A System-Level Specification & Design Methodology

Daniel Gajski Sanjiv Narayan Frank Vahid

Technical Report #92-102
October 17, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425
(714) 856-4155

Abstract

Given an abstract specification of a system, we present a methodology for specification capture and refinements that will result in synthesizable descriptions. It must be emphasized here that this report is about methodology - the set of descriptions and the manual refinements to derive one description from the other. Some of these refinements can be automated and collected as a set of system-level synthesis tools which are not a part of this report.

Contents

1 Introduction **1**

2 System Level Design Methodology **2**

2.1 Specification Capture 3

2.2 Specification Refinement 3

2.3 Behavioral Implementation 6

3 Specification Capture **6**

3.1 Capturing the Designer's Conceptual View 6

3.2 The SpecCharts Language 8

4 Specification Refinement Example **10**

4.1 Example : SpecChart of the RP Processor 10

4.2 Storage Grouping 11

4.3 Storage Implementation 12

4.4 Behavioral Grouping 14

4.5 Channel Grouping 16

4.6 Interface Implementation 17

5 Behavioral Implementation **20**

5.1 Hardware Implementation 20

5.2 Software Implementation 20

6 Conclusion **21**

7 Acknowledgements **21**

8 References **21**

List of Figures

1 System Level Synthesis Domain 1

2 System Level Synthesis Methodology 4

3 (a) Specifying actions and sequencing (b) Grouping and refining behaviors 7

4 Representing leaf-level behaviors using programming constructs 8

5 Partial SpecChart of the answering machine 9

6 Partial SpecChart of the RP 10

7 Storage Grouping in RP 12

8 Storage Implementation in RP 14

9 Behavioral Grouping in RP 15

10 Channel Grouping in RP 16

11 (a) Behaviors communicating over a channel. (b) Interfacing the RP with external memory (c) Generating an arbiter for the Program and Data buses. 17

12 Interface Process Generation in RP 19

1 Introduction

System design efforts can be focused on several different levels of abstraction. Focus on the gate level is quite common, as there are a variety of tools to support capture and simulation at this level. The register transfer level (RTL) is becoming more common as RTL and logic simulation and synthesis tools gain acceptance. However, design effort is rarely focused on the behavioral level, where the functionality of a system is described using a software-like language.

The lack of design effort at the behavioral level can be attributed to two reasons: first, there is no proven design methodology at this level, and second, there are few tools to support the design tasks at this level. We present in this report, a system design methodology that starts with an executable specification.

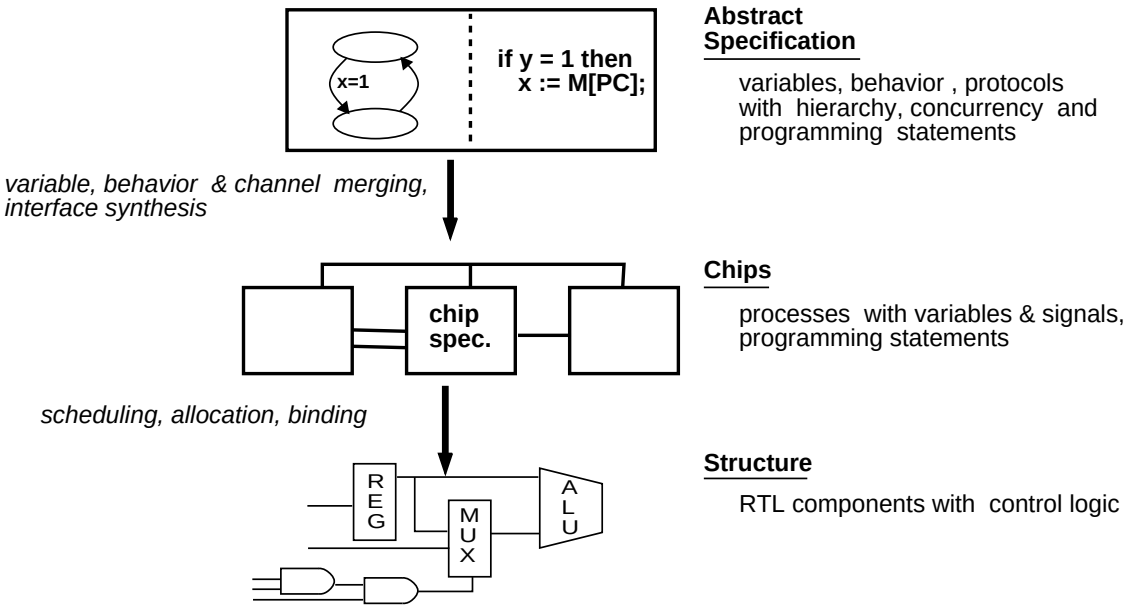

Figure 1: System Level Synthesis Domain

Figure 1 shows the basic steps in our methodology. We start with an executable specification of the system which describes the system as a set of hierarchically structured concurrent behaviors. Each behavior is expressed as operations or functions performed on scalar and array variables. Communication is achieved using shared memory variables. To obtain manufacturable hardware, we perform several refinements on this abstract specification. We group all variables into memories, register-files and registers, group all behaviors into processes implemented as processors, and group all communication into channels implemented as buses with well defined protocols. Finally, we partition the specification into chips that satisfy given constraints such as silicon area, packaging cost, performance,

power and testability among others.

Each chip consist of one or more processes with internal variables and external signals. Each process will be implemented as a Finite State Machine with a Datapath (FSMD). The next step consists of refining each process into a RTL block diagram consisting of interconnected RTL components such ALU's, multipliers, selectors, counters, registers and memories and associated control logic.

Starting design efforts at the system-level of abstraction has several advantages:

1. The specification language at the system level can very easily and concisely reflect the designer's conceptualization of the system, requiring little designer effort. Future redesign of a system would be simplified since the description also serves as the system's specification, and thus efforts spent in reverse engineering from a structural design are greatly reduced, if not entirely eliminated.

2. The functionality of the design as captured in the specification can be verified at an early stage by simulation or any other technique.

3. Architectural decisions (such as bus insertion, memory selection, parallel/serial tradeoffs and software/hardware tradeoffs) can be made early in the design process. Since chip-partitioning is done at the system-level, the functionality of each chip is known. Functional test patterns for individual chips can be generated, thus reducing the testing overhead.

4. Gradual refinement is possible since the the same representation is used for the design throughout the design process, enabling the designer to evaluate the effects of each synthesis step easily.

In Section 2 we present a brief overview of the proposed system-level design methodology. Specification capture, specification refinement and behavioral implementation are discussed in Sections 3, 4 and 5 respectively. We present our conclusions in Section 6.

2 System Level Design Methodology

The system-level design methodology (summarized in Figure 2) can be divided into three distinct phases: Specification Capture, Specification Refinement and Design Implementation.

2.1 Specification Capture

We need to be able to capture the description of a design in a form which will serve as an input to our design methodology. Any specification language is essentially a tradeoff between the degree of expressibility and the ease of implementation. For example, specifying a design as a structural netlist requires a great amount of effort by the designer, but it is easy to realize an implementation from such a structural description. On the other hand, describing a design in English gives the designer virtually unlimited power of expression, but it is extremely difficult, if not possible, to synthesize from English specifications. A system specification language should be able to capture the designer's conceptual view of the system without sacrificing a realizable path to design implementation.

At the system-level, a specification language should be capable of representing features and characteristics which are and integral part of real systems. The specification must be able to represent hierarchy and concurrency. It should provide the capability for specifying actions (using constructs similar to that found in programming languages) and sequencing between these actions similar to those in state diagrams. Finally the description language must be executable, thus permitting the designer to verify the functionality of the design by simulation.

In addition to the specification, the designer must specify the design constraints. At the system-level, these constraints could be the number and area of chips that the design has to be implemented in, the performance of certain processes and the number of pins at each chip boundary. The library of components including available off-the shelf components that will be used to implement the design must also be specified by the designer.

2.2 Specification Refinement

Specification Refinement refers to the set of system synthesis tasks and the subsequent incorporation of their results into the initial specification to reflect design decisions made. Specification refinement serves the following purposes:

1. It attempts to bridge the gap between the designer's conceptual view of the design (as captured by the system specification) and the component set that will be used for the implementation. This is achieved by performing a set of synthesis tasks such that each portion of the specification

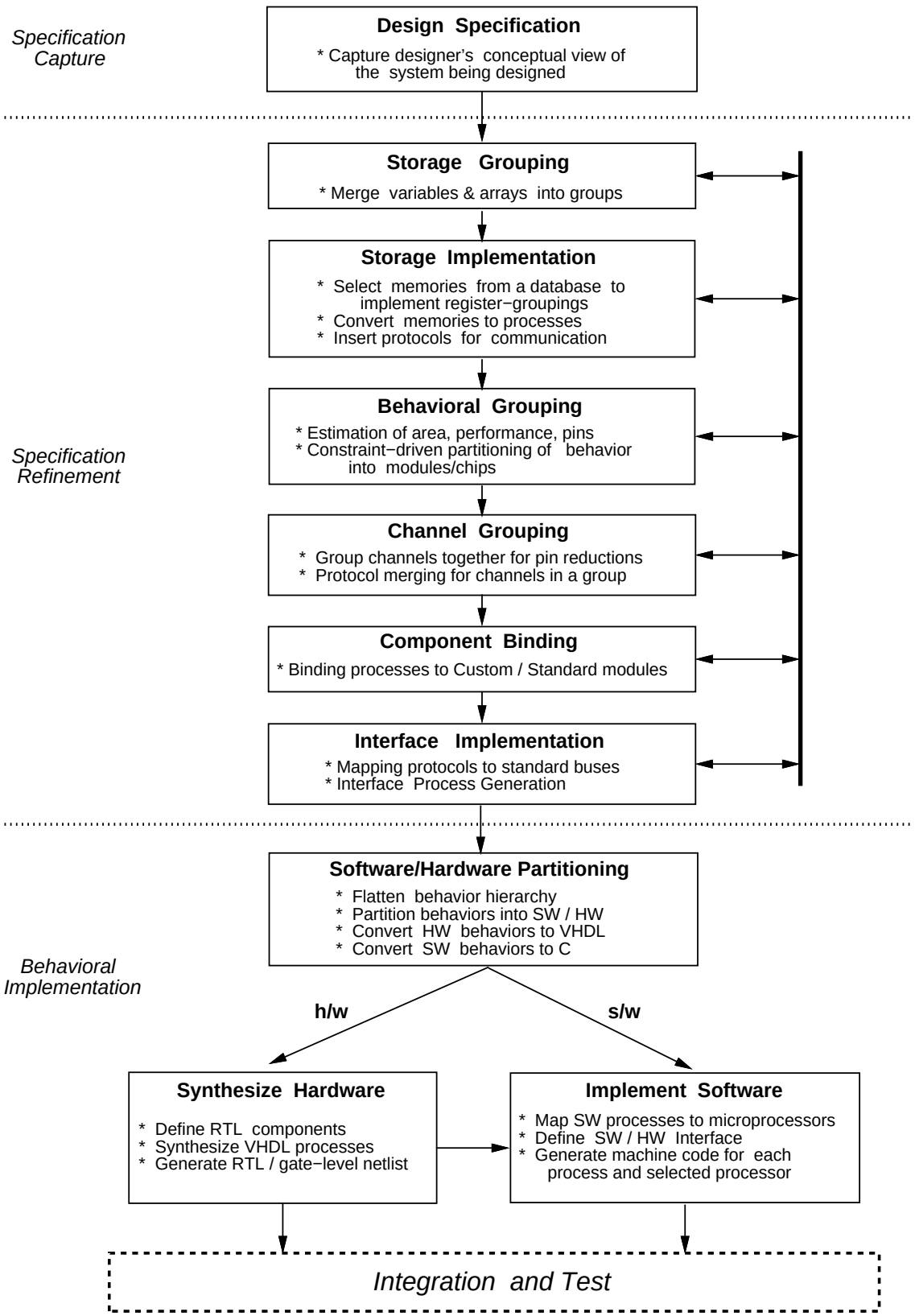

Figure 2: System Level Synthesis Methodology

can be easily mapped to an implementable component. The synthesis tasks are driven by the constraints specified for the design.

2. The specification refinement paradigm enhances *designer interaction*, since the results of any synthesis step are reflected back to the designer in the same language as the initial specification. Having a single language reflect the design throughout synthesis, does away with the need for other intermediate representations (e.g. a control/dataflow graph), which the designer must then correlate with the initial specification.

3. Since the specification language is capable of representing the added detail, the designer could have directly included the details in the initial specification, thus supporting partial design by the designer.

Specification refinement at the system-level consists of the following synthesis steps:

1. **Storage Grouping** is defined as the merging of storage elements in the specification such as variables and arrays into *groups*. Each group of such elements will be implemented by individual registers, register files or memories. Storage grouping minimizes interconnect cost at the expense of potential parallelism.

2. **Storage Implementation** selects register files or memories from the design library to implement the storage groups obtained in the previous step. This provides subsequent steps with information regarding memory access times, number of concurrent accesses that can be made and the communication protocol required to access the memories.

3. **Behavioral Grouping** partitions the set of behaviors and storage elements in the specification into specifications of chips/modules. The partitioner must satisfy the given constraints and thus, estimates of area, performance and interconnect complexities of each of the groups must be determined and compared to the constraints for the group.

4. **Channel Grouping** combines different communication channels in order to minimize package cost (number of pins) at the expense of possibly lower performance.

5. **Component Binding** which may bind behavioral groups to standard off-the-shelf components or indicate that they are to be custom designed. In such cases, the behaviors that are to be custom-designed may have to be modified to make them compatible with the interface characteristics of the behaviors which have been mapped to standard components.

6. **Interface Synthesis** resolves interface consistency issues as a result of the application of some of the above synthesis tasks. Thus, channel-grouping will require that a protocol be selected for the channels grouped together. Component binding may bind two communicating behaviors to standard components. In case their communication protocols are different, we need to generate the necessary interface logic which will be inserted between the two protocols to make them compatible.

2.3 Behavioral Implementation

Behavioral implementation provides the path to the final design. The output of system-level synthesis is a set of *interconnected chip descriptions*, which satisfy the constraints specified by the designer. In case a hardware implementation is desired, RTL blocks are defined and described for each behavior. Finally an RTL/gate netlist is then generated from which a design can be synthesized using commercial tools. However, if a software implementation is desired, the behaviors are first converted to a standard programming language (e.g. C) which permit easy compilation into machine instructions. In this case, interfaces between the software and hardware components in the design have to be defined. Finally, code is generated for each concurrent behavior (process) in the design.

Having briefly introduced the system-level design methodology, we will examine each of the steps outlined above in greater detail.

3 Specification Capture

In this section, we will examine system-level specification issues. First, we briefly introduce some of the features necessary for capturing system-level designs by examining the specification of an answering machine telephone. We will then briefly introduce the SpecCharts specification language.

3.1 Capturing the Designer's Conceptual View

Consider a simple answering machine telephone. Conceptually, the designer should have the capability to specify distinct actions which constitute the behavior of the answering machine. For example, we

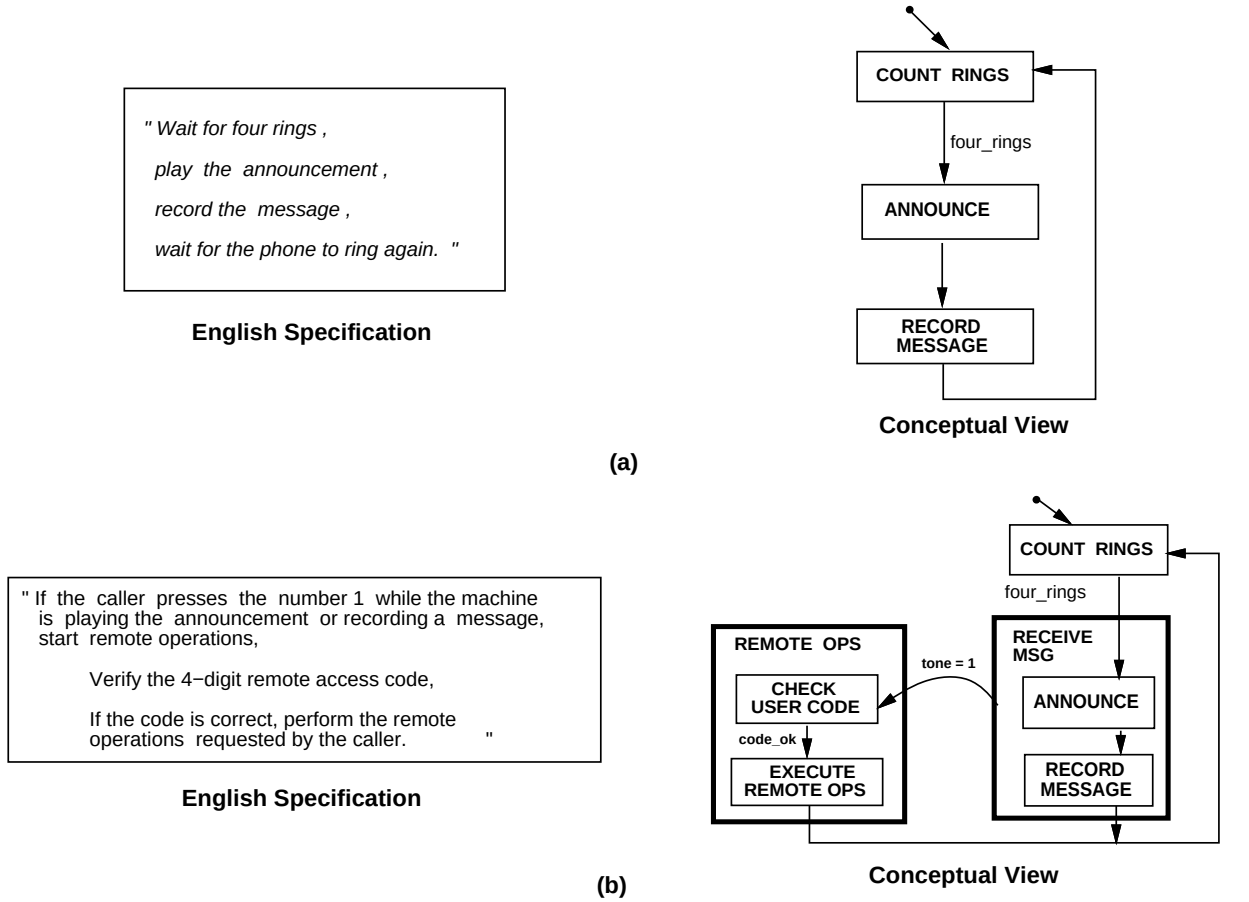

Figure 3: (a) Specifying actions and sequencing (b) Grouping and refining behaviors

want to be able to specify that after four rings, the announcement must be played, the message must be recorded and the machine should wait for the phone to ring again. The English language specification and the conceptual view represented by state diagrams is shown in Figure 3(a). There are three distinct actions or *behaviors* $COUNT_RINGS, ANNOUNCE$ and $RECORD_MESSAGE$, with some sequencing specified between these actions.

It may be the case that the designer may need to group certain behaviors as they share some common property (e.g. a common transition to another state). For example, if we are playing the announcement or recording a message, and the caller presses $button_1$, then we should suspend the behavior we are executing and start remote operations. Thus, in Figure 3(b), behaviors $ANNOUNCE$ and $RECORD_MESSAGE$ are grouped together into a super behavior $RECEIVE_MESSAGE$, with a transition labeled `(tone = 1)` causing them to terminate and start performing remote operations. **Secondly**, we want have the capability to decompose behaviors into sub-behaviors. For example, $REMOTE_OPS$ is decomposed into two distinct behaviors - verifying the user access code,

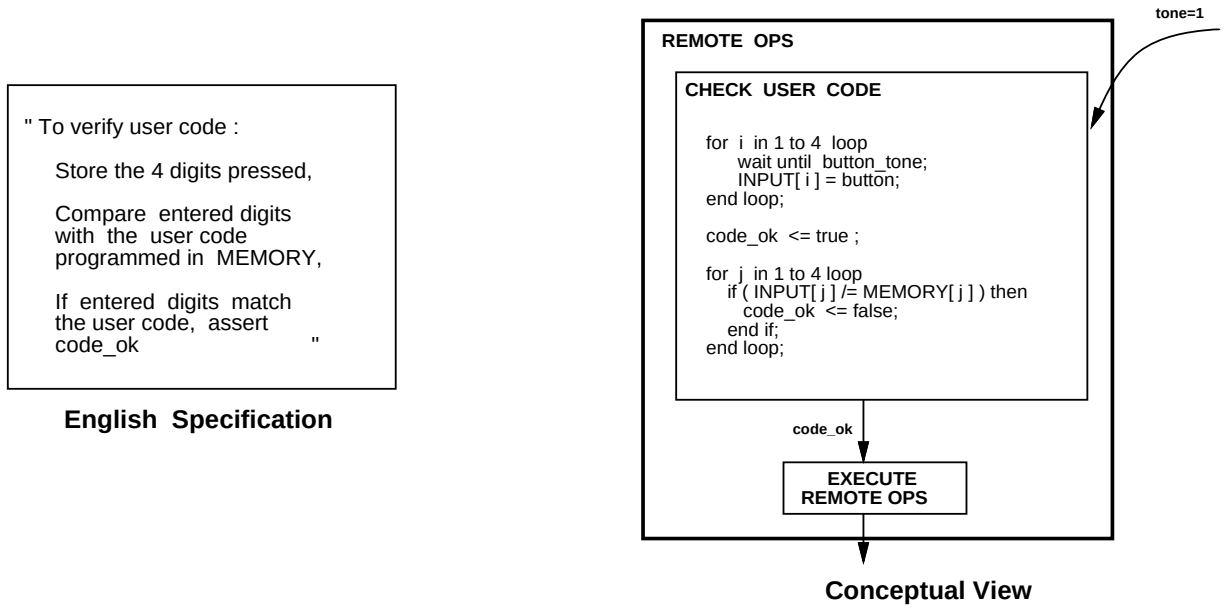

Figure 4: Representing leaf-level behaviors using programming constructs

and then performing the desired remote operation like playing back messages or resetting the machine.

As we decompose a design's behavior into several sub-behaviors, we might reach a stage where we may want to specify the actions using programming constructs like FOR/WHILE loops and CASE statements. For example, in Figure 4, the details of $CHECK_USER_CODE$ have been specified directly using programming constructs.

A partial SpecChart of the answering machine is given in Figure 5. We have shown through the example of the answering machine, that for every system that has to be designed, there exists a conceptual view of the system. A system level specification language should be capable of capturing this conceptual view of the design. This was the main motivation behind the development of the SpecCharts language.

3.2 The SpecCharts Language

As a result of modeling several systems, the **SpecCharts** language was developed for system specification capture [1]. SpecCharts combines the three aspects of system specification (control, behavior, and structure) into a single, unified concept. Essentially a concept of Finite State Machine (FSM) and VHDL, the language permits a system to be described as a set of *behaviors*.

The language supports *behavioral decomposition* in that each behavior can be decomposed into concurrent behaviors (often thought of as processes) or sequential behaviors (often thought of as states) sequenced by conditional arcs. A behavior is also describable using VHDL sequential statements.

Graphical conventions used are similar to those presented in [2]. Briefly, behaviors are represented as boxes, sequential behaviors are sequenced by transition arcs and concurrent behaviors are separated by dotted lines. *Hierarchical Activation/Deactivation* is used to terminate any sub-behavior in response to an external event. This is very useful in modeling events like RESETS which occur commonly in designs. SpecCharts have the *transition immediately* arc which causes the current behavior to suspend execution and effects a transition to the appropriate next state. SpecCharts also allows *behavioral completion*, i.e. the capability of a behavior to indicate completion and that of other states to detect such a completion. The *transition on completion* arc in SpecCharts cause a transition only when the source behavior has completed execution of its actions and the associated condition is true. Behavioral completion is very useful for modeling sequential behaviors like the FETCH, DECODE, and EXECUTE states of a microprocessor.

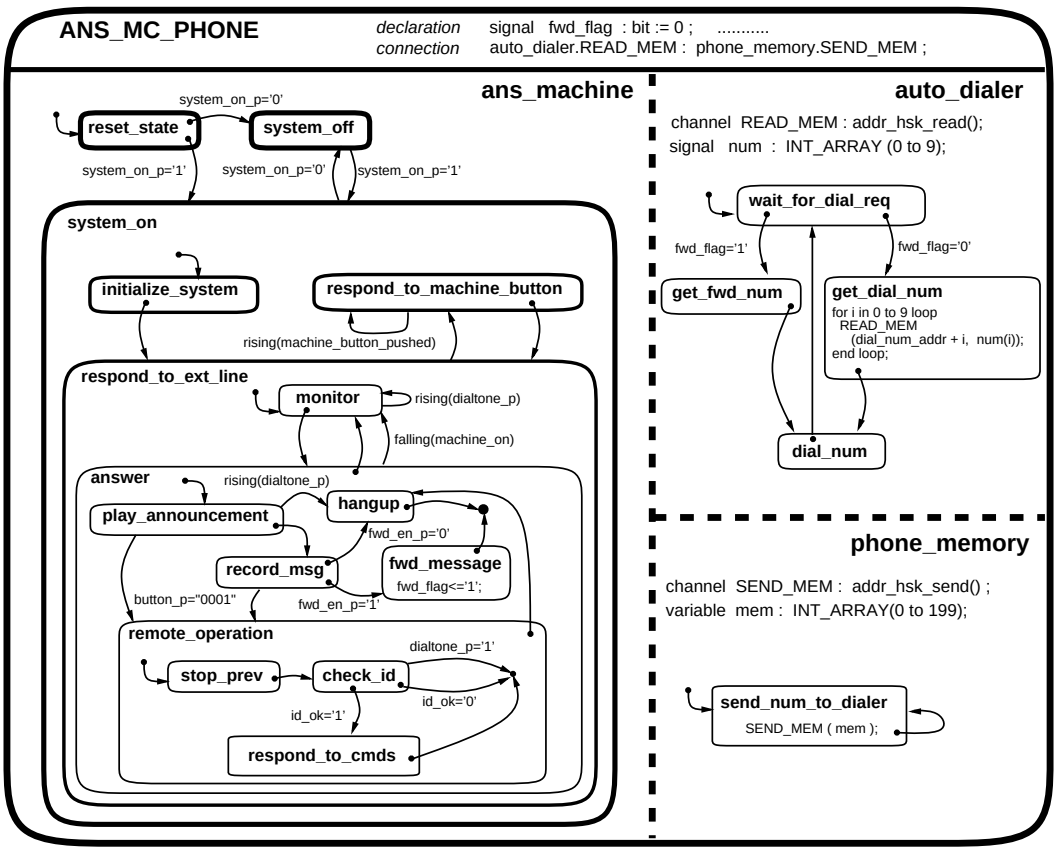

Figure 5: Partial SpecChart of the answering machine

Any language which supports Behavioral Decomposition, Behavioral Completion, and Hierarchical Activation/Deactivation is said to support a *Behavioral Hierarchy*. Behavioral Hierarchy occurs frequently in designs. For example, the answering machine has 10 levels of behavioral hierarchy of which only six are shown in Figure 5. The Intel 8237 DMA Controller had 4 levels of behavioral hierarchy. The Pipelined Processor in [3] also had 4 levels of hierarchy. Detailed descriptions of the above systems can be found in [3].

4 Specification Refinement Example

We have chosen the example of a RISC processor to demonstrate each system-level synthesis task. At each synthesis step, the corresponding details (or *refinements*) that are added to the system specification are highlighted.

4.1 Example : SpecChart of the RP Processor

The RP (RISC Processor) consists of a RISC controller and a signal processor. The RP has a 4 stage execution pipeline - instruction fetch, register fetch, ALU stage and the memory access stage. Each stage takes exactly one clock cycle to execute. The RP also has a register file (RF). The RP has an on-chip program and data memories as well as a large external memory (4M). In addition, there is a timer-counter circuit, a serial interface, host interface and an exception handler [4]. The SpecChart specification of the RP is shown in Figure 6.

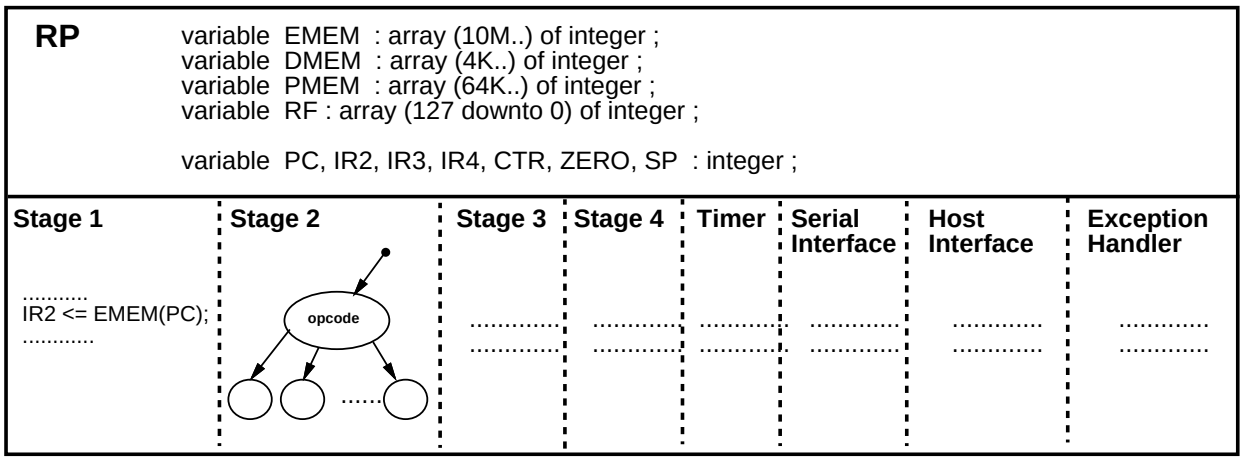

Figure 6: Partial SpecChart of the RP

The SpecChart specification models the 4-stage instruction pipeline as 4 concurrent behaviors $Stage1$, $Stage2$, $Stage3$ and $Stage2$. There were three instruction registers $IR2$, $IR3$ and $IR4$ associated with $Stage2$, $Stage3$ and $Stage4$ respectively, which contained the instruction being executed by that stage. The instruction being executed is passed from one instruction register to the next as it is executed in the instruction pipeline.

Other units of the RP such as the Timer-Counter Circuits and the Serial Interface are also modeled as concurrent behaviors. The register file and memories were modeled as arrays of integers - RF (register file), $PMEM$, $DMEM$ and $EMEM$ (program, data and external memory). The memories were all global in that they could be accessed by all the concurrent behaviors in the SpecChart. In addition to the above addressable memories, the PC (program counter), $IR1$, $IR2$, $IR3$, $ZERO$ (register containing all zeros), SP (stack pointer) and CTR (timer-counter register) were modeled as integers. Each of the concurrent behaviors were described using VHDL sequential statements.

4.2 Storage Grouping

The input description has arrays and variables which will be implemented using either individual registers, register files or memories. The variables and arrays have to be grouped into clusters which would be implemented by a single storage component. The main motivation behind such a grouping is to minimize the number of pins and the routing area. Each group is then mapped to a register file or memory if it has an array or multiple variables, or mapped to register if the group has a single variable.

Storage grouping is essentially a tradeoff between interconnect cost and the performance measured by the time required to access the variable or array. The factors which determine whether two variables can be grouped together are:

1. **Sequential/Concurrent access**: Two variables that are accessed concurrently are less likely to be merged into a register file/memory as would be two variables that are always accessed sequentially. This is due to the fact that the access conflict in the presence of insufficient number of access ports would adversely affect performance.

2. **Access Frequency** : A variable that is accessed very frequently is less likely to be merged with other variables. Again, this is due to the fact that a frequently accessed variable will cause more access conflicts and thus, performance deterioration.

3. **Variable Size**: Two arrays are less likely to be merged if their combined size exceed the maximum size memory available in the design library.

RP Example

Figure 7 shows how storage elements were grouped in the RP example based on the above criteria. The PC, $IR1$, $IR2$, $IR3$ and CTR variables were not grouped with any other variable since they are accessed very frequently and often concurrently. The RF, SP and $ZERO$ were grouped together because they were implementable using a 4-port register file to account for all possible concurrent accesses forming the register space, REG_SPACE. The arrays $PMEM$, $DMEM$ and $EMEM$ remain in separate groups.

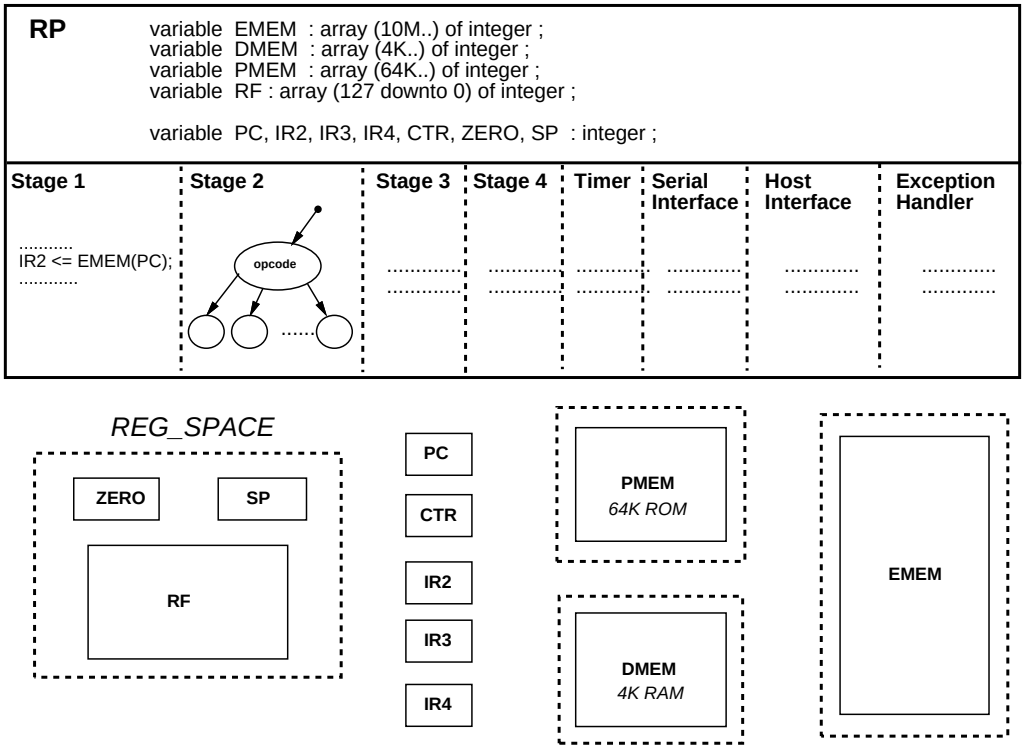

Figure 7: Storage Grouping in RP

4.3 Storage Implementation

Having grouped the arrays and variables, we need to select from the design library the component that will be needed to implement the groupings. The main reason for determining the exact components that will implement the storage groupings is to obtain the behavioral and performance characteristics

of storage elements. For example, having selected a specific memory from the library to implement a group of variables, we know the access times for those variables, as well as the details of the interface to those variables (number of control lines and data lines, communication protocol etc.). This will enable us to get more accurate estimates for metrics that are required for the next step in the methodology - for performance and the number of pins required for communication.

Given a group of variables/arrays, the storage component is selected from the component library based on the following:

1. number of *words* in the group.
2. length of each *word*.
3. access time.

Once the appropriate memory/register file has been selected, the individual variables and arrays in the group have to be mapped to the address space of the selected component. The declarations for the arrays/variables in the group are now removed from the SpecChart and a new process is added for each group, reflecting the behavioral and structural characteristics of the storage component selected.

In the original SpecChart specification, the memories were being accessed as arrays. Now each storage group is an independent process. Appropriate communication protocols have to be inserted in all the processes which were accessing the memories for communication with the new storage component processes. This enables us to simulate the entire SpecChart with the *models* of the storage components. In case there more concurrent accesses to a memory with fewer ports, then an arbiter process may have to be inserted to resolve access conflicts based on some priority scheme.

RP Example

Figure 8 shows the modified SpecChart after storage implementation. A `128 x 32` register file, a `64K x 32` ROM and a `4K x 32` RAM were selected from the design library for implementing the $REG_SPACE, PMEM$ and $DMEM$ respectively, whereas a memory is selected for the external memory, $EMEM$. The declarations for these objects are replaced with concurrent behaviors (processes) containing the behavioral description of the selected storage components. Communication to the storage components is now through communication channels which have a well-defined protocol associated with them.

$Stage1$ and $Stage4$ both communicate with the external memory, $EMEM$, over the communication

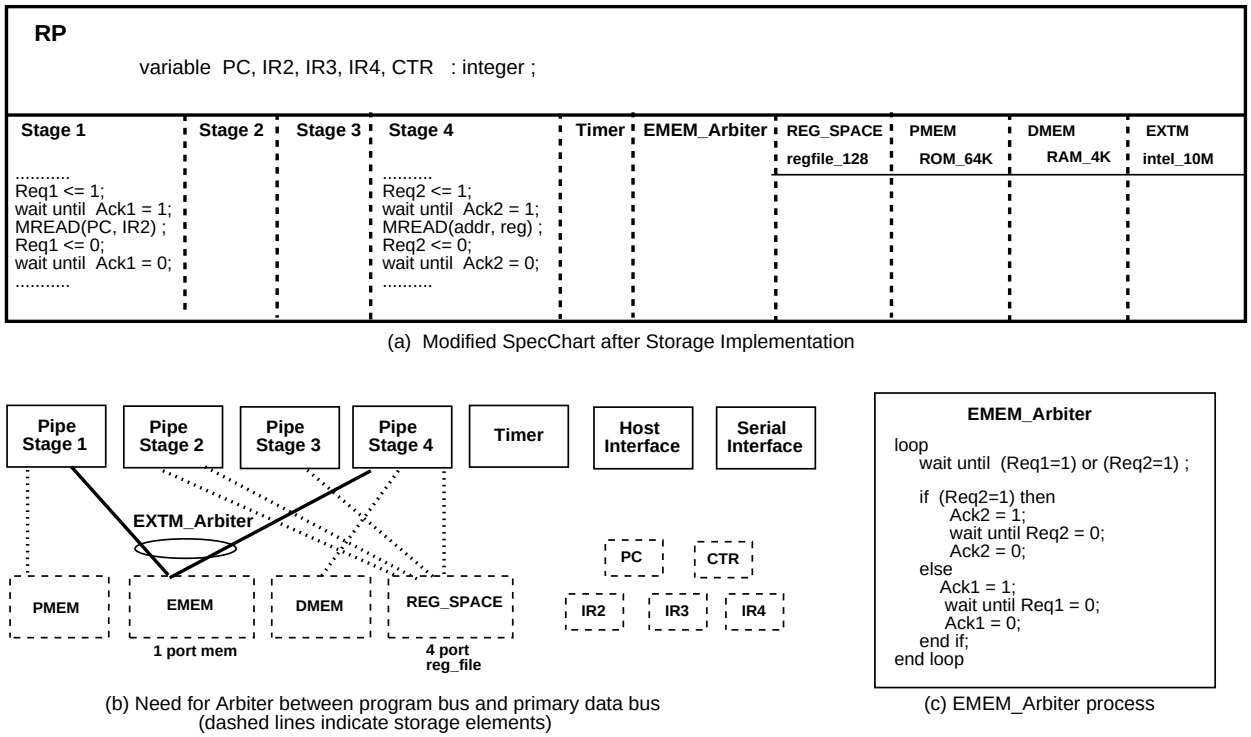

Figure 8: Storage Implementation in RP

channels shown by the bold lines in Figure 8(b). Assume that the $EMEM$ was mapped to a single port memory. There is a possibility of an access conflict in case both the stages access the memory at the same time. Thus an arbiter process, $EMEM_Arbiter$ (Figure 8(c)), has to be added which resolves contention for memory accesses between $Stage1$ and $Stage2$. The two stages will have to issue requests to the arbiter process prior to accessing the memory. The arbiter will decide which stage gets access to the memory, In the RP, $Stage4$ has a higher priority than $Stage1$. The pipeline will stall in case such a conflict occurs.

4.4 Behavioral Grouping

The specification of the system now consists of a collection of behaviors. These behaviors are now grouped into chips/modules. The motivation for behavioral grouping is:

1. **Minimizing interconnect** between behaviors by grouping the those that communicate most frequently together.

2. **Minimize cost** by using fewer and/or smaller chips or mapping behaviors to standard components.

3. **Reduce problem size** for subsequent synthesis tools which can now can now perform synthesis on groups of behaviors taken individually.

Before behavioral grouping can be performed, estimates of design parameters such as area, performance and interconnect must be determined for each behavior. Based on the estimates, a constraint-driven partitioner will group the behaviors in the design into chips/modules. The specification must be updated to reflect the behavioral groupings determined.

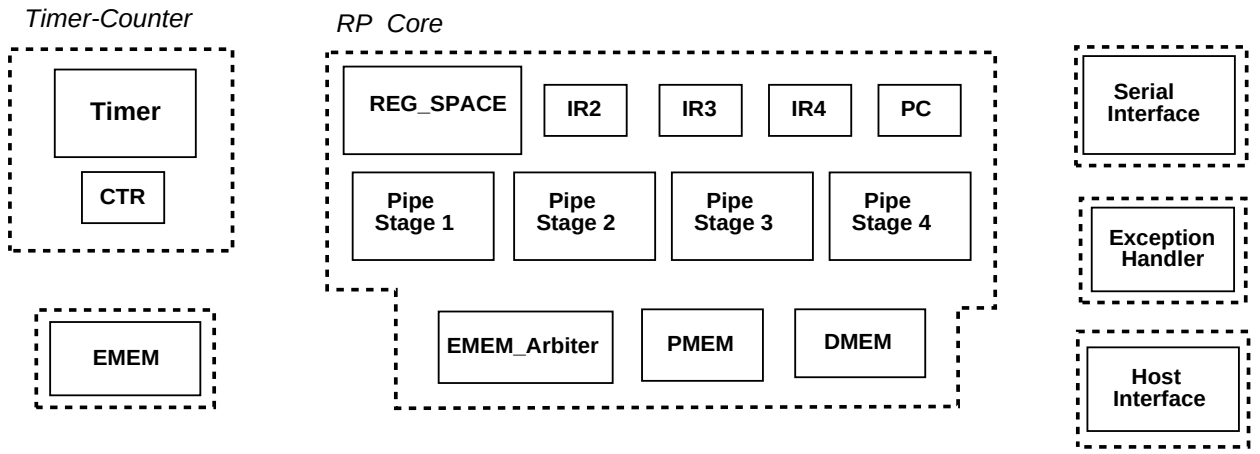

Figure 9: Behavioral Grouping in RP

RP Example

In the RP example, the results of behavioral grouping are shown in Figure 9. The four stages of the execution pipe along with the register space, program counter PC, the three instruction registers, $EMEM_Arbiter$, program memory $PMEM$ and data memory $DMEM$ are grouped together to from the RP Core. The counter register, CTR, is grouped together with the behavior $TIMER$ which accesses them frequently, to minimize execution time. The other behaviors - $EMEM$, the serial and host interfaces and the exception handler are assigned to individual groups by themselves.

It should be mentioned here, that if a storage component is grouped with the behaviors which access it, the storage component need not be replaced with the corresponding storage behaviors. The storage component REMAINS in the SpecChart as a variable declaration. Thus, in the refined SpecChart after behavioral grouping PC and REG_SPACE will remain as variable declarations global to the behaviors $Stage1, Stage2, Stage3$ and $Stage4$. The separate process created for REG_SPACE in the Storage Implementation step is deleted.

4.5 Channel Grouping

The specification at this stage consists of groups of behaviors communicating over communication channels. Each communication channel has an associated protocol which defines the number of control and data ports in the channel as well as the behavior over those ports. To satisfy the pin constraints, we may have to group channels into a bus, thus sharing data and control lines over several channels.

Channel grouping is essentially a pin vs. performance tradeoff. Communication channels are merged based on the following criteria:

1. **Frequency of Concurrent Accesses**: Two channels are less likely to be merged if they could possibly be in use concurrently. Access conflicts for transfering data across a merged channel will result in performance degradation for the behaviors involved. The frequency of data transfer across a channel is used to determine the feasibility of grouping that channel with another concurrently used channel.

2. **Reduction in Interconnect cost**: Two channels are more likely to be grouped together if the merged channel results in a greater reduction in the interconnection cost measured as the number of control and data lines (or pins).

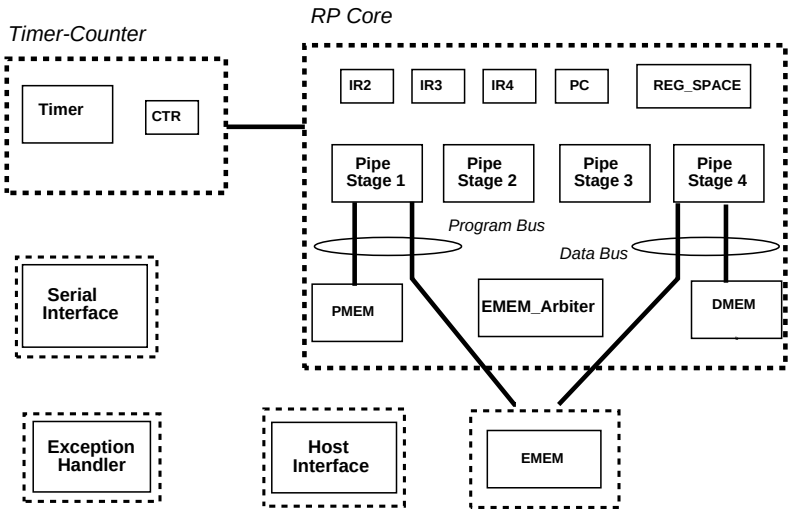

Figure 10: Channel Grouping in RP

RP Example

Based on the criteria above, Figure 10 shows channel grouping applied to the RP. The figure shows the communication channels only the groups of behaviors determined in the previous step. $Stage1$

reads data from both $PMEM$ and $EMEM$. Since, at any time, the data can be read from either $PMEM$ or $EMEM$, the two communication channels are mutually exclusive and can be merged to form what is known as the Program Bus in the RP. Similarly the two channels used by $Stage4$ to transfer data to/from the $DMEM$ and $EMEM$ can be merged to form the Data Bus. The channel for communicating between the RPcore and the Timer-Counter circuits is not merged with either of the two buses since it can be used concurrently with both of them to transfer data.

4.6 Interface Implementation

Given a set of behaviors communicating over channels and a grouping of channels representing shared buses, several tasks need to be performed with respect to the interface between the behaviors. To understand the various tasks performed in Interface Synthesis, consider Figure 11(a). Two behaviors, A and B are communicating over channel C. For communication to take place, the two protocols (P_a and P_b) at the ends of channel C, must be *compatible*, i.e. the number of control and data lines and the order in which values are assigned to them must be identical.

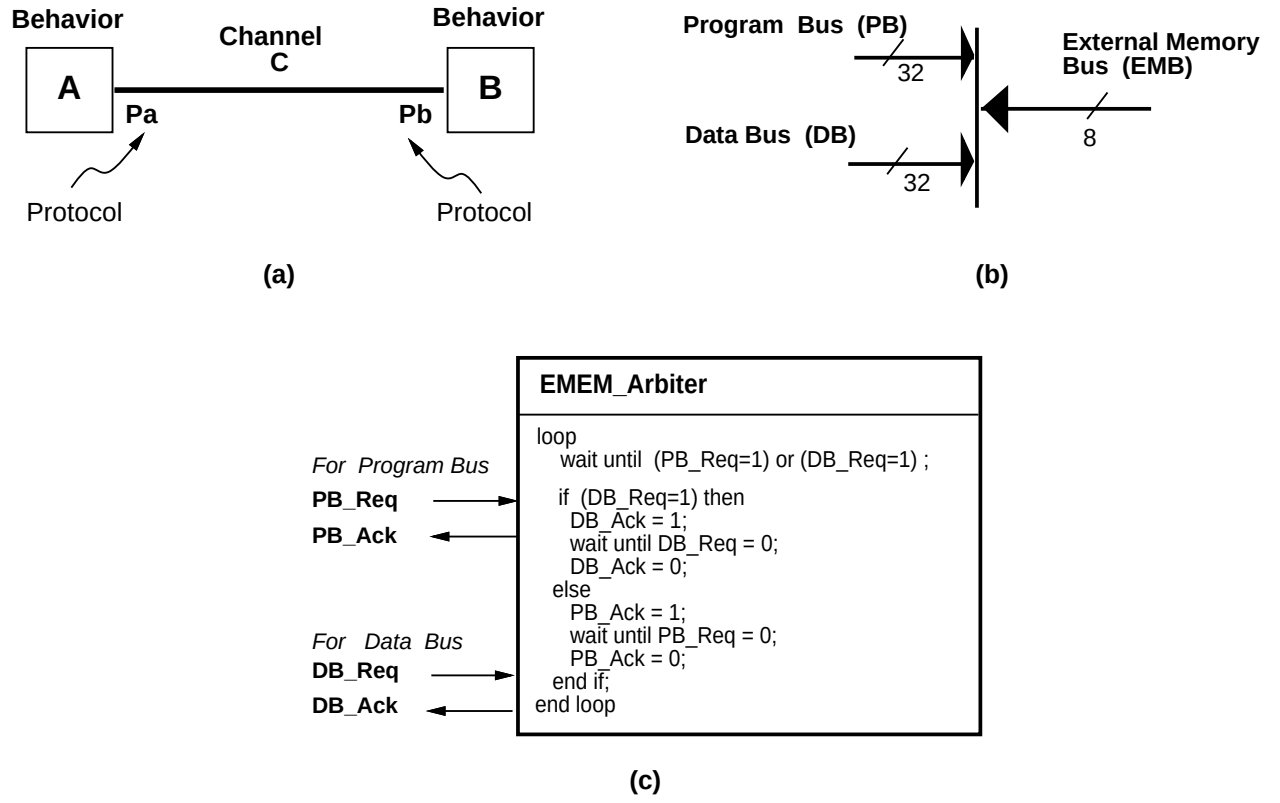

Figure 11: (a) Behaviors communicating over a channel. (b) Interfacing the RP with external memory (c) Generating an arbiter for the Program and Data buses.

Depending on whether the communicating behaviors are mapped to custom or standard components, three cases arise.

1. If both the communicating behaviors are mapped to custom components, we have the flexibility of changing the communication protocol in both the behaviors. For example, assume that two channels transferring 32 and 8 bits of data have been grouped. The common protocol is first determined from a given pin-constraint or a desired data transfer rate. If a pin constraint of 10-pins is given for the merged channel, we can have an 8 bit wide data bus, where the 32 bits of the first channel are transferred in four steps. If a high bit-rate is specified as a constraint, then we might want to implement the merged channel using a 32-bit data bus.

2. In case one of the communicating behaviors is bound by the designer to a standard off-the-shelf component, the protocol for that behavior is fixed. We must modify the description of the other behavior to include an exact complementary protocol to ensure that the two behaviors can still communicate over the same channel.

3. Finally, the designer may decide to bind both the behaviors communicating over a channel to standard components. In this case, the corresponding protocols in both behaviors may be different and incompatible with each other. Thus, an *interface process* has to be generated and inserted between the two communicating behaviors to ensure compatibility. The two behaviors now communicate with the interface process which effectively masks the two incompatible protocols from each other.

In addition to generating a merged protocol for the channel group, it may be necessary to generate an arbiter process which will resolve access conflicts to the merged channel. In this case, modifications can be made to the behaviors to account for the request/acknowledge control lines.

A group of channels can also be implemented by mapping the channel group onto a designer-preferred standard bus (e.g. MULTIBUS) protocol. Thus, for each channel in the group, the control and data lines will have to be mapped onto that of the specified standard bus.

Example

Figure 11(b) shows how the RP Program and Data buses communicate with the external memory bus. The Program and Data buses have 32-bit wide data lines, whereas the memory consists of 8-bit wide data. The Program bus can only read from the memory while the Dat bus can read or write to the memory.

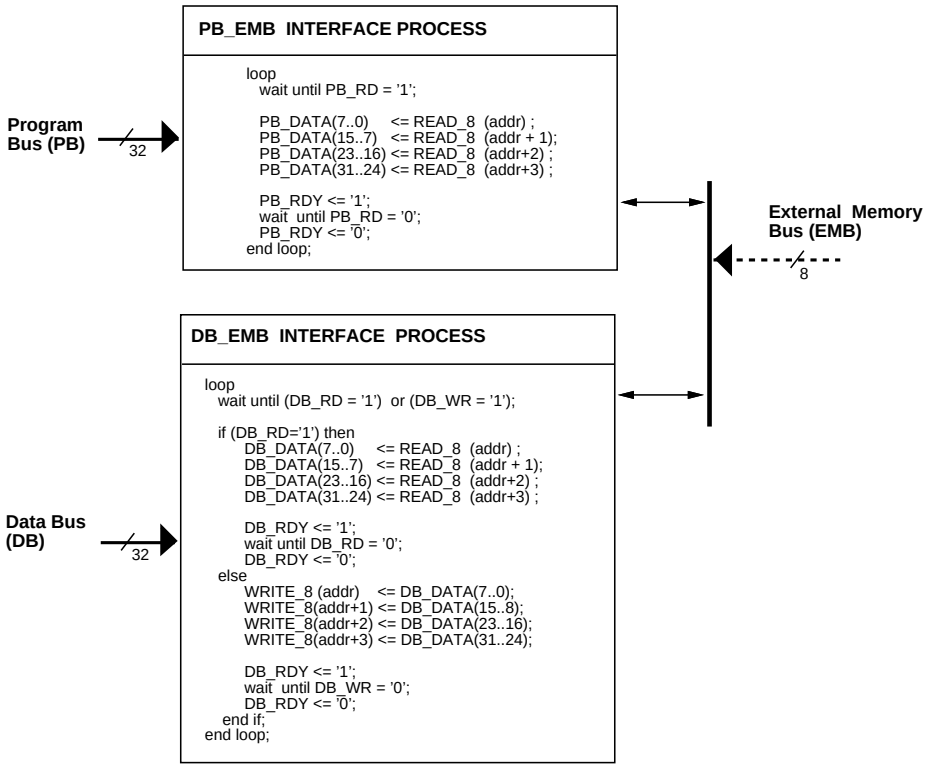

Figure 12: Interface Process Generation in RP

The $EMEM_Arbiter$ gives priority to the Data bus over the Program bus in case both of them attempt to access the external memory simultaneously. At this stage, the REQ and ACK lines required for arbitration are considered separately from the control signals of the Program and Data buses as shown in Figure 11(c).

The 32-bit Program and Data buses have to be made compatible with the 8-bit wide external memory. This is achieved by generating an *interface process* which would ensure that whenever the Program and Data buses access the memory, four accesses of the 8-bit memory are made. The resulting interface processes are shown in Figure 12. These interface processes form the Expansion Bus Interface Unit in the RP.

As a final stage of interface implementation, it may be possible to optimize the number of control signals by possibly merging the REQ/ACK signals required for any arbitration with the control signals of the communication protocol. Thus, in the RP example, the *read* and $\overline{PB\ data\ rdy}$ signal of the Program bus can also serve as the REQ and ACK lines for arbitration.

5 Behavioral Implementation

Behavioral implementation provides a path to a design implementation. For each group of behaviors, the system specification must be mapped to hardware or software components. Hierarchical behaviors are flattened. Behaviors communicating over channels are now converted to processes communicating over global signals and ports. For each concurrent behavior we must determine whether it is to be implemented as a software or hardware component.

5.1 Hardware Implementation

In order to implement the design in hardware, for each VHDL process, an RT block structure must be defined which should be capable of performing the computations specified in that process. This is achieved by using a VHDL synthesis tool such as the VHDL Synthesis System (VSS) [5, 6, 7]. VSS takes a VHDL description as an input. Given a resource constraint (such as the maximum number of adders that can be used), VSS schedules operations in the description to maximize the system performance. Alternatively, given a performance constraint, VSS will attempt to schedule the operations to satisfy the constraint using a minimal number of resources. In addition to determining a schedule for the operations in the description based on resource or performance constraints, VSS also binds operations to specific instances of the functional units which can execute the operation. Finally, an RT structural netlist is generated from which a layout can be obtained by using logic and layout synthesis tools.

5.2 Software Implementation

In case a software implementation is desired, suitable code targeted towards a standard microprocessor must be generated. To achieve this, first, each behavior is converted to equivalent programming language (C language for example) statements. Each process has to be mapped to a specific microprocessor. To ensure that the communication between the processes is maintained, interfaces need to be defined between the software and hardware components to which these processes are mapped. Finally, the machine code for the microprocessor has to be generated for each process from the programming language representation.

6 Conclusion

We have defined the SpecChart language for specification capture and a set of transformations that are required to refine that specification into a description that can be used as an input to commercial synthesis tools. Our methodology also includes software/hardware partitioning that allows the final description to be mapped on standard off-the-shelf components such as processors. This is achieved by compiling the description into machine code using standard programming language compilers available for selected standard processors.

Since the refinement process must be iterated several times to achieve the best possible tradeoffs, we envision a conceptualization environment that will allow designers to quickly evaluate the gains in his/her description. This requires work on three components of such an environment: (1) a component base with VHDL models for different technologies, (2) fast estimators of design quality metrics for different architectural styles and different technologies (e.g., custom, gate-arrays, FPGA's, standard components, software implementations, etc.) and (3) an appropriate human interface for display of quality metrics, different user views and subsequent refinements of the specification [8]. The SpecSyn [1] framework is the first step in this direction.

7 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #91-DJ-269). We are grateful for their support.

8 References

[1] S. Narayan, F. Vahid, and D. Gajski, "System Specification and Synthesis with the SpecCharts Language," in *Proc. of the International Conference on Computer-Aided Design*, 1991.

[2] D. Harel, "Statecharts : A Visual Formalism for Complex Systems," *Science of Computer Programming 8*, 1987.

[3] S. Narayan, F. Vahid, and D. Gajski, "Modeling with SpecCharts." UC Irvine, TR ICS 90-20, 1990.

[4] H. Juan, N. Holmes, S. Bakshi and D. Gajski, "Top-Down Modeling of RISC Processors in VHDL." UC Irvine, TR ICS 90-21, 1990.

[5] J. Lis, *Behavioral Synthesis from VHDL Structured Modeling.* PhD thesis, University of California, Irvine, January 1992.

[6] J. Lis and D. Gajski, "Synthesis from VHDL," in *Proc. of the ICCD*, 1988.

[7] J. Lis and D. Gajski, "VHDL Synthesis using Structured Modeling," in *Proc. of the Design Automation Conference*, 1989.

[8] D. Gajski, N. Dutt, C. Wu, and Y. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1991.