COMPSYS723 Embedded Systems Design

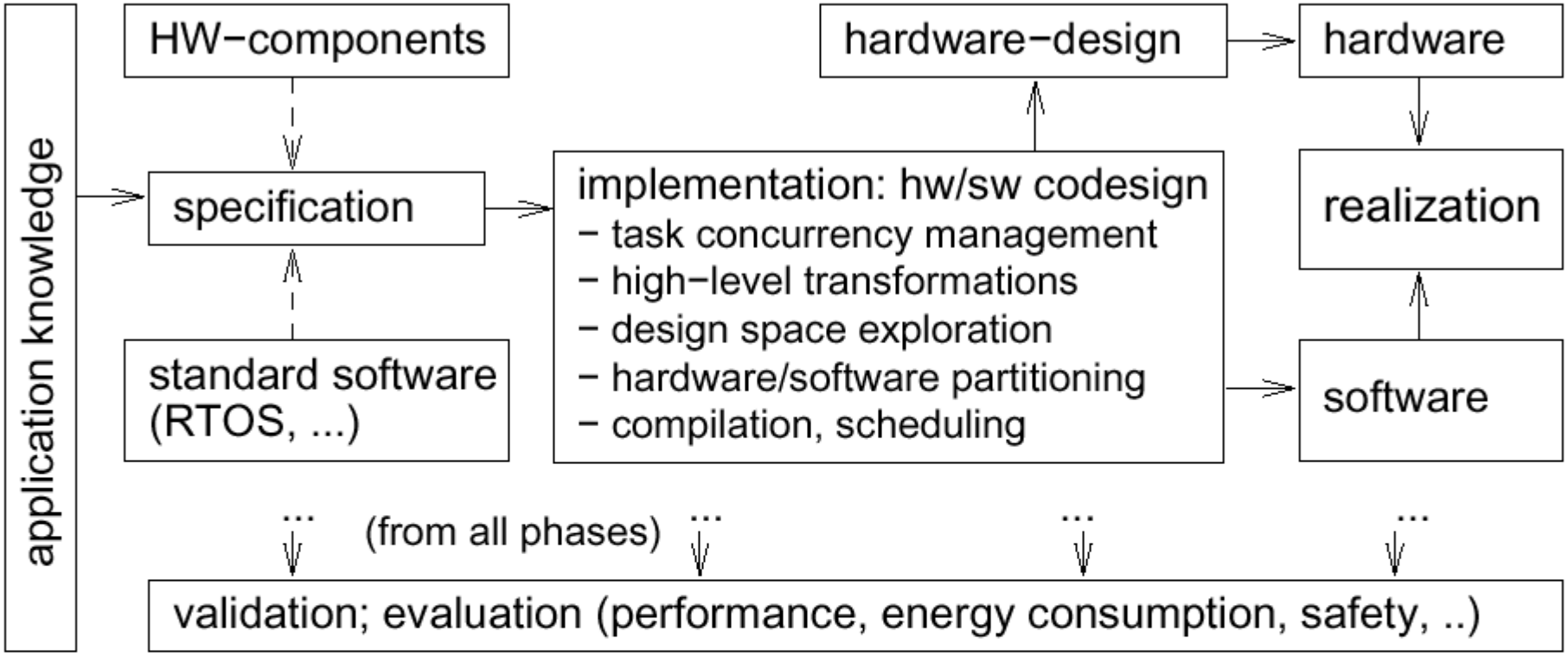
LS-8 Embedded Systems Design- From Specification to Implementation

Zoran Salcic
Auckland University
Semester 1, 2024

Outline

- System-level tasks
  - Specification
  - Exploration of design alternatives
  - Partitioning
  - Hardware and software synthesis
  - Verification
- Models of Computation
  - Models of computation for control- and data-dominated/driven systems
- Embedded systems specification/design and programming languages
  - Hardware-description languages
  - Programming languages
  - RTOS
  - System-level languages

System design flow – a simplified view



Hardware vs. Software

- Hardware - Functionality implemented using customized architecture (typically RTL datapath + FSM)
- Software – Functionality implemented on a programmable processor
  - Allocation of system components and
  - Specification of their physical and performance constraints

- Key differences: -
  - Multiplexing
    - software modules multiplexed with others on a processor (e.g. using an OS); exception multiprocessor and multicore systems
    - hardware modules are typically mapped individually on dedicated hardware
  - Concurrency
    - processors usually have s single thread of control
    - hardware may have multiple parallel (concurrent) datapaths

System-Level Design Tasks

- Specification capture - The description of the model in a language
- Exploration of design alternatives - To find one that best satisfies constraints
  - Allocation of system components and
  - Specification of their physical and performance constraints

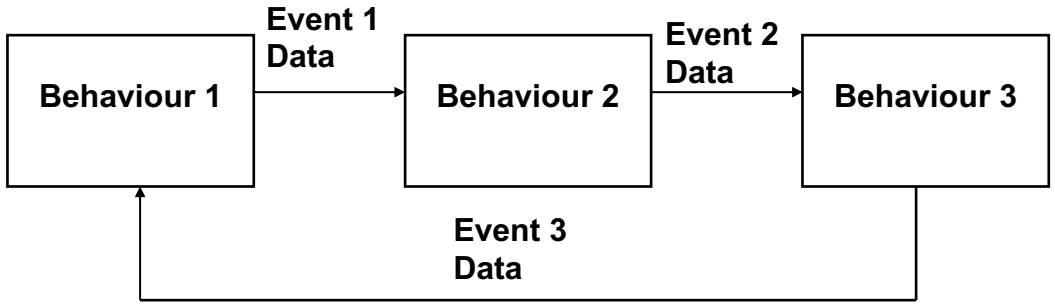
- Specification refinement - It includes
  - Allocation of variables to memories
  - Insertion of interface protocols between components
  - Addition of arbiters to resolve conflicts when concurrent processes access common resources
  - Specification of processors, memories and buses

System-Level Design Tasks

- Software and hardware design - An implementation is created for each component Using software and hardware design techniques.
  - A processor component requires software synthesis.
  - RT-level netlists are generated for hardware components
- Physical design - It generates manufacturing data for each component.
  - For software implemented components it is as simple as compiling code into an instruction set sequence.
  - For hardware part, RT-level netlists are converted into layout data for gate arrays, FPLDs, or custom ASICs using physical design tools for placement, routing, and timing.

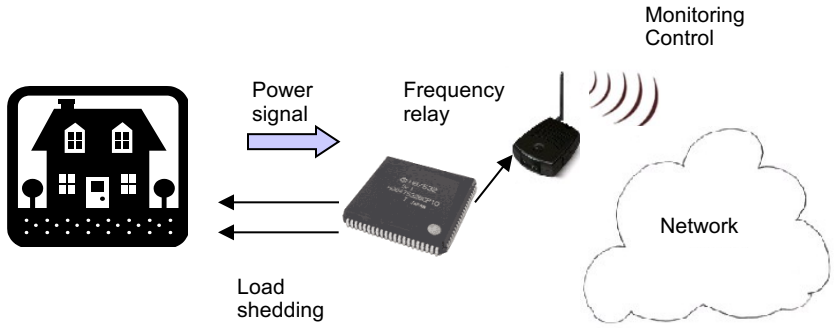
Specification Capture

- Model creation, description generation and simulation using conceptual modeling
- Functionality of the system decomposed into pieces
- The result is functional specification without implementation details
- Relationships between the pieces are described in terms of their execution order and data passing between them

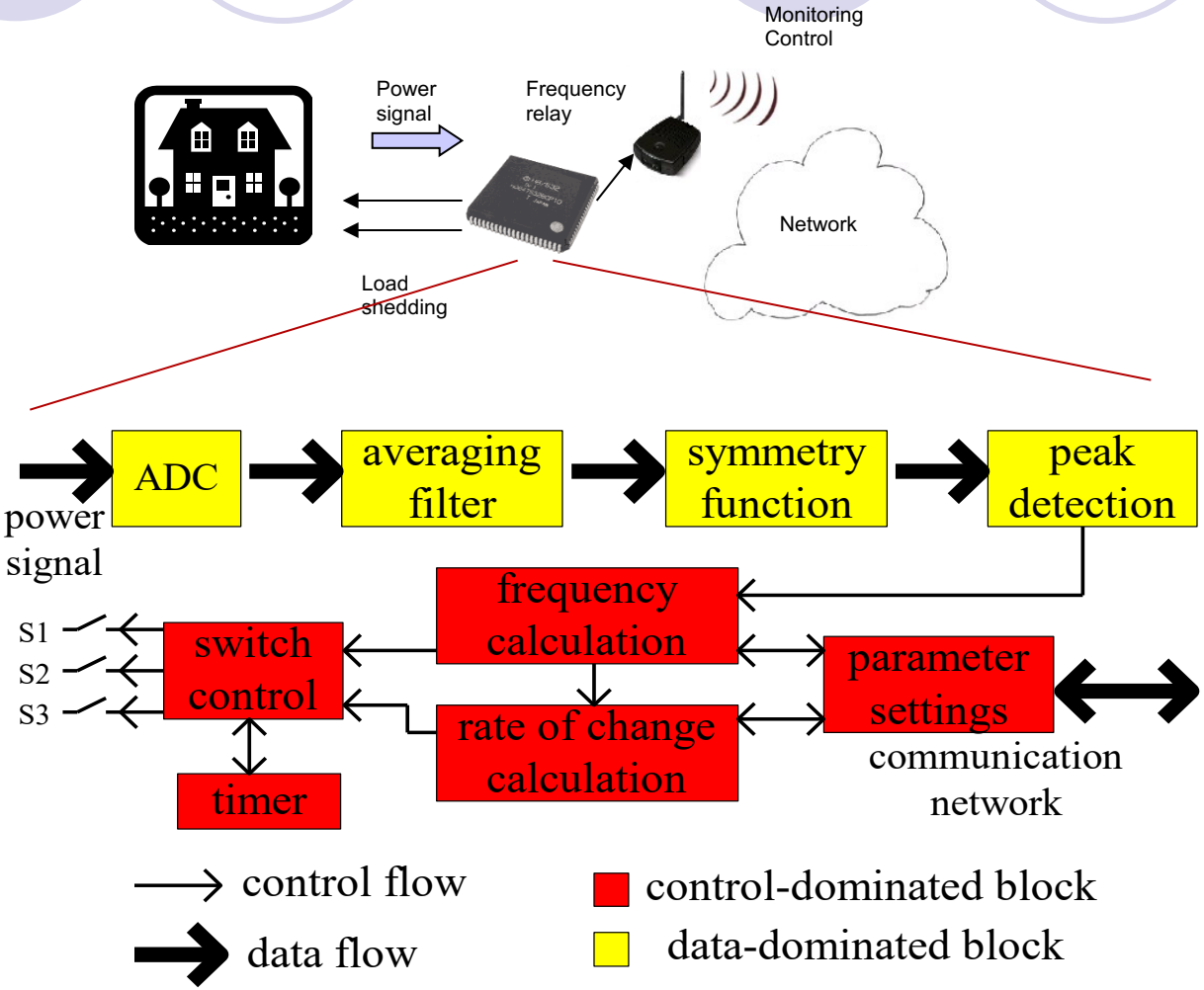


A Heterogeneous Embedded System Example: Internet-enabled Frequency Relay

- Measures current frequency and its rate of change in power signal
- Sheds the loads if necessary (switches loads off and on)
- Communicates with other supervisory and control system through landline or wireless system (higher level control)
- Operates non-stop
- Low-cost for use at household level



Internet-enabled Frequency Relay



Specification Capture

Model Creation for Embedded Systems

- No model is ideal for all types of systems and behaviors
- The best is one that most closely matches the characteristics of the system it models
- Embedded systems include:
  - hierarchy of functional modules
  - concurrency
  - state transitions
  - exceptions and
  - program instructions (algorithmic style descriptions)

Specification Capture

Description generation

- System functionality is captured in a specification using one of many different languages
- Very often there is not one-to-one correspondence between model characteristics and language constructs
- VHDL and Verilog are popular standards that support most of the features – but suitable only for hardware design
- Programming languages (sequential and concurrent) used for software functionalities
- If some of the features are not supported, a greater effort is needed to describe them, which usually ends in more lines of code

Specification Refinement

The refinement process consists of adding further details about system functionality and then generating a system-level description

- Memories (grouping of variables and memory access protocol)
- Interfacing (data communication between modules – bus size, protocol generation, protocol matching)
- Arbitration (insures only one component/behaviour/process accesses the same resource at a given time for modification – fixed priority schemes or dynamic priority schemes which make decision in run-time)

Software and Hardware Synthesis

- Software synthesis converts a complex description into a traditional program that can be compiled by traditional compilers
- Several issues when transferring from system-level description to traditional program:
  - System-level description using concurrent tasks mapped to a single processor must be scheduled to execute sequentially
  - “busy-waiting” time must be reduced
  - A given task must satisfy some timing constraints (response times) on input events, and to output data at certain rate to satisfy system performance

Software and Hardware Synthesis

Hardware synthesis from a system-level description - hardware parts must be synthesized:

- System component's functional description is transformed into a structure of RTL components such as registers, multiplexers and ALUs
- It usually contains datapath and controller implementing finite state machine
- The controller controls register transfers in the datapath and generates signals for communication with the other components

Validation

- Specification must be validated for its completeness and correctness
  - It is **complete** if it includes all possible input sequences that the environment might provide to the system
  - It is **correct** if it generates expected output for every such input sequence
- Validation of correctness and completeness may be performed by formal verification techniques or simulation techniques or their combination

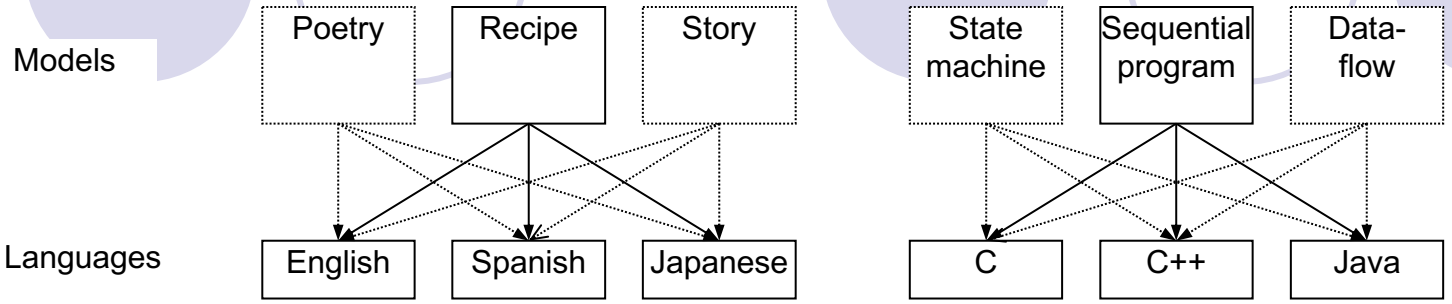
Validation

- Simulation involves executing the specification and then comparing the generated output sequence with the sequence of expected values (However, it does not entirely validate a specification's completeness)
  - Simulation is performed by writing own testbenches or using functional or timing simulators with input vectors or waveforms
- Cosimulation - simulation is performed by integrating simulations at various levels of abstraction (RT-level components (hardware) along with the instructions running on a processor (software))

Models and languages

- How can we (precisely) capture behavior?
  - We may think of languages (e.g. C, C++), but *computation model* is the key
- Common computation models:
  - Sequential program model
    - Statements, rules for composing statements, semantics for executing them
  - Communicating process model
    - Multiple sequential programs running concurrently
  - State machine model
    - For control dominated systems, monitors control inputs, sets control outputs
  - Dataflow model
    - For data dominated systems, transforms input data streams into output streams
  - Object-oriented model
    - For breaking complex software into simpler, well-defined pieces

Models vs. languages



Recipes vs. English **Sequential programs vs. C**

- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C
- Variety of languages can capture one model
  - e.g., sequential program model → C,C++, Java
- One language can capture variety of models
  - e.g., C++ → sequential program model, object-oriented model, state machine model
- Certain languages better at capturing certain computation models

Model of Computation

- Model of Computation (MoC) is a formal representation of the operational semantics of networks of functional blocks describing the computation
- Concurrent semantics used in different MoCs
  - Differential equations (multiple equations→ concurrency)
  - Spatial/temporal models
  - Discrete time (difference equations)
  - Discrete-event systems (DE)
  - State based models
  - Process networks (Kahn)
  - Communicating sequential processes with rendezvous (CSP)
  - Dataflow
  - Synchronous-reactive (S/R) models
  - Globally Asynchronous Locally Synchronous (GALS)

State-Oriented Models

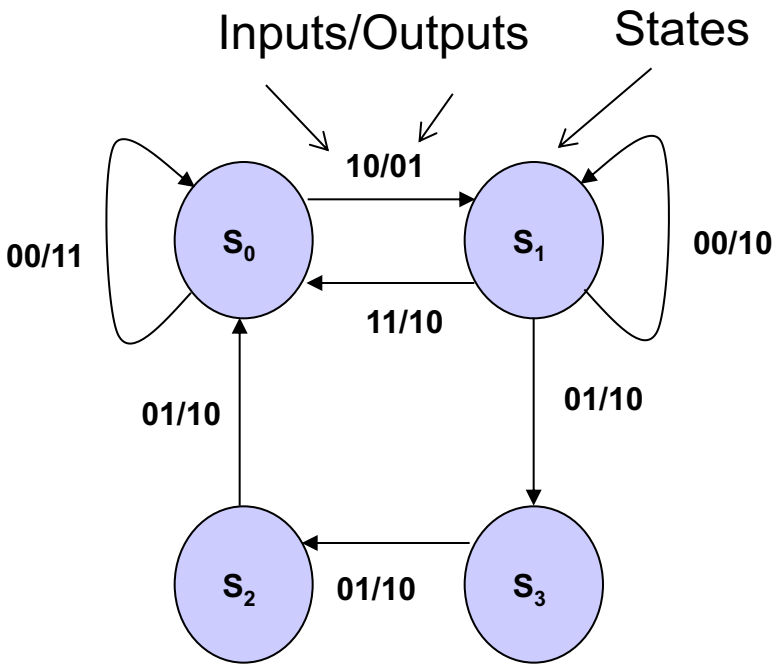
- Function of the system described by a set of states and a set of transitions between them
- State transitions react on external events and transfer system from one state to another
- Suitable for control-dominated applications such as reactive systems
- State-oriented models:
  - Finite-State Machines (FSM)
  - Hierarchical Concurrent FSMs (HCFSM)
  - Petri nets

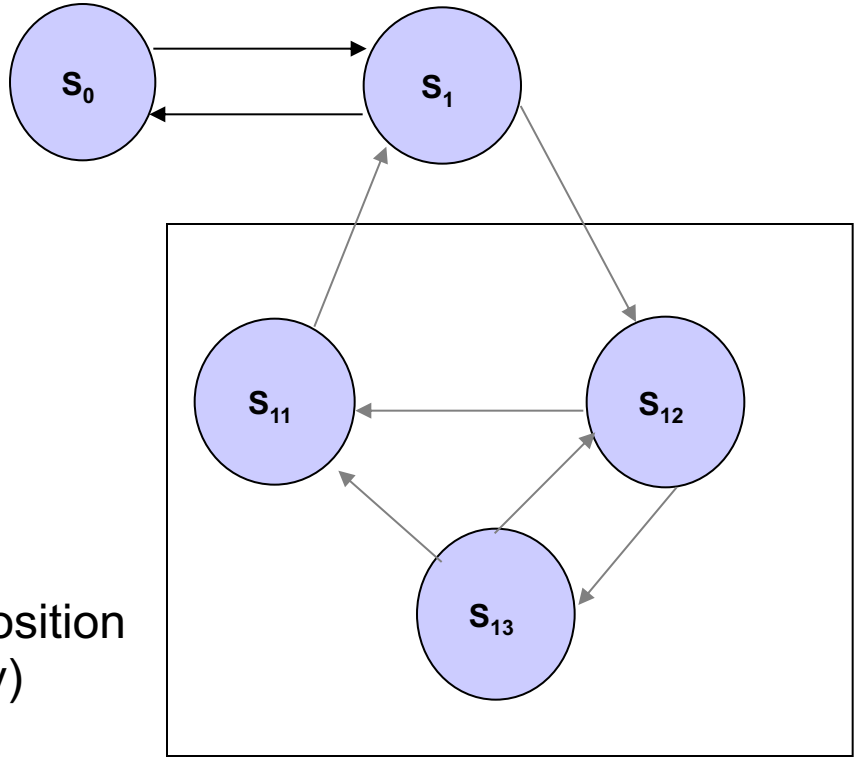
State-Oriented Models – FSM and HFSM

- FSM consists of a set of states, a set of transitions connecting states and a set of actions, with an initial state
- Two well known types:
  - Mealy (actions associated with transitions) transition-based
  - Moore (actions associated with states) state-based

- Well suited for control-dominated problems
- Lack of concurrency and hierarchy → not suitable for complex systems
- Hierarchical FSM (HFSM) – states can be decomposed into a set of sub-states; sub-states can be concurrent communicating via global variables
- Extended FSM (EFSM), Co-design FSM (CFSM), Hierarchical Concurrent FSM (HCFSM) suitable for HW/SW Co-design

State-Oriented Models – FSM and HFSM

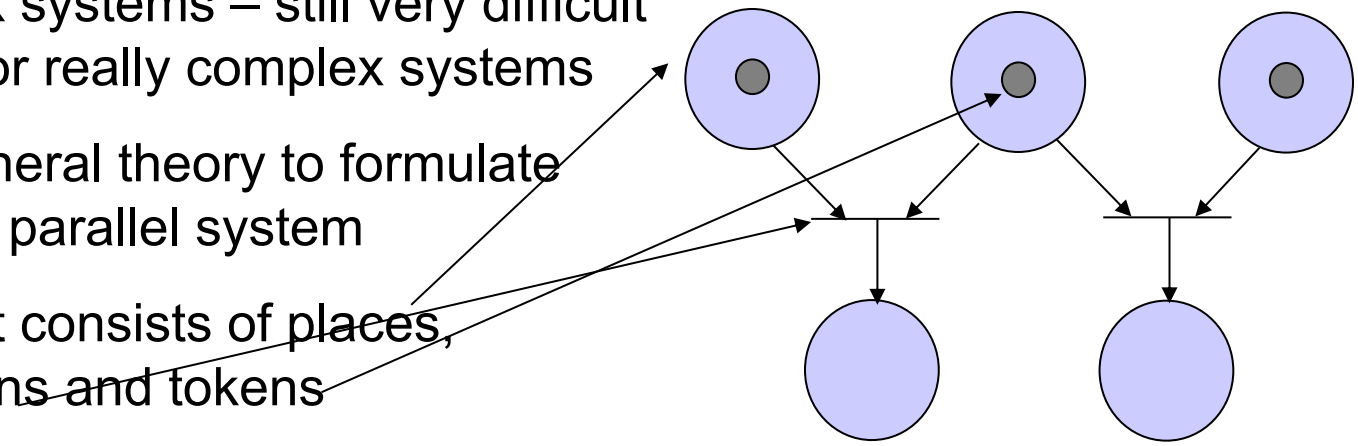




State decomposition (hierarchy)

State-Oriented Models – Petri Nets

- Graphical language for modeling complex systems – still very difficult to use for really complex systems
- First general theory to formulate discrete parallel system
- Petri net consists of places, transitions and tokens
- Tokens are stored in places and consumed and produced whenever a transition fires
- Suitable to model and analyze concurrent systems

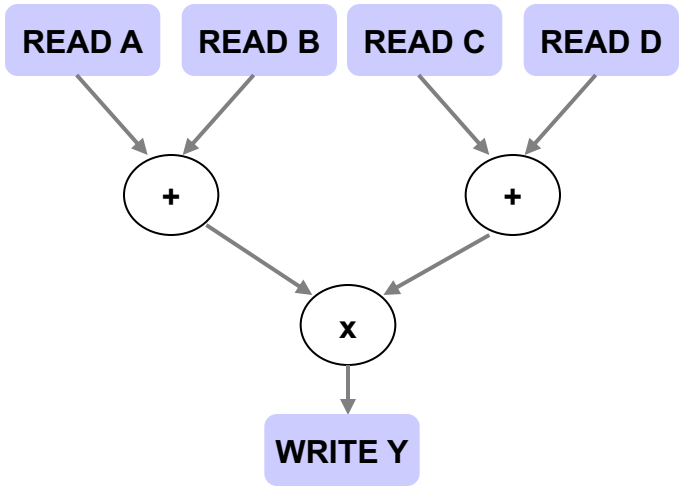


Activity-Oriented Models of Computation

- Specify the system functionality by a set of activities related by data or control dependencies
- There are no internal states
- Suitable for data-dominated applications
- Examples:
  - Data flow graphs (DFG)
  - Control flow graphs (CFG)

Activity-Oriented Models of Computation

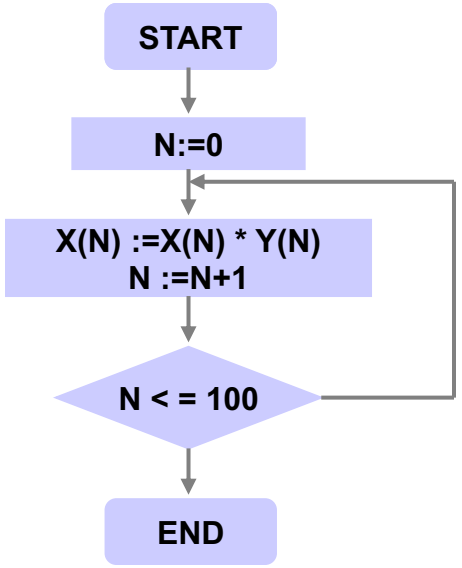
Data flow graphs (DFG)



Directed graph with:

- Vertices/nodes (inputs, outputs, storage and operations)
- Edges (dependences between nodes)

Control flow graphs (CFG)

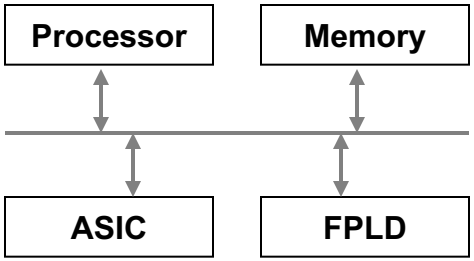


Directed graph with:

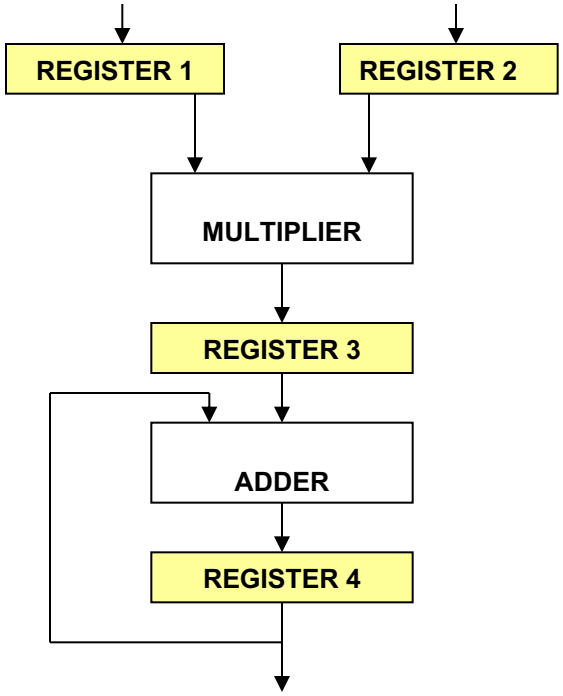
- Vertices/nodes (basic processing blocks and decision nodes)
- Edges (control flow between nodes)

Structural Models

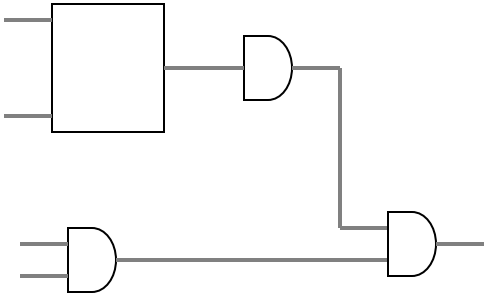
- They describe the physical modules and interconnections of the system



Block diagram



Register Transfer Netlist



Gate Netlist

Requirements on Specification/Design Languages

Embedded systems specification languages requirements include:

- Hierarchy of functional modules - simplifies top-down decomposition and bottom-up integration, can be structural or behavioral
- Concurrency – to describe concurrent operations and parallelisms
- State transitions – to model internal systems states; important for control-dominated systems – can be modeled by programming constructs
- Exceptions – interaction with environment – current state terminated and transition into another state occurs (e.g. reset and interrupt)
- Communication and synchronization – (for exchange of data between concurrently operating parts e.g. shared memory, message passing)
- Program instructions – necessary to simplify specifications (data types, decision constructs, assignments etc)
- Time – to describe timing constraints (response time for example) and functional times

Requirements on Specification/Design Languages

- Embedded system behaviour analysis at specification time:
  - Verifiability and formal analysis
  - Model executability (some models are only for documentation!)

Embedded Systems Design Languages

- Each embedded system has a diverse set of tasks to perform
- Tasks can be described with different models of computation that are built-in into different languages
- No language includes all those models (more domain-specific languages have evolved)
- Languages have evolved that are easier to write, analyse, and compile
- For example, a language for signal-processing is often more convenient for a particular problem than assembly, but might be poor for control-dominated behavior
- Categorisation of languages:
  - Hardware design - description - languages
  - Software design (programming) languages
  - Dataflow languages
  - Hybrid languages (which combine features of the above ones)

Hardware Description - Design - Languages

- VHDL and Verilog are the best known examples (standards)
- Both languages provide modeling tools for
  - Hierarchy
  - Concurrency
  - Behavioral and structural models
  - Module communication facilities
  - Multiple-valued logic
  - Support simulation and synthesis
  - Gate-level modeling
- VHDL strongly-typed, supports user-defined types, interfaces, implementation, local variables

Programming Languages

- 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 two numbers)
- 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 Languages - Comparison

	C	C++	Java
Expressions	xxx	xxx	xxx
Control-flow	xxx	xxx	xxx
Recursion	xxx	xxx	xxx
Exceptions	x	xxx	xxx
Classes & Inheritance		xxx	xxx
Templates		xxx	xx
Namespaces		xxx	xxx
Multiple Inheritance		xxx	x
Threads & Locks			xxx
Garbage collection		x	xxx

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.

Programming Language C++

- 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

Real-Time Operating Systems – Programming Language Extension

- Many embedded systems use a real-time operating system (RTOS) to simulate concurrency on a single processor or on multiprocessor or multi-core platforms
- An RTOS manages multiple running processes, each written in sequential language
- The processes perform the system's computation and the RTOS schedules them — attempts to meet deadlines by deciding which process runs when.

Real-Time Operating Systems – Programming Language Extension

- Most RTOSs use fixed-priority preemptive scheduling in which each process is given a particular priority (a small integer) when the system is designed
- At any time, the RTOS runs the highest-priority running process, which is expected to run for a short period of time before suspending itself to wait for more data
- Priorities are usually assigned using rate-monotonic analysis, which assigns higher priorities to processes that must meet more frequent deadlines.

Hybrid (System-Level) Languages

	Esterel	Polis	SystemC	SystemJ (UoA)
Concurrency	xxx	xxx	xxx	xxx
Hierarchy	xxx	xxx	xxx	xxx
Preemption	xxx		xxx	xxx
Determinism	xxx		x	xx
Synchr.comm.	xxx		xxx	xxx
Buffered comm.		xxx	xxx	xxx
FIFO comm.			x	xxx
Procedural descriptions	xxx	x	xxx	xxx
FSMs	xxx	xxx	x	xxx
Data-flow (DF)		xxx	xxx	xxx
Multi-rate DF				xxx
Software implem.	xxx	xxx	xxx	xxx
Hardware implem.	xxx	xxx	xxx	x

Esterel

- Intended for specifying control-dominated reactive systems
- Combines the control constructs of an imperative software language with
  - concurrency
  - preemption and
  - synchronous model of time like in synchronous digital circuits

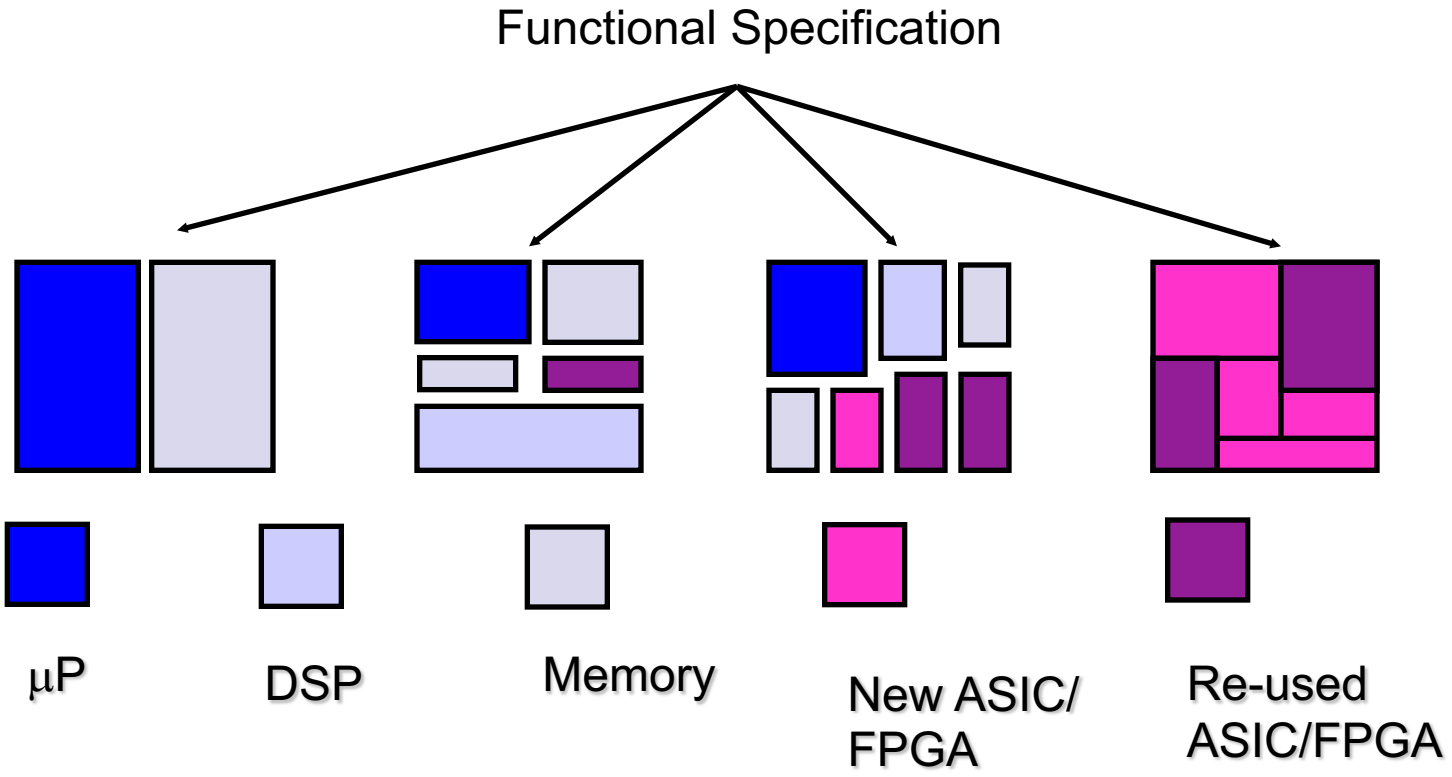
- In each clock cycle, the program awakens, reads its inputs, produces outputs, and suspends
- An Esterel program communicates through signals that are either present or absent each cycle - in each cycle, each signal is absent unless an emit statement for the signal runs and makes the signal present for that cycle only
- Esterel guarantees determinism by requiring each emitter of a signal to run before any statement that tests the signal

SystemC

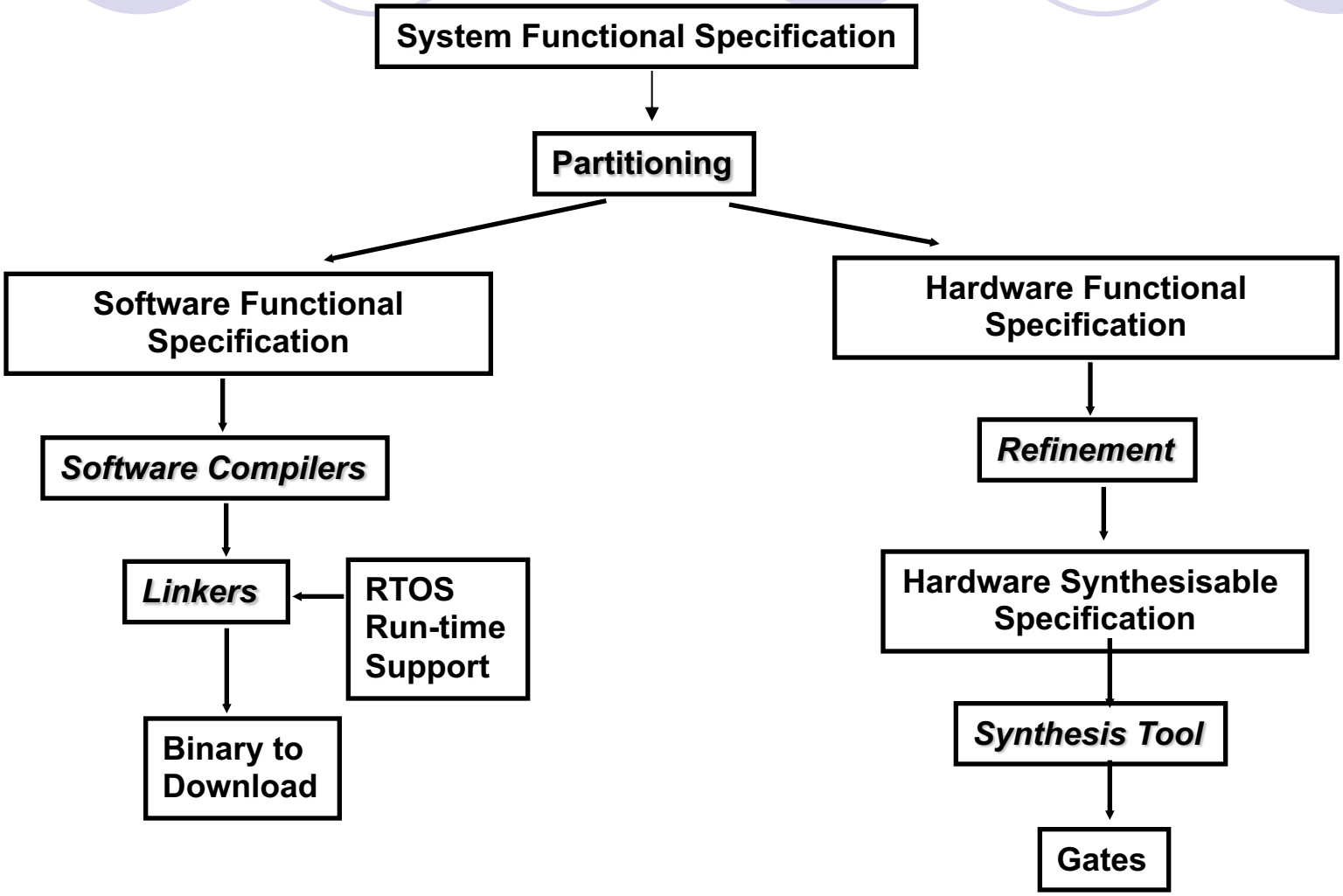
- The SystemC language is a C++ subset for specifying and simulating synchronous digital hardware
- A SystemC specification can be simulated by compiling it with a standard C++ compiler and linking in freely-distributed class libraries (from www.systemc.org)
- The SystemC language builds systems from 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
- 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

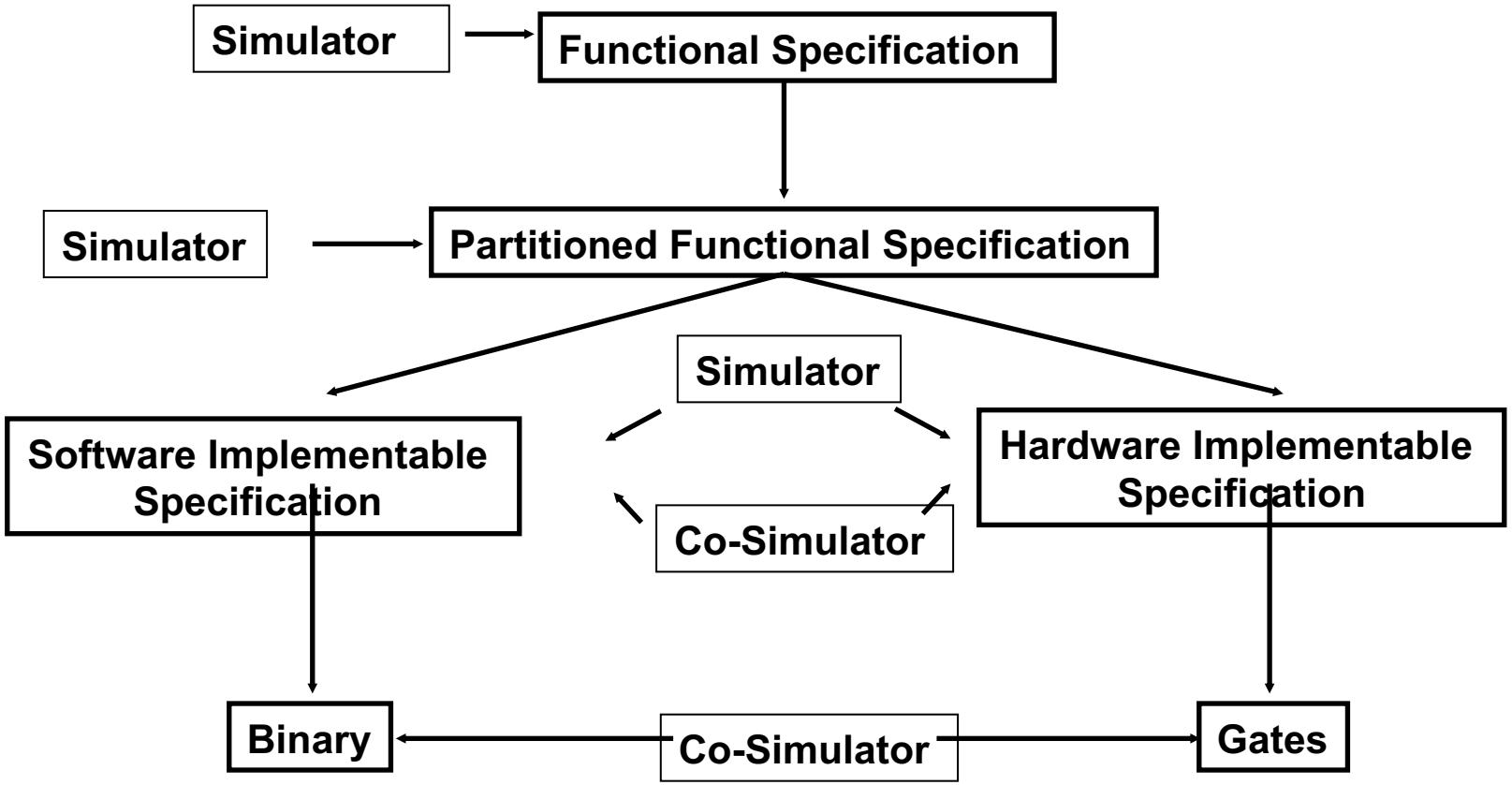
- One language for the specification of entire system
- Specification and implementation from the same language for both hardware and software



From SystemC to Implementation



From System C to Verification



System-level Programming and Design Language: SystemJ

SystemJ Approach to System-Level Design

- Developed at UoA – Embedded Systems Lab
- New system-level language
- SystemJ = Java + Esterel + CSP
- Enables multiclock designs in GALS type model of computation
- Combination of locally synchronous designs using global asynchrony
- Synchronous designs are reactive - S/R model of computation (MoC)
- Asynchrony implemented using channels with fully-locked handshaking (rendezvous)

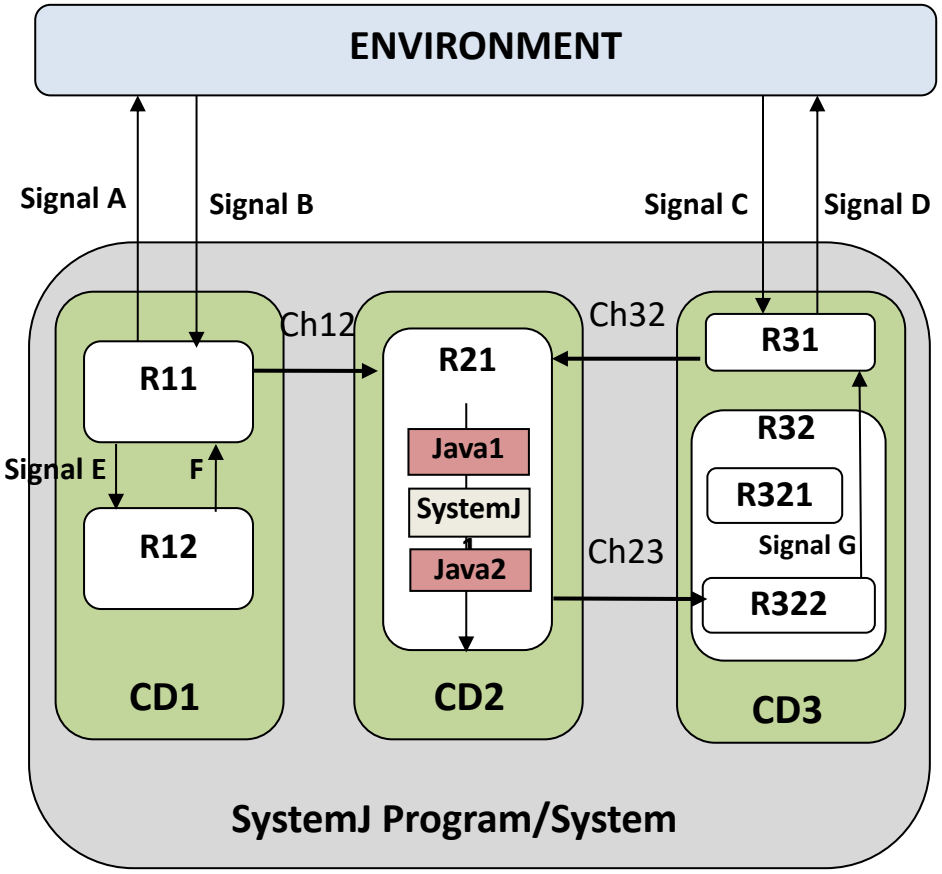
SystemJ Approach to System-Level Design

- Separation of computation from communication
- GALS formal model of computation (MoC)
- Behavioural hierarchy
- Explicit control of time (“logical” time, ticks)
- Basic behavioural entities, reactions, synchronise with the “local” clock
- Support for exceptions and exception handling
- Mix of data-dominated and control-dominated processing
- Support for formal verification
- Language has full constructive operational semantics
- Suitable for (hard) real-time systems on selected platforms

SystemJ Program

- Basic programming unit is called reaction
- Program composed of a number of concurrent reactions
- Import declarations for pure Java computations
- Named and unnamed reactions
- "System" level reaction which initialises the various global defined named and unnamed reactions
- Most of synchronous constructs borrowed from Esterel
- Has a set of synchronous and asynchronous kernel statements from which all other statements can be derived
- Uses delayed semantics (internally emitted signals visible in the next tick of time)

SystemJ Program – A Graphical Representation



- Clock-domains (CDs)
  - Top-level asynchronous processes
- Reactions
  - Light-weight synchronous threads spawned within clock-domains
- Communication object
  - Signals (Synchronous broadcast)
  - Channels (message passing)

SystemJ Kernel Statements

Statements	Description
`p;q`	p and q in sequence
`pause`	Consumes a logical instant of time (defines a tick boundary)
`[input|output] [type] signal ID`	Signal declaration; type is any valid Java type
`emit ID[(exp)]`	Emitting a signal with a possible value
`while(true) p`	Temporal loop; loop takes a t least one logical tick
`present(ID) p else q`	If signal ID is present enter p else q
`[weak] abort ([immediate] ID) p do q`	Preempt if signal ID is present
`[weak] suspend ([immediate] ID) p`	Suspend for one tick if signal ID is present
`trap(ID) p; do q`	Software exception; preempt if exit(ID) is executed in p
`exit(ID)`	Throw an exception with a token ID
`p || q`	Run p and q in lock-step parallel (synchronously)
`[input|output] type channel ID`	Channel declaration
`send ID(exp)`	Send data exp over the channel ID
`receive ID`	Receive data on channel ID
`#ID`	Retrieve data from a valued signal ID or channel ID; typically assigned to a Java variable
`CD_name (input|output)-> p`	Defines p as a clock domain with name CD_name and specified interface signal and channel ports (input, output)

SystemJ Derived Statements
(syntactic sugar)

Statement	Expansion
`await([immediate] S)`	`abort([immediate] S){while(true)pause;}`
`sustain(S)`	`while(true){emit S; pause;}`

`abort p`
`case (S1) do q1`
`case (S2) do q2`
	`abort(S1){abort(S2){ … p … } do q2 } do q1`

`halt`	`while(true) pause`
`loop`	`while(true) p`

`waitl (NUMBER (ms | s))`	`trap(T){`
	`long start = getTimeMs();`
	`while(true){`
	`if(getTimeMs() – start > NUMBER)`
	`exit(T);`
	`pause;`
	`}`
	`}`

SystemJ Program Example: Simple Example – Complex Control

Original syntax for asynchronous clock domains

```
system
    Interface{ input int channel Ch;
         output int channel Ch;
         input signal s;}
{
   {  signal A;
      {
         await(s);
         emit A;
      }
      ||
      {
         await(A);
         while(true){
            send Ch(1);
            pause;
      }
   }
   ><
   {
      receive Ch;
      int i = #Ch;
      i = i+1;
      System.out.println(i);
   }
}
```

Interface declaration
Inputs, outputs, channels

Synchronous reaction 1

Clock domain 1

Synchronous reaction 2

Clock domain 2

Java data call

SystemJ Program Example: Simple Example – Complex Control

New syntax

```
CD1→ {output int channel Ch;
      input signal s}
      {
         signal A;
         await(s);
         emit A;
      }
      ||
      {
         await(A);
         while(true){
            send Ch(1);
            pause;
         }
      }
CD2→ {input int channel Ch}
     {
      receive Ch;
      int i = #Ch;
      System.out.println(i);
   }
```

Interface declaration CD1

Reaction 1

Clock domain 1

Reaction 2

Interface declaration CD2

Clock domain 2

More on SystemJ towards the end of Part 1 lectures