The Chinook Hardware/Software Co-Synthesis System [1]

Pai Chou, Ross Ortega, Gaetano Borriello

Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350

Technical Report 95–03–04
March 14, 1994

[1] This work was supported by by the ARPA/CSTO Microsystems Program under an ONR monitored contract (N00014-91-J-4041). The authors' email addresses are {chou,ortega,gaetano}@cs.washington.edu.

The Chinook Hardware/Software Co-Synthesis System

Pai H. Chou, Ross B. Ortega, Gaetano Borriello

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

March 14, 1995

Abstract

Embedded systems are becoming more commonplace and are being designed by larger numbers of designers with ever tighter constraints on design time. Unfortunately, computer aided design tools for embedded systems have not kept pace with these trends and are still fundamentally identical to those used twenty years ago. The Chinook co-synthesis system, under development at the University of Washington, addresses the automation of the most time-consuming and error-prone tasks in embedded controller design, namely: the synthesis of interface hardware and software needed to integrate system components; the migration of functions between processors and/or custom logic; and the co-simulation of system specifications before, during, and after synthesis. In this paper, we describe the principal elements of the Chinook system and discuss its application to a variety of embedded designs.

Category of interest: (C) Hardware-software Co-design.

1 Introduction

Embedded system designers, in varied industry segments that include consumer electronics, automotive control, and medical equipment, are facing increased pressure to create products quickly and inexpensively. This trend is coupled to the increasing levels of integration, performance, and programmability achievable in off-the-shelf integrated circuits including microprocessors, programmable logic, and devices such as LCDs, network interface controllers, and speech generators. Designers find using these devices to be advantageous because of their low cost and the way in which they facilitate rapid realization of designs not only for prototyping but for production as well. In fact, with embedded controllers now found in everything from consumer electronics and automobiles to smart credit cards and medical equipment, many products have declining lifetimes that make custom integrated circuits a less economically viable option.

The job of the embedded system designer has also changed. In addition to time to market constraints, the designer must worry about correctness and cost effectiveness of the implementation. Thus, designers have a need to explore a large design space of potential solutions, yet no integrated CAD tools are available to help them with this task. The design must be quickly defined and simulated and then mapped onto the cheapest combination of components. Unlike general-purpose computers embedded systems are designed and optimized to provide specific functionality. Thus, the most time consuming and error-prone task in embedded system design is precisely the detailed mapping of the abstract functional specification onto the target components. In fact, the process is so time-consuming that many designers fix the target architecture and system components well before a complete evaluation of the final system and perform only one mapping. This often leads designers to *over-design* their systems with faster processors or larger capacity logic devices then really needed, thereby increasing the cost. If the target architecture were to prove inadequate due to performance or capacity constraints, designers would face a costly re-mapping process.

It is clear that design exploration tools to automate the mapping process and thus provide faster feedback on design decisions are sorely needed. Many design automation tools and frameworks have been proposed to address a few of these problems. These tools either look at high-level specifications but do not assist with the actual implementation, or they help with individual parts of the implementation but do not provide a system view. Examples of the former include behavioral simulators and formal specification languages while examples of the latter include compilers, board layout tools, and logic synthesis systems. Recently, tools for dealing with the hardware and software portions of the system have been proposed, but these have not addressed the system integration issues that dominate the design cycle.

2 Taxonomy of Hardware/Software Co-design

The field of hardware/software co-design of real-time embedded systems can be organized along three principal dimensions: the implementation technology, the application domain, and the aspect of the design cycle.

2.1 Technology

An embedded system may be implemented with a number of technologies, including off-the-shelf components, programmable logic, and full-custom or semi-custom ASICs. Examples of such technologies include interface controllers, FPGAs, standard processor cores, or cores that are either custom designed or enhanced with custom datapath and I/O logic. The choice of technologies has a significant impact on the price/performance of the embedded system. ASICs provide higher performance but can be expensive to design and are difficult to modify once fabricated. FPGAs and processors are reprogrammable and can be used to quickly prototype a system. Because they are available in large quantities, they often have competitive price/performance ratios to custom logic. Increasingly, more functionality is being moved into software because microprocessors can deliver the desired performance, obviating the need for much custom logic. Thus, the design burden is shifting to software and increases pressure to automate tasks such as device driver generation and scheduling to meet real-time constraints.

2.2 Domains

Embedded systems can be divided into two principal domains, control-dominated and data-flow, based upon the characteristics of the application. In the data-flow domain, data is sampled at regular intervals and processed in the same order. The behavior of the system remains the same over time. In each time step, a set of mathematical operators are applied to a window of data samples. Digital signal processing (DSP) systems are the canonical example for the data-flow domain. Control dominated-systems span a much wider range and are characterized by complex conditional or modal behavior rather than math-intensive computations. An example of a control-dominated system is a network controller or avionics control system. Of course, many systems contain elements of both domains but usually one or the other dominates the designer's attention.

2.3 Design Problems

The problems in embedded systems design include specification of behavior and timing constraints, partitioning, interfacing, scheduling, code-generation, analysis, simulation and debugging. Point tools either exist or are being contemplated for all these aspects of the design process. We focus our discussion of this dimension on control-dominated applications.

Specification captures the behavior and requirements of a design. This is for the most part done informally using a mixture of natural language documents, pseudo-code, and block diagrams. This approach has made design maintenance, upgrading, and retargeting very time intensive and sometimes impossible. Several formal specification methods have been proposed including finite state machines [4], Petri nets [18], and CSP [15]. Today's tools lie somewhere in the middle – specifications are written in a high-level programming or hardware description language that is simulatable. This enables some early validation of the design. There is still no accepted formalization of the timing and performance constraints that are critical in many embedded applications. Without these constraints explicitly represented, designers must devise and validate software schedules and interactions between components by hand. Simulators can help with this task but are limited to the tests performed explicitly. Formal verification or synthesis techniques are needed to guarantee that constraints are satisfied.

Partitioning is the process of determining the components on which to implement portions of system functionality. This may be a split between a processor and auxiliary logic or between a set of processors. Attempts at automating partitioning have included simulated annealing algorithms [9] and hardware to software migration [12] but have for the most part ignored the problems of interfacing and communication between the parts. In fact, designers spend a large fraction of the time in interfacing system components to each other and the operating environment (including user interfaces) because it is where the bulk of errors lie. Yet, interfacing remains one of the least addressed areas in many co-synthesis tools. The interfacing task may involve both hardware and software aspects of the interface as well as low level timing concerns that may require glue logic. Interface generation has been described in [20], though the synthesis of interface software is not addressed.

Other aspects of the embedded system design cycle include retargetable code-generation (for different off-the-shelf processors as well as custom designed ones). Performance estimation of software is of critical importance and is required if partitioning to meet timing constraints is to be done with confidence. Analysis tools are needed to predict execution times, and possibly the size, of code fragments. Finally, simulators, debuggers, and profilers are needed to evaluate the final design at a detailed level.

3 The Chinook Co-Synthesis System

At the University of Washington, our approach to the co-synthesis of real-time reactive embedded systems is embodied in Chinook, a tool that generates complete design specifications given a single high-level specification of the desired system functionality. Several features distinguish Chinook from other work in this area. Each is motivated by the observations and concerns outlined in the previous two sections. Using the taxonomy of section 2, Chinook is intended for control-dominated designs constructed from off-the-shelf components. It addresses the aspects of the design process whose automation will provide the most benefit to designers in terms of shortening the design cycle, permitting more design space exploration, and automating tasks that are error-prone or cumbersome.

The following elements of the Chinook system are where the principal innovations lie. It is important to note that what makes Chinook unique is the combination of these elements rather than any single one.

- **Single specification.** A designer writes one specification in a single specification language with explicit timing/performance constraints rather than separate netlist, hardware description, and software languages all with implicit constraints. This is key to the retargetability and maintainability of the design.
- **One simulation environment.** The high-level specification of the design can be simulated directly to help debug the designer's intent as well as operational aspects of the design. The final synthesized result, and any intermediate steps, can be simulated in the same environment and augmented with additional tools (e.g., debuggers and profilers for software).
- **Comprehensive software scheduling.** Chinook synthesizes the appropriate software architecture for the timing requirements of the system: low-level partitioning to ensure signal-

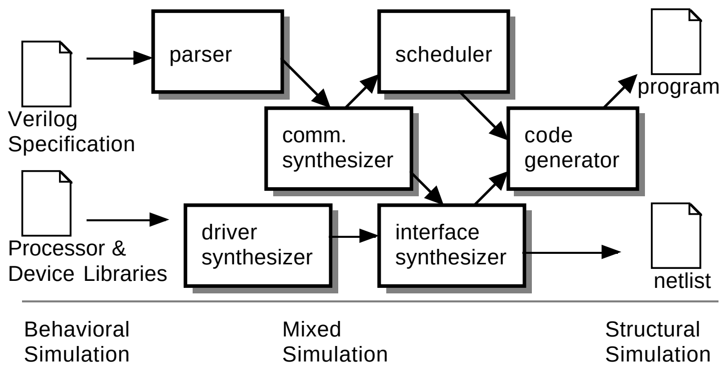

Figure 1: The Chinook Co-Synthesis System

ing constraints are satisfied (possibly by synthesized hardware modules), static fine-grained scheduling to tailor device drivers, and customized dynamic schedulers and interrupt handlers.

- **Interface synthesis.** Interface hardware and software between system components (including peripheral devices as well as multiple processors) is automatically synthesized with appropriate changes reflected in interprocessor communication and device drivers.
- **Complete information for physical prototyping.** Chinook generates a complete netlist for assembling the system and complete code for its processors to run. After co-synthesis, the system is ready to be assembled and evaluated in its intended environment.

The Chinook co-synthesis system consists of the parser, the processor/device library, the device-driver synthesizer, the interface synthesizer, the communication synthesizer, the scheduler, and the simulator (see Figure 1). The parser accepts a system description in annotated Verilog. In addition to a behavioral specification, it also contains a structural specification that instantiates the principal components of the system, including processors, peripheral devices, and standard interfaces. The device library contains detailed generic specification of device interfaces (in the form of timing diagrams and Verilog code) and models for their simulation (in C). For processors it contains specifications of their interfaces as well as timing schemas for software run-time estimation [19]. The device-driver synthesizer compiles the timing diagrams and Verilog device drivers into customized code for the given processor and makes low-level partitioning decisions to meet signaling constraints. The interface synthesizer allocates I/O resources to connect a processor to the peripheral devices it will control, and customizes the access routines to reflect these assignments. The communication synthesizer generates the hardware and software needed for interprocessor communication. With all resources allocated, the scheduler generates C code to meet real-time constraints in software. The C code is compiled by a processor-specific compiler. Chinook also outputs the netlist, including the necessary glue logic, to construct the desired system.

Chinook does not attempt does not attempt several tasks. It does no high-level partitioning of functionality between hardware or software or between processors. Instead, it assumes that designers involved in design exploration are in a better position to make these assignments at the module and/or task level. Chinook does not compile code to the target processor(s). It assumes not only the existence of the appropriate C compilers but also that these will be able to provide the scheduler with feedback on the expected run-time of code fragments. Finally, in its focus on real-time reactive systems, it assumes that caches are not employed.

4 Specification

The single Verilog file provided as input to Chinook contains both behavioral and structural constructs. The behavioral style imposed by Chinook enables the expression of real-time reactive behavior as well as facilitating partitioning. The structural component merely lists the processors, peripheral devices, and communication interfaces that will be used. That is, the principal components of the system to which the designer would like to evaluate a mapping of the desired functionality. Chinook expects the designer to *tag* tasks and/or modules with the processor that is preferred for their implementation. The implementation of untagged modules/tasks is assumed to be in hardware. This separation of functionality from components allows the designer to quickly explore the design space by instantiating different processors and alternative peripheral devices without modifying the behavioral specification. All interactions with the devices and interfaces are specified using a procedural abstraction layer. As long as two interfaces (e.g., SCSI and PCMCIA) support the same access routines (e.g, read and write) they can be easily interchanged.

To model the reactive behavior of control-dominated applications, we organize the control states of the system as a set of *modes*. Each mode defines a behavioral regime, that is, how the system should respond to its inputs. A mode also defines a scope for a set of timing constraints that must be satisfied while the system is within that mode but not necessarily when it is operating outside of it. Modes are similar to the hierarchical states of [14] in that they can capture both sequential and concurrent behavior.

Chinook allows the specification of real-time requirements in terms of minimum and maximum separation between I/O events, namely events between system components or between the system and the environment. At the low level, the constraints may correspond to setup and hold times, or simply the sequencing constraints between successive I/Os. At the high level, min/max separation can also be used to express *response times* to system inputs and *rate* constraints on performance [8].

In a given mode, the system's responses are defined by a set of *handlers*. Conceptually, they are event-triggered routines, but their activation conditions are checked by a time-triggered loop. Handlers respond by generating I/O events and/or causing a mode transition. A handler consists of a trigger condition and a body. The trigger condition is an event expression consisting of inputs from the environment and other handlers. When the trigger condition evaluates to true, the handler body is executed. For example, a network interface chip may signal that a message is pending and this triggers a handler to read that message. Note that the handler body can be in software, hardware, or a combination of the two, depending on its tag and the ability of the processor to meet the timing constraints in the handler. From a specification point of view, a handler is executed atomically, but may be interleaved by the scheduler.

5 Scheduling

Embedded systems have timing constraints at different levels. Their interaction with the devices and the environment must respect not only low-level signaling constraints but also performance requirements such as rate and response time constraints. To satisfy these high-level constraints, designers have used *process-based* scheduling techniques based on operating systems concepts [17, 3]. These techniques are coarse-grained, priority-driven, and dynamically preemptive. They assume that the processor does not perform I/O directly and the processes are independent of each other. Since all timing constraints are coarse-grained, overhead incurred by the executive during preemption can be dismissed. However, many embedded systems must perform direct I/O and meet fine-grained timing constraints. These constraints are much more difficult to meet because the scheduler cannot afford to incur much, if any, run-time overhead, and at the same time must handle uncertainties in the execution delays. Instead, Chinook statically schedules all low-level I/O and high-level operations as grouped in modes. A customized dynamic scheduler may be generated for the larger modes (*i.e.*, those at the top of the mode hierarchy).

Chinook uses a static, nonpreemptive scheduling algorithm to meet min/max timing constraints on fine-grained operations with delay ranges [6]. It determines a serial ordering for the operations, and inserts delays to meet minimum constraints, if necessary. Because the complexity of the problem is NP-hard, we employ heuristic ordering functions to help the exact algorithm quickly find a valid and short schedule. Experimental results show that our best heuristic consistently outperforms a heuristic that solves the same problem inexactly [13].

At the high level, rate constraints are specified on a reference event between successive iterations, and response times are constraints on the time it takes to do a mode transition. In statically scheduling the software, Chinook first converts handlers within a mode into a single handler containing their bodies, possibly using unrolling, and then schedules this single partially-ordered handler by interleaving [8]. Note that a mode transition may be triggered by one of the handlers before other handlers run to completion, and the scheduler must maintain the integrity of all handler states. We do not use critical regions to achieve atomic execution because they disable interleaving, which is necessary when servicing devices with long separation between sequential events. Instead, Chinook allows the user to define *safe points* in the handlers, where potential mode transitions can safely occur [5]. All parallel handlers must reach their safe points before a mode transition is allowed to take effect.

6 Interface Synthesis

Interface synthesis is the realization of communication between components via both hardware and software elements. Chinook handles a wide range of interface synthesis problems. At the lowest level, Chinook synthesizes device drivers directly from timing diagrams. It generates customized code for the particular processor being used, and separates out the portions that cannot be implemented in software by synthesizing the required external hardware. For processors with general purpose I/O ports, Chinook employs an efficient heuristic for connecting devices and processors using minimal interface hardware. For processors without I/O ports, Chinook automatically implements the interface using memory-mapped I/O including allocating address spaces and generating the required bus logic and instructions.

These synthesis solutions require knowledge about the interfaces of the processors and the devices, which are captured in the libraries. A processor is defined by its I/O resources, built-in functionality (e.g., serial-line controller, timer, etc.), and detailed architecture templates (e.g., down to the specific resistors and capacitors required for power-up reset). A device description contains interface information including ports and skeletal access routines that encapsulate timing diagrams. After successful interface synthesis, Chinook updates the access routines by binding the device ports to the processor's I/O ports or memory bus, and taking into account any intervening glue logic that it may have synthesized. By managing these connectivity details and generating the interface across the hardware/software boundary, the interface synthesizer completes the design and enables simulation and evaluation at the final implementation level.

6.1 Driver Synthesis from Timing Diagrams

At the most detailed level, device interfaces are described in data sheets in the form of timing diagrams. They show the sequences of signaling events that make up I/O transactions across the interface. These timing diagrams are usually annotated with timing requirements, timing delays, and timing guarantees. The first of these three are requirements imposed on the user of the interface, while the second two are timing promises made by the device as long as the user conforms to the requirements. When new devices are added to the device library, these constraints and their corresponding timing diagrams are entered via a waveform editor [11]. Chinook parses these waveforms and synthesizes the device driver code by choosing a linear schedule of controller events, and inserting additional interface glue logic where necessary [22].

6.2 I/O Port Allocation

Many processors used in embedded systems include I/O ports that can be used to directly sense and manipulate the processor's environment. These ports can be accessed from software like registers thus providing a low-cost and straightforward interfacing mechanism. Chinook provides a port allocation scheme that uses the minimal amount of glue logic. Furthermore, device access routines are customized to reflect the assignments of pins [7]. The key idea is that an I/O port may be able to service multiple devices without glue logic and without performance penalties. These devices have interfaces that are able to isolate themselves from the shared bus, and become active only when the appropriate control signals, or *guards*, enable them. Thus, a guarded interface of a device can share the same I/O port with other devices because their interfaces cannot be active at the same time. If necessary, the port allocator inserts glue logic to add guards to previously unguarded interfaces, so that they can share busses. Chinook can also synthesize ports to create a new interfacing point for additional devices. Hardware is synthesized to create a new port on the processor's memory bus. This module decodes addresses and translates them into control signals to read and write the new I/O pins.

6.3 Memory-Mapped I/O

When I/O ports are too inefficient (due to multiple instructions to manipulate their values or too much additional hardware) or are unavailable (as is the case for higher-performance processors), Chinook synthesizes the interface using memory-mapped I/O. First, many parts, processors as well as peripheral devices, are designed with memory-mapped I/O in mind. They contain built-in address matching logic and can be connected to the memory bus with little or no glue logic. Second, many components without built-in address comparators can still be connected with little or no glue logic, depending on the available address space the user reserves for I/O. Devices are allocated portions of the address space of the processor controlling them. If the allocation is done intelligently (i.e., using one-hot, binary, or Huffman encodings when possible) the amount of address matching logic required can be minimized.

Memory-mapped I/O is also a preferred method of interprocessor communication and can be used to support both point-to-point and shared memory schemes. If we are to allow a designer to explore mapping of functions to multiple processors, then the mapping tools must automatically synthesize the interprocessor communication hardware and software. Essentially, the view from one processor is that the other processors are just more peripheral devices requiring their own device drivers.

7 Communication Synthesis

Requirements for faster response times and increased modularity frequently guide embedded system designers to employ multiple processors. These processors are often heterogeneous as cost and modularity concerns drive designers to tailor processors to specific functions. CAD support is non-existent for these types of systems. There are not even debuggers to support concurrent development of programs on two identical processors. Designers find heterogeneous multiple processor systems the most difficult to debug and thus constrain designs unnecessarily just to make debugging tasks tractable.

Chinook provides support for interprocessor communication by synthesizing the hardware and software needed to transfer data between processors. A designer tags the procedures and modules with the processor that should be used to implement them. Chinook then determines the data that must be transferred and the mechanism to use for those transfers including the interconnections between the processors, glue logic, and/or buffers and memory.

In meeting timing constraints, Chinook will adjust the interface between the software running on the processors. Consider the case of a fast processor communicating with a slow one. Handshaking with the slow processor may cause the fast one to violate its constraints. Buffers can reduce the load on the fast processor by eliminating direct handshaking. The communication becomes non-blocking and data may be processed in bursts.

7.1 Interprocessor Communication Synthesis

When considering communication in multiple processor systems, many new issues arise including predictability, interconnect topology, access to peripheral devices, and communication protocols. The interconnect topology could be bus-based, point-to-point, or a hybrid scheme. A peripheral device may only be accessible via a designated processor or many processors may have shared access. The communication protocol may be contention based or statically scheduled, blocking or non-blocking, and master-slave or peers. Each choice has impacts on performance, predictability, and the complexity of scheduling and hardware required. Chinook supports most of these choices, but by default uses a model suitable for real-time control-dominated applications. It is based on non-blocking communication among peers with designated peripheral processors. The interconnect may be either point-to-point or bus-based.

A handler communicates with the environment through device driver calls and with other handlers via messages. A message is an event that triggers another handler with an optional data value. Intraprocessor messages are implemented with shared variables. Interprocessor messages are transmitted via communication channels synthesized with elements from a communication library that contains buffers, FIFOs, arbiters, and interconnect templates. Given a partitioning of handlers provided by the user, Chinook will synthesize communication channels to satisfy timing and resource constraints. Once the communication components are chosen, they are connected to the respective processors using the interfacing techniques in section 6. If there are multiple communication channels between processors, each channel may be mapped to its own physical connection or they may share connections.

7.2 Migration between Processors

Keeping in mind Chinook's focus on aiding the designer's exploration of the design space, it is important that the designer be free to easily allocate functionality to different processors. Through assignment tags in the high-level specification, a designer can rapidly change the partitioning of functionality - between two processors, or between a processor and a direct hardware implementation. Because Chinook synthesizes interprocessor communication channels and optimizes their use, this task is greatly simplified for the designer. No longer does the designer need to radically alter code running on one processor and then propagate the changes to the others while keeping track of all the potential implications on timing requirements and resource access. These adjustments are made automatically by Chinook.

Migrating functionality is divided into three parts: input parameter sending, control sequencing, and output parameter receiving. Input and output parameters are mapped to latches or memory locations which are connected to the processor using the interfacing techniques discussed earlier. The control sequencing may simply be moved to another processor or be moved to hardware where it will be instantiated as a finite-state-machine and data-path. The general solution to this requires behavioral synthesis but is quite straightforward in most cases involving I/O. The original software is replaced with routines that pass the inputs, kick-start the hardware or the software handler on the other processor, and then read back the outputs.

8 Simulation

The design can be simulated at different levels of detail. The initial specification is compatible with behavioral Verilog and is simulated without exact timing or detailed I/O. As the synthesis steps refine abstract communications and operations into more concrete signals and components, outputs from intermediate design steps and the final implementation can also be simulated with cycle-level accuracy.

The simulator uses the Verilog-XL Programming Language Interface [2] to communicate with peripheral device models. The device models are written in C and make X-window calls to visually

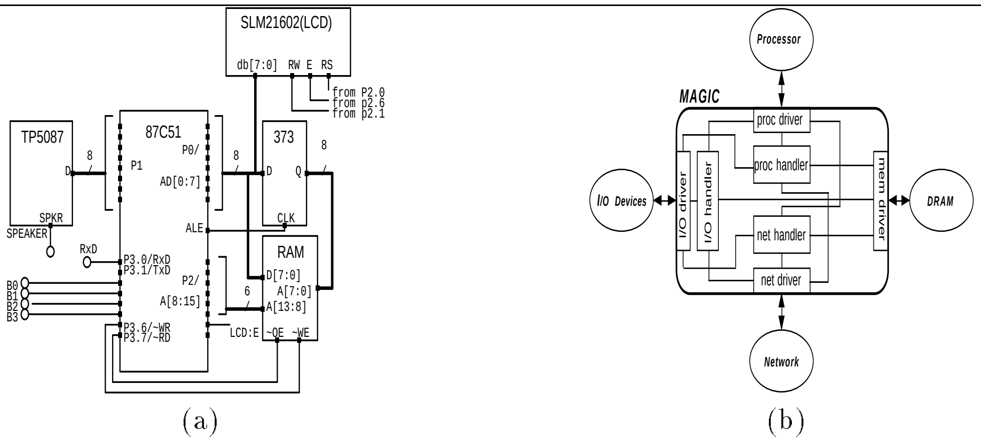

Figure 2: Examples synthesized in Chinook. (a) Portable Electronic Phonebook. (b) Communication in the MAGIC system.

represent the simulated device. Each device model exports the same API (application program interface) for simulation and synthesis. To simulate the specification during the early stages of the design, the API is bound to a behavioral simulation model. For example, a SCSI device exports a `send` routine. During simulation, the user may pop-up a window containing the various fields of a SCSI packet. After creating a new packet, the designer selects the send option which calls the `send` routine. This enables the user to simulate the environment of the system being designed in a consistent manner. During structural simulation of the system, the device's pin interface is modeled by running multiple FSMs to recognize all possible I/O sequencings in parallel. The FSM that matches the given I/O invokes the corresponding behavioral routine to simulate the device's reaction to the given waveform.

Chinook uses RTL-level processor models for simulating the final system implementation. The processor model, also written in C, interprets the same machine code that runs on the actual processor. At this stage, it is possible to execute the software with a debugger (although this is the synthesized code and not the original Verilog source). The binary code is disassembled and the registers, program counter, stack, internal memory, and built-in devices are visible in the processor status window. The processor model faithfully reproduces, within cycle-level accuracy, the appropriate waveforms on the processor's pins.

9 Examples

Several embedded systems have been designed using the Chinook tools. The following examples show the type of complexity that the current version supports. They are a portable electronic phonebook, a node controller for a distributed system, and a mobile defibrillator.

9.1 Portable Electronic Phonebook

The Portable Electronic Phonebook was originally designed by senior undergraduate students. Taking their implementation, we reverse-engineered a high level specification which was run through the Chinook tools (see Figure 2a). The generated solution required less hardware than the original implementation due to the interface synthesis algorithm. We were able to simulate the entire system at the behavioral and structural levels to validate the design. After building this application in hardware according to the generated netlist, the system operated correctly upon applying power.

9.2 MAGIC

The MAGIC (Memory and General Interconnect Controller) is a custom node controller for the FLASH architecture [16]. It communicates with a processor, network, I/O devices and DRAM (see Figure 2b). We modeled this architecture with three handlers, one for the processor requests, one for the network requests and one for the I/O requests. Since the DRAM does not initiate activity, it does not require its own handler. All communication with the DRAM occurs via device driver calls. We used the MAGIC application to experiment with using a common API for different peripherals. The specification was written so that it is easy to select a SCSI or Ethernet network interface chip. This demonstrates that designers can easily explore different high level options and observe their ramifications on other parts of the system. Now that we have both SCSI and Ethernet chips and drivers in the device library, it is straightforward to implement other systems that require these protocol chips. Using the results synthesized by Chinook, we performed our experiments with the simulator.

9.3 A Mobile Defibrillator

The purpose of the mobile defibrillator is to revive heart-attack victims with a powerful electrical shock. We consider the digital control subsystem containing an extensive interface including display of ECG waveforms, voice synthesis, digital audio recording, and PC-Card non-volatile storage. Because of the difficulty of guaranteeing that all timing constraints would be respected, the commercial version of this application was designed with a microcontroller and an ASIC. We are currently exploring solutions using reprogrammable components.

10 Conclusion

With the increasing availability of inexpensive and powerful microprocessors and FPGAs, designers of embedded systems are faced with more implementation choices than ever and given less time to realize their designs. Unfortunately, computer aided design tools are not tracking these trends. The Chinook co-synthesis system facilitates design space exploration and automates many aspects of system integration. These are often the most time-consuming and error-prone tasks in the embedded system design process.

Design space exploration is enabled by the use a single system specification that captures the reactive real-time behavior of the system and appropriately abstracts interactions with the environment to enhance retargetability. Since timing requirements are critical for many embedded applications, Chinook uses static scheduling to guarantee their satisfaction by construction. Several interface synthesis techniques are employed to interconnect system components. The necessary interface hardware and software is generated automatically and minimal glue logic is introduced.

At a higher level, Chinook facilitates easy migration of functionality among processing elements and manages the communication requirements between processors. This enables designers to rapidly evaluate different architectural templates and partitionings. Simulation is supported throughout the design cycle from the initial behavioral specification through the final structural implementation. Chinook's output consists of a netlist, logic specification, and code for each processor - all the elements needed for the construction of the complete system.

We have used to Chinook to synthesize several embedded systems including an electronic phonebook, SCSI interface to a VLSI chip tester, hand-held logic analyzer, and an infrared network transceiver. We are currently experimenting with its use in evaluating the design spaces for an automatic defibrillator and a multi-processor I/O subsystem. Future work includes developing synthesis methods for more efficient communication using higher level knowledge about the dataflow and control dependencies of the handlers. For instance, routing data around a processor may reduce processor load and yield higher performance at the cost of additional hardware. Ongoing work includes making Chinook more robust and more integrated, especially between the scheduler and compiler/estimator. In addition, we are investigating techniques to permit partitioning between software running on a workstation/PC and functionality in a peripheral device, which is an embedded system on a board attached to the system bus or other standard interface such as serial-line or PCMCIA slot.

References

[1] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9), September 1991.

[2] CADENCE Design Systems, Inc. *Programming Language Interface Reference Manual.* CADENCE Design Systems, Inc., 1992.

[3] D. Cathey. All things considered... important factors in choosing a real-time development system. *Real-Time Magazine*, 2nd quarter 1993.

[4] M. Chiodo et al. Hardware-software codesign of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.

[5] P. Chou and G. Borriello. Software scheduling in the co-synthesis of reactive real-time systems. In *Proceedings of the Design Automation Conference*, June 1994.

[6] P. Chou and G. Borriello. Interval scheduling: Fine-grained software scheduling for embedded systems. In *Proceedings of the Design Automation Conference*, June 1995.

[7] P. Chou, R. Ortega, and G. Borriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1992.

[8] P. Chou, E. A. Walkup, and G. Borriello. Scheduling for reactive real-time systems. *IEEE Micro*, 14(4):37–47, August 1994.

[9] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, December 1993.

[10] D. D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design and Test of Computers*, 12(1):53–67, Spring 1995.

[11] B. Gladstone. Specification of timing in a digital system. *ASIC and EDA*, pages 46–52, August 1993.

[12] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *Computers and Electrical Engineering*, 10(3):29–41, September 1993.

[13] R. K. Gupta and G. De Micheli. Constrained software generation for hardware-software systems. In *Proceedings of the Third International Workshop on Hardware/Software Codesign*, pages 56–63, September 1994.

[14] D. Harel. StateCharts: a visual formalism for complex systems. *Science of Programming*, 8, 1987.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[16] J. Kuskin et al. The Stanford FLASH multiprocessor. In *21st Annual Internation Symposium on Computer Architecture*, pages 302–313, 1994.

[17] A. K. Mok. The design of real-time programming systems based on process models. In *Real Time Systems Symposium*, pages 5–17, 1984.

[18] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[19] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs.* PhD thesis, University of Washington, 1992. Technical Report 92-08-02, Department of Computer Science & Engineering.

[20] M. Srivastava, B.C.Richards, and R.W.Brodersen. System level hardware module generation. *IEEE Transactions on VLSI Systems*, 3(1), March 1995.

[21] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic, 1991.

[22] E. A. Walkup and G. Borriello. Interface timing verification with application to synthesis. In *Proceedings of the Design Automation Conference*, June 1994.