System Design for DSP Applications in Transaction Level Modeling Paradigm

Abhijit K. Deb, Axel Jantsch, Johnny Öberg
Department of Microelectronics and Information Technology
Royal Institute of Technology, 164 40 Kista, Sweden
Email: { abhijit | axel | johnny } @ imit.kth.se

ABSTRACT

*In this paper, we systematically define three **transaction level models** (TLMs), which reside at different levels of abstraction between the functional and the implementation model of a DSP system. We also show a unique language support to build the TLMs. Our results show that the abstract TLMs can be built and simulated much faster than the implementation model at the expense of a reasonable amount of simulation accuracy.*

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD); C.3 [**Special-Purpose and Application-Based Systems**]: Signal processing systems, Real-time and embedded systems.

General Terms

Design, Languages.

Keywords

Transaction level modeling, system design, DSP, grammar.

1. INTRODUCTION

The difficulties of system design are persistently increasing owing to the well-known reasons, like the integration of more functionality on a system, time-to-market pressure, productivity gap, and performance requirements. To manage these difficulties, the interface based design methodologies have been proposed [1][2]. Here, the basic idea is to separate the communication and the computation aspects of a design so that systems could be modeled with ease at higher levels of abstraction.

Vissers et al. describe a heterogeneous approach to achieve the separation between the communication and the computation of a DSP system [3][4]. Sangiovanni-Vincentelli et al. advocates the separation between function (i.e., computation) and architecture (i.e., communication), and employs the abstract-CFSMs to tackle design challenges [5]. Keutzer et al. have elaborately discussed the separation between these two design issues, and proposed the platform based design methodology [6]. They explain that, function is the behavior of a system that describes the input-output relation. The implementation architecture, on the other hand, states how the function is implemented. Separation of these two aspects splits the design task into smaller and more manageable problems. Their ideas are commercialized in the Cadence VCC tool.

DAC 2004, June 7-11, 2004, San Diego, California, USA

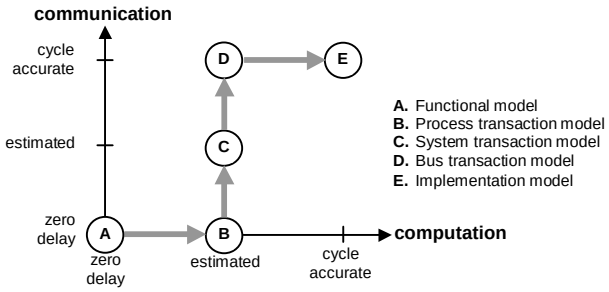

Figure 1: Coordinates of different system models in system modeling graph

The separation between the communication and the computation aspects of a design can be effectively achieved using ***transaction level models*** (TLMs). The concept of TLM first appeared in the domain of system modeling languages, like SystemC [7] and SpecC [8]. In [7], a TLM is defined as a model where communication between modules is modeled in a way that is accurate in terms of what is being communicated, but not in a way that is structurally accurate (i.e., the actual wires and pins are not modeled). They have also insisted that the communication between modules be modeled using function calls.

Gajski et al. have defined a TLM as a model where the details of communication among components are separated from the details of computation within the components [9]. They also present the *system modeling graph* where computation and communication are shown in two axes. Each axis has three degrees of time accuracy: untimed, estimated, and cycle accurate.

Typically, the ***functional model*** (FM) of a DSP system is built at the most abstract level where both communication and computation takes place in zero-time. In the ***implementation model*** (IM), however, both communication and computation are cycle accurate. In the context of DSP systems design, we present three TLMs between the FM and IM. These TLMs are the ***process transaction model*** (PTM), the ***system transaction model*** (STM), and the ***bus transaction model*** (BTM). The coordinates of all these models are shown in the system modeling graph of Figure 1. The thick arrow in gray shows the design flow proposed in this paper. Throughout the design flow, design decisions are added to an abstract model to create a less abstract model.

The contribution of this paper is the systematic formulation of the three TLMs in the context of DSP systems design. In addition, we show how these models can be built with ease using the grammar based language of MASIC. The DSP system design methodology MASIC, short for *Maths to ASIC*, provides an elegant grammar based language to build abstract system models [10]. The technique of building cycle accurate models in MASIC is presented separately in [11].

Our results show that the models at the higher abstraction level can be built with ease and simulated much faster than the less abstract models without significant loss of accuracy. Next, in Section 2 we discuss the commonly practiced DSP systems modeling styles and the MASIC language, which is used to build the TLMs. Our methodology is described in Section 3, followed by an illustrative example in Section 4. Section 5 presents the experimental results. Finally, we conclude this work in Section 6.

2. BACKGROUND

Functional modeling of signal processing applications usually begins using Kahn Process Network (KPN) [12] or different forms of dataflow networks like SDF [13], DDF [14], etc. Processes in a KPN are connected through infinite length point-to-point FIFOs. Graphically, processes are drawn as nodes and FIFOs as arcs. Processes read from the input FIFO when data is available (i.e., blocking read), perform computation in their private memory, manipulate their own state space, and write results in an output FIFO (non-blocking write). An important issue regarding an optimum implementation of these networks is to find a schedule to determine which process executes on which resource at which point in time. There exists an efficient technique of scheduling SDF networks for a sequential or parallel implementation [13].

In our design flow, from the functional model through different TLMs to implementation model, the functionality remains the same. It is only the protocol of data transaction that evolves from abstract FIFO channels to bus protocols, component interfaces, etc. There are different ways of describing a communication protocol. One way is to specify a state machine that implements the protocol using an HDL description. This is a low level approach. A more abstract way is to specify the grammar of the protocol and synthesize a controller from it.

There exist academic [15][16][17] and commercial [18] grammar based tools for protocol description. Though these approaches do not address the problem of system design, they demonstrate two clear advantages, firstly: the ease of protocol description using grammar, and secondly: the smooth path to HW synthesis from a grammar description. Inspired by these advantages, we have adopted the grammar based style in our methodology. Abstract communication modeling has become a part of different system modeling languages like, SystemC [7] and SpecC [8]. However, we argue that grammars provide a more intuitive way of describing protocols than the C++ or C language.

Grammars are primarily used in compiler development for pattern matching [19]. We use grammars to recognize signaling pattern, which represents a signaling protocol, and to produce the desired action when the pattern is seen at the input stream. The actions could be to generate a synchronization signal, store data in a buffer, or call a C function with the data stored in a buffer. There are two major sections of the MASIC description of a model:

- ***Grammar rules***: describe the protocol of data transaction.
- ***Constraints to the rules***: specify architectural resources like FIFOs, synchronization signals, buses, memories, IOs (i.e., interface), etc.

```
(stream) [@clock_name] : [(condition)] pattern1 {action1}
                                     |  pattern2 {action2}
                                     | .. ..
                                     | reset {action-n}
                                     ;
```

Figure 2: General syntax of a grammar rule

The syntax of the MASIC grammar rule is shown in Figure 2. We have added an optional `clock_name` on top of the YACC-like grammar rule. Reading different streams at different speeds symbolizes multiple clocks in a system and allows modeling of multi-rate systems. The rule in the figure says: a `stream` is read at the rate of the given `clock`. Next, it says, if a given `condition` is met and a certain `pattern` is seen at the stream, then the corresponding `action(s)` inside the curly braces would take place. The pattern-action pair of a grammar rule provides a natural way of describing transactions.

3. TLM BASED DESIGN OF DSP SYSTEMS

3.1 Functional Model (FM)

At this level, individual DSP functions are developed in C or MATLAB like environment. These functions are connected by infinite length FIFOs. At this stage, design issues are primarily algorithmic and verification is concerned with making sure that the specified signal processing figures of merits are met. The output of this level is a set of DSP functions in C without side effects.

3.2 Process Transaction Model (PTM)

The coordinate of PTM in the system modeling graph is shown in Figure 1, which reveals the fact that the communication in PTM takes place without any delay and the computation delay of the model is at an estimated level. We reuse the C functions developed at the FM level and add design decisions to build a PTM. The design decisions at this level are: the sizes of the FIFOs, the process to resource mapping, and a schedule of process execution. The sizes of the FIFOs for an SDF network can be found using the *balance equation* for each arc between the processes of a network [14]. Though, finding the FIFO size is not possible for the general case of KPN, there exist scheduling techniques to find a reasonable upper bound using simulations with varying input data set [20].

While a function is mapped to SW to achieve flexibility, mapping to a HW block is done for performance critical parts. Traditionally, the mapping of functions on the architectural resources is viewed as a scheduling problem. However, the order of process execution given by a schedule is not enough to build a PTM, because:

- scheduling technique, for example in [13], only provides a sequence of process execution, where execution is an atomic operation.
- scheduling assumes the ideal situation that processes have their local memories.
- scheduling [13] does not deal with the system interface to the environment and buffering of input data. Hence, the assumption of being able to schedule an input node at *any time* has to be synchronized with the availability of valid data at input.

To build a PTM, we split the atomic process execution into a sequence of read, compute and write operations, and these operations are properly synchronized. Let us consider the example SDF network shown in Figure 3(a). Here process `p1` reads one token from FIFO `f1`, which has a delay of 2 unit; and produces one token in `f2`, which has no delay. Corresponding values for the other processes and FIFOs are shown in the figure. Considering a run time of 1-unit for process `p1` and `p2`, and 3-units for process `p3`, an optimal schedule for the SDF network is shown in Figure 3(b). Here process `p1` and `p2` are running on processor1, and process `p3` is running parallelly on processor2. This schedule runs well if the `f1` has a depth of two.

Now, if processor2 does not have enough internal memory it would write directly to the output FIFO `f1`. Since the size of `f1`

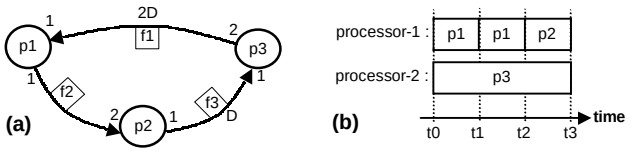

Figure 3: (a) An SDF network, (b) schedule for two parallel processors

is bounded, this operation would overwrite the FIFO and cause processor1 to read new data before it has finished processing the old data. To synchronize them properly, processor2 has to wait until processor1 has finished reading tokens from arc1 (assuming processor1 has enough input buffer), or until processor1 has finished executing process `p1` twice (assuming processor1 has no input buffer). We shall show how such synchronization can be easily expressed using grammar rules.

The invocation of each function is controlled by a grammar rule. The functions communicate over dedicated FIFO channels using `get` and `put` procedures, and hence no physical address is needed. The `get` and `put` procedures provide atomic bulk transfer capability and timing is modeled only for major synchronization events. Several grammar rules read their input streams in parallel and have an inherent end recursion. Thus the model represents the concurrent non-terminating behavior of a system. The concurrent rules are arbitrated by an abstract controller, which provides scheduling and synchronization of events.

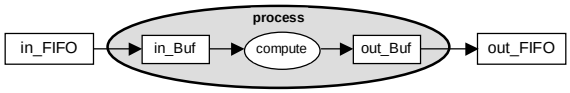

Figure 4: A process of a network

Let us consider the process shown in Figure 4 that reads data from `in_FIFO` and writes data to `out_FIFO`. It reads the `data_Rdy` and `room_Rdy` synchronization signals from the controller to know if there is data available in `in_FIFO` and if there is room available in `out_FIFO`, respectively. Assuming that the process has an input and an output data buffer, a grammar rule with two alternatives is shown in Figure 5(a). The first alternative specifies: if data is ready in `in_FIFO` then it would be copied to `in_Buf`; `read_Rdy` would be asserted so that another process can start writing to this FIFO; and a C function would execute on `in_Buf` and write result in `out_Buf`. To model the computation delay, the process waits for an estimated amount of time before asserting the `result_Rdy` signal. The second alternative says: when there is room in the `out_FIFO`, the `out_Buf` is copied to `out_FIFO;` and the `result_Rdy` signal is deserted.

Figure 5(b) is more interesting. In this case, we assume that the process shown in Figure 4 has neither input nor output data buffer. So, if it receives a `data_Rdy` signal it can not start executing the C function. However, if it receives both `data_Rdy` and `room_Rdy` signal, then it executes the C function directly on the data in `in_FIFO` and saves the result in `out_FIFO`.

The addition of such simple rules keeps the order of process execution as suggested by a schedule and adds the necessary synchronization to make the PTM work correctly with different implementation restrictions. The MASIC compiler reads the PTM written using MASIC language and generates a VHDL description. The VHDL model imports the DSP functions in C and performs a cosimulation using the Foreign Language Interface (FLI) of VHDL. Simulation of this model shows the system performance

```
(data_Rdy, room_Rdy) @clk:
      '1', '0'  { get(in_FIFO, in_Buf);
                  read_Rdy  <= '1';
                  wait until clk = '1';
                  read_Rdy  <= '0';
                  call c_fun1(in_Buf, out_Buf);
                  wait for 3 us;
                  result_Rdy <= '1'; }
    | '0', '1'  { put (out_Buf, out_FIFO);
                  result_Rdy <= '0'; }
    | ...  ...
                                                  (a)
```

```
(data_Rdy, room_Rdy) @clk:
      '1', '0'  { null; }
    | '1', '1'  { call c_fun2(in_FIFO, out_FIFO);
                  wait for 5 us;
                  read_Rdy  <= '1';
                  wait until clk = '1';
                  read_Rdy  <= '0'; }
    | ...  ...
                                                  (b)
```

Figure 5: Example grammar rules

based on the estimated computation delay. If the run-time of a function is data independent, the computation delay of the function on a target architecture can be found using system level estimators. Even for the data dependent case, the computation time is bounded in hard real time applications. If performance requirements are not met then design decisions are changed. For example, the number of resources can be increased to add computational power. Again, as shown in [20], FIFO sizes can be increased to obtain a better schedule with higher performance. It is necessary to note that, though a process to processor mapping is assumed no hard implementation decision has been made. Hence, it does not require an expensive redesign effort to change the mapping decision.

3.3 System Transaction Model (STM)

While PTM captures the process to processor mapping and the computation delay, it does not capture any information regarding the implementation of the FIFOs and the communication architecture. There is a wide design space related to the implementation of the communication architecture. For example, the FIFOs could be implemented using message passing or shared memory; the single address space of the shared memory could have centralized or distributed physical memory; the bus might have different widths, protocols and arbitration priorities, etc. Instead of buses a NoC based architecture, as proposed in [21], can be used to implement the communication architecture.

STM is truly the system model that captures information on both computation and communication aspects of the design, however both of them are at an estimated level. At this level, we decide how the FIFOs would be implemented and what type of architecture would be used to exchange data. Communication architectures add significant amount of delay due to synchronization overhead. The delays would depend on bus width, bus protocol, priorities, etc. Currently, the communication delay is estimated by observing the

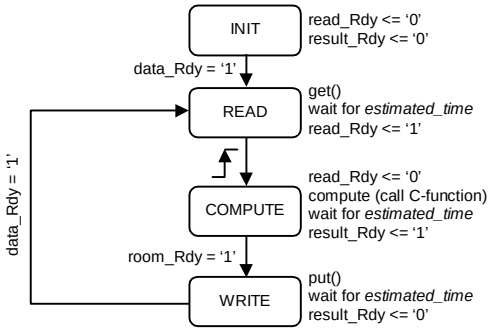

Figure 6: Control flow of a typical process in STM

communication protocol of the target bus architecture in an ad-hoc basis. However, more accurate delay figures can be estimated using the trace based analysis technique presented in [22].

The transactions modeled at this level are same like the PTM. Processes read from and write to FIFOs using `get` and `put` procedures. However, at the end of each `get/put` operation, estimated communication delay is added using the `wait` statement. The control flow synchronization points of a process in STM are shown in the control flow of Figure 6. As shown in the figure, a process reads and writes using atomic bulk transfer. However, the `read_Rdy` signal does not appear before the estimated amount of delay. Hence, the simulation of this model shows both communication and computation delays at an estimated level. If simulation result of the STM is not satisfactory then we need to change the design decisions. Since system transactions have not yet been modeled using the actual wires and pins, changing design decisions is fairly easy at this level.

3.4 Bus Transaction Model (BTM)

So far, we have already decided how our processes and FIFOs would be implemented. At this level we model the communication among the resources at a structurally accurate level. We elaborate the atomic bulk transfers between the synchronization points of the STM and access the memory through shared communication medium (e.g., bus) using physical address. The BTM does not alter the execution semantics of the STM because the points of synchronization are maintained. Simulation of the BTM reveals the clock true communication delay of the implementation architecture. The design steps to create a BTM from an STM involves the following tasks:

- Elaborating the abstract communication channels into the detailed signaling mechanism required to express the bus protocol and the arbitration logic. If several functions are mapped on a single core, the channel between the functions needs to be implemented using the communication primitive offered by the operating system.
- Describing the interface of the hardware blocks onto which the functions are mapped to. To ease reuse of predesigned components we create the *bus functional models* (BFMs) of embedded cores and interface description of IP blocks. MASIC descriptions are used to build these models and they are saved in a library from where they are instantiated.
- Adapting the component interfaces to the bus protocol. We describe the glue logic between the pre-designed blocks and the bus architecture. The adapters can be saved in the library and reused. Currently the glue logic is written manually. However, this task can be automated, for example, using the approach presented in [23].

3.5 Implementation Model (IM)

As shown in the system modeling graph of Figure 1, both the communication and computation are clock true at this level. For a HW implementation, the C functions are replaced with the RT level description of the IP block. For a SW implementation, on the other hand, the C functions are compiled to the target architecture. Since a *bus functional model* represents the external interface of a core and generates the cycle accurate transactions supported by the core (e.g., read, write, burst read, burst write, etc.), the glue logic developed at the BTM level works at the IM level as well. A comparison of different system models is given in Table 1.

4. AN ILLUSTRATIVE EXAMPLE

Here we are using the Linear Predictive Coding (LPC) example. It samples input values at a rate of 8 kHz and buffers 160 input samples. Then it performs the windowing operation on the samples and generates another 160 data. Next, the autocorrelation function takes this result and computes 11 autocorrelation lags, which the LPC block reads to compute 10 coefficients. Finally, the reflection coefficients are computed from the LPC values. At the FM level we build four C functions that compute the windowing (`win`), autocorrelation (`cor`), LPC (`lpc`) and reflection (`rfl`) coefficients.

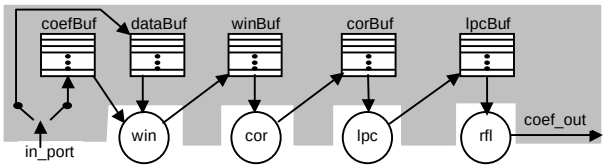

Figure 7: PTM of the LPC codec

We decide to implement our four DSP functions on four MIPS32-4K processor cores. For the application at hand, we use SDF network scheduling technique, which also gives us the FIFO depth. The interface specification of the system requires sharing the `in_port` for downloading the windowing coefficients at startup, and then starting regular processing of data. Hence the startup sequence would look like: reset, downloading coefficients, and then regular data processing. We reuse the C functions built at the FM level to create a PTM, as shown in Figure 7. The C functions, interface to the environment, and the FIFO channels are declared using grammar constraints. The grammar rules are used to describe the atomic data transaction among the functions, and to model the necessary synchronization primitives to handle the input data buffering and startup sequence. MASIC description of abstract PTM-like model has been shown elaborately in [10].

Table 1: Comparison of different system models

	Communication	Computation	Transaction/Operation	Model characteristic
Functional Model (FM)	zero-delay	zero-delay	- process execution (atomic execution consisting of read, computation, and write operations)	process to resource mapping is not done, communication through ***infinite*** length FIFO
Process Transaction Model (PTM)	zero-delay	estimated	- bulk read from FIFO - computation (C function call, with estimated delay) - bulk write to FIFO	process to HW/SW mapping is assumed, communication through ***finite*** length FIFO, read/write to FIFO using get/put procedures
System Transaction Model (STM)	estimated	estimated	- bulk read from FIFO (with estimated delay) - computation (C function call, with estimated delay) - bulk write to FIFO (with estimated delay)	communication through get/put procedures from/to FIFO with estimated delay for memory access using shared medium
Bus Transaction Model (BTM)	cycle accurate	estimated	- memory read (Req, Ack, Address, Data, Split) - computation (C function call, with estimated delay) - memory write (Req, Ack, Address, Data, Split)	communication through cycle accurate component interface and shared medium, read/write to memory using physical address
Implementation Model (IM)	cycle accurate	cycle accurate	- memory read (Req, Ack, Address, Data, Split) - cycle accurate computation (RTL, ISS) - memory write (Req, Ack, Address, Data, Split)	cycle accurate computation– RT level for HW implementation, or instruction level for SW implementation

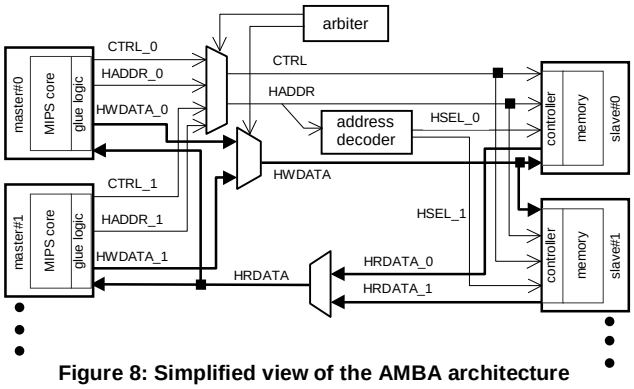

Figure 8: Simplified view of the AMBA architecture

Next, we decide to realize communication architecture using the AMBA on-chip bus [24]. To build an STM from the PTM, we add estimated communication delay after each `get` and `put` operation of the PTM. This it still a fairly abstract model as the data transaction are not modeled using pins and wires.

Next, at the BTM level, the communication is modeled at the cycle accurate level. The *bus functional model* (BFM) of the MIPS32-4K core [25] is used as the bus master module. The communication channels are elaborated according to the AMBA AHB specification. A simplified view of the AMBA architecture is shown in Figure 8. The data and control lines are shown in solid and thin arrows, respectively. The request and grant lines between masters and arbiter are not drawn.

The AMBA AHB uses separate read and write buses operating through a centrally multiplexed architecture. To gain performance, it works in a pipelined fashion where the address-phase of a transfer proceeds simultaneously with the data-phase of the previous transfer. Now we shall show how these complex transactions can be described using our grammar based technique.

The control, address and data buses are represented as internal signals using grammar constraints. We connect the read and write data buses to the read and write data ports of the bus functional model of the core. The address and control information of the master, which wins the arbitration, are propagated to the slaves through the address bus `HADDR`. The address decoder, shown in Figure 8, selects a slave by combinational decode of the higher bits of the address bus `HADDR`. Figure 9 shows the grammar rule for the address decoder. The clock information is absent in the grammar rule, which symbolizes a combinational behavior, and the decoding logic is described using the pattern-action pairs.

Depending on the slave select signal, the appropriate slave unit is selected and the control signal `CTRL` tells the type of transaction that needs to be performed. The aggregate signal `CTRL` is composed of several AHB signals like `HWRITE`, `HTRANS`, `HBURST`, etc. Figure 10 shows the grammar description involved in the slave module that includes a memory and a controller. The memory behavior is described as: if the condition `CS` is true, then a `'1'` at `RWS` causes a write and a `'0'` at `RWS` causes a read. The next line fetches the address from the address-phase of the transfer. It says

```
-- grammar rule for the address decoder
(HADDR_HIGH_BIT) : "00"  { HSELv <= "0001"; }
                 | "01"  { HSELv <= "0010"; }
                 | "10"  { HSELv <= "0100"; }
                 | "11"  { HSELv <= "1000"; }
                 ;
```

Figure 9: MASIC description of address decoder

```
-- the memory block of slave_0
(RWS) : (CS)  '1' { MEM(ADDR) <= HWDATA; }
            | '0' { HRDATA_0  <= MEM(ADDR); }
            ;


-- fetching the HADDR at the rising edge of the HCLK
-- of the address cycle
(HSEL) @HCLK : '1' { ADDR <= HADDR };


-- Protocol of AMBA Slave Controller
(HSEL, HWRITE, HTRANS, HBURST) @HCLK
      : '1', '0', "10", "000"  { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
      | '1', '1', "10", "000"  { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
      | '1', '0', "10", "001"  { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
         branch1

      | '1', '1', "10", "001"  { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
         branch2

      | reset                  { CS     <= '0';
                                 RWS    <= '0';
                                 HREADY <= '0'; }
      ;

branch1  : '-', '-', "11", '-' { CS     <= '1';
                                 RWS    <= '0';
                                 HREADY <= '1'; }
            branch1
         | '-', '-', "00", '-'
         ;

branch2  : '-', '-', "11", '-' { CS     <= '1';
                                 RWS    <= '1';
                                 HREADY <= '1'; }
           branch2
         | '-', '-', "00", '-'
         ;
```

Figure 10: MASIC description of the slave

to read the address (`HADDR`) at the arrival of the `HCLK` if the `HSEL` signal is high, which is the address-phase. By default, the clock is implemented as rising edge triggered and the reset as an asynchronous reset.

Next, the slave controller reads the aggregate control word, separated by commas. If the first bit pattern is seen, then it asserts the chip select signal and de-asserts the `RWS` so that the RAM outputs (i.e. a read operation) a single word. The second pattern causes a write operation. The third pattern initiates a burst read of unspecified length. The first transfer of a burst uses `HTRANS="10"`, followed by `"11"` for the remaining transfer and terminates with a `"00"`. This whole information is described as follows: if a pattern of `('1', '0', "10", "001")` is seen, the first data of the burst is supplied and then it looks for a pattern labeled as `branch1`. The `branch1`, as described below, is repeated while there is a `"11"` at the `HTRANS` input; the other inputs are don't cares, represented by the `'-'`s. Finally, `branch1` terminates when a `"00"` is seen at `HTRANS`.

5. RESULTS

We have performed several experiments using two core examples: the LPC codec described in the previous section and a ΣΔ demodulator. The ΣΔ demodulator has two FIR filters of length 31 and 69, one integrator and one differentiator. The DSP functions are developed in C during the functional modeling phase, and reused to build a PTM. We have decided to implement the system using the AMBA AHB architecture. AMBA allows single cycle bus master hand over. The master hand over cycle, the data transfer cycles, and the delay in the glue logic is considered to make a static estimation of the communication delay. The `wait` statements are used to add the delay time in the STM.

Next, the *bus functional models* (BFM) of MIPS32-4K cores, cycle accurate description of the AMBA architecture and the glue logic are used to build a BTM. The computation is still performed using C functions with approximated delay figures.

Table 2 shows the code size of the MASIC and VHDL description of the three TLMs for the two examples. The increase in productivity in term of the design-hour could be guessed from the bulk of VHDL code and the number of states needed in the VHDL model. In the MASIC approach, the system transactions are expressed in abstract grammar, from where the VHDL is generated by our compiler.

Table 2: Code size and number of states

	LPC		ΣΔ	
	MASIC	VHDL	MASIC	VHDL
PTM (word count)	305	2596	276	2409
STM (word count)	337	2628	300	2447
BTM (word count)	2998	15865	2437	11952
Number of states in BTM	-	176	-	142

Table 3: VHDL Simulation time

	LPC	ΣΔ
PTM	13 min. 47.7 sec	8 min. 36.1 sec
STM	13 min. 47.7 sec	8 min. 36.2 sec
BTM	52 min. 41.2 sec	37 min. 23.5 sec

Table 3 shows the VHDL simulation time of these TLMs for 1 sec of input data. The PTM/STM simulates much faster than the BTM, because they do not perform the intricate signaling protocol of data transaction, instead they just uses `get` and `put` procedures to FIFO. Such speedups are highly beneficial in an iterative design flow. The simulation time of PTM and STM remains more or less the same, as these models just differ by a couple of `wait` statements. We compare the communication delay found from the STM with the clock true delay found from the BTM. The communication delay figures found from the STM vary only by ±2.98% compared to the cycle accurate delay figures found from the BTM simulation.

6. CONCLUSION

We have presented three TLMs in the context of system design for DSP applications. The formulation of these TLMs eases the design task of realizing an implementation model from an abstract functional model, the exploration of the design space at a higher level of abstraction, and the reuse of predesigned cores and HW blocks using their interface description. In addition, we have shown a unique language support to build the TLMs with ease. Though the work is described in the context of the MASIC methodology, any DSP system design methodology would benefit from the systematic formulation of these TLMs.

While the BTM provides a synthesizable description of the communication architecture, the PTM and STM are meant for efficient system simulation. To gain further simulation speedup we are considering the compilation of the grammar description of the PTM and STM to SystemC, instead of VHDL. Currently the glue logic for embedded cores is written manually. This task can be automated using the ideas presented in [23]. So far, only SDF examples have been considered. However, the methodology is not restricted to SDFs. To model a general process network a dynamic scheduler needs to be built. Currently the communication delay is not estimated using any rigorous technique. The simulation accuracy of the STM can be improved by using the delay figures found from the trace based analysis technique shown in [22].

7. REFERENCES

[1] J.A. Rowson and A. Sangiovanni-Vincentelli, "Interface Based Design," in *Proc. DAC*, pp. 178-183, Jun. 1997.

[2] C. K. Lennard, P. Schaumont, G. Jong, A. Haverinen, and P. Hardee, "Standards for system-level design: Practical reality or solution in search of a question," in *Proc. DATE Conf.*, pp.576-583, Mar. 2000.

[3] P. van der Wolf, P. Lieverse, M. Goel, D.L. Hei and K. Vissers, "An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology," in *Proc. CODES*, pp. 33-37, May 1999.

[4] E.A. de Kock, et al. "YAPI: Application Modeling for Signal Processing Application," in *Proc. DAC*, pp. 402-405, Jun. 2000.

[5] M. Sgroi, L. Lavagno and A. Sangiovanni-Vincentelli, "Formal Models for Embedded Systems Design," *IEEE Design & Test of Comp*., vol. 17, no. 2, pp. 14-27, Jun. 2000.

[6] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey and A. Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform based Design," *IEEE Trans. CAD*, vol. 19, pp. 1523-1543, Dec. 2000.

[7] T. Grötker et al., *System Design with SystemC*, Kluwer Academic Publishers, Norwell, MA, 2002.

[8] R. Dömer, D.D. Gajski and A. Gerstlauer, "SpecC Methodology for High-Level Modeling," in *Proc. 9th IEEE/DATC Electronic Design Processes Workshop*, Monterey, CA, Apr. 2002.

[9] L. Cai and D. Gajski, "Transaction Level Modeling: An Overview", in *Proc. CODES+ISSS*, pp. 19-24, Newport Beach, CA, Oct. 2003.

[10] A. Hemani, Abhijit K. Deb, J. Öberg, A. Postula, D. Lindqvist and B. Fjellborg, "System Level Virtual Prototyping of DSP SOCs Using Grammar Based Approach," *Kluwer Design Automation for Embedded Systems*, vol. 5, no. 3, pp. 295-311, Aug. 2000.

[11] Abhijit K. Deb, A. Jantsch, and J. Öberg, "System Design for DSP Applications Using the MASIC Methodology," in *Proc. DATE Conf.*, vol. 1, pp. 630-635, Feb. 2004.

[12] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress '74*, pp. 471-474, Aug. 1974.

[13] E.A. Lee and D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. Comp*., vol.C-36, no. 1, pp. 24-35, Jan. 1987.

[14] J.T. Buck and E.A. Lee, "Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model," in *Proc. Int. Conf. Acoustics Speech & Signal Processing*, pp. 429-432, Apr. 1993.

[15] A. Seawright, F. Brewer, "Clairvoyant: A Synthesis System for Production-based Specification," *IEEE Trans. VLSI*, vol. 2 no. 2, pp. 172-185, June 1994.

[16] J. Öberg, A. Kumar, and A. Hemani, "Grammar-Based hardware synthesis from port size independent specifications," *IEEE Trans. VLSI*, vol. 8, no. 2, pp. 184-194, April 2000.

[17] R. Siegmund and D. Müller, "Automatic Synthesis of Communication Controller Hardware from Protocol Specifications," *IEEE Design & Test of Comp*., vol. 19 no. 4, pp. 84-95, Jul-Aug. 2002.

[18] A. Seawright, U. Holtmann, W. Meyer, B. Pangrle, R. Verbrugghe, J. Buck, "A system for compiling and debugging structured data processing controllers," in *Proc. Euro DAC.*, pp. 86-91, Sept. 1996.

[19] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, MA, 1986.

[20] T. Basten and J. Hoogerbrugge, "Efficient Execution of Process Networks," in *Proc. Communicating Process Architectures*, pp. 1-14, IOS Press, Amsterdam, 2001.

[21] S. Kumar et al., "A Network on Chip Architecture and Design Methodology," in *Proc. IEEE Comp. Society Annual Symposium on VLSI*, pp. 105-112, Apr. 2002.

[22] K. Lahiri, A. Raghunathan and S. Dey, "System-Level Performance Analysis for Designing On-Chip Communication Architectures," *IEEE Trans. CAD*, vol. 20, no. 6, pp. 768-783, Jun. 2001.

[23] R. Passerone, J.A. Rowson and A. Sangiovanni-Vincentelli, "Automatic Synthesis of Interfaces between Incompatible Protocols," in *Proc. DAC*, pp. 8-13, Jun. 1998.

[24] AMBA on-chip bus specification [Online], http://www.arm.com

[25] MIPS32 4K Processor Core Family Integrator's Manual, [Online], http://www.mips.com