# SystemC abstractions and design refinement for HW-SW SoC design

Dündar Dumlugöl

Vice President of Engineering, CoWare, Inc.

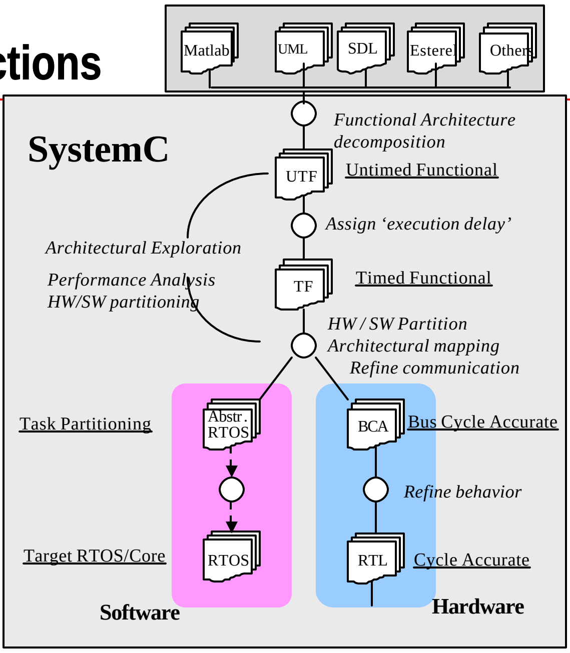
CoWare™

## Overview

- SystemC abstraction levels & design flow
- Interface Synthesis
- Analyzing system performance
- Examples
- Key modeling paradigms & semantics
- Conclusions

CoWare™

## Abstractions



CoWare

## Simulation speed for different abstractions

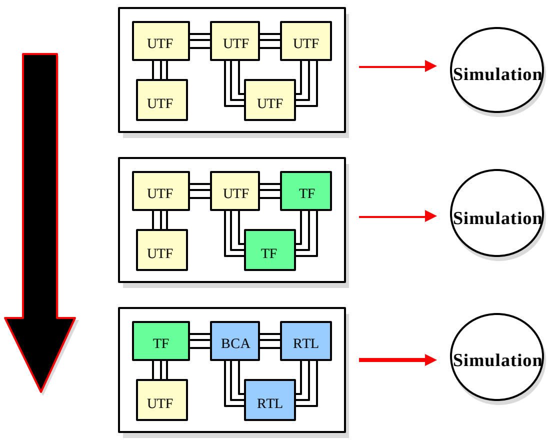
| | Simulation speed | Real time (s) simulated in 10h | Comments<br>WCDMA test platform @100 Mhz |
|---|---|---|---|
| SoC | 100 MHz | 3,300,000x | Real chip |
| Emulation | 300 kHz | 10,000x | Estimated |
| UTF | 3 MHz | 100,000x | |
| Timed Trans. | 300 kHz | 10,000x | Minimum required for SW/firmware developer |
| CATrans. | 30 kHz | 1,000x | Minimum required for platform developer |
| BCA | 300 Hz | 10x | With ISS for core |
| RTL | 30 Hz | 1x | No ISS |

CoWare

## Gradual Refinement of the Design

Key to the methodology is that a design may be refined in a gradual step-wise fashion, rather than in one giant step… it need not be "all or nothing".

***Details added to portions of the system.***



## Algorithmic level

- ❑ Algorithm described in a sequential language such as C
- ❑ Functional structure has no bearing to implementation architecture
- ❑ Example: JPEG/ MPEG algorithm development
- ❑ Design solutions: Matlab, SPW, Cossap, etc

## Untimed Functional (UTF)

- An architectural/structural decomposition of the system into functional blocks which are structurally interconnected over abstract communication channels
- Models control- and data-flow at an abstract functional level with abstract data types
- Enables easy refinement to HW-SW architecture
- Re-use of test benches & simulation results as references in subsequent design refinements

## Sequential (in-lined) process execution

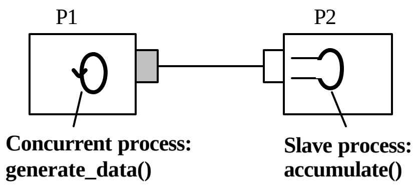
- A process triggers execution of a slave process in-lined with itself
- Emulates function execution as in C programs

## Example: sequential C program

```
static int sum = 0; // state variable

void generate_data() {
    for (int i = 0;  i < 10;  i++) {
        accumulate(i) ;
    }
}

void accumulate(int in1) {
    sum += in1;
    cout << "Sum = " << sum << endl;
}
```

## Same example with in-lined process execution

Sequential behavior of C-code is preserved
Separation in 2 processes allows
structural decomposition and re-use



Structure is
key for re-use

Transaction triggers slave process execution

Equivalent to function call but without function pointer

## Example: producer module

```
SC_MODULE(producer) {
    sc_outmaster<int>  out1;


    void generate_data() {
        for (int i = 0;  i < 10;  i++) {
            out1 = i ; // this will invoke
                           //the slave;
        }
    }

    SC_CTOR(producer) {
        SC_METHOD(generate_data);
    }
};
```

CoWare

## Slave process

```
SC_MODULE(consumer) {
    sc_inslave<int>     in1;
    int   sum;      // state variable

    void accumulate() {
        sum += in1;
        cout << “Sum = “ << sum << endl;
    }

    SC_CTOR(consumer) {
        SC_SLAVE(accumulate,in1);
        sum = 0;    // initialize
    }
};
```

CoWare

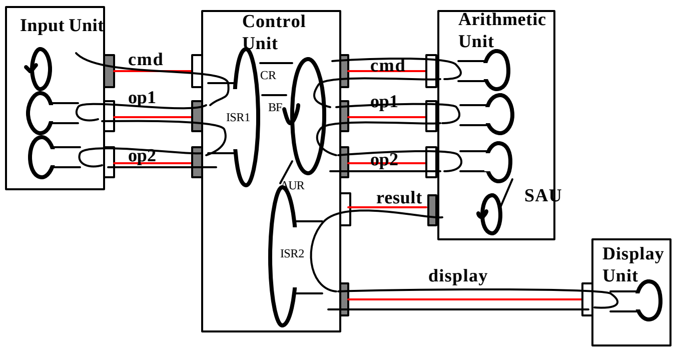
## Structural module

```
SC_MODULE(top) { // structural module
    producer *A1;
    consumer *B1;
    sc_link_mp<int> link1;

    SC_CTOR(top) {
        A1 = new producer(“A1”);
        A1->out1(link1);
        B1 = new consumer(“B1”);
        B1->in1(link1);
    }
};
```

## Sequential (in-lined) process : summary

❑ SC_SLAVE process executes in-line with another process
- Slave process has a single slave port that connects to a sequential link (sc_link_mp)
- Caller process connects with a master port to a sc_link_mp
- Invocation occurs by master accessing master port
- Slave process can call other slaves

## Untimed Functional example

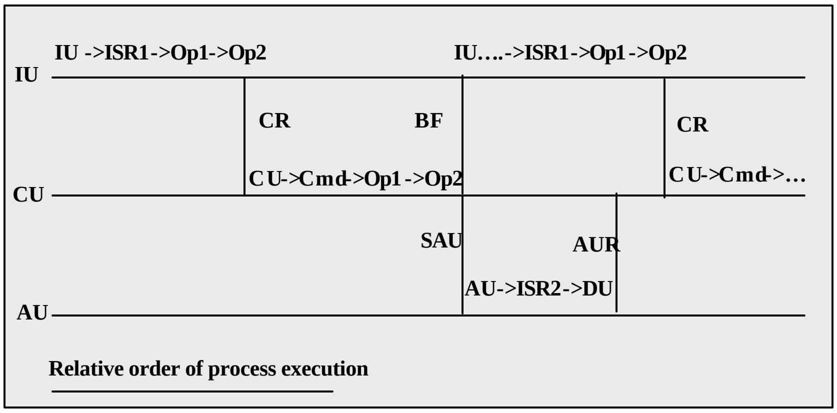


**System is described in terms of 3 concurrent threads with in-lined slave processes. The concurrent threads (IU, CU, AU shown as loops) are synchronized over events: CR (Command Ready), BF (Buffer Free), AUR (AU Ready), SAU (Start AU)**

## What the example does

- A simple system with an Input Unit (IU), a Controller Unit (CU) and an arithmetic co-processor (AU) unit takes a triplet consisting of an arithmetic operator and two operands, calculates the result and displays it.
- The control flow is as follows: An Input Unit generates triplets (can be a test bench) asynchronously. Sends the triplet to the Control Unit (CU) if the latter's input buffer is free. CU writes the triples into the register of the Arithmetic Unit (AU) if the latter is not busy and generates a buffer free event. This will start the AU. The CU then waits for an AU ready event. The AU calculates the result and sends it back to the CU (ISR2) which then dispatches the result to the Display Unit and generates an AU ready event.

## Process execution order for the example



Vertical arrows depict synchronization events between the 3 concurrent threads
Horizontal lines show in-lined process executions within each concurrent thread (IU, CU, AU)

## UTF summary

- Combines sequential and concurrent processes
- Transaction & computation order is modeled, time is not
- Allows abstraction of (initially unwanted or unknown) implementation details : data/ control flow, clocking, concurrency, pipelining, time
- Assumptions: infinite resources which are infinitely fast

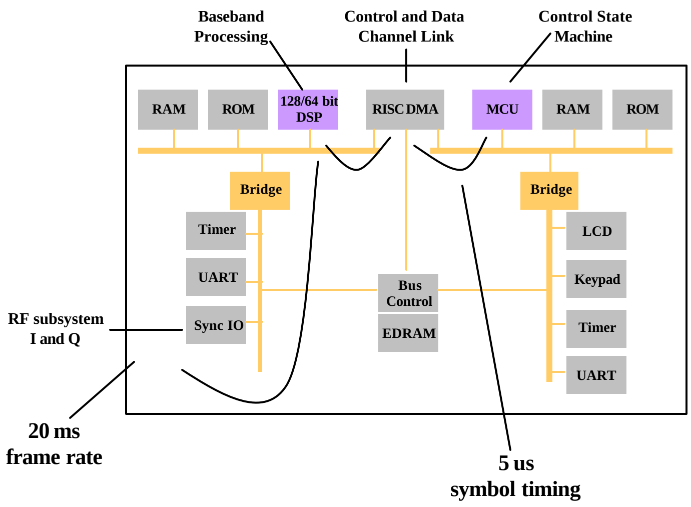
## Benefits of UTF

- Allows control/data flow description before timing is available
- Can be easily partitioned into SW-HW
- Very compact descriptions of complex data- and control-flows (10x-100x less code)
  - No FSM's required for behavior or communication
  - Abstract data types
- 5 orders of magnitude faster simulation than at RTL level

## Timed Functional Abstractions

- Pre-partitioned (HW-SW)
  - Add timing to the UTF system in the form of timing estimates, timing constraints or time budgets
  - Time delays are in absolute time units
  - System is timed but not clocked
- Post-partitioned (HW-SW)
  - Add timing to the UTF system for HW-SW partitioning, architecture exploration and optimization (bus hierarchy, memory architecture, cache size, etc)
  - Map SW blocks to cycle accurate models of processor core & bus/ memory architecture
  - HW is either untimed (assuming fast enough to not affect the critical path) or timed with estimated cycle delays

## Optimizing Wireless Handset architecture



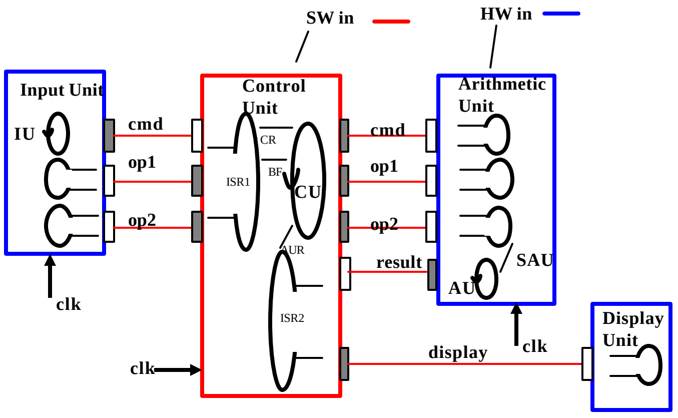
## Optimizing SOC bus architecture

- Bus architecture - choosing the correct bus architecture is one of the biggest problems in SOC design.
- Parameters
  - Standard (e.g. AMBA) versus custom
  - Arbitration (scheduling algorithms)
  - Bus width
  - Bus pipelining (wait state versus split transaction)
  - DMA
  - Hierarchy (e.g. system bus versus peripheral bus)

## Optimizing SOC memory architecture

- ❑ Memories - choosing memory components and hierarchy can have large affect on system performance
  - cache, SRAM, DRAM, flash, etc.
  - size and hierarchy
  - power consumption
  - wait states and system throughput

## AU example after HW-SW partitioning



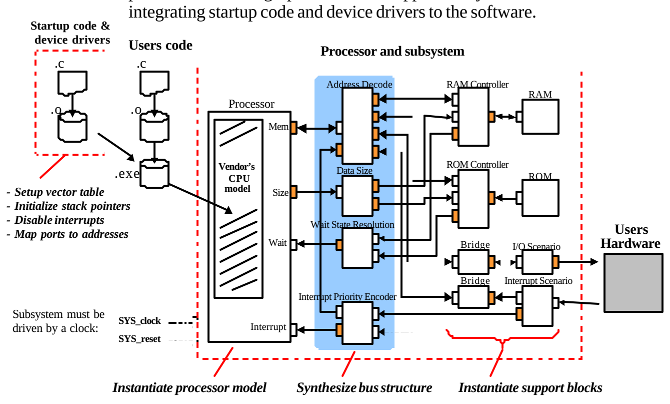
**Map the processes in the Control Unit block to SW running on a MCU core. Use interrupt scenarios on cmd (from IU) and result channels with ISR1 and ISR2 as respective Interrupt Service Routines.**

## Interface Synthesis (IFS)

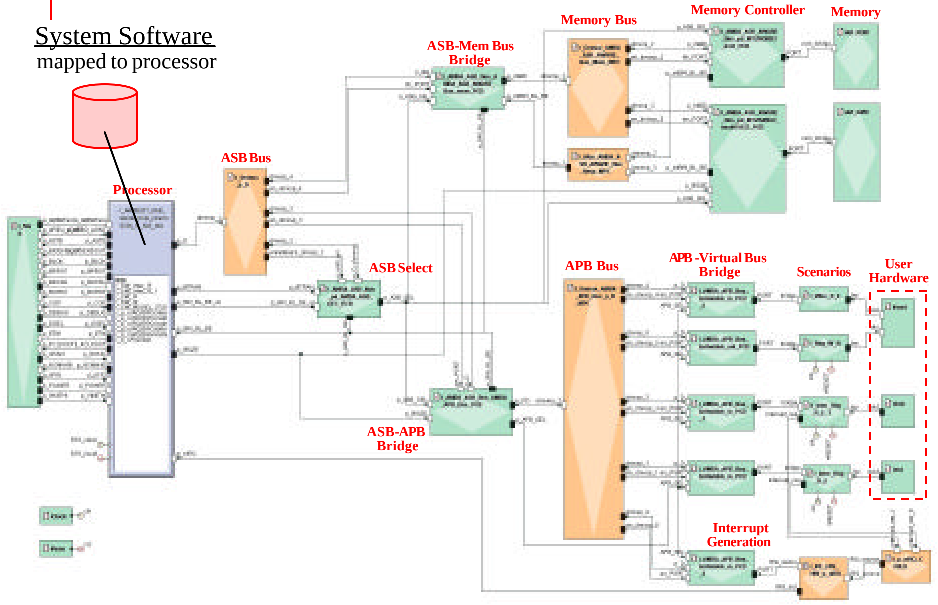
- What you get from IFS
  - Ability to configure your platform with multi-master, multi-layer, multi-bus and hierarchical memory architectures
  - Cycle Accurate processor model, e.g. ARM926-EJS CCM model
  - Fast Cycle Accurate model of the above platform
  - Automatic mapping of your UTF system to this platform including generation of boot code, device drivers, address maps, address decoders in the bus and bus bridges
  - IFS is an enormous productivity tool since you don’t have to know details about the core, its bus protocol, pin timing, etc. because the IFS tool has knowledge about the core and its platform.
  - You can now very easily (in a matter of hours) change your HW-SW partition at the UTF level or choose a different core or modify your platform, regenerate the cycle accurate model of the system and analyze the impact of your changes on system performance

## Interface Synthesis

Interface Synthesis divides the system into distinct hardware and software portions, connecting a processor and support subsystem to the hardware, and integrating startup code and device drivers to the software.



## Sample IFS-generated System



## Analyzing System Performance

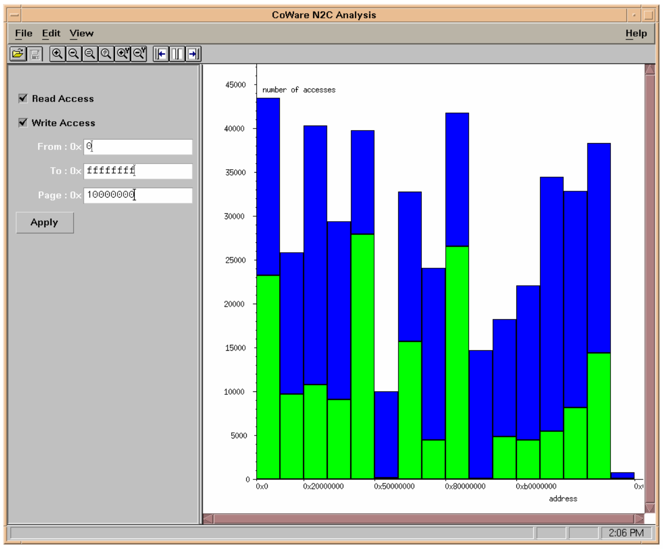
- Architectual analysis
  - Analyze token-based performance model
  - Study throughput and bottlenecks
  - Look at bus switching and cache usage to reduce power
  - Optimize bus & memory architecture
- Functional analysis
  - Look at system response and task scheduling
  - Analyze complexity to drive partitioning
  - Profile software for optimization

## Analysis views (HW/SW)

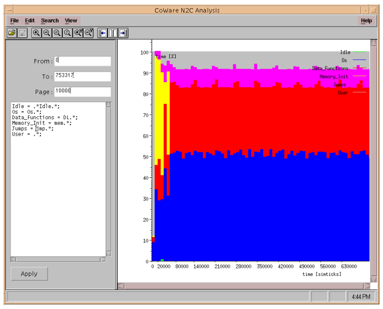
- HW Views
  - Gantt chart
  - Call Graph
  - Operation Count
  - Variable Info / Operator Count.
  - Variable Tracing
  - Function/Thread Call Stack

- SW Views
  - Gantt chart
  - Function/task call stack
  - Memory access / time
  - Memory page access
  - CPU load
  - Memory access + SW trace
  - Memory map counts

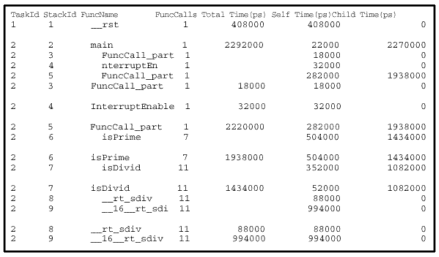
## Memory access patterns



## CPU loading/SW profiling



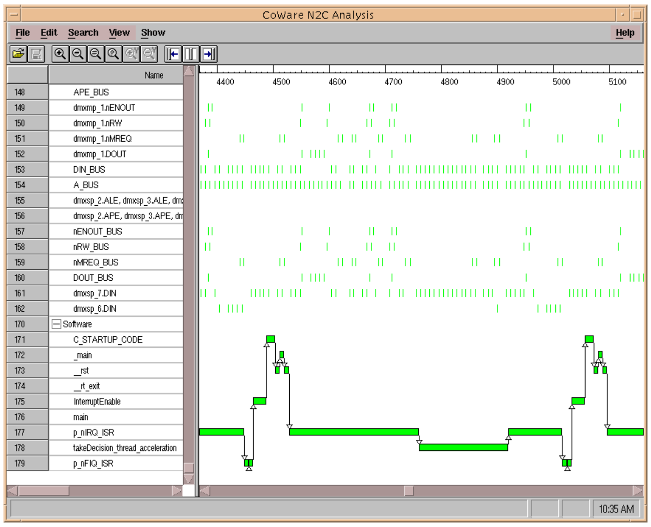
```
Idle = .*Idle.*;
Os = Os.*;
Data_Functions = DL.*;
Memory_Init = mem.*;
Jumps = Jmp.*;
User = .*;
```



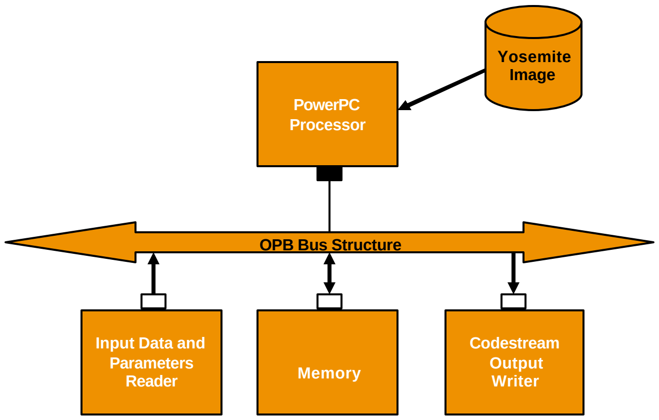
```
TaskId StackId FuncName        FuncCalls Total Time(ps) Self Time(ps)Child Time(ps)
1       1       __rst              1         408000        408000              0

2       2       main               1        2292000         22000        2270000
2       3         FuncCall_part    1                        18000              0
2       4         nterruptEn       1                        32000              0
2       5         FuncCall_part    1                       282000        1938000
2       3       FuncCall_part      1          18000         18000              0

2       4       InterruptEnable    1          32000         32000              0

2       5       FuncCall_part      1        2220000        282000        1938000
2       6         isPrime          7                       504000        1434000

2       6       isPrime            7        1938000        504000        1434000
2       7         isDivid         11                       352000        1082000

2       7       isDivid           11        1434000         52000        1082000
2       8         __rt_sdiv       11                        88000              0
2       9         __16__rt_sdi    11                       994000              0

2       8       __rt_sdiv         11          88000         88000              0
2       9       __16__rt_sdiv     11         994000        994000              0
```

## Design results

- Gantt charts show performance results



## JPEG2000 VirtexII Pro (Xilinx) SW Platform



## JPEG2000 platform

- Architectural exploration focused on two alternatives
  - A) Implement JPEG algorithm with Wavelet transform in SW
  - B) Implement Wavelet algorithm in HW
- Architectural analysis reveals that bottleneck is not in the Wavelet algorithm as initially expected but instead in the code stream output writer due to memory accesses
- Without this analysis, RTL designers would have selected the wrong HW-SW partitioning
- Greatest gain in performance and power is achieved in architectural optimizations

## Modeling HW in the TF Abstraction

- Some blocks may remain untimed if their timing is not critical to the system function or performance (they're assumed to be fast enough)
- Other blocks are in UTF annotated with cycle delays
  - Concurrent HW threads must be attached to a clock
  - Slave processes get their clock tick from the concurrent thread that calls them

CoWare™

## SW design flow

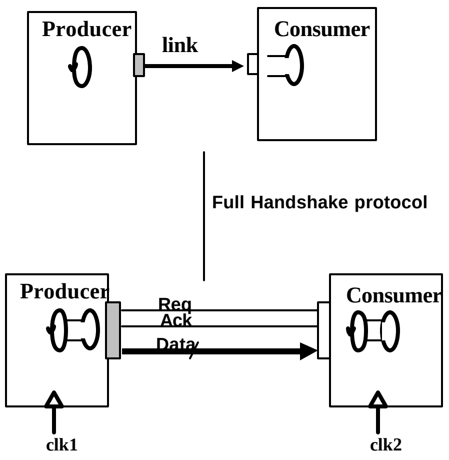
- Abstract RTOS level
  - Functional threads are partitioned into SW processes per core processor in a multi-core system
  - SW processes may be created dynamically
  - SW processes are scheduled using an abstract (generic) RTOS; scheduling policy and process priorities are defined
  - Inter-process communication is implemented (shared memory, semaphores, sockets, mailboxes, etc)
- Target RTOS level
  - Abstract RTOS calls are now mapped to a target RTOS with further refinement of communication and scheduling

CoWare™

## The RTL Implementation level

- IFS generates automatically RTL implementation of the platform from the specifications of the TF platform
- HW blocks are implemented at the RTL level as follows:
  - All HW (concurrent and slave) processes are implemented as clocked processes
  - Functional communication channels are refined to handshake bus protocols

## BCA example: full-handshake protocol



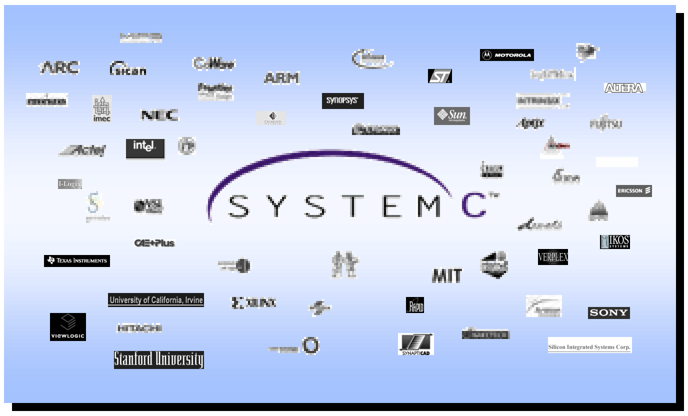
After in-lining of communication channel processes into the master and slave blocks

## Conclusions

- SystemC supports modeling abstractions for HW/SW co-design from Functional to RTL
- Abstraction levels can be mixed for gradual refinement
- Benefits
  - Fast design exploration & HW/SW partitioning
  - Enables synthesis of communication
  - Fast IP embedding & retargeting
  - Orders of magnitude faster than RTL verification

CoWare™

## SystemC$^{TM}$ Community



CoWare™

## Backup slides

- TF communication details
- Key modeling paradigms

## TF communication revisited

- Timed Transactional (TT)
  - Mostly for SW modeling
  - Delay is in absolute time units
- Cycle Accurate Transactional (CAT)
  - For HW modeling including Bus/ Memory architecture
  - Delay in clock cycles

## Cycle Accurate Transactional Modeling

- Models transactions on a bus between bus masters & slaves
  - Cycle accurate
  - Bus protocol specific
  - Arbitration, address, data cycles
- Protocol independent generic API allows re-use
  - Mapping to a bus protocol through protocol specific libraries (re-use)
  - Specify attributes of a transaction: data width, data size, burst, status, etc

CoWare™

## Benefits of CA Transactional modeling

- Platform architecture exploration and optimization
  - Bus hierarchy: number of buses, assignment of resources to buses
  - Memory hierarchy / size
  - Cache size
  - HW-SW partitioning
  - Analysis for bus throughput, bottlenecks
- Much less user code than in RTL

CoWare™

## Key modeling paradigms

- Model of hierarchy and re-use
- Model of
  - Process execution
  - Process communication
  - Process synchronization
- Model of Time

## Model of hierarchy and re-use

- All behavior is described in processes; statements in a process execute sequentially
- Processes are "encapsulated" in modules which have ports
- All external communication takes place exclusively over module ports (to enable module re-use)
- Inside a module processes can communicate over shared variables and internal channels
- Modules are interconnected through their ports to communication channels

## Model of communication

- Abstract functional
- Transactional
  - Timed
  - Cycle accurate/ bus protocol accurate
- Bus Cycle Accurate communication
  - Cycle & Pin accurate

## Model of time

- Untimed
- Timed (but not clocked)
- Clocked (cycle accurate)

## Model of process execution

- Concurrent process execution
  - Communication & synchronization
- Sequential (in-lined) process execution
  - Abstract point-to-point & multi-point sequential communication

## Concurrent processes

- SC_THREAD process
  - Thread type (can be suspended)
  - Process execution is triggered from its sensitivity list
  - Process can suspend at a wait() statement & resume
  - Has a stack to store its state
  - Can have static and/ or dynamic sensitivity
- SC_METHOD process
  - Method process (functional call, can not suspend)
  - Process execution is triggered from its sensitivity list
  - Has static sensitivity only

## Concurrent process communication

- ❑ Concurrent processes communicate over a concurrent channel (=user defined module)
  - Shared variables for intra-module communication
  - Signals for intra- & inter-module communication
  - Signals are a special form of concurrent communication channel (for HW, supports evaluate-update paradigm)

## Model of process synchronization

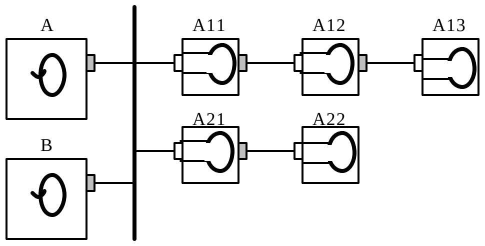
- ❑ Concurrent processes synchronize over events

  sc_event my_event; // fundamental sync object
- ❑ Two types of process sensitivity:
  - Static : wait() without arguments; sensitivity in constructor
  - Dynamic: changes during simulation
    - ❑ wait(ev1 | ev2 | …) // on any one or more events
    - ❑ wait(ev1 & ev2 & …) // on all events
    - ❑ event.notify(0) // next delta cycle
    - ❑ event. notify(delay); // timed notification
    - ❑ event.notify(); // immediate notify, not in v1.2
- ❑ Mutex & semaphores built as libraries on core sync objects

## Sequential (in-lined) process execution

- A process triggers execution of a slave process in-lined with itself
- Emulates function execution as in C programs

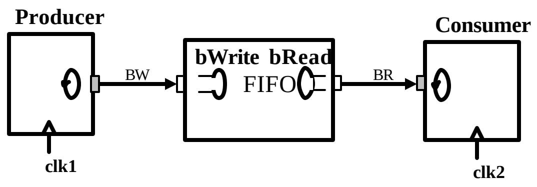
## Multi-point sequential communication



Abstract model for a bus communication:
A transaction by a master causes all slaves on the bus to be called;
Based on index value the slaves decide whether to respond or not
Multi-point will be later refined into a real bus communication
with address decoding & arbitration, concurrent behavior

## Sequential & concurrent communication combined

BW: blocking write with time-out
BR: blocking read with time-out



FIFO channel