Embedded System Design and Modeling

EE382V, Fall 2008

Lecture Notes 4

System Design Flow and Design Methodology

Dates: Sep 16&18, 2008
Scribe: Mahesh Prabhu

SpecC:

Import Directive:

This is different from a *#include* directive which is used for sharing definitions from a different file. 'Import' allows the user to add behaviors and channels that were defined earlier to be used in the current design. The predefined behaviors and channels may not be available in source format but may be given in the form of 'sir' files. This makes the import directive ideal for sharing IPs. Another difference between import and include directives is that import adds a behavior/channel exactly once in the design, no matter how many times the directive is used on the same behavior/channel. But 'include' work differently and need guards to operate without compile time errors if we include the same file twice.

Usage:
import "s1"; // This looks for *s1.sir* file first and if this is not available looks for *s1.sc*

Import works hierarchically, so that imports in the .sir and .sc files that are imported are further imported.

Notes:

'note' is used to add some additional meta information to the design specification. The notes do not impact the design and a 'note' is not used by SpecC compiler in any special way. Hence the use of notes does not affect simulation. However this annotated information may be used by tools down the flow for say synthesis.

Q: Why is SpecC used?

SpecC is used for modeling the entire system which includes H/W and S/W, so that all the functionality of the entire system is captured in a single language. SpecC can also be used as a common language starting from system specification to the final gate level implementation. The use of multiple languages in the flow brings in the need for more sophistication in the tools being used in between (because now they need to handle more language semantics) and also requires the engineers to be trained to use multiple languages.

Q: Is SpecC faster than C/C++ implementation?

This highly depends upon the implementation of the simulation kernel. C/C++ would need some additional help form libraries like posix threads to capture parallelism and other semantics, however the speed would only be determined by the implementation and not the language since C/C++ and SpecC would capture the same level of detail. If we had compared system level specification and RTL implementation, we can clearly see that RTL simulation is slow.

System Design:

We start from a system level specification which is purely behavioral with zero cycle time simulation semantics. This is finally converted into a completely structural description with clear s/w and h/w boundaries and also the timing being captured at a clock cycle accurate level. This transformation from the initial untimed specification to the final cycle accurate level design takes place through phases involving iterations and decision making adding more and more detail in each phase. Hence the semantic gap between the untimed model and the cycle accurate model is too large to be done in a single step or by a single tool.

Use of IPs:

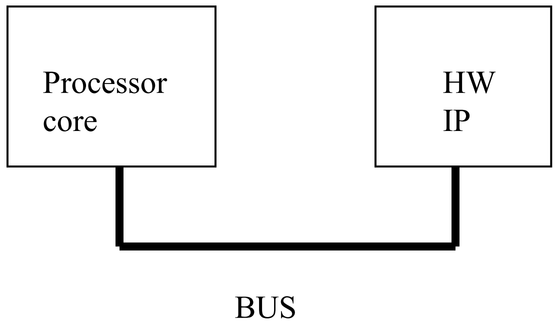
IP providers like ARM give both the RTL (in Verilog/VHDL) and Instruction Set Simulators (ISS) (in the form of dynamically loadable/statically linkable binaries with the appropriate header files) to the designers. Initially at the system design phase we do not have much use of the RTL. However, the ISS can be used during this phase. Our design can be connected to the IP through a bus and the communication with the supplied library happens through API's provided along with the IP. The SpecC code for this would look something like this.

Iss.sc:

```
#include <iss.h>
behavior ISS {
        ...
        void main(void)
        {
                ...
                while(1) {
                        // Set up bus here with APIs
                        iss_exec();
                        //wait for 20. This adds some delay
                } // Perform Execution for as many cycles that is required
        }
        ...
};
```

IP Problem: How to use exiting IPs that were designed years ago which are stable and are being used currently?
These IPs are written in RTL and there are tools for RTL→ SystemC conversion. So this is one way to solve the problem. Or the IP provider needs to ship a C/SpecC/SystemC model of the IP (for simulation) together with its RTL (for implementation).

There are many simulators that provide mixed level of simulation where parts of the design can be in RTL (IP) and the rest of the design is in C/C++/SystemC(/Spec?). The simulators provide APIs for communication between the RTL and rest of the design to perform a full system simulation. This is another way to get around the IP problem.



Design Methodology:

Design Methodology gives a step by step process that we can use to transform an untimed specification into a cycle accurate structural (RTL) specification. Lecture-4/Slide-8 shows a top down design flow and the different stages in the transformation process. The different intermediate timing models and structures that we might have to work with are also shown.

We notice that computation mapping/refinement is done before communication mapping/refinement. This is because computation is often the focus of most systems and the processors that we choose will determine the type of bus required thus simplifying the communication mapping/refinement phase. However when the focus is on the communication aspect of the system for performance reasons, like a wireless router, the communication mapping takes place before the computation mapping.

SpecC code outline of the model show in Lecture-4/Slide-11 (with the focus on communication) is given below. The design has communication through ports and events which is commented out and replaced by standard communication channels.

```
behavior B1(out v1) {
        void main (void) {
                ...
                v1 = ..;
                ...
        }
};

behavior B2B3(in v1) {
        //event e2;
        //float v2;
        c_double_handshake c2; // this replaces the above two in communication
refinement phase
        B2 b2( v1,  c2 /*this replaces v2 and e2*/);
        B3 b3( v1, c2 /*this replaces v2 and e2*/);
        void main(void) {
                par {
                        b2;
                        b3;
                }
        }
};

behavior B2( in v1, i_sender s2/*this replaces out float v2, out event e2*/) {
        void main(void) {
                ...
                // v2 = ..;
                // notify e2;
                s2.send( v2); // this replaces above two statements
        }
};

behavior B3( in v1, i_receiver s2/*this replaces in float v2, in event e2*/) {
        void main(void) {
                ...
                // wait( e2);
                // ... = v2;
                s2.receive( v2); // this replaces above two statements
        }
};

behavior main() {
        int v1;
        B1 b1(v1);
        B2B3 b2b3(v1);
        void main(void) {
                seq {
                        b1;
                        b2b3;
                }
        }
};
```

Note that *in* and *out* are important while specifying the ports of a behavior because it decides if writing on a specific port is allowed or not.

Computation Refinement:

Processing Element (PE): These include execution units or storage units. We can have just memory as a processing element or just a processor core without any memory. The latter two examples are extreme case and most PEs will be a combination of the two.

Synchronization:

If multiple processes are running on the same PE we need some sort of scheduling mechanism to multiplex the use of the common resource.

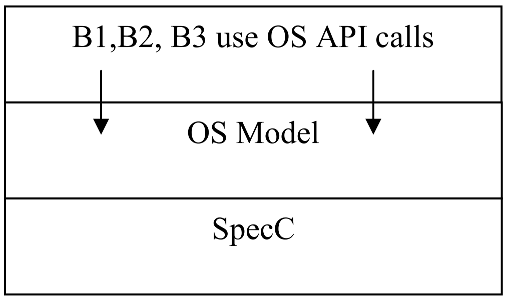
Consider the earlier example (see Lecture-4/Slide-16). We partition the behaviors that run on 2 PEs say PE1 and PE2 with behavior B1 & B2 scheduled on PE1 and behavior B3 on PE2. The problem here is that B1 & B3 were running sequentially and in the PE allocation there is a possibility they may run in parallel. To ensure sequential semantics, or the execution ordering in general, we need to have additional communication elements (variables, events, channels etc).

Timing:

Timing is as important as functionality. Delays should be put in every basic block in the behaviors using *waitfor* statements to simulate the timing aspect of the design even though this increases the actual simulation time.

Scheduling:

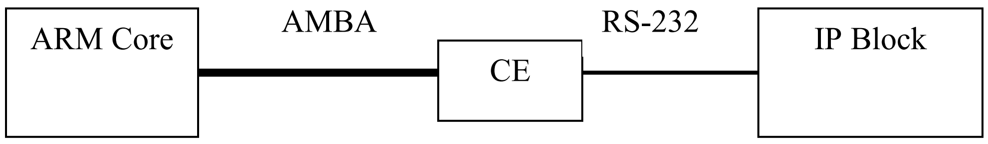
For scheduling processes within each PE, we would have calls in the behaviors to the OS. The OS model would be inserted into the system like a channel which provides API's like create_task(), fork(), join(). The scheduling schemes that are available include schemes like priority and round robin.



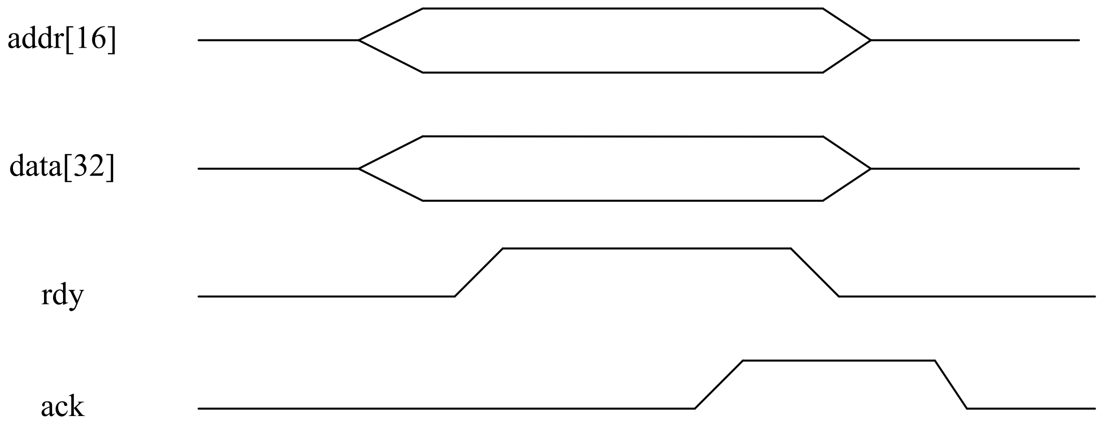
Communication Refinement:

Inlining is converting protocol stack into wires (i.e we flatten out the communication part of the system).

A single channel can exist over multiple buses; this aspect of the channel depends on what it is design to do. Consider an ARM core communicating with an IP block. The ARM sits on an AMBA bus and the IP communicates using RS-232. Hence there needs to be a CE in between which is able to do this translation. We see that there will be 2 buses in the model, one the AMBA bus and the other a RS-232 interface. For a SpecC model we can remove these two buses and replace it with a single channel with appropriate APIs and which internally simulates the working of the two buses to the required granularity.



Consider a channel which is expected to maintain some timing constraints. Shown below is the timing diagrams and the SpecC code outline for specifying those constraints.



```
channel Db1Hs implements master, slave
{
        signal bit[16] addr;
        signal bit[32] data;
        signal bit[1]  rdy, ack;

        boid masterWrite( bit[16] a, bit[32] d) {
                do {
                        t1: addr = a;
                        data = d;
                        waitfor(10 /*some typical estimated value*/)
                        t2: rdy = 1;
                        wait (rising ack);
                        /* this is similar to having while(!ack) wait(ack); */
                        /* but: if ack is already '1', they behave differently! */
                        t3: rdy = 0;
                        addr = 0;
                        data = 0;
                        wait(falling ack);
                } timing {
                  /*use range statements to give the time constraints between labels tx */
                }
        }

        void slaveServeWrite( bit[16] a, bit[32] *d) {
          do {
                do {
                        wait( rising rdy);
                } while( addr != a);
                t1: *d = data;
                waitfor(5);
                t2: ack = 1;
                wait(falling rdy);
                t3: waitfor(5);
                t4: ack = 0;
          timing {
                  /*use range statements to give the time constraints between labels tx */
          }
        }
};
```

Transaction-Level Modeling (TLM):

Channels can be defined at the pin level with signals operating at the clock cycle level with timing details or we can have channels that are defined at a “transaction” level where we are only worried about what information is exchanged without considering the timing details of the exchange.

Consider the example of a master communicating with a slave device through a bus. This is shown in Lecture-4/slide-27. We can have a detailed SpecC channel as above.

Here we notice that the bus protocol is specified in great detail down to the timing level. This increases the number of events that the SpecC simulator has to handle and slows down the simulation. The simulation speed is directly proportional to the number of contexts switches which in turn is directly proportional to the number of events. Instead of this we can have an implementation which removes the bus details at the wire level i.e what happens at each clock cycle. This reduces the number of events to the simulator kernel making the simulation faster. Such a bus model is called a Transaction Level Model (TLM).

The earlier Master Slave protocol would have the following SpecC TLM:

```
Channel DblHsTLM {
  unsigned int addr, data;
  event rdy, ack;

  void masterWrite( bit[16] a, bit[32] d) {
          addr = a;
          data = d;
          waitfor(10);
          notify rdy;
          wait ack;
   }

   void slaveServeWrite(bit[16] a, bit[32] *d) {
          do {
             wait rdy;
          } while (addr != a);
          *d = data;
          waitfor(10);
          notify ack;
};
```

We can similarly have TLMs for other buses. Consider the communication between an Instruction Set Simulator and memory. Here the loads and stores from the processor are handled by the TLM which internally follows the bus protocol.