COMPSYS 701: Advanced Digital Design

Overview, Trends and Challenges of Modern Digital Design

Zoran Salcic

Embedded Systems Research Group

Department of Electrical, Computer and Software Engineering

Semester 1, 2025, University of Auckland

COMPSYS701 FYI - Assessments

The course is assessed through the coursework only;

Overarching theme is a design of a heterogeneous multiprocessor as system on chip (HMPSoC) and all assessments are related to it

- Lab 1 and Lab2 with small assignments (10% each) done in groups of 2 students, assessed individually; treated as individual work it provides the foundations for the tools and technologies used to design HMPSoC
- Individual Project (IP) (20%) that results in an Application Specific Processor (nodes) for HMPSoC (one task for each student)
- Group Project (GP) (60%) is done in two parts (each 30%):
  1) GP1 - Design of a simple RISC processor called ReCOP from ISA and programming model specification (30%, 3 students in group)
  2) GP2 - Integration of all components created in GP1 and IP into HMPSoC (30%); this includes an application that uses ASPs from IPs, as well as ReCOP and general-purpose processor (NiOS 2) that must be integrated into HMPSoC

Outline

Setting a scene for the course

- Technology as the Driving Factor in Modern Digital Systems
- Hardware vs Software – Blurring Boundaries
- Hardware Description Languages for Hardware Design
- Programming Languages for Software Development
- System-level Design Languages: SystemC
- System-level Programming Languages: SystemJ and SystemGALS
- Some Concluding Observations

Technology as the Driving Factor in Modern Digital Systems

Technology as a Driving Factor in Digital World

- Technology enables integration of huge number of transistors
- Operation frequencies increasing twofold every two years – limit achieved
- Integration density increases twofold every 18 months – still, but new architectures needed
- Mixing analog, digital circuits and RF on a chip
- Low power requirements increasingly important
- Hardware software trade-offs and co-design
- New applications emerge and become possible

Key issue: Complexity of Solutions

How to Describe, Synthesise and Verify Solutions

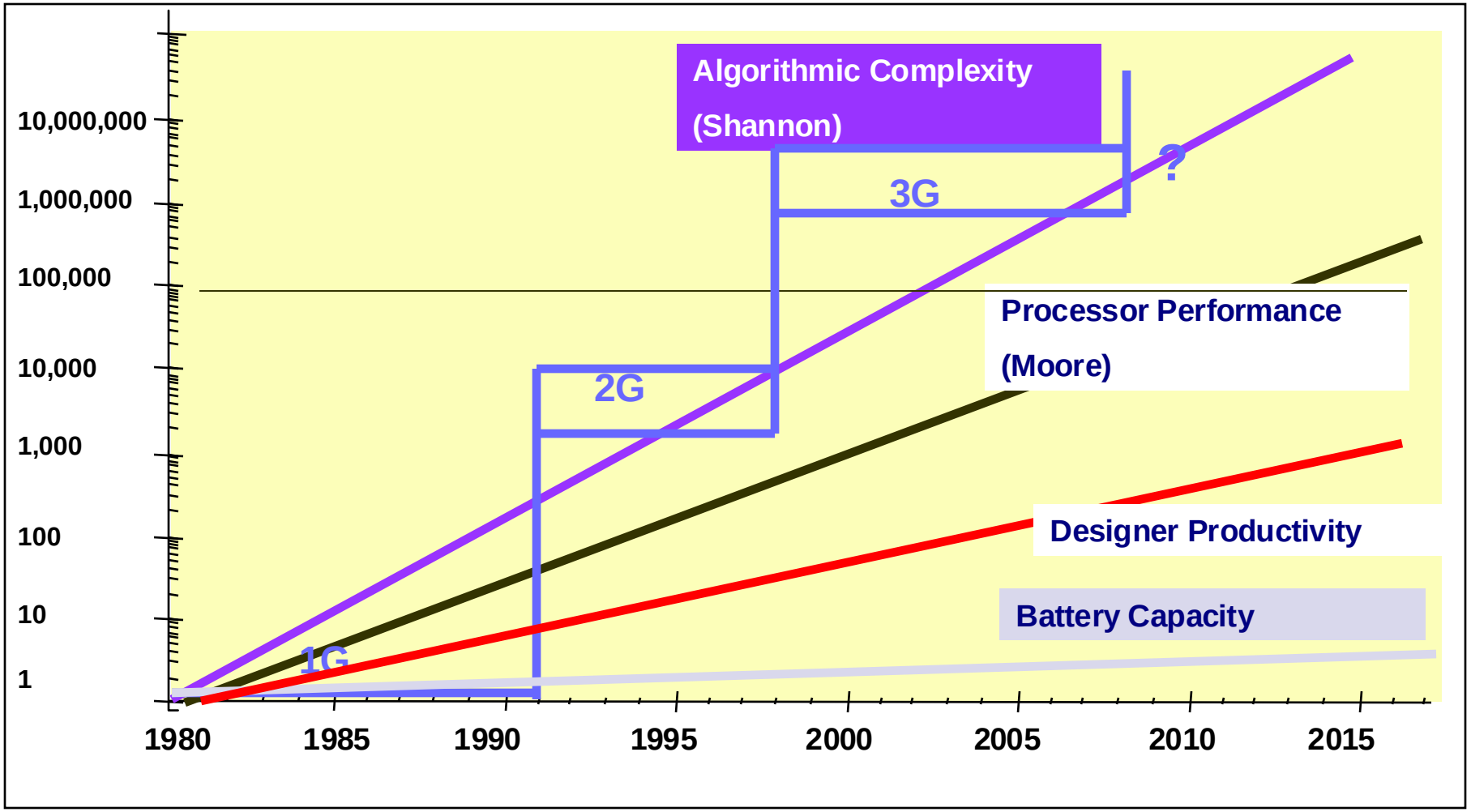
Major Digital Systems Applications

- General purpose computers
- Wireless systems
- Data communications systems
- Vehicle automation
- Intelligent sensors – smart objects in Internet of Things
- Home networking and automation
- Medical systems
- Mechatronics and robotics
- Control systems
- Entertainment
- Other …

Application Requirements vs Technology vs Productivity

(The plots are outdated, but the trends pertain)

Power Consumption Becomes Very Important!



Questions for Future – Challenges Ahead

- What will all those transistors do
  - currently they are with 2- 5 nm features
- What are probable architectures?
  - Homogeneous and heterogeneous multi-core architectures, NoCs
- What kind of applications will absorb all that fabrication capacity?
  - Many applications are now possible, especially related to video processing, sensor fusion, machine learning….
- How will these chips be designed?
  - How to up-scale capabilities of components and integrate them in meaningful ways
- How to partition solutions on HW and SW
  - Manual and automatic HW/SW partitioning and co-design
- How will we trust that they operate correctly?
  - Formal languages may help make these systems “correct-by-design/specification”

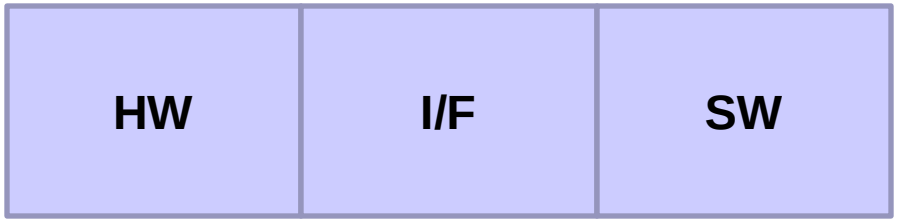
Digital Implementation Technologies and Options System-on-Chip Leading Way Forward

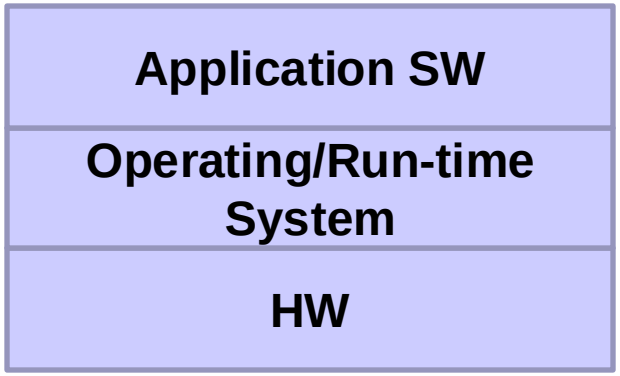
- Application-Specific Integrated Circuits (ASICs)
- Field-Programmable Gate Arrays (FPGAs)
- General Purpose Microprocessors
- DSP processors
- GPUs, NN processors
- Dynamically reconfigurable logic – still in infancy after 30 years of development
    - **Increasing trend towards combinations of the above – Homogeneous and Heterogeneous**

Hardware vs Software – Blurring Boundaries

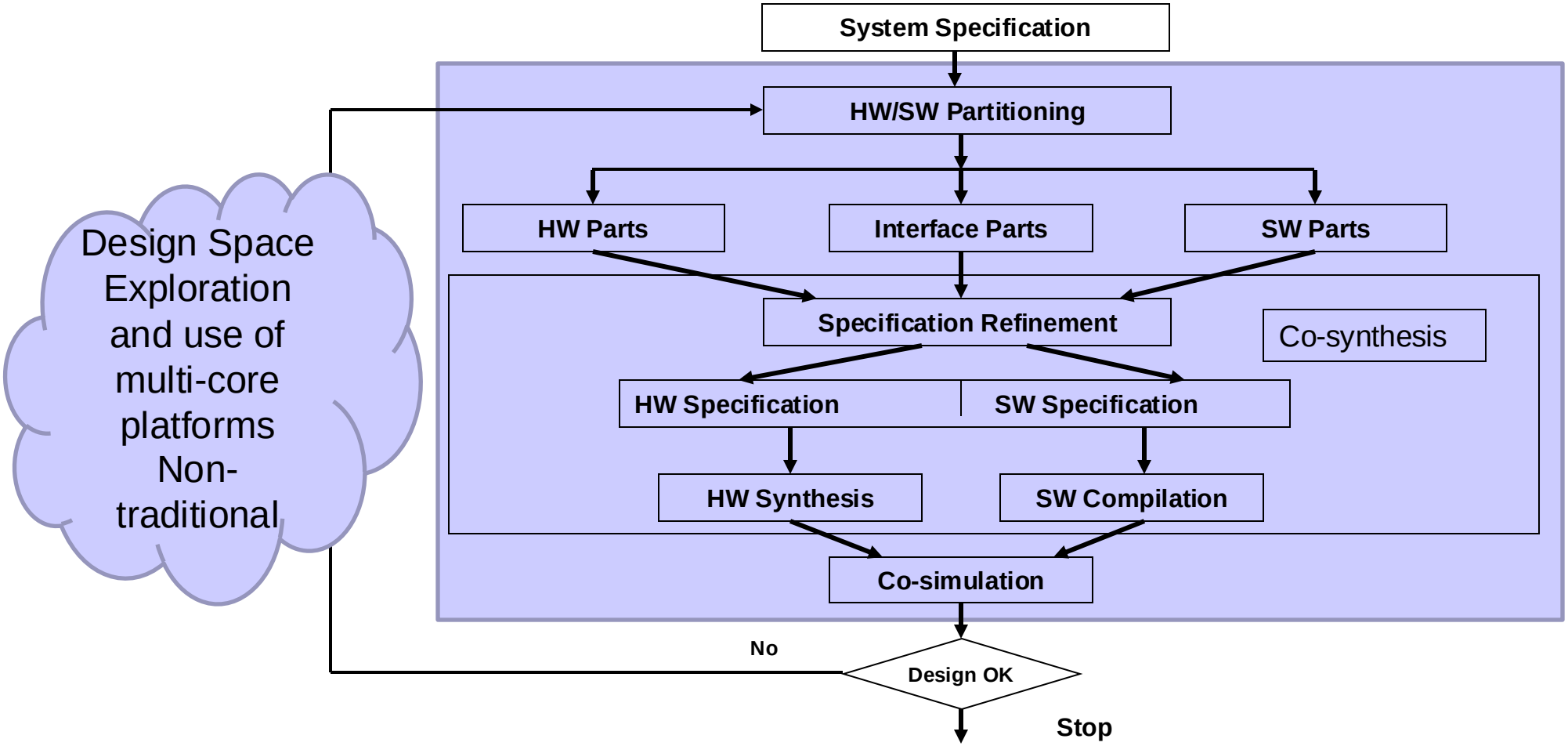
Hardware and Software Working Together

- Comprise of functions implemented in hardware and software
- HW and SW functions collaborate when executing overall system functionality
- Typically HW and SW functionalities separated in early design phase and designed independently
- HW for speed/parallelism; SW for flexibility
- HW typically designed using Hardware Description Languages (HDL) like VHDL, Verilog, System Verilog
- SW typically designed using high level programming languages like C, C++, Java, sometimes low-level machine (assembly) languages





Hardware/Software Co-Design Flow



Hardware/Software Co-Design

- Traditional approach/flow: partitioning done manually at the design phase and then HW and SW designed independently – careful selection of interfaces makes system operation likely successful when integrating HW and SW parts
- Use of FPGAs introduced the need for automated partitioning, which is difficult
- Separation of control flow of an application from data processing intuitive way for HW/SW partitioning
- Design space exploration (evaluation of alternative designs) helps in automating the process of partitioning and co-design
- High-level specification languages – system-level descriptions

Hardware Description Languages for Hardware Design

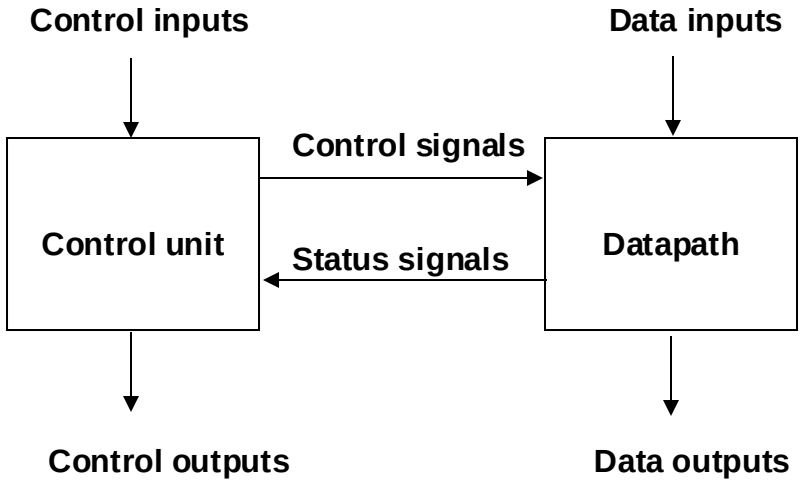
Hardware Description Languages to Design HW

- VHDL and Verilog are the best-known examples (standards)
- Both languages provide modeling tools for
  - Hierarchy
  - Concurrency
  - Behavioral, data flow and structural models
  - Module communication facilities
  - Multiple-valued logic
  - Support simulation and synthesis

- Initially aimed at simulation of digital systems behaviours (discrete event simulators) and then used for synthesis as well
- Ambiguous semantics; different designers can use them for their own design styles
- Increasing trend towards using programming language as the specification language, then high level synthesis (HLS)

Register transfer level (RTL) design model

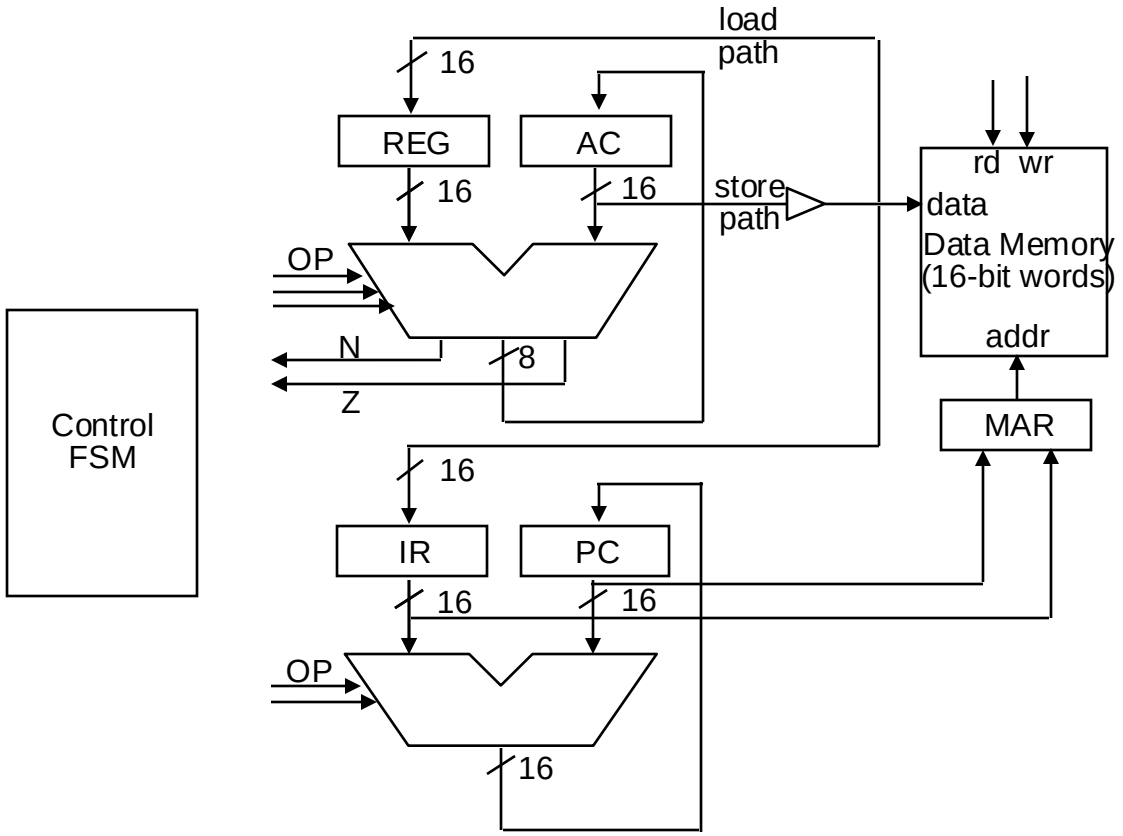
- The general design model for RTL design consists of control unit and datapath
- Two types of I/O ports:
  - data ports (inputs and outputs) to exchange data with the outside environment
  - control ports to control the operations performed by the datapath and receive information about the status of selected registers in the datapath



Block Diagram of a Simple Processor

Register Transfer View of von Neumann Architecture

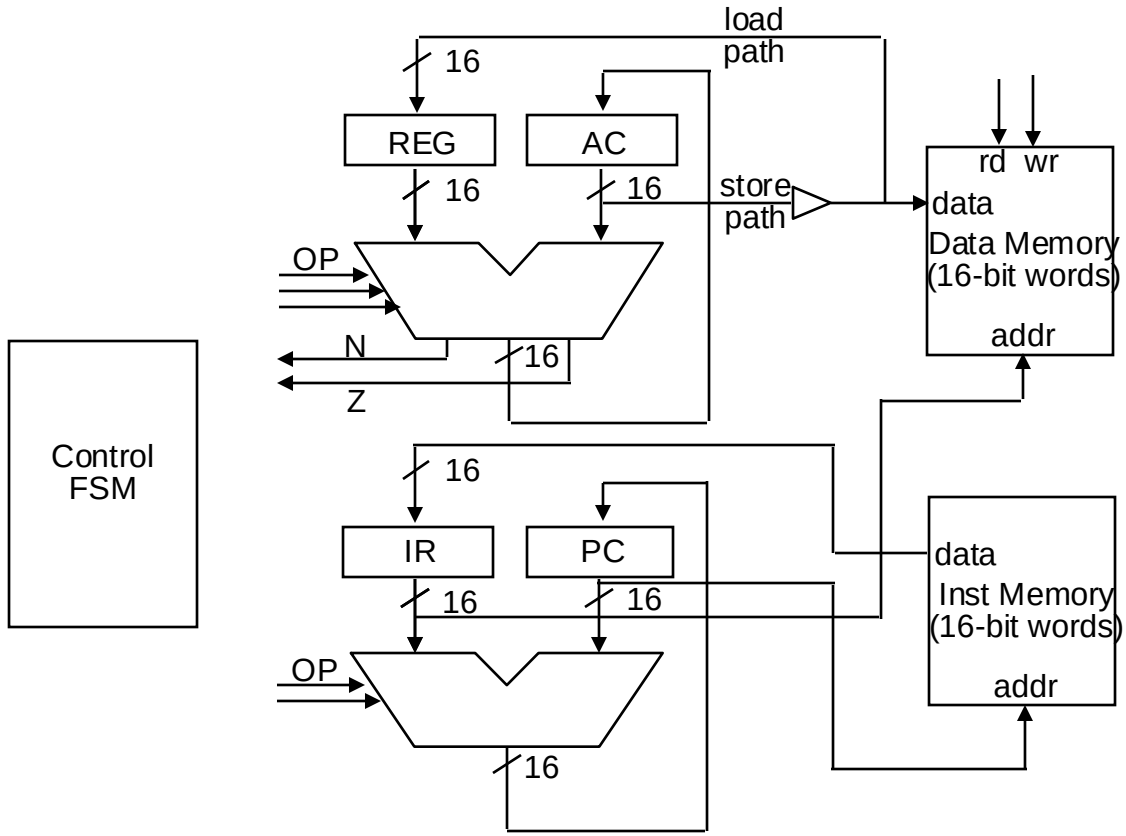
- Which register outputs are connected to which register inputs
- Arrows represent data-flow, other are control signals from control FSM
- MAR may be a simple multiplexer rather than separate register
- MDR is split in two (REG and IR)
- Load control for each register



Block Diagram of a Simple Processor

Register transfer view of Harvard architecture

- Which register outputs are connected to which register inputs
- Arrows represent data-flow, other are control signals from control FSM
- Two MARs (PC and IR)
- Two MDRs (REG and IR)
- Load control for each register



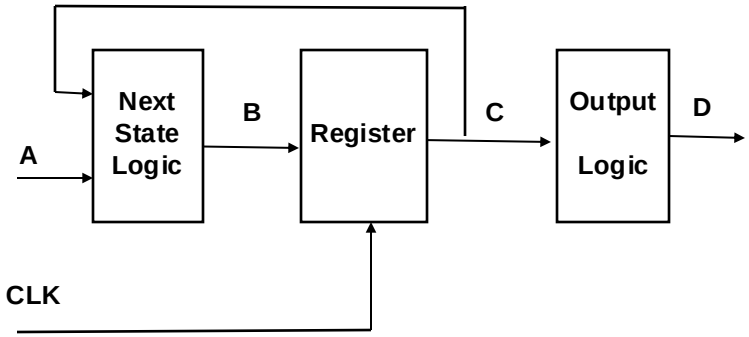
VHDL – Sequential Logic

FSMs - Moore Machines

- Outputs of combinational functions are the functions of their respective current inputs
- Register block holds the current state of the FSM
- Outputs of the Moore FSM are the functions of the state of the machine
- The Moore FSM can be represented by three processes each corresponding to one of the functional blocks

VHDL – Sequential Logic

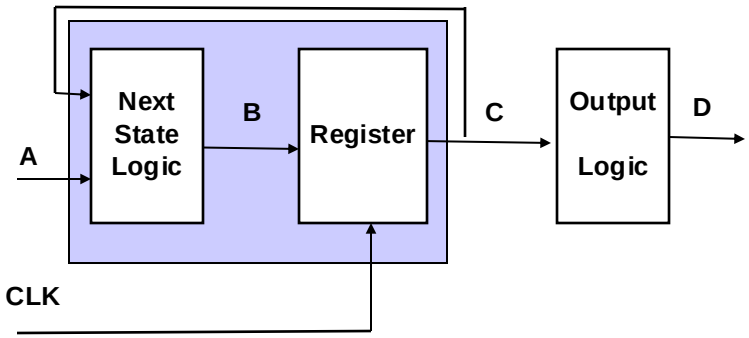
- The Moore FSM can be represented by three processes each corresponding to one of the functional blocks




```
ENTITY system IS
  port (clock: IN STD_LOGIC; a: IN some_type;
          d: OUT some_type);
END system;

ARCHITECTURE moore1 OF system IS
  SIGNAL b, c: some_type;
BEGIN
next_state: process (a, c) – next state logic
BEGIN
  b <= next_state_logic(a, c);
END process next_state;
system_Output: process (c)
BEGIN
  d <= Output_logic(c);
END process system_output;
state_reg: process
BEGIN
wait until rising_edge(clock);
  c <= b;
END process state_reg;
END moore1;
```

VHDL – Sequential Logic

More compact description of Moore FSM

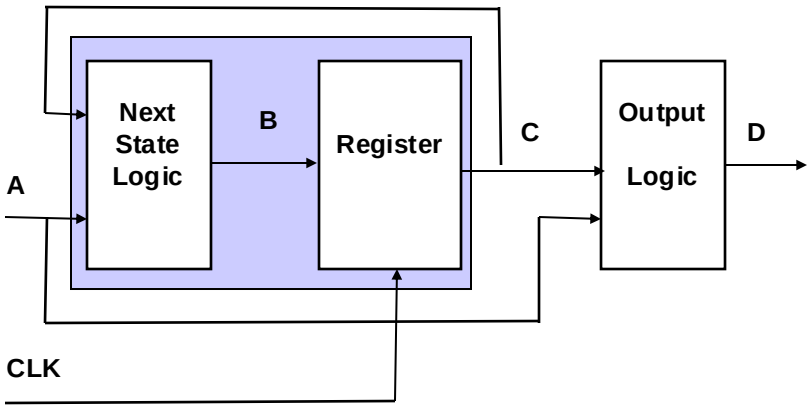



```
ENTITY system IS
  port (clock: IN STD_LOGIC; a: IN some_type;
          d: OUT some_type);
END system;

ARCHITECTURE moore2 OF system IS
  SIGNAL c: some_type;
BEGIN
system_output: process (c)--combinational logic
BEGIN
  d <= output_logic(c);
END process system_output;

next_state: process-- sequential logic
BEGIN
  wait until rising_edge(clock);
  c <= next_state_logic(a, c);
END process next_state;
END moore2;
```

VHDL – Sequential Logic

FSMs – Mealy Machine

- Outputs of the Mealy FSM are functions of both state and current inputs




```
ENTITY system IS
port (clock: IN STD_LOGIC; a: IN some_type;
  d: OUT some_type);
END system;

ARCHITECTURE mealy OF system IS
  SIGNAL c: some_type;
BEGIN

system_output: process (a, c)--combin. logic
BEGIN
  d <= output_logic(a, c);
END process system_OUTput;

next_state: process -- sequential logic
BEGIN
  wait until rising_edge(clk);
  c <= next_state_logic(a, c);
END process next_state;
END mealy;
```

Programming Languages for Software Development

Programming Languages - for SW Development

- Programs describe sequences of instructions for a processor to execute
- Instructions communicate through memory, which is an array of storage locations that hold their values until changed
- Each machine instruction typically performs an elementary operation (e.g. add/subtract two numbers, compare two numbers etc.)
- High-level languages aim to specify many instructions concisely and intuitively
- The C language provides arithmetic expressions, control-flow constructs such as loops and conditionals, and recursive functions.
- Additionally, the C++ language provides classes as a way to build new data types, templates for polymorphic code, exceptions for error handling, and a standard library of common data structures.
- Java is a still higher-level language that provides automatic garbage collection, threads, and monitors to synchronize them.

Programming Language C

- Instructions in a C program run sequentially, but control-flow constructs such as loops of conditionals can affect the order in which instructions execute
- When control reaches a function call in an expression, control is passed to the called function, which runs until it produces a result, and control returns to continue evaluating the expression that called the function.
- C derives its types from those a processor manipulates directly: signed and un-signed integers ranging from bytes to words, floating point numbers, and pointers
- These can be further aggregated into arrays and structures—groups of named fields.
- C programs use three types of memory:
  - Space for global data is allocated when the program is compiled
  - The stack stores automatic variables allocated and released when their function is called and returns
  - The heap supplies arbitrarily-sized regions of memory that can be deallocated in any order

Programming Language C++

- C++ extends C with structuring mechanisms for big programs:
  - user-defined data types
  - a way to reuse code with different types
  - namespaces to group objects and avoid accidental name collisions when program pieces are assembled
  - exceptions to handle errors
- The C++ standard library includes a collection of efficient polymorphic data types such as arrays, trees, strings for which the compiler generates custom implementations
- A class defines a new data type by specifying its representation and the operations that may access and modify it.
- Classes may be defined by inheritance, which extends and modifies existing classes.
- A template is a function or class that can work with multiple types. The compiler generates custom code for each different use of the template. For example, the same template could be used for both integers and floating-point numbers.

Programming Language Java

- Java language resembles C++ but is incompatible
- Java is object-oriented, providing classes and inheritance
- It is a higher-level language than C++ since it uses object references, arrays, and strings instead of pointers
- Java's automatic garbage collection frees the programmer from memory management
- Java provides concurrent threads - Creating a thread involves extending the *Thread* class, creating instances of these objects, and calling their *start* methods to start a new thread of control that executes the objects' *run* methods
- *Synchronizing* a method or block uses a per-object lock to resolve contention when two or more threads attempt to access the same object simultaneously
- A thread that attempts to gain a lock owned by another thread will block until the lock is released, which can be used to grant a thread exclusive access to a particular object

System-level Design Languages: SystemC

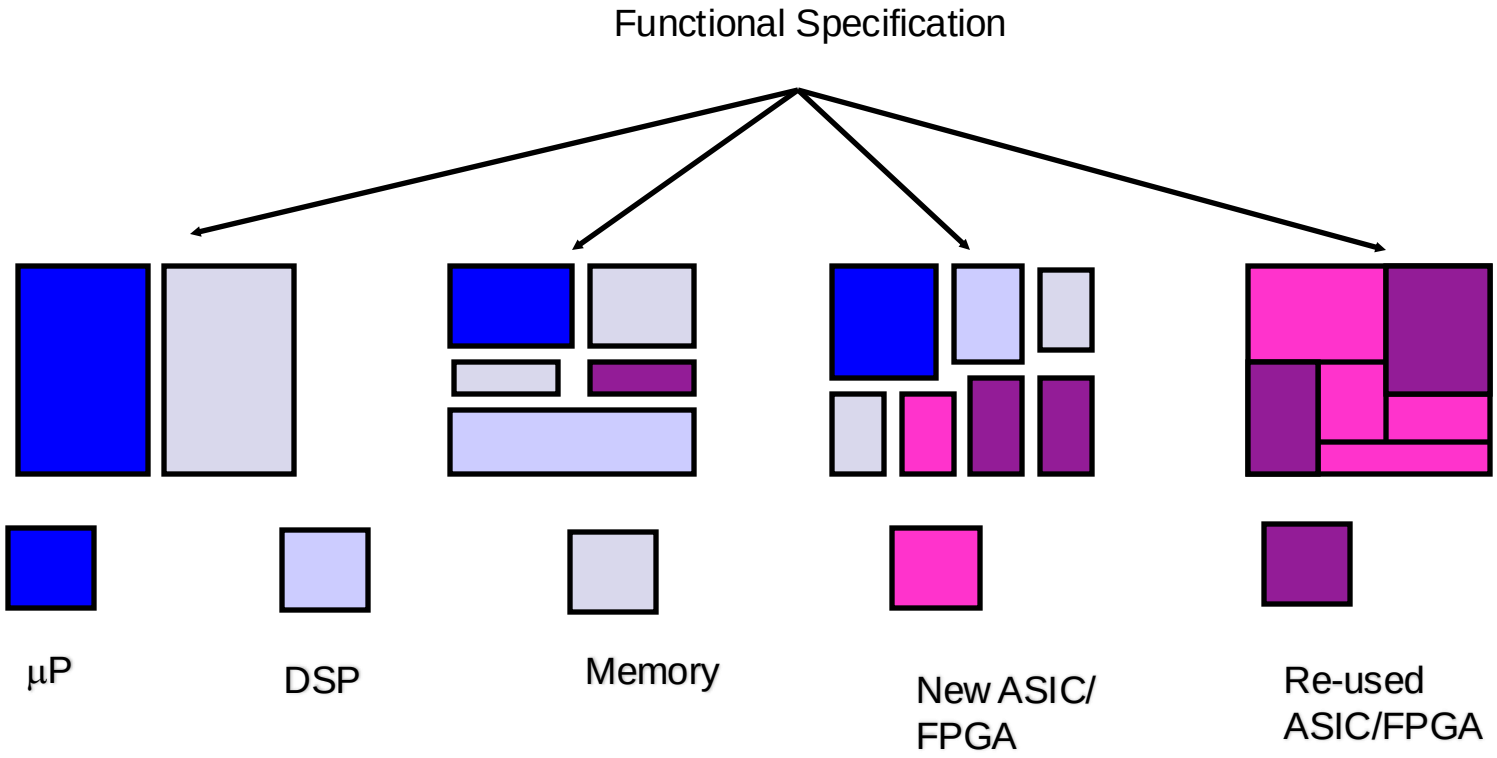
System-level Design Languages: SystemC

- SystemC language is a C++ subset for specifying and simulating synchronous digital hardware
- A SystemC specification can be simulated by compiling it with a standard C++ compiler and linking in freely-distributed class libraries
- The SystemC language builds systems from Verilog- and VHDL-like modules.
- Each has a collection of I/O ports and may contain instances of other modules or processes either synchronous or asynchronous
- As it is based on C++, it allows modelling design entities using traditional programming => it allows specification of behaviour of entire system
- SystemC's simulation semantics are synchronous:

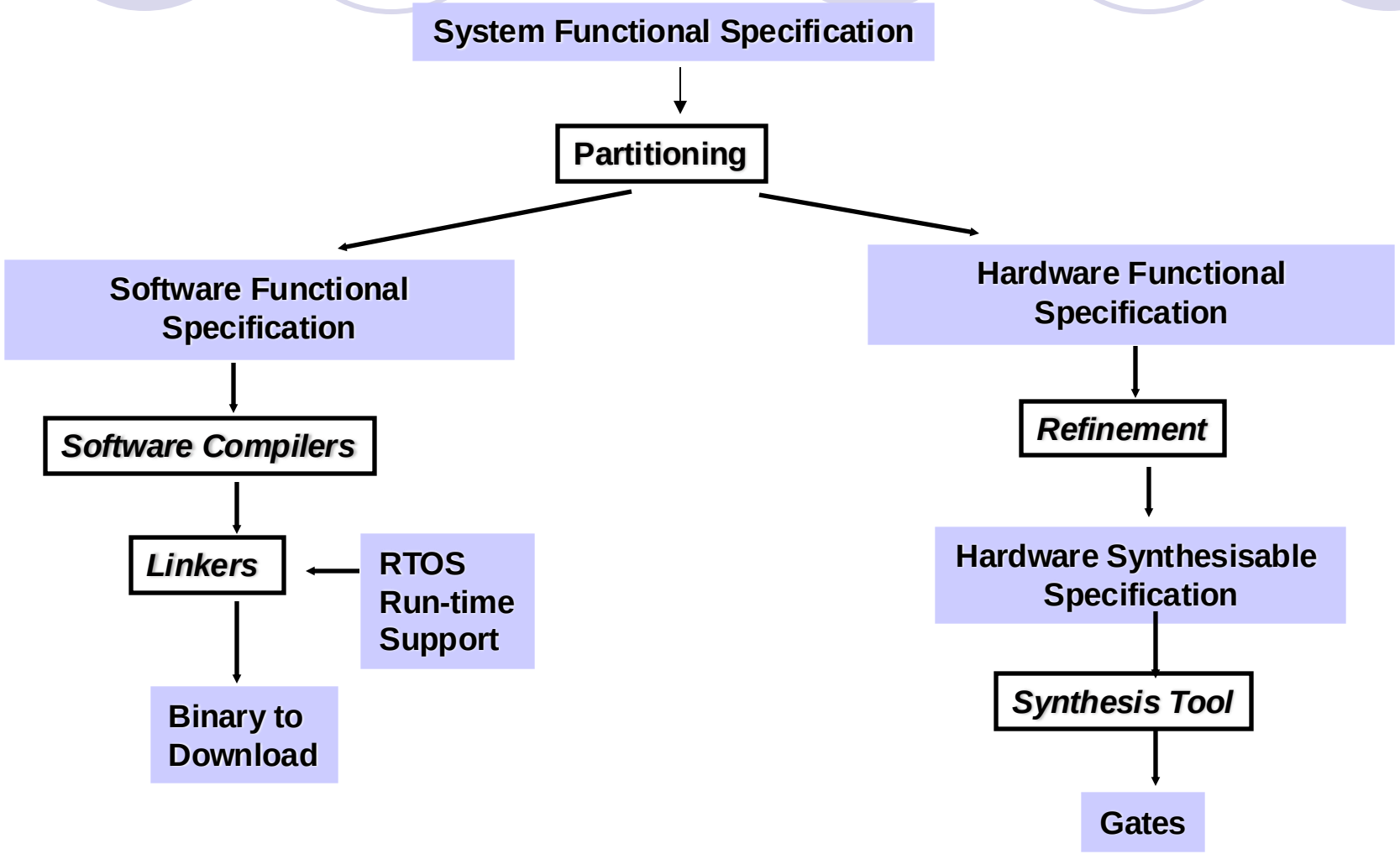
  when a clock arrives, each synchronous process sensitive to that clock runs, then asynchronous processes sensitive to changes on the outputs of those processes run until they stabilize, and the process repeats

SystemC as Co-Design Language

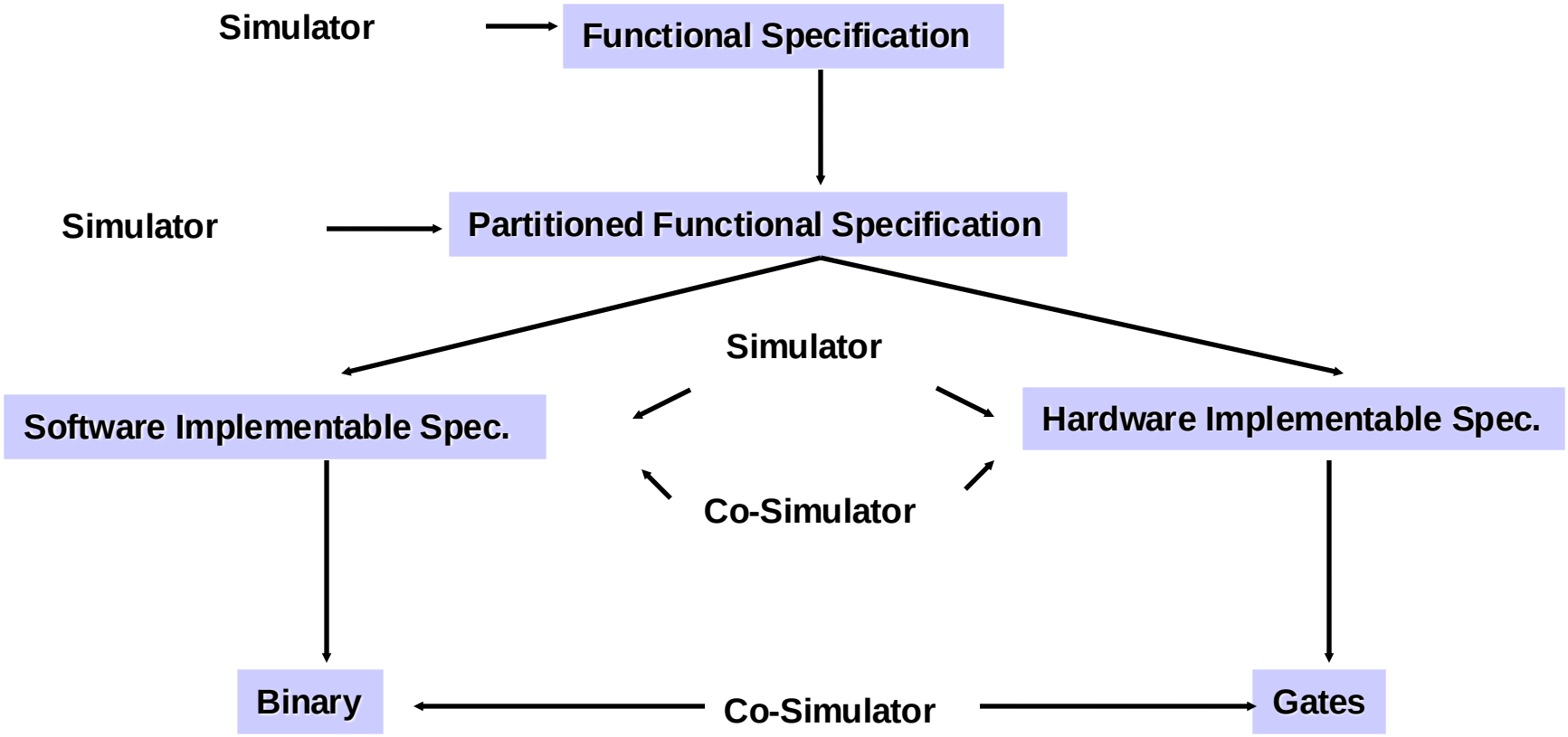
- One language for the specification of entire system
- Specification and implementation from the same language for both hardware and software



From SystemC to Implementation



From SystemC to Verification



System-level Programming Languages: SystemJ and SystemGALS

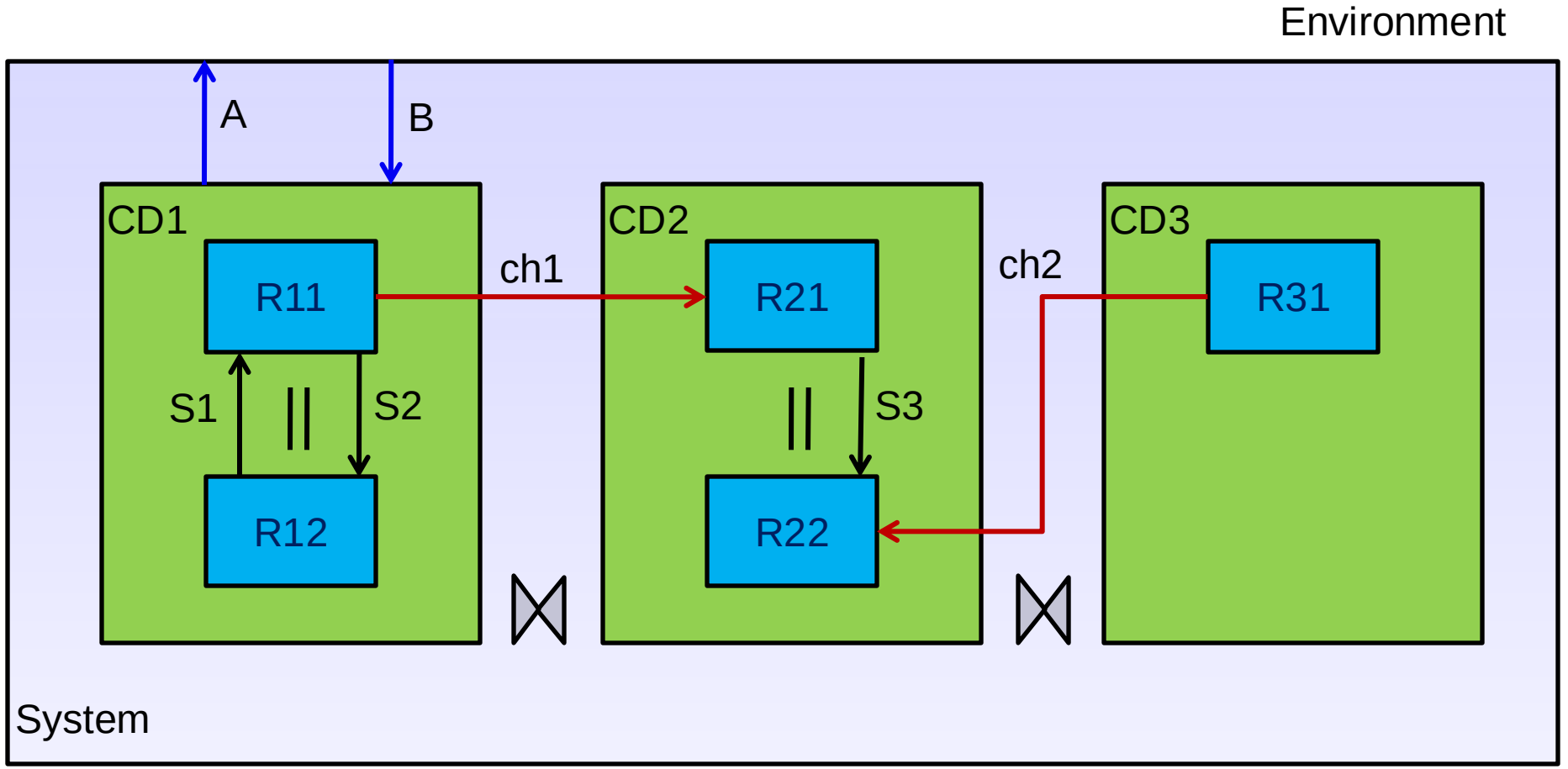
SystemJ Approach to System-Level Software Design

- Developed at UoA
- New system-level language
- SystemJ = Java + Esterel + CSP
- Enables multiclock designs with GALS (Globally asynchronous Locally Synchronous) formal model of computation (MoC)
- Combination of locally synchronous designs using global asynchrony
- Synchronous designs are reactive - S/R model of computation
- Asynchrony implemented using channels
- Communication with the environment through signals
- Communication within single clock domain using signals

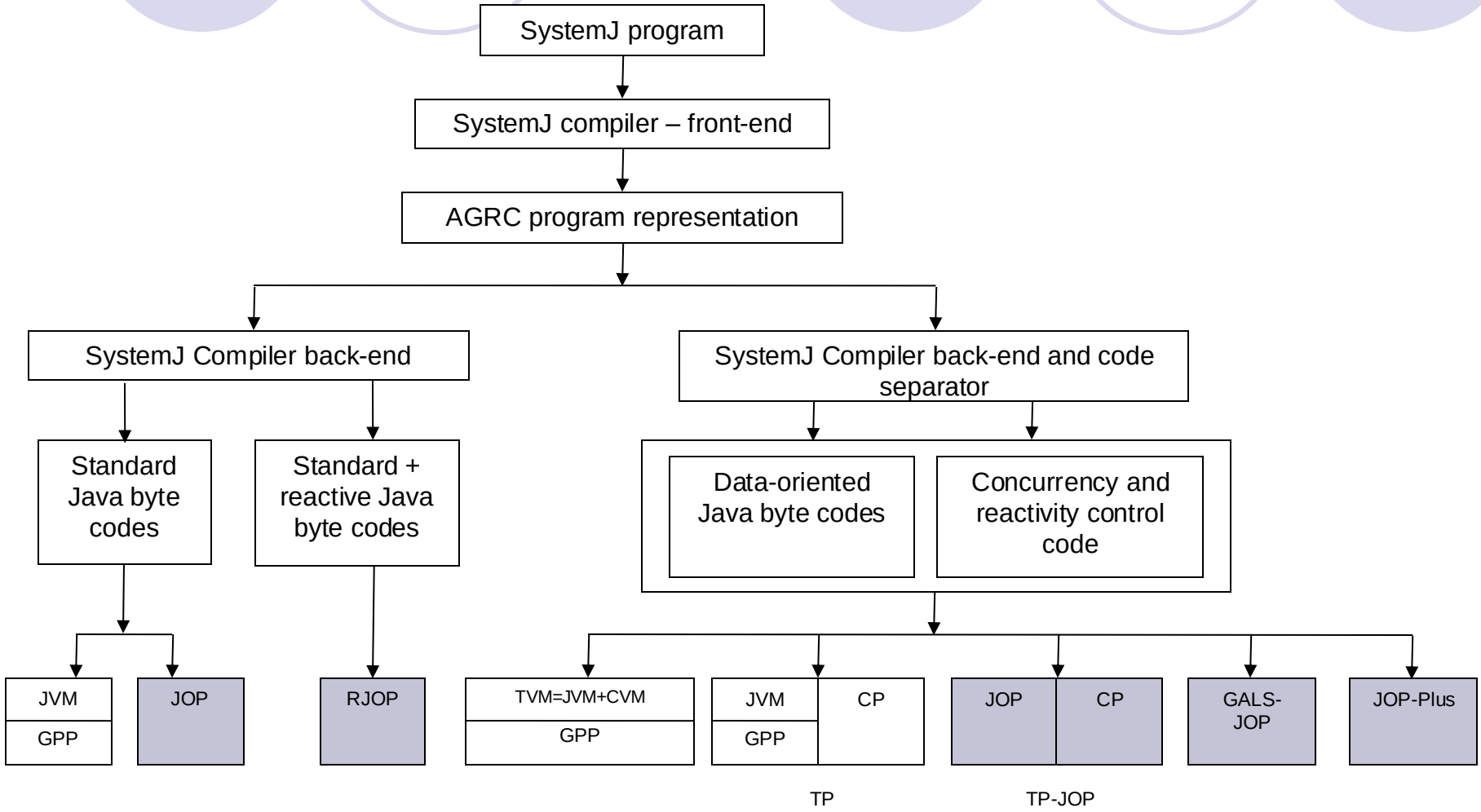
SystemJ Approach to System-Level Design

- Separation of computation from communication
- GALS model of computation
- Behavioural hierarchy
- Support for exceptions and exception handling:
- Mix of data-dominated and control-dominated behaviours and processing
- Support for formal verification
- Language has full constructive operational semantics
- May look like HDL!!

SystemJ Program (System) Example



Execution on Embedded Processors



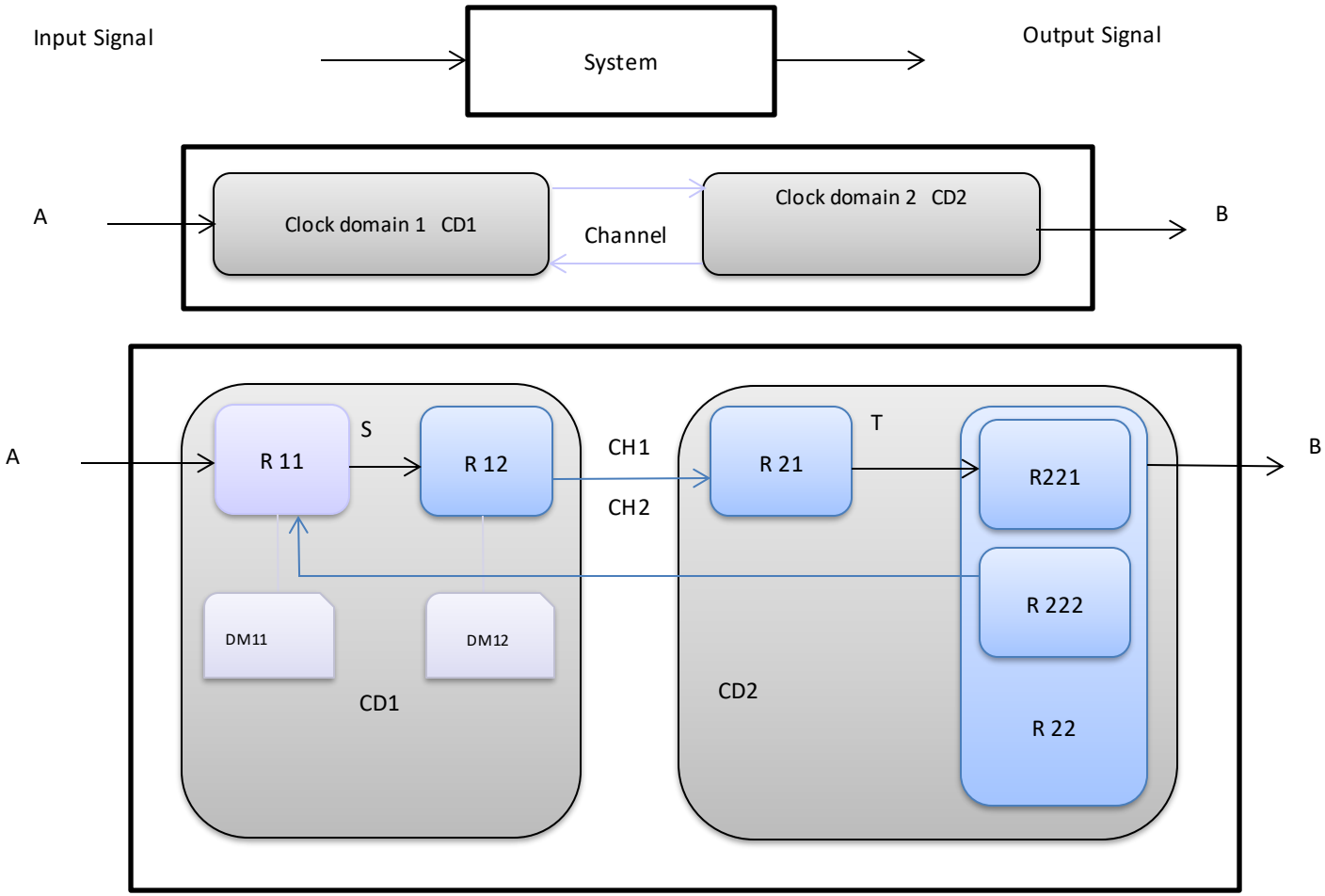
System-level Programming Languages: SystemGALS

SystemGALS Approach to Software Systems Design

- Being developed at UoA
- New language for software systems design
- SystemGALS = Esterel + CSP + C (as a host language) + Other
- Enables multiclock designs with GALS (Globally asynchronous Locally Synchronous) formal model of computation (MoC)
- Shares most of the features with SystemJ with major differences:
  - Stricter separation of control and data computations
  - Data computations conceptually can be specified in any programming language or HDL
  - Supports HW/SW co-design
  - By default translates programs into C
  - Suitable for heterogeneous multi-core/multiprocessor implementations

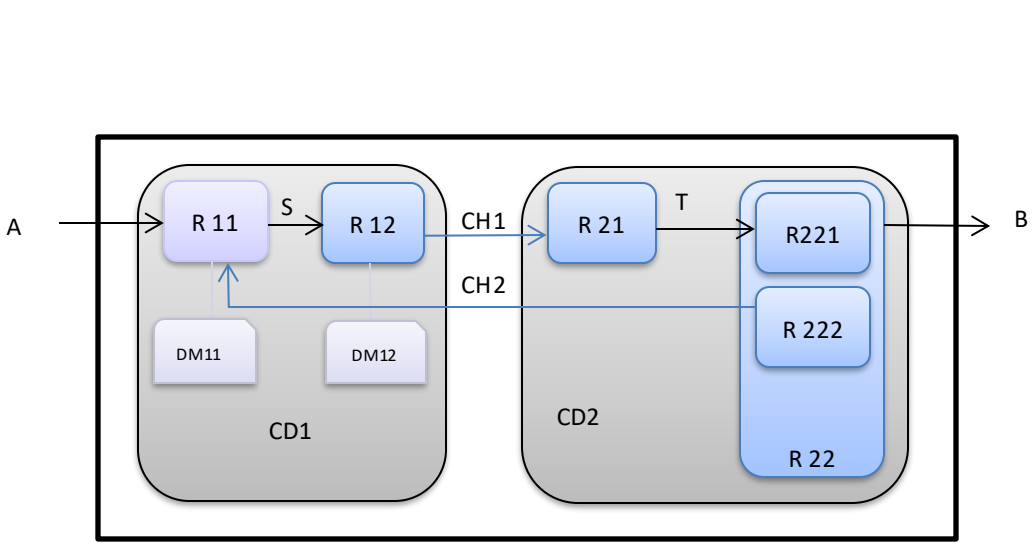
SystemGALS Approach to Software Systems Design

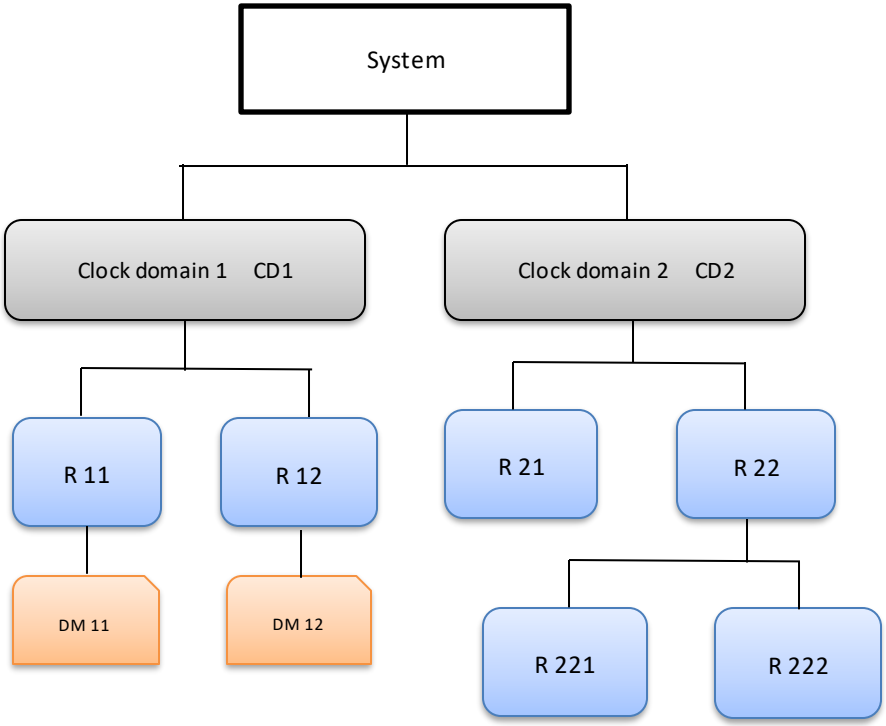
Composition of software systems



SystemGALS Approach to Software Systems Design

Hierarchy of design components





More on SystemGALS in separate presentation

Some General Observations

- Trends and Challenges -

Some Observations - Designing the Chips Faster

- Chips developed for narrow market segments and smaller production runs
- Large System-on-Chip (SoCs) that push the limits of Moore's Law in mass products (e.g. mobile phones)
- Focus shifts towards optimisation, market requirements and time to market (automotive, the cloud, industrial market)
- Emphasis first on how chips behave but then how they meet power, performance and cost metrics, security
- In Internet of Things different market segments require different products

Some Observations - Designing the Chips Faster

- Need for faster and more integrated tools
- Emulation (in FPGAs getting traction) before production runs
- System-level design including system-level simulation
- Virtual prototyping “must-to-have”
- Importance of “correct-by-construction” approach
- How to go beyond RTL designs => High-Level Synthesis => System-Level Design
- Need for automation and algorithmic content
- Breeding good designers – those who can deal with multiple levels of abstraction used in system design
- Master abstraction
- Key is in understanding of hardware/software relationship