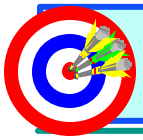


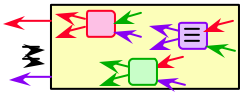
This Tutorial

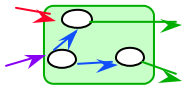
- ✓ Introduction
  - Embedded systems, characteristics, applications
- Hardware-Software Co-Design
  - Identify technologies important to co-design
    - What is involved in system design?
    - What are the steps, and where are the bottlenecks?
  - Indicate the state of the art
    - Existing concepts, established tools
    - Research ideas & exploratory tools
  - Provide examples to illustrate technologies & tools.
- Caveats
  - Not exhaustive, definitely a biased view

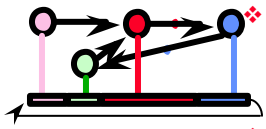
Acknowledgements

- Sandeep Shukla, Virginia Tech
- Mani Srivastava, UCLA
- P. Subrahmanyam, Stanford

Embedded systems design: Major subtasks







- Modeling
  - the system to be designed, and experimenting with algorithms involved;
- Refining (or "partitioning")
  - the function to be implemented into smaller, interacting pieces;
- HW-SW partitioning: Allocating
  - elements in the refined model to either (1) HW units, or (2) SW running on custom hardware or a general microprocessor.
- Scheduling
  - the times at which the functions are executed. This is important when several modules in the partition share a single hardware unit.
- Mapping (Implementing)
  - a functional description into (1) software that runs on a processor or (2) a collection of custom, semi-custom, or commodity HW.

What is HW/SW Co-design?

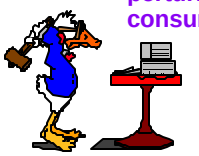
- Traditional Design
  - SW and HW partitioning is decided at an early stage, and designs proceed separately from then onward.
- CAD today addresses synthesis problems at a purely hardware level:
  - efficient techniques for data-path and control synthesis down to silicon.
- ECS use diverse (commodity) components
  - uP, DSP cores, network and bus interfaces, etc.
- "New fangled" Co-design
  - A flexible design strategy, wherein the HW/SW designs proceed in parallel, with feedback and interaction occurring between the two as the design progresses.
  - Final HW/SW partition/allocation is made after evaluating trade-offs and performance of options
    - ➜Seek delayed (and even dynamic) partitioning capabilities.

CAD for Embedded Computing Systems

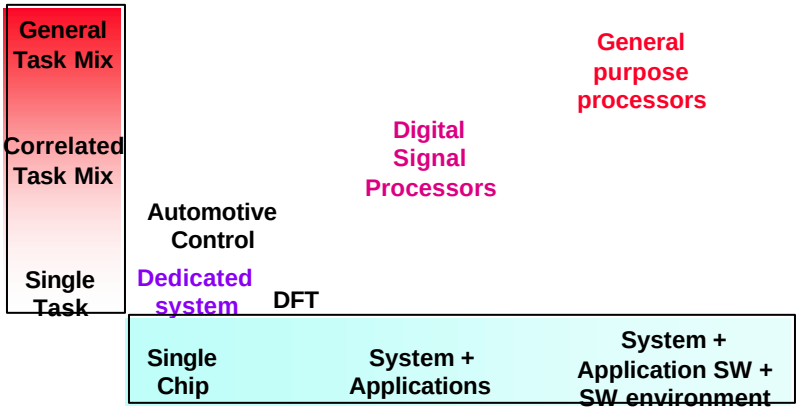
- Co-design: joint optimization of Hardware and software
  - cost-performance tradeoffs as a part of product implementation, as opposed to product specification.
- Co-synthesis: synthesis assisting co-design
  - designs derived from (formal) specifications
  - rapid exploration of design alternatives.

Complicating factors in the design of embedded systems

- Many of the subtasks in design are intertwined.
  - Allocation depends on the partitioning, and scheduling presumes a certain allocation.
- Predicting the time for implementing the modules in hardware or software is not very easy, particularly for tasks that have not been performed before.
  - If a particular module has been implemented earlier, or there is data about an almost similar design, then prediction can exploit this precedence. e.g.,
    - Building a processor (2- years?)
    - C/FORTRAN compiler for a standard architecture.
  - Even then, details and personnel may change, causing perturbations in the actual time and resources consumed.



Ingredients of the task: Scope of Codesign



- The specific issues that need to be addressed in codesign depend to some extent on
  - the scope of the application at hand, and
  - the richness of the system delivered.

Disciplines used in embedded system design

- The design of embedded systems draws upon several disparate disciplines in CS and EE.
- Application domain (Signal processing, Comm./network engg. …)
- Software engineering (Programming Languages, Compilers)
  - For implementing the software components; this can be a major component of several systems.

- VLSI (computer aided) design
  - For implementing the custom and semi-custom hardware components.
- Parallel/Distributed system design
  - since many embedded systems and multiprocessors are structured as a network of communicating processors; the network may be loosely coupled or tightly coupled.
- Real-time systems (Hard- & soft- real time systems)

System Specification and Modeling

Warning! This is really a 'review' material.

System Specification: Goals & Characteristics

- Main purpose: provide clear and unambiguous description of the system function, and to provide a
  - documentation of the initial design process
- Support
  - diverse models of computation
    - A MOC describes rules to mimic a certain behavior
      - How computation proceeds and info transferred
  - allow the application of computer-aided design tools for
    - design space exploration
    - partitioning
    - software-hardware synthesis
    - validation (verification, simulation)
    - testing
- Should not constrain the implementation options.
  - diverse implementation technologies.

Examples of useful Computation Models

- Petri nets
- Data flow
  - Static (Synchronous), multi-rate, dynamic, multidimensional, ...
  - Process networks
- Discrete event models
- Communicating Sequential Processes (CSP)
- Finite State Machines (FSM)
  - Hierarchical, Nondeterministic, ...
- Object-oriented models
- Imperative models
- Functional models

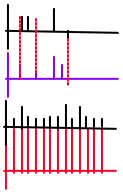
Choosing Models & Languages

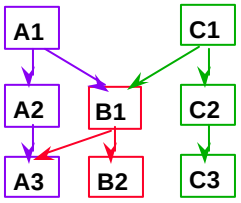
- Model choice: depends on the application domain, e.g.,
  - DSP (digital signal processing) applications use data flow models;
  - Control intensive applications use finite state machine models;
  - HW simulation (algorithms & engines) use simulation models;
  - Event driven applications use reactive models;
- Language choice: depends on:
  - Underlying semantics:
    - the language syntax must have a semantics in the model appropriate for the application.
  - Available tools
  - Personal taste and/or company policy.

An Event-based characterization: <Events> & <Processes> can vary

- Continuous time models, e.g., Analog

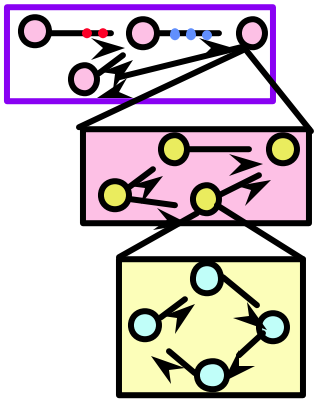
- Discrete-time models
  - Totally ordered events: <time-stamp, event> e.g., VHDL
    - "Discrete-time cycle driven"; e.g., DSP
      - <clock-tick, event>
      - Events with the same clock tick may be ordered by data dependencies.
      - Used in: DSP systems.
    - "Multi-rate discrete time, cycle driven", e.g., multi-rate DSP
      - Every n-th signal in one process aligns with another.
    - Synchronous Finite state machines.
  - Partially ordered events
    - e.g., Petri nets, process algebras, ...





Graphs as an abstract syntax for models

- Graph = <Nodes, Arcs>
- Many computation models are obtained by ascribing specific interpretations (semantics) to the nodes and arcs, and sometimes by constraining the structure of the graph.
- Examples of graph-based computation models:
  - Petri nets
  - Data flow
  - Networks of processes
  - Queuing models
  - Control-data flow graphs
  - Finite State Machines
- Hierarchical graphs thus offer a useful visual representation for many application domains.

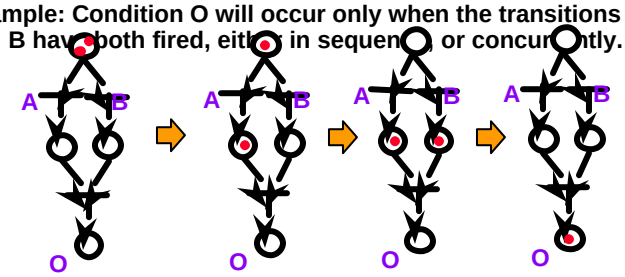


Queuing Models

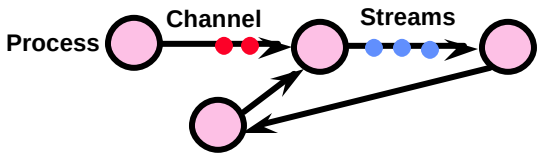
- Queuing models are graph-based system level models.
  - Nodes: complex operators e.g., Poisson queues; computations & decision nodes.
  - Arcs: Events/tokens/requests.
- Used for:
  - performance estimation, e.g., determining overall throughput, latency, of a network of queuing nodes
- Some commercial modeling systems
  - combine queuing-theory based models with other "program like" (c-code) nodes.
    - e.g., SES modeling system

Petri Nets

- Petri nets
  - events (transition nodes) execute ("fire") when certain conditions hold ("markers are present in the places nodes preceding events").
- Models “true” concurrency (as opposed to interleaving event sequences). Somewhat weak in supporting hierarchical descriptions and compositionality.
  - Many variants e.g., Colored Petri nets.
  - Very powerful; many problems undecidable unless Petri nets are very constrained in structure & semantics.
- Example: Condition O will occur only when the transitions A and B have both fired, either in sequence or concurrently.



Process Networks



- Network of
  - concurrent sequential processes,
  - communicating via 1-way FIFO channels;
  - the channels have unbounded capacity,
  - one producer and consumer;
  - writes to the channel are non-blocking, and
  - reads from the channel are blocking.
- Kahn process semantics:
  - Mapping from one or more input sequences to one or more output sequences
  - Flexible (but hard to schedule)
  - Simplified/restricted in Synchronous Data Flow (SDF)
    - Easy to schedule

Example

```
Process f (in int u, in int v, out int w) {
    int i; bool b = true;
    for (;;) {
        i = b ? wait (u) : wait (v); // wait returns next token in FIFO, blocks if empty
        printf("%i\n", i);
        send (i, w); // writes a token into a FIFO w/o blocking
        b = !b;
    }
}
```

- A process can not check whether data is available before attempting a read
- A process can not wait for data on more than one port at a time
- Therefore, order of reads, writes depend only on data, not its arrival time
- It is a challenge to schedule KN without accumulating tokens.

Variants of data flow models

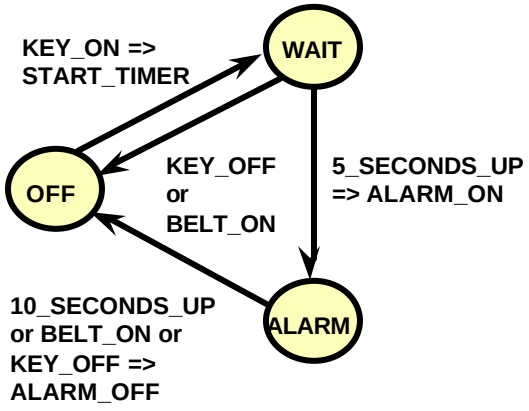
- Synchronous data flow
  - Flow of control is predictable at compile time.
  - Schedule can be constructed once, and repeatedly executed.
  - Applications: synchronous multirate signal processing.
- Dynamic data flow
  - The consumption and production of data tokens (node firing) is data dependent.
  - Turing complete => many questions undecidable.
  - Algorithms exist that work most of the time. Combine these with dynamically constructed schedule.
- Multidimensional data flow
  - useful for 2-D operations, e.g., images & video.

Finite State Machines (FSMs)

- Properties of FSMs
  - Good for specifying sequential control.
  - Not Turing complete.
    - More amenable to formal analysis.
- Typical domains of application
  - Control-intensive tasks.
  - Protocols (Telecom, cache-coherency, bus, ...)
- Many variants of the formulation
  - Differ in communication, determinism, ...

FSM Example: Seat Belt Alarm Control

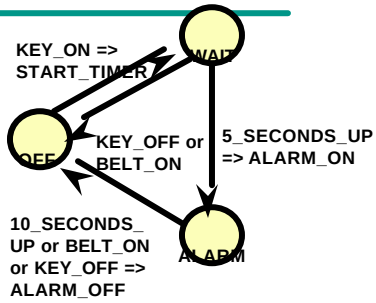
- Informal Specification
  - If the driver
    - turns on the key, and
    - does not fasten the seat belt within 5 seconds
  - then sound the alarm
    - for 5 seconds, or
    - until the driver fastens the seat belt
    - or until the driver turns off the key



No explicit condition => implicit self-loop in the current state

Finite State Machine: Example + Definition

- FSM = (Inputs, Outputs, States, InitialState, NextState, Outs)
  - Inputs = {KEY_ON, KEY_OFF, BELT_ON, BELT_OFF, 5_SECONDS_UP, 10_SECONDS_UP}
  - Outputs = {START_TIMER, ALARM_ON, ALARM_OFF}
  - States = {OFF, WAIT, ALARM}
    - InitialState = OFF
  - NextState: CurrentState, Inputs -> NextState
    - e.g., NextState(WAIT, {KEY_OFF}) = OFF
      - ➢ All inputs other than KEY_OFF are implicitly absent
  - Outs (function): CurrentState, Inputs -> Outputs
    - e.g., Outs(OFF, {KEY_ON}) = START_TIMER



Non-deterministic Finite State Machines

- A finite state machine is said to be non-deterministic when
  - The NextState and Output functions may be RELATIONs (instead of functions).
- Non-determinism can be user to model
  - unspecified behavior
    - incomplete specification
  - unknown behavior
    - e.g., the environment model
  - abstraction
    - (the abstraction may result in insufficient detail to identify previously distinguishable situations)

Reactive (Real-time) Systems

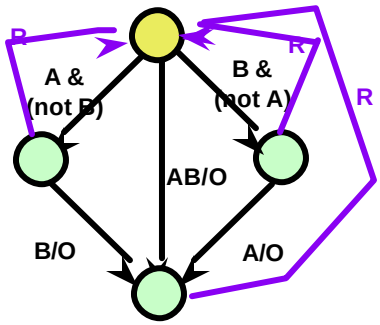
- Reactive Systems
  - "React" to events
    - e.g., in the external environment, other subsystems
- Suited for modeling "non-terminating" interactions
  - e.g., operating systems, interrupt handlers, process control systems.
  - Often subject to external timing constraints
    - "real-time"

Reactive Synchronous Languages

- Assumptions
  - the system reacts to internal and external events by emitting other events
  - events can occur only at discrete time instances
  - the reactions are assumed to be "instantaneous"
    - In practice, this means that it takes negligible or a relatively small time to process the event.
    - If the processing is significant, start and end events can be associated with this task.
- Control flow oriented (imperative) languages
  - Esterel
- Data flow languages
  - Lustre, Signal
- Simple and clean semantics
  - based on FSMs.
- Deterministic behavior
- Simulation, software and hardware synthesis, verification

Esterel: An imperative language

```
module EsterelFSM
input A, B, R;
output O;
    loop
        do
            [await A || await B];
            emit O;
            halt;
        watching R
    end loop
end module;
```



- As the number of interrupts (signals to watch) increases,
  - the size of the Esterel program grows linearly,
  - while the FSM complexity grows exponentially.

Esterel: Example

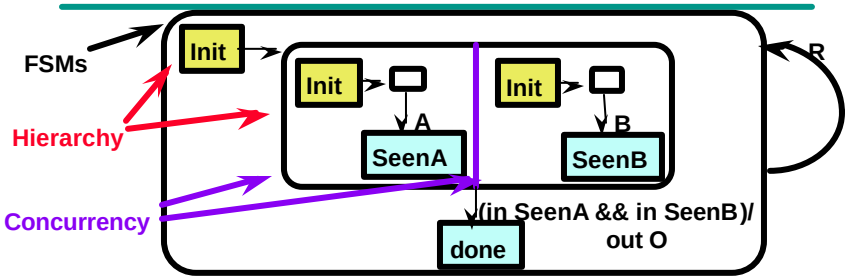
- Deterministic Parallelism

```
trap END in
  await SECOND;
  emit ALARM;
  exit END
||
  await BUTTON;
  emit ACTION;
  exit END
end
```

Esterel Backgrounder

- **To Language:** Origins in control theory – programming control that provided support for time, delays, preemption
  - A new vision of concurrency that allowed for determinism and broadcast (rooted in G. Plotkin's Structural Operational Semantics, SOS)
- **Language to Automata:** Combined with Brozozowski's derivative algorithmic for translating REs into automata that can used in SOS languages
  - Improvements by Gonthier in state encoding
  - Early adoption by Dassault avaiation
- **Language to Circuits:** Adapted to controller specification. Structural translation from language to circuits and their optimization. Later to software generation. Avoided state explosion inherent in earlier efforts.
- **Handling Causality and Constructive Semantics:** handling cyclic circuits was tough for the circuit generation from Esterel until Malik/Shiple's work that enabled Esterel to be used in compiling cyclic programs.
- Esterel now finding its way with C (ECL); Java (Jester) and by itself
  - Current extension into Multiclock Esterel (other synch languages are also doing the same, e.g., POLYCHRONY from Signal).

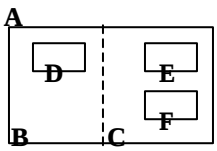
State Charts: An Example

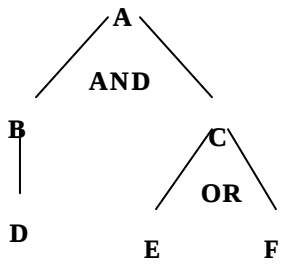


- Visual syntax: FSMs + concurrency + hierarchy.
- 2 concurrent state machines monitor the signals A and B.
- When both FSM transition to their final state,
  - the "higher level" FSM transitions to its "done" state.
  - Reset signal (R) =>
    - self-loop at the highest level of the hierarchy is triggered,
    - reinitializing all FSMs in the initial state.
- Program size grows linearly with signals being monitored.

StateCharts

- Objects
  - states
  - events
  - conditions
  - actions (outputs)
- Events and conditions cause state transitions
- AND or OR composition of states
  - leads to a configuration.





StateCharts (continued)

- Transitions
  - Between states and/or configurations
  - Can be across hierarchy
  - Unique *source* state identified
  - Multiple *sink* states
  - Multiple simultaneous transitions
  - A transition can cause subsequent transitions
  - Transitions or states can be labeled for output actions.

- There are many variants of such a formalism.
  - Commercial systems
    - Statemate (iLogix), VisualHDL (Summit Design Inc), SpeedChart, StateVision, ...

Concurrent & RT Programming Languages

- Primitives
  - semaphores (shared variables + atomic test-and-set mechanisms)
  - monitors (more elaborate forms of semaphores: shared data structures + interface to them via functions/procedures)
  - message passing/interprocess communication
- Interprocess communication is supported by a combination of
  - programming languages
  - operating systems
  - hardware
- Real-time programming languages
  - Specify real-time requirements
  - These cannot be guaranteed unless the infrastructure supports the required primitives (the compilers, operating systems, networks hardware, I/O, peripherals, etc.)

How models influence an application design?

- Consider the following problem: Given input from a camera, digitally encode it using MPEG II encoding standards.
- This task involves:
  - storing the image for processing
  - going through a number of processing steps
    - e.g., Discrete cosine transform (DCT), Quantization, encoding (variable length encoding), formatting the bit stream, Inverse Discrete Cosine transform (IDCT), ...
- Is this problem appropriate for
  - Reactive Systems, Synchronous Data flow, CSP, ...
- More than one model be reasonable.
- Choice may be influenced by
  - availability of tools
  - efficiency of the model
    - in terms of simulation time
    - in terms of synthesized circuit/code.

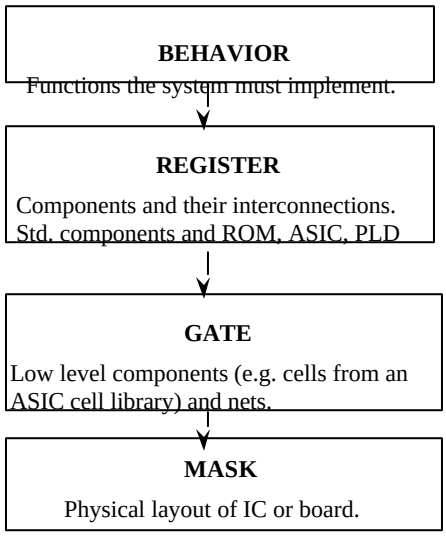
Co-Design and Co-Synthesis

Language-level Design for Hardware, Software
Hardware and software synthesis
Modeling with network interfaces
Integrated HW, SW modeling, simulation

Why Computer-Aided Design?

- In system software CAD corresponds to compiler tools
- In hardware, CAD refers to a collection of tools for circuit synthesis and optimizations
- Increasing role of *design methodology*
  - A design environment consists of
    - Design tools to carry out various design tasks
    - A suggested or preferred method of using tools
  - Methodology ensures timely and correct completion of tasks.
    - e.g..., implementing an engineering change.
  - Often needed for team logistics reasons.

Example: Simplified HW Design Flow



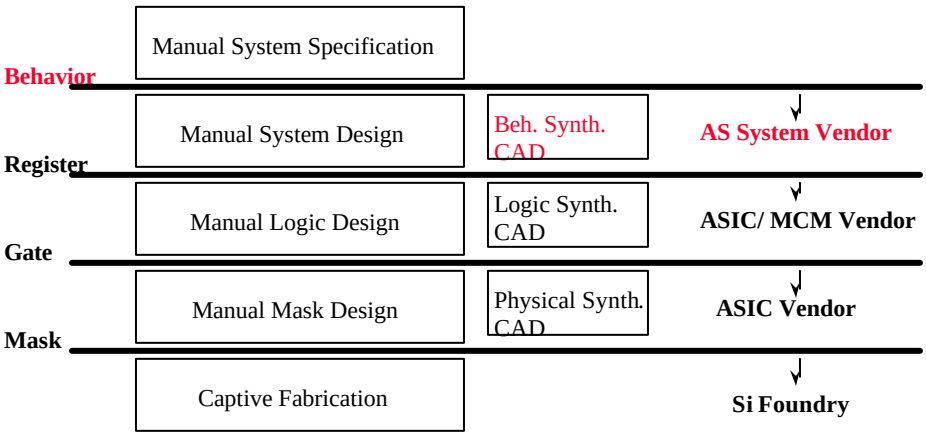
DS based on cycle-time, area latency.

ROM: read-only memory
ASIC: Application-Specific Integrated-Circuit
PLD: Programmable LogicDevice

cell: logic component with pre-determined electrical characteristics.

net: set of terminals connected together.

The Evolving Design Flow



High-level Design and Its Language

- Input *specification* using programming languages
- Knowledge about target hardware technology not necessary
- Large exploration space for design implementation

- The choice of language is important
  - Language often embodies the methodology
  - Specific to domains
    - VHDL, Verilog: model system, testbench
    - DSP: Blocks with fixed IO rates
    - Java: threads with per object locks

Hardware Specification

- Behavioral specification
  - Operations and ordering between operations
  - Timing behavior is relative
  - Resource usage partially or completely unidentified
- Register-Transfer Level (RTL) specification
  - Represents micro-architecture
  - Operations as synchronous transfer between functional units
- Behavioral to RTL translation is manual or automatic
  - Substantial growth industry in circuit synthesis and optimization tools at various levels.

Hardware vs. Software Languages

- Programming languages are often used for constructing system models
- Hardware
  - concurrency in operations
  - I/O ports and interconnection of blocks
  - exact event timing is important: open computation
- Software
  - typically sequential execution
  - structural information is less important
  - exact event timing is not important: closed computation.

Language Distinctions

- Distinctions based on data types, interface abstraction, communication and time. For instance:

- Communication
  - shared variables using explicit communication architectures
  - synchronous handshaking using implicit communications (ADA task entry call)
  - instantaneous broadcast (Esterel)
  - asynchronous message passing using explicitly communication architectures
- Time
  - global, multiple clocks, logics.

Languages for Hardware

- Declarative or imperative styles
  - Functional level descriptions often use a mixture of behavioral and structural views.
  - Based on view supported:
    - Behavioral: mainly imperative
    - Structural: mostly declarative
    - Physical: declarative or procedural
- Variables (implemented as storage, wires, mux)
  - Unbuffered 'wire' communication
  - data-structures
  - multiple-assignments
  - implementation choices determined by variable resolution
- Discrete event semantics
  - Timing semantics

Popular HDLs

- Cycle-based or event-based semantics
- Behaviors described as a collection of *processes*
  - sequential processes (VHDL)
  - parallel processes (Verilog, HardwareC)
  - global storage across processes
- Concurrency at the operation or process level
- Structural as well behavioral constructs
  - control flow

Time in VHDL

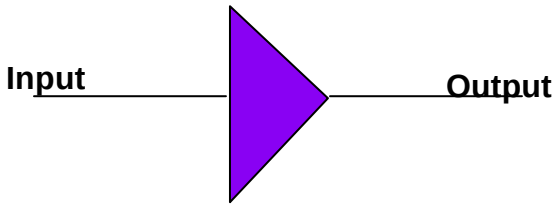
- Discrete-event semantics
  - Event = assignment to a signal
- Processes are triggered by events
- A simulation frame lasts until all processes are dead-locked
- A simulation run consists of many frames
- In given frame a process can generate multiple events

A two-level timing model.

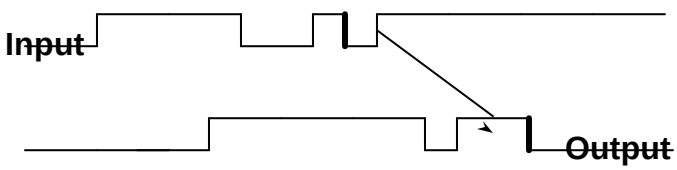
Pre-emption

- HDLs model physical components and their behaviors
- Two kinds of event propagation through a component:
  - inertial mechanism
  - transport mechanism
- Pre-emption needed to ensure event causality

Example



Output <= ‘1’ after 10 ns when input = ‘1’ else
‘0’ after 14 ns

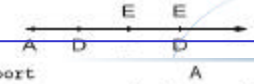


Time in Esterel

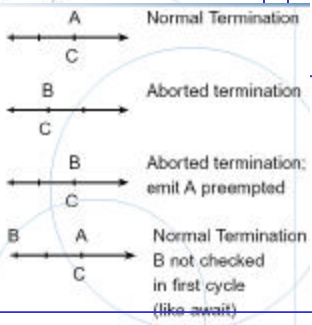
- Global clock with precise control over when events appear
  - At every tick:
    - Inputs presented, computation, outputs ready
- Statements
  - A bounded number can be packed in one cycle, or
    - Emit, present, loop
  - Take multiple cycles:
    - Pause, await, sustain

Esterel Examples

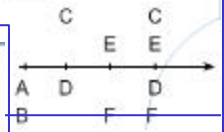
```
emit A;
emit B;
pause;
loop
  present C then emit D end;
  present E then emit F end;
  pause;
end
```



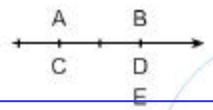

```
abort
  pause;
  pause;
  emit A
when B;
emit C
```




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




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



Source: Stephen Edwards, Columbia

Co-Design and Co-Synthesis

Language-level Design for Hardware, Software
Hardware and software synthesis
Modeling with network interfaces
Integrated HW, SW modeling, simulation

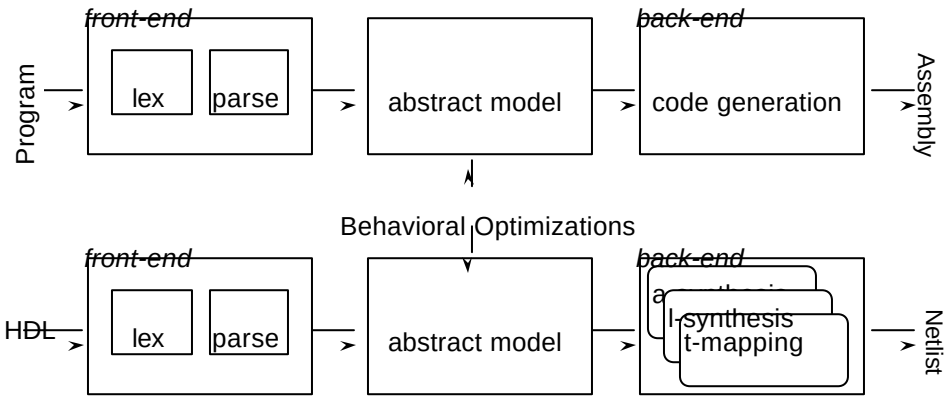
Compilation & Synthesis

- Compilation spans programming language theory, architecture and algorithms
- Synthesis spans concurrency, finite automata, switching theory and algorithms
- In practice, the two tasks are inter-related.
- Compilation & Synthesis in three steps:
  - front-end, intermediate optimizations, back-end.

Compilation

- Program compilation for software target
  - Front-end parsing into intermediate form
  - Optimization over the intermediate form
  - Back-end code-generation for a given processor
- HDL compilation for hardware target
  - Front-end parsing into intermediate form
  - Optimization over the intermediate form
  - Back-end architecture, logic and physical synthesis.

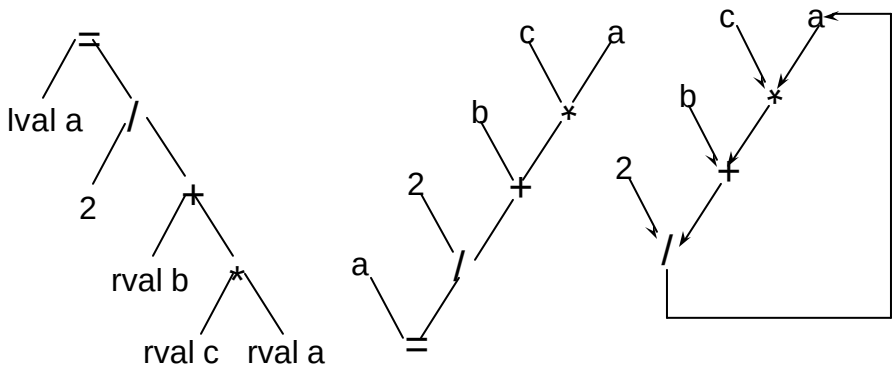
Compilation Anatomy



Target independent **Language independent**

Front End

a = (b+c*a)/2



Behavioral Optimizations

- Semantic preserving transformations
- Implemented as multiple-pass traversals over the intermediate form
- Types
  - Data-flow based
  - Control-flow based
  - Synthesis oriented

Data-oriented Transformations

- Traditional compiler
  - common sub-expression elimination
  - constant propagation
  - tree-height reduction
  - dead-code elimination
  - variable renaming
  - operator strength reduction, copy propagation, etc.
- Concurrency enhancing
  - pipeline interleaving
  - block processing
  - unfolding with look-ahead

Control, Synthesis Transformations

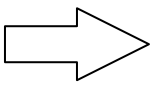
- Control-oriented Transformations
  - Loop transformations
  - FSM-based transformations
    - Explicit versus implicit state transitions
    - Minimization of state machines
- Synthesis oriented Transformations
  - Concurrency enhancing transformations
  - Combinational conditional and block coalescing
  - Variable resolution and multiplexor structures
  - Incorporation of *Don't Care* conditions

Conditional Coalescing

- If branches contain only 'combinational' logic operations then they can be merged to larger logic blocks.
- Supports operation chaining
- Oriented towards subsequent logic synthesis
- Can derive don't care information and pass it on to the logic synthesis tools.

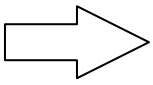
Example

```
if (q) {
   a = b & c;
   d = e | f;
   u = b + d;
} else {
   h = i xor j;
   x = y z;
   u = b d;
}
```



```
a = b & c;
d = e | f;
h = i xor j;
x = y z;
u = q(b+d)+q'(bd);
```

```
T1 = a & b;
T2 = T1 & c;
write b = T1;
x = read(b);
T3 = x | y;
T4 = z & w;
T5 = T3 & T4;
```



```
{ T1 = a & b;
T2 = T1 & c;}
write b = T1;
x = read(b);
{ T3 = x | y;
T4 = z & w;
T5 = T3 & T4;}
```

Hardware Synthesis Objectives

- Generate a structure suitable for synchronous and single-phase circuits
  - resource performance in terms of execution delay
  - in number of clock cycles
- Design space:
  - area, cycle time, latency, throughput
- Optimal implementation
  - maximum performance subject to area constraints
  - minimum area subject to performance constraints

Synthesis Tasks

- Operation scheduling, resource binding, control generation
- Scheduling determines operation start times
  - minimize latency
- Resource binding: resource selection, allocation
  - minimize area (maximize sharing)
- Problem:
  - scheduling affects area; binding affects latency

Putting it together

- Hardware constituents:
  - data-path = “connectivity synthesis”
    - detailed resource connections
    - steering logic
    - connection to the interface
  - control synthesis
    - synthesize controller that provides operations/resource enables, operation synchronization, resource arbitration

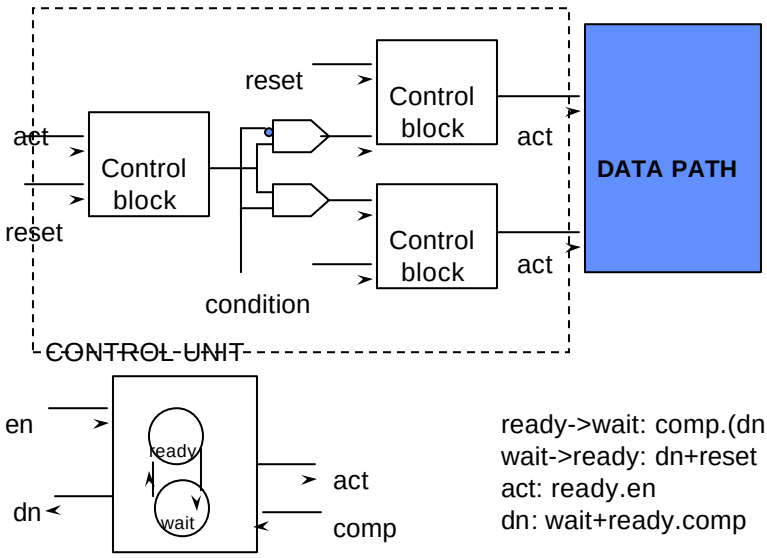
Control Generation

- Dependent upon the model of control
- Two types
  - Micro-programmed
    - micro-code, PLA or ROM implementations
  - FSM-based
    - Single FSM
    - Network of FSMs

FSM-based Control Implementations

- Simple model
  - one state for each control step
  - next-state function: unconditional
  - output function: enable operations
- Extended model
  - branching and iteration: conditional next-state function
  - hierarchy: interconnection of FSMs

Example



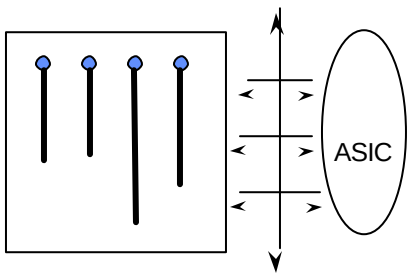
ready->wait: comp.(dn+reset)'
wait->ready: dn+reset
act: ready.en
dn: wait+ready.comp

A CAD Methodology for SW

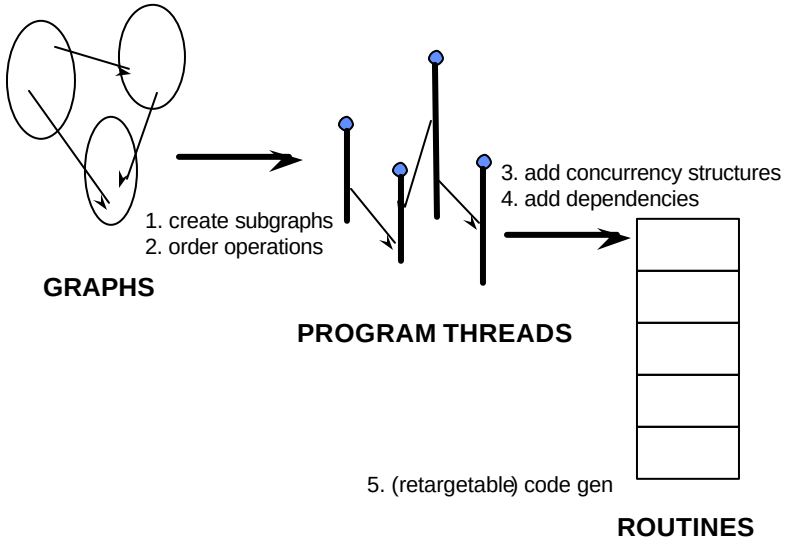
- Automated software synthesis from specs.
- Synthesis tools generate implementation
  - Global optimization of the program.
  - One-time compilation costs.
- Optimization used to achieve design goals.
- Analysis and verification tools for feedback.

Software Synthesis

- Software system model
  - set of program threads
    - latency
    - reaction rate
  - implemented as coroutines



Steps in Software Synthesis



Program Thread Generation

- Constraint linearization
- Overhead reduction is important
- Thread latency versus overhead trade-offs
  - Thread frames (Goosens, IMEC)
- Choice of runtime system
  - control FIFO scheduler
    - non-preemptive
  - extension to preemptive scheduling proposed by Goosens, et. al.
- Techniques finding use in software synthesis for very small footprint sensor networks
  - E.g., TinyOS construction

Co-Design and Co-Synthesis

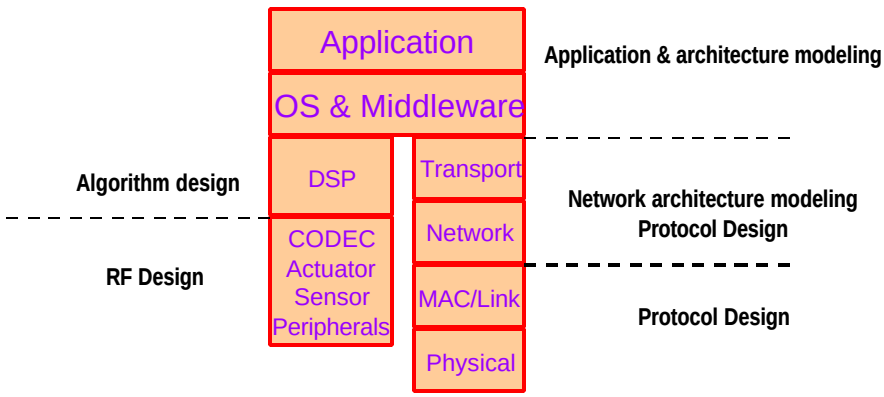
Language-level Design for Hardware, Software
Hardware and software synthesis
Modeling with network interfaces
Integrated HW, SW modeling, simulation

Networked Embedded Systems

These are embedded systems with interesting network interfaces.

- Unique challenges in design technology
- Two views of Networked SOCs
  - compositional (or ASIC view)
  - architectural (or network-centric view)
- Scope and categories of design tools for NSOCs
- System-level composition through OO mechanisms
- Network architectural modeling

Network Systems View



71

ASIC & Network Models

- Complementary models
  - ASIC models focus on "node" implementation
  - Network model keeps "multi-node" system view
- Example: Synopsys Protocol Compiler, NS models.

- "Theoretically" both models can support either view
- Designers often need the ability
  - to tradeoff across layers (easier in ASIC models) while
  - keeping the system view (easier in network models).

- Hence, a convergence in works on integration of ASIC and Network models
  - MIL3 OPNET, Cadence Bones, Diablo
  - HP EEsof's ADS, AnSoft HFSS, Cadence Allegro, Anadigics, White Eagle DSP, ...

72

Scope of NSOC Design Tools

- Design of single-chip systems with radio transceivers requires tools
  - to explore new architectures containing heterogeneous elements
  - to explore circuit design containing analog/digital, active/passive components (mixed signal design)
  - to accurately estimate parasitic effects, package effects
- Typically mixed-system design entails
  - antennae design
  - network design: interference, user mobility, access to shared resources
  - algorithmic simulations
  - protocol design
  - circuit design, layout and estimation tools

73

Categories of Design Tools

- Architectural design tools
  - network, protocol simulations
  - algorithmic simulations, partitioning and mapping tools
- Design environment tools
  - encapsulated libraries, library management for design components
- Module design
  - low noise integrated frequency synthesizers
  - base-band over-sampled data converters
  - design of RF, analog, digital VLSI modules
- Modeling, characterization and validation tools
  - characterization of mixed-mode designs, RF coupling paths, EMI
  - simultaneous modeling, design and optimization of antenna, passive RF filter, RF amp, RF receiver, power amp. components

74

Network Architectural Design

or “behavioral design” for wireless systems

- Design network architecture
  - point-to-point, cellular, etc
- Design protocols
  - specification
  - verification at various levels: link, MAC, physical
- Tools in this category
  - Matlab, Ptolemy (and likes)
  - network, protocol simulators
- Tools are designed for simulations specific to a design layer:
  - simulation tools for algorithm development
  - simulation tools for network protocols
  - simulation tools for circuit design, hardware implementation, etc.

75

Network Architecture Modeling: NS

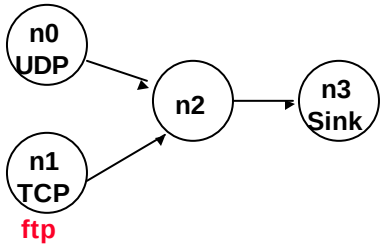
- Developed under the Virtual Internet Testbed (VINT) project (UCB, LBL, USC/ISI, Xerox PARC)
- Captures network nodes, topology and provides efficient event driven simulations with a number of “schedulers”
- Interpreted interface for
  - network configuration, simulation setup
  - using existing simulation kernel objects such as predefined network links
- Simulation model in C++ for
  - packet processing
  - changing models of existing simulation kernel classes, e.g., using a special queuing discipline.

76

Example:
A 4-node system with 2 "agents", a traffic generator

- "Agents" are network endpoints where network-layer packets are constructed or consumed.

```
set ns [new Simulator]
set f [open out.tr w]
$ns trace-all $f
set n0 {$ns node}
set n1 {$ns node}
set n2 {$ns node}
set n3 {$ns node}
$ns duplex-link $no $n2 5Mb 2ms  DropTail
$ns duplex-link $n1 $n2 5Mb 2ms DropTail
$ns duplex-link $n2 $n3 1.5Mb 10ms DropTail
set udp0 [newagent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [newapplication/Traffic/CBR]
$cbr0 attach-agent $udp0
..
$ns at 3.0 "finish"
proc finish () {
            …
}
$ns run
```



77

NS v2 Implementation and Use

- A "Split-level" simulator consisting of
  - C++ compiled simulation engine
  - Object Tcl (Otcl) interpreted front end
- Two class hierarchies (compiled, interpreted) with 1-1 correspondence between the classes
  - C++ compiled class hierarchy
    - allows detailed simulations of protocols that need use of a complete systems programming language to efficiently manipulate bytes, packet headers, algorithms over large and complex data types
    - runtime simulation speed
  - Otcl interpreted class hierarchy
    - to manage multiple simulation "splits"
    - important to be able to change the model and rerun
- NS pulls off this trick by providing *tclclass* that provides access to objects in both hierarchies.

78

NS Implementation

- Example:
  - Otcl objects that assemble, delay, queue.
  - Most routing is done in Otcl
  - HTTP simulations with flow started in Otcl but packet processing is done in C++
- Passing results to and from the interpreter
  - The interpreter after invoking C++ expects results back in a private variable *tcl_->result*
  - When C++ invokes Otcl the interpreter returns the result in *tcl_->result*
- Building simulation
  - Tclclass provides simulator with scripts to create an instance of this class and calling methods to create nodes, topologies etc.
  - Results in an event-driven simulator with 4 separate schedulers: FIFO (list); heap; calendar queue; real-time.
  - Single threaded, no event preemption.

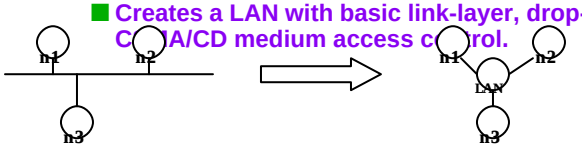
79

NS Usage: LAN nodes

- LAN and wireless links are inherently different from PTP links due to sharing and contention properties of LANs
  - a network consisting of PTP links alone can not capture LAN contention properties
  - a special node is provided to specify LANs
- LanNode captures functionality of three lowest layers in the protocol stack, namely: link, MAC and physical layers.
  - Specifies objects to be created for LL, INTF, MAC and Physical channels.
  - Example:

```
$ns make-lan <nodelist> <bw> <delay> <LL> <ifq> <MAC> <channel> <phy>
$ns make-lan "$n1 $n2" $bw $delay LL queue/DropTail Mac/CSMA/CD.
```

  - Creates a LAN with basic link-layer, drop-tail queue and CSMA/CD medium access control.

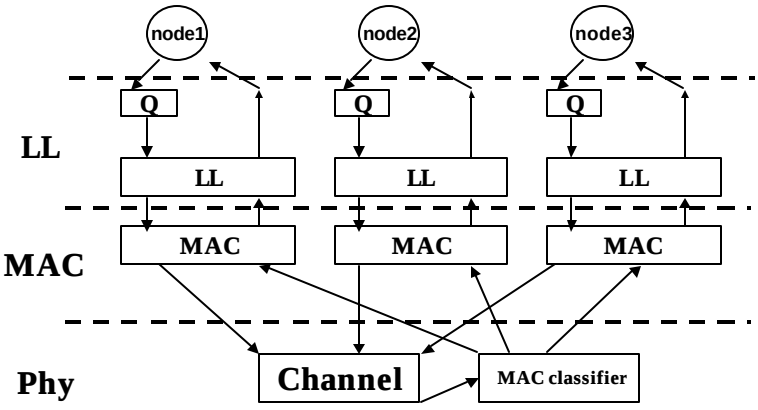


The LAN node collects all the objects shared on the LAN.

80

Network Stack simulation for LAN nodes in *ns*

Objects used in LAN nodes. Each of the underlying classes can be specialized for a given simulation.



Channel object simulates the shared medium and supports the medium access mechanisms of the MAC objects on the sending side.

On the receiving side, MAC classifier is responsible for delivering and optionally replicating packets to the receiving MAC objects.

Modeling of Mobile Nodes

From CMU Monarch Group

- Allows simulation of multihop ad hoc networks, wireless LANs etc.
- Basic model is a *MobileNode*, a split object specialized from ns class *Node*
  - allows creation of the network stack to allow channel access in MobileNode
- A mobile node is not connected through "Links" to other nodes
- Instead, a MobileNode includes the following mobility features
  - node movement (two dimensional only)
  - periodic position updates
  - maintaining topology boundary

82

Mobile Nodes

- As in wireline, the network "plumbing" is scripted in Otcl
- Four different routing protocols (or routing agents) are available
  - destination sequence distance vector (DSDV)
  - dynamic source routing (DSR)
  - Temporally ordered routing algorithm (TORA)
  - Adhoc on-demand distance vector (AODV)
- A mobile node creation results in
  - a mobile node with a specified routing agent, and
  - creation of a network stack consisting of
    - LL (with ARP), INT Q, MAC, Network Interface with an antenna.
- ➡ Enables integrated event driven simulation of mixed networks.

83

Mobile Node

```
Node/MobileNode instproc add-interface {channel pmodel lltype mactype qtype qlen iftype anttype } {

$self instvar arptable_ nifs_
$self instvar netif_ mac_ ifq_ ll_
set t $nifs_

set netif_($t) [new $iftype]          ;# net-interface
set mac_($t) [new $mactype]           ;# mac layer
set ifq_($t) [new $qtype]             ;# interface queue
set ll_($t) [new $lltype]             ;# link layer
set ant_($t) [new $anttype]
..
}

set topo [topography]
$topo bind_flatgrid $opt(x) $opt(y)
$node set x_ <x1>
$node set y_ <y1>
..
$ns at $time $node setdest <x2> <y2> <speed>
or
$mobilenode start
```

84

Network Simulation using OPNET

- Commercially available from MIL3
- Heterogenous models
  - for network
  - for node
  - for process
- Network, node, process editors
- Network models consist of node and link objects
- Nodes represent hardware, software subsystems
  - processors, queues, traffic generators, RX, TX
- Process models represent protocols, algorithms etc
  - using state-transition diagrams
- Simulation outputs typically include
  - discrete event simulations, traces, first and second order statistics
  - presented as time-series plots, histograms, prob. density, scattergrams etc.

85

OPNET Wireless System Modeling

- OPNET modeler with radio links and mobile nodes
- Mobile nodes include three-dimensional position attributes that can change dynamically as the simulation progresses.
- Node motion can be scripted (position history) or by a position control process.
- Links modeled using a 13-stage model where each stage is a function (in C)
- Transmitter stages:
  - Transmission delay model: time required for transmission
  - Link closure model: determine reachable receivers
  - Channel match model: determine which RX channel can demodulate the signal (rest treat it as noise)
  - Transmitter antenna gain: computes gain of TX antenna in the direction of the receiver
  - Propagation delay model: time for propagation from TX to RX.

86

Link Model Stages

- Receiver stages:
  - RX antenna gain: in the direction of the receiver
  - Received power model: avg. received power
  - Background Noise Model: computes the in-band background noise for a receiver channel
  - Interference noise model: typically total power of all concurrent in-band transmission
  - SNR model: SNR of transmission fragment based on the ratio of received power and interference noise
  - BER model: computes mean BER over each constant SNR fragment of the transmission
  - Error Allocation Model: determines the number of bit error in each fragment of the transmission
  - Error Correction Model: determines whether the allocated transmission errors can be corrected and if the transmitted data should be forwarded in the node for higher level processing.

87

Communications Toolbox (MATLAB)

- Part of the MATLAB DSP workshop suite
  - functionality models from MATLAB
    - sources, sinks and error analysis
    - coding, modulation, multiple access blocks, etc.
  - communication link models from SIMULINK
    - channel models: Rayleigh, Rician fading, noise models
- Good front-end simulations through vector processing
  - handles data at different time-points in large vectors
  - used in modeling physical layer component such as modems
  - useful in algorithm development and performance analysis
    - for modulation, coding, synchronization, equalization, filter design.

http://www.mathworks.com/products/communications/index.shtml

88

Co-Design and Co-Synthesis

Language-level Design for Hardware, Software
Hardware and software synthesis
Modeling with network interfaces
Integrated HW, SW modeling, simulation

Coming Generation of Embedded Systems & System-Chips

- Package boundary is enlarging
  - analog/RF, digital baseband, applications, RTOS, DSP, …
- Hardware-type 'behavioral' modeling just does not cut it
  - Substantial networking, communications, infrastructure software needs to be modeled as well.
- Learning from practice
  - People generally use C or C++ to model at system Level
  - Typically performance model and ISA models are built with C/C++
- → Why not 'standardize' use of C++ for system modeling purposes?
  - We already do software, network modeling.

Enter SystemC

- SystemC developed by Synopsys, Coware
  - Initially Scenic project (Synopsys and UC Irvine)
  - SystemC-0.9 (Sept 1999) based on Scenic
  - SystemC-1.0 (Early 2000) performance enhancements
  - SystemC-2.0 (mid 2001) – ideas from SpecC (UC Irvine) incorporated
  - SystemC-3.0 (yet to be released) – software APIs
- Other players that influenced SystemC
  - OCAPI library (IMEC Belgium)
  - Cynlib (FORTE Design Systems, formerly CynApps)
  - SpecC
  - SuperLOG (now SystemVerilog) from Coware (now Synopsys)

91

What Is SystemC?

- A C++ library that helps designers to use C++ to model/specify synchronous digital hardware
- Built in simulation libraries (simulation kernel) that can be used to run a SystemC program
- Any C++ compiler can compile SystemC
  - Simulation is free in comparison to Verilog/VHDL
- A compiler that translates the "synthesis subset" of SystemC into a netlist (Synopsys, FORTE)
- Language definition is publicly available
  - (Open SystemC Initiative or OSCI)
- Libraries are freely distributed
- Compiler is an expensive commercial product

92

Quick Overview

- A SystemC program consists of module definitions plus a top-level function that starts the simulation
- Modules contain processes (C++ methods) and instances of other modules
- Ports on modules define their interface
  - Rich set of port data types (hardware modeling, etc.)
- Signals in modules convey information between instances
- Clocks are special signals that run periodically and can trigger clocked processes
- Rich set of numeric types (fixed and arbitrary precision numbers)

93

Modules

- Hierarchical entity
- Similar to Verilog's module
- Actually a C++ class definition
- Simulation involves
  - Creating objects of this class
  - They connect themselves together
  - Processes in these objects (methods) are called by the scheduler (simulation kernel) to perform the simulation

94

Modules

```
SC_MODULE(mymod) {
  /* port definitions */
  /* signal definitions */
  /* clock  definitions */

  /* storage and state variables */

  /* process definitions */

  SC_CTOR(mymod) {
    /* Instances of processes and modules */
  }
};
```

95

Ports

- Define the interface to each module
- Entities through which data is communicated
- Port consists of a direction
  - input sc_in
  - output sc_out
  - bidirectional sc_inout
- and any C++ or SystemC type

96

Ports

```
SC_MODULE(mymod) {
  sc_in<bool> load, read;
  sc_inout<int> data;
  sc_out<bool> full;

  /* rest of the module */
};
```

97

Signals

- Convey information between modules within a module
- Directionless: module ports define direction of data transfer
- Type may be any C++ or built-in type

98

Signals

```
SC_MODULE(mymod) {
 /* port definitions */
 sc_signal<sc_uint<32> > s1, s2;
 sc_signal<bool> reset;

  /* … */
  SC_CTOR(mymod) {
    /* Instances of modules that connect to the signals */
  }
};
```

99

Instances of Modules

- Each instance is a pointer to an object in the module

```
SC_MODULE(mod1) { … };
SC_MODULE(mod2) { … };
SC_MODULE(foo) {
  mod1* m1;
  mod2* m2;
  sc_signal<int> a, b, c;
  SC_CTOR(foo) {
    m1 = new mod1("i1");  (*m1)(a, b, c);
    m2 = new mod2("i2");  (*m2)(c, b);
  }
};
```

Connect instance's ports to signals

100

Processes

- Procedural code with the ability to suspend and resume
  - (Not all kinds)
- Methods of each module class

101

Three Types of Processes

- METHOD
  - Usually Models combinational logic
  - Triggered in response to changes on inputs
- THREAD
  - Usually Models testbenches
- CTHREAD
  - Usually Models synchronous FSMs

102

METHOD Processes

```
SC_MODULE(onemethod) {
  sc_in<bool> in;
  sc_out<bool> out;

  void inverter();

  SC_CTOR(onemethod) {

    SC_METHOD(inverter);
    sensitive(in);

  }
};
```

Process is simply a method of this class

Instance of this process created

and made sensitive to an input

103

METHOD Processes

- **Invoked once every time input "in" changes**
- **Runs to completion: should not contain infinite loops**
  - **No mechanism for being preempted**

```
void onemethod::inverter() {
  bool internal;
  internal = in;
  out = ~internal;
}
```

Read a value from the port

Write a value to an output port

104

THREAD Processes

- Triggered in response to changes on inputs
- Can suspend itself and be reactivated
  - Method calls wait to relinquish control
  - Scheduler runs it again later
- Designed to model just about anything

105

THREAD Processes

```
SC_MODULE(onemethod) {
  sc_in<bool> in;
  sc_out<bool> out;

  void toggler();

  SC_CTOR(onemethod) {

    SC_THREAD(toggler);
    sensitive << in;
  }

};
```

Process is simply a method of this class

Instance of this process created

alternate sensitivity list notation

106

THREAD Processes

- **Reawakened whenever an input changes**
- **State saved between invocations**
- **Infinite loops should contain a wait()**

```
void onemethod::toggler() {
  bool last = false;
  for (;;) {
    last = in; out = last; wait();
    last = ~in; out = last; wait();
  }
}
```

Relinquish control until the next change of a signal on the sensitivity list for this process

107

CTHREAD Processes

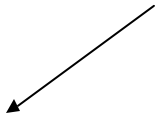
- **Triggered in response to a single clock edge**
- **Can suspend itself and be reactivated**
  - **Method calls wait to relinquish control**
  - **Scheduler runs it again later**
- **Designed to model clocked digital hardware**

108

CTHREAD Processes

```
SC_MODULE(onemethod) {
  sc_in_clk clock;
  sc_in<bool> trigger, in;
  sc_out<bool> out;

  void toggler();

  SC_CTOR(onemethod) {

    SC_CTHREAD(toggler, clock.pos());
  }

};
```

Instance of this process created and relevant clock edge assigned



109

SystemC Built-in Types

- sc_bit, sc_logic
  - Two- and four-valued single bit
- sc_int, sc_unint
  - 1 to 64-bit signed and unsigned integers
- sc_bigint, sc_biguint
  - arbitrary (fixed) width signed and unsigned integers
- sc_bv, sc_lv
  - arbitrary width two- and four-valued vectors
- sc_fixed, sc_ufixed
  - signed and unsigned fixed point numbers

110

SystemC Semantics

- Cycle-based simulation semantics
- Resembles Verilog, but does not allow the modeling of delays
- Designed to simulate quickly and resemble most synchronous digital logic

111

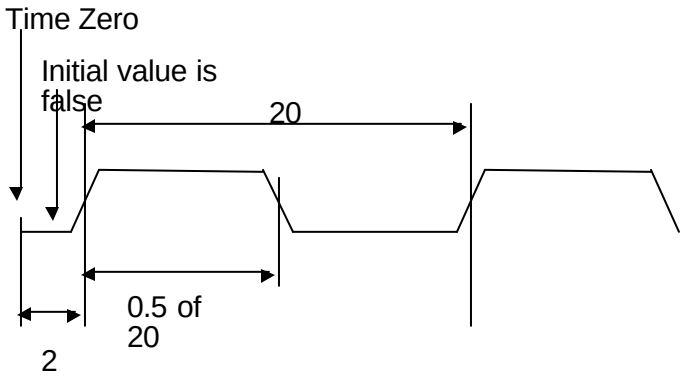
Clocks

- The only thing in SystemC that has a notion of real time
- Triggers SC_CTHREAD processes
  - or others if they decided to become sensitive to clocks

112

Clocks

- **sc_clock clock1("myclock", 20, 0.5, 2, false);**



113

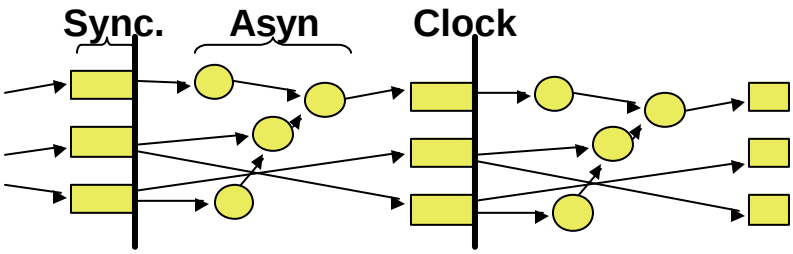
SystemC 1.0 Scheduler

- **Assign clocks new values**
- **Repeat until stable**
  - **Update the outputs of triggered SC_CTHREAD processes**
  - **Run all SC_METHOD and SC_THREAD processes whose inputs have changed**
- **Execute all triggered SC_CTHREAD methods. Their outputs are saved until next time**

114

Scheduling

- Clock updates outputs of SC_CTHREADs
- SC_METHODs and SC_THREADs respond to this change and settle down
- Bodies of SC_CTHREADs compute the next state



115

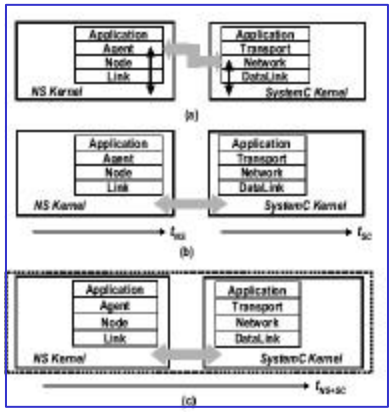
Recap: SC can connect to 'anything'

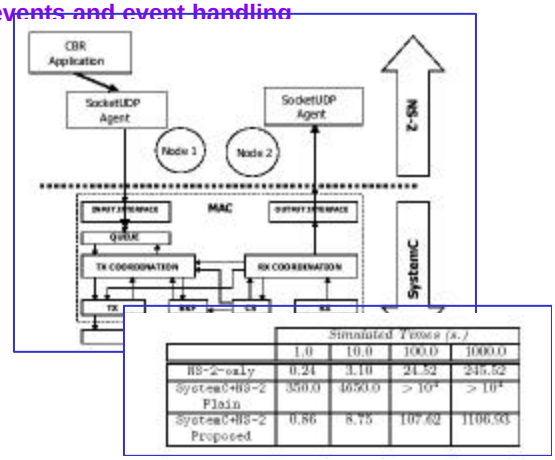
- SC_METHOD
  - Designed for modeling purely functional behavior
  - Sensitive to changes on inputs
  - Does not save state between invocations
- SC_THREAD
  - Designed to model anything
  - Sensitive to changes
  - May save variable, control state between invocations
- SC_CTHREAD
  - Models clocked digital logic
  - Sensitive to clock edges
  - May save variable, control state between invocations

116

SystemC and NS-2

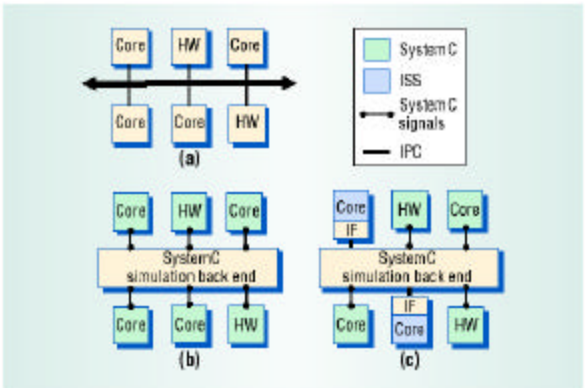
- Used in description of a 802.11 MAC Layer
  - Fummi *et al* in DAC 2003
- Integration possible because of DE MOC used in both
  - Different notion of events and event handling

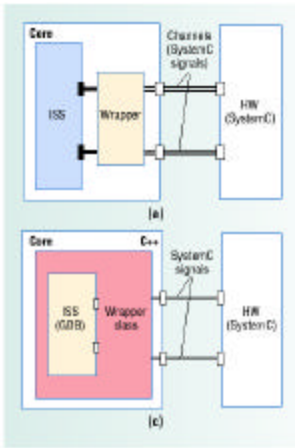




Complete Models Through Integration With ISS

- Too frequent communication with ISS can slow down the system simulation (usually through IPC)
- ISS 'wrapper' can be SystemC (interface) modules (IF)





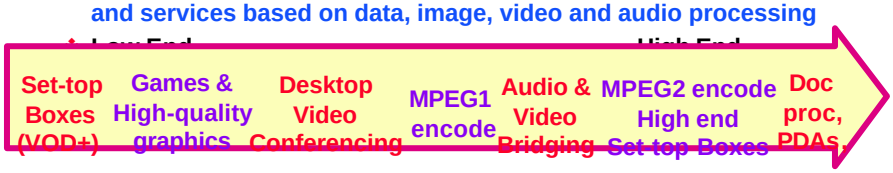
Source: Benini, Drago, Fummi, Computer 03

Industry Developments

Embedded Systems:
Desktop and Mobile Systems

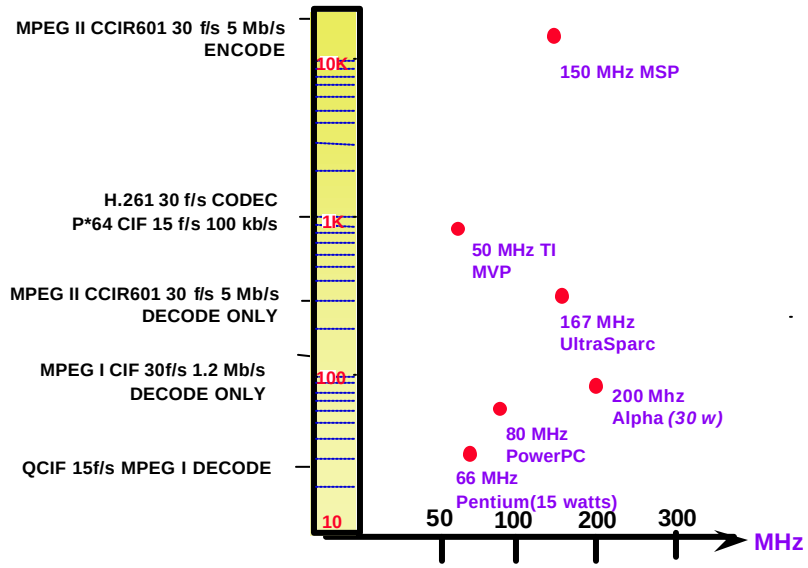
Multimedia: Market Opportunities & Challenges

- Progress in
  - (Audio, Video, Graphics) algorithms & standards
  - High performance CMOS VLSI
  - Local & wide area digital networks
  - Workstations & PCs
- ⤷ has yielded opportunities to develop new multimedia products and services based on data, image, video and audio processing



- Challenges: Designs & methodology must enable
  - Flexibility (adaptability to changes in standards)
  - Different cost-performance points
  - Product line management & migration paths
  - Differentiators (product features, quality, time to market)

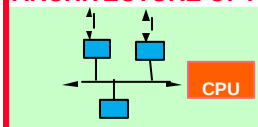
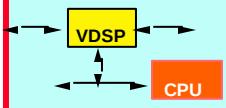
Computational Requirements for Video Coding



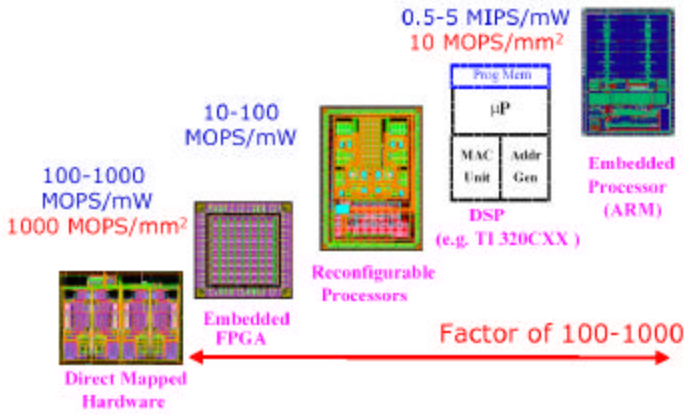
Multimedia Processing Requirements

- Multimedia signal processing places extreme demands on on the underlying computing platform
  - High throughput (hundreds to thousands of MIPS)
  - High I/O & memory bandwidth (tens - hundreds of MB/s)
  - Low latency
  - Real-time performance constraints
  - Predictable throughput, delay
  - Flexibility (rapidly evolving markets)
    - Rapid time to market critical
  - Low cost (< $100)
- These will lead to new architectures at the system, software & hardware level.

Hardware: Architectures & business models

ARCHITECTURE OPTION (EXAMPLE)	ASSUMPTIONS / BUSINESS MODEL
 Dedicated hardware functions (AVP)	Point solution for niche market(s)
 Super CPU, multimedia instns (Intel native DSP)	CPU has major market share
 Multimedia enhanced CPU (UltraSparc, Philips TriMedia, MicroUnity)	CPU does not (yet) have a major market share -s: Major SW development
 Video DSP + CPU (TI - MVP, Philips TriMedia, Chromatic MPACT, Lucent)	"In between solution": Leverages CPU HW & SW. -s: Less integration, greater cost

OTOH, Efficiency Varies



Source: Teresa Meng, Atheros

Announced Media Processor Architectures

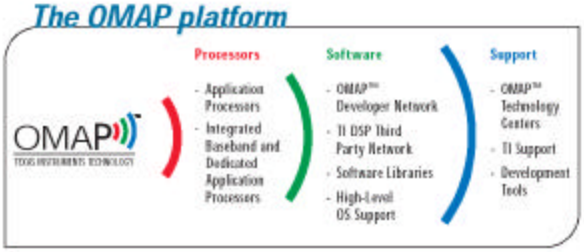
	TI MVP	Chromatics MPACT	Philips TriMedia	IBM MFAST	Intel P55C
Architecture	4 x 64b DSPs + 32 b RISC +cross-bar	VLIW/SIMD 4 ALUs ME engine 792b bus	VLIW 25 exec units+ VLD	VLIW 4x4 folded array SIMD columns 32b mesh	mP with split-word DSP instrs
Clock	40 MHz	62 MHz	100 Mhz	50 MHz	200 MHz
Peak Performance	1.2 Gops	2 Gops	4 Gops	20 Gops	800 Mops
Memory	DRAM 400 MB/s	RAMBUS 500 MB/s	SDRAM 400 MB/s	SDRAM 800 MB/s	DRAM 100 MB/s
Volume Production	95	2Q96	4Q96	2Q97	4Q96

Architectural Prototypes: Wireless NES

- Let us consider the following commercial "Platforms"
  - TI OMAP (Open Multimedia Applications Platform)
  - Intel PCA (Personal Internet Client Architecture)
  - Motorola MXC (Mobile Extreme Convergence)
- Different timelines, but all three directed at wireless SOC based systems (cellular, 802.11)
  - Tasks:
    - Communication : Networking : Applications
  - Generally seek energy efficient processing through division of labor (among multiple processing elements)
    - Additional datapath; memory mapped fun; coprocessing

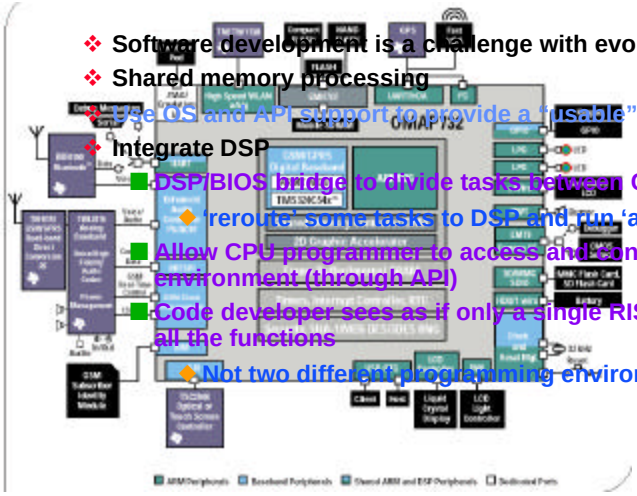
TI OMAP

- ‘Platform’ = Processor + Software + Support
- Scalable processors:
  - For applications, communications
- Software from application to system software
  - DSP libraries; J2ME, Linux, MS WinCE, Palm, Symbian

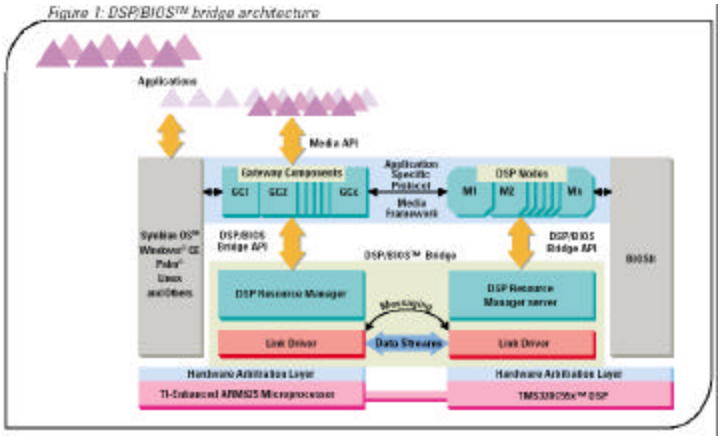


OMAP Hardware

- Multiple (heterogenous) processors (on-chip)
- Software development is a challenge with evolving processors
- Shared memory processing
- Use OS and API support to provide a “usable” programming model
- Integrate DSP
  - DSP/BIOS bridge to divide tasks between CPU, DSP
    - reroute some tasks to DSP and run ‘asynchronously’
  - Allow CPU programmer to access and control DSP runtime environment (through API)
  - Code developer sees as if only a single RISC processor is doing all the functions
    - Not two different programming environments.



OMAP Software



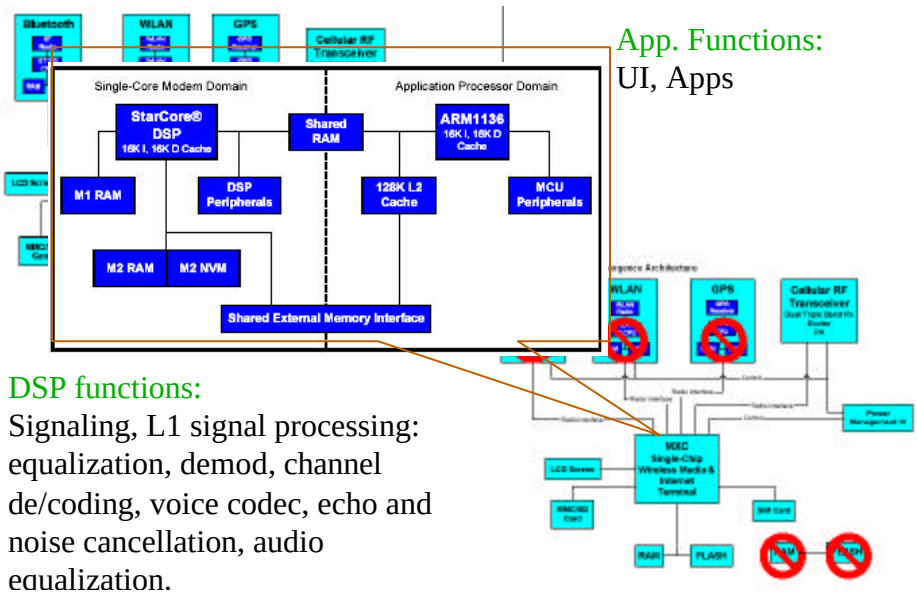
Intel PCA

- Primarily wireless handheld clients
  - PDAs, handsets, tablets, PCS phones,…
- Common hardware platform
  - Xscale and other comm/networking processors
  - Strong separation of communications and applications
    - Comm: DSP, or microcontrollers running RTOS
      - Cellular and 'baseband' processors
      - DSP using ADI, Intel version called Micro Signal Architecture (MSA)
    - App: Xscale running General Purpose OS
- Functionality separation also means separate development platforms
  - Use APIs and services between the two (comm+app) pieces of software
  - Multiple OS, air interfaces

Motorola MXC

- ARM core + StarCore DSP on a shared memory bus
- Radios: WLAN, BT, GPS
- Air Interfaces: GSM/GPRS, EDGE, WCDMA
- Follows Intel PCA in separating comm from app
  - Protocol stack completely in DSP
- Security as a platform differentiator

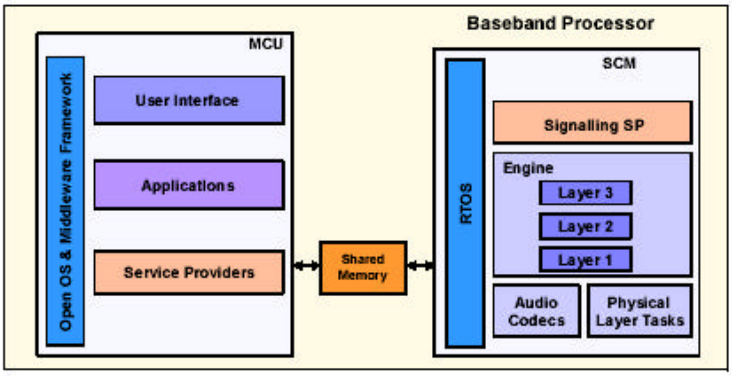
MXC Hardware



App. Functions:
UI, Apps

DSP functions:
Signaling, L1 signal processing: equalization, demod, channel de/coding, voice codec, echo and noise cancellation, audio equalization.

MXC Software



Summary, Conclusions and Discussions

Hardware-Software Co-design of Embedded Systems: Summary

- Embedded Systems are widespread.
  - Growth in $ volume, units, and system complexity
- Embedded systems have heterogenous components
  - Designs involve many aspects, steps, and disciplines
- These components need to be
  - modeled (computational models, specification languages)
  - analyzed (coverification)
  - simulated together (cosimulation, speedup => emulation)
    - for behavior validation, and
    - performance validation
  - their performance analyzed
  - mapped into HW & SW
    - (partitioning, scheduling, cosynthesis)
- Software is a major (and growing!) cost and bottleneck.

Co-specification

- Co-specification is important, but not mature;
- Description/Specification techniques can be domain specific;
- is still in the evolutionary stage: lots of options, no single "universal, industrial strength solution"
- Many of the underlying computational models are maturing
  - New models and algorithms are needed for some of the multimedia applications.

Co-simulation

- Simulation is very important at all levels of the design.
- Enables early analysis of the system behavior & performance characteristics, and permits HW-SW trade-off analysis.
- Techniques used include:
  - C/C++ models at a higher system level
  - VHDL models at a lower level
  - Hardware simulation models at the gate & transistor (switch) level.
- Main bottlenecks:
  - Time to develop models
  - Complexity of models and simulations needed
  - Time/complexity of developing software that runs on the HW models.
- These bottlenecks impede the degree of "serious" design-space exploration that can be done.

Emerging techniques (cosimulation)

- Richer, more flexible co-simulation environment;
  - These will enable:
    - Modeling of systems with heterogeneous system components (e.g., digital HW, analog, software, micro processors, DSPs)
  - Reduced overhead in "interfacing" the models for different parts of the system;
  - Libraries
    - A rich set will eventually enable a suitable "mix-and-match" strategy to work reasonably well to a first approximation, thus reducing the time spent in developing models.
    - The need to model and perhaps even design ASIPs and ASICs will be reduced.

Remarks

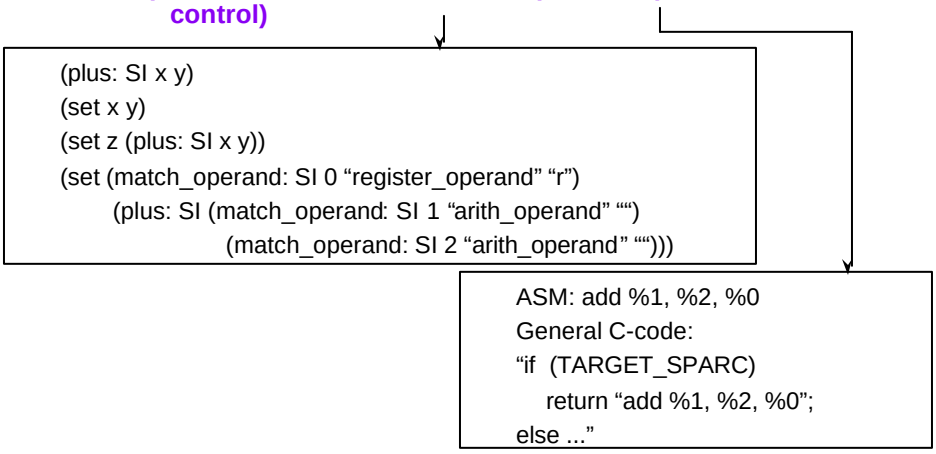
- Embedded Computing Systems are driven by the proliferation of computing elements in system design.
- Design of ECS requires good modeling, simulation, and design tools
  - can't program a computer without a compiler
- Design tools currently fall into separate categories for hardware and software tools.
- System design tools define a new growth area for system architects and CAD developers.

Appendix: Machine Description Examples

Gcc: MD w/ Architecture Only

- **Gcc RTL format**

  (define_insn, "name", RTL-template, output-control)



MD with Organization

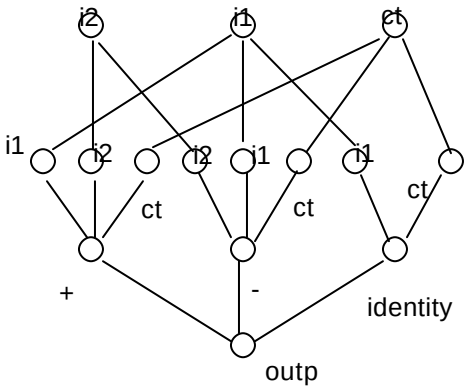
- **MIMOLA (Marwedel, MICRO-17, 1984)**
- **Rimey**
  - **Architecture for ASSP**
  - **Irregular datapaths and horizontal uCode**

Mimola RTL Structure

- All register transfer (RT) modules
- RT operations and interconnect
- Compiler produces uCode for a given application (in Pascal-like language) and an RT structure
  - Map resource conflicts to instruction field conflicts
- Machine description compiled into M-graphs
  - Inputs as leaves, Output root
  - One tree of depth 2 for every operation.

Example

```
MODULE Processor (OUT res:(15:0); IN ClockIn:(0));
  STRUCTURE AtRtLevel OF Processor IS
   TYPE
      word = (15:0);
      Instr = FIELDS
            Alu : (1:0); Mux : (2);
            R0 : (3); R1 : (4);
            R2 : (5); Imm : (21:6);
            NextAddr : (37:22);
            END;
   PARTS
      Alu : MODULE AluT(IN i1, i2: word;
            OUT outp: word; FCT ct: (1:0));
             BEHAVIOR AtRtLevel of AluT IS
               BEGIN
                case ct OF
                   %00 : outp <- i1 + i2 AFTER 10;
                   %01 : outp <- i1 - i2 AFTER 10;
                    %10 : outp <- i1 AFTER 5;
               END; END;
....
   CONNECTIONS:
```



RL (Rimey & Hilfinger, '88)

- Architecture
  - Data-path
  - Open horizontal uCode
- uCode avoids instruction encoding/format issues. Though later optimization is always possible for a given application.

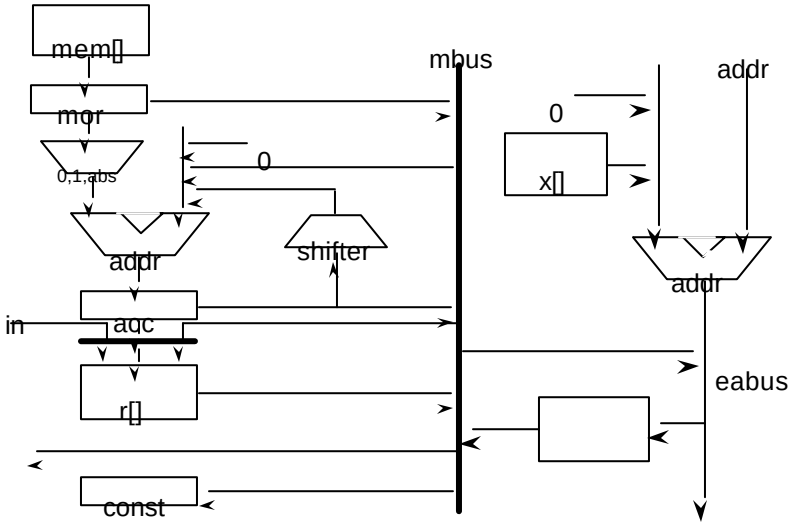
RL Usage

- *Inputs:* An application in Silage, a Data path
- *Output:* Compiled application
- If compiled application OK goto hardware synthesis else modify DP and retarget compiler.
- Output quality strongly depends upon types of functional units and their interconnections.

Architecture

- Three components:
  1. Data-path: integer unit and address unit
  2. Boolean unit: logic array
  3. Control unit: program sequencer
- Data-path consists of register, register banks, functional units (typically w/ saturation arith.)

Data-path



Boolean & Control Units

- Boolean Unit
  - Devoted to logical operations
  - Evaluate Boolean expressions (ops on Bool types)
  - Inputs from DP (sign bit) or external
  - Outputs as cc or to external
- Control Unit
  - Generate addresses for program memory
  - PC, state machine to affect PC
  - Branch addresses
  - Inputs from BU, DP or external

RL Micro Operations

- Transfer micro-operations

  x = y
  x = y[l]
  x[l] = y
  x = l
- Function micro-operations
  - Indirect read and write (x = y[z]; x[z]= y)
  - Indexed read and write (x=y[l+z]...)
  - Port input and output
  - Arithmetic
  - Shift

RL Machine Description

- Declaration of data-path nodes
- Implemented micro-operations
- Example:

```
#define bus node: delay = 0
#define reg node: delay = 0
#define file reg: bank

bus addr, xbus, xsum, xsign, eabus
micro addr = Immediate
```

- Micro-operations impose scheduling constraints
  - Two uops may not write to the same node simultaneously.

RL Machine Description

- Micro operations

```
micro addr = immediate
micro xsum = addr + xbus
micro x[N] = eabus
```

- Constraints
  - Reserve: a node implicitly modified by a uop
  - Grab: resource required
  - Sequence: ordering of grab operations
- Output: sequence of uops. Each macro instr is a collection of uops.

Code Generation

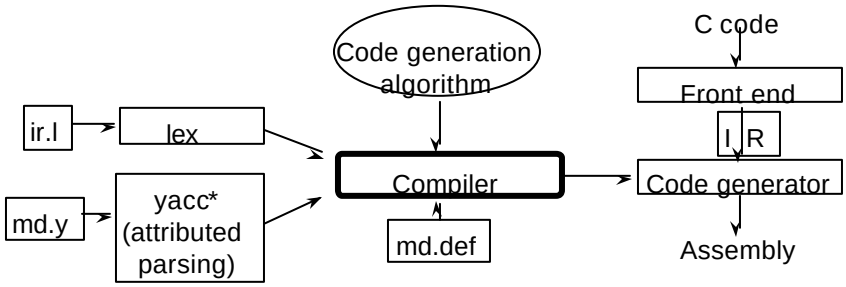
- Mapping of source language data types to machine data types/formats
- Storage allocation and binding
- Instruction selection
- Machine-specific optimizations
  - Special addressing
  - Special instructions (e.g., AOBLEQ on VAX)

Code Generation

- Three types
  1. Interpretive (with case analysis)
     - generate code for a virtual machine
     - expand generated code into real target code
     - use hand-written interpreters to implement mapping
     - example: Pascal P-code, Open boot F-code
  2. Pattern matching
     - PM inplace of interpretation: Heuristic or Parsing
     - separate MD from code generation algorithm
  3. Table driven

Attributed Grammar

- Code-generation algorithm independent from the target machine
- Machine described using YACC grammar
- Produces code generator
- Intermediate representation, IR



IR

- C code

```
void Example (int n)
{
      int i;
      i = 0;
      do {
          i = i + 1;
      } while ( i <= n);
}
```

- IR

```
:Example^1 {
  :i^local^integer^1
  :=i0
  LBL
      :=i  + i  1
      <= i n LBL
}
```

- IR variables assigned before code selection.

Productions using Attributed Grammar

- Three types:
  1. Instruction selection productions
  2. Addressing mode productions
  3. Transfer production
- Instruction selection
  - Code generator consists of a set of transition tables and a driver for these tables.
  - The driver is an automata that parses the IR form
  - Instructions are selected during parsing
  - For a given machine, generate transition tables directly from affix grammar description.