	Example
OS	• Android • IOS • Windows • RTOS • Redhat
Kernel	Linux is a kernel. Redhat is OS which lunches linux kernel
Virtualization	VMware [35]
System Initialization	Booting

3. Services: These are the various services that are provided as part of the ecosystem. They can include services for hosting, deployment, maintenance, and support. Services are often provided by the ecosystem owner or third-party vendors and are designed to help users get the most out of the ecosystem.
4. Development Tools: These are the various tools that are used to develop software products for the ecosystem. They can include IDEs (Integrated Development Environments), SDKs (Software Development Kits), and APIs (Application Programming Interfaces). Development tools are essential for developers who want to create software products that integrate seamlessly with the ecosystem.
5. Marketplace: This is the platform where users can find and download software products, add-ons, and services that are part of the ecosystem. The marketplace is often managed by the ecosystem owner and provides a centralized location for users to discover and download new products and services.
6. Community: This is the group of users, developers, and contributors who are part of the ecosystem. The community is an essential component of the ecosystem as it provides support, feedback, and contributions to the development of new software products and services.

1.1.2.2 Software Versus Firmware

Firmware is a low-level software necessary for hardware to work with software. Firmware is located in read-only memory (ROM), serving as a bridge between the hardware and the software as it contains specific instruction sets that allow the hardware to interface with **higher-level software** like the operating system. Devices with firmware includes BIOS, TV Remote Controls, USB hard drives, Keyboard, and many more. In other words, firmware is a type of software that is permanently stored in a hardware device and controls its operation.

Software is a more general term that refers to a program that are used to perform a specific on a computer. It is typically stored on a hard drive or other storage medium. Examples of software include web browsers, video games and operating systems.

Updating firmware involves downloading a new firmware image file, and then using a specialized tool to load the new firmware onto the device's memory while updating software is much simpler as it can be done using an automated update process or by downloading and running an installer program. Comparison between software and firmware is shown in Table 1.4.

1.1.2.3 Operation System

The operating system (OS) manages all of the software and hardware on the computer. It performs basic tasks such as file, memory and process management, handling input and output, and controlling peripheral devices such as disk drives and printers [36]. The operating system performs a variety of functions, including:

Table 1.4 Comparison between software and firmware

	Software	Firmware
Definition	Programs and data that run on a computer or device	Software that is embedded in a hardware device
Function	Provides functionality to a computer or device	Controls hardware functionality
Types	System software, application software	BIOS, device drivers, microcode, embedded software
Modifiability	Can be easily modified and updated	Difficult to modify and update once deployed
Portability	Designed to run on specific operating systems and hardware configurations	Designed to run on specific hardware configurations
Execution	Runs on the central processing unit (CPU)	Runs on a microcontroller or microprocessor
Examples	Microsoft Word, Google Chrome, Adobe Photoshop	Firmware for a printer, router, or smart thermostat

1. Process management: The operating system manages the execution of programs, allocating system resources such as memory and CPU time, and ensuring that different programs run smoothly without interfering with each other.
2. Memory management: The operating system manages memory usage, allocating and deallocating memory for programs as needed to ensure efficient use of system resources.
3. File system management: The operating system manages access to files and directories, ensuring that data is stored and retrieved correctly.
4. Device management: The operating system manages input and output devices, such as keyboards, mice, printers, and network devices, ensuring that they work correctly and can be accessed by applications.
5. Security management: The operating system provides security features such as user authentication, access control, and data encryption to protect against unauthorized access and data breaches.

Some examples of popular operating systems include Microsoft Windows, macOS, Linux, Android, and iOS. Different types of operating systems are optimized for different types of devices, such as desktops, laptops, servers, smartphones, and embedded systems.

1.1.3 Introduction: How to Develop HW/SW Together?

Language is a structured system of communication used by humans. When a human wishes to communicate with a computer system, a programming language is required. A programming language can convert a set of instructions, known as the source code, to perform a specific task. There are many common programming languages, such as C, C++, and JAVA. Each programming language requires a specific compiler, which can translate the source code into machine code. There are also other mechanisms to produce machine code that are interpreter based, and these use step-by-step executors of the source code. A language can be implemented with either a compiler or interpreter. A combination of both platforms is possible too where the compiler generates the machine code and then passes it to the interpreter for execution.

There are several high levels modeling language like SystemVerilog [37] and SystemC [17]. Modeling abstractions are summarized in Table 1.4. Table 1.5 shows how to develop SW/HW together. The different levels of abstraction and the different modeling languages are shown in Fig. 1.9, Tables 1.7 and 1.8.

MathWorks offers HDL Coder, which generates RTL from MATLAB and/or Simulink models [38]. But, without easy access to gate-level manipulations of algorithms. Python-based hardware modeling framework can enable agile hardware design flows this reduces the cost of validation by rapidly iterating towards "building the right thing" and increase productivity by enabling rapid implementation and verification [39]. Developing hardware and software together is a crucial process that involves close collaboration between hardware and software development teams. Here are some steps that can help ensure successful development of hardware and software together:

1. Define requirements: The first step is to define the requirements for both the hardware and software. This includes identifying the desired functionalities, performance criteria, and constraints for both components.

Table 1.5 Modeling abstraction

Modeling abstraction	Example
Functional level	Python, MATLAB
Cycle-Level	SystemC, C, C++ [40], SystemVerilog
Register transfer Level	Verilog [41], VHDL

Table 1.6 HW/SW languages

SW	HW
C, C++, SystemC	RTL
QEMU-based	SYSTEMC

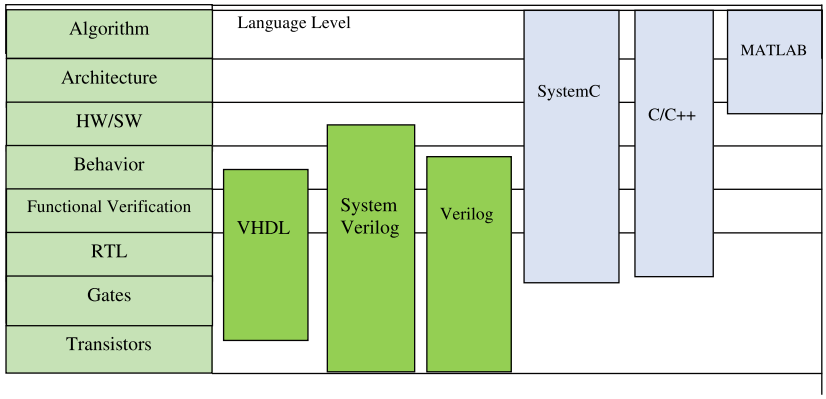


Fig. 1.9 Comparison between different modeling languages Error! Reference source not found.

Table 1.7 The modeling languages comparison

	MATLAB	SystemC	SystemVerilog	Verilog	VHDL
Architecture	YES	YES	NO	NO	NO
HW/SW	NO	YES	NO	NO	NO
Behavior modeling	NO	YES	YES	NO	YES
Functional verification	NO	YES	YES	NO	NO
Testbench	NO	YES	YES	YES	YES
RTL	NO	YES	YES	YES	YES
Gates	NO	NO	YES	YES	YES
Transistors	NO	NO	YES	YES	NO

2. Establish communication: The hardware and software development teams need to establish effective communication channels to share information, resolve issues, and ensure that both components are developed in sync.
3. Develop a system architecture: Develop a system architecture that considers both the hardware and software components, including the interfaces between them. This architecture should consider the requirements defined in step one.
4. Collaborative design: Collaborative design involves hardware and software development teams working together to design components that work together seamlessly. Hardware designers need to consider the software needs, while software developers need to consider hardware requirements. This helps to ensure that both components work together as intended.

Table 1.8 The modeling languages comparison: more details

Language	Description	Main Applications	Advantages	Disadvantages
Verilog	Hardware Description Language (HDL) used for digital circuit design	Digital circuit design, FPGA and ASIC design	Concise syntax, widely used in industry, simulation and verification tools available	Limited support for analog circuits, syntax can be difficult to read and understand
VHDL	Another HDL used for digital circuit design	Digital circuit design, FPGA and ASIC design	Strong typing, supports concurrent processes, widely used in industry	Steep learning curve, syntax can be verbose
SystemC	A C++ library used for hardware/software co-design and system-level design	System-level modeling, hardware/software co-design	High-level modeling capabilities, supports hardware/software co-design, well-suited for complex systems	Steep learning curve, syntax can be complex
C	A general-purpose programming language	Operating system development, firmware development, software development	Fast and efficient, widely used in industry, good support for system-level programming	Low-level programming can be difficult, manual memory management required
C++	An object-oriented programming language that builds upon C	Game development, software development, system-level programming	Object-oriented programming features, good support for system-level programming, widely used in industry	Steep learning curve, manual memory management required
MATLAB	A numerical computing environment and programming language	Engineering and scientific computing, data analysis and visualization	High-level language, good support for numerical computation, widely used in academia	Proprietary software, can be slow for large-scale computations, limited support for low-level programming

5. Prototype development: Develop a prototype of the hardware and software together to test and validate the system architecture and collaborative design. This helps to identify any issues or design flaws early in the development process.

6. Integration and testing: After developing the prototype, integrate the hardware and software components and perform comprehensive testing to ensure that they work together as intended.
7. Iterative development: Iterate the design and development process as needed to refine the system architecture, collaborative design, and prototype until the hardware and software components work together seamlessly.

1.1.4 Introduction: How to Run/Communicate HW/SW Together?

There are different approaches for communication between SW and HW. These approaches can be summarized in Fig. 1.10. The microprocessor and the HW are both connected to an on-chip communication mechanism, such as an on-chip bus. The *on-chip bus* transports data from the microprocessor module to the custom-hardware module. While typical on-chip buses are shared among several masters and slaves, they can also be implemented as dedicated point-to-point connections.

The SoC consists of buses and peripherals, where buses are for communication between different blocks inside the chip and peripherals for communications with outer world. Buses are the simplest and most widely used SoC interconnection networks to connect between different IPs in the SoC. The bus is a collection of signals (wires) to which one or more IP components (which need to communicate data with each other) are connected. Only one IP component can transfer data on the shared bus at any given time. To implement SoC buses we need standards to make it easy to connect diverse IPs quickly [42], where standards important for seamless **integration** of SoC IPs—helps avoid integration mismatches, where mismatches require development of "logic wrappers" at IP interfaces to ensure correct data transfers and it consumes time to be created, reduces performance, and takes up area [43].

- e.g., 1—connecting IP with 32 data pins to a 30 bit data bus.
- e.g. 2—connecting IP supporting data bursts to a bus with no burst support.

Two categories of standards for SoC communication are existing:

(1) **Standard bus architectures**:
   - define interface between IPs and bus architecture.
   - define at least some specifics of bus architecture that implements data transfer protocol.

(2) Socket based bus interface standards:
   - define interface between IPs and bus architecture.
   - freedom w.r.t choice and implementation of bus architecture.

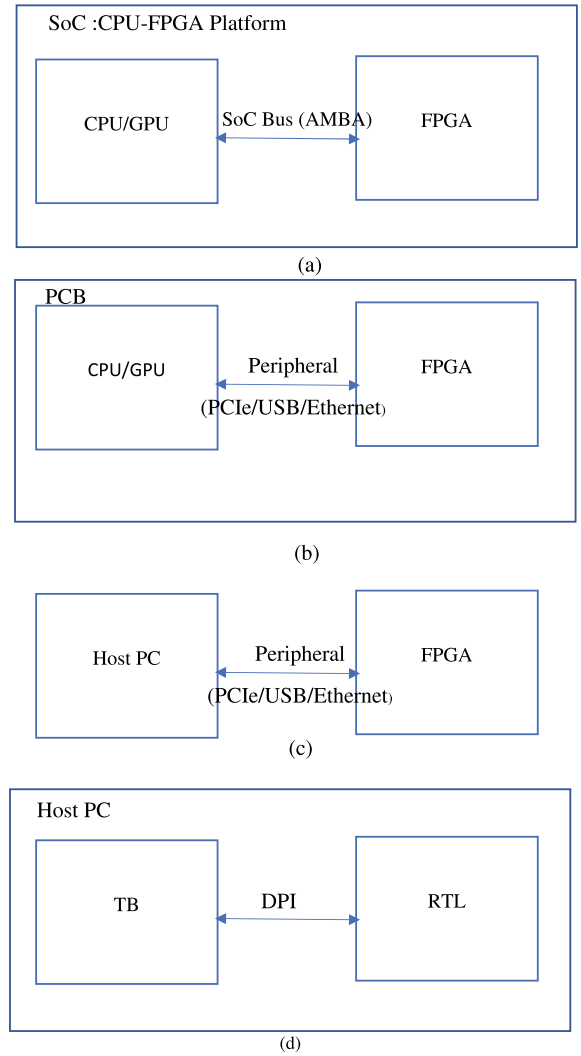


Fig. 1.10 **a** SoC Approach, **b** PCB-Approach, **c** Emulation approach, **d** Simulation approach

I2C is one of the famous SoC Buses. I2C eliminates the need for address decoders and glue logic, and it reduces space requirements, which keeps designs simple and flexible. It also supports simple constructions and enables easy upgrades. I2C buses are popular in the marketplace for low-speed peripheral devices such as radios, televisions, and personal digital assistants (PDA). I2C has a physical layout of two bidirectional wires, Serial Data Line (SDA) and Serial Clock Line (SCL), which transmit information between devices. Each device connected to the bus has a unique address assigned to it and can operate in receive and/or transmit mode with a designation as a master or slave. I2C offers the

possibility of having multiple masters; however, only one master can transmit data over the bus at a time [44].

Running and communicating hardware and software together requires a well-designed system architecture that considers both components and the interfaces between them. Here are some ways in which hardware and software can communicate and work together:

1. Drivers and APIs: Hardware devices often require device drivers, which are software programs that provide a standardized interface for communication with the hardware. APIs (Application Programming Interfaces) are also used to provide a standardized interface between software components. By providing a common interface, drivers and APIs allow hardware and software components to communicate and work together seamlessly.
2. Interrupts: Interrupts are a mechanism used by hardware devices to communicate with the CPU. When an event occurs, such as a button press or a data transfer, the hardware device sends an interrupt signal to the CPU, which stops the current process and responds to the event.
3. Direct Memory Access (DMA): DMA allows hardware devices to transfer data directly to and from memory, bypassing the CPU. This reduces the load on the CPU and improves system performance.
4. Shared memory: Shared memory is a region of memory that is accessible by both hardware and software components. Hardware devices can write data to shared memory, which can then be read and processed by software components.
5. Real-time operating systems: Real-time operating systems (RTOS) are designed to manage real-time applications, which require precise timing and responsiveness. RTOS often include specialized features such as interrupts, event-driven programming, and memory management, which allow hardware and software components to work together seamlessly in real-time systems.

1.1.5 Introduction: SoC Case Studies

Applications of SoC covers different domains as Aerospace, Communications, Healthcare, Energy, Appliances, Industrial Control, Automotive. The system on a chip (SoC) is an integrated circuit (IC), which includes various electronic parts such as central processing unit (CPU), input and output ports (I/O Ports), internal memory, analog input, output blocks, and others. All these components are incorporated on a single and very small chip. System on chip (SoC) can perform various applications such as wireless communication, signal processing, use of artificial intelligence, and others. The different components that make up a typical System-on-Chip (SoC) system:

1. Processing Units: Processing units are the heart of any SoC system. They are responsible for executing software programs and performing computations. Some of the commonly used processing units in SoC systems include:
   - Microprocessors: These are general-purpose processors that can execute a wide range of software applications. They are commonly used in embedded systems and consumer devices.
   - Digital Signal Processors (DSPs): These are specialized processors designed for performing digital signal processing tasks, such as audio and video processing.
   - Graphics Processing Units (GPUs): These are specialized processors designed for performing graphics processing tasks, such as rendering 3D graphics and video playback.
   - Other specialized processors: SoC systems may also include other specialized processors such as cryptography accelerators, neural network accelerators, and image signal processors.
2. Memory Units: Memory units are used for storing data and instructions. They can be integrated within the SoC or can be separate components that are connected to the SoC. Some of the commonly used memory units in SoC systems include:
   - Random Access Memory (RAM): This is a volatile memory that is used for storing data and instructions that are actively being used by the processor.
   - Read-Only Memory (ROM): This is a non-volatile memory that is used for storing firmware and other software programs that are not expected to change.
   - Flash memory: This is a non-volatile memory that is used for storing data and software programs that can be updated.
   - Cache memory: This is a small, high-speed memory that is used for temporarily storing frequently used data and instructions to improve performance.
3. Input/Output (I/O) Units: I/O units provide the interface between the SoC and the external world. They are responsible for transmitting and receiving data between the SoC and external devices. Some of the commonly used I/O units in SoC systems include:
   - Universal Serial Bus (USB) controllers: These are used for connecting devices such as keyboards, mice, and flash drives.
   - Ethernet controllers: These are used for connecting to wired networks.
   - Wi-Fi and Bluetooth controllers: These are used for wireless connectivity.
   - Audio and video codecs: These are used for encoding and decoding audio and video data.
   - Display interfaces: These are used for connecting to displays such as LCDs and OLEDs.
   - Sensor interfaces: These are used for connecting to sensors such as accelerometers, gyroscopes, and magnetometers.

4. Power Management Units: Power management units are responsible for managing the power consumption of the SoC. They are responsible for optimizing power consumption and extending the battery life of the system. Some of the commonly used power management units in SoC systems include:
   - Voltage regulators: These are used for regulating the voltage supplied to the SoC and other components.
   - Battery chargers: These are used for charging the batteries used in the system.
   - Power management integrated circuits (PMICs): These are used for managing the power consumption of the SoC and other components.

SoC's integrates IP-cores and processors with an on-chip interconnects where a system software program is running on a processor core for computation and managing the IP-cores which usually serve as hardware accelerators. *software refers* to the software program running on a processor core and the term *hardware* for an IP-core, which is managed by the software.

System on Chip (SoC) is a type of integrated circuit that integrates all components of a computer or other electronic systems onto a single chip. There are many examples for SoC. Each market has its own standard and each standard has its own requirement. Here are some examples of SoC:

1. **Mobile devices**: SoCs are commonly used in mobile devices such as smartphones and tablets. These SoCs include a processor, graphics processing unit (GPU), memory, and other components necessary for running mobile operating systems and applications. Examples of SoCs used in mobile devices include the Qualcomm Snapdragon series, Samsung Exynos series, and Apple A-series chips. Mobile phone SoCs are designed to provide high-performance computing capabilities in a compact and energy-efficient form factor. Some of the typical components of a mobile phone SoC are:
   - CPU (Central Processing Unit): The CPU is the brain of the SoC and is responsible for executing instructions and performing calculations. Mobile phone SoCs typically use ARM-based CPUs.
   - GPU (Graphics Processing Unit): The GPU is responsible for rendering graphics and images on the screen. It also assists with general-purpose computing tasks. Mobile phone SoCs typically use Adreno, Mali or PowerVR GPUs.
   - Modem: The modem is responsible for connecting the mobile phone to a cellular network. It handles the transmission and reception of data over the network. Mobile phone SoCs typically use Qualcomm or Mediatek modems.
   - Memory Controller: The memory controller manages the access to memory and storage devices, such as RAM, eMMC, and UFS. It ensures that data is transferred quickly and efficiently between these devices and the CPU.
   - Image Signal Processor (ISP): The ISP is responsible for processing image and video data from the camera sensors. It handles tasks such as noise reduction, color

correction, and image stabilization. Mobile phone SoCs typically use ISPs from Sony or Samsung.

2. **Internet of Things** (IoT): SoCs are also widely used in IoT devices, which require low power consumption, small form factor, and high performance. These SoCs often include a microcontroller unit (MCU), wireless communication modules, and other peripherals necessary for IoT applications. Examples of SoCs used in IoT include the Nordic Semiconductor nRF52 series and the STM32 series from STMicroelectronics.
3. **Automotive**: SoCs are increasingly used in automotive applications for infotainment systems, advanced driver-assistance systems (ADAS), and autonomous driving. These SoCs include multiple processing cores, image processing units, and other peripherals necessary for automotive applications. Examples of SoCs used in automotive include the NVIDIA Drive series and the Renesas R-Car series.
4. **Networking**: SoCs are also used in networking equipment such as routers, switches, and gateways. These SoCs often include multiple processing cores, Ethernet controllers, and other peripherals necessary for networking applications. Examples of SoCs used in networking include the Broadcom StrataDNX and StrataXGS series, and the Marvell ARMADA series. Some of the typical components of a router SoC are:
   - CPU: The CPU is responsible for managing the network traffic, routing packets, and handling other networking tasks. Router SoCs typically use ARM-based CPUs.
   - Ethernet Controller: The Ethernet controller manages the wired network connections and ensures that data is transmitted quickly and efficiently between devices.
   - Wireless Controller: The wireless controller manages the wireless network connections and ensures that data is transmitted quickly and efficiently between devices. Router SoCs typically use wireless controllers from Qualcomm or Broadcom.
   - Security Accelerator: The security accelerator provides hardware-based encryption and decryption capabilities for secure data transmission over the network.
   - Memory Controller: The memory controller manages the access to memory and storage devices, such as RAM and flash memory. It ensures that data is transferred quickly and efficiently between these devices and the CPU.
5. **Medical devices**: SoCs are used in medical devices such as portable patient monitoring devices and medical imaging equipment. These SoCs include low-power processing cores, analog-to-digital converters (ADCs), and other peripherals necessary for medical applications. Examples of SoCs used in medical devices include the Texas Instruments MSP430 series and the Analog Devices ADuCM350 series.
6. **Machine learning**: Machine learning can be integrated into SoCs as a component to provide hardware acceleration for AI applications. This is accomplished through the integration of dedicated hardware accelerators for tasks such as matrix operations, convolution, and activation functions. By offloading these tasks to specialized hardware, machine learning algorithms can be executed much more quickly and efficiently compared to running on a general-purpose processor [38]. Some SoCs designed for

AI applications, such as the NVIDIA Jetson series and Google's Tensor Processing Unit (TPU), include dedicated hardware accelerators specifically designed for machine learning tasks. These accelerators can process large amounts of data in parallel and provide a significant increase in performance compared to running the same tasks on a CPU or GPU. In addition to dedicated hardware accelerators, SoCs designed for machine learning applications can also include specialized software libraries and frameworks to simplify the development of machine learning algorithms. Examples of such libraries and frameworks include TensorFlow, PyTorch, and Caffe.

Figure 1.11 shows System block diagram of main chip of mobile phone. Figure 1.12 depicts system block diagram of a generic SoC and its key elements. Where here is a summary of how it works:

- The processor can boot an operating system (linux/android): First we boot the boot-code then we boot the firmware. We can use s SW debugger to download the boot-code and the firmware on a flash. Then the processor starts booting and running firmware.
- Then it starts talking to peripherals such as PCI, USB via AXI bus and bridges: there will be software sequence of events and hardware sequence of events.
- Any SoC generally have several interfaces (Peripherals) through which it communicates with the outside world. These could be USB, Ethernet, WLAN, I2C etc.

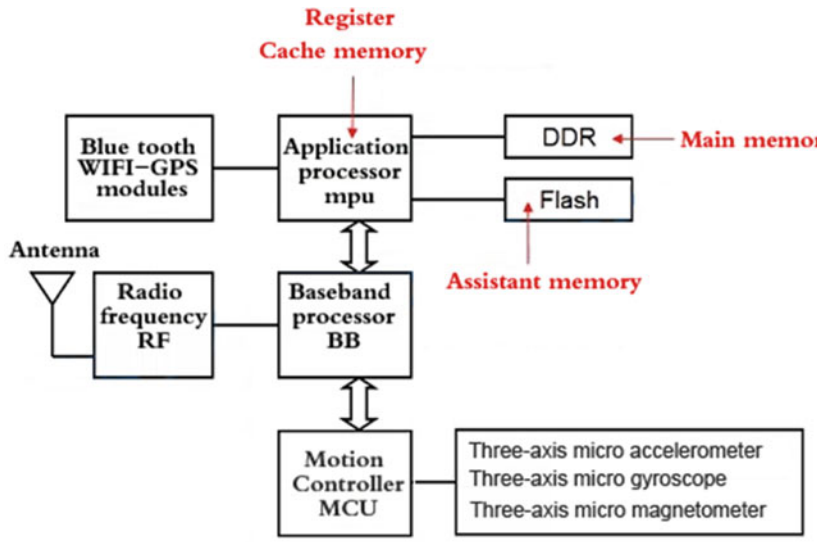


Fig. 1.11 System block diagram of main chip of mobile phone

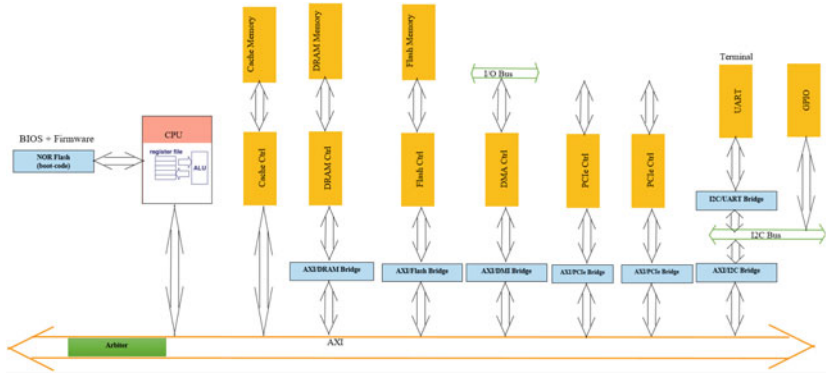


Fig. 1.12 System block diagram of a generic SoC. It is composed of microprocessors (single or multiple), hardware accelerators, on-chip memory hierarchies, and I/O. Moreover, it has standard bus interface port such as AXI. It has also **Cache/DRAM/ Flash/DMA** controllers. The program is loaded into the processor and then it starts executing it. Sensors are usually connected to I2C bus

- Direct memory access (DMA) is a method that allows an input/output (I/O) device to send or receive data directly to or from the main memory, bypassing the CPU to speed up memory operations. The process is managed by a chip known as a DMA controller (DMAC).

The booting process of Linux on a System-on-a-Chip (SoC) typically involves several stages before executing user-provided programs. While the specific stages may vary depending on the SoC architecture and configuration, a common sequence of boot stages includes:

- Power-On: When the SoC receives power or is reset, the initial hardware initialization and configuration take place. This includes setting up the processor, memory, and peripheral devices.
- Boot ROM: The Boot ROM, also known as the Primary Boot Loader, is a small piece of firmware stored in non-volatile memory on the SoC. It is responsible for initializing the hardware and loading the next stage bootloader.
- Secondary Bootloader: The Secondary Bootloader, often referred to as the First Stage Bootloader (FSBL) or Preloader, is loaded by the Boot ROM. It performs essential initialization tasks such as configuring memory, initializing device drivers, and loading the next stage bootloader or kernel image.
- Bootloader: The Bootloader is a software component that handles the boot process and loads the Linux kernel into memory. Common bootloaders used in the Linux ecosystem include U-Boot, GRUB, or Das U-Boot. The bootloader may also provide

functionalities like kernel command-line configuration, device tree loading, and other system-specific configurations.
- Linux Kernel: The Linux Kernel is the core component of the operating system. Once loaded by the bootloader, the kernel initializes the system, sets up essential services, and starts the user space initialization process.
- Init System: After the kernel initializes, it starts an init system or process (e.g., sysvinit, systemd, or initramfs). The init system is responsible for starting system services, mounting filesystems, and launching user space processes.
- User Space: Once the init system completes its tasks, it launches the user space environment. This includes starting essential system daemons, launching login services, and providing a shell or graphical user interface for user interaction.

1.1.6 Introduction: IP Management

There are various IP (intellectual property) management tools available in the market that can help manage the IP components used in the SoC (system-on-chip) design cycle. Some of the common IP management tools are:

- **Jira**: Jira is a project management tool that can be used to track issues, bugs, and tasks related to the SoC design project. It can be used to manage the design files and documents related to the IP components used in the SoC design, track the status of the IP components, and collaborate with team members working on the IP components.
- **SVN**: SVN (Subversion) is a version control system that can be used to manage the source code and design files related to the IP components used in the SoC design. It can help track changes made to the IP components, manage different versions of the IP components, and provide a centralized repository for the IP components.
- **GitHub**: GitHub is a web-based platform for hosting and collaborating on software projects. It can be used to store and share the design files related to the IP components used in the SoC design, collaborate with team members working on the IP components, and manage version control.
- **Jenkins**: Jenkins is an open-source automation server that can be used to automate the testing and verification of the IP components used in the SoC design. It can help ensure that the IP components meet the design specifications and function correctly, and can help streamline the verification process.

1.1.7 Conclusions

This chapter provides an in-depth overview of System-on-chip (SoC) and intellectual property (IP) era. It starts by discussing the motivation behind SoC, which is to increase

performance and decrease costs and time-to-market. It then goes on to explain the trade-offs between hardware and software in SoC design, highlighting the benefits of using a hybrid HW/SW Co-Design approach. This chapter also covers the design flow of hybrid SoC design, including IP reuse. It explains that pre-designed components are used to conquer the complexity of SoC, with hardware IP cores emerging as an integral part of modern SoC designs. The importance of IP reuse is emphasized as it helps reduce development time and cost while improving quality. Furthermore, it provides a solution hierarchy for any VLSI-based problem. It explains how to transform an algorithmic model into a FSM or flowchart, which is essential in designing a computer system. The overall design flow of hybrid SoC design is also shown. This chapter concludes by highlighting that designing a computer system is different from designing a phone or data-center due to different platforms and design requirements/constraints such as area, power, and performance. However, there are many common fundamental concepts that apply across all platforms.

References

1. Wolf, W., Modern VLSI Design: System-on-chip Design, Prentice Hall (2002) 3rd ed.
2. Nekoogar, F. and Nekoogar, F., From ASICs to SOCs: A Practical Approach, Prentice Hall (2003).
3. Asheden, P.J. and Mermet J., System-on-Chip Methodologies and Design Languages, Kluwer Academic (2002).
4. Uyemura, J.P., Modern VLSI Design – SOC Design, Prentice Hall (2001).
5. Rajsuman, R., System-on-a-chip: Design and Test, Artech House (2000).
6. J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," Proc. IEEE, Vol. 100, No. 13, pp. 1411–1430, May 2012.
7. S. Lee and J. S. Vetter, "Early evaluation of directive-based GPU programming models for productive exascale computing," in SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis. Salt Lake City, Utah, USA: IEEE press, 2012.
8. S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with xeon phi KNL," in Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International. IEEE, 2017, pp. 297–306.
9. A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in ACM Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU). Pittsburgh, Pennsylvania: ACM, 2010, pp. 63–74.
10. S. Lee and J. S. Vetter, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC). Vancouver: ACM, 2014.
11. S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing," in IEEE International Parallel & Distributed Processing Symposium (IPDPS). Chicago: IEEE, 2016.

12. S. Lee, S.-J. Min, and R. Eigenmann, “OpenMP to GPGPU: a compiler framework for automatic translation and optimization,” in 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. Raleigh, NC, USA: ACM, 2009.
13. S. Lee and R. Eigenmann, “OpenMPC: Extended openMP programming and tuning for GPUs,” in 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010, pp. 1–11.
14. M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin, Pragmatic Performance Portability with OpenMP 4.x. Cham: Springer International Publishing, October 2016, pp. 253–267.
15. N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, “Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers,” in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC ’11. New York, NY, USA: ACM, 2011, pp. 11:1–11:12. [Online]. Available: https://doi.org/10.1145/2063384.2063398.
16. C. LengauerSven, A. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, “Exastencils: Advanced stencil-code engineering,” in Euro-Par 2014: Parallel Processing Workshops, 2014, pp. 553–564.
17. David C. Black and Jack Donovan. SystemC: From the Ground Up. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
18. “Digital Design: Principles and Practices,” by John F. Wakerly, Pearson Education, 2017.
19. “Digital Design and Computer Architecture,” by David Money Harris and Sarah L. Harris, Morgan Kaufmann, 2012.
20. “SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling,” by Stuart Sutherland, Simon Davidmann, and Peter Flake, Springer, 2005.
21. “Verilog HDL: A Guide to Digital Design and Synthesis,” by Samir Palnitkar, Prentice Hall, 2003.
22. “ASIC Design in the Silicon Sandbox: A Complete Guide to Building Mixed-Signal Integrated Circuits,” by Keith Barr, John Wiley & Sons, 2007.
23. “Digital Design with RTL Design, Verilog and VHDL,” by Frank Vahid and Roman Lysecky, John Wiley & Sons, 2010.
24. “Digital Systems Engineering,” by William J. Dally and John W. Poulton, Cambridge University Press, 1998.
25. “System-on-Chip Design and Test,” by Rochit Rajsuman and Michael K. Keating, CRC Press, 2000.
26. “ASIC and FPGA Verification: A Guide to Component Modeling,” by Richard Munden, Morgan Kaufmann, 2004.
27. R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen, “Coming challenges in microarchitecture and architecture,” Proceedings of the IEEE, vol. 89, no. 3, pp. 325–340, 2001.
28. M. Flynn, P. Hung, and K. Rudd, 1999, “Deep submicron microprocessor design issues,” IEEE Micro, vol. 19, no. 4, pp. 11–22.
29. Mohammed, Khaled Salah. “FPGA implementation of enhanced data rate transceiver for Bluetooth 2. 0 application. “WSEAS Transactions on Communications 6.2 (2007): 359–363.
30. “Software Ecosystem: Understanding an Indispensable Technology and Industry,” by David G. Messerschmitt and Clemens Szyperski, The MIT Press, 2005.
31. “The Software Paradox: The Rise and Fall of the Commercial Software Market,” by Stephen O’Grady, O’Reilly Media, 2015.
32. “Managing Software Ecosystems,” by Ivan J. Jureta, Brian Henderson-Sellers, and Pnina Soffer, Springer, 2013.
33. “Open Source Strategies for the Enterprise,” by Bernard Golden, O’Reilly Media, 2007.
34. “Digital Ecosystems: Society in the Digital Age,” by Joel C. H. Ong, Springer, 2019.

35. https://www.vmware.com/.
36. https://www.uow.edu.au/student/learning-co-op/technology-and-software/operating-systems/.
37. Accellera. System Verilog 3.1 Accellera's Extensions to Verilog. www.systemverilog.org.
38. H. Genc, S. Kim, A. Amid, et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in Proceedings of the 58th Annual Design Automation Conference (DAC), 2021.
39. D .Lockhart, G .Zibrat, and C. Batten "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research" the Proceedings of the 47th Int'l Symp. on Microarchitecture (MICRO-47), December 2014.
40. Barr, M. & Massa, A. Oram, A. *(ed.)* Programming Embedded Systems in C and C++, 2nd Edition. *O'Reilly & Associates, Inc.,* **2006**.
41. Mohamed, K.S. (2016). Verilog for Implementation and Verification. In: IP Cores Design from Specifications to Production. Analog Circuits and Signal Processing. Springer, Cham. https://doi.org/10.1007/978-3-319-22035-2_5.
42. Mohamed, K.S. (2016). SoC Buses and Peripherals: Features and Architectures. In: IP Cores Design from Specifications to Production. Analog Circuits and Signal Processing. Springer, Cham. https://doi.org/10.1007/978-3-319-22035-2_4.
43. Rohita P. Patil, Pratima V. Sangamkar "A Review of System-On-Chip Bus Protocols" International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering, Vol. 4, Issue 1, January 2015.
44. D. Paret, the I2C Bus: From Theory to Practice. Chichester, New York: John Wiley & Sons, 1997.

SoC Design Methodologies

2

2.1 Introduction: Hardware Design Methodologies

Hardware design modeling methodologies are critical in the development of hardware systems. They provide a systematic approach for designing, simulating, verifying, and validating complex hardware systems. These methodologies are critical to the success of hardware design projects as they enable engineers to model the system and identify any issues before committing to the physical implementation. There are four primary hardware design methodologies for modeling a hardware IP, which are FPGA-based modeling, ASIC-based modeling, processor-based modeling, and PCB-based modeling (Fig. 2.1).

- FPGA-Based Modeling: FPGAs are programmable logic devices that allow engineers to configure hardware functionality using an HDL such as VHDL or Verilog. FPGA design is typically done using RTL modeling methodology and is often used for prototyping, rapid development, and low-volume production of digital circuits. FPGAs are highly flexible and can be reprogrammed, making them an excellent choice for applications where hardware changes are likely.
- ASIC-Based Modeling: ASICs are custom-designed integrated circuits tailored for a specific application or function. ASIC design is typically done using RTL or behavioral modeling methodology and is often used for high-volume production of digital circuits. ASICs are more costly to design and manufacture than FPGAs, but they offer higher performance, lower power consumption, and lower cost per unit for high-volume production.
- Processor-Based Modeling: Processors run programs written using a pre-defined fixed set of instructions (ISA). Processor-based modeling is typically used for applications that require high-performance computing and are well-suited to applications that require repetitive operations, such as video encoding, encryption, or decryption.

K. S. Mohamed, *Heterogeneous SoC Design and Verification*, Synthesis Lectures on Digital Circuits & Systems, https://doi.org/10.1007/978-3-031-56152-8_2

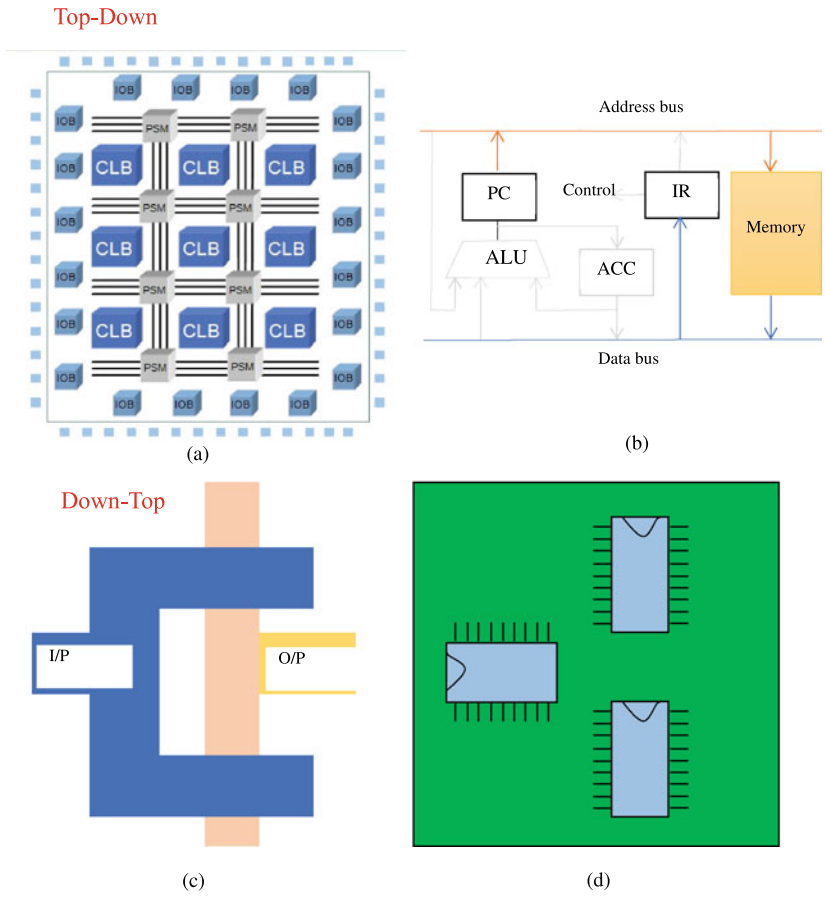


Fig. 2.1 **a** FPGA-based Modeling, **b** Processor-based Modeling, **c** ASIC-based Modeling, **d** PCB-based Modeling

- PCB-Based Modeling: PCBs are boards that contain electronic components connected by conductive traces. PCB design is typically done using schematic capture and layout tools, and PCBs are used for a wide range of electronic devices, from consumer electronics to aerospace and defense systems. PCB-based modeling uses standard ICs such as 74xx (TTL) and 40xx (CMOS), and it is not VLSI but uses discrete components. PCB-based modeling is well-suited to applications where the cost of the hardware is a significant factor, and volume production is not required.

The choice of hardware implementation technology depends on the specific requirements of the design, such as performance, power consumption, cost, and time-to-market. FPGA and ASIC design methodologies are more focused on the design process itself, while

Table 2.1 Comparison between different types of hardware

	Processor		ASIC	FPGA	PCB
	GPP	DSP			
Examples	μP, μC	MAC, FFT	–	–	–
Software/ hardware	Software	Software	Hardware	Hardware	Hardware
Spatial/ temporal	Temporal	Temporal	Spatial	Spatial	Spatial
Functionality	Programmable	Programmable	Fixed	Programmable	Fixed
Time to market	High	High	Low	High	Medium
Performance	Low	Medium	High	Med-High	Low
Cost	Low	Medium	High	Low	Low
Power	High	Medium	Low	Low-Med	High
Memory Bandwidth	Low	Low	High	High	Low
Companies	Intel-ARM	TI	TSMC	Xilinx [4]-Altera-Actel	Valor
Design alternative	Digital	Digital	Digital Analog RF Mixed	Digital	Digital Analog RF Mixed
Languages	C Assembly	C	–	Verilog VHDL	–

processor-based and PCB-based modeling methodologies are focused on the physical implementation of the design. In general, the choice of hardware implementation technology depends on the application and its requirements. The comparison between the different hardware options is shown in Table 2.1 and Fig. 2.2. Engineers must carefully consider the advantages and disadvantages of each technology before selecting the most appropriate methodology for their specific project.

2.1.1 FPGA-Centric SoC Design

The computing landscape has shifted towards heterogeneous systems with general-purpose graphic processing units (GPGPUs) or field programming gate arrays (FPGAs) to speed up computation in recent years. Compared with GPGPUs, FPGAs are more attractive for real-time sound field rendering because of their re-programmability and customization, which make it possible to tailor input/output interfaces and system data path

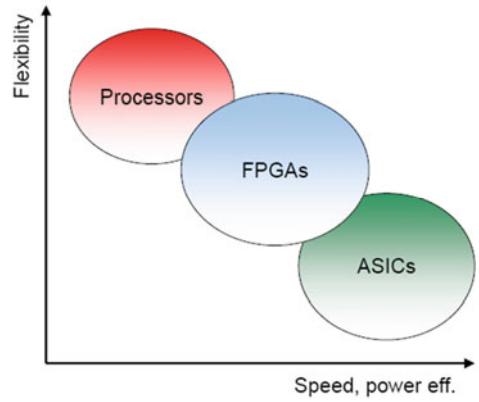


Fig. 2.2 Comparison between processors, FPGAs, and ASICs in terms of Flexibility versus speed

according to data flows in applications. The architecture of an FPGA differs from that of a processor (Fig. 2.3). A processor has a fixed architecture that relies on instructions that control the data path and peripheral resources in order to perform computations. An FPGA is made up of many configurable logic blocks (CLBs), switches, interconnecting wires, and input/output ports. Modern FPGA architectures consist of five types of modules: Configurable Logic Blocks (CLB), Digital Signal Processing (DSP) units, Block Random Access Memories (BRAM), I/O Blocks (IOB), and a configurable Interconnection Network. The **CLB** includes a configurable Look-Up Table (LUT) to implement bit-level logic functions, carry logic to support arithmetic operations such as binary adders, dedicated multiplexors, and Flip-Flops (FF). The **DSP** units implement large two's-complement multipliers and accumulators. The configurable interconnection network connects these modules together to implement digital circuits. **BRAMs** can be used as large storage areas or large LUTs with multiple outputs to implement logic functions. FPGAs can be reconfigured with new firmware; this process is very different from software development. FPGA firmware is typically written using a hardware description language (HDL) such as Verilog or VHDL (Very High-Speed Integrated Circuit Hardware Description Language) [2, 7, 8, 15]. As an accelerator, FPGA has [22]:

- High performance.
- Low power.
- Programmability and Flexibility.
- Short time-to-market.

For maximizing resource utilization, Dynamic partial reconfiguration (**DPR**) can be used. The configuration memory of the (reconfiguration region) RR consists of SRAM memory cells that control the content of the look-up tables and the state of the routing switches. To implement a circuit in the RR, a configuration needs to be generated that contains the binary values that need to be written in the RR's memory cells. Figure 2.4 shows

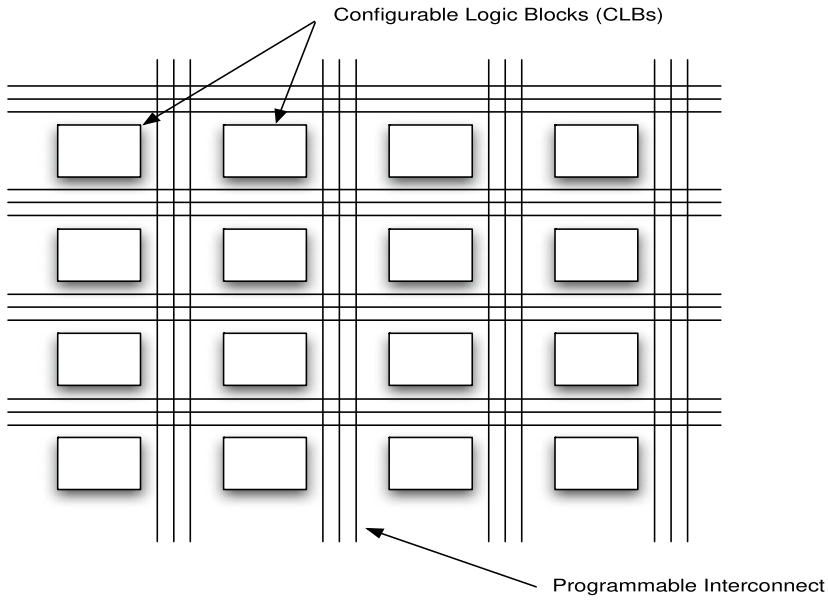


Fig. 2.3 FPGA architecture. Basic FPGA Architecture consists of: LUT, Flip-flop, DSP Block, Storage elements: BRAM

an example describes the role of configuration memory. In conventional DPR systems, a configuration bitstream is generated for every mode by implementing it separately in the RR, where every RR memory cell corresponds to a collection of binary values, one value for each mode. When these binary values are the same, this collection is called a static bit. If they are not the same, this collection is called a dynamic bit. Memory cells containing a static bit do not need to be rewritten during run-time [17–21].

The important steps of the conventional FPGA flow can be summarized in Fig. 2.5. The logic synthesis of the design to transform a hardware description language, such as Verilog into Boolean gates, can be accomplished using numerous technology independent techniques, with the ultimate objective of optimizing the Boolean network. Placement is meant to optimize the routing resources when routing the connection between multiple connected blocks. Those connected blocks are placed near each other to further optimize the FPGA architecture by balancing the wire density.

After the netlist instances are placed, connections are routed between them using the limited routing resources, with the objective of each signal using a unique routing resource. For this purpose, the resources of FPGA are represented as a directed graph and an example of this representation is shown in Fig. 2.6. Since the FPGA routing resources are limited, routing is deemed to be the most time-consuming phase of the CAD Flow requiring days for the largest commercial designs [23].

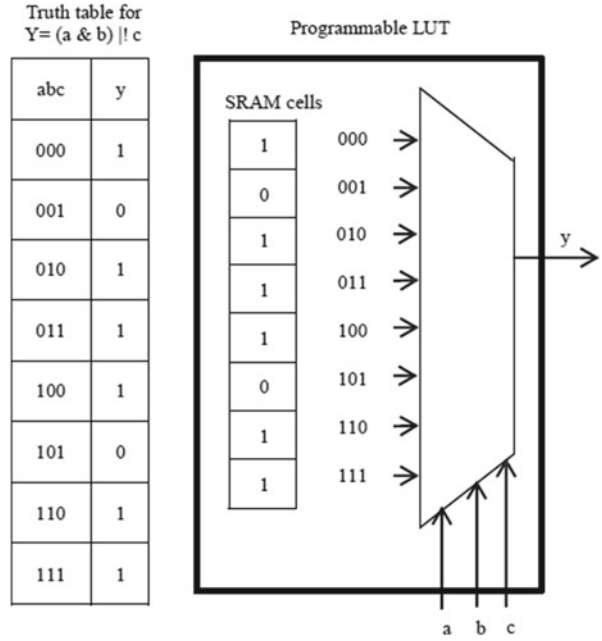


Fig. 2.4 An example describes the role of configuration memory

In the context of FPGA design, **SLR** stands for Super Logic Region. SLR is a physical region on an FPGA that contains a group of logic elements, such as lookup tables (LUTs), flip-flops, and interconnect resources. SLRs are typically used in large FPGAs to partition the design into smaller, more manageable sections, allowing for better design performance and timing closure. SLRs provide a number of benefits for FPGA designers. By partitioning the design into smaller sections, SLRs can help reduce congestion and improve timing closure, making it easier to achieve the desired performance and functionality. SLRs can also help improve routing efficiency by allowing designers to place related logic elements closer together, reducing the length of interconnects and improving signal integrity. In addition to these benefits, SLRs can also help with design reuse and IP integration. By partitioning the design into logical blocks, SLRs make it easier to reuse design blocks across multiple projects and to integrate third-party IP into the design. SLRs are an important feature of modern FPGAs that provide a range of benefits for FPGA designers, including improved performance, timing closure, routing efficiency, and design reuse.

2.1.1.1 ASIC RTL Versus FPGA RTL

ASIC RTL and FPGA RTL are two different types of digital design implementations that serve different purposes.

ASIC RTL (Register Transfer Level) design is a process of creating a digital circuit using a hardware description language (HDL) such as Verilog or VHDL, and optimizing

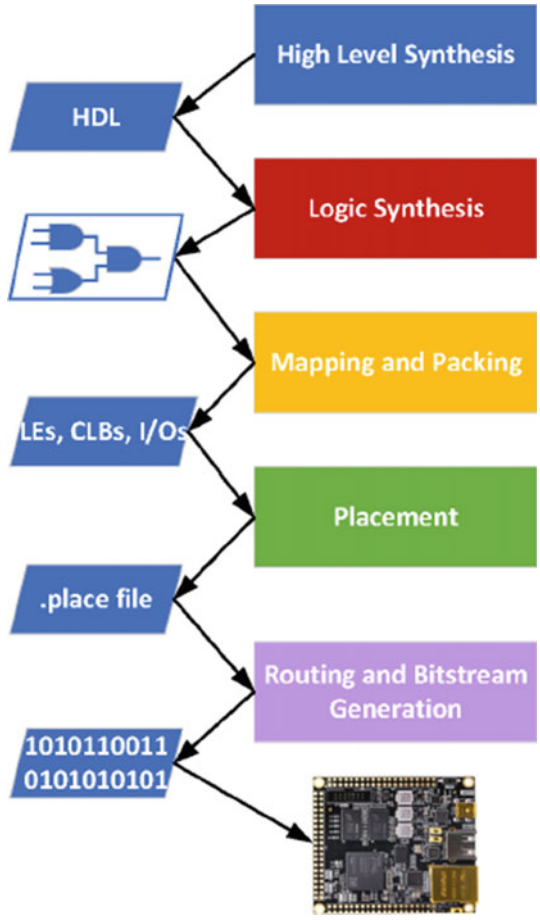


Fig. 2.5 FPGA flow

it for implementation in an Application-Specific Integrated Circuit (ASIC). The design is tailored to a specific application, and the ASIC is typically produced in large quantities for a specific purpose. ASIC designs can be more complex and optimized than FPGA designs, but they are also more expensive and have longer development times.

FPGA RTL design is also a process of creating a digital circuit using HDL, but the design is optimized for implementation in a Field-Programmable Gate Array (FPGA) instead of an ASIC. FPGAs are programmable logic devices that can be reconfigured to perform a wide range of functions. FPGA designs are typically less complex than ASIC designs and can be developed faster and at a lower cost. They are also more flexible and can be reprogrammed, making them suitable for prototyping and low-volume production.

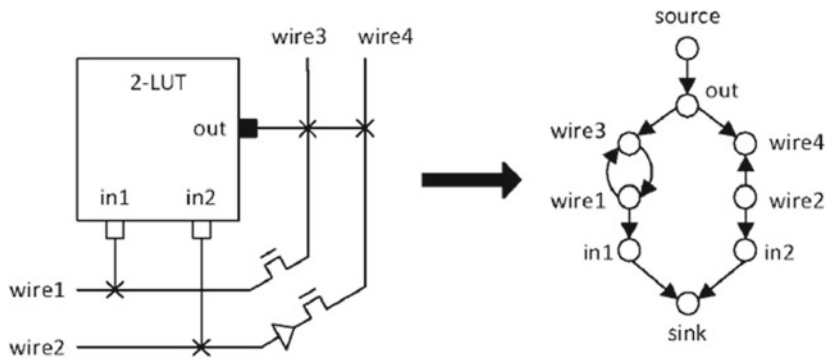


Fig. 2.6 Directed graph representation for FPGA routing task

ASIC RTL and FPGA RTL have some differences in terms of clocking, primitives, and memory.

- **Clocking**: ASIC designs typically have more stringent clocking requirements than FPGA designs. The clock tree in an ASIC design must be carefully designed to ensure that the clock signal is distributed evenly throughout the circuit and meets timing requirements. In contrast, FPGAs have more flexible clocking options, and clock signals can be distributed more easily using the FPGA's programmable routing resources. An example is shown below.

Listing 1

ASIC	FPGA

module clk_tree (
  input clk_in,
  output reg clk_out
);
 	
  wire clk_int;
 	
  BUFG buf (.I(clk_int), .O(clk_out));
 	
  assign clk_int = clk_in;
 	
endmodule
	module clk_divider (
	  input clk_in,
	  input [7:0] divisor,
	  output reg clk_out
	);
	  reg [7:0] count = 8'h0;
	  always @(posedge clk_in) begin
	  if (count == divisor) begin
	  count <= 8'h0;
	  clk_out <= ~clk_out;
	  end else begin
	  count <= count + 1;
	  end
	  end
	  endmodule

- **Primitives**: ASIC designs typically use standard-cell libraries that contain a set of pre-designed logic gates and flip-flops. These libraries are optimized for the specific fabrication process and technology, and provide high-performance and low-power consumption. In contrast, FPGAs use programmable logic blocks, such as Look-Up Tables

(LUTs), Flip-Flops, and Multiplexers, that can be programmed to implement any logic function. These logic blocks are more flexible, but may not be optimized for a specific application or fabrication technology. An example is shown below.

Listing 2

ASIC	FPGA
module and_gate (input a, input b, output c); and and_inst (.A(a), .B(b), .Z(c)); endmodule	module and_gate (input a, input b, output c); wire [1:0] lut_out; assign lut_out = a & b;

- **Memory**: ASIC designs often use custom memory structures that are optimized for a specific application or target technology. For example, an ASIC may use a custom memory structure that reduces power consumption, increases density, or provides faster access times. In contrast, FPGAs typically use embedded memory blocks, such as RAM and ROM, that are programmable and can be used to implement any memory function. These embedded memory blocks are not as optimized as custom memory structures, but they provide more flexibility and ease of use.

2.1.2 Processor-Centric SoC Design

Most SoC designs adopt a processor-centric approach since a microprocessor can easily handle all communication tasks between the different hardware components integrated into a SoC. Processors can be classified based on their intended use and functionality. In this context, three common classifications are:

- **GPP** (General Purpose Processor): A processor designed for general computing tasks, such as browsing the web, word processing, and running general software applications. Examples of GPPs include Intel Core processors and AMD Ryzen processors.
- **DSP** (Digital Signal Processor): A processor optimized for processing signals such as audio, video, and other data types that require fast and efficient processing. DSPs are commonly used in devices such as smartphones, digital cameras, and home theater systems.
- **GPU** (Graphics Processing Unit): A processor optimized for rendering and processing graphics, including 3D modeling, video games, and video playback. GPUs are commonly used in devices such as gaming computers, workstations, and high-performance computing clusters [36].

Each of these classifications has specific design features and optimization to suit its intended use case. While there can be some overlap between the different classifications, processors are generally optimized for specific tasks to provide the best performance and efficiency in their intended applications.

When it comes to the architecture evolution of processors, the industry has witnessed a progression from designing processors using schematics (depicting transistors and gates) to Hardware Description Language (HDL) designs (representing cells and blocks). This evolution further advanced to the use of Intellectual Properties (IPs), which are pre-designed and pre-verified functional units that can be integrated into larger systems. The development of 2D System-on-Chip (SoC) designs followed, where multiple IPs are combined on a single chip to create a complete system.

In recent times, the trend has shifted towards 3D System-on-Chip (SoC) designs. 3D SoCs involve stacking multiple layers of chips vertically, allowing for increased integration density and improved performance. This approach helps overcome the limitations imposed by traditional 2D scaling, where the continuous reduction in transistor size becomes challenging due to physical constraints and power consumption concerns.

Architecture evaluation of processors started with design with schematics (sea of transistors and gates, then design with HDL (sea of cells and blocks, then design with Ips, then 2D SoC and now 3D SoC. A comparison between Microprocessor system versus reconfigurable system (FPGA-based) is shown in Fig. 2.10. Consider factors such as processing power, memory, and input/output capabilities when selecting a processor. When comparing a microprocessor system (typical CPU-based system) with a reconfigurable system based on Field-Programmable Gate Arrays (FPGAs), several factors should be considered:

- **Processing Power**: Microprocessors are optimized for sequential processing and complex control flow. They excel in tasks that require a single thread or process to execute efficiently. On the other hand, FPGAs offer highly parallelizable computing resources and can achieve massive parallelism. They are suitable for applications with high data parallelism, where multiple computations can be executed simultaneously.
- **Memory**: Microprocessors generally have larger on-chip cache memories and support a wide range of memory hierarchy. They provide efficient memory management and caching mechanisms. FPGAs, however, have limited on-chip memory resources, and data storage is often implemented using off-chip memory elements. The memory requirements should be carefully considered when choosing between a microprocessor and an FPGA-based system.
- **Input/Output Capabilities**: Microprocessors typically have extensive input/output capabilities, including various peripheral interfaces and communication protocols. They are well-suited for applications requiring extensive interaction with external devices. FPGAs, on the other hand, can be highly flexible and customizable in terms of

input/output interfaces. They can be programmed to support specific communication protocols or implement custom interfaces as required.

2.1.2.1 GPP

The general architecture for a very simple processor consists of: program counter (PC), accumulator (ACC), arithmetic logic unit (ALU), instruction register (IR). The PC holds the address of next instruction to be executed, ACC holds the data to be processed, ALU performs operation on data, IR holds the current instruction code being executed. The operation can be summarized in the following steps (Fig. 2.7):

(1) **Instruction fetch**: The value of PC is outputted on address bus, memory puts the corresponding instruction on data bus, where it is stored in the IR.
(2) **Instruction decode**: The stored instruction is decoded to send control signals to ALU which increment the value of PC after pushing its value to the address bus.
(3) **Operand fetch**: The IR provides the address of data where the memory outputs it to ACC or ALU.
(4) **Execute instruction**: ALU is performing the processing and store the results in the ACC.

The instruction types include data transfer, data operation (arithmetic, logical), and program control such as interrupts.

Theses cycles are continuous and called fetch-decode-execute cycle. The processors can be programed using high level language such as C or mid-level language such as

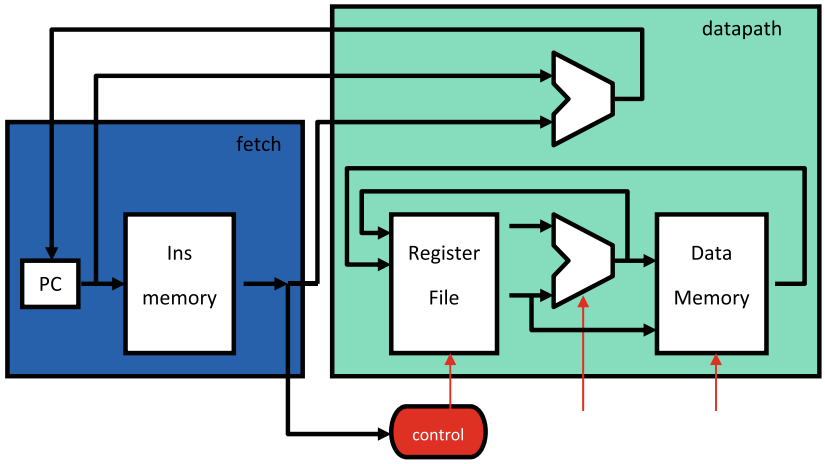


Fig. 2.7 Fetch → Decode → Execute "cycle". Datapath performs computation. Control determines which computation is performed. Fetch: get instruction, translate opcode into control

assembly. Assembly is used for example in nuclear application because it is more accurate. At the end the compiler translates this language to the machine language which contains only ones and zeroes. Algorithm development in the application domains of computer vision, big data, sensor fusion, and wireless communication has greatly outpaced our ability to deliver dedicated hardware accelerators that promise orders-of-magnitude energy-efficiency and performance improvements over CPU/GPU realizations. As shown in Fig. 2.4, Processors have more flexibility and ease of use over FPGA.

Traditionally, the CPU is targeted for general-purpose computation; it is evolved from single instruction single data (SISD) architecture to single instruction multiple data (SIMD). It employs multiple processor cores (2, 4, 6, 8, or 16 cores) to support multi-thread processing for parallel computation. There are many processors such as **X86, ARM, and RISC-V** (Table 2.2). CISC, RISC, and VLIW are three different processor architectures used in modern computer systems (Table 2.3).

Little endian is a byte ordering format used in computer systems and processors where the least significant byte (LSB) is stored first in memory or sent first in data transmission. This means that the lower-order bytes of a multi-byte value are stored at lower memory addresses, while the higher-order bytes are stored at higher memory addresses. In contrast, big endian is another byte ordering format where the most significant byte (MSB) is stored first in memory or sent first in data transmission. In big endian format, the higher-order bytes of a multi-byte value are stored at lower memory addresses, while the lower-order bytes are stored at higher memory addresses.

Whether a processor uses little endian or big endian format can affect the way it processes data and communicates with other devices or systems. Some processors, like ARM processors, allow for both little endian and big endian formats to be used, while others, like x86 processors, typically only use little endian format.

Reset is a vital mechanism in the operation of any processor, serving as a means to initialize the system and establish a consistent state. When a processor undergoes a reset event, it begins by reading the "Reset vector." This vector holds the memory address that indicates where the processor should start executing code after the reset. By examining the reset vector, the processor can determine whether it should boot from ROM, RAM, or even a peripheral device. Once the reset vector is determined, the bootloading process commences. This process relies on the program counter (PC), which keeps track of the address of the next instruction to be executed. The PC is set to the address specified by the reset vector, ensuring that the bootloading code is the first code to be fetched and executed. This code, typically stored in ROM or other non-volatile memory, takes control of the processor and performs crucial initialization tasks. During the bootloading phase, the code configures various hardware components, initializes memory, and sets up the initial state of the system. It may also be responsible for loading subsequent bootloaders or operating systems into the appropriate memory locations. By bootloading based on the PC, the processor follows a defined sequence of instructions that sets the stage for further software execution.

Table 2.2 Comparison between different processors architecture

Processor	x86	ARM	RISC-V	MIPS	Apple	Texas instruments
Instruction Set Architecture	CISC	RISC	RISC	RISC	ARM-based custom	VLIW
Register Size	16–64 bits	32–64 bits	32–64 bits	32–64 bits	64 bits	64 bits
Endianness	Little-endian	Little/Big	Little-endian	Big-endian	Little-endian	Little/Big
Applications	**Desktop** and server computers	**Mobile** devices, IoT devices, and embedded systems	Embedded systems, **IoT** devices, and specialized computing	Embedded systems and networking devices	iPhone, iPad, and Mac computers	Digital Signal Processing (**DSP**), multimedia and graphics
Operating Systems	Windows, Linux, macOS	Android, iOS, Linux	Linux, FreeRTOS, Zephyr	Linux, FreeBSD	iOS and macOS	Windows, Linux,
Performance	High power consumption and high performance	Low power consumption and moderate performance	Moderate power consumption and moderate performance	Moderate power consumption and moderate performance	High performance with low power consumption	High performance processing capabilities
Cost	Higher cost due to complexity and branding	Lower cost due to simpler architecture	Lower cost due to open-source architecture	Lower cost due to simple architecture	Higher cost due to custom design	Higher than other general-purpose processors due to its specialized DSP capabilities
Vendor	Intel, AMD	ARM Holdings, Qualcomm, Samsung	SiFive, Western Digital	MIPS Technologies	Apple Inc	Texas Instruments

Table 2.3 Comparison between different instruction sets

Processor architecture	Examples of processors	Endianness	Word size	Instruction set	Applications
CISC	Intel Pentium 4, Motorola 68000, DEC VAX	Little/Big	8-bit to 64-bit	Complex instruction set	Desktop computers, servers, mainframes, scientific computing
RISC	ARM, MIPS, RISC-V	Little/Big	32-bit, 64-bit, 128-bit	Reduced instruction set	Mobile devices, embedded systems, routers, consumer electronics
VLIW	Intel Itanium, Texas Instruments TMS320C6x, STMicroelectronics ST200	Little/Big	64-bit	Very Long Instruction Word	Digital Signal Processing (DSP), multimedia and graphics, scientific and high-performance computing, networking, embedded systems

CPUs (Central Processing Units) need L1, L2, and L3 caches to improve overall system performance by reducing the latency or delay in accessing data from main memory. Let's understand each cache level and its purpose (Figs. 2.8, 2.9 and 2.10):

1. L1 Cache (Level 1 Cache): The L1 cache is the smallest but fastest cache located directly on the CPU core. It is divided into two parts: the instruction cache (L1i) and the data cache (L1d). The L1 cache stores frequently accessed instructions and data that the CPU needs for immediate processing. Its proximity to the CPU core ensures quick access, reducing the need to access data from slower memory sources like RAM. The L1 cache has low capacity but provides extremely low latency, enabling fast data retrieval.
2. L2 Cache (Level 2 Cache): The L2 cache is the second level of cache in the CPU hierarchy, and it is larger than the L1 cache. It acts as a middle ground between the L1 cache and the main memory. The L2 cache stores additional instructions and data that are frequently accessed but may not fit in the L1 cache. It has a larger capacity than the L1 cache but has slightly higher latency.
3. L3 Cache (Level 3 Cache): The L3 cache is a shared cache that is present on some CPU architectures. It is larger than both the L1 and L2 caches and is shared among all the CPU cores in a multi-core processor. The L3 cache serves as a common pool of data

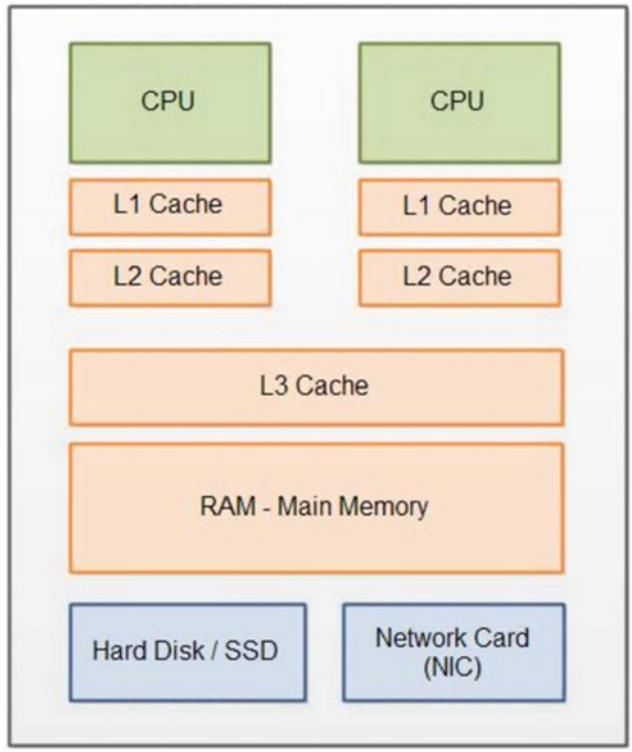


Fig. 2.8 Cache hierarchy

Fig. 2.9 Processors flexibility and ease of use

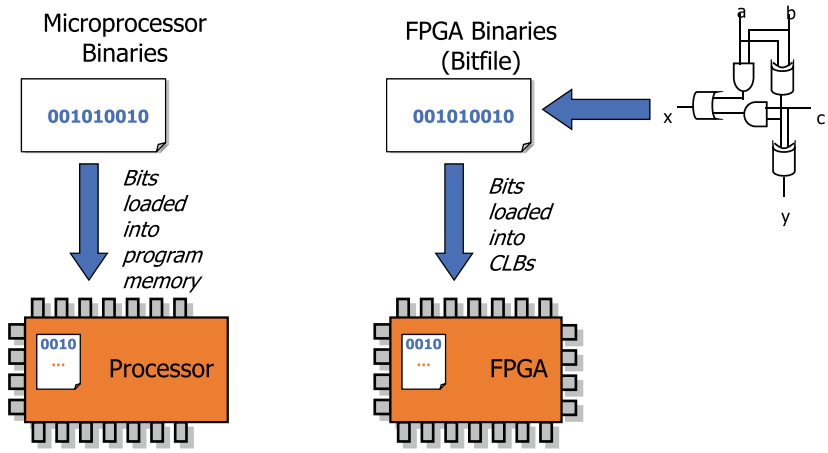


Fig. 2.10 Microprocessor system versus reconfigurable system

that can be accessed by any core when needed. It helps in reducing data duplication and improves overall performance by providing faster access to shared data.

The purpose of these cache levels is to minimize the time required to fetch data or instructions from main memory, which is relatively slower compared to the cache. By storing frequently accessed data closer to the CPU cores, caches help reduce the number of memory accesses and minimize the performance impact of memory latency.

2.1.2.2 MCU

A **microcontroller** is a specialized computing device that combines a microprocessor, memory, and input/output peripherals, all on a single chip. Unlike a microprocessor, a microcontroller is designed for specific tasks and applications. Microcontrollers are commonly used in embedded systems, such as control systems, robotics, and consumer electronics devices. Some of the key differences between microcontrollers and microprocessors are:

- Architecture: Microprocessors have a separate chip for memory and input/output peripherals, while microcontrollers have them all on a single chip.
- Cost: Microcontrollers are generally less expensive than microprocessors, as they have fewer components.
- Power consumption: Microcontrollers are designed to operate on low power, making them suitable for battery-powered devices, while microprocessors require more power.
- Applications: Microprocessors are suitable for general-purpose computing tasks, while microcontrollers are used in specialized applications that require real-time processing, control, and monitoring.

Microcontrollers are designed to be slower than microprocessors because they are intended for embedded systems, which require low power consumption, small form factor, and low cost. They are designed to execute simple and specific tasks, rather than the complex operations that microprocessors are capable of performing. An **AVR** (Alf and Vegard's RISC processor) is a type of microcontroller, developed by Atmel (now owned by Microchip Technology). AVR microcontrollers are known for their simple instruction set, low power consumption, and fast processing speed, which make them suitable for a wide range of applications, including industrial automation, robotics, and consumer electronics. There are many microcontrollers available in the market from different manufacturers, but some of the most famous ones include:

- **Atmel AVR**: AVR microcontrollers are popular for their ease of use, wide availability, and low cost. They are widely used in many embedded systems applications.
- **Microchip PIC**: The PIC microcontroller is another popular choice for embedded systems development. It is known for its ease of use, low cost, and wide range of features.
- **ARM Cortex-M**: ARM Cortex-M is a family of microcontrollers based on the ARM architecture. They are known for their low power consumption, high performance, and flexibility.
- **Texas Instruments MSP430**: The MSP430 is a popular ultra-low-power microcontroller family from Texas Instruments. It is widely used in applications where power consumption is critical, such as battery-powered devices.
- **Intel 8051**: The 8051 is an 8-bit microcontroller that has been around since the 1980s. It is still widely used in many embedded systems applications due to its simplicity, low cost, and wide availability.
- **Raspberry Pi**: Although not strictly a microcontroller, the Raspberry Pi is a popular single-board computer that is widely used in embedded systems and DIY projects. It is known for its low cost, high performance, and flexibility.
- **Arduino:** is a popular microcontroller board based on the Atmel AVR microcontroller. It is an open-source platform that allows for easy programming and prototyping of electronic projects. Arduino boards typically have a variety of input and output pins that can be used to interact with sensors, actuators, and other electronic components. They also have a built-in USB interface, making it easy to program the board and communicate with other devices. The popularity of Arduino has led to a large community of users, which has contributed to a wealth of resources and support available online.

Both microprocessors and microcontrollers can have **watchdog** timers. A watchdog timer is a component that is used to monitor the operation of a system and reset it if a fault or malfunction occurs. It is designed to prevent the system from getting stuck in an undesirable state. Microprocessors are the central processing units (CPUs) of computers and other devices, while microcontrollers are integrated circuits that combine a microprocessor core

with peripherals and memory on a single chip. Microcontrollers are often used in embedded systems and devices where real-time control and monitoring are required. While watchdog timers are commonly associated with microcontrollers, many modern microprocessors also include watchdog timer functionality. This allows the microprocessor-based system to monitor itself and reset if necessary, providing an additional level of reliability and fault tolerance. The implementation of watchdog timers may vary between different microprocessors and microcontrollers, but the concept and purpose remain the same.

2.1.2.3 GPU

GPU consists of massively parallel Processing Elements (PEs) designed for computer graphics and image processing. In recent years, it is also targeted for artificial intelligent development (image classification, speech recognition, and autonomous vehicle), especially for neural network training [24]. One of the key characteristics of a GPU is its massively parallel architecture. It consists of a large number of Processing Elements (PEs), also known as CUDA cores or shader cores, which can perform calculations concurrently. This parallel design enables GPUs to process multiple data streams simultaneously, leading to significant speedup in tasks that can be divided into parallel subtasks. While GPUs were initially developed for graphics processing, their architecture is well-suited for many parallel computing tasks beyond graphics. This led to the emergence of General-Purpose GPU (GPGPU) computing, where GPUs are utilized for non-graphics computations. GPGPU applications involve leveraging the parallel processing power of GPUs to accelerate tasks such as scientific simulations, data analysis, cryptography, and more. In recent years, GPUs have gained significant attention in the field of artificial intelligence (AI) and machine learning (ML), particularly for neural network training. Deep learning models, which are often composed of millions or even billions of interconnected neurons, require substantial computational resources to train effectively. GPUs excel in this domain due to their parallel architecture, allowing for the efficient execution of matrix operations commonly found in neural network computations.

2.1.2.4 DSP

DSP stands for Digital Signal Processing. It is a field of study and a technology that involves the manipulation, analysis, and interpretation of digital signals using mathematical algorithms and computing techniques. DSP is widely used in various applications such as telecommunications, audio and video processing, radar systems, medical imaging, control systems, and many others.

Digital signals are discrete-time signals represented by a sequence of numbers, typically obtained from real-world analog signals through analog-to-digital conversion. These signals can be processed and transformed using DSP techniques to extract information, enhance quality, remove noise, or perform various other operations.

Some key aspects of DSP include:

- **Signal Representation**: Digital signals are represented as sequences of discrete samples. Each sample represents the amplitude of the signal at a specific time instant.
- **Filtering and Frequency Analysis**: DSP allows for the design and implementation of various filters to manipulate signals. Filtering operations can include low-pass, high-pass, band-pass, and notch filters. Frequency analysis techniques such as Fourier analysis and Fast Fourier Transform (FFT) are commonly used to analyze the frequency content of signals.
- **Convolution**: Convolution is a fundamental operation in DSP used for various purposes, including filtering, linear time-invariant system analysis, and signal convolutional coding.
- **Sampling and Reconstruction**: Sampling is the process of converting continuous-time signals into discrete-time signals, and reconstruction involves converting the discrete-time signals back into continuous-time signals. These processes are essential for analog-to-digital and digital-to-analog conversion.
- **Signal Compression**: DSP techniques enable efficient data compression and storage of digital signals. Techniques like lossless compression (e.g., run-length encoding) and lossy compression (e.g., transform coding) are used to reduce the size of digital signals while minimizing the loss of important information.
- **Adaptive Filtering**: Adaptive filters are used in DSP applications where the characteristics of the signal change over time. These filters adjust their parameters automatically based on the input signal, allowing for effective noise cancellation, echo cancellation, and equalization.
- **Speech and Audio Processing**: DSP plays a crucial role in speech and audio processing applications, including speech recognition, speech synthesis, audio coding (e.g., MP3), and audio effects (e.g., equalization, reverb).
- **Image and Video Processing**: DSP techniques are used extensively in image and video processing applications. Image enhancement, denoising, compression (e.g., JPEG), object recognition, and video coding standards (e.g., H.264) are some of the areas where DSP is employed.

Digital Signal Processing has revolutionized numerous fields by enabling advanced signal analysis, manipulation, and interpretation. Its applications span a wide range of domains and continue to grow as technology advances. From communication systems to multimedia processing and biomedical engineering, DSP plays a crucial role in shaping our modern digital world.

2.1.3 HLS-Centric SoC Design

Recently, high-level synthesis (HLS) hardware design methodologies have gained popularity as an alternative to using HDLs and the RTL level of design abstraction. With HLS,

C/C++ software program is written to specify the desired behavior. Then, the program is compiled and run on a standard microprocessor to verify its functionality and correctness. Following this, one uses an HLS tool to compile" the C/C++ specification automatically into a hardware circuit in RTL. HLS allows hardware design to happen at a higher level of abstraction and makes hardware accessible to those with solely software skills. HLS allows the use of behavioral-level programming languages (C/C++) to be the abstract descriptions of hardware functions. It thus helps reduce considerable amounts of lines of code compared to RTL programming languages [6, 9].

High level synthesis is a powerful tool for early performance analysis and **architecture optimization**. Moreover, designing and verification is faster with high level synthesis (HLS). HLS-flow is nearly 50% faster than traditional flow. SystemC is an event driven simulator like Verilog, but the SystemC model is at a higher level of abstraction (more abstract level) so there are fewer events, so it is faster in simulation. C-level modeling is faster in terms of simulation and easier in terms of coding. Pre-HLS and Post-HLS simulations should be functionally the same. However, they may differ in latency. HLS reduces IP development time as algorithms can be easily modified and regenerated. Designs can be described by C++ or SystemC and HLS tools will synthesize them to RTL code. At the same time, the synthesizable RTL can meet performance/area goals and it will behave the same as the C++/SystemC code. Moreover, RTL/C++ verification can re-use test stimulus, metrics19 and practices. Thus, resulting in speeding up verification. Fast verification makes a rapid product/design changes feasible. Design productivity alone is not enough for time to market. Verification speedup is also needed. Figure 2.8 shows HLS methodology [3, 10, 11].

HLS is a process to convert untimed behavioral descriptions into efficient hardware that implements that behavior. The inputs to the HLS process are the behavioral description (SystemC, C++, ...etc.) to be synthesized and a technology library that contains the area and delay information of basic operations, a target synthesis frequency, and a set of synthesis directives in the form of pragmas to allow the designer to control the synthesis process. For example, to control how to synthesize arrays (RAM or registers), loops (unroll, partially unroll, not unroll, or pipeline), and functions (inline or not). The HLS process then parses the behavioral description and constraints and performs three main steps: **(1) resource allocation, (2) scheduling, and (3) binding.** In the resource allocation stage, the number and type of hardware resources are extracted (functional units, storage elements, type of buses). In the scheduling phase, the different operations in the behavioral description are assigned to individual clock steps based on the number of available resources. Depending on whether the time constraint or the area constraint is more difficult to meet, resource constrained scheduling or time constrained scheduling has to be chosen. Finally, in the binding stage, the hardware resources are bound to different **operations** in the scheduled operations [13, 14].

It is important to mention that, although HLS facilitates the procedure of converting code developed in C/C++ to an HDL module, it usually results in low-performance kernels. because the codes are designed for sequential execution.

SystemC is implemented as a C++ class library, which provides the language elements and an event-driven simulation kernel. The language comprises constructs for modularization and structuring, for hardware, software, and communication modeling, and for synchronization and coordination of concurrent processes. From a structural point of view, a SystemC design is a set of modules, connected by channels. The structure strictly separates between computation and communication units (*i. e.*, modules and channels) and is highly flexible due to a communication concept that allows transaction level modeling and communication refinement. The event-driven simulation kernel regards the SystemC design as a set of concurrent processes that are synchronized and coordinated by events and communicate through channels (Fig. 2.4). Recently, **tensor flow** can be synthesized to RTL.

Although HLS facilitates the procedure of converting code developed in C/C++ to an HDL module, it usually results in low-performance kernels because the codes are designed for sequential execution (see Figs. 2.11 and 2.12).

2.1.4 Data-Centric/Memory-Centric SoC Design

Data-centric System-on-Chip (SoC) design is an approach to designing electronic systems that focuses on the efficient processing, storage, and communication of data. It involves building a system that is optimized for handling large amounts of data, rather than just performing specific functions. In a data-centric SoC design, the architecture is designed to support data-centric workloads, with an emphasis on data movement and data storage. The system is optimized for data-intensive applications, such as machine learning, big data analytics, and high-performance computing. Data-centric SoC design typically involves the integration of several hardware and software components, such as processors, memory, storage, and I/O interfaces. These components are designed to work together seamlessly to optimize data flow and minimize bottlenecks. One of the key advantages of data-centric SoC design is that it can significantly improve the efficiency and performance of data-intensive applications, compared to traditional SoC designs. This is because data-centric SoC designs can minimize the movement of data between different components, reducing latency and improving throughput. Overall, data-centric SoC design is an important approach to building high-performance, efficient electronic systems that are optimized for handling large amounts of data [31].

Memory-centric System-on-Chip (SoC) design is an approach to designing electronic systems that places a strong emphasis on optimizing the memory architecture of the system. The goal of memory-centric SoC design is to build a system that can efficiently manage and process large amounts of data by leveraging the most suitable memory

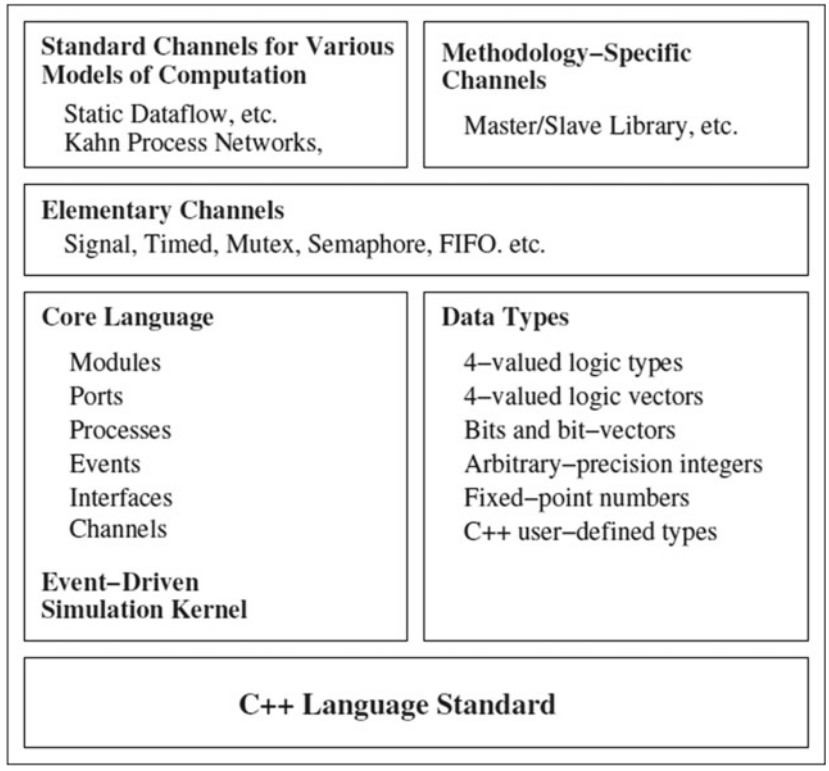
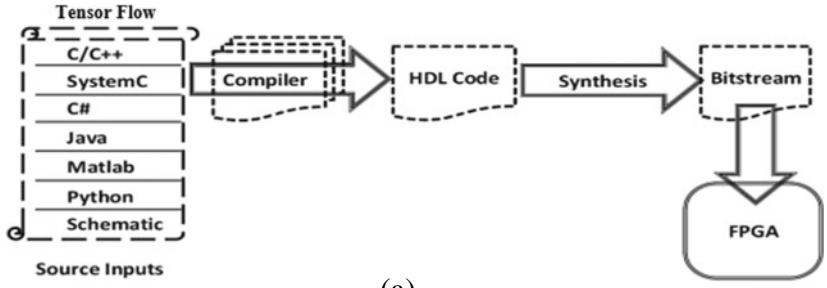
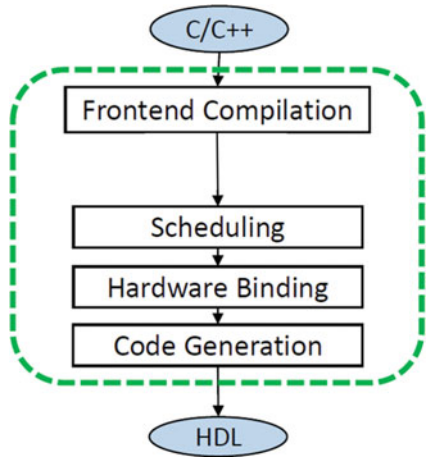


Fig. 2.11 SystemC language architecture

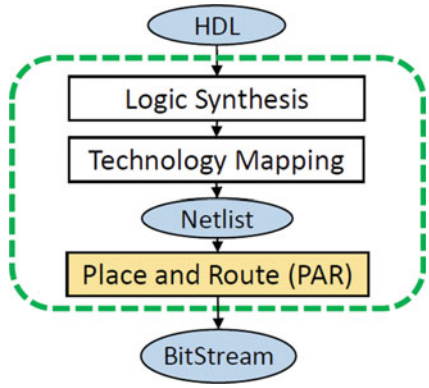
technology for each specific use case. In a memory-centric SoC design, the memory architecture is designed to be the central point of the system, with other components optimized to support its performance. This means that the memory hierarchy is carefully designed to match the requirements of the application, including the size, speed, and latency of different memory types. Memory-centric SoC design typically involves the integration of several memory types, such as DRAM, SRAM, flash, and non-volatile memory. These memory types are selected and configured to optimize the system's performance, power consumption, and cost. One of the key advantages of memory-centric SoC design is that it can significantly improve the efficiency and performance of memory-intensive applications, such as machine learning, computer vision, and graphics processing. This is because memory-centric SoC designs can minimize memory access time and improve data bandwidth, reducing the overall latency of the system. Overall, memory-centric SoC design is an important approach to building high-performance, efficient electronic systems that are optimized for managing and processing large amounts of data. It allows designers to



(a)



(b)



(c)

Fig. 2.12 **a** HLS methodology. The C/C++ codes should be a hardware-friendly codes, **b** compiler step, **c** RTL implementation flow

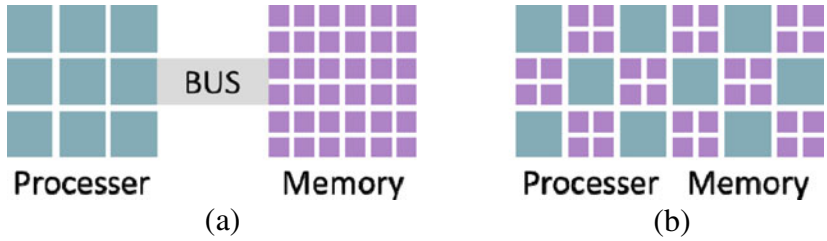


Fig. 2.13 Conventional processing (**a**) versus (**b**) PIM concept

select and configure the most suitable memory types for each specific application, leading to improved performance, reduced power consumption, and lower costs [26–30].

Computation is already orders-of-magnitude cheaper than moving data. Processing-in-memory (**PIM**) designs cores enjoy fast, high-bandwidth access to nearby memory (Fig. 2.13). Processing-in-memory (PIM) is a computing architecture where computation is performed within the memory system, rather than relying solely on the processor or central processing unit (CPU) to perform computations. In PIM architecture, the memory and computation functions are tightly integrated, and the memory is designed to perform some of the computation functions typically performed by the CPU. Traditionally, the CPU fetches data from memory, performs computations, and writes the result back to memory. In PIM architecture, the memory can be designed to perform some of the computations, reducing the need for data movement between the CPU and memory. This can lead to significant improvements in system performance and power efficiency, particularly for memory-intensive applications. PIM architecture can be implemented in different ways, depending on the specific application requirements. For example, some PIM architectures use specialized memory units, such as content-addressable memory (CAM) or associative memory, to perform specific computations. Others use specialized processing elements integrated into the memory itself, such as multipliers, adders, and logic gates. PIM architecture has the potential to overcome the memory bandwidth bottleneck that is often encountered in conventional computer architectures. By bringing computation closer to the data, PIM architecture can reduce the amount of data that needs to be transferred between the CPU and memory, thereby reducing data movement and latency.

2.1.4.1 Hybrid-Centric Platforms for SoC Design

Hybrid platforms for System-on-Chip (SoC) design refer to the integration of different types of hardware platforms in a single design environment. This approach combines the advantages of multiple platforms, such as field-programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), and processors, to create a more efficient and flexible SoC design. Hybrid platforms for SoC design can be used in various applications, such as wireless communication, image processing, and artificial intelligence. For

example, a hybrid platform for wireless communication may include an FPGA for hardware acceleration of digital signal processing tasks, a processor for running control and management software, and an ASIC for implementing custom communication protocols. One advantage of hybrid platforms for SoC design is that they can reduce the development time and cost. By using pre-designed components and leveraging the strengths of multiple platforms, designers can achieve a more efficient and optimized SoC design. Additionally, hybrid platforms can provide a higher degree of flexibility compared to using a single platform, as different types of hardware can be combined and reconfigured as needed. However, hybrid platforms for SoC design also pose some challenges, such as the need to integrate different hardware components and ensure their compatibility, as well as the complexity of managing the different design tools and methodologies associated with each platform. Hybrid platforms such as ZYNQ-SoC combines the software flexibility of ARM processors with the parallel processing capability of reconfigurable hardware. **Real-time low latency** applications can be offloaded to the programmable logic [1].

2.1.5 Hardware Accelerators-Centric SoC Design

Using general-purpose hardware such as CPUs and GPUs has some limitations. CPUs lack the compute throughput needed for kernels with high operational intensities, while GPUs require significant amount of power. This has led to tremendous efforts in building specialized hardware (accelerators) to maximize data reuse for high usage compute kernels. Dedicated hardware accelerators popular for imaging, vision, and machine learning (ML) applications. One of the most popular accelerators being the Google Tensor Processing Unit (**TPU**) [12]. Cloud TPU v2 Chip Architecture is depicted in Fig. 2.14.

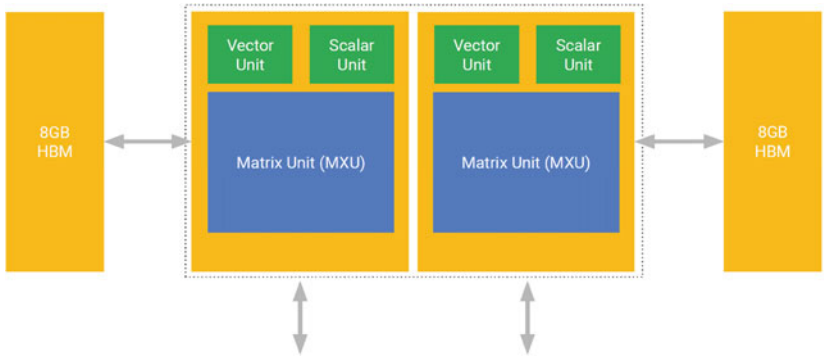


Fig. 2.14 Cloud TPU v2 Chip Architecture. Scalar unit for control and data flow. Vector unit for pooliong, activation, and dropout. Matrix unit for convolutions

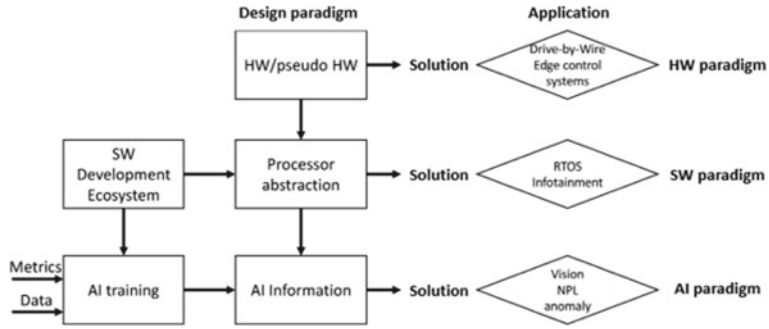


Fig. 2.15 AI as a new design paradigm

The TPU targets accelerating matrix multiplications. Heterogeneous computing combines conventional software-based processing on host CPUs with specialized hardware accelerators to perform for higher performance or better efficiency. The end of **Dennard's scaling** and the forthcoming emerging end of Moore's Law are the key enablers for heterogeneous design. Also, as applications change rapidly, we need reconfigurable accelerators. Noting that cost is an overhead for accelerators. **AI** is a new design paradigm [25] (Fig. 2.15).

Hardware accelerators-centric SoC design is an approach that involves designing a system-on-chip (SoC) with one or more specialized hardware accelerators that are optimized for specific tasks. These accelerators are designed to provide higher performance and energy efficiency compared to software-based solutions. Examples of hardware accelerators-centric SoC designs include:

1. Graphics processing units (GPUs): GPUs are specialized hardware accelerators designed to perform the complex mathematical computations required for graphics rendering. GPUs are commonly used in gaming consoles, desktop computers, and mobile devices to improve the performance of graphics-intensive applications.
2. Digital signal processors (DSPs): DSPs are specialized hardware accelerators designed to perform digital signal processing tasks such as audio and video processing, speech recognition, and image processing. DSPs are commonly used in consumer electronics such as smartphones, digital cameras, and home theater systems.
3. Neural processing units (NPUs): NPUs are specialized hardware accelerators designed to perform artificial neural network computations for machine learning and artificial intelligence applications. NPUs are commonly used in edge devices such as smartphones and smart home devices to improve the performance of machine learning applications.
4. Cryptographic accelerators: Cryptographic accelerators are specialized hardware accelerators designed to perform encryption and decryption operations for data security

applications. Cryptographic accelerators are commonly used in network devices such as routers and firewalls, as well as in mobile devices and smart cards.

The main advantage of hardware accelerators-centric SoC design is that it can significantly improve the performance and energy efficiency of specialized applications. By offloading computation-intensive tasks to specialized hardware accelerators, the CPU can focus on other tasks, resulting in overall faster and more efficient computing. Additionally, hardware accelerators can reduce the cost of implementing specialized functionality by reducing the amount of CPU time and energy required. However, hardware accelerators-centric SoC design also poses some challenges, such as the need to design and optimize specialized hardware accelerators for specific tasks, and the need to integrate them into the SoC design. Additionally, hardware accelerators-centric SoC designs can be more complex than traditional SoC designs, requiring specialized knowledge and expertise in hardware design and optimization. Artificial neural networks have become a widely recognized tool across many scientific disciplines. This recognition has led to the development of deep learning techniques and specialized hardware, such as GPUs with tensor cores. The popularity of neural networks lies in their ability to be viewed as multiple layers of approximation functions, represented by a series of weighted multiplications and additions called neurons. Fully connected deep neural networks have gained the most attention, especially in image processing. Given the flexibility of neural networks in approximation and the availability of specialized hardware, researchers have explored the use of neural acceleration to approximate computationally expensive parts of code. The approach involves replacing small subroutines of code with an inexpensive neural network approximation during execution, which costs less due to optimized hardware. To achieve this, programmers would identify specific subroutines of interest and provide a small training set of inputs and outputs for the subroutine. At compile time, a search space algorithm would select a simple and low-cost neural network to be used as an approximation for the subroutine at execution time.

2.1.6 ASIC-Based SoC Design

Physical design converts a circuit description into a geometric description. This description is used to manufacture a chip. Geometric shapes which correspond to the patterns of metal, oxide, or semiconductor layers that make up the components of the integrated circuit. It is top view of the cross-sectional device [5]. As the number of transistors in IC was increasing dictated by Moors law, the size of transistors is also decreasing almost double in size since 1971. The projected transistor size in 2023 is to be 2–3 nm [16]. ASIC performance is optimized for a certain application domain ASIC energy efficiency against GPU and CPU is shown in Fig. 2.16. ASIC-based accelerators provide satisfactory performance and power consumption; however, they only execute a fixed program, and

it is hard or impossible to change their functionality. It is impossible to yield full wafer with zero defects as Silicon and process defects are inevitable even in mature process. So, redundancy is important while fabrication as if a defect happens, hardware will remap and reconnect using extra links (Fig. 2.17). The following are some examples of ASIC-based SoC design [32–35]:

1. Mobile Processors: Many mobile devices, such as smartphones and tablets, use ASIC-based SoC designs to provide high-performance processing while minimizing power

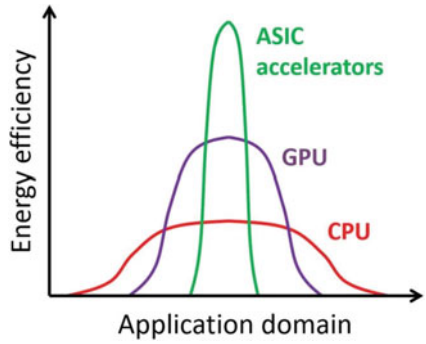


Fig. 2.16 ASIC energy efficiency against GPU and CPU

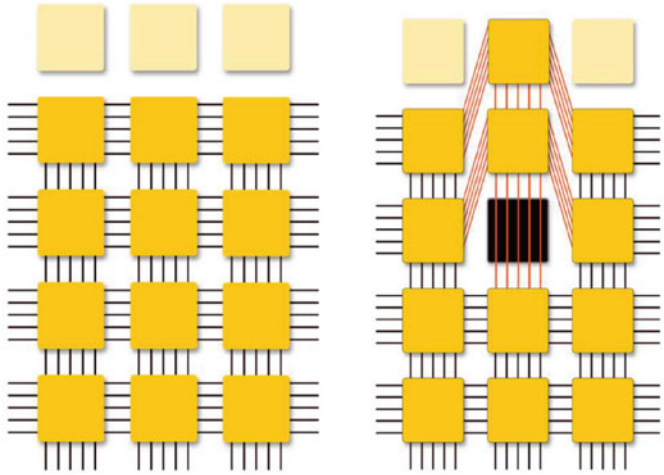


Fig. 2.17 **a** No defects, **b** defects

consumption. Companies like Qualcomm and Apple design custom ASICs to integrate multiple functions, such as graphics processing, video decoding, and wireless communication, into a single chip.
2. Networking Devices: ASIC-based SoC designs are also commonly used in networking devices, such as routers, switches, and network interface cards. These devices require high-speed processing of large amounts of data, and custom ASICs can be designed to handle specific network protocols and traffic patterns.
3. Automotive Systems: ASIC-based SoC designs are increasingly being used in automotive systems, such as advanced driver assistance systems (ADAS) and autonomous vehicles. Custom ASICs can be designed to perform tasks such as image processing, sensor fusion, and decision-making, all of which are critical for safe and efficient operation of autonomous vehicles.
4. Medical Devices: ASIC-based SoC designs are also used in medical devices, such as implantable devices, diagnostic tools, and monitoring systems. Custom ASICs can be designed to provide precise control and monitoring of medical devices, as well as to process and analyze large amounts of patient data in real-time.
5. Industrial Control Systems: ASIC-based SoC designs are also used in industrial control systems, such as programmable logic controllers (PLCs) and robotic control systems. Custom ASICs can be designed to perform real-time control and monitoring of industrial equipment, as well as to analyze and process data from sensors and other sources.

2.1.7 PCB-Based SoC Design

Standard logic ICs provides fixed function devices which can be connected together on PCB to implement a system. Standard logic ICs has limited speed and limited number of pins. For single-sided PCB, components are on one side and conductor pattern on the other side. Routing is very difficult. For double-sided PCB, conductor patterns are on both sides of the board, and we connect between the two layers through vias. Via is a hole in the PCB, filled or plated with metal and touches the conductor pattern on both sides. Since routing is on both sides, double-sided boards are more suitable for complex circuits than single-sided ones. It is always better to minimize the number of vias. For multilayer PCB, these boards have one or more conductor pattern inside the board. Several double-sided boards are glued together with insulating layers in between. For interlayer connections, there is blind via to connects an inner layer to an outer layer and buried via to connects two inner layers. The layers are classified as: Signal layers, Ground plane, and Power plane. Power planes may have special restrictions such as wider track widths. At the heart of high speed, PCB design is an issue of interference. The faster your data rates are, the more issues you have trying with to protect the integrity of your signals. Most of these problems stem from electromagnetic radiation. Signal integrity (SI), power

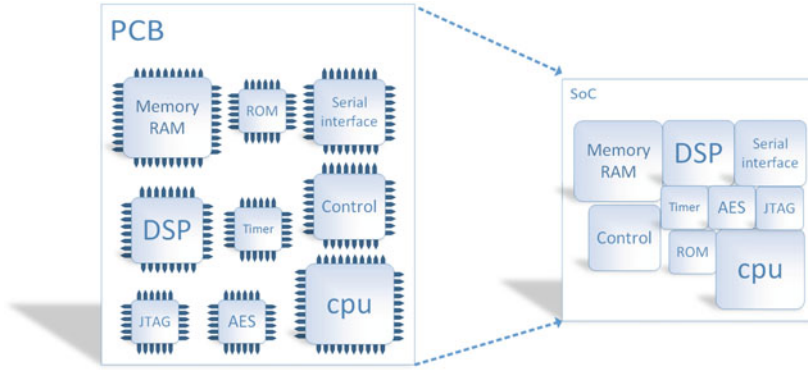


Fig. 2.18 From PCB to SoC

integrity (PI), electromagnetic compatibility (EMC), and design for manufacturing (DFM) verification are very crucial for PCB design. DFM, ensures that the PCB design meets the requirements imposed by the manufacturing process, based on characteristics such as minimum trace width, minimum distance between traces, minimum hole width and more, which need to be verified before the circuit board goes into production. To achieve this, the PCB layout level goes through a set of rules or controls called Design Rule Checking (DRC).The trend now is moving from PCB to SoC as possible (Fig. 2.18). To enable faster time to market, common IP or technology that already has been silicon-proven can be utilized. Time and resources needed to redesign IP can be saved by integrating existing tested technologies and reusing common IP across products. PCB can provide a cost-effective alternative to custom ASIC-based designs. The following are some examples of PCB-based SoC design:

- Embedded Systems: PCB-based SoC design is commonly used in embedded systems, such as industrial control systems, medical devices, and automotive systems. These systems typically require multiple SoC components, such as microcontrollers, sensors, and communication modules, to be integrated onto a single PCB.
- IoT Devices: PCB-based SoC design is also used in Internet of Things (IoT) devices, such as smart home devices, wearable devices, and smart sensors. These devices require multiple components, such as microcontrollers, wireless communication modules, and sensors, to be integrated onto a single PCB.
- Consumer Electronics: PCB-based SoC design is used in a wide range of consumer electronics products, such as smartphones, tablets, and gaming consoles. These devices require multiple SoC components, such as processors, memory, and communication modules, to be integrated onto a single PCB.

- Communication Systems: PCB-based SoC design is used in communication systems, such as routers, switches, and base stations. These systems require multiple SoC components, such as processors, memory, and communication modules, to be integrated onto a single PCB.
- Robotics: PCB-based SoC design is also used in robotics applications, such as autonomous vehicles and industrial robots. These systems require multiple SoC components, such as microcontrollers, sensors, and actuators, to be integrated onto a single PCB.

There are several types of failures that can occur in a printed circuit board (PCB) after fabrication. Some of the common types of failures include:

- Electrical Failures: Electrical failures can occur due to various reasons, such as short circuits, open circuits, voltage spikes, and power surges. These types of failures can be caused by faulty components, incorrect wiring, and inadequate insulation.
- Mechanical Failures: Mechanical failures can occur due to issues such as warping, cracking, delamination, and bending. These types of failures can be caused by improper handling, excessive stress, or manufacturing defects.
- Thermal Failures: Thermal failures can occur due to issues such as overheating, thermal cycling, and thermal shock. These types of failures can be caused by improper design, inadequate cooling, or improper component selection.
- Chemical Failures: Chemical failures can occur due to issues such as corrosion, oxidation, and contamination. These types of failures can be caused by exposure to chemicals or environmental factors, such as humidity or moisture.
- Environmental Failures: Environmental failures can occur due to issues such as exposure to extreme temperatures, humidity, or moisture. These types of failures can be caused by improper storage or usage conditions.
- Manufacturing Defects: Manufacturing defects can occur due to issues such as improper etching, drilling, or soldering. These types of failures can be caused by errors in the manufacturing process, inadequate quality control, or improper equipment maintenance.

2.1.8 Application-Centric SoC Design

Application-centric SoC design is an approach in which the design of a System-on-a-Chip (SoC) is optimized for a specific application or use case. This approach focuses on developing an SoC that can efficiently and effectively run a particular application or set of applications, rather than trying to create a more general-purpose design that can handle a wide range of tasks. The goal of an application-centric SoC design is to maximize performance, power efficiency, and cost-effectiveness for a specific application or set of

applications. This is achieved by customizing the hardware and software components of the SoC to meet the specific requirements of the target application. Here are some examples of application-centric SoC designs:

- Autonomous Vehicles: Autonomous vehicles require a high level of computational power and specialized hardware to enable perception, decision-making, and control functions. An application-centric SoC design for autonomous vehicles would optimize the hardware components such as sensors, vision processing, and connectivity, to enable safe and efficient autonomous driving. The software components would also be tailored to support the specific requirements of autonomous driving applications, such as real-time sensor fusion, machine learning, and deep learning algorithms.
- Wearable Devices: Wearable devices, such as smartwatches, fitness trackers, and medical wearables, require a combination of low power consumption, compact form factor, and high-performance computing to deliver an optimal user experience. An application-centric SoC design for wearables would optimize the hardware components, such as sensors, processors, and wireless connectivity, to deliver the required performance while minimizing power consumption. The software components would also be customized to support specific wearable applications, such as health monitoring, fitness tracking, and messaging.
- Smart Home Automation: Smart home automation devices, such as smart speakers, smart thermostats, and smart security systems, require specialized hardware and software components to enable seamless integration and interoperability with other devices and platforms. An application-centric SoC design for smart home automation would optimize the hardware components, such as sensors, connectivity, and voice recognition, to enable easy and intuitive interaction with the user. The software components would also be tailored to support the specific requirements of smart home applications, such as natural language processing, cloud connectivity, and security protocols.
- high-performance gaming console: would be optimized for running complex graphics and high-speed networking. The hardware components of the SoC, such as the graphics processing unit (GPU), would be designed to deliver high-performance graphics processing, while the software components, such as the operating system and device drivers, would be tailored to support the specific requirements of gaming applications.

2.1.9 Conclusions

This chapter introduces hardware design methodologies, which are critical in the development of hardware systems. The four primary hardware design methodologies for modeling a hardware IP are FPGA-based modeling, ASIC-based modeling, processor-based modeling, and PCB-based modeling. Engineers must carefully consider the advantages and disadvantages of each technology before selecting the most appropriate methodology for

their specific project. These methodologies provide a systematic approach for designing, simulating, verifying, and validating complex hardware systems. They enable engineers to model the system and identify any issues before committing to physical implementation. This helps to reduce costs and improve the overall quality of the final product. The chapter also includes references to several case studies and research papers that demonstrate the practical applications of these methodologies in real-world scenarios. This chapter provides a comprehensive overview of hardware design methodologies and their importance in modern engineering projects.

References

1. Pezzarossa, L.; Schoeberl, M.; Sparso, J. A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems. In Proceedings of the 2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC), Toronto, ON, Canada, 16–18 May 2017; pp. 92–100.
2. R. Woods, J. McAllister, G. Lightbody, and Y. Yi, FPGA-based Implementation of Signal Processing Systems. Wiley Publishing, 2nd ed., 2017.
3. Matlab simulink. https://www.mathworks.com/products/simulink.html.
4. Xilinx. https://www.xilinx.com.
5. D. Markovic, C. Chang, B. Richards, H. So, B. Nikolic, and R. W. Brodersen, \ASIC design and verification in an FPGA environment," in CICC 2007, pp. 737{740, Sept 2007.
6. Q. Li et al., "Implementing neural machine translation with bidirectional GRU and attention mechanism on FPGAs using HLS," in Proc. of Asia and South Pacific Design Automation Conference (ASPDAC), 2019.
7. D. Petrisko, F. Gilani, M. Wyse, et al., "Blackparrot: An agile open source RISC-V multicore for accelerator SoCs," IEEE Micro, vol. 40, no. 4, pp. 93–102, 2020.https://doi.org/10.1109/MM.2020.2996145
8. H. Genc, S. Kim, A. Amid, et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in Proceedings of the 58th Annual Design Automation Conference (DAC), 2021.
9. D. Gajski, N. Dutt, A. Wu, S. Lin: High-level Synthesis – Introduction to Chip and System Design. Kluwer Academic Publishers, 1992.
10. David C. Black and Jack Donovan. SystemC: From the Ground Up. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
11. Accellera. System Verilog 3.1 Accellera's Extensions to Verilog. www.systemverilog.org.
12. N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al., "In-datacenter performance analysis of a tensor processing unit," in Proceedings of the 44th annual international symposium on computer architecture, 2017, pp. 1–12.
13. You, M.K.; Song, G.Y. Case study: Co-simulation and co-emulation environments based on SystemC SystemVerilog. In Proceedings of the 2009 IEEE Region 10 Conference (TENCON 2009), Singapore, 23–26 November 2009; pp. 1–4.
14. Wang, Z.; Schafer, B.C. Machine Leaming to Set Meta-Heuristic Specific Parameters for High-Level Synthesis Design Space Exploration. In Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 20–24 July 2020; pp. 1–6.

15. R. Tessier, K. Pocek, and A. DeHon, “Reconfigurable Computing Architectures,” Proceedings of the IEEE, vol. 103, no. 3, pp. 332–354, 2015.
16. Wolf, W., Modern VLSI Design, Prentice Hall (2008) 3rd ed.
17. Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite, Xilinx Corporation, July 2012.
18. R. Dunkley, “Supporting a wide variety of communication protocols using partial dynamic reconfiguration,” Proc. IEEE AUTOTESTCON 2012, pp. 120–125, 2012.
19. N. Marques, H. Rabah, E. Dabellani, and S. Weber, “Partially reconfigurable entropy encoder for multi standards video adaptation,” in Consumer Electronics (ISCE), 2011 IEEE 15th International Symposiumon, june 2011, pp. 492 –496.
20. Wang Lie, Wu Fengyan, “Dynamic Partial Reconfiguration in FPGA”, Third International Symposium on Intelligent Information Technology Application, IEEE Computer Society, 2009.
21. K. Salah, An Area Efficient Multi-mode Memory Controller Based on Dynamic Partial Reconfiguration. In 2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON). IEEE (2017), pp. 328–331.
22. Mohamed, K.S. Reconfigurable and Heterogeneous Computing. In: Neuromorphic Computing and Beyond. Springer,2020.
23. Wang, C., Lou, W., Gong, L., Jin, L., Tan, L., Hu, Y., Li, X., Zhou, X., Reconfigurable hardware accelerators: Opportunities, trends, and challenges. arXiv preprint arXiv:1712.04771, 2017.
24. Cong, J., Fang, Z., Lo, M., Wang, H., Xu, J., Zhang, S., Understanding performance differences of FPGAs and GPUs. 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 93–96, 2018.
25. R. Razdan, M. İ. Akbaş, R. Sell, M. Bellone, M. Menase and M. Malayjerdi, "PolyVerif: An Open-Source Environment for Autonomous Vehicle Validation and Verification Research Acceleration," in *IEEE Access*, https://doi.org/10.1109/ACCESS.2023.3258681.
26. “Memory Systems: Cache, DRAM, Disk,” by Bruce Jacob, Spencer Ng, and David Wang, Morgan Kaufmann, 2007.
27. “High-Performance Memory Systems,” by Hyesoon Kim, Onur Mutlu, and Mike Marty, Morgan & Claypool Publishers, 2014.
28. “Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems,” by Benny Akesson, KTH Royal Institute of Technology, 2012.
29. “Heterogeneous System Architecture: A New Compute Platform Infrastructure,” by Wen-mei Hwu and Jason Cong, Morgan Kaufmann, 2016.
30. “Memory Management: Algorithms and Implementations in C/C++,” by Bill Blunden, Jones & Bartlett Learning, 2002.
31. “Designing Embedded Systems with PIC Microcontrollers: Principles and Applications,” by Tim Wilmshurst, Newnes, 2009.
32. “ASIC Design in the Silicon Sandbox: A Complete Guide to Building Mixed-Signal Integrated Circuits,” by Keith Barr, Wiley-IEEE Press, 2007.
33. “ASIC and FPGA Verification: A Guide to Component Modeling,” by Richard Munden, Morgan Kaufmann, 2004.
34. “Design of System on a Chip: Devices & Components,” by Ricardo Reis and Marcelo Lubaszewski, Springer, 2018.
35. “ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies,” by Ashok B. Mehta, Springer, 2010.
36. Salah, K., AbdelSalam, M. (2019). Solving Sparse Matrices: A Comparative Analysis Between FPGA and GPU. In: Ntalianis, K., Croitoru, A. (eds) Applied Physics, System Science and Computers II. APSAC 2017. Lecture Notes in Electrical Engineering, vol 489. Springer, Cham.

HW/SW Co-Exploration and Co-Design

3

Now, SoC vendors face a significant challenge to keep the development time as short as possible as in order to win the competition, they need to deliver a competitive product ahead of the competitors at a lower cost. Moreover, they need to keep up with the rapid evolution of different IPs standards.

Traditional design techniques, based on independent design of HW/SW components are no longer sufficient to support the integration of subparts of such SoCs. Here, HW/SW co-design methodologies, where designers can easily check system-level constraints satisfaction and evaluate cost/performance trade-off for different architectural solutions, are of renovated relevance.

Assume we need to design an algorithm for a SoC [8]. Then we have three options: HW, SW, and HW/SW. Each has their own advantages and weaknesses (Table 3.1):

A. **Implementing the entire algorithms in Hardware**: the hardware design promises a faster speed of operation, more reliable process, but requires longer development time and less flexibility. The hardware can be divided into two categories, i.e. fixed hardware and the reconfigurable hardware. The fixed hardware includes discrete components, daughter boards, and PCB-based extension board. The fixed hardware design requires a lot of development time, special skill of design, and can only a limited function. While the reconfigurable hardware is a platform that implement a particular hardware model (HDL). FPGA is a well-recognized platform to deal with design issues such as reconfigurability, development time, and support of concurrent software development. HLS tools can also be used to design the hardware part [7].
B. **Implementing the entire algorithms in software**: the advantage of software design is higher flexibility, the ability to support more functions, and shorter development time.

K. S. Mohamed, *Heterogeneous SoC Design and Verification*, Synthesis Lectures on Digital Circuits & Systems, https://doi.org/10.1007/978-3-031-56152-8_3

Table 3.1 Advantages and disadvantages of HW, SW, and HW/SW co-design implementation

	HW implementation	SW implementation	HW/SW implementation
Advantages	Provides higher performance via hardware speeds and parallel execution of operations	• High flexibility since firmware can be updated anytime • May run on high-performance processors at low cost (due to high-volume production)	• High Speed • Flexibility • Reduce Cost
Disadvantages	• No flexibility after fabrication • High cost: Usually more expensive than software solutions	Usually slower than hardware	Complex in design

However, design in software will produce slower speed of operation and requires more memory resources. We can divide the software into two kinds, i.e., the application software and the embedded software. The application software is developed in X86 CPU (host PC) as a higher layer software, while the embedded software (firmware) is developed in embedded processor (e.g., ARM) as lower-level software.

C. **Implementing the algorithms in Hardware and software.*i.e.*, partitioning the design between SW/HW**:

HW/SW co-design is a methodology that enables hardware and software to be designed concurrently. We can choose time critical tasks and computationally complex algorithms to be implemented in HW and non-critical tasks to be implemented in SW. Before starting partitioning. you need to draw a block diagram for the main module/algorithm with a detailed dataflow along with showing the data going in and out for each sub-module. HW/SW co-design allows the trade-off between attainable performance and flexibility.

Hardware/software (HW/SW) co-design is the process of designing a system that consists of both hardware and software components. The objective of HW/SW co-design is to optimize the performance, cost, and reliability of the system by designing the hardware and software components together, rather than separately. The following are the key objectives of HW/SW co-design:

- Improved Performance (power, area, speed, memory space): HW/SW co-design can help improve the performance of a system by optimizing the interaction between the hardware and software components. For example, the hardware can be designed to accelerate certain software tasks, such as image processing or signal filtering, resulting in faster overall system performance.

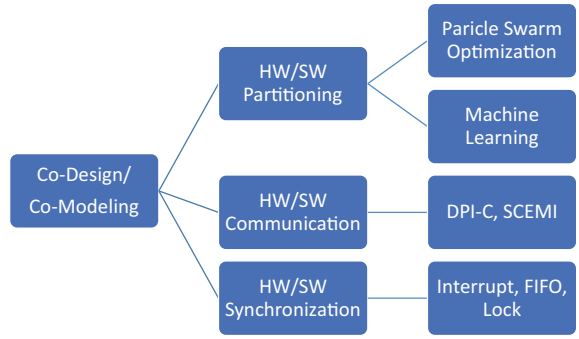


Fig. 3.1 Co-modeling anatomy

- Reduced Cost: HW/SW co-design can help reduce the cost of a system by optimizing the hardware and software components to work together efficiently. For example, the use of hardware accelerators can reduce the amount of processing required by the software, resulting in lower system costs.
- Reduced Time-to-Market: HW/SW co-design can help reduce the time-to-market of a system by allowing hardware and software components to be designed and tested concurrently. This can help identify and resolve issues earlier in the development cycle, resulting in a faster time-to-market.
- Increased Flexibility: HW/SW co-design can help increase the flexibility of a system by allowing hardware and software components to be designed to work together seamlessly. This can help reduce the complexity of the system and make it easier to modify or update in the future.
- Improved Reliability: HW/SW co-design can help improve the reliability of a system by allowing hardware and software components to be designed to work together in a more fault-tolerant manner. For example, redundant hardware components can be used to ensure that the system remains operational even in the event of a hardware failure.

Co-Modeling anatomy is shown in Fig. 3.1. While hardware/software (HW/SW) co-design can bring numerous benefits, it also presents several challenges. Some of the main challenges of HW/SW co-design are [17–21]:

1. Complexity: HW/SW co-design involves designing two highly complex and interdependent systems. Coordinating the development of the hardware and software components, and ensuring that they work seamlessly together, can be a daunting task.
2. Communication and synchronization: Since HW/SW co-design involves teams with different backgrounds and expertise, there can be communication gaps between the hardware and software teams. These communication gaps can lead to misunderstandings and delays.

3. Timing: HW/SW co-design requires tight coordination between the hardware and software teams to ensure that they are working in sync. This coordination can be challenging, especially when there are changes to the design.
4. Verification: The verification of the hardware and software components is critical to ensure that the system is functioning as intended. However, verification can be challenging, especially when dealing with complex systems that are highly interdependent.
5. Intellectual Property (IP) Ownership: HW/SW co-design often involves the use of third-party IP, which can create ownership issues. This is especially true when the IP is being used in both the hardware and software components.
6. Tool Integration: Coordinating the use of different design tools and languages can be a challenge in HW/SW co-design. Integrating these tools can be complex and time-consuming.
7. Hardware-Software Partitioning: Deciding how to partition the system between hardware and software can be challenging. This decision can have a significant impact on the performance, cost, and reliability of the system.
8. Design Trade-offs: HW/SW co-design often requires making design trade-offs to optimize the system. These trade-offs can be challenging to make, especially when they involve conflicting objectives such as performance and cost.

3.1 HW/SW Partitioning

Several design methods can be explored to perform the implementation of a specific IP. Usually, the SW solutions are more flexible and don't require a lot of time to verify and validate the IP which is not the case of the HW implementation. This last is more tended to satisfy real time constraint at low power cost rather than software at the cost of increasing the simulation time. In order to ensure the best trade-off between flexibility and performances, the HW/SW concept is considered as a best solution which combines a microprocessor system and a programmable logic both in the same chip [1]. Co means hybrid.

The idea of HW/SW co-design came up in the early 1990s. The classical design flow for Co HW/SW systems starts with an abstract specification. The specification is then partitioned into hardware and software modules. The decision which parts of the systems should be implemented in hardware and which in software is often guided by experience (manual). The separation of hardware and software design flows severely increases the system integration effort and makes it difficult and error prone [13].

The HW/SW partitioning steps can be summarized in Fig. 3.2. Partitioning maps or allocates design parts to either hardware or software. This mapping is left to the user as it can be done manually or automatically using particle swarm optimization or machine

Fig. 3.2 HW/SW partitioning steps

learning [27]. Performance feedback is provided by simulation. Moreover, there are two major approaches for partitioning [2]:

1. Software-Oriented Partitioning: Start with all functionalities in software and move portions into hardware which are time-critical and cannot be allocated to software
2. Hardware-Oriented Partitioning: Start with all functionalities in hardware and move portions into software implementation

Also, we can describe the whole system behavioral then partitioning it structurally or from the beginning we partition them structurally (Fig. 3.3). In general, high-level partitioning can be done by hand or can be automated using various techniques such as machine learning or particle swarm optimization. HW/SW partitioning from specification to integration is shown in Fig. 3.4. Typically, the designer starts from C/C++/SYSTEMC model. Then it is segmented into software part running on processors, and hardware part running on accelerator hardware. Finding the right partitioning for a system is a complex task and usually requires multiple iterations. Exploring possible hardware/software partitioning scenarios is a time consuming and challenging job. In general, Processor can offload any heavy-power task to HW accelerator. One methodology is calculating the power at SYSTEMC level then modeling the parts with high power into hardware i.e., offloading them to a specific accelerator [22–25].

HW/SW partitioning can be performed at different levels of abstraction, such as system level, module level, or function level. At the system level, the partitioning determines the overall architecture of the system, and the number and type of hardware and software components required. At the module level, the partitioning determines which modules will be implemented in hardware and which will be implemented in software. At the function level, the partitioning determines which functions will be implemented in hardware and which will be implemented in software.

HW/SW partitioning is a complex optimization problem, as the system performance, power, and cost are often trade-offs of each other. Many different techniques have been proposed for HW/SW partitioning, such as dynamic programming, simulated annealing, genetic algorithms, and heuristic algorithms.

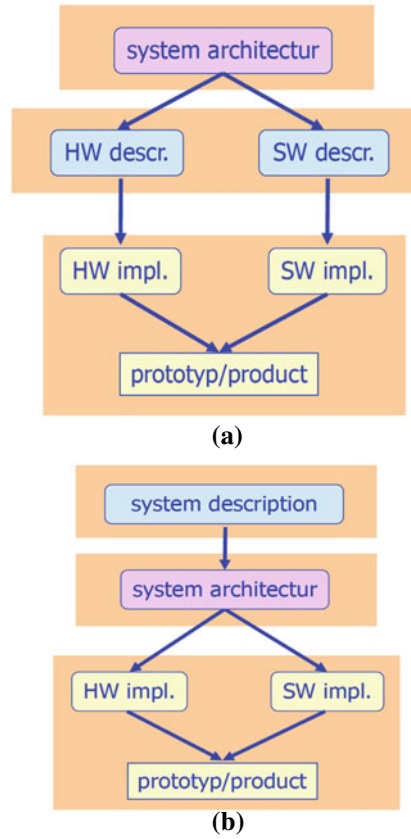


Fig. 3.3 HW/SW partitioning **a** structure first, **b** behavior first. Implementing the testbench as software running on the host PC reduces routing congestion in HW

3.1.1 Design Space Exploration (DSE): HW/SW Co-Exploration Tradeoff

Design space is large and it has many dimensions to explore. So, we need to approach it systematically. We have many design choices, options, and alternatives to try and many parameters to tune for implementation. However, this choice with impact area, delay, energy. i.e., design space is constrained by budget and design goals. So, we need to find one that bests meets design constraints and goals (Fig. 3.5). The larger the design space, the more opportunities to find good solutions. Design Space Exploration (DSE) is the process of exploring and analyzing the design space of a given system in order to identify the best design configurations that meet the specified design constraints and objectives. The design space refers to the range of possible design choices that can be made in terms of different system components, parameters, and design trade-offs. In the context of hardware and software co-design, DSE involves exploring the design space of both the hardware

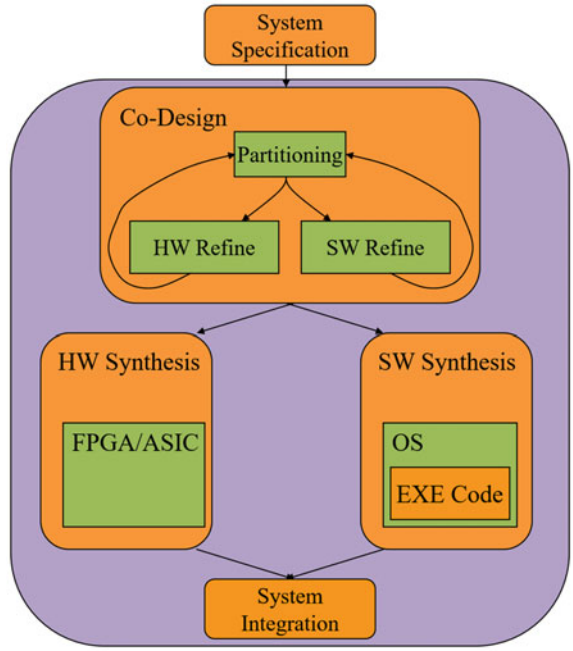


Fig. 3.4 HW/SW partitioning: from specification to integration

and software components to find the optimal configuration that satisfies the desired performance, power, area, and other design criteria. DSE can help designers to identify the best system architecture, hardware and software partitioning, and component selection, among other design decisions. DSE can be performed using various techniques such as analytical modeling, simulation, and optimization. In analytical modeling, mathematical models are used to represent the system behavior and performance. Simulation-based DSE involves using simulation tools to evaluate the system performance under different design configurations. Optimization-based DSE uses mathematical optimization algorithms to search for the optimal design configuration based on the specified design constraints and objectives. DSE is an important process in the design of complex systems as it enables designers to explore the design space in an efficient and systematic manner, leading to improved design quality and reduced design time and cost.

Assume we have three design alternative and when we explore them, we found that design 2 gives the better result for the target criteria (Fig. 3.6). Design space exploration could be security-aware, power-aware, energy-aware, cost-aware, and performance-aware. DSE could be multi-objective. Since the objectives are often in conflict, there cannot be a single optimal solution that simultaneously optimizes all objectives. Therefore, optimal decisions need to be taken in the presence of trade-offs between design criteria [14].

RTL simulation is intractable for design space exploration because it is too time consuming to design and evaluate. Design space exploration options/tools and strategies can be summarized in the following points:

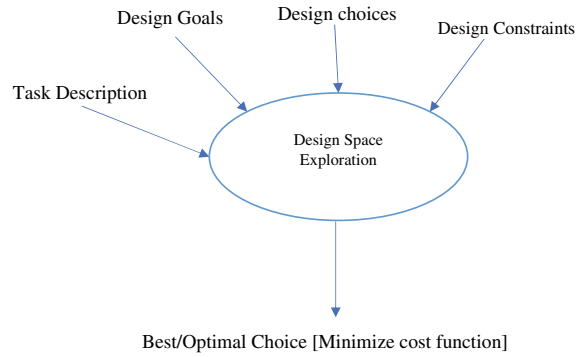


Fig. 3.5 Design space exploration

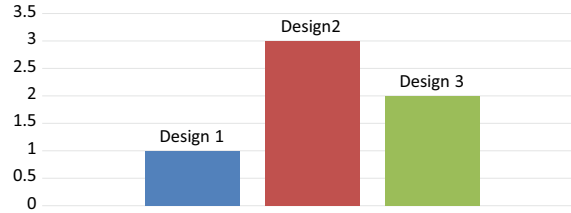


Fig. 3.6 Design space exploration example

For example, consider the design of a digital signal processing (DSP) system for a hearing aid. HW/SW Co-Exploration would involve exploring different hardware architectures (e.g., FPGA, ASIC, custom-designed chip) and different software algorithms (e.g., FFT, FIR filter, IIR filter) to find the best combination that meets the design specifications, such as low power consumption, high processing speed, and low noise. Another example is the design of a self-driving car. In this case, HW/SW Co-Exploration would involve exploring different hardware platforms (e.g., CPU, GPU, FPGA) and different software algorithms (e.g., object detection, path planning, control) to find the optimal combination that achieves the desired level of performance, such as high accuracy, real-time processing, and low latency. The key challenge in HW/SW Co-Exploration is to find the best trade-offs between hardware and software components to achieve the desired performance while minimizing power consumption and cost. This requires sophisticated modeling, simulation, and optimization techniques that can explore the design space efficiently and accurately.

3.1.1.1 System Modeling/Description (Behavioral/Algorithmic, Structural, High Level)

System modeling is the process of creating a simplified representation of a system, which can be used to better understand its behavior, analyze its performance, and design new or improved versions of the system. There are three main types of system modeling: behavioral modeling, structural modeling, and high-level modeling.

A. **Low-level Behavioral models** describe the behavior of the system. Implementation of behavioral models may be in SW or HW. However, some models are better suited for HW design others better for SW They can de described using state diagram or data flow graph (behavioral views of architectural models). It involves creating a mathematical or logical model that can simulate the behavior of the system under different conditions. For example, behavioral modeling can be used to simulate the behavior of a control system for a robot or a temperature control system in a building.
B. **Low-level Structural models** focus on the structure of the system, i.e., its components, modules rather than its behavior. They can de described using block diagrams or netlist. It involves creating a graphical representation of the system, such as a block diagram, that shows the components of the system and how they are interconnected. Structural modeling can be used to design and analyze the layout of printed circuit boards, electrical networks, and mechanical systems.
C. **High-level models**: raise the abstraction level of modeling to give up some accuracy to enable speed, flexibility, and quick simulator design. We can implement system-level to HW/SW partitioning. High-level modeling can be used to design and analyze complex systems, such as software systems, communication networks, and information systems.

3.1.1.2 HW Selection/Implementation (System-on-Chip, ASIC, FPGA, DSP, NP, uC, uP)

There are numerous solutions to select HW. These solutions were discussed in Chapter two. Hardware platform selection refers to the process of selecting the appropriate hardware platform or architecture for a given application or system design. The choice of hardware platform has a significant impact on the overall performance, power consumption, and cost of the system. Therefore, selecting the right hardware platform is critical to achieving the desired system performance and meeting project goals. The selection process involves evaluating various hardware platforms based on their capabilities, performance, power consumption, scalability, ease of use, and cost. The evaluation is typically done based on the system requirements, which may include factors such as the desired processing speed, memory, connectivity, power consumption, and form factor. Hardware platforms can range from simple microcontrollers to complex multi-core processors, FPGAs, GPUs [26], and ASICs. Each platform has its strengths and weaknesses, and the selection process involves determining which platform can best meet the system requirements and provide the optimal balance of performance, power consumption, and cost. The selection process may involve prototyping and testing the system on multiple hardware platforms to determine the optimal solution. The selection process may also involve trade-offs between different hardware platforms, such as between performance and power consumption or cost.

3.1.1.3 HW Design Methods (Languages, HL-Synthesis, RTL-Synthesis)

There are numerous HW design methods. HW design methods refer to the different approaches and techniques used in designing hardware circuits and systems. These methods include hardware description languages, high-level synthesis, and register transfer level (RTL) synthesis. Hardware description languages (HDLs) are programming languages used to describe hardware circuits and systems. HDLs, such as Verilog and VHDL, allow designers to describe hardware circuits using code, similar to how software is developed using programming languages. HDLs provide a way to describe the functionality, behavior, and structure of hardware circuits, and they can be used to simulate and verify the design before implementation. High-level synthesis (HLS) is a design methodology that allows designers to create hardware designs from high-level specifications written in a programming language such as C or C++. HLS tools automatically generate hardware designs from the high-level specifications, which can significantly reduce design time and improve design quality. HLS is particularly useful for designing complex digital signal processing (DSP) systems and other data-intensive applications. RTL synthesis is a design methodology that converts a high-level hardware description into a register transfer level (RTL) description. RTL is a hardware design abstraction that describes the behavior of a circuit in terms of the flow of data between registers. RTL synthesis tools automatically generate RTL descriptions from high-level descriptions or HDLs. RTL synthesis is used to optimize the design for area, speed, and power consumption, and it is commonly used in the design of digital logic circuits, microprocessors, and other digital systems.

Hardware modeling languages (HMLs) are programming languages used to describe the behavior, structure, and functionality of digital hardware systems. HMLs provide a formal and precise method to describe hardware circuits and systems, and they are used to create models that can be simulated and tested before being implemented in hardware. There are several types of HMLs, including:

1. HDLs: Hardware Description Languages such as VHDL (VHSIC Hardware Description Language) and Verilog are used to describe digital hardware at different levels of abstraction. These languages allow designers to model the behavior and functionality of hardware systems.
2. SystemC: A system-level hardware description language used to model hardware systems at a high level of abstraction. It allows designers to describe hardware systems as a collection of interconnected modules.
3. SystemVerilog: An extension of Verilog, SystemVerilog provides additional features for modeling complex hardware systems. It supports both hardware and software modeling and allows designers to create testbenches for hardware verification.
4. VHDL-AMS: VHDL-AMS is a hardware modeling language used for modeling mixed-signal systems, which include both digital and analog circuits.

HMLs are used extensively in the design of digital hardware systems, including microprocessors, digital signal processing circuits, and memory systems. By using HMLs, designers can create models of hardware systems that can be simulated and tested before being implemented in hardware, which can reduce the risk of design errors and improve design quality.

3.1.1.4 SW Description (Programming Languages, Algorithms, and Implementation)

There are so many methods to describe software. Software description refers to the process of defining software systems in terms of programming languages, algorithms, and implementation details. It involves the creation of a software design that can be used to guide the implementation of the software system. Programming languages are used to define the syntax and structure of the software system. There are many programming languages available, each with their own strengths and weaknesses. Some popular programming languages include C++, Java, Python, and JavaScript. The choice of programming language depends on the requirements of the software system and the preferences of the development team. Algorithms are used to define the logic of the software system. They describe the steps that the software system must take to accomplish its intended task. Algorithms can be simple or complex, and they can be implemented using a variety of programming languages. Implementation details refer to the specific implementation choices made by the development team. This includes decisions about data structures, control flow, and other implementation details. Implementation details can have a significant impact on the performance and reliability of the software system. Software description is an important step in the software development process because it provides a blueprint for the implementation of the software system. It allows developers to work collaboratively and ensures that everyone on the team has a clear understanding of the software system's requirements and design. By defining the software system in terms of programming languages, algorithms, and implementation details, developers can create software that meets the needs of users and stakeholders.

Software platforms provide a framework for the execution of software programs. They provide a set of tools and services that are used to develop, test, and deploy software applications. Software platforms can be categorized as desktop, mobile, web, or cloud-based. Some popular software platforms include Windows, macOS, iOS, Android, and Linux. The choice of software platform depends on the requirements of the software system and the preferences of the development team. In addition to programming languages and software platforms, software development also requires the use of software tools and frameworks. These tools and frameworks are used to automate software development tasks and to provide additional functionality to software applications. Examples of software tools and frameworks include integrated development environments (IDEs), version control systems, testing frameworks, and deployment tools.

3.1.2 HW/SW Interfacing

Hardware/Software (HW/SW) interfaces, mostly implemented as devices and device **drivers** [3]. HW/SW interfacing refers to the process of connecting hardware components and software components in a system so that they can communicate with each other and work together effectively. In most systems, the hardware components and software components are designed and developed independently, but they need to interact with each other to achieve the desired functionality of the system. The primary goal of HW/SW interfacing is to ensure that hardware and software components can communicate with each other seamlessly and without any issues. This is achieved by defining a set of standards and protocols that govern the communication between hardware and software components. These standards and protocols may include interfaces, **drivers, APIs**, and other communication mechanisms. HW/SW interfacing involves several tasks, including identifying the requirements for communication between hardware and software components, designing and implementing the interfaces that enable communication between the components, and testing and verifying that the communication works as expected. This process often involves collaboration between hardware and software engineers to ensure that the system is well-integrated and functions correctly. HW/SW interfacing is an essential aspect of system design, particularly in embedded systems and other complex systems where hardware and software components must work together seamlessly to achieve the desired functionality. Effective HW/SW interfacing can result in better system performance, increased reliability, and lower development costs, while poor interfacing can lead to errors, performance issues, and other problems in the system.

3.1.3 Task Graph and Cost Function: Problem Definition

Specification is analyzed to generate a task graph representation of the system functionality. Designers analyze task graph to determine each task's placement (HW or SW). A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes (Fig. 3.7). Given an undirected graph $G = (V, E)$, where V is the vertex set and E is the edge set. Each partitioning object corresponding to a vertex $v_i \in V$ is essentially a function that can be mapped to HW or SW. Each edge $e_{ij} \in E$ represents a call or an access to the callee function v_j from the caller function v_i.

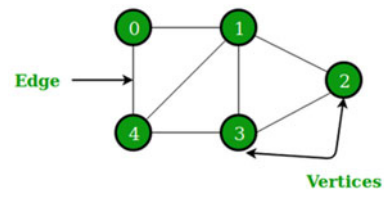


Fig. 3.7 Task graph

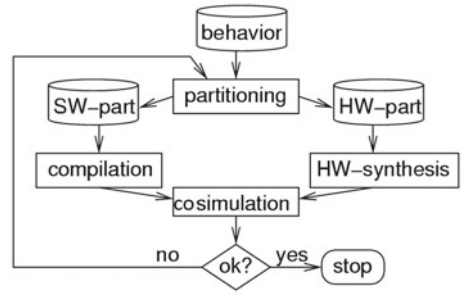


Fig. 3.8 HW/SW co-simulation

We need to minimize the HW/SW cost function. When we choose to implement a sub-task as a software or hardware, we compare its cost versus the total cost when all nodes are mapped to Hardware or Software. The cost should be less than the constraints. The communication cost is the cost incurred due to the data and control passing from one node to another in the graph representation of the design. The cost could be area, delay, and communication cost (if v_i is mapped to SW and its child v_j is mapped to HW or vice-versa). We need co-simulation to verify the functionality (Fig. 3.8).

3.1.4 Petri Nets

Specification can be analyzed to generate a Petri Net representation of the system functionality. A Petri Net is a graphical and mathematical modeling tool used **to describe and study information processing systems of various types**. Petri nets can provide a model that allows for formal qualitative and quantitative analysis in order to perform hardware/software partitioning. Petri nets as an intermediate model allows one to analyze properties of the specification and formally compute performance indices which are used in the partitioning process [16] (Fig. 3.9).

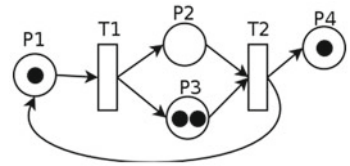


Fig. 3.9 Petri Net example. T is token (for control). P is process

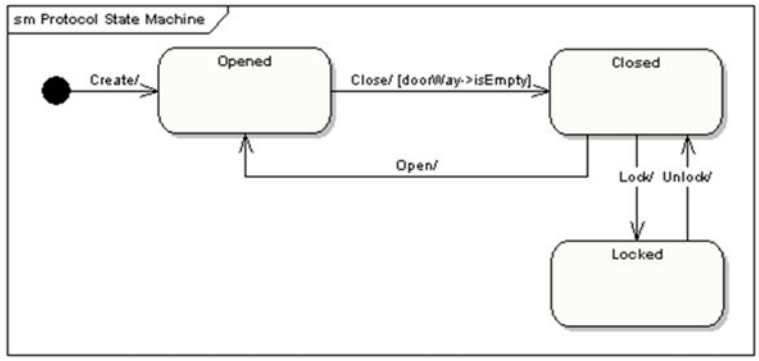


Fig. 3.10 Class UML diagram

3.1.5 UML Diagrams

Unified Modeling Language (UML) is used to describe a software system at a high level of abstraction. So, it helps acquire an overall view of a system. UML is *not* dependent on any one language or technology. Types of UML Diagrams are:

- Use Case Diagram.
- Class Diagram.
- **Sequence** Diagram.
- Collaboration Diagram.
- **State** Diagram.

State diagrams are created using UML state diagram notation. These diagrams describe system behavior using states, events and actions and correspond closely with the high-level RTL design approach. An example of Class UML diagram is shown in Fig. 3.10.

3.1.6 Optimization Techniques for Manual HW/SW Partitioning: ML-Based Approach

HW/SW partitioning is a complex task that involves determining which part of the system functionality will be implemented in hardware and which part will be implemented in software. The objective is to optimize the design for performance, power, area, and other design metrics. There are several optimization techniques that can be used for HW/SW partitioning, including:

1. Performance modeling: This technique involves creating models of the system behavior and analyzing their performance to determine the optimal partitioning.
2. Algorithmic mapping: This technique maps the system algorithms onto the hardware or software components based on their performance and resource requirements.
3. Data-flow analysis: This technique analyzes the data flow in the system to determine the optimal partitioning.
4. Constraint-based optimization: This technique formulates the partitioning problem as an optimization problem and uses constraints to find the optimal solution.
5. Heuristic algorithms: This technique uses heuristic algorithms such as simulated annealing, genetic algorithms, and tabu search to find a near-optimal solution.
6. Interactive design: This technique involves iterative partitioning between hardware and software designers to achieve an optimal partitioning.
7. Code profiling: This technique involves analyzing the software code to determine which parts of the code are performance-critical and should be implemented in hardware.
8. Hardware-software codesign tools: These tools provide automated support for the manual partitioning process by analyzing the system requirements and generating optimal hardware-software partitioning solutions.

The traditional heuristic algorithms include hardware-oriented and software-oriented ones. The hardware-oriented approach starts with a complete hardware solution and swaps parts to software until constraints are violated, while the software-oriented approach means that the initial implementation of the system is supposed to be a software solution. Particle swarm optimization and Simulated annealing are General-purpose heuristics.

3.1.6.1 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a well-known bio-inspired algorithm that is adopted in various applications [9]. PSO simulates the behavior of bird flocking. Suppose the following scenario: a group of birds is randomly searching for food in an area. There is only one piece of food in the area being searched. Not all the birds know where the food is. However, during every iteration, they learn via their inter-communications, how far the food is. Therefore, the best strategy to find the food is to follow the bird that is nearest to the food. PSO learned from this bird-flocking scenario and used it to solve optimization problems. In PSO, each single solution is a "bird" in the search space. We call it "particle". All of particles have fitness values which are evaluated by the fitness function (the cost function to be optimized) and have velocities which direct the flying of the particles. The particles fly through the problem space by following the current optimum particles. PSO is initialized with a group of random particles (solutions) and then searches for optima by updating generations. During every iteration, each particle is updated by following two "best" values [11, 15].

3.1.6.2 Simulated Annealing Optimization

The SA algorithm essentially tries to find an optimal solution to a "hard" problem such as partitioning, by conceptually representing the problem as a physical system with a huge number of particles at an initial temperature T. The system is randomly perturbed and the temperature is successively decremented allowing the system to reach statistical equilibrium at each temperature step: a state of minimal energy of the system corresponds to an optimal solution to the "hard" problem. Thus, key parameters in any formulation are the initial temperature T, the cooling (annealing) schedule mandating how the temperature is decremented, and the number of iterations at each temperature step. In HW-SW partitioning, perturbation is commonly defined as a move of a single vertex from HW to SW and vice versa [10].

3.2 HW/SW Communication

Performance is one of the key factors with respect to HW/SW Co-Design. Thus, reducing communication overhead is an inevitable need. HW/SW Communications should **provide low-latency and high-bandwidth** communications between HW/SW. DPI-C and SCEMI are two major mechanisms used in HW/SW communications.

In an embedded system, HW/SW communication refers to the exchange of information and commands between the hardware and software components of the system. This communication is necessary for the system to function as a cohesive unit and achieve the desired performance goals. Hardware components, such as processors, memory, and peripherals, communicate with software components, such as device drivers, operating systems, and application software, through various mechanisms such as buses, interrupts, memory-mapped registers, and direct memory access (DMA). The main challenge in HW/SW communication is ensuring that the communication is fast, efficient, and reliable, while also minimizing power consumption and avoiding conflicts between hardware and software components. There are several techniques used to optimize HW/SW communication in embedded systems, including:

1. Memory-mapped I/O: This technique maps hardware registers onto memory locations, allowing software components to access hardware components as if they were accessing memory.
2. Direct Memory Access (DMA): DMA allows hardware components to directly access system memory, bypassing the processor and reducing latency and power consumption.
3. Interrupts: Interrupts are used to signal the processor that an event has occurred and requires immediate attention, allowing the processor to respond quickly to hardware events.
4. Bus arbitration: This technique ensures that multiple hardware components can share the same bus without conflicts by implementing a protocol for bus access.

5. Communication protocols: Standard communication protocols, such as SPI, I2C, and UART, are used to ensure compatibility between different hardware and software components.
6. Hardware accelerators: Hardware accelerators can be used to offload computation-intensive tasks from the processor to specialized hardware, reducing the communication overhead, and improving system performance.

3.2.1 SCEMI

SCE-MI standard defines a multi-channel communication interface that addresses these challenges and caters to the needs of verification (both simulation and emulation) end-users and suppliers and providers of verification IP. Accelera Standard Co-Emulation API Modeling Interface (SCE-MI) connects a model written in HDL to a model running on a workstation. Moreover, it connects transactor models to a C model (untimed or RTL) running on a workstation [6].

It Establishes a communication bridge between a simulator and an emulator. The Establishment of a common API that supports both simulators and emulators and answers the question: How do I make my testbench running in a simulator communicate with my design running in an emulator. The broad intent of the standard is to create a modeling interface that meets all the requirements for simulation and enables transactor models to be easily migrated from a simulation environment to an emulation environment. The simulator stimulates the design in the emulator [4, 5].

The standard specifies a modeling interface which provides multiple channels of communication that allow software models detailing system behavior (SystemC on simulator) to connect to structural models (HW in emulator) describing implementation of a device under test (DUT).

connect software proxy models to message port interfaces on the hardware side of the design. the focus of this standard is to interface purely un-timed software models with a register transfer level- (RTL) or gate-level DUT.

The focus of this interface is to avoid communication bottlenecks that might become apparent when interfacing software models to an emulator as compared to interfacing them to a slower software HDL simulator or even an HDL accelerator. The communication channels are message- or transaction-oriented, rather than event-oriented. The Interface is designed to bridge two modeling environments, each of which supports a different level of modeling abstraction. SCE-MI supports multi-threaded environments, such as SystemC, or single-threaded environments, such as simple C programs. There is no thread-specific code anywhere in the SCE-MI.

There are three types of SCEMI interface (Fig. 3.11 and Table 3.2):

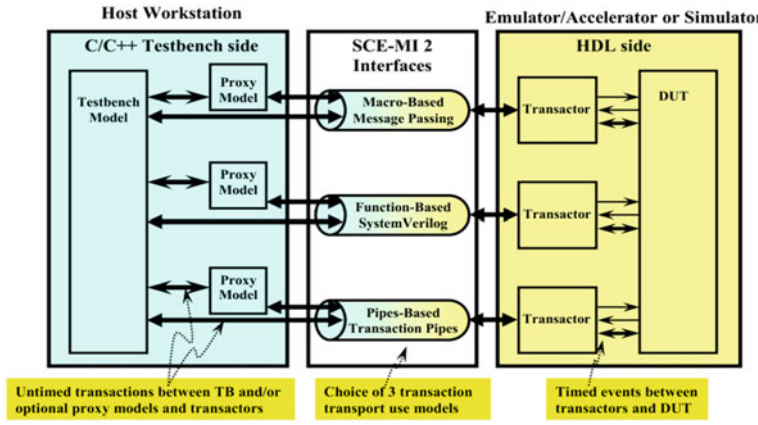


Fig. 3.11 SCEMI architecture

Table 3.2 SCE-MI as a bridge: different interfaces

	Macro-based	Function-based	Pipe-based
Definition	Uses predefined macros/API to send data from simulator to emulator (testbench in C/C++)	Use SystemVerilog DPI to call functions written in RTL running on the emulator (based on SystemVerilog DPI)	Supports constructs called *transaction pipes* which can be implemented as *built- in* library functions
Compatibility with UVM	Not compatible with UVM (since testbench in C/C++)	Applicable to UVM based testbench	applicable to UVM based testbench
Abstraction level	Low (closer to RTL) on HW side, High on SW side (C/C++)	Mid-level abstraction	High level abstraction
API Adherence	Strict adherence to a predetermined API	API is user defined (API-less)	Can potentially be implemented with reference source code that uses basic DPI, or can be implemented in an optimized specific manner

- Macro/API-based interface (no DPI, no UVM).
- Function-based interface (DPI, UVM).
- Pipes/Transactions/FIFO-based interface (DPI, UVM).

3.2.2 DPI-C

The SystemVerilog Direct Programming Interface (DPI) is basically an interface between SystemVerilog and a foreign programming language, in particular the C language. It allows the designer to easily call C functions from SystemVerilog and to export SystemVerilog functions, so that they can be called from C. DPI-C is particularly useful for system-level verification, where the goal is to verify the interaction between software and hardware. Here are some examples of how DPI-C can be used:

1. A C++ program can be used to generate stimuli for a hardware design being simulated in SystemVerilog. The C++ program can send data and events to the simulation environment via DPI-C.
2. A hardware design being simulated in SystemVerilog can send data and events to a C++ program via DPI-C. The C++ program can then perform some analysis on the data or use the data to drive further simulation.
3. DPI-C can be used to create a co-simulation environment where a hardware design being simulated in SystemVerilog and a software program running on a host computer can interact with each other.

3.3 HW/SW Synchronization

Synchronization, in HW/SW Co-Design context, relates to the ability to exchange messages in an **orderly manner**. Interrupt, FIFO, Semaphore, handshake, Mutex and lock are some mechanisms used for HW/SW synchronization. i.e. for moving data from HW to SW or vice versa. A comparison between these techniques is shown in Table III. Sometimes, you may need to add guard statement around the C code. HW/SW synchronization refers to the coordination of hardware and software components in a system to ensure that they work together seamlessly. This is important for achieving high performance, minimizing latency, and avoiding errors in complex systems. One example of HW/SW synchronization is in embedded systems, where the software running on a processor must coordinate with the hardware peripherals to ensure that data is transferred correctly and in a timely manner. Another example is in graphics processing, where the software running on a CPU must synchronize with the hardware graphics accelerator to ensure that graphics rendering is smooth and fast.

3.3.1 Semaphore

A well-known software primitive to implement synchronization points is the semaphore. It was proposed in 1962. A binary semaphore can be in two possible states: taken or

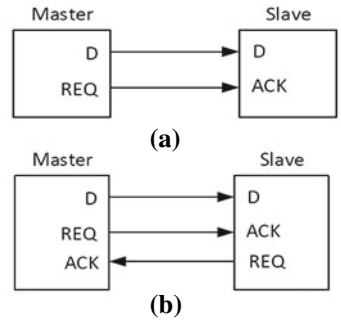


Fig. 3.12 Handshake
a one-way, **b** two-way

not taken. A database system might use a semaphore to limit the number of concurrent connections to the database.

3.3.2 Handshake

A *handshake* protocol is a signaling sequence between two concurrent entities with the purpose of achieving synchronization. A one-way handshake requires one control wire, called **request**, from the master module to the slave module (Fig. 3.12a). To transfer a data item, the master will assert the control wire. A two-way handshake requires two control wire, one called **request** and the other called **acknowledge** (Fig. 3.12b). Two computers might use a handshake protocol to establish a connection over a network.

3.3.3 Bus Locking

Bus Locking is a feature supported by some bus specifications that blocks re-arbitration and allows a master to retain the bus until the required operations are performed. Thus, with bus locking the processor can perform atomic operations. A real-time system might use a lock to prevent priority inversion, where a lower-priority task holds a resource that a higher-priority task needs to complete.

3.3.4 Mutex

It is a simple memory, with a configurable number of slots (each of them represents a mutex), that can be accessed, alternatively, from multiple ports. Each memory slot is composed by a bit that identifies if the lock has been taken or not, and up to n bits that save the identifier of the processor currently retaining the mutex. A multithreaded

program might use a mutex to prevent two threads from accessing a shared resource, such as a database or a file.

3.3.5 Interrupt

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts. Interrupts are asynchronous. i.e. they don't happen at predictable places in the user code or "passive" since the interrupt handler must wait for them to happen eventually. An input/output device might send an interrupt signal to the processor to indicate that it has data ready to be read.

3.3.6 FIFO

The use of a hardware FIFO to transfer data neither requires interrupting the CPU nor does it increase the load on the memory bus [12]. A network switch might use a FIFO buffer to store packets while they are being processed, ensuring that they are processed in the correct order (Table 3.3).

3.4 HW/SW Co-Design Metrics

HW/SW co-design metrics are used to evaluate the performance of a system that involves both hardware and software components. These metrics are crucial in determining whether the system meets its design requirements and can help identify potential areas for improvement. Some commonly used metrics in HW/SW co-design include:

1. Execution time (HW/SW Run-time): The time taken by the system to complete a task or a set of tasks.
2. Power consumption: The amount of power consumed by the system during operation.
3. Area (HW area and SW program/data memory): The amount of silicon area occupied by the system.
4. Cost: The cost of manufacturing the system, including the cost of components and labor.
5. Reliability: The probability that the system will operate correctly over a given period of time.
6. Throughput: The amount of data that can be processed by the system in a given period of time.

Table 3.3 Comparison between different methods of HW/SW synchronization

Synchronization mechanism	Functionality	Example
Interrupt	A signal used by hardware to request the attention of the processor to execute a specific task	An I/O device signaling a request for data transfer
FIFO (First In First Out)	A buffer that stores data and retrieves it in the same order in which it was received	A printer spooler where the print jobs are queued and printed in the order they were received
Semaphore	A variable that controls access to shared resources by multiple threads, processes or tasks	A traffic signal controlling the access of vehicles on a road
Handshake	A two-way communication protocol that ensures that two systems are synchronized and ready to communicate	A greeting or introduction between two people
Mutex (Mutual Exclusion)	A synchronization object that allows only one thread to access a shared resource at a time	A bathroom that can be used by only one person at a time
Lock	A synchronization mechanism that allows multiple threads to share the same resource, but only one thread can modify the resource at a time	A shared bank account that can be accessed by multiple users, but only one user can withdraw money at a time

7. Latency: The time taken by the system to respond to an input.
8. Scalability: The ability of the system to handle increasing levels of workload or complexity.
9. Maintainability: The ease with which the system can be maintained and modified over time.
10. Reusability: The extent to which the system components can be reused in future designs.
11. Quality: absence of errors.

The choice of metrics depends on the specific requirements of the system being designed. For example, in a real-time system, latency and throughput may be more important than area or power consumption, while in a low-cost system, cost and area may be the most critical metrics.

3.5 Conclusions

This chapter discusses the challenges faced by System-on-Chip (SoC) vendors in keeping up with the rapid evolution of different IP standards and delivering competitive products ahead of their competitors at a lower cost. Traditional design techniques based on independent design of HW/SW components are no longer sufficient to support the integration of subparts of such SoCs. Therefore, HW/SW co-design methodologies are becoming increasingly relevant. The chapter explains that when designing an algorithm for a SoC, there are three options: HW, SW, and HW/SW. Each option has its own advantages and weaknesses. Implementing the entire algorithm in hardware promises faster speed of operation and more reliable process but requires longer development time and less flexibility. On the other hand, implementing the entire algorithm in software offers more flexibility but may result in slower performance. HW/SW co-design methodologies allow designers to easily check system-level constraints satisfaction and evaluate cost/performance trade-offs for different architectural solutions. The chapter also mentions that IPs standards are evolving rapidly, making it challenging for SoC vendors to keep up with them.

References

1. M. Kammoun, M. Elleuchi, M. Abid, and A. M. Obeid "HW/SW Architecture Exploration for an Efficient Implementation of the Secure Hash Algorithm SHA-256 "Journal of Communications Software And Systems, Vol. 17, NO. 2, JUNE 2021.
2. J. Teich: Digitale Hardware/Software Systeme. Springer, 1997.
3. A. Kadav and M. M. Swift, "Understanding modern device drivers," in Proc. of ASPLOS, 2012.
4. Ha, Soonhoi, and Jürgen Teich. Handbook of hardware/software codesign. Springer, 2017.
5. Andrews, Jason. Co-verification of hardware and software for ARM SoC design. Elsevier, 2004.
6. Accellera, SCE-MI_v24 standard,. https://accellera.org/images/downloads/standards/sce-mi/SCE-MI_v24-Nov2016.pdf.
7. Gajski, D. Daniel, et.al., "High-Level Synthesis: Introduction to Chip and System Design," Springer Science & Business Media. 2012.
8. L. Pomante, P. Serri. "SystemC-based HW/SW Co-Design of Heterogeneous Multiprocessor Dedicated Systems". International Journal of Information Systems, 2014.
9. M. Clerc, Particle swarm optimization, Vol. 93, John Wiley & Sons, 2010.
10. S Kirkpatrick, C D Gelatt, M P Vechi, "Optimization by simulated annealing", Science, V-220, 1983.
11. Eberhart, R. C., and Shi, Y. 2001. Particle swarm optimization: developments, applications and resources. In Processions of 2001 congress on evolutionary computation (Seoul, Korea). 81–86.
12. J. A. Williams, N. W. Bergmann, and X. Xie, "FIFO Communication Models in Operating Systems for Reconfigurable Computing," in Proceedings of the 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2005, pp. 277–278.
13. Ha, S.; Teich, J.; Haubelt, C.; Glaß, M.; Mitra, T.; Dömer, R.; Eles, P.; Shrivastava, A.; Gerstlauer, A.; Bhattacharyya, S.S. Introduction to hardware/software codesign. In Handbook of Hardware/Software Codesign; Springer: Dordrecht, The Netherlands, 2017; pp. 3–26.

14. L. Nardi et al., “Practical design space exploration,” Proc. IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2019.
15. J. Kennedy, R. Eberhart, 47-Particle swarm optimization proceedings, IEEE international conference, in: Proc. ICNN’95 - Int. Conf. Neural Networks, 1995.
16. Maciel, Paulo & Barros, Edna & Rosenstiel, Wolfgang. (1999). A Petri Net Model for Hardware/Software Codesign. Design Autom. for Emb. Sys.. 4. 243-310. https://doi.org/10.1023/A:1008969621405.
17. "Hardware/Software Co-design: Principles and Practice” by Jari Nurmi and Jouni Isoaho.
18. "Co-Design for System Acceleration: A Quantitative Approach” by Masahiro Fujita, Kenichi Osada, and Satoshi Ohtake.
19. "Embedded Systems: Hardware, Design, and Implementation” by Krzysztof Iniewski.
20. “System Design: A Practical Guide with SpecC” by Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner.
21. “Hardware/Software Co-Design and Optimization for Cyberphysical Integration in Digital Microfluidic Biochips” by Krishnendu Chakrabarty.
22. Cordone, R.; Redaelli, F.; Redaelli, M.A.; Santambrogio, M.D.; Sciuto, D. Partitioning and scheduling of task graphs on partially dynamically reconfigurableFPGAs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 2009, 28, 662–675, doi:https://doi.org/10.1109/TCAD.2009.2015739.
23. Ma, Y.; Liu, J.; Zhang, C.; Luk, W. HW/SW partitioning for region-based dynamic partial reconfigurable FPGAs. In Proceedings of the 2014 32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, Korea, 19–22 October 2014; pp. 470–476, doi:https://doi.org/10.1109/ICCD.2014.6974721.
24. Banerjee, S.; Bozorgzadeh, E.; Dutt, N.D. Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration. IEEE Trans. Very Large Scale Integr. (VLSI) Syst.\2006, 14, 1189–1202, doi:https://doi.org/10.1109/TVLSI.2006.886411.
25. Chen, S.; Huang, J.; Xu, X.; Ding, B.; Xu, Q. Integrated Optimization of Partitioning, Scheduling, and Floorplanning for Partially Dynamically Reconfigurable Systems. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 2020, 39, 199–212, doi:https://doi.org/10.1109/TCAD.2018.2883982.
26. Salah, Khaled, and Mohamed AbdelSalam. "Performance Comparison of FPGAs and GPUs: Solving Sparse Matrices Case-Study." International Journal of Mathematical and Computational Methods 2 (2017).
27. Mohamed, K.S. (2018). Nature-Inspired Machine Learning Algorithm: Particle Swarm Optimization, Artificial Bee Colony. In: Machine Learning for Model Order Reduction . Springer, Cham. https://doi.org/10.1007/978-3-319-75714-8_4.

4 Pre-Silicon Verification and Post-Silicon Validation Methodologies

4.1 Introduction

The testing can be divided into two categories Pre-Silicon verification and post-Silicon validation. Pre-Silicon verification deals with simulating and verifying the RTL code while post-Silicon validation deals with silicon validation after fabrication.

The big picture for different verification technologies is shown in Figs. 4.1, 4.2 and 4.3. Advantages of different verification methodologies is shown in Table 4.1. The verification methods are divided between those based on simulations of test benches and those that are formal. Simulation-based verification includes directed tests coming directly from the requirements, random tests to cover corner cases, and assertions to assure that the design meets the designer's intent. Formal verification methods include sequential equivalence checking to prove model equibalance and model checking to prove properties.

In the Hardware market, there are so many **prototyping and emulation hardware platforms**. These platforms are based on commercial FPGAs to accelerate simulation. Each platform can be suitable for different performance requirements and applications/use modes. Prototyping platforms are used to quickly bring-up and debug hardware designs. These platforms are competing in terms of their debugging capabilities, ability to communicate with software side, performing data analytics, and the ability to scale the system. Software-driven hardware verification is a major trend for verification of complex Socs.

Given the extremely high and increasing costs of manufacturing microchips, the consequences of flaws going unnoticed in system designs until after the production phase, would be very expensive. At the same time, RTL verification is still one the most challenging activities in digital system development. Given this scenario, it is easy to see why many digital IC development teams report that more than 70% of the design time and engineering resources are spent in verification, and why verification is thus the bottleneck in the time-to-market for integrated circuit development. Verification methodology still

K. S. Mohamed, *Heterogeneous SoC Design and Verification*, Synthesis Lectures on Digital Circuits & Systems, https://doi.org/10.1007/978-3-031-56152-8_4

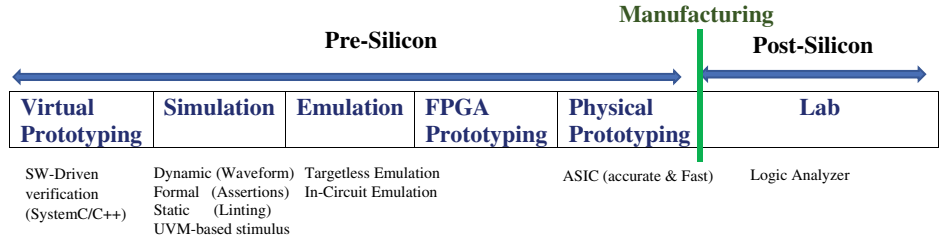


Fig. 4.1 The big picture for different verification technologies

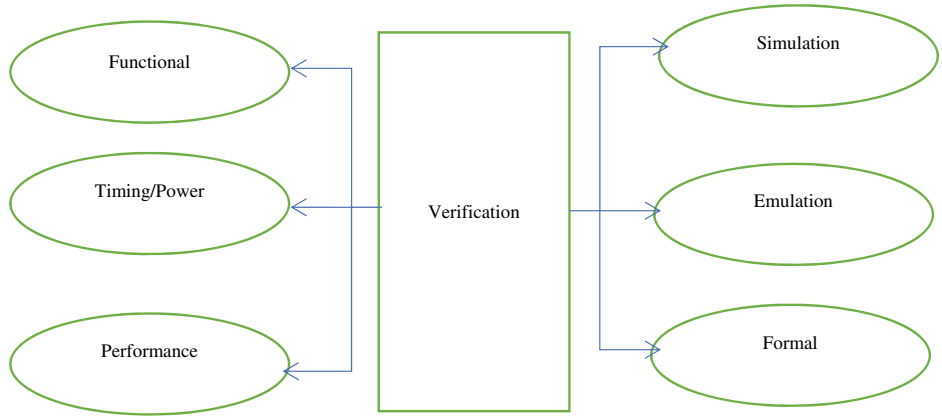


Fig. 4.2 Verification strategies

lacks any standard or even a commonly accepted approach, with the consequence that each hardware engineering team has its own distinct verification practices which often change with subsequent designs by the same team.

Comparison between different verification platforms in terms of speed and verification time is shown in Table 4.2 and Fig. 4.4. Table 4.3 shows comparison between different verification Platforms in terms of connection to Verilog DUT.

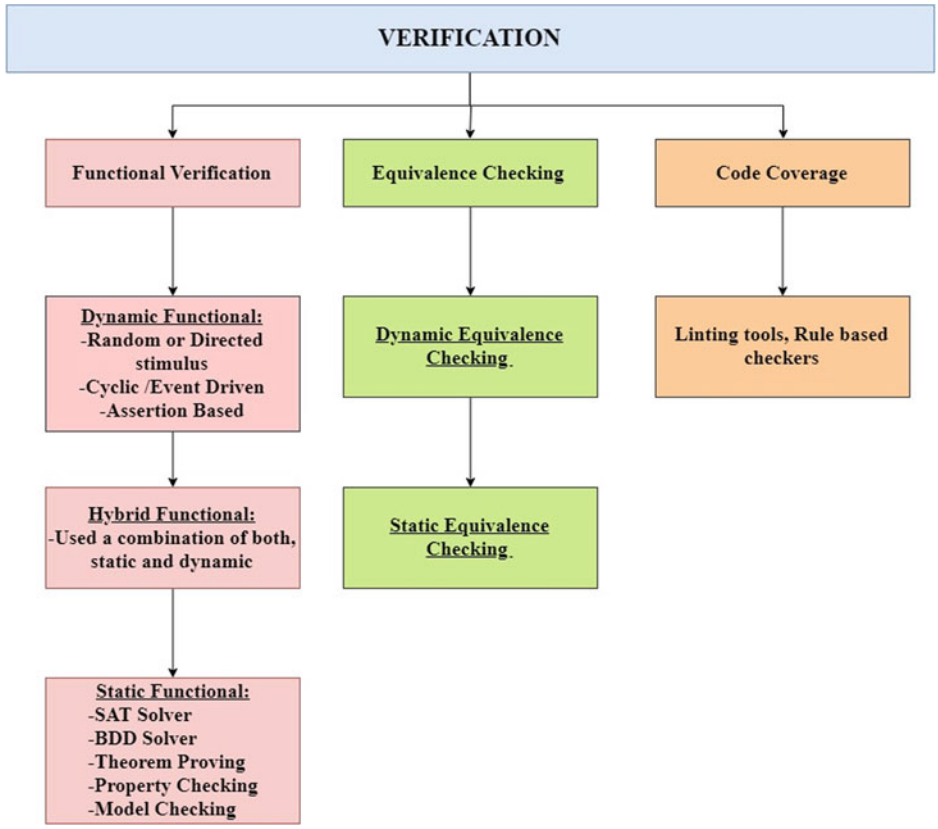


Fig. 4.3 Verification technologies

4.2 Pre-Silicon Verification

4.2.1 Virtual Prototyping

The SystemC and TLM libraries have become the de facto standard for virtual prototyping. This modeling method allows HW developers to perform early architecture exploration and SW developers to design and validate their applications early in the development phase of a system. RTL models suffer from long development and verification times. Consequently, complex SoCs may fail to meet the market's growing constraints. For this reason, virtual prototyping based SystemC and TLM is important to tackle such problems as they shorten development and debugging iterations. Virtual prototypes are fast, fully functional software models of hardware systems, which enable unmodified execution of software code [1, 2]. It can help the shift-left of SW development (Fig. 4.5). In Virtual prototyping, Stimulus is done from memory not from peripheral.

Table 4.1 Advantages of different verification methodologies

Verification technologies	Advantages
Virtual prototyping	• Early available in the design cycle as it allows virtual models (CPU and more) running with RTL design • Cheaper than hardware • Better and faster debugging capabilities • Performance tuning • Design exploration • It could be System-C based or QEMU-based virtual prototyping
Simulation	• High visibility
Emulation	• Faster than simulation as in some cases, for example to simulate linux boot, it may take infinite time to do this on simulation while it can be done on emulation in hours
FPGA prototyping	• Faster than emulation
Physical prototyping	• Accurate
Lab	• Using logic analyzers

Table 4.2 Comparison between different verification platforms in terms of speed and verification time

Platform	Relative speed	Verification time
Real time	1	1 h
FPGA prototyping	10 X slower	~1 day
Emulation	10^2 X slower	~week
Simulation: SystemC	10^2X slower	~month
Simulation: RTL	10^3X slower	~months
Simulation: gate-level	10^4X slower	~year

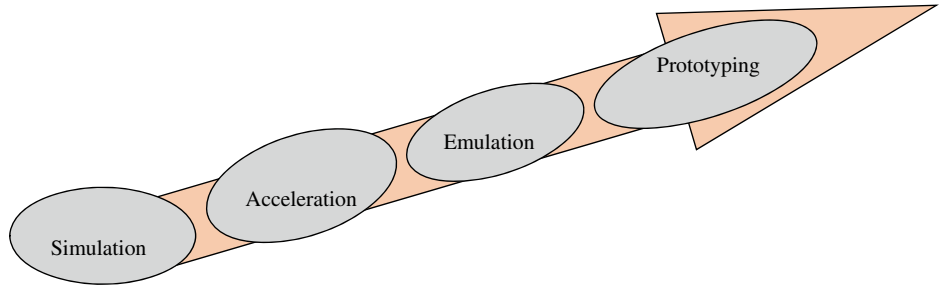


Fig. 4.4 Verification platforms: speed comparison

Table 4.3 Comparison between different verification Platforms in terms of connection to Verilog DUT

Method	Language/ environment	Interface with Verilog DUT	Development complexity	Debugging support	Simulation speed
C/C++ Testbench	C/C++	Requires low-level interfacing DPI	Moderate to high	Good	Fast
SystemC Testbench	SystemC	Native support	Moderate to high	Good	Fast
SystemVerilog Testbench	SystemVerilog	Native support	Moderate to high	Good	Fast
UVM Testbench	SystemVerilog	Native support	High	Good	Fast
QEMU Testbench	C	Requires custom QEMU emulation module	High	Limited	Medium to fast
Real hardware Testbench	N/A	Hardware interfaces and protocols	High	Limited	Real-time

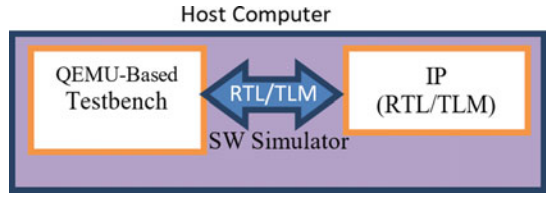


Fig. 4.5 Virtual prototyping

4.2.1.1 SystemC

Similar to RTL and gate-level modeling, SystemC-based virtual prototyping encapsulates HW components into modules, which can be nested within other modules to create a system hierarchy. Two or more modules can be connected using ports which describe the directional flow of data [3–5].

SystemC is a system design and modeling language. It is a lib of C++ language, evolved to meet a system designer's requirements for designing and integrating today's complex electronic systems very quickly while assuring that the final system will meet performance expectations. SystemC is capable of modeling at the cycle accurate RTL abstraction level, but it is ideally suited for untimed modeling. The most famous use

of SystemC is create a model with a TLM level of abstraction. SystemC includes Hardware-oriented features: Time modeling (sc_clock, sc_time classes), Hardware data types (double, long long int), Module hierarchy to manage structure and connectivity, Communications management between concurrent units of execution and Concurrency modeling [6, 7]. SYSTEMC is also considered electronic system level (**ESL**) language [8].

4.2.1.2 TLM

Because SystemC models follow a similar modeling paradigm to RTL and gate-level models, the transaction level modeling (TLM) library has been created to add another layer of abstraction to virtual prototyping. TLM models structure HW information hierarchically using so-called initiator and target modules. The communication among these modules is achieved using initiator and target **sockets instead of ports**. Sockets pass data around modules using abstract payloads instead of bit-wise signals (Fig. 4.6) [9, 10].

4.2.2 FPGA Prototyping

Every SoC needs a prototyping. FPGAs are reprogrammable silicon chips that provide hardware-timed speed and reliability. FPGAs have a matrix of Configurable Logic Blocks (CLB) connected through programmable interconnects and they can be reconfigured at any point of the design cycle. CLBs include logic gates, Look-Up Tables (LUT) and Flip-Flops (FF). Today's FPGAs also contain configurable embedded Static Random-Access Memory (SRAM), high-speed transceivers and high-speed inputs and outputs (I/

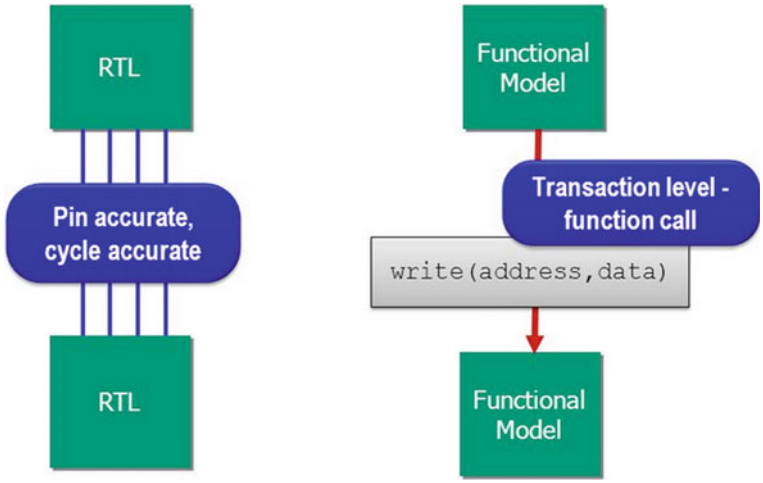


Fig. 4.6 TLM versus RTL

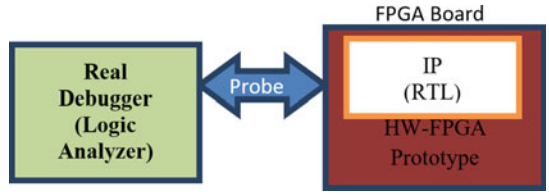


Fig. 4.7 FPGA prototyping

O). Therefore, they are an interesting solution in digital HW development. FPGA Prototyping is a key strategic approach to check ASIC hardware and speed up SoC software development when the silicon is not available yet. Interconnected FPGAs on a board provide the number of gates required to map a complex ASIC and start developing software with some ability to troubleshoot the hardware. FPGA Prototyping can have some real wall clock behavior (Fig. 4.7).

4.2.3 Physical Prototyping

Physical/Hardware Prototype refers to the construction of custom hardware or the use of reusable hardware to construct a hardware representation of the system. A prototype is a representation of the final system that can be constructed faster and is available sooner than the actual product. This is achieved by making tradeoffs in product requirements, such as performance and packaging. A common path to a prototype is to save time by substituting programmable logic for **ASICs**.

4.2.4 Emulation-Based Verification: Using FPGAs to Simulate ASICs

Emulation platforms are used to quickly bring-up and debug hardware designs (Fig. 4.8). These platforms are competing in terms of their debugging capabilities, ability to communicate with software. To simulate one second of real data of ASIC design running at 100 MHz on a simulator running at 10 Hz, this will take 10 M seconds and to run it on emulator running at 1 MHz, it would take 100 s [11]. However, emulation performance cannot match real time behavior.

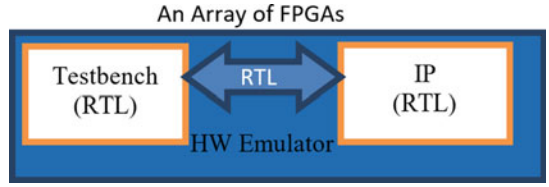


Fig. 4.8 Emulation

4.2.4.1 Targetless Emulation

In Targetless Emulation, the focus is on functional validation of the SoC components and subsystems, as opposed to testing against a predefined set of requirements. This approach allows for comprehensive and exhaustive testing, enabling designers to find and fix bugs and design flaws before the product is released to the market. It enables designers to perform pre-silicon verification and validation of their designs at speeds much faster than traditional simulation tools.

4.2.4.2 In-Circuit Emulation

In-circuit emulation (ICE) is a technique used to test and debug microcontrollers and other embedded systems using an emulator that is physically connected to the target system. In this technique, the emulator is connected to the target system through a JTAG or other debug interface, and the code is executed on the target hardware. In-circuit emulation is particularly useful when the target hardware is available and when the developer needs to test the code in a real-world environment. This technique enables developers to observe the behavior of the target hardware and to identify and fix bugs that may only appear when the code is executed on the actual hardware (Fig. 4.9). ICE works as **speed adaptor** between high-speed real HW and low-speed emulator.

4.2.5 Simulation-Based Verification

There are three metrics to evaluate a simulator: Speed, Flexibility, and Accuracy. Speed is how fast the simulator runs. Flexibility is how quickly one can modify the simulator to evaluate different algorithms and design choices. Accuracy is how accurate the performance numbers the simulator generates are vs. a real design i.e., Simulation error. HDL simulators are good for hardware debug in the early stages of the design cycle, when the design is focused at the block level.

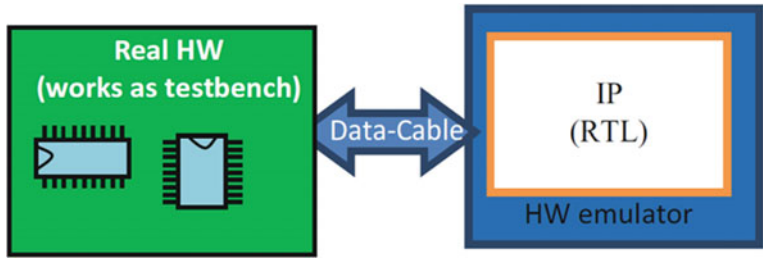


Fig. 4.9 ICE. Stimulus is done from real HW via peripheral

4.2.6 Functional Verification

RTL verification, which is verifying the correctness of an RTL description, is still one the most challenging activities in digital systems development where it consumes more than 70% of the development time as shown in Fig. 4.10, so we can say that verification is the bottleneck in the time-to-market for ICs development. The verification complexity and the simulation time increases incrementally with the design complexity. For complex designs it may take several hours or days to run some tests on the design, so our interest will be in the RTL functional verification [12, 13].

There are several methodologies to perform the RTL Functional verification such as:

- Directed Verification:
  - Conventional directed verification
  - Assertion Based Verification (ABV)
  - Coverage based verification (CBV).
- Class library based verification:
  - Universal Verification Methodology (UVM)
  - Open Verification Methodology (OVM)
  - Advanced Verification Methodology (AVM)
  - Verification Methodology Manual (VMM)
  - Reference Verification Methodology (RVM)
  - E Reuse Methodology (ERM)
  - Universal Reuse Methodology (URM).

Currently, UVM is dominating the market.

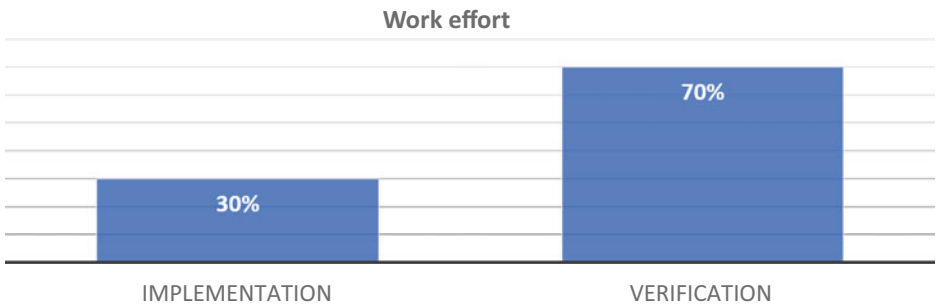


Fig. 4.10 Verification work effort versus implementation

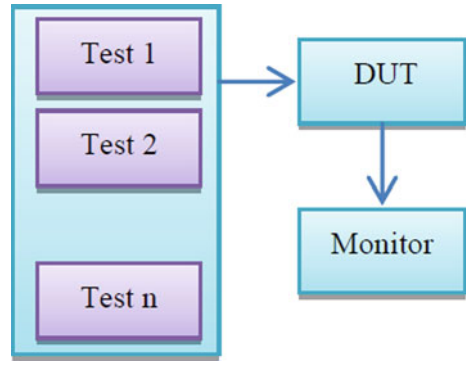


Fig. 4.11 The direct testbench environment

4.2.7 Directed-Testing Verification

In a direct environment, the task of checking all input vectors in the device is infeasible because the coverage space increases when the complexity of the device becomes bigger. Generally, the main objective of the informal verification techniques is to increase design space coverage and changes for finding errors in the digital system design. When the verification of an implemented device is performed in a hardware description language such as Verilog, VHDL, and SystemVerilog [14], the verification engineer needs to use specialized software tools [15].

The directed functional verification has an important role to meet the conditions. It has been found that it is hard to cover all corner cases by using the pseudo-random test generation methods, it is necessary to propose new test vector generation methods as shown in Fig. 4.11.

4.2.8 Coverage-Driven Verification

Coverage is a metric to measure verification progress and completeness. Coverage metrics identify portions of the design that were never activated during simulation, which allows us to adjust our input stimulus to improve verification. There are two metrics of coverage, Code coverage and Functional coverage. Functional coverage is manually defined and implemented by the designer. But, code coverage is implemented automatically by the tool [16].

1. Code coverage: is a measurement of structures within the source code that have been activated during simulation. One limitation with code coverage metrics is that you might achieve 100% code coverage during your regression run, which means that your testbench provided stimulus that activated all structures within your RTL source code, yet there are still bugs in your design. For example, the input stimulus might have

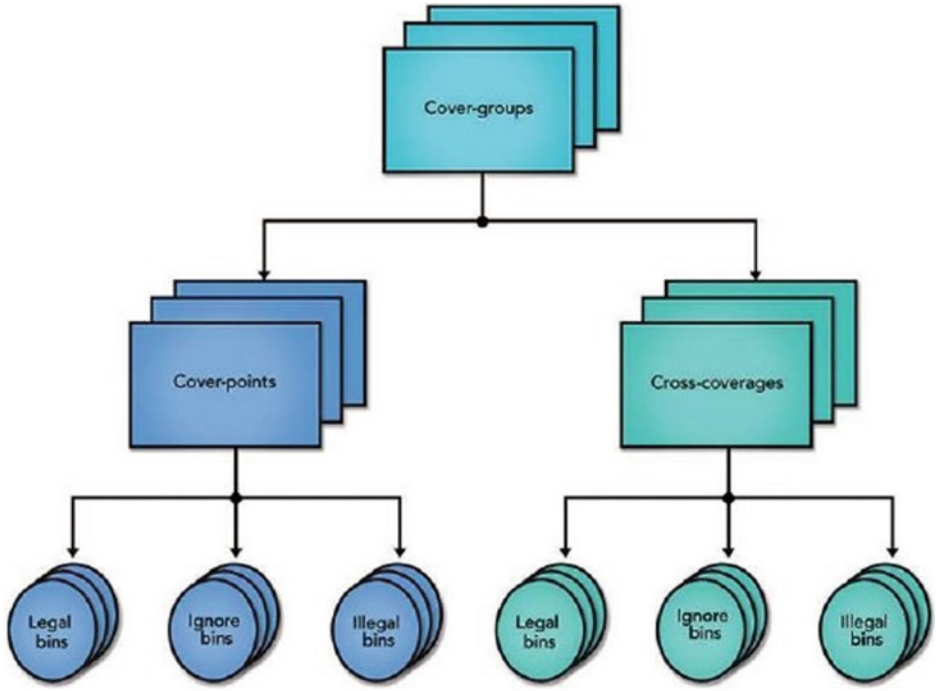


Fig. 4.12 Functional coverage hierarchy

activated a line of code that contained a bug, yet the testbench did not generate the additional required stimulus that propagates the effects of the bug to some point in the testbench where it could be detected. Functional coverage hierarchy is depicted in Fig. 4.12.

2. Functional coverage: determine if the design requirements, as defined in the specification, are functioning as intended. The functional coverage has three main metrics:
   - **Covergroups**: it's a user defined type that sample all the variables included inside it at the same sampling edge.
   - **Coverpoint**: it's a user defined variable/expression that cover a certain design specification.
   - **Bins**: collect information. An example for functional coverage is shown in Listing 1.

Listing 1: An example for functional coverage

```
bit [2:0] op
covergroup  COVERGROUP_NAME;
        coverpoint COVERPOINT_NAME;{
                bins b1= { };
                bins b2={ };
}
endgroup

COVERGROUP_NAME  COVERGROUP_NAME_INST;

initial begin
    COVERGROUP_NAME_INST = new();
    COVERGROUP_NAME_INST.sample();
end
```

4.2.9 UVM

A generic testbench architecture for functional verification is depicted in Fig. 4.13. UVM is a methodology for functional verification created by Accellera. It is based on Open Verification Methodology (OVM) that is created by Cadence and Mentor, and VMM from Synopsis. The UVM package contains a class library that consists of three main types of classes:

- UVM components, used to construct a class based hierarchical testbench structure.
- UVM objects, used as data structures for configuration of the testbench.
- UVM transactions, used in stimulus generation and analysis.

The top level class in a UVM testbench is called "test". It is responsible for configuring the testbench, initiating the construction process by building the next level down in the hierarchy, and by initiating the stimulus by staring the main sequence. The environment and the agent are components used in order to enable reuse. The design under test (DUT) which is the device that is to be verified by the testbench is connected to the testbench through interfaces. It receives stimulus from the testbench, and returns the output of that stimulus. UVM testbench is shown in Fig. 4.14. Agents encapsulate a driver, sequencer, and monitor as shown in Fig. 4.15 (Figs. 4.16 and 4.17).

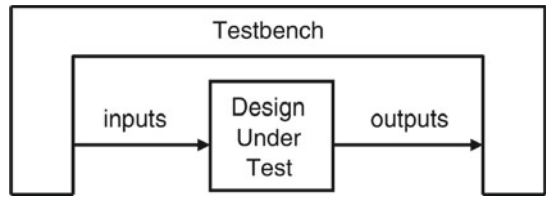


Fig. 4.13 A generic testbench

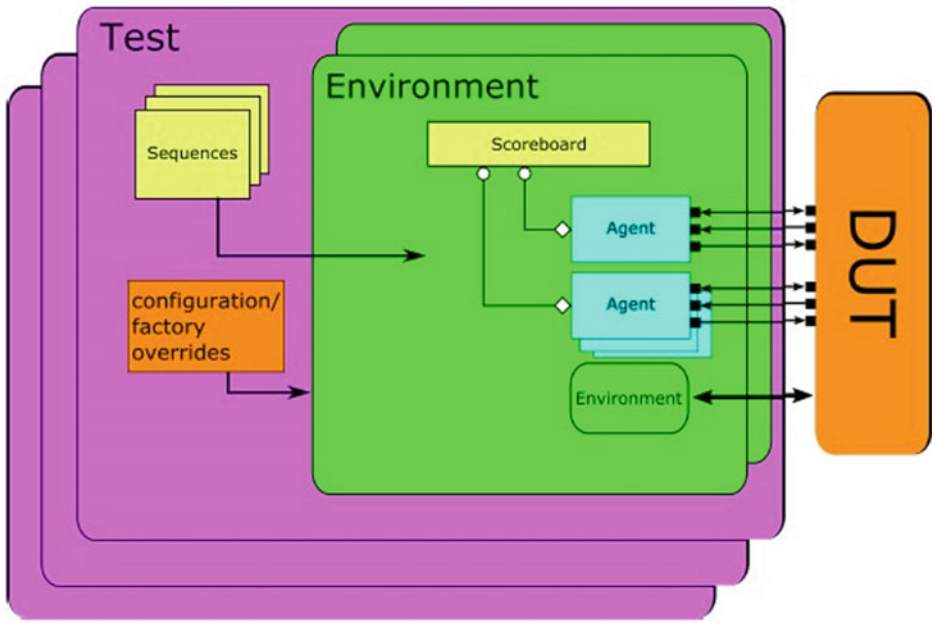
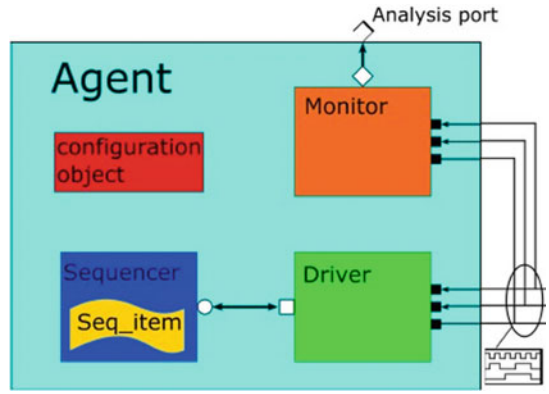


Fig. 4.14 UVM testbench

Fig. 4.15 Agent component. Agents can be active or passive



There are many similarities between the different SoC buses protocols. A generic UVM framework can encapsulate common functionality for most busses and let the user adjust the behavior of this UVC according to his need. This could decrease the amount of code needed to write. Construction flow in a UVM testbench consists of two domains: static and dynamic as depicted in Fig. 4.18.

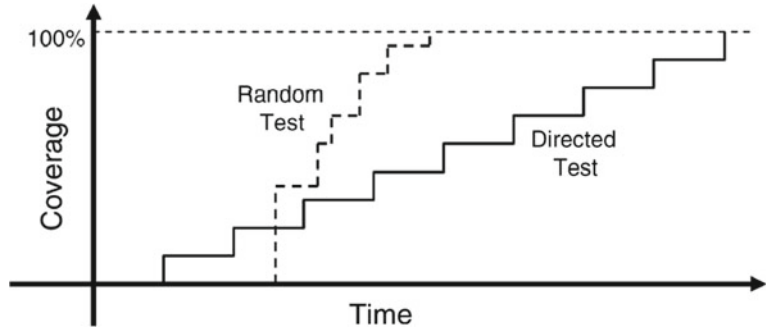


Fig. 4.16 Directed versus constrained-Random testing time to reach 100%

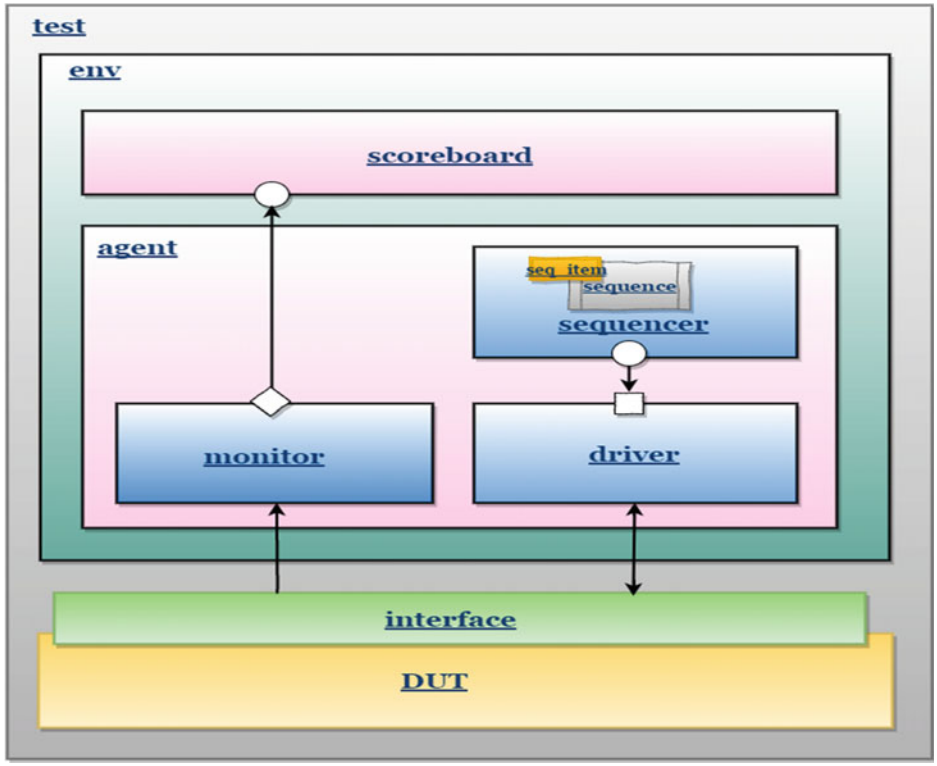


Fig. 4.17 General UVM architecture

Static Domain	Dynamic Domain
Module top; //DUT Initial begin Run_test(); end end module	1- Build phase: Calling run_test() causes the selected test to be constructed. Test = new(); build process starts from the test and works top-down. 2- Connect, run, report phases. 3- When all the UVM phases are complete $finish, control returns to the testbench module initial block

Fig. 4.18 Construction flow in a UVM testbench

UVM is an open source SystemVerilog library allowing creation of flexible and reusable verification components utilizing **constrained random stimulus** generation and functional coverage methodologies (Directed vs Constrained-Random testing time to reach 100% is shown in Fig. 4.14). It uses the Object Oriented Programming (**OOP**) concept which allows inheritance. Using the Transaction Level Modeling (**TLM**) standard for communications between different components and objects to hide the communications details for ease of **reusability**. It is separating test cases and sequences from the test environment to support reusability and work efficiency.

The UVM main **components** as shown in the figure are as follows:

- Driver
- Monitor
- Sequencer
- Agent
- Scoreboard
- Environment (Env)

The main architecture of UVM is shown previously in Fig. 4.18.

The UVM main **objects** as shown in the figure are as follows:

- Transaction (seq_item)
- Sequences
- Tests
- SystemVerilog **interface** for DUT communication.

The standard flow of the UVM can be as follows:

- The sequencer generates sequences which generates transactions that will be sent to the driver through TLM ports.
- The driver wiggles the DUT ports through the interface according to the transactions.
- The monitor is monitoring the DUT interface to feedback the scoreboard and the coverage components with the interface values through TLM ports.
- The scoreboard then applies the inputs to the reference model and checks the outputs and finally it reports the failure or success of the test case.

It is important to realize that when using UVM, relatively small set of features that Test Writers, Environment Writers and Sequence Writers will actually use. Indeed, the set of features identified in this work consists of **10 classes**, **30 methods** and **7 macros** that users need to be familiar with, in order to use UVM. When compared to the **357 classes** and **1037 unique methods** (938 functions and 99 tasks) that comprise UVM1.2, this means that UVM users really only need to learn **3% of UVM** (**2% of classes** and **3% of methods**) to be productive [17].

4.2.9.1 Advantages of UVM

1. Modularity and reusability, the methodology is designed as modular components (Driver, Sequencer, Monitor, Agent, …, etc.).
2. Separating test from test bench, hence can be used for different unites or projects.
3. Simulator independent.
4. Sequence methodology gives good control on stimulus generation.
5. Factory, which is a class, make component overriding or object become easier.
6. TLM is the transaction level modeling, which is port to port communication not component to component communication as the system Verilog mailbox.
7. Reusable and portable.

4.2.9.2 Disadvantages of UVM

1. Hard to learn and too many functions and tasks.
2. There are limitations for using UVM with emulators.
3. UVM is very complicated, so it is not suitable for small designs.
4. There are challenges in using UVM at SOC level.

4.2.9.3 UVM Phases

UVM phases and verification setup are as follows:

- **Build phase**

Where the test bench is constructed, connected and configured.

- **Run phase**

Where stimulus generation and execution take place here.

- **Clean up phase**

Where the test results are collected and reported.

4.2.9.4 How to Reduce the Compilation Time of UVM

- Avoid auto-configuration [18].
- Minimize the use of the uvm_config_db.
- Use configuration objects to pass configuration data to components.
- Minimize the number of uvm_config_db #(…)::get() calls.
- Use specific strings with the uvm_config_db set() and get() calls.
- Minimise the number of virtual interface handles passed via uvm_config_db.
- Using the test bench package to pass virtual interface handles.
- Consolidate the virtual interface handles into a single configuration object.
- Pass configuration information through class hierarchical references.
- Minimize the use of the UVM Factory.
- Avoid polling the uvm_config_db for changes.
- Do not use the UVM field macros in transactions.
- Minimise factory overrides for stimulus objects.
- Avoid embedding covergroups in transactions.
- Use the UVM reporting macros.
- Do not use the uvm_printer class.
- Avoid the use of get_xxx_by_name() in UVM register code.
- Minimize the use of get_registers() or get_fields() in UVM register code.
- Use UVM objections, but wisely.
- Minimize the use of UVM call-backs.

4.2.9.5 UVM: From Simulation to Emulation

The UVM testbench, to be reusable across simulation and emulation, must follow these basic principles [14, 19]:

- The environment must be partitioned into:

  1. A timed HDL: containing the DUT and the bus functional modeling (BFM) logic. BFM is responsible for bit wiggling.
  2. An untimed HVL testbench.

- There should not be any delays in the HVL. HVL should be truly untimed.
- The HDL side must comply with the synthesis rules allowed by the tool for the emulation platform.
- All communication between HVL and HDL must be through remote function calls.
- No HDL signals should be accessed directly from any HVL class.

4.2.10 Formal Verification

Formal verification is the process of mathematically checking that the behavior of a system, described using a formal model, satisfies a given property, also described using a formal model. The two models may or may not be the same but must share a common semantic interpretation. Simulation and formal verification are two complementary techniques for checking the correctness of hardware and software designs. Formal verification proves that a design property holds for all points of the search space while simulation checks this property by probing the search space at a subset of points [15, 20]. The actual effort of the formal tool is to apply as many input permutations and sequence of events and values to see if it can cause your assertion to fail as long as the formal tool is not over constrained in its ability to perform or generate the input permutations. Formal verification is the use of tools that mathematically analyze the space of possible behaviors of a design, rather than computing results for particular values. It is an exhaustive verification technique that uses mathematical proof methods to verify if the design implementation matches design specifications [21, 22].

4.2.10.1 Linting

Formal verification is a technique used in different stages in ASIC project life cycle to prove the correctness of the design. A **linting** strategy targeted as an initial check in the complete line of verification tools is the most cost-effective method of finding design errors. Linting tool performs a static analysis on RTL based on series of rules and guidelines to guaranteed perfect code style. At first, it is used in syntax analysis. Then the tools have gradually incorporated formal verification techniques that can identify the problems early before the simulation. Figure 4.19 shows Linting in verification cycle. Lint on a design includes for example Syntax error such as missing end of line operator, or incorrect instantiation signal width, or uncovered FSM states.

Static timing analysis (STA) is method to validate the time performance of the design by checking all possible paths for timing violations under worst case conditions without simulation (**dynamic timing analysis**) [23]. Standard Delay Format (**SDF**) is an IEEE standard for the representation and interpretation of timing data for use at any stage of an electronic design process. It finds wide applicability in design flows and forms an efficient bridge between dynamic timing verification and static timing analysis [24]. STA tool checks that each path meets the setup and hold time for the design library that the designers specify. STA can also predict false paths and Multi-Cycle Paths. A false path is a path, which exists in the chip, but it would never be exercised in the operation of the chip. A multi-cycle path is when a signal takes more than one clock cycle to propagate to the end point. Linting tools can be used at RTL level and at gate level simulations. Verilog can be used to create timing Check Tasks for verification of timing properties of designs and for reporting timing violations. For example, the **$period** checks that a period of signal is sufficiently long [25].

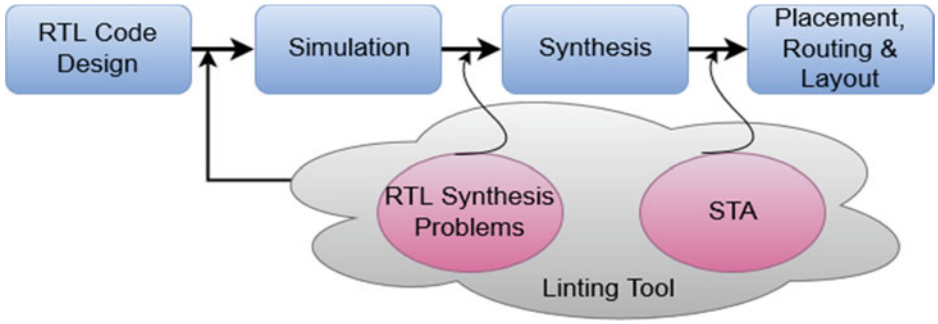
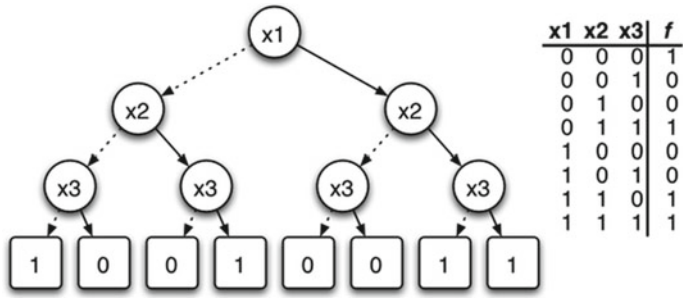


Fig. 4.19 Linting in verification cycle



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Fig. 4.20 An example of binary decision tree

4.2.10.2 Formal/Equivalence Checking

Equivalence checking is an emerging technology that can take **two designs** that have fundamentally different timing, enabling an un-timed or partially timed model to be compared against an RTL model, or between RTL models that have undergone retiming or other transformation to improve power or other design qualities. This is a necessary technology for mass adoption of high-level synthesis. Simulation works by providing stimulus to the primary inputs to the design along with a definition of a cyclic pattern to the clock input and an initial reset condition. This input stimulus generates an expected pattern on the outputs of the design. This same input pattern is applied for both the RTL and the netlist. The outputs of both designs are compared for equivalence. Covering every possible combination with simulation of the RTL would easily take weeks or months if not more even with a constraint random methodology such as UVM. One of the major benefits of equivalence checking is to discover **synthesis bugs and optimizations bugs** as it inputs the same stimulus to both the working RTL and the **netlist**. The equivalency tools simply convert each design to a **binary decision tree** and compare the logic. An example of binary decision tree is shown in (Fig. 4.20).

4.2.10.3 Property/Assertion-Based Checking

For property checking, we use property specification languages, such as PSL, and SVA, to write the requirements of the system. Then we create the mathematical model of the system and compare the specifications with the mathematical model in a model checker [26, 27].

Assertion-based verification is one of the promising verification techniques used in the industry for hardware designs. Using assertions will improve the controllability and observability which will help faster localization of errors and reduce debug time. There are different abstraction levels, verification levels and verification techniques associated with assertions. Assertions can be utilized for pre-silicon validation as well as post-silicon debug. Assertions can be embedded at different abstraction levels including Transaction-Level Model (TLM) and Register-Transfer Level (RTL). Compared to RTL models,

TLM is more abstract and faster in simulation. Therefore, TLM is more suitable for the validation of large designs and hardware/software co-design.

SystemVerilog assertions (SVA) are widely adopted in Register Transfer Level (RTL) IPs verification process to check the correctness of the corresponding RTL IP. Most assertions are generated from the requirements as the RTL code is being written. Mainly, we have two types of assertions:

1. Immediate Assertions: they check the conditions immediately and do not wait for time as opposed to concurrent assertions. They are written in always block which contains a sensitivity list. The "sensitivity_list" can be edge or level triggered. The assertion statement checks a condition [16]. If it is correct, it displays a pass message, else it displays a fail message. An example for immediate assertions is shown in Listing 2.
2. Concurrent Assertions: unlike immediate assertions, concurrent assertions allow sampling on positive or negative edge of the clock. The assertion checks the condition before of the implication sign "|->" which is called the antecedent. If the condition is true, the assertion checks for the part after the implication sign which is called the consequent and if it is true, the condition is successful. A message can be displayed when asserting the property. An example for concurrent assertions is shown in Listing 3. Assertion's messages can be controlled using severity level and the whole assertion block can be disabled in a certain condition using "disable iffy" feature. We can give a name to the assertion, so we can deactivate it, monitor it, etc. [14, 17, 18].

Assertions are classified into a few categories as follows:

- Logical Assertions (Table 4.4).
- Timing Assertions (Table 4.5).
- Commands Assertions.
- Arguments Assertions.
- Response Assertions.

Table 4.4 Example of logical assertions classification

Logical expression	Description
a && b	a = 1 and b = 1
!a	a = 0
not (a &&b)	a and b can't be = 1 at the same time this can be used in negative testing

Table 4.5 Example of timing assertions classification

Timing expression	Description
a |-> b	b = 1 at the same time a = 1
a |=> b	b = 1 on the next clock tick from a = 1
a ##N b	b = 1 on the Nth clock tick after a = 1
a ##[N1:N2] b	b = 1 between N1 and N2 clock ticks after a = 1

Listing 2: An example for immediate assertions

```
always @(sensitivity_list)
begin
assert (condition) $display ("Pass");
        else $display("Fail");
end
```

Listing 3: An example for concurrent assertions

```
property PROPERTY_NAME;
        @(posedge clk) antecedent  |-> Consequent;
Endproperty
assert property(PROPERTY_NAME;)  $display("Pass")
else $display("Fail");
```

When you run SVA properties in simulation, the infrastructure of your simulation environment executes the SVA properties with respect to the test vectors generated by the testbench. The SVA properties are also executable in the full formal tool, where the inputs and sequence of events is generated by the formal tool engines, no reliance on any generated test vectors. Adding **assumptions** on the inputs is one way to reduce the execution time of the formal tool. Along with 'assert' and 'cover'—'**assume**' is a reserved word in the formal verification language indicating how you want to use the property.

Formal tools do not require any kind of specific language as long as the language can mathematically generate a binary decision diagram. so, you can use VHDL RTL. In the simulation space, VHDL files can use the 'bind' construct of the SVA to associate SVA assertions within the simulation space, depending on your toolset.

Assertions can improve the controllability and observability that can lead to faster error detection and localization.

4.2.10.4 Formal CDC and RDC Verification

Metastability and **glitches** is unavoidable for clock domain crossing (CDC) and reset domain crossing (RDC) designs. Metastability refers to signals that do not assume stable 0 or 1 states for some duration of time at some point during normal operation of a design (Fig. 4.20) [28]. Glitches are undesired transitions that occur before the signal settles to its intended value (Fig. 4.20). CDC means data transfer from logic governed by one clock net to logic in another clock net (Fig. 4.20). A reset domain crossing (RDC)

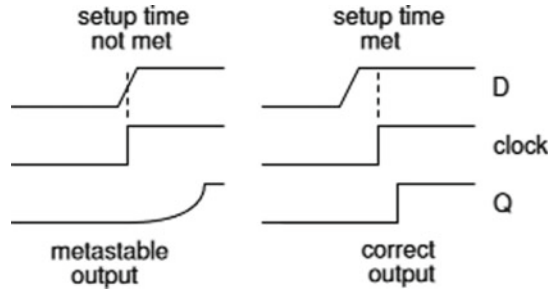


Fig. 4.21 Metastability example

occurs when a path's transmitting flop has an asynchronous reset, and the receiving flop has an uncorrelated reset or no reset (Fig. 4.21). Typical design techniques to ensure safe CDC operations include the usage of synchronization schemes such as inserting two FFs between the two domains. One of the basic synchronizer circuits is a dual flip flop synchronizer, also called 2-FF synchronizer (Fig. 4.25) [29]. Glitch can be detected in hardware implementations on FPGAs using delay-based sampling techniques [30, 31].

Verification engineers can potentially write assertions to verify certain aspects of CDC functionality or use automatic formal CDC verification [32]. Synchronizers are used too to solve RDC issues [33]. **Clock-gate isolation** technique is also used to solve RDC where clock of Rx flop is turned off before Tx reset asserted. Moreover, **Data-Isolation** technique can be used where isolation signal from a reset controller isolates the output of the first flop when its reset is asserted. There is a handshake protocol between the enables and the corresponding resets.

The CDC and RDC flow are a verification tool flow which reads the RTL source files, checks for missing or inadequate synchronization logic and reports them back to the engineer.

In hardware design, various types of reset signals are used to initialize and control the operation of digital circuits. Here are some commonly used types of resets:

- **Power-On Reset** (POR): Power-On Reset is a type of reset that is triggered when power is first applied to a circuit. It ensures that the circuit initializes in a known state and stabilizes before normal operation begins. POR circuits typically use a combination of voltage detection and delay circuits to generate a reset signal that remains active for a specific period after power is applied.
- **Hard Reset**: A hard reset, also known as a system reset or master reset, is a type of reset that forcefully clears all the internal states of a circuit or system. It is typically triggered by a dedicated signal or by a specific sequence of control signals. Hard reset is used to bring the system back to a well-defined state, often used in situations such as system hang-ups or software glitches.
- **Soft Reset**: A soft reset is a type of reset that is initiated by software or through a specific command rather than a dedicated hardware signal. It allows a controlled restart

of a system or a specific subsystem without affecting the entire circuit. Soft resets are commonly used in microcontrollers and embedded systems to handle software errors or reconfigure certain settings.

- **Watchdog Reset**: A watchdog reset is a type of reset mechanism that monitors the operation of a system or subsystem and performs a reset if a fault or error condition is detected. It typically involves a watchdog timer circuit that must be periodically refreshed or reset by the system's software. If the software fails to refresh the watchdog timer within a specified time frame, the watchdog circuit triggers a reset to bring the system back to a known state.
- **External Reset**: An external reset, also known as a hardware reset or manual reset, is a type of reset that is triggered by an external signal, such as a push-button switch or a dedicated reset pin. It allows an external agent, such as a user or an external control system, to initiate a reset operation. External resets are often used for system debugging, recovery, or to handle critical events.
- **Debug Reset**: A debug reset is a type of reset specifically used for debugging and testing purposes. It is typically designed to facilitate the development and verification of hardware or software by providing a controlled mechanism to halt or reset the system during debugging sessions. Debug resets are commonly used in embedded systems, microcontrollers, and digital logic circuits to enable breakpoints, step-by-step execution, and other debugging features. A debug reset can be triggered by various means, such as a dedicated debug port, a software command sent through a debugging interface, or specific conditions specified in the debug logic. When a debug reset is activated, the system is typically brought to a known state, allowing the debugger to halt the execution, examine the internal states, and modify the behavior of the system for debugging purposes. Debug resets are crucial for identifying and resolving issues during the development and testing phases of hardware design. They provide engineers with the ability to investigate and trace the behavior of complex circuits, identify timing or functional errors, and validate the correctness of the design (Figs. 4.22, 4.23, 4.24 and 4.25).

Fig. 4.22 Glitch example

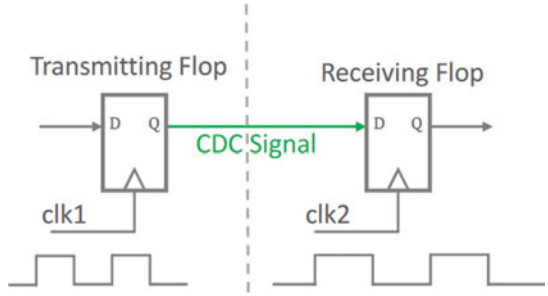


Fig. 4.23 CDC concept

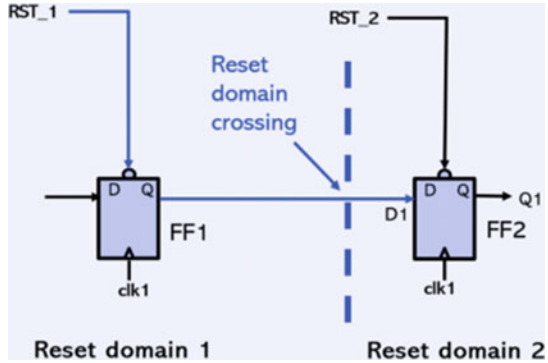


Fig. 4.24 RDC concept

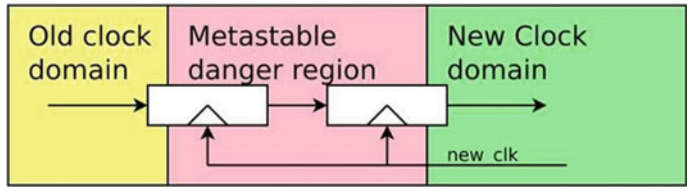


Fig. 4.25 2-FF synchronizer

4.2.11 Verification IP Versus Design IP

Verification IP (VIP) and Design IP are two different types of Intellectual Property (IP) used in the semiconductor industry. Verification IP (VIP) is used for functional verification of a design. It provides a set of pre-built tests that can be run to verify that a design is functioning correctly. VIP can be used to verify various protocols such as PCIe, USB,

Ethernet, and others. The VIP is designed to mimic the behavior of the protocol it is verifying, so it can generate the necessary test scenarios and check the design's response. On the other hand, **Design IP** provides pre-designed building blocks for the implementation of a particular function in a semiconductor design. Design IP includes things like pre-designed processors, memory blocks, and various other circuit blocks that can be integrated into a larger design. Design IP allows designers to focus on the higher-level design aspects rather than the implementation of lower-level functions. In summary, Verification IP (VIP) is used to verify that a design works correctly, while Design IP is used to speed up the design process by providing pre-designed building blocks for various functions.

4.2.12 Power-Aware Verification

Power is everywhere in all applications such as mobile, cloud, automotive, aerospace, etc. Not only Power can impact performance and battery life, but also it can impact your chip reliability. There are power bugs that can lead to inefficiency, product failures or accelerate aging. Clock gating reduces the switching power by turning off clocks to specific registers when there is no update in the register values or when the updated value is not going to be used by downstream logic. However, this can introduce a bug if there are corner-case conditions where the clock is turned off when it needs to be on for correct operation of the design. Sequential equivalency checking (**SEC**) can find those corner-case bugs by detecting differences between the behavior of the original design and the design with clock gating.

Power verification can start very early in the design cycle by using architectural power exploration (system level power analysis) or later in the design cycle after synthesis and beyond till signoff (RTL level or gate level). There are many tools in the market that can generate a power-aware optimized RTL design or generate a report that highlights the power-related issues. Moreover, it can provide automatic power optimization.

UPF-based verification at the RTL consists of creating power domains, inserting power aware cells such as isolation, level-shifter, and retention cells and defining a supply network to propagate power. RTL **power-aware** verification ensures that power-aware cells are placed at all the required ports/state elements and the power distribution network is valid.

Low-power static checkers are tools used in electronic design automation (EDA) to perform static analysis on digital hardware designs with a focus on power consumption. These checkers help identify potential power-related issues in a design without the need for actually simulating or synthesizing the entire design, which can be time-consuming. The goal of these low-power static checkers is to detect potential power issues early in the design phase, reducing the overall development time and ensuring that the final design meets power requirements. They complement other design verification and optimization

techniques used during the design process. Some common low-power static checkers include:

- **Power Rule Checkers**: These checkers analyze the design to ensure that it adheres to certain power-related rules or guidelines. For example, they may check that certain power domains are correctly defined and isolated, or that specific power-saving techniques are employed in critical areas of the design.
- **Clock Gating Checkers**: Clock gating is a popular power-saving technique where the clock to certain parts of the design is turned off when they are not actively in use. Clock gating checkers verify that clock gating is properly implemented and does not lead to any issues like glitches or improper synchronization.
- **Multi-Voltage Domain Checkers**: In designs with multiple voltage domains, it is essential to handle level shifting and isolation correctly. Multi-voltage domain checkers verify that the interface between voltage domains is well-designed to prevent potential power-related problems.
- **Data-Path Power Analysis**: These checkers focus on analyzing the power consumption of data paths within the design, such as adders, multipliers, and other arithmetic circuits. They help identify power-hungry sections that might require optimization.
- **Sequential Power Analysis**: Sequential elements like flip-flops and registers can contribute significantly to power consumption. These checkers analyze the sequencing and usage of sequential elements to suggest power optimizations.
- **Memory Power Analysis**: Memory elements (e.g., RAMs, ROMs) can be major power consumers in a design. Memory power analysis checkers help identify power inefficiencies in memory usage.
- **Clock Tree Power Analysis**: The clock distribution network can also consume considerable power. Clock tree power analysis checkers identify potential optimizations in the clock tree design to reduce power consumption.

4.2.13 Timing Verification

Timing verification means checking that the design met its required input-to-output and internal path delays and that no violation times (such as setup and hold parameters). Stating timing analysis (**STA**) tools are one of the methods used for timing verification and it will be discussed later in this chapter. Simulation (Dynamic timing analysis) is another method for timing verification.

4.2.14 Safety Verification: Fault Simulation

Fault simulation is important for evaluating the robustness of a product's safety mechanism against unexpected errors. Faults are injected into the design to emulate the unexpected errors. Functional verification validates designs outside of stimulus integrity (Fig. 4.27). Fault simulation analyzes potential stimulus failure due to defects factors. Fault simulation aims at developing a fault free design before manufacturing. Fault coverage are obtained by generated set of faults within the testbench.

Many parameters affect the fault model of the circuit such as the change in physical parameter, different environmental condition, etc. The logical fault modeling mainly focusses on how the physical failure of the circuit affects the behavior [34].

Common Fault Models (Fig. 4.26):

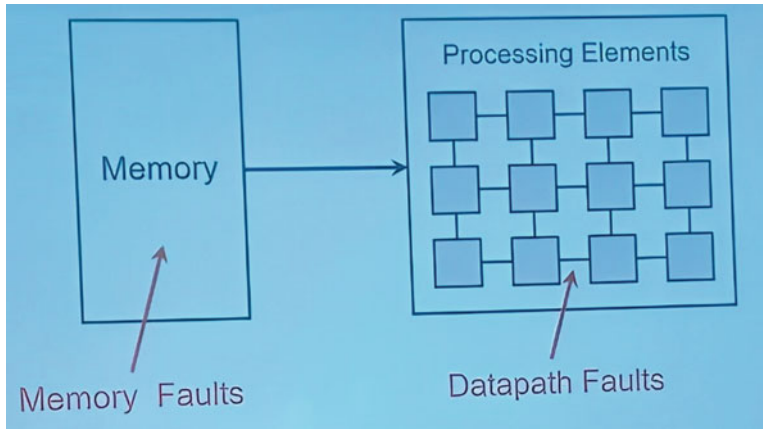


Fig. 4.26 Types of faults

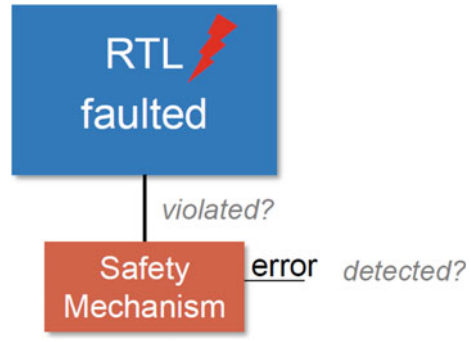


Fig. 4.27 Conceptional view of fault testing

- **Stuck at fault**: An individual signal or pins in the circuit are permanently stuck at logic 0 or at a logic 1 is termed as stuck at fault. It is static fault.
- **Delay Fault**. It is Dynamic Faults: Pin transitions from zero to one (slow-to-rise) or from one to zero (slow to-fall). This can be produced in any pin in the model, the principal dynamic pin faults are inputs or outputs of primitives slow-to-rise and slow-to-fall. Dynamic faults are mainly caused by delay defects, these defects are a physical manufacturing problem which slows down the propagation of a logical data value's transition from logic 0 to logic 1, or vice versa. There are many manufacturing problems which can result in the creation of delay defects. Impurities in the materials used can cause the resistance of a material to change, resulting in the Genus time constant to increase. Etching processes that remove too much material, or additive processes, which add or remove too much or not enough material, can also increase the resistance of a transmission line. Poorly filled vias are also a common cause of delay defects.
- **Memory fault**: Such as a particular address location access has no cell, a particular cell is never accessed, few address locations accessing a single cell, multiple cells are being accessed by a single address.

The huge complexity of electronic systems has led to growth in reliability needs have increased dramatically with the introduction of nanometer technologies, which impact adversely noise margins; process, voltage, and temperature variations; aging and wear-out; soft error and EMI sensitivity; power density and heating. The use of **design for robustness and reliability** techniques for extending, yield, reliability, and lifetime of modern SoCs is mandatory for reducing power dissipation, reducing noise margins and thus the sensitivity to soft-errors and EMI, and reducing the severity of **timing faults**.

Aging simulations is referring to simulating the effects of aging on the components of a System on Chip. Aging can affect various aspects of a chip's performance, including transistor degradation, interconnect reliability, and power consumption. To study and mitigate the effects of aging on SoCs, researchers often employ simulation techniques. These simulations involve modeling the behavior of individual components over time and considering factors such as voltage variations, temperature changes, and wear-out mechanisms. By running aging simulations, engineers can assess the impact of aging on system-level performance, identify potential vulnerabilities, and develop strategies to enhance the chip's longevity and reliability. Aging simulations can also aid in optimizing power management techniques, thermal management strategies, and fault tolerance mechanisms within SoCs. Additionally, they can contribute to the design and validation of adaptive systems that can dynamically adjust their behavior based on aging-related variations. Aging simulations in the context of SoCs play a crucial role in understanding and addressing the long-term performance and reliability challenges associated with integrated circuits.

4.2.15 Performance Verification: *Meeting Bandwidth and Latency*

Performance verification means that the design will meet its targeted bandwidth and latency levels.

4.2.16 Verification Challenges

A. **Reduce time to develop and improve robustness**
   - Develop verification components (e.g., UVM driver, sequencer, monitor, agents) that are reusable.
   - Use System Verilog Assertions (SVA) to reduce time to develop complex sequential and combinatorial checks.
   - Raise abstraction level of developing the tests.

B. **Reduce time to simulate and improve simulation accuracy and throughput**
   - Higher-level abstractions simulate much faster than pure RTL test bench, Use transaction level test bench (e.g., UVM, TLM 2.0).
   - Use coverage-driven verification (CDV) methodologies.

C. **Reduce time to debug and improve efficiency**
   - Use System Verilog Assertion methodology to quickly reach to the source of the bug.
   - Constrained random verification reduces time to debug.

D. **Reduce time to "comprehensive" cover: Check How Good Is Your Test bench**
   - Use System Verilog *functional coverage* language to measure the *intent* of the design.

4.2.17 How to Verify the Verification

Golden Reference Models can be generated in C++, VHDL, MATLAB or Verilog RTL model, that run in parallel with the Verilog model. Golden reference models are high-level descriptions of a design and are used to compare to the results of the model under test during simulation. Reference models usually model interaction between components at the transaction level (e.g. read transaction/write transaction) instead of at the signal level. At the end of each transaction the outputs for the MUT (model under test) and the reference model are compared. Test Bencher generates all of the sub-functions for the golden reference model, keeping the transaction interface to the reference model the same as the HDL level model [35].

4.2.18 Regression Testing Techniques

Regression testing is a type of software testing that ensures that previously developed and tested software still performs the same way after it is changed or interfaced with other software. Regression Testing is required when there is a change in requirements and code is modified, new feature is added to the software, defect fixing, and performance issue fix. In regressions we may:

- **Retest all**: all the tests in the existing test bucket or suite should be re-executed. This is very expensive as it requires huge time and resources.
- **Select**: Instead of re-executing the entire test suite, it is better to select part of test suite to be run.
- **Prioritization**: Selection of test cases based on priority will greatly reduce the regression test suite.

4.2.19 IP-XACT Methodology

XML is a software- and hardware-independent tool for storing and transporting data. Software languages divide into two sections, Imperative software languages and declarative ones. The XML is one of the declarative software languages. Those declarative languages have 3 types, functional, markup and logic. The XML is a declarative markup software language. It was developed from Standard Generalized Markup Language (SGML) as shown in Fig. 4.28. It's a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. XML files are self-descriptive that don't do anything in and of themselves except describing the transportation, structure, and storage of data. XML files just contain the sender information, receiver information, the heading, and the message body. The most important thing about XML is validation. Validation is the process of checking to see if a XML document conforms to a schema.

IP-XACT is an XML schema for language and vendor-neutral IP descriptions that includes a generator interface for "plug-in" functionality. It's also uses XML format that defines and describes electronic components and their designs. It was created by the Structure for Packaging, Integrating, and Reusing IP within Tool-flows (SPIRIT) Consortium as a standard to enable automated configuration and integration through tools. The electronics industry uses tools produced by a number of vendors in the design process. Many of these tools use unique and proprietary formats.

A standard for description of electronic design information lets developers exchange this information quickly between different design environments (DE). This speeds the design flow and leads to quicker implementation. The Goal of IPXACT is to provide a well-defined XML Schema for meta-data that documents the characteristics of Intellectual

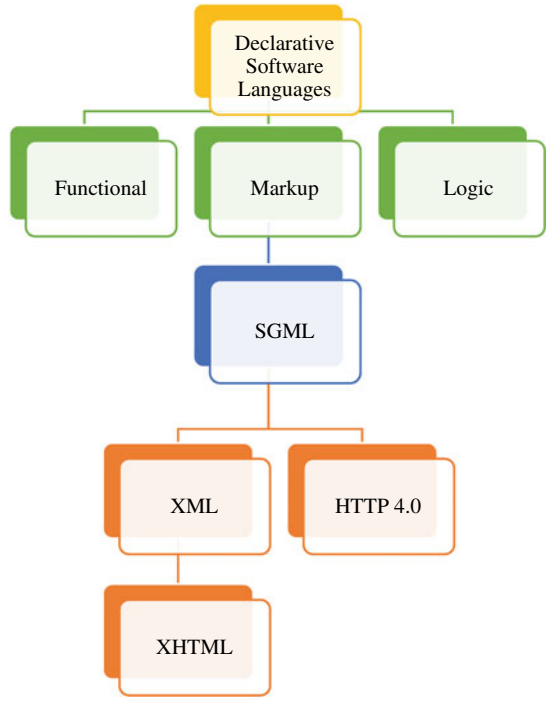


Fig. 4.28 XML language tree

Property (IP) required for the automation of the configuration and integration of IP blocks; and to define an Application Programming Interface (API) to make this meta-data directly accessible to automation tools.

IP-XACT is a standard that specifies how to describe different types of electronic IPs in the form of an XML document. An IP component has several attributes that can map directly to XML. Memory maps, registers, bus interfaces, ports, views, parameters, generators, and file sets are some of the attributes that the component contains and can be mapped to XML. When multiple components are connected together, it becomes an IPXACT design file. A component document represents a single IP block that you can instantiate as a single entity in a design. Components have bus interfaces whose types are specified by a bus definition. A component document can also define one or more views of the implementation, or an interconnect infrastructure in the form of a bridge or a channel. Typically, the component document contains the following elements:

- A model element represents the top-level signal list for that component.
- Signal map elements map signals to appropriate bus interfaces.
- Other elements document the memory map for slave bus interfaces and provide information on accessible locations and their structure. These locations are called *registers*

in the IP-XACT standard. You can implement these as functional or *Register Transfer Level* (RTL) registers.

The IP-XACT specification is a mechanism to express and exchange information about design IP and its required configuration. 2 While the IP-XACT description formats are the core of this standard, describing the IP-XACT specification in the context of its basic use model, the design environment (DE), more readily depicts the extent and limitations of the semantic intent of the data. The DE coordinates a set of tools and IP, or expressions of that IP (e.g., models), through the creation and maintenance of meta-data descriptions of the system on chip (SoC) so that its system design and implementation flows are efficiently enabled and reuse centric [36].

A DE enables the designer to work with IP-XACT design IP through a coordinated front-end and IP design database. These tools create and manage the top-level meta-description of system design and may provide two basic types of services: *design capture*, which is the expression of design configuration by the IP provider and design intent by the IP user, and *design build*, which is the creation of a design (or design model) to those intentions.

Before using the IPXACT we had to work in three parallel tasks with no exchange of information. First when we have the IP specifications, we start system profiling and exploration. After this step we move to verification task. The testbench is written for the IP then we start the verification solution for the system. Finally, we make the design of the IP using RTL. After writing the RTL code of the IP, we start the synthesis solution.

We will notice here that there is no relation between the design process and the verification process. This way we waste a lot of time that will delay the time to market, which will increase cost. However, the IPXACT solved this problem as it made this process much easier. There a great difference happened when we used the IPXACT. With IPXACT now the process will have less time as it generates an XML file, the design and the verification processes will be done in one task instead of two tasks. So now it's obvious that IPXACT gave us a great advantage over the traditional flow. Making standard that shares one specification for all information saved a lot of time and made the processes less [37, 38].

4.2.20 Portable Stimulus Standard: Graph-Based Testing

Portable stimulus standard (PSS) is a **language** for capturing test scenarios and verification logic in an abstract way ([Like graphical representation in terms of states), which can then be applied on multiple platforms and testbench implementations (Fig. 4.29) [39]. PSS borrows its core concepts from object-oriented programming languages, hardware-verification languages, and behavioral modeling languages. PSS features native constructs for system notions, such as data/control flow, concurrency and synchronization, resource

requirements, and states and transitions. It also includes native constructs for mapping these to target implementation artifacts. A key purpose of PSS is to automate the generation of test cases and test suites. The randomization mechanism in PS based verification starts from an abstract description of the legal transitions between the high-level states of the DUT and automatically enumerates the minimum set of tests needed to cover the paths through this state space. This is a very promising feature which can ensure a better-quality test as compared to manually written tests [40]. The tests can be visually seen allowing users to understand the control and data flows in a better way [41]. Expectation of increasing verification productivity with using PSS compared to other methodologies is illustrated in Fig. 4.30. PSS is useful creating a single representation of stimulus and test scenarios, usable by a variety of users across different levels of integration under different configurations, enabling the generation of different implementations of a scenario that run on a variety of execution platforms, including, but not necessarily limited to, simulation, emulation, FPGA prototyping, and post-Silicon [42].

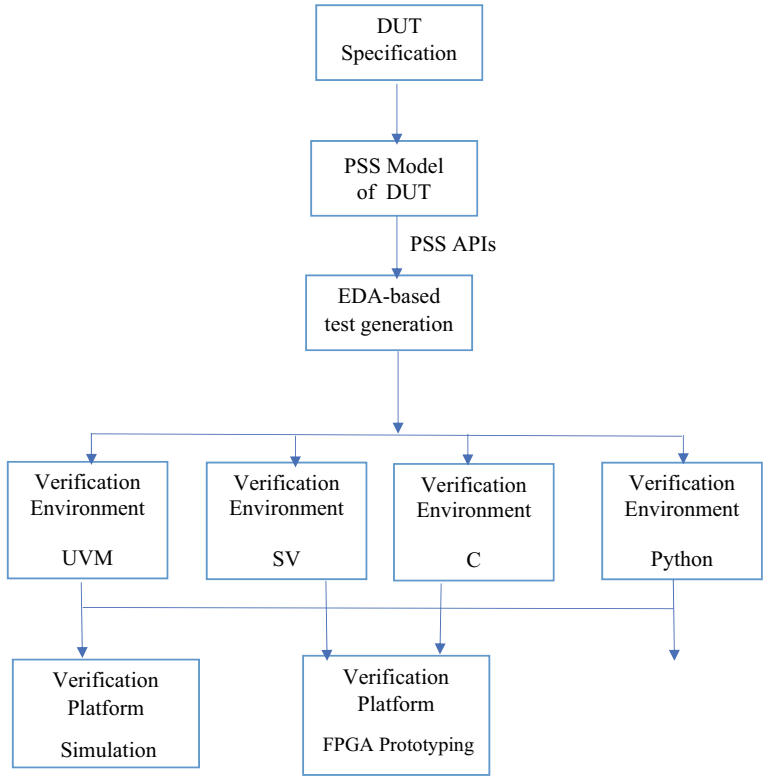


Fig. 4.29 PSS methodology

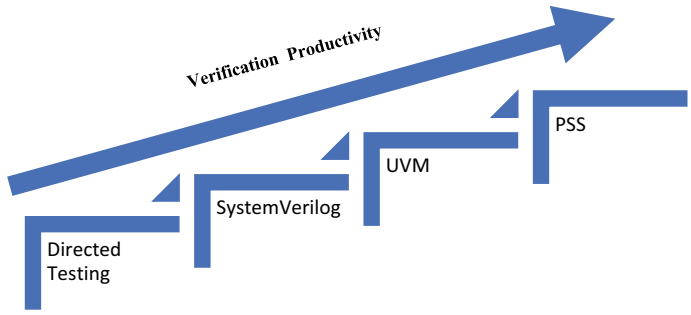


Fig. 4.30 Verification productivity

4.3 Post-Silicon Validation

Post silicon validation is the final process in semiconductor chip manufacturing. Pre-silicon verification techniques are not capable to establish full correctness of the design because simulation of the design is at least million orders of magnitude slower than the real execution of the design. Thus, during post-silicon validation step, we target the detection of design errors because of the fast execution speed of the manufactured chip. Post-silicon validation is the activity where one uses an actual silicon instead of an RTL model.

Although the designers always try to ensure an error-free hardware design through simulation before fabrication, few design errors (bugs) are likely to escape the simulation process. Such bugs subsequently appear post-silicon. Finding such bugs is time-consuming due to inherent invisibility of the hardware and hence significantly affect the time-to-market. As stated in the International Technology Roadmap for Semiconductors (ITRS) report, post-silicon validation remains a key challenge in the design process.

Scan chains and design for testability (**DFT**) can be utilized for a periodic monitoring scheme wherein their contents are taken out after some pre-defined intervals [43–46]. DFT can find defective units in large-scale production. DFT replace functional tests by **structural tests**, which focus on **physical defects** instead of functionality. Debugging at that phase can be quite hard. Moreover, fixing a bug requires a new silicon spin. Real defects are too numerous and hard to be analyzed. For exaple, an output is always shortened to "1" after fabrication. Standard Test Interface Language (**STIL**) is a general-purpose test pattern language used in DFT [47]. DFT insertion in the design cycle can be shown in Fig. 4.31. DFT is considered a structural test method.

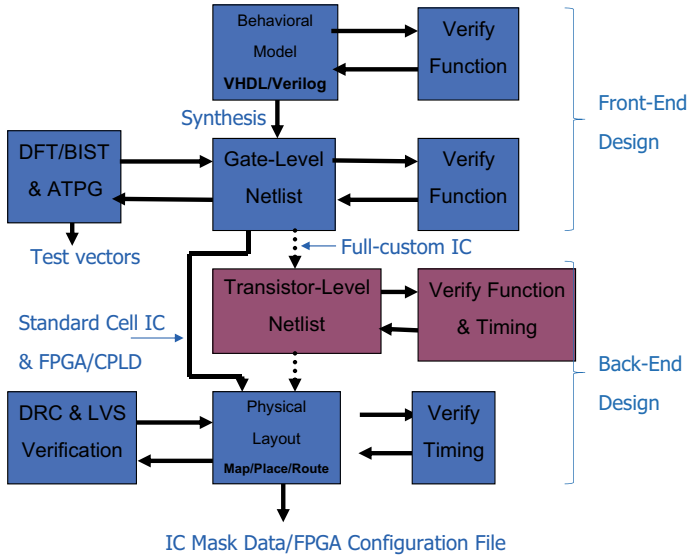


Fig. 4.31 DFT in the design cycle

4.3.1 DFT Verification: Gate-Level Simulations

As scaling approaches the physical limits of devices, we will continue to see increasing levels of process variations, noise, and defects. One of the main objectives of DFT is identifying the location of these defects. Design for testability (DFT) is a methodology. BIST and ATPG are two techniques or two flavors to implement DFT. Design for Test (DFT) is used make sure that no defective chips are shipped from the factory and to detect performance degradation and aging defects. Scan chains are generally inserted after the gate level netlist has been created. Hence, **gate level simulations** are often used to determine whether scan chains are correct. Figure 4.32 shows DFT insertion step in the ASIC design flow.

The process where the chip is tested for manufacturing defects by analyzing the logic on the chip to detect logic faults is called test process. During testing, the test process puts the circuit into the following test modes:

- **Normal mode**: corresponds to the intended operation of the chip. At this system mode, any logic dedicated to DFT purposes is active.
- **Scan-shift mode**: is the part of the test process in which the scan-flops act as shift registers in a scan chain. Test vector data is shifted into the scan flip-flops and the captured data are shifted out of the scan flip-flops.

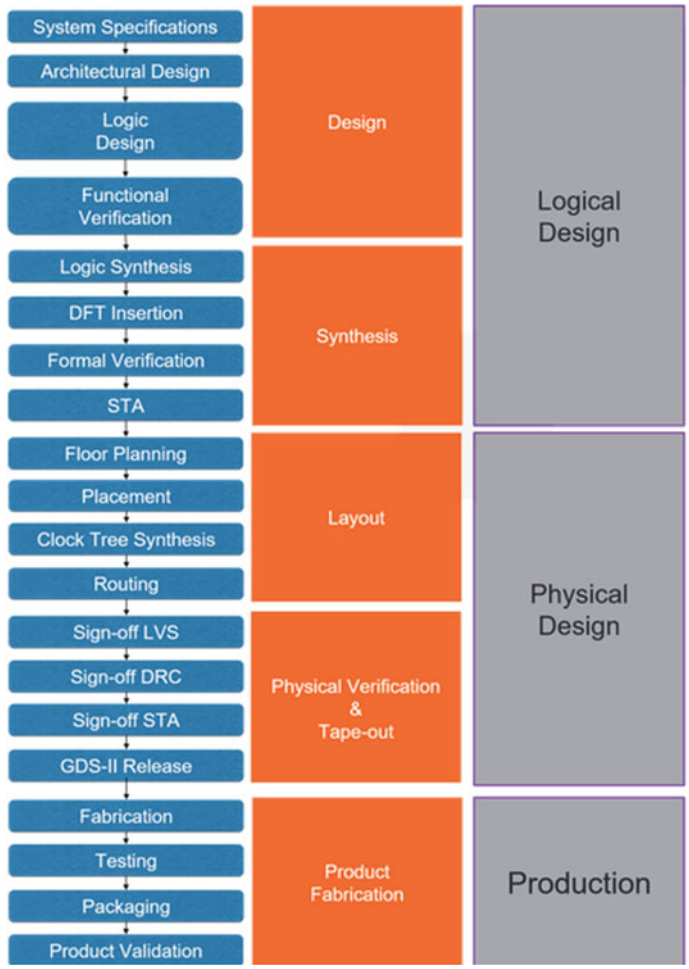


Fig. 4.32 DFT insertion step in the ASIC design flow

- **Capture mode**: is the part of the test process that analyses the combinational logic on the chip. The scan-flops act first as pseudo-primary inputs (using ATPG-generated test vector data), and then as pseudo-primary outputs (capturing the output of the combinational logic).

Nowadays, high-end chips contain tens of thousands of flops/latches so the traditional full scan ATPG vectors will take up a lot of Automatic Test Equipment (ATE) memory and test time to apply the vectors to the devices under test and as design size increased, the I/O interface did not scale with the increasing flops. As a result, the scan chains became too long. There are different ways to reduce ATE execution time. We can increase the

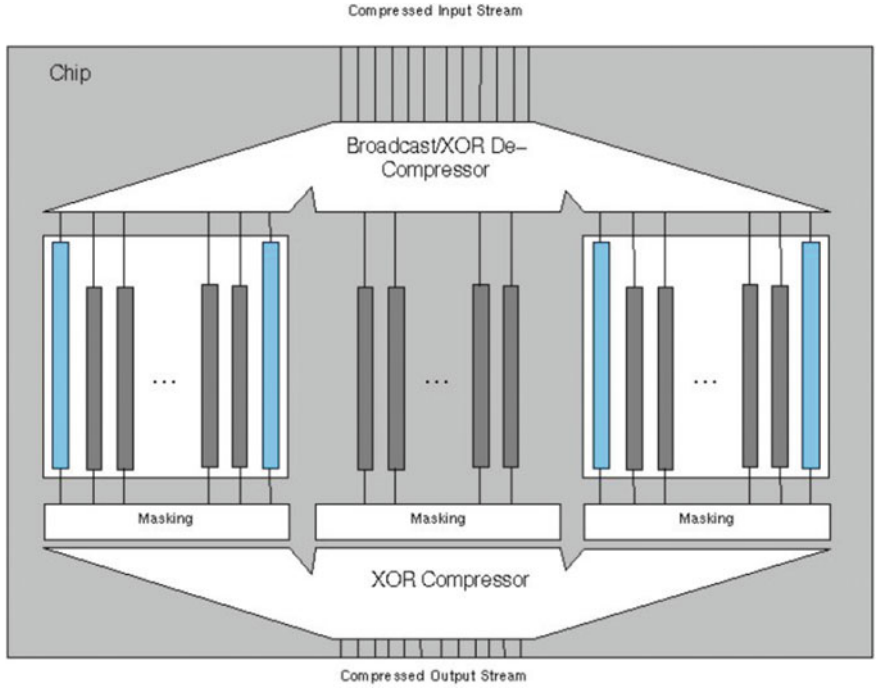


Fig. 4.33 High-level view of the DFT compression architecture

number of scan chains in the design, however, without any other design changes, this results in the need for more test pins and the ATE may not have enough test pins available. Other options could be to reduce the volume of test data, but it could decrease the number of patterns negatively affecting the test coverage, or increase the frequency of shifting patterns, in exchange for a raise in test power. More advanced options are inserting signature-based output verification, making the design harder to diagnose, or adding self-test LBIST, that produces overhead and less control over coverage. The solution to this problem is called **scan compression**, this is a technique of reducing test data volume and test application time (TAT) while retaining test coverage. This test compression solution requires ATPG capabilities via software and compression logic via adding some additional on-chip hardware before and after the scan chains, that means some form of de-compressor on the scan input side and some form of a compactor on the scan output side (Fig. 4.33) [48–50].

4.3.1.1 Automatic Test Pattern Generation (ATPG)

One of the available techniques for DFT is internal scan insertion, this technique replaces the normal flip-flops in the design with special flip-flops that contain built-in logic targeted for testability (scan-flops). Scan logic allows us to control and observe the sequential state

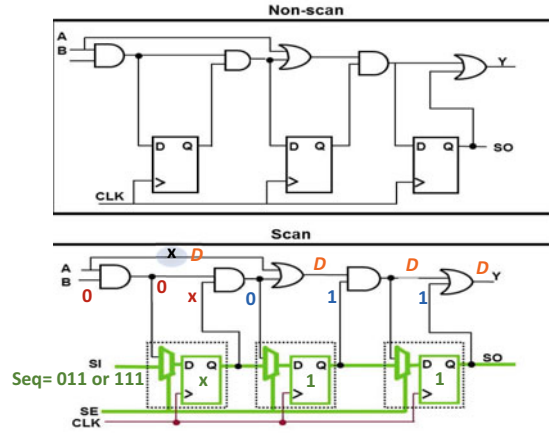


Fig. 4.34 ATPG example. SE signal stands for shift enable. SI is scan-in. SO is scan-out

of the design through the test pins of the scan flip-flops during test mode. By replacing the flip-flops with their scan-equivalent, the Automatic Test Pattern Generator (ATPG) tool can achieve higher fault coverage and generate a more compact test pattern set for the design.

ATPG algorithms generate vectors which stimulate physical defects and check the response. ATPG is an electronic design automation method/technology used to find an input (or test) sequence to distinguish between the correct circuit behavior and the faulty circuit behavior caused by defects. The generated patterns are used to test semiconductor devices after manufacture [15]. Figure 4.34 shows an ATPG example. It is used for (design for testability) **DFT** [51]. Patterns are applied from an **external** tester.

To **generate tests** for a design the first thing that must be done is to identify the behaviors that require testing and the targets of those tests. In the digital logic test industry, the most common targets for testing are **static and dynamic faults**. Static faults, such as stuck-at faults, affect the behavior of the design without regard to timing. Dynamic faults, also called transition or delay faults, affect the time dependent behavior of the design. Since these static and dynamic faults are the logic model equivalent of physical defects, we can see these referenced as logic faults. Other faults or objectives are identified for special testing purposes such as ensuring that a driver in the physical design is working prior to applying tests to the entire design.

4.3.1.2 Scan-Based BIST Technique

As test cost is directly related to the time each product stays on the tester, test time reduction has long been an important issue. To overcome these shortcomings, BIST (Built-in self-test) is found to be one of the promising techniques to verify the chips by performing self-testing with distinctive test methodologies and time reduction techniques.

A built-in self-test (BIST) engine is **built inside** the chip and requires only an access mechanism like the test access port (TAP) to start. When a device is powered on, BIST

can also check that the logic is working properly before starting any functional tests. The pseudorandom generator produces test patterns, which are loaded into **scan chains** to test the logic. The results are collected and compared to an expected signature. You can have **memory BIST (MBIST)** or **logic BIST (LBIST)** (Fig. 4.35). Logic BIST performs scan-chain operations (vector shift-in, capture, vector shift-out, comparison) and provides PASS/FAIL results controlling test signals and clock. Scan-Chains are **configurable shift register** (called **scan chain**), used in test mode to have access to internal nodes. Vectors are shifted in, applied to logic, response captured is shifted out. **MBIST** provides an effective solution for testing of large memories after manufacturing. MBIST is a self -test logic that generates patterns through data and address generator and read/write controller to detect possibly all faults that could be present inside a typical RAM cell (Fig. 4.36) [52–54]. Scan-chain insertion and connection is **automated** by most synthesis and PnR EDA tools. BIST is consistent with the pattern generator which provides stimulus to the circuit under test and output analyzer that monitors the response of the circuit and compares that response to a **fault-free** model to determine test coverage [55].

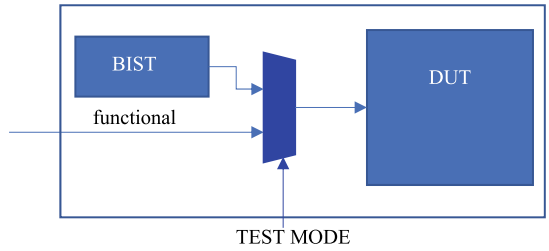


Fig. 4.35 Logic BIST. TEST-MODE controls signal needed to divide test from normal inputs

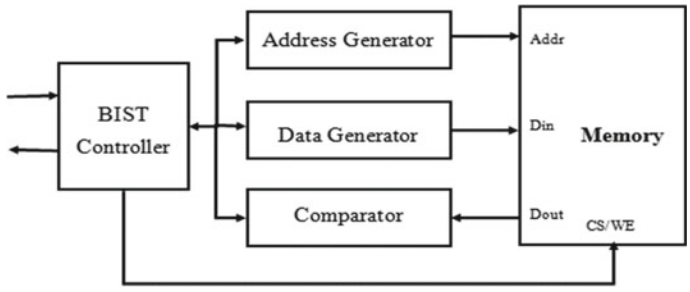


Fig. 4.36 Memory BIST

4.3.2 Physical Probing/Trace-Based Technique

To acquire data in real-time during post-silicon validation, one can connect the signals of interest directly to the device pins so that it can be monitored by external **logic analyzer** equipment [56].

Also, a modern IC design includes debug mechanisms such as **embedded logic analyzers (ELA)** to record values of internal design signals during silicon execution. An ELA consists of trigger and sampling units; trigger units are used to specify events that trigger recording initiation, and sampling units then record a small set of signals in the trace buffer for a specified number of cycles. The sampled signals can then be transferred from the trace buffer for off-chip analysis [57].

4.3.3 Synthesizable Assertions for Post-Silicon Debug

Since SVA is a separate language from the procedural language used to define RTL, the synthesis tool can simply ignore SVA within your design. However, there is a recent trend to have synthesized assertions.

Full visibility of the running design is always an important demand for rabid error localization, and efficient debugging process. However, it is difficult to be achieved, as there is nothing to tell about the state of internal nodes and signals, so it is necessary to find out a solution for this obstacle. There are several EDA tools that try to find a solution that allows observability of the running design, to find errors and to be able to debug the design. However, most of these tools do not use assertions to catch bugs, rather they define a trigger condition once upon it occurs, the software tool captures the state of signals, registers, and memories, then send it to a graphical waveform display. This approach may help if the examined DUT is small and has a finite amount of information to be displayed, but if the DUT is large (i.e., SoC) with plentiful signals and different IPs/ cores, then it will not be a very reasonable nor efficient task for the human operator. Lately, several have been released to target synthesis of assertions; but few of them are generic and support all SystemVerilog Assertions' operators. On the other hand, although several researches have been done on how to synthesize assertions into hardware checkers, but they are different in the approach of how each one implements assertions.

The synthesized assertions provide a way to compare, how the design is really running, with how it must run in the way intended, which makes bug localization and fixing easier, especially for those corner bugs, which requires very complicated test benches to be detected if we use the traditional verification methodologies [58–62].

Also, we can re-create the environment in the testbench and SVA properties to locate and recognize errors and bugs discovered in post-silicon. In fact, returning to the testbench AND property space with the design sometimes provides better observability than the post-silicon environment.

4.4 Full-Chip SoC Verification: SoC Integration Testing

SoCs are composed of primarily pre-verified third-party IPs and some in-house IPs. A full chip SoC is generally too complicated to test as a single unit as it is too **complicated** to control the deep internals. Thus, "**divide and conquer**" concept is used to break it into manageable pieces (IPs or blocks). Each piece can be individually tested, and then more limited and faster testing done at the full chip level to exercise interactions between the pieces. In other words, IP and Subsystem Verification is a vital step towards Full-chip SoC verification along with using all the previous verification techniques. So Ful-chip SoC verification is more about **integration testing**. Moreover, SoC verification **includes functional, timing, power, reliability, thermal, EMI, security, and mechanical verification** (Fig. 4.37). All these verifications should be done in pre and post silicon phases. Recent advances in systems, packaging and process technologies are enabling the computation of hundreds of teraflops per chip. This enormous demand for computing per Silicon-based SoC creates new challenges with respect to storage, memory, security, reliability, power, and functional verification.

While computer-aided power analysis tools can provide power consumption estimates for various circuit blocks, these estimates can substantially deviate from the actual power consumption of working silicon chips. Thus, Post-silicon **power characterization** is inevitable thermal infrared emissions from the backside of silicon die are captured using a state-of-the-art **infrared camera**, then characterization techniques are applied to invert the thermal emissions to power [63]. Moreover, ML techniques Can use existing embedded **temperature sensors** and **workload-independent** utilization information, which are available in real-time to track thermal activities [64]. There are many temperature forcing devices that could be used to inject heat into silicon chip to help testing thermal.

SoC **security** is vital in designing trustworthy systems. Assertion-based verification can be applied to detect SoC security vulnerabilities. Authors in [65] perform automated

Fig. 4.37 SoC verification tasks

vulnerability analysis of RTL models to generate security assertions. Experimental results show that the generated security assertions can detect a wide variety of vulnerabilities.

Design for manufacturing (DFM) refers to **actions taken during the physical design stage of IC development to ensure that the design can be accurately manufactured**. Dummy transistors are commonly used as part of DFM techniques in integrated circuit (IC) design. They serve several important purposes to enhance manufacturability and improve the performance and reliability of the final IC.

4.5 *Data-Driven* Verification: AI-Powered Verification

Data from test runs along with suitable machine learning algorithms can be used to accelerates verification closure or can be used to localize bugs in designs. Data-Driven Verification, particularly in the realm of AI-Powered Verification, represents a cutting-edge methodology that harnesses the potential of machine learning to analyze data obtained from test runs. This approach significantly accelerates the verification closure process and enhances bug detection and localization in complex designs. The heart of Data-Driven Verification lies in the extraction of meaningful insights from extensive datasets collected during testing phases. Utilizing sophisticated machine learning algorithms, these insights are transformed into predictive models capable of recognizing patterns, correlations, and anomalies within the verification data. By leveraging the power of AI, engineers can efficiently prioritize their efforts based on critical areas identified by these models. This approach not only expedites the verification closure by predicting potential challenges but also streamlines bug detection and resolution through targeted analysis. Overall, Data-Driven Verification, bolstered by artificial intelligence, represents a paradigm shift in verification methodologies, offering efficiency improvements and predictive capabilities that redefine the landscape of hardware and software verification.

4.6 Conclusions

This chapter introduces to Pre-Silicon Verification and Post-Silicon Validation Methodologies. The chapter begins by explaining that testing can be divided into two categories: Pre-Silicon verification and post-Silicon validation. Pre-Silicon verification deals with simulating and verifying the RTL code, while post-Silicon validation deals with silicon validation after fabrication. The chapter goes on to provide a big picture overview of different verification technologies. Additionally, the advantages of different verification methodologies are shown. The verification methods are divided between those based on simulations of test benches and those that are formal. The chapter then focuses on post-silicon validation, which is the final process in semiconductor chip manufacturing. It explains that pre-silicon verification techniques are not capable of establishing full

correctness of the design because simulation of the design is at least million orders of magnitude slower than the real execution of the design. Thus, during post-silicon validation step, we target the detection of design errors because of the fast execution speed of the manufactured chip.

References

1. Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. Coverage evaluation of post silicon validation tests with virtual prototypes. In DATE, 2014.
2. Li Lei, Fei Xie, and Kai Cong. Post-silicon conformance checking with virtual prototypes. In DAC, 2013.
3. W. Müller, W. Rosenstiel, and J. Ruf, SystemC: Methodologies and Applications. Springer, 2003.
4. Z. Navabi, "The Role of SystemC in the Evolution of Hardware Design", Worcester Polytechnic Institute Vanthournout, "SoC design methodology Using SystemC", Coware, 2003.
5. F. Ghenassia, "Transaction Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Springer, 2005.
6. S. Park and S.-I. Chae, "A C/C++-based functional verification framework using the SystemC verification library," Rapid System Prototyping, IEEE International Workshop on, vol. 0, pp. 237–239, 2005.
7. K. R. G. da Silva, E. U. K. Melcher, G. Araujo, and V. A. Pimenta, "An automatic testbench generation tool for a SystemC functional verification methodology," in SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design. New York, NY, USA: ACM, 2004, pp. 66–70.
8. Black, David & Donovan, Jack & Bunton, Bill & Keist, Anna. (2009). Why SYSTEMC: ESL and TLM. https://doi.org/10.1007/978-0-387-69958-5_1.
9. F. Ghenassia et al., Transaction-Level Modeling with SystemC. Springer, 2005.
10. D. Gajski, L. Cai, "Transaction Level Modeling: An Overview", Center for Embedded Computer Systems, University of California, Irvine, 2004.
11. https://www.electronicproducts.com/when-to-use-simulation-when-to-use-emulation/
12. N. K. Doshi, S. Suryawanshi, and G. N. Kumar, "Development of generic verification environment based on UVM with case study on HMC controller," in 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT), pp. 550–553, May 2016.
13. S. Jain, P. Govani, K. B. Poddar, A. K. Lal, and R. M. Parmar, "Functional verification of DSP based on-board VLSI designs," in 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), pp. 1–4, Jan 2016.
14. https://verificationacademy.com/resources/technical-papers/from-simulation-to-emulation-a-fully-reusable-uvm-framework
15. http://semiengineering.com/kc/knowledge_center/Scan-Test/173
16. K. Salah "A UVM-based smart functional verification platform: Concepts, pros, cons, and opportunities." 9th International Design & Test Symposium (IDT), 2014.
17. Sutherland, S. and Fitzpatrick, T., 2015. UVM Rapid Adoption: A Practical Subset of UVM.
18. https://verificationacademy.com/cookbook/uvm/performance_guidelines#UVM.C2.A0Testbench_Run-Time_Performance_Guidelines

19. Mohamed, K.S. (2016). New Trends in SoC Verification: UVM, Bug Localization, Scan-C0068ain-Based Methodology, GA-Based Test Generation. In: IP Cores Design from Specifications to Production. Analog Circuits and Signal Processing. Springer, Cham. https://doi.org/10.1007/978-3-319-22035-2_6.
20. Chockler, H., Kupferman, O., and Vardi, M., "Coverage Metrics for Formal Verification," International Journal on Software Tools for Technology Transfer, vol. 8, no. 4–5, 2006, pp. 373–386.
21. B. R. Harry Foster Lawrence Loh and V. Singhal, "Guidelines for creating a formal verification testplan," New York, USA: DAC, 2006.
22. E. Seligman, T. Schubert, and M. V. A. K. Kumar, Formal Verification, An Essential Toolkit for Modern VLSI Design. Morgan Kaufmann Publishers, 2015.
23. http://vlsi.pro/formal-verification-an-overview/
24. https://docs.verilogtorouting.org/en/latest/tutorials/timing_simulation/
25. https://peterfab.com/ref/verilog/verilog_renerta/mobile/source/vrg00052.htm
26. Ashok B Mehta. 2020. System verilog assertions. In System Verilog Assertions and Functional Coverage. Springer, 11–31.
27. Yangdi Lyu and Prabhat Mishra. Automated Test Generation for Activation of Assertions in RTL Models. In 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 223–228.
28. P. Narain and C. Cummings, "Clock Domain Crossing Demystified: The Second Generation Solution for CDC Verification," Sunburst Design, 2008.
29. Jackie Hsiung, Ashish Hari, Sulabh-Kumar Khare, "Preventing Chip-Killing Glitches on CDC Paths with Automated Formal Analysis", Mediatek, DVCon 2018.
30. Rajesh Velegalati, Kinjal Shah and Jens-Peter Kaps "Glitch Detection in Hardware Implementations on FPGAs using Delay Based Sampling Techniques" 16th Euromicro Conference on Digital System Design, 2013.
31. Plassan, Guillaume, et al. "Improving the Efficiency of Formal Verification: The Case of Clock-Domain Crossings." *24th IFIP/IEEE International Conference on Very Large Scale Integration-System on a Chip (VLSISOC)*. Springer International Publishing, 2016.
32. https://www.eetimes.com/unleash-the-power-of-formal-technology-for-cdc-verification/.
33. https://www.design-reuse.com/articles/36975/analysis-of-rdc-paths-for-a-million-gate-soc.html
34. Ye Li, Yun-Ze Cal, Ru-Po Yin and Xiao-Ming Xu, "Fault diagnosis based on support vector machine ensemble," 2005 International Conference on Machine Learning and Cybernetics, Guangzhou, China, 2005, pp. 3309–3314, Vol. 6.
35. http://www.syncad.com/web_manual_testbencher/test_bench_generator_main_index.html?golden_reference_models.htm
36. El-Shiekh, Ahmad, Ahmad El-Alfy, Ahmad Ammar, Mohamed Gamal, Mohammed Dessouky, Khaled Salah, and Hassan Mostafa. "IPXACT-Based RTL Generation Tool." In 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES), pp. 71–74. IEEE, 2020.
37. IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows.
38. Timo D. Hämäläinen, & Esko Pekkarinen. (2015). Kactus2: Open Source IP-XACT tool. Accellera. (2018). IP-XACT User Guide.
39. https://www.accellera.org/images/downloads/standards/Portable_Test_Stimulus_Standard_v20.pdf
40. A. Vintila, I. Tolea, AMIQ Consulting, "Portable Stimulus Driven SystemVerilog/UVM verification environment for the verification of a high-capacity Ethernet communication bridge," Design and Verification Conference US, 2019.

41. Bhatnagar, Gaurav & Brownell, David. (2018). Portable Stimulus vs Formal vs UVM A Comparative Analysis of Verification Methodologies Throughout the Life of an IP Block, DVCON 2018.
42. https://www.design-reuse.com/articles/52508/soc-verification-flow-and-methodologies.html
43. Kai hui Chang, I. L. Markov, and V. Bertacco. Automating post-silicon debugging and repair. In 2007 IEEE/ACM International Conference on Computer-Aided Design, pages 91–98, Nov 2007. doi: https://doi.org/10.1109/ICCAD.2007.4397249.
44. K. Rahmani, S. Proch, and P. Mishra. E_cient selection of trace and scan signals for post-silicon debug. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 24(1):313–323, Jan 2016. ISSN 1063-8210. doi: https://doi.org/10.1109/TVLSI.2015.2396083.
45. M. Gao, P. Lisherness, and K. T. Cheng. Post-silicon bug detection for variation induced electrical bugs. In 16th Asia and South Paci_c Design Automation Conference (ASP-DAC 2011), pages 273–278, Jan 2011. doi: https://doi.org/10.1109/ASPDAC.2011.
46. A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann. A unified methodology for pre-silicon verification and post silicon validation. In DATE, 2011.
47. IEEE Standards Association, Standard Test Interface Language (STIL) for Digital Test Vectors, IEEE std. 1450–1999.
48. M. J. Geuzebroek, J. T. van der Linden, and A. J. van de Goor. Test point insertion for compact test sets. In Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159), pages 292–301, 2000.
49. R. Kapur, S. Mitra, and T. W. Williams. Historical perspective on scan compression. IEEE Design Test of Computers, 25(2):114–120, 2008.
50. N. A. Touba. Survey of test vector compression techniques. IEEE Design Test of Computers, 23(4):294–303, 2006.
51. Xinli Gu, Weili Wang, K. Li, Heon Kim, and S. S. Chung. Re-using dft logic for functional and silicon debugging test. In Test Conference, 2002. Proceedings. International, pages 648–656, 2002. doi: https://doi.org/10.1109/TEST.2002.1041816.
52. Liang Che Li, Wen Hsuan Hsu, Kuen Jong Lee, and Chun Lung Hsu. An efficient 3DIC on-chip test framework to embed TSV testing in memory BIST. 20th Asia and South Pacific Design Automation Conference, ASP-DAC 2015, pages 520–525, 2015. doi: https://doi.org/10.1109/ASPDAC.2015.7059059.
53. Chung Fu Lin and Yeong Jar Chang. An area-efficient design for programmable memory built-in self test. 2008 International Symposium on VLSI Design, Automation, and Test, VLSI-DAT, pages 17–20, 2008. doi:https://doi.org/10.1109/VDAT.2008.4542402.
54. Reinaldo Silveira, Qadeer Qureshi, and Rodrigo Zeli. Flexible architecture of memory BISTs. 2018 IEEE 19th Latin-American Test Symposium, LATS 2018, 2018-January:1–6, 2018. doi: https://doi.org/10.1109/LATW.2018.8349666.
55. Manibha Sharma and Jasdeep Dhanoa. Smart Logic Built In Self-Test In SOC. IEEE International Conference on Recent Advances and Innovations in Engineering, 2020.
56. B. J. Hammond. Development in Logic Analysers. In Proceedings of IEE Colloquium on Instrumentation in Electronic Product Manufacture, pages 2/1,1989.
57. K. Rahmani, S. Ray and P. Mishra, "Postsilicon Trace Signal Selection Using Machine Learning Techniques," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 570–580, Feb. 2017, doi: https://doi.org/10.1109/TVLSI.2016.2593902.
58. Sayantan Das, RiziMohanty, PallabDasgupta, P.P. Chakrabarti, "Synthesis of System Verilog Assertions", Design, Automation and Test in Europe, München, Germany, 2006
59. Ivan Kastelan, ZoranKrajacevic, "Synthesizable System Verilog Assertions as a Methodology for SoC Verification", First IEEE Eastern European Conference on the Engineering of Computer Based Systems, 2009.

60. Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00), pages 538–542, 2000.
61. Boulé, Marc, and Zeljko Zilic. "Efficient automata-based assertion checker synthesis of SEREs for hardware emulation." 2007 Asia and South Pacific Design Automation Conference. IEEE, 2007.
62. Boulé, Marc, Jean-Samuel Chenard, and Zeljko Zilic. "Adding debug enhancements to assertion checkers for hardware emulation and silicon debug." 2006 International Conference on Computer Design. IEEE, 2007.
63. R. Cochran, A. N. Nowroz and S. Reda, "Post-silicon power characterization using thermal infrared emissions," *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Austin, TX, USA, 2010, pp. 331–336, doi: https://doi.org/10.1145/1840845.1840914
64. S. Sadiqbatcha, J. Zhang, H. Amrouch and S. Tan, "Real-Time Full-Chip Thermal Tracking: A Post-Silicon, Machine Learning Perspective" in IEEE Transactions on Computers, vol. 71, no. 06, pp. 1411–1424, 2022. doi: https://doi.org/10.1109/TC.2021.3086112.
65. Lyu Yangdi and Mishra Prabhat. 2020. System-on-chip security assertions. arXiv preprint arXiv:2001.06719 (2020).

HW/SW Co-Verification and Co-Debugging

5

With the growing complexity of modern SoCs, debug, verification, and validation have become critical design concerns. Traditionally, the software developers wait for a hardware prototype for the final system integration. Solving problems that may arise during integration such as misunderstanding of specifications may be costly and time-consuming. So, moving the system integration phase forward in the design cycle (creating a Hardware/Software (HW/SW) co-verification) would help in detecting these integration problems earlier and minimizing the product time-to-market [12]. Co-Verification techniques is summarized in Fig. 5.1, where it includes Co-Simulation and Co-Emulation. Validation is a continuous process applied in different phases of the development process and to different models of the system to ensure conformance with various requirements of the system [1, 2]. After integration, we need to verify system integration. Hardware/Software (HW/SW) co-verification is the process of testing and validating the interactions between hardware and software components of a system. It ensures that the hardware and software components work together seamlessly to achieve the intended functionality of the system. The co-verification process involves simulating the behavior of the hardware and software components of the system in a virtual environment to test their interactions. This simulation enables engineers to identify and resolve any potential issues before the actual hardware and software are integrated into a final system. An embedded system is a combination of hardware and software designed to perform a specific task. The hardware consists of a microcontroller or a processor, sensors, and other electronic components, while the software includes the firmware and the application code. A System-on-Chip (SoC) is a complex electronic system that integrates multiple components, including a processor, memory, and peripherals, onto a single chip. Field-Programmable Gate Arrays (FPGAs) are programmable electronic devices that can be configured to perform specific

K. S. Mohamed, *Heterogeneous SoC Design and Verification*, Synthesis Lectures on Digital Circuits & Systems, https://doi.org/10.1007/978-3-031-56152-8_5

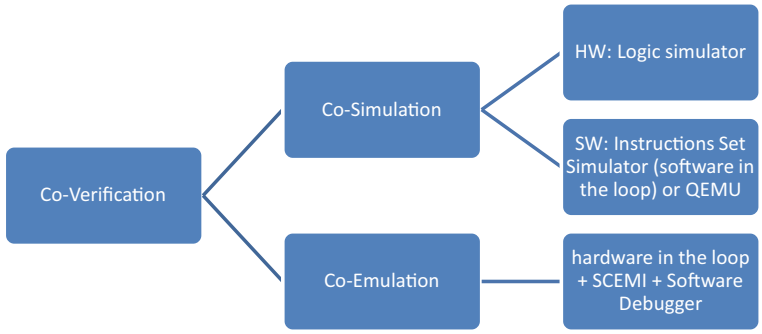


Fig. 5.1 Co-verification techniques

functions. In FPGA co-verification, the hardware and software components are tested together to ensure that the FPGA functions as expected. In embedded system or SoC or FPGA co-verification, the hardware and software are tested together to ensure that the system functions as expected. The co-verification process involves simulating the behavior of the hardware and software components in a virtual environment. The software is tested using a software simulator (**instruction set simulator**), while the hardware is tested using a hardware simulator (**cycle accurate simulator**). The test cases are designed to exercise the interactions between the hardware and software components of the system. Hardware/software co-debugging is the process of debugging a system that contains both hardware and software components. This can be a challenging task because it requires knowledge of both hardware and software. In **co-debugging**, hardware and software engineers work together to debug issues in the system. The software engineer may use software debugging tools, such as a debugger, to identify and fix issues in the software. The hardware engineer may use hardware debugging tools, such as a logic analyzer, to identify and fix issues in the hardware. In an embedded system, both hardware and software are tightly integrated. If the software is not working correctly, it may be difficult to determine if the issue is in the hardware or software. In this case, the hardware and software engineers may work together to identify the root cause of the issue. For example, they may use a logic analyzer to examine signals on the hardware and a debugger to examine the software.

5.1 Co-Simulation

After HW/SW partitioning and once RTL for the hardware and the C code for SW is ready, then you can simulate it and see whether it meets the design requirements or not. With the suitable Co- simulator, you can see your C/C++ code and see correspondingly the hardware signals in the waveform viewer. HW/SW co-simulation tools cover a very

important role in a HW/SW co-design flow because they allow a fast and correct analysis of the system properties [10, 11].

In co-simulation, the hardware and software components are simulated together, allowing the engineers to verify that the software works correctly with the hardware. This is done by simulating the behavior of the hardware using models or simulation software and running the software on a simulated hardware platform. When developing a new microprocessor, hardware and software engineers can use co-simulation to test the processor before it is built. They can create a model of the processor and simulate it with software that runs on the processor. This allows them to verify that the software works correctly with the processor and to identify any issues that need to be addressed before the processor is built, reducing development time and costs.

Software Simulation refers to an event-based logic simulator that operates by propagating input changes through a design until a steady state condition is reached. Software simulators run on workstations and use Verilog or VHDL as a simulation language to describe the design and the testbench. Some portions of the hardware may be more abstractly modeled using a high-level language, such as SystemC. **Software in the loop** testing means the design is co-simulated on host machine [19]. In HW/SW co-simulation, the design under test, including hardware and software components, is executed in a simulator.

C/RTL co-simulation is a technique used in hardware design and verification to simulate the interaction between the hardware at the Register-Transfer Level (RTL) and software running on a Central Processing Unit (CPU) written in the C programming language. The co-simulation process involves integrating a hardware model, typically written in RTL, with a software model, written in C, and simulating their interactions. The software model can communicate with the hardware model through a set of well-defined input and output interfaces, allowing the two to interact in a manner that mimics real-world behavior. The co-simulation technique is useful because it allows designers to test the functionality of a hardware design in conjunction with software that will eventually run on the design, ensuring that the hardware and software components work together correctly. This can help to reduce design iterations and improve overall design quality. C/RTL co-simulation uses a C test bench, running the main() function, to automatically verify the RTL design running in behavioral simulation. Direct Programming Interface (DPI) in SystemVerilog serves as translator, allowing hardware models to communicate with software processes (Fig. 5.3). Verilog hardware description language is used to describe the hardware and C or C++is used to describe the software. Here is an example of how C/RTL co-simulation might be used in practice: Suppose we are designing a hardware module that performs audio signal processing. The module has a set of inputs for audio data and a set of outputs for processed audio data. We want to test the functionality of this module in conjunction with a software application that will eventually run on a CPU. To perform the co-simulation, we would first develop an RTL model of the audio signal processing module, including input and output interfaces. We would then develop a

C program that sends audio data to the hardware module and receives processed audio data from the module. We would then use a co-simulation tool to simulate the interaction between the C program and the RTL model. The co-simulation tool would execute the C program on a virtual CPU and communicate with the hardware model through the input and output interfaces. During the simulation, we would monitor the behavior of the hardware and software components to ensure that they are interacting correctly. We would also analyze the output from the audio signal processing module to ensure that it is producing the expected results. If any issues were discovered during the simulation, we could adjust either the hardware or software components and rerun the simulation until we were satisfied with the overall design [20].

Co-simulation means mixed languages simulators. Co-simulation is important for early integration. Co-simulation correlates software source code to simulation timestamp. Co-simulation is useful in realizing an efficient simulation process as it enables different simulations to work together. Functional Mock-up Interface (FMI) is a generic tool for co-simulation and it is a standardized interface (Figs. 5.2 and 5.4), [3]. The Functional Mock-up Interface (FMI) is proposed as an interface for model exchange and co-simulation of dynamic models. FMI can be used for simulating complex systems that may include multiple disciplines, such as mechanical, electrical, and hydraulic systems. This interface is devised to support model exchange and co-simulation. A component which implements the FMI interface is known as Functional Mock-up Unit (FMU). As a package, it contains model functionality as C code and information about the model as XML document. For model exchange, one modeling environment exports a model as FMU. The FMU can be imported and used by another FMI-compliant modeling environment. With FMI two or more simulation environments can be coupled to form a co-simulation environment. One of the simulation environments must be master while all others are slaves. The master simulator is responsible for synchronized data exchanges and executions of all slave simulators relative to its own execution cycles. An FMI adapter needs to be developed for slave simulator to communicate with the master. The adapter is a dedicated simulation component responsible for information retrieval from FMU and data exchange between master and slave simulators.

FMI provides two main functionalities, model exchange, and co-simulation. In the Model Exchange mode, the simulation model is exported as a functional mock-up unit (FMU). An FMU is a standalone executable file that contains the simulation model and the

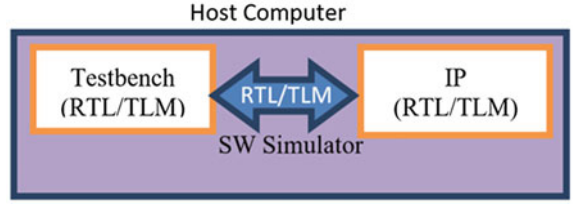


Fig. 5.2 HW/SW co-simulation: conceptual-level

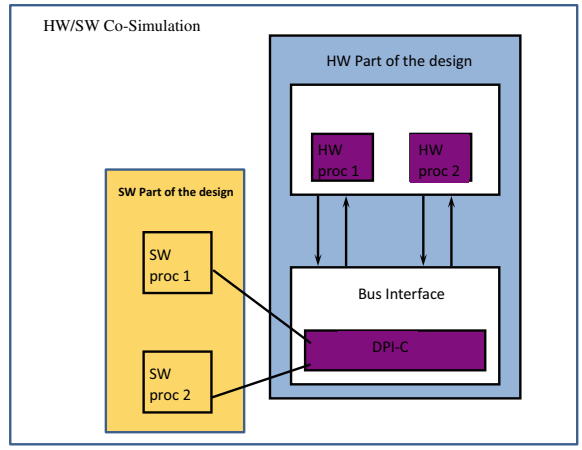


Fig. 5.3 HW/SW co-simulation

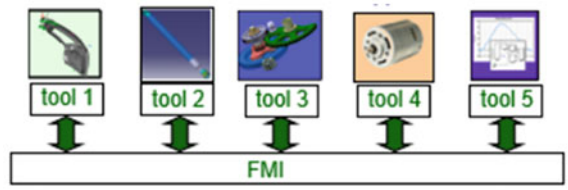


Fig. 5.4 FMI example

required inputs and outputs. FMUs can be imported into different simulation environments that support FMI, and the simulation results can be obtained without requiring access to the original simulation environment. In the co-simulation mode, the simulation model is split into two parts, a master simulation environment, and a slave simulation environment. The master simulation environment provides inputs to the slave environment, and the slave environment provides outputs to the master environment. The two environments communicate through the FMI interface. This allows for co-simulation between two or more software tools.

QEMU is an open-source machine emulator and virtualizer that emulates a wide range of CPU architectures (Fig. 5.5). Together with a HW/SW co-simulation platform, it simplifies the verification process for challenging hybrid designs [16]. It can emulate a wide range of hardware platforms and architectures, including $\times 86$, ARM, PowerPC, and MIPS. QEMU can also emulate a variety of peripherals and devices, such as network adapters and storage devices, enabling users to create virtual environments that closely resemble real-world systems. Suppose we are a software developer who wants to test an application on a different architecture than our development machine. For example, we want to test our application on an ARM-based system. Rather than purchasing an ARM-based computer, we can use QEMU to emulate an ARM-based system on our development machine. To do this, we would first install QEMU on our development machine.

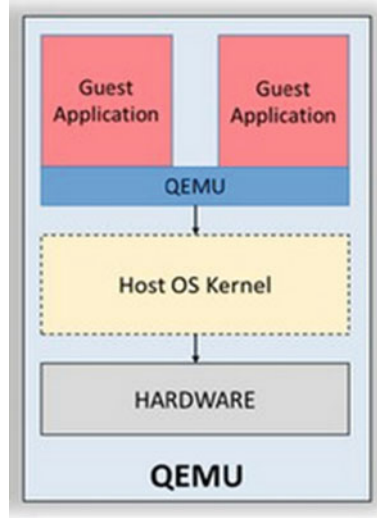


Fig. 5.5 QEMU representation

We would then download an image of an ARM-based operating system, such as Raspbian for the Raspberry Pi. This image would include the necessary drivers and software to run the ARM-based system. We would then use QEMU to create a virtual machine and load the ARM-based operating system image. QEMU would emulate an ARM-based system, including the CPU, memory, and peripherals. We could then install our application on the emulated system and test it as if it were running on a real ARM-based system. QEMU can be used to test applications on different operating systems and hardware platforms without requiring separate physical machines. QEMU can be used to migrate physical machines to virtual machines. This can be useful for consolidating multiple physical machines onto a single host machine or for moving systems to a cloud environment. Overall, QEMU is a versatile tool that can be used for a variety of tasks related to virtualization and emulation. It provides a powerful and flexible environment for developers.

5.2 Co-Emulation

Emulation refers to the process of mapping an entire design into a hardware platform that performs parallel processing to increase performance. There is no constant connection to the workstation during execution, and the hardware platform receives no input from the workstation. By eliminating the connection to the workstation, the hardware platform now runs at its full speed and does not need to wait for any communication.

Co-Emulation has many techniques. SCEMI protocol is an essential one for Co-Emulation (Table 5.1). **SCEMI** stands for Standard Co-Emulation Modeling Interface. It describes a standard way for untimed software to interact with timed hardware. We use

SCEMI to communicate between **software** running on the host processor and a **hardware** design running on the FPGA. The **transactor** converts the high-level commands from the testbench into the wire-level, protocol-specific sequences required by the DUT, and vice versa. **Transactors** can be implemented in SystemVerilog or C++. Transactors are used to interface between hardware design and software testbench environments when using hardware verification languages like SystemVerilog or C++ for verification tasks. They enable communication and synchronization between the hardware and software components, allowing simulation or emulation of the hardware design with test scenarios written in a higher-level programming language.

SCEMI protocol provides a handshaking/bridging mechanism between hardware platform and simulator as depicted in Fig. 5.6.

SCEMI facilitates the co-emulation of hardware and software designs in a virtual environment. It is a standard interface used to connect the two models and simulate the interaction between them. The SCE-MI protocol consists of a set of rules and guidelines that define how hardware and software models communicate with each other during co-emulation. The protocol defines the signals, data types, and communication methods

Table 5.1 Difference between co-simulation and co-emulation

	Co-simulation	Co-emulation
HW/SW communication technique	DPI-C	SCEMI
Speed	Slower than co-emulation	Faster than co-simulation

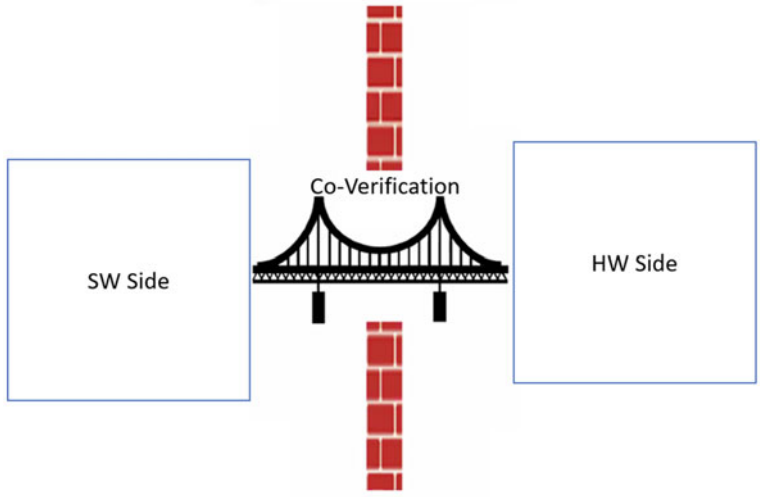


Fig. 5.6 SCEMI bridging concept

used to exchange information between the models. In the development of processors, the SCE-MI protocol is used to co-simulate the processor hardware model and the software model. The processor hardware model is simulated in an HDL, while the software model is simulated on a processor model. SCE-MI is used to connect the two models, and the software can interact with the processor model as if it were running on the actual hardware.

Co-simulation and co-emulation are both techniques used to simulate the interaction between hardware and software designs. While both techniques serve similar purposes, there are key differences between them. The key difference between co-simulation and co-emulation is the former typically couples software models to a traditional HDL simulator interface through a proprietary **API**, whereas the latter couples software models an emulator through an optimized transaction-oriented interface, such as **SCEMI**. **Hardware in the loop** testing means using prototyping hardware platform to verify the design, i.e. using real hardware to verify the design [17, 18]. Co-Emulation Conceptual-Level is shown in Fig. 5.7. The key difference between co-simulation and co-emulation is in the level of abstraction and the timing of the simulation. Co-simulation operates at a higher level of abstraction, where the hardware and software components are simulated separately, and the communication between them is simulated at a higher level. Co-emulation operates at a lower level of abstraction, where the hardware and software components are simulated together, and the communication between them is simulated at a lower level. In terms of timing, co-simulation is typically used in the early stages of the design process when the focus is on verifying the functionality of the hardware and software components, while co-emulation is used in the later stages of the design process when the focus is on verifying the interaction between the hardware and software components.

In-Circuit Emulation (**ICE**) refers to the use of external hardware coupled to a hardware platform for the purpose of providing a more realistic environment for the design being simulated. This hardware commonly takes the form of circuit boards, sometimes called target boards or a target system, and test equipment cabled into the hardware platform. Emulation without the use of any target system is defined as targetless emulation (Fig. 5.8). Hardware/software co-emulation is a valuable technique that helps to identify and fix issues early in the development cycle. It is used in various industries, including automotive, mobile devices, and IoT, to ensure that the software and hardware components of electronic systems work seamlessly together. In-Circuit Emulation (ICE) is a

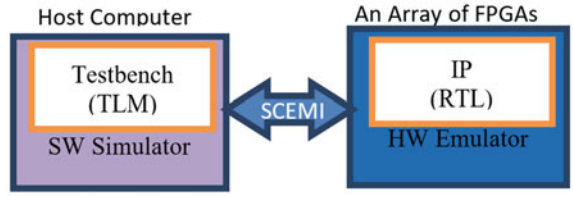


Fig. 5.7 Co-emulation: conceptual-level

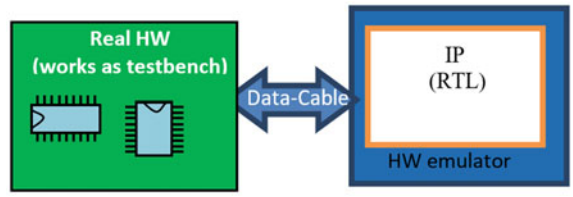


Fig. 5.8 ICE co-emulation representation

hardware-based debugging technique used to test and debug software on a target hardware platform. In ICE, the actual hardware platform is used for debugging and testing, instead of a software simulation or model. ICE is typically used in the later stages of the design process, after the hardware design has been completed, and the software has been loaded onto the target hardware. The ICE system consists of an emulator device, which is connected to the target hardware platform, and a host computer, which is used to control the emulation process. The emulator device is designed to mimic the behavior of the target hardware platform and is used to replace the actual processor on the target hardware platform. The emulator device also provides the necessary debugging and tracing features, such as breakpoints, watchpoints, and trace buffers. The host computer is used to control the emulation process and to interact with the emulator device. The host computer typically runs a software debugger, which communicates with the emulator device and allows the developer to test and debug the software on the target hardware platform. The software debugger provides features such as source-level debugging, memory viewing, and execution control. One advantage of ICE is that it provides a real-world testing environment, where the software can be tested and debugged on the actual hardware platform, which is essential for debugging hardware-dependent issues. Another advantage of ICE is that it allows the developer to test and debug the software at full speed, without the overhead of a simulation or model. However, there are also some disadvantages to ICE. One major disadvantage is that it can be expensive to set up, as it requires specialized hardware, such as an emulator device, and a host computer with debugging software. Additionally, ICE can be difficult to set up and configure, as it requires precise timing and synchronization between the emulator device and the target hardware platform. In-Circuit Emulation is a hardware-based debugging technique used to test and debug software on a target hardware platform. ICE provides a real-world testing environment and allows the developer to test and debug the software at full speed. However, ICE can be expensive and difficult to set up and configure.

5.3 Co-Debugging

When an error is found in a set of instructions given to a computer, it is called a bug. The process of finding the error in a set of computer instructions is called debugging. **Bug source**s can be summarized in Table 5.2.

Table 5.2 Bug types

Bug type	Description
Sporadic/intermittent	Occurs when unanticipated conditions trigger an unexpected error
Heisenburg (time sensitive)	A bug that seems to disappear or change when you try to debug it
Race condition	Arise in software when a computer program has multiple code paths that are executing at the same time
Buffer overflow	Exists when a program attempts to put more data in a buffer than it can hold

Software bugs can arise from a wide variety of sources, ranging from programming errors to design flaws. Here are some common software bug sources with examples:

1. Programming errors: These types of errors arise from coding mistakes made by the programmer. Some examples include:
   - Syntax errors: These errors occur when the programmer writes incorrect code syntax, such as using a wrong punctuation mark or a missing semicolon.
   - Logical errors: These errors occur when the programmer's code does not work as intended. For example, a program may produce incorrect output because of a flawed algorithm.
2. Design flaws: These types of bugs arise from problems in the design of the software. Examples include:
   - Inadequate user requirements: If the software is designed without considering the needs of the end-users, it may not meet their needs, resulting in a bug.
   - Architecture flaws: If the software's architecture is not properly designed, it can lead to bugs in the software. For example, if the software is designed to be monolithic instead of modular, it can make it difficult to isolate and fix bugs.
3. Environmental issues: These types of bugs arise from problems in the software's operating environment. Examples include:
   - Incompatibility issues: If the software is not compatible with the user's operating system or other software components, it can lead to bugs.
   - Hardware failures: If the software relies on hardware components that are not functioning properly, it can lead to bugs.

It's important for software developers to be aware of these sources and take steps to prevent them during the software development process. This can include using best practices for coding, conducting thorough testing, and regularly updating documentation.

Debugging is a standard part of any SoC design process. The concept of debugging hardware (cycle-level) and software (instruction-level) together is not a new one. Co-Debugging means debugging software code and custom hardware simultaneously in an early design stage [4]. Software debugging is performed at the source/instruction-level

while the hardware is debugged at the cycle-level. Tests are Written in Native Softwar such as C or assembly then Software Program is Loaded into Memory. Processor Executes Program. Moreover, RTL Testbenches may also be Executed.

A **tracer**, on the other hand, is a tool used to collect runtime information about the execution of a program. It provides insights into the sequence and timing of various program events, such as function calls, system calls, memory accesses, and other program-specific events.

Example of SW debuggers are:

- Lauterbach Trace32 Debugger [7].
- Codelink [8].
- GDB: a GNU debugger [5].
- MIPS SP55 Debugger [9].
- Arm DS5 Debugger [6].
- OpenOCD Debugger.

when using a software debugger, you typically need to define the interface and target in order to establish a connection with the target device and perform debugging operations.

- **Interface**: The interface refers to the hardware or protocol used to communicate between the debugger and the target device. Examples of interfaces commonly used in embedded systems are JTAG (Joint Test Action Group) and SWD (Serial Wire Debug). The debugger needs to be configured with the appropriate interface settings to establish a connection.
- **Target**: The target represents the specific hardware device or microcontroller you are debugging. Each target device has its own configuration requirements, memory map, and debugging features. The target configuration specifies the details of the target device, such as its architecture, memory layout, and debug support. This information is necessary for the debugger to interact with the target correctly.

Using these debuggers, the developer can read and write variables. Moreover, he can navigate through statements and procedures in the source code using breakpoint, stepping, etc. Most of the existing debugging methods perform software debugging and hardware debugging separately. This causes the information of software debugging and hardware debugging be independent from each other. With the help of conventional software debugger such as GNU Debugger, a programmer could navigate between the source codes and examine the correctness of the software behavior by breakpoint setting, single stepping, and rea/write variables. In contrast, hardware designers often detect the bugs by examining the bit-information or the waveform dumped from debug supporting hardware such as tracer [13].

These SW debuggers can be used too in processor booting process as they can be used to download bootloader and **FW** (with Linux Kernel) onto **a flash**. Then, as the debugger is hooked to the DUT, it can be used to debug it. *HW*/*SW* debuggers can also be used within a *simulation* environment.

5.3.1 GDB: A GNU Debugger

GDB is a standard debugger in GNU project, which supports debugging for many programming languages such as C, C++, Java, etc. It is a debugger which is familiar to almost every software designer.

GDB could fetch the information and help the software developers locate the causes of a malfunction in their software. GDB also provides Remote Serial Protocol (RSP) to let GDB connect to any target remotely if the target has implemented the server side of the RSP. This is very valuable when debugging in embedded environment, where it is not possible to run GDB natively on the target.

Here's an example of how GDB can be used to debug a C program:

1. First, the program must be compiled with debugging symbols. This can be done by adding the "-g" flag to the compiler command:
   gcc -g -o my_program my_program.c
2. Once the program is compiled with debugging symbols, it can be loaded into GDB by typing the following command in the terminal:
   gdb my_program.
3. After the program is loaded into GDB, the developer can set breakpoints in the code at specific locations where they suspect a bug might be occurring. This is done using the "break" command, followed by the line number or function name:
   break 15 // Set breakpoint at line 15
   break my_function // Set breakpoint at function my_function
4. The developer can then run the program in GDB using the "run" command. When the program reaches a breakpoint, it will stop executing, and GDB will display information about the program's state at that point in time, including variable values and the call stack.
5. From there, the developer can use GDB's commands to inspect the program's state, step through the code, and diagnose the bug. For example, the "next" command can be used to execute the next line of code, while the "print" command can be used to print the value of a variable:
   next // Execute the next line of code
   print my_variable // Print the value of the variable my_variable

6. Once the bug has been diagnosed, the developer can modify the code to fix the issue and recompile the program with debugging symbols to verify that the bug has been resolved.

GDB also provides several commands for examining the stack trace and memory of a program, which can be helpful for identifying issues such as memory leaks or segmentation faults.

Memory leakage and segmentation faults are both issues that can occur in computer programs and can cause them to crash or behave unpredictably. Here are some common reasons for each:

Memory Leakage:

- Not freeing dynamically allocated memory when it is no longer needed.
- Using a recursive function that doesn't have a base case, causing it to keep allocating memory until none is left.
- Creating circular data structures (e.g., linked lists) where objects refer to each other in a way that prevents them from being deallocated.
- Forgetting to close open file handles or sockets, which can use up system resources.

Segmentation Fault:

- Dereferencing a null pointer or uninitialized pointer.
- Attempting to access an array or other memory location outside of its bounds.
- Passing an incorrect type or number of arguments to a function
- Incorrect use of pointers, such as using a pointer after it has been freed.
- Stack overflow due to an excessive amount of recursion or a large amount of local variables being declared.

In addition to its command-line interface, GDB also provides a graphical user interface (GUI) called the GNU Data Display Debugger (**DDD**). This tool provides a visual representation of the program's state and allows developers to set breakpoints and interact with the program in a more intuitive way.

5.3.2 Arm DS5 Debugger

Arm® Debugger is part of Arm Development Studio and helps you find the cause of software bugs on Arm processor-based targets and Fixed Virtual Platform (FVP) targets [14]. It provides a comprehensive suite of features and capabilities for debugging and analyzing code, including source-level debugging, instruction-level debugging, and real-time tracing. Here are some of the key features and capabilities of Arm® Debugger:

1. Source-level debugging: Arm® Debugger supports source-level debugging, allowing developers to set breakpoints, step through code, and examine variables and memory locations in their source code.
2. Instruction-level debugging: Arm® Debugger supports instruction-level debugging, allowing developers to debug and analyze the assembly code generated by their compiler.
3. Real-time tracing: Arm® Debugger provides real-time tracing capabilities, allowing developers to trace the execution of their software in real-time and analyze the performance of their software.

5.3.3 MIPS SP55 Debugger

MIPS SysProbe SP55E, the newest probe technology from MIPS, supports all MIPS cores including the latest Warrior class. By taking advantage of advanced on-chip debug and optional trace features in MIPS IP cores, the SysProbe SP55E allows fast and efficient debug without requiring an intrusive software monitor or additional target I/O resources [9]. Here are some of the key features and capabilities of MIPS SP55 Debugger:

1. Multi-core debugging: MIPS SP55 Debugger supports multi-core debugging, allowing developers to debug and analyze software running on multi-core MIPS-based systems.
2. Integration with other MIPS development tools: MIPS SP55 Debugger integrates with other MIPS development tools, such as the MIPS32® Software Development Kit and the MIPS Navigator Integrated Development Environment, providing a comprehensive development and debugging environment for MIPS-based systems.
3. Customizable user interface: MIPS SP55 Debugger provides a customizable user interface, allowing developers to customize the layout and behavior of the debugger to suit their needs.

5.3.4 Lauterbach Trace32 Debugger

The TRACE32 debugger allows you to test your embedded hardware and software by using the on-chip debug interface. The most common on-chip debug interface is **JTAG**. A single on-chip debug interface can be used to debug all cores of a multi-core chip [7].

JTAG (Joint Test Action Group) is a standard for debugging and testing integrated circuits, especially digital systems. It was first introduced in the 1980s and has become a standard feature on many digital devices. JTAG is based on a chain of logic signals that connects all the integrated circuits on a board together. This chain is called a “boundary scan chain” and allows signals to be sent in and out of each device in turn. The JTAG interface typically includes four signals: TCK (Test Clock), TMS (Test Mode Select), TDI

(Test Data In), and TDO (Test Data Out). TCK is used to synchronize the data transfer, TMS is used to switch between test modes, TDI is used to input test data, and TDO is used to output test results. One of the key features of JTAG is its ability to access the internal registers of digital devices, even if they are not physically accessible. This allows developers to test and debug their designs at a much more detailed level than would otherwise be possible. JTAG is commonly used in a number of different applications, including:

- Boundary scan testing: This involves testing the connections between different integrated circuits on a board to ensure they are working correctly.
- Flash programming: This involves programming the internal memory of a device using JTAG, allowing for firmware updates or other changes.
- Debugging: JTAG is often used for debugging digital systems, allowing developers to step through code and examine the state of internal registers in real-time.

TRACE32 debugger supports for various architectures: Lauterbach Trace32 Debugger supports a wide range of architectures, including ARM, PowerPC, MIPS, $\times$ 86, and many others. Trace32 Debugger is a powerful and comprehensive tool for debugging and analyzing software running on various microcontrollers, microprocessors, and SoCs. Its support for source-level and instruction-level debugging, real-time tracing, and multi-core debugging make it an essential tool for embedded system development.

5.3.5 OpenOCD Debugger

OpenOCD (Open On-Chip Debugger) is an open-source software tool that provides debugging, in-system programming, and boundary scan testing capabilities for embedded systems. It supports a wide range of target devices, including microcontrollers and processors from various manufacturers. Here are some key features and capabilities of OpenOCD [21]:

- Debugging: OpenOCD allows you to debug target devices using JTAG, SWD, or other compatible interfaces. It provides features such as halting and resuming execution, single-stepping through code, setting breakpoints, and inspecting and modifying register values and memory contents. **Halting** refers to stopping the execution of a specific hardware thread or processor core. When a hardware thread is halted, it means that its execution is paused, and the thread is effectively stopped at its current program counter (PC) value.
- Flash Programming: OpenOCD enables in-system programming of flash memory on target devices. It supports different flash programming algorithms and provides commands to erase, program, and verify flash memory.

- Boundary Scan Testing: OpenOCD supports boundary scan testing, which is a technique used for testing and diagnosing faults on integrated circuits. It allows you to perform boundary scan operations, such as shifting data into and out of boundary scan cells, and performing boundary scan tests on devices that support this feature.
- Multi-core and Multi-thread Debugging: OpenOCD supports debugging and control of multi-core or multi-threaded systems. It provides mechanisms to select and halt individual hardware threads (harts) in systems based on architectures like RISC-V.
- Scripting and Automation: OpenOCD offers a command-line interface (CLI) that allows you to interact with the tool and perform operations using commands. It also supports scripting using Tcl (Tool Command Language), allowing you to automate complex debugging or programming tasks.
- Platform and Device Support: OpenOCD supports a wide range of target devices, including microcontrollers, application processors, and system-on-chip (SoC) devices from various manufacturers. It provides configuration files and support for specific target architectures, interfaces, and debug adapters.

5.3.6 Codelink

The Codelink product is an integrated, source-level debug environment targeting processor driven tests. Codelink Software Debug Environment automates debugging for embedded software and correlates embedded software and hardware debug of complex SoC's. During debug, Visualizer with Codelink allows embedded software to be debugged using software debug techniques, including source code debug, assembly code debug, all while monitoring the CPU registers and memory, and correlating these with hardware debug. Hardware and Software debug in one integrated tool helps find bugs faster [8]. Codelink gives the software developer a very productive environment where they can debug efficiently off-line. It correlates software to hardware on waveforms, code, and CPU register contents, so you can correlate what software part is responsible for what hardware part in case of debugging an issue. CodeLink works by connecting the microcontroller to a host computer via a debugging interface, such as JTAG or Serial Wire Debug (SWD). This allows developers to interact with the microcontroller and debug their code in real-time.

5.4 HW/SW Co-Verification Metrics

HW/SW co-verification is the process of verifying that the hardware and software components of a system are designed to work together as intended. Metrics are used in this process to measure and analyze the performance of the co-verification process. The HW/SW Co-Verification Metrics can be summarized as:

- **Performance** (speed) [15]: This metric measures the timing performance of the system, including factors such as latency and throughput. Timing analysis can help identify bottlenecks in the system and ensure that the hardware and software components are synchronized and working together as intended.
- **Accuracy**: This metric measures the accuracy of the co-verification process, including factors such as simulation accuracy and emulation accuracy. Ensuring accuracy is important to ensure that the system functions as intended and to avoid issues that could arise later in the development process.
- **Synchronization**: synchronization overhead is an important metric.
- **Time to integrate models**: This metric measures the time required to debug and fix issues that arise during the co-verification process. Reducing debug time can help speed up the overall development process and reduce costs.
- **Resource utilization**: This metric measures the utilization of system resources, including factors such as memory and CPU utilization. Optimizing resource utilization can help improve system performance and reduce costs.
- **Ability to do hardware debugging (visibility).**
- **Ability to do performance analysis**.

These metrics provide insight into the performance of the co-verification process and help ensure that the hardware and software components of a system are designed to work together as intended. By measuring and analyzing these metrics, developers can identify areas where improvements can be made and optimize the co-verification process to improve overall system performance and reduce development costs.

5.5 Conclusions

This chapter discusses the importance of Co-Verification and Co-Debugging in the design process of modern System-on-Chips (SoCs). With the growing complexity of SoCs, debug, verification, and validation have become critical design concerns. Traditionally, software developers wait for a hardware prototype for the final system integration. However, this approach can be costly and time-consuming if problems arise during integration. Therefore, moving the system integration phase forward in the design cycle by creating a Hardware/Software (HW/SW) co-verification can help detect these integration problems earlier and minimize product time-to-market. Co-Verification involves simulating the behavior of hardware and software components in a virtual environment. The software is tested using a software simulator, while the hardware is tested using a hardware simulator. The test cases are designed to exercise the interactions between the hardware and software components of the system. Co-Debugging is the process of debugging a system that contains both hardware and software components. This can be challenging

because it requires knowledge of both hardware and software. In co-debugging, hardware and software engineers work together to debug issues in the system. The software engineer may use software debugging tools such as a debugger to identify and fix issues in the software. The chapter also discusses Co-Simulation and Co-Emulation techniques used in Co-Verification. Co-Simulation involves simulating both hardware and software components together using simulation models. On the other hand, Co-Emulation involves running actual hardware with simulated or emulated interfaces for other components.

References

1. N. K. Doshi, S. Suryawanshi, and G. N. Kumar, Development of generic verification environment based on uvm with case study on hmc controller," in 2016 IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT), pp. 550–553, May 2016.
2. S. Jain, P. Govani, K. B. Poddar, A. K. Lal, and R. M. Parmar, \Functional verification of dsp based on-board vlsi designs," in 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), pp. 1–4, Jan 2016.
3. https://fmi-standard.org/
4. L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino, “Systemc co-simulation and emulation of multiprocessor soc designs,” Computer, vol. 36, no. 4, pp. 53–59,2003.
5. GDB: The GNU Project Debugger. Available: https://www.gnu.org/software/gdb/
6. DS-5 Development Studio. Available: https://developer.arm.com/products/softwaredevelopment-tools/ds-5-development-studio.
7. https://www.lauterbach.com
8. https://webinars.sw.siemens.com/embedded-software-debug-using
9. https://www.mips.com/develop/tools/mips-debug-and-trace-probes/sp55e/
10. W. Müller, W. Rosenstiel, and J. Ruf, SystemC: Methodologies and Applications. Springer, 2003.
11. F. Ghenassia et al., Transaction-Level Modeling with SystemC. Springer, 2005.
12. Hong, Sungpack, Oguntebi, Tayo, Casper, Jared, Bronson, Nathan, Kozyrakis, Christos and Olukotun, Kunle (2012) "A Case of System-level Hardware/Software Co-design and Co-verification of a Commodity Multi-processor System with Custom Hardware" Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, ACM, 513–520.
13. K. Goossens, B. Vermeulen, R. Van Steeden, and M. Bennebroek, “Transaction-based communication-centric debug,” in 2007. NOCS 2007. First International Symposium on Networks-on-Chip, 2007, pp. 95–106.
14. https://developer.arm.com/documentation/101470/2022-0/?lang=en.
15. https://www.embedded.com/hw-sw-co-verification-basics-part-4-co-verification-metrics/
16. https://semiengineering.com/hw-sw-co-verification-for-hybrid-systems/
17. M. M. Ashtaputre, M. S. Sutaone and M. Rajne, “A Low Cost Solution for Hardware-in-Loop Software Testing,” *2021 International Symposium of Asian Control Association on Intelligent Robotics and Industrial Automation (IRIA)*, Goa, India, 2021, pp. 449–452, doi: https://doi.org/10.1109/IRIA53009.2021.9588753
18. A. Taksale, V. Vaidya, P. Shahane, G. Dronamraju and V. Deulkar, “Low cost hardware-in-loop for automotive application,” 2015 International Conference on Industrial Instrumentation and

Control (ICIC), Pune, India, 2015, pp. 1109–1114, doi: https://doi.org/10.1109/IIC.2015.7150913.

19. Sooyong Jeong, Yongsub Kwak and Woo Jin Lee, "Software-in-the-Loop simulation for early-stage testing of AUTOSAR software component," 2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN), Vienna, Austria, 2016, pp. 59–63, doi: https://doi.org/10.1109/ICUFN.2016.7536980.
20. Salah, Khaled. "An online RTL-level scan-chain-based methodology for accelerating IP emulation debugging at run-time." 2013 4th Annual International Conference on Energy Aware Computing Systems and Applications (ICEAC). IEEE, 2013.
21. https://openocd.org/

HW/SW Co-Optimization and Co-Protection

6

6.1 HW/SW Co-Optimization

Optimization is the process of finding the best input variable values from among all possibilities without explicitly evaluating each possibility. Optimization technology uses various techniques to identify the input variables to specify the optimum solution.

Optimization techniques can be categorized as follows:

- **Trial and error optimization**: it involves iteratively refining the design by making changes and evaluating the results. Designers adjust based on their experience, intuition, and understanding of the design requirements. They may modify the code, tweak the design parameters, or adjust the architecture to improve performance, area utilization, or power consumption. This approach relies on the designer's expertise and involves multiple iterations of compilation, synthesis, and implementation to achieve the desired results. While trial and error optimization can be effective, it can also be time-consuming and may not always lead to optimal results. It requires a deep understanding of the design and its requirements.
- **Simulation-Driven optimizations**: reduce the trial-and-error design work [5]. Instead of relying solely on manual adjustments, designers use simulation results to identify bottlenecks, analyze timing issues, and evaluate design trade-offs. Simulation allows designers to verify the correctness and functionality of the design before physically implementing it on an FPGA. By simulating the design with different input scenarios, designers can assess performance, identify potential issues, and explore alternative design choices. Simulation-driven optimizations can include techniques such as pipelining, parallelization, resource sharing, and algorithmic optimizations. Through simulation-driven optimizations, designers gain insights into the behavior of

K. S. Mohamed, *Heterogeneous SoC Design and Verification*, Synthesis Lectures on Digital Circuits & Systems, https://doi.org/10.1007/978-3-031-56152-8_6

the design and can make informed decisions to improve performance, reduce area utilization, or minimize power consumption. This approach helps in validating the design against performance targets, identifying critical paths, and adjusting achieve the desired outcomes.

Hardware/software (HW/SW) co-optimization is the process of optimizing the design of an electronic system by simultaneously considering both hardware and software components. This approach involves creating a tight integration between the hardware and software components, such that they can be optimized together to achieve the desired performance, power consumption, and other design goals. In the traditional approach to system design, the hardware and software components are designed separately, and then integrated later in the design process. However, this approach can result in suboptimal performance and higher costs due to the lack of synergy between the hardware and software components. HW/SW co-optimization involves co-designing the hardware and software components from the beginning of the design process. This approach enables designers to take advantage of the strengths of both hardware and software, resulting in a more efficient and effective design. For example, hardware components can be designed to perform specific tasks that are difficult or inefficient to implement in software, while software components can be designed to take advantage of the flexibility and programmability of software. HW/SW co-optimization is particularly important in the design of complex electronic systems, such as system-on-chip (SoC) designs, where the integration of hardware and software is critical to achieving optimal performance, power consumption, and other design goals. By co-designing the hardware and software components of an SoC, designers can achieve a more efficient and effective design, resulting in better performance, lower power consumption, and lower costs. There are several methods for HW/SW co-optimization [30]:

1. Co-design: Co-design is a method that involves simultaneously designing the hardware and software components of a system. This approach allows for closer collaboration between hardware and software designers, leading to more efficient system designs.
2. Model-based design: Model-based design involves creating a system model that incorporates both the hardware and software components. This model can be used to simulate and optimize the system's performance before it is built, reducing the time and cost of physical prototyping.
3. Profiling and optimization: Profiling involves analyzing the performance of a system's hardware and software components to identify bottlenecks and areas for optimization. Optimization techniques such as loop unrolling, cache optimization, and pipelining can then be applied to improve performance.
4. Partitioning: Partitioning involves dividing the functionality of a system between hardware and software components in a way that optimizes performance and power consumption. The partitioning process can be done manually or using automated tools.

5. Co-simulation: Co-simulation involves simulating the hardware and software components of a system together to evaluate their performance and identify potential issues. This approach can help designers identify and resolve design flaws before the system is built.
6. Hardware acceleration: Hardware acceleration involves offloading compute-intensive tasks from the software to dedicated hardware components. This can significantly improve performance and reduce power consumption.

6.1.1 Software Optimization

Software optimization refers to the process of improving the performance and efficiency of software programs by reducing their resource utilization, such as CPU time, memory usage, and I/O operations. Here are some examples of software optimization techniques (Table 6.1):

1. Code optimization: Code optimization involves analyzing the source code of a program and making modifications to improve its efficiency. This can include techniques such as loop unrolling, function pipelining, and reducing the number of memory accesses, replacing non-optimal data-types with optimal one: using non-optimal data-types can increases the number of required memory accesses and processor instructions, removing redundant arrays, not using long loops,
2. Compiler optimization: Compiler optimization involves using advanced algorithms and techniques to automatically optimize the code generated by a compiler. This can include techniques such as instruction scheduling, register allocation, and code reordering.
3. Parallelization: Parallelization involves dividing a program into smaller tasks that can be executed simultaneously on multiple processing units. This can include techniques such as multi-threading, SIMD (Single Instruction, Multiple Data), and GPU acceleration.
4. Memory management: Memory management involves optimizing the use of memory in a program to reduce the number of memory accesses and improve cache locality. This can include techniques such as memory pooling, pre-fetching, and reducing memory fragmentation.
5. I/O optimization: I/O optimization involves reducing the time and resources required for input/output operations. This can include techniques such as buffering, compression, and caching.
6. Algorithm optimization: Algorithm optimization involves selecting or modifying algorithms to improve the efficiency of a program. This can include techniques such as reducing the time complexity of algorithms, using more efficient data structures, and avoiding unnecessary calculations.

Table 6.1 Comparison between different software optimization techniques

Technique	Description	Advantages	Disadvantages
Code optimization	Analyzing source code and making modifications to improve efficiency	Improves execution speed and reduces resource usage	Can be time-consuming and difficult to implement
Compiler optimization	Using advanced algorithms and techniques to optimize compiled code	Can improve execution speed and reduce resource usage	Can sometimes result in incorrect or inefficient code
Parallelization	Dividing a program into smaller tasks executed simultaneously on multiple processing units	Can greatly improve execution speed	Can be difficult to implement and requires careful design to avoid race conditions and deadlocks
Memory management	Optimizing memory usage in a program to reduce memory accesses and improve cache locality	Can significantly reduce resource usage	Can be time-consuming and requires careful design to avoid memory leaks and segmentation faults
I/O optimization	Reducing the time and resources required for input/output operations	Can improve overall system performance	Can sometimes require more complex code
Algorithm optimization	Selecting or modifying algorithms to improve efficiency	Can greatly improve execution speed	Can be difficult to implement and may require significant changes to program logic
Profile-guided optimization	Analyzing runtime behavior to optimize performance	Can improve overall system performance	Requires careful collection and analysis of profiling data, which can be time-consuming

7. Profile-guided optimization: Profile-guided optimization involves analyzing the runtime behavior of a program and using this information to optimize its performance. This can include techniques such as using runtime profiling data to guide code optimizations.

Software optimization is essential for improving the performance and efficiency of software programs, particularly for resource-constrained systems such as embedded systems, mobile devices, and servers.

6.1.2 Hardware Optimization

Hardware optimization is the process of improving the performance, efficiency, and reliability of hardware components or systems. The goal is to maximize the functionality of the hardware while minimizing the resources it requires. This is achieved through a variety of techniques such as reducing the size of the hardware, improving power efficiency, increasing clock speed, optimizing circuitry, and more. Hardware optimization is important because it allows designers to create more powerful and efficient hardware solutions that can perform complex tasks in real-time. It is often used in the design of computer systems, microcontrollers, and other embedded systems that require high performance and low power consumption. Hardware optimization can be achieved using various methods, such as [6, 7]:

1. Circuit optimization: This technique involves optimizing the circuits within the hardware to improve their performance, reduce their size, and lower their power consumption. This can be achieved by using smaller transistors, optimizing the layout of the circuit, or using specialized circuit topologies.
2. Timing analysis: Timing analysis involves analyzing the timing behavior of the hardware to identify bottlenecks and optimize the timing of signals between different components. This can be achieved by adjusting clock frequencies or using specialized timing analysis tools.
3. Power optimization: Power optimization involves optimizing the power consumption of the hardware to minimize energy usage while maintaining performance. This can be achieved by using power-saving techniques such as clock gating, power gating, or voltage scaling.
4. Synthesis optimization: Synthesis optimization involves optimizing the hardware design by automatically generating the most efficient implementation from a high-level description. This can be achieved using synthesis tools that optimize the design based on a set of constraints such as area, power, or performance.
5. Design for testability: Design for testability involves designing hardware with built-in self-test features that allow for more efficient testing and debugging. This can be achieved by using specialized test patterns or built-in self-test circuits.
6. RTL coding optimization: RTL coding optimization involves optimizing the Register Transfer Level (RTL) code used to describe the hardware. This can be achieved by using high-level synthesis tools, optimizing the RTL code by hand, or using specialized coding techniques such as pipelining or parallelization.
7. Memory optimization: Memory optimization involves optimizing the use of memory within the hardware to improve performance and reduce power consumption. This can be achieved by using specialized memory architectures or optimizing the use of cache memory.

Hardware optimization techniques are often used in combination to achieve the desired performance and efficiency goals. The optimization process typically involves multiple iterations of simulation, testing, and refinement. It requires a deep understanding of hardware design principles, as well as the ability to work with a wide range of design tools and technologies.

Hardware optimization is often a complex and iterative process that involves multiple rounds of simulation, testing, and refinement. It requires a deep understanding of hardware design principles, as well as the ability to work with a wide range of design tools and technologies. The optimization objective is to reduce area, delay, latency, and power and to increase performance and speed to meet the requirement.

6.1.3 HW/SW Compilation Time Optimization

Compilation Time Optimization (CTO) is the process of optimizing the time it takes to compile software code. It is an important aspect of software development because the faster the code can be compiled, the faster developers can test and iterate on their code, resulting in a shorter development cycle. CTO techniques can be broadly classified into two categories: static and dynamic. Static techniques focus on optimizing the code before it is compiled, while dynamic techniques optimize the code as it is being compiled.

Static CTO techniques include [8–14]:

1. Code refactoring: This involves restructuring the code to improve its efficiency and performance.
2. Code analysis: This involves analyzing the code to identify potential performance bottlenecks and then making changes to address them.
3. Compiler flags: These are options that can be set when compiling the code to improve performance, such as enabling optimization features and disabling debugging symbols.
4. Parallel compilation: This involves splitting the compilation process across multiple processors to reduce the overall compile time.

Dynamic CTO techniques include:

1. Just-In-Time (JIT) compilation: This is a technique used in some programming languages (such as Java and.NET) where the code is compiled at runtime, rather than ahead of time. JIT compilers can optimize the code as it is being compiled, resulting in faster code execution.
2. Profile-guided optimization (PGO): This involves compiling the code multiple times while collecting performance data to optimize the code based on how it is actually used.

3. Incremental compilation: This involves compiling only the parts of the code that have changed since the last compilation, rather than compiling the entire codebase every time.

Overall, CTO techniques are important for improving the productivity of software development teams by reducing the time it takes to compile code and allowing developers to iterate on their code more quickly.

Compilation Time Optimization (CTO) is also important in the context of hardware compilers, where the goal is to optimize the time, it takes to compile hardware description languages (HDLs) into hardware designs. Hardware designs can be complex, with many thousands or even millions of gates and components, and the process of compiling them can be time-consuming. CTO techniques can help reduce this compilation time, allowing designers to iterate on their designs more quickly and efficiently. Some common CTO techniques used in hardware compilers include:

1. Algorithmic optimizations: This involves optimizing the algorithms used by the compiler to generate the hardware design, such as optimizing the logic synthesis and placement and routing process.
2. Parallelization: This involves splitting the compilation process across multiple processors or cores to reduce the overall compile time.
3. Resource sharing: This involves sharing resources (such as memory or computational resources) across different compilation processes to reduce the overall resource usage and speed up compilation.
4. Incremental compilation: This involves compiling only the parts of the design that have changed since the last compilation, rather than compiling the entire design every time.
5. High-level synthesis: This involves using high-level language constructs (such as loops and arrays) to describe the design, which can then be automatically translated into hardware.
6. Hardware/software co-design: This involves designing the hardware and software components of a system together, optimizing them together to achieve better overall performance and reduced compilation time.
7. Best practice design methodology:
   - Do not use long loops.
   - Store large data in memory not in a register.
   - Reduce the use of power"**" and the division "\", instead use log and shift right.
   - Do not write long ternary statement "()?: () ?: () ? ….".
   - Use 2D memory instead of 1D memory as 2-D memory reduce the compile as it is mapped directly to the memory blocks not to the logic.
   - Split logic circuits to shorten the critical path.
   - Choose faster logic circuit architectures.

6.1.4 HW Maximum Frequency Optimization

Maximum frequency optimization is a technique used in digital hardware design to achieve the highest possible clock frequency for a given design. This optimization technique is used to improve the performance of digital systems by maximizing the operating frequency of the system. The maximum frequency of a digital design is determined by the critical path, which is the path in the design that takes the longest time to complete. The critical path is made up of a series of logic gates and interconnects that limit the clock frequency of the design. Therefore, to optimize the maximum frequency, it is essential to optimize the critical path. There are several techniques used to optimize the maximum frequency of a digital design. These include:

1. Logic restructuring: This technique involves restructuring the design's logic gates to minimize the number of levels of logic and reduce the critical path. By reducing the critical path, the maximum frequency of the design can be increased.
2. Pipelining: This technique involves breaking up the critical path into several smaller stages, which can be pipelined to reduce the delay on the critical path. Pipelining helps in reducing the critical path delay and, hence increasing the maximum frequency.
3. Register retiming: This technique involves moving registers in the design to balance the delay between the registers and the logic gates, reducing the critical path. This technique helps in reducing the critical path delay and, hence increasing the maximum frequency.
4. Clock gating: This technique involves selectively gating the clock signal to portions of the design that are not needed at a particular time to reduce power consumption and improve the maximum frequency. Clock gating helps in reducing the power consumption, which can lead to an increase in the maximum frequency.
5. Physical layout optimization: This technique involves optimizing the physical layout of the design, including the placement of logic gates and interconnects, to reduce the critical path. By optimizing the physical layout, the critical path delay can be reduced, and the maximum frequency can be increased.
6. Timing analysis: This technique involves analyzing the timing characteristics of the design to identify the critical path and areas that need optimization. By analyzing the timing characteristics of the design, the critical path can be identified, and the optimization techniques can be applied to improve the maximum frequency.
7. Power analysis: This technique involves analyzing the power consumption of the design to identify areas that can be optimized for power consumption. By optimizing the power consumption, the maximum frequency can be increased.

By using these techniques, designers can optimize the maximum frequency of a digital design and achieve better performance. However, it is important to note that these techniques may also result in an increase in power consumption or area usage. Therefore,

designers must strike a balance between optimizing the maximum frequency and other design considerations such as power consumption, area, and reliability.

6.1.5 HW/SW Power Optimization

Hardware/software power optimization is the process of reducing the power consumption of a system while maintaining or improving its performance. This is particularly important for battery-powered devices and embedded systems, where power consumption is a critical factor in determining the device's functionality and battery life.

There are several techniques used in hardware/software power optimization, including [15–21]:

1. Dynamic Voltage and Frequency Scaling (DVFS): This technique involves dynamically adjusting the voltage and frequency of the processor to match the workload. This can significantly reduce power consumption when the workload is low.
2. Power Gating: This technique involves turning off power to specific blocks of the system when they are not in use. This can significantly reduce power consumption when these blocks are idle.
3. Clock Gating: This technique involves turning off the clock signal to specific parts of the system when they are not in use. This can save power by reducing the number of clock cycles required to execute instructions (Fig. 6.1).
4. Pipeline Optimization: This technique involves optimizing the pipeline design to reduce power consumption. This can be done by reducing the number of pipeline stages or by adding pipeline registers to reduce the power consumed by each stage.

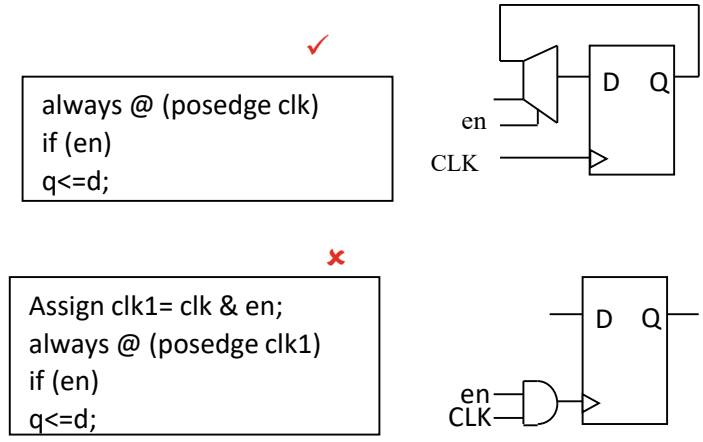


Fig. 6.1 Clock gating

5. Compiler Optimization: This technique involves optimizing the code generated by the compiler to reduce power consumption. This can be done by using techniques such as loop unrolling, instruction scheduling, and register allocation to reduce the number of instructions executed and the number of memory accesses.
6. Low Power Modes: This technique involves putting the system into low power modes when it is not in use. This can be done by reducing the voltage and frequency of the processor or by turning off power to specific parts of the system.
7. Energy-Aware Scheduling: This technique involves scheduling tasks based on their energy requirements. This can be done by selecting tasks that have lower energy requirements first or by scheduling tasks that have higher energy requirements when the system is in a low power mode.
8. Use gray-coding FSM: Encoding of FSMs including different encoding styles, the most famous one is binary encoding. There is also gray encoding and one-Hot encoding. Binary-encoding implements very less logic. Also, it used minimum number of FFs. Possible state values for a 4-state binary state machine (00, 01, 10, 11). Gray encoding especially useful when the outputs of the state bits are used asynchronously. This kind of state coding avoids intermediate logics. For example, if a state wants to change its state from "01" to "10". In Gray coding between state transitions only one bit will change. Possible state values for a 4-state gray state machine (00, 01, 11, 10). One-hot encoding uses one flip-flop for each state. For example, if there are 10 states in logic then it will use 10 flip-flops. This type of encoding is fast because only one bit needed to check for each state. It implies complex logic and more area inside the chip due to a greater number of flip-flops. FPGAs are "Flip-flop rich", therefore one-hot state machine encoding is often a good approach. It also reduces hardware's logic switching rate. Possible state values for a 4 state one-hot state machine (0001, 0010, 0100, 1000) [3].
9. Use line coding: to reduce transitions (8b/10b encoder): reduce α (switching activity factor).
10. Increase data bus width: to reduce transfer cycles: reduce α.

These techniques can be used in combination to achieve the desired power savings. The choice of technique depends on the system architecture, the workload, and the power requirements.

6.1.6 HW/SW Speed Optimization

HW/SW speed optimization refers to the process of improving the performance of a system by optimizing both the hardware and software components together. This approach aims to achieve the best possible speed and efficiency by taking advantage of the strengths of both hardware and software.

Hardware speed optimization techniques include:

1. Pipeline optimization: This involves dividing the computation into several stages, where each stage performs a specific task. This can help to improve the overall speed of the system.
2. Memory hierarchy optimization: This involves organizing the memory subsystem to reduce the time required to access data. This can include techniques such as cache optimization and prefetching.
3. Clock frequency optimization: This involves increasing the clock frequency of the system to improve its performance. However, this can also increase power consumption and heat generation.
4. Keep critical path logic in a separate module: optimize the critical path logic for speed and optimize the noncritical path logic for area.

Software speed optimization techniques include:

1. Loop unrolling: This involves duplicating the loop body to reduce the number of iterations required, which can help to improve the speed of the program.
2. Code optimization: This involves modifying the code to reduce the number of instructions executed or the number of memory accesses required. This can include techniques such as instruction scheduling and register allocation.
3. Parallelization: This involves breaking up the computation into smaller pieces that can be executed simultaneously on different cores or processors. This can help to improve the speed of the program.

By combining these hardware and software optimization techniques, it is possible to achieve the best possible performance from a system while minimizing power consumption and other resources.

6.1.7 HW Area Optimization

Area optimization is a process of minimizing the amount of silicon area required to implement a digital circuit or system. It involves reducing the number of transistors or gates or re-organizing them in a more compact and efficient manner. Area optimization is an important consideration in integrated circuit design, as it directly impacts the cost, power consumption, and performance of the chip. There are several techniques that can be used to optimize the area of a digital circuit or system, including [22, 23]:

1. Gate-level optimization: This technique involves optimizing the logic gates used in the design to reduce their area. One example of gate-level optimization is gate merging,

where multiple gates are merged into a single gate to reduce the number of transistors and save area. Another example is gate sharing, where multiple gates share a common input or output, resulting in a reduction in the number of transistors and area.
2. Circuit-level optimization: This technique involves optimizing the circuit topology to reduce its area. One example of circuit-level optimization is folding, where a circuit is folded to reduce its length and save area. Another example is retiming, where the clock period is adjusted to optimize the timing and reduce the number of registers, resulting in an area reduction.
3. Technology mapping: This technique involves mapping the logic gates in the design to a specific technology library, with the goal of minimizing area. Technology mapping tools use a library of pre-characterized cells that have been optimized for area and timing and map the logic gates in the design to the most area-efficient cells in the library.
4. High-level synthesis: This technique involves automatically generating a hardware implementation from a high-level description of the functionality. High-level synthesis tools can optimize the hardware architecture to minimize area, by exploring different architectural choices and selecting the most area-efficient implementation.
5. Design reuse: This technique involves reusing pre-designed circuits or IP blocks that have already been optimized for area. By using pre-optimized circuits or IP blocks, designers can avoid the need for extensive optimization and reduce the time-to-market for their designs.
6. Dynamic Partial Reconfiguration (DPR) is also used to optimize area usage. With DPR, it is possible to implement different circuits that are not needed at the same time, and that do not operate simultaneously, on the same FPGA area, resulting in considerable area savings [1, 2].

Area optimization is a complex and multi-dimensional problem, as it requires balancing competing design constraints such as performance, power consumption, and cost. A designer must carefully evaluate different area optimization techniques and trade-offs to arrive at an optimal solution for a given design.

6.2 HW/SW Co-Protection

HW/SW co-protection refers to the process of designing a system-on-chip (SoC) in such a way that the hardware (HW) and software (SW) components of the system work together to provide a high level of security against various types of attacks. In this approach, the hardware and software components of the system are designed to work together in a mutually supportive way, each reinforcing the other's defenses against potential attacks. The hardware components of the system are designed to detect and prevent unauthorized access to the system, while the software components are designed to provide additional

layers of protection against attacks that may bypass the hardware defenses. Examples of hardware-based protections include secure boot processes, encryption and decryption engines, and tamper-resistant hardware modules. Software-based protections include secure key management, data encryption and decryption, and secure communication protocols. HW/SW co-protection is an important consideration in the design of modern SoCs, especially those used in mission-critical applications such as aerospace, defense, and healthcare. By combining the strengths of both hardware and software protections, designers can create systems that are highly secure and resistant to attack. Without IP protection, companies can lose revenue and market share [24–26].

HW/SW Co-protection techniques aim to ensure the security and privacy of a system by combining hardware and software solutions to detect and prevent potential threats. These techniques include:

1. Hardware-based encryption: This technique uses dedicated hardware to perform encryption and decryption operations, making it faster and more secure than software-based encryption.
2. Trusted Platform Module (TPM): A TPM is a dedicated hardware module that can securely store cryptographic keys and perform secure boot operations, providing a trusted platform for system protection.
3. Secure boot: A secure boot process ensures that only trusted software is loaded into a system during boot-up, preventing the execution of malicious software.
4. Memory protection: Hardware and software techniques can be used to protect against attacks that exploit vulnerabilities in the memory of a system. This includes techniques such as address space layout randomization (ASLR), which randomizes the location of memory, making it difficult for attackers to exploit memory vulnerabilities.
5. Hardware-based authentication: Hardware-based authentication techniques, such as biometric authentication, use dedicated hardware to ensure that only authorized users are granted access to a system.
6. Intrusion detection and prevention: A combination of hardware and software techniques can be used to detect and prevent intrusions, such as intrusion detection systems (IDS) and intrusion prevention systems (IPS).
7. Hardware-based monitoring: Hardware-based monitoring techniques can be used to detect and prevent unauthorized access to a system, such as hardware-based firewalls.

IP vendors are facing major challenges to protect hardware IPs from IP piracy as, unfortunately, recent trends in IP-piracy and reverse engineering efforts to produce counterfeit ICs have raised serious concerns in the IC design community. IP piracy can take several forms, as illustrated by the following scenarios:

(1) A chip design house buys an IP core from an IP vendor and makes an illegal copy or "clone" of the IP. The IC design house then sells it to another chip design house (after minor modifications) claiming the IP to be its own.
(2) An untrusted fabrication house makes an illegal copy of the GDS-II database supplied by a chip design house and then illegally sells them as hard IP.
(3) An untrusted foundry manufactures and sells counterfeit copies of the IC under a different brand name.
(4) An adversary performs post silicon reverse engineering on an IC to manufacture its illegal clone.

These scenarios demonstrate that all parties involved in the IC design flow are vulnerable to different forms of IP infringement which can result in loss of revenue and market share. Hence, there is a critical need of a piracy-proof design flow that equally benefits the IP vendor, the chip designer, as well as the system designer. A desirable characteristic of such a secure design flow is that it should be transparent to the end-user, i.e., it should not impose any constraint on the end-user with regard to its usage, cost, or performance.

To secure an IP, we need to obfuscate it then encrypt the contents before sending it to the customer. **Obfuscation** is a technique that transforms an application or a design into one that is functionally equivalent to the original but is significantly more difficult to reverse engineer [4]. So, Obfuscation changes the name of all signals to numbers and characters combination. The second level is to encrypt the whole file. Although **encryption** is effective, code obfuscation is an effective enhancement that further deters code understanding for attacker. Moreover, **Watermarking** can be used to protect Soft-IPs. It includes modules duplication or module splitting. Overall, HW/SW co-protection techniques aim to provide a more secure and reliable system by combining the strengths of hardware and software solutions to detect and prevent potential threats.

6.2.1 Hardware Oriented Security and Trust

Hardware Oriented Security and Trust (HOST) is a field of study and research focused on developing secure and trustworthy hardware systems. It encompasses various approaches, techniques, and methodologies that aim to enhance the security, privacy, and resilience of hardware systems against various types of attacks, such as side-channel attacks, fault attacks, hardware Trojans, and other forms of hardware-based cyber threats. The HOST field involves various aspects of hardware security, including the design and implementation of secure hardware architectures, the development of secure hardware modules and components, the evaluation and verification of hardware security properties, the integration of hardware security mechanisms into larger system architectures, and the deployment and management of secure hardware systems. Some of the specific research areas within

HOST include secure hardware design techniques, hardware-based root of trust, hardware security testing and evaluation, trusted execution environments, secure boot and firmware, hardware-based secure storage, and hardware-based cryptography. Secure hardware design techniques refer to a set of methodologies and practices used to design hardware systems that are resistant to various types of attacks and threats, such as side-channel attacks, fault attacks, hardware Trojans, and other forms of hardware-based cyber threats [27–29, 29, 31].

Some of the key techniques used in secure hardware design include:

1. Secure hardware architectures: This involves designing hardware systems that are resistant to various types of attacks, by incorporating security mechanisms such as access controls, firewalls, secure boot, and trusted execution environments.
2. Side-channel analysis resistance: Side-channel attacks are a type of attack that exploits information leakage from a system to extract sensitive information, such as cryptographic keys. Techniques such as power analysis, electromagnetic analysis, and timing analysis can be used to protect against side-channel attacks.
3. Fault attack resistance: Fault attacks involve introducing faults into a system to manipulate its behavior and extract sensitive information. Hardware design techniques such as error correcting codes, redundancy, and fault detection and recovery mechanisms can be used to mitigate the effects of fault attacks.
4. Hardware Trojan detection and prevention: Hardware Trojans are malicious modifications to a hardware design that can introduce vulnerabilities and backdoors into a system. Techniques such as formal verification, design-for-trust, and hardware Trojan detection algorithms can be used to detect and prevent hardware Trojans.
5. Cryptographic hardware design: This involves designing hardware systems that can perform cryptographic operations securely, such as encryption and decryption, key generation, and authentication.

By incorporating these secure hardware design techniques, hardware systems can be made more resistant to a wide range of cyber threats, thereby enhancing overall system security and trustworthiness.

6.3 Conclusions

This chapter discusses the importance of HW/SW co-optimization in the design of complex electronic systems, such as system-on-chip (SoC) designs. The integration of hardware and software is critical to achieving optimal performance, power consumption, and other design goals. HW/SW co-optimization involves creating a tight integration between the hardware and software components, such that they can be optimized together to achieve the desired performance, power consumption, and other design goals. This

approach allows for closer collaboration between hardware and software designers, leading to more efficient system designs. The traditional approach to system design involves designing the hardware and software components separately and then integrating them later in the design process. However, this approach can result in suboptimal performance and higher costs due to the lack of synergy between the hardware and software components. There are several methods for HW/SW co-optimization, including co-design which involves simultaneously designing the hardware and software components of a system. This approach allows for closer collaboration between hardware and software designers, leading to more efficient system designs.

References

1. R. Dunkley, “Supporting a wide variety of communication protocols using partial dynamic reconfiguration,” Proc. IEEE Autotestcon 2012, pp. 120-125, 2012.
2. N. Marques, H. Rabah, E. Dabellani, and S. Weber, “Partially reconfigurable entropy encoder for multi standards video adaptation,” in Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on, June 2011, pp. 492–496.
3. M. Cassel and F. L. Kastensmidt, “Evaluating One-Hot encoding finite state machines for SEU reliability in SRAM-based FPGAs,” in Proc. 12th IEEE Int. On-Line Testing Symp. (IOLTS 2006), 2006, p. 6.
4. Meha Kainth, Lekshmi Krishnan, Chaitra Narayana, Sandesh Gubbi Virupaksha and Russell Tessier “Hardware-Assisted Code Obfuscation for FPGA Soft Microprocessors” DATE, 2015.
5. J. Liu *et al*., "Simulation Data Driven Design Optimization for Reconfigurable Soft Gripper System," in *IEEE Robotics and Automation Letters*, vol. 7, no. 2, pp. 5803–5810, April.
6. A Slowik and M Bialko, "Design and optimization of combinational digital circuits using modified evolutionary algorithm", *Proc. 7th International Conference on Artificial Intelligence and Soft Computing*, vol. 3070, pp. 468–473, 2004.
7. Y. J. Pavitra, S. Jamuna and J. Manikandan, “Hardware Resource Optimization for Embedded System Design: A Brief Review,” 2018 3rd International Conference for Convergence in Technology (I2CT), Pune, India, 2018, pp. 1–6, doi: https://doi.org/10.1109/I2CT.2018.8529817
8. W. Jiang, X. Jianjun, M. Xiankai, Z. Zhuo, Z. Nan and Z. Haoyu, “High-Reliability Compilation Optimization Sequence Generation Framework Based ANN,” *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Macau, China, 2020, pp. 347–355, doi: https://doi.org/10.1109/QRS51102.2020.00053.
9. J. Tang, J. Zhang, Q. Song, S. Liu, Z. Yu and W. Zou, “The Compilation Performance Tests and Analysis of RISC-V Compilation Toolchain On the SPEC CPU®2017,” *2021 9th International Symposium on Next Generation Electronics (ISNE)*, Changsha, China, 2021, pp. 1–4, doi: https://doi.org/10.1109/ISNE48910.2021.9493648
10. Wang Hongsheng, Research on LLVM JIT compilation optimization technology based on domestic platform [D], Zhengzhou University, 2020.
11. Gao Yuchen, Research on OpenMP program compilation optimization technology for domestic processors [D], Strategic Support Force Information Engineering University, 2018.

12. B. -P. Tine, S. Yalamanchili, H. Kim and J. Vetter, "POSTER: Tango: An Optimizing Compiler for Just-In-Time RTL Simulation," 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), Seattle, WA, USA, 2019, pp. 481–482, doi: https://doi.org/10.1109/PACT.2019.00055.
13. F. H. Alexey Kupriyanov and J. Teich, "High-speed event-driven rtl compiled simulation", 2004.
14. G. Nazarian, C. Strydis and G. Gaydadjiev, "Compatibility Study of Compile-Time Optimizations for Power and Reliability," 2011 14th Euromicro Conference on Digital System Design, Oulu, Finland, 2011, pp. 809–813, doi: https://doi.org/10.1109/DSD.2011.108.
15. A. Roelke, R. Zhang, K. Mazumdar, K. Wang, K. Skadron and M. R. Stan, "Pre-RTL Voltage and Power Optimization for Low-Cost, Thermally Challenged Multicore Chips," *2017 IEEE International Conference on Computer Design (ICCD)*, Boston, MA, USA, 2017, pp. 597–600, doi: https://doi.org/10.1109/ICCD.2017.104.
16. S. Li et al., "McPAT: An Integrated Power Area and Timing Modeling Framework for Multicore and Manycore Architectures", *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, 2009.
17. A. Mathur and Q. Wang, "Power Reduction Techniques and Flows at RTL and System Level," *2009 22nd International Conference on VLSI Design*, New Delhi, India, 2009, pp. 28–29, doi: https://doi.org/10.1109/VLSI.Design.2009.113.
18. Y. Attaoui, M. Chentouf, Z. E. A. A. Ismaili and A. El Mourabit, "Clock Gating Efficiency and Impact on Power Optimization During Synthesis Flow," *2021 International Conference on* Microelectronics (ICM), New Cairo City, Egypt, 2021, pp. 13–16, doi: https://doi.org/10.1109/ICM52667.2021.9664896.
19. G. Pouiklis and G. C. Sirakoulis, "Clock gating methodologies and tools: a survey", International Journal of Circuit Theory and Applications, vol. 44, no. 4, pp. 798-816, 2016.
20. K. SP and K. BS, "A Novel Approach for Power Optimization in Sequential Circuits Using Latch Based Clock Gating", International Journal of Electrical Engineering and Technology, vol. 11, no. 4, 2020.
21. P. Saraswat and T. Goyal, "Novel methods of clock gating techniques: a review", international research journal of engineering and technology (irjet), vol. 5, no. 01, 2018.
22. S. Ghandali, B. Alizadeh, M. Fujita and Z. Navabi, "RTL datapath optimization using system-level transformations," Fifteenth International Symposium on Quality Electronic Design, Santa Clara, CA, USA, 2014, pp. 309–316, doi: https://doi.org/10.1109/ISQED.2014.6783341.
23. B. Alizadeh and M. Fujita, "Modular Datapath Optimization and Verification Based on Modular-HED," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 29, pp. 1422–1435, 2010.
24. W. Adi *et al.*, "IP-core protection for a non-volatile Self-reconfiguring SoC environment," *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, Istanbul, Turkey, 2013, pp. 252–255, doi: https://doi.org/10.1109/VLSI-SoC.2013.6673284.
25. S. Drimer, Security for Volatile FPGAs, Ph.D. dissertation, University of Cambridge, August 2009.
26. P. Vanhauwaert, M. Portolan, R. Leveugle and P. Roche, "Usefulness and effectiveness of HW and SW protection mechanisms in a processor-based system," *2008 15th IEEE International Conference on Electronics, Circuits and Systems*, Saint Julian's, Malta, 2008, pp. 113–116, doi: https://doi.org/10.1109/ICECS.2008.4674804.
27. "The Hardware Trojan War: Attacks, Myths, and Defenses", Chapter 10, J.
28. Plusquellic and F. Saqib, "Detecting Hardware Trojans using Delay Analysis", Springer, 2018, ISBN 978–3–319–68511–3
29. "Fundamentals of IP and SoC Security, Design, Verification, and Debug", Chapter 6, J. Plusquellic, "PUF-Based Authentication", Springer, ISBN 978–3–319–50057

30. “Physically Unclonable Functions: Constructions, Properties and Applications”, Roel Maes, Springer, SBN 978–3–642–41394–0, ISBN 978–3–642–41395–7.
31. “Handbook of Applied Cryptography”, A. J. Menezes, P. C. van Oorschot and S. A. Vanstone.
32. Salah, Khaled, and Mohamed AbdelSalam. “IP cores design from specifications to production: Modeling, verification, optimization, and protection.” 2013 25th International Conference on Microelectronics (ICM). IEEE, 2013.