

EE382V-ICS: System-on-a-Chip (SoC) Design

Lecture 7 – Models of Computation

Sources:
Prof. Margarida Jacome, UT Austin

Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu



Lecture 7: Outline

- **Introduction**
 - Embedded systems
 - Design process
- **HW/SW co-design**
 - Specification and modeling
 - Models of Computation
- **Summary**

Embedded Systems

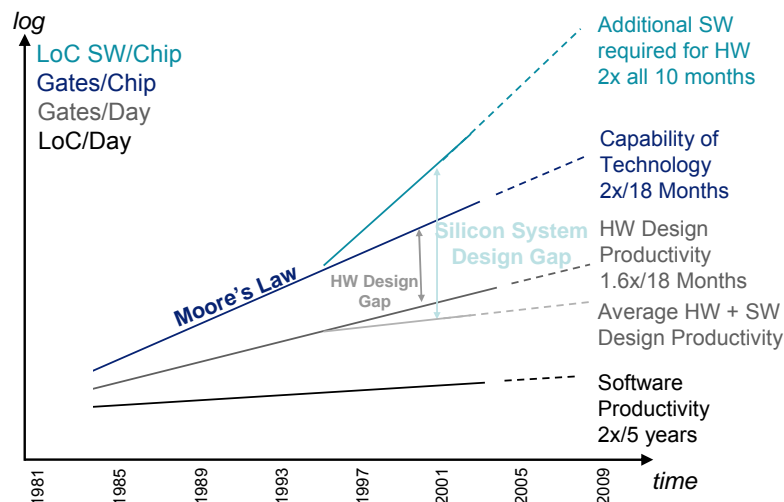
- **Ubiquitous computing**
 - Signal processing systems
 - Radar, sonar, real-time video, set-top boxes, DVD players, medical equipment
 - Mission critical systems
 - Avionics, space-craft control, nuclear plant control
 - Distributed control
 - Network routers, mass transit systems, elevators in large buildings, sensors
 - “Small” systems
 - Cellphones, appliances, toys, MP3 players, PDAs, digital cameras, smart badges
- **Part of a larger system - masquerading as non-computers**
 - Not a “computer with keyboard, display, etc.”
- **Application-specific – not general-purpose**
 - Application is known a priori
- **Reactive – not transformative / interactive**
 - Interact (sense, manipulate, communicate) with the external world
 - Never terminate (ideally)
- **Constrained**
 - Timing: latency, throughput (real time)
 - Power, size, weight, heat, reliability, etc.
 - Increasingly high-performance (DSP) & networked

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

3

Productivity Gaps



Source: W. Ecker, W. Müller, R. Dömer, *Hardware-dependent Software - Principles and Practice*, Springer 2009.

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

4

Embedded System Design

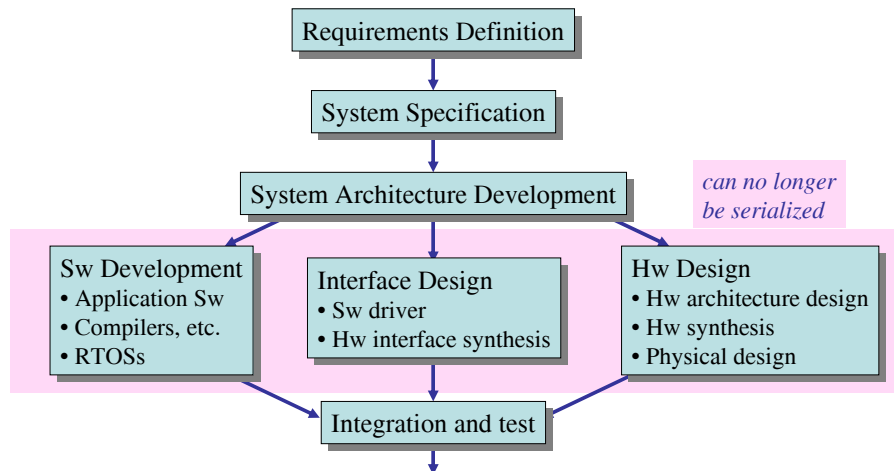
- **Computer-Aided Design (CAD)**
Electronic Design Automation (EDA)
 - Tools take care of HW fairly well (at least in relative terms)
 - Productivity gap emerging
- **Situation in SW is worse**
 - HLLs such as C help, but can't cope with exponential increase in complexity and performance constraints

Holy Grail for Tools People: HW-like synthesis & verification from a behavior description of the whole system at a high level of abstraction using formal computation models

Desirable Design Methodology

- **Design should be based on the use of one or more *formal models* to describe the *behavior* of the system at a high level of abstraction**
 - Such behavior should be captured on an unbiased way, that is, before a decision on its decomposition into hardware and software components is taken
- **The final implementation of the system should be generated as much as possible using *automatic synthesis* from this high level of abstraction**
 - To ensure implementations that are "correct by construction"
- **Validation (through *simulation* or *verification*) should be done as much as possible at the higher levels of abstraction**

Embedded System Design Process



EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

7

Lecture 7: Outline

✓ Introduction

✓ Embedded systems

- Design process

• HW/SW co-design

- Specification and modeling
- Models of Computation

• Summary

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

8

Embedded System Design

- **The design of an embedded system consists of correctly implementing a specific set of *functions* while satisfying constraints on**
 - Performance
 - Dollar cost
 - Energy consumption, power dissipation
 - Weight, etc.

The choice of a *system architecture* impacts whether designers will implement a *function* as custom hardware or as (embedded) software running on a programmable component (processor).

Design Problems

- **Design a heterogeneous multiprocessor architecture that satisfies the design requirements.**
 - Use computational unit(s) dedicated to some functions
 - Processing elements (PE): hardwired logic, CPU
 - Program the system
- **A significant part of the design problem is deciding which parts should be in SW on programmable processors, and which in specialized HW**
 - Deciding the HW/SW architecture
- **Ad-hoc approaches today**
 - Based on earlier experience with similar products
 - HW/SW partitioning decided a priori, designed separately

HW/SW Co-Design

- **Concurrent design & joint optimization of mixed HW/SW systems**
 - Specification
 - Modeling
 - Performance analysis
 - Synthesis
 - HW/SW partitioning (resource allocation & binding)
 - Scheduling
 - HW & SW implementation
 - SW compilation
 - HW synthesis
 - Validation
 - Integration, verification & debugging

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

11

Reliability and Safety

- **Embedded systems often are used in life critical situations, where reliability and safety are more important criteria than performance**
 - Today, embedded systems are designed using a somewhat ad hoc approach that is heavily based on earlier experience with similar products and on manual design
- **Formal verification and automated synthesis are the surest ways to guarantee safety**
 - Both, formal verification and synthesis from high levels of abstraction have been demonstrated only for small, specialized languages with restricted semantics
 - Insufficient, given the *complexity* and *heterogeneity* found in typical embedded systems

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

12

Heterogeneity, Complexity

- **Managing complexity and heterogeneity challenge**
 - Mix of hardware design with software design
 - Mixes design styles within each of these categories
 - Mix of abstraction/detail/specificity
- **Different specification and modeling techniques**
 - Rigorous and unambiguous
- **Formal models for analysis and synthesis are key**
 - It requires reconciling
 - Simplicity of modeling required by verification and synthesis
 - Complexity and heterogeneity of real world design



Key need ⇒ understanding which formal models are more appropriate to capture/express the various types of behavior at different abstraction levels, and how those diverse formal models interact.

Lecture 7: Outline

- ✓ **Introduction**
 - ✓ Embedded systems
 - ✓ Design process
- **HW/SW co-design**
 - Specification and modeling
 - Models of Computation
- **Summary**

Formal Model of a Design

- **Most tools and designers describe the behavior of a design as a relation between a set of inputs and a set of outputs**
 - This relation may be informal, even expressed in natural language
 - Such informal, ambiguous specifications may result in unnecessary redesigns...
- **A formal model of a design should consist of the following components:**
 - Functional specification
 - Set of properties
 - Set of performance indices
 - Set of constraints on performance indices

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

15

Formal Model of a Design (2)

- **A functional specification, given as a set of explicit or implicit relations which involve inputs, outputs and possibly internal (state) information**
Fully characterizes the operation of a system
- **A set of properties that the design must satisfy**
Redundant: in a properly constructed system, the functional specification satisfies these properties. Yet properties are simpler / more abstract compared to the functional specification.
- **A set of performance indices that evaluate the quality of the design in terms of cost, reliability, speed, size, etc.**
- **A set of constraints on performance indices, specified as a set of inequalities**
Bound the cost of a system

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

16

Properties


- **A property is an assertion about the behavior, rather than a description of the behavior**
 - It is an abstraction of the behavior along a particular axis
- **Examples:**
 - *Liveness* property: when designing a network protocol, one may require that the design never *deadlocks*
 - *Fairness* property: when designing a network protocol, one may require that *any request will eventually be satisfied*

The above properties do not completely specify the behavior of the protocol, they are instead properties we require the protocol to have

Properties & Models

- **Properties can be classified in three groups:**
 1. Properties that are *inherent* to the model (i.e., that can be shown formally to hold for *all specifications* described using that model)
 2. Properties that can be verified *syntactically* for a given specification (i.e., that can be shown to hold with a simple, usually polynomial-time analysis of the specification)
 3. Properties that must be verified *semantically* for a given specification (i.e., that can be shown to hold by executing, at least implicitly, the specification for all inputs that can occur)

Model Validation

- **By construction**
 - property is inherent
 - **By verification**
 - property is provable syntactically
 - **By simulation**
 - check behavior for all inputs
 - **By intuition**
 - property is true, I just know it is...
- 
- better be higher
in this list...*

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

19

Model of Computation (MoC)

- **A MoC is a framework in which to express what actions must be taken to complete a computation**
 - Objects and their relationships
- **MoCs need to**
 - Be *powerful/expressive enough* for the application domain
 - Have appropriate *synthesis* and *validation* semantics
- **Why different models?**
 - Different models \Rightarrow different properties
 - Turing complete models are too powerful!
 - Imperative programming models are poor match
 - Reactive instead of transformation systems

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

20

Consider a Simple Example

“The Observer pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated automatically.”

Eric Gamman Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addison-Wesley, 1995

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

21

Example: Observer Pattern in Java

```
public void addListener(listener) {...}

public void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue)
    }
}
```

Will this work in a multithreaded context?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

22

Observer Pattern with Mutexes

```
public synchronized void addListener(listener)
{...}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue)
    }
}
```

JavaSoft recommends against this.
What's wrong with it?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

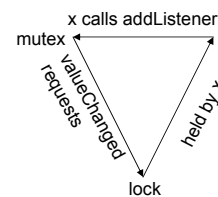
23

Mutexes using Monitors are Minefields

```
public synchronized void addListener(listener)
{...}

public synchronized void setValue(newvalue) {
    myvalue=newvalue;
    for (int i=0; i<mylisteners.length; i++) {
        myListeners[i].valueChanged(newvalue)
    }
}
```

- **valueChanged()** may attempt to acquire a lock on some other object and stall.
- If the holder of that lock calls **addListener()**: **deadlock!**



Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

24

Observer Pattern Gets Complicated

```
public synchronized void addListener(listener) {...}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

while holding lock, make a copy of
listeners to avoid race conditions

notify each listener outside of the
synchronized block to avoid deadlock

This still isn't right. What's wrong with it?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

25

How to Make it Right?

```
public synchronized void addListener(listener) {...}

public void setValue(newValue) {
    synchronized (this) {
        myValue=newValue;
        listeners=myListeners.clone();
    }
    for (int i=0; i<listeners.length; i++) {
        listeners[i].valueChanged(newValue)
    }
}
```

Suppose two threads call setValue(). One of them will set the value last, leaving that value in the object, but listeners may be notified in the opposite order. The listeners may be alerted to the value-changes in the wrong order!

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

26

Problems with Thread-Based Concurrency

- ***Nontrivial software written with threads, semaphores, and mutexes is incomprehensible to humans***

- Nondeterministic, best effort
 - Explicitly prune away nondeterminism
- Poor match for embedded systems
 - Lack of timing abstraction
- Termination in reactive systems
 - Composability?



- Search for non-thread-based models: which are the requirements for appropriate specification techniques?

Source: Ed Lee, UC Berkeley, Artemis Conference, Graz, 2007

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

27

MoCs for Reactive Systems

- **Consider essential aspects of reactive systems:**

- Time/synchronization
- Concurrency
- Heterogeneity

- **Classify models based on**

- How to specify behavior
- How to specify communication
- Implementability
- Composability
- Availability of tools for validation and synthesis

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

28

Main MoCs for Reactive Systems

- **Programming models**
 - Imperative & declarative
 - Synchronous/reactive
- **Process-based models**
 - Discrete event
 - Kahn Process Networks (KPNs)
 - (Synchronous) Dataflow models ((S)DF)
- **State-based models**
 - Finite State Machines (FSM)
 - Hierarchical, Concurrent State Machines (HCFSM)
 - Petri Nets

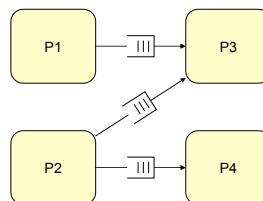
EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

29

Kahn Process Network (KPN) [Kahn74]

- **C-like processes communicating via FIFO channels**
 - Unbounded, uni-directional, point-to-point queues
 - Sender (`send()`) never blocks
 - Receiver (`wait()`) blocks until data available



- **Deterministic**
 - Behavior does not depend on scheduling strategy
 - Focus on causality, not order (implementation independent)

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

30

Kahn Process Network (KPN) (2)

- **Determinism**
 - Process can't peek into channels and can only wait on one channel at a time
 - Output data produced by a process does not depend on the order of its inputs
 - Terminates on global deadlock: all process blocked on `wait()`
- **Formal mathematical representation**
 - Process = continuous function mapping input to output streams
- **Turing-complete, undecidable (in finite time)**
 - Terminates?
 - Can run in bounded buffers (memory)?

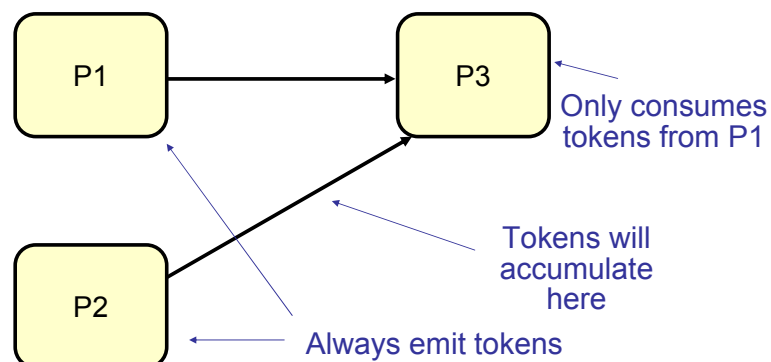
EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

31

KPN Scheduling

- **Scheduling determines memory requirements**
- **Data-driven scheduling**
 - Run processes whenever they are ready:



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

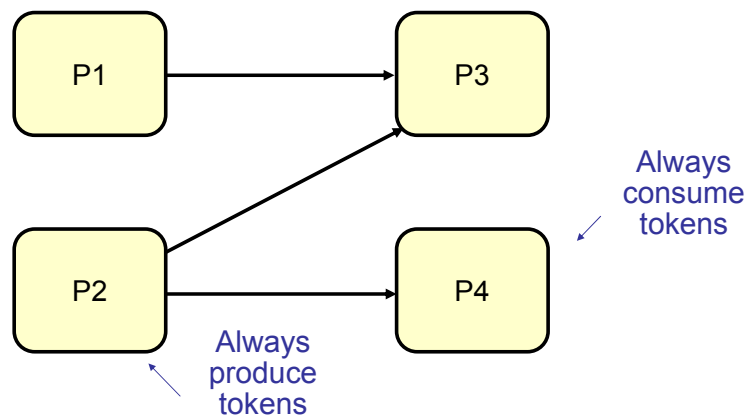
EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

32

Demand-Driven Scheduling

- Only run a process whose outputs are being solicited
 - Synchronous, unbuffered message-passing
- However...



Source: S. Edwards. "Languages for Digital Embedded Systems", Kluwer 2000.

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

33

KPN Scheduling

- Inherent tradeoffs
 - Completeness
 - Run processes as long as they are ready
 - Might require unbounded memory
 - Boundedness
 - Block senders when reaching buffer limits
 - Potentially incomplete, artificial deadlocks and early termination
 - Data driven: completeness over boundedness
 - Demand driven: boundedness over completeness and even non-termination
- Hybrid approach [Parks95]
 - Start with smallest bounded buffers
 - Schedule with blocking `send()` until (artificial) deadlock
 - Increase size of smallest blocked buffer and continue

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

34

Kahn Process Networks (KPN)

- **Difficult to implement**
 - Size of infinite FIFOs in limited physical memory?
 - Dynamic memory allocation, dependent on schedule
 - Boundedness vs. completeness vs. non-termination (deadlocks)
 - Dynamic context switching
- **Parks' algorithm**
 - Non-terminating over bounded over complete execution
 - Does not find every complete, bounded schedule
 - Does not guarantee minimum memory usage
 - Deadlock detection?

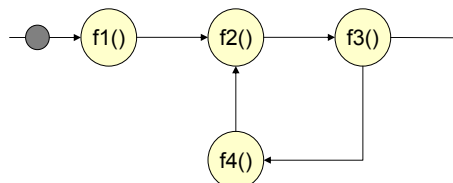
EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

35

Dataflow

- **Breaking processes down into network of actors**
 - Atomic blocks of computation, executed when *firing*
 - Fire when required number of input *tokens* are available
 - Consume required number of tokens on input(s)
 - Produce number of tokens on output(s)
 - Separate computation & communication/synchronization
 - Actors (indivisible units of computation) may fire simultaneously, any order
 - Tokens (units of communication) can carry arbitrary pieces of data
- **Directed graph of infinite FIFO arcs between actors**
 - Boundedness, completeness, non-termination?



- **Signal-processing applications**

EE382V-ICS: SoC Design, Lecture 7

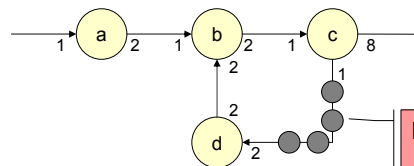
© 2010 A. Gerstlauer

36

Synchronous Dataflow (SDF) [Lee86]

- **Fixed number of tokens per firing**

- Consume fixed number of inputs
- Produce fixed number of outputs



Initialization
➤ Delay
➤ Prevent deadlock

- **Can be scheduled statically**

- Flow of data through system does not depend on values

- **Find a repeated sequence of firings**

- Run actors in proportion to their rates
- Fixed buffer sizes, no under- or over-flow

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

37

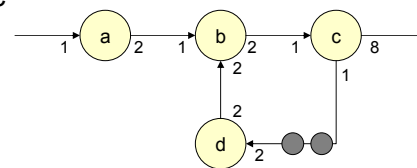
SDF Scheduling (1)

- **Solve system of linear rate equations**

- Balance equations per arc

- $2a = b$
- $2b = c$
- $b = d$
- $2d = c$

➤ $4a = 2b = c = 2d$



- Inconsistent systems

- Only solvable by setting rates to zero
- Would otherwise (if scheduled dynamically) accumulate tokens

- Underconstrained systems

- Disjoint, independent parts of a design

- **Compute repetitions vector**

- Linear-time depth-first graph traversal algorithm

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

38

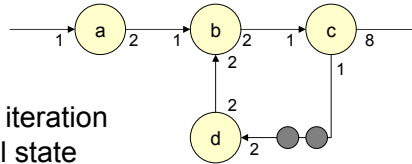
SDF Scheduling (2)

- Periodically schedule actors in proportion to their rates

- Smallest integer solution

- $4a = 2b = c = 2d$

- $\rightarrow a = 1, b = 2, c = 4, d = 2$



- Symbolically simulate one iteration of graph until back to initial state

- Insert initialization tokens to avoid deadlock

- $\rightarrow adbccdbcc = a(2db(2c))$

- $\rightarrow a(2db)(4c)$

- Schedule determines memory requirements

- $\rightarrow a(2db(2c))$: 2 token slots on each arc for total of 8 token buffers

- $\rightarrow a(2db)(4c)$: extra initialization tokens, 12 token buffers

- Single appearance schedule to reduce code size

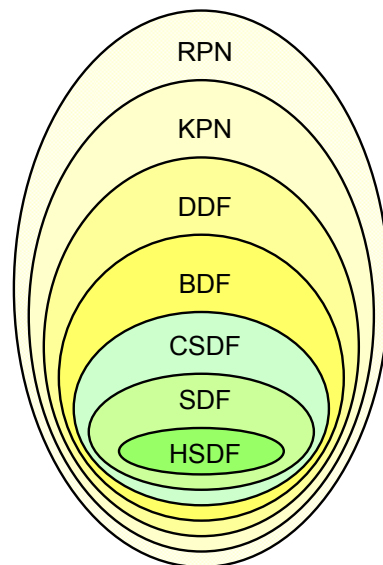
- Looped code generation and compiler optimizations

EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

39

Process-Based MoCs



Yellow: Turing complete

RPN	Reactive Process Network
KPN	Kahn Process Network
DDF	Dynamic Dataflow
BDF	Boolean Dataflow
CSDF	Cyclo-Static Dataflow
SDF	Synchronous Dataflow
HSDF	Homogeneous SDF

Source: T. Basten, MoCC 2008.

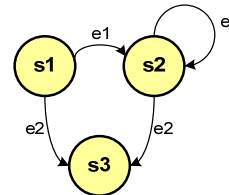
EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

40

State-Based Models

- **Status and reactivity (control flow)**
 - **Explicit enumeration of computational states**
 - State represents captured history
 - **Explicit flow of control**
 - Transitions in reaction to events
- **Stepwise operation of a machine**
 - Cycle-by-cycle hardware behavior
 - Finite number of states
 - Not Turing complete
- **State-oriented imperative representation**
 - State only implicit in control/data flow (CDFG)
- **Formal analysis**
 - Reachability, equivalence, ...



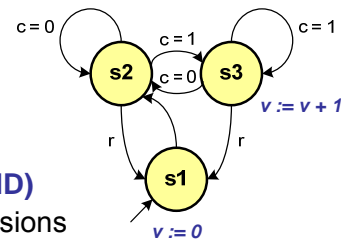
EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

41

Finite State Machines

- **Finite State Machine (FSM)**
 - Basic model for describing control and automata
 - Sequential circuits
 - States S , inputs/outputs I/O , and state transitions
 - FSM: $\langle S, I, O, f, h \rangle$
 - Next state function $f: S \times I \rightarrow S$
 - Non-deterministic: f is multi-valued
 - Output function h
 - Mealy-type (input-based), $h: S \times I \rightarrow O$
 - Moore-type (state-based), $h: S \rightarrow O$
 - Convert Mealy to Moore by splitting states per output
- **Finite State Machine with Data (FSMD)**
 - Computation as control and expressions
 - Controller and datapath of RTL processors
 - FSM plus variables V
 - FSMD: $\langle S, I, O, V, f, h \rangle$
 - Next state function $f: S \times V \times I \rightarrow S \times V$
 - Output function $h: S \times V \times I \rightarrow O$



EE382V-ICS: SoC Design, Lecture 7

© 2010 A. Gerstlauer

42

Hierarchical & Concurrent State Machines

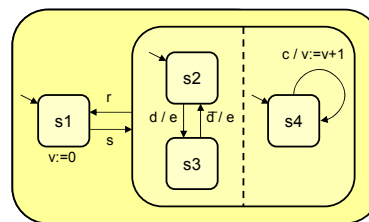
- **Superstate FSM with Data (SFSMD)**

- Hierarchy to organize and reduce complexity
 - Superstates that contain complete state machines each
 - Enter into one and exit from any substate

- **Hierarchical Concurrent FSM (HCFSM)**

- Hierarchical and parallel state composition
 - Lock-step concurrent composition and execution
- Communication through global variables, signals and events
 - Synchronous: zero time

- Graphical notation
[StateCharts]



Lecture 7: Outline

- ✓ **Introduction**

- ✓ Embedded systems
- ✓ Design process

- ✓ **HW/SW co-design**

- ✓ Specification and modeling
- ✓ Models of Computation

- **Summary**

Design Process

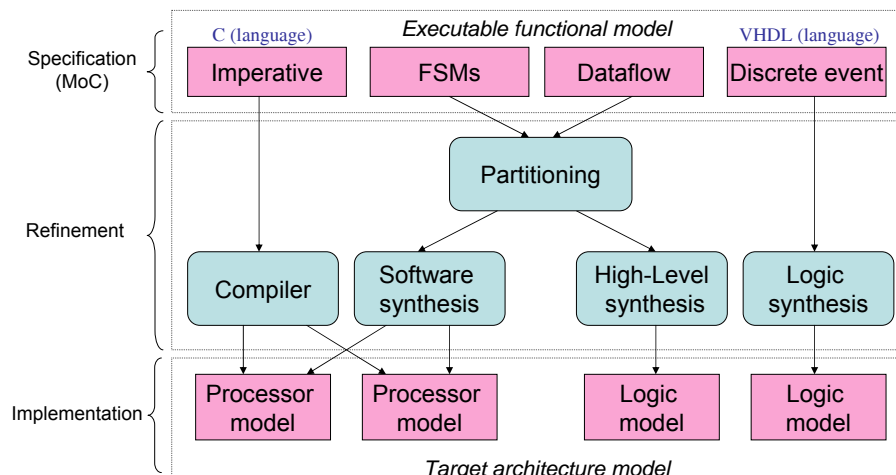
- **Sequence of steps that transforms a set of requirements described informally into a detailed description that can be used for manufacturing**
 - Intermediate steps with transformation from a more abstract description to a more detailed one (*refinement*)
- **A designer can perform step-by-step refinement**
 - The “input” description is a *specification*
 - The final description of the design is an *implementation*
- **Take a model of the design at a level of abstraction and refine it to a lower one (level of detail ↑).**
 - Ensure that the **properties at the lower level of abstraction are verified**, and that the **performance indices are satisfactory**
 - Thus, refinement process involves **mapping constraints, performance indices and properties to the lower level**, so that they can be computed for the next level down

EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

45

Design Synthesis



EE382V-ICS: SoC Design, Lecture 7

© Margarida Jacome, UT Austin

46