Modeling Embedded Systems

Tajana Simunic Rosing
Department of Computer Science and Engineering
University of California, San Diego.
Models, Languages and Tools

State machine
Sequent. program
Data-flow
Concurrent processes

Verilog
C/C++
Java
VHDL

Implementation A
Implementation B
Implementation C
Components of a formal design model

- **Functional specification**
  - Relationships between inputs, outputs & states

- **Properties**
  - Relations between I/O/S that can be checked against the functional specification
  - 3 types: *inherent* in model of computation, those that can be verified *syntactically & semantically* for a given specification

- **Performance indices**
  - Evaluate quality of design

- **Constraints**
  - On performance indices, usually inequalities
Design process

**Design:**
- A set of components interacting with each other and with the environment that is not a part of design

**Model of Computation (MOC):**
- Defines the behavior and interaction of the design blocks

**Design process:**
- Takes a model at a higher level of abstraction and refines it to a lower level along with mapping constraints, performance indices and properties to the same level

**Validation:**
- Process of checking if design is correct
  - Simulation/emulation, formal verification of specification or implementation

**Synthesis:**
- Design refinement where more abstract specifications are translated into less abstract specifications
Models of Computation Elements

• **State**
  – e.g. in HW:
    • Combinational states: one state for a given time t
    • Sequential states: multiple states possible for time t

• **Decidability**
  – Can a property be determined in a finite amount of time?

• **Concurrency and communication**
  – Embedded systems usually have coordinated concurrent processes -> communication required
## Modes of Communication

<table>
<thead>
<tr>
<th></th>
<th>Transmitters</th>
<th>Receivers</th>
<th>Buffer Size</th>
<th>Blocking Reads</th>
<th>Blocking Writes</th>
<th>Single Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsynchronized</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Read-Modify-write</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Unbounded FIFO</td>
<td>one</td>
<td>one</td>
<td>unbounded</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Bounded FIFO</td>
<td>one</td>
<td>one</td>
<td>bounded</td>
<td>no</td>
<td>maybe</td>
<td>yes</td>
</tr>
<tr>
<td>Single Rendezvous</td>
<td>one</td>
<td>one</td>
<td>one</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multiple Rendezvous</td>
<td>many</td>
<td>many</td>
<td>one</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Communication

- Message Passing
  - Non-blocking
  - Blocking

- Extended rendezvous

- Shared memory

```
process a {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```

```
process b {
  ..
  P(S)  //obtain lock
  ..    // critical section
  V(S)  //release lock
}
```
# Models of computation comparison

<table>
<thead>
<tr>
<th>Communication/local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Synchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>StateCharts</td>
<td>SDL</td>
</tr>
<tr>
<td>Data flow</td>
<td>(Not useful)</td>
<td>Kahn networks, DFG, SDF</td>
</tr>
<tr>
<td>Petri nets</td>
<td></td>
<td>various versions of Petri Nets</td>
</tr>
<tr>
<td>Discrete event (DE) models</td>
<td>VHDL*, Verilog*, SystemC*, …</td>
<td>Only experimental systems, e.g. distributed DE in Ptolemy</td>
</tr>
<tr>
<td>Von Neumann models⁺</td>
<td>C, C++, Java</td>
<td>C, C++, Java &amp; libraries, ADA</td>
</tr>
</tbody>
</table>

*Classification based on implementation with centralized data structures

⁺Consists of a processing unit, control unit and storage for instructions & data
Model of Computation Examples

- **State machine models**
  - FSMs, StateCharts, SDL
- **Petri nets**
- **Communicating processes**
  - Kahn processes, Communicating Sequential Processes
- **Ada**
- **Dataflow models**
  - DFG, SDFG
- **Discrete event models**
  - VHDL, Verilog, SystemC, SpecC
- **Synchronous languages**
  - Cycle based models, Esterel, Lustre
Classical automata

- **Moore-automata:**
  \[ O = H(S); \quad S^+ = f(I, S) \]

- **Mealy-automata**
  \[ O = H(I, S); \quad S^+ = f(I, S) \]
Finite-state machines (FSMs)

**Elevator Control process using a state machine**

```
GoingUp
req > floor
!(req > floor)
req > floor
!(timer < 10)
req < floor
DoorOpen
timer < 10
u,d,o,t = 0,0,1,0
req == floor
GoingDn
req < floor
!(req<floor)
req < floor
u,d,o,t = 0,1,0,0
DoorOpen
u,d,o,t = 0,0,1,1

u is up, d is down, o is open
t is timer_start
```

u, d, o, t = 1,0,0,0
u, d, o, t = 0,0,1,0
u, d, o, t = 0,1,0,0
• FireMode
  – When fire is true, move elevator to 1st floor and open door
Models of Computation

• **State Machine Models**
  – FSM, StateCharts, CFM, SDL

• **Petri nets**

• **Communicating Processes**
  – Kahn processes, Communicating Sequential Processes

• **Ada**

• **Dataflow models**
  – DFG, SDFG

• **Discrete Event Systems**
  – VHDL, Verilog, SystemC, SpecC

• **Synchronous languages**
  – Cycle based models, Esterel, Lustre
StateCharts: Hierarchy
Back to Elevator Example

UnitControl

RequestResolver

ElevatorController

NormalMode

FireMode

!fire

fire

idle

GoingUp

req>floor

u,d,o = 1,0,0

!(req>floor)

timeout(10)

Idle

req<floor

req==floor

DoorOpen

u,d,o = 0,0,1

!(req>floor)

FireGoingDn

FireDrOpen

floor>1

floor==1

u,d,o = 0,1,0

u,d,o = 0,0,1

fire

!fire

req<floor

GoingDn

req<floor

req>floor

req>floor
StateCharts: Default state
StateCharts: History
History & default state

same meaning
StateCharts: Concurrency
Conditional Transitions
StateCharts: Timers

20 ms

timeout
Example: Answering machine
StateCharts: edge labels

- **Events:**
  - Exist only until the next evaluation of the model
  - Can be either internally or externally generated

- **Conditions:**
  - Refer to values of variables that keep their value until they are reassigned

- **Reactions:**
  - Can either be assignments for variables
  - or creation of events

- **Example:**
  - `service-off [not in Lproc] / service:=0`
Propagations and Broadcasts

Source: B. P. Douglass & iLogix
StateCharts: Simulation

Status = values of all variables + set of events + current time
Step = execution of the three phases

Three phases:
1. Effects of external changes on events and conditions are evaluated,
2. The set of transitions to be made in the current step and right hand sides of assignments are computed,
3. Transitions become effective, variables obtain new values.
StateCharts Simulation Example

```
swap

\[
\begin{align*}
  e/a & \rightarrow = b \\
  e/b & \rightarrow = a
\end{align*}
\]

/a := 1; b := 0
```
Statecharts – Example 1

Statechart Example

Equivalent FSM Representation
Statecharts – Example 2

Equivalent FSM Representation
StateCharts to FSM
StateCharts: Application Examples

• Power converter system for trams, metros & trains
  – System-level modeling and automated code generation
  – Guarantee latencies less than 10 microseconds
  – Cut development time by 50%
  – Time from design completion to first prototype down to 1hr from 3 months
  – Defect free code automatically generated for a number of RTOS implementations
StateCharts: Application Examples

- Development of defibrillator and pacemaker technology
  - Model system behavior before requirements are finalized
  - Check that SW provides mathematically consistent representation of the product’s system – correct and unambiguous representation of model behavior
  - More accurate and extensive verification – guarantee device works 100% of the time for at least 7-10yrs
  - Cut product verification costs by 20%
  - 15-20% overall cost reduction per projects on future development
StateCharts: Application Examples

• Jet engine electronic controller design
  – System level specification and consistency check
  – Construction of on-screen simulation of the cockpit display
  – Evaluate correctness in normal and faulty operation modes
  – Project so successful that now StateCharts are used for:
    • Independent overspeed protection system
    • Thrust reverse control system
    • Engine fuel control system

By BMW
StateCharts: Summary

- Hierarchy
- AND- and OR-super states
- Default state, History
- Timing behavior
- State oriented behavior
- Edge labels
- Concurrency
- Synchronization & communication  
  - Broadcast, shared memory
- Simulation
- Cross compiling
Models of Computation

• **State Machine Models**
  – FSM, StateCharts, SDL

• Petri nets

• **Communicating Processes**
  – Kahn processes, Communicating Sequential Processes

• Ada

• Dataflow models
  – DFG, SDFG

• **Discrete Event Systems**
  – VHDL, Verilog, SystemC, SpecC

• **Synchronous languages**
  – Cycle based models, Esterel, Lustre
Specification & Description Language (SDL)

• Designed for specification of distributed systems.
  • Dates back to early 70s, formal semantics defined in the late 80s, updates from 1984 to 1999 by International Telecommunication Union (ITU)
  • Provides textual and graphical formats
• Similar to StateCharts, each FSM is called a *process*, but it uses message passing instead of shared memory for communication
• Supports operations on data
SDL-representation of FSMs/processes

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>B</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>C</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>D</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>E</td>
<td>f</td>
<td>v</td>
</tr>
</tbody>
</table>

Process P1
Communication among SDL-FSMs

- Communication between FSMs (or “processes“) is based on **message-passing**, assuming a **potentially indefinitely large FIFO-queue**.

  - Each process fetches next entry from FIFO,
  - checks if input enables transition,
  - if yes: transition takes place,
  - if no: input is ignored (exception: SAVE-mechanism).
Determinate?

• Let tokens be arriving at FIFO at the same time:
  Order in which they are stored is unknown:

All orders are legal so simulators can show different behaviors for the same input, all of which are correct.
Operations on data

- Variables can be declared locally for processes.
- Their type can be predefined or defined in SDL itself.
- SDL supports abstract data types (ADTs)

```
DCL
Counter Integer;
Date String;

Counter := Counter + 3;
```

```
(1:10)  (11:30)  ELSE
```
Process interaction diagrams

- Interaction between processes can be described in process interaction diagrams, which are a special case of block diagrams.
- In addition to processes, these diagrams contain channels and declarations of local signals.
Hierarchy in SDL

- Process interaction diagrams can be included in **blocks**. The root block is called **system**.

Processes cannot contain other processes, unlike in StateCharts.
Timers

- Timers can be declared locally
- Elapsed timers put signal into queue, but are not necessarily processed immediately; RESET removes a timer from FIFO-queue
Description of network protocols
Vending machine example

Machine sells pretzels, chips, cookies, & doughnuts

Accepts nickels, dime, quarters, and half-dollars

It is not a distributed application.

Overall view of vending machine

System VendingMachine

Ccoins
[nickel, dime, quarter, half]

CoinInterface
[add]
Ccointctrl
[reject_further_coins, accept_coins]

Crequest
[pur_pretzel, pur_chip, pur_cookie, pur_doughnut, reload_pretzel, reload_chip, reload_cookie, reload_doughnut]

Creject
[reject_coin]

DecodeRequests
[amount_entered]
Cemptystate
[pretzel_empty, chip_empty, cookie_empty, doughnut_empty]

CspitPurchased
[spit_pretzel, spit_chip, spit_cookie, spit_doughnut]

CexaktDisplay
[exact_only]

CspitChange
[spit_nickel, spit_dime]

CHANGEINTERFACE

SIGNAL
[dime, nickel, quarter, half, pur_pretzel, pur_cookie, pur_doughnut, pur_chip, add(int), spit_change(int), amount_entered(int), reject_further_coins, exact_only, accept_coins, reject_coins, spit_dime, spit_nickel, pretzel_empty, spit_pretzel, chip_empty, spit_chip, cookie_empty, spit_cookie, doughnut_empty, spit_doughnut, reload_pretzel, reload_chip, reload_cookie, reload_doughnut]

SYNTAX items=INTEGER
CONSTANTS 0:7
ENDSYNTAX items;

SYNTAX int=INTEGER
CONSTANTS 0:127
ENDSYNTAX int;
Decode Requests

CONNECT Cadd AND Radd;
CONNECT Ccoincrl AND Rcointctrl;
CONNECT Cchange AND Rchange;
CONNECT CAmountDisplay AND RamountDisplay;
CONNECT Crequest AND Rpretzel,Rchip,Rcookie, Rdoughnut;
CONNECT CemptyDisplay AND Rpretzel_e,Rchip_e, Rcookie_e,Rdoughnut_e;
CONNECT CspitPurchased AND Rpretzel_s, Rchip_s,Rcookie_s,Rdoughnut_s;

SYNONYM PRETZEL int=50
SYNONYM PCHIP int=15;
SYNONYM PCOOKIE int=55;
SYNONYM PDOUGHNUT int=60;
SYNONYM PMAX int=60;
SYNONYM NITEMS items=7;

SIGNAL sub(int);
Process ChipHandler

DCL nchip items:=NITEMS;
VIEWED current int;

pur_wait

pur_chip

VIEW(current) >= PCHIP

no

pur_wait

yes

sub(PCHIP)

nchip:= nchip-1;

spit_chip

nchip=0

yes

no

pur_wait

chip_empty

empty

reload_chip

nchip:=NITEMS

pur_wait
SDL: Real World Examples

• FlexRay RTIO protocol specification
• ADSL design
  – Ideal language for telecom design; communication between components and their different states of operation can be easily modeled with SDL
  – Object orientation and automatic code generation significantly simplified system design verification
  – Early testing done on SDL – significant savings in the cost of expensive test equipment
SDL summary

- FSM model for the components,
- Non-blocking message passing for communication,
- Implementation requires bound for the maximum length of FIFOs; may be very difficult to compute,
- Not necessarily determinate
- Timer concept adequate just for soft deadlines,
- Limited way of using hierarchies,
- Limited programming language support,
- No description of non-functional properties,
- Excellent for distributed applications (used for ISDN),
- Commercial tools available (see http://www.sdl-forum.org)
Models of Computation

• State Machine Models
  – FSM, StateCharts, SDL, CFSM
• Petri nets
• Communicating Processes
  – Kahn processes, Communicating Sequential Processes
• Ada
• Dataflow models
  – DFG, SDFG
• Discrete Event Systems
  – VHDL, Verilog, SystemC, SpecC
• Synchronous languages
  – Cycle based models, Esterel, Lustre
Petri net definitions

A Petri net is a 5-tuple, \( PN = (P, T, F, W, M_0) \) where:

- \( P = \{ p_1, p_2, \ldots, p_m \} \) is a finite set of places,
- \( T = \{ t_1, t_2, \ldots, t_n \} \) is a finite set of transitions,
- \( F \subseteq (P \times T) \cup (T \times P) \) is a set of arcs (flow relation),
- \( W: F \rightarrow \{1, 2, 3, \ldots\} \) is a weight function,
- \( M_0: P \rightarrow \{0, 1, 2, 3, \ldots\} \) is the initial marking,
- \( P \cap T = \emptyset \) and \( P \cup T \neq \emptyset \).

A Petri net structure \( N = (P, T, F, W) \) without any specific initial marking is denoted by \( N \).

A Petri net with the given initial marking is denoted by \( (N, M_0) \).

<table>
<thead>
<tr>
<th>Input Places</th>
<th>Transition</th>
<th>Output Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>Event</td>
<td>Postconditions</td>
</tr>
<tr>
<td>Input data</td>
<td>Computation step</td>
<td>Output data</td>
</tr>
<tr>
<td>Input signals</td>
<td>Signal processor</td>
<td>Output signals</td>
</tr>
<tr>
<td>Resources needed</td>
<td>Task or job</td>
<td>Resources released</td>
</tr>
<tr>
<td>Conditions</td>
<td>Clause in logic</td>
<td>Conclusion(s)</td>
</tr>
<tr>
<td>Buffers</td>
<td>Processor</td>
<td>Buffers</td>
</tr>
</tbody>
</table>

- \( \text{H}_2\text{O} \text{Example} \)
Petri net – Train tracks model

train wanting to go right

train going to the right

track available

train going to the left
Concurrency, Causality, Choice
Concurrency, Causality, Choice
Concurrency, Causality, Choice

Causality, sequencing
Concurrency, Causality, Choice
Concurrency, Causality, Choice
Petri nets: confusion 😊

- Concurrency & conflict lead to confusion....
  - Symmetric
  - Asymmetric
Conflict for resource „track“

- Train wanting to go right
- Train going to the right
- Track available
- Train going to the left
Communication Protocol
Communication Protocol
Communication Protocol
Communication Protocol
Communication Protocol
Communication Protocol

![Communication Protocol Diagram](image-url)
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Producer-Consumer Problem
Modeling synchronization control when sharing resources

- Multiprocessor CPUs, distributed systems
A Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where:

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places,
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions,
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation),
- $W: F \rightarrow \{1, 2, 3, \ldots\}$ is a weight function,
- $M_0: P \rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking,
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted by $N$.

A Petri net with the given initial marking is denoted by $(N, M_0)$.

---

### Input Places | Transition | Output Places
---|---|---
Preconditions | Event | Postconditions
Input data | Computation step | Output data
Input signals | Signal processor | Output signals
Resources needed | Task or job | Resources released
Conditions | Clause in logic | Conclusion(s)
Buffers | Processor | Buffers

---

- $H_2O$
Petri Net Properties

• Behavioral
  – Reachability
    • Marking M reachable from marking Mo
  – k - Boundedness
    • Number of tokens in each place does not exceed finite number k
    • Safe if 1-bounded
  – Liveness
    • A transition is live if it can never deadlock.
    • A transition is deadlocked if it can never fire.

• Structural
  – Controlability
    • Any marking can be reached from any other marking
  – Structural boundedness
  – Conservativeness – weighted sum of tokens constant
PN Properties - Reachability

\[ M_0 = (1,0,1,0) \]
\[ M = (1,1,0,0) \]
PN Properties – Deadlock-free
PN Properties – Deadlock-free
Petri Net Liveness

• **L0-live (dead):** a particular transition can never fire
  – In the figure below, $t_0$ can never fire
    • see reachability graph
Petri Net Liveness

• **L1-live**: a particular transition can fire at least once for some firing sequence
  – In the figure below, \( t_1 \) can only fire once, for some firing sequence
Petri Net Liveness

• **L2-live**: a particular transition can fire \( k \) times for a particular firing sequence, for any \( k \).
  
  – In the figure below, \( t_2 \) can only fire once, twice, etc, for different firing sequences.
Petri Net Liveness

- **L3-live**: a particular transition can fire infinitely in a particular firing sequence.
  - In the figure below, $t_3$ can fire infinitely for the firing sequence $t_3, t_3, t_3, t_3, \ldots$
  - Note that the number of times $t_1$ and $t_2$, fire is finite for any firing sequence.
PN Properties - Boundedness
PN Properties - Boundedness
PN Properties - Boundedness

Unbounded
PN Properties - Boundedness

Unbounded
PN Properties - Boundedness
PN Properties - Conservation
PN Properties - Conservation

Not conservative
PN Properties - Conservation
Petri Nets - Analysis

• Structural
  – Incidence matrix

• State Space Analysis techniques
  – Coverability Tree
  – Reachability Graph
PN Properties – Analysis

Incident Matrix

State Equations
Petri Nets – Coverability Tree
Example

• Assume
  – e=6
  – Mo=[0 12]

• Can we reach M=[6 0] from Mo?
• Can it be statically scheduled?
• What size buffers are needed at P1 & P2?
Consider the Petri net defined by:

\[ P = \{ p_1, p_2, p_3 \} \]
\[ T = \{ t_1, t_2, t_3 \} \]
\[ A = \{ (p_1 t_1) (p_1 t_3) (p_2 t_1) (p_2 t_2) (p_3 t_3) (t_1 p_2) (t_1 p_3) (t_2 p_3) (t_3 p_1) (t_3 p_2) \} \]

With all weights being one except \( w(p_1 t_1) = 2 \)
Urn Game with Petri Nets

Draw a Petri net that describes a game in which there is an urn with black and red balls. In each round the player takes two balls:

- If both are black, then the player returns one black
- If both are red, then the player returns one red
- If one is red and one is black, then the player returns one red.
Petri nets: Applications

• Model, simulate and analyze networking protocols (e.g. TCP, Ethernet, etc)
• Model, simulate and analyze complex network elements (e.g. router, switch, optical mux); check their logical behavior
• Design and analyze network performance and AoS with logical models of traffic generators, protocols and network elements
• Study network behavior characteristics (e.g. throughput, blocking probability etc)
Petri nets in practice

- Example apps:
  - TCP performance
  - Security system design and automated code generation
  - MAC design
  - ....
Petri Nets - Summary

• PN Graph
  – places (buffers), transitions (action), tokens (data)

• Firing rule
  – Transition enabled if enough tokens in a place

• Properties
  – Structural (consistency, structural boundedness)
  – Behavioral (reachability, boundedness, etc.)

• Analysis techniques

• Applications
  – Modeling of resources, mutual exclusion, synchronization
Models of Computation

- State Machine Models
  - FSM, StateCharts, SDL, CFSM
- Petri nets
- Communicating Processes
  - Communicating Sequential Processes, Ada, Kahn processes
- Dataflow models
  - DFG, SDFG
- Discrete Event Systems
  - VHDL, Verilog, SystemC, SpecC
- Synchronous languages
  - Cycle based models, Esterel, Lustre
- HW Models
- Unified Modeling Language (UML)
Process

• A sequential program
  – Executes concurrently with other processes

• Basic operations on processes
  – Create & terminate, suspend & resume, join

• Process communication
  – Shared memory (and mutexes)
  – Message passing
  – Rendezvous
task screen_out is
entry call_ch(val:character; x, y: integer);
entry call_int(z, x, y: integer);
end screen_out;
task body screen_out is
...
select
  accept call_ch ... do ..
end call_ch;
or
  accept call_int ... do ..
end call_int;
end select;

Sending a message:
begin
  screen_out.call_ch('Z',10,20);
exception
  when tasking_error =>
    (exception handling)
end;
Task graphs

Sequence constraint

Nodes are a „program“ described in some programming language
Task graphs - Timing

Arrival time \rightarrow \text{deadline}

(0,7] \rightarrow T_1

(1,8] \rightarrow T_2

(3,10] \rightarrow T_3
Task graphs - I/O
Task graphs - Shared resources
Task graphs - Periodic schedules

.. infinite task graphs
Task graphs - Hierarchy
Your job is to design a process $P$ with the sub tasks:

$Q_1(x,y) = (u,v) = (2x, x+y)$
$Q_2(x,y) = 2x + y$
$Q_3(x,y) = x + 2y$

Assume you have an adder (2s, 1W) and a shifter (1s, 0.5W) to implement $P$. Show the fastest possible schedule, and get the total energy consumption for $P$. Do your results change if battery can supply maximum 1W at any point in time?
Models of Computation

- State Machine Models
  - FSM, StateCharts, SDL, CFMS
- Petri nets
- Communicating Processes
  - Communicating Sequential Processes, Ada
- **Dataflow models**
  - Kahn processes
  - DFG, SDFG
- Discrete Event Systems
  - VHDL, Verilog, SystemC, SpecC
- Synchronous languages
  - Cycle based models, Esterel, Lustre
- Petri nets
- HW Models
- Unified Modeling Language (UML)
Kahn process network

KPN - executable task graphs
Communication is via infinitely large FIFOs
Nonblocking write, blocking read => DETERMINATE

Important properties of KPN:
Continuous
Output signals can be gradually produced; never have to consume all input to produce some output
Monotonic
Output depends on input but doesn’t change previous output

Continuous monotonic processes => ITERATIVE
Petri net model of a Kahn process

- KPNs are deterministic:
  - Output determined by
    - Process, network, initial tokens
Dataflow Process Networks

• Dataflow:
  – Maps input tokens to output tokens
  – Outputs function of current inputs
    • No need to keep state on suspend/resume
  – Scheduling for resource sharing

\[ Z = (A + B) \times (C - D) \]
Dataflow process examples

- HW resource scheduling
  - Constraints: 2 mult, 2 ALU
  - List scheduler
  - Additional constraints
    - Performance
    - Power
    - Area etc.
Synchronous dataflow
• Nodes have rates of data production or consumption.

• Edges have delays.
  – Delays do not change rates, only the amount of data stored in the system at startup.
SDF examples
SDF Scheduling

- By building a set of “flow and conservation” equations

\[
\begin{align*}
3a - 2b &= 0 \\
4b - 3d &= 0 \\
b - 3c &= 0 \\
2c - a &= 0 \\
d - 2a &= 0
\end{align*}
\]

Solution: \(a = 2c; \ b = 3c; \ d = 4c\)

Possible schedules:
- BBBCDDDDDA
- BDBDBCADDA
- BBDBBDCCAA
- …
SDF problem
Dataflow process examples

- Design of a first 802.11b WLAN card
  - New product – combines RF, MAC and PHY; lots of DSP
  - Implemented on an ASIC
  - Previous design hand coded VHDL

- System level design with COSSAP by Synopsys
  - Explore architectural trade-offs – what in gates, what on DSP
  - Scheduling trade-offs: latency, area, propagation delay between clocks, and vendor library

- Results:
  - Reduced time to market by a factor of TWO!
  - Architectural exploration within 10% of actual measurements in terms of timing and area

by Symbol Tech.
Models of Computation

• State Machine Models
  – FSM, StateCharts, SDL, CFSM
• Petri nets
• Communicating Processes
  – Kahn processes, Communicating Sequential Processes
• Ada
• Dataflow models
  – DFG, SDFG
• Discrete Event Systems
  – VHDL, Verilog, SystemC, SpecC
• Synchronous languages
  – Cycle based models, Esterel, Lustre
• HW Models
• Unified Modeling Language (UML)
Discrete Events

• Notion of time is fundamental: global order
  – events are objects which carry ordered time info
  – there is a casual relationship between events
• DE simulator maintains global event queue
  – Verilog, VHDL
• Expensive - ordering tame stamps can be time consuming
• Large state & low activity => effective simulation
• Simultaneous events lead to non-determinacy
  – require complex conflict resolution schemes
  – e.g. delta delays
Simultaneous Events in the Discrete Event Model

B has 0 delay

B has delta delay

A → B → C

A → B → C

A → B → C

A → B → C

B has 0 delay

B has delta delay
VHDL: Entities

entity full_adder is
port(a, b, carry_in: in Bit; -- input ports
    sum, carry_out: out Bit); -- output ports
end full_adder;

Entity declaration

Architecture 1
Architecture 2
Architecture 3
....
**VHDL: Architectures**

**architecture** structure of full_adder **is**

- **component** half_adder
  - **port** (in1, in2: **in** Bit; carry, sum: **out** Bit);
  - **end** component;

- **component** or_gate
  - **port** (in1, in2: **in** Bit; o: **out** Bit);
  - **end** component;

- **signal** x, y, z: Bit; -- local signals

**begin** -- port map section

- i1: half_adder **port map** (a, b, x, y);
- i2: half_adder **port map** (y, carry_in, z, sum);
- i3: or_gate **port map** (x, z, carry_out);

**end** structure;

**architecture** behavior of full_adder **is**

**begin**

- sum <= (a xor b) xor carry_in after 10 Ns;
- carry_out <= (a and b) or (a and carry_in) or (b and carry_in) after 10 Ns;

**end** behavior;
VHDL processes & waits

Processes model HW parallelism

```
process
begin
  a <= b after 10 ns
end
```

Waits:

```
wait until signal list;
wait until a;
wait until condition;
  wait until c='1';
wait for duration;
  wait for 10 ns;
wait; suspend indefinitely
wait on = sensitivity list
```

```
process
begin
  prod <= x and y ;
  wait on x,y;
end process;

process (x, y)
begin
  prod <= x and y ;
end process;
```
VHDL Simulation

Start of simulation

Future values for signal drivers

Assign new values to signals

Evaluate processes

Activate all processes sensitive to signal changes
architecture one of RS_Flipflop is
begin
process: (R,S,Q,nQ)
begin
Q <= R nor nQ;
nQ <= S nor Q;
end process;
end one;

δ cycles reflect the fact that no real gate comes with zero delay.
VHDL Summary

- Entities and architectures
- Multiple-valued logic
- Modeling hardware parallelism by processes
  - Wait statements and sensitivity lists
- VHDL simulation cycle
  - $\delta$ cycles
SystemC: Motivation
SystemC: Methodology
Models of Computation

- State Machine Models
  - FSM, StateCharts, SDL, CFSM
- Petri nets
- Communicating Processes
  - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
  - DFG, SDFG
- Discrete Event Systems
  - VHDL, Verilog, SystemC, SpecC
- HW Models
- Synchronous languages
  - Cycle based models, Esterel, Lustre
- Unified Modeling Language (UML)
Levels of hardware modeling

1. System level
2. Algorithmic level
3. Instruction set level
4. Register-transfer level (RTL)
5. Gate-level models
6. Switch-level models
7. Circuit-level models
8. Device-level models
9. Layout models
10. Process and device models
## Instruction level

<table>
<thead>
<tr>
<th>Assembler (MIPS)</th>
<th>Simulated semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>or $1,$2,$3</code></td>
<td><code>Reg[1] := Reg[2] \lor Reg[3]</code></td>
</tr>
<tr>
<td><code>andi $1,$2,100</code></td>
<td><code>Reg[1] := Reg[2] \land 100</code></td>
</tr>
<tr>
<td><code>sll $1,$2,10</code></td>
<td><code>Reg[1] := Reg[2] \ll 10</code></td>
</tr>
<tr>
<td><code>srl $1,$2,10</code></td>
<td><code>Reg[1] := Reg[2] \gg 10</code></td>
</tr>
</tbody>
</table>
Register transfer level: MIPS
Gate-level model
Switch level model
Circuit level model

```
.SUBCKT NAND2 VDD VSS A B OUT
MN1 I1 A VSS VSS NFET W=8U L=4U AD=64P AS=64P
MN2 OUT B I1 VSS NFET W=8U L=4U AD=64P AS=64P
MP1 OUT A VDD VDD PFET W=16U L=4U AD=128P AS=128P
MP2 OUT B VDD VDD PFET W=16U L=4U AD=128P AS=128P
CA A VSS 50fF
CB B VSS 50fF
COUT OUT VSS 100fF
.ENDS
```
Device level

Measured and simulated currents

\[ I_D \quad [\text{mA}] \]

\[ -V_{DS} \]

\[ 0.5 \]

\[ 2 \quad 4 \quad 6 \]
Layout model
Process model

- Simulated
- Measured
Models of Computation

- State Machine Models
  - FSM, StateCharts, SDL, CFSM
- Petri nets
- Communicating Processes
  - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
  - DFG, SDFG
- Discrete Event Systems
  - VHDL, Verilog, SystemC, SpecC
- HW Models
- Synchronous reactive languages
  - Cycle based models, Esterel, Lustre
- Unified Modeling Language (UML)
Reactive Synchronous Languages

• Assumptions
  – Instantaneous reactions
  – Discrete events
  – Static

• Cycle based models
  – Excellent for (single) clocked synchronous circuits

• Control flow oriented (imperative) languages
  – Esterel

• Data flow languages
  – Lustre, Signal

• Deterministic behavior

• Simulation, software and hardware synthesis, verification
Lustre example
Reactive Synchronous Models: Esterel Statements

- Emit S
- Present S then p else q end
- Pause
- P; Q, P||Q
- Loop p end
- Await S
- Abort p when S
- Suspend p when S
- Sustain S = (loop emit S; pause end)
Abort Statement

```plaintext
abort
  pause;
  pause;
  emit A
when B;
emit C
```

- **Normal termination**
- **Aborted termination**
- **Aborted termination, Emit A preempted**
- **Normal termination**
  B not checked in the first cycle (like await)
Esterel Examples

emit A;
emit B;
pause;
loop
    present C then emit D end;
present E then emit F end;
pause;
end

C       C
 E
A       D       D
B       F       F
Esterel Examples

[ await A; emit C
  ||
  await B; emit D
];
emit E

A B
C D
E
Time in Esterel

- Global clock with precise control over when events appear
  - At every tick: read inputs, compute, output
- Statements
  - A bounded number in one cycle
  - Emit, present, loop
  - Take multiple cycles:
    - Pause, await, sustain
- Causality analysis
  - Deterministic & non-contradictory
Esterel Application Examples

- TI used Esterel to automatically synthesize full coverage tests for a safety-critical design
  - Showed functional coverage covered only 30% of the design, with Esterel 100% covered

- Airbus
  - Significant decrease in errors due to increase in automated code generation (40-70%)
    - e.g fly by wire controls & automatic flight control 70%, display computer 50%, warning & maintenance computer 40%
  - Major increase in productivity

<table>
<thead>
<tr>
<th>Computer Aided Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft</td>
</tr>
<tr>
<td>Number of digital units</td>
</tr>
<tr>
<td>Volume of on-board software in MB</td>
</tr>
<tr>
<td>Errors found per 100 KB</td>
</tr>
</tbody>
</table>
Esterel Summary

- Reactive synchronous language
- Control flow oriented
- Imperative syntax
- Synchrony assumption useful for safety critical embedded systems
  - Convert timing relations to causal ordering
  - Used in verification
    - e.g. TI, Airbus
Models of Computation

- State Machine Models
  - FSM, StateCharts, SDL, CFSM
- Petri nets
- Communicating Processes
  - Kahn processes, Communicating Sequential Processes
- Ada
- Dataflow models
  - DFG, SDFG
- Discrete Event Systems
  - VHDL, Verilog, SystemC, SpecC
- HW Models
- Synchronous reactive languages
  - Cycle based models, Esterel, Lustre
- Unified Modeling Language (UML)
UML (Unified modeling language)
UML - StateCharts
UML Summary

- UML
  - State machine diagram (StateChart-like)
  - Activity diagram (extended Petri nets)
  - Deployment diagram (exec. arch.)
  - Use case diagram
  - Package diagram (hierarchy)
  - Class diagrams
  - Timing diagrams (UML 2.0), UML for real-time
Models and Languages Summary

• Multiple models and languages are essential for high-level design
  – Managing complexity by abstraction
  – Formality ensures refinement correctness
  – Model choice depends on
    • Class of applications
    • Required operations (synthesis, scheduling, ...)

• Multiple MOCs can co-exist during all phases of design
  – Specification
  – Architectural mapping and simulation
  – Synthesis, code generation, scheduling
  – Detailed design and implementation
  – Co-simulation
Sources and References

- Alberto Sangiovanni-Vincentelli @ UCB
- Mani Srivastava @ UCLA
- Rajesh Gupta @ UCSD
- Nikil Dutt @ UCI